Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Santiago Miguel Palladino

Algoritmo de Branch and Cut para el Problema de Coloreo Particionado

Directoras

Isabel Méndez Díaz
Paula Lorena Zabala

Mayo 2011

Agradecimientos

A Isabel y Paula, mis directoras, no solamente por la guía indispensable que fueron para el desarrollo de este trabajo (con notable paciencia), sino por haberme enseñado todo lo que sé acerca de Métodos Numéricos e Investigación Operativa, y haberme transmitido su gran pasión por estas áreas. Su trabajo y dedicación como directoras es solamente comparable a su desempeño como las excelentes docentes que son.

A Irene Loiseau y Esteban Feuerstein, por haber aceptado ser jurados de este trabajo, y por las correcciones que permitieron concluirlo como corresponde.

A todo el Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la UBA, especialmente a sus docentes, por haberme brindado una educación excelente y gratuita; así como a los grupos de Algoritmos y Métodos Numéricos, por darme un espacio como ayudante en el cual devolver una pequeña parte de lo aprendido.

A mis compañeros de la cursada, la gente-de-la-facu, junto a quienes recorrí la carrera y a quienes les debo las alegrías dentro de la facultad en estos últimos seis años: Matías Blanco, Fernando Bugni, Luis Brassara, Facundo Carreiro, Bruno Cuervo Parrino, Diego Freijo, Maxi Giusto, Pablo Laciana, Sergio Medina, Ariel Neisen, Leo Rodríguez, Nati Rodriguez, Viviana Siless, Javier Silveira, Leo Spett, Pablo Zaidenvoren, Eddy Zoppi; con una mención más que especial para Leandro Radusky, Andrés Taraciuk y Martín Verzilli, mis compañeros infaltables de TPs, con quienes aprendí a trabajar codo a codo y pudimos llevar adelante todos los trabajos con los que nos enfrentamos.

A la gente de investigación de la facultad, por haberme hecho un lugar dentro de la gran familia que representan: Nicolás Botbol, Juan Manuel Chaneton, Alexandra Diehl, Marina Groshaus, Fernando Hernández, Leandro Montero, Mercedes Pérez Millán, Juan Pablo Puppo, Francisco Soulignac, Juanjo Miranda y Pablo Factorovich; especialmente a estos dos últimos por haberme ayudado en incontables ocasiones y servido de ejemplo a seguir acerca de lo que significa el desarrollo en Investigación Operativa.

A Manas, mi lugar de trabajo en la industria desde hace más de tres años: como empresa, por haberme brindado un espacio dentro de la misma para el desarrollo de mi tesis; y más especialmente, como grupo humano, por ser un lugar donde trabajar y compartir el día a día con excelentes profesionales y aún mejores personas. Una mención especial para Caro Hadad, por la paciencia para hacer de audiencia ante los ensayos de la defensa de este trabajo.

A mis amigos que aún están a mi lado desde el colegio: Mariana Lavia, Virginia Raschia, Gaby Revale, Pablo Cagnoni, Mauro Lampo, Ale Maggi, Nicolás Muschirintello, Martín Kalos, Leandro Paizal y Hernán Sánchez; con quienes hemos compartido ya la mitad de lo que va de la vida, si no más en algunos casos, y a quienes agradezco por su amistad durante todo este tiempo.

A mis padres y a Mariano, mi hermano; por el apoyo incondicional siempre, por haberme hecho la persona que soy, y por haberme permitido llegar hasta esta etapa de mi vida.

A todos ellos, y a todos los que fueron una parte de esta larga carrera o de los pasos para llegar hasta ella, muchas gracias!!

Resumen

El problema de coloreo particionado, PCP, es una generalización del clásico problema de coloreo de grafos. En esta variante el conjunto de nodos del grafo de entrada se encuentra particionado, y el problema consiste en colorear un solo nodo por partición utilizando la menor cantidad de colores posible, manteniendo la restricción de que dos nodos adyacentes no pueden compartir color.

Este problema fue propuesto por Li y Simha en el contexto del problema de *ruteo y asignación de longitudes de onda* (RWA) en redes *multiplexadas por división de longitud de onda* (WDM). Dichos autores proponen una resolución en dos etapas: una primera en la que se generan posibles caminos como soluciones factibles para el problema de ruteo, y una segunda en la que se determinan los caminos a usar y se les asignan longitudes de onda, buscando minimizar la cantidad de longitudes de onda usadas. Esta última etapa se corresponde con una instancia del PCP.

El PCP, al igual que coloreo tradicional de grafos, es un problema NP completo, con lo que no se conoce un algoritmo que pueda resolverlo en tiempo polinomial. Por este motivo, la mayoría de los enfoques para resolver este problema se basan en técnicas heurísticas, dejando poco lugar a algoritmos exactos para la resolución del mismo.

En este trabajo modelamos el PCP como un problema de programación lineal entera, generalizando el modelo propuesto por Méndez-Díaz y Zabala para coloreo de grafos, lo que nos permite resolverlo mediante la técnica de *branch and cut*. Para ello, desarrollamos una heurística inicial, una heurística primal, estrategias de branching, y algoritmos de separación para distintas familias de desigualdades válidas que caracterizamos para el poliedro. A partir de estos componentes implementamos el algoritmo de *branch and cut* para la resolución del PCP.

Abstract

The partitioned graph coloring problem, PCP, is a generalization of the classic graph coloring problem. In this variant the set of the input graph's nodes is partitioned, and the problem relies in coloring exactly one node per partition using the lowest possible number of different colors, maintaining the constraint that two adjacent nodes may not use the same color.

This problem was first stated by Li and Simha in the context of the *routing and wavelength assignment* (RWA) problem in *wavelength division multiplexed* (WDM) networks. The authors propose a two-stage resolution: a first stage in which possible *lightpaths* are generated, which are feasible solutions to the routing problem, and a second stage where the lightpaths to be used are selected and each of them is assigned a wavelength, looking to minimize the number of different wavelengths. This last stage can be modelled as an instance of the PCP.

The PCP, like traditional graph coloring, is an NP complete problem, which means that there are no polynomial algorithms known for its resolution. Therefore, most of the work on this problem in the literature is targeted towards heuristic approaches, with only a few efforts for developing exact algorithms.

In this work we modelled the PCP as an integer linear programming problem, generalizing the model proposed by Méndez-Díaz and Zabala for graph coloring, which can be solved via *branch and cut* algorithms. In order to develop such an algorithm, we implemented an initial heuristic, a primal heuristic, branching strategies and separation algorithms for the families of valid inequalities we found; these components were the building blocks for our *branch and cut* algorithm for solving the PCP.

Resumen Extendido

Una red óptica de tipo WDM (Wavelength Division Multiplexing, o multicanalización por división de longitud de onda) permite la transmisión simultánea de distintos paquetes de datos a través de una misma fibra utilizando distintas longitudes de onda dentro de la misma.

La conexión punto a punto entre dos nodos de la red se denomina, en estos casos, lightpath (camino óptico). Si bien en estas redes existen tipos de nodos más avanzados con la capacidad de modificar la longitud de onda usada por un lightpath durante su recorrido, a efectos del problema nos concentraremos en la versión más sencilla en la que el lightpath utiliza la misma longitud en todo su recorrido, imponiendo lo que se denomina la wavelength continuity constraint (restricción de continuidad de longitud de onda).

Dado un segmento de fibra, no puede ocurrir que dos lightpaths distintos intenten transmitir en la misma longitud de onda, o habrá colisión entre los datos enviados. Esto impone la denominada wavelength clash constraint (restricción de conflicto de longitud de onda), lo que sumado a la anterior lleva a que dos lightpaths distintos no pueden tener la misma longitud de onda si comparten algún segmento de fibra óptica.

A partir de estas restricciones se genera el problema de RWA (Routing and Wavelength Assignment, o ruteo y asignación de longitudes de onda), que consiste en, dado un conjunto de conexiones a satisfacer, determinar los enlaces a utilizar y longitud de onda para cada uno de los caminos ópticos.

Si bien existe una gran cantidad de soluciones para el problema anterior, se busca alguna que sea óptima en un determinado criterio. En este caso nos enfocamos en minimizar la cantidad de longitudes de onda requeridas para satisfacer las conexiones, lo cual permite utilizar fibras de menor costo al tener que soportar una menor cantidad de longitudes de onda distintas por enlace. Este problema es el denominado min-RWA.

El RWA suele ser atacado bajo dos enfoques: o bien como un único problema en el que se busca resolver simultáneamente el ruteo y la asignación de longitudes de onda, o bien como un problema de dos etapas en las que se separan dichas fases. Es esta última opción en la que nos concentramos en este trabajo.

En trabajos como [13] y [18] se buscan primero los caminos a utilizar para el ruteo, utilizando criterios de camino mínimo o edge disjoint path respectivamente, para luego resolver la asignación de longitudes de onda mediante un problema de coloreo de grafos.

El problema de coloreo de grafos, extensamente estudiado en la literatura, consiste en asignar un color a cada nodo de un grafo con la restricción de que

dos nodos adyacentes no tengan el mismo color, con el objetivo de utilizar la menor cantidad de colores distintos como sea posible. Modelando cada lightpath como un nodo, y conectando dos nodos si corresponden a lightpaths que comparten al menos un segmento de fibra óptica, es posible resolver el problema de asignación de manera óptima.

Li y Simha, por otra parte, proponen en [17] generar un conjunto de caminos candidatos para cada conexión en la primera fase. Es decir, la etapa de ruteo no genera un lightpath por cada pedido de conexión, sino varios, todos ellos posibles candidatos. Esto implica que en la segunda etapa se derive una instancia del problema de coloreo particionado, cuya resolución es el principal objetivo de trabajo.

El problema de coloreo particionado toma un grafo en el que el conjunto de nodos se encuentra particionado, y tiene por objetivo asignar un color a un solo nodo de cada partición, de manera que se mantenga la restricción de que dos nodos adyacentes tengan distinto color y se minimice la cantidad de colores distintos usada. Es fácil notar que este problema es una generalización del problema de coloreo de grafos tradicional, ya demostrado por Karp en [14] que es NP-Completo, lo que implica que su resolución de manera exacta requiere un tiempo de procesamiento exponencial en el tamaño del grafo de entrada.

Considerando cada lightpath posible como un nodo, agrupados en una misma partición si satisfacen el mismo pedido de conexión punto a punto, es posible resolver el problema de asignación utilizando la menor cantidad de longitudes de onda posibles para los candidatos propuestos. El manejar conjuntos de lightpaths para la fase de asignación, en lugar de un solo lightpath por conexión, permite llegar a una mejor solución para el problema.

Buena parte del trabajo sobre resolución del problema de coloreo particionado (PCP) es heurístico. Solamente en [9], Frota et al presentan un algoritmo de branch and cut para su resolución de manera exacta, basándose en un modelo de programación lineal entera, generalizado a partir del modelo de coloreo por representantes, desarrollado en [5] y [6].

La programación lineal entera es una técnica usada frecuentemente para atacar la resolución exacta de problemas de optimización combinatoria, en los que una exploración exhaustiva de las diferentes soluciones en busca del óptimo se vuelve impracticable debido a la explosión exponencial ocurrida al enumerar dichas soluciones. Se suelen utilizar algoritmos de planos de corte, de branch and bound o una combinación de ambos, como ser cut and branch o branch and cut.

En este trabajo fue desarrollado un algoritmo de branch and cut para resolver el problema de forma exacta, utilizando un modelo de programación lineal entera basado en el desarrollado para el problema de coloreo tradicional por Méndez-Díaz y Zabala en [21]. La elaboración de un algoritmo de branch and cut específico para un determinado problema requirió el desarrollo de los siguientes componentes:

- La formulación de un modelo lineal entero para el problema. Distintas variantes generalizadas a partir del modelo de coloreo tradicional, incluyendo restricciones de eliminación de simetría, entre otras, fueron evaluadas sobre distintas instancias del problema hasta arribar a una formulación definitiva.

- La búsqueda de desigualdades válidas para el modelo elegido. Una vez determinado el modelo, se buscan desigualdades satisfechas por toda solución entera del modelo pero no así por las fraccionarias. Esto permite luego aplicar dichas desigualdades eliminando los puntos fraccionarios, derivando en un algoritmo de planos de corte. Generalizando desde las familias conocidas para coloreo tradicional, hallamos seis familias distintas de desigualdades válidas.

- La implementación de heurísticas de separación para las desigualdades válidas halladas. Dada una solución fraccionaria, se debe hallar una desigualdad válida de alguna de las familias desarrolladas que sea violada por dicha solución, de manera de agregarla al modelo de la relajación y moverse a otra solución. Se implementaron heurísticas para cada una de las familias encontradas.

- Una heurística inicial para obtener rápidamente una solución inicial aceptable para el algoritmo de branch and cut. Esto permite reducir notablemente el tamaño del modelo sobre el cual se trabaja. Como heurística inicial se evaluaron distintas alternativas, hasta optar por una variante para coloreo particionado del algoritmo DSATUR [4] que desarrollamos. Este algoritmo es de enumeración implícita, con lo que genera una solución exacta dado el suficiente tiempo, pero tiene la particularidad de hallar buenas soluciones con poco tiempo de ejecución, lo que lo hace un excelente candidato como heurística, interrumpiendo su ejecución luego de determinado tiempo.

- Una heurística primal para derivar soluciones enteras intermedias a partir de las soluciones de las relajaciones obtenidas a lo largo del árbol de branching. Para esto reutilizamos la heurística inicial, modificándola de manera tal que reaprovechase la información provista por la solución de la relajación en su exploración del conjunto de soluciones.

- Una estrategia para generación del árbol de branching. En cada nodo, debe determinarse cuántos nodos hijos se abren y con qué criterio. Si bien se suele forzar una variable binaria con valor fraccionario a cero y a uno en cada hijo, hay otras alternativas posibles. Luego de evaluar distintas estrategias, optamos por una que fuerza no solo una determinada variable sino todas las demás implicadas lógicamente; además de establecer un criterio de selección de dicha variable en función de su grado de saturación y valor fraccional.

- Una estrategia para el recorrido y poda del árbol. Evaluamos opciones clásicas como DFS, BFS o best bound sobre distintas instancias hasta determinar cuál se comportaba mejor dentro de nuestro esquema de branch and cut dependiendo de la densidad del grafo. Asimismo efectuamos una poda del árbol de enumeración una vez alcanzada altura suficiente, ejecutando una corrida exhaustiva de DSATUR, la cual ejecuta a una mayor velocidad que el branch and cut una vez fijada la suficiente cantidad de colores.

Todos estos componentes fueron implementados y sus distintas variantes evaluadas sobre múltiples instancias. La implementación fue realizada en Java utilizando como framework CPLEX 12.

La versión final del algoritmo fue evaluada contra los motor de mixed integer programming search y dynamic search de CPLEX 12 sin realizar modificaciones sobre la configuración original, y se verificó que el algoritmo desarrollado para PCP obtiene mejores gaps y tiempos, con lo que los componentes desarrollados efectivamente favorecen la resolución del problema. Asimismo evaluamos el rendimiento del algoritmo contra el reportado por el branch and cut basado en el modelo de representantes [9].

Contents

Chapter 1

Introduction

1.1 Coloring

Needless to say, graphs are widely used for modeling different scenarios in multiple areas of expertise, as well as for solving problems on those scenarios by translating them into well-known problems.

One of those problems is the graph coloring problem, which consists in assigning a color to each node in a graph, with the constraint that two adjacent nodes must not have the same color. The objective is to generate a valid coloring using the minimum number of colors.

One of the most famous real life problems which led to the graph coloring problem was the *4 colors problem*. In 1852, the question of whether any planar map could be colored using only four colors, in such a way that no two regions sharing a border had the same color, was posed. Modeling neighbour regions as adjacent nodes in a planar graph led to the planar graph coloring problem, which was eventually generalized into coloring a generic graph.

Graph coloring is widely used in multiple applications, such as schedule assignment to solve time incompatibilities, assignment of radio frequencies to prevent interference between neighboring radios, or even assigning variables to registers during the flow of a program.

The coloring of a graph is defined formally as a function that, given an input graph $G =< V, E >$, being V the set of nodes and E the set of undirected edges, assigns a natural number which represents a color to each node $v \in V$, such that no two adjacent nodes have the same color. A *k-coloring* is an assignment which uses exactly k different colors.

The *chromatic number* $\chi(G)$ of a graph is the minimum number of colors that can be used to color the graph, this is, the minimum k such that a valid k-coloring exists. The *graph coloring problem*, then, is defined as the problem of finding both the minimum number of colors required to color a

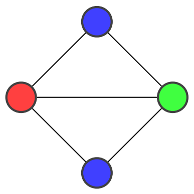


Figure 1.1: Sample 3-coloring of a diamond graph.

graph ($\chi(G)$), and a valid $\chi(G)$-coloring.

This problem has been proved to be *NP-Complete*, and has been widely studied in the literature, being approached both by heuristic and exact methods for its resolution.

Previous Work

Simplest heuristic approaches consist in greedy algorithms, using different criteria such as *largest-first* [30], *smallest-last* [19] or *degree of saturation*[1] [4]. While the first two rely on a static ordering based on the degree of each node, the last one uses a dynamic ordering based on the number of different colors being used in the neighbourhood of each vertex.

These criteria may also be used in implicit enumeration techniques for choosing the ordering of the nodes. These techniques enumerate all possible colorings by constructing a decision tree: each node of the tree represents a node of the original graph, and each decision consists in which color is to be assigned to the current node. Using a good strategy for deciding the order for picking the next node to be colored is vital for finding good solutions as soon as possible, therefore pruning a great number of possible colorings. The implicit decision tree may be traversed in a BFS, DFS or best bound fashion. All of these algorithms eventually find the optimal solution for the problem.

The DSATUR enumeration algorithm (proposed in [4]), which uses degree of saturation criteria for picking the next node to be colored, has proven to be one of the most efficient implicit enumeration methods for the coloring problem; having several improvements such as [26].

More complex heuristic algorithms, using different metaheuristics, have also been used for the coloring problem.

There is also extensive work using integer linear programming formulations for the coloring problem by using different models:

- In [20] a column generation approach is used based on an independent set formulation of the problem, in which a binary variable x_S defines

[1]The degree of saturation of a node is defined as the number of different colors being used to color its neighbourhood.

whether the independent set S is given a color label or not; this formulation requires a variable for each possible color class in the graph.

- An ILP model for acyclic orientations with path constraints is presented in [8] and then applied to solve the vertex coloring problem.
- The representatives model presented in [6] and [5] uses x_{uv} variables which determine whether vertex v *represents* color u; having exactly one node represent each color class allows easy symmetry breaking.
- In [21, 22] both branch-and-cut and cutting planes algorithms were developed for a standard formulation of the problem, using x_{ij} variables to specify whether node i used color j, and w_j variables as witnesses to whether color j was in use. Several symmetry breaking constraints were added to the model to ensure a fast resolution.

Application to Frequency Assignment

As it has already been mentioned, the graph colouring problem has multiple applications, amongst which is the problem of assigning frequencies to a given set of nodes in order to establish communications between them, considering availability of frequencies, interference, etc. This problem, called the *frequency assignment problem*, has multiple variants, which lead to different generalizations of the graph coloring problem.

In this work we will consider its application to routing and wavelength assignment in WDM networks, which gives rise to the *partitioned coloring problem.*

1.2 Routing and Wavelength Assignment in WDM Networks

A Wavelength Division Multiplexed (WDM) optical network consists in a network in which links are optical fibers capable of transmitting a specified number of different wavelengths. The Routing and Wavelength Assignment (RWA) problem consists in, given a desired set of connections between pairs of nodes, establish routes between those nodes using the network's links.

Every route is composed by a set of consecutive lightpaths. A lightpath is defined as a point to point connection between two adjacent nodes in the network using a certain wavelength. Although there are networks in which the nodes are capable of transforming wavelengths within the same route, we will assume that every route uses the same wavelength across all of its lightpaths; this restriction is known as the *wavelength continuity constraint.*

The second restriction to be satisfied is the *wavelength clash constraint* which imposes that different lightpaths in the same physical link must have different wavelengths. Together with the previous constraint, it is implied that two different routes that share at least one physical link must use different wavelengths.

In the offline or static version of the RWA problem, the set of connections to be established is known beforehand. The counterpart of this version is the *dynamic* RWA in which connections must be satisfied as they are requested in an online fashion. In this work we will take only the former version into consideration.

The goal of the min-RWA is to minimize the number of different wavelengths required to establish all the routes desired. Note that there are multiple criteria that can be used to evaluate the quality of a set of routes, such as the number of lightpaths used for each route, or generating particular traffic patterns. In this work we will be focusing only in optimizing the number of wavelengths.

Previous Work

Initial techniques to solve the min-RWA problem as a two-stage problem, such as [13], pick a single route for every connection using shortest-path algorithms and then use different heuristics to solve a standard coloring problem in the assignment stage. In [18] the shortest-path routing solution is replaced by a maximum edge disjoint path solution in order to reduce conflicts between routes.

Other approaches to the problem tackle the routing and wavelength assignment as a single problem, without decomposing it in two separate phases. In [27], for example, bin packing heuristic algorithms are used to handle the problem, whereas [24] embeds this heuristic into a genetic evolutionary framework.

An exact approach using an integer programming formulation with column generation is used in [16], which solves both the routing and the wavelength assignment problems in the same formulation.

Resolution of min-RWA using graph coloring

In [17], Li and Simha proposed a two-phase approach for solving the min-RWA problem: a routing phase and an assignment phase. In the routing phase, a set of candidate routes is generated for every pair of nodes to be connected, mostly using shortest-path or edge-disjoint criteria.

The assignment phase, before actually asigning a wavelength to each route, must pick a single route from the set of candidates for each connection. Each selected route is then assigned a wavelength, ensuring that no two routes sharing any physical link have the same wavelength assigned. These two processes that compose the assignment phase can be solved simultaneously through the *partitioned graph coloring problem* (PCP).

1.3 Partitioned graph coloring problem

A *partitioned graph* is defined as a tuple $G =< V, E, P >$ of n vertices, m edges and q partitions respectively. The set P contains $P_1, \ldots, P_q$ sets of nodes which constitute a partition of V. Therefore, for every node $v \in V$, there is exactly one $P_k \in P$ such that $v \in P_k$, and every $P_i \in P$ is nonempty.

The partitioned coloring problem is defined as an assignment of colors to the nodes of the graph G, with the restriction that no two adjacent nodes may have the same color, but requiring only one node per partition to be colored. Once again, the goal is to minimize the number of colors required.

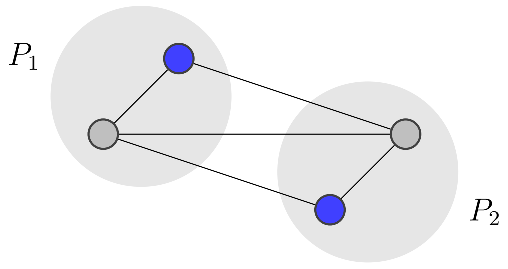


Figure 1.2: Sample 1-coloring of a partitioned diamond graph.

In order to solve the min-RWA problem using PCP, a partitioned graph G can be constructed in the following way:

- Every potential route generated in the routing phase is represented by a node $v \in V$.
- Nodes belong to the same partition iff the routes they represent satisfy the same connection request.
- An edge between two nodes u, v is created if the routes share any physical link.

Each wavelength is represented as a color. The problem then consists in coloring a single node within each partition, this is, assigning a wavelength to a single route from the set of candidates for each connection request. The fact that two nodes may not be colored if they are adjacent guarantees that no wavelength conflicts may occur between two different lightpaths in the same link. An example of this is shown in figures 1.3, 1.4 and 1.5.

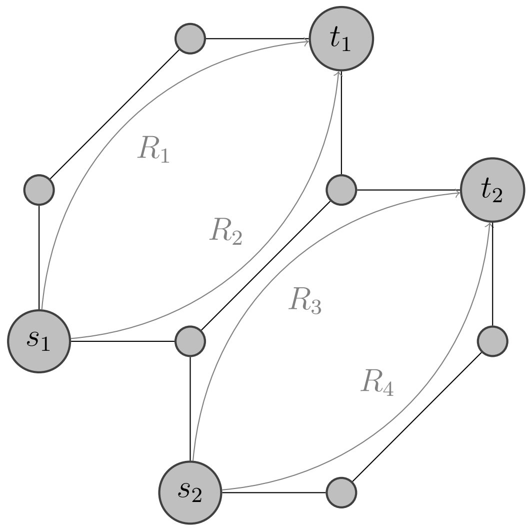


Figure 1.3: Sample network in which connections $s_1 \rightarrow t_1$ and $s_2 \rightarrow t_2$ are to be implemented. Potential routes R_1, R_2 are proposed for the first, while routes R_3, R_4 are proposed for the second one. The corresponding partitioned graph is presented in figure 1.4.

In this work we will focus on finding an exact solution for the partitioned coloring problem, using a branch and cut algorithm based on a generalization of the coloring model proposed in [21, 22].

Complexity

It is easy to see that when $|P_i| = 1 \; \forall P_i \in P$, this is, there is a single node per partition, the partitioned coloring problem is equivalent to the standard graph coloring problem previously mentioned. In terms of complexity classes, PCP belongs to the same class as the standard coloring problem.

Theorem 1. *The decision version of PCP is NP-Complete.*

Proof. We will prove NP-Completeness by proving both belonging to NP and NP-Hard classes.

- *NP:* Given an input partitioned graph $G =< V, E, P >$ and an assignment of colors for a subset of nodes, checking that the number of colors used is k is trivial, and a simple algorithm such as 1.1 can easily check the validity of the coloring in polynomial time.

- *NP-Hard:* Any instance of standard graph $k-coloring$ can be converted to an instance of PCP by partitioning the initial graph G in such a way

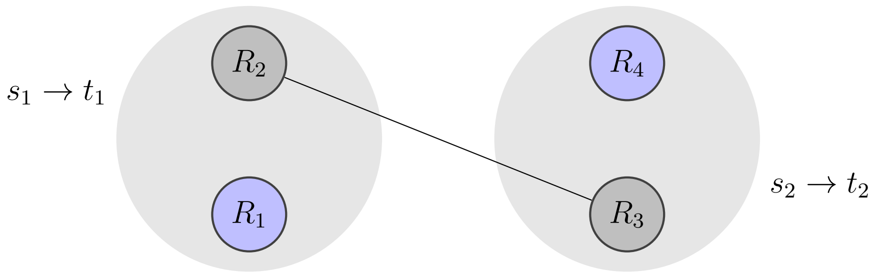


Figure 1.4: Conflicts partitioned graph for network from figure 1.3. Routes R_1 and R_2 satisfy the same connection request, as such, they are contained in the same partition; same happens for R_3 and R_4. Since routes R_2 and R_3 share a physical link, the corresponding nodes are adjacent to prevent that they are assigned the same frequency. A 1-coloring, which assigns the same label to R_1 and R_4 is shown, and the corresponding lightpaths generated are shown in figure 1.5.

that every partition contains a single node. The solution to the original $k-$*coloring* problem is the same as the solution to the constructed PCP. Since standard coloring is NP-Hard, this implies that PCP is NP-Hard as well.

□

Algorithm 1.1 Polynomial time algorithm for checking validity of a partition coloring

```
for all partition $p$ in $P$ do
  for all node $v$ in $p$ do
    if $v$ has a color $j$ assigned then
      mark $p$ as colored
      for all neighbour $u$ to $v$ do
        if $u$ has the same color assigned as $v$ then
          return false
  if no node $v$ in $P$ was colored then
    return false
```

Previous work

In [17], two groups of heuristics were developed for solving the PCP: one-step and two-step. The former iteratively picks the easiest node in every partition, and then picks the hardest one from that set using different criteria

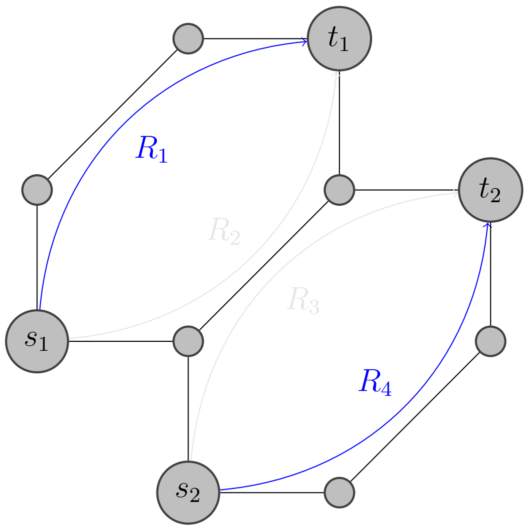


Figure 1.5: Solution for the network presented in figure 1.3 using the coloring obtained in 1.4. Since R_1 and R_4 were the colored nodes, using the same label, then those are the routes established and lightpaths using that label are created to satisfy the connection requests.

(largest-first, smallest-last, color-degree) in order to color it with the lowest-label available color, and then proceeds to the next node; the latter makes an initial pass picking the easiest nodes in every partition and inducing a non-partitioned graph, onto which a standard heuristic is applied in a second stage.

In [25] the one-step color-degree constructive heuristic is used in a tabu search approach, TS-PCP. Routes are generated in an initial stage using a k-EDR constructive procedure, based on the maximum edge disjoint path heuristic by [15], and the resulting partitioned coloring problem is solved with TS-PCP.

Due to the complexity of the problem, most of the work on PCP is composed by heuristic approaches. However, in [9], a branch and cut algorithm is devised, using an integer linear programming model based on the asymmetric representatives formulation for the standard coloring problem, presented in [6] and [5].

1.4 Objective of this work

A common approach for obtaining exact solutions to complex combinatorial optimization problems is to model them as integer linear programming problems, and solve them using branch and bound, cutting planes or branch and cut algorithms, among others.

Therefore, the objective of this work will be to develop a branch and cut algorithm for solving the partitioned coloring problem, by modelling it as an integer linear programming problem, with a generalization of the standard coloring model presented in [21, 22].

We will be using CPLEX as a branch and cut framework and introduce custom initial heuristics, cutting planes, primal heuristics and branching strategies, designed specifically for this problem, and evaluate their performance against the default implementation provided by CPLEX.

This work is structured in seven chapters. In chapter 2 we present different models for representing an instance of PCP, starting with a basic model that captures all necessary restrictions, for later strengthening it by modifying the constraints with stronger ones or introducing new ones, such as symmetry breaking constraints.

Chapter 3 dwelves deeper into the polyhedron defined in the previous chapter by presenting valid inequalities derived for PCP. These inequalities will be later applied as cutting planes in both cutting planes and branch and cut algorithms.

Alternative algorithms for solving the problem are presented in chapter 4. We present enumeration algorithms for solving the standard coloring problem, and generalize them for partition coloring, focusing in the DSATUR algorithm [4] and its generalization. These algorithms will be adapted to be used as initial and primal heuristics during the branch and cut process.

The implemented branch and cut algorithm is presented in chapter 5, where we present the general structure for cutting plane, branch and bound and branch and cut algorithms, as well as different components of a branch and cut (separation heuristics, initial heuristics, primal heuristics, branching strategies, node selection strategies) and how we implemented them for dealing with the PCP.

This implementation is then tested in chapter 6. Given a test suite of binomial, powerlaw cluster and DIMACS challange graphs, we first evaluated multiple configurations for all of the different components, starting by choosing a model to be used in the algorithm, and testing the effectiveness of the different heuristics and strategies with different parameterizations. We then test the performance of the algorithm, once we have all parameters fixed, against a fresh test suite and report the obtained MIP gap.

Finally, in chapter 7, we sum up the work achieved, draw conclusions from it and present possible future research lines.

1.5 Definitions

In this section we will define all concepts and conventions to be used throughout this work:

- **Colors:** The set of valid color labels $C = \{1, \ldots, c\}$, where c may be any upper bound to the chromatic number of the graph, such as n.
- **Graph:** Defined as tuple $< V, E >$ where V is the set containing the n nodes and E contains the m undirected edges.
- **Partitioned Graph:** Defined as tuple $< V, E, P >$, being V and E the same sets as above, and P the set of $P_1, \ldots, P_q$ partitions of V.
- **Partition function:** For every node v in a partitioned graph, $p(v)$ returns the partition that contains that node.
- **Neighbourhood:** $N(v)$ is the set of nodes in V adjacent to node v.
- **Partition Neighbourhood:** $N_P(v)$ is the set of partitions that contain at least one node adjacent to v.
- **Degree:** $\delta(v)$ is the cardinal of the neighbourhood of v.
- **Partition Degree:** $\delta_P(v)$ is the cardinal of the partition neighbourhood of v (see figure 1.6).

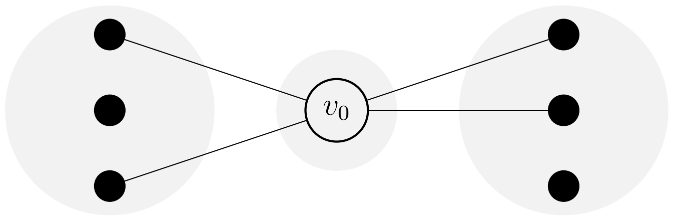


Figure 1.6: Node v_0 has partition degree $\delta_P(v_0) = 2$.

- **Color Degree:** Number of different colors used in $N(v)$ for a node $v \in V$; also degree of saturation (see figure 1.7).
- **Path:** Subset P of V such that each node is adjacent only to the next one in the path in the subgraph induced by V' in G; formally, being $P = \{v_1, \ldots, v_k\}$, P is a path if $G[P]$ has edges $[v_i, v_{i+1}]$ for $1 \leq i < k$, and no more edges between vertices of P (see figure 1.8).

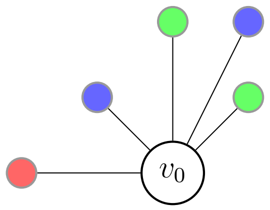


Figure 1.7: Node v_0 has degree of saturation 3.

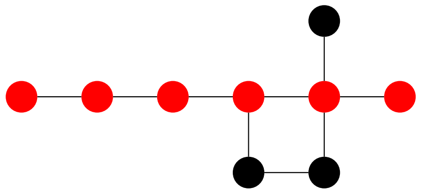


Figure 1.8: Nodes highlighted in red form a path.

- **Hole:** Subset H of V such that each node is adjacent only to the next one in the hole in the subgraph induced by V' in G, and the last node is adjacent to the first; formally, being $H = \{v_1, \ldots, v_k\}$, H is a hole if $G[H]$ has edges $[v_1, v_k]$ and $[v_i, v_{i+1}]$ for $1 \leq i < k$, and no more edges between vertices of H (see figure 1.9).

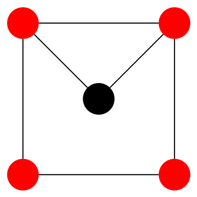


Figure 1.9: Nodes highlighted in red form a hole.

- **Component Independent Set:** Subset of V such that for every pair of nodes u, v, u is not adjacent to v and they belong to different partitions (see figure 1.10).
- **Component Path:** Path P in G that verifies that every node in P belongs to a different partition.
- **Component Hole:** Hole H in G that verifies that every node in H belongs to a different partition.
- **Component Clique:** Clique K in G that verifies that every node in K belongs to a different partition (see figure 1.11).

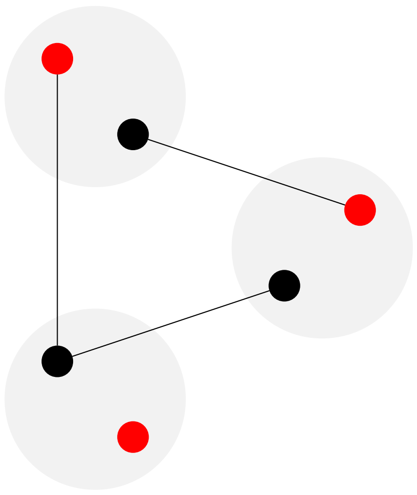


Figure 1.10: Nodes highlighted in red form a component independent set.

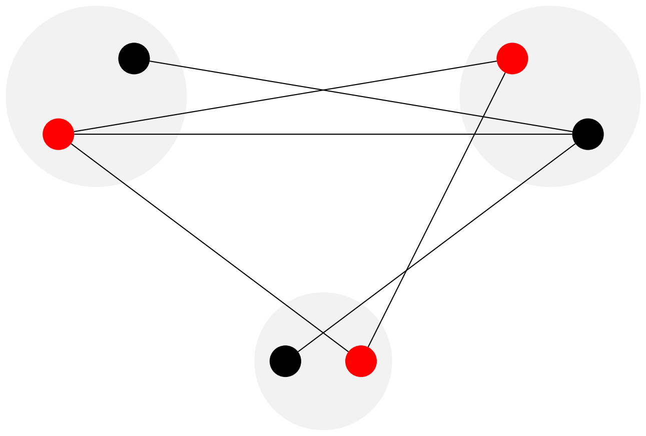


Figure 1.11: Nodes highlighted in red form a component clique.

- **Extended Clique:** Subset of V such that for every pair of nodes $[u, v]$, either u is adjacent to v, or u and v are contained in the same partition (see figure 1.12).
- **Partition Graph:** The *partition graph* G' of a partitioned graph G is a standard graph $G' =< V', E' >$ in which every node $v'_k \in V'$ corresponds to a partition $P_k \in P$, and two nodes $v'_i, v'_j \in V'$ are adjacent if and only if every node in P_i in G is adjacent to every node in P_j (see figure 1.13).

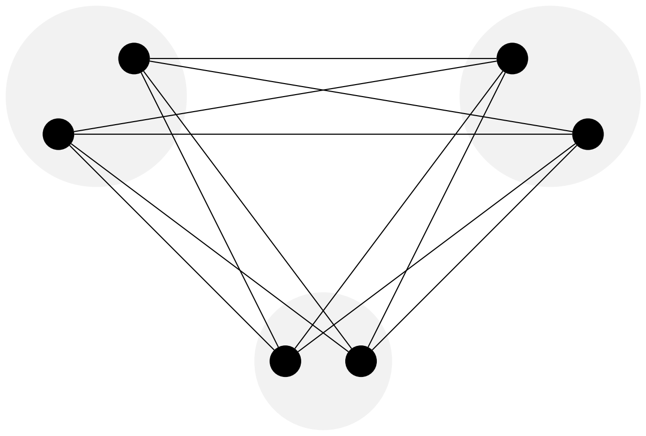
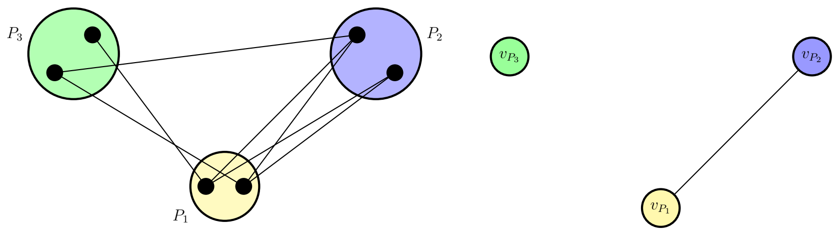


Figure 1.12: Example of an extended clique.



(a) Partitioned graph (b) Partition graph

Figure 1.13: Sample partitioned graph along with its partition graph.

Chapter 2

Model

In this chapter we will present various binary integer programming formulations for the PCP, generalized from the CP model presented in [22], which will be used in the branch and cut algorithm.

2.1 Formulation

Let $G =< V, E, P >$ a partitioned graph, being V the set of nodes numbered from 1 to n, E the set of m edges, and P the set of partitions numbered from 1 to q; and let C be the set of color labels numbered from 1 to n.

The standard coloring problem formulation, SCP, uses the following $(n+1)c$ binary variables, where $i \in V$ and $j \in C$:

- x_{ij} equals 1 if and only if the node i is colored with label j
- w_j equals 1 if there is at least one node in the graph which uses color j

The goal is to minimize the total number of colors used, this is, the number of w_j variables set to 1.

$$\begin{aligned}
\text{MINIMIZE} \quad & \sum_{j \in C} w_j & \\
\text{SUBJECT TO} \quad & \sum_{j \in C} x_{ij} = 1 \quad \forall i \in V & (2.1)\\
& x_{ij} + x_{kj} \leq w_j \quad \forall (i,k) \in E,\ \forall j \in C & (2.2)\\
& x_{ij}, w_j \in \{0,1\} \quad \forall i \in V,\ \forall j \in C &
\end{aligned}$$

Equation 2.2 implies that two adjacent vertices may not use the same color, and also ensures that any variable x_{ij} set to 1 will cause w_j to be set as well.

Restriction 2.1 requires that every node is assigned exactly one color. Since the difference between standard coloring and partition coloring relies solely in the fact that, in the latter, only one node per partition must be colored, adjusting this last restriction provides a simple model for PCP.

$$\begin{aligned}
\text{MINIMIZE} \quad & \sum_{j \in C} w_j && (2.3)\\
\text{SUBJECT TO} \quad & \sum_{i \in P_k} \sum_{j \in C} x_{ij} = 1 \quad \forall P_k \in P && (2.4)\\
& x_{ij} + x_{kj} \leq w_j \quad \forall (i,k) \in E,\ \forall j \in C && (2.5)\\
& x_{ij}, w_j \in \{0,1\} \quad \forall i \in V,\ \forall j \in C
\end{aligned}$$

This model has $(n+1)c$ variables as well, q restrictions 2.4 and $m.c$ 2.5 restrictions, plus all integrality constraints.

2.2 Variants

There are several variants for the previously presented model for PCP, all of which provide valid partition colorings. We will explore different alternatives to the basic formulation composed by restrictions 2.3, 2.4 and 2.5, presented in section 2.1.

2.2.1 Color a single node per partition

Restriction 2.4 can be relaxed by requiring that at least one node is colored per partition, instead of requiring that exactly one node is colored. Even more, we may also accept colorings in which a single node is assigned more than one color.

$$\sum_{i \in P_k} \sum_{j \in C} x_{ij} \geq 1 \qquad (2.6)$$

The minimization of the total number of colors used will ensure that no additional colors will be used, and a valid coloring can be extracted from the resulting solution by picking any color from any node on every partition, as no color conflicts will occur since restriction 2.5 is still in place.

2.2.2 Color conflicts

An alternative to restriction 2.5, for both preventing color conflicts and set w_j variables upon usage of color j, is to decouple this two concepts into different restrictions. Therefore, instead of restricting $x_{ij}+x_{kj} \leq w_j$ for every edge, we may require:

$$x_{ij} + x_{kj} \leq 1 \quad \forall (i,k) \in E,\ \forall j \in C \tag{2.7}$$
$$x_{ij} \leq w_j \quad \forall i \in V,\ \forall j \in C \tag{2.8}$$

This alternative, however, yields an even larger number of restrictions than the original 2.5. Looking forward to reducing the number of equations in the model, we propose the following alternative:

$$\sum_{i \in P_k \cap N(i_0)} x_{ij} + x_{i_0 j} \leq w_j \quad \forall j \in C,\ \forall P_k \in P,\ \forall i_0 \in V \tag{2.9}$$

This equation establishes that either node i_0 may use color j, or at most one neighbor in every adjacent partition may use it (as no more than a single node may be colored per partition). This formulation considerably reduces the amount of restrictions when partition sizes are large; otherwise, it does not report any benefits over the original version 2.5.

Another variant that further reduces the number of restrictions makes heavy use of the maximum number of nodes that may use the same color within a neighbourhood:

$$\sum_{i \in N(i_0)} x_{i_0 j} + r x_{i_0 j} \leq r w_j \quad \forall j \in C,\ \forall i_0 \in V \tag{2.10}$$

A simple value for r could be the number of different partitions in the neighbourhood of node i_0. In that case, the restriction implies that either node i_0 uses color j, or at most r nodes in its neighbourhood may use it simultaneously.

However, we may tighten the restriction by replacing r by the number of components in an extended clique coverage of the node's neighbourhood, as this value provides an upper bound for the maximum number of colors that can be used for a set of nodes. We use a simple greedy heuristic to generate a standard clique cover of the partition graph induced by the neighbourhood of i_0 to obtain the value r.

This restriction generates a much lower number of equations in dense graphs, as it requires just c restrictions per node instead of per edge.

2.3 Breaking symmetry

One of the main issues with the model presented is that it allows for multiple symmetric solutions; the same happens with the standard coloring model SCP.

Since it does not matter which color labels are used in the coloring, any χ colors can be used, resulting in $P(c,k)$[1] different solutions for every equivalent coloring; therefore, introducing additional constraints with the purpose of removing symmetric solutions is expected to produce an improvement in the algorithm, as it greatly limits the solution space. Once again, we adapted some of the constraints presented in [22] for SCP to our model.

The easiest restriction to generate is to prevent color $j+1$ from being used unless color j is used in the coloring. This ensures that only colors with labels $1 \dots k$ are used in a k-coloring, leaving always the last $k+1 \dots c$ unused, thus reducing the number of symmetric solutions from $P(c,k)$ to $k!$.

$$w_j \geq w_{j+1} \quad \forall 1 \leq j < c \tag{2.11}$$

A stricter requirement that can be imposed is to forbid having more vertices colored with label $j+1$ than with label j. This removes all symmetric solutions in the case that every color class has a different node count, which is a vast improvement from the previous restriction.

$$\sum_{i \in V} x_{ij} \geq \sum_{i \in V} x_{ij+1} \quad \forall 1 \leq j < c \tag{2.12}$$

However, this restriction allows symmetric solutions by exchanging labels between color classes with same cardinal, which is likely to occur in regular graphs.

In order to further remove symmetric solutions, it is possible to enforce the following restriction, which implies that among all possible assignments to the set of partitions defined by color classes, only the one that assigns the lowest possible color label to the partition with the lowest index is used:

$$x_{ij} = 0 \quad \forall j > p(i) + 1 \tag{2.13}$$

$$x_{ij} \leq \sum_{l=j-1}^{k-1} \sum_{u \in P_l} x_{uj-1} \quad \forall 1 < k \leq q,\ \forall i \in P_k,\ \forall 1 < j \leq k \tag{2.14}$$

Equation 2.13 establishes that color with label j may not be used for a partition with index greater than j; whereas equation 2.14 imposes that color

[1]Number of different ordered subsets of size k from a set of size c, equals to $c!/(c-k)!$

j cannot be used for a partition unless color $j-1$ was used in a previous partition.

As an example, suppose a graph such that $P = \{P_1, \dots, P_q\}$ and $P_k = \{x_{2k-1}, x_{2k}\}$, this is, every partition has two nodes. The first instantiations of restriction 2.14 would be:

$$\begin{array}{ll}
k=2, i=3, j=2 & x_{3,2} \leq \sum\limits_{u \in P_1} x_{u,1} = x_{1,1} + x_{2,1} \\
k=2, i=4, j=2 & x_{4,2} \leq \sum\limits_{u \in P_1} x_{u,1} = x_{1,1} + x_{2,1} \\
\\
k=3, i=5, j=2 & x_{5,2} \leq \sum\limits_{u \in P_1} x_{u,1} + \sum\limits_{u \in P_2} x_{u,1} = x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} \\
k=3, i=6, j=2 & x_{6,2} \leq \sum\limits_{u \in P_1} x_{u,1} + \sum\limits_{u \in P_2} x_{u,1} = x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} \\
\\
k=3, i=5, j=3 & x_{5,3} \leq \sum\limits_{u \in P_2} x_{u,2} = x_{3,2} + x_{4,2} \\
k=3, i=6, j=3 & x_{6,3} \leq \sum\limits_{u \in P_2} x_{u,2} = x_{3,2} + x_{4,2} \\
\qquad\qquad\quad\vdots & \vdots
\end{array}$$

2.4 Objective function

Another way of reducing the number of symmetric solutions is to prefer lower-label colors in the objective function, therefore choosing the lowest labels possible in each coloring. To achieve this, we simply multiply w_j variables with a lower index j by a lower coefficient in the objective function; since we are minimizing, the variables multiplied by the lowest factors should be chosen first.

$$\text{MINIMIZE} \quad \sum_{j \in C} j w_j \tag{2.15}$$

However, early tests showed that this objective function offers poor computational results, so it was early discarded from the set of possible formulations.

2.5 Strengthening the model

There are other inequalities, besides the already mentioned symmetry breaking ones, that are not necessary for the formulation of a valid coloring, yet they strengthen the model relaxation, helping during the branch and cut process. These inequalities are entirely optional in the formulation, their inclusion depends strictly in the tradeoff between building a more complex model that takes more time to solve and strengthening its relaxation so the algorithm's overall performance is increased.

A simple restriction, which is already implied by the objective function, consists in preventing a w_j variable from being set unless there is a node painted with color j:

$$w_j \leq \sum_{i \in V} x_{ij} \quad \forall j \in C \tag{2.16}$$

The usage of this restriction will become clear when we present the branching strategies in 5.7, in which we directly enforce bounds on the w_j variables based on additional information on the coloring problem.

Another equation, which improved the obtained results in [21], avoids the generation of fractional solutions such as $x_{ij} = 1/c$:

$$\sum_{j \in C} w_j \geq \sum_{j \in C} j x_{ij} \quad \forall i \in V \tag{2.17}$$

The rationale behind that restriction is that only one x_{ij} may be set per node, therefore the right hand side of the inequality is the label of the color assigned to node i, which cannot be greater than the total of colors used.

This restriction can be further strengthened by extending the sum of the x_{ij} variables over partitions instead of nodes:

$$\sum_{j \in C} w_j \geq \sum_{j \in C} \sum_{i \in P_k} j x_{ij} \quad \forall P_k \in P \tag{2.18}$$

Chapter 3

Valid inequalities

In the previous chapter, we defined the basic PCP polyhedron as the set of points that satisfy the inequalities presented in 2.1 (2.3, 2.4 and 2.5), along with a number of variants of that model, which span alternative polyhedra.

In this chapter we will derive valid inequalities for the basic PCP polyhedron, which will be used as cutting planes in the branch-and-cut algorithm. It is important to note that the inequalities derived are valid for both the basic polyhedron and for all its presented alternatives; therefore, they can be used as cuts in the branch and cut algorithm regardless of which model is implemented.

3.1 Extended clique inequalities

A classical inequality for the standard coloring problem is the clique inequality, which establishes that within a clique K, at most one node can be colored with a label j.

$$\sum_{i \in K} x_{ij} \leq w_j \quad \forall j \in C$$

Combining this inequality with the fact that in PCP at most one node per partition can be colored with a label j, we define the *extended clique inequality* for PCP. Recall from 1.5 that an extended clique is a maximal subset K_P of V such that every pair of nodes is either adjacent or belong to the same partition. These inequalities specify that among all nodes in K_P at most one of them may use color j.

$$\sum_{i \in K_P} x_{ij} \leq w_j \quad \forall j \in C \tag{3.1}$$

Similar inequalities were developed by [9], based on the asymmetric representatives formulation, but applied only on component cliques[1]. Extended cliques have the added benefit of covering a larger set of nodes, being stronger than their component-based counterparts, and maintain their effectiveness regardless of the partition size used.

3.2 Component independent set inequalities

As was defined in 1.5, a component independent set I_P is a standard independent set with the added restriction that every node must belong to a different partition, and $\alpha_P(G)$ is the size of the largest component independent set of a graph. These definitions allow us to adapt the independent set inequality directly from the standard coloring problem in [22]:

$$\sum_{i \in W} x_{ij} \leq \alpha_P(W) w_j \quad \forall j \in C \tag{3.2}$$

The restriction is applied to a subgraph of G induced by the nodes $W \subseteq V$. Since the cardinal of the maximum component independent set of the subgraph, $\alpha_P(W)$, is not easy to calculate, as it is as difficult as the coloring problem itself, this inequality is applied to particular subsets of the graph with an α_P easy to determine: component holes and component paths.

Note that even though all component cliques have $\alpha_P = 1$ and can be used to generate component independent set inequalities[2], these are superseded by the already described *extended clique inequalities*, as the latter apply over a larger set of nodes; this occurs because every component clique is an extended clique, but not the other way around. Therefore, component clique inequalities will not be considered in this work..

3.2.1 Component hole inequalities

A simple instantiation of the previous inequality can be done by picking a subset W that induces a component hole H[3] in the partitioned graph. As in a standard hole, it holds that $\alpha_P(H) = \left\lfloor \frac{n}{2} \right\rfloor$, where n is the length of the hole, therefore the only effort required lies in finding a component hole in the graph.

[1]Clique in which every node belongs to a different partition.

[2]In this case, *component clique inequalities*, which force every vertex in a component clique to have a different color.

[3]A component hole is a chordless cycle in which every node belongs to a different partition.

Therefore, given a component hole H in the partitioned graph, the component hole inequality is:

$$\sum_{i \in H} x_{ij} \leq \left\lfloor \frac{n}{2} \right\rfloor w_j \quad \forall j \in C \tag{3.3}$$

3.2.2 Component path inequalities

Similar to the previous case, the component independent set can be instantiated with a component path P, which is a standard path where every node belongs to a different partition. In this case, it holds that for every component path of length n, $\alpha(P) = \left\lceil \frac{n}{2} \right\rceil$, and the inequality results:

$$\sum_{i \in P} x_{ij} \leq \left\lceil \frac{n}{2} \right\rceil w_j \quad \forall j \in C \tag{3.4}$$

3.2.3 Strengthening by breaking symmetry

Component independent set inequalities can be strengthened by taking into consideration symmetry breaking constraints, which forbid using a color unless all of the previous colors are used (2.11 being the weakest). This means that the following inequalities will only be valid for models which include symmetry breaking constraints.

In case a component independent of size α_P set is colored with label $j^* \leq q - \alpha_P$, then it is possible to ensure that the colors with the highest $\alpha_P + 1$ labels, $j_{q-\alpha_P+2} \ldots$, will be left unused, since there are α_P nodes using the same color j^*.

Being W the component independent set, in the worst case, in which all nodes in $V \setminus W$ use different colors, an assignment as the one shown in table 3.2.3.1 will occur. This coloring uses only the first $q - \alpha_P + 1$ colors, and leaves all colors with a greater label unused.

This assignment may happen only if every node in $V \setminus W$ uses a different color; if there is any node repeating color then label $q - \alpha_P + 1$ is unused as well. Therefore, at most one node may be colored using $q - \alpha_P + 1$, while colors with a greater label will never be used, so the following inequality holds:

$$\sum_{j=j_t+1}^{c} \sum_{i \in V} x_{ij} \leq w_{j_t+1} \quad j_t = q - \alpha_P(W) \tag{3.5}$$

Combining this inequality with 3.2, results in the following component independent set inequality, which becomes strengthened via symmetry breaking:

color label	partitions count
j_1	1
j_2	1
$\vdots$	$\vdots$
j^*	α_P
$\vdots$	$\vdots$
$j_{q-\alpha_P}$	1
$j_{q-\alpha_P+1}$	1
$j_{q-\alpha_P+2}$	0
$\vdots$	$\vdots$
j_q	0

Table 3.2.3.1: Worst-case color assignment when a component independent set of size α_P is found in the partitioned graph.

$$\sum_{i\in W} x_{ij_0} + \sum_{j=j_t+1}^{c} \sum_{i\in V} x_{ij} \leq \alpha_P(W) w_{j_0} + w_{j_t+1} \quad \forall j_0 \leq j_t,\ j_t = q - \alpha_P(W) \quad (3.6)$$

Both component hole (3.3) and component path inequalities (3.4) can be strengthened using this argument.

3.3 Partition graph inequalities

Let $G' =< V', E' >$ be the partition graph[4] of G. Most bounds found for coloring G' can be reused in the original G by extending the constraint over every node represented by each $p \in V'$.

A clear example are independent set inequalities. Let $W' \subseteq V'$ a subset of nodes inducing a subgraph in G', then the independent set inequality holds:

$$\sum_{i\in W'} x_{ij} \leq \alpha(W') w_j \quad \forall j \in C \quad (3.7)$$

As in inequalities 3.6, these inequalities can be strengthened considering symmetry breaking constraints:

[4]The *partition graph* G' of a partitioned graph G is a standard graph $G' =< V', E' >$ in which every node $v'_k \in V'$ corresponds to a partition $P_k \in P$, and two nodes $v'_i, v'_j \in V'$ are adjacent if and only if every node in P_i in G is adjacent to every node in P_j.

$$\sum_{i \in W'} x_{ij_0} + \sum_{j=j_t+1}^{c} \sum_{i \in V'} x_{ij} \leq \alpha(W') w_{j_0} + w_{j_t+1} \quad \forall j_0 \leq j_t,\ j_t = q - \alpha(W') \tag{3.8}$$

These constraints over G' can be converted to constraints G by replacing every node $p \in V'$ with the sum over the nodes $v \in P_p$. Let $W \subseteq P$ be the set of partitions represented by the nodes in $W' \in V'$ in G', then:

$$\sum_{P_k \in W} \sum_{i \in P_k} x_{ij_0} + \sum_{j=j_t+1}^{c} \sum_{i \in V} x_{ij} \leq \alpha(W') w_{j_0} + w_{j_t+1} \quad \forall j_0 \leq j_t,\ j_t = q - \alpha(W') \tag{3.9}$$

Once again, since the size α of the maximum independent set of a graph is NP-hard to calculate, the subgraph induced by W' is chosen in such a way that this number is trivial to obtain. This inequality is then specialized with W' inducing either a path or a hole in G', having $\alpha(W)$ equal to $\lceil |W|/2 \rceil$ and $\lfloor |W|/2 \rfloor$, yielding *partition graph path inequalities* and *partition graph hole inequalities*, respectively. These kind of inequalities are easy to work with, as they have been extensively studied for the standard coloring problem, and they have not been applied to this problem in any previous work (such as [9]).

Partition graph clique inequalities predicate over a set of nodes which are all adjacent to each other or belong to the same partition, which is the very definition of an *extended clique*. Since partition graph clique inequalities are more restrictive than the extended clique ones, as they require that every node in every partition involved is part of the clique, the former will not be considered and we will be applying only the latter.

Note that partition graph independent set inequalities are less frequent than component independent set ones, since G' tends to be less dense as partition sizes increase; however, the former are much stronger as they impose restrictions over all the nodes in the partitions covered, instead of over a single node per partition.

3.4 Block color inequalities

Block color inequalities arise from the symmetry breaking constraints 2.11 $w_j \geq w_{j+1}$. Given a partition P_k and a color j_0, every coloring of partition P_k using label $j > j_0$ requires color j_0 to be already used in the graph, since 2.11 implies that a color cannot be used unless all previous ones had ben used.

As only a single x_{ij} is set in every partition, or equivalently, exactly one node is painted in each partition, the following inequality holds:

$$\sum_{j \geq j_0} \sum_{i \in P_k} x_{ij} \leq w_{j_0} \quad \forall P_k \in P, j_0 \in C \tag{3.10}$$

These $c.q$ inequalities are extremely easy to generate and, as will be analyzed further in this work, have proven to greatly improve the cutting planes scheme.

Chapter 4

Enumeration algorithms

Implicit enumeration algorithms walk through all possible colorings for the graph, restricting the solution set as much as possible and pruning non-optimal solutions using known bounds. In this chapter we will review implicit enumeration techniques for the classical coloring problem and discuss different generalizations for PCP.

4.1 Classical scheme

A classical scheme for enumeration algorithms is presented in 4.1.

The algorithm picks a node to be colored in each recursive call, attempting to color it with one of the already used labels if possible; it also assigns a fresh color to the node, in order to explore all possible colorings for the graph. Note that several symmetric colorings are left out of the exploration.

At every iteration, the partial solution is pruned if the coloring is using as many labels as the best coloring found by the algorithm, as this implies that the best solution cannot be improved using the current one. The algorithm runs until all possible colorings have been explored, therefore it effectively returns a minimum coloring of the graph.

The strategy used for picking the node to be colored in each recursive call gives place to different algorithms. A very simple strategy is to use the degree of the node, coloring nodes with highest degree first, based on the assumption that difficult nodes should be handled early.

Another algorithm, which is one of the most widely used for the coloring problem, is DSATUR[4]. This algorithm always picks the node with the highest degree of saturation[1], using different strategies for tie-breaking, such as picking the node with the largest number of uncolored neighbours[26]. It has proved to be one of the best enumeration algorithms available.

[1]Number of different colors used in $N(v)$ for a node $v \in V$.

Algorithm 4.1 Classical coloring implicit enumeration scheme for simple graphs $G =< V, E >$

```
call color(0,1)
procedure color(painted, label)
  if current coloring is no better than best coloring then
    prune current solutions subtree
  else if painted equals to $|V|$ then
    update best coloring with current coloring
  else
    call paintnext(painted,label)
procedure paintnext(painted, label)
  pick next uncolored node to color
  for j = 1 to label do
    if can paint node with color $j$ then
      assign color $j$ to node
      call color(painted+1, label)
      uncolor node
  {try coloring node with a new label}
  assign color label + 1 to node
  call color(painted+1, label+1)
  uncolor node
```

For an implementation of the DSATUR algorithm, we used the code provided by Trick in [28], which we ported to Java and adapted for partitioned coloring as described in 4.2.

4.2 Enumerating partitioned colorings

The previous scheme must be modified in order to generate valid partitioned colorings. A simple modification would be to simply pick a new partition instead of a new node on each recursive call, and iterate over all nodes in the partition using all candidate labels. This modification is presented in algorithm 4.2.

Algorithm 4.2 Modification of enumeration scheme for partitioned graphs $G =< V, E, P >$, picking partitions on every call

```
[...]
procedure paintnext(painted, label)
  pick next uncolored partition to color
  for all node in partition do
    for j = 1 to label do
      if can paint node with color j then
        assign color j to node
        call color(painted+1, label)
        uncolor node
    {try coloring node with a new label}
    assign color label + 1 to node
    call color(painted+1, label+1)
    uncolor node
```

However, this modification imposes that all the nodes within the same partition are explored together in the enumeration, regardless of the criteria being used to choose each candidate. For example, if a largest-degree criteria is used, and the remaining partitions (with their nodes' degrees) are $P_1\{v_1(10), v_2(1)\}$ and $P_2\{v_3(5)\}$, algorithm 4.2 would enumerate nodes v_1, v_2, v_3 instead of v_1, v_3, v_2. This severely hurts the efectiveness of the strategy being used.

Therefore, we propose another modification, presented in algorithm 4.3. In this case we use the original enumeration scheme, picking a node from an unpainted partition on every call, but before returning from the procedure we create another branch in which we do not color the chosen node, so that the partition can be later colored using another node.

Algorithm 4.3 Partitioned coloring implicit enumeration scheme for partitioned graphs $G =< V, E, P >$

```
call color(0,1)
procedure color(painted, label)
  if current coloring is greater than or equal best coloring then
    prune current solutions subtree
  else if painted equals to $|P|$ then
    update best coloring with current coloring
  else
    call paintnext(painted, label)
procedure paintnext(painted, label)
  pick next uncolored node to color from all uncolored partitions
  for j = 1 to label do
    if can paint node with color $j$ then
      assign color $j$ to node
      call color(painted+1, label)
      uncolor node
  {try coloring node with a new label}
  assign color $label + 1$ to node
  call color(painted+1, label+1)
  uncolor node
  {leave node unpainted}
  if there are other nodes left in the partition then
    mark node as unavailable
    call color(painted, label)
    mark node as available again
```

It is within this scheme that we embedded our two different strategies based on degree of saturation for partition coloring.

4.3 Partitioned DSatur

Classical DSATUR picks the node with the highest color degree on each iteration, based on the assumption that nodes difficult to color should be handled first, which usually works well for most heuristics. In the case of partition coloring, as suggested in [17], nodes with lower degree are easier to color and should be preferred within a partition; also, it is better to color larger partitions first in order to reduce the problem size as early as possible.

Based on these assumptions, we generalized two different versions for partitioned DSATUR: *easiest node* and *hardest partition.*

4.3.1 Easiest node

The easiest node variant is based on the *onestepCD* heuristic proposed in [17]. In order to pick the node to color, the easiest node is chosen from every uncolored partition, where *easiest* is defined in terms of lowest color degree, with tie breaking on lowest number of uncolored neighbours. From the resulting set, the node with the highest color degree is chosen, as in classic DSATUR.

In other words, this algorithm picks the hardest node from the set of the easiest nodes on each uncolored partition. Note that if every partition contains a single node, this algorithm behaves exactly as classic DSATUR.

Also, in an attempt to explore different solutions earlier to obtain better upper bounds, we also implemented a randomized version of the algorithm. From the set of the easiest nodes in every partition, only half of the time the hardest node is chosen, the other half another candidate is chosen with probability decreasing as its color degree decreases.

4.3.2 Hardest partition

Instead of choosing the easiest node from each partition and then picking the hardest one to color, this algorithm first chooses the hardest partition to handle, and then picks the easiest node from that partition. While the easiest node is picked using the usual color degree criteria, choosing the hardest partition requires a new strategy.

Therefore, in order to determine the hardest partition to color, we experimented with combinations of the following metrics:

- Color degree of the partition, defined as the number of different colors adjacent to all of the nodes in the partition; considering that a larger color degree implies a harder partition to color
- Size of the partition, as a larger partition being colored earlier helps reducing the problem size, therefore, the larger the partition the earlier it should be handled
- Number of uncolored partitions adjacent to the partition, equivalent to the tie breaking criteria used for classic DSATUR

Note that using the color degree of the easiest node in the partition as a criteria, under the premise that a partition can be considered to be as hard as its easiest node, yields the *easiest node* algorithm.

Different combinations of these criteria, as well as the different variants of the algorithm, will be compared in chapter 6.2.

4.3.3 Ad-hoc modifications

As it will be presented in chapter 5, bounded versions of this algorithm will be used during the branch and cut algorithm as an initial heuristic, primal heuristic and subtree pruning. As such, certain adaptations were made to the algorithm.

Since the preprocessing step identifies a large clique before any coloring is performed, the algorithm supports forcing a set of partitions to be assigned a specific set of colors; therefore all partitions $P_{K_1}, \ldots, P_{K_\omega}$ contained in the initial clique are colored with labels $1, \ldots, \omega$. Since we still have to determine which node to color in each partition, we have to try every possible way of picking a single node from each partition to be colored. Therefore, the algorithm colors $P_{K_1}, \ldots, P_{K_\omega}$ first, trying all possible $\prod_{i=1}^{\omega} |P_{K_i}|$ node combinations, before proceeding with the partitioned DSATUR on the rest of the graph.

Also, since the algorithm is used as a primal heuristic within the branch and cut tree (see 5.8), it supports forcing the coloring of certain nodes, keeping those assignments fixed during the enumeration process. In case certain node-color combinations are forbidden due to the restrictions imposed during branching, solutions assigning those combinations are not explored in the algorithm.

In case symmetry breaking constraints 2.12 or 2.14 are used, the color classes obtained by the algorithm are sorted based on node count or minimum partition index respectively, so the obtained solution can be successfully injected into the branch and cut scheme.

Chapter 5

Branch and Cut

In this chapter we will present the implemented branch-and-cut algorithm, including all the components that compose the branch and cut structure, such as initial heuristic, branching strategies, separation algorithms and primal heuristics.

5.1 Algorithm structure

The structure of a branch and cut algorithm can be considered a combination of cutting planes and branch and bound schemes, which we describe below.

5.1.1 Cutting planes

Cutting planes algorithms rely solely on valid inequalities for solving the problem. Given a solution of the model's linear relaxation [1], cutting planes are added to the formulation in order to remove the fractional solution obtained.

Recall that valid inequalities have the property of being satisfied by all integer solutions of the model, but not necessarily by all fractional solutions in the relaxation; therefore, given a fractional solution x^*, there is a cutting plane generated by a linear inequality that holds for all integral solution but is not satisfied by x^*.

The algorithm consists in repeating this process, re-solving the relaxation with all added cuts in each iteration, until an integer-feasible solution is obtained, no more violated inequalities can be found or any defined stopping criteria is achieved.

[1]The linear relaxation of an integer linear programming consists in removing all integrality constraints on the variables.

Algorithm 5.1 General scheme for a cutting planes algorithm

```
loop
  calculate solution of the linear relaxation
  if solution is integer then
    return obtained solution
  else if can generate inequalities to cut off the fractional solution then
    add the cutting planes to the model
  else
    return failure
```

The algorithm depends on having an adequate set of cutting planes families, along with good separation heuristics, in order to find valid inequalities that cut off the relaxation's fractional solution on every iteration. Excessive generation of cutting planes may have the drawback that the relaxation becomes larger and larger on every iteration, requiring a greater computational effort to solve. Even worse, the cutting planes families chosen may be such that the algorithm requires an infinite number of iterations to converge to the integer solution, which forces the addition of an algorithm stopping condition based on elapsed time, number of iterations or relative improvement in the solution, among others.

Note that, besides the valid inequalities specific to each problem, such as the ones we have presented for PCP in chapter 3, there are generic families of cutting planes that can be applied to any problem, like Gomory cuts [10], disjunctive cuts [3], clique cuts, cover cuts, etc. However, problem-specific cuts usually have better performance than generic ones.

5.1.2 Branch and bound

The branch and bound scheme explores different solutions by setting bounds on fractional variables on every partial solution. The algorithm starts by solving the relaxation, like cutting planes, but instead of removing the fractional solution by adding a valid inequality, it creates two subproblems, usually by fixing a particular variable with a fractional value to either zero or one, in the case of binary integer programming problems in which variables are restricted to these two possible values.

Each subproblem is then solved using the same strategy until the relaxation's solution is integer feasible, in which case that branch is closed; note that eventually all variables are fixed to an integer value, so an integer solution is always found. The algorithm runs until all nodes are explored.

Besides the *branching* behaviour of the algorithm, both upper and lower *bounds* for each node are considered. A global upper bound (in case of a minimization problem) is updated whenever a new integer feasible solution is found, keeping the one with the lowest objective value. This upper bound is compared with each node's lower bound provided by the relaxation's solution. Since the integer solution will always be greater than the relaxation's, when the lower bound of the node is larger than the global upper bound, the node can be fathomed.

Algorithm 5.2 General scheme for a branch and bound algorithm

```
initialize tree with the problem's formulation

while there are open nodes in the tree do
    pick an open node from the tree
    calculate solution $x^*$ of the linear relaxation of the node
    if the linear relaxation is infeasible then
        close the node as the branch is infeasible
    else if solution $x^*$ is integer feasible then
        update best integer solution $x_M$ with $x^*$ if $f(x^*) < f(x_M)$
        close the node as an integer solution was found
    else if $f(x^*) > f(x_M)$ then
        close the node as best solution cannot be improved
    else
        generate new subproblem nodes by branching
return best integer solution $x_M$
```

The branch and bound scheme does not specify which node to select on each iteration, or how to generate the subproblems on each node. The node selection strategy and the branching strategy are chosen depending on the problem to solve.

5.1.3 Cut and branch

The cut and branch scheme is simply an execution of the cutting plane algorithm until a certain threshold is reached (expressed in running time, number of cuts iterations or mip gap). Then the generation of cutting planes is stopped and a branch and bound algorithm is executed, using the initial formulation augmented with all the generated cutting planes.

This algorithm usually yields better results than the previous ones, as a cutting plane one may not find inequalities that lead to an optimal solution in a finite number of steps, and a pure branch and bound one usually takes too long to solve as the enumeration may be too large. However, since the cutting planes may cause that the relaxations in the branch and bound become tougher to solve, a good balance between the two phases is extremely important for a good overall performance.

It is worth noting that, since the cut and branch algorithm is ultimately reduced to a branch and bound scheme, this scheme always arrives to an optimal solution, unlike a pure cutting planes algorithm which may fail to obtain an integer solution.

5.1.4 Branch and cut

While the cut and branch algorithm applies cutting planes only at the root of the branch and bound tree, the branch and cut version uses cutting planes throughout the whole tree. Although cuts are applied more aggressively at the root, on certain internal nodes more iterations of cuts are executed.

Note that these generated cuts can use local information, exploiting the variables fixed in the node due to the branching process. In this case, cuts generated on the node are only valid on the node and its subtree; otherwise, cuts generated on an internal node may be reused globally.

An improvement to this scheme consists in deriving integral solutions from node relaxations, obtaining global upper bounds earlier during the traversal, which allow to prune branches earlier. The generation of these solutions is done via *primal heuristics*.

As with all previously mentioned algorithms, branch and cut schemes have a number of parameters and strategies to determine. To begin with, all the separation heuristics for cutting planes iterations must be chosen adequately to quickly find a reasonable number of cuts; on the branch and bound side, node selection and branching strategies must also be determined. Also, the number of cut iterations to perform on the root and on internal nodes must be set, as well as choosing on which nodes cutting planes and primal heuristics should be executed.

Throughout this section we will go over all these missing pieces in the branch and cut structure, and point out how we filled them in the context of the partitioned coloring problem.

5.2 Preprocessing

The first step in solving a PCP instance consists in preprocessing the graph, applying all the following rules until no more modifications are made to the graph:

1. As an initial step, every edge with both ends within the same partition is removed. Since only one node is colored per partition, there can be

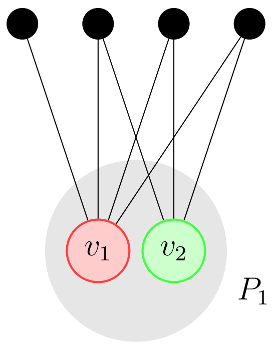


Figure 5.1: Neighbourhood inclusion example: node v_1 will be removed from the graph as its neighbourhood completely contains $N(v_2)$.

no color conflicts between nodes of the same partition, and all edges connecting them can be removed, in order to greatly reduce the size of the graph and the number of adjacency restrictions generated.

2. Partitions containing isolated nodes can be completely removed from the graph, as any isolated node can be trivially colored using the lowest possible label, and coloring a single node within a partition marks the whole partition as colored, therefore allowing us to completely remove it.

3. Neighbourhood inclusion criteria is applied within a single partition in order to remove higher-order nodes. Let u, v be two different nodes in a partition P_k, if $N(u) \subseteq N(v)$, then we can remove node v from the graph. Since only one node per partition needs to be colored, any valid coloring that assigns color j_0 to node v can be modified to assign color j_0 to node u instead, still satisfying all color constraints. Intuitively, we are removing *difficult* nodes from a partition when we find an easier one to substitute it. See figure 5.1 for an example.

4. A lower bound for the chromatic number of the graph is obtained by finding a maximal clique in the partition graph G'. Finding a clique of size ω in G' implies that at least ω different colors are needed for coloring the partition graph, and the same lower bound clearly holds for G. All partitions in the clique will have their colors fixed to $1, \ldots, \omega$ in order to reduce the number of possible colorings, since each of them must be painted using a different color.

5. As in step 2, partitions that contain at least one node with degree less than the lower bound found in the previous step are removed. A node with strictly less than ω neighbours can be assigned a color among

$1, \ldots, \omega$, knowing that no color conflicts will occur; and since there are already ω colors required, the chromatic number is not increased by that assignment, and therefore the node can be discarded.

Last 3 steps are performed until no more changes are made to the graph. The resulting largest clique found is used to fix the colors of the partitions included in it.

Every step is processed by brute force, since their running time is polynomial in the size of the graph, except for step 4 for which we use the algorithm presented below.

5.2.1 Partition graph clique detection

To find the maximum clique in the partition graph we use a simple backtracking algorithm. Since the running time of this algorithm can be excessive for a preprocessing step, we bound the running time of this algorithm to five seconds; however, the algorithm usually ends much sooner, as the partition graph is not only smaller but also much less dense than the original graph.

In case the time bound is reached, the best solution found so far is returned. As the backtracking uses DFS to explore all possible solutions, a reasonably good solution is generated early in the algorithm, therefore a valid result is obtained regardless its interruption.

Starting with an initial node, the algorithm keeps a list of valid candidates for the clique, which is updated on each iteration by removing all nodes that are not adjacent to the clique. Keeping both the candidates list and all adjacency lists sorted by degree makes computing the intersection between these lists faster, and produces better initial solutions that can be used to prune other solutions later.

5.3 Initial heuristic

A good initial heuristic solution gives an upper bound on the solution, eliminates a large number of variables and restrictions in the model, and can be used as an initial incumbent solution for the branch and cut algorithm.

In order to generate this solution, we use the modification of the DSATUR algorithm for partitioned graphs presented in 4. Since the algorithm generates an implicit enumeration of all possible colorings, which might take too long to compute, its running time is bounded to five seconds. The coloring of the partitions in the clique K is fixed to labels $1, \ldots, \omega$ in order to reduce the solutions set.

Algorithm 5.3 Finding a maximum clique in a simple graph $G =< V, E >$

```
sort all nodes and adjacency lists descendingly by degree
for all initial node $v$ in $V$ do
    initialize clique with node $v$
    initialize candidates with $N(v)$
    call clique
procedure clique
    if candidates is empty then
        update best solution if current clique is better
    else if clique.size + candidates.size $\leq$ best.size then
        prune current tree
    else
        pop node $u$ from candidates and add it to clique
        intersect candidates with $N(u)$ and store removed nodes
        call clique
        remove node $u$ from clique
        add removed nodes back to candidates
        call clique
```

5.4 Initialization

Using the initial solution as an upper bound $\hat{\chi}$ for the chromatic number, it is possible to eliminate all x_{ij} and w_j variables with $j > \hat{\chi}$, thus greatly reducing the number of involved variables and restrictions in the model.

Another optimization is to fix the colors for all partitions involved in the clique K found during the preprocessing stage. Since it is not possible to determine which node within the partition is to be colored, we simply set to zero all x_{ij} variables for nodes within the partitions that use a different color than the one assigned. Formally, let $K = \{P_{K_1}, \dots, P_{K_\omega}\}$ be the initial clique, then:

$$\begin{aligned} x_{ij} &= 0 \quad \forall i \in K[l],\ \forall 1 \leq l \leq \omega,\ \forall j \neq l \\ w_j &= 1 \quad \forall 1 \leq j \leq \omega \end{aligned}$$

Also, in case the partition being fixed to a color j_k has a single node in it, then variable x_{ij_0}, where i is the single node in the partition, is fixed to 1.

Another bound based on nodes degree is imposed. A node v of partition degree $\delta_P(v)$ can always be colored with a label j_k such that $1 \leq j_k \leq \delta_P(v)+1$, since it will be neighbour to at most $\delta_P(v)$ different colors, therefore any set of $\delta_P(v) + 1$ colors contains at least one valid label that does not generate color conflicts.

$$x_{ij} = 0 \quad \forall i \in V,\ \forall j > \delta_P(v) + 1$$

Finally, in case minimum partition index breaking symmetry restrictions (2.14) are being used, this is, the color with the lowest label is assigned to the color class containing the partition with the lowest index, another bound can be imposed on x_{ij} variables. Since in the worst case all partitions will have a different color, then a partition with index k will never be colored with label greater than k. Therefore, the following bound is imposed:

$$x_{ij} = 0 \quad \forall P_k \in P,\ \forall i \in P_k,\ \forall j > k$$

5.5 Cuts separation

For each family of valid inequalities listed in 3, an heuristic is implemented to find a set of valid inequalities being violated by a solution of the linear relaxation of the model. Note that finding a violated inequality in a solution is NP-Complete[2], so heuristic procedures must be applied.

Since these algorithms are applied frequently during the branch and cut tree, it is imperative that their running time is as fast as possible, in order to minimize the added overhead to the whole algorithm.

5.5.1 Extended clique cuts

Separation of extended clique cuts (3.1) is done using a very similar algorithm to 5.3, adapted to partitioned graphs and without backtracking, so running time is acceptable for a separation algorithm. This algorithm is executed once for each color, and nodes are sorted based not on their degree but on their x_{ij} value in the current solution.

For each initial node, a clique is constructed until the corresponding inequality is broken, and extended to up to κ maximal cliques using backtracking, making use of the *candidates* collection (note that in this case, *candidates* is initialized with not only the initial node's neighbours, but also with all the nodes in its partition). In case no clique breaking the inequality is found, the next initial node is picked.

In order to avoid exploring the same solution space multiple times for different initial nodes, restrictions on how many times a node or an edge can be visited are applied. The resulting algorithm is presented in 5.4.

[2]Grötschel, Lovász and Schrijver [11] related the complexity of the optimization problem being solved to the complexity of the separation problem: the former is polynomial if and only if the latter is also polynomial.

Algorithm 5.4 Separation algorithm for extended clique cuts

```
for all color $j$ such that $w_j \geq \mu$ do
  sort all nodes and adjacency lists descendingly by $x_{ij}$ value
  for all initial node $v$ in $V$ do
    initialize clique with node $v$
    initialize candidates with $N(v) \cup P(v)$
    while candidates is not empty do
      if current clique breaks inequality then
        for all maximal cliques $K$ containing clique up to $\kappa$ do
          add extended clique cut using $K$ and color $j$
        continue with next initial node
      else if next candidate $u$ can be used then
        add $u$ to clique and remove it from candidates
        remove nodes not adjacent to $u$ from candidates
      else
        remove $u$ from candidates
```

5.5.2 Component independent set inequalities

Component hole 3.3 and component path 3.4 inequalities are separated within the same procedure using a greedy heuristic. In a similar fashion to algorithm 5.4, for every color the graph is sorted according to x_{ij} values, and for each initial node a component path or hole is greedily constructed. Once again, bounds for a maximum number of visits on each node are imposed, thus rejecting nodes with a certain number of visits or belonging to a partition already in the path, since this would violate the *component* property.

On every iteration the most promising node is added to the path being built. In case this node is adjacent to a node already in the path (thus generating a hole), it is added only if it violates the corresponding inequality, otherwise, the next candidate is picked, and so forth. Algorithm 5.5 resumes this process.

As an alternative to the previous algorithm, we also implemented the hole detection algorithm presented in [23]. We adapted the algorithm presented in the paper to reject a node if it belongs to a partition already in the path, therefore exploring only component holes; testing this implementation against the previous one showed that our custom algorithm performed slightly better, and generated path as well as hole inequalities.

5.5.3 Partition graph independent set inequalities

For both path and hole inequalities (3.7) over the partition graph G', algorithms equivalent to the ones used for component independent sets are

Algorithm 5.5 Separation algorithm for component independent set cuts

```
for all color $j$ such that $w_j \geq \mu$ do
  sort all nodes and adjacency lists descendingly by $x_{ij}$ value
  for all initial node $v$ in $V$ do
    initialize *path* with node $v$
    loop
      for all valid node $u$ adjacent to last node in the path do
        if $u$ is adjacent to a previous node $w$ in the path then
          if hole $H = [u, \ldots, w]$ violates inequality 3.3 then
            add component hole inequality with hole $H$ and color $j$
            continue with next initial node
        else
          add node $u$ to *path*
          if current *path* breaks inequality 3.4 then
            add component path inequality with *path* and color $j$
            continue with next initial node
```

applied as separation heuristics.

Graph G' is constructed once at the beginning of the branch and cut and is then used as input for these heuristics. Since there are no x_{ij} variables to use for sorting the nodes of the graph, the value $\sum_{i \in P_k} x_{ij}$ is used for each partition P_k, this is, the sum of the values of the partition's nodes.

5.5.4 Block color inequalities

Block color inequalities (3.10) are explored using brute force, since there are no more than $c.q$ of them, and checking whether they are violated or not can be performed fast enough.

Alternatively, they can be added initially to the cut pool provided by the branch and cut framework, instead of manually checking them at each cuts iteration.

5.6 Node selection strategies

On each iteration of the branch and cut algorithm, an unprocessed node must be chosen to be solved. Determining which node will be handled next is known as the *node selection* strategy.

There are different standard node selection strategies, all of them implemented by the branch-and-cut framework we used. In chapter 6 we experiment with the behaviour of the algorithm under different strategies:

- **DFS:** Depth-first search, picks the last node opened, attempting to generate an integral solution by diving to the bottom of the tree as fast as possible. This strategy has low memory consumption as relatively few nodes are left unprocessed on every iteration.

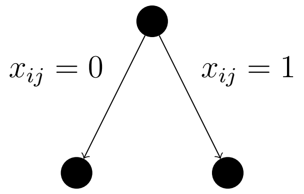
- **BFS:** Breadth-first search, picks the first node opened; this strategy solves every open node at the same depth before proceeding to the following depth level. It usually has memory issues, as a large number of nodes tend to be left open, thus making this a poor choice in most cases.

- **BestBound:** Best-bound, picks the open node with best objective function available, which tends to be near the root of the branch and bound tree. This strategy usually reports the best results.

- **BestEstimate:** Best-estimate uses *an estimate of a given node's progress toward integer feasibility relative to its degradation of the objective function*[29]; it improves the chance of finding feasible solutions when they are difficult to generate.

5.7 Branching strategies

After each node in the branch and cut tree is processed, new child nodes are created by subdividing the problem into two easier subproblems; this is usually done by *branching* on a certain variable. Usually, in the case of binary variables, a variable x with a fractional value in the relaxation is chosen, and the two subproblems are created by fixing $x = 0$ and $x = 1$ and re-processing. Alternatively, bounds on expressions, instead of on variables, can be set.



Whichever method of branching is chosen, it requires that the solutions represented by the union of all subproblems cover all integral solutions represented by the parent node; otherwise, feasible solutions may be left unexplored.

Choosing which variable to branch on, and what bounds are implied within each branch, is part of what is called the branching strategy.

5.7.1 Static priorities

The simplest way to choose which variable to branch on is to assign a priority to each x_{ij}, which will be used to pick the branching variable when necessary: the integer infeasible variable with the highest priority is picked on every branching.

We experimented in section 6.3 with different criteria for selecting variable $x_{i^*j^*}$. We first pick the node v_{i^*} based on either the number of partitions adjacent to it or the size of the partition it belongs to, or a combination of both. Once the node is chosen, we choose from the set of variables $x_{i^*j_0} \dots x_{i^*j_C}$ the one with the lowest j such that its value in the relaxation is greater than zero (as we choose only integer infeasible variables for branching).

Using this criteria has the huge drawback that no information regarding the actual value of the x_{ij} variable is used, so these priorities work best as a tie-breaker for another strategy.

5.7.2 Fractional values

A common practice is to pick the most fractional variable to branch on. We determine such variable as:

$$\min_{x_{ij}} \{|x_{ij} - 0.5|\}$$

In case of a tie, we use the static priority set for the variables to determine which one use to branch on. We also experimented in section 6.3 with the opposite criteria, this is, branching on the less fractional value (excluding those variables with already integral values).

This is a common branching technique, but does not exploit any particular feature of the problem being studied, unlike the one described below.

5.7.3 Degree of saturation

A branching strategy specifically related to the partitioned coloring problem is to branch on a node with the highest degree of saturation. Since these nodes are usually the most difficult ones to handle, it is reasonable to fix their values as early as possible in the branch and cut tree.

This criteria for picking the branching variable requires first to compute an approximate degree of saturation for every node and choosing the one with the largest value, i^*, in order to obtain a set of candidate variables $x_{i^*j_0}, x_{i^*j_1}, \dots, x_{i^*j_c}$ (once again, ties between nodes are broken using the already defined priorities).

Since the only available values are those of the fractional solution, we color one node in each partition using the largest value within the partition and neighbours: this is, for every node i and color j combination, if the value x_{ij} is larger than all of its neighbours and nodes in the same partition, as well as larger than an arbitrary lower bound (0.7)[3], we assign color j to node i. Note that some partitions might be left uncolored, in this case they will not contribute to the degree of saturation of their neighbours.

$$v_i \leftarrow j \text{ if } x_{ij} > 0.7 \wedge x_{ij} > x_{kj} \; \forall k \in N(i) \cup P(i) \tag{5.1}$$

Having chosen a node v_i for branching, we must choose which variable $x_{i^*j^*}$ from the set of candidate $x_{i^*j_0}, x_{i^*j_1}, \ldots, x_{i^*j_c}$ will be branched on. We have implemented two different strategies for this:

- **DSatur-2:** Choose the variable $x_{i^*j^*}$ with the highest value from the set, and branch on $x_{i^*j^*} = 0$ and $x_{i^*j^*} = 1$; this results in a classic $0-1$ branching on the variable corresponding to the most saturated node with its most *likely* color.

- **DSatur-(C+1):** Create up to $C+1$ subproblems, one for each possible coloring of the node, branching on $x_{i^*j_0} = 1, \ldots, x_{i^*j_c} = 1$, plus another child which sets all $x_{i^*j_0}$ variables to zero, in case the node is not colored within its partition; this idea was first defined in [4], revisited in [26], and used in [21].

5.7.4 Implied bounds

When manually specifying the branching variable and creating the subproblems, it is also possible to fix more variables that would be affected by the value assigned to the first one.

Regardless of the branching variable, it is possible to fix all color variables w_j to 1 for $j = 1, \ldots, \lceil z \rceil$, where z is the value of the objective function of the current node's relaxation. For example, if the sum of all w_j variables is 5.3, which is a lower bound on the chromatic number, we can be sure that at least 6 different colors are needed to color the graph, and therefore all $w_1, \ldots, w_6$ can be set to 1.

When branching down on the selected variable[4], there are no more logical implications than the previous one that can be used to bound more variables. This is easy to see since setting an x_{ij} variable to zero implies that a certain

[3]Note that if the classic constraints are being used, 2.4 and 2.5, by specifying a lower bound higher than 0.5 it is not necessary to verify that the node has the highest value among its neighbours or within the partition, as it will be assured by the restrictions; the check is required in case alternative restrictions are used, such as allowing more than one node to be colored or grouping multiple color conflicts into single constraints.

[4]Fixing the branch variable's value to 0.

color will not be used for a certain node, but does not grant any information on which *node* on the partition will be colored and with which *color*.

Branching up, on the other hand, provides much more information. Whenever a variable $x_{i^*j^*}$ is set to 1, this is, node i^* in partition $P(i^*)$ is assigned color j^*, we may specify the following conditions for that branch:

- Every other color-node combination in partition $P(i^*)$ can be set to zero, as only one node must be assigned a color in the partition.

$$x_{ij} = 0 \quad \forall i \in P(i^*),\ \forall j \in C,\ i \neq i^* \vee j \neq j^*$$

- Every node adjacent to i^* cannot use color j^* in order to avoid color conflicts.

$$x_{ij^*} = 0 \quad \forall i \in N(i^*)$$

5.8 Primal heuristic

The algorithm used to create an integer feasible solution from the relaxation's solution is called the *primal heuristic*. A typical primal heuristic consists in rounding the values of every fractional variable to the nearest integer value, as long as this process satisfies all the restrictions imposed by the model.

For PCP we implemented a primal heuristic based on the DSATUR algorithm. Given a fractional solution x^*, for every variable x_{ij} with a large enough value, we fix that node-color combination. The criteria used for determining when a variable is fixed is the same as the one depicted in 5.1.

Also, for every variable x_{ij} with an upper bound set to 0 as a product of the branching in the branch and cut tree, we forbid that node-color combination.

Having all these values fixed, an extremely short run of DSATUR is executed, bounded to 200 milliseconds. The algorithm works reasonably fast as more and more variables are fixed, and bounds for the optimal coloring can be inferred from the branch and cut tree, further shortening the exploration of possible solutions.

- Value $\left\lceil \sum_{j \in C} w_j \right\rceil$ of the node's relaxation is a lower bound to the integer solution, so in case DSATUR finds a solution using that number of colors, it can be assured that it is the local optimum.

- The solution of the primal heuristic will be used as the global upper bound, replacing the current incumbent solution, only if it uses less colors. Therefore, DSATUR is bounded to exploring solutions that use strictly less colors than the incumbent.

The best coloring obtained by the algorithm is then used as an incumbent solution for the node. In case certain symmetry breaking restrictions are in place, a reordering of the labels assigned to each color class might have to be performed.

5.9 Implicit enumeration

Early experimentation with the branch and cut algorithm and with the DSATUR algorithm has shown that, for relatively small instances, the latter explores all possible solutions much faster than the former, since it does not have all the overhead imposed by the different artifacts present in a full branch and cut.

Therefore, when we have reduced the problem size to a relatively small one by fixing node-color assignments during the branching process, instead of proceeding with the traditional branch and cut algorithm, we execute a full run of DSATUR. Since most partitions are already colored, the number of possible solutions is reasonably small to be fully explored.

In chapter 6 we experiment with different values for the number of uncolored partitions in the branch and cut tree to be used as the threshold for stopping the branch and cut and starting an unbounded execution of DSATUR.

5.10 Implementation details

The branch and cut algorithm was implemented in Java 1.6 using CPLEX version 12.1 both as a branch-and-cut framework and a linear programming solver for relaxations.

We made use of branch, heuristic and cut callbacks provided by the CPLEX API to manage the branching strategy, inject primal solutions and apply custom cuts on both the root and internal nodes.

- The branch callback is invoked once the processing of a certain node has been completed in order to determine how to create the child subproblems; inside this callback we implement the different dynamic branching strategies described in 5.7. Static priorities are fixed during the initialization of the problem. This callback is also used to prune the branch and cut tree once a certain number of partitions have been fixed in order to proceed with the implicit enumeration from 5.9.

- The heuristic callback is invoked after the linear relaxation of a node has been solved, and provides the fractional values from the relaxation's solution to derive an integral feasible solution, using the color degree strategy explained in 5.8. We make use of this callback to inject the integral solution derived from implicit enumeration (5.9).

- The cut callback is invoked after the linear relaxation is solved; every certain number of nodes the separation heuristics are invoked in an attempt to add planes to cut off the current linear solution. After the cuts are added, the relaxation is solved again, and more iterations of cutting planes may be optionally executed; while few iterations are performed in the internal nodes, a larger number is executed in the root.

The framework was configured to use standard branch and cut search, instead of dynamic search, to correctly determine the performance of the developed strategies. Multi-core processing was also disabled.

Chapter 6

Computational results

This chapter contains all the results obtained from the computational experiments executed. We devised a test suite for each algorithm or strategy to parametrize, and picked the configuration with the best results for each subsequent test. Tested components include the model formulation, DSATUR strategy, branch criteria, primal heuristic, separation algorithms, etc.

6.0.1 Testing environment

We executed all tests on an Intel Core 2 Duo E7400 (2.80GHz each) with 3.5 GB RAM, running Windows XP SP3, with a JVM version 1.6.0 update 16. In most experiments execution was confined to a single core, in order to eliminate distortions in measures caused by parallelization.

We used IBM Ilog Cplex version 12.1 as a branch and cut framework, using its Java libraries for programming custom routines.

6.0.2 Graphs instances

We used different types of instances on each test depending on the focus of the test itself. Overall, we used mostly random graphs, as well as certain DIMACS challenge instances[1] for our last tests. In the case of random graphs, we used multiple instances (between three and five, depending on running times) for each different value of number of nodes, density and partition size; and reported the average results.

Random graphs were built according to two different schemes:

- **Erdos-Rényi:** Also known as binomial graphs[7], each of the possible edges between the set of n nodes is chosen to be included with probability p; this parameter determines the density of the graph. This procedure generates graphs with an equal distribution of degrees among its nodes.

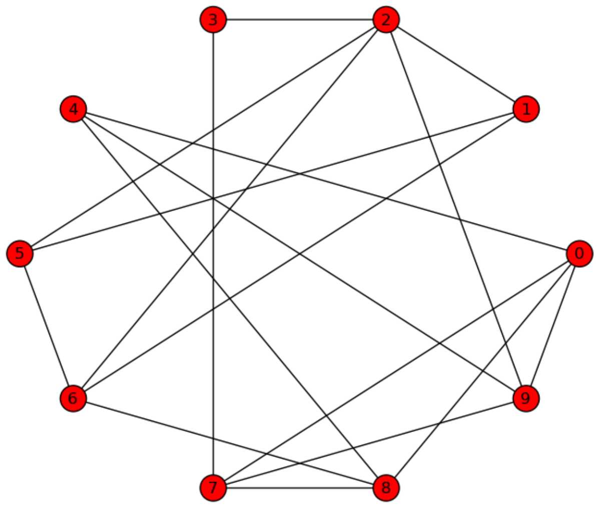


Figure 6.1: Sample random binomial graph with 10 nodes and a 40% probability to create an edge between each pair of nodes.

- **Holme-Kim:** Also known as powerlaw cluster graphs[12], the graph is constructed iteratively by attaching each new node to a number m of already existing nodes using preferential attachment based on the degree of existing nodes. Also, after the addition of each edge, there is a probability p that a triangle will be created by adding an additional edge. These graphs are controlled by two parameters: node count n and probabilty p, with $0 \leq p \leq 1$; value m is inferred as the product between p and n.

Since both of the previous procedures generate unpartitioned graphs, they are partitioned after their generation by grouping nodes with consecutive indices in partitions of a certain size. For most instances we used a fixed partition size of 2, although in some cases we used random sizes varying from 1 to 4.

These random graphs were generated using the *NetworkX* python package[2], version 1.1.

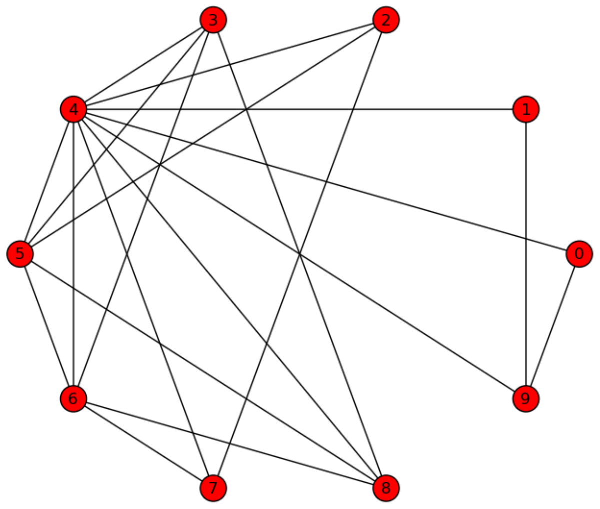


Figure 6.2: Sample random powerlaw cluster graph with 10 nodes, note how certain nodes have a much higher degree than the others.

6.1 Model comparison

We executed a test suite for determining which inequalities to use in the formulation of the problem. Recall from section 2 that there are several restrictions that can be applied to define the model, as well as additional ones that may strengthen the model or reduce the number of symmetrical solutions.

In order to test the effectiveness of the different formulations, we applied a fixed number of cutting planes iterations, using all implemented cuts with a slightly aggressive configuration, and reported the resulting MIP gap and running time (in seconds), as well as how many rounds of cutting planes were executed. It is worth noting that in some cases fewer iterations than total were applied as the separation heuristics were not able to find any more violated inequalities.

For these tests we used binomial graphs with a fixed size of 100 nodes with exactly 2 nodes per partition and densities from 20% to 80%, and powerlaw cluster graphs with the same size with different values of p and m. Five instances for each density for each random family were used, and the reported values are always the average of the ones obtained. All graphs were preprocessed beforehand.

6.1.1 Adjacency constraints

We first tested the four different adjacency (or color conflict) constraints we had proposed, using arbitrarily chosen constraints 2.4, 2.11 and 2.16 to complete the model:

$$\sum_{i \in P_k} \sum_{j \in C} x_{ij} = 1 \qquad \forall P_k \in P \tag{2.4}$$

$$w_j \geq w_{j+1} \qquad \forall 1 \leq j < c \tag{2.11}$$

$$w_j \leq \sum_{i \in V} x_{ij} \qquad \forall j \in C \tag{2.16}$$

The different adjacency constraints being tested in this experiment are the following:

$$x_{ij} + x_{kj} \leq w_j \qquad \forall (i,k) \in E,\ \forall j \in C \tag{2.5}$$

$$x_{ij} + x_{kj} \leq 1 \qquad \forall (i,k) \in E,\ \forall j \in C \tag{2.7}$$

$$\sum_{i \in N(i_0)} x_{i_0 j} + r x_{i_0 j} \leq r w_j \qquad \forall j \in C,\ \forall i_0 \in V \tag{2.10}$$

$$\sum_{i \in P_k \cap N(i_0)} x_{ij} + x_{i_0 j} \leq w_j \qquad \forall j \in C,\ \forall P_k \in P,\ \forall i_0 \in V \tag{2.9}$$

Results are displayed on table 6.1.1.1. Differences between gaps are almost non existent, whereas time required changes greatly between graphs with different densities. On higher density graphs, constraints 2.10 using a clique coverage of the neighbourhood report a better running time than the others; while on lower density 2.7 works better than 2.5, even though the former uses $n.c$ additional constraints 2.8.

Graphics 6.3 show the evolution of the gap for four particular representative instances, one for each of the evaluated densities in binomial graphs. Gap evolution is very similar among different configurations, except for 2.10 which is remarkably slower to achieve the same gap in low density graphs. At 40% and 60% densities, 2.7 not only works better than 2.5 as reported in table 6.1.1.1, but also achieves the same gap slightly faster.

6.1.2 Colored nodes per partition

A quick test we also ran in parallel was to determine whether to paint exactly one node per partition (2.4), or to relax this constraint and allow for painting more than a single node (2.6).

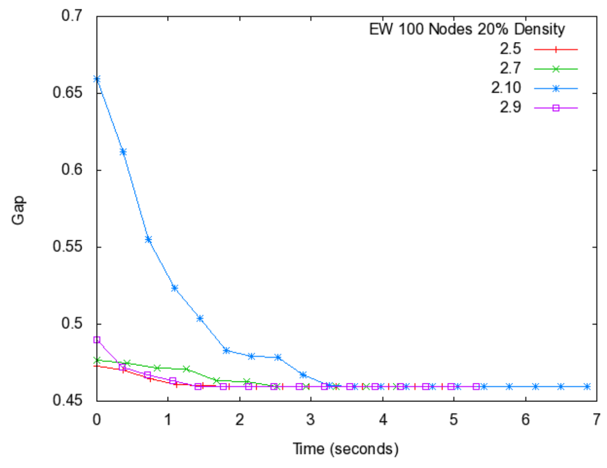
Results on table 6.1.2.1 confirm our expectations: while the former has a slightly larger running time, it also reports a slightly lower gap than the latter in some cases. The simplicity provided by 2.4 when extracting solutions from the model (when constructing the the partial solutions to be processed during the primal heuristic, or during the branching process) makes us choose this option in our formulation.

6.1.3 Model strengthening

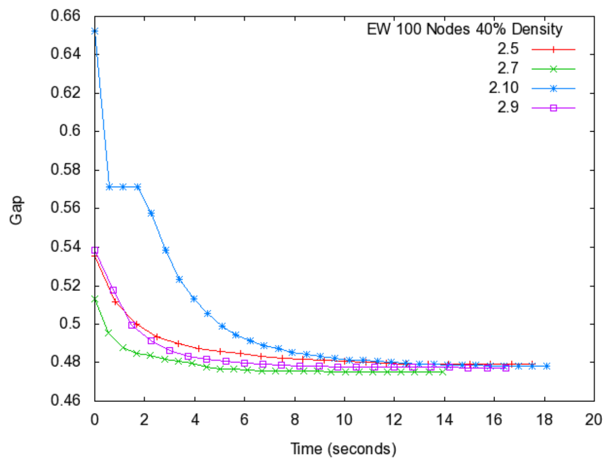
We also compared applying only constraint 2.16, which ensures that variable w_j is set only if a node uses color j (regardless of the objective function), to adding restrictions 2.17 and 2.18:

Id	Constraint 2.5			Constraint 2.7			Constraint 2.10			Constraint 2.9		
	gap	rounds	time	gap	rounds	time	gap	rounds	time	gap	rounds	time
EW 20%	45.8%	14.6	5.632	45.8%	10.6	**4.568**	45.8%	20.4	7.915	45.8%	16.2	5.728
EW 40%	46.6%	22.4	16.998	46.6%	25.6	**14.09**	46.6%	32.6	17.884	46.6%	24.8	16.976
EW 60%	42.0%	62.8	98.642	42.0%	57.4	**77.694**	42.0%	72.6	87.575	42.0%	78.6	120.138
EW 80%	29.2%	193.0	449.054	29.6%	181.2	469.844	29.2%	192.4	**349.557**	29.4%	160.0	451.126
HK P=0.1	20.0%	0.8	0.156	20.0%	0.8	0.128	20.0%	0.8	**0.106**	20.0%	0.8	0.168
HK P=0.2	12.0%	0.6	0.278	12.0%	0.6	0.324	20.0%	1.0	**0.181**	12.0%	0.6	0.306
HK P=0.3	0%	0.8	0.308	0%	0.8	**0.276**	0%	3.2	0.489	0%	0.8	0.318
HK P=0.4	4.8%	2.2	**0.28**	4.8%	3.0	0.312	4.8%	5.2	0.416	4.8%	2.6	0.292

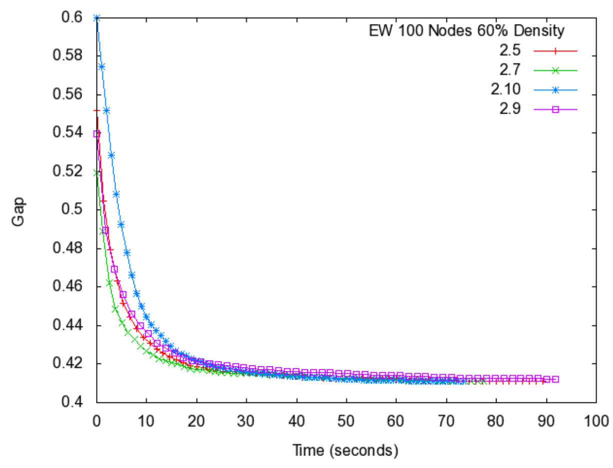
Table 6.1.1.1: Comparison of different color conflict constraints on the model formulation: adjacent nodes sum bounded by w_j (2.5), adjacent nodes sum bounded by 1 (2.7), adjacencies grouped by partition (2.9) and using clique coverage of the neighbourhood (2.10).



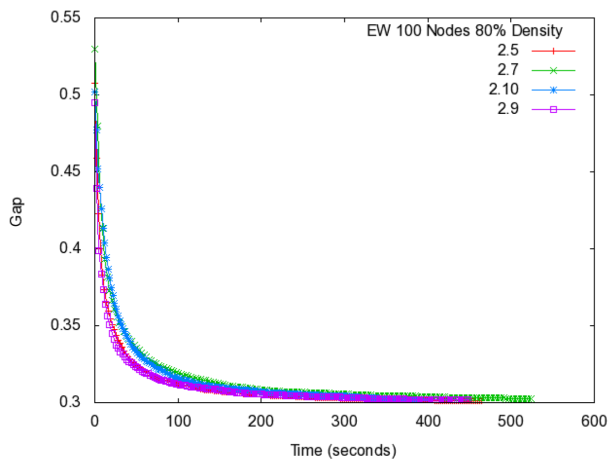
(a) EW 100 Nodes 20% Density



(b) EW 100 Nodes 40% Density



(c) EW 100 Nodes 60% Density



(d) EW 100 Nodes 80% Density

Figure 6.3: Comparison of the inclusion of different color conflict constraints in the model, visualizing evolution of the gap during time in a cutting planes algorithm. Compared constraints are: adjacent nodes sum bounded by w_j (2.5), adjacent nodes sum bounded by 1 (2.7), adjacencies grouped by partition (2.9) and using clique coverage of the neighbourhood (2.10).

Id	At least 1		Exactly 1	
	gap	time	gap	time
EW 20%	46.0%	5.472	45.8%	7.915
EW 40%	46.6%	17.324	46.6%	17.884
EW 60%	42.0%	93.578	42.0%	87.575
EW 80%	29.4%	354.612	29.2%	349.557
HK P=0.1	20.0%	0.112	20.0%	0.106
HK P=0.2	20.0%	0.136	20.0%	0.181
HK P=0.3	0%	0.434	0%	0.489
HK P=0.4	7.6%	0.398	4.8%	0.416

Table 6.1.2.1: Comparison of constraints specifying whether exactly one node must be assigned one color in the partition, or at least one node should be painted with at least one color.

$$\sum_{j \in C} w_j \geq \sum_{j \in C} j x_{ij} \quad \forall i \in V \tag{2.17}$$

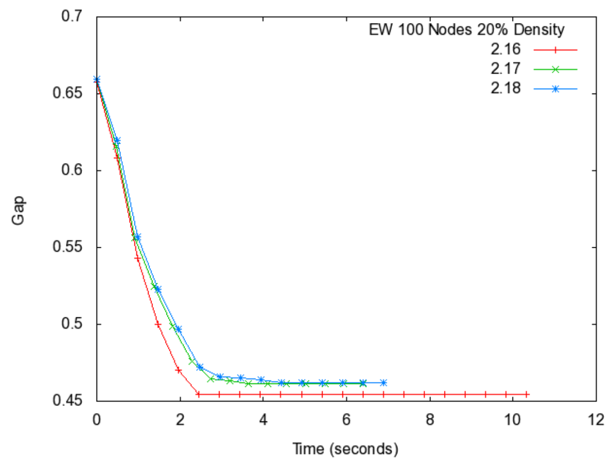
$$\sum_{j \in C} w_j \geq \sum_{j \in C} \sum_{i \in P_k} j x_{ij} \quad \forall P_k \in P \tag{2.18}$$

Results on table 6.1.3.1 show that there is very little difference between the three variants. Overall, the simplest one, 2.16, seems to be the fastest one to execute, although taking slightly more cuts iterations in non-medium density graphs.

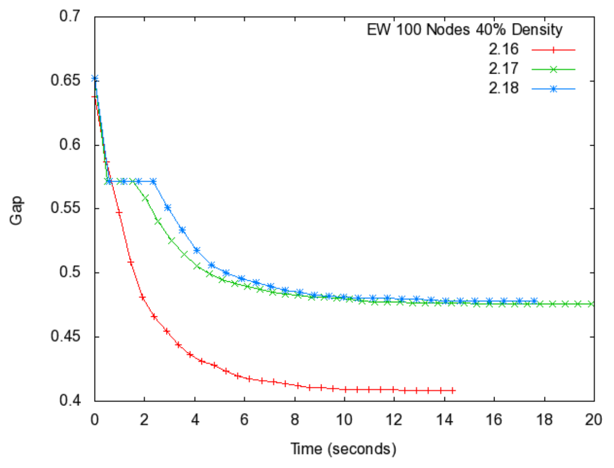
The graphics 6.4 present, as before, the evolution of the gap for these different configurations on a representative binomial instance for each tested density. Configuration 2.16 is shown to be not only the one returning the best gap (even for very little difference with the others), but also the fastest one.

6.1.4 Symmetry breaking

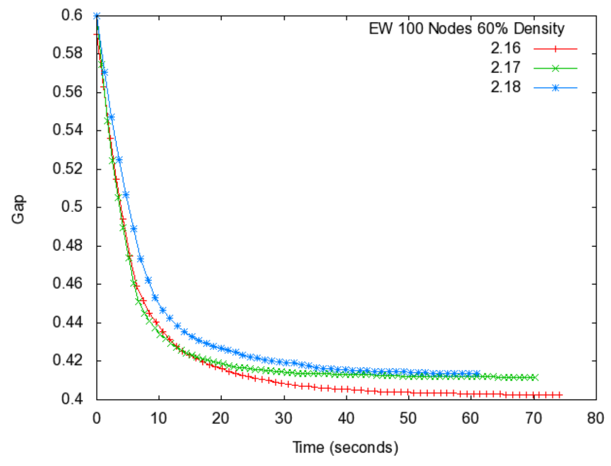
Results obtained from comparing no symmetry breaking constraints whatsoever with color label (2.11), node count (2.12) and minimum node label (2.13,2.14) ordering restrictions are shown on table 6.1.4.1. The evolution of the obtained gap in time for different densities is shown in figure 6.5.



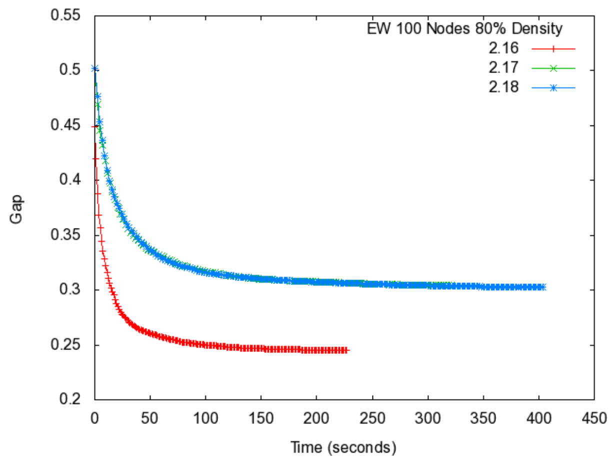
(a) EW 100 Nodes 20% Density



(b) EW 100 Nodes 40% Density



(c) EW 100 Nodes 60% Density



(d) EW 100 Nodes 80% Density

Figure 6.4: Comparison of the inclusion of model strengthening constraints in the model, visualizing evolution of the gap during time in a cutting planes algorithm. Compared constraints are: (2.16) which ensures that variable w_j is set only if a node uses color j, and (2.17) and (2.18) which eliminate certain fractional constraints, adding over all colors of node and of a partition, respectively.

Id	2.16			2.17			2.18		
	gap	niters	time	gap	niters	time	gap	niters	time
EW 20%	45.8%	20.4	7.915	45.8%	15.8	7.318	45.8%	15.4	**7.286**
EW 40%	46.6%	32.6	17.884	46.6%	37.6	19.464	46.6%	32.0	**17.748**
EW 60%	42.0%	72.6	**87.575**	42.0%	84.2	91.272	42.0%	79.4	89.03
EW 80%	29.2%	192.4	**349.557**	29.4%	170.0	355.518	29.4%	180.6	378.936
HK P=0.1	20.0%	0.8	**0.106**	20.0%	0.8	0.108	20.0%	0.8	0.112
HK P=0.2	20.0%	1.0	**0.181**	20.0%	1.0	0.198	20.0%	1.0	0.184
HK P=0.3	0%	3.2	**0.489**	0%	3.2	0.552	0%	3.2	0.51
HK P=0.4	4.8%	5.2	**0.416**	4.8%	4.0	0.398	4.8%	5.2	0.438

Table 6.1.3.1: Comparison of different model strengthening constraints: (2.16) which ensures that variable w_j is set only if a node uses color j, and (2.17) and (2.18) which eliminate certain fractional constraints, adding over all colors of node and of a partition, respectively.

$$w_j \geq w_{j+1} \qquad \forall 1 \leq j < c \qquad (2.11)$$

$$\sum_{i \in V} x_{ij} \geq \sum_{i \in V} x_{ij+1} \qquad \forall 1 \leq j < c \qquad (2.12)$$

$$x_{ij} = 0 \qquad \forall j > p(i) + 1 \qquad (2.13)$$

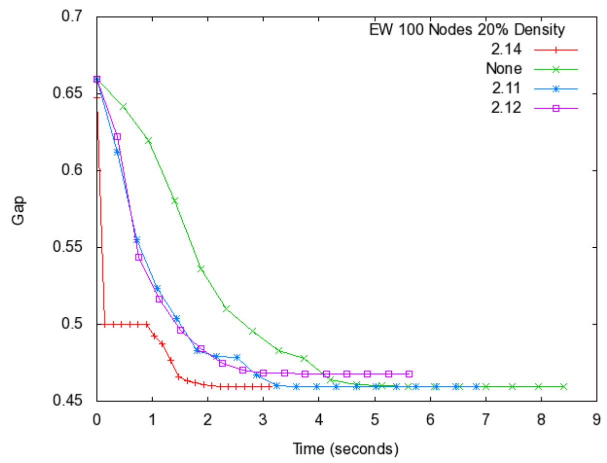
$$x_{ij} \leq \sum_{l=j-1}^{k-1} \sum_{u \in P_l} x_{uj-1} \qquad \forall 1 < k \leq q,\ \forall i \in P_k,\ \forall 1 < j \leq k \qquad (2.14)$$

It is with these constraints that significative changes in solution gaps are found. While there is little difference between applying or not the simplest restrictions 2.11 (although they are required for the validity of other inequalities and bounds), stricter restrictions that further eliminate symmetrical solutions report much lower gaps, in some cases even reaching optimality at this stage.

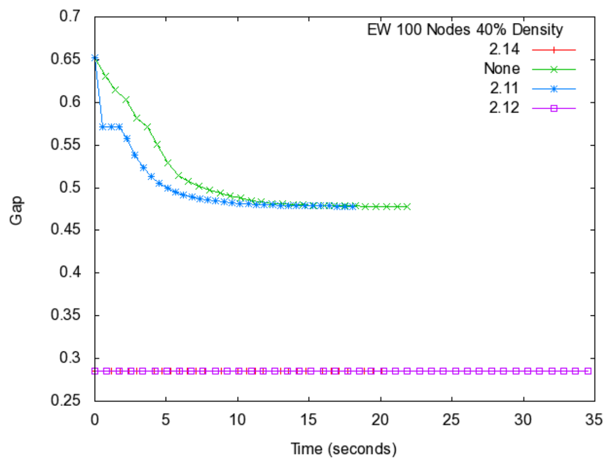
Minimum partition index constraints (2.13,2.14) have the best gaps, require fewer cutting planes iterations, and run within acceptable times (in some cases even faster than its counterparts). The graphics in 6.5 support this, showing that 2.14 either reaches a better gap faster than the others, or simply returns a better gap when solving the first relaxation (without obtaining a big improvement when applying cutting planes).

Id	Constraint 2.14			None			Constraint 2.11			Constraint 2.12		
	gap	niters	time	gap	niters	time	gap	niters	time	gap	niters	time
EW 20%	46.0%	20.8	3.086	**45.8%**	18.2	8.45	**45.8%**	20.4	7.915	46.6%	16.0	6.032
EW 40%	**29.8%**	34.4	20.342	46.6%	31.6	21.244	46.6%	32.6	17.884	31.4%	39.8	31.484
EW 60%	**4.0%**	34.8	153.702	42.0%	73.2	96.82	42.0%	72.6	87.575	16.0%	86.4	433.19
EW 80%	**5.2%**	43.2	299.66	29.4%	192.2	401.3	29.2%	192.4	349.557	16.0%	100.6	202.282
HK P=0.1	20.0%	0.6	0.082	20.6%	4.4	0.26	20.0%	0.8	0.106	**15.0%**	0.6	0.084
HK P=0.2	**12.0%**	0.6	0.144	20.6%	9.8	0.752	20.0%	1.0	0.181	**12.0%**	0.6	0.12
HK P=0.3	0%	2.0	0.314	0%	3.4	0.516	0%	3.2	0.489	0%	2.4	0.454
HK P=0.4	4.8%	3.8	0.316	5.0%	6.2	0.494	4.8%	5.2	0.416	**2.4%**	4.4	0.376

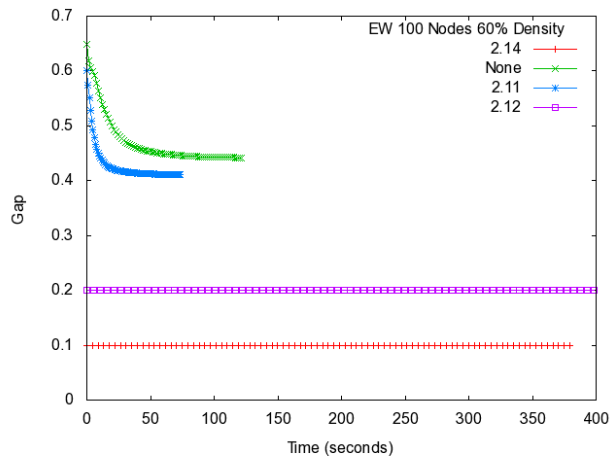
Table 6.1.4.1: Comparison of the inclusion of different symmetry breaking constraints in the model: assigning the lowest color label to the color class with the lowest node index (2.14), applying no constraint whatsoever, forcing lower labels to be used first (2.11) and assigning the lowest color label to the color class with the greatest number of nodes (2.12).



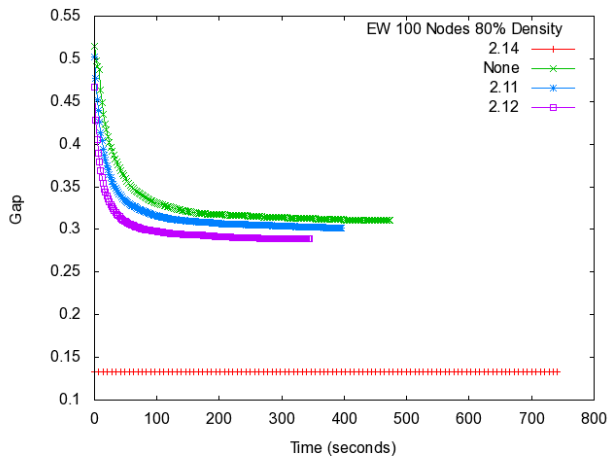
(a) EW 100 Nodes 20% Density



(b) EW 100 Nodes 40% Density



(c) EW 100 Nodes 60% Density



(d) EW 100 Nodes 80% Density

Figure 6.5: Comparison of the inclusion of different symmetry breaking constraints in the model, visualizing evolution of the gap during time in a cutting planes algorithm. Compared constraints are: assigning the lowest color label to the color class with the lowest node index (2.14), applying no constraint whatsoever, forcing lower labels to be used first (2.11) and assigning the lowest color label to the color class with the greatest number of nodes (2.12).

6.1.5 Chosen formulation from cutting planes

Taking into account all previous results in a cutting planes algorithm, the set of constraints that we will use in the PCP formulation for subsequent computational experiments will be the following:

$$\sum_{i \in P_k} \sum_{j \in C} x_{ij} = 1 \quad \forall P_k \in P \tag{2.4}$$

$$\sum_{i \in N(i_0)} x_{i_0 j} + r x_{i_0 j} \leq r w_j \quad \forall j \in C,\ \forall i_0 \in V \tag{2.10}$$

$$w_j \leq \sum_{i \in V} x_{ij} \quad \forall j \in C \tag{2.16}$$

$$x_{ij} \leq \sum_{l=j-1}^{k-1} \sum_{u \in P_l} x_{uj-1} \quad \forall 1 < k \leq q,\ \forall i \in P_k,\ \forall 1 < j \leq k \tag{2.14}$$

$$x_{ij}, w_j \in \{0,1\} \quad \forall i \in V,\ \forall j \in C$$

First two constraints define the problem itself, by specifying that a node must be colored in each partition and no color conflicts must occur; constraints 2.16 simply strengthen the linear relaxation; and 2.14 eliminate symmetrical solutions. Last set of constraints are the binary restrictions.

Note that while adjacency restrictions 2.7 reported better results than the chosen ones (2.10) in most cases, the latter worked better in dense graphs, which are the ones that, due to a larger problem size, take longer to solve their linear relaxation. Therefore, we opt for improving the resolution of the hardest graphs instead of getting slightly better results in the rest.

6.1.6 Branch and bound testing

While the previous formulation was chosen for working on a cutting planes algorithm, we are also interested in the behaviour of different models in standard branch and bound algorithms.

We tested many variations to the chosen model in a branch and bound algorithm, bound to 1800 seconds, with graphs with 90 nodes, partition size 2 and different densities. The branch and bound uses default CPLEX settings, no custom callbacks were yet applied.

We present in table 6.1.6.2 the following configurations, chosen based on their results:

- **C1:** Chosen model from cutting planes experimentation phase.

Id	C1	C2	C3	C4	C5	C6	C7
EW 20 N=90	**0%**	**0%**	25.0%	**0%**	43.0%	25.0%	25.0%
EW 40 N=90	33.0%	**22.0%**	33.0%	33.0%	33.0%	28.0%	35.0%
EW 60 N=90	39.0%	37.0%	41.0%	37.0%	48.0%	**15.0%**	38.0%
EW 80 N=90	38.0%	43.0%	31.0%	39.0%	41.0%	**10.0%**	38.0%

Table 6.1.6.1: Gap obtained in a standard branch and bound algorithm for different models.

- **C2:** Relaxes that exactly one node must be painted per partition (2.4) by replacing it with at least one painted per partition (2.6).
- **C3:** Uses simple color conflict constraints, requiring that two adjacent nodes cannot use the same color (2.7).
- **C4:** Strengthens the model using not only 2.16 restrictions but also applying 2.18.
- **C5:** Uses no symmetry breaking constraints whatsoever.
- **C6:** Bases symmetry breaking on the number of nodes of each color class (2.12).
- **C7:** Applies constraints 2.9 for color conflict.

Results were most interesting. The formulation chosen for the cutting planes algorithm yielded good results only for lowest density graphs. In other cases, using different models returned better results:

- In graphs with 40% density, relaxing the 2.4 constraint on painting one node per partition greatly reduces the obtained gap, as can be seen in the results for $C2$.
- In the most dense graphs, varying the strategy for symmetry breaking to use constraint 2.12 yields much better results.

Considering the results returned by using constraint 2.12, we re-tested all of the previous configurations changing the default symmetry breaking strategy from 2.14 to 2.12, except for C1 which kept the original settings. This time, we executed a branch and bound algorithm on much smaller graphs (60 nodes), restricting ourselves to 10-minute runs and **not** providing the algorithm with any initial solution, in order to obtain zero gap and compare running times.

Strategy	40%	60%
C1	17.41	151.12
C2	15.42	55.51
C4	4.47	30.30
C3	**0.37**	30.06
C6	4.48	30.50
C5	**0.78**	56.68
C7	51.47	**15.22**

Table 6.1.6.2: Running time until resolution obtained in a standard branch and bound algorithm for different models, in binomial graphs with 60 nodes, without providing an initial heuristic solution.

The most interesting results are found in 40% and 60% density graphs, and are displayed in table 6.1.6.2. It is also worth noting that C7 was the only one to solve all 80% density graphs to optimality.

This time, using a simpler model, both regarding color conflict and symmetry breaking, yielded very good results in low density graphs. In higher densities, applying constraints 2.9 for color conflicts produces excellent results, which is surprising as the efectiveness of these constraints was highly related to partition sizes, not densities.

Throughout this chapter we have tested a significant number of different models, which were obtained by combining different constraints, and tested them in different scenarios by using different algorithms. Results changed greatly between those scenarios.

We arrived to a formulation in 6.1.5 based in data reported by a cutting planes algorithm, which will be used throughout the following sections when testing other components of the algorithm.

However, experimentation in 6.1.6 with branch and bound algorithms showed that that formulation may not be the best choice for all scenarios. Although we will stick to the 6.1.5 formulation for the following sections, all the data obtained in this last subsection will be used to know which alternative models should be tested under the branch and cut algorithm, once primal heuristic, separation heuristics and branching strategies have been properly set.

6.2 Partitioned dsatur

Considering we had three different partitioned DSATUR implementations (see section 4.3), we ran quick tests on multiple graphs to determine how they performed when executed for short periods of time. For each instance, we executed the different algorithms for one minute, and report the lowest bound for the chromatic number obtained, as well as how fast was this bound obtained. Since we will be using DSATUR mostly as an heuristic, it is of our interest that good solutions are found as quickly as possible.

6.2.1 Hardest partition parametrization

Before comparing the three different algorithms, we had to fix the criteria used to pick the *hardest partition* on each call in this variant of the algorithm[1]. As we had already mentioned, the criteria used is a combination of:

- Color degree of the partition, defined as the number of different colors adjacent to all of the nodes in the partition; considering that a larger color degree implies a harder partition to color
- Size of the partition, as a larger partition being colored earlier helps reducing the problem size, therefore the larger the partition the earlier it should be handled
- Number of uncolored partitions adjacent to the partition, equivalent to the tie breaking criteria used for classic DSATUR

We generated six different configurations, based on all different possible orderings of these items. For example, for the first configuration, we first compared by the number of adjacent uncolored partitions, then by the degree of saturation, and finally by the size of the partition. The following are the configurations we established:

- **C1:** Uncolored, saturation, size
- **C2:** Saturation, uncolored, size
- **C3:** Uncolored, size, saturation
- **C4:** Saturation, size, uncolored
- **C5:** Size, uncolored, saturation

[1]Recall that this strategy picked the hardest partition on each call, and then the easiest node from it.

- **C6:** Size, saturation, uncolored

Results in table 6.2.1.1 show little difference for most instances in which partition size is constant (fixed to two nodes). Whereas in some cases, mostly those with lower density, configurations C1, C3 and C5 (those who choose the hardest partition by number of uncolored neighbour partitions before than by saturation degree) find the solution earlier, in other cases configurations C2, C4 and C6 report better times. All of them find always nearly the same values for the chromatic number.

Differences arise, however, when we have different partition sizes. Configurations C1 and C3, those who pick the partition on uncolored neighbours, obtain better bounds in less time than the others. Surprisingly, configurations based on partition sizes offer the worst results for these cases.

For graphs with partition sizes fixed to 2, which are the graphs with which we will be working mostly, we will not take into consideration configurations that use partition size as a criteria; also, as graphs with higher density have been taking the longest time to process, we will prefer a configuration that best deals with these cases, such as C2.

6.2.2 Strategies comparison

After fixing the *hardest partition* strategy to C2, we will compare its performance with both the *easiest node* and the *randomized easiest node* variants. We ran the same tests as before, and present the results on table 6.2.2.1.

Regardless of the configuration chosen for the *hardest partition* variant, both *easiest node* strategies find much better bounds within the one-minute running time. Within those two, the deterministic one offers slightly better results, so it is chosen as the algorithm that we will be using for the following experimentations.

As for the randomized version, we speculate that it might perform better in larger graphs for lengthier running periods, since it has a chance to find a different solution than the deterministic and obtain a sudden improvement on the bound, whereas the deterministic may spend several iterations trying similar assignments. However, in very short runs, like these ones, the deterministic version is clearly preferred.

Graphs	C1		C2		C3		C4		C5		C6	
	chi	found	chi	found	chi	found	chi	found	chi	found	chi	found
EW 20% N=140	6.0	0.192	**5.8**	1.514	6.0	0.196	**5.8**	1.504	6.0	0.184	**5.8**	1.5
EW 40% N=140	10.0	6.372	**9.6**	16.878	10.0	6.384	**9.6**	16.874	10.0	6.382	**9.6**	16.886
EW 60% N=140	**14.2**	27.692	14.4	16.208	**14.2**	27.758	14.4	16.154	**14.2**	27.79	14.4	16.184
EW 80% N=140	21.8	8.29	21.8	8.138	21.8	8.314	21.8	8.14	21.8	8.294	21.8	8.148
EW 50% N=140 P=(1..2)	16.2	1.414	16.0	5.232	16.2	1.414	**15.6**	0.428	18.0	19.652	**15.6**	0.612
EW 50% N=140 P=(1..3)	**10.4**	6.61	11.4	11.576	**10.4**	6.648	12.4	9.7	13.0	1.17	12.6	0.068
EW 50% N=140 P=(1..4)	**9.2**	6.788	9.4	10.542	**9.2**	6.794	10.8	5.37	11.0	30.262	10.8	14.542
EW 50% N=140 P=(2..3)	8.8	3.366	8.8	0.486	8.8	3.35	9.8	0.37	9.6	6.964	9.8	0.382
EW 50% N=140 P=(2..4)	**7.2**	0.366	7.2	16.604	**7.2**	0.36	8.2	20.728	8.8	0.15	8.2	25.316
EW 50% N=140 P=(3..4)	6.6	1.504	6.6	4.122	6.6	1.506	**6.6**	0.23	**6.6**	0.246	**6.6**	0.23
EW 50% N=080	7.0	2.248	7.0	**0.658**	7.0	2.252	7.0	**0.648**	7.0	2.25	7.0	**0.656**
EW 50% N=100	8.2	19.472	**8.0**	10.01	8.2	19.504	**8.0**	10.008	8.2	19.526	**8.0**	10.014
EW 50% N=120	10.0	12.04	10.0	**1.578**	10.0	12.066	10.0	**1.588**	10.0	12.058	10.0	**1.586**
EW 50% N=140	**11.6**	8.88	11.8	3.832	**11.6**	8.892	11.8	3.81	**11.6**	8.904	11.8	3.826
EW 50% N=160	13.4	6.066	13.4	6.352	13.4	6.07	13.4	6.344	13.4	6.068	13.4	6.35
EW 50% N=180	14.0	**1.376**	14.0	6.962	14.0	**1.382**	14.0	6.97	14.0	**1.382**	14.0	6.92
EW 50% N=200	16.2	0.57	**15.4**	0.766	16.2	0.574	**15.4**	0.776	16.2	0.572	**15.4**	0.774
HK P=0.1 N=140	6.4	0.106	6.4	0.02	6.4	0.106	6.4	0.02	6.4	0.114	6.4	0.02
HK P=0.2 N=140	9.4	0.114	**9.0**	9.804	9.4	0.112	**9.0**	9.824	9.4	0.11	**9.0**	9.846
HK P=0.3 N=140	12.4	3.066	**11.8**	20.746	12.4	3.066	**11.8**	20.758	12.4	3.066	**11.8**	20.818
HK P=0.4 N=140	14.8	21.552	**14.2**	6.8	14.8	21.576	**14.2**	6.798	14.8	21.588	**14.2**	6.792

Table 6.2.1.1: Best value obtained for the chromatic number and time at which this value was obtained in one-minute runs of the *hardest partition* version of DSATUR, using different combinations of strategies to pick the hardest partition at each call.

Graphs	*easiest node*		*randomized node*		*hardest partition*	
	chi	found	chi	found	chi	found
EW 20 N=140	**5.0**	**0.003 (5)**	5.0	0.016 (1)	5.8	1.513 (0)
EW 40 N=140	**8.0**	**4.172 (5)**	8.2	27.334 (0)	9.6	16.875 (0)
EW 60 N=140	**12.4**	**10.062 (3)**	12.4	4.656 (2)	14.4	16.210 (0)
EW 80 N=140	18.6	25.522 (2)	**18.4**	**10.769 (3)**	21.8	8.137 (0)
EW 50 N=60	**5.0**	**0.019 (5)**	5.0	0.097 (3)	5.0	2.972 (0)
EW 50 N=80	**6.0**	**2.584 (5)**	6.4	0.778 (0)	7.0	0.659 (0)
EW 50 N=100	**7.2**	**1.647 (5)**	7.2	15.728 (1)	8.0	10.009 (0)
EW 50 N=120	**9.0**	**0.041 (5)**	9.0	1.619 (1)	10.0	1.578 (0)
EW 50 N=140	**10.2**	**1.594 (4)**	10.0	33.262 (1)	11.8	3.831 (0)
EW 50 N=160	**11.2**	**5.134 (4)**	11.8	2.575 (1)	13.4	6.353
EW 50 N=180	**12.6**	**10.241 (5)**	13.0	4.819 (1)	14.0	6.960 (0)
EW 50 N=200	**13.6**	**14.634 (4)**	14.0	0.966 (1)	15.4	0.766 (0)
EW 50 N=140 P=(1..2)	**13.6**	**4.353 (3)**	13.8	2.347 (2)	16.0	5.231 (0)
EW 50 N=140 P=(1..3)	9.6	31.634 (1)	**9.8**	**8.309 (4)**	11.4	11.575 (0)
EW 50 N=140 P=(1..4)	**8.0**	**6.466 (5)**	8.8	9.797 (0)	9.4	10.544 (0)
EW 50 N=140 P=(2..3)	**7.4**	**11.172 (5)**	8.0	0.047 (2)	8.8	0.488 (0)
EW 50 N=140 P=(2..4)	**6.8**	**0.703 (5)**	7.0	1.225 (0)	7.2	16.606 (0)
EW 50 N=140 P=(3..4)	**5.8**	**1.497 (5)**	6.0	0.650 (1)	6.6	4.122 (0)
HK P=0.1 N=140	**4.0**	**0.016 (5)**	4.0	0.044 (3)	6.4	0.019 (0)
HK P=0.2 N=140	**6.0**	**0.016 (5)**	6.0	0.166 (3)	9.0	9.803 (0)
HK P=0.3 N=140	**8.0**	**0.009 (5)**	8.0	0.119 (1)	11.8	20.744 (0)
HK P=0.4 N=140	**9.8**	**0.003 (5)**	9.8	0.006 (4)	14.2	6.800 (0)

Table 6.2.2.1: Best value obtained for the chromatic number and time at which this value was obtained in one-minute runs for the *hardest partition*, *easiest node* and *randomized easiest node* versions of DSATUR. Value between parentheses indicates for how many of the five instances that strategy was considered the winner.

6.3 Branching strategies

We evaluated the different branching strategies described in section 5.7 on regular graphs with fixed size and different density, in order to determine which reports the best results. We used a simple branch and bound algorithm bounded to 15 minutes of running time.

6.3.1 Priorities

The first test we implemented applied only priorities on the variables during the problem's initialization. Priorities were assigned according to the following formula:

$$prio(x_{ij}) = \alpha\delta_P(v_i) + \beta j$$

We tested with different values for α and β, both positive and negative, to generate different priorities. Although we found hardly any differences in higher density graphs, the ones with the lowest densities (20%) did have significative differences.

In table 6.3.1.1 we report those (α, β) values which gave results better or near the ones obtained when not using priorities, this is, allowing CPLEX to choose automatically which variable to branch on.

Priorities	**Time**	**Gap**
$\alpha = 10$, $\beta = -1$	232.57	0%
$\alpha = 10$, $\beta = 1$	523.10	0%
cplex	570.72	0%

Table 6.3.1.1: Gap and running time for branch and bound executions on 20% density graphs with different priorities on the branching variables.

Clearly giving the highest priority to nodes with the highest $\delta_P(v_i)$ value, tie-breaking in favor of higher color labels, is the best static branching priority.

6.3.2 Dynamic strategies

Having fixed the priorities to set on the variables, we use them as tie breaking strategies for the two devised strategies which depend on the variable's value (5.7.2 and 5.7.3).

We set up a suite of graphs of different size and density to test most and less fractional strategies, as well as both degree of saturation strategies: branching on a particular x_{ij} variable or creating one subproblem for each possible color for a particular node v_i. Results are displayed in table 6.3.2.1;

we report the resulting MIP gap, on which node that gap was obtained, and how many nodes were explored during the 15 minutes of execution.

Within fractional strategies, branching on the *most fractional* variable generates the best results, although there is little difference with the *less fractional* criteria. There is a significative difference, mostly in low-density graphs, between fractional and degree of saturation strategies. Whereas the former requires less computational time to execute and allows the algorithm to explore a larger number of nodes, the latter obtains a much smaller gap much earlier in the exploration.

We will be using degree of saturation criteria, and test its both alternatives in conjunction with a custom primal heuristic to determine the best branching and primal configuration for the problem.

However, the obtained gaps were better using fixed priorities on the variables than using either dynamic strategy. We may infer that the overhead generated by overriding the engine's default behavior, and the computational effort required iterating all the variables, cause the custom strategies to behave poorer than the fixed priorities. We will compare them again once we add the custom primal heuristic in section 6.4.

6.3.3 Implied bounds

As explained in section 5.7.4, whenever we fix a variable $x_{i^*j^*}$ to 1 when branching, we have the possibility of fixing the value of other variables due to logical implications:

- Fix all other variables in the partition to zero, as only one node must be painted within the partition.

$$x_{ij} = 0 \quad \forall i \in P(i^*),\ \forall j \in C,\ i \neq i^* \vee j \neq j^*$$

- Fix all variables corresponding to adjacent nodes with same color to zero, due to color conflict restrictions.

$$x_{ij^*} = 0 \quad \forall i \in N(i^*)$$

- Fix all color variables w_j to 1 for all labels less or equal than the current lower bound χ_{LB} on the chromatic number.

$$w_j = 1 \quad \forall j \leq \lceil \chi_{LB} \rceil$$

We re-ran the previous test with the DSATUR-2 configuration, with and without these bounds, to check if the overhead generated by forcing multiple

Graph	DSATUR-2			DSATUR-(C+1)			Less fractional			Most fractional		
	gap	found	nodes	gap	found	nodes	gap	found	nodes	gap	found	nodes
EW 20% N=90	**17%**	77	94	**17%**	**76**	95	25%	177	250	25%	134	178
EW 40% N=100	**33%**	20	24	**33%**	**14**	24	39%	27	39	33%	30	44
EW 60% N=80	37%	**7**	18	37%	18	19	37%	30	32	37%	23	27
EW 80% N=100	42%	2	2	42%	**1**	3	44%	3	4	42%	4	4

Table 6.3.2.1: Results for fractional and degree of saturation (spanning either 2 or $C + 1$ subproblems) branching strategies on branch and bound schemes. Data reported is MIP gap after 15 minutes of execution, on which node (in thousands) that gap was found, and how many nodes (in thousands) were explored in total.

x_{ij} bounds	w_j bounds	gap
On	On	32.25%
On	Off	42.25%
Off	On	42.25%
Off	Off	42.25%

Table 6.3.3.1: Gaps on branch-and-bound executions for different bounds set during branching, using DSATUR-2 branching strategy.

bounds on each branching is compensated by the reduction in the branch-and-bound tree. Table 6.3.3.1 reflects the average gaps for different bounds set.

Applying all bounds during the process clearly reports the best results. The total number of nodes generated during the process was similar between all the configurations, so the implied bounds do not generate a noticeable overhead.

Therefore, we will be applying all bounds for both x_{ij} and w_j variables for all upcoming tests.

6.4 Primal Heuristic

In this section we evaluate the effectiveness of the devised primal heuristic, in comparison with the default heuristic provided by the CPLEX engine, in the context of a branch and bound algorithm. We used the same test suite as in section 6.3.

6.4.1 Using priorities branching

We first tested the primal heuristic using the simple priorities branching scheme, which offered a good performance according to the results presented in table 6.3.1.1. We ran our branch and bound using CPLEX's default heuristic, our custom degree-of-saturation primal heuristic, and a combination of both. Executions were bounded to 30 minutes each, and we executed three instances of each graph kind. Results are presented in table 6.4.1.1; for these tests we evaluated not only the resulting gap, but also the improvement in the upper bound from the initial solution (obtained by the initial heuristic) to the final solution returned by the algorithm.

Graph	CPLEX		Priorities		Priorities+CPLEX	
	Initial (impr)	Gap	Initial (impr)	Gap	Initial (impr)	Gap
EW 20 N=90	4.0 (0/3)	-	4.0 (0/3)	-	4.0 (0/3)	-
EW 40 N=100	6.0 (0/3)	44.8%	6.0 (0/3)	45.6%	6.0 (0/3)	45.6%
EW 60 N=80	8.0 (0/3)	38.7%	8.0 (0/3)	39.3%	8.0 (0/3)	39.3%
EW 80 N=100	14.3 (0/3)	45.9%	14.3 (**2**/3)	43.8%	14.3 (**2**/3)	43.7%

Table 6.4.1.1: Number of colors obtained in the initial heuristic, number of instances in which this upper bound was improved by the primal heuristic and resulting gap, for different primal heuristics in a branch and bound using priorities branching.

These results show that the solution obtained initally is very difficult to be improved during the branch and bound process, as it is very close to (or effectively is) the optimal solution. Only in the most dense graphs a coloring better than the initial one was found, and it occured in two of three instances, and in both cases the difference was just a single color. It is worth noting that it is the DSATUR heuristic that improves the initial solution, whereas the default one provided by CPLEX does not.

However, in graphs other than the 80%-density ones, the resulting gap is better when the custom DSATUR primal heuristic is turned off. This is because the time spent executing the heuristic is invested in walking the branch and bound tree, this improving the lower bound and reducing the gap.

It is important to note thet we also tried alternative configurations, in which the primal heuristic was applied much more aggressively, or for a longer period of time each time it was invoked, obtaining exactly the same improvements in the upper bound as the ones just presented.

6.4.2 Using dsatur-(C+1) branching

Next, we used the DSATUR-(C+1) branching criteria instead of the priorities branching strategy. We analyze the same results as before for the same three settings: using only CPLEX default primal heuristic, using only our custom DSATUR primal heuristic, and using both at the same time. Results are presented in table 6.4.2.1.

Graph	CPLEX		DSATUR		DSATUR+CPLEX	
	Initial (impr)	Gap	Initial (impr)	Gap	Initial (impr)	Gap
EW 20 N=90	4.0 (0/3)	16.7%	4.0 (0/3)	16.7%	4.0 (0/3)	16.7%
EW 40 N=100	6.0 (0/3)	33.3%	6.0 (0/3)	33.3%	6.0 (0/3)	33.3%
EW 60 N=80	8.0 (0/3)	37.2%	8.0 (0/3)	37.5%	8.0 (0/3)	37.5%
EW 80 N=100	14.3 (0/3)	41.4%	14.3 (**2**/3)	40.3%	14.3 (**2**/3)	38.9%

Table 6.4.2.1: Number of colors obtained in the initial heuristic, number of instances in which this upper bound was improved by the primal heuristic and resulting gap, for different primal heuristics in a branch and bound using DSATUR-(C+1) branching.

The conclusions that can be drawn from these results are the same as the ones from the previous set, only that in this case the increase in the gap when the primal heuristic is turned on, is much smaller in 40% and 60% density graphs. The custom DSATUR primal heuristic does find a better integral solution in the same two graphs, and CPLEX still does not.

What is worth noting is that there is a considerable reduction in the resulting gap when the default primal heuristic is used along with the custom one. Even though the upper bounds obtained in both cases is exactly the same, the gap becomes actually smaller when more effort is put into the primal heuristic, as *the lower bound is increased when* CPLEX*'s primal heuristic is used.* We could not find a sensible explanation for this situation, and CPLEX not disclosing its internal behaviour regarding to default procedures was certainly not helpful.

6.4.3 Comparison

Even though the priorities branching scheme by itself generated better results than the degree of saturation branching strategy, when adding a primal heuristic the results changed. Except for low-density graphs, in which the overhead generated by the branching strategy seems counter-productive, DSATUR-(C+1) branching performed better than its counterpart.

Regarding the primal heuristic itself, as long as the initial heuristic produces solutions very close to the optimum, any primal heuristic will not be able to improve the upper bound, and will end up reducing the time available for exploring the tree without producing any useful results. This seems to be the case for most graphs, except for the ones with the highest density.

6.5 Cuts

Given the different cutting planes families we had implemented, we tested their effectiveness in a cutting planes algorithm. The algorithm was executed until no further planes could be generated, which took no more than a few hundred rounds. We used binomial graphs with fixed size and partitions of exactly two nodes, with different densities.

The tests we executed were aimed at determining the performance of each family, as well as the engine's default generic separation algorithms. We first measured the resulting gap and required time after all the cutting planes rounds that could be executed; these values are shown in table 6.5.1.1.

We were also interested in how many cuts of each family were generated by our algorithm, in order to know with how many inequalities each algorithm was contributing to the cutting planes scheme. These data, for the same graphs and families combination, with and without CPLEX's cuts enabled, is presented in table 6.5.1.2.

6.5.1 Results

The first clear conclusion, drawn from the first table, is that extended clique and block color cuts alone produce a great improvement in the obtained gap, with respect to CPLEX's generic cuts, which is easy to explain as they are cuts created particularly for this problem. Since denser graphs have larger cliques, it is also understandable that in those cases the gap improvement is even larger.

As for the addition of the other cuts families, there is little difference with respect to the results obtained using only clique and block color cuts, not only in gap but also in total running time, which implies that the running time of their separation algorithms is rather low. The addition of CPLEX's generic cuts reports a small benefit in low-density graphs, in which clique cuts cannot be exploited much.

It is worth noting from table 6.5.1.2 that there is a huge difference between the number of clique cuts generated and the other families. Even in low-density graphs, extended clique cuts are an order of magnitude greater than component hole or path cuts.

As for partitions graph independent set cuts, they are considerably difficult to generate, as partitions graphs tend to have a very high number of unconnected nodes. Only a few of these cuts are generated in some medium-density graphs, which explains the lack of change in gap or rounds when adding these cuts to the cutting planes algorithm.

Given these observations, our strategy for the branch and cut algorithm will be to aggressively apply block color and extended clique cuts until no more than a number K is generated in a round, in which case we also enable component independent set cuts in an attempt to remove the fractional solution. Therefore, most of the computational effort will be centered around cliques, which are the ones that provide the best cuts, and only fall back to the rest when these family fails to find a valid cut.

Graph	None			EClique + BColor			EClique + BColor + CISet			All		
	gap	rounds	time	gap	rounds	time	gap	rounds	time	gap	rounds	time
EW 20 N=100	50%	4.00	0.71	46%	24.40	2.63	46%	29.00	3.81	46%	29.00	3.83
EW 40 N=100	49%	8.80	2.60	39%	35.00	11.37	39%	37.40	10.97	39%	40.80	11.86
EW 60 N=100	47%	34.00	14.34	38%	71.20	55.59	38%	81.80	55.44	38%	84.40	56.78
EW 80 N=100	40%	27.80	14.77	25%	120.80	156.07	25%	136.40	165.60	25%	136.40	166.77
EW 20 N=100				47%	25.00	2.57	48%	24.40	3.17	48%	24.40	3.18
EW 40 N=100				39%	35.00	9.67	39%	39.00	11.90	39%	30.80	10.38
EW 60 N=100				38%	61.60	43.44	38%	70.80	48.40	38%	70.80	48.64
EW 80 N=100				25%	157.80	166.97	25%	127.00	150.30	25%	127.00	151.46

Table 6.5.1.1: Gap, total number of cutting planes rounds in the root, and total time for different cutting planes families enabled in the cut and branch algorithm. The first group has CPLEX default cuts enabled as well, while the second one does not.

Graph	Cplex OFF						Cplex ON					
	BColor	Clique	Hole	Path	PGHole	PGPath	BColor	Clique	Hole	Path	PGHole	PGPath
	EClique + BColor											
EW 20 N=100	23.60	2072.60	0.00	0.00	0.00	0.00	19.80	2121.80	0.00	0.00	0.00	0.00
EW 40 N=100	49.60	3467.20	0.00	0.00	0.00	0.00	42.00	3274.40	0.00	0.00	0.00	0.00
EW 60 N=100	105.80	6233.40	0.00	0.00	0.00	0.00	128.40	6286.20	0.00	0.00	0.00	0.00
EW 80 N=100	98.40	12178.00	0.00	0.00	0.00	0.00	95.60	11266.80	0.00	0.00	0.00	0.00
	EClique + BColor + CISet											
EW 20 N=100	32.80	2088.60	394.60	335.00	0.00	0.00	23.00	2220.80	398.20	269.60	0.00	0.00
EW 40 N=100	47.20	3679.40	164.00	248.00	0.00	0.00	45.00	3215.60	121.60	169.80	0.00	0.00
EW 60 N=100	106.00	6326.80	87.40	37.00	0.00	0.00	127.40	6222.40	73.80	30.80	0.00	0.00
EW 80 N=100	106.00	11362.00	26.60	20.40	0.00	0.00	93.60	11561.60	23.80	16.60	0.00	0.00
	EClique + BColor + CISet + PG CISet											
EW 20 N=100	32.80	2088.60	394.60	335.00	0.00	0.00	23.00	2220.80	398.20	269.60	0.00	0.00
EW 40 N=100	44.00	3469.40	179.80	246.00	0.00	4.40	45.60	3343.20	120.00	176.40	0.00	2.40
EW 60 N=100	106.00	6326.80	87.40	37.00	0.00	0.00	126.60	6258.80	73.20	30.60	0.00	0.20
EW 80 N=100	106.00	11362.00	26.60	20.40	0.00	0.00	93.60	11561.60	23.80	16.60	0.00	0.00

Table 6.5.1.2: Number of cuts generated in the cutting planes algorithm for each kind with different configurations enabled.

6.6 Branch and cut

Last but not least, in determining the best configuration for the different components of a branch and cut algorithm, we evaluated the algorithm's performance with different settings relative to the whole branch and cut process. We evaluated different criteria for running the exhaustive implicit enumeration in subtrees, as described in 5.9, and also different MIP relative parameters in the underlying CPLEX framework we used.

6.6.1 Exhaustive implicit enumeration

Our first test, once most parameters in the branch and cut algorithm were fixed, was to determine the threshold to run a full DSATUR on a node once enough partitions' colors had been fixed during the branching process. Since the algorithm considered only non-fixed partitions for its execution, we experimented with values within acceptable ranges for an exhaustive enumeration: we chose 20, 40 and 60 as the number of remaining partitions to color which triggered the enumeration.

We used graphs of 100 nodes with 2 nodes per partition and different densities, in branch and cut executions of 30 minutes, to check the behaviours of these strategies.

Results were not encouraging, as shown in table 6.6.1.1. Setting a low number of unfixed partitions as the threshold to start the exact algorithm caused the algorithm to be never invoked, as the branch and cut itself could prune the whole subtree after very few partitions were colored in the branch process.

On the other hand, making the exact DSATUR start earlier caused the algorithm to consume much more time than the available, surpassing the 1800 seconds bound for high-density graphs, or simply left less time to explore a larger number of nodes, in both cases greatly hurting the obtained gap.

As a result of these experiments, we will be enabling exhaustive implicit enumeration only for very low thresholds (20 partitions pending) in order to avoid any problems caused by running the exact algorithm for long periods. Although this setting may cause the algorithm to never be triggered, in other cases such as larger graphs with a greater time bound there is a possibility for this algorithm to actually be effective.

6.6.2 Probing

An available setting in the CPLEX framework is the probing level. This controls how much processing is invested in a preprocessing stage to derive

Graph	20			40			60		
	# times	nodes	gap	# times	nodes	gap	# times	nodes	gap
EW 20%	0.00	11833	25%	0.00	11834	25%	115.33	11877	25%
EW 40%	0.00	2341	22%	0.00	2341	22%	192.33	2368	30%
EW 60%	0.00	1226	22%	0.00	1226	22%	345.00	1050	29%
EW 80%	0.00	308	19%	2.00	314	19%	28.00	119	21% (*)

Table 6.6.1.1: Average number of times the enumeration was triggered, number of nodes in the tree and resulting gap, for different number of uncolored partitions for triggering the exhaustive enumeration; on graphs with 100 nodes and 2 nodes per partition. The execution marked with a (*) indicate that the execution of the enumeration algorithm took an unacceptable amount of time for the imposed bounds.

logical implications from setting binary variables to a fixed value. As CPLEX's manual [29] explains:

> Probing is a technique that looks at the logical implications of fxing each binary variable to 0 (zero) or 1 (one). It is performed after preprocessing and before the solution of the root relaxation. Probing can be expensive, so this parameter should be used selectively. On models that are in some sense easy, the extra time spent probing may not reduce the overall time enough to be worthwhile. On diffcult models, probing may incur very large runtime costs at the beginning and yet pay off with shorter overall runtime.

We experimented with binomial graphs of fixed size and different densities, as usual, with different probing levels set. Results are shown in table 6.6.2.1, and differ greatly between different densities.

For low densities, a moderate level of probing seems to be the best option, as it managed to explore a greater amount of nodes in the tree during the imposed 1800 seconds.

On the other hand, greater densities seems to benefit more from disabling probing whatsoever, as the custom bounds implied during the branch process (see 5.7.4) benefit largely from higher-degree nodes, making the engine's probing unnecesary and yielding a better gap.

Therefore, we will be using moderate probing settings for low density graphs, and disabling probing altogether for higher densities.

Graph	disabled		moderate		aggressive		very aggressive	
	nnodes	gap	nnodes	gap	nnodes	gap	nnodes	gap
EW 20%	11319.00	25%	23284.33	25%	24523.67	25%	24517.67	25%
EW 40%	2366.67	22%	**5396.67**	**22%**	2348.00	22%	2345.33	22%
EW 60%	1227.67	22%	1227.33	22%	1171.67	22%	1171.33	22%
EW 80%	**347.00**	**15%**	346.00	19%	308.33	19%	309.00	19%

Table 6.6.2.1: Average number of nodes in the tree and resulting gap, for different MIP probing levels; on graphs with 100 nodes and 2 nodes per partition.

6.6.3 Emphasizing feasibility and optimality

Arriving to an optimal solution in a branch and cut algorithm requires both (1) obtaining integral feasible solutions of decreasing objective value, and (2) generate a proof that the best integral solution obtained is actually an optimum. The emphasis the framework puts on these two parts of the algorithm is controlled by a *MIP emphasis* parameter, which can be given the following values:

- **Balanced:** Have a reasonable balance between feasibility and optimality, which is the default behaviour.
- **Emphasize feasibility:** Focus on feasibility instead of optimality, which produces better solutions earlier and works better under tight time constraints when an optimality proof is not necessary.
- **Emphasize optimality:** Focus on the proof of optimality by attempting to raise the best bound[2] faster.
- **Emphasize best bound:** Focus even more in the proof of optimality by attempting solely to move the best bound; this causes intermediate optimal solutions to be rarely found as it cares exclusively to arrive to a final optimal solution.
- **Hidden feasibility:** Attempts to find high quality feasible solutions that are considered hidden, this is, difficult to obtain through the branch and cut process; this causes the proof of optimality to take longer than with other settings.

We evaluated these different configurations in the usual set of binomial graphs, reporting both gaps and number of nodes explored in the tree; results

[2]The best bound is the lowest possible value that an integer feasible solution could have.

are shown in table 6.6.3.1. All of them arrived to the same gap values, but there were observable differences between the number of nodes explored in the tree.

For low density graphs, emphasizing the best bound yielded the highest number of nodes explored within the same time frame, while in higher density graphs a balanced approach managed to explore more nodes. These configurations will be used for further experimentation, depending on the processed graph's density.

Graph	balanced		feasibility		optimality		best bound		hidden	
	nodes	gap	nodes	gap	nodes	gap	nodes	gap	nodes	gap
EW 20% N=100	11840	25%	11705	25%	18266	25%	**20810**	25%	11841	25%
EW 40% N=100	2343	22%	2186	22%	3055	17%	**3707**	**17%**	2342	22%
EW 60% N=100	**1226**	22%	1219	22%	820	22%	675	22%	1225	22%
EW 80% N=100	**308**	19%	305	19%	188	19%	104	19%	308	19%

Table 6.6.3.1: Average number of nodes in the tree and resulting gap, for different MIP emphasis settings.

Graph	S1			S2			S3		
	gap	time	nodes	gap	time	nodes	gap	time	nodes
EW 20% N=90	16.7%	5295	56954	16.7%	4412	46356	**0.0%**	**3014**	32668
EW 40% N=90	16.7%	5400	17160	16.7%	5400	17237	16.7%	5400	**27439**

Table 6.6.4.1: Average solution gap, running time and number of nodes in the tree, for different model strategies for low density binomial graphs. For each tested graph, all strategies obtained the same number of colors in the solution.

6.6.4 Alternative models

After fixing all of the parameters involved in the different components of the branch and cut algorithm, we decided to revisit the first step and re-evaluate alternative model formulations, this time using the full branch and cut algorithm for comparing the different configurations in executions bounded to 90 minutes.

Most strategies tested are variations of the model chosen originally in section 6.1.5, usually motivated by the results obtained in 6.1.6. Nevertheless, some of them include certain modifications to other artifacts such as primal heuristic, branching strategies or probing; the rationale being evaluating already obtained results in the context of the complete branch and cut algorithm.

We tested separately low-density and high-density binomial graphs, with a fixed number of 90 nodes and 2 nodes per partition.

Low density graphs

For graphs with 20% and 40% density, the following strategies were proposed:

- **Strategy 1:** Relax symmetry breaking constraints, using simple 2.11 instead of 2.14 and 2.13; also disables CPLEX probing feature.
- **Strategy 2:** Relax the restriction that every partition must have exactly one node colored, and allow for more than a single node to have a single color; this is, replace restrictions 2.4 for restrictions 2.6.
- **Strategy 3:** Model being executed in previous tests, as described in section 6.1.5.

High density graphs

As for higher density graphs, 60% and 80%, we tested the following five different strategies:

- **Strategy 1:** Uses 2.7 and 2.8 equations for color conflicts, this is, requires that the sum of two adjacent vertices for the same color is at most 1, which are far more simple than the 2.10 constraints used in previous experiments. Also simplifies symmetry breaking by relying solely in using labels with a lower index first (2.11).
- **Strategy 2:** Model being executed in previous tests, as described in section 6.1.5.
- **Strategy 3:** Color conflicts are handled with constraints 2.9, and uses symmetry breaking constraints 2.11 as strategy 1.
- **Strategy 4:** Uses number of vertices in each color class as symmetry breaking constraint, this is, replaces constraints 2.14 with 2.12; and further strengthens the model by applying contraints 2.18 instead of 2.16, which predicate over whole partitions instead of individual nodes.
- **Strategy 5:** Same as the previous one, but color conflicts are handled with constraints 2.9 as in strategy 3.

Graph		S1	S2	S3	S4	S5
EW 60% N=90	Gap %	23.1%	23.1%	23.1%	23.1%	23.1%
	Solutions	[8,9,9]	[8,9,9]	[8,9,9]	[8,9,9]	[8,9,9]
EW 80% N=90	Gap %	15.8%	15.4%	**11.1%**	15.8%	**11.1%**
	Solutions	[12,13,13]	[13,13,13]	**[12,12,12]**	[12,13,13]	**[12,12,12]**

Table 6.6.4.2: Average resulting gap and number of colors in the obtained solutions, for different model strategies for high density binomial graphs. Running time is not reported as all runs hit the 5400 seconds time bound.

Results

Results for both low-density and high-density tests are presented in tables 6.6.4.1 and 6.6.4.2. Even though graphs with 40% and 60% density exhibited little to no differences among the strategies (the only exception being the higher number of nodes explored in the branch and cut tree by strategy 3 on 40%-density graphs), there were remarkable differences in the most sparse and dense graphs.

All graphs with 20% density were solved to optimality when using the model exhibited in section 6.1.5, which was the original formulation derived from cutting planes experiments.

80%-density graphs, on the other hand, obtained better solutions on strategies 3 and 5, which are the ones that use 2.9 constraints for color conflicts. Recall from section 2.2.2 that this family is simpler than 2.10, which relied on an extended clique coverage of each vertex's neighbourhood; also, in graphs with large partitions or high density, it spans less inequalities than the standard 2.5 constraints.

6.6.5 Primal heuristic and DSatur branching

We also wanted to test the impact of disabling dynamic branching strategies and custom primal heuristic in low density graphs, as these artifacts had proven to have a negative impact in branch and bound algorithms for graphs with very low density (see 6.3 and 6.4).

DSatur Branch	Primal Heuristic	EW 20% N=90	EW 40% N=90
Enabled	Enabled	16.7%	16.7%
Disabled	Enabled	25.0%	27.8%
Disabled	Disabled	25.0%	27.8%

Table 6.6.5.1: Average gap obtained in low density graphs for *Strategy 1* when disabling DSATUR dynamic branching strategy and custom primal heuristic on a branch and cut algorithm.

However, in a branch and cut algorithm, the aforementioned results are reverted, as can be seen in table 6.6.5.1

6.6.6 Cuts Iterations

We also wanted to evaluate the effect of changing how cuts are applied throughout the branch and cut tree. In section 6.5 we tested how the addition of different cuts families improved the obtained results; now we check how varying the number of cuts rounds in each node affected the results.

Results in 6.6.6.1 show that the configuration used so far (100 cut iterations in the root node, 1 iteration per internal node, and local cuts) yields the best results for all densities. As usual, the most noticeable differences appear in graphs with very low or high density.

Cuts settings			Graph density			
iters root / inner	locality	max depth	20%	40%	60%	80%
100/1	local	unbounded	0.0%	16.7%	23.1%	11.1%
100/20	global	10	16.7%	16.7%	25.8%	16.9%
100/20	local	10	0.0%	16.7%	23.1%	13.4%
100/5	local	15	0.0%	16.7%	23.1%	13.4%
500/1	local	unbounded	0.0%	16.7%	23.1%	11.1%

Table 6.6.6.1: Average gap obtained for different cuts settings, varying the number of iterations in the root node and in internal nodes, whether cuts are applied globally to the whole branch and cut tree or only to the local subtree, and maximum tree depth at which cuts are still applied. All graphs are binomial with 90 nodes and 2 nodes per partition, and only density is changed.

6.7 Final Results

Having fixed the best configurations of the algorithm for binomial random graphs in the previous sections, such as models, initial heuristic, branching strategies, primal heuristic and cuts, we now compare our branch and cut algorithm for PCP against other solutions.

6.7.1 Comparison with CPLEX

The first evaluation to perform is to analyze the improvement introduced by the custom modifications we made on top of the CPLEX engine, by comparing our results to those returned by an unmodified execution of CPLEX[3].

We used binomial random graphs with 90 nodes, 2 nodes per partition, and picked 2 instances for each node-density pair; with running time of 2 hours.

First, we compared our algorithm to CPLEX's default branch and cut, both with and without fixing an initial clique and performing other simplifications to the model (described in section 5). In all cases we provided the same initial solution χ_0, which considerably reduced the number of variables in the model by eliminating those x_{ij} and w_j with $j > \chi_0$.

Then, we performed the same tests, but this time using CPLEX dynamic search algorithm, instead of the traditional branch and cut we were using. This algorithm, introduced in version 11 of CPLEX and improved in version 12, uses the same building blocks as traditional branch and cut, but does not allow for user customization via callbacks, therefore working as a black box solver, often yielding better results than its counterpart.

Graph	Cplex branch and cut w/o initial clique		Cplex branch and cut with initial clique		Custom PCP branch and cut	
	gap	time	gap	time	gap	time
EW 20% N=90	0.0%	1.49	0.0%	1.48	0.0%	0.141
EW 40% N=90	16.7%	7200	16.7%	7200	16.7%	7200
EW 60% N=90	33.3%	7200	36.7%	7200	22.2%	7200
EW 80% N=90	26.7%	7200	23.3%	7200	11.3%	7200

Table 6.7.1.1: Average gap and running time in seconds for graphs with different densities, comparing our custom PCP branch and cut algorithm with CPLEX's default branch and cut, with and without fixing an initial clique for the resolution.

[3] All tests were performed against version 12.1 of CPLEX.

Graph	Cplex Dynamic Search w/o initial clique		Cplex Dynamic Search with initial clique		Custom PCP branch and cut	
	gap	time	gap	time	gap	time
EW 20% N=90	0.0%	0.758	0.0%	0.758	0.0%	0.141
EW 40% N=90	16.7%	7200	16.7%	7200	16.7%	7200
EW 60% N=90	22.2%	7200	22.2%	7200	22.2%	7200
EW 80% N=90	11.8%	7200	12.0%	7200	11.3%	7200

Table 6.7.1.2: Average gap and running time in seconds for graphs with different densities, comparing our custom PCP branch and cut algorithm with CPLEX's dynamic search, with and without fixing an initial clique for the resolution.

The obtained results showed that the customizations oriented towards solving the PCP did produce an improvement in the solution. The difference with CPLEX's traditional branch and cut algorithm is remarkable, requiring 10% of the time to solve to optimality in sparse instances, and achieving more than a 10% improvement in graphs with a high density.

As for CPLEX dynamic search, it is clear that it performs much better than its branch and cut, but there are still improvements attained by our algorithm in graphs with very low and high density, in terms of time and gap respectively.

Something interesting to notice is that fixing the initial clique does not always report a benefit when running CPLEX algorithms, even though it did report a significative improvement on our customized algorithm.

6.7.2 Alternative partition sizes

Since we used 2 as the *de facto* partition size for most of our tests, it was pending to analyze how the algorithm performed when varying the size of the partitions. Using binomial random graphs, with 90 nodes, 60% density and partition sizes ranging from 1 to 6, we executed our branch and cut algorithm, as well as CPLEX's branch and cut, and reported gap and running time.

Results in table 6.7.2.1 show that the most difficult graphs to solve are those with partition sizes equal to 1, 2 and 3, regardless of which algorithm is being used. In all those cases, the implemented branch and cut algorithm performed better (or same as) CPLEX's branch and cut. Note that a partition size equal to 1 makes this problem equivalent to standard graph coloring, which is known to be difficult to solve.

Larger partition sizes, such as 4, and specially 5 and 6, are much easier to solve. All instances were solved to optimality by all the algorithms, although

Graph	Cplex branch and cut w/o initial clique		Cplex branch and cut with initial clique		Custom PCP branch and cut	
	gap	time	gap	time	gap	time
EW P=1	37.1%	7200	19.8%	7200	**17.6%**	7200
EW P=2	45.2%	7200	24.0%	7200	**18.8%**	7200
EW P=3	26.7%	7200	26.7%	7200	26.7%	7200
EW P=4	-	2880	-	2878	-	5119
EW P=5	-	16	-	16	-	86
EW P=6	-	20	-	20	-	255

Table 6.7.2.1: Average gap and running time in seconds for graphs with different partition sizes, comparing our custom PCP branch and cut algorithm with CPLEX's branch and cut, with and without fixing an initial clique for the resolution. All graphs are random binomial graphs, have 60% density and 90 nodes.

our branch and cut required slightly more time than CPLEX's. This was an expected result, as we fine-tuned all the parameters of our algorithm in order to excel at dealing with small partition sizes (which are the most difficult to handle); therefore, in other scenarios for which we did not customize it, it can be outperformed by generic solvers.

6.7.3 DIMACS instances

To check the performance of our algorithm on particularly difficult graphs, we chose a few graphs from the DIMACS [1] graph coloring challenge, and arbitrarily partitioned them in partitions of 2 nodes. We once again compared our algorithm to CPLEX's branch and cut, and reported running time and solution gap.

The results in table 6.7.3.1 report only one value for CPLEX's branch and cut, even though we executed it with and without fixing the values for an initial clique. The reason for this is that fixing those values did not report any modification on the obtained results; this could be due to the particular nature of the graphs being tested, since some of them, such as Mycielski's, can have relatively large chromatic numbers while keeping a comparatively very low clique number.

The obtained gaps show there is no winner between both algorithms: depending on the structure of the graph, the custom branch and cut algorithm implemented for PCP performs better than CPLEX's, or vice-versa. The former works better than the latter in medium and high density random graphs (DSJC), as most of the developed artifacts were oriented to-

Graph	Cplex B&C		Custom PCP	
	time	gap	time	gap
dimacs1-FullIns_5	7200	**16.7%**	7200	31.7%
dimacs1-Insertions_5	7200	33.3%	7200	33.3%
dimacs1-Insertions_6	7200	67.0%	7200	**57.1%**
dimacs2-FullIns_4	**452**	**0.0%**	7200	16.7%
dimacs2-FullIns_5	7200	**28.6%**	7200	39.3%
dimacs2-Insertions_5	7200	**50.0%**	7200	56.7%
dimacs3-FullIns_4	7200	14.3%	7200	14.3%
dimacs3-Insertions_4	7200	40.0%	7200	40.0%
dimacs4-FullIns_4	7200	12.5%	7200	12.5%
dimacs4-Insertions_3	3084	0.0%	774	0.0%
dimacs4-Insertions_4	7200	40.0%	7200	40.0%
dimacsDSJC125	7200	32.8%	7200	**22.9%**
dimacsDSJC250	7200	N/A	7200	**49.4%**
dimacsmyciel6	7200	28.6%	7200	26.8%
dimacsmyciel7	7200	50.0%	7200	**49.7%**

Table 6.7.3.1: Average gap and running time in seconds for certain DIMACS challenge graphs, with arbitrary partitions of size 2, comparing our custom PCP branch and cut algorithm with CPLEX's branch and cut.

wards these cases; whereas small or sparse graphs derived from Mycielski's (FullIns-5, Insertions-5) are better handled by CPLEX, the only exception being Insertions-6, with 3% density and 600 nodes.

In most cases, obtained gaps were large for both algorithms, showing the difficult nature of this graphs for the coloring problem, which is clearly extended to partitioned coloring.

6.7.4 Comparison with Asymmetric Representatives Branch and Cut

We also compared our algorithm to the other branch and cut algorithm designed specifically for the PCP we found in the literature: the one devised by Frota, Maculan, Noronha and Ribeiro in [9], based on the asymmetric representatives formulation for traditional coloring, ([6],[5]).

It is worth noting that both implementations and execution environments differ considerably. While the aforementioned algorithm was run under Linux, implemented in C++ and using XPRESS to solve linear relaxations, our algorithm was executed in Windows, implemented in Java and built on top of the CPLEX engine using its Java API. This makes both algorithms difficult to compare using the reported results of their respective implementations; nevertheless, we will be presenting this comparison as an informative result.

Even though multiple results are presented in [9], we focused in the number of instances reported to have been solved to optimality in random graphs with 90 nodes and 2 nodes per partition, with different edge densities (which is also the kind of graphs in which we focused our testing in these last sections).

For each density, we executed our algorithm in five instances of binomial (Erdos-Rényi) graphs and five instances of powerlaw-cluster[4] (Holme-Kim) graphs. Every graph instance was ran until the optimal solution was found, or for up to two hours.

Graph Density	B&C Binomial		B&C Holme-Kim		Frota et al.	
20%	**100%**	(5/5)	**100%**	(5/5)	20%	3/15
40%	0%	(0/5)	**100%**	(5/5)	7%	1/15
60%	0%	(0/5)	60%	(3/5)	**80%**	12/15
80%	0%	(0/5)	0%	(0/5)	**100%**	15/15

Table 6.7.4.1: Fraction of the tested instances that were solved to optimality using the implemented branch and cut algorithm on both binomial and powerlaw cluster graphs of different densities, and fraction solved to optimality as reported by Frota et al. in [9].

The obtained results (presented in table 6.7.4.1) are most interesting. Whereas our algorithm outperforms [9] in low density graphs, the latter wins in high density graphs. It is also worth noting that our algorithm handles clustered graphs much better than binomial ones, most probably because of the high level of symmetry usually found in binomial graphs.

Also, both algorithms have a tight bound on the difference between the lower bound and the solution found in most cases: for algorithm by Frota et al. this difference is never greater than one color, and in our algorithm it reaches its maximum difference of two colors only in a few high-density binomial graphs.

[4]As described in 6.0.2, these graphs are generated by three parameters: node count n, number of nodes m to which each new node is attached, and probability p to add an extra edge generating a triangle when a new node is added. These graphs are constructed with an initial empty graph of size m. In order to attain densities higher than 50% with this kind of graphs, we modified the algorithm to start with a binomial graph of size m, and iteratively expand it to the desired node count n using the original procedure.

These results show that different models and different algorithms can tackle the same problem efficiently in different cases. While our algorithm easily solved low density instances, the branch and cut based on the asymmetric representatives model for the PCP performs clearly better in high density graphs. Instances with medium density are still the most difficult to solve, as also happens in standard coloring problems.

Chapter 7

Conclusions

The graph coloring problem is one of the most vastly studied problems on graphs. In this work, we studied one of its many variants: the partitioned graph coloring problem, or PCP, which is closely related to the min-RWA problem in WDM networks. Our main objective was to solve this problem efficiently using binary integer programming techniques, eventually leading to a branch and cut algorithm.

7.1 Recapitulation

We built our initial model of the problem by generalizing the one proposed by Méndez-Díaz and Zabala in [22] for standard coloring, and experimented with multiple alternatives for expressing the constraints that shape the PCP, as well as with different symmetry breaking constraints adapted from standard coloring. It soon became clear that the symmetry of solutions was one of the most difficult parts of this problem, as happens with standard coloring.

A partial analysis of the polyhedron defined by our proposed model was made, in order to obtain valid inequalities to be used as cuts in the algorithm. We found out that most of the cuts present in [22] could be generalized to partitioned coloring, converting *clique cuts* into *extended clique cuts*, as well as *independent set cuts* into *component independent set cuts.* We also defined the concept of *partition graph* for a partitioned graph, and defined how to migrate certain families of inequalities found in the former into the latter.

Having done an analysis of the polyhedron, we focused in developing a strong heuristic for the problem, which would later be used not only as an initial heuristic to provide a starting integer solution for the algorithm, but also as a primal heuristic and as a component of the branching strategy. The chosen heuristic was actually an exact method, DSATUR, which is an implicit enumeration algorithm, but produces solutions of very high quality very early in its exploration, thus executing it under a strict time bound makes it an excellent heuristic. Since DSATUR is an algorithm for standard coloring, we tested different generalizations of it, eventually choosing the one proposed in [17], and introduced certain optimizations for our particular algorithm.

Armed with several families of valid inequalities and an heuristic algorithm for our problem, we used them as building blocks for our branch and cut algorithm. Starting with a generic B&C scheme provided by the CPLEX engine, we implemented a custom initial heuristic, primal heuristic, and branching strategy, as well as our set of cutting planes along with their separation procedures. A strong preprocessing stage allowed for larger instances to be handled by our algorithm.

We made extensive testing of all of these custom artifacts, evaluating different configurations for each of them under different circumstances, and eventually fine-tuned our algorithm for it to handle different instances of the PCP.

7.2 Results obtained

We obtained several results from the experimentation. First of all, we found out that our heuristic procedure quickly generated solutions very close to the optimum, and the most difficult part in the branch and cut algorithm was to improve the lower bound to prove the optimality of the solution. This made cutting planes play a critical part in our work.

In the analysis we made of the polyhedron, making heavy use of PORTA for identifying the facets of particular instances of our model, we discovered that several families of the standard coloring problem were also found in our models, albeit with slight variations. Since the standard coloring problem can be considered a particular case of the PCP, it is reasonable to deduce that a partial characterization of the latter can be derived from one of the former; this is the method we used to obtain valid inequalities for our problem.

We also found certain restrictions without an equivalent in the standard coloring polytope: this could have been either because they relate to a yet non-discovered facet of standard coloring, or because the addition of partitions to the problem introduces whole new families, pending to be studied.

Regarding performance in comparison with CPLEX's default MIP engine, our algorithm showed that the development of custom artifacts for the PCP did generate a difference in the quality of the obtained solutions.

Also, it was most interesting to find out that, compared to the other integer linear programming model existing in the literature ([9]), the model we used performs much better in graphs with low density, whereas this trend is exactly reversed in high density graphs. This proves that certain models may be more adequate than others for dealing with certain families of graphs.

Consequently, it is critical to identify for which families a particular model (and therefore, a particular algorithm) is more suitable than other, in order to find the correct tool for each particular problem. Discovering which properties of each graph family are being exploited by each model would also provide valuable insight for further optimizing the algorithms being developed.

7.3 Future work

There are several paths for continuing this work. The first of them is to dwelve deeper into the characterization of the PCP polyhedron defined by the propsed model. Even though we presented certain inequalities in this work, there are yet many remaining to provide a complete description. This task has not been even accomplished for the standard coloring problem; nevertheless, there are certain inequalities found for coloring which we have not yet generalized into PCP, such as multicolor inequalities, which could provide more cutting planes to work towards a more effective proof of optimality for the obtained solutions in the branch and cut. Since improving the lower bound was the most difficult task for our algorithm, implementing more families of cutting planes might yield some interesting improvements in terms of obtained gaps.

Further analysis of the heuristic procedures developed is also a pending subject. In this work we evaluated the heuristic methods only with respect to their usefulness in the context of a branch and cut algorithm; however, these procedures have proved to obtain very good solutions by themselves, and an analysis of their performance as stand-alone long-running algorithms would be most interesting. To illustrate this point, for small graphs, the implicit enumeration done by partitioned DSATUR actually proved the optimality of the solution it produced, much faster than its branch and cut counterpart.

Testing the branch and cut algorithm under other scenarios is also another possibility for further development of this work. Throughout our experimentation, we evaluated our algorithm mostly in random graphs with two particular structures (binomial and clustered) with bounded partition sizes. Implementing an *edge-disjoint* heuristic to generate graphs from actual WDM networks would provide the means to test our algorithm under real-world scenarios for the min-RWA problem.

Bibliography

[1] Dimacs challenge instances. `http://mat.gsia.cmu.edu/COLOR02/`.

[2] Networkx version 1.1, April 2010. `http://networkx.lanl.gov/`.

[3] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0–1 programs. Mathematical Programming, 58(1):295–324, 1993.

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

[5] M. Campêlo, V.A. Campos, and R.C. Corrêa. On the asymmetric representatives formulation for the vertex coloring problem. Discrete Applied Mathematics, 156(7):1097–1111, 2008.

[6] M. Campêlo, R. Corrêa, and Y. Frota. Cliques, holes and the vertex coloring polytope. Information Processing Letters, 89(4):159–164, 2004.

[7] P. Erdos and A. Renyi. On Random Graphs. Publicationes Mathematicae 6, pages 290–297, 1959.

[8] R.M.V. Figueiredo, V.C. Barbosa, N. Maculan, and C.C. Souza. Acyclic orientations with path constraints. Arxiv preprint cs/0510064, 2005.

[9] Y. Frota, N. Maculan, T.F. Noronha, and C.C. Ribeiro. A branch-and-cut algorithm for partition coloring. Networks, 55(3):194–204, 2010.

[10] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society, 64(5):275–278, 1958.

[11] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. Combinatorica, 1(2):169–197, 1981.

[12] P. Holme and B.J. Kim. Growing scale-free networks with tunable clustering. Physical Review E, 65(2):26107, 2002.

[13] E. Hyytia and J. Virtamo. Wavelength assignment and routing in WDM networks. In Proc. of the 14th Nordic Teletraffic Seminar (NTS-14). Citeseer, 1998.

[14] R.M. Karp. Reducibility among combinatorial problems. Complexity of computer computations, pages 85–104, 1972.

[15] J.M. Kleinberg. Approximation algorithms for disjoint paths problems. PhD thesis, Citeseer, 1996.

[16] K. Lee, K.C. Kang, T. Lee, and S. Park. An optimization approach to routing and wavelength assignment in WDM all-optical mesh networks without wavelength conversion. ETRI journal, 24(2):131–141, 2002.

[17] Guangzhi Li and Rahul Simha. The partition coloring problem and its application to wavelength routing and assignment. In In Proceedings of the First Workshop on Optical Networks, 2000.

[18] P. Manohar, D. Manjunath, and RK Shevgaonkar. Routing and wavelength assignment in optical networks from edge disjoint path algorithms. IEEE Communications Letters, 6(5):211, 2002.

[19] D.W. Matula, G. Marble, and J.D. Isaacson. Graph coloring algorithms. Graph theory and computing, pages 109–122, 1972.

[20] A. Mehrotra and M.A. Trick. A column generation approach for graph coloring. INFORMS Journal on Computing, 8(4):344, 1996.

[21] I. Méndez-Díaz and P. Zabala. A branch-and-cut algorithm for graph coloring. Discrete Applied Mathematics, 154(5):826–847, 2006.

[22] I. Méndez-Díaz and P. Zabala. A cutting plane algorithm for graph coloring. Discrete Applied Mathematics, 156(2):159–179, 2008.

[23] S.D. Nikolopoulos and L. Palios. Hole and antihole detection in graphs. In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, page 859. Society for Industrial and Applied Mathematics, 2004.

[24] T.F. Noronha, M.G.C. Resende, and C.C. Ribeiro. A random-keys genetic algorithm for routing and wavelength assignment. In Seventh Metaheuristics International Conference, Montréal, Canadá. Citeseer, 2007.

[25] T.F. Noronha and C.C. Ribeiro. Routing and wavelength assignment by partition colouring. European Journal of Operational Research, 171(3):797–810, 2006.

[26] E. Sewell. An improved algorithm for exact graph coloring. Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993, page 359, 1996.

[27] N. Skorin-Kapov. Routing and wavelength assignment in optical networks using bin packing based algorithms. European Journal of Operational Research, 177(2):1167–1179, 2007.

[28] Michael Trick. DSATUR C implementation, November 1994. `http://mat.gsia.cmu.edu/COLOR/solvers/trick.c`.

[29] IBM ILOG CPLEX V12.1. User's manual for cplex, 2009.

[30] D.J.A. Welsh and MB Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. The Computer Journal, 10(1):85, 1967.