ANDRÉ DUQUE MADEIRA

Coloração de Grafos: teoria e aplicações à síntese VLSI

Relatório final da disciplina de Trabalho de Conclusão II do curso de Bacharelado em Informática

Orientador: Prof. Dr. Ney L. V. Calazans

Porto Alegre
1998/II

Coloração de Grafos: teoria e aplicações à síntese VLSI

André Duque Madeira

Problems worthy
of attack
prove their worth
by hitting back

Piet Hein

Sumário

Lista de Figuras

Agradecimentos

Em primeiro lugar agradeço a todos meus professores desde o colégio até a faculdade, de todas as áreas e épocas, que souberam expressar seu conhecimento e pensamento de maneira construtiva e produtiva, e que me fizeram enxergar e compreender alguma sabedoria que eu não poderia ter conseguido sozinho. Sou agradecido também à universidade e órgãos de formento à pesquisa que proporcionaram nos últimos anos condições necessárias para que eu pudesse me desenvolver intelectual e socialmente.

Agradeço com minha mais sincera humildade e admiração ao professor Ney Calazans que soube, ao longo dos anos em que foi meu orientador e amigo, conscientizar-me e projetar-me em um mundo de descobertas e facínios constantes que somente a incansável busca pelo conhecimento pode nos oferecer. Seus pensamentos, ensinamentos e contribuições à minha formação profissional e pessoal tornaram-se cada vez mais inmensuráveis à medida que eu conseguisse gradativamente entender seus inestimáveis valores agregados. Meus singelos muito obrigado e votos de muito sucesso profissional na vida.

Agradeço igualmente aos professores Fernando Moraes e João Batista Oliveira pelo apoio fornecido não somente durante o trabalho de conclusão mas em muitas outras ocasiões como professores e amigos. É com grande satisfação que posso me orgulhar em poder ter dialogado e vivido momentos de grande experiência intelectual com essas pessoas tão brilhantes que serviram e servem de exemplo em minha vida.

Agradeço enormemente a todos meus amigos que puderam compreender o valor de uma amizade sólida, duradoura e sincera. Acima de tudo, foram amigos a todas as horas e estiveram sempre dispostas a me ajudar nos momentos em que eu mais

necessitava. Obrigado e espero poder contar com vocês durante mais um bom tempo.

O papel de apoio dos pais nessas horas de crescimento pessoal e profissional é indiscutível e por essa razão, gratifica-me muito poder dizer que posso ter sido privelegiado em ter os pais que tenho, os quais sempre me incentivaram a crescer estudando e me conduziram para caminhos corretos advertindo-me dos perigos e conseqüências que eu iria enfrentar. Agradeço muit'ıssimo a meu Pai e minha Mãe, assim como a minha Irmã, por simplesmente serem o que são e por terem me ajudado bastante no decorrer de meu curso, moral, financeira e fraternamente falando.

Não poderia me esquecer de parabenizar minha namorada pelo sucesso que obteve em seu curso e pelo apoio incondicional que me proporcionou durante o periodo da escrita deste relatório. Agradeço-a, também, profundamente pela sua compreensão nos momentos difíceis vividos e das diversas diversões que deixamos de aproveitar. Com meu mais glorioso carinho, JTA, Eu.

CAPÍTULO 1

Introdução

O presente trabalho é um estudo sobre o problema de coloração de grafos (CG) e suas aplicações à síntese de Sistemas Digitais (SDs) de Muito Alta Escala de Integração (do inglês Very Large Scale Integration ou VLSI). Diversas etapas compuseram o trabalho:

- a pesquisa de algoritmos de coloração de grafos existentes na literatura;
- o estudo de algoritmos em síntese VLSI que utilizam parcial ou totalmente coloração de grafos em sua solução;
- construção de uma biblioteca de estruturas básicas para a manipulação de grafos (visando coloração) e de algoritmos de coloração de grafos que seja de fácil uso ao programador, em particular aquele desejando mapear problemas concretos para uma instância de CG;
- mapeamento de um algoritmo em VLSI para o problema de coloração de grafos como estudo de caso para o presente trabalho.

A seguir, ainda neste Capítulo, apresenta-se o escopo de desenvolvimento deste trabalho, um pequeno histórico sobre a Teoria de Grafos e a motivação para o presente trabalho. Algumas definições de grafos e coloração de grafos importantes e

necessárias a um bom acompanhamento na leitura do relatório são apresentadas no Capítulo 2, Definições Preliminares. O Capítulo 3 discute o problema de coloração em si, sua complexidade, suas classificações e apresenta a implementação de alguns algoritmos existentes enquanto que o Capítulo 4 apresenta a exploração realizada na área de em síntese VLSI para a realização deste trabalho através de estudos de casos selecionados. A implementação da biblioteca e de outros algoritmos desenvolvidos encontra-se no Capítulo 5. Este Capítulo detalha toda a implementação realizada, desde sua construção e funcionamento interno, passando por seu manuseio, até a comparação de resultados com outros algoritmos do estado da arte. O Apêndice, contendo os objetivos e o cronograma de atividades descritos na proposta do presente trabalho, e as Referências Bibliográficas encerram o relatório.

1.1 Escopo do Trabalho

O projeto auxiliado por computador ou CAD (computer-aided design) de sistemas digitais é um tópico de pesquisa e desenvolvimento que vem assumindo crescente importância nas últimas três décadas, desenvolvendo-se enormemente o interesse pela área. Diversas técnicas em CAD possibilitaram o projeto eficiente de circuitos de alto desempenho e de grande porte para um amplo campo de aplicações, desde o processamento de informação (ex: computadores) até telecomunicações, controle de manufaturas, transportes etc.

Neste trabalho, endereça-se, sobretudo, um dos importantes tópicos em CAD: a síntese e otimização de projeto de sistemas digitais VLSI (Very Large Scale Integration), enfatizando os níveis lógico e arquitetural de abstração. A maioria dos problemas nesta área, tais como minimização de relações Booleanas e a minimização de estados, está baseada na teoria de grafos e na álgebra Booleana [21] e pode ser reduzida a problemas fundamentais como *coloração* , *cobertura* e *satisfactibilidade* de grafos conforme ilustrado na Figura 1.1 retirada de [21]. Uma das propostas realizadas foi o desenvolvimento de algoritmos eficazes que implementam soluções do problema de ***coloração de vértices de um grafo*** (a definição do problema pode

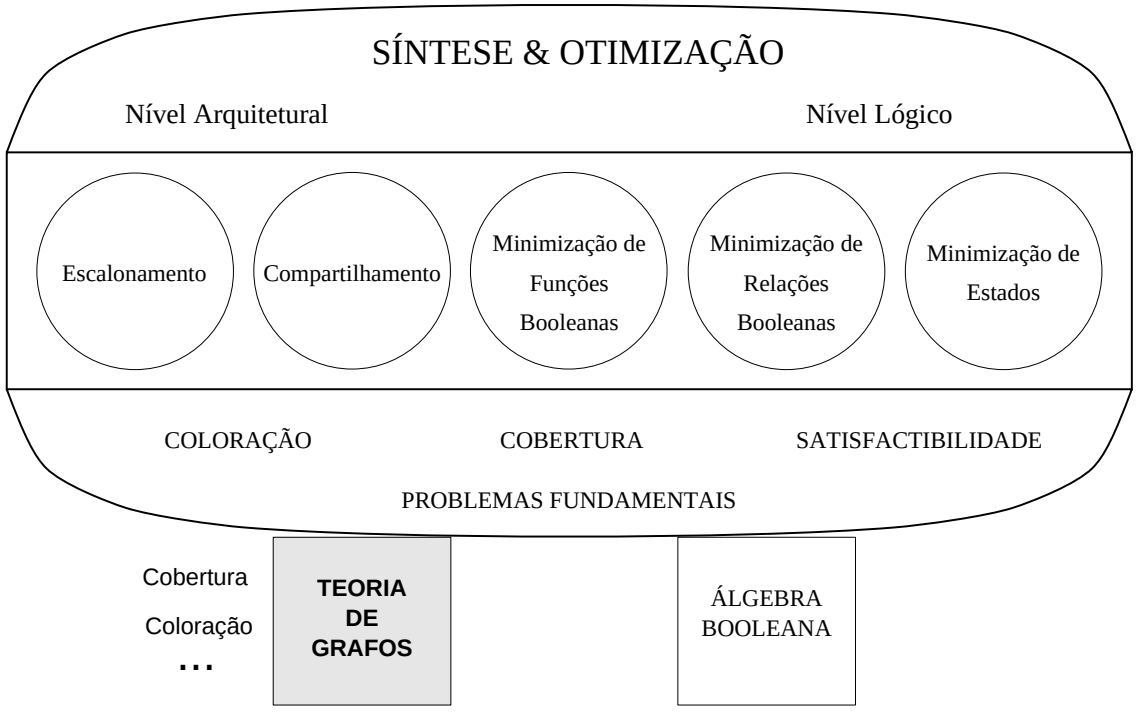
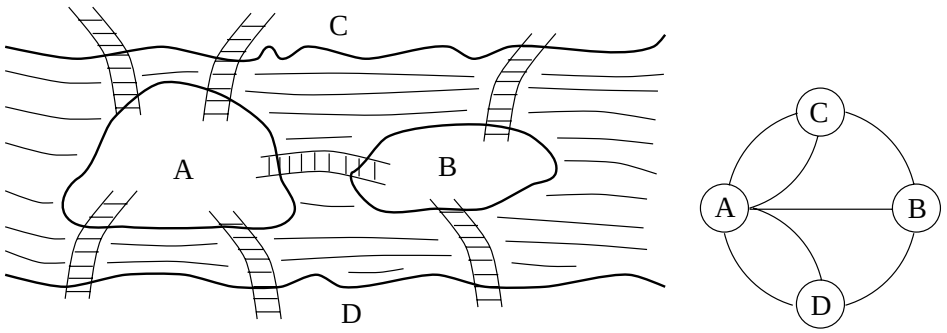


Figura 1.1: Problemas de Síntese e Otimização em VLSI têm base teórica de resolução na Teoria de Grafos e Álgebra Booleana.

ser encontrada na página 23). Outro problema em coloração de grafos é o problema de ***coloração de arestas de um grafo***, menos conhecido, cujas descrição e aplicabilidade podem ser encontradas em [15, 24, 31, 53]. No resto deste relatório, quando não especificado, estipular-se-á ***coloração de grafos*** como significando ***coloração de vértices de um grafo*** de acordo com a convenção corrente em Computação.

1.2 Histórico da Teoria de Grafos

A solução do conhecido *problema da ponte de Königsberg* , resolvido por Euler em 1736 *apud* [47], é considerada o primeiro teorema em teoria de grafos. No rio Pregel, junto à cidade de Königsberg (hoje Kaliningrado, Rússia), existem duas ilhas formando quatro regiões distinguíveis de terra (A, B, C e D) e um total de sete pontes interligando-as, conforme a disposição ilustrada na Figura 1.2(a). O problema consiste em determinar um trajeto pelas pontes segundo o qual se possa



(a) (b)

Figura 1.2: O problema da ponte de Königsberg.

retornar à região de partida, após atravessar cada ponte exatamente uma vez. Euler, generalizando o problema, utilizou um modelo de grafos, ilustrado na Figura 1.2(b), para mostrar que não existe tal trajeto. Provou também que o desejado trajeto só existe quando e somente quando este contiver um número par de pontes.

Poucos trabalhos foram realizados nos anos subseqüentes ao trabalho de Euler. Algumas contribuições para o despertar do interesse pela área surgiram somente em meados do século XIX, quase 200 anos depois. Uma dessas contribuições é a formulação do *problema das quatro cores* supostamente feita por Francis Guthrie em 1852[1] que por sua vez comunicou a De Morgan, primeiro a escrever uma contribuição técnica ao assunto.

No problema, é necessário colorir todas as regiões contíguas de um mapa plano, cada região recebendo uma cor, de tal forma que regiões fronteiriças possuam cores diferentes. O problema então consiste em provar que não são jamais necessárias mais do que quatro cores para colorir qualquer mapa plano ou ***grafos planares*** (a definição de grafo planar pode ser encontrada na página 22). O mapa da Figura 1.3, por exemplo, pode e deve ser colorido com quatro cores neste contexto. Este problema é um caso especial do problema geral de coloração de grafos e é também o pioneiro nesta área. O atual teorema das quatro cores permaneceu uma

[1] uma breve história sobre o problema encontra-se na url
http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/The_four_colour_theorem.html

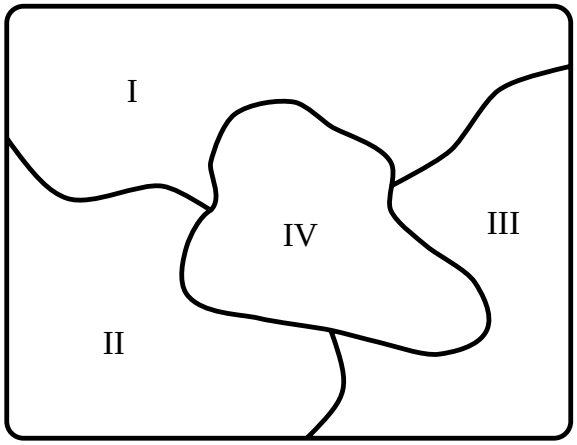


Figura 1.3: Mapa onde 4 cores são necessárias para obter uma coloração.

conjetura até Appel & Haken o provarem[2] em 1976 [1, 2, 3]. Uma prova menos complicada e mais completa do teorema pode ser encontrada em [43] ou na URL http://www.math.gatech.edu/∼thomas/FC/fourcolor.html.

Além da importância do tópico de coloração, o problema das 4 cores desempenhou um papel muito relevante para o desenvolvimento geral da teoria de grafos, pois serviu de motivação para o trabalho na área e ensejou o desenvolvimento de outros aspectos teóricos, na tentativa de resolver a questão.

O interesse pela pesquisa em teoria de grafos aumentou por volta da década de 1930, onde resultados fundamentais na teoria foram obtidos por Kuratowski[38], König[35] e Menger[41]. Os anos mais recentes confirmam, de certa forma, a idéia de ser a teoria de grafos ainda uma área com vastas regiões inexploradas.

Grafos são representações abstratas de um conjunto de elementos, chamados de vértices, e de um outro conjunto que relaciona os elementos do primeiro de acordo com certas regras, e cujos elementos são chamados de arestas. Grafos são portanto abstrações de dados amplamente utilizadas em diversos campos onde relações binárias e adjacências entre elementos de um conjunto devem ser modeladas.

Uma das grandes vantagens do uso da teoria de grafos em relação aos outros

[2]a prova inclui uma verificação computacional de mais de 1400 casos, utilizando cerca de 1200 horas de computação em um IBM 370.

campos da matemática discreta como a álgebra Booleana e relacional é o fato da primeira possuír uma *representação gráfica*, que auxilia enormemente a apreensão e a solução de diversos problemas. Por outro lado, um problema típico de representação gráfica é o problema de ***isomorfismo***[3]: dadas duas representações gráficas, correspondem elas a um mesmo grafo? Os grafos G_1 e G_2, por exemplo, são grafos ***isomorfos*** conforme ilustrado na Figura 1.4. Esse problema pode ser visto como

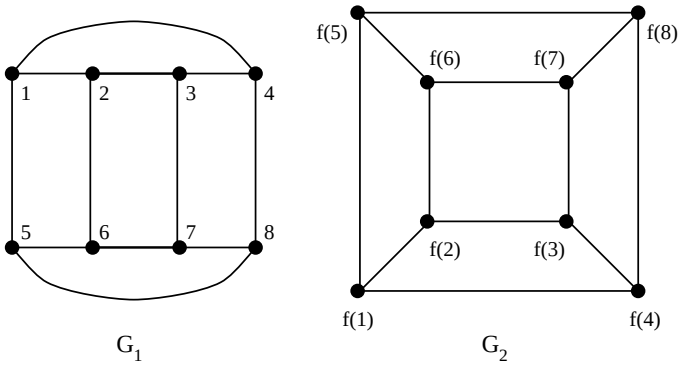


Figura 1.4: Grafos isomorfos entre si.

o de verificar se é possível associar cada vértice do primeiro grafo a exatamente 1 vértice do segundo, de modo a preservar a relação de adjacência entre os elementos. Isto pode ser obtido no caso dos grafos da Figura definindo a bijeção f entre os vértices de G_1 e G_2.

1.3 Motivação

Grafos são amplamente utilizados nas mais diversas áreas da Informática, da Engenharia em geral, da Física, da Economia e da Matemática. Soluções de diversos problemas que se apóiam em caminhamento em grafos, coloração de grafos, verificação de planaridade, entre outros, necessitam de algoritmos eficientes em termos de tempo computacional e qualidade de resultados.

[3]a definição de isomorfismo pode ser encontrada na Seção de Definições Preliminares, na página 21.

A teoria de coloração de grafos, por sua vez, tem uma posição central em Matemática Discreta. Coloração de grafos lida com o problema fundamental de particionar um conjunto de objetos em classes (classificar elementos do conjunto) de acordo com certas regras. Muitos problemas específicos nas diversas áreas em computação, como Arquiteturas de Computadores, Computação Gráfica e CAD de Sistemas Digitais, têm como solução particionamentos ou classificações de elementos de um conjunto, e utilizam algoritmos específicos de coloração em suas resoluções. Pode-se citar alguns problemas já modelados desta forma:

- codificação Booleana de entradas e estados em Máquinas de Estados Finitas (FSMs) para implementação em hardware[13, 18];
- decomposição de PLAs [14];
- problema de roteamento para redes Clos [32];
- problemas em síntese VLSI de alto nível, como escalonamento e alocação de recursos em blocos operacionais de processadores [46];
- a alocação de recursos em um sistema operacional [25];
- escalonamento e seqüenciamento de processos [40, 44];

O fato do problema de coloração de grafos ser um problema abstrato mas mapeável para inúmeros problemas reais contribui fortemente para motivar o presente trabalho.

O problema de colorir (isto é, obter uma coloração de) um grafo com o número mínimo de cores[4] é bem conhecido por ser NP-hard[5], mesmo quando restrito a grafos k-coloríveis[6] para k constante e $k \geq 3$. Garey & Johnson em [26] mostram que o problema de obter o número cromático de um grafo é intratável, i.e., não se conhece um algoritmo que resolva o problema em tempo polinomial.

[4] esse número mínimo de cores é chamado de *número cromático* de um grafo.

[5] um bom resumo sobre NP-completude e sobre complexidade de algoritmos em geral pode ser encontrado em [16].

[6] grafos passíveis de coloração com k cores.

Neste trabalho, o desenvolvimento de boas heurísticas para resolver o problema de coloração de grafos, visando a solução de problemas em síntese VLSI, serve como forte motivação. Recentemente, Coudert[18] fez pequenas modificações ao algoritmo de coloração de grafos *DSATUR* [6] para resolver o problema de codificação restrita baseado em dicotomias[7], conseguindo bons resultados. Coudert precisou, contudo, reimplementar o código do algoritmo de coloração e houve também a necessidade de entender a fundo os conceitos de coloração. Este é mais um bom motivo para propor a construção de uma biblioteca de estruturas e algoritmos visando coloração que possa facilmente ser adequada às necessidades de diversos outros problemas que necessitem da coloração em sua resolução.

A procura por heurísticas eficazes a aplicar durante a coloração de grafos é constante e será discutida brevemente no Capítulo 3.

Outro fator importante para o bom desenvolvimento deste trabalho é o fato do autor desta proposta ser bolsista de iniciação científica há quase três anos no Grupo de Apoio ao Projeto de Hardware (GAPH) da Faculdade de Informática da PUCRS. Nesses anos, o autor trabalhou com síntese lógica na mesma linha de trabalho do líder do grupo, o prof. Ney Calazans. Adquiriu-se experiência em Microeletrônica em geral e em algoritmos de síntese para o projeto de sistemas digitais. Percebeu-se, neste período, que existiam diversos problemas que poderiam ser mapeados para o problema de coloração de grafos, inclusive o problema de satisfação de restrições baseadas em pseudo-dicotomias desenvolvido e implementado como parte da tese de doutorado do prof. Calazans [8]. A idéia inicial foi utilizar algoritmos de coloração de grafos para satisfazer essas restrições e integrá-los ao ambiente ASS†UCE[10] (ver Seção 4.2.3).

[7] a definição pode ser encontrada na página 41

CAPÍTULO 2

Definições Preliminares

O intuito desta Seção é apresentar algumas definições básicas em teoria de grafos e coloração de grafos. As definições de grafo e do problema de coloração de grafos, necessárias à leitura do resto do trabalho, são apresentadas.

Utiliza-se a notação corrente em Computação [16] para a representação de grafos, vértices (nodos), arestas e definições associadas. As definições de grafos planares, completos, caminhos em grafos entre outros foram em sua maioria retirados de [47] enquanto que as definições de coloração de grafos foram extraídas de vários autores como [16, 26, 47].

2.1 Grafos

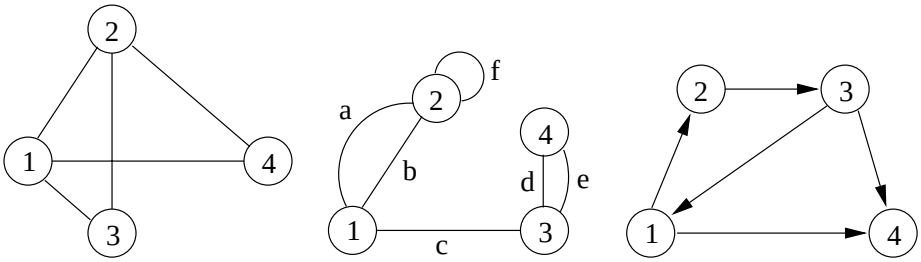
Definição 2.1 (grafo) *Um* **grafo** $G(V, E)$ *é um conjunto finito não-vazio* V*, ou* $V(G)$*, e um conjunto* E*, ou* $E(G)$*, de pares não ordenados de elementos distintos de* V*. Os elementos de* V *são denominados os* **vértices** *e os de* E *as* **arestas** *de* G*, respectivamente. Utilizar-se-á a notação* $n = |V|$ *e* $m = |E|$ *para designar a cardinalidade dos conjuntos.*

Cada aresta $e \in E$ *será denotada pelo par de vértices* $e = \{v, w\}$ *que a determina. Nesse caso, os vértices* v *e* w *são os* **extremos** *(ou* **extremidades***) da aresta* e*, sendo denominados* **adjacentes***. A aresta* e *é dita* **incidente** *a ambos* v*,* w*. Duas arestas*

que possuem um extremo comum são chamadas de **adjacentes**.

Seja o elemento $t \in V(G) \cup E(G)$, $G - t$ *é um pequeno abuso notacional utilizado para designar, no caso de* t *ser um vértice, o grafo* G *do qual se retira o vértice* t *e todas as arestas incidentes a* t *e no caso de* t *ser uma aresta, o grafo* G *do qual se retira a aresta* t.

Um **multigrafo** *é um grafo no qual é permitido ter duas ou mais arestas associadas a um mesmo par de vértices. Um* **digrafo** *é um grafo onde as arestas são* dirigidas, *ou seja, cada aresta corresponde a um par ordenado de vértices. Na Figura 2.1 ilustra-se um grafo, um multigrafo e um digrafo em* (a),(b) *e* (c) *respectivamente.*



(a) grafo (b) multigrafo (c) digrafo

Figura 2.1: Exemplos de grafo, multigrafo e digrafo.

Definição 2.2 (grau de um vértice) *Define-se* ***grau de um vértice*** $v \in V$, *denotado por* $grau(v)$, *como sendo o número de vértices adjacentes a* v.

Cada vértice v *é incidente a* $grau(v)$ *arestas e cada aresta é incidente a 2 vértices. Logo, para qualquer grafo,* $\sum_{v \in V} grau(v) = 2|E|$. *Um vértice que possui grau* 0 *é chamado* ***isolado***. *Um grafo é* regular *de* grau r, *quando todos os seus vértices possuírem o mesmo grau* r.

Definição 2.3 (grafo completo) *Um grafo é* ***completo*** *quando existe uma aresta entre cada par de seus vértices. Utiliza-se a notação* K_n, *para designar um grafo completo com* n *vértices. O grafo* K_n *possui, portanto, o número máximo possível de arestas para um dado* n, *ou seja* $\binom{n}{2}$ *arestas. Exemplos de grafos completos estão ilustrados na Figura 2.2.*

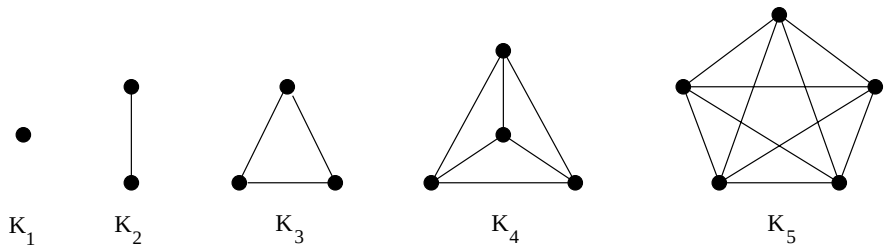


Figura 2.2: Grafos Completos.

Definição 2.4 (grafo bipartido) *Um grafo $G(V, E)$ é* ***bipartido*** *quando o seu conjunto de vértices V puder ser particionado em dois subconjuntos V_1, V_2, tal que toda aresta de G une um vértice de V_1 a outro de V_2.*

Um grafo ***bipartido completo*** *possui uma aresta para cada par de vértices v_1, v_2, com $v_1 \in V_1$ e $v_2 \in V_2$. Sendo $n_1 = |V_1|$ e $n_2 = |V_2|$, um grafo bipartido completo é denotado por K_{n_1,n_2} e obviamente possui $n_1 n_2$ arestas.*

Definição 2.5 (caminho) *Uma seqüência de vértices $v_1, \ldots, v_k$ tal que $\{v_i, v_{i+1}\} \in E$, com $1 \leq i < k-1$, é denominado* ***caminho*** *de v_1 a v_k. Diz-se então que v_1* ***alcança*** *ou* ***atinge*** *v_k. Se entre v_1 e v_k existir apenas uma aresta, diz-se que o caminho é* ***direto*** *caso contrário ele é* ***indireto****. Se todos os vértices do caminho $v_1, \ldots, v_k$ forem distintos, a seqüência recebe o nome de* ***caminho simples*** *ou* ***elementar****. Se as arestas forem distintas, a seqüência denomina-se* ***trajeto****.*

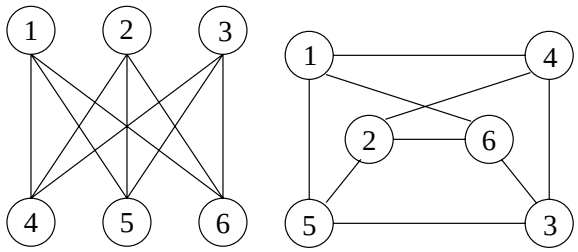
Definição 2.6 (grafo conexo) *Dois vértices estão* ***conectados*** *se existe um caminho direto ou indireto entre os dois. Um grafo é* ***conexo*** *se para qualquer par de vértices do grafo, estes estão conectados, caso contrário o grafo é dito* ***desconexo****.*

Definição 2.7 (subgrafo) *Um grafo $H = (V_H, E_H)$ é dito um* ***subgrafo*** *de um grafo $G = (V, E)$ se $V_H \subseteq V$ e $E_H \subset E$. Os* ***subgrafos conexos maximais****, também chamado de* ***subgrafos completos****, são subgrafos conexos de um grafo que não estão estritamente contidos em outros subgrafos conexos do mesmo grafo.*

Definição 2.8 (isomorfismo) *Dois grafos $G = (V, E)$ e $H = (V_H, E_H)$ são* ***isomorfos*** *se existe uma bijeção entre os vértices de V e V_H que preserva adjacência,*

isto é, se $v, w \in V$ são adjacentes em G, então os vértices correspondentes em V_H pela bijeção também são adjacentes em H e vice-versa. Esta relação induz também um relação 1-1 entre os conjuntos de arestas dos grafos respectivos.

Definição 2.9 (planaridade) *Um grafo é* ***planar*** *se é possível desenhá-lo no plano de modo que as linhas correspondentes às arestas não se cruzem. Tal desenho é*



(a) *(b)*

Figura 2.3: Grafos $K_{3,3}$.

uma realização gráfica planar *do grafo, ou simplesmente,* realização planar.

O grafo da Figura 1.4 da página 16 é planar pois satisfaz a condição enquanto que os grafos isomorfos (a) e (b) da Figura 2.3 não são.

Definição 2.10 (clique e conjunto independente de vértices) *Denomina-se* ***clique*** *de um grafo G um subgrafo de G que seja completo, i.e., um conjunto de vértices mutuamente adjacentes. Chama-se* ***conjunto independente de vértices*** *um subgrafo induzido de G, que seja totalmente desconexo. Um* ***clique máximo*** *é um clique cujo número de vértices é no mínimo maior que qualquer outro clique em G. O* ***tamanho*** *de um clique ou conjunto independente de vértices é igual à cardinalidade de seu conjunto de vértices.*

Num clique, portanto, cada par de vértices distintos contém uma aresta interligando-as, enquanto que num conjunto independente de vértices não existe aresta entre qualquer par de vértices.

O problema de achar um clique ou conjunto independente de vértices com um dado tamanho k é NP-hard [26]. O problema de achar um *clique máximo* e o de obter

uma *coloração mínima* estão intimamente relacionados. Note-se que o tamanho de um clique máximo é um *limite inferior* no número mínimo de cores necessárias para colorir um grafo.

2.2 Coloração de Grafos

Definição 2.11 (coloração) *Seja $G(V, E)$ um grafo e $K = \{1, 2, \ldots, k\}$ um conjunto de* ***cores****. Uma* ***coloração correta*** *de G é uma atribuição de alguma cor de K para cada vértice de V, de tal modo que nenhuma aresta contenha suas extremidades coloridas com a mesma cor. Uma coloração correta de G é, portanto, uma função $\varphi : V \longrightarrow K$ tal que para cada par de vértices $v, w \in V$ tem-se $\{v, w\} \in E \Longrightarrow \varphi(v) \neq \varphi(w)$.*

Uma ***coloração imprópria***[1] *de G ocorre quando existe pelo menos uma aresta $\{v, w\} \in E$ onde $\varphi(v) = \varphi(w)$. Doravante, neste trabalho e em Computação em geral, coloração sem qualificador é assumido como coloração* correta.

O ***grau de saturação de um vértice*** *$v \in V$, denotado por $grau\_sat(v)$, é o número de vértices previamente coloridos e adjacentes a v ($|adj(v)|$).*

O grau de saturação de um vértice é utilizado em alguns algoritmos para a resolução do problema de coloração. Caracteriza-se por ser uma definição algorítmica e transitória, pois seu valor depende de uma coloração prévia de vértices adjacentes.

Definição 2.12 (grafo k-colorível) *Um grafo G é dito* ***k-colorível*** *quando é possível obter uma coloração correta de G com no máximo k cores.*

Problema de Decisão 2.1 (coloração de grafos [34])
INSTÂNCIA: um grafo $G(V, E)$ e um inteiro positivo $1 \leq k \leq |V|$
DECISÃO: É G k-colorível? Isto é, existe ou não uma função $\varphi : V \longrightarrow \{1, 2, \ldots, k\}$ tal que $\forall \{v, w\} \in E, \varphi(v) \neq \varphi(w)$?

[1] *alguns algoritmos de coloração (ver Seção 3.2) caracterizam-se por produzir, transitoriamente ou não, colorações impróprias.*

O problema de decisão acima é solúvel em tempo polinomial para $k = 2$, mas permanece NP-hard para $k \geq 3$.

Definição 2.13 (número cromático) *O* ***número cromático*** *de um grafo G, denotado por $\chi(G)$, é o número mínimo de cores necessárias para obter uma k-coloração de G. Uma* **coloração mínima** *de G é aquela onde $|K| = \chi(G)$.*

Um grafo k-cromático é aquele em que o número cromático do grafo é exatamente *k, ao contrário de k-colorível onde o número cromático do grafo é no* máximo *k.*

Em particular, se um grafo é 2-colorível, isto significa que seu conjunto de vértices se particiona em dois subconjuntos disjuntos (cada um com uma cor), tal que cada aresta reune dois vértices dos subconjuntos distintos. Um grafo, portanto, é 2-colorível *se e somente se* ele for *bipartido* [47]. Grafos k-coloríveis, por extensão deste conceito, podem ser chamados de grafos k-partidos.

Definição 2.14 (elemento crítico e grafo crítico) *Um elemento $t \in V(G) \cup E(G)$ é* crítico *se $\chi(G-t) < \chi(G)$. G é crítico, ou mais precisamente, $\chi(G)$-crítico, se* todas *as arestas e vértices de G são críticos.*

Existem ainda muitas definições, conjeturas e teoremas sobre coloração de grafos. Coloração de arestas de um grafo, teoremas de número cromático baseado nos graus dos vértices, coloração de grafos perfeitos e outros tipos de grafos são explorados em diversos problemas citados em [33]. Apresentou-se aqui somente as definições suficientes e necessárias à compreensão deste trabalho.

CAPÍTULO 3

Coloração de Grafos

Diversos problemas de interesse prático podem ser modelados como uma instância do problema de coloração de grafos. Problemas que envolvam o particionamento de um conjunto de objetos em classes de acordo com certas regras são freqüentes e fundamentais em Matemática e Informática. A forma geral desses problemas envolve a construção de um grafo como modelo, sendo, em geral, os vértices os objetos de interesse do problema e as arestas a relação entre pares de objetos do conjunto a ser particionado. O problema de coloração resume-se, portanto, em achar classes de objetos particionados identificando-as através de uma cor. O problema de coloração mínima, por sua vez, acha o menor conjunto de classes que obedeça às restrições impostas pelas arestas.

Há mais de 100 anos, desde a formulação do *problema das quatro cores* , resultados interessantes de problemas em coloração de grafos vêm sendo obtidos, proporcionando uma maior flexibilidade na resolução de problemas práticos. Existe, também, uma grande quantidade de problemas sendo formulados, conjeturas tentando ser provadas e teoremas sendo reescritos. Muitos dos problemas insolúveis até agora foram descritos e reunidos em um livro redigido por Jensen & Toft [33]. Este livro é uma boa referência na área de coloração, pois contém informações abundantes e previamente publicadas em artigos especiais, anais de conferências e relatórios

técnicos. Cada um dos mais de 200 problemas são citados em um formato acessível, seguido de comentários de sua história, resultados e literatura, posicionando o leitor adequadamente no estado da arte da solução de cada problema. Para uma revisão mais detalhada sobre coloração de grafos em geral, assim como do estado da arte até a data, pode-se consultar [36].

Apresenta-se nesse Capítulo, a descrição dos algoritmos de coloração mais conhecidos, discutindo suas complexidades (Seção 3.1), classificações (Seção 3.2) e implementações (Seção 3.3).

3.1 Complexidade do Problema

O problema de colorir um grafo possui sempre uma solução trivial muito simples que consiste em utilizar um total de $|V|$ cores, uma para cada vértice, garantindo obviamente cores diferentes a vértices adjacentes. Todavia, encontrar uma coloração *mínima* ou *quase-mínima* é bastante dificil e computacionalmente custoso. Neste aspecto, o problema é muito similar ao problema de codificação Booleana de um conjunto de símbolos [8], onde soluções corretas estão trivialmente disponíveis mas soluções *corretas ótimas* são extremamente difíceis de encontrar!

Algoritmos que necessariamente obtêm uma solução ótima à instância do problema a ser resolvido são chamados de ***exatos***. No caso de coloração de grafos, um algoritmo exato seria aquele que, para qualquer grafo G é capaz de obter (sempre) o número cromático $\chi(G)$. Existe um algoritmo bem simples, de fácil implementação, mas altamente custoso em tempo de CPU (ver Seção 5).

A solução exata do problema para qualquer grafo dado é intratável [26], situando-se na classe NP-hard de problemas. Isso significa que mesmo a verificação do problema tem complexidade exponencial, ou seja, dado um número k a verificação consiste em analisar se existe uma coloração válida com k cores. O *estado da arte* em algoritmos exatos para coloração de grafos está em geral baseado em Enumeração Implícita citando por exemplo as implementações propostas por Brélaz [6] e Kubale [37].

Devido à inviabilidade de aplicar algoritmos exatos à resolução de problemas

encontrados na prática (devido ao tamanho do grafo a colorir), utiliza-se algoritmos ***aproximados***, baseados na aplicação de heurísticas à solução do problema, com o intuito de acelerar o processamento. Em coloração, algoritmos aproximados são aqueles em que a resposta (número de cores necessárias à coloração) se aproxima do número cromático do grafo. Mede-se a qualidade do algoritmo pela razão:

$$\frac{número\ de\ cores\ \ utilizadas}{número\ \ cromático}$$

e pela sua complexidade em termos de tempo de CPU, e/ou memória ocupada.

A maioria dos algoritmos desenvolvidos até hoje são algoritmos aproximados como costuma ocorrer para problemas da classe NP-hard. Classificam-se a seguir os algoritmos aproximados de acordo com sua técnica de resolução ao problema.

3.2 Classificação de Algoritmos de Coloração

A técnica mais utilizada por algoritmos aproximados de coloração é a do aumento sucessivo, ou seja, uma coloração parcial é encontrada num pequeno número de vértices e estende-se esta vértice a vértice, até o grafo inteiro ser colorido. Exemplos de variações dessa abordagem incluem as propostas de Welsh and Powell [51], Grimmet and McDiarmid [28], Brélaz [6], Wigderson [52], Berger and Rompel [4] e Halldórsson [29].

O algoritmo que proporciona a melhor razão entre o número de cores utilizadas pelo número cromático, no momento da escrita deste relatório, deve-se a Halldórson [29] que garantiu uma razão não maior que

$$O\left(\frac{n(\log\log n)^2}{(\log n)^3}\right)$$

onde n é o número de vértices e log é na base 2. Halldórson superou algoritmos anteriores de Wigderson ($O(n^{1-(1/(k-1))})$) e Berger and Rompel ($O((n/\log n)^{1-(1/(k-1))})$) para grafos k-coloríveis.

Muitos desses algoritmos de coloração de grafos genéricos, classificam-se, segundo

Culberson[19], em uma das seguintes classes:

Algoritmos Gulosos (Greedy). Procuram grandes conjuntos independentes de vértices e colorem cada um com uma cor. A implementação mais simples desta classe é o Algoritmo apresentado na página 29 a seguir. Intuitivamente, parece razoável admitir o fato de que quanto maior os conjuntos independentes, menor será o número de cores utilizadas. Entretanto, sendo guloso nos primeiros passos, evita-se muitas vezes uma boa escolha posterior, segundo Cormen[16]. Essas abordagens, portanto, produzem colorações aproximadas. Aplicações interessantes utilizando Algoritmos Gulosos podem ser encontradas em [5, 51].

Algoritmos de Partição. Particionam os vértices de acordo com um critério especificado e remove conflitos, movendo vértices de uma partição a outra. Esses métodos produzem uma *coloração imprópria* ou *aproximação de coloração correta*, pois no fim alguns conflitos ainda restam. O algoritmo *TABU search* [30] é um exemplo desta classe.

Algoritmos baseados em Clique. Baseiam-se na coloração de cliques como primeiro passo, obtendo limites inferiores ao número cromático. Posteriormente, colore-se os outros vértices de acordo com algum critério (maior grau de saturação, maior grau de vértice, número de verificações). O algoritmo mais conhecido desta classe é o algoritmo de Brélaz chamado *DSATUR* [6] (Degree of SATURation) por ser um algoritmo que leva em conta o grau de saturação dos vértices.

Algoritmos de Zykov. Esses algoritmos utilizam a estrutura recursiva imposta aos grafos pelas decomposições de Zykov [56], mas usam heurísticas para achar uma boa, senão ótima, coloração, ao invés de fazer uma pesquisa exaustiva.

A maioria dos algoritmos resolvem o problema de coloração passando somente uma vez pelo conjunto de vértices. Joe Culberson [19] propôs um algoritmo baseado em iterações de algoritmos da classe Guloso e obteve bons resultados para alguns

grafos randômicos específicos [19]. Culberson também conseguiu melhores resultados utilizando um método híbrido, isto é, juntou algoritmos pertencentes a classes diferentes como o IG (iterated greedy) e *TABU search* num único algoritmo para resolver aproximadamente o problema de coloração.

Os algoritmos acima são utilizados para resolver o problema de coloração de grafos genéricos. Existem algoritmos específicos para determinados tipos de grafos como grafos planares, esparsos, cordais entre outros. Alguns algoritmos, teoremas e conjeturas podem ser encontradas em [33].

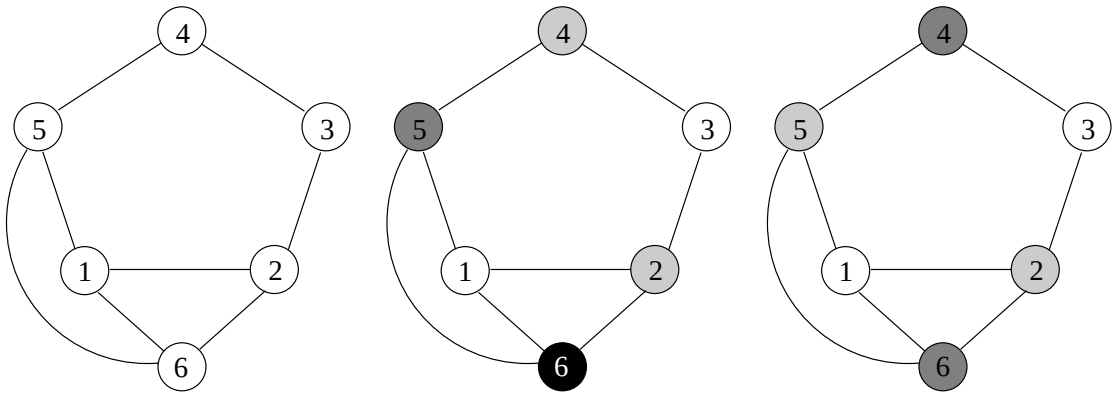
3.3 Implementação de Algoritmos de Coloração

Grande parte dos algoritmos de coloração de grafos são baseados numa varredura seqüencial do conjunto de vértices, onde estes são coloridos um de cada vez. O Algoritmo GULOSO, detalhado abaixo, é um exemplo de algoritmo simples de coloração mínima aproximada.

Algoritmo 1 ALGORITMO_GULOSO ($G(V, E)$)

para $i \leftarrow 1$ **até** $|V|$ **faça**
  $k \leftarrow 1$
  enquanto $\exists$ vértice adjacente a v_i com cor k **faça**
    $k \leftarrow k + 1$
  fim enquanto
  $Cor(\ v_i\ ) \leftarrow k$
fim para

Neste algoritmo, o número de cores utilizado é o valor de k ao final da execução e pode ser maior ou igual ao número cromático $\chi(G(V, E))$. Infelizmente, esse número pode ser bem maior que o número cromático, dependendo da ordem em que os vértices forem coloridos. Por exemplo, aplicando o Algoritmo 1 ao grafo da Figura 3.1(a), a partir do vértice 1 até o 6 seqüencialmente, obtém-se o resultado da Figura 3.1(b). A solução com 4 cores não é uma solução *mínima*. Com apenas 3 cores, obtém-se uma coloração mínima do grafo como ilustrado na Figura 3.1(c) colorindo os vértices na seguinte ordem: 1, 5, 2, 3, 4 e 6. O comportamento do



(a) grafo não colorido (b) coloração não-mínima (c) coloração mínima

Figura 3.1: Coloração Seqüencial.

algoritmo GULOSO é, portanto, **afetado** pela mudança no ordenamento dos vértices. Culberson cita algumas heurísticas utilizadas para a ordenação inicial dos vértices em [19].

Não é intuitivamente trivial entender a razão pela qual este algoritmo não resulta em uma solução ótima. Considerando os vértices ordenados de acordo com seus identificadores, os quatro primeiros vértices são coloridos com cores alternadas. Para colorir o vértice de número 5, é necessário uma terceira cor pois o vértice é adjacente aos vértices 1 e 4 que possuem cores distintas. Analogamente, o vértice 6 é colorido com uma quarta cor.

Com a finalidade de obter uma solução ótima, é necessário achar a melhor ordenação ou realizar uma recoloração de vértices previamente coloridos (em inglês utiliza-se o termo *backtracking*). Neste caso, o algoritmo deveria retornar ao ponto de tratamento do vértice 4, recolorir esse vértice e a partir daí prosseguir o algoritmo GULOSO, obtendo assim uma coloração ótima como ilustra a Figura 3.1(c). A utilização dessa técnica eleva a complexidade do algoritmo, consumindo espaço e tempo de CPU adicionais.

O Algoritmo GULOSO comporta-se, em termos de tempo de execução, em $O(|V|)$ no melhor caso (grafo não possuir nenhuma aresta) e em $O(|V|^2)$ no pior caso (grafo ser completo). Quando não há arestas, basta colorir todos os vértices com uma única

cor, percorrendo, portanto, $|V|$ vértices. No caso de grafo completo, é necessário verificar todos os vértices já coloridos a cada passo, crescendo, portanto, como uma progressão arimética de razão 1. Logo a complexidade é proporcional a

$$\sum i = 1|V| - 1i = \frac{|V|^2 - |V|}{2} \longrightarrow O(|V|^2)$$

.

Uma pesquisa exaustiva de todo possível conjunto de cores K que satisfaz a condição de coloração correta seria necessária para obter a solução ótima do problema. Sabemos que esse algoritmo, de complexidade exponencial, super-exponencial ou fatorial, é inviável na maioria dos casos reais onde o número de vértices em um grafo chega a centenas e ou milhares.

Algoritmo 2 ALGORITMO_EXATO ($G(V, E)$)

```
para k ← 1 até |V| faça
  repita
    K ← uma coloração entre as k^|V| possíveis
    se K é uma coloração correta de G então
      fim do algoritmo
    fim se
  até ∄ mais uma coloração possível com k cores
fim para
```

O Algoritmo EXATO, número 2, exemplifica um possível algoritmo exato exaustivo. Para cada número k, sendo k o número de cores necessário sendo testado e $1 \leq k \leq |V|$, são testadas $k^{|V|}$ combinações de conjunto de cores. Ou seja, verifica-se se o conjunto de cores K a ser atribuído ao conjunto de vértices corresponde a uma coloração correta do grafo. Exemplo: dado um grafo G, com 5 vértices, existem $1^5 + 2^5 + 3^5 + 4^5 + 1$ combinações possíveis, totalizando 1.301 combinações a serem testadas no algoritmo. Com 6 e 7 vértices, o número de combinações aumenta drasticamente para 20.516 e 376.762 respectivamente, com crescimento fatorial! Obviamente, o algoritmo apresentado é extremamente ingênuo. Um exemplo é o fato de testar casos iguais como $K_1 \leftarrow \{1, 2, 1\}$ e $K_2 \leftarrow \{2, 1, 2\}$ para um grafo de três vértices. K_1 é exatamente igual a K_2 pois a cor dada a cada vértice serve sim-

plesmente como referência aos vértices particionados, não contendo nenhum outro significado.

Um algoritmo exato e um pouco mais eficiente que o supra-citado foi proposto por Brélaz[6] baseado no algoritmo de Randall-Brown[7]. Este algoritmo evita redundância na enumeração de soluções (exemplificado acima) ao problema de coloração. Além disso, utiliza estruturas em árvores para acelerar o processo de busca no espaço de soluções. Todavia, a complexidade continua sendo super-exponencial, e portanto, inviável a aplicação de problemas reais.

Um dos algoritmos mais conhecidos em coloração aproximada é o algoritmo DSATUR de Brélaz[6] (Algoritmo 3), também exato para alguns casos. Este algoritmo serviu de base para muitos outros algoritmos que adaptaram-no a necessidades específicas, como no caso do algoritmo de Coudert[18].

Algoritmo 3 ALGORITMO_DSATUR ($G(V, E)$)

1: $V \leftarrow Ordem\_decrescente\_de\_grau(V)$
2: $Cor(\ V.pop()\ ) \leftarrow 1$
3: **repita**
4: /* ESCOLHE PRÓXIMO VÉRTICE */
5: $v \leftarrow grau\_max\_saturacão(G)$
6: **se** $\exists\ w \in V$ **tal que** $grau\_saturacão(w) = grau\_saturacão(v)$ **então**
7: $v \leftarrow V.pop()$
8: **fim se**
9: /* ESCOLHE COR PARA VÉRTICE */
10: $k \leftarrow 1$
11: **enquanto** $\exists$ vértice adjacente a v com cor k **faça**
12: $k \leftarrow k + 1$
13: **fim enquanto**
14: $Cor(\ v\ ) \leftarrow k$
15: **até** $\nexists\ v$ não colorido

Detalha-se o algoritmo de DSATUR para uma melhor compreensão:

1. O algoritmo utiliza, inicialmente, uma heurística de ordenamento de vértices, através da função $Ordem\_descrescente\_de\_grau$ que retorna uma lista de vértices em ordem decrescente de grau. Isso facilita a coloração, pois os vértices de maior grau serão, portanto, coloridos primeiro, ou seja, retirados da fila de não coloridos mais rapidamente que os de menor grau.

2. Seleciona-se, portanto, o ou um vértice de maior grau, colorindo-o com a cor 1. Utiliza-se a função *pop()* que devolve o primeiro da lista e o exclui da lista em seguida.

3-5. A cada iteração, seleciona-se o vértice com grau máximo de saturação[1] retornado pela função $grau\_max\_saturacão(G)$[2]. Com isso, Brélaz propõe que é interessante colorir aquele vértice que contiver o maior número de vizinhos já coloridos. Esse método acaba colorindo um clique do grafo.

6-8. Se existir algum vértice com o mesmo grau de saturação, substitui-se o vértice previamente escolhido pelo vértice de maior grau[3].

9-14. Escolhe-se a cor de menor índice possível a ser atribuída ao vértice escolhido.

15. Repete-se a iteração até que todos os vértices estejam coloridos.

Uma peculiaridade desse algoritmo encontra-se na escolha do próximo vértice a ser colorido. Primeiramente, colore-se o vértice com maior grau de saturação mas se houver algum outro que também possua esse mesmo grau (o que se denomina de *empate*), seleciona-se o vértice de maior grau. Se neste último caso ocorrer outro empate, o algoritmo original DSATUR escolhe aleatoriamente um dos vértices empatados. Esse é um dos pontos fracos que podem e são atacados por outros algoritmos que se baseiam no algoritmo DSATUR

Este algoritmo colore inicialmente um dos maiores cliques do grafo, encontrando limites inferiores ao número cromático. Desta maneira, ele é exato para grafos bipartidos e é um bom método para verificar, em $O(n^2)$, se o grafo é ou não bipartido. As provas podem ser encontradas em [6].

A ordenação inicial de vértices de acordo com um critério é extremamente importante antes da coloração propriamente dita. O próprio algoritmo GULOSO, por

[1] vértice que tiver maior número de vértices já coloridos entre os seus adjacentes.

[2] é necessário o grafo inteiro e não só o conjunto de vértices para verificar seu grau de saturação, por que este último depende da execução do algoritmo de coloração.

[3] outras heurísticas baseadas no algoritmo de DSATUR exploram este ponto de empate.

exemplo, é capaz de produzir uma coloração ótima de um grafo dependendo da ordenação de seus vértices[20]. Todavia, o teste de todas as possíveis ordenações para um determinado número n de vértices é $n!$, sendo, portanto, impraticável para grafos reais de 100 vértices ou mais. É comum, por conseguinte, encontrar ordens especiais para os vértices de um grafo, de maneira que esses permitam uma coloração mais fácil. Chvátal [12] definiu ordenamentos *perfeitos* e *grafos perfeitamente ordenáveis* e propôs técnicas de coloração eficientes em tempo polinomial.

Mais informações sobre a implementação e eficiência desses três algoritmos citados podem ser encontrados na Seção 5.

CAPÍTULO 4

Aplicações em VLSI

Nesta Seção, apresentam-se alguns problemas encontrados em síntese VLSI que utilizam coloração de grafos em suas resoluções. A coloração pode ser apenas uma pequena parte da modelagem do problema a ser resolvido ou pode ser um mapeamento completo do problema. Neste último caso, são problemas que consistem na partição de objetos de acordo com determinados critérios. Analisa-se em cada estudo de caso o tipo de grafo que deve ser colorido a fim de obter características especiais que podem ajudar na coloração.

Escolheu-se o problema de *codificação Booleana restrita* como o estudo de caso a ser implementado utilizando a biblioteca de algoritmos desenvolvida. Na Seção 4.2 encontram-se definições básicas necessárias ao entendimento do problema e logo a seguir explicam-se os problemas escolhidos como estudos de caso.

4.1 Estudos de Caso

Em síntese de alto nível de hardware, Springer et al [46] identificaram tipos especiais de grafos que aparecem freqüentemente na área, sendo utilizados para a alocação de operadores, valores e transferência de dados em recursos compartilhados. Entre esses tipos especiais, identificaram dois tipos de grafos: *cordais* e *de comparabilidade*, além dos grafos de *intervalo* e de *arco-circular*, utilizados em sistemas já

existentes de síntese de alto nível. No caso de grafos cordais e grafos de intervalo existem algoritmos que resolvem o problema de forma ótima em tempo polinomial [27].

Ciesielski & Yang [14] em seu algoritmo *PLADE*, utilizaram um procedimento baseado em coloração e particionamento de grafos para resolver o problema de codificação restrita de entradas em PLAs (Programmable Logic Arrays).

Na área de síntese de circuitos assíncronos, Chu et al. [11] propuseram uma técnica de codificação de estados livre de corridas críticas para máquinas de estados. Utilizou-se um algoritmo guloso heurístico de coloração de grafos.

Wan & Perkowski [50] desenvolveram um método de coloração de grafos para realizar uma codificação quase-ótima de *don't cares* em funções Booleanas, visando resolver o problema de mapeamento tecnológico para FPGAs (Field Programmable Gate Arrays). Esse método recebeu o nome de "Método da Influência da Cor". A idéia principal é avaliar a influência da atribuição da cor a um vértice no grafo inteiro, e escolher a cor que resulta numa influência mínima.

Yang e Ciesielski [55] propuseram uma formulação teórica do problema de codificação de entrada[1], baseado na compatibilidade de dicotomias (partições de dois blocos de um subconjunto de um dado conjunto, ver definição na página 42). Eles estudaram a viabilidade de manipular essa compatibilidade através de um grafo de incompatibilidade de dicotomias.

Outra aplicação recente foi a de Coudert [18] que propõe colorir grafos gêmeos (*twin graphs*) para resolver o problema de satisfactibilidade de restrições em codificação Booleana. Coudert obteve ótimos resultados, utilizando um algoritmo modificado, baseado no algoritmo de DSATUR [6].

Um dos objetivos específicos deste trabalho foi, baseado na idéia de Coudert, criar um algoritmo de satisfação de restrições de pseudo-dicotomias e integrá-lo ao codificador ASS†UCE [10]. Estudou-se a interface necessária para mapear o problema de satisfação das dicotomias [13, 18] para o problema de coloração de

[1] Codificação de entrada é um problema de codificação Booleana que associa códigos aos símbolos do domínio de uma função Booleana. Uma definição mais formal pode ser encontrada em [8].

grafos.

4.2 Codificação Booleana Restrita

Dentre todos os estudos de caso, escolheu-se estudar o problema de ***codificação Booleana restrita*** [8] por três motivos principais:

- pela existência de trabalhos semelhantes já publicados que mapeiam o problema original para o problema de coloração de grafos [13, 18, 55];
- pelo fato do problema ser um dos resolvidos heuristicamente pelo orientador deste trabalho em sua tese de doutorado; o autor, por sua vez, por ter trabalhado como bolsista com o orientador durante um bom tempo, familiarizou-se com o problema;
- o problema que se deseja resolver, é o problema de codificação Booleana restrita baseada em ***pseudo-dicotomias***[8], uma abordagem distinta de propostas anteriores;

Um passo fundamental no processo de síntese de sistemas digitais é a codificação em binário dos conjuntos de símbolos manipulados na especificação do sistema, denominado de ***codificação Booleana***.

Definição 4.1 (codificação Booleana) *Atribui-se o nome de* ***codificação Booleana*** *de S ao mapeamento ou função completa $e : S \longrightarrow \mathcal{P}(\mathcal{B}^n)$, onde S é um conjunto finito de* ***símbolos****, $\mathcal{B} = \{0, 1\}$, $\mathcal{B}^n$ é o produto cartesiano $\overbrace{\mathcal{B} \times \mathcal{B} \times \ldots \times \mathcal{B}}^{n\,vezes}$ e $\mathcal{P}(\mathcal{B}^n)$ é o conjunto potência[2] de vetores binários de comprimento fixo n. Para todo $s \in S$, a imagem $e(s)$ é denominada o* ***código*** *do elemento s, onde n é o* ***comprimento*** *do código. Dois códigos $e(s)$, $e(t)$ são* ***disjuntos*** *se e somente se $e(s) \cap e(t) = \emptyset$, caso contrário os códigos são* ***intersectantes****.*

Pode-se definir alguns casos especiais de codificação Booleana, tais como codificação injetiva e codificação funcional.

[2] *conjunto potência é o conjunto de todos os subconjuntos distintos de um conjunto.*

Definição 4.2 (codificação Booleana injetiva) *É uma codificação onde a função completa e é uma injeção*[3] *e os códigos gerados são dois a dois* disjuntos, *ou seja,*

$$\forall(s,t \in S),\ s \neq t \Longrightarrow e(s) \cap e(t) = \emptyset.$$

Qualquer codificação que não seja injetiva é, naturalmente, dita ***não-injetiva****.*

Definição 4.3 (codificação Booleana funcional) ***Codificação funcional*** *de um conjunto de símbolos S é uma codificação onde todo código é um conjunto* ***unitário***[4]*. Coerentemente, uma codificação funcional é redefinida como uma função $e : S \longrightarrow \mathcal{B}^n$, sem perda de generalidade. Qualquer codificação que não é funcional é denominada* ***não-funcional****.*

As Figuras 4.1 e 4.2 ilustram exemplos de codificação Booleana. Os símbolos a, b e c do conjunto S, no primeiro exemplo, são codificados com os vetores Booleanos disjuntos 00, 10 e 11, caracterizando, conseqüentemente, uma codificação Booleana *injetiva* e *funcional*. Em contrapartida, no exemplo da Figura 4.2, os símbolos a e b e c possuem códigos intersectantes, caracterizando uma codificação Booleana *não injetiva*. Ao mesmo tempo, a codificação é *não funcional* pois os códigos associados aos símbolos b e c não são *unitários*.

Define-se informalmente compatibilidade de símbolos necessária à definição de uma *codificação Booleana restrita válida* a seguir.

Definição 4.4 (compatibilidade de símbolos) *Dois símbolos s ao compatíveis quando não existir nenhuma restrição de codificação para separá-los.*

Definição 4.5 (codificação Booleana restrita[8]) *Considerem-se dados um conjunto de símbolos S e um conjunto de* ***restrições de codificação*** *sobre os símbolos de S, tal que a* satisfação *de uma restrição possui um* ***ganho*** *associado. Resolver o problema de* ***codificação Booleana restrita*** (CBR) *consiste em obter uma codificação*

[3] *não existe uma mesma imagem para dois elementos distintos do conjunto (mapeamento 1-1).*
[4] *não possui mais de 1 código no conjunto*

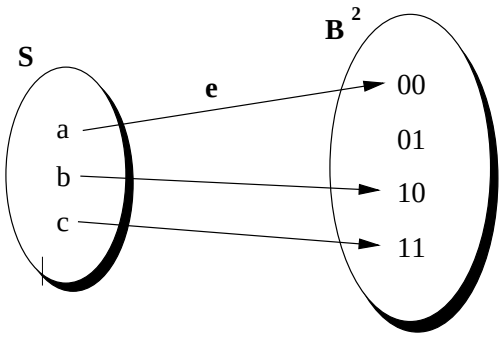


Figura 4.1: Exemplo de codificação Booleana injetiva *e* funcional.

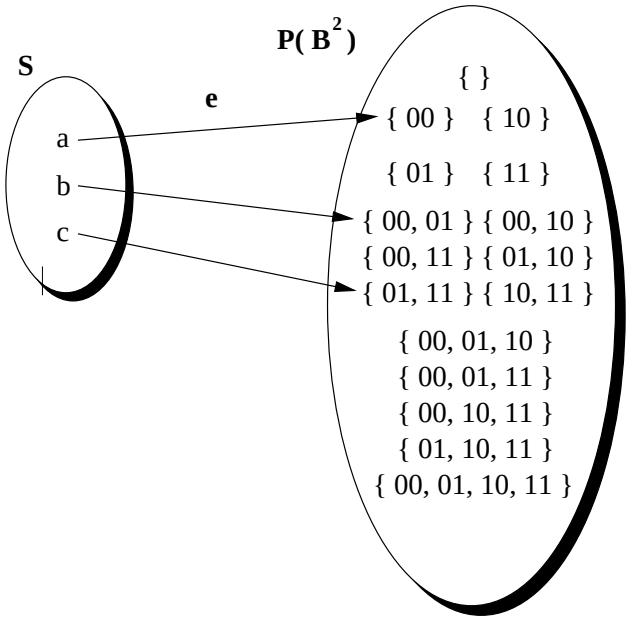


Figura 4.2: Exemplo de codificação Booleana não injetiva *e* não funcional.

Booleana de S que atende uma ***função de satisfação*** *(que depende das restrições impostas sobre S) e tal que:*

- *esta codificação possui o menor comprimento possível;*
- *maximize-se o* ganho *obtido pela codificação quando computado sobre todos os ganhos sugeridos pela satisfação de restrições de codificação.*

Define-se informalmente abaixo uma codificação Booleana restrita *válida*:

Definição 4.6 (codificação Booleana restrita válida) *Uma codificação Booleana restrita é* ***válida*** se e somente se *sob esta esta codificação dois símbolos quaisquer são intersectantes apenas quando eles são compatíveis.*

Neste escopo, uma codificação funcional injetiva *sempre é* válida.

Resolver o problema geral de CBR é uma tarefa extremamente complexa, seja por se tratar normalmente de problemas cuja solução é computacionalmente custosa, seja porque a própria formulação exata da função ganho adequada é difícil quando não inviável. Definem-se, então, para este problema, duas aproximações:

Codificação Booleana Restrita Completa - codificação de menor comprimento possível que respeita todas as restrições. Obviamente, entende-se que uma codificação que respeite todas as restrições nem sempre irá obter uma codificação mínima;

Codificação Booleana Restrita Parcial - codificação com um dado comprimento (com freqüência, o mínimo absoluto necessário) que satisfaz o maior número possível de restrições.

A Codificação Booleana Restrita Parcial, portanto, não necessariamente satisfaz todas as restrições não excluindo o fato de achar uma codificação válida.

O problema de *codificação Booleana restrita* baseia-se em duas grandes tarefas: a *geração* das restrições impostas ao conjunto de símbolos a ser codificado e a *satisfação* das mesmas. No presente trabalho, tenta-se mapear o problema de satisfação de restrições para o problema de coloração de grafos. O problema de geração das restrições depende do problema original a ser tratado e não é abordado neste trabalho.

Originalmente introduzido por Tracey [48], o problema de codificação Booleana restrita baseada em dicotomias pode ser usada para resolver os problemas abaixo entre outros citados em [18]:

- gerar uma implementação assíncrona crítica livre de corridas [48] ou independente de portas e capacitância de atrasos [49];

- gerar uma implementação de PLA de área mínima [22, 21, 55];

- gerar uma implementação PLA ótima de expressões Booleanas [23];

As dicotomias e ***pseudo-dicotomias*** são, portanto, um conceito útil para modelar restrições em problemas de codificação Booleana. Em geral, *restrições de codificação* consistem em indicações para *separar* ou *não separar* os códigos de um conjunto de símbolos. Partições de dois conjuntos são adequadas para modelar tal comportamento. Uma partição de dois blocos pode ser interpretada como uma indicação para fazer bits dos códigos de símbolos em um bloco distintos dos bits dos códigos de símbolos no outro bloco (separando assim os códigos dos símbolos posicionados em blocos distintos).

As pseudo-dicotomias [8] são uma estrutura algébrica com duas partes: uma relação binária que captura o conceito similar a partições de dois blocos, e uma função de chaveamento geral[5]. A relação binária captura a característica de separação (ou não) de símbolos, enquanto que a função indica como satisfazer os requisitos específicos de cada classe de restrições.

Descreve-se aqui uma definição informal de pseudo-dicotomia baseada na definição formal utilizada na descrição de restrições de codificação do trabalho realizado por Calazans[8].

Definição 4.7 (pseudo-dicotomia) *Seja S um conjunto de símbolos e S' um subconjunto de S. Uma* ***pseudo-dicotomia*** (PD) *é uma partição de dois blocos de S', A e B, e uma função de chaveamento f que associa esta partição a códigos que a satisfazem, ($A : B : f$). Uma* ***pseudo-dicotomia semente*** *é uma pseudo-dicotomia onde pelo menos um dos dois blocos da partição é* unitário. *Dado um vetor Booleano $\varkappa = x_{n-1} \ldots x_0$, uma pseudo-dicotomia é* ***satisfeita*** *por $\varkappa$* se e somente se *o resultado de sua aplicação à função de chaveamento da* PD *resultar no valor 1. Uma* PD *é* ***ordenada*** *se ($A : B : f$) é diferente de ($B : A : f$) caso contrário ela é considerada*

[5] definições formais de partição, função de chaveamento, e pseudo-dicotomias podem ser encontradas em [8].

não ordenada. *Uma* **dicotomia** *é um caso especial de uma pseudo-dicotomia onde* $S' = S$.

Considere um conjunto de símbolos $S = \{\ foo, bar, lala, xuxu\ \}$ e um conjunto de restrições $R = \{$ *separe o código de 'foo' dos códigos de 'bar' e 'lala'*, *separe o código de 'bar' dos códigos de 'lala' e 'xuxu'* $\}$. As pseudo-dicomotias geradas seriam: $(\ foo : bar, lala : f\ )$ e $(\ bar : lala, xuxu : g\ )$. $f = (\ 0, 1, 1, -\ )$ e $g = (\ -, 0, 1, 1\ )$ podem ser definidos desta maneira, utilizando a notação de *vetor valor*[8]. Note que o código de 'xuxu' e 'foo' não importam na satisfação da restrição, recebendo portanto um *don't care* na função f e g respectivamente. A Figura 4.3 ilustra este exemplo.

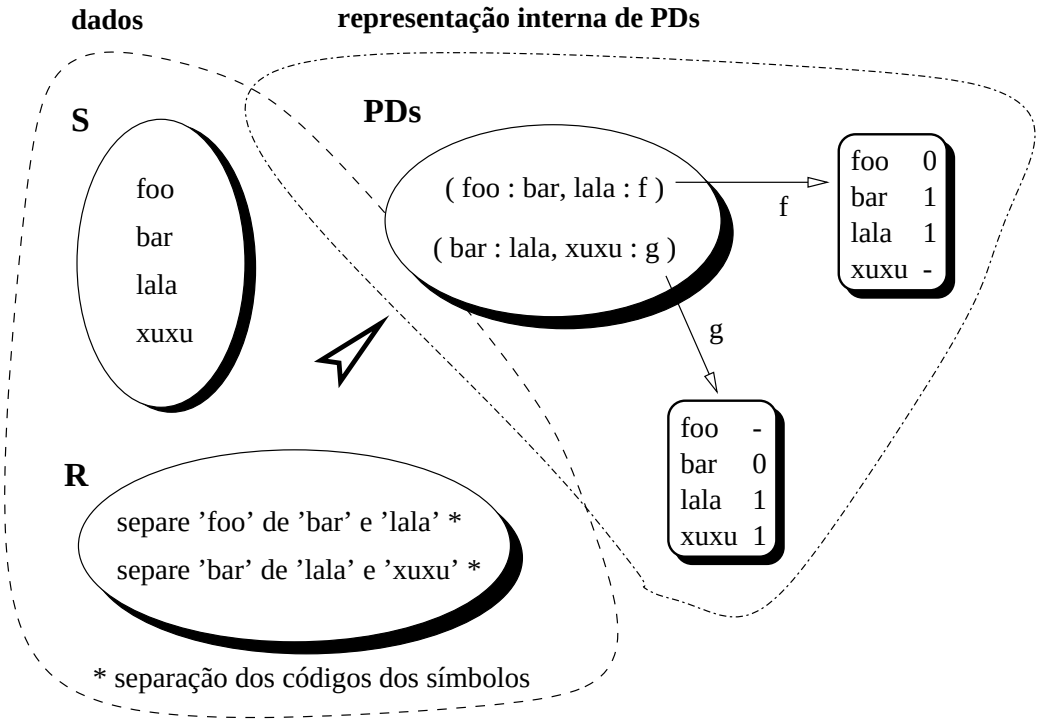


Figura 4.3: Exemplo de representação de pseudo-dicotomia (PD).

Definição 4.8 (compatibilidade de pseudo-dicotomias) *Uma pseudo-dicotomia* ω *é compatível com outra pseudo-dicotomia* ϖ (compatibilidade 2 a 2) *quando uma das proposições abaixo é verdadeira:*

a) a interseção da união do bloco A de ω com o bloco A de ϖ, com a união do bloco B de ω com o bloco B de ϖ é vazia, ou seja, $(A_\omega \cup A_\varpi) \cap (B_\omega \cup B_\varpi) = \emptyset$;

b) a interseção da união do bloco A de ω com o bloco B de ϖ, com a união do bloco B de ω com o bloco A de ϖ é vazia, ou seja, $(A_\omega \cup B_\varpi) \cap (B_\omega \cup A_\varpi) = \emptyset$;

No caso de PD ordenada, *a preposição a) deve ser verdadeira.*

Três pseudo-dicotomias ω, ϖ e φ são compatíveis quando uma das proposições abaixo é verdadeira:

a) $(A_\omega \cup A_\varpi \cup A_\varphi) \cap (B_\omega \cup B_\varpi \cup B_\varphi) = \emptyset$;

b) $(A_\omega \cup A_\varpi \cup B_\varphi) \cap (B_\omega \cup B_\varpi \cup A_\varphi) = \emptyset$;

c) $(A_\omega \cup B_\varpi \cup A_\varphi) \cap (B_\omega \cup A_\varpi \cup B_\varphi) = \emptyset$;

d) $(A_\omega \cup B_\varpi \cup B_\varphi) \cap (B_\omega \cup A_\varpi \cup A_\varphi) = \emptyset$;

*Analogamente define-se compatibilidade de pseudo-dicotomias **n a n** seguindo a mesma linha de raciocínio, alternando todas as combinações possíveis a serem testadas.*

Definição 4.9 (compatível de PD**)** *Um **compatível** é um conjunto de* PD*s compatíveis n a n. Um compatível Υ é **máximo** quando não existir outro compatível distinto Ψ que contenha todas as* PD*s que formam Υ.*

Pseudo-dicotomias e dicotomias são utilizadas com freqüência na modelagem de restrições de codificação. Yang [55], Ciesielski [13], Calazans [8, 9] e Coudert [18] são exemplos de trabalhos recentes neste tipo de aplicação.

4.2.1 Pseudo-coloração de Yang

Yang e Ciesielski [55] propuseram uma formulação teórica do problema de codificação de entrada[6], baseado na compatibilidade de dicotomias. Estes autores descrevem três técnicas para resolver o problema:

[6]Codificação de entrada, é um problema de codificação Booleana que associa códigos aos símbolos do domínio de uma função Booleana. Uma definição mais formal pode ser encontrada em [8].

1. técnicas derivadas de minimização lógica clássica (geração de dicotomias primas e resolução através do problema de cobertura);

2. coloração de grafos aplicada ao grafo de incompatibilidade de dicotomias sementes;

3. extração de dicotomias primas essenciais, reduzindo o tamanho do problema de coloração de grafos e cobertura;

Do artigo de Yang [55] pode-se extrair as definições abaixo. Note que Yang chama dicotomia como na verdade sendo uma pseudo-dicotomia. Para maior coerência no presente trabalho, as definições abaixo baseam-se na definição de pseudo-dicotomias (PD).

Definição 4.10 (grafo de incompatibilidade de PDs sementes) *Um* ***grafo de incompatibilidade de PDs sementes*** (GIPDS) *é um grafo* $G(\ V, E\ )$ *onde* V *é o conjunto de PDs sementes e* E *é o conjunto de pares de PDs sementes incompatíveis representando os vértices e as arestas respectivamente. Cada aresta* $(\ v, w\ ) \in E$ *representa que a PD* v *é incompatível com a PD* w*.*

Definição 4.11 (coloração compatível do GIPDS) K *é uma* ***coloração compatível do grafo de incompatibilidade de PDs sementes*** $G(\ V, E\ )$ *quando* K *colore* G *corretamente e cada PD semente associada a uma mesma cor pode ser representada por uma* única PD *para todas as cores utilizadas na coloração.*

Essa definição, portanto, difere do problema geral de coloração de grafos por impor uma segunda restrição, estranha ao problema clássico de coloração. Para resolver este problema não é suficiente associar cores diferentes a PDs sementes incompatíveis, é também necessário verificar se existe uma PD, para cada conjunto de PDs sementes de mesma cor, que as substitua e as contenha.

Na Figura 4.4, ilustra-se uma coloração (b) do grafo de incompatibilidade de PDs semente (a). O grafo é bipartido, portanto, duas cores são suficientes para sua coloração. As PDs semente $(12 : 3)$, $(12 : 4)$ e $(3 : 4)$ foram coloridas com a cor 1

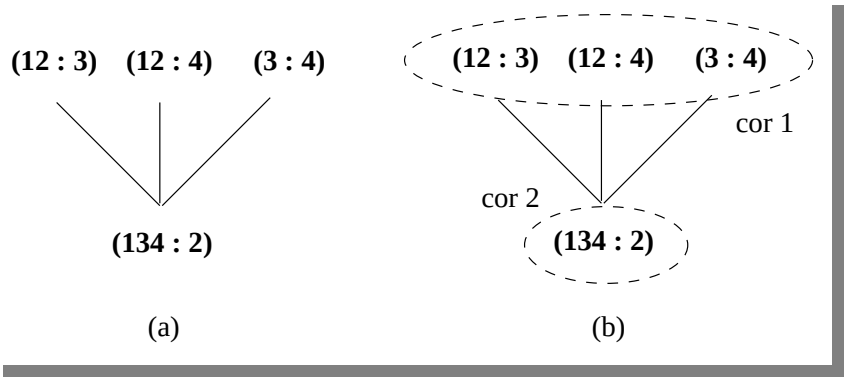


Figura 4.4: Coloração do grafo de incompatibilidade de PDs sementes.

e devem portanto ser reduzidas a uma *única* PD que as contenha. É fácil perceber que não existe tal PD, pois a junção das duas primeiras geraria a PD (12 : 34), incompatível com (3 : 4). Analogamente as junções da primeira com a terceira, e da segunda com a terceira gerariam PDS incompatíveis com a segunda e primeira respectivamente.

O problema, portanto, não é um problema de coloração de grafos geral. Essa é uma das razões pela qual Yang, em [55], propôs uma heurística de *coloração compatível* de grafos não utilizando algoritmos de coloração existentes, havendo a necessidade de uma reimplementação de código. Essa implementação é similar à implementação de um algoritmo GULOSO de coloração de grafos, com algumas modificações.

Todavia, estudando a relação entre algoritmos de coloração de grafos clássicos e algoritmos de coloração compatível de PDs sementes, derivaram-se observações e teoremas importantes.

Observação 4.1 ***Relação entre*** **compatíveis máximos** ***e o*** GIPDS *A relação entre* compatíveis máximos *e o* GIPDS *é que o primeiro não precisa de mais de uma cor na coloração do segundo, ou seja, se obtivermos o número cromático do* GIPDS*, garante-se que não existem compatíveis máximos com mais de uma cor. É fácil observar a validade dessa observação, pois um compatível máximo é composto de um conjunto de* PDs*, todas compatíveis n a n entre si. Portanto, uma cor é suficiente*

para representá-las.

Para a resolução do problema de *codificação Booleana restrita completa* (CBRC), os seguintes teoremas podem ser definidos:

Teorema 4.1 *Resolver* exatamente *o problema de coloração do grafo de incompatibilidade de* PDs *sementes* (GIPDS) *fornece um* limite inferior não-estrito *ao comprimento de código para o problema de codificação de símbolos.*

Prova. *Resolver exatamente o problema de coloração do* GIPDS *consiste obviamente em obter o número cromático χ do grafo. A partir da Definição 4.10 e da Observação 4.1, o teorema afirma que este número cromático é um limite inferior para o comprimento mínimo de uma codificação* válida *do problema* CBR *que gerou o* GIPDS.

Provamos esta afirmativa por contradição. Assume-se que existe, para tal problema, uma codificação válida com comprimento $\xi < \chi$. Neste caso, pela Definição 4.8 de compatibilidade/incompatibilidade de PDs*, pela Definição 4.10 de* GIPDS *e pela Observação 4.1, tomando-se o* GIPDS*, é possível partir desta codificação e* recolorir *o grafo de partida com ξ cores e esta constitui uma coloração válida do* GIPDS*, o que contradiz o fato de χ ser o número cromático. Logo ξ deve ser $\geq \chi$, provando o teorema.* ■

4.2.2 Pseudo-coloração de Coudert

Coudert propôs, em [18], heurísticas e um algoritmo exato para a solução do problema de *codificação Booleana restrita* baseada em dicotomias.

Sua proposta difere de outras anteriores pela representação interna das dicotomias obtidas da geração de restrições. Coudert utiliza o conceito de ***grafo gêmeo*** (do inglês, *twin graph*), para representação das dicotomias e colore o grafo através de um algoritmo modificado baseado no algoritmo DSATUR.

A idéia de grafo gêmeo é ter uma representação de dicotomias na qual estas estejam ordenadas, veja Figura 4.5. Isto evita a necessidade da segunda restrição imposta pela coloração do grafo de incompatibilidade de dicotomias, visto anteriormente,

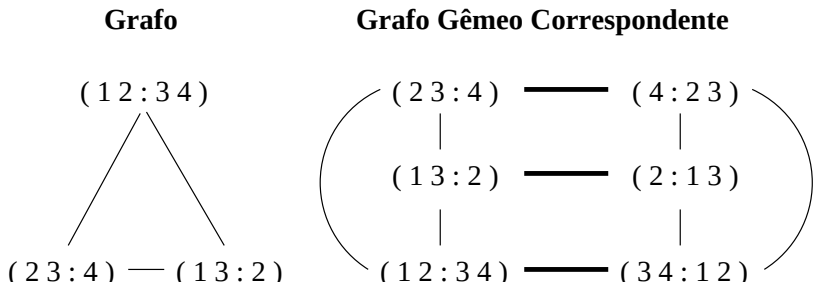


Figura 4.5: Exemplo de grafo *gêmeo*.

pois as dicotomias que receberão a mesma cor, são garantidamente compatíveis n a n. A prova é direta da definição de compatibilidade de dicotomias na página 42.

Todavia, o problema de colorir um grafo gêmeo não é um problema de coloração pura, pois necessita que se escolha um *gêmeo representativo* [18]. Cada par ordenado de dicotomias gerado por uma dicotomia não ordenada deve ter um gêmeo representativo que irá fazer parte da coloração do grafo. A idéia de gêmeo representativo expressa o fato de uma dicotomia não ordenada gerar sempre duas dicotomias ordenadas e estas representarem a **mesma restrição**, não havendo necessidade de colorir as duas, bastando simplesmente escolher qual a melhor a ser colorida, dependendo dos conflitos que elas apresentam no grafo, no momento da coloração. Isso impõe portanto uma segunda restrição ao problema de coloração: escolher qual das duas dicotomias deve ou não ser colorida.

Assim, Coudert re-implementou o algoritmo de coloração de grafos DSATUR realizando pequenas modificações como a inserção da segunda restrição recém citada e criando, portanto, uma *pseudo-coloração*. Se Coudert tivesse à disposição um algoritmo de coloração que pudesse levar em conta essa segunda restrição imposta pelo seu problema ao problema de coloração de grafos, seu trabalho teria sido menos penoso. Uma das grandes motivações para a criação de uma função que pudesse interferir no algoritmo de coloração a cada vez que este colorisse um vértice de um grafo nasceu das necessidade de facilitar tarefas de implementação como as empreendidas por Yang e Coudert. Essa função foi incluída na biblioteca como visto nos detalhes da implementação na Seção 5.1.1.

Seus resultados, porém, são bem interessantes, melhores que suas próprias abordagens anteriores baseadas em BDD/ZBDDs [17]. Esses resultados deram motivações para estudar o problema mais complexo resolvido por ASS†UCE[10] e utilizar a idéia de colorir o grafo de incompatibilidade de dicotomias para acelerar o processo de satisfação de restrições, um dos gargalos do software.

4.2.3 ASS†UCE

ASS†UCE[8] é um ambiente exploratório para Máquinas de Estado Finitas (FSM). O ambiente, atualmente, codifica e minimiza os estados de uma FSM utilizando um codificador baseado nas restrições representadas por pseudo-dicotomias. Através das restrições de *minimização de estados* e *codificação de estados*, ASS†UCE constrói as pseudo-dicotomias juntando essas duas restrições. ASS†UCE tem um embasamento teórico diferente do utilizado por outras abordagens em codificação que minimizam primeiramente os estados seguido de uma codificação dos mesmos. No ambiente ASS†UCE, realiza-se a minimização de estados juntamente com a codificação representando-as num arcabouço unificado de restrições.

Como visto anteriormente, a codificação Booleana resume-se em dois grandes passos:

1. a geração de restrições;

2. a satisfação das mesmas.

O codificador integrado ao ambiente ASS†UCE, atualmente, realiza uma codificação Boolena gulosa não baseada em coloração de grafos, mas sim em uma iteração utilizando uma *estrutura algébrica de manipulação* de PDs via matrizes esparsas.

Percebeu-se que o desempenho do passo que domina o tempo de execução do algoritmo de satisfação de restrições ASS†UCE não era muito eficiente e poderia ser melhorado. Surgiu a idéia de resolver este problema em parte como um problema de coloração de grafos.

O problema de satisfação de restrições do ASS†UCE é o mesmo do problema citado na Seção 4.2.1. A diferença é a representação das restrições de dicotomias

para pseudo-dicotomias.

A partir do estudo da viabilização de representar o problema de codificação Booleana restrita baseado em coloração, implementaram-se dois algoritmos baseados na *coloração do grafo de incompatibilidade de* PDs que resolvem o problema de satisfação de restrições, descrito pelos Algoritmos SATISFAÇÃO_SIMULTÂNEA e SATISFAÇÃO_SERIAL. Denominou-se de SATISFAÇÃO_SIMULTÂNEA uma coloração do grafo de incompatibilidade de PDs levando em conta uma segunda restrição para a coloração. A restrição é que as PDs que forem agrupadas em uma mesma classe, ou seja, receberem uma mesma cor na coloração, devem ser compatíveis n a n. Chamou-se essa coloração, portanto, de uma pseudo-coloração. O Algoritmo de SATISFAÇÃO_SERIAL realiza uma *coloração pura*[7] do grafo de incompatibilidade de PDs. A Figura 4.6 ilustra a implementação desses algoritmos e sua integração com o ambiente ASS†UCE.

Ambos os algoritmos retornam um conjunto de PDs que satisfazem a coloração. A partir desse conjunto, gera-se a codificação de maneira trivial, atribuindo uma coluna de bit para cada PD resultante.

Algoritmo 4 ALGORITMO_SATISFAÇÃO_SERIAL ($G\_PD\ (V, E)$)

1: Colore o grafo de incompatibilidade de PDs (G_PD);
2: **para todo** conjunto de PDs que foram agrupadas em uma classe através da coloração **faça**
3: **enquanto** $\nexists$ uma única PD que agrupe todas as PDs dessa classe **faça**
4: quebra-se esse conjunto em duas partes e testa-se novamente com as novas classes;
5: **fim enquanto**
6: substitui-se essa PD por todas as outras da classe em questão;
7: **fim para**
8: retira-se a redundância gerada pela divisão de classes das PDs, retirando aquelas PDs que já estão contidas em outras.
9: retorna o conjunto de PDs que satisfazem a codificação;

Outra característica importante do passo de satisfação de restrições do codificador ASS†UCE também implementado nesses novos algoritmos é levar em conta

[7] coloração sem nenhuma outra restrição além das arestas

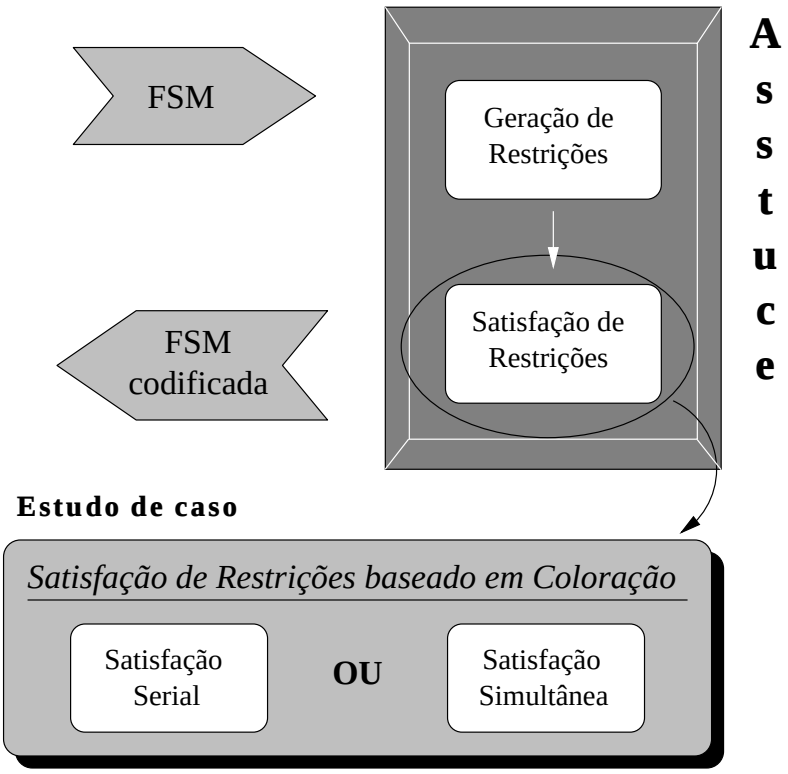


Figura 4.6: Esquema do algoritmo de satisfação de restrições integrado ao ASS†UCE.

o tipo de satisfação a ser empregado, ou seja, é necessário haver uma opção para realizar uma Codificação Booleana Restrita *completa* ou *parcial* (ver página 40).

Para tal, criou-se um processo pós-construção do grafo de incompatibilidade de PDs e pré-satisfação de restrições que analise se é necessário eliminar algumas PDs no caso da codificação ser *parcial*. Neste caso utilizamos uma heurística de eliminação de PD. A heurística constitui em eliminar aquelas PD que representam restrições já previamente representadas por outra ou outras PDs. Considere, por exemplo, as PDs

Algoritmo 5 ALGORITMO_SATISFAÇÃO_SIMULTÂNEA ($G\_PD\,(V, E)$)

1: $f \leftarrow$ função que indica se o vértice representando uma PD pode ou não juntar-se com os outros vértices já previamente coloridos com a cor em questão;
2: Colore o grafo de incompatibilidade de PDs (G_PD, f);
3: /* f SERÁ CHAMADO A CADA VEZ QUE UM VÉRTICE FOR RECEBER UMA DETERMINADA COR */
4: retorna o conjunto de PDs que satisfazem a codificação;

ilustradas na Figura 4.7 e as respectivas restrições que elas impõem.

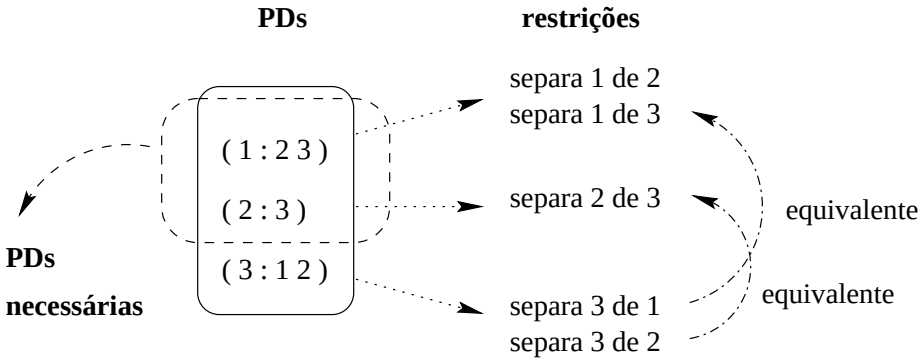


Figura 4.7: Exemplo do processo de eliminação de PDs em uma codificação Booleana restrita parcial.

Pode-se observar que a PD (3 : 1 2) não é necessária ao algoritmo de satisfação pois as duas restrições que ela impõe, a de separar 3 de 1 e 3 de 2 já estão representadas pelas outras duas PDs, (1 : 2 3) e (2 : 3).

Exemplo

Detalha-se um exemplo de satisfação de restrição de uma máquina de estados finita para a melhor compreensão dos algoritmos citados anteriormente.

Utilizou-se a FSM `dk15.kiss2` como exemplo. ASS†UCE produz as PDs ilustradas no grafo de incompatibilidade da Figura 4.8 como saída da *geração de restrições*.

O Algoritmo de SATISFAÇÃO_SERIAL colore esse grafo obtendo, portanto, as classes de PDs ilustradas na Figura 4.9. Cada cor representa uma classe de PDs onde todas as PDs da mesma classe (ou de mesma cor) são compatíveis duas a duas. Não é garantido, porém, que as PDs de uma mesma classe são compatíveis entre si, ou seja, se existe uma única PD que possa as substituir. Para resolver tal problema o Algoritmo de SATISFAÇÃO_SERIAL impõe uma condição a cada classe estabelecendo que deve existir uma única PD que represente a classe. Através de um método iterativo, o algoritmo particiona as classes de PDs nas quais não é possível encontrar essa única PD e testa novamente a mesma condição para cada uma das

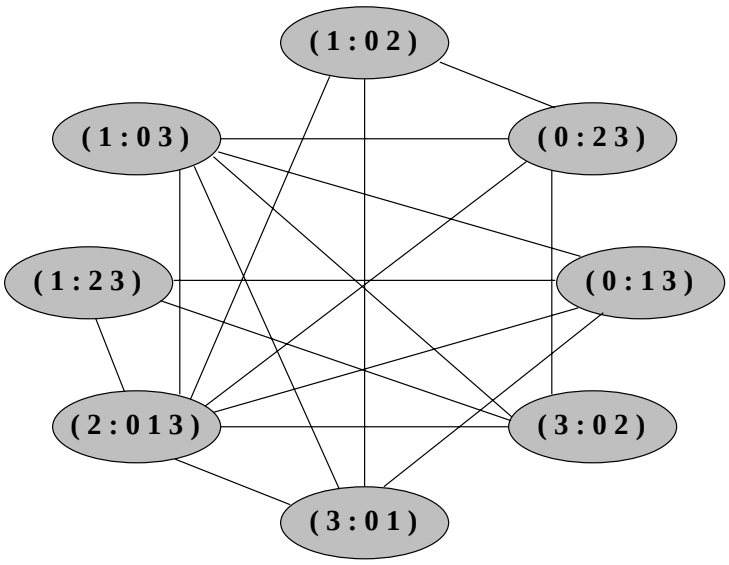


Figura 4.8: Grafo de incompatibilidade de PD da FSM DK15.

classes particionadas. Esse processo termina quando todas as classes geradas da coloração ou do particionamento de outras classes têm uma PD como representante.

No caso da FSM DK15, as classes geradas não precisaram de um particionamento. É fácil encontrar uma PD que represente cada uma das classes: (1 : 0 2 3) para a classe #1, (0 : 1 2 3) para a classe #2, (3 : 0 1 2) para a classe #3 e a última classe permanece com sua única PD (2 : 0 1 3).

Percebeu-se que era ainda necessário um passo posterior a esse particionamento. Devido ao próprio particionamento, em alguns casos, pode ocorrer de uma PD ser compatível com outra PD. É indispensável, portanto, um procedimento que agrupe essas PDs que se tornaram compatíveis ao final do particionamento e que conseqüentemente geraram redundâncias nas restrições.

As PDs finais, geradas pela junção das outras PDs da mesma classe, encontram-se ilustradas na Figura 4.10.

O Algoritmo de SATISFAÇÃO_SIMULTÂNEA, por sua vez, caracteriza-se por impor uma segunda restrição à coloração. Essa segunda restrição é a mesma imposta por Yang[55] em seu trabalho. Consiste em verificar, no momento da coloração, se a PD pode ou não ser colorida com a cor em questão através de uma análise de

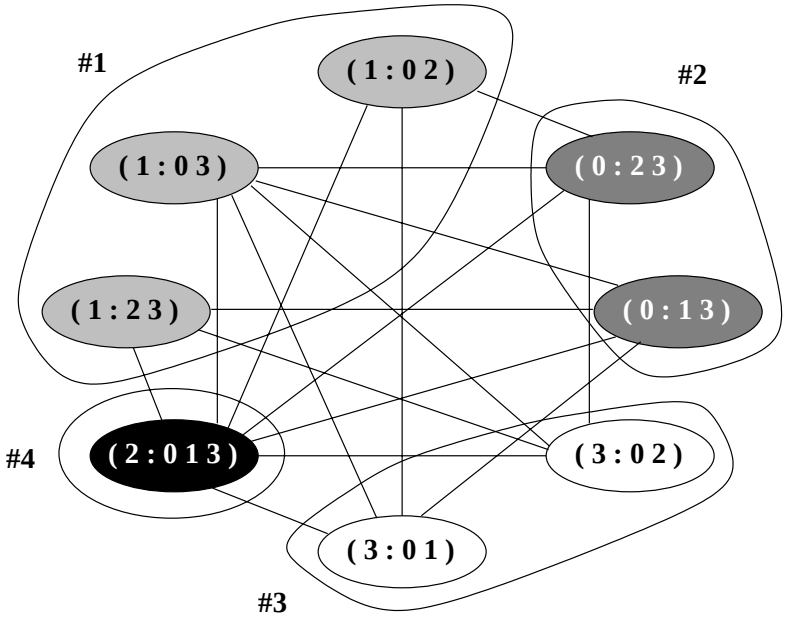


Figura 4.9: Classes de PDs geradas pela coloração (DK15).

compatibilidade com as outras PDs que possuem essa cor. Constrói-se uma função Booleana que retorna se uma PD passada como parâmetro pode ou não pertencer a classe associada a uma cor também passada por parâmetro. Essa função será chamada pelo algoritmo de coloração a cada vez que este julgar que uma certa cor pode pertencer a tal vértice respeitando as restrições impostas pelas arestas do grafo.

Colore-se, portanto, o grafo de incompatibilidade de PDs passando a função construída como parâmetro. Denominamos a coloração que exige uma segunda restrição além daquela imposta pelas arestas do grafo como uma *pseudo-coloração*. O resultado da pseudo-coloração garante que para cada classe, existe uma PD que a representa sem necessidade de fazer qualquer particionamento. Isso ocorre pelo fato da função construída garantir que todas as PDs com a mesma cor (portanto da mesma classe) são compatíveis entre si n a n[8].

O resultado do algoritmo, ilustrado na Figura 4.10, no caso da FSM DK15, é coincidentemente o mesmo para os dois algoritmos. Detalhes dos algoritmos citados serão analisados na Seção 5.2.

[8] ver definição de compatibilidade n a n na Seção 4.2.

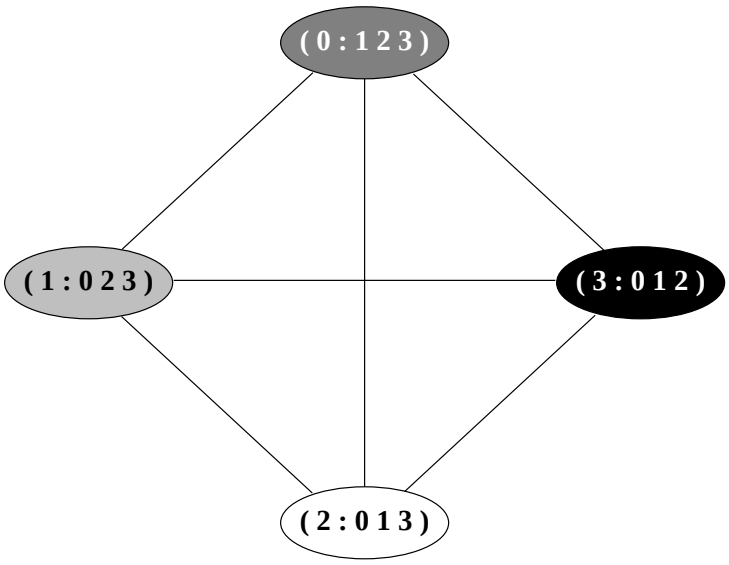


Figura 4.10: Conjunto final de PDs a serem satisfeitas (DK15).

Após se achar o conjunto de PDs que satisfazem a codificação, codificam-se os estados de maneira trivial. Para cada PD gerada, atribui-se uma coluna da codificação, gerando, por exemplo, uma codificação de comprimento 4 ilustrada na Tabela 4.2.3. A primeira coluna atribui ao estado 0 o código '0' e aos estados restantes o código '1' respeitando, portanto, à restrição imposta pela PD (0 : 1 2 3). As demais colunas seguem analogamente o mesmo raciocínio.

estado	código
0	0111
1	1110
2	1101
3	1011

Tabela 4.1: Exemplo de Codificação da FSM DK15. Cada bit da codificação representa a posição associado a uma PD utilizada na codificação.

CAPÍTULO 5

Implementação

A idéia básica do trabalho foi o desenvolvimento de algoritmos eficazes (tanto do ponto de vista computacional quanto da qualidade de resultados) que resolvam o problema de coloração de grafos assim como uma estrutura adequada de armazenamento e manuseio do grafo. Juntamente com essa idéia, necessitou-se implementar um estudo de caso para mostrar a qualidade e eficiência dos algoritmos e estruturas. Como visto nos Capítulos 1 e 3, o problema de coloração de grafos pode ser mapeado para diversos outros problemas em diversas áreas do conhecimento humano. Interessa aqui a aplicação a problemas de síntese de sistemas digitais VLSI.

As implementações foram desenvolvidas com dois objetivos principais:

- obter uma biblioteca de algoritmos e estruturas de dados associadas que resolvam o problema de coloração de grafos, de fácil manuseio, evitando a reimplementação de código para os problemas que podem ser mapeados parcial ou totalmente ao problema de coloração;

- possibilitar a personalização dos algoritmos de coloração e das estruturas de dados associadas de acordo com alguns critérios (detalhados mais adiante) para que o mapeamento seja o menos penoso possível;

Para tal, necessitou-se basicamente da execução cooperativa de três tarefas. Primeiramente, desenvolveram-se e implementaram-se algoritmos eficientes que resol-

vessem o problema de coloração de grafos de froma exata e/ou aproximada.

Em segundo lugar, selecionou-se um subconjunto de problemas em síntese VLSI a abordar, visando gerenciar, dentro do prazo, a escolha de algoritmos e heurísticas apropriadas. O problema de codificação Booleana restrita serviu de escolha inicial, justificado pelo familiaridade do autor com o problema. Atacaram-se outros problemas também, não se restrigindo somente ao problema de codificação Booleana.

Por fim, estudaram-se mais profundamente as características das instâncias típicas dos problemas a resolver, traçando um perfil das mesmas. Especificamente falando, será que as instâncias de grafos da maioria dos problemas em síntese VLSI possuem uma porcentagem de arestas elevada ou baixa? Ou seja, suponha-se que as arestas representem incompatibilidade dos objetos em um determinado problema, pergunta-se então: existem muitos pares de objetos incompatíveis[1]? Será que os grafos construídos possuem diversos vértices isolados? Será que a maioria dos grafos ou uma porção significativa destes são bipartidos, sendo, portanto bicoloríveis? Em casos assim, podemos escolher o algoritmo a ser utilizado, DSATUR, GULOSO e outros. Um bom levantamento dessas características a partir do estudo de benchmarks sintéticos e industriais influencia e contribui fortemente para a escolha de um bom subconjunto de problemas a abordar.

Essas e outras peculiaridades modificam, na prática, a estrutura e desenvolvimento de um algoritmo de coloração. Ou seja, um bom resultado de um algoritmo de coloração de grafos não significa necessariamente que o algoritmo em si seja eficiente para qualquer tipo de grafo. Grafos cordais, por exemplo, podem ser coloridos eficientemente em tempo polinomial e como demonstrado em [46] são utilizados freqüentemente em problemas em síntese de alto nível. Pode-se, portanto, construir diversas heurísticas que produzem bons resultados de acordo com o tratamento prévio dessas características particulares.

Outro problema tratado é aquele de como mapear eficientemente os problemas de síntese VLSI para o problema de coloração de grafos, sem que haja a necessidade

[1] esse número pode ser tão grande quanto o seu máximo de $\binom{n}{2}$, representando o número máximo de arestas que um grafo pode ter.

de entender a fundo o problema de coloração. A fim de obter melhores resultados, os algoritmos geralmente são adaptados, necessitando de pequenas, ou às vezes grandes, modificações no código fonte. O mapeamento deve ser fácil e não custoso, em termos de eficiência computacional, para que o mesmo possa fazer a diferença. Este mapeamento está intimamente ligado com a estrutura básica utilizada para a representação do grafo e seu manuseio. Algoritmos com boa eficiência em termos de tempo de execução, como o de Culberson[19], são extremamente difíceis de serem integrados para não dizer inviáveis se o usuário/programador não entender a fundo a implementação dos mesmos. A presente proposta tenta minimizar o custo desse mapeamento, disponibilizando, além de algoritmos eficientes, uma estrutura básica mais amigável, baseada nas estruturas da biblioteca de algoritmos LEDA[42].

Gerou-se, portanto, um pacote de algoritmos de coloração e de estruturas básicas associadas facilmente aplicáveis a uma ampla gama de problemas. Esse pacote de algoritmos e estruturas de dados, associados em uma biblioteca, facilita a integração de problemas diversos com o problema de coloração de grafos de forma prática e pouco penosa, descartando a necessidade de modificar extensamente algoritmos existentes, como descrito, por exemplo, em [11, 14, 18, 50, 55].

A seguir, apresenta-se a biblioteca desenvolvida, citando suas características principais, a integração com outros algoritmos e seu manuseio. A Seção 5.1.1 explica o funcionamento interno da biblioteca e de suas rotinas. Encontra-se também, na Seção 5.1.3, uma análise comparativa dos algoritmos implementados com os algoritmos já existentes. Por fim, é detalhada a implementação do estudo de caso escolhido explicando suas peculiaridades.

5.1 Biblioteca de Algoritmos GraphCol

A biblioteca foi desenvolvida na linguagem C/C++ em ambiente UNIX tanto em SunOS Solaris quanto em Linux. Decidiu-se utilizar essa plataforma e linguagem devido à fácil integração com os diversos algoritmos em síntese lógica em VLSI, majoritariamente desenvolvidos nas mesmas plataformas. Outra motivação foi a

utilização do paradigma orientado a objetos que permite uma ótima modularização da construção do software, de portabilidade (compilável em Linux, SunOS, Solaris e DOS/Windows) e de manutenção do software.

As principais características da biblioteca são:

- utiliza o paradigma orientado a objetos, acrescentando maior grau de abstração na interação com outros programas;
- utiliza as classes da biblioteca LEDA[42] para armazenar as estruturas internas como arrays, listas e filas entre outros;
- possui duas classes como estrutura básica de armazenamento que implementam os algoritmos de coloração:
  - GRAPHCOL → armazena os vértices e arestas no formato NODE e EDGE da LEDA e implementa todos os algoritmos disponíveis;
  - GRAPHCOL → classe derivada de GRAPHCOL e é utilizada para parametrizar o grafo com informações nos vértices e manuseá-los;
- é facilmente compilável e ligável através de uma interface com apenas um cabeçalho de definições (header) `_graphcol.h` e bibliotecas (libraries) `libcol.a` e `libcol.so`;
- possui um help on-line acessível através do comando `man` no UNIX ou através de um browser leitor de formato HTML.

Além da biblioteca, implementou-se um programa teste para cada algoritmo implementado. Cada programa executa de forma *stand-alone*, recebendo um grafo de um arquivo ASCII no formato DIMACS[2] e devolvendo na saída uma coloração válida. Estes programas foram utilizados na comparação com outros algoritmos tornando-se úteis para a execução de benchmarks.

A Figura 5.1 resume graficamente a estrutura básica da biblioteca implementada.

[2] ver Apêndice na página 85.

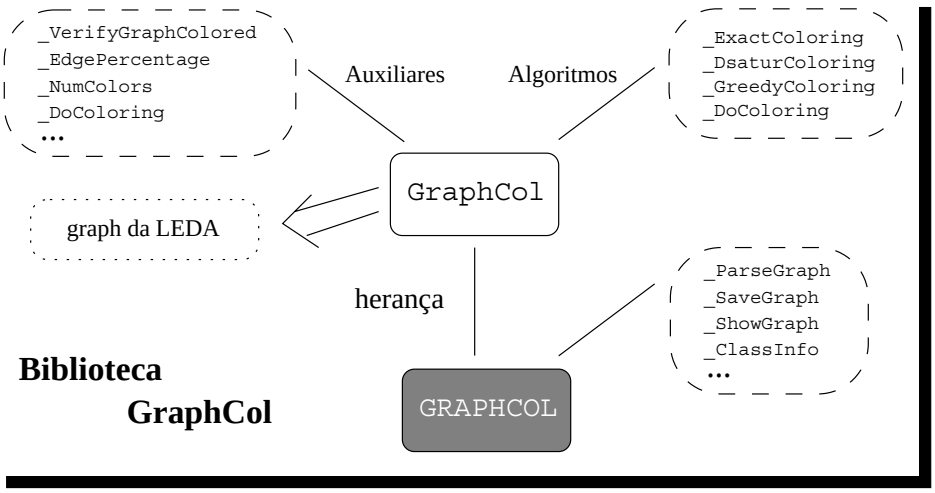


Figura 5.1: Estrutura básica da biblioteca implementada.

5.1.1 Detalhes da Implementação da Biblioteca

Os algoritmos descritos na Seção 3.3 foram implementados através de programas computacionais. Os algoritmos são:

Exato → este algoritmo testa todas as possibilidades de conjunto de cores a ser atribuído aos vértices de tamanho k com k variando de 1 a $|V|$; O algoritmo pára quando encontrar uma coloração válida tendo k como o número de cores utilizadas.

Guloso → este algoritmo colore o primeiro vértice com a cor 1 e colore os outros vértices sucessivamente com a mínima cor possível (cor de menor número) a ser atribuída;

Dsatur → este algoritmo baseia-se em um prévio ordenamento de vértices antes de começar a coloração propriamente dita; o algoritmo leva em conta o *grau de saturação dos vértices* a medida de sua coloração;

Como ferramentas de programação utilizou-se o compilador **GNU g++**[3] e as bibliotecas de estruturas de dados leda[42], ferramentas publicamente disponíveis

[3]http://www.gnu.org

para uso em ambiente acadêmico. Como ferramenta de análise de resultados utilizou-se os próprios recursos da linguagem de programação **C++** para medir o tempo de processamento utilizado.

A biblioteca possui as classes principais GRAPHCOL e GRAPHCOL como descritas anteriormente. A implementação através de classes possibilita a criação de diversas instâncias de grafos distintos, permitindo, por exemplo, colorir cada um com um algoritmo diferente ao mesmo tempo. No caso de outras implementações como [19], a estrutura básica utilizada para o armazenamento dos grafos é capaz de manter *somente um* grafo de cada vez necessitando limpar a estrutura global e reconstruir o grafo.

Métodos da classe GRAPHCOL

- Algoritmos:
  - _ExactColoring() ↪ executa o algoritmo EXATO de coloração;
  - _GreedyColoring() ↪ executa o algoritmo GULOSO de coloração;
  - _DsaturColoring() ↪ executa o algoritmo original de coloração DSATUR, implementado *ipsis litteris* [6];
  - _DoColoring() ↪ verifica, de acordo com alguns critérios, qual é o melhor algoritmo a ser executado para o determinado grafo armazenado; o critério utilizado neste caso foi o número de vértices combinado com a percentagem de arestas; Através dos resultados dos benchmarks obtidos determinou-se um limite máximo do número de vértice para a execução do algoritmo de coloração EXATA[4]; caso o número de vértices seja elevado, este método chama o algoritmo DSATUR;
- Auxiliares:
  - _VerifyGraphColored() ↪ verifica se a atual coloração é válida; retorna verdadeiro caso ela seja, falso caso contrário;

[4] o número ficou entre 10 vértices nao importando a porcentagem de arestas e 13 vértices para grafos com porcentagem de arestas abaixo de 50%.

  - _EdgePercentage() ↪ retorna a porcentagem de arestas do grafo armazenado;
  - _NumColors() ↪ retorna o número de cores utilizado pela coloração atual;

- Parametrizáveis pelo usuário:
  - _SetCanColor() ↪ método que armazena que função será chamada *antes* de um vértice receber uma determinada cor; a função indica se o vértice *pode* ou *não* receber essa cor de acordo com critérios estabelecidos pelo usuário; esta função é de grande utilidade para casos como a implementação do Algoritmo de Satisfação_Simultânea descrito na página 50;
  - _SetUpdateInfoClass() ↪ método que armazena que função será chamada *após* a coloração de um vértice com uma determinada cor; essa função possibilita a atualização da informação que cada classe de mesma cor armazena e que será utilizada pelo usuário através da função atribuída em _SetCanColor();

Métodos da classe GRAPHCOL

- Entrada e Saída:
  - _ParseGraph() ↪ lê, de um arquivo ou da entrada padrão, um grafo no formato ascii da dimacs[5] e o armazena internamente; _ParseGraph() chama internamente _ParseAsciiGraph();
  - _SaveGraph() ↪ grava em um arquivo ou na saída padrão, o grafo no formato ascii da dimacs;
  - _SetInput() ↪ indica de que entrada _ParseGraph() deve ler o grafo;
  - _ShowGraph() ↪ apresenta o grafo através de uma interface gráfica; a interface disponibiliza funções para visualização e impressão do grafo

[5] ver Apêndice na página 85.

úteis para impressão de relatórios e controle do processo de coloração dentro de um sistema mais complexo;

- Manipulação de vértices e arestas:
  - NEW_NODE(), NEW_EDGE(), DEL_NODE(), DEL_EDGE(), FIRST_NODE(), FIRST_EDGE(), SUCC_NODE(), SUCC_EDGE()
- Auxiliares:
  - CLEAR() $\hookrightarrow$ limpa o grafo excluindo todos seus vértices, arestas e informações associadas à coloração;
  - _CLASSINFO() $\hookrightarrow$ retorna a informação armazenada por _SETUPDATEINFOCLASS() associada a uma determinada classe;

Ainda existem alguns operadores especiais como FORALL_COLORS(i, G) que iterativamente percorre todas as cores do grafo G associando, a cada passo, i com a cor corrente. Faz-se acesso aos vértices atráves de outro operador, nomeado de OPERADOR [ ], que devolve uma lista encadeada de vértices que foram coloridos com uma determinada cor. O Programa 5.1.1 exemplifica esses operadores especiais.

Programa 5.1.1 Exemplo de utilização dos operadores especiais

```
...
forall_colors( i, G )              // 'i' recebe a cor
   forall_nodes( n, G[i] )         // 'n' recebe cada vertice da
                                   // lista devolvida por 'G[i]'
      cout << "Vertice: " << n;     // Imprime o vertice 'n'
...
```

Essas funções aplicam-se tanto à classe GRAPHCOL quanto a classe GRAPHCOL.

O *manual da biblioteca* GRAPHCOL, em anexo, descreve, em mais detalhes, cada método acima, exemplificando cada um deles. O formato DIMACS para reprensentação de grafos também está contido no Apêndice na página 85.

Um dos trabalhos futuros é gerar, como saída, um grafo no formato do programa **xvcg**[39]. Este programa é um excelente visualizador e manipulador de grafos, desde organogramas pequenos até grafos com mais de 5000 vértices utilizados cientificamente. Suas funções de desenho de grafos e de zoom são excelentes e auxiliaram na depuração do programa e na criação de exemplos para o presente trabalho.

5.1.2 Usabilidade da Biblioteca

A biblioteca implementada possui um arquivo *header* em C++, que é o arquivo base para sua compilação, nomeado `_graphcol.h`. Este arquivo contém todas as definições das classes implementadas e dos seus respectivos algoritmos e métodos. Esses últimos são compilados e ligados em uma biblioteca para facilitar o reuso da mesma em diversos programas. A biblioteca gerada pode ser *estática* ou *dinâmica* correspondendo aos arquivos `libcol.a` e `libcol.so` respectivamente. Procedimentos de instalação da biblioteca estão contidos no *manual da biblioteca* GraphCol encontrados em anexo.

Utiliza-se a biblioteca *estática* para obter maior rapidez na performance da implementação pois o código da biblioteca é incluido no código final do programa ligado com ela. Uma ligação dinâmica caracteriza-se por colocar chamadas especiais no código do programa ligado para suas funções, não aumentando o código do programa final mas sacrificando levemente a velocidade dos algoritmos. As diferenças são mínimas, mas optou-se por realizar das duas formas para que o usuário decida o que é melhor para seu tipo de aplicação.

Para a compilação de um programa com a biblioteca, basta incluir o header `_graphcol.h` no programa. O Programa 5.1.2 exemplifica uma pequena utilização da biblioteca.

O Programa 5.1.2 inclui primeiramente o header `_graphcol.h` para que as definições da biblioteca possam ser compiladas. Instancia-se um grafo g da classe GRAPHCOL parametrizando-a com o tipo *int* (inteiro). Esse comando cria um grafo que contém informações do tipo *inteiro* nos seus vértices. Automaticamente quando se cria um vértice, deve-se dizer qual a informação que o mesmo receberá,

Programa 5.1.2 Exemplo de utilização da biblioteca GRAPHCOL.

```
#include <iostream.h>
#include <_graphcol.h>      // inclui header da classe

main()
{
   GRAPHCOL< int > g;
   int i;
   node n;

   g.new_edge( g.new_node( 1 ), g.new_node( 2 ) );

   g._DoColoring();

   forall_colors( i, g ) {
      cout << "Cor " << i << '\n';
      forall( n, g[i] )
         cout << "Vertice " << g[n] << '\n';
   }
}
```

parametrizando o método NEW_NODE() como descrito na linha seguinte. Colore-se o grafo utilizando o método _DOCOLORING() que escolherá qual o melhor algoritmo a ser executado para este tipo de grafo. Especificamente neste caso, o método direcionará o controle do programa para o algoritmo EXATO pois o número de vértices é bem reduzido. Finalmente imprime-se na tela as cores utilizadas para a coloração do grafo, cada cor com seus respectivos vértices. Utiliza-se a função de iteração FORALL_COLORS explicada anteriormente através do Programa 5.1.1.

O uso da biblioteca, como visto acima, é fácil para qualquer programador, seja amador ou experiente. A compilação e a ligação também são passos simples. Basta que, na compilação, o header `_graphcol.h` esteja no *caminho de procura* do compilador e na ligação, que a biblioteca `libcol.a` ou `libcol.so` esteja no *caminho de procura* do ligador e que se acrescente o parâmetro **-lcol** no comando de ligação. Exemplo:

```
# make
```

```
gcc -I/usr/local/include -c exemplo.c
ld -L/usr/local/lib -lcol exemplo.o -o exemplo
```

A primeira linha, após o comando, compila o programa `exemplo.c` assumindo que o arquivo `_graphcol.h` esteje em `/usr/local/include` e a segunda linha liga o código objeto `exemplo.o` e gera o programa `exemplo` utilizando o parâmetro `-lcol` e assumindo que `libcol.a` ou `libcol.so` esteje em `/usr/local/lib` por exemplo.

O *manual da biblioteca* GraphCol apresenta mais exemplos de programas que utilizam a biblioteca e uma explicação mais detalhada da compilação e ligação da própria biblioteca e de programas exemplos que a utilizam.

5.1.3 Análise de performance da Biblioteca

Basicamente, foram realizadas dois tipos de análise. A primera consistiu na comparação dos algoritmos desenvolvidos, exato, guloso e dsatur entre si. A segunda na comparação do algoritmo dsatur implementado com outras implementações encontradas. Todos os testes foram realizados numa máquina *UltraSparc I 170* da Sun Microsystems de 167Mhz com 128MB de RAM e 256MB de memória virtual.

Utilizou-se grafos randômicos de 10 a 5000 vértices, cada um com 20, 50 e 75% de arestas no grafo. Para tal, utilizou-se o programa *generator*[6] de Joe Culberson. O programa *generator* gera grafos, levando em conta a porcentagem de arestas fornecida, no formato ascii ou binário da dimacs, mesmo formato utilizado na leitura da biblioteca GraphCol.

Os grafos com 5000 vértices e 50% de arestas e os grafos de 4000 e 5000 vértices com 75% de arestas provocaram falta de memória na execução dos algoritmos. Esses grafos são muito grandes, requerendo muita memória para seu armazenamento. Os resultados gerados pelos grafos com menos de 200 vértices não são exibidos, visto que suas execuções são praticamente instantâneas e não informam dados interessantes nem relevantes à comparação. Omite-se também a informação sobre os grafos com

[6] este e outros programas encontram-se em http://web.cs.ualberta.ca/~joe/Coloring/

75% de arestas pois apresentaram comportamento muito semelhante aos de 50%. Outro fator para essa omissão é que não encontramos nenhum grafo com 75% de arestas de utilidade em nossos estudos de caso.

Comparou-se os algoritmos EXATO, GULOSO e DSATUR. Verificou-se que o algoritmo EXATO é inviável para grafos acima de 10 vértices, consumindo muito tempo de CPU. O algoritmo executou instantâneamente para os grafos de 10 vértices e mais de 5 dias para achar uma coloração exata do grafo de 20 vértices com porcentagem de arestas de 25%.

A Figura 5.2 mostra a comparação dos algoritmos GULOSO e DSATUR em termos de tempo de execução (medindo somente o tempo de CPU utilizado para a coloração) e em termos de cores utilizadas para grafos com 50% de arestas[7].

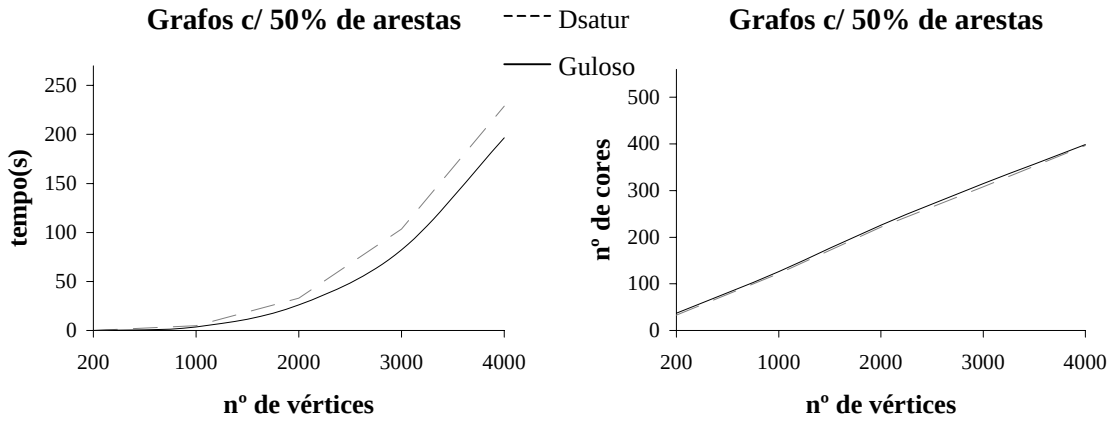


Figura 5.2: Análise dos algoritmos DSATUR vs GULOSO.

Pode-se constatar que os dois algoritmos possuem a mesma ordem de grandeza em relação ao tempo de CPU. Observa-se que o algoritmo GULOSO é ligeiramente mais rápido que o algoritmo DSATUR. Isso deve-se a dois procedimentos realizados pelo algoritmo de DSATUR: a ordenação inicial de vértices e o critério de seleção do próximo vértice a colorir que atrasam um pouco o processo de coloração. No algoritmo GULOSO, os vértices do grafo são coloridos na mesma ordem em que foram lidos, não estabelecendo nenhuma ordenação nem critério de seleção. Em relação

[7] Os gráficos de grafos com 25% e 75% de arestas são muito semelhantes ao gráfico de 50%.

à qualidade de resultado, o algoritmo de DSATUR é sempre superior para qualquer porcentagem de arestas e para qualquer número de vértices. A diferença na qualidade dos resultados não é grande, mas é sistemática para todos os grafos testados. O algoritmo de DSATUR, por conseguinte, apresenta um melhor compromisso de *desempenho por qualidade* comparado ao algoritmo GULOSO.

A Figura 5.3 mostra a comparação do algoritmo de DSATUR da biblioteca GraphCol com o algoritmo de DSATUR de Culberson[8] em termos de tempo de execução e qualidade de resultado respectivamente.

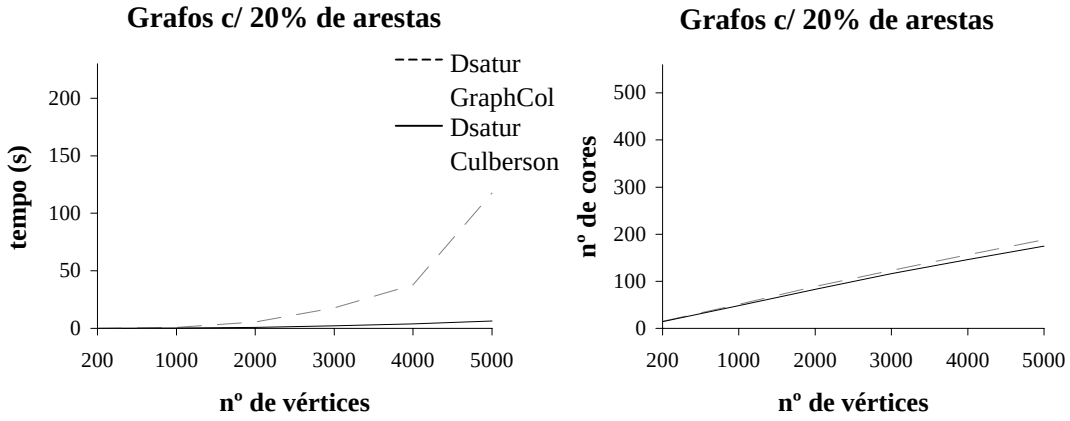


Figura 5.3: Análise dos algoritmos DSATUR GraphCol vs DSATUR de Culberson.

Observa-se que o algoritmo de Culberson é superior na velocidade de execução enquanto que o número de cores utilizadas cresce na mesma ordem de grandeza nos dois algoritmos.

Culberson realizou pequenas modificações ao algoritmo original de DSATUR proposto por Brélaz em [6]. As modificações que Culberson realizou são da mesma natureza que outros algoritmos realizaram ao se basear no algoritmo de DSATUR. Basicamente, modificam-se o ordenamento inicial de vértices, importante para uma boa coloração [19], e a decisão da escolha do vértice a ser colorido quando há um empate entre dois vértices que possuam o mesmo grau de saturação (vide descrição do algoritmo DSATUR, na página 32).

[8] ver nota de rodapé na página 65.

A superioridade, em termos de tempo de execução, do algoritmo de Culberson deve-se às modificações acima descritas e à estrutura de dados utilizadas no armazenamento do grafo. O armazenamento dos vértices e das arestas é estático, possibilitando um acesso mais rápido às informações mas por outro lado consumindo *mais memória*, em casos de grafos pequenos, que uma implementação dinâmica como a desenvolvida neste trabalho. Uma desvantagem da implementação estática, neste caso, é a necessidade da atribuição de um valor máximo ao número de vértices a nível de compilação, tornando-se tendioso o seu uso por outros programas. Uma vantagem dos algoritmos desenvolvidos, em relação aos encontrados, é que eles estão encapsulados como métodos de classes, facilitando assim a criação de diversas instâncias do problema no mesmo programa.

Como conclusão, a presente implementação é mais flexível e adaptável que a de Culberson, embora sacrificando a velocidade de execução. Uma implementação de heurísticas de ordenamento similares a de Culberson pode equalizar as curvas de número de cores $\times$ número de vértices de ambas implementações e não foi realizada devido à escassez de tempo.

5.2 Codificador ASS†UCE baseado em Coloração

Na Seção 4.2.3, apresentou-se o embasamento teórico necessário ao compreendimento do funcionamento dos algoritmos de satisfação desenvolvidos.

Implementou-se os Algoritmos 4 e 5 descritos nas páginas 49 e 5 respectivamente. Utilizou-se o software ASS†UCE versão 1.1 [10] para a inclusão desses algoritmos aproveitando todos os outros algoritmos e estruturas necessárias para a implementação de um codificador Booleano.

O ambiente ASS†UCE contém uma classe chamada ASSTUCE responsável por todo o funcionamento do ambiente. Essa classe contém vários outros objetos de outras classes inclusive objetos que representam a *geração de restrições* e *satisfação das mesmas* considerados e citados na página 48 como os dois passos fundamentais da Codificação Booleana Restrita. Utilizou-se a classe responsável pela *satisfação*

de restrições para a implementação dos dois algoritmos desenvolvidos no presente trabalho

Criou-se, portanto, um método na classe de *satisfação de restrições* chamado SATISFY_USING_GRAPHCOL ao contrário do método SATISFY que executa o algoritmo baseado em uma estrutura algébrica utilizando matrizes esparsas [9]. Adicionou-se ao software os arquivos `graphsat.h` e `graphsat.cc` que implementam esse método e outras funções auxiliares associadas.

Esse novo método aproveita as estruturas internas de representação de *pseudo-dicotomias* e cria um grafo a partir dela, passando esse a ser a nova representação das PDs e também das restrições impostas agora pelas arestas. A representação em forma de grafo fica mais fácil de ser manipulada pois não é necessário ter duas estruturas, uma para a representação de todas as PDs e outra para a representação de suas incompatibilidades e/ou compatibilidades.

O fluxo de controle desse método, ou do novo processo de satisfação, pode ser controlado através de parâmetros passados ao programa, ou seja, pode-se escolher, por exemplo, uma opção em que o programa imprima o grafo antes de processar a coloração ou se quiser, após a coloração para verificar se as restrições estão corretamente inseridas ou corretamente respeitadas respectivamente. As opções de codificação restrita *total* e *parcial* também são acionadas através dessa parametrização. O manual do codificador ASS†UCE, encontrado em anexo, apresenta mais explicações sobre a usabilidade do software detalhando seus possíveis parâmetros, opções de entrada e saída e utilização em outros ambientes como SIS[45].

5.2.1 Análise de desempenho do Codificador ASS†UCE

Realizou-se vários testes comparativos entre o novo codificador ASS†UCE que utiliza os algoritmos baseados em coloração de grafos e o codificador antigo baseado nas estruturas algébricas utilizando matrizes esparsas. Chamaremos de ASS†UCE GRAPHCOL e ASS†UCE ORIGINAL daqui em diante para indicar as versões do algoritmo de satisfação utilizado.

A Tabela 5.2.1 lista as máquinas de estados finitas (FSMs) utilizadas como ben-

chmarks para a realização das comparações. Todas as máquinas foram extraídas dos benchmarks da MCNC[54].

A comparação foi feita entre o codificador ASS†UCE ORIGINAL e o ASS†UCE GRAPHCOL. Utilizou-se o algoritmo de SATISFAÇÃO_SIMULTÂNEA para a realização dos testes, visto que este obtém uma performance não comparável ao algoritmo de SATISFAÇÃO_SERIAL devido a complexidade deste último.

A Tabela 5.2 mostra o quadro comparativo entre os algoritmos ASS†UCE Original e ASS†UCE GRAPHCOL.

Percebeu-se a partir dos benchmarks recolhidos que alguns resultados da execução do Algoritmo ASS†UCE GRAPHCOL não estavam corretos. Após profundo estudo para achar qual seria o problema já que a formulação teórica está comprovadamente correta, descobriu-se um erro no algoritmo básico do ASS†UCE em alguns casos especiais (o que não diretamente relacionado com o mapeamento para um problema de coloração, mas sim com a geração de restrições). Este erro, resumindo informalmente, encontra-se no relaxamento de restrições [8, 9] realizada pelo Algoritmo de geração de restrições interno ao ASS†UCE. Algumas restrições associadas à minimização de estados[9] não são unificadas com as restrições associadas à codificação de estados. Por essa razão, o Algoritmo ASS†UCE Original trata discretamente essas restrições de minimização em seu algoritmo de satisfação, produzindo, portanto, uma codificação válida.

Porém, ambos os Algoritmos SATISFAÇÃO_SERIAL e SATISFAÇÃO_SIMULTÂNEA dependem das restrições geradas pelo gerador de restrições do ASS†UCE para realizar a satisfação das mesmas. Não é possível, portanto, obter uma codificação válida a partir dos algoritmos de satisfação implementados já que as restrições não estão completas. Exemplos são os casos `ex2` e `ex5` onde o comprimento de código obtido pelo Algoritmo ASS†UCE GRAPHCOL é extreamente inferior proporcionalmente ao comprimento obtido pelo Algoritmo ASS†UCE Original. Foi a partir desses exemplos que esse erro foi identificado.

Entretando, pode-se concluir que os algoritmos implementados são eficientes e

[9] restrições de fechamento, ver [8].

que as restrições que faltam ser representadas não influênciariam no tempo de execução dos algoritmos somente aumentando um pouco o comprimento de código da máquina como se pode ver pela Tabela 5.2. Uma outra observação que pode ser extraída da execução desses benchmarks é a relação entre o número de arestas existentes a partir da geração de restrições com o tempo de coloração. Quanto maior o número de arestas no grafo de incompatibilidade de PDs, maior tempo de CPU será gasto para a coloração do mesmo. Exemplos são as máquinas S510 e SCF que obtém um aumento significativo na execução do algoritmo de codificação baseado em coloração.

A re-escrita da geração de restrições no algoritmo básico do ASS†UCE será certamente um trabalho futuro por dois motivos principais:

- para resolver o erro supra-citado e conseguir portanto aproveitar os algoritmos baseados em coloração;
- por ser o *gargalo* do ASS†UCE em termos de execução;

O segundo motivo foi obtido através da ferramenta **gprof** que possibilita obter uma descrição detalhada do comportamento do programa. O número de vezes que uma função foi chamada e o tempo que esta demorou são apenas algumas medidas que a ferramenta nos proporciona. Abaixo segue o resultado de uma das saídas do algoritmo ASS†UCE executando a maior máquina disponível com 218 estados.

```
.
.
.
index % time    self  children    called     name
                0.00  105.74       1/1           _start [2]
[1]     81.0    0.00  105.74       1         main [1]
                0.00  105.73       1/1           Asstuce::Execute(void) [3]
                0.00    0.01       1/1           Asstuce::Asstuce(int, char **) [120]
                0.00    0.00       1/1           Asstuce::~Asstuce(void) [749]
-----------------------------------------------
                                                  <spontaneous>
[2]     81.0    0.00  105.74                 _start [2]
                0.00  105.74       1/1           main [1]
-----------------------------------------------
                0.00  105.73       1/1           main [1]
[3]     81.0    0.00  105.73       1         Asstuce::Execute(void) [3]
                0.00  105.66       1/1           Asstuce::launch_execution(void) [5]
                0.00    0.07       1/1           Asstuce::get_FSM_from_file(void) [58]
-----------------------------------------------

.
.
.

-----------------------------------------------
              22.75   73.30   35881/35881       greedy_satisfier::greedy_satisfier(...) [6]
```

```
[7]     73.6   22.75   73.30   35881         greedy_satisfier::insert_seed(int, my_int_set const &, int) [7]
                0.47   31.30 1019523/1019523     my_int_set::operator|(my_int_set const &) [8]
               17.28    0.00 76558517/101457141     my_int_set::member(int) const [10]
                5.16    0.00 16383951/16451664     leda_h_array<int, leda_h_array<int, ... [16]
                3.58    0.00 2772463/2811308      my_int_set::my_int_set(int, int) [19]
                0.06    3.37  159100/159100       my_int_set::operator&(my_int_set const &) [20]
                3.16    0.00 9111239/9111239      leda_h_array<int, leda_dictionary< ... [22]
                2.00    0.00 9184919/9184919      leda_h_array<int, leda_h_array<int, ... [24]
                1.12    0.63 2772463/2880446      my_int_set::insert(int) [25]
                1.53    0.00 1031525/3416419      my_int_set::my_int_set(my_int_set const &) [17]
                1.24    0.00 4322225/4344578      leda_h_array<int, leda_dictionary<my_int_set, int> *>::operator[](int const &) [27]
                0.83    0.00 2802553/2829087      leda_dictionary<my_int_set, int>::next_item(ab_tree_node *) const [29]
                0.62    0.00 2582829/2582829      leda_dictionary<my_int_set, int>::lookup(my_int_set const &) const [32]
                0.54    0.00 1629568/1651921      leda_dictionary<my_int_set, int>::first_item(void) const [33]
                0.39    0.00 2234152/2238333      leda_dictionary<my_int_set, int>::key(ab_tree_node *) const [36]
                0.02    0.00   23467/23467        leda_dictionary<my_int_set, int>::insert(my_int_set const &, int const &) [81]
                0.00    0.00   56724/56724        leda_dictionary<my_int_set, int>::size(void) const [138]
                0.00    0.00   19286/19286        leda_dictionary<my_int_set, int>::del(my_int_set const &) [140]
                0.00    0.00    5460/5460         leda_dictionary<my_int_set, int>::inf(ab_tree_node *) const [150]
                0.00    0.00    5460/5460         leda_dictionary<my_int_set, int>::change_inf(ab_tree_node *, int const &) [149]
-----------------------------------------------
.
.
.
```

Percebe-se pelos dados extraídos pelo **gprof** que a função que mais ocupa tempo de CPU é a função INSERT_SEED() responsável pela construção interna das PDs *sementes* e por 73.6% do tempo total de execução da codificação. Pode-se concluir, portanto, que esse é um motivo forte para a re-escrita de parte da geração de restrições do ASS†UCE.

FSM	# de estados	# de produtos
dk15	4	32
lion	4	11
mc	4	10
tav	4	49
train4	4	14
bbtas	6	24
s27	6	34
beecount	7	28
dk14	7	56
dk27	7	14
dk17	8	32
ex6	8	34
shiftreg	8	16
ex5	9	32
lion9	9	25
bbara	10	60
ex3	10	36
ex7	10	36
train11	11	25
s386	13	64
ex4	14	21
dk512	15	30
mark1	15	22
bbsse	16	56
cse	16	91
sse	16	56
s208	18	153
s420	18	137
ex2	19	72
keyb	19	170
ex1	20	138
s1	20	107
tma	20	44
pma	24	56
s820	25	232
s832	25	245
dk16	27	108
styr	30	166
sand	32	184
tbk	32	1569
s510	47	77
planet	48	115
planet1	48	115
s1488	48	251
s1494	48	250
scf	121	166
s298	218	1096

Tabela 5.1: FSMs extraídas dos benchmarks da MCNC.

FSM	número de estados	produtos	**Asstuce Original**		**Asstuce GraphCol**	
			Compr. Código	Tempo	Compr. Código	Tempo
dk15	4	32	2	0,02	2	0,00
lion	4	11	2	0,03	2	0,00
mc	4	10	2	0,02	2	0,00
tav	4	49	2	0,02	2	0,01
train4	4	14	2	0,02	2	0,00
bbtas	6	24	3	0,06	3	0,00
s27	6	34	3	0,05	4	0,01
beecount	7	28	2	0,05	2	0,01
dk14	7	56	4	0,07	3	0,01
dk27	7	14	3	0,04	4	0,01
dk17	8	32	5	0,08	4	0,02
ex6	8	34	5	0,13	4	0,01
shiftreg	8	16	4	0,07	4	0,01
ex5	9	32	3	0,04	1	0,00
lion9	9	25	4	0,15	2	0,01
bbara	10	60	4	0,08	4	0,01
ex3	10	36	4	0,05	2	0,01
ex7	10	36	4	0,05	3	0,00
train11	11	25	4	0,11	2	0,01
s386	13	64	7	0,30	7	0,04
ex4	14	21	4	0,13	5	0,04
dk512	15	30	6	0,27	5	0,05
mark1	15	22	5	0,28	5	0,02
bbsse	16	56	7	0,25	6	0,02
cse	16	91	6	0,45	5	0,04
sse	16	56	7	0,26	6	0,03
s208	18	153	6	0,41	6	0,05
s420	18	137	6	0,44	6	0,06
ex2	19	72	10	0,20	2	0,01
keyb	19	170	7	0,44	7	0,04
ex1	20	138	8	0,64	6	0,04
s1	20	107	6	0,34	7	0,09
tma	20	44	8	0,75	6	0,07
pma	24	73	7	0,72	7	0,15
s820	25	232	7	0,86	6	0,11
s832	25	245	7	0,87	6	0,11
dk16	27	108	9	1,85	6	0,20
styr	30	166	9	1,03	7	0,17
sand	32	184	6	0,59	8	0,51
tbk	32	1569	11	4,66	7	0,21
s510	47	77	6	1,22	10	3,39
s1488	48	251	15	6,10	8	0,92
s1494	48	250	12	9,07	9	0,84
scf	121	166	8	14,55	15	77,04
s298	218	1096	16	183,84	16	38,53

Tabela 5.2: Tabela comparativa entre ASS†UCE Original vs ASS†UCE GraphCol.

CAPÍTULO 6

Conclusões

O trabalho desenvolvido consistiu em um estudo teórico sobre o problema de coloração de grafos. Optou-se por abordar o problema de coloração de vértices de um grafo, deixando de lado o problema de coloração de arestas, menos citado na literatura e de aplicação igualmente menos ampla. A procura por algoritmos eficientes e que fossem fáceis de utilizar na integração com outros algoritmos foi um ponto chave no desenvolvimento do trabalho propiciando o surgimento da biblioteca de algoritmos e estruturas de dados associadas para a resolução do problema de coloração de grafos.

A biblioteca de algoritmos de coloração e estruturas de dados associados foi desenvolvida e implementada em ambiente UNIX utilizando a linguagem C++ como descrito no capítulo anterior. As estruturas de dados desenvolvidas possuem uma grande facilidade de uso inclusive para programadores com pouca experiência em programação Orientada a Objetos na qual baseou-se a implementação. Essas estruturas estão baseadas nas classes da biblioteca LEDA[42], disponível livremente em domínio público para uso acadêmico. Tentou-se ao máximo seguir os padrões impostos pela estrutura das classes LEDA por dois motivos principais:

- para facilitar o uso integrado das diversas classes existentes na biblioteca LEDA e das classes desenvolvidas na biblioteca GraphCol;

- para facilitar uma futura integração dos algoritmos desenvolvidos nos pacotes originais da biblioteca LEDA;

Atingui-se um nível fácil de utilização da biblioteca após a implementação da mesma, ou seja, um simples exemplo é suficiente para ilustrar o funcionamento básico de sua utilização ainda que contenha algumas peculiaridades. O objetivo importante do trabalho foi alcançado pela implementação das classes da biblioteca GRAPHCOL que facilitam o emprego de modelagem de problemas baseados em coloração de grafos. Essa estrutura utilizada permite uma boa flexibilidade na aplicação da modelagem de outros problemas não diretamente mapeáveis para coloração de grafos. Um exemplo bem factível é o estudo de caso implementado (software ASS†UCE) cujo problema pode ser resolvido parcialmente através de coloração de grafos. Nesta implementação utilizou-se de alguns artifícios, como imposição de uma segunda restrição ao algoritmo de coloração, para conseguir um eficiente mapeamento. Artifícios estes contidos na biblioteca que foram implementados e colocados de maneira modular visando futura expansão.

A biblioteca implementada está sendo disponibilizada publicamente para uso acadêmico[1]. Uma página da biblioteca está em construção e pode ser visitada através do endereço `URL: http://www.inf.pucrs.br/~madeira/softwares/GraphCol`.

No que tange ao estudo de caso, a versão nova do ASS†UCE, chamada de ASS†UCE GRAPHCOL, demonstra que o estudo de caso escolhido para mapeamento para coloração de grafos conduz a uma solução mais eficiente que o algoritmo original, uma vez resolvidos os problemas de implementação pendentes. Estes problemas são oriundos da descoberta de um erro no algoritmo de geração de restrições na implementação original do ASS†UCE. Este erro, todavia, não afetava o funcionamento do algoritmo original pois este resolvia este problema transparentemente no algoritmo de satisfação de restrições, não dividindo corretamente um módulo de outro, o que seria o correto.

Acredita-se que este trabalho possa ter uma continuação pelo próprio autor ou por qualquer outro acadêmico que se interessar pelo assunto abordado no presente

[1] sem fins lucrativos

trabalho. Contatos com o autor pode ser obtido através de mensagem eletrônica para o endereço:

madeira@kriti.inf.pucrs.br

Apêndice

Objetivos e Cronograma de Atividades proposto

Replicou-se aqui os Objetivos e o Cronograma de Atividades descrito na proposta de trabalho afim de proporcionar um melhor acompanhamento do trabalho desenvolvido em TC1 e TC2.

Objetivos

Para melhor descrever os objetivos do trabalho, estes foram divididos em duas partes: *estratégicos* e *específicos*.

Objetivos Estratégicos

- dominar alguns conteúdos relevantes da teoria de grafos: coloração; grafos k-coloríveis; grafos k-críticos; número cromático; coloração mínima; grafos bipartidos; planaridade; cliques; ciclos;
- dominar aspectos essenciais de complexidade de algoritmos: problemas tratáveis, intratáveis; classe P e NP; problemas NP-completos, NP-hard;
- dominar as técnicas algorítmicas mais recentes de resolução **exata** e **aproximada** do problema de coloração de grafos;
- melhorar o desempenho de métodos de solução de problemas VLSI desenvolvidos pelo grupo GAPH, utilizando os algoritmos a serem desenvolvidos;
- formar um especialista capaz de resolver problemas de coloração de grafos e de mapear problemas diversos para o de coloração;

Objetivos Específicos

- desenvolver um algoritmo que implemente uma solução exata do problema de coloração de vértices;
- desenvolver um ou mais algoritmos que implementem soluções heurísticas eficientes de coloração de vértices;
- desenvolver um pacote de algoritmos (biblioteca em ambiente UNIX) que contenha os algoritmos desenvolvidos. Esse pacote deverá conter uma interface adequada de integração com outros programas assim como uma documentação clara e objetiva;
- modelar um problema de síntese VLSI (provavelmente em síntese lógica) para o problema de coloração de grafos e avaliar seu desempenho e eficiência através da comparação de resultados com abordagens anteriores;

Cronograma de Atividades

O cronograma geral de atividades pode ser dividido nas seguintes tarefas:

1. Estudo sobre conceitos em teoria de grafos e mais especificamente em coloração de grafos;
2. Estudo de algoritmos *exatos* e *aproximados* e de técnicas de resolução do problema de coloração de grafos;
3. Estudo (visando implementação) de problemas em síntese VLSI mapeáveis para o problema de coloração de grafos;
4. Implementação de um algoritmo exato de coloração de grafos;
5. Levantamento de características particulares dos problemas VLSI para modelização dos algoritmos heurísticos;

6. Implementação de diversos algoritmos heurísticos comparando-os uns com os outros;
7. Testes e avaliação de desempenho dos algoritmos implementados;
8. Documentação das técnicas e dos algoritmos desenvolvidos;
9. Escrita do relatório final de TC1 e TC2;
10. Publicação de resultados;

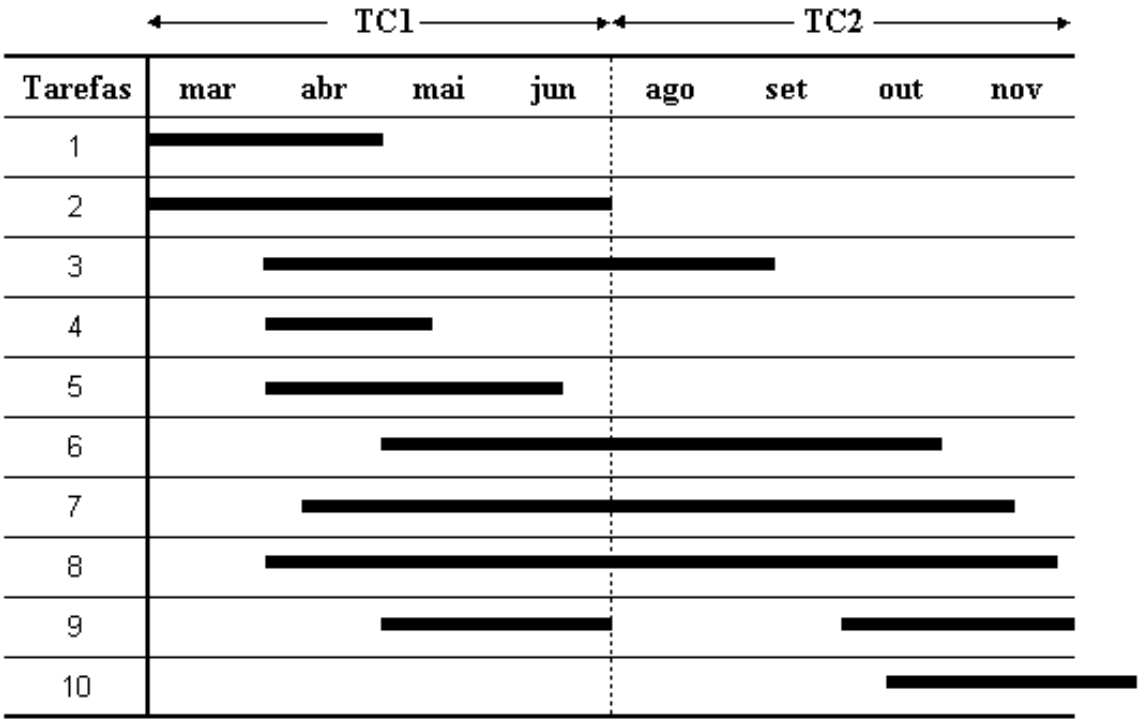


Figura 6.1: Cronograma de execução de tarefas

A Tabela 6 ilustra a distribuição das tarefas de acordo com o tempo de realização das disciplinas *Trabalho de Conclusão I* (TC1) e *Trabalho de Conclusão II* (TC2).

Formato ASCII da DIMACS

Clique and Coloring Problems Graph Format

This paper outlines a suggested graph format. If you have comments on this or other formats or you have information you think should be included, please send a note to `challenge@dimacs.rutgers.edu`.

Introduction

One purpose of the DIMACS Challenge is to ease the effort required to test and compare algorithms and heuristics by providing a common testbed of instances and analysis tools. To facilitate this effort, a standard format must be chosen for the problems addressed. This document outlines a format for graphs that is suitable for those looking at graph coloring and finding cliques in graphs. This format is a flexible format suitable for many types of graph and network problems. This format was also the format chosen for the First Computational Challenge on network flows and matchings.

This document describes three problems: unweighted clique, weighted clique, and graph coloring. A separate format is used for satisfiability.

File Formats for Graph Problems

This section describes a standard file format for graph inputs and outputs. There is no requirement that participants follow these specifications; however, compatible implementations will be able to make full use of DIMACS support tools. (Some tools assume that output is appended to input in a single file.)

Participants are welcome to develop translation programs to convert instances to and from more convenient, or more compact, representations; the Unix **awk** facility is recommended as especially suitable for this task.

All files contain ASCII characters. Input and output files contain several types of *lines*, described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

Input Files

An input file contains all the information about a graph needed to define either a clique problem or a coloring problem. Some information may be included that is not relevant to one problem (for instance, node weights are not needed for coloring problem) so that information may be ignored.

In this format, nodes are numbered from 1 up to n. There are m edges in the graph.

Files are assumed to be well-formed and internally consistent: node identifier values are valid, nodes are defined uniquely, exactly m edges are defined, and so forth. A input checker will be made available to ensure compatibility with this standard.

- **Comments.** Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character **c**.

```
c This is an example of a comment line.
```

- **Problem line.** There is one problem line per input file. The problem line must appear before any node or arc descriptor lines. For network instances, the problem line has the following format.

  ```
  p FORMAT NODES EDGES
  ```

  The lower-case character `p` signifies that this is the problem line. The `FORMAT` field is for consistency with the previous Challenge, and should contain the word "edge". The `NODES` field contains an integer value specifying n, the number of nodes in the graph. The `EDGES` field contains an integer value specifying m, the number of edges in the graph.

- **Node Descriptors.** For this Challenge, a node descriptor is required only for the weighted clique problem. These lines will give the weight assigned to a node in the clique. There is one node descriptor line for each node, with the following format. Nodes without a descriptor will take on a default value of 1.

  ```
  n ID VALUE
  ```

  The lower-case character `n` signifies that this is a node descriptor line. The `ID` field gives a node identification number, an integer between 1 and n. The `VALUE` gives the objective value for having this node in the clique. This value is assumed to be integer and can be either positive or negative (or zero).

- **Edge Descriptors.** There is one edge descriptor line for each edge the graph, each with the following format. Each edge (v, w) appears exactly once in the input file and is not repeated as (w, v).

  ```
  e W  V
  ```

  The lower-case character `e` signifies that this is an edge descriptor line. For an edge (w, v) the fields `W` and `V` specify its endpoints.

- **Optional Descriptors.** In addition to the required information, there can be additional pieces of information about a graph. This will typically define the parameters used to generate the graph or otherwise define generator–specific information. The following list may be added to as interesting problem generators are decided on:

  – Geometric Descriptors. One common method to generate or display graphs is to have the nodes be embedded in some space and to have the edges be included according to some function of the distance between nodes according to some metric. The node information can be defined by a dimension descriptor and a vertex embedding descriptor.

  ```
  d DIM METRIC
  ```

  is the dimension descriptor. DIM is an integer giving the number of dimensions of the space, while METRIC is a string representing the metric for the space. METRIC is a string that can take a number of forms. Lp (i.e. L1, L2, L122, and so on) denotes the ℓ_p norm where the distance between two nodes embedded at $(x_1, x_2, \ldots, x_d)$ and $(y_1, y_2, \ldots y_d)$ is $(\sum_{i=1}^{d} |x_i - y_i|^p)^{1/p}$. The string LINF is used to denote the ℓ_∞ norm. L2S denotes the squared euclidean norm (which can be less susceptible to computer–differences in round–off and accuracy issues).

  ```
  v  X1  X2  X3  . . . XD
  ```

  The lower-case character `v` signifies that this is a vertex embedding descriptor line. The fields `X1, X2 . . .XD` give the d coordinate values for the vertex. Note that these lines must appear after the `d` descriptor.

  – Parameter Descriptors. The parameter descriptors are used to give other information about how the graph was generated. The lines are generator–specific, and as such it is not expected that most codes will use most (or any) of them. They are included only to aid those codes specifically

designed to attack specially structured problems. The general form of the parameter descriptor is:

```
x PARAM VALUE
```

The lower-case character `x` signifies that this is a parameter descriptor line. The PARAM field is a string that gives the name of the parameter, while the VALUE field is a numeric value that gives the corresponding value. The following PARAM values have been defined:

PARAM	Description
MINLENGTH	(Geometric Graphs) Edge included only if length greater than or equal to VALUE
MAXLENGTH	(Geometric Graphs) Edge included only if length less than or equal to VALUE

Note that this information is in addition to the required edge descriptors.

Output Files

Every algorithm or heuristic should create an output file. This output file should consist of one or more of the following lines, depending on the type of algorithm and problem being solved.

- **Solution Line**

  ```
  s TYPE SOLUTION
  ```

  The lower–case character `s` signifies that this is a solution line. The `TYPE` field denotes the type of solution contained in the file. This should be one of the following strings: "col" denotes a graph coloring, "clq" denotes a maximum weighted clique, and "cqu" denotes a maximum unweighted clique (one that has ignored the `n` descriptor lines).

  The `SOLUTION` field contains an integer corresponding to the solution value. This is the clique size for unweighted clique, clique value for weighted clique, or number of colors used for graph coloring.

- **Bound Line**

  ```
  b BOUND
  ```

  The lower–case character `b` signifies that this is a bound on the the solution. The `BOUND` field contains an integer value that gives a bound on the solution value. This bound is an upper bound on the maximum clique value for cliques and weighted clique and a lower bound on the number of colors needed for coloring the graph.

- **Clique Line**

  ```
  v V
  ```

  The lower-case character `v` signifies that this is a clique vertex line. The `V` field gives the node number for the node in the clique. There will be one clique line for each node in the clique.

- **Label Line**

  ```
  l V N
  ```

  The lower-case character `l` signifies that this is a label line, generally used for graph coloring. The `V` field gives the node number for the node in the clique while the `N` field gives the corresponding label. There will be one label line for each node in the graph.

Referências Bibliográficas

[1] K. Appel and W. Haken. Every planar map is four colorable. *American Mathematical Society Bulletin*, 82(5):711–712, 1976.

[2] K. Appel and W. Haken. Every planar map is four colorable part I: Discharging. *Illinois Journal of Mathematics*, 21:429–490, 1977.

[3] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable part II: Reducibility. *Illinois Journal of Mathematics*, 21:491–567, 1977.

[4] B. Berger and J. Rompel. A better performance guarantee for approximate graph coloring. *Algorithmica*, 5(4):459–466, 1990.

[5] Béla Bollobás. The chromatic number of random graphs. *Combinatorica*, 8(1):49–55, 1988.

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

[7] J. Randall Brown. Chromatic scheduling and the chromatic number problem. *Management Science*, 19(4):456–463, December, Part I 1972.

[8] N. L. V. Calazans. *State minimization and state assignment of finite state machines: their relationship and their impact on the implementation.* PhD thesis, Université Catholique de Louvain, Laboratoire de Microélectronique, Louvain-la-Neuve, Belgium, 1993.

[9] Ney L. V. Calazans. Boolean constrained encoding: a new formulation and a case study. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 702–706, San Jose, November 1994.

[10] Ney L. V. Calazans and André D. Madeira. Asstuce - an exploratory environment for finite state machines. In *XXIII Conferencia Latinoamericana de Informática - CLEI*, pages 117–126, Valparaíso, Chile, November 1997.

[11] Tam-Anh Chu, Narayana Nani, and Clement K.C. Leung. An efficient critical race-free state assignment technique for asynchronous finite state machine. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 2–6, 1993.

[12] V. Chvátal. Perfectly ordered graphs. In C. Berge and V. Chvátal, editors, *Topics on Perfect Graphs*, volume 21 of *Annals of Discrete Mathematics*, pages 63–65. North-Holland Publishing Co., 1984.

[13] M. J. Ciesielski, J.-J. Shen, and M. Davio. A unified approach to input-output encoding for FSM state assignment. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 176–181, San Francisco, CA, June 1991.

[14] Maciej J. Ciesielski and Saeyang Yang. PLADE: A two-stage pla decomposition. *IEEE Transactions on Computer-Aided Design*, 11(8):943–954, August 1992.

[15] Richard Cole and John Hopcroft. On edge coloring bipartite graphs. *SIAM Journal on Computing*, 11(3):540–546, August 1982.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT Electrical Engineering and Computer Science Series. McGraw-Hill Book Company, Cambridge, MA, 1990.

[17] O Coudert and C.-J. Richard Shi. Exact dichotomy-based constrained encoding. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors - ICCD*, October 1996.

[18] Olivier Coudert. A new paradigm for dichotomy-based constrained encoding. In *Proceedings of the ACM/IEEE European Design Automation & Testing Conference - DATE*, Paris, France, February 1998.

[19] Joseph C. Culberson. Iterated greedy graph coloring and the difficulty landscape. Technical Report TR 92-07, University of Alberta Department of Computing Science, Edmonton, Alberta, Canada T6G 2H1, 1992. ftp ftp.cs.ualberta.ca pub/TechReports.

[20] Celina M. H. de Figueiredo, João Meidanis, and Célia Picinin de Mello. Coloração em grafos. In *XVI Jornada de Atualização em Informática*, pages 39–83, Brasília, August 1997. Sociedade Brasileira de Computação.

[21] G. de Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, Inc., New York, NY, 1994.

[22] G. de Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, CAD-4(3):269–284, July 1985.

[23] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment, and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):13–27, January 1991.

[24] Stanley Fiorini and Robin J. Wilson. Edge-colourings of graphs. In Lowell W. Beineke and Robin J. Wilson, editors, *Selected Topics in Graph Theory*, chapter 5, pages 103–126. Academic Press, Inc., London, 1978.

[25] S. Fujita, T. Kameda, and M. Yamashita. A resource assignment problem on graphs. *Lecture Notes in Computer Science*, 1004:418, 1995.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, San Francisco, CA, 1979.

[27] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Acadmeic Press, New York, NY, 1980.

[28] G. R. Grimmett and C. J. H. McDiarmid. On colouring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77:313–324, 1975.

[29] Magnus M. Halldorsson. A still better performance guarantee for approximate graph coloring. *Information Processing Letters*, 45(1):19–23, January 1993.

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

[31] Ian Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, November 1981.

[32] F. K. Hwang. A modification to a decomposition algorithm of Gordon and Srikanthan. *IEEE Transactions on Computers*, 46(8):958–960, August 1997.

[33] Tommy R. Jensen and Bjarne Toft. *Graph Coloring Problems*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., 1995.

[34] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[35] D. Koenig, editor. *Theorie der endlichen und unendlichen Graphen*. Chelsea, New York, 1935.

[36] Marek Kubale. Graph coloring. In Allen Kent and James G. Williams, editors, *Encyclopedia of Microcomputers*, volume 8, pages 47–69. Marcel Dekker, Inc., New York, 1991.

[37] Marek Kubale and Boguslaw Jackowski. A generalized implicit enumeration algorithm for graph coloring. *Communications of the ACM*, 28(4):412–418, April 1985.

[38] C. Kuratowski. Sur les problèmes des courbes gauches en Topologie. *Fund. Math.*, 15:271–283, 1930.

[39] Iris Lemmke and Georg Sander. *The CVG Tool – A Visualization Tool for compiler graphs.* the Compare Consortium. Software package, available by ftp from *ftp.cs.uni-sb.de:/pub/graphics/vcg/.*

[40] Vahid Lotfi and Sanjiv Sarin. A graph coloring algorithm for large scale scheduling problems. *Computers and Operations Research*, 13(1):27–32, 1986.

[41] K. Menger. Zur allgemeine Kurventheorie. *Fund. Math.*, 10:96–115, 1927. (in german).

[42] Stefan Näher. LEDA *User Manual - Version 3.7.* Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998. http://www.mpi-sb.mpg.de/LEDA.

[43] N. Robertson, D. Sanders, and R. Thomas. The four-colour theorem. *Journal of Combinatorial Theory Series B*, 70:2–44, 1997.

[44] A. Salazar and R. V. Oakford. A graph formulation of a school scheduling algorithm. *Communications of the ACM*, 17(12):696–698, December 1974.

[45] Ellen M. Sentovich and et al. Sis: A system for sequential circuits synthesis. Memorandum no ucb/erl m92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[46] D. L. Springer and D. E. Thomas. Exploiting the special structure of conflict and compatibility graphs in high-level synthesis. In Satoshi Sangiovanni-Vincentelli, Alberto; Goto, editor, *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 254–257, Santa Clara, CA, November 1990. IEEE Computer Society Press.

[47] Jayme Luiz Szwarcifter. *Grafos e Algoritmos Computacionais.* Editora Campus Ltda., Rio de Janeiro, Brasil, 1984.

[48] James H. Tracey. Internal state assignment for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15(4):551–560, August 1966.

[49] S. H. Unger. *Asynchronous sequential switching circuits.* Wiley-Interscience – John Wiley & Sons, New York, NY, 1969.

[50] W. Wan and M. A. Perkowski. A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to FPGA mapping. In *Proceedings of the European Conference on Design Automation*, pages 230–237, Los Alamitos, CA, USA, September 7–10 1992. IEEE Computer Society Press.

[51] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its applications to timetabling problems. *The Computer Journal*, 10:85–86, 1967.

[52] Avi Wigderson. Improving the performance guarantee for approximate graph coloring. *Journal of the ACM*, 30(4):729–735, October 1983.

[53] Douglas Woodall. Improper colourings of graphs. In Roy Nelson and Robin J. Wilson, editors, *Graph Colourings*, Pitman Research Notes in Mathematics Series, pages 45–63. Longman Scientific & Technical, Longman house, Burnt Mill, Harlow, Essex, UK, 1990.

[54] Saeyang Yang. Logic synthesis and optimization benchmarks. Technical report, Microelectronics center of North Carolina, Research Triangle Park, NC, January 1991. Version 3.0.

[55] Saeyang Yang and Maciej J. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):4–12, January 1991.

[56] A. A. Zykov. On some properties of linear complexes. *Matematicheskii Sbornik*, 24:163–188, 1949. English Trans. Amer. Soc. Translation no. 79, 1952.

Índice Remissivo

Anexo