Experimental Analysis of Precoloring Extensions

Completed by: Traci Lim

A Dissertation submitted to the Department of Mathematics of the London School of Economics and Political Science for the degree of Master of Science

September 10, 2018

Summary

The precoloring extension problem, or PrExt, is a computationally hard problem that belongs to a subfamily of graph coloring. The idea is to extend the coloring of a partially colored graph to a full one. The main challenge of real-life applications of PrExt is that ordinary graph coloring algorithms cannot be applied to precolored graphs because these algorithms do not take into account that colors of precolored vertices must stay unchanged. As such, there is a lack of literature in the study of algorithmic applications of precoloring extensions. To resolve this problem, we developed a conversion algorithm that converts a precolored graph into a graph fit for any ordinary graph coloring algorithms, while retaining the fixed-color property of precolored vertices. Apart from an overview of the existing literature of PrExt and graph coloring algorithms, we also delivered detailed expositions of two local search heuristics, TabuCol and PartialCol. The main highlight is a series of experimentation that focuses on applying local search heuristics to solve precoloring extension problems. Using the converted precolored graphs as instances for the experiments, we found that PartialCol with a reative tabu tenure surpassed TabuCol in terms of performance and reliability. We deduced that, when the number of unique colors in the precoloring reaches a certain threshold, TabuCol and PartialCol are ineffective in providing a better solution than a greedy coloring algorithm. We also observed a linear relationship between the the number of colors found by these heuristics and the number of unique colors in the precoloring. Lastly, we proposed explanations for the results based on our understanding of PrExt and algorithmic processes of TabuCol and PartialCol.

Contents

List of Figures

Chapter 1

Introduction

The study on graph coloring has always been prevalent in Graph Theory. Graphs are mathematical structures that are represented as networks of points (vertices) connected by lines (edges). They are used to model pairwise relations between objects. The main idea of "coloring" a graph is to assign the fewest number of possible colors to all its vertices so that no two adjacent vertices have the same color.

Precoloring extension is a special case of the ordinary graph coloring problem. Instead of coloring a graph with no colored vertices, the precoloring extension problem is interested in coloring a graph which contains both colored and uncolored vertices. This process is also known as 'extending' a partial coloring of a graph to a complete coloring of the whole graph. To put it crudely, if you were to have a conversation with PrExt, you will be asked, "Here's a partially colored graph, now finish coloring the whole graph for me."

1.1 Problem At Hand

The challenge of extending any partial coloring of graphs is that the precolored vertices must have their colors fixed throughout the extension process. This poses a huge problem because conventional graph coloring algorithms cannot be applied on these kind of graphs. They only accept uncolored graphs as input.

In the present day, real-world graphs are usually dynamic in nature. A dynamic graph is a graph that gets updated by edge and/or vertex insertion or deletion. The difference between dynamic and static graphs is the dimension of time. Dynamic graphs can be seen as a series of different static graphs with each time step. This translates to a sequence of precoloring extension problems—if we were to only consider vertex insertions. In existing graph coloring literature, only a few algorithms deal specifically with partially colored graphs, whereas conventional graph coloring algorithms still remain in active development.

To bridge the gap between precoloring extensions and graph coloring, we will de-

velop a reliable conversion algorithm to convert a precolored graph into an graph fit to run on any graph coloring algorithms. Furthermore, we will investigate how precoloring extensions affects the quality of solutions produced by local search graph coloring heuristics. Here TabuCol and PartialCol are specific local search heuristics for graph coloring that we will study in detail later. The *solution* refers to the k-coloring found by TabuCol or PartialCol. The *quality* or *performance* of a solution is evaluated by looking at two things: the number of additional colors needed to extending the precoloring, and the number of successive runs (runs that result in a k-coloring found by the algorithm). More specifically, we are interested in the following questions.

- How does the *number of unique colors in the precoloring* affect the solution?
- How does the *density* of a graph affect the solution?
- How do TabuCol and PartialCol compare in terms of quality of solutions?
- How do *dynamic* and *reactive* tabu tenure schemes compare in terms of quality of solutions?

1.2 Chapter Overview

In the first chapter, we set forth on the problem definition and objectives.

To provide a sufficient background on the topic, Chapter 2 includes a survey on existing literature of precoloring extensions. Although there is a sizable amount of theoretical results on complexity and color bounds pertaining to the problem, only a handful of papers have touched on the application side. In this chapter, we include literature in both theory and real-life applications of PrExt.

Chapter 3 moves into graph coloring algorithms in general. Since this dissertation houses a bulk of experimentation and algorithmic application of graph coloring, a brief survey of graph coloring algorithms is provided to make sense of the its progression. Due to the popularity of local-search type algorithms, like TabuCol and PartialCol, we give expositions on both heuristics.

Before we can understand the implications of precoloring extensions on performance, a conversion algorithm is explained in Chapter 4, coupled with its pseudocode, running time analysis and example. With this algorithm, we can model any precoloring extension problem as an ordinary graph coloring problem.

Chapter 5 features experimentation. With a clearer understanding of what is happening in the field of precoloring extensions, we proceed to run controlled experiments to investigate the implications precoloring extensions have on the quality of solutions, and draw an intuitive conclusion based on the results.

1.3 Notation and Terminology

We define the graph classes considered in this survey and other notation and terminology. All definitions, unless otherwise stated, are taken from the book: *A Guide to Graph Colouring: Algorithms and Applications* [28], which was authored by R.M.R. Lewis. We let $G = (V, E)$ be a graph, which consists of a set of n vertices V and a set of m edges E. An edge between vertices $u, v \in V$ is denoted as $uv \in E$.

Definition 1. If $uv \in E$, vertices u and v are said to be *adjacent*. Vertices v and u are also said to be incident to the edge $uv \in E$. If $uv \notin E$, then vertices u and v are *nonadjacent*.

Definition 2. A *clique* is a subset of vertices $C \subseteq V$ that are mutually adjacent, $\forall u, v \in C,\ uv \in E$. A *k-clique* is a clique of size k. The clique number or maximum clique size is denoted by $\omega(G)$.

Definition 3. The *neighbourhood* of a vertex v, written $N_G(v)$, is the set of vertices adjacent to v in the graph G. That is, $N_G(v) = \{u \in V : vu \in E\}$. The *degree* of a vertex v is the cardinality of its neighbourhood set, $|N(v)|$, usually written $deg_G(v)$. When the graph being referred to is made clear by the text, these can be written in their shorter forms, $N(v)$ and $deg(v)$, respectively.

Definition 4. A proper k-coloring is a function $c : V \to \{1, 2, \ldots, k\}$ such that $uv \in E$ implies $c(u) \neq c(v)$. (from Biró et al. [6])

Definition 5. A *color class* is a set containing all vertices in a graph that are assigned to a particular color in a solution. That is, given a particular color $i \in 1, \ldots, k$, a color class is defined as the set $\{v \in V : c(v) = i\}$.

Definition 6. A coloring of a graph is called *complete* if all vertices $v \in V$ are assigned a color $c(v) \in 1, \ldots, k$; else the coloring is considered *partial*.

Definition 7. Given a coloring c of the graph G, a *clash* describes a situation where a pair of adjacent vertices $u, v \in V$ are assigned the same color, i.e. $uv \in E$ and $c(v) = c(u)$.

Definition 8. A coloring is *feasible* if and only if it is both complete and proper.

Definition 9. The *chromatic number* of a graph G, denoted by $\chi(G)$, is the minimum number of colors required in a feasible coloring of G. A feasible coloring of G using exactly $\chi(G)$ colors is considered optimal.

Definition 10. An *independent set* is a subset of vertices $I \subseteq V$ that are mutually nonadjacent. That is, $\forall u, v \in I$, $uv \notin E$.

Definition 11. Given a graph $G = (V, E)$, an *adjacency matrix* is a matrix $\mathrm{A}_{n \times n}$ for which $\mathrm{A}_{ij} = 1$ if and only if vertices v_i and v_j are adjacent, and $\mathrm{A}_{ij} = 0$ otherwise.

Definition 12. The *density* of a graph $G = (V, E)$ is the ratio of the number of edges to the number of pairs of vertices. For a simple graph with no loops, it is calculated as $m/((n(n-1))/2)$, where n is the number of vertices in the graph, and m is the number of edges in the graph. Graphs with low densities are often referred to as *sparse* graphs; those with high densities are known as *dense* graphs.

Definition 13. An *empty graph* G consists of isolated vertices with no edges.

Definition 14. A *path* is a non-empty graph $P = (V, E)$. A path from vertex x_0 to x_k, or x_0, x_k-path, is denoted by $P = x_0x_1 \dots x_k$, where the vertices x_i are all distinct. The number of edges of a path is its *length*, and the path of length k is denoted by P^k. (from Diestel [11])

Definition 15. The *distance* of two vertices x, y in G is the length of a shortest x, y-path in G. (from Diestel [11])

Definition 16. Given a graph $G = (V, E)$ and $H = (V', E')$, if $V' \subseteq V$ and $E' \subseteq E$, then H is a *subgraph* of G, written as $H \subseteq G$. If $H \subseteq G$ and H contains all edges $xy \in E$ with $x, y \in V'$, then H is an *induced subgraph* of G; we say that V' induces or spans H in G, and write $H = G[V']$. (from Diestel [11])

Definition 17. Let $r \geq 2$ be an integer. A graph $G = (V, E)$ is called *r-partite* if V admits a partition into r classes such that every edges has its ends in different classes, or vertices in the same partition class must not be adjacent. A 2-partite graph is usually known as a *bipartite* graph. (from Diestel [11])

Definition 18. A non-empty graph G is called *connected* if any two of its vertices are linked by a path in G. (from Diestel [11])

Definition 19. A *family $\mathcal{F}$ of circular arcs* is a set $\{A_1, A_2, \dots, A_n\}$, where each A_i is an ordered pair (a_i, b_i) of positive integers, with $a_i \neq b_i$.

Definition 20. A graph G is called a *circular arc graph* if its vertices can be placed in one-to-one correspondence with a family $\mathcal{F}$ of arcs of a circle in such a way that two vertices of G are joined by an edge if and only if the corresponding two arcs in $\mathcal{F}$ intersect one another.

Definition 21. A split graph is a graph whose vertex set is the union of a complete subgraph and an independent set. (from Hujter and Tuza [25])

Definition 22. An *acyclic graph* is a graph that does not contain any cycles, also known as a *forest*. A connected forest is called a *tree*. (from Diestel [11])

Definition 23. A graph is called *perfect* if every induced subgraph $H \subseteq G$ has chromatic number $\chi(H) = \omega(G)$. (from Diestel [11])

Definition 24. A graph is *chordal* if each of its cycles of length at least 4 has a chord, i.e. if it contains no induced cycles other than triangles. (from Diestel [11])

Chapter 2

Precoloring Extensions

The precoloring extension problem is a variant of the graph coloring problem. In a precoloring extension problem, one is typically challenged by the following concerns:

- Extending the coloring of a partially colored (precolored) graph using as few colors possible.
- Keeping the colors of partially colored vertices unchanged.

2.1 Precoloring Extension Problem

Let $G[W]$ denote the subgraph of G induced by a vertex set W, where $W \subseteq V$. We define the Precoloring Extension problem, or PrExt as:

Precoloring Extension (PrExt):
Instance: A positive integer $k \geq 2$, a graph $G = (V, E)$, a vertex subset $W \subseteq V$, a proper k-coloring c_W of $G[W]$.
Question: Can c_W be extended to a complete and proper k-coloring of G?

The problem was first defined by Biró et al. [6] in 1992. In other words, given a partially colored graph G, can the precoloring be extended to a complete and proper coloring with at most k colors (for some given k). *Precoloring* refers to the proper k-coloring c_W of $G[W]$. We can think of an instance of PrExt as a *precolored* or *partially k-colored graph*, where k is the *color bound*. The set W will be used throughout this dissertation to refer to the set of *precolored* vertices, and we refer $V - W$ as the set of *precolorless* or *uncolored* vertices. The color classes of c_W are the sets $W_i = \{x \in W : c_W(x) = i\}$, where $i \in 1, \ldots, k$. Given a nonnegative interger d, we define the subproblem d-PrExt as the problem in which the instances of PrExt are restricted to those partially k-colored graphs where the size of each precolored class at most d; e.g. 1-PrExt presents the problem of PrExt as the event when every color is used at most once in the precoloring. All mentioned basic notions in this chapter are taken

from the same paper that introduced PrExt—Biró et al. [6].

The interest of extending precolorings surfaced in the nineties when Biró et al. [5] reviewed the practice of operative aircraft and maintenance scheduling at the Hungarian Airlines. The paper modeled the problem of scheduling flights and maintenance as a Precoloring Extension problem. Think of maintenance and flights as time intervals, and the vertices in a graph represent the fleet of planes in Hungarian Airlines. Since planes have to be assigned a flight and maintenance, precolored vertices correspond to planes which are assigned a particular maintenance interval, and precolorless vertices correspond to planes which are assigned a flight. Any vertices connected by an edge has an overlap in time intervals. We can easily see that this remodeled scheduling problem is actually PrExt in a special class of graphs—interval graphs. The following definition is taken from Biró et al. [6].

Definition 25 (Interval Graphs)**.** A graph $G = (V, E)$ is an *interval graph* if its vertices can be represented by open real intervals in such a way that two vertices are adjacent if and only if the corresponding intervals intersect each other. A *unit interval graph* is an interval graph that has an interval representation in which each interval has unit length.

Interval graphs are well known because of their theoretical and practical importance. Even in the 21st century, PrExt on interval graphs still remain prevalent in the domain of scheduling. In Section 2.3, we will elaborate on two real-life applications of PrExt: campsite scheduling and cloud resource allocation.

Key Results

It is clear that the PrExt is at least as difficult as the ordinary $\mathcal{NP}$-complete Graph Coloring problem. Thus, PrExt is $\mathcal{NP}$-complete in every class of graphs that is $\mathcal{NP}$-complete in the Graph Coloring problem. The study on the complexity status of PrExt produced results pertaining to several special classes of graphs. After Biró et al. [6] introduced the problem of PrExt and proved that 1-PrExt is polynomially solvable on interval graphs, Hujter and Tuza [25] proved that PrExt is solvable in polynomial time for some classes of perfect graphs: split graphs, complements of bipartite graphs and forests. For some other classes of perfect graphs, Hujter and Tuza [25] proved that PrExt is $\mathcal{NP}$-complete on bipartite graphs, it was proved using a series of reductions that starts from the 3-Coloration problem. The paper's main result is on the $\mathcal{NP}$-completeness of the bipartite PrExt problem, which reviewed that even when restricting the assignment of colors in precoloring to at most one vertex—B-1-PrExt, the problem remains $\mathcal{NP}$-complete.

PrExt in the case of trees, cographs, and connected bipartite graphs with $k = 2$ were found to be solvable in linear time in [26]. The main results of the paper

included some open problems that inquired on the complexity of PrExt on unit interval graphs and chordal graphs. Marx [32, 33] followed up and proved that 1-PrExt on chordal graphs and PrExt on unit interval graphs are both $\mathcal{NP}$-complete. The above mentioned results are on PrExt problems with unbounded k, i.e. k is part of the input. In the case of fixed k, i.e. not part the input, Marx [31] discovered that PrExt on chordal graphs becomes polynomially solvable in $O(kn^{k+2})$ if the number of colors used in precoloring is at most k.

Taking reference from [6], the proof of 2-PrExt will be presented. This complexity result is crucial in showing the hardness of the PrExt problem on interval graphs, and how the problem blows up by merely allowing every color used in the precoloring to appear at most two times.

Theorem 1. *2-PrExt is $\mathcal{NP}$-complete on interval graphs.*

Proof of Theorem 1—referenced from [6]. To show 2-PrExt $\in \mathcal{NP}$, we let the certificate be the assignment of colors to vertices. The verification algorithm checks that this assignment of colors is a feasible k-coloring, by going through every edge in G and ensure its endpoints are assigned different colors. It then counts the number of colors used and compare it with the given k. It is easy to see that we can verify the certificate in polynomial time because we are only iterating though all edges once.

To show 2-PrExt $\in \mathcal{NP}$-hard, we will prove that Circular-Arc Coloring $\leq_p$ (reduces to) 2-PrExt. The idea is simple. We want to construct an instance of 2-PrExt from an instance of Circular-Arc Coloring. Garey et al. [19] proved that the Circular-Arc Coloring problem becomes $\mathcal{NP}$-complete when k is not fixed.

Circular-Arc Coloring problem [19]:
Instance: A positive integer $k \geq 2$, a family of $\mathcal{F}$ of circular arcs.
Question: Can the circular arc graph $G = (V, E)$ be colored with k colors?

There are two objectives we want to achieve with this reduction: convert a circular arc graph into an interval graph; precolor some of the vertices such that every color is used at most twice in the precoloring. Consider an instance of the Circular-Arc Coloring, $< G, \mathcal{F}, k >$, where $G = (V, E)$, $\mathcal{F}$ is its arc representation, and k is an arbitrary integer. Without loss of generality, assume that $V = \{1, 2, \ldots, n\}$, and that the arc representation of G is a cycle of length m, which is defined as $C = \{u_1, u_2, \ldots, u_m\}$, and with edges $\{u_1u_2, u_2u_3, \ldots, u_{m-1}u_m, u_mu_1\}$, where m is a sufficiently large integer. Note that the size of m does not affect the original setup of arcs in $\mathcal{F}$, the use of cycle C is for an alternative representation of $\mathcal{F}$. Suppose the arcs $A_i \in \mathcal{F}$, $i = \{1, \ldots, n\}$ contain at least 2 and at most $m-1$ vertices of C. We pick an arbitrary arc, say (u_m, u_1), and observe that every arc that contains the edge u_mu_1 corresponds to a clique in G.

These set of overlapping arcs is represented by a set of paths $P_1, P_2, \ldots, P_t$ in C, where $t \leq \omega(G)$.

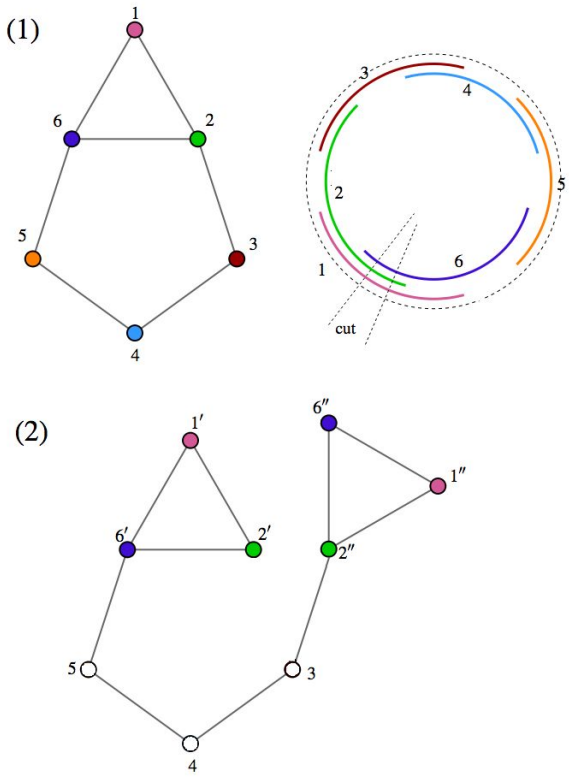
We first convert the circular arc graph into an interval graph by constructing a new graph G' from G. For every path $P_1, P_2, \ldots, P_t$, remove the edge $u_m u_1$ to separate each path into two shorter paths P_i' and P_i''. This will 'flatten' the circular arc representation into an interval graph. The interval graph now has $n + t$ intervals because splitting t paths results in additional t paths. Note that the intervals corresponding to the interval graph are represented as subpaths of $C - u_m u_1$.

Then, precolor the vertices in G' that correspond to the split paths P_i' and P_i'' with the color i. Vertices that correspond to paths that do not contain $u_m u_1$ are precolorless. This gives us a precolored interval graph where every color is used at most twice. Thus, the reduction is complete. Figure 2.1 shows an example of this conversion. It is easy to see that the precoloring of G' can be extended to a feasible k-coloring of G' if and only if the circular arc graph has a feasible k-coloring.

Tucker [37] proved that circular arc graphs and its arc representations are recognizable and detectable in polynomial time. Since the transformation from G' to G takes linear time, the reduction can be achieved in polynomial time. Therefore, 2-PrExt is $\mathcal{NP}$-complete.

□

Figure 2.1: Example of Conversion of Circular Arc Graph to Interval Graph



2.2 Distance Constraints

Because of the hardness of precoloring extension problem, a study on imposing distance constraints onto precolored vertices emerged. We associate the meaning of 'distance constraint' as limiting the distance of precolored vertices to a certain number. In other words, it is a study that investigates how far apart the precolored vertices need to be for the precoloring to be extendable. The study on imposing distance constraints onto precolored vertices is critical in the field of precoloring extensions, as it presents new perspectives for the $\mathcal{NP}$-complete PrExt problem. By laying down certain constraints on distances between precolored vertices, one might hope that PrExt becomes solvable even in general graphs.

Following the mathematical preliminaries from [3, 4], let $d(W)$ be the minimum distance between components of $G[W]$. The chromatic number $\chi(G)$ and precolored vertex set will be labeled r and W respectively for the rest of this section. We also assume $r \geq 3$. Out of all possible r-colorings of G, we pick one that maximises the minimum distance between vertices colored r, and denote this distance as d_r.

The first key result in this study was an answer to an open problem asked by Thomassen [36], which asked the following:

Problem 1. Suppose G is a planar graph and $W \subseteq V$ such that the distance between any two vertices in W is at least 100. Can a 5-coloring of $G[W]$ be extended to a 5-coloring of G?

Albertson's [1] answer to this open problem was the first key result in this study, which is the following theorem:

Theorem 2. *Suppose a graph $G = (V, E)$, where $\chi(G) = r$ and $W \subseteq V$ such that the distance between any pair of vertices in W is at least 4. Then any $(r+1)$-coloring of $G[W]$ can be extended to a $(r+1)$-coloring of G.*

Proof of Theorem 2—referenced from [1]. Suppose we want to extend a proper coloring of $W \subseteq V$ to a feasible coloring of G. Let the proper coloring of W be defined by the function $c_W : W \to \{1, 2, \ldots, r+1\}$, and let the coloring of G be defined by $c : V \to \{1, 2, \ldots, r\}$. Observe that c_W uses an extra color on top of $\chi(G) = r$. Ultimately, we want to modify c to agree with c_W on W.

Fix a vertex $u \in W$. If $c_W(u) = c(u)$, then no extension of c_W is required. If $c_W(u) \neq c(u)$, we assign the color $r+1$ to all neighbours of u that are colored $c_W(u)$. Since $d(W) \geq 4$, no pair of vertices that were recolored $r+1$ can be adjacent. The purpose of this is to allow the coloring c to be modified to get $c(u) = c_W(u)$. This means that other than the recoloring of neighbours of u, no recoloring of any vertices in W is required to extend a proper coloring of $G[W]$ to a feasible coloring of G. This

is because we can make a valid alteration to the coloring c for any $u \in W$, so that they receive the same color as what they were assigned before the extension. □

Key Results

With analogous arguments used in the above proof, Albertson [1] also showed that if W is made up of cliques with size at most j, and $d(W) \geq 6j-2$, then any $(r+1)$-coloring of $G[W]$ can be extended to a $(r+1)$-coloring of G. The distance bound was later improved by Kostochka to $d(W) \geq 4j$, who was credited in [2], where the strategy to the proof lies in recoloring the r-coloring of G clique by clique, until it agrees with the precoloring of precolored set W. Albertson and Moore [2] then followed up with a proof which showed that if $j = r$, then $d(W) \geq 3j$ is enough to extend any $(r+1)$-coloring of $G[W]$ to a $(r+1)$-coloring of G.

We are able to see the development of new theorems regarding different bounds on distance between any pair of vertices in W in [2, 3]. Albertson and Moore [2] showed that if $d(W) \geq 3$, then any $(r+1)$-coloring of $G[W]$ can be extended to a $\lceil \frac{3r+1}{2} \rceil$-coloring of G. In [3], both $d(W)$ and d_r are found to be crucial in extending any r-coloring of $G[W]$ to an r-coloring of G. This result holds in the following presets:

1. $d_r = 3$ and $d(W) \geq 12$
2. $d_r = 4$ and $d(W) \geq 8$
3. $d_r \geq 6$ and $d(W) \geq 6$

In general, there are no known bounds on $d(W)$ that guarantees the extendability from any r-coloring of $G[W]$ to an r-coloring of G, other than imposing some constraint on the distribution of r-th color.

Turning to the recent years, Ojima et al. [35] investigated how the number of pairs of vertices with distance 2 and 3 in W affected the bounds on the number of additional colors required to extend any r-coloring of $G[W]$.

We proceed to prove the following theorem from Albertson and Moore [2] to highlight how the extension problem escalates in the number of additional colors required for extension when the distance between any pair of vertices in W has a lower bound compared to Theorem 2.

Theorem 3. *Suppose a graph $G = (V, E)$, where $\chi(G) = r$ and $W \subseteq V$ such that the distance between any pair of vertices in W is at least 3. Then any $(r+1)$-coloring of $G[W]$ can be extended to a $\lceil \frac{3r+1}{2} \rceil$-coloring of G.*

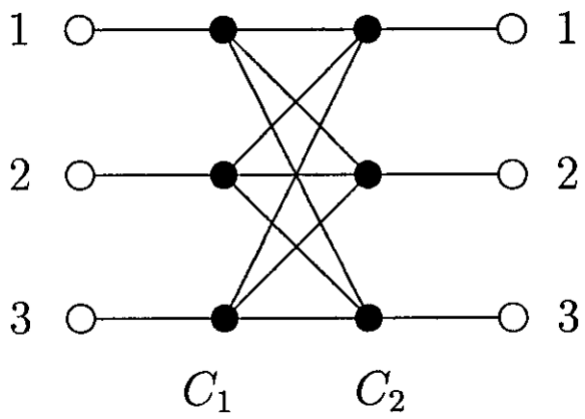
Proof of Theorem 3—referenced from [2]. Suppose we want to extend a proper $(r+1)$-coloring of $W \subseteq V$ to a feasible coloring of G. We first construct an r-partite graph where each part contains $r+1$ vertices. There are r such parts, denoted by C_i, $i = \{1, 2, \ldots, r\}$. Next, add an edge to every vertex and attach a vertex at its endpoint;

we label this constructed graph as G. Figure 2.2 shows an example of G when $r = 2$. We let $G[W]$ be the subgraph induced by the set W, which is the vertex set of attached vertices. Observe that for every pair of vertices $u, v \in W$, there are at distance 3 or 4 apart. We give a proper $(r + 1)$-coloring to $G[W]$, and the goal is to extend this coloring to all C_i's.

The coloring strategy for each C_i is simple. All vertices in each C_i must either be assigned two different colors $\{1, 2, \ldots, r + 1\}$, or one new color. Observe that the colors used on any C_i and C_j, $i \neq j$, must not have any overlap due to the r-partite property of the graph. We can understand why we need two different colors from colors $\{1, 2, \ldots, r + 1\}$ for each part by considering the following scenario. If each part gets one unique color from colors $\{1, 2, \ldots, r + 1\}$ and with no overlap of colors between parts, then we have a clash when at least one of the attached vertices have the same color as the one assigned to C_i. Thus, we must have two different colors from the $r+1$ colors for every part.

Also, if we do not allow a new color to be introduced, only $\lfloor \frac{r+1}{2} \rfloor$ parts are assigned two different colors from colors $\{1, 2, \ldots, r+1\}$ until we run out of colors. Consequently, a new color is introduced for every part in $\{C_{\lfloor \frac{r+1}{2} \rfloor + 1}, \ldots, C_{r-1}, C_r\}$. Note that one color is enough for each of these parts because vertices in each part are not adjacent to each other. Furthermore, there are $(r - \lfloor \frac{r+1}{2} \rfloor)$ many new colors introduced. Therefore, we need $(r + 1) + (r - \lfloor \frac{r+1}{2} \rfloor)$ colors for the extension, which equals a total of $\lceil \frac{3r+1}{2} \rceil$ colors. □

Figure 2.2: Graph with $r = 2$, from Alberston et al. [2]



2.3 Applications

Even in the 21st century, real-life problems are still modeled as precoloring extension problems. In this section, we highlight two real-life applications pertaining to PrExt on interval graphs: campsite scheduling and cloud resource allocation.

Campsite Scheduling

Ehmsen and Larsen [13] probed into a real-life campsite scheduling problem, modeling it as a precoloring extension problem for some fixed number of colors. A campsite scheduling problem is the problem of assigning bookings to campsites. While all bookings are represented as intervals, there are primarily two kinds of bookings described in the paper: bookings made on specified campsites or unspecified campsites. Assume you are planning to book a campsite through a campsite booking website, you can either specify which campsite you want (specified campsites) or not specify and let the system assign you whichever campsite that is available on the day of your arrival (unspecified campsites). Customers who make bookings on specified campsites are usually experienced and more familiar with the campground, and the ones who make bookings without any specifications are often first-time customers. There are two main challenges. The first challenge is to make an irreversible decision of whether or not to accept a booking every time a booking is received. The second is to come up with an assignment of bookings to unspecified campsites while considering that some booking-to-campsite assignments cannot be changed.

To understand this as a graph coloring problem, we let time intervals corresponding to all bookings be represented as vertices in a graph, which means that the total number of vertices is equal to the total number of bookings made. A color will correspond to a campsite, so 200 campsites correspond to 200 colors, or $k = 200$.

The first challenge can be modeled as an online graph coloring problem, where bookings arrive one by one, or in other words, vertices are added to the graph one by one. The second challenge is to address the constraints imposed by bookings made on specified campsites, which also means certain vertices must be assigned a certain color that must not be changed.

We can model the online graph coloring problem as an offline problem, i.e. a series of ordinary graph coloring problems on incrementally larger graphs. Thus, both challenges can be modeled as a precoloring extension problem for some fixed k, since the number of campsites is fixed.

Although polynomial algorithms are already developed on the topic of PrExt on interval graphs, the paper stressed that even polynomial algorithms are not feasible for large k, especially in this problem where the number of campsites can reach 150. They designed an algorithm that solves PrExt on interval graphs by taking into account that most bookings start and end on the same days, managing a running time $O(k^k n)$.

Cloud Resource Allocation

Cloud computing is the delivery of cloud services over the internet. Very much like paying your gas and electricity bills, companies pay for cloud services like physical servers, storage and high performance computing to cloud providers like Amazon and

Microsoft. Infrastructure as a Service (IaaS) data servers is a computing infrastructure that offers on-demand cloud services to end-users. The advent of big data has fueled the demand for such services, since most companies lack the IT infrastructure to maintain their growing data needs. In the following, we will try to understand how Ghribi and Zeghlache [20] modeled a cloud resource allocation problem as a precoloring extension problem.

The typical request for a cloud resource is as follows. Firstly, the end user makes a request of virtual machines (VM). These VMs are split into requested resource units (RRUs). In this context, resources refers specifically to virtual resources, or virtual resource units (VRUs). They comprise of compute, storage, communications resources, all of which are provided by physical servers in the IaaS cloud data center block. To put in perspective, an RRU is an abstract representation of a VRU at a request level. Thus, each RRU will be assigned a unique VRU.

We want to build a graph where vertices represent RRUs, colors represent VRUs, and any edge denote an overlap of time interval between the RRUs. The objective is to give a feasible coloring to such a graph, and taking into consideration of the capacity of infrastructure servers (number of VRUs), and making sure that servers used are high in performance per watt (PPW) value. The PPW of a server is defined as the rate of transactions or computations that can be delivered by a computer for every watt of power consumed. Making sure that resource units are allocated virtual resources with high PPW value is crucial in allowing less efficient servers to be shutdown and save energy.

Such graphs are often dynamic in nature, since vertices are often added and/or deleted. If we were to zoom in to a particular time step and consider a static graph, then we will be facing a precoloring extension problem. This case is when there is already a feasible coloring of the graph, but new requests made by users correspond to adding a new set of uncolored vertices to the graph. Therefore, we are now left with the problem of extending the existing coloring to a coloring of a bigger graph.

On top of graph coloring, the paper associated a weight w_j to each color. This weight represents the PPW of server j. Ghribi and Zeghlache [20] presented a fast Energy Efficient Graph Precoloring (EEGP) heuristic to find a feasible extension of a precoloring, one that maximizes the average power efficiency of servers.

Chapter 3

Overview of Graph Coloring Algorithms

The purpose of this section is to provide a detailed exposition on local search graph coloring algorithms, which will be used to solve for PrExt in the Section 5. The use of local search algorithms remains widely prevalent. It is often used as a subroutine in hybrid algorithms even today. We turn to graph coloring algorithms because papers documenting algorithmic design for precoloring extension problems are severely lacking in literature, as compared to algorithms for graph coloring problems.

3.1 Trends

There are two main types of algorithms: exact and inexact. The main difference is that exact algorithms only outputs an optimal solution to the problem, whereas inexact algorithms do not provide the guarantee of an optimal solution. Although exact approaches may seem highly desired compared to inexact ones, a quoted rule of thumb [17] is that inexact algorithms (heuristics) are considered more useful in graphs where $n > 100$ because exact approaches have to exhaustively search the solution space. In the following text, we briefly highlight some trends in graph coloring heuristics.

The simplest and most fundamental form of an inexact graph coloring algorithm is a greedy heuristic. It takes vertices one at a time according to some ordering and assign every vertex the first available color. DSATUR [8] and Recursive Largest First [27] are examples of efficient greedy heuristics that uses use more polished rules for ordering vertices. These constructive approaches are fast, but they usually require many more colors than the chromatic number $\chi(G)$ to color a graph. However, in spite of all this, they are often seen in initialization procedures in other better performing algorithms.

Local search heuristics, or metaheuristics (coined by Glover [21] in 1986), are highly-improved algorithms that start from an initial solution and perform local transformations (or moves) to improve the coloring. We define an incumbent solution to be

the current best solution found during an algorithmic search procedure. These transformations are 'local' because they are examined and evaluated at each incumbent solution. Thus, rather than considering all possible solutions, the moves are evaluated and chosen 'locally', one solution at a time. The aim is to reach an optimal solution (or k-coloring) by making the best 'move' for each incumbent solution. A metaheuristic typically requires much less work than designing a heuristic from scratch, and near-optimal solutions are often achievable in reasonable running time. Examples of metaheuristics include: stimulated annealing, tabu search, population-based algorithms. We will delve into this category of heuristic in the next section.

The first population-based approach for graph coloring by Davis [10] emerged in 1991, which was based on genetic and evolutionary principles. These kinds of algorithms works with a population of solutions, and use an evolutionary-type process for optimization, which are generally inspired by Darwinian principles. Since then, many population-based approaches have been developed [12, 14, 16, 18, 29, 30, 34], many were used in conjunction with local search procedures. These algorithms are powerhouses in terms of performance, which can be attributed to the wider pool of solutions considered, hybridization with local search, and powerful search operators. We will not be elaborating more on this type of heuristics because the focus of our dissertation is on local search approaches. To find out more on this, look towards Galinier et al. [15] for a detailed review.

In the more recent years, few other coloring approaches like reduce-and-solve [23, 38] have also proved its usefulness in coloring large graphs. Furthermore, in 2018, Zhou et al. [39] adopted reinforcement learning techniques in their graph coloring heuristic. The paper commented that reinforcement learning techniques can enhance local search capabilities by predicting sequential searching actions. Therefore, it is highly probable that the research efforts of graph coloring algorithms will lean towards reinforcement learning in the following years.

3.2 Local Search Heuristics

We can classify local search heuristics according to how their solution space are defined, such as spaces of complete improper k-colorings and spaces of partial proper k-colorings.

Spaces of improper k-colorings typically operate in fixed k context, where a fixed number of colors is proposed. Next, vertices are assigned colors using heuristics, possibly at random. The point to note is that when some of these vertices run out of colors without inducing a clash, they will be assigned a random color anyway. This results in an improper but complete coloring. Algorithms that operate in this solution space typically measure the quality of coloring solution by counting the total number of clashes. So the aim is to use neighborhood operators to reduce the number of clashes

to zero. If the clashes cannot be reduced to zero, the initially input k will be increased, and the process repeats. Neigborhood operators are schemes for changing a particular solution, in this case, the operator is to take some vertex v assigned color i, and assign it color j, where $(1 \leq i \neq j \leq k)$.

Spaces of partial proper k-colorings operate similar to spaces of improper k-colorings. A fixed k is also proprosed, as well as an assignment of colors using heuristics takes place. However, if the vertex cannot be feasibly assigned a color anymore, it is placed in a separate set of uncolored vertices, U. The idea is to make changes to the solution so that all vertices in U can be colored. The evaluation function is also different. Each solution in the space can be evaluated based on (i) size of U or (ii) the sum of degrees of vertices in U. The neighborhood operator also differs. It assigns a color i to some vertex $v \in U$, and any vertices adjacent to v that are colored i will be moved to U.

We introduce the most basic form of a local search method with a neighborhood operator—Random Descent method. It first generates an initial solution (not necessarily proper), and evaluates the solution by counting the number of clashes. At every iteration, the algorithm attempts a move in the solution space by randomly applying the neighborhood operator to the current solution to form a new solution. In the case when the new solution is evaluated to be better than the initial solution, then it becomes the new incumbent for the next iteration. Unfortunately, Random Descent are likely to get trapped in local optima, which happens when all neighbors of the incumbent solution fail to reduce the number of clashes. Two approaches to circumvent this problem were deveoloped in 1987: Stimulated Annealing (SA) [9] and TabuCol [24]. Stimulated Annealing is a generalization of Random Descent, except that instead of ignoring all worse moves, the algorithm accepts a worse move with probability, $\exp(-\delta/t)$, where δ is the absolute difference in number of clashes between incumbent and new solution, and t is the temperature parameter, which is set to a relatively high value at the start. This incorporation of randomness opens SA to a larger span of solution space, allowing it to escape from local optima. However, the problem with SA lies in its inability to remember solutions it previously observed within the solution space. This causes it to often 'cycle' within the same subset of solutions. To prevent such 'cycling' tendencies, TabuCol algorithm was developed by Hertz and Werra [24] in 1987, shortly after SA for graph coloring was introduced. Considered as one of the key breakthroughs in the field of graph coloring algorithms, TabuCol uses tabu search, which employs a short term memory structure known as a tabu list. A tabu list keeps tracks of moves and forbids the algorithm from returning to these moves for a period of time. We will present a more precise elaboration in the following sections.

3.2.1 TabuCol

The solution space of TabuCol operates in spaces of complete improper k-colorings. TabuCol explores a search space consisting of a set of k-colorings of a given graph G. We define a solution to be S, which is a paritition of vertex set $V(G)$ into k sets, where $S = (S_1, S_2, \ldots, S_k)$. The method of evaluation is the assessment of the total number of clashes in solution S, also known as cost, defined as the output of $f_1(S)$ in Equation 3.1. Any move $(u \hookrightarrow S_j)$ in this solution space is carried out by selecting a s vertex $u \in S_i$ that induce a clash, and assign it a new color class $S_j, i \neq j$.

The purpose of a tabu list is to record every visited solution, and disallow any re-visits to this solution for the next t iterations (t refers to tabu tenure). The tabu list is stored using a matrix $\mathrm{T}_{n \times k}$, where n is the number of vertices in the input graph and k is the input k. The value of each element T_{vj} is the number of iterations that vertex v is not allowed to be assigned the color j, we call such a move tabu. For example, if at iteration l a neighborhood operator moves a vertex v from S_i to S_j, then $\mathrm{T}_{vj} = l + t$. In other words, all solutions with $v \in S_j$ is tabu for the next t iterations. The original TabuCol algorithm proposed by Hertz and Werra [24] uses a static tabu tenure, where t is constant. In 1999, Galinier et al. [16] suggested that TabuCol will benefit with a dynamic tabu tenure setting, which depends on how high the incumbent solution's cost is. They suggested setting $t = 0.6 f_1(S) + r$, where r is an integer uniformly selected from 0 to 9. When cost $f_1(S)$ is high, higher values of t will force algorithm to explore different regions within the solution space. Low costs will naturally lead to lower values of t and the algorithm will be limited to a smaller focused region.

Two types of tabu tenure are available for use in this algorithm: dynamic and reactive. Reactive tabu tenure is based on how the cost fluctuates along the search process. For instance, if the cost experience small or no changes for a long period, we can speculate that the search process is stuck in an uninteresting region in the solution space. The algorithm can then react to this behavior by increasing the tabu tenure to escape that region, which is the same as forcing the search process to move away from the region. Three parameters have to be defined in reactive tabu tenure: frequency φ, increment μ, and threshold b. For every φ iterations, we first calculate α, the difference between the maximum and minimum values that the cost has taken during the past φ iterations. If $\alpha \leq b$, then we increase tabu tenure t by μ, and decrease t by 1 otherwise. These parameters must be tuned to avoid two extreme events. If b or μ are too large, or if φ is too small, then the tabu tenure rises unfavorably. If b or μ are too small, or if φ is too large, then tabu tenure hovers around zero, which traps the search process in an uninteresting region of the search space. To overcome these extremities, Blöchliger and Zufferey [7] suggested to choose the values for the parameters uniformly at random over some fixed intervals after every φ iterations.

The randomness can prevent over-training in the tuning phase and enable some level of robustness.

Recall that Random Descent repeatedly applies a neighborhood operator and then evaluate its solution at each iteration. TabuCol uses a Steepest Descent method. Instead of evaluating a solution at each iteration, all solutions in the neighborhood are evaluated at each iteration. This means that it calculates the cost of moving each clashing vertex (a vertex involved in a clash) into each of the $k-1$ color classes every time it attempts a move. Let x be the number clashing vertices in the incumbent solution S. This part of TabuCol is the most time consuming because of the $x(k-1)$ moves taking place at every iteration. The version of TabuCol used in this dissertation is from Galinier et al. [16], in which an improved data structure is adopted. Instead of performing $x(k-1)$ neighborhood moves and evaluating them one by one, a matrix $\mathrm{C}_{n\times k}$ is used to lessen the evaluation time, where n is the number of vertices in the input graph, and k is the input k. Each element C_{vj} denotes the number of vertices adjacent to v that is colored j. With this matrix, every move will update its corresponding entry in the matrix. So rather than counting the number clashes from scratch for each of the $x(k-1)$ moves, the cost of each neighbouring solution can simply be counted with the following formula:

$$f_1(S') = f_1(S) + \mathrm{C}_{vj} - \mathrm{C}_{vi} \tag{3.1}$$

Formula 3.1 is the evaluation function used by TabuCol, its output is the cost (number of clashes) of a solution in which vertex v is moved from color class S_i to S_j. Therefore, by simply updating the matrix C for each move and scanning each row of C that correspond to a clashing vertex, we can avoid naively counting the number of clashes of $x(k-1)$ solutions from scratch each time.

Update of Matrix C in TabuCol

Algorithm 1: UpdateC-T $(u \hookrightarrow S_j)$

```
1 for all v ∈ N(u) do
2 |   C_vi := C_vi − 1
3 |   C_vj := C_vj + 1
  end
```

The matrix update pseudocode is straightforward. For all neighbors of vertex u, we minus 1 from the element C_{vi} because the vertex u is no longer colored i. Line 3 shows a plus 1 because vertices adjacent to $v \in N(u)$ that are colored j has a new neighbor, i.e. vertex u.

An aspiration criterion exists in TabuCol, which permits tabu moves that are seen to be an improvement to the best solution throughout the run. Thus, in the event where a tabu move leads to a solution S' with $f_1(S') = 0$, the algorithm can halt.

This dissertation adopts the improved TabuCol algorithm by Galinier et al. [16]. We show its pseudocode in Algorithm 2.

Algorithm 2: TabuCol(G, k)

Input: Graph G, k, a tabu tenure scheme: Dynamic or Reactive
Output: A feasible k-coloring of G, or no k-coloring has been found.
Parameters: MaxIter, Dynamic or Reactive tabu tenure

```
1  Generate initial solution S = {S_1, S_2, ..., S_k} using Dsatur
2  S* := S
3  iter := 0
4  Initialize Tabu List T_{n×k}
5  Initialize evaluation matrix C_{n×k}
6  while f_1(S*) ≠ 0 or iter ≠ MaxIter do
7      iter := iter + 1
8      Evaluate cost all non-tabu moves M = {(u ↪ S_q) | u is clashing vertex ∈ S}
9      Update matrix C_{n×k}
10     Choose and apply the move with minimum f_1(S')
11     Introduce the chosen move (u ↪ S_q) to Tabu List T for t iterations
12     S := S'
13     if f_1(S) < f_1(S*) then
14         S* := S
15         iter := 0
       end
   end
16 if f_1(S*) = 0 then
17     return S*
   end
18 else
       return no k-coloring is found
   end
```

Dsatur is a greedy graph coloring heuristic that takes an ordering of vertices of G based on the saturdation degree of the vertices. The saturation degree of v is the number of different colors assigned to adjacent vertices. In line 1, the algorithm will call on Dsatur to generate an initial coloring. Vertices that cannot be given a feasibly color class are assigned a random color. In line 2, S^* refers to the optimal solution, i.e. solution with the lowest cost. In line 3–5, we initialize iter as a counter for the number

of iterations, a matrix $T_{n\times k}$ as tabu list, and matrix $C_{n\times k}$ as evaluation matrix. The algorithm enters a loop in line 6–15, stopping only when the number of clashes in S^* is 0, or when it reaches the maximum number of iterations set (denoted by parameter MaxIter. In line 8, A 'clashing' vertex refers to a vertex that induces a clash in the incumbent solution S. For all clashing vertices in S, evaluate the cost of every non-tabu moves and update matrix C by calling UpdateC-T on every move evaluated. Line 10 calculates $f_1(S')$ for every move by using formula 3.1, and selects the move with minimum cost. As seen in line 10, the move then becomes tabu for the next t iterations, which will vary depending on the type of tabu tenure specified. Finally, let $S^* = S$ if the cost of S is lesser than cost of S^*. Algorithm returns a solution S^* when the stopping criterion is met, which is when the solution has zero cost, or when the number of iterations has reached the maximum. If the cost of the solution fails to reach zero, the heuristic outputs 'no k-coloring is found'.

3.2.2 PartialCol

The solution space PartialCol operates in spaces of partial proper k-colorings. Developed by Blöchliger and Zufferey [7] in 2008, PartialCol also uses a tabu search technique. It differs with TabuCol in both solution space and evaluation function. Since it does not consider improper solutions, after the initial solution is generated, vertices that cannot be assigned feasible colors are placed into a set of uncolored vertices, U. Then, by making changes to the partial solution S, vertices of U can be assigned feasible colors so that $|U| = 0$.

Referenced from [7], we define the solution to be $S = (S_1, S_2, \ldots, S_k; U)$. It consists of k disjoint independent sets and a set $U = V \setminus \bigcup_{i=1}^{k} S_i$. The neighborhood operator is different from TabuCol. A move $(u \hookrightarrow S_c)$ in the solution space is taking an uncolored vertex $u \in U$ and assigning the color class S_c. Since vertices within each k disjoint sets must be independent, all vertices $v \in S_c$ that is adjacent to u are also transferred to U.

The evaluation function used in PartialCol is $f_2(S) = |U|$. At each iteration, $k \times |U|$ neighbouring solutions are evaluated. It means that every vertex in set $|U|$ are placed in k different color classes, so that PartialCol picks the move with the lowest f_2 score. Similar to TabuCol, we can utilize a more efficient data structure to lower the evaluation time. The same matrix $C_{n\times k}$ is used, although the update procedures is different. In this case, the cost of any neighboring solution is easily calculated with the following formula:

$$f_2(S') = f_2(S) + C_{uj} - 1 \tag{3.2}$$

Formula 3.2 is the evaluation function that PartialCol uses, its output is the cost (number of vertices in U) of a solution that resulted from a move $(u \hookrightarrow S_j)$. After the move is conducted, two things happen immediately. Firstly, all vertices in S_j that are

adjacent to u are moved to U. Secondly, all moved vertices are marked tabu for the next t iterations, where t is the tabu tenure. Similar to TabuCol, both dynamic and reactive tabu tenure schemes are available for use in this algorithm.

Update of Matrix C in PartialCol

Algorithm 3: UpdateC-P ($u \hookrightarrow S_j$)

```
1 for all v ∈ N(u) do
2     C_vj := C_vj + 1
3     if v ∈ S_j then
4         for all w ∈ N(v) do
5             C_wj := C_wj − 1
          end
      end
  end
```

Recall that in matrix C, each element C_{vj} stores the number of vertices adjacent to v that are colored j, where $j \in [1, k]$. In UpdateC-P, line 2 adds 1 to the element C_{vj} for all $v \in N(u)$ because u is moved to S_j, and 1 correspond to vertex u added to S_j. Line 3-5 considers all vertices in S_j that are adjacent to v. Since the algorithm transfers these vertices to U, the element C_{wj} receive a -1 for all vertices $w \in N(v)$.

This dissertation adopts the PartialCol algorithm by Blöchliger and Zufferey [7]. We show its pseudocode in Algorithm 4.

We can already spot some similarities with the pseudocode of TabuCol. The input, output and parameters are identical. Depending on the type of tabu tenure set, the value of t will vary accordingly. The initial solution is generated by a Dsatur graph coloring algorithm. We denote k_D as the number of colors used by the Dsatur algorithm. Since Dsatur generates a coloring for graph G with unbounded k, it may not use the same number of colors as k. In the event of this, if $k_D > k$, then any vertices that were assigned colors $\{k+1, k+2, \ldots, k_D\}$ will be moved to set U. If $k_D \leq k$, then PartialCol will return the coloring of Dsatur. Line 5–15 enters a loop, which stops only when $U = \varnothing$ or when the number of iterations reached the specified maximum iterations. In line 7, for every vertex in U, the algorithm evaluates the cost of every move to each of k colors. So, at most $k \times |U|$ neighboring solutions are evaluated, provided that they are non-tabu moves. Line 8 calls UpdateC-P for every move evaluated, which is explained in the next subsection. The move with minimum cost is selected and applied. In event of a tie, the algorithm chooses them randomly. Line 10–11 updates the Tabu List matrix. This part is slightly different from the one seen in Tabucol. There is an additional update in line 11 because we need to account

for all vertices that are moved to U after the chosen move. Lastly, the solution with $|U| = 0$ is returned. If no solution with $|U| = 0$ exists, then PartialCol outputs 'no k-coloring is found'.

Algorithm 4: PartialCol(G, k)

Input: Graph G, k, a tabu tenure scheme: Dynamic or Reactive
Output: A feasible k-coloring of G, or no k-coloring has been found.
Parameters: MaxIter, Dynamic or Reactive tabu tenure

```
1  Generate initial solution S = {S_1, S_2, ..., S_k; U} using Dsatur
2  S* := S
3  Initialize Tabu List T_{n×k}
4  Initialize evaluation matrix C_{n×k}
5  while |U| ≠ 0 or iter ≠ MaxIter do
6      iter := iter + 1
7      Evaluate cost of each non-tabu move M = {(u ↪ S_q) | u ∈ U, q ∈ [1, k]}
8      Update matrix C_{n×k}
9      Choose and apply the move (u ↪ S_q) with minimum f_2(S'), break ties
         randomly
10     Introduce the chosen move (u ↪ S_q) to Tabu List T_{n×k} for t iterations
11     Introduce all moves (w ↪ S_q) to Tabu List T_{n×k} for t iterations, for all
         w ∈ S_q adjacent to u
12     S := S'
13     if f_2(S) < f_2(S*) then
14         S* := S
15         iter := 0
       end
   end
16 if |U| = 0 then
17     return S*
   end
18 else
       return no k-coloring is found
   end
```

Chapter 4

Conversion Algorithm

Algorithmic studies of precoloring extensions are not as active as studies on graph coloring algorithms. This could be because real-life graphs with precoloring properties typically have certain inherent traits that can be exploited. Therefore, graph coloring may not be the only concern. We can see examples of these cases in the precoloring extension applications highlighted in section 2.3. The precoloring extension algorithm for campsite scheduling from Ehmsen and Larsen [13] was designed while taking into account that most bookings start and end on the same days. Ghribi and Zeghlache [20] proposed a precoloring extension algorithm for cloud resource allocation that maximizes average power efficieny of servers. However, these problems are still graph coloring problems in essence, any advancements of the graph coloring field should be transferable to precoloring extensions. With this objective at heart, this brings us to develop a conversion algorithm that converts a precoloring extension problem to a graph coloring problem, whose coloring represents a feasible solution to the precoloring extension problem.

Algorithm 5: PrextToGCP (G, k)

Input: Graph G in DIMACS format, set of precolored vertices W, k

Output: Graph G' in DIMACS format

Initialize G' by copying graph G and adding k additional vertices $u_1, \ldots, u_k$

Add edges between all $\binom{k}{2}$ pairs of added vertices to form a complete graph K_k

for all $v \in W$ **do**

  add an edge between v and u_i, $\forall i \in [1, k],\ i \neq j$, where $v \in S_j$

end

return G'

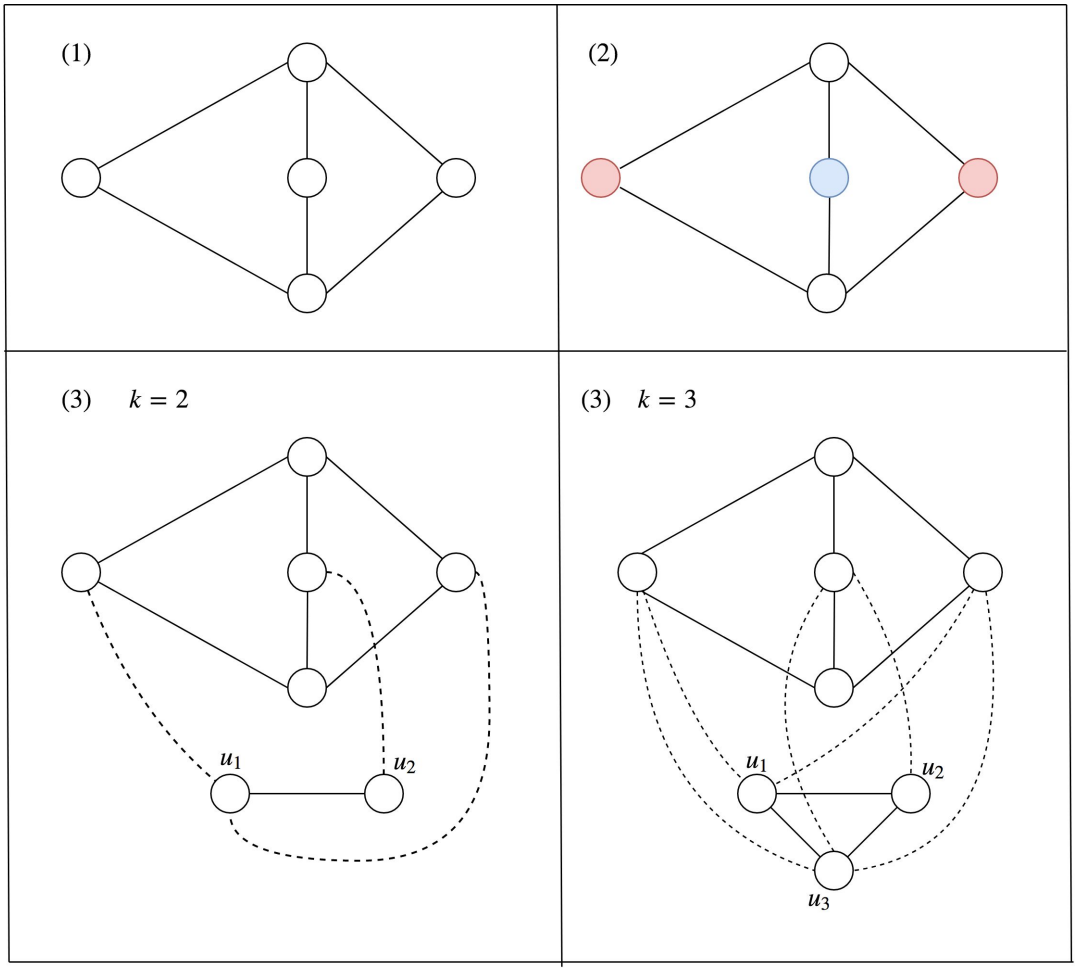
The *PrextToGCP* algorithm is inspired by a list coloring problem to graph coloring problem conversion mentioned by Lewis [28]. *PrextToGCP* ouputs an uncolored graph G' when given a precolored graph and integer k as input. We can refer to these graphs

as *converted precolored graphs*. The addition of a complete graph K_k (or k-clique) introduced in line 1-2 helps the converted problem retain the precoloring property. Now that the converted precolored graph G' contains a k-clique, **the idea is to 'force' certain vertices in G' to be feasibly colored one color.** Every vertex v in the precolored set W was given a k-coloring in G. Thus, depending on the color class each $v \in W$ was assigned to, an edge was added between v and all vertices $u_1, \ldots, u_k \in K_k$ except u_j, where j is the color v was assigned to. In other words, if two vertices of W were assigned the same color j, then connecting each of these two vertices to the same $k-1$ vertices of a k-clique ensures that any graph coloring algorithm will be forced to assign them the same color j. Note that the graph coloring algorithm may or may not assign the color j to these vertices, but this is easily achievable if we permute the indices of color classes of G' to agree with the indices of color classes of G. Once the converted precolored graph is colored, we can simply remove the k-clique and its edges to obtain an extended proper k-coloring of the initial precolored graph.

Figure 4.1 shows an example application of the conversion algorithm. Part (1) and (2) shows an uncolored and precolored graph. Part (3) shows the output of *PrextToGCP* for $k = 2$ and $k = 3$. The initial input k is important for applying an ordinary graph coloring algorithm on the converted graph. For any converted precolored graphs, applying an ordinary graph coloring algorithm with a fixed k that is not equal to initial input k will not retain the precoloring property of the converted precolored graphs. This means that vertices that were precolored before the conversion will not be assigned the same color as the color used in the precoloring. Therefore, to extend a coloring of a precolored graph, we have to convert it and specify a k beforehand, and then applying a graph coloring algorithm for that fixed k.

The running time of this algorithm is polynomial in the size of k. Line 1–2 takes $O(k^2)$ time because of the construction of complete graph K_k requires iterating through $\binom{k}{2}$ pairs of vertices. Line 3–4 takes $O((k-1)|W|)$ time because $|W|$ vertices were considered for the addition of $k-1$ edges for each of them. Hence, the running time for this algorithm is polynomial in the size of k and linear in the size of $|W|$.

Figure 4.1: Example Application of Conversion Algorithm



Chapter 5

Experimentation

Building proofs has always been the "gold standard" in mathematics. Although mathematicians understand that experimental evidence does not translate to conclusive proof, experimental mathematics still play an integral role. With the evidence it generates, a mere computational test to look for patterns may yield new conjectures and assertions, and perhaps lead to discoveries of formal proofs.

Let us recall the objectives of this dissertation. We are interested in the following questions.

1. How does the *number of unique colors in precoloring* affect the solution?
2. How does the *density* of a graph affect the solution?
3. How does the TabuCol and PartialCol compare in terms of quality of solutions?
4. How does the Dynamic and Reactive tabu tenure schemes compare in terms of quality of solutions?

Before we discuss the experimentation approaches, we must turn our attention to data preparation.

5.1 Generating Precoloring Instances

Since there are no known public datasets that has precolored vertices, all graph instances used has to be generated from scratch.

It is obvious that there are many approaches on how to generate a precoloring, like assigning a random color to random vertices, or assigning one color to vertices that are separated by distance 3, 4 or more. However, we must pin down an approach that allows for some important variables to be tested in a controlled manner. Variables such as graph density, number of unique colors in precolored vertices, types of tabu tenure schemes and proportion of precolored vertices have the potential to cause significantly different results if the method of generating precolorings is not controlled or fixed. To

avoid inconsistency and bias, the method of generating precoloring samples must not be underrated.

With the objective of generating a precoloring that generalizes well, we chose to generate the initial set of precolored vertices from a maximal independent set.

5.1.1 *Minimum-Degree-Greedy* Algorithm

An independent set in a graph is a group of vertices that are mutually nonadjacent. A maximum independent set is an independent set of largest possible size for a given graph G. A maximal independent set is an independent set that is not a subset of any other independent set. While it is known that the Maximum-Independent-Set problem is NP-hard, we turn to a specific approximation algorithm—the *Minimum-Degree-Greedy* algorithm. Developed by Halldórsson et al. [22], the algorithm finds a maximal independent set in a graph. The independent set found will be used as the set of precolored vertices.

The algorithm iterates through all vertices in the graph to find the vertex with the minimum degree, and removes the selected node and its neighbours from the graph. This process then repeats itself until the graph is empty. The pseudocode can be examined below.

Algorithm 6: Minimum-Degree-Greedy(G)

Input: Graph G

Output: Maximal Independent Set S

$S := \emptyset$

1 **while** $G \neq \emptyset$ **do**

2 choose v such that $d(v) = \min_{w \in V(G)} d(w)$, (break ties randomly)

3 $S := S \cup \{v\}$

4 $G := G - \{v \cup N(v)\}$

return S

end

The purpose of choosing the maximal independent set as precolored vertices can be explained in two folds. Firstly, having an independent set as precolored vertices provides more control during the experimentation process. Since no precolored vertices are adjacent with each other, we can vary the number of unique colors without violating any graph coloring rules. Secondly, other than distance constraints imposed on graphs, there are no documented studies on how precolored vertices can be chosen to minimize the number of additional colors required to extend the precoloring. Thus, the idea of constraining precolored vertices as the maximal independent set could be meaningful.

5.1.2 *Precoloring Generator*

The following pseudocode generates a precoloring by taking an uncolored graph (in DIMACS format) and p (number of unique colors for precoloring) as input, and outputs a precoloring as a text file.

Algorithm 7: Precoloring-Generator(G, p)

Input: Graph G, p,
Output: A precolored graph G'

```
1  x := 0
2  C := {C_1, C_2, ..., C_p}
3  S := Minimum-Degree-Greedy(G)
4  S := {s_1, s_2, ..., s_|S|}
5  if p ≥ |S| then
6      for q = 1 to |S| do
7          C_q := C_q ∪ s_q
       end
   end
8  else
9      for q = 1 to |S| do
10         C_q := C_q ∪ s_q
       end
11     for r = p + 1 to |S| do
12         x := random number between 1 to p − 1
13         C_x := C_x ∪ s_r
       end
   end
```

The *Precoloring-Generator* essentially takes an empty graph G and number of unique colors in precoloring, p as input, and assign colors to them. If p is 3, then the output of *Precoloring-Generator* is a precoloring with 3 unique colors.

In line 2, the set C refers to all color classes of the precoloring, and sets $C_1, C_2, \ldots, C_p$ contain the indices of vertices. e.g. If $C_1 = \{1, 4\}$, then it means vertices `1` and `4` are assigned the color 1. Line 3 calls *Minimum-Degree-Greedy*(G) as a subroutine, which outputs S, the maximal independent set, where vertices are labelled in line 4. Next, if the number of unique colors for precoloring is at least the size of maximal independent set, then it assigns color 1 to vertex `1` and color 2 to vertex `2`, until all vertices in S are receive a color, as seen in line 5–7. This reason why we disallow vertices to receive random colors is to avoid an irregular distribution of colors; i.e. when all vertices are randomly assigned color 1 even though $p > 1$.

Line 8–13 represents the case when $p < |S|$. In line 9–10, The first few vertices are assigned a unique color until there are no more unique color left; line 11–13 generates a random number between 0 and $p - 1$ for each uncolored vertex in S, and assign it the color x.

```
1 p edge 5 6
2 e 1 2
3 e 1 3
4 e 2 3
5 e 2 4
6 e 3 4
7 e 4 5
```

Listing 5.1: Input of *Precoloring-Generator*

In code listing 5.1, it shows how a typical graph is represented in DIMACS format. In line 1, the character '`p`' indicates the total number of vertices and edges in the graph respectively. The above graph has 5 vertices and 6 edges. In line 2–7, the character '`e`' is appended at the start of each line. Each line represents an edge of the graph, e.g. line 2 represents an edge between vertex `1` and `2`.

```
1 5
2 1 1
3 2 -1
4 3 -1
5 4 -1
6 5 2
```

Listing 5.2: Output of *Precoloring-Generator*

As seen in code listing 5.2 above, line 1 displays the total number of vertices in the graph. The first integer in line 2–6 is the index of the vertex, and the second integer refers to the color it was assigned to. In line 2 and 6, vertices `2`, `3` and `4` is assigned a color indexed with the integer `-1`. In this case, vertices `2`, `3` and `4` are uncolored.

5.2 Experimental Setup and Goals

Now that the data has been prepared, we will proceed to experimentation. All experiments are conducted on an Intel(R) Core(TM) i7-7700HQ Lenovo laptop, with 16GB of RAM. All code is programmed in `c++`, using Microsoft Visual Studio.

This section comprises of four main objectives. We look at the number of additional colors used in the solution to assess the performance. The lesser the number of additional colors used, the better the performance of the solution. The *solution* refers to the k-coloring found by TabuCol or PartialCol when given the converted

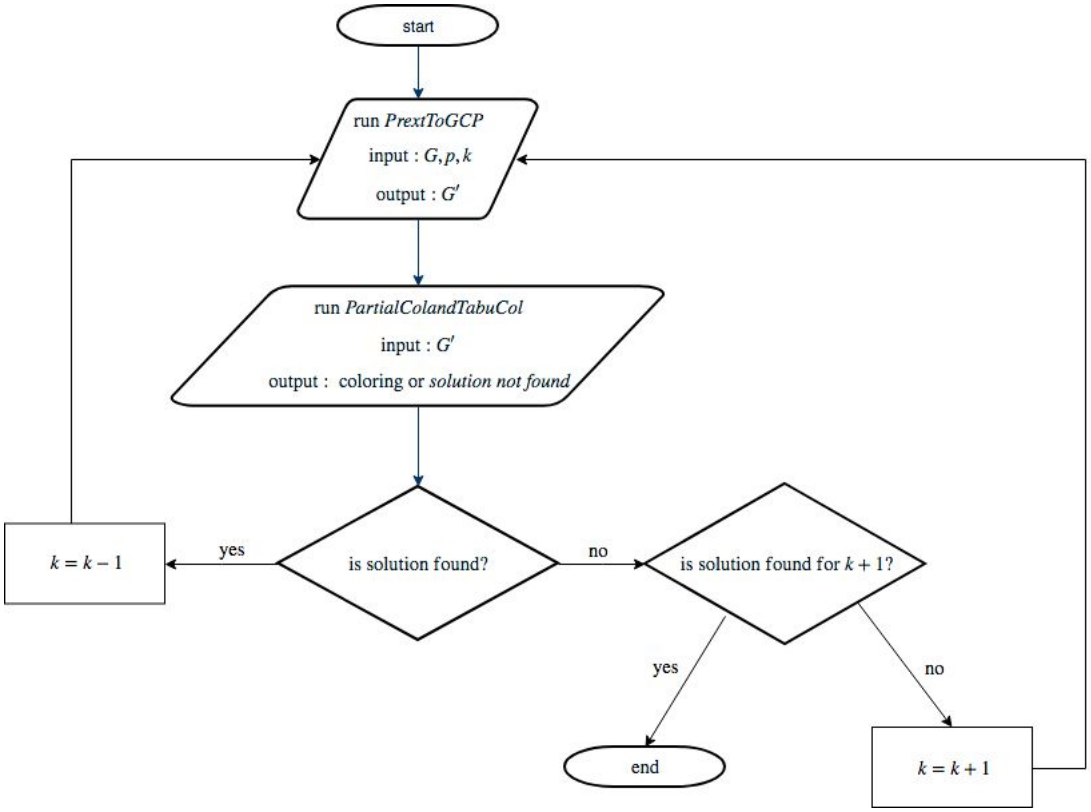
precolored graph as input. The following list shows a breakdown and the objective each hopes to achieve.

- **Graph Density vs. Performance**:
  This experiment seeks to understand how graph density affects the solution. While varying levels of density, we are interested in inspecting the number of additional colors used in the solution when compared to the precoloring. The graph density in this case refers to the probability of an edge being present between any pair of vertices in the graph, which can be expressed with the forumula: $\frac{2|E|}{|V|(|V|-1)}$.
- **TabuCol vs. PartialCol**:
  These two local search heuristics are both local search heuristics that uses the same tabu structure, but they are fundamentally different in many areas. The solution space they operate in, as well as their evaluation functions are totally different. Our experiment will investigate the quality of solutions produced by these two heuristics. The results will help us conclude which of the two offers a better solution for precolored graphs.
- **Number of Unique Colors in Precolored Vertices vs. Performance**:
  The number of unique colors in used in the precoloring may be indicative of how good the solution is. We denote this variable as p. Looking at a range of values of p, we will present the relationship between p and performance of the solution.
- **Dynamic Tabu Tenure vs. Reactive Tabu Tenure**:
  We explore two types of tabu schemes to determine which of the two stands out in terms of performance.

5.2.1 Data

The data used in this experiment are random graphs that vary in density. These graphs are randomly generated through the code attached in Section A. All experiments are conducted on 3 random graphs with densities: 0.1, 0.5, 0.9. The algorithm of generating random graphs first generates a set of isolated vertices and then adds edges between them with probability equals density, independently for each pair of vertices. To understand the density variable in random graphs, a density of 1 refers to a complete graph, 0 would mean a graph with isolated vertices, and 0.5 refers to a graph in which each vertex has $n/2$ edges on average. We chose an initial sample of three random graphs with 1000 vertices each. They are then precolored in a controlled manner to form the instances used for this experiment.

Figure 5.1: Methodology Flowchart for Experimental Process



5.2.2 Methodology

Figure 5.1 summarizes the general course of action during the experimental process. *PrextToGCP* and *PartialColandTabuCol* are separate `c++` programs. We first input a graph G in DIMACS format; number of unique colors in precoloring, p; and the desired number of colors to test, k, to *PrextToGCP*. It is a program that packages the code of *Precoloring-Generator* mentioned in Algorithm 7, and the code of *PrextToGCP* mentioned in Algorithm 5. Its output will then be used as input to *PartialColandTabuCol*, a program that contains TabuCol and PartialCol, as seen in Algorithm 2 and 4. The program, *PartialColandTabuCol*, was retrieved from the book from Lewis [28], *A Guide to Graph Colouring: Algorithms and Applications*. There will be two cases after this stage. If a solution is found for G', the initial specified k will be decreased by 1 and *PrextToGCP* will be run again. The process repeats until no solution can be found for G'. In this case, if a solution was found for the instance $k+1$, it means that k can no longer be lowered to result in a solution, since k itself failed to achieve a feasible solution. Then, the process ends. On the other hand, if no solution is found for k and no solution was found for $k+1$, k will be incrementally added until a solution is found for some k. More instructions on how to recreate the experimental process can be found at Section A.

Local search heuristics like TabuCol and PartialCol are have to be run multi-

ple times to measure their reliability. Convergence to a solution for both heursitics is dependent on the initial graph coloring generated from DSATUR, which can produce different colorings depending on the random seed. Therefore, to redress this, all instances tested on both TABUCOL and PARTIALCOL were ran 5 times with 5 different seeds. Additionally, we ran *PrextToGCP* a total of 10 times with 10 different seeds, and selected the graph that has the largest maximal independent set. Results will be presented along with the number of successes, out of the 5 runs for each instance. We did not report the time taken to run heurisitcs on any of the instances because they typically take seconds for 5 runs. For larger graphs with higher densities, the running time did not exceed 5 minutes for each instance. Both heuristics were stopped when a fixed number of iterations have been performed without improving the incumbent solution S. The number of maximum iterations set was 2×10^9.

For each given instance of precolored graph, its converted graph will serve as input to TABUCOL and PARTIALCOL, with two types of tabu tenure schemes: dynamic and reactive. Suggested by Galinier et al. [16], the dynamic tabu tenure was fixed at $t = 0.6f_i(S) + r$, for $i = \{1, 2\}$, and r is an integer uniformly selected from 0 to 9. Recall from Equation 3.1, $f_1(S)$ is the evaluation function used by TABUCOL that calculates the number of clashes in incumbent solution S. Likewise, from Equation 3.2, $f_2(S)$ is the evaluation function used by PARTIALCOL, which outputs the number of vertices in set of uncolored vertices, U. For the reactive tabu tenure scheme, we used the same parameters for both TABUCOL and PARTIALCOL. Since the code base was originally developed by Blöchliger and Zufferey [7], we kept the reactive tabu tenure configurations unchanged. After every φ iterations, the parameters: frequency φ and increment μ are selected uniformly at random over a list of 12 configurations, which contains a range of values: $\varphi = \{500, 1000, 5000, 10000\}$, and $\mu = \{5, 10, 15\}$. These values were not altered or tuned in our experimentation because the random graphs used in our experiment are of the same size and densities as the random graphs used by Blöchliger and Zufferey [7] in their series of tests. We also want to avoid the risk of over-tuning the parameters, which may result in some instances achieving biased results.

5.3 Results

Table 5.1 shows the recorded results for all experiments conducted. For every k-coloring found, number of successes were reported. A long dash indicates that the heuristic was only able to find a k-coloring with the initial coloring generated from DSATUR. We denote the k-coloring found by DSATUR to be a k_D-coloring. The DSATUR column shows the k_D-coloring generated from DSATUR when it produced a k_D-coloring with the same value as initial input k. This column is displayed because the at least one of

Table 5.1: Recorded Results

$|V|$: Number of vertices in the graph;
$|S|$: Size of maximal independent set;
p: number of unique colors in precoloring;
k_T, k_P: the number of colors for which a k-coloring was found at least 1 out of 5 times for TabuCol and PartialCol respectively;
Succ.: Number of successes out of 5 different seeds.

				TabuCol				PartialCol				Dsatur	
				Dynamic		Reactive		Dynamic		Reactive			
Density	$\|V\|$	$\|S\|$	p	k_T	Succ.	k_T	Succ.	k_P	Succ.	k_P	Succ.	k_D	Succ.
	1000	N.A	0	21	5 of 5	21	5 of 5	21	5 of 5	21	5 of 5	—	
	1000	53	1	—		23	1 of 5	—		22	5 of 5	77	5 of 5
	1000	53	2	—		25	2 of 5	—		24	5 of 5	77	5 of 5
	1000	53	3	—		34	1 of 5	—		29	1 of 5	69	5 of 5
	1000	53	4	—		43	1 of 5	—		35	1 of 5	61	5 of 5
	1000	53	5	—		45	1 of 5	—		44	2 of 5	62	5 of 5
	1000	53	6	—		53	1 of 5	—		43	1 of 5	59	3 of 5
	1000	53	7	—		55	2 of 5	—		50	1 of 5	57	5 of 5
0.1	1000	53	8	—		56	2 of 5	—		52	1 of 5	57	5 of 5
	1000	53	9	—		60	2 of 5	—		54	1 of 5	66	5 of 5
	1000	53	10	—		—		—		—		57	3 of 5
	1000	53	20	—		—		—		—		58	3 of 5
	1000	53	21	—		—		—		—		55	2 of 5
	1000	53	30	—		—		—		—		64	3 of 5
	1000	53	40	—		—		—		—		62	3 of 5
	1000	53	50	—		—		—		—		85	5 of 5
	1000	9	N.A	93	4 of 5	95	3 of 5	93	2 of 5	94	2 of 5	—	
	1000	9	1	525	1 of 5	—		523	2 of 5	98	1 of 5	531	5 of 5
	1000	9	2	—		—		—		98	1 of 5	531	5 of 5
	1000	9	3	524	1 of 5	—		—		99	2 of 5	531	5 of 5
	1000	9	4	—		—		—		100	1 of 5	531	5 of 5
0.5	1000	9	5	300	1 of 5	—		—		99	3 of 5	531	5 of 5
	1000	9	6	—		210	1 of 5	100	5 of 5	97	1 of 5	531	5 of 5
	1000	9	7	—		200	1 of 5	100	1 of 5	100	4 of 5	531	5 of 5
	1000	9	8	100	1 of 5	104	1 of 5	96	1 of 5	96	2 of 5	—	
	1000	9	9	95	2 of 5	97	2 of 5	94	2 of 5	96	3 of 5	—	
	1000	4	N.A	261	3 of 5	261	1 of 5	250	4 of 5	251	2 of 5	—	
0.9	1000	4	1	295	1 of 5	318	1 of 5	251	2 of 5	252	1 of 5	—	
	1000	4	2	314	1 of 5	316	1 of 5	254	1 of 5	254	1 of 5	—	
	1000	4	3	292	1 of 5	313	1 of 5	254	1 of 5	254	1 of 5	—	
	1000	4	4	263	3 of 5	262	1 of 5	250	3 of 5	252	4 of 5	—	

the heuristics with its tabu tenure scheme was unable to find a feasible coloring for the graph instance. We also like to note that when k reaches a sufficiently large number, all k-coloring output by any of the heuristics are generated solely from DSATUR. The reason will be explained below. When this happens, we mark these entries in the corresponding heuristic with a long dash, —.

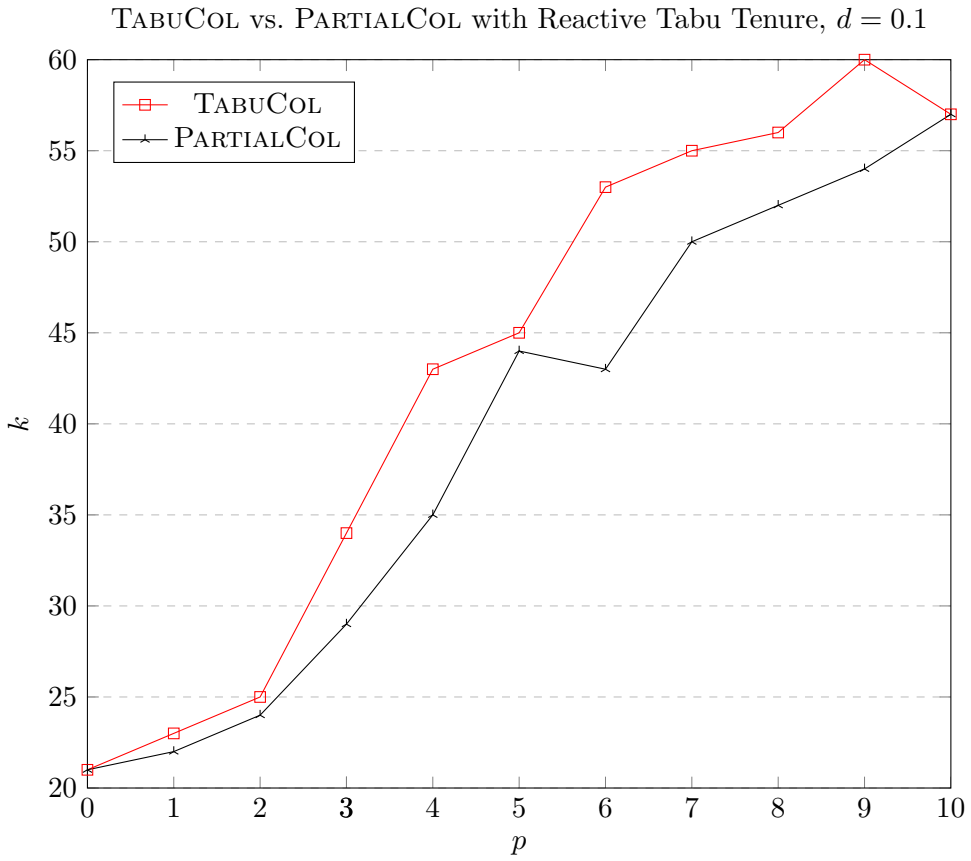
To understand why all k-coloring output by any of the heuristics are generated solely from DSATUR when initial input k is sufficiently large, we have to recall that for every increment of initial input k, the converted graph G' increases in size. Due to our conversion algorithm, the number of vertices of graph G' changes with respect to the k-clique formed. Hence, when k increase, the k-clique will also increase in size. This leads to a larger graph G'. When k gets sufficiently large, G' becomes sufficiently large to allow DSATUR to provide a k_D-coloring that is lower or equal to k. This case disposes the need to carry on any algorithmic processes of TABUCOL and PARTIALCOL, and both heuristics will just output the coloring produced by DSATUR.

A rough scan of Table 5.1 tells us that there are many long dashes recorded for TABUCOL and PARTIALCOL with dynamic tabu tenure. We observe that for all instances converted from precolored graphs with density $d = 0.1$, a dynamic tabu tenure failed to yield a feasible k-coloring for a k lesser than k_D, even for a relatively large k. One observation during experimentation was when both heuristics explore the solution space with a dynamic tabu tenure, they were both stuck in the solution space whenever the incumbent solution has a low cost, which was typically 1 or 2. Cost refers to the output of the equations 3.1 and 3.2 for TABUCOL and PARTIALCOL respectively. A dynamic tabu tenure adjusts tabu tenure values according to how high the incumbent solution's cost is. When the cost is lower, the tabu tenure t will be limited to a lower value. This means that the solution space will be be confined to a smaller region, since the algorithm will ignore all moves with higher costs. This essentially traps the search space in a small and unfavorable region. A reactive scheme on the other hand, circumvents this problem by 'reacting' to this behavior. Its increment parameter μ, has the chance to force the search process to move away from the problem region when it gets sufficiently large. The uniformly randomized settings for its parameters also helps to avoid μ getting too large, which could cause the tabu tenure to rise unfavorably. We believe that this is the advantage a reactive tabu tenure scheme has over a dynamic scheme.

A reactive tabu tenure scheme showed a more reliable record, except certain instances with density $d = 0.5$. Both heuristics were able to find k-colorings lesser than k_D for values of $p \in [1, 9]$. While they were found with a relatively low success rate, the values of k appears to be growing with respect to p. When p reaches 10, all heuristics failed to find a k-coloring lesser than the k_D-coloring DSATUR produced. In fact, for all values of $p \in [10, 50]$, no k-coloring was found by the heuristics with both tabu tenure schemes. We can safely deduce that, when p reaches a certain threshold, TABUCOL

and PartialCol are ineffective in providing a better k-coloring than a greedy coloring algorithm. Intuitively, when the number of unique colors in the precoloring increases, vertices that are adjacent to the precolored vertices will be allowed fewer number of colors to choose from. This constraint imposed will indefinitely reach a stage where coloring with k colors becomes impossible, and k has be increased to compensate for these constraints. When k gets sufficiently large, Dsatur will easily find a k_D-coloring that agrees with k. This explains why TabuCol and PartialCol output the k_D-coloring generated from its initial coloring stage when p becomes reasonable large.

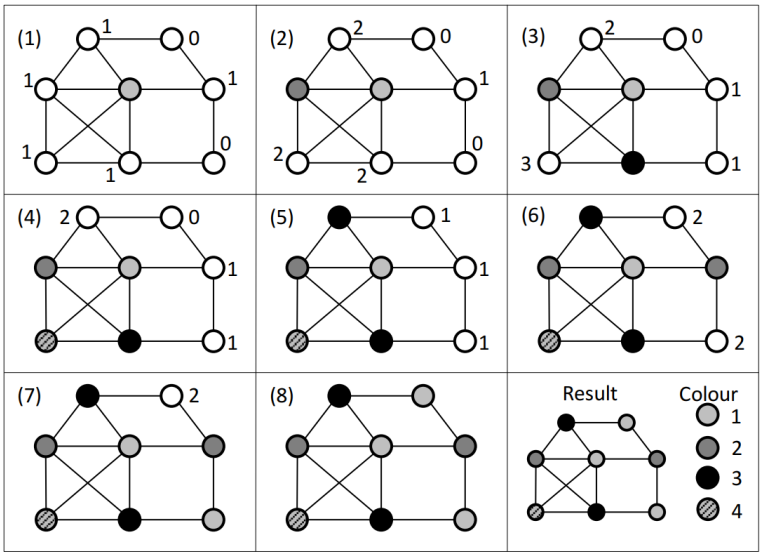
Figure 5.2: Line Chart for Comparison, $d = 0.1$



We turn to Figure 5.2 for a comparison between TabuCol and PartialCol with a reactive tabu tenure, on converted precolored graphs with density $d = 0.1$. When the number of unique colors in preocoloring is less than 10, PartialCol was observed to outperform TabuCol in terms of k-coloring produced. PartialCol consistently achieved a lower k-coloring than its counterpart. This speaks a lot about the advantage PartialCol has over TabuCol for the case of precolored graphs which are converted. These converted graphs are essentially random graphs joined with a complete graph of size k. Even in converted precolored graphs with density $d = 0.5$ and 0.9 (Figure 5.4, 5.5), PartialCol seemed to color these graphs more effectively than TabuCol. To further investigate the cause of this, we must understand how Dsatur produce the

initial k-coloring. DSATUR algorithm colors each vertex according to some ordering, and this ordering is decided heuristically based on the saturation degree of the vertices. The saturation degree of v is the number of different colors assigned to adjacent vertices. Figure 5.3 shows an example of applying DSATUR on a graph with a clique of size 4.

Figure 5.3: Example Application of DSATUR from Lewis [28]



Naturally, when a graph than contains a k-clique is presented to DSATUR, the k-clique typically gets colored first. Since the next vertex to be colored is the vertex with maximal saturation degree, vertices which are more 'constrained' are prioritized, i.e. vertices with the fewest feasible colors available to them. Less 'constrained' vertices are then dealt with later in the algorithm. However, in the graph instances used in this experiment, vertices in the k-clique is colored last. Given a graph instance of size 1000 with density $d = 0.1$, the converted graph contains a k-clique of size at least 22. Since each vertex has 100 edges on average, every precolored vertex will have at least $100 + 22 = 122$ edges on average. Using DSATUR, the vertex with the highest degree is the first to be colored, which is one of the precolored vertices. All vertices adjacent to the first colored vertex will have saturation degree of 1. The second vertex colored will be one of the first vertex's neighbors, whichever has the largest degree. The third vertex colored will reserved for a neighbor of the first and second colored vertex, which is definitely not a vertex in the k-clique. It is because vertices in k-clique are either connected to vertices in the clique or precolored vertices. Therefore, we can surmise that vertices in the k-clique are colored last.

For TABUCOL, when DSATUR used up all k colors during its coloring process, remaining vertices are assigned a random color from 1 to k. For PARTIALCOL, remaining vertices are put into a set U. Then, depending on the evaluation function

of each heuristic, all moves in the solution space are evaluated to choose a non-tabu move that reduces the cost. TabuCol evaluates each move by counting the number of clashes of the neighboring solution. Once it identifies a non-tabu move with lower cost, it will proceed and change the color of the clash vertex to a feasible color. In the input graph G', vertices can be categorized into three different types: vertices which were precolored before the conversion, vertices which were uncolored before the conversion, vertices forming the k-clique. Clash vertices are vertices in the k-clique because they are ones which are colored last by Dsatur. The problem with TabuCol is that it does not consider or make any move to vertices which were uncolored before the conversion, despite the possibility that these vertices may be assigned poor choices of colors by Dsatur. This is where PartialCol outperforms TabuCol in converted precolored graph instances. PartialCol makes moves to vertices in U after evaluating $k \times |U|$ neighboring solutions for each move. Then, any vertices affected by the move are moved into U. This may lead to a vertex outside of the k-clique being stripped of its color and put into U. This fixes the problem TabuCol has. Vertices that were uncolored before the conversion have a chance to change its initially assigned color classes, leaving its performance to be less affected by the initial coloring stage. On the other hand, TabuCol's performance suffers from a strong dependence on the initial coloring stage. **Therefore, we can conclude that PartialCol can produce a better k-coloring than TabuCol when p is smaller than a certain threshold.** When p reaches the threshold, TabuCol and PartialCol will not produce any k-coloring lower than k_D, because after all, they are reliant on the the initial coloring stage.

Figure 5.4: Line Chart for Comparison, $d = 0.9$

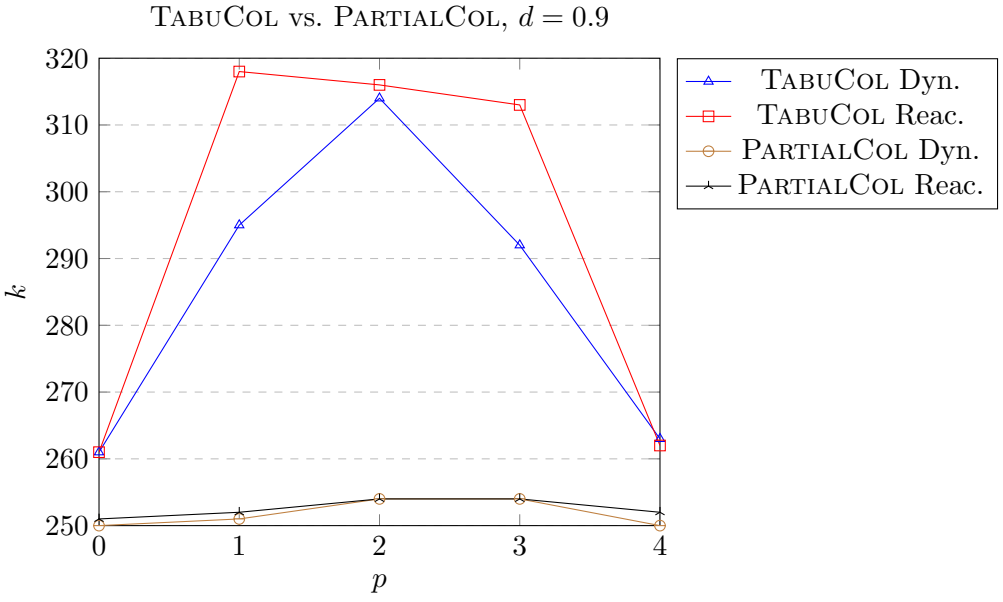
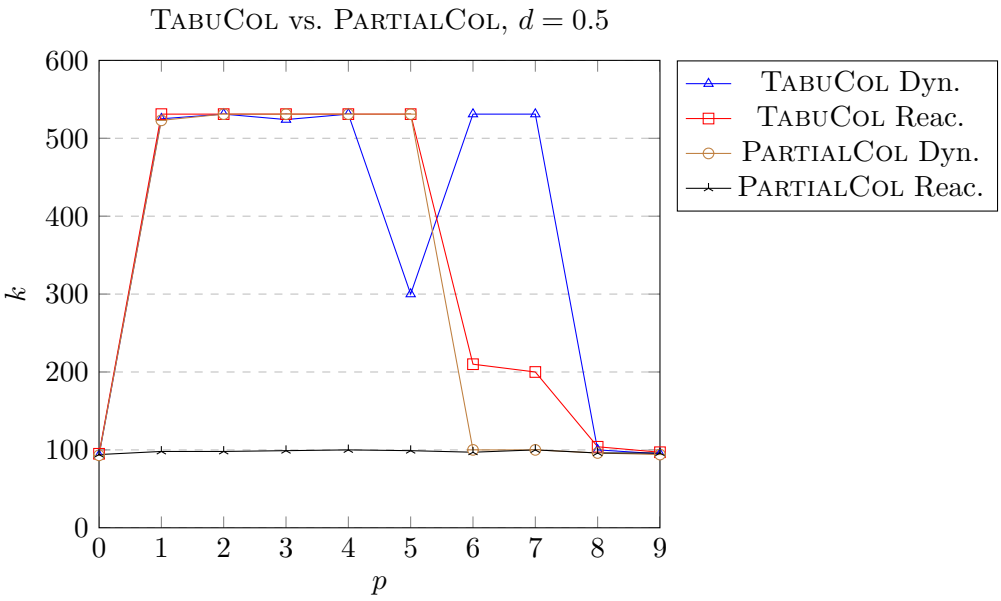


Figure 5.5: Line Chart for Comparison, $d = 0.5$



Interestingly, in graph instances with $d = 0.5$ and 0.9, performance of TabuCol and PartialCol seemed unaffected when p grows. This is contrary to the fact that we have previously observed a linear looking relationship between the k-coloring both heuristics output and the value of p (Table 5.2). To investigate whether density affects the relationship between k and p, it would be more appropriate to conduct more experiments that involve instances with higher levels of p, or p to k ratio. We can only speculate that density of the graph does not affect the interaction between k and p, since the evidence is not enough for us to make any acceptable deductions.

For instances with density $d = 0.9$, dynamic and reactive tabu tenure appeared to be stable in producing k-colorings, as compared to instances with density $d = 0.5$. TabuCol oddly showed a non-linear relationship between k and p, as seen in Figure 5.4. A dynamic tabu tenure looked like it outperformed a reactive tabu tenure. We could not explain non-linear relationship with any logical reasoning, except that the non-linear pattern was caused by inconsistent coloring generated at the initial coloring stage. In high density graphs, the coloring produced by Dsatur could heavily influence the k-coloring TabuCol produces because there are much more edges as compared to lower density graphs. More edges typically mean stricter constraints, where vertices are given fewer choices of colors.

Chapter 6

Conclusion

Overall, we can confirm a few things. When facing converted precolored graphs with low, medium and high densities, and out of the four presets of heuristics tested, PartialCol with a reactive tabu tenure stood out to be the most reliable heuristic for coloring converted precolored graphs. It provided consistent k-colorings for almost all instances tested. This can be credited to PartialCol's framework and the advantage a reactive tabu tenure has over a dynamic one. Although the success rates were mostly hovering around 1 and 2 out of 5 runs for each instance, it can be easily overlooked because of the fast running time for all instances.

In this experimental setting, both TabuCol and PartialCol have to basically compete on 'fixing' the problem of vertices in the k-clique running out of feasible colors. Using Dsatur as the inital coloring algorithm for both heuristics, they are both very susceptible to the initial coloring generated at the start. PartialCol was observed to produce more effective k-colorings than TabuCol because of the solution space and evaluation function it operates on.

For graph instances with low density, the number of unique colors in precolored vertices had a linear relationship with the k-coloring found. Although we could not confirm this fact for graph instances with higher densities, we concluded that the values of p experimented were too low for us to show the linear relationship. We highly suspect that it is because the p to k ratio for instances with larger densities is small. When p to k ratio is sufficiently large, we may be able to witness a direct relationship between the values of p and k.

Blöchliger and Zufferey [7] reported that the performance of PartialCol was competitive with TabuCol's, only outperforming TabuCol on certain flat graphs, i.e. graphs with a flatness parameter giving the maximal allowed difference between the degrees of two vertices. They also compared both dynamic and reactive tabu tenure schemes, with dynamic tabu tenure mostly coming out on top. The results on converted precolored graphs were encouraging. The conversion algorithm proved to be practical in coloring precolored vertices. We not only experienced solving the compu-

tationally hard problem of precoloring extensions, but also drew certain conclusions on how PartialCol with a reactive tabu tenure excels in coloring converted precolored graphs.

On the whole, there are many areas of improvement for this experiment. When tested on converted precolored graphs with more than 5000 vertices, both heuristics are met with memory errors. Although local search heuristics are highly popular and exhibits good performance in graph coloring, it is still not practical to color even larger graphs with them. Although the conversion algorithm works in translating a precoloring extension problem to a graph coloring problem, it is still inefficient, especially because it enlarges the graph before coloring it. This speaks a lot about the current state on the study of precoloring extensions. Precoloring extensions is a field that will remain relevant and practical in solving real-life problems. We must find better ways to incorporate the advances of ordinary graph coloring to precoloring extensions.

Bibliography

[1] Michael O Albertson. You can't paint yourself into a corner. *Journal of Combinatorial Theory, Series B*, 73(2):189–194, 1998.

[2] Michael O Albertson and Emily H Moore. Extending graph colorings. *Journal of Combinatorial Theory, Series B*, 77(1):83–95, 1999.

[3] Michael O Albertson and Emily H Moore. Extending graph colorings using no extra colors. *Discrete Mathematics*, 234(1-3):125–132, 2001.

[4] Michael O Albertson and Douglas B West. Extending precolorings to circular colorings. *Journal of Combinatorial Theory, Series B*, 96(4):472–481, 2006.

[5] M Biró, I Simon, and C Tánczos. Aircraft and maintenance scheduling support, mathematical insights and a proposed interactive system. *Journal of Advanced Transportation*, 26(2):121–130, 1992.

[6] Miklós Biró, Mihály Hujter, and Zs Tuza. Precoloring extension I. interval graphs. *Discrete Mathematics*, 100(1-3):267–279, 1992.

[7] Ivo Blöchliger and Nicolas Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3):960–975, 2008.

[8] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[9] M Chams, A Hertz, and D De Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research*, 32(2):260–266, 1987.

[10] Lawrence Davis. Order-based genetic algorithms and the graph coloring problem. *Handbook of Genetic Algorithms*, pages 72–90, 1991.

[11] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, 4th edition, 2010.

[12] Raphaël Dorne and Jin-Kao Hao. A new genetic local search algorithm for graph coloring. In *International Conference on Parallel Problem Solving from Nature*, pages 745–754. Springer, 1998.

[13] Martin R Ehmsen and Kim S Larsen. A technique for exact computation of precoloring extension on interval graphs. *International Journal of Foundations of Computer Science*, 24(01):109–122, 2013.

[14] Charles Fleurent and Jacques A Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, 63(3):437–461, 1996.

[15] Philippe Galinier, Jean-Philippe Hamiez, Jin-Kao Hao, and Daniel Porumbel. Recent advances in graph vertex coloring. In *Handbook of Optimization*, pages 505–528. Springer, 2013.

[16] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial optimization*, 3(4):379–397, 1999.

[17] Philippe Galinier and Alain Hertz. A survey of local search methods for graph coloring. *Computers & Operations Research*, 33(9):2547–2562, 2006.

[18] Philippe Galinier, Alain Hertz, and Nicolas Zufferey. An adaptive memory algorithm for the k-coloring problem. *Discrete Applied Mathematics*, 156(2):267–279, 2008.

[19] Michael R Garey, David S Johnson, Gary L Miller, and Christos H Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic Discrete Methods*, 1(2):216–227, 1980.

[20] Chaima Ghribi and Djamal Zeghlache. Exact and heuristic graph-coloring for energy efficient advance cloud resource reservation. In *Cloud computing (CLOUD), 2014 IEEE 7th international conference on*, pages 112–119. IEEE, 2014.

[21] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations research*, 13(5):533–549, 1986.

[22] Magnús M Halldórsson and Jaikumar Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1):145–163, 1997.

[23] Jin-Kao Hao and Qinghua Wu. Improving the extraction and expansion method for large graph coloring. *Discrete Applied Mathematics*, 160(16-17):2397–2407, 2012.

[24] Alain Hertz and Dominique de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[25] M Hujter and Zs Tuza. Precoloring extension II. graph classes related to bipartite graphs. *Acta Mathematica Universitatis Comenianae*, 62(1):1–11, 1993.

[26] Mihaly Hujter and Zs Tuza. Precoloring extension III. classes of perfect graphs. *Combinatorics, Probability and Computing*, 5(1):35–56, 1996.

[27] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–506, 1979.

[28] R Lewis. *A Guide to Graph Colouring*. Springer, 2015.

[29] Zhipeng Lü and Jin-Kao Hao. A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203(1):241–250, 2010.

[30] Enrico Malaguti, Michele Monaci, and Paolo Toth. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2):302–316, 2008.

[31] Dániel Marx. Parameterized coloring problems on chordal graphs. *Theoretical Computer Science*, 351(3):407–424, 2006.

[32] Dániel Marx. Precoloring extension on chordal graphs. In *Graph Theory in Paris*, pages 255–270. Springer, 2006.

[33] Dániel Marx. Precoloring extension on unit interval graphs. *Discrete Applied Mathematics*, 154(6):995–1002, 2006.

[34] Craig Morgenstern. Distributed coloration neighborhood search. *Discrete Mathematics and Theoretical Computer Science*, 26:335–358, 1996.

[35] Chihoko Ojima, Akira Saito, and Kazuki Sano. Precoloring extension involving pairs of vertices of small distance. *Discrete Applied Mathematics*, 166:170–177, 2014.

[36] Carsten Thomassen. Color-critical graphs on a fixed surface. *Journal of Combinatorial Theory, Series B*, 70(1):67–100, 1997.

[37] Alan Tucker. An efficient test for circular-arc graphs. *SIAM Journal on Computing*, 9(1):1–24, 1980.

[38] Qinghua Wu and Jin-Kao Hao. An extraction and expansion approach for graph coloring. *Asia-Pacific Journal of Operational Research*, 30(05):1350018, 2013.

[39] Yangming Zhou, Béatrice Duval, and Jin-Kao Hao. Improving probability learning based local search for graph coloring. *Applied Soft Computing*, 65:542–553, 2018.

Appendix A

Source Code

This section contains instructions on how to run the *PrextToGCP* and *PartialColandTabuCol* algorithm used in the experiments conducted. All algorithms are programmed in `c++`. They are compiled in Windows using Microsoft Visual Studio Community 2017, Version 15.7.4. To run the programs attached, simply double-click on the `.sln` files if you have Microsoft Visual Studio installed.

Usage

Three source codes are included in this appendix: `GenRandomGraphDensity.cpp`, `PrextToGCP.cpp`, `PreGCPFixedKTransformation.cpp`. Due to the length of the source code of *PartialColandTabuCol*, we will not be including its source code in this appendix. All codes are accessible in the thumb drive submitted along with this dissertation.

- ***GenRandomGraphDensity***
  Input: Graph Size, Density
  Output: A random graph `graph.txt` in DIMACS format

- ***PrextToGCP***
  Example command: `graph-1000-10.txt -r 1 -v -v -p 1 -c 23 -n 10`
  "`graph-1000-10.txt`" is the input graph in DIMACS format with 1000 vertices and 10% density.
  "`-r 1`" sets the random seed to 1.
  "`-v`" sets the verbosity. If present, output is sent to screen. If -v is repeated, more output is shown.
  "`-p 1`" sets the number of unique colors in the precoloring to 1.
  "`-c 23`" sets $k = 23$, where k is the k-coloring to be fixed.
  "`-n 10`" outputs ten converted precolored graphs with 10 consecutive levels of k.

Output:
`precolorSolution.txt`: shows indices of vertices and its assigned color class (look at Code Listing 5.2 to understand how to interpret this file).
`resultsLog.log`: shows history of commands
`newgraph.txt`: A converted precolored graph with $k = p$.
`newnewgraphX.txt`: A converted precolored graph with the specified $k =$ `X`.

- ***PartialColandTabuCol***
  Example command: `newnewgraph70.txt -t -tt -T 70 -v -v`
  "`newnewgraph70.txt`" is a converted precolored graph with $k = 70$ in DIMACS format
  "`-t`" If present, TabuCol is used. Else PartialCol is used.
  "`-tt`" If present, a dynamic tabu tenure is used, otherwise a reactive tenure is used.
  "`-T 70`" sets target number of colours to be 70. Algorithm halts if this is reached.
  "`-v`" sets the verbosity. If present, output is sent to screen. If -v is repeated, more output is shown.

  Output:
  `solution.txt`: shows indices of vertices and its assigned color class (which can be compared with `precolorSolution.txt` to confirm that vertices which were precolored gets the correct color, although the permutation might not be the same).
  `resultsLog.log`: shows history of commands, results, number of successes.

Workflow

A workflow of the experimental process is described in the following steps:

1. Run ***GenRandomGraphDensity*** program to get the random graphs with desired densities.

2. Copy the output file from step 1 to the main directory of ***PrextToGCP***.

3. Run ***PrextToGCP*** to generate converted precolored graphs for some fixed k.
   `RandomGraphGenerated.txt -r 1 -v -v -p 1 -c 22 -n 10`

4. Copy the output file(s) of ***PrextToGCP*** to the directory of ***PartialColandTabuCol***.

5. Run ***PartialColandTabuCol*** with the following commands to test all the heuristics.

```
newnewgraph22.txt -t -tt -T 22 -v -v
newnewgraph22.txt -t -T 22 -v -v
newnewgraph22.txt -tt -T 22 -v -v
newnewgraph22.txt -T 22 -v -v
```

6. Check `resultsLog.log` file to see which heuristic succeeded at least one time out of 5 seeds. If no k-coloring is found for all heuristics, go back to step 2 and set `-c` to 23.

...generate_random_graph_w_density\GenRandomGraphDensity.cpp 1

```
1  /*****************************************************************************/
2  //  This program generates a random graph for a specified number of vertices and
     density:
3  //
4  //
5  //  To run the program, no command arguments are needed.
6  //  ------------------------------------------------
7  /*****************************************************************************/
8
9  #include "stdafx.h"
10 #include <ctime>
11 #include <cstdlib>
12 #include <fstream>
13 #include <iostream>
14
15 using namespace std;
16
17 double prob() {
18     return (static_cast<double>(rand()) / RAND_MAX);
19 }
20
21 int main()
22 {
23     int edgeCount = 0;
24     int size = 15;
25     double density;
26
27     cout << "graph size?" << endl;
28     cin >> size;
29     cout << "graph density (0,1)?" << endl;
30     cin >> density;
31
32     bool** graph;
33     int** color;
34     int** cost;
35     srand(2); //seed random number generator
36
37     // Construct the different components, basically 3 matrices
38     graph = new bool*[size];
39     color = new int*[size];
40     cost = new int*[size];
41     for (int i = 0; i < size; i++) {
42         graph[i] = new bool[size];
43         color[i] = new int[size];
44         cost[i] = new int[size];
45     }
46
47     // Generate undirected edges
48     for (int i = 0; i < size; ++i) {
```

...generate_random_graph_w_density\GenRandomGraphDensity.cpp 2

```
49          for (int j = i; j < size; ++j) {
50              if (i == j) {
51                  graph[i][j] = false; //no loops
52              }
53              else {
54                  graph[i][j] = graph[j][i] = (prob() < density);
55              }
56          }
57      }
58      // Generate costs and colors
59      for (int i = 0; i < size; ++i) {
60          for (int j = i; j < size; ++j) {
61              if (graph[i][j]) {
62                  color[i][j] = color[j][i] = rand() % 3;
63                  cost[i][j] = cost[j][i] = prob() * 30;
64
65              }
66          }
67      }
68      // Output graph is saved in graph.txt
69      ofstream outp("graph.txt");
70      for (int i = 0; i < size; ++i) {
71          for (int j = i; j < size; ++j) {
72              if (graph[i][j]==true) {
73                  edgeCount++;
74              }
75          }
76      }
77      // display the number of vertices and edges in graph.txt
78      outp << "p edge "<< size << ' '<< edgeCount << "\n";
79      for (int i = 0; i < size; ++i) {
80          for (int j = i; j < size; ++j) {
81              if (graph[i][j] == true) {
82                  outp << "e " << i + 1 << ' ' << j + 1 << "\n";
83              }
84          }
85      }
86
87      return 0;
88  }
89
90
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 1

```
1  /*****************************************************************************/
2  //  This code precolors any graph in two steps:
3  //  1) Find the maximal independent set with minimum-degree-greedy over 10
     different seeds.
4  //  2) Randomly assign a color from a specified number of colors to each vertex
     in the independent set.
5  //
6  //  The code was written by Traci Lim, the code skeleton was adapted from the
     code package from
7  //  R.M.R Lewis's book: A Guide to Graph Colouring: Algorithms and Applications,
8  //  which was taken from rhydlewis.eu/resources/gCol.zip
9  //
10 //  To run the program, input the following example command.
11 //  ------------------------------------------------
12 //  "graph-1000-10.txt -r 1 -v -v -p 1 -c 23 -n 10"
13 //  ------------------------------------------------
14 //  "graph-1000-10.txt" is a graph in DIMACS format
15 //  "-r 1" is the random seed set as 1
16 //  "-v -v" sets verbose to true, which shows more information
17 //  "-p 1" sets the number of unique colors in precoloring to 1
18 //  "-c 23" sets k=23, where k is the k-coloring to be fixed
19 //  "-n 10" outputs ten converted precolored graphs with 10 consecutive levels of
     k
20 /*****************************************************************************/
21 #include "stdafx.h"
22 #include "PreGCPFixedKTransformation.h"
23 #include <string.h>
24 #include <fstream>
25 #include <iostream>
26 #include <vector>
27 #include <stdlib.h>
28 #include <time.h>
29 #include <limits.h>
30 #include <iomanip>
31 #include <algorithm>
32 #include <string>
33
34 using namespace std;
35
36 unsigned long long numConfChecks;
37
38 //-------------------------------------------------------------------------
     ------
39 void swap(int &a, int &b) {
40     int x = a; a = b; b = x;
41 }
42
43 //-------------------------------------------------------------------------
     ------
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 2

```
44  void readInputFile(ifstream &inStream, int &numNodes, int &numEdges, vector<
      vector<bool> > &adjacent, vector<int> &degree, vector< vector<int> > &adjList)
45  {
46      //Reads a DIMACS format file and creates the corresponding degree array and
          adjacency matrix
47      char c;
48      char str[1000]; // char array size 1000
49      int line = 0, i, j;
50      numEdges = 0;
51      int edges = -1;
52      int blem = 1;
53      int multiple = 0;
54      while (!inStream.eof()) {
55          line++;
56          inStream.get(c);
57          if (inStream.eof()) break;
58          switch (c) {
59          case 'p': // if first char == 'p'
60              inStream.get(c);
61              inStream.getline(str, 999, ' '); //max # char to write to char array
                  str is 999, operation of extracting succesive characters stops as
                  soon as next char is whitespace
62
63              // compare str and string "edge" character by character; returns 0
                    (false) if str contains "edge", 1 (true) otherwise
64              if (strcmp(str, "edge") && strcmp(str, "edges")) {
65                  cerr << "Error reading 'p' line: no 'edge' keyword found.\n";
66                  cerr << "'" << str << "' found instead\n";
67                  exit(-1);
68              }
69
70              inStream >> numNodes;
71              inStream >> numEdges;
72
73              //Set up the 2d adjacency matrix
74              adjacent.clear(); //Removes all elements from the vector (which are
                  destroyed), leaving the container with a size of 0.
75
76              // new container size, number of elements = numNodes
77              // new elements are initialized with a bool vector of size numNodes
78              adjacent.resize(numNodes, vector<bool>(numNodes));
79
80              for (i = 0;i<numNodes;i++) for (j = 0;j<numNodes;j++) {
81                  if (i == j)adjacent[i][j] = true;
82                  else adjacent[i][j] = false; // if i != j, then give it 0.
83              }
84              blem = 0;
85              break;
86          case 'n':
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 3

```
 87             if (blem) {
 88                 cerr << "Found 'n' line before a 'p' line.\n";
 89                 exit(-1);
 90             }
 91             int node;
 92             inStream >> node;
 93             if (node < 1 || node > numNodes) {
 94                 cerr << "Node number " << node << " is out of range!\n";
 95                 exit(-1);
 96             }
 97             node--;
 98             cout << "Tags (n Lines) not implemented in g object\n";
 99             break;
100         case 'e':
101             int node1, node2;
102             inStream >> node1 >> node2;
103 
104             //if node index is out of range
105             if (node1 < 1 || node1 > numNodes || node2 < 1 || node2 > numNodes)
                  { //
106                 cerr << "Node " << node1 << " or " << node2 << " is out of range!
                      \n";
107                 exit(-1);
108             }
109             node1--;
110             node2--;
111 
112             // if node1 and node2 has no edge recorded in adj matrix, increase
                  edges count
113             if (!adjacent[node1][node2]) {
114                 edges++;
115             }
116             else { // if node1 and node2 already has an edge recorded, increase
                  multiple count
117                 multiple++;
118                 if (multiple<5) {
119                     cerr << "Warning: in graph file at line " << line << ": edge
                       is defined more than once.\n";
120                     if (multiple == 4) {
121                         cerr << "  No more multiple edge warnings will be issued
                       \n";
122                     }
123                 }
124             }
125 
126             // record an edge between node1 and node2 in the adj matrix
127             adjacent[node1][node2] = true;
128             adjacent[node2][node1] = true;
129             break;
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 4

```
130         case 'd':
131         case 'v':
132         case 'x':
133             cerr << "File line " << line << ":\n";
134             cerr << "'" << c << "' lines are not implemented yet...\n";
135             inStream.getline(str, 999, '\n');
136             break;
137         case 'c':
138             inStream.putback('c'); //Attempts to decrease the current location in
                   the stream by one character, making the last character extracted
                  from the stream once again available to be extracted by input
                  operations.
139             inStream.get(str, 999, '\n');
140             break;
141         default:
142             cerr << "File line " << line << ":\n";
143             cerr << "'" << c << "' is an unknown line code\n";
144             exit(-1);
145         }
146         inStream.get(); // Kill the newline; //Extracts a single character from
              the stream.
147     }
148     inStream.close();
149     if (multiple) {
150         cerr << multiple << " multiple edges encountered\n";
151     }
152     //Now use the adjacency matrix to construct the degree array and adjacency
          list
153     degree.resize(numNodes, 0); //degree is a vector, initialize vector degree
          with all 0's
154     adjList.resize(numNodes, vector<int>()); //2d int vector
155     for (i = 0; i<numNodes; i++) {
156         for (j = 0; j<numNodes; j++) {
157             if (adjacent[i][j] && i != j) {
158                 adjList[i].push_back(j); //at the ith index of adjList vector,
                      add index j to it. So for each node (row), it will contain the
                      indices of all nodes it's adjacent to.
159                 degree[i]++; // increase the degree of node i by 1
160             }
161         }
162     }
163 }
164
165 //------------------------------------------------------------------------------
      ------
166 inline
167 void removeElement(vector<int> &A, int i) {
168     //Constant time operation for removing an item from a vector (note that
          ordering is not maintained)
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 5

```
169     swap(A[i], A.back()); //swap the element you want to delete with the reference to the last element
170     A.pop_back(); //delete last element
171 }
172
173 inline
174 void chooseMinDegVertex(vector<int> &X, int &v, int &vPos, vector<int> &Deg) {
175     //Select the vertex in X that has the min corresponding value in vector Deg (aka minimum degree)
176     int i, min = Deg[0], numMin = 1;
177     v = X[0]; // choose a arbitrary vertex and degree
178     vPos = 0;
179
180     // Go through the all nodes in vertex vector, take the index of the vertex with smallest degree,
181     for (i = 1;i<X.size();i++) {
182         if (Deg[i] <= min) {
183             if (Deg[i]<min) numMin = 0; // once a vertex with degree less than current min is spotted, set numMin to 0
184             if (rand() % (numMin + 1) == 0) { // use rand() to generate a random selection of vertices with same minimal degree
185                 min = Deg[i];
186                 v = X[i];
187                 vPos = i;
188             }
189             numMin++;
190         }
191     }
192 }
193 //-------------------------------------------------------------------------------------
194 inline
195 void updateX(vector<int> &X, vector<int> &XDeg, int v, int vPos, vector<int> &Y, vector<int> &YDeg, vector< vector<bool> > &adjacent, vector<int> &NInY) {
196     int i = 0, j;
197     //Remove v from X and update the relevant vectors XDeg, NInY
198     removeElement(X, vPos);
199     removeElement(XDeg, vPos);
200     removeElement(NInY, vPos);
201     //Transfer all vertices in X that are adjacent to v (which has already been removed) into Y. Also update the degree vectors
202     while (i<X.size()) {
203         numConfChecks++;
204         if (adjacent[X[i]][v]) { //for any vertex adjacent to v
205             // Move vertex at X[i] to Y.
206             // Also transfer X[i]'s degree minus 1 (because the edge between v and X[i] has been removed)
207             Y.push_back(X[i]); // add it to vector Y
208             YDeg.push_back(XDeg[i] - 1);
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 6

```
209             removeElement(X, i);
210             removeElement(XDeg, i);
211             removeElement(NInY, i);
212 
213             //Since a new vertex is being moved to Y, any vertex in X that is adjacent has its NInY entry updated
214             for (j = 0;j<X.size();j++) {
215                 numConfChecks++;
216                 if (adjacent[X[j]][Y.back()]) {
217                     NInY[j]++;
218                 }
219             }
220         }
221         else {
222             i++;
223         }
224     }
225 }
226 inline
227 void removeDuplicateColors(vector< vector<int> > &candSol, vector<int> &colNode) {
228 
229     // We need a replica of candSol vector because we will be removing elements from candSol, and we want to keep the indices unchanged
230     vector< vector<int> > candSolTemp;
231     candSolTemp = candSol;
232 
233     // loop through all color classes 1 onwards, if identical color with color class 0 exists, remove node from color class 0.
234     for (int i = 1; i<candSol[0].size(); i++) {
235         for (int group = 1; group<candSol.size(); group++) {
236             for (int h = 0; h<candSol[group].size(); h++) {
237                 if (candSol[group][h] == candSol[0][i]) {
238 
239                     // Finding an element in vector
240                     vector<int>::iterator it = std::find(candSolTemp[0].begin(), candSolTemp[0].end(), candSolTemp[group][h]);
241 
242                     // Get index of element from iterator
243                     int index = distance(candSolTemp[0].begin(), it);
244 
245                     removeElement(candSolTemp[0], index);
246                     //colNode[candSol[group][h]] = INT_MIN;
247                 }
248             }
249         }
250     }
251     candSol = candSolTemp;
252 }
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 7

```
253
254
255 // This function randomly recolor the precolored vertices to numColorsChosen different colors
256 // It makes sure that if numColorsChosen is equal to number of precolored nodes, every node gets a distinct color
257 inline
258 void precolor(vector< vector<int> > &candSol, vector<int> &colNode, int numColorsChosen)
259 {
260     // Open up new colors
261     // start from 1 since color 0 is already opened
262     for (int p = 1; p < numColorsChosen; p++) {
263         candSol.push_back(vector<int>());
264     }
265
266     // If numColorsChosen is more/equal than number of nodes assigned color 0
267     if (numColorsChosen >= candSol[0].size()) {
268         // Go through all vertices colored 0,
269         for (int q = 1; q < candSol[0].size(); q++) {
270             candSol[q].push_back(candSol[0][q]);
271             colNode[candSol[0][q]] = q;
272             //candSol[0][q] = INT_MIN;
273         }
274         removeDuplicateColors(candSol, colNode);
275
276     }
277     // if numColorsChosen < candSol[0].size(), remaining vertices must be randomly assigned a color
278     else {
279         // Assign the first few vertices with distinct colors until all colors are used up,
280         // We do this to avoid all vertices being randomly assigned the same color.
281         for (int q = 1; q < numColorsChosen; q++) {
282             candSol[q].push_back(candSol[0][q]);
283             colNode[candSol[0][q]] = q;
284         }
285         // Remaining vertices are randomly assigned a color.
286         for (int r = numColorsChosen; r < candSol[0].size(); r++) {
287             int randColor = rand() % (numColorsChosen);
288
289             if (randColor != 0) {
290                 candSol[randColor].push_back(candSol[0][r]);
291                 colNode[candSol[0][r]] = randColor;
292             }
293         }
294         removeDuplicateColors(candSol, colNode);
295     }
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 8

```
296 }
297
298
299
300 //------------------------------------------------------------------------------
    ------
301 inline
302 void makeSolution(vector< vector<int> > &candSol, int verbose, vector<int>
      &degree, vector< vector<int> > &adjList, vector<int> &colNode, vector<
      vector<bool> > &adjacent, int numNodes, vector<int> &numNodesIndepSet)
303 {
304     int i, c, v, vPos;
305     candSol.clear(); // 2d vector size is now 0
306
307     //X is a vector containing all unplaced vertices that can go into the current
        colour c (initially contains all unplaced vertices).
308     //XDeg contains the degree of the vertices induced by the subgraph of X (i.e.
        XDeg[i] == degree of X[i])
309     //Y is used to hold vertices that clash with vertices currently assigned to
       colour c (initially empty).
310     //YDeg contains the degree of the vertices induced by the subgraph of Y (i.e.
        YDeg[i] == degree of Y[i])
311     //NInY contains the number of neighbours vertex X[i] has in Y (initially zero
        because Y is empty at the start of each iteration)
312     vector<int> X(numNodes), Y, XDeg, YDeg, NInY(numNodes, 0);
313     for (i = 0;i<numNodes;i++) X[i] = i; //initialize vector X
314     XDeg = degree;
315     c = 0;
316
317     // Run the generatePrecoloring algorithm
318     // Two main steps:
319     // while X is not empty,
320     if (c == 0) {
321         //while (!X.empty()) {
322         //Open a new colour c, by adding a new element(vector in this case) at
            the end of the vector
323         candSol.push_back(vector<int>());
324
325         //Choose the vertex v in X that has the largest degree in the subgraph
            induced by X, then add v to colour c
326         //chooseVertex(X, v, vPos, XDeg); // this o/p v, the vertex with largest
            degree
327         chooseMinDegVertex(X, v, vPos, XDeg);
328
329         // Assign color c to vertex v
330         candSol[c].push_back(v);
331         colNode[v] = c;
332
333         updateX(X, XDeg, v, vPos, Y, YDeg, adjacent, NInY);
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 9

```
334         while (!X.empty()) {
335             // Choose the vertex v that has the smallest degree, then remove v
                and its neighbors from X, repeat until X is empty
336             chooseMinDegVertex(X, v, vPos, XDeg);
337             candSol[c].push_back(v);
338             colNode[v] = c;
339             updateX(X, XDeg, v, vPos, Y, YDeg, adjacent, NInY);
340         }
341         X.swap(Y); // Since X is now empty, swap contents with Y
342         XDeg.swap(YDeg);
343         NInY.resize(X.size(), 0); //reset size vector NInY and set all 0's
344                                   //c++;
345     }
346     numNodesIndepSet = candSol[0];
347 }
348
349
350 //------------------------------------------------------------------------------
      ------
351 inline
352 void prettyPrintSolution(vector< vector<int> > &candSol, vector<int>
      &numNodesIndepSet, int &numNodes, int &numEdges, int &maximal, int &setseed)
353 {
354     int i, count = 0, group;
355     cout << "\n\n";
356     for (group = 0; group<candSol.size(); group++) {
357         cout << "C-" << group << "\t= {";
358         if (candSol[group].size() == 0) cout << "empty}\n";
359         else {
360             for (i = 0; i<candSol[group].size() - 1; i++) {
361                 cout << candSol[group][i]+1 << ", ";
362             }
363             cout << candSol[group][candSol[group].size() - 1]+1 << "}\n";
364             count = count + candSol[group].size();
365         }
366     }
367     cout << "Total Number of Nodes and Edges = " << '(' << numNodes << ", " <<
        numEdges << ')' << endl;
368     cout << "Number of Nodes in maximal independent set = " <<
        numNodesIndepSet.size() << endl;
369     cout << "Precolored vertices have " << candSol.size() << " unique number of
        colors."<< endl;
370     cout << "Maximum size of maximal independent set over 10 random seeds is " <<
         maximal << ", with seed = "<< setseed << endl;
371 }
372
373 //------------------------------------------------------------------------------
      ------
374 inline
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 10

```
375 void checkSolution(vector< vector<int> > &candSol, vector< vector<bool> > &adjacent, int numNodes)
376 {
377     int j, i, count = 0, group;
378     bool valid = true;
379 
380     // Finally, check for illegal colourings: I.e. check that each colour class contains non conflicting nodes
381     for (group = 0; group < candSol.size(); group++) {
382         for (i = 0; i < candSol[group].size() - 1; i++) {
383             for (j = i + 1; j < candSol[group].size(); j++) {
384                 if (adjacent[candSol[group][i]][candSol[group][j]]) {
385                     cout << "Error: Nodes " << candSol[group][i] << " and " << candSol[group][j] << " are in the same group, but they clash" << endl;
386                     valid = false;
387                 }
388             }
389         }
390     }
391     if (!valid) cout << "This solution is not valid" << endl;
392 }
393 
394 
395 //-------------------------------------------------------------------------------------
396 int main(int argc, char ** argv) {
397 
398     if (argc <= 1) {
399         cout << "Generating precolored graph from uncolored graph\n\n"
400             << "USAGE:\n"
401             << "<InputFile>     (Required. File must be in DIMACS format)\n"
402             << "-r <int>        (Random seed. DEFAULT = 1)\n"
403             << "-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)\n"
404             << "-c <int>        (Number of distinct colors for precoloring. DEFAULT = 1)\n"
405             << "****\n";
406         exit(1);
407     }
408 
409     int i, verbose = 0, randomSeed = 1, numNodes, numEdges = 0, numPrecolorsChosen=0, numColorsChosen = 0, precoloringMode=0, prepareForGCP=1, many=1;
410     vector<int> degree;
411     vector< vector<int> > adjList;
412     vector< vector<bool> > adjacent;
413     numConfChecks = 0;
414 
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 11

```
415     for (i = 1; i < argc; i++) {
416         if (strcmp("-r", argv[i]) == 0) {
417             randomSeed = atoi(argv[++i]);
418         }
419         else if (strcmp("-v", argv[i]) == 0) {
420             verbose++;
421         }
422         else if (strcmp("-p", argv[i]) == 0) {
423             precoloringMode = 1;
424             numPrecolorsChosen = atoi(argv[++i]);
425         }
426         else if (strcmp("-c", argv[i]) == 0) {
427             prepareForGCP = 2;
428             numColorsChosen = atoi(argv[++i]);
429         }
430         else if (strcmp("-n", argv[i]) == 0) {
431             many = atoi(argv[++i]);
432         }
433         else {
434             //Set up input file, read, and close (input must be in DIMACS format)
435             ifstream inStream;
436             inStream.open(argv[i]);
437 
438             // initialize adj matrix, adjList, degree vector, edges count,
                  multiple count, numEdges, numNodes
439             readInputFile(inStream, numNodes, numEdges, adjacent, degree,
                  adjList);
440             inStream.close();
441         }
442     }
443 
444     //Set Random Seed
445     srand(randomSeed);
446 
447     //Check to see if there are no edges. If so, exit straight away
448     if (numEdges <= 0) {
449         if (verbose >= 1) cout << "Graph has no edges. Optimal solution is
              obviously using one colour. Exiting." << endl;
450         exit(1);
451     }
452 
453     //Start the timer
454     clock_t runStart = clock();
455 
456     //Declare strucures used for holding the solution
457     int maximal, setseed;
458     vector< vector<int> > candSol;
459     vector<int> colNode(numNodes, -1), numNodesIndepSet;
460 
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 12

```
461     // Over 10 seeds, find the seed that produce the maximum size of maximal
          independent set
462     maximal = 0;
463     for (int i = randomSeed; i < randomSeed+10; i++) {
464         int randomnum = rand() % 10;
465         srand(i+ randomnum);
466         makeSolution(candSol, verbose, degree, adjList, colNode, adjacent,
              numNodes, numNodesIndepSet);
467         if (numNodesIndepSet.size() >= maximal) {
468             maximal = numNodesIndepSet.size();
469             setseed = i+ randomnum;
470         }
471     }
472
473     // Greedily find the maximal indepedent set and assign color 0 to all
          vertices
474     srand(setseed);
475     makeSolution(candSol, verbose, degree, adjList, colNode, adjacent, numNodes,
          numNodesIndepSet);
476
477     // Precolor maximal independent set with numColorsChosen number of colors
478     if (precoloringMode = 1) {
479         precolor(candSol, colNode, numPrecolorsChosen);
480
481         // Output the solution to a text file
482         ofstream solStrm;
483         solStrm.open("precolorSolution.txt");
484         solStrm << numNodes << "\n";
485         // from the first line, for each node, print the color class it was
              assigned to
486         for (i = 0;i < numNodes;i++) solStrm << i + 1 << ' ' << colNode[i] <<
              "\n";
487         solStrm.close();
488     }
489
490     //Stop the timer.
491     clock_t runFinish = clock();
492     int duration = (int)(((runFinish - runStart) / double(CLOCKS_PER_SEC)) *
          1000);
493
494     if (verbose >= 1) cout << " COLS     CPU-TIME(ms)\tCHECKS" << endl;
495     if (verbose >= 1) cout << setw(5) << candSol.size() << setw(11) << duration
          << "ms\t" << numConfChecks << endl;
496     if (verbose >= 2) {
497         prettyPrintSolution(candSol, numNodesIndepSet, numNodes, numEdges,
              maximal, setseed);
498         checkSolution(candSol, adjacent, numNodes);
499     }
500
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 13

```
501
502     // Create a new graph from the precolored vertices by adding a complete graph K_k, k = numColorsChosen
503     // Add an edge from every precolored vertex to all nodes in K_k except its own color
504     // Output: A converted precolored graph in DIMACS format with k = p
505     for (i = 1; i < argc; i++) {
506         if (strcmp("-r", argv[i]) == 0) {
507             randomSeed = atoi(argv[++i]);
508         }
509         else if (strcmp("-v", argv[i]) == 0) {
510             verbose++;
511         }
512         else if (strcmp("-p", argv[i]) == 0) {
513             numPrecolorsChosen = atoi(argv[++i]);
514         }
515         else {
516             ofstream resultsLog("resultsLog.log", ios::app);
517             resultsLog << argv[i] << " sizeOfMaxIndepSet " << numNodesIndepSet.size() << " numUniqueColors " << candSol.size() << " numColorsChosen "<< numColorsChosen << endl;
518             resultsLog.close();
519
520
521             //Set up input file, read, and close (input must be in DIMACS format)
522             ifstream in1;
523             in1.open(argv[i]);
524
525             ofstream newStrm;
526             newStrm.open("newgraph.txt");
527
528             newStrm << "c Adapting Precoloring Extensions problem to Graph Coloring problem\n"
529                 << "c \n"
530                 << "c Initial input graph has [" << numNodes << "] nodes and [" << numEdges << "] edges.\n"
531                 << "c A total of [" << numNodesIndepSet.size() << "] nodes has been precolored with [" << numPrecolorsChosen << "] distinct number of colors.\n"
532                 << "c Your input graph has been precolored and transformed.\n"
533                 << "c It can now be run as an ordinary Graph Coloring Problem.\n"
534                 << "c newgraph.txt is in DIMACS format.\n"
535                 << "c ************************************************************ **********\n"
536                 << "a " << numPrecolorsChosen << "\n";
537                 for (int i = 0; i < numNodesIndepSet.size(); i++) {
538                     newStrm << "d " << numNodesIndepSet[i] +1 << "\n";
539                 }
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 14

```
540
541
542             //newStrm << numNodes << '+' << candSol.size() << "\n";
543             newStrm << "p edge " << numNodes + candSol.size() << ' ' << numEdges
                  + ((candSol.size())*(candSol.size() - 1) / 2) << "\n";
544
545             vector<int>::iterator pos1;
546             pos1 = max_element(colNode.begin(), colNode.end());
547
548             for (i = 0;i < numNodes;i++) {
549                 if (colNode[i] != -1) {
550                     for (int j = 0; j < *pos1 + 1; j++) {
551                         // add an edge between precolored node and all nodes in
                        K_k except its own color
552                         if (j != colNode[i]) {
553                             // i+1 because edges are defined with nodes that
                            start from index 1
554                             // numNodes+1 because adding a complete graph to
                            existing graph requires new nodes to start from index
                            numNodes+1
555                             //FOR DEBUGGING: newStrm << "e " << i + 1 << ' ' <<
                            numNodes + 1 << '+' << j << ' ' << "\n";
556                             newStrm << "e " << i + 1 << ' ' << numNodes + 1 + j
                            << "\n";
557                         }
558                     }
559                 }
560             }
561
562             // Specify edges for first K_k
563             for (int g = numNodes + 1; g < numNodes + candSol.size() + 1; g++) {
564                 for (int f = numNodes + 1; f < numNodes + candSol.size() + 1; f+
                    +) {
565                     if (f != g) {
566                         newStrm << "e " << g << ' ' << f << "\n";
567                     }
568                 }
569             }
570
571
572
573             // Combine the K_k and new edges to original graph.txt file
574             std::string line;
575             while (std::getline(in1, line))
576             {
577                 if (line.find('e') == 0)
578                     // copy line to output
579                     newStrm << line << '\n';
580                 //continue;
```

...nts\C++_projects\Prext_to_GCP\Prext_to_GCP\PrextToGCP.cpp 15

```
581             }
582             in1.close();
583             newStrm.close();
584 
585             // if k is specified, algorithm outputs newnewgraph.txt for the
                  specified k.
586             if (prepareForGCP = 2) {
587                 preStepsGCP(numNodes, numEdges, numNodesIndepSet,
                      numPrecolorsChosen, numColorsChosen, many);
588 
589             }
590 
591             // Produce a colorsolution.txt file, which shows the indices of
                  vertices in their respective color classes
592             ofstream colStrm;
593             colStrm.open("colorsolution.txt");
594             colStrm << candSol.size() << "\n";
595 
596             int k, count = 0, group;
597             for (group = 0; group < candSol.size(); group++) {
598                 colStrm << "C-" << group << "\t= {";
599                 if (candSol[group].size() == 0) colStrm << "empty}\n";
600                 else {
601                     for (k = 0; k < candSol[group].size() - 1; k++) {
602                         colStrm << candSol[group][k]+1 << ", ";
603                     }
604                     colStrm << candSol[group][candSol[group].size() - 1]+1 << "}
                          \n";
605                     count = count + candSol[group].size();
606                 }
607             }
608             colStrm << "Total Number of Nodes and Edges = " << '(' << numNodes <<
                    ", " << numEdges << ')' << endl;
609             colStrm << "Number of Nodes in maximal independent set = " <<
                  numNodesIndepSet.size() << endl;
610             colStrm << "Precolored vertices have " << candSol.size() << " unique
                  number of color(s)." << endl;
611             colStrm.close();
612 
613         }
614     }
615 }
616 
```

...\Prext_to_GCP\Prext_to_GCP\PreGCPFixedKTransformation.cpp 1

```
1  #include "stdafx.h"
2  #include <string.h>
3  #include <fstream>
4  #include <iostream>
5  #include <vector>
6  #include <stdlib.h>
7  #include <time.h>
8  #include <limits.h>
9  #include <iomanip>
10 #include <algorithm>
11 #include <string>
12 using namespace std;
13
14 // Before a precolored graph can be put into a graph coloring algorithm, this function calls precoloring converter for fixed k
15 void preStepsGCP(int &numNodes, int &numEdges, vector<int> &numNodesIndepSet, int &numPrecolorsChosen, int &numColorsChosen, int &many) {
16
17    int orig_numColorsChosen = numColorsChosen;
18
19    // for loop to generate -n files
20    for (int i = 0; i < many+1; i++) {
21        // Open newgraph.txt and copy its contents to newnewgraph.txt
22        ifstream in1;
23        in1.open("newgraph.txt");
24
25        numColorsChosen = orig_numColorsChosen + i;
26        ofstream newStrm;
27        newStrm.open("newnewgraph"+to_string(numColorsChosen) + ".txt");
28        newStrm << "c A Graph Transformation of Precoloring Extensions to run on ordinary Graph Coloring Algorithms\n"
29            << "c \n"
30            << "c Initial input graph has [" << numNodes << "] nodes and [" << numEdges << "] edges.\n"
31            << "c [" << numColorsChosen << "] number of colors has been selected to test.\n"
32            //<< "c A total of [" << numNodesIndepSet.size() << "] nodes has been precolored with [" << numPrecolorsChosen << "] distinct number of colors.\n"
33            << "c A total of [" << numNodesIndepSet.size() << "] nodes has been precolored with [" << numPrecolorsChosen << "] distinct number of colors.\n"
34            << "c Your input graph has been precolored and transformed.\n"
35            << "c It can now be run as an ordinary Graph Coloring Problem.\n"
36            << "c This graph is in DIMACS format.\n"
37            << "c ********************************************************************************\n";
38
```

...\Prext_to_GCP\Prext_to_GCP\PreGCPFixedKTransformation.cpp 2

```
39          newStrm << "p edge " << numNodes + numColorsChosen << ' '
40              << numEdges + ((numNodes + numColorsChosen)*(numNodes +
                  numColorsChosen - 1) / 2) << "\n";
41
42          // Specify edges for K_k
43          for (int d = numNodes + 1; d < numNodes + numColorsChosen + 1; d++) {
44              for (int f = numNodes + 1; f < numNodes + numColorsChosen + 1; f++) {
45                  if (f != d) {
46                      newStrm << "e " << d << ' ' << f << "\n";
47                  }
48                  //for (int m = g.n - g.numPrecoloredNodes + 1; m < g.n + 1; m++)
49                  //  newStrm << "e " << d << ' ' << m << "\n";
50              }
51          }
52          for (int i = 0; i < numNodesIndepSet.size(); i++) {
53              for (int s = numNodes + 1 + numPrecolorsChosen; s < numNodes +
                    numColorsChosen + 1; s++) {
54                  newStrm << "e " << numNodesIndepSet[i] + 1 << ' ' << s << "\n";
55              }
56          }
57
58          // Combine the K_k and new edges
59          std::string line;
60          while (std::getline(in1, line))
61          {
62              if (line.find('e') == 0)
63                  // copy line to output
64                  newStrm << line << '\n';
65              //continue;
66          }
67
68          in1.close();
69          newStrm.close();
70      }
71
72  }
73
```