UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ANNE ROSE ALVES FEDERICI MARINHO

ALGORITMOS CERTIFICADORES E VERIFICADORES

Rio de Janeiro
2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ANNE ROSE ALVES FEDERICI MARINHO

ALGORITMOS CERTIFICADORES E VERIFICADORES

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Vinícius Gusmão Pereira de Sá

Rio de Janeiro
2015

CBIB Marinho, Anne Rose Alves Federici

Algoritmos Certificadores e Verificadores / Anne Rose Alves Federici Marinho. – 2015.

82 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2015.

Orientador: Vinícius Gusmão Pereira de Sá.

.

1. Algoritmos certificadores. 2. Algoritmos verificadores. 3. Complexidade de algoritmos. – Teses. I. Pereira de Sá, Vinícius Gusmão (Orient.). II. Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática. III. Título

CDD

ANNE ROSE ALVES FEDERICI MARINHO

Algoritmos Certificadores e Verificadores

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado em: Rio de Janeiro, _____ de _____ de _____.

_____

Prof. Dr. Vinícius Gusmão Pereira de Sá (Orientador)

_____

Prof. Dr. Jayme Luiz Szwarcfiter, Ph.D., UFRJ

_____

Profa. Dra. Lilian Markenzon, Ph.D., UFRJ

_____

Prof. Dr. Danilo Artigas da Rocha , D.Sc., UFF

Rio de Janeiro
2015

Aos meus pais, Alcideia e Luiz Carlos, que estiveram presentes em todos os momentos.

AGRADECIMENTOS

Agradeço ao meu marido, Paulo Marinho, que me incentivou ao recomeço. Obrigada pela paciência, pela força e principalmente pelo carinho e amor. Valeu a pena cada momento ausente. Essa vitória é nossa! Quero agradecer também aos nossos filhos, Eduarda e Danilo, que me proporcionaram momentos de lazer fazendo eu até esquecer das minhas ansiedades e angústias. Iluminaram meus pensamentos me levando a buscar mais conhecimentos. Vocês três são tudo pra mim!

Ao professor e orientador Vinícius Gusmão, por toda paciência e dedicação. Agradeço por transmitir seus conhecimentos, por fazer do meu trabalho uma experiência positiva e por ter confiado em mim. Tenho certeza que sem todo seu apoio e compreensão eu não teria chegado ao fim.

Agradeço aos meus pais e à minha sogra, que mesmo diante de muitas dificuldades não mediram esforços para cuidarem dos meus filhos nos meus momentos de ausência.

Às minhas irmãs, Lilian e Danielle, por sempre me lembrarem que eu não podia desistir.

À minha amiga Carolina Szkruc, pelas alegrias, tristezas e ansiedades compartilhadas. As trocas de mensagens entre um parágrafo e outro fizeram com que tudo ficasse mais leve.

Aos amigos e colegas do LC3, pelo incentivo e pelo apoio constante. Obrigada por tudo!

A todos os professores do Programa de Pòs Graduação em Informática, que foram tão importantes na minha vida acadêmica.

A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de estudos concedida.

Enfim, a todos que de forma direta ou indireta auxiliaram na concretização deste trabalho. O meu agradecimento!

RESUMO

Marinho, Anne Rose Alves Federici. **Algoritmos Certificadores e Verificadores**. 2015. 79 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Algoritmos certificadores são algoritmos que retornam não apenas a solução para o problema que se prestam a resolver, mas também um certificado, direta e eficientemente verificável, de que tal solução é correta. Nesta dissertação, argumentamos a favor do emprego de algoritmos certificadores para a solução computacional de problemas em geral, em detrimento a algoritmos clássicos, não-certificadores. O ponto central é que os primeiros estão protegidos contra possíveis — e, certamente, comuns — erros introduzidos inadvertidamente pelo programador, ao passo que os últimos, nesses casos, apresentariam respostas imprevisíveis, provavelmente incorretas e com efeitos indesejados. O objetivo desta pesquisa foi estudar o emprego de algoritmos certificadores, divulgando a técnica e estimulando seu uso. Foram considerados (analisados e implementados) quatro problemas: o problema do reconhecimento de grafos bipartidos e o do emparelhamento de cardinalidade máxima, ambos já descritos na literatura de algoritmos certificadores; o problema da seleção dos k menores elementos de uma lista (k-selection); e o problema do reconhecimento de grafos de disco unitário. Esses dois últimos problemas fogem intencionalmente do padrão do algoritmo certificador que provê um certificado que pode ser verificado de forma muita simples e eficiente, pois, no primeiro caso, o certificado é a própria saída do algoritmo, que se presta imediatamente à verificação (como ocorre, na prática, em muitas situações), e no segundo caso, a verificação não é formalmente eficiente (por demandar tempo exponencial), mas pode ser computacionalmente viável para instâncias pequenas, permitindo, por exemplo, ser usada em provas computacionais para teoremas.

Palavras-chave: Algoritmos certificadores, algoritmos verificadores, complexidade de algoritmos.

ABSTRACT

Marinho, Anne Rose Alves Federici. **Algoritmos Certificadores e Verificadores**. 2015. 79 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Certifying algorithms are those which return not only the solution to the problem they mean to solve, but also a correctness certificate, whose verification can be tackled in an efficient and straightforward way. In this dissertation, we argue for the use of certifying algorithms, rather than their classic, non-certifying counterparts, whenever possible. The central idea is that the former are shielded against implementation errors inadvertently introduced by the programmer, whereas the latter, in these cases, would helplessly present unforeseeable, incorrect outputs, probably with undesirable consequences. After a brief account of the theory of certifying algorithms, we look into three problems illustrative examples, highlighting the necessary care one should devote to the completeness of the verification phase. The goal of this research was to study the applicability of certifying algorithms, disclosing the technique and stimulating its use. We have considered (analysed and implemented) four problems: the recognition of bipartite graphs and the maximum-cardinality matching problem, both already tackled in the literature of certifying algorithms; the k-selection problem; and the recognition of unit disk graphs. The two latter problems intentionally deviate a little from the standard pattern of the certifying algorithm which provides a certificate that can be verified in a very simple and efficient way. In the first case, no certificate is actually called for, since the very output of the classic algorithm can be verified right away (as is the case in many practical situations); in the second case, even though the verification step cannot be undertaken in a formally efficient way (demanding exponential time), it can be computationally feasible for small instances, which shall make it suitable for theorem proofing.

Keywords: Certifying algorithm,verifiers algorithm, complexity algorithm.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

GDU	Grafo de disco unitário
CCTI	Cobertura por conjuntos de tamanho ímpar

SUMÁRIO

1 INTRODUÇÃO

A solução computacional de qualquer problema envolve a concepção e a programação de um algoritmo, isto é, de uma sequência bem definida de instruções a serem executadas pelo computador. Embora a corretude e a eficiência de um algoritmo possa ser garantida analiticamente, atestando-se que a solução por ele obtida é sempre correta, o mesmo não pode ser dito, em geral, sobre sua realização em uma linguagem qualquer. Com efeito, para que um algoritmo de fato resolva determinado problema, não é suficiente que ele funcione no papel — é crucial que ele seja corretamente traduzido para a linguagem de programação escolhida, isto é, que sua *implementação* esteja livre de erros. Provar que a implementação de um algoritmo — com todos os possíveis fluxos de execução e todas as estruturas de dados envolvidas, bem como eventuais recursão, paralelismo, etc. — está livre de erros é tarefa extremamente complexa, beirando a impossibilidade. Como garantir, portanto, que a solução apresentada por um algoritmo para uma instância qualquer de determinado problema está correta, se a exatidão da implementação do algoritmo não pode ser atestada? Tornar isto possível é o objetivo dos algoritmos ditos certificadores [18], que apresentam não apenas a solução de um problema, mas também um certificado da corretude da resposta, certificado esse que pode ser facilmente verificado.

A possibilidade de se verificar a resposta obtida para um problema (e a possibilidade de fazê-lo em tempo hábil) não é assunto novo. Por exemplo, a classe de complexidade **NP** para problemas de decisão (i.e., que admitem resposta SIM ou NÃO) é definida como sendo a classe dos problemas cuja resposta SIM pode ser verificada em tempo polinomial no tamanho da entrada. O ponto de nosso interesse, no entanto, é um pouco diverso. Não queremos provar que tal ou qual problema *admite* verificação eficiente para determinada resposta; queremos permitir que, qualquer que seja a resposta

obtida para o problema considerado (seja ele de decisão, otimização, localização, etc.), uma verificação computacional eficiente, simples e completa, cuja implementação admita uma prova formal de corretude (ou, o que é igualmente satisfatório para todos os fins práticos, uma implementação a respeito da qual seja irrazoável temer pela existência de erros ou vícios ocultos), possa ser de fato realizada. Indo mais além, nosso objetivo é o de recomendar que, tanto quanto seja possível fazê-lo, prefira-se a adoção de um algoritmo certificador para a solução de um problema, incorporando a produção dos certificados *e a verificação dos mesmos* ao próprio código do algoritmo, outorgando-lhe assim máxima confiabilidade.

Sullivan e Masson [21] foram precursores da ideia de certificação. Eles preconizavam que todo programa deveria deixar um rastro de informações que pudesse ser usado para atestar seu bom funcionamento, permitindo a verificação de erros a posteriori (auditoria), mas não imediatamente. Suas ideias foram aplicadas, inicialmente, para verificar o comportamento de estruturas de dados. Em um trabalho posterior [4], eles combinaram certificação com verificação formal. O termo "algoritmo certificador" foi usado pela primeira vez em 2006 por McConnell et al [14]. Antes, sem formalizar o conceito, Mehlhorn e Naher [20, 19] já haviam trabalhado com as ideias de verificação de resultados e de "programas de verificação".

Tanto quanto sabemos, não há ainda na literatura nenhuma recomendação explícita de que a própria verificação seja incorporada ao programa que irá resolver determinado problema, o que, de certa forma, enfraquece a ideia da certificação (pois, na prática, a verificação caberia ao *usuário final* do programa, que nem sempre disporá de meios de consumar a desejada verificação). Portanto, assim como, em Engenharia de Software, recomenda-se a incorporação de *unit tests* ao código principal de qualquer sistema, nossa recomendação é a de que, em Computação Científica, se volte à produção de certificados — *e à verificação automatizada desses certificados, incorporada aos próprios programas* — a mesma atenção exigida pelas boas práticas da Engenharia com relação aos testes de

software de nível industrial.

1.1 Organização da dissertação

Esta dissertação está dividida em sete capítulos, a saber: introdução; teoria de algoritmos certificadores; problema do reconhecimento de grafos bipartidos; problema do emparelhamento de cardinalidade máxima; $k-$Selection; problema de reconhecimento de disco unitário; conclusão. Após a breve introdução ao tema feita neste Capítulo 1, apresentamos os preceitos básicos sobre os algoritmos certificadores e verificadores no Capítulo 2, estabelecendo uma analogia com algumas técnicas conhecidas de testes em engenharia de *software*. Nos Capítulos 3 e 4, respectivamente, apresentamos os problemas do reconhecimento de grafos bipartidos e do emparelhamento de cardinalidade máxima em grafos, mostrando como podemos tornar os algoritmos clássicos em algoritmos certificadores, e exemplificando formas de procedermos à verificação das soluções. Nos Capítulos 5 e 6, descrevemos problemas que fogem intencionalmente do padrão do algoritmo certificador que permite, por meio de um certificado ou testemunha, uma verificação simples e eficiente, sendo, de todo modo, interessantes do ponto de vista de certificação e verificação. Primeiramente, o problema da seleção dos k menores elementos, para o qual um algoritmos verificador prescinde de certificados, sendo perfeitamente capaz de efetuar a verificação de forma simples e eficiente diretamente da resposta obtida. Finalmente, o problema do reconhecimento de grafos de disco unitário, em que o certificado não permite uma verificação formalmente eficiente, demandando tempo exponencial (o algoritmo mais eficiente que se conhece para resolver o problema demanda tempo duplamente exponencial), mas que, para certas instâncias pequenas, é computacionalmente viável, permitindo por exemplo a criação de provas computacionais para teoremas. No Capítulo 7, apresentamos as conclusões da dissertação.

2 ALGORITMOS CERTIFICADORES E VERIFICADORES

Algoritmos certificadores produzem como saída, além da solução para determinado problema, um *certificado* (ou *testemunha*) de que a solução encontrada está correta, isto é, que não foi comprometida, por exemplo, por um erro na implementação do algoritmo. Ao inspecionar a testemunha, devemos ser capazes de nos convencermos de que a saída está correta, ou de rejeitarmos a saída como errada (ou como inconclusiva, por não termos sido capazes de atestá-la). Desta forma, não é preciso confiar cegamente no resultado obtido, eliminando qualquer dúvida quanto à resposta apresentada.

Um algoritmo certificador recebe como entrada uma instância x, retornando como saída a resposta y e o certificado w. É possível, então, procedermos à verificação de que w de fato comprova que y é uma saída correta para a entrada x. O processo de verificação de w pode ser automatizado com um algoritmo *verificador*, isto é, um programa que, usando w, consegue provar a exatidão da resposta y para a entrada x, ou rejeitar y. Idealmente, o verificador admite implementação simples e eficiente. Uma prova formal da corretude da implementação de um algoritmo certificador pode estar fora de nosso alcance; porém, uma prova formal de corretude do algoritmo verificador pode ser viável [18]. Diante de uma verificação bem-sucedida, podemos ter plena confiança que a saída y é exata.

A Figura 2.1a mostra a entrada e a saída de um algoritmo convencional, não certificador, para calcular uma função $f(x)$. Não podemos ter certeza de que y é realmente igual a $f(x)$. A Figura 2.1b mostra a entrada e a saída de um algoritmo certificador, que calcula a saída y e a testemunha w. Tanto a saída do algoritmo certificador (resposta e certificado) quanto a instância original do problema são então submetidas ao algoritmo verificador, que, auxiliado por w, valida (ou não) a igualdade $y = f(x)$.

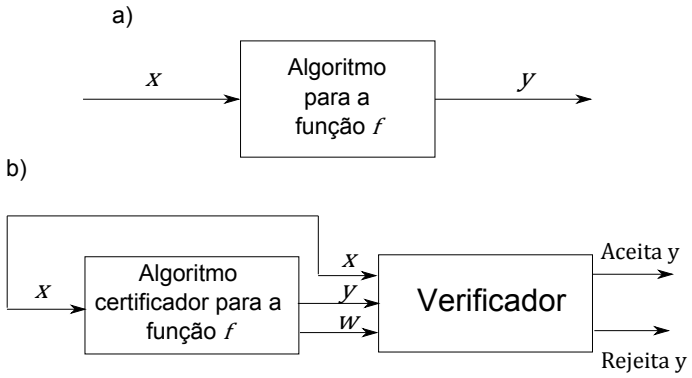

Figura 2.1: Comparação entre um algoritmo convencional e um algoritmo certificador para computar uma função f.

Um dos grandes problemas em engenharia de *software* é provar que um programa está correto [18]. *Testes unitários* (*unit tests*) [23] são programas escritos para testar um sistema de forma controlada, verificando funcionalidades específicas de unidades como métodos, funções, módulos ou classes. O nível de abstração desses testes depende do tipo de sistema que está sendo desenvolvido. Para a consecução destes testes, são empregados pares de entrada e saída (x_i, y_i) conhecidos para o problema. Desta forma, um programa é aceito se, para cada x_i na lista, o programa retorna y_i. A boa prática recomenda a cobertura de todas as linhas de código em testes unitários, e a incorporação desses testes ao próprio código do programa principal. Isto é, o programa passa a *conter seus testes*, sendo, por assim dizer, "auto-testado" automaticamente em tempo de compilação. Algumas objeções comuns contra os testes unitários são que eles podem revelar a presença de erros, porém não conseguem atestar sua completa ausência. Além disso, um programa só pode ser testado com respeito a entradas para os quais a saída correta já foi revelada por outros meios. É essencial, portanto, cuidado na hora de preparar os testes unitários, para que todos os (ou a maioria dos) possíveis casos de erros de implementação possam ser por eles capturados.

Quando resolvemos computacionalmente um problema fundamental, digamos, em Teoria dos Grafos, as análises de complexidade e de corretude dos algoritmos emprega-

dos são em geral feitas cuidadosamente. O mesmo cuidado, no entanto, não costuma ser visto com relação à implementação desses mesmos algoritmos. A implementação, quando chega a ser feita, frequentemente não é testada a contento. O fato de apresentarem a resposta correta para um pequeno número de instâncias (em testes rápidos feitos por um humano) não garante que a implementação esteja livre de falhas. Os algoritmos certificadores permitem incorporar à resposta fornecida uma maneira de verificar, computacionalmente, sua corretude, estabelecendo assim uma analogia ao software que se "auto-testa".

2.1 Certificação e verificação

Um bom certificado é aquele que admite um algoritmo verificador que seja conceitualmente mais simples do que o próprio algoritmo que o produz. Um bom algoritmo certificador é aquele que produz um bom certificado sem pagar por isso o preço de ser computacionalmente menos eficiente do que um algoritmo não-certificador para o mesmo problema.

McConnell, em [18] definiu três tipos de algoritmos certificadores: algoritmos fortemente certificadores, algoritmos certificadores (propriamente ditos) e algoritmos fracamente certificadores. Os algoritmos fortemente certificadores são o tipo mais desejável. Para cada entrada, é provado que o algoritmo funcionou corretamente ou que o usuário forneceu uma entrada incorreta, fora do domínio aceito pelo algoritmo; ele também retorna qual das duas alternativas ocorreu. Em outras palavras, para qualquer entrada x, ou ele produz uma testemunha mostrando que x não satisfaz a *pré-condição* (condições que devem ser atendidas para que o programa possa ser executado), ou ele produz uma saída y e uma testemunha que mostra que o par (x, y) satisfaz a *pós-condição* (condições que devem ter sido atendidas ao final da execução do programa).

Em outras situações, temos que nos contentar com menos. O algoritmo retorna apenas se a pré-condição foi violada ou se a pós-condição foi satisfeita. Nem sempre é possível indicar qual das duas alternativas ocorreu. Como exemplo, podemos citar a busca binária. Sua entrada é um número x e um vetor A de n números. A pré-condição afirma que A é ordenado em ordem crescente (ou não-decrescente). A saída é SIM, se x é pertence ao vetor, e é NÃO, se x não pertence ao vetor. O algoritmo certificador da busca binária não retorna qualquer erro na sua pré-condição.

Por último, um algoritmo certificador fraco, para entrada e saída (x, y) e certificado w é um algoritmo que termina sua execução em tempo finito para todas as entradas x que satisfazem a pré-condição. Para entradas que não satisfazem a pré-condição, ele pode parar ou não.

Um algoritmo certificador é dito eficiente se sua saída puder ser verificada por um algoritmo verificador, de tal modo que a complexidade assintótica tanto do certificador quanto do verificador sejam, no máximo, equivalentes à do algoritmo conhecido mais eficiente para o mesmo problema [14].

Evidentemente, todo o conceito de algoritmo certificador depende da capacidade de escrevermos os algoritmos verificadores corretamente. Ao projetarmos um algoritmo verificador, devemos observar os seguintes aspectos [18]:

- Corretude e completude: para toda instância respondida e certificada corretamente pelo algoritmo certificador, o algoritmo verificador deve atestar sua validade; para toda instância respondida incorretamente pelo algoritmo, o verificador deve rejeitar a resposta.

- Tempo: sempre que possível, o tempo de execução de um verificador deve ser linear no tamanho de sua entrada, ou seja, no tamanho da tripla (x, y, w), e jamais

excedendo assintoticamente a complexidade do próprio algoritmo que resolve o problema.

- Simplicidade: o verificador deve ser um algoritmo simples, cuja corretude lógica possa ser provada analiticamente e cuja implementação possa ser verificada, preferencialmente, por meio de simples inspeção de código.

McConnell, em [18] citou diversas vantagens de algoritmos certificadores e verificadores. Podemos destacar:

- Em um algoritmo certificador podemos ter certeza da saída retornada. Se o verificador aceitar como entrada (x, y, w), então podemos afirmar que a testemunha w prova que $f(x) = y$ e que w é um certificado correto, e se rejeitar (x, y, w), podemos afirmar que o algoritmo cometeu um erro, seja no cálculo da saída y, seja no cálculo do certificado w.

- Podemos realizar a verificação de *todas* as entradas. Quando falamos em testes de *software*, sabemos que os mesmos são executados apenas para as entradas para as quais já conhecemos a saída correta, e provam apenas a presença de erros, mas não a ausência deles. Ao implementarmos um algoritmo certificador, possibilitamos analisar qualquer entrada x, e não apenas as entradas para as quais a saída correta já seja conhecida.

- Os algoritmos certificadores e verificadores são resistentes a alterações. Se uma alteração for feita na implementação do algoritmo, pode-se analisar a verificação para algumas entradas típicas, já conhecidas. Por exemplo, se o verificador rejeitar a entrada (x, y, w) não aceitando a testemunha w que era aceita anteriormente, deve-se rever as alterações feitas; caso contrário, se apesar das alterações w for aceita, não é necessário se preocupar com as modificações realizadas, pois elas provavelmente não introduziram nenhum tipo de erro. Além disso, caso tenham introduzido erro,

a verificação de cada saída específica (executada durante o próprio uso pretendido do algoritmo) revelará sua existência, evitando que uma resposta errada seja aceita.

Os problemas que estudaremos a seguir são exemplos de como a certificação e verificação podem apoiar-se mutuamente.

3 PROBLEMA 1: RECONHECIMENTO DE GRAFOS BIPARTIDOS

O primeiro problema que abordaremos é de fato um exemplo fácil em Teoria dos Grafos. Trata-se do reconhecimento de grafos bipartidos, isto é, reconhecer se os vértices de um grafo G podem ser particionados em dois conjuntos, de forma que que cada aresta de G tenha suas extremidades em conjuntos distintos.

Um algoritmo não-certificador retorna um único bit (verdadeiro ou falso), informando se G é bipartido ou não.

Um algoritmo certificador, por outro lado, permite a verificação da corretude dessa resposta obtida, fazendo-a vir acompanhada de uma testemunha w. A testemunha para a resposta SIM consiste em uma partição dos vértices de G em dois conjuntos independentes (um conjuto independente não contém quaisquer dois vértices que sejam vizinhos no grafo), isto é, uma bicoloração para G. Diferentemente, a testemunha para o NÃO vem na forma de um ciclo de G de tamanho ímpar, isto é, uma sequência de vértices $v_0, v_1, v_2, \ldots, v_{2k+1}$, para k inteiro positivo, onde $v_{2k+1} = v_0$ e onde (v_j, v_{j+1}) é uma aresta de G para todo $0 \leq j \leq 2k$.

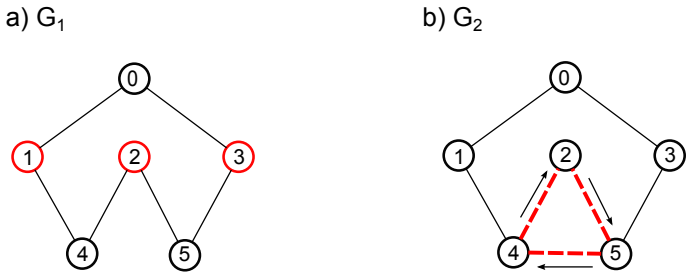

Figura 3.1: (a) Grafo bipartido. (b) Grafo não bipartido.

A imagem 3.1a mostra um grafo bipartido, onde os conjuntos são separados pela cor do vértice. Na Figura 3.1b, as arestas pontilhadas em vermelho apresentam um ciclo ímpar$\{2, 5, 4\}$.

A verificação, em cada caso, é bastante simples e direta. Uma bicoloração certifica, por definição, que o grafo é bipartido. A existência de um ciclo ímpar, por outro lado, certifica que o grafo não é bipartido. É interessante frisar que sequer precisaríamos *entender a razão* pela qual um grafo bipartido não pode conter um ciclo ímpar. Trata-se de um resultado conhecido, analiticamente *comprovado* (nesse caso, algo que pode ser encontrado em qualquer livro introdutório sobre Teoria dos Grafos). O que precisa ser feito, para garantir que o grafo dado é ou não é bipartido, é *validar* a resposta obtida, isto é, verificar se os dois conjuntos fornecidos de fato constituem uma bicoloração do grafo (no caso de uma resposta SIM), ou verificar se o ciclo ímpar apresentado como testemunha é de fato um ciclo ímpar do grafo. No primeiro caso, a verificação consiste em se examinar se os conjuntos dados são mesmo uma bipartição dos vértices de G, e se de fato todas as arestas de G incidem em vértices de conjuntos distintos daquela bipartição. No segundo caso, em que a testemunha é uma sequência de vértices que é alegadamente um ciclo ímpar de G, é preciso verificar se cada par de vértices na sequência apresentada consititui uma aresta de G, não esquecendo de verificar também que o tamanho do ciclo fornecido é ímpar!

3.1 Testes computacionais

O algoritmo certificador e o verificador para o reconhecimento de grafos bipartidos foram implementados para realização de testes. O grafo é armazenado como uma matriz de adjacências, em que a verificação da existência de uma aresta pode ser realizada em tempo constante. O código-fonte está listado no Apêndice A.

Utilizamos os grafos da Figura 3.1: o grafo G_1 para os testes que precisavam de um grafo bipartido como entrada, e o grafo G_2 para os testes que precisavam de um grafo não-bipartido como entrada.

Foram produzidas, propositalmente, entradas com erros para analisarmos a resposta do verificador acusando os erros que podem ser obtidos por "defeitos" na implementação do algoritmo.

1. Entrada: G_1
   Resposta: SIM
   Certificado: Cor 0: {0, 4, 5}; Cor 1: {1, 2, 3}
   Verificação ok! O verificador analisa o certificado, confirma a bicoloração e aceita a resposta para essa entrada.

2. Entrada: G_1
   Resposta: NÃO
   Certificado: ciclo 0, 1, 3
   A verificação falha, pois o certificado apresenta um ciclo ímpar que não existe no grafo de entrada. A aresta $(1, 3)$ não existe.

3. Entrada: G_1
   Resposta: SIM
   Certificado: Cor 0: {0, 4}; Cor 1: {1, 2, 3}
   A verificação falha. Apesar de apresentar uma bicoloração para o grafo, o verificador não pode confirmar a resposta, pois apresenta um vértice (no caso, o vértice 5) sem atrbuição de cor.

4. Entrada: G_1
   Resposta: SIM
   Certificado: Cor 0: {0, 4}; Cor 1: {1, 2, 3}; Cor 2: {5}

A verificação falha. O certificado apresenta uma cor a mais, logo não é uma bicoloração e não é possível afirmar que a resposta retornada pelo algoritmo está correta ou não.

5. Entrada: G_1
   Resposta: SIM
   Certificado: Cor 0: {0,1, 4, 5}; Cor 1: {2, 3}
   O verificador rejeita a resposta, pois o certificado retornado apresenta dois vértices adjacentes com a mesma cor.

6. Entrada: G_2
   Resposta: NÃO
   Cerificado: ciclo 4, 1, 0, 3, 5
   Verificação ok! O cerificador confirma a existência do ciclo ímpar no grafo e aceita a resposta.

7. Entrada: G_2
   Resposta: NÃO
   Certificado: 4,1,0,3,5,2
   O verificador rejeita a resposta, pois o ceritificado retornado é um ciclo de tamanho par. Desta forma, não pode garantir a resposta que o grafo não é bipartido.

8. Entrada: G_2
   Resposta: NÃO
   Certificado: ciclo 4, 1, 0, 3, 2
   Novamente o verificador rejeita a resposta. O ciclo ímpar apresentado como certificado contém uma aresta que não pertence ao grafo.

Com esses testes, pretendemos mostrar que a verificação precisa ser meticulosa. Analisamos diversos possíveis erros de implementação, incluindo tanto casos de respostas incorretas (com certificados, evidentemente, errados!) quanto de respostas corretas

com certificados que nada certificam (e que não podem, portanto, ser validados pelo verificador). É importante percebermos cada detalhe para que não sejam aceitas respostas incorretas, primando, dessa forma, pela completude do verificador.

4 PROBLEMA 2: EMPARELHAMENTO DE CARDINALIDADE MÁXIMA EM GRAFOS

Encontrar um emparelhamento de cardinalidade máxima é um problema clássico no estudo de algoritmos e otimização combinatória [1]. Um *emparelhamento* no grafo G é um conjunto de arestas de G no qual não há duas arestas compartilhando uma mesma extremidade.

Um *caminho* no grafo G é dado por uma lista de vértices $(v_1, v_2, \ldots, v_n)$ onde $n \geq 2$ e $(v_i, v_i + 1)$ é uma aresta para $1 \leq i < n$. O caminho é chamado de *simples* se nenhum vértice ocorre mais de uma vez na lista. Um *circuito* ou *ciclo* é um caminho $(v_1, v_2, \ldots, v_n, v_1)$ tal que $n \geq 3$ e $(v_1, v_2, \ldots, v_n)$ é um caminho simples [9].

Dado um emparelhamento M, vértices incidentes a alguma aresta de M são ditos *saturados*, sendo os demais vértices do grafo ditos *não-saturados* ou *livres*. Um emparelhamento é chamado *maximal* se não existe um outro emparelhamento no grafo que propriamente o contenha. Um emparelhamento *máximo* é um emparelhamento de cardinalidade máxima no grafo. Se todos os vértices do grafo são saturados em um emparelhamento M, então M é um emparelhamento *perfeito*. Evidentemente, todo emparelhamento perfeito é máximo [6].

Na Figura 4.1, o grafo G_1 ilustra um exemplo de grafo com emparelhamento M de cardinalidade 2 (as arestas emparelhadas são representadas por linhas vermelhas). Se adicionarmos qualquer outra aresta, deixamos de ter um emparelhamento; portanto M é maximal. Como não existe outro emparelhamento com cardinalidade maior, M é também máximo.

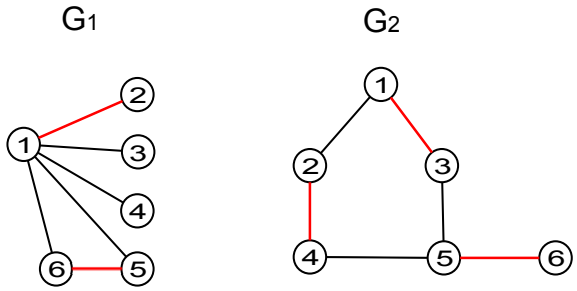

Figura 4.1: Emparelhamento máximo e emparelhamento perfeito.

O grafo G_2 ilustra um exemplo de grafo com um emparelhamento perfeito.

Um caminho $M-$*alternante* é um caminho simples cujas arestas se alternam entre M e $E-M$. Um caminho *aumentante* é um caminho alternante entre dois vértices livres, isto é, o primeiro e o último vértice do caminho $M-$alternante não são cobertos por M.

Dado um emparelhamento M e um caminho aumentante P é possível obter um emparelhamento M', fazendo $M' = (M \cup P) - (P \cap M)$, a diferença simétrica entre M e P. O caminho P é chamado de aumentante pois $|M'| = |M| + 1$.

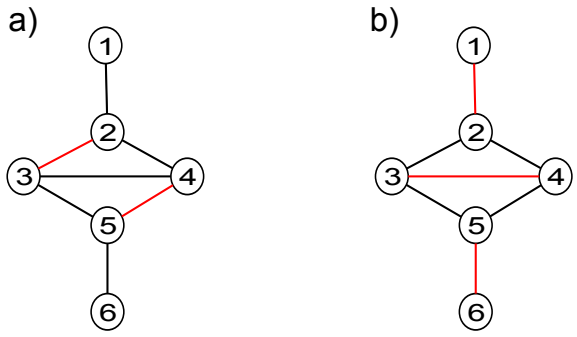

Figura 4.2: a) Emparelhamento. b) Emparelhamento máximo.

A Figura 4.2a mostra um grafo com emparelhamento de cardinalidade 2. Os vértices 1 e 6, são vértices livres e são os extremos do caminho aumentante $(1, 2, 3, 4, 5, 6)$. Ao trocarmos as arestas deste caminho obtemos o emparelhamento da Figura 4.2b.

Edmonds, em [8] escreveu o primeiro algoritmo eficiente para emparelhamento de cardinalidade máxima, com um tempo de execução $O(n^4)$. Ele introduziu o conceito de *flor* (um circuito de comprimento ímpar) e como manipulá-la para encontrar caminhos aumentantes de maneira eficiente.

4.1 Algoritmo de Edmonds

O algoritmo de Edmonds [8] está diretamente ligado ao conceito de grafos bipartidos. O algoritmo de emparelhamento máximo para grafos bipartidos é menos complexo do que para grafos genéricos. Os grafos bipartidos possuem algumas propriedades que facilitam a estrutura desse algoritmo. Inicialmente faremos uma breve descrição sobre o funcionamento de um algoritmo para emparelhamento máximo em grafos bipartidos, para em seguida explicarmos as modificações feitas por Edmonds para que seja encontrado o emparelhamento máximo em grafos gerais.

4.1.1 Algoritmo para emparelhamentos em grafos bipartidos

O algoritmo executa uma busca em um grafo bipartido. Essa busca é iniciada a partir de um vértice livre x e percorre o grafo com o objetivo de encontrar um outro vértice livre y para formar um caminho aumentante. Se esse vértice y for encontrado, então as arestas deste caminho são trocadas para que a cardinalidade do emparelhamento aumente. Porém, se a busca percorrer todo o grafo e não encontrar um outro vértice livre, significa que não existe um caminho aumentante partindo deste vértice livre x, não sendo possível aumentar o emparelhamento a partir de x. O vértice x é marcado como apagado, e o algoritmo começa uma nova busca a partir de um outro vértice livre z que não esteja apagado. Esta busca se repete até que não exista mais vértices livres não-apagados. Logo, se o algoritmo já não tem mais vértices livres para iniciar a busca, foi encontrado um

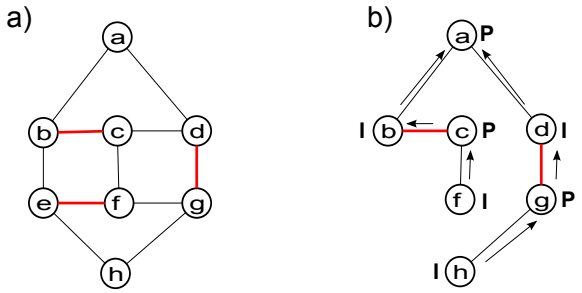

Figura 4.3: a) Grafo Bipartido. b) Árvore de busca.

emparelhamento máximo.

A cada busca, o algoritmo reduz o número de vértices livres não-apagados (isto ocorre quando um vértice livre é apagado, nos casos de busca que não foram bem-sucedidas) ou aumenta a cardinalidade do emparelhamento (caso seja encontrado um caminho aumentante).

A busca no grafo é executada através de uma árvore de busca em largura a partir do vértice livre. Os vértices visitados pela busca recebem um rótulo par ou ímpar. O rótulo par é dado se a distância em número de arestas entre os vértices for par e recebe o rótulo ímpar se a distância entre os vértices for de comprimento ímpar. Sempre que um vértice livre, diferente do vértice de partida, receber um rótulo ímpar, foi identificado um caminho aumentante.

A Figura 4.3a mostra um grafo bipartido e um emparelhamento (arestas destacadas em vermelho). Os vértices $\{a, c, e, g\}$ pertencem a um conjunto X e os vértices $\{b, d, f, h\}$ pertencem a um conjunto Y. As arestas ligam somente vértices do conjunto X com os vértices do conjunto Y. A Figura 4.3b apresenta o crescimento da busca em largura sobre o grafo bipartido da Figura 4.3a. Os vértices com rótulo par são representados pela letra P, os com rótulo ímpar pela letra I e as setas representam as marcações de predecessores. A raiz da árvore de busca é o vértice a. A busca se expande partindo

de a para os vértices b e d, pois não são vértices livres. Como os vértices b e d possuem rótulo ímpar, a ação a ser feita é estender a busca para os vértices c e g respectivamente. A partir do vértice c, a busca somente poderá ser expandida para o vértice f, pois o vértice d já foi visitado. A partir do vértice g é dado um rótulo ímpar ao vértice livre h, identificando o caminho aumentante (a, d, g, h). Este caminho é estabelecido pelas marcações de predecessores a partir de h até a raiz a.

Esse algoritmo encontra o emparelhamento máximo, pois os grafos bipartidos possuem rotulagem única dos vértices, ou seja, não é possível um vértice receber um rótulo par e também um rótulo ímpar durante uma busca. Nos grafos bipartidos se a raiz da árvore de busca estiver no conjunto A então todos os vértices de A receberão rótulos par e todos os vértices de B receberão rótulos ímpar. Porém nos grafos genéricos não é possível conseguir o mesmo, pois não podemos garantir a rotulação única dos vértices.

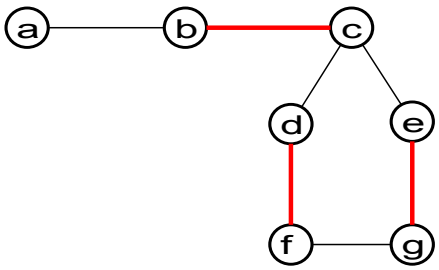

Figura 4.4: Grafo genérico.

A Figura 4.4 mostra um exemplo em que o vértice pode receber um rótulo par e também um rótulo ímpar. O vértice f pode receber um rótulo par através do caminho da busca (a, b, c, d, f) e pode receber um rótulo ímpar através do caminho da busca (a, b, c, e, g, f). Isto acontece quando podemos ligar um vértice à raiz através de dois caminhos com comprimentos distintos, um caminho de comprimento par e outro caminho de comprimento ímpar.

4.1.2 Algoritmo para emparelhamento máximo em grafos genéricos

Para encontrar o emparelhamento de cardinalidade máxima em grafos genéricos, Edmonds [8] introduziu o conceito de haste e flor, para atender ao problema da ambiguidade de rótulos em um grafo genérico. Apresentamos a seguir essas definições formais [5].

Definição 4.1. *Uma* haste *é um caminho alternado de comprimento par que parte da raiz da busca em largura r e vai até um vértice w. É permitido que r seja igual a w, onde dizemos que a haste é vazia.*

Definição 4.2. *Uma* flor *é um circuito de comprimento ímpar (não necessariamente induzida) que começa e termina no vértice w da haste e não tem nenhum outro vértice em comum com ela. O vértice w é a base da flor.*

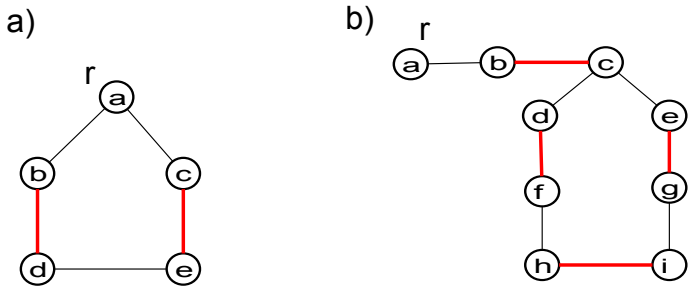

Figura 4.5: a) Flor com haste vazia. b) Flor com haste.

A Figura 4.5a representa uma flor com uma haste vazia e a Figura 4.5b representa uma flor com haste não vazia.

Dado um emparelhamento maximal M, observamos as seguintes propriedades da flor e sua haste:

1. Uma haste possui $2i + 1$ vértices e contém $i \geq 0$ arestas de M.

2. Uma flor possui $2j + 1$ vértices e contém $j \geq 1$ arestas de M. Tais arestas cobrem todos os vértices da flor exceto a base, a qual é um vértice com rótulo par.

3. Pelo fato de uma flor ser um circuito ímpar, todos os seus vértices alcançam a haste por dois caminhos, um deles de comprimento par e outro de comprimento ímpar.

Com estas propriedades Edmonds [8] conclui que uma aresta que não pertence a uma flor, mas que incide em um de seus vértices, é uma aresta não emparelhada e os caminhos aumentantes que passam por uma tal aresta alcançam a base. Então, uma flor não forma uma objeção para um caminho que passe através dela, o que conduz à idéia de contrair a flor em um pseudo-vértice sobre a sua base. As arestas que antes eram incidentes sobre vértices da flor agora são incidentes sobre esse pseudo-vértice.

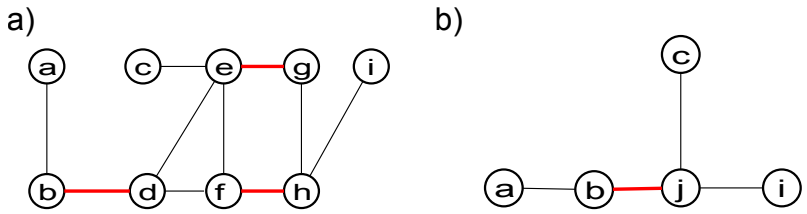

Figura 4.6: a) Identificação de uma flor. b) Contração da flor.

Na Figura 4.6 podemos observar a identificação e a contração de uma flor. A raiz da árvore de busca é o vértice a. Um ramo da busca segue pelos vértices (a, b, d, e, g) e outro por (a, b, d, j, h, g). O vértice g pode receber por um ramo da busca um rótulo par e por outro ramo da busca o rótulo ímpar. O que nos mostra a presença de uma flor composta pelos vértices $\{d, e, f, g, h\}$. A sua base é o vértice d e a haste é o caminho (a, b, d). A flor é contraída sobre a sua base criando um pseudo-vértice j. A Figura 4.6b apresenta o resultado da contração.

A alteração feita no algoritmo para emparelhamento bipartido acontece durante a busca a partir de um vértice livre. Nesse momento é verificado se existe a ambigüidade no rotulamento de um vértice v. No momento que é encontrada essa ambiguidade, a

busca é suspensa e se iniciam duas buscas em profundidade a partir dos dois caminhos que chegam ao vértice v. Estas duas buscas seguirão as marcações de predecessores e têm a finalidade de encontrar o primeiro vértice em comum entre elas. Este vértice é a base da flor e os dois caminhos percorridos contém os vértices da flor. O algoritmo contrai a flor sobre a sua base criando um pseudo-vértice, o qual recebe rótulo par e é inserido na fila para prosseguir a expansão da busca.

Visualizando a Figura 4.6a um ramo da busca em largura segue pelo caminho (a, b, d, e, g), atribuindo um rótulo par ao vértice g, e um ramo segue pelo caminho (a, b, d, f, h), atribuindo um rótulo par ao vértice h. A partir do vértice h é possível atribuir um rótulo ímpar ao vértice visitado g, o qual já possui um rótulo par. Isto caracteriza uma ambigüidade, logo nos mostra a presença de uma flor. Neste momento são iniciadas duas buscas em profundidade. Uma segue o caminho (g, e, d) e a outra segue (g, h, f, d). O vértice em comum entre elas, d, é a base da flor e os vértices $\{d, e, f, g, h\}$ são os vértices que a compõem. Esta estrutura é contraída num pseudo-vértice j como na Figura 4.6b.

Podem ocorrer diversas contrações até que seja encontrado um caminho aumentante. Quando um caminho aumentante é identificado, o algoritmo segue as marcações de predecessores para verificar se existem pseudo-vértices no caminho. Se existir então ele é expandido e verifica-se se o caminho até a base segue pelo caminho de comprimento par ou de comprimento ímpar. Podemos observar na Figura 4.6b que foi identificado o caminho aumentante (a, b, j, c). Seguindo as marcações de predecessores, percebemos que o vértice j é, na realidade, um pseudo-vértice. Neste momento, a flor é expandida para a sua forma original, e podemos observar que o caminho segue pelo arco de comprimento par até a base d da flor. Sendo assim, o caminho aumentante é (a, b, d, f, h, g, e, c).

4.2 Algoritmo certificador e verificador

O algoritmo certificador que veremos para esse problema emprega o conceito de cobertura por conjuntos de tamanho ímpar, e se baseia no teorema, também de autoria de Edmonds, apresentado logo a seguir.

Uma *cobertura por conjuntos de tamanho ímpar* (CCTI) de um grafo G é uma rotulação de vértices de G com números inteiros não negativos, tais que, para toda aresta e de G, temos que e incide em pelo menos um vértice rotulado 1, ou e conecta dois vértices rotulados com o mesmo número $i \geq 2$. Além disso, o número de vértices rotulados i, para $i \geq 2$, é ímpar.

Teorema 4.1 (Edm65 [7])**.** *Seja G um grafo. Se M é um emparelhamento de G, e existe uma CCTI de G (definindo uma coloração com n_i vértices coloridos com a cor i para todo i) tal que*

$$|M| = n_1 + \sum_{i \geq 2} \left\lfloor \frac{n_i}{2} \right\rfloor,$$

então M é máximo.

A prova do teorema de Edmonds é baseada no seguinte fato. Se n_i é o número de vértices rotulados i, para $i \geq 1$, segundo uma CCTI *qualquer* de G, então todo emparelhamento M de G satisfaz

$$|M| \leq n_1 + \sum_{i \geq 2} \left\lfloor \frac{n_i}{2} \right\rfloor.$$

Na Figura 4.7, os rótulos dos vértices constituem uma CCTI, e portanto certificam que o emparelhamento (indicado com arestas coloridas de vermelho) é de cardinalidade máxima.

Novamente, como mencionamos no exemplo do problema de reconhecimento de

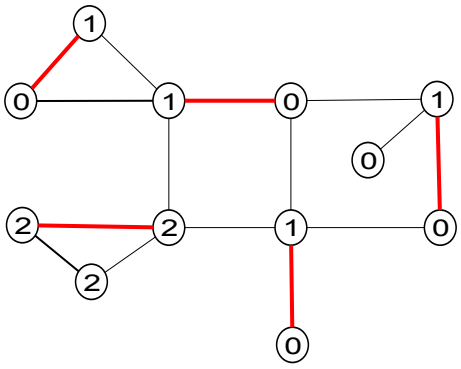

Figura 4.7: Emparelhamento máximo com rotulação de vértices (CCTI).

grafos bipartidos, a compreensão do teorema que garante a certificação *não é requisito para que se possa proceder à verificação da resposta encontrada*, de forma simples e eficiente, protegendo o utilizador contra qualquer erro eventualmente introduzido na implementação do algoritmo de Edmonds que encontra um emparelhamento máximo. O que precisamos fazer é verificar se o emparelhamento M e a testemunha w retornada pelo algoritmo certificador satisfazem a igualdade enunciada no teorema. Além disso, como não devemos esquecer, nosso algoritmo verificador precisa confirmar que M é de fato um emparelhamento, isto é, verificar que cada aresta pertencente ao conjunto de arestas que foi retornado é de fato uma aresta do grafo de entrada, e que não há qualquer vértice do grafo incidindo em mais de uma aresta daquela conjunto. Por último, o verificador precisa também analisar se a rotulação dada como certificado é realmente uma CCTI, o que pode ser feito verificando se cada rótulo aparece um número ímpar de vezes e se todas as arestas incidem em algum vértice de rótulo 1 ou em dois vértices com o mesmo rótulo $i \geq 2$.

Um problema interessante, e mais fácil, do ponto de vista da certificação é o problema de se decidir se um grafo bipartido G admite emparelhamento perfeito. Um certificado simples para o NÃO pode ser dado pelo Teorema de Hall. Se houver um subconjunto S de uma das partes tal que a união dos conjuntos de vizinhos de seus elementos tem cardinalidade menor do que a cardinalidade de S, então não é possível haver emparelhamento

perfeito para G. Um certificado simples para o SIM seria, evidentemente, a exibição de um emparelhamento perfeito.

4.3 Exemplos de verificação

Como exemplo, utilizaremos o grafo da Figura 4.8, e algumas possíveis saídas produzidas pelo algoritmo certificador (um suposto emparelhamento máximo M e uma suposta CCTI C), bem como os resultados da verificação que se sucedeu.

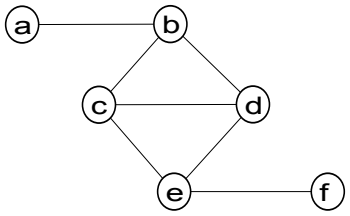

Figura 4.8: Grafo para entrada dos testes do emparelhamento máximo.

1. $M = \{(a, b), (c, d), (e, f)\}$,
   $C = \{a \leftarrow 3, b \leftarrow 1, c \leftarrow 2, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   Verificação ok!

2. $M = \{(b, c), (d, e)\}$,
   $C = \{a \leftarrow 3, b \leftarrow 1, c \leftarrow 2, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   A verificação falha, pois o certificado C não satisfaz a igualdade exigida pelo teorema de Edmonds. Isto não quer dizer que o conjunto de arestas retornado *não seja* um emparelhamento máximo do grafo; quer dizer apenas que não foi possível *verificar isto com o "certificado" que foi apresentado*! Seria preciso um outro certificado (um certificado que de fato certifique!), ou confiar cegamente na resposta, da mesma forma como faríamos no caso de um algoritmo não-certificador. Nesse caso, é fácil ver que a resposta está de fato errada. O emparelhamento retornado

tem cardinalidade 2, ao passo que o grafo admite um emparelhamento (perfeito!) com cardinalidade 3.

3. $M = \{(a, b), (c, d), (e, f)\}$,
   $C = \{a \leftarrow 2, b \leftarrow 1, c \leftarrow 2, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   Nesse caso, mais uma vez, a etapa de verificação falha, pois o certificado que foi retornado pelo algoritmo certificador não é correto — não se trata de uma CCTI, o total de vértices rotulados 2 é par. Este fato, assim como no caso anterior, não nos permitiria dizer que a resposta dada pelo algoritmo (isto é, o conjunto de arestas M) *não é* um emparelhamento máximo do grafo; o verificador apenas diria que não foi capaz de verificar a resposta. De fato, neste exemplo, ao contrário do anterior, a resposta está correta!

4. $M = \{(a, b), (c, d), (e, f)\}$,
   $C = \{a \leftarrow 2, b \leftarrow 1, c \leftarrow 3, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   Com essas entradas, o verificador rejeita a resposta, o certificado não é uma CCTI, apresenta dois vértices com rótulos, diferentes, ≥ 2, ligados pela mesma aresta. Apesar da igualdade estar correta, não podemos afirmar que o conjunto M retornado é, de fato, um emparelhamento máximo.

5. $M = \{(a, b), (c, d), (e, f)\}$,
   $C = \{a \leftarrow 1, b \leftarrow 1, c \leftarrow 2, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   Nesse caso, ao verificarmos o certificado, vemos que a mesma não é uma CCTI, ela possui uma aresta ligada a dois vértices rotulados 1. O que não certifica, logo não é possível afirmar que o emaprelhamento retornado é máximo.

6. $M = \{(a, c), (b, d), (e, f)\}$,
   $C = \{a \leftarrow 3, b \leftarrow 1, c \leftarrow 2, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   A verificação falha, pois o emparelhamento retornado, não é um emparelhamento. O verificador rejeita a resposta quando verifica que no conjunto de arestas retornadas como um emparelhamento, contém uma aresta que não existe no grafo de

entrada.

7. $M = \{(a, b), (b, d), (e, f)\}$,
   $C = \{a \leftarrow 3, b \leftarrow 1, c \leftarrow 2, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   O verificador rejeita a resposta. O conjunto M retornado apresenta duas arestas que incidem em um mesmo vértice, o que, por definição, não é um emparelhamento.

8. $M = \{(a, b), (b, d), (e, f)\}$,
   $C = \{a \leftarrow 1, b \leftarrow 2, c \leftarrow 1, d \leftarrow 2, e \leftarrow 2, f \leftarrow 1\}$.
   O último exemplo, mostramos uma entrada para o verificador que M é um emparelhamento, C é uma CCTI, mas o verificador não pode afirmar se o emparelhamento é máximo, pois não satisfaz a igualdade exigida pelo teorema de Edmonds. Sendo $|M| = 3$ e $(n_1 + \sum_{i \geq 2} \lfloor \frac{n_i}{2} \rfloor) = 4$. O que impossibilita a confirmação da resposta.

Os casos em que a resposta está correta e a verificação se dá de forma bem sucedida são, entre si, análogos — todas as exigências do verificador são atendidas. Enumeramos diversos exemplos, em que a verificação falha (tanto para resposta correta quanto para resposta incorreta): o conjunto de arestas retornado pode não ser um emparelhamento do grafo de entrada (pode conter uma não-aresta do grafo, ou conter duas arestas incidindo em um vértice comum, por exemplo); o certificado retornado pode não ser uma CCTI (por diversas razões); o certificado retornado pode ser uma CCTI que não satisfaz o teorema de Edmonds, etc. É preciso, como se vê, bastante cuidado com a completude do verificador, para que nenhum caso possa haver em que uma resposta incorreta seja validada.

5 PROBLEMA 3: $K-$SELECTION

O terceiro problema estudado é de seleção dos k menores números de uma lista dada [2]. O objetivo do estudo desse problema é mostrar que em determinados casos, não há necessidade de fazermos um novo algoritmo para obtermos, além da resposta, um certificado. A resposta do algoritmo clássico já é, por assim dizer, o próprio certificado, de forma que precisamos nos preocupar apenas com a etapa da verificação.

O problema conhecido como $k-$Selection pode ser resolvido por diversos algoritmos. Uma das formas mais ingênuas de resolvê-lo é através do algoritmo de ordenação por seleção (*Selection sort*) [13]. Trata-se, no caso, de uma execução parcial daquele algoritmo, em que a lista dada será percorrida k vezes. A cada iteração j, para $1 \leq j \leq k$, o algoritmo percorre a lista e encontra o menor elemento da lista, removendo-o da lista e o adicionando ao conjunto que será retornado. Apesar de parecer clara a forma de implementação, ela só é eficaz se o k for pequeno, pois o algoritmo é assintoticamente ineficiente, demandando tempo $O(nk)$ [12].

Outro algoritmo que podemos usar para selecionarmos os k menores elementos é o *Quicksort* (ou qualquer outro algoritmo de ordenação). Uma vez ordenada a lista, retornar os k menores elementos é tarefa trivial.

Uma terceira maneira é utilizarmos uma *heap* de máximo para encontrarmos os k menores elementos de uma lista. Inicializamos a *heap* com os k primeiros elementos da lista de entrada e percorremos os elementos restantes da lista. A cada elemento observado, verificamos se ele é menor do que a raiz da *heap*. Caso seja, ele deve ser incluído na *heap*, e a raiz (que é o maior elemento da *heap*) é removida. Ao final do processo, os k elementos que estiverem na *heap* serão os k menores elementos da lista original. O tempo

de execução é no pior caso $O(n \log k)$.

Finalmente, podemos solucionar esse problema com o algoritmo clássico *Quick Select* em tempo linear $O(n)$, para qualquer $k \leq n$, um desempenho, portanto, ótimo [16]. A ideia do *Quick Select* é muito simples, análoga à do *Quick Sort* [11].

Um pivô é selecionado aleatóriamente, em uma lista L com n elementos, os elementos maiores que ele são colocados na primeira metade da Lista, e os elementos que são iguais ou maiores, na segunda metade (a "metade" não é necessariamente exata, uma vez que é possível que o resultado não seja exatamente "metade"). No entanto, ao fazer a seleção, já sabemos em qual partição o elemento desejado encontra-se, pois o pivô, ao final de cada iteração, já está em sua posição final.

Qualquer que seja o algoritmo escolhido para retornar os k menores elementos, a própria saída (o conjunto dos k menores) já é por si só um certificado, no sentido de se prestar à verificação simples e direta. O verificador, então, recebe a lista de entrada do algoritmo e os k menores elementos retornados pelo algoritmo, podendo, a partir daí, validar (ou não) essa resposta.

5.1 Algoritmo verificador

Para a validação desta resposta, o algoritmo verificador analisa a solução em tempo esperado $O(n + k) = O(n)$. O algoritmo é dividido em etapas, analisando os possíveis erros em uma lista com os k menores elementos.

Inicialmente é verificado se a quantidade de números retornados é igual a k. Trata-se de uma verificação trivial, mas que não pode deixar de ser feita. A segunda verificação analisa se todos os elementos que são menores ou iguais, na lista de entrada, ao k-ésimo

elemento retornado, aparecem no conjunto retornado. A terceira verificação diz respeito ao número de ocorrências de cada elemento menor do que o k-ésimo no conjunto retornado. É preciso que esse número, para cada elemento, seja precisamente sua multiplicidade na lista de entrada. A última verificação analisa se o número de ocorrências do k-ésimo menor elemento não excede a sua multiplicidade na lista de entrada.

5.2 Testes computacionais

A implementação do algoritmo para o funcionamento do $k-$*Selection* foi feita baseada na implementação de John Kurlak [**?**]. A partir desse código, implementamos o algoritmo verificador para conferirmos os resultados. O algoritmo verificador roda em tempo linear. Foram utilizadas tabelas *hash* para as operações de inserção e de consulta da lista de entrada, e o tempo esperado para cada operação é $O(1)$. O Apêndice B apresenta o código fonte. Primeiramente foi feita uma análise da implementação e o verificador retornou que a saída estava correta. E em seguida fizemos simulações de respostas erradas do *Quick Select*, que poderiam passar despercebidos e que nos retornaria uma resposta errada.

Para todos os testes realizados utilizamos a seguinte lista como entrada:

$$\text{Entrada: } \{2, 67, 19, 40, 40, 40, 18, 2, 10, 10\}$$

Resultados dos testes realizados com e sem a simulação de erros:

- Verificação para uma saída correta
  - **Saída do algoritmo** $\rightarrow \{2, 10, 18, 2, 10\}$

para k = 5

  - **Saída do algoritmo verificador** $\rightarrow$ "Verifição ok!"

- Verificação para uma saída correta, com multiplicidade do $k-$ésimo elemento diferente da lista de entrada.

  - **Saída do algoritmo** $\rightarrow \{2, 10, 2\}$

    para k = 3

  - **Saída do algoritmo verificador** $\rightarrow$ "Verifição ok!"

- Verificação para uma saída incorreta, faltando um elemento.

  - **Saída do algoritmo** $\rightarrow \{2, 2, 19, 10, 10\}$

    para k = 5

  - **Saída do algoritmo verificador** $\rightarrow$ "O elemento 18, menor ou igual ao $k-$ésimo na lista de entrada, não aparece na lista retornada!"

  Por terem k elementos na lista de saída, essa resposta passaria pela nossa primeira verificação, mas na segunda verificação o verificador retorna que a resposta está errada, pois encontra o elemento 18 na lista de entrada, mas não aparece na lista de saída.

- Verificação para uma saída incorreta, com o número de elementos retornados errados.

  - **Saída do algoritmo** $\rightarrow \{2, 2, 10, 10, 18, 19\}$

    para k = 5

  - **Saída do algoritmo verificador** $\rightarrow$ ""Foram retornados mais do que 5 elementos!"

  A resposta retornada mostra que a verificação falhou, isto é, a saída do algoritmo *Quick Select* está incorreta, pois apresentou uma lista com mais do que k elementos.

- Verificação para uma saída incorreta, com retorno de um número que não pertence a lista de entrada.

  - **Saída do algoritmo** $\rightarrow \{2, 2, 18, 11, 10\}$

    para k = 5

  - **Saída do algoritmo verificador** $\rightarrow$ "O elemento 11, menor do que o $k-$ésimo, aparece com multiplicidade 1 na lista de saída do algoritmo; no entanto, sua multiplicidade na lista de entrada é 0"

  Nesse caso, todos são menores do que o $k-$ésimo, e a lista de saída tem k elementos, desta forma passaria pela primeira e pela segunda verificação, mas não pela terceira que verifica se o número de ocorências de um elemento na lista de saída é igual ao número de ocorrências desse mesmo elemento na lista de entrada. O verificador encontra o elemento 11 que só aparece na lista de saída e não aparece na lista de entrada.

- Verificação para uma saída incorreta, com a multiplicidade do menor elemento errada.

  - **Saída do algoritmo** $\rightarrow \{2, 10, 10, 2, 2\}$

    para k = 5

  - **Saída do algoritmo verificador** $\rightarrow$ "O elemento 2, menor do que o $k-$ésimo, aparece com multiplicidade 3 na lista de saída do algoritmo; no entanto, sua multiplicidade na lista de entrada é 2"'

  Esse teste, como o anterior, passaria pela 1ª e 2ª verificação.

- Verificação para uma saída incorreta, multiplicidade do $k-$ésimo elemento errada.

  - **Saída do algoritmo** $\rightarrow \{2, 10, 10, 2, 18, 18\}$

    para k = 6

– **Saída do algoritmo verificador** $\rightarrow$ "O elemento 18, que foi retornado como sendo o $k-$ésimo menor, aparece com multiplicidade 2 na lista de saída do algoritmo, maior, portanto, que sua multiplicidade na lista de entrada, que é 1."

Após a criação dos testes, testamos a importância de cada um separadamente, comentando o código da implementação. Todas as verficações foram comentadas uma a uma para analisarmos a necessidade das mesmas. Em todas elas verificamos que pelo menos um dos nossos testes falharam.

Tabela 5.1: Resultados para listas com n números (em milisegundos)

Teste $k-$Select		
n	**Tempo QuicSelect**	**Tempo Verificador**
1.000	0,00013440	0,00007086
10.000	0,00116314	0,00025364
100.000	0,00446295	0,00183073
1.000.000	0,00809997	0,00648183
10.000.000	0,01283904	0,00601609

Foi realizado também a análise de tempo, fazendo uma comparação entre o tempo gasto pelo *Quick Select* com o tempo gasto pelo algoritmo verificador. Usamos a função {*System.nanoTime()*} do java, para fazer essa comparação. Como podemos ver na Tabela 5.1 o número de elementos n para entrada foi aumentando. O tempo gasto pelo algoritmo verificador é sempre menor que o tempo gasto pelo algoritmo *Quick Select*, apesar de nesse caso, os dois rodarem em tempo linear.

6 PROBLEMA 4: RECONHECIMENTO DE GRAFOS DE DISCO UNITÁRIO

O último problema que vamos abordar é o do reconhecimento de grafos de disco unitário, uma classe de grafos que encontra diversas aplicações, com destaque em modelagens de redes sem fio [17]. Um *grafo de disco unitário* (GDU) é um grafo cujos vértices podem ser representados por pontos no plano e cujas arestas são definidas por pares de pontos que estão distantes no máximo uma unidade um do outro. Os GDU também podem ser entendidos como grafos de interseção de discos congruentes coplanares (de diâmetro 1).

Definição 6.1. *Dado um grafo G, um* modelo de discos unitários *(modelo GDU) de G é uma função $\phi : V(G) \to \mathbb{R}^2$, tal que, para algum número real $d > 0$, quaisquer vértices $u, v \in V(G)$ satisfazem*

- $||\phi(u) - \phi(v)|| \leq d$, *se* $uv \in E(G)$;
- $||\phi(u) - \phi(v)|| > d$, *se* $uv \notin E(G)$.

Um grafo é um GDU se e somente se admitir um modelo GDU. Normalmente é considerado 1 para o valor de d (por isso o nome de disco unitário) [22], e dessa forma, consideraremos esse valor no decorrer desse capítulo. A Figura 6.1 ilustra formas de representar um grafo de disco unitário.

O interessante, ao se estudar algoritmos certificadores e verificadores para esse problema, é mostrar que em alguns casos a testemunha exibida permite apenas uma verificação que não é formalmente eficiente, por demandar tempo exponencial, mas que, para

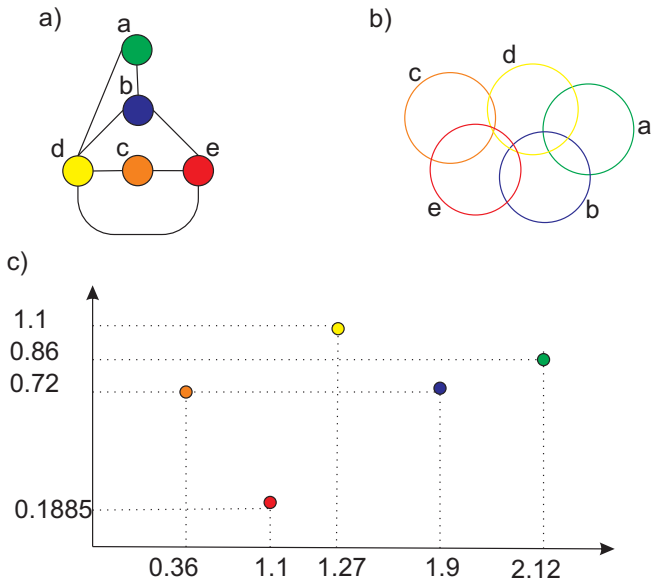

Figura 6.1: a) Grafo de disco unitário G. b) Modelo de discos congruentes de G. c) Representação de G por pontos em $\mathbb{R}^2$.

instâncias pequenas, é computacionalmente viável, permitindo por exemplo a criação de provas computacionais para teoremas.

Existem instâncias de grafos que ninguém sabe dizer se são ou não GDU. Um algoritmo cerificador que classifique essas instâncias e que permita a verificação simples pode ser usado para provar, que alguma instância específica é (ou não é) GDU. Não existe certificado para os grafos (a) e (b) da Figura 6.2, no entanto, seria muito importante conseguirmos provar que esses grafos não são GDU. A prova acarretaria em consequências imediatas, existem resultados que dependem disso.

O problema de reconhecer se um dado grafo é um GDU é NP-difícil [3]. Não se sabe sequer se esse problema pertence a NP, pois não se conhece nenhum certificado polinomial para o *SIM*. Não é conhecido certificado polinomial para o NÃO para nenhum problema NP-completo. Isto é, tanto quanto se sabe, a interseção de NP-completo e de co-NP é vazia.

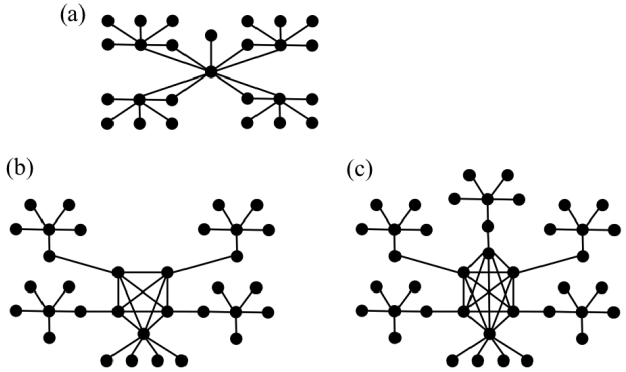

Figura 6.2: (a) Wu et al. [22] conjecturou que não é GDU. (b) Grafo de disco unitário. (c) Não se sabe se é um GDU.

O certificado natural para o *SIM* é um modelo em $\mathbb{R}^2$. O problema é que pode ser necessário um número duplamente exponencial de bits para as coordenadas desse modelo. Então, o algoritmo certificador pode não ser eficiente, mas para entradas pequenas, mesmo exigindo tempo exponencial, pode ser possível obter o certificado para o *SIM*.

O certificado para o *NÃO*, por outro lado, não é tão simples. Os possíveis mapeamentos de vértices em pontos do plano são infinitos, pois $\mathbb{R}^2$ é infinito! Para resolver essa dificuldade, Fonseca et al, em [10], apresentaram uma nova técnica para o reconhecimento de grafos de disco unitário. Eles utilizaram o conceito de *modelo relaxado UDG* que é o mapeamentos de vértices em um subconjunto finito (enumerável) do plano, sendo que a distância entre dois vértices pode pertencer a uma faixa (intervalo) reservada a pares de vértices adjacentes, a uma faixa reservada a pares de vértices não-adjacentes ou podem ainda pertencer a uma faixa neutra, que aceita pares de vértices dos dois tipos.

Dado um $\epsilon > 0$ e $\epsilon \in \mathbb{Q}$, considere o conjunto $Q_\epsilon = \{x \in \mathbb{Q} : x = d\epsilon, d \in \mathbb{Z}\}$, e $C_\epsilon = Q_\epsilon \times Q_\epsilon$ um conjunto discreto de coordenadas no plano. Chamamos C_ϵ uma $\epsilon-$grade. Quanto menor o valor de ϵ, maior a granularidade da grade. Dizemos que $C_{\epsilon 1}$ é

mais estreita do que $C_{\epsilon 2}$ (equivalente, $C_{\epsilon 2}$ é mais espessa do que $C_{\epsilon 1}$) se a granularidade de $C_{\epsilon 1}$ é maior do que a de $C_{\epsilon 2}$, ou seja, se $\epsilon 1 < \epsilon 2$. Uma *grade* pode ser considerada como um conjunto de pontos de interseção de retas verticais e horizontais igualmente espaçadas. Chamamos de *célula da grade*, os pontos do plano que se encontram dentro dos quadrados definidos por duas linhas horizontais consecutivas e duas linhas verticais consecutivas.

Definição 6.2. *Seja G um grafo e seja C_ϵ uma grade de granularidade ϵ. Um modelo $\epsilon-$discreto do grafo G é uma função $\psi_\epsilon : V(G) \to C_\epsilon$, de tal modo que para todo $u, v \in V(G)$:*

- $||\psi_\epsilon(u) - \psi_\epsilon(v)|| < 1+ \epsilon\sqrt{2}$, *se* $uv \in E(G)$;
- $||\psi_\epsilon(u) - \psi_\epsilon(v)|| > 1- \epsilon\sqrt{2}$, *se* $uv \notin E(G)$.

Podemos observar que o modelo relaxado do grafo G não é necessariamente um modelo GDU de G, mas por outro lado, todo modelo GDU de um grafo G é um modelo relaxado de G.

Dado C_ϵ e um modelo relaxado ψ_ϵ de G, definimos um modelo relaxado $H[\psi_\epsilon]$ como o grafo completo, cujo conjunto de vértices é $V(G)$ e em que cada aresta de uv em $E(H[\psi_\epsilon)]$ indica uma de três tipos possíveis adjacências:

1. Aresta obrigatória, se $||\psi_\epsilon(u) - \psi_\epsilon(v)|| \leq 1 - \epsilon\sqrt{2}$;
2. Aresta proibida, se $||\psi_\epsilon(u) - \psi_\epsilon(v)|| \geq 1 + \epsilon\sqrt{2}$;
3. Aresta opcional, se $1 - \epsilon\sqrt{2} < ||\psi_\epsilon(u) - \psi_\epsilon(v)|| < 1 + \epsilon\sqrt{2}$.

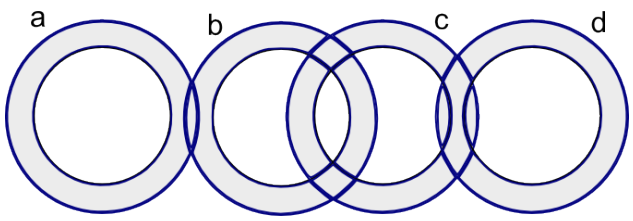

Figura 6.3: Modelo de discos relaxados.

Um modelo relaxado de G pode, portanto, ser considerado como um mapeamento de $V(G)$ para pontos de uma $\epsilon-$grade, tal que um modelo relaxado $H[\psi_\epsilon]$, satisfaz as seguintes condições para todos os vértices $u, v \in V(G)$:

1. Se $u, v \in E(G)$, então as arestas correspondentes em $H[\psi_\epsilon]$, são do tipo obrigatória ou opcional.
2. Se $u, v \notin E(G)$, então as arestas correspondentes em $H[\psi_\epsilon]$, são do tipo proibida ou opcional.

Diferentemente do apresentado na Figura 6.1b, cada vértice, no modelo relaxado de discos, é representado por dois círculos concêntricos. Um interior de diâmetro fechado $1 - \epsilon\sqrt{2}$, e o outro exterior com o diâmetro aberto $1 + \epsilon\sqrt{2}$. A área entre os dois círculos é considerada como incerta. Na Figura 6.3, essa área esta representada pela cor cinza. A Figura 6.3 mostra um grafo G com quatro vértices. Podemos obsevar que a única aresta obrigatória é $\{b, c\}$; as arestas proibidas são $\{a, c\}$, $\{a, d\}$ e $\{b, d\}$; e as arestas opcionais são $\{a, b\}$ e $\{c, d\}$.

A seguir serão descritos dois lemas provados no artigo de Fonseca et al [10], que são fundamentais para a compreensão do algoritmo.

Lema 6.1. *Seja G um grafo, e ψ_ϵ um modelo relaxado de G. Se todas as adjacências opcionais em $H[\psi_\epsilon]$ correspondem a pares de vizinhos em G (alternativamente, a pares de não-vizinhos em G), então G é um GDU.*

Lema 6.2. *Se G é um GDU, então G admite um modelo relaxado ϵ-discreto para todo racional positivo ϵ.*

6.1 Algoritmo para reconhecimento de grafos de disco unitário

O Algoritmo para o reconhecimento de grafos de disco unitário desenvolvido por Fonseca et al [10] realiza uma busca por modelos discretos em um grafo G em uma determinada $\epsilon-$grade, até que um modelo GDU (não-relaxado!) seja encontrado (resposta *SIM*), ou até que não exista um modelo relaxado disponível para G para esse valor de ϵ (resposta *NÃO*). A busca exaustiva em uma grade muito estreita requer um esforço computacional elevado. Por esse motivo, a busca deve ser iniciada por um ϵ maior (o valor inicial de ϵ é passado por parâmetro, e normalmente é definido por um valor ligeiramente abaixo de $1/\sqrt{2}$, como por exemplo $0,7$), para que o número de nós na grade seja pequeno e seja possível pesquisar todos os possíveis modelos relaxados $\epsilon-$discretos de G.

Cada vértice v de $V(G)$ é posicionado, um a um, em um ponto escolhido a partir de um conjunto de locais candidatos para v na $\epsilon-$grade, locais esses que são obtidos em função do posicionamento daqueles dentre seus vizinhos e não-vizinhos que já tiverem sido posicionados. Quando um vértice não tem qualquer local candidato possível, é realizado um retorno (*backtrack*) e uma nova tentativa de posicionamento. Se um modelo relaxado, que satisfaça as condições do Lema 6.1, for encontrado, o algoritmo retorna *SIM* (é um GDU). Se não existir um modelo relaxado na granularidade, então, pelo Lema 6.2, G não é um GDU, e o algoritmo retorna *NÃO*.

Finalmente, se um ou mais modelos relaxados $\epsilon-$discreto existem, mas nenhum deles satisfaz o lema 6.1, então, a busca recomeça, de forma iterativa, em uma grade mais estreita, por exemplo com ϵ de valor igual à metade do valor utilizado na grade anterior, no que corresponderia a uma sequência de granularidades tal como $0,7$, $0,7/2$, $0,7/3$,

$0, 7/4$, etc. Podemos notar que, quanto mais estreita a grade, maior a probabilidade de se obter uma resposta conclusiva, mesmo precisando de um tempo computacional maior.

6.2 Algoritmo certificador e verificador

O algoritmo certificador para o problema retorna, além da resposta esperada (*SIM* ou *NÃO*), um certificado para essa resposta. Quando o algoritmo retorna *SIM*, apresenta também uma lista com coordenadas para cada vértice (um modelo GDU). Quando o algoritmo retorna *NÃO*, apresenta um valor para a granularidade ϵ para a qual não existe modelo relaxado GDU.

Para validarmos essas saídas, o algoritmo verificador recebe, como de costume, o grafo de entrada, a resposta obtida e o certificado retornado pelo algoritmo certificador. Se a resposta é *SIM*, deve-se verificar se os pares de vértices adjacentes apresentam coordenadas a uma distância menor ou igual a 1, e se os pares de vértices não-adjacentes apresentam coordenadas a uma distância maior que 1. Se a resposta é *NÃO*, o verificador analisa se realmente não existe um modelo relaxado para a granularidade dada, o que pode ser feito por um algoritmo simples de busca exaustiva.

6.3 Testes computacionais

O algoritmo foi rodado para duas instâncias, uma *SIM* e uma *NÃO*, simulando situações de erro em ambos os casos. Apresentaremos exemplos de verificações que aceitaram a resposta retornada pelo algoritmo certificador e de verificações que rejeitaram as respostas. Essas verificações que foram rejeitadas ocorrem ou porque a resposta do algoritmo está errada, ou porque o certificado não certifica de fato (certificado errado).

Para os testes na Instância *SIM*, usamos o grafo $K_{1,4}$ como entrada. A Figura 6.5a apresenta a entrada do primeiro teste. Na Figura 6.4 podemos ver um modelo de disco unitário.

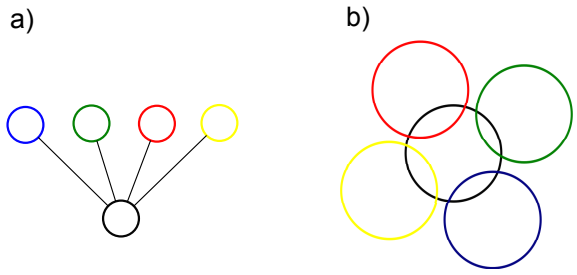

Figura 6.4: $K_{1,4}$.

Entrada para o verificador:

Instância:	$K_{1,4}$
Resposta do algoritmo:	SIM
Certificado:	Coordenadas

1: (0.7071,0)
2: (-0.7071,-0.21213)
3: (0,0)
4: (1.4142, -0.7071)
5: (0, -1.4142)

Figura 6.5: Resposta correta - Certificado errado.

O verificador recebe esses parâmetros como entrada e nos retorna uma lista com os vértices adjacentes e suas distâncias baseadas nas coordenadas.

Nesse caso, esse certificado retornou um modelo GDU inválido, pois o verificador obteve o seguinte:

$(1, 5)$ vizinhos em G [distância = 1.58112]

A distância entre os vértices 1 e 5 é maior do que 1 no modelo apresentado, mesmo eles sendo adjacentes no grafo $K_{1,4}$. Então percebemos que esse certificado não está

correto. Dessa forma, o verificador nos retorna que a verificação falhou.

O segundo exemplo (Figura 6.6) também mostra o funcionamento do verificador em um teste na instância *SIM* para a seguintes entrada:

Entrada para o verificador:	
Instância:	$K_{1,4}$
Resposta do algoritmo:	NÃO
Certificado:	0.7017

Figura 6.6: Resposta errada - Certificado errado.

Ao rodarmos o algoritmo verificador para essas entradas, ele *encontra* um modelo relaxado para esse grafo, na granularidade dada $\epsilon = 0.7071$.

Modelo relaxado encontrado:

- Coordenadas:

  1: (0.7071, 0)
  2: (0,0)
  3: (0.7071, 0)
  4: (0.7071,-0.7071)
  5: (0,-1.4142)

- Tipos de adjacências e suas distâncias:

  (1,2) vizinhos em G [distância= 0.7071] – aresta opcional
  (1,3) vizinhos em G [distância= 0.0] – aresta obrigatória
  (1,4) vizinhos em G [distância= 0.7071] – aresta opcional
  (1,5) vizinhos em G [distância= 1.5811] – aresta opcional
  (2,3) não-vizinhos em G [distância= 0.7071] – aresta opcional

(2,4) não-vizinhos em G [distância$= 0.9999$] – aresta opcional

(2,5) não-vizinhos em G [distância$= 1.414$] – aresta opcional

(3,4) não-vizinhos em G [distância$= 0.7071$] – aresta opcional

(3,5) não-vizinhos em G [distância$= 1.5811$] – aresta opcional

(4,5) não-vizinhos em G [distância$== 0.9999$] – aresta opcional

Com esse modelo relaxado, a resposta do verificador é: "A verificação falhou!" O verificador não pode afirmar que o grafo não é um GDU, pois ele possui um modelo relaxado na granularidade dada. (E, de fato, como sabemos, o $K_{1,4}$ *é* um GDU.)

No último teste para instância *SIM* (Figura 6.7), analisamos como seria a reposta do verificador quando as entradas estivessem corretas.

Entrada para o verificador:

Instância:	$K_{1,4}$
Resposta do algoritmo:	SIM
Certificado:	Coordenadas
	1: (0.7071, 0)
	2: (0, 0)
	3: (1.4142, 2.8284)
	4: (1.4142, -2.8284)
	5: (3.5355, 0)

Figura 6.7: Resposta correta - Certificado correto.

O verificador apresenta os tipos de adjacências e suas distâncias.

(1,2) vizinhos em G [distância = $7071.0/30000$]

(1,3) vizinhos em G [distância = $29154.5/30000$]

(1,4) vizinhos em G [distância = $29154.5/30000$]

(1,5) vizinhos em G [distância = $28284.0/30000$]

(2,3) não-vizinhos em G [distância = $31622.5/30000$]

(2,4) não-vizinhos em G [distância = $31622.5/30000$]
(2,5) não-vizinhos em G [distância = $35355.0/30000$]
(3,4) não-vizinhos em G [distância = $56568.0/30000$]
(3,5) não-vizinhos em G [distância = $35355.0/30000$]
(4,5) não-vizinhos em G [distância = $35355.0/30000$]

E como resposta: Verificação ok! O grafo é um GDU. Apresenta para arestas obrigatórias uma distância ≤ 1 e para arestas proibidas uma distância > 1.

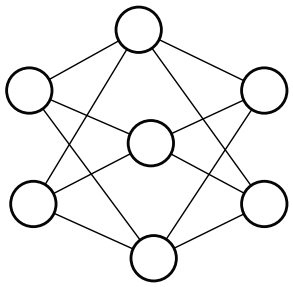

Figura 6.8: $K_{3,4}$.

Para os testes na Instância *NÃO*, usamos o grafo $K_{3,4}$ (Figura 6.8) como entrada.

Entrada para o verificador:	
Instância:	$K_{3,4}$
Resposta do algoritmo:	NÃO
Certificado:	0.35355

Figura 6.9: Resposta correta - Certificado errado.

Durante a execução da verificação para o *NÃO*, o verificador tentará exaurir as possibilidades de posicionamento dos vértices na granularidade dada como certificado. No entanto, não será possível exauri-la, pois encontra, por exemplo o seguinte modelo relaxado:

- Coordenadas:

  1: (0.35355, 0)
  2: (-0.353550, 0.7071)
  3: (0.7071, 0.7071)
  4: (0, 0)
  5: (-0.35355, 0.7071)
  6: (0, 1.4142)
  7: (0.35355, 0.7071)

- Tipos de adjacências e suas distâncias:

  (1,2) não-vizinhos em G [distância = 0.99999] – aresta opcional
  (1,3) não-vizinhos em G [distância = 0.79056] – aresta opcional
  (1,4) vizinhos em G [distância = 0.35355] – aresta obrigatória
  (1,5) vizinhos em G [distância = 0.99999] – aresta opcional
  (1,6) vizinhos em G [distância = 1.457725] – aresta opcional
  (1,7) vizinhos em G [distância = 0.7071] – aresta opcional
  (2,3) vizinhos em G [distância = 1.06065] – aresta opcional
  (2,4) vizinhos em G [distância = 0.79056] – aresta opcional
  (2,5) vizinhos em G [distância = 0.0] – aresta obrigatória
  (2,6) vizinhos em G [distância = 0.79056] – aresta opcional
  (2,7) vizinhos em G [distância = 0.7071] – aresta opcional
  (3,4) vizinhos em G [distância = 0.99999] – aresta opcional
  (3,5) vizinhos em G [distância = 1.06065] – aresta opcional
  (3,6) vizinhos em G [distância = 0.99999] – aresta opcional
  (3,7) vizinhos em G [distância = 0.35355] – aresta obrigatória
  (4,5) não-vizinhos em G [distância = 0.79056] – aresta opcional
  (4,6) não-vizinhos em G [distância = 1.4142] – aresta opcional
  (4,7) não-vizinhos em G [distância = 0.79056] – aresta opcional
  (5,6) não-vizinhos em G [distância = 0.79056] – aresta opcional

(5,7) não-vizinhos em G [distância = 0.7071] – aresta opcional

(6,7) não-vizinhos em G [distância = 0.79056] – aresta opcional

O resultado da verificação é inconclusivo, pois foi encontrado um modelo relaxado na granularidade 0.35355. Desta forma, não é possível garantir que o grafo não seja um GDU.

O próximo exemplo (Figura 6.10) apresenta, como entrada do verificador, a resposta errada e o certificado errado.

Entrada para o verificador:

Instância:	$K_{3,4}$
Resposta do algoritmo:	SIM
Certificado:	Coordenadas:

1: (0.35355, 0)
2: (-0.35355, 0.7071)
3: (0, 0)
4: (0, 0)
5: (0.35355, 0.35355)
6: (0, 0)
7: (-1.4142, 0.7071)

Figura 6.10: Resposta errada - Certificado errado.

Quando o verificador retornou a lista de adjacências e suas distâncias, apresentou:

(1,2) não-vizinhos em G [distância = 0.99999]

Os vértices 1 e 2, que não são adjacentes, aparecem no modelo apresentado com distância menor do que 1.

A última análise do algoritmo verificador para o reconhecimento de grafos de disco unitário testa a verificação na instância *NÃO*, para resposta correta e certificado correto (Figura 6.11).

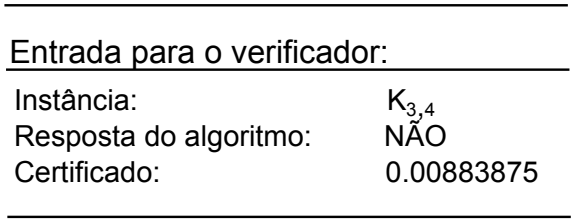

Figura 6.11: Resposta correta - Certificado correto.

O verificador executa sua busca na granularidade dada pelo certificado e de fato não encontra qualquer modelo relaxado, retornando como resposta: "Verificação ok! O grafo não é um GDU". Para esse caso, o programa executou em $1h54$minutos.

7 CONCLUSÃO

Apresentamos neste trabalho uma pesquisa sobre algoritmos certificadores e verificadores. Inicialmente, apresentamos conceitos e propriedades básicas desses algoritmos, fazendo uma analogia com os algoritmos clássicos, não-certificadores. Tanto quanto sabemos, trata-se do primeiro texto em língua portuguesa sobre o tema.

No decorrer do trabalho foram apresentados quatro problemas: o reconhecimento de grafos bipartidos, o emparelhamento de cardinalidade máxima, $k-$*Selection*, o problema de seleção dos k menores elementos de uma lista, e o problema de reconhecimentos de grafos de disco unitário. Ilustramos com exemplos didáticos o uso dessa técnica que ainda é pouco difundida.

Para cada um dos problemas, foram feitos estudos de algoritmos certificadores e verificadores. Para primeiro problema, o reconhecimento de grafos bipartidos, apresentamos um algoritmo certificador e um verificador e mostramos os resultados para testes computacionais. No problema do emparelhamento de cardinalidade máxima, fizemos uma revisão bibliográfica apresentando o algoritmo clássico e o um algoritmo certificador e um verificador. Para terceiro problema, apresentamos uma forma de realizar a verificação sem precisar de um algoritmo certificador. Desta forma, mostramos como essa técnica se aplica sem a necessidade de se produzir um certificado, mas apenas agregar um verificador ao algoritmo clássico. Implementamos um algoritmo verificador para o problema de seleção dos k menores elementos e realizamos testes computacionais, mostrando que um algoritmo de verificação pode ser simples e eficiente. Por último, foi estudado o problema do reconhecimento de grafos de disco unitário. Nesse caso, notamos que mesmo quando a verificação não é formalmente eficiente, para algumas instâncias ela é computacionalmente viável e pode ser util. Mostramos assim, como a verificação pode ser usada

para validar provas de teoremas, ainda que o tempo computacional possa ser exponencial, pois, nesses casos, estamos interessados em instâncias específicas.

Defendemos a idéia que as implementações de algoritmos em Computação Científica devem, preferencialmente, seguir o paradigma de "auto-teste", pelo qual o programa, sempre que possível, deve conter o verificador embutido.

A importância de testes automatizados que permitam a cobertura completa do código de programas é ponto pacífico na comunidade da Engenharia de Software. A boa prática considera a escrita dos chamados "unit tests", que não raro antecede a escrita dos próprios programas (no que é chamado de *Test Driven Development*), como parte inerente e inseparável do desenvolvimento de qualquer software. A computação científica, por outro lado, parece raramente se preocupar com esse tipo mais mundano de questão. Uma vez provado que seus algoritmos são corretos (e completos, e eficientes), a tarefa de implementá-los, para que eles possam afinal ser úteis, parece constituir detalhe de somenos importância, algo que pode ser desempenhado por qualquer um sem que maiores esforços precisem ser despendidos para garantir que funcionem como se espera. Trata-se, no entanto, de grande equívoco, uma vez que os erros mais comuns aparecem justamente na implementação dos algoritmos. De que adianta o algoritmo mais eficiente para certo problema ter sido meticulosamente analisado e escrutinado pela comunidade científica, se sua encarnação real na forma de um programa de computador é algo que não pode receber o mesmo tratamento? De que adiantam elaboradas provas matemáticas sobre a *ideia* do algoritmo, se sua *realização* é algo intangível na prática, a respeito do qual os eventuais usuários precisarão devotar boa dose de fé, resignando-se à esperança de que erros mais ou menos sutis não tenham sido introduzidos em sua fase de implementação?

Algoritmos certificadores estão protegidos contra falhas de implementação. Os certificados produzidos por eles, e que acompanham as próprias soluções apresentadas, podem ser verificados de forma direta e eficiente por programas sensivelmente mais sim-

ples, cuja corretude admitiria até mesmo uma prova formal. Além disso, a verificação das respostas produzidas pode ser conduzida (computacionalmente ou não) pelo próprio usuário do programa original, prescindindo assim de qualquer dose daquela "fé" a que nos referíamos.

REFERÊNCIAS

[1] BATISTA, Rogério da Silva. **Implementações eficientes para problemas de caminhos e ciclos em grafos com arestas coloridas**. 2014. Dissertação (Mestrado) - Universidade Federal Fluminense, Niterói, 2014.

[2] BLUM, Manuel et al. Time bounds for selection. **Journal of Computer and System Sciences**, v. 7, n. 4, p. 448-461, 1973.

[3] BREU, Heinz; KIRKPATRICK, David G. Unit disk graph recognition is np-hard. **Computational Geometry**, v. 9, n. 1-2, p. 3-24, 1998.

[4] BRIGHT, Jonathan D.; SULLIVAN, Gregory F.; MASSON, Gerald M. A formally verified sorting certifier. **IEEE Transactions on Computers**, v. 46, n. 12, p. 1304-1312, 1997.

[5] CRUZ, Carlos Fernando Bella; SETUBAL, João Carlos. Uma adaptação do algoritmo de emparelhamento de Edmonds para execução em paralelo. In: OFICINA NACIONAL DE PROBLEMAS COMBINATÓRIOS: Teoria, Algoritmos e Aplicações, 2., 1995, Campinas. **Anais...** Campinas: UNICAMP, 1995. p. 39-49. (Relatório Técnico DCC, 95-17).

[6] DIESTEL, Reinhard. **Graph theory**. 2. ed. New York: Springer, 2000.

[7] EDMONDS, Jack. Maximum matching and a polyhedron with 0, l-vertices. **Journal of Research of the National Bureau of Standards-B. Mathematics and Mathematical Physics**, v. 69B, n. 1-2, p. 125-130, 1965.

[8] _____. Paths, trees, and flowers. **Canadian Journal of mathematics**, v. 17, n. 3, p. 449-467, 1965.

[9] FEOFILOFF, Paulo; KOHAYAKAWA, Yoshiharu; WAKABAYASHI, Yoshiko. **Uma introdução sucinta à teoria dos grafos**. Disponível em: <http://www.ime.usp.br/ pf/teoriadosgrafos/texto/TeoriaDosGrafos.pdf>. Acesso em: 13 ago. 2015.

[10] FONSECA, Guilherme Dias da et al. On the recognition of unit disk graphs and the distance geometry problem with ranges. **Discrete Applied Mathematics**, v. 197, p. 3-19, 2015.

[11] HOARE, Charles A. R. Quicksort. **The Computer Journal**, v. 5, n. 1, p. 10-16, 1962.

[12] JADOON, Sultanullah; SOLEHRIA, Salman Faiz; QAYUM, Mubashir. Optimized selection sort algorithm is faster than insertion sort algorithm: a comparative study. **International Journal of Electrical and Computer Sciences**, v. 11, n. 2, p. 18-23, 2011.

[13] JADOON, Sultanullah et al. Design and analysis of optimized selection sort algorithm. **International Journal of Electrical and Computer Sciences**, v. 11, n. 1, p. 16-21, 2011.

[14] KRATSCH, Dieter et al. Certifying algorithms for recognizing interval graphs and permutation graphs. **SIAM Journal on Computing**, v. 36, n. 2, p. 326-353, 2006.

[15] KURLAK, John. **https://github.com/JohnKurlak/Algorithms/blob/ master/selection/Quickselect.java**. Visualizado em 04 de março de 2015.

[16] MAHMOUD, Hosam M. Distributional analysis of swaps in Quick Select. **Theoretical Computer Science**, v. 411, n. 16-18, p. 1763-1769, 2010.

[17] MARATHE, Madhav V. et al. Simple heuristics for unit disk graphs. **Networks**, v. 25, n. 2, p. 59-68, 1995.

[18] McCONNELL, Ross M. Certifying algorithms. **Computer Science Review**, v. 5, n. 2, p. 119-161, 2011.

[19] MEHLHORN, Kurt; NÄHER, Stefan. From algorithms to working programs: on the use of program checking in LEDA. In: MATHEMATICAL FOUNDATIONS OF COMPUTER SCIENCE, 23., 1998, Brno. **Proceedings...** Berlin: Springer, 1998, p. 84-93.

[20] MEHLHORN, Kurt; NÄHER, Stefan; UHRIG, Christian. The leda platform for combinatorial and geometric computing. In: DEGANO, P.; GORRIERI, R.; MARCHETTI-SPACCAMELA, A. (Ed.). **Automata, Languages and Programming**. Berlin: Springer, 1997. p. 7-16. (Lecture Notes in Computer Science, 1256).

[21] SULLIVAN, Gregory F.; MASSON, Gerald M. Certification trails for data structures. In: FAULT-TOLERANT COMPUTING, 21., 1991, Montreal. **Proceedings...** IEEE, 1991, 240-247.

[22] WU, Weili et al. Minimum connected dominating sets and maximal independent sets in unit disk graphs. **Theoretical Computer Science**, v. 352, n. 1-3, p. 1-7, 2006.

[23] XIE, Tao et al. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In: TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS, 11., 2005, Edimburgo. **Proceedings...** Berlin: Springer, 2005, p. 365-381.

APÊNDICE A CÓDIGO FONTE — RECONHECIMENTO DE GRAFOS BIPARTIDOS

Código-fonte (em linguagem Java) dos algoritmos utilizados para realizar os testes computacionais apresentados no Capítulo 3.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Main {
5
6      static String obtainOddCycle(int vertex1, int vertex2, int[]
           predecessors) {
7          List<Integer> path1 = new ArrayList<Integer>();
8          List<Integer> path2 = new ArrayList<Integer>();
9          while (vertex1 != vertex2) {
10             path1.add(vertex1);
11             vertex1 = predecessors[vertex1];
12             path2.add(vertex2);
13             vertex2 = predecessors[vertex2];
14             if (vertex1 == -1 || vertex2 == -1) {
15                 return "Nao foram encontrados ciclos impares";
16             }
17         }
18
19         /* Cria o ciclo */
20         List<Integer> oddCycle = new ArrayList<Integer>(path1);
21         oddCycle.add(vertex1);
22         for (int i = path2.size() - 1; i >= 0; i--) {
23             oddCycle.add(path2.get(i));
24         }
25
26         /* Converte a string */
27         StringBuffer sb = new StringBuffer();
28         for (int v : oddCycle) {
29             sb.append(v).append(',');
30         }
```

```
31          sb.deleteCharAt(sb.length() - 1);
32          String result = sb.toString();
33
34          return result;
35      }
36
37      static String obtainBipartition(int[] colors, int n) {
38
39          List<Integer> setColor0 = new ArrayList<Integer>();
40          List<Integer> setColor1 = new ArrayList<Integer>();
41
42          for (int i = 0; i < n; i++) {
43              if (colors[i] == 0) {
44                  setColor0.add(i);
45              }
46              if (colors[i] == 1){
47                  setColor1.add(i);
48              }
49              if (colors[i] != 0 && colors[i] != 1){
50                  return "Nao se trata de uma bicoloracao!";
51              }
52          }
53
54          StringBuffer ob = new StringBuffer();
55          ob.append("Cor 0: ");
56          for (int v : setColor0) {
57              ob.append(v).append(',');
58          }
59          ob.deleteCharAt(ob.length() - 1);
60          ob.append(';');
61          ob.append(" Cor 1: ");
62          for (int v : setColor1) {
63              ob.append(v).append(',');
64          }
65          ob.deleteCharAt(ob.length() - 1);
66          String result = ob.toString();
67
68          return result;
69      }
70
71
72      static AnswerAndCertificate
           runCertifyingAlgorithmForBipartite(Graph graph) {
```

```
73          String answer = null;
74          String certificate = null;
75
76          int n = graph.getNumberOfVertices();
77          int predecessors[] = new int[n];
78          int colors[] = new int[n];
79
80          for (int i = 0; i < n; i++) {
81              predecessors[i] = -1;
82              colors[i] = -1;
83          }
84
85          List<Integer> queue = new ArrayList<Integer>();
86          int v = 0;
87          colors[v] = 0;
88          queue.add(v);
89          int queueHeadIndex = 0;
90
91          while (queueHeadIndex < queue.size()) {
92              v = queue.get(queueHeadIndex);
93              queueHeadIndex++;
94              int thiscolor = colors[v];
95              int othercolor = 1 - colors[v];
96
97              for (int w : graph.getNeighbors(v)) {
98                  if (colors[w] != -1) {
99                      if (colors[w] != othercolor) {
100                         answer = "NAO";
101                         certificate = obtainOddCycle(v, w,
                               predecessors);
102                         return new AnswerAndCertificate(answer,
                               certificate);
103                     }
104                 } else {
105                     colors[w] = othercolor;
106                     predecessors[w] = v;
107                     queue.add(w);
108                 }
109             }
110         }
111
112         answer = "SIM";
113         certificate = obtainBipartition(colors, n);
```

```
114         return new AnswerAndCertificate(answer, certificate);
115     }
116
117     static boolean runVerifier(Graph graph, String answer,
            String certificate) {
118
119          // Grafo bipartido
120         if (answer == "SIM"){
121
122             int posicao = certificate.indexOf ("Cor 1: ");
123             String group1 = certificate.substring(9, posicao -
                    2);
124             String group2 = certificate.substring(posicao + 9);
125
126             String[] vertex1 =  group1.split(",");
127             List<Integer> neighbors1 = new ArrayList<Integer>();
128             int[] vertexInt1 = new int[vertex1.length];
129
130               for (int i = 0; i < vertex1.length; i++) {
131                 vertexInt1[i]= Integer.parseInt(String.valueOf(
                        vertex1[i]));
132             }
133
134             // verifica se os vertices com a mesma cor sao
                    adjacentes
135             for (int i = 0; i < vertexInt1.length; i++){
136
137                 neighbors1 = graph.getNeighbors(vertexInt1[i]);
138                 for (int w = 0; w < vertexInt1.length; w++){
139                     if (neighbors1.contains(vertexInt1[w])){
140                         return false;
141                     }
142                 }
143             }
144
145             String[] vertex2 =  group2.split(",");
146             List<Integer> neighbors2 = new ArrayList<Integer>();
147             int[] vertexInt2 = new int[vertex2.length];
148
149             for (int i = 0; i < vertex2.length; i++) {
150                 vertexInt2[i]= Integer.parseInt(String.valueOf(
                        vertex2[i]));
151             }
```

```
152
153             // verifica se os vertices com a mesma cor sao
                   adjacentes
154             for (int i = 0; i < vertexInt2.length; i++){
155
156                 neighbors2 = graph.getNeighbors(vertexInt2[i]);
157                 for (int w = 0; w < vertexInt2.length; w++){
158                     if (neighbors2.contains(vertexInt2[w])){
159                         return false;
160                     }
161                 }
162             }
163
164
165             return true;
166         }
167
168
169         if (answer == "NAO"){
170
171             String[] oddCycle = certificate.split(",");
172             int[] oddCycleInt = new int[oddCycle.length];
173             List<Integer> neighbors = new ArrayList<Integer>();
174
175             for (int i = 0; i < oddCycle.length; i++) {
176                 oddCycleInt[i]= Integer.parseInt(String.valueOf(
                       oddCycle[i]));
177             }
178
179             // verifica se eh um ciclo
180             for (int w = 0; w < oddCycleInt.length -1; w++){
181                 // verifica se o vertice pertence ao grafo
182                 if (w < 0 || w >= graph.getNumberOfVertices()) {
183                     return false;
184                 }
185
186             // verifica se os vertices consecutivos no ciclo sao
                   adjacentes no grafo
187
188                 neighbors = graph.getNeighbors(oddCycleInt[w]);
189                 if (w == 0){
190                     if (!neighbors.contains(oddCycleInt[
                           oddCycleInt.length -1])){
```

```
191                         return false;
192                     }
193                 }
194                 if (!neighbors.contains(oddCycleInt[w + 1])){
195                     return false;
196                 }
197             }
198 
199             if (oddCycle.length % 2 != 1){
200                 return false;
201             }
202         }
203 
204         return true;
205     }
206 
207 
208     public static void main(String[] args) {
209         Graph graph = Graph.readGraph();
210         AnswerAndCertificate answerAndCertificate =
               runCertifyingAlgorithmForBipartite(graph);
211         String answer = answerAndCertificate.getAnswer();
212         String certificate = answerAndCertificate.getCertificate
               ();
213         System.out.println("\nResposta: " + answer);
214         System.out.println("\nCertificado: " + certificate);
215         boolean resultOk = runVerifier(graph, answer,
               certificate);
216         if (resultOk) {
217             System.out.println("\nVerificacao ok!");
218         } else {
219             System.out.println("\nVerificacao falhou!");
220         }
221     }
222 
223     static class AnswerAndCertificate {
224         private String answer;
225         private String certificate;
226 
227         public AnswerAndCertificate(String answer, String
               certificate) {
228             this.answer = answer;
229             this.certificate = certificate;
```

```
230             }
231
232             public String getAnswer() {
233                 return answer;
234             }
235
236             public String getCertificate() {
237                 return certificate;
238             }
239         }
240     }
```

Programa para criar o grafo de entrada.

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.util.ArrayList;
4   import java.util.List;
5
6   public class Graph {
7
8       private int n;
9       private int m;
10      private List<List<Integer>> adjacencyLists;
11
12      public Graph(int n_vertices) {
13          this.n = n_vertices;
14          this.m = 0;
15          this.adjacencyLists = new ArrayList<List<Integer>>(
                n_vertices);
16          for (int i = 0; i < n_vertices; i++) {
17              this.adjacencyLists.add(new ArrayList<Integer>());
18          }
19      }
20
21      public void addEdge(int vertex1, int vertex2) {
22          List<Integer> vertex1neighbors = this.adjacencyLists.get
                (vertex1);
```

```
23          List<Integer> vertex2neighbors = this.adjacencyLists.get
                (vertex2);
24          if (!vertex1neighbors.contains(vertex2)) {
25              vertex1neighbors.add(vertex2);
26              vertex2neighbors.add(vertex1);
27              this.m++;
28          }
29      }
30
31      public int getNumberOfVertices() {
32          return this.n;
33      }
34
35      public List<Integer> getNeighbors(int v) {
36          return this.adjacencyLists.get(v);
37      }
38
39      public static Graph readGraph() {
40
41          InputStreamReader reader = new InputStreamReader(System.
                in);
42          BufferedReader in = new BufferedReader(reader);
43
44
45          int n = 0;
46          System.out.println("\nQuantos vertices? ");
47          boolean success = false;
48          while (!success) {
49              try {
50                  n = Integer.parseInt(in.readLine());
51                  success = true;
52              } catch (Exception e) {
53                  System.out.println("\nValor invalido.");
54              }
55          }
56
57
58          Graph graph = new Graph(n);
59
60
61          boolean readMoreEdges = true;
62          while (readMoreEdges) {
```

```
63                System.out.println("\nProxima aresta (formato: <
                     vertice1>,<vertice2>): ");
64                success = false;
65                while (!success) {
66                    try {
67                        String edgeStr = in.readLine();
68                        if (edgeStr.equals("")) {
69                            readMoreEdges = false;
70                            break;
71                        }
72                        int commaIndex = edgeStr.indexOf(',');
73                        int vertex1 = Integer.parseInt(edgeStr.
                             substring(0, commaIndex));
74                        int vertex2 = Integer.parseInt(edgeStr.
                             substring(commaIndex + 1));
75                        graph.addEdge(vertex1, vertex2);
76                        success = true;
77                    } catch (Exception e) {
78                        System.out.println("\nAresta invalida.");
79                    }
80                }
81            }
82
83            return graph;
84        }
85    }
```

APÊNDICE B CÓDIGO-FONTE — $K-$SELECTION

Código-fonte (em linguagem Java) dos algoritmos utilizados para realizar os testes computacionais apresentados no Capítulo 5.

```
1  import java.util.*;
2
3  public class QuickSelect {
4      public static void main(String[] args) {
5          run(args);
6      }
7      public static boolean run(String args[]) {
8          int k = 0;
9          int n = 0;
10         int[] inputList = {};
11         boolean verbose = false;
12         Map<Integer, Integer> inputElementsMap = new HashMap<
               Integer, Integer>();
13
14     for (int i = 0; i < args.length; i++) {
15                 if (args[i].equals("-k")) {
16                     k = Integer.valueOf(args[++i]);
17                 } else if (args[i].equals("-random")) {
18                     n = Integer.valueOf(args[++i]);
19                     List<Integer> list = new ArrayList<>(n);
20                     inputList = new int[n];
21                     Random random = new Random();
22                     for (int j = 0; j < n; j++) {
23                         int x = random.nextInt(Integer.MAX_VALUE
                               );
24                         list.add(x);
25                         inputList[j] = x;
26                         int occurrences = 0;
27                         if (inputElementsMap.containsKey(x)) {
28                             occurrences = inputElementsMap.get(x
                                   );
29                         }
```

```
30                      inputElementsMap.put(x,  occurrences +
                           1);
31                  }
32                  if (verbose) {
33                  Collections.sort(list);
34                  System.out.println(list.toString());
35              }
36          } else if (args[i].equals("-list")) {
37              n = Integer.valueOf(args[++i]);
38              inputList = new int[n];
39              for (int j = 0; j < n; j++) {
40                  inputList[j] = Integer.valueOf(args[++i]);
41              }
42              countOccurrences(inputList, inputElementsMap);
43          } else if (args[i].equals("--verbose")) {
44              verbose = true;
45          }
46      }
47
48      int[] outputList = new int[k];
49
50
51      System.out.println("\nExecutando o k-selection...");
52      long start = System.nanoTime();
53      kSelection(inputList, outputList, k);
54      long end = System.nanoTime();
55      long duration = end - start;
56      System.out.println(String.format("Fim. Tempo
           transcorrido: %.8f milissegundos.", (1.0 * duration /
            1000000)));
57
58      System.out.println("Os elementos retornados sao: ");
59      for (int i = 0; i < outputList.length; i++) {
60          System.out.print(outputList[i] + " ");
61      }
62
63      System.out.println("\nExecutando a verificacao...");
64      start = System.nanoTime();
65      boolean verificationResult = verify(inputList,
           outputList, k, inputElementsMap);
66      end = System.nanoTime();
67      duration = end - start;
```

```
68          System.out.println(String.format("Fim. Tempo
               transcorrido: %.8f milissegundos.", (1.0 * duration /
                1000000)));
69          if (!verificationResult) {
70              System.out.println("Verificacao falhou!!!");
71              return false;
72          }
73          System.out.println("Verificacao ok!");
74          return true;
75      }
76
77      public static boolean verify(int[] inputList, int[]
           outputList, int k, Map<Integer, Integer> inputElementsMap
           ) {
78
79          // 1: Verifica o tamanho da lista de saida
80          if (outputList.length > k) {
81              System.out.println("Foram retornados mais do que " +
                    k + "elementos!");
82              return false;
83          }
84          if (outputList.length < k) {
85              System.out.println("Foram retornados menos do que "
                   + k + "elementos!");
86              return false;
87          }
88
89          // Descobre qual elemento eh o maior entre aqueles que
               foram retornados
90          int kthElement = outputList[0];
91          for (int x : outputList) {
92              if (x > kthElement) {
93                  kthElement = x;
94              }
95          }
96
97          // 2: Verifica se todos os elementos que sao menores do
               que ou igual ao k-esimo na lista de entrada aparecem
               na lista de saida
98
99          Map<Integer, Integer> outputElementsMap = new HashMap<
               Integer, Integer>(outputList.length);
100         countOccurrences(outputList, outputElementsMap);
```

```
101
102         for (int i = 0; i < inputList.length; i++){
103             int element = inputList[i];
104             if (element <= kthElement) {
105                 if (!outputElementsMap.containsKey(element)) {
106                     System.out.println("O elemento " + element +
                            ", menor do que o " + k + "-esimo na
                           lista de entrada, nao aparece na lista
                           retornada!");
107                     return false;
108                 }
109             }
110         }
111            // 3: Verifica se:
112            // (I) o numero de ocorrencias de cada elemento da
                  lista de saida menor do que o k-esimo eh igual a
                  sua multiplicidade na lista de entrada; e
113
114            // (II) o numero de ocorrencias do k-esimo elemento
                  nao excede sua multiplicidade na lista de entrada
115
116         for (int i = 0; i < outputList.length; i++) {
117             int element = outputList[i];
118
119             int outputElementCount = 0;
120             if (outputElementsMap.containsKey(element)) {
121                 outputElementCount = outputElementsMap.get(
                        element);
122             }
123
124             int inputElementCount = 0;
125             if (inputElementsMap.containsKey(element)) {
126                 inputElementCount = inputElementsMap.get(element
                        );
127             }
128
129             if (element != kthElement) {
130                 if (outputElementCount != inputElementCount) {
131                        System.out.println("O elemento " + element
                               + ", menor do que o " + k +
132                        "-esimo, aparece com multiplicidade " +
                           outputElementCount +
```

```
133                     " na saida do algoritmo; no entanto, sua
                           multiplicidade na lista de entrada eh " +
134                     inputElementCount + ".");
135                      return false;
136                 }
137             } else {
138                 if (outputElementCount > inputElementCount) {
139                        System.out.println("O elemento " + element
                               +
140                     ", que foi retornado como sendo o " + k +
141                     "-esimo menor, aparece com multiplicidade " +
                            outputElementCount +
142                     " na saida do algoritmo, maior, portanto, que
                            sua multiplicidade na lista de entrada,
                           que eh " +
143                      inputElementCount + ".");
144                      return false;
145                 }
146             }
147         }
148
149         return true;
150     }
151
152     public static void countOccurrences(int[] list, Map<Integer,
            Integer> ocurrencesMap) {
153         for (int i = 0; i < list.length; i++) {
154             int inputElement = list[i];
155             int count = 0;
156             if (ocurrencesMap.containsKey(inputElement)) {
157                 count = ocurrencesMap.get(inputElement);
158             }
159             count++;
160             ocurrencesMap.put(inputElement, count);
161         }
162     }
163
164    // Encontra os k menores elementos de uma lista
165     public static void kSelection(int[] inputList, int[]
           outputList, int k) {
166         quickselect(inputList, 0, inputList.length - 1, k);
167         int count = 0;
168         for (int i = 0; i < k; i++) {
```

```
169             int inputListElement = inputList[i];
170                 outputList[count++] = inputListElement;
171         }
172     }
173
174     // Encontra o k-esimo elemento (quick select)
175     public static Integer quickselect(int[] list, int k) {
176         return quickselect(list, 0, list.length - 1, k);
177     }
178
179   // Parte recursiva do quick select
180     public static Integer quickselect(int[] list, int leftIndex,
        int rightIndex, int k) {
181
182         if (k < 1 || k > list.length) {
183             return null;
184         }
185
186         if (leftIndex == rightIndex) {
187             return list[leftIndex];
188         }
189
190         int pivotIndex = randomPartition(list, leftIndex,
            rightIndex);
191         int sizeLeft = pivotIndex - leftIndex + 1;
192
193          if (sizeLeft == k) {
194             return list[pivotIndex];
195         } else if (sizeLeft > k) {
196             return quickselect(list, leftIndex, pivotIndex - 1,
                k);
197         } else {
198             return quickselect(list, pivotIndex + 1, rightIndex,
                 k - sizeLeft);
199         }
200     }
201
202
203     public static int randomPartition(int[] list, int leftIndex,
        int rightIndex) {
204         int pivotIndex = medianOf3(list, leftIndex, rightIndex);
205         int pivotValue = list[pivotIndex];
206         int storeIndex = leftIndex;
```

```
207
208         swap(list, pivotIndex, rightIndex);
209
210         for (int i = leftIndex; i < rightIndex; i++) {
211             if (list[i] <= pivotValue) {
212                 swap(list, storeIndex, i);
213                 storeIndex++;
214             }
215         }
216
217         swap(list, rightIndex, storeIndex);
218         return storeIndex;
219     }
220
221
222     public static int medianOf3(int[] list, int leftIndex, int
        rightIndex) {
223         int centerIndex = (leftIndex + rightIndex) / 2;
224
225         if (list[leftIndex] > list[rightIndex]) {
226             swap(list, leftIndex, centerIndex);
227         }
228
229         if (list[leftIndex] > list[rightIndex]) {
230             swap(list, leftIndex, rightIndex);
231         }
232
233         if (list[centerIndex] > list[rightIndex]) {
234             swap(list, centerIndex, rightIndex);
235         }
236
237         swap(list, centerIndex, rightIndex - 1);
238         return rightIndex - 1;
239     }
240
241     public static void swap(int[] list, int index1, int index2)
        {
242         int temp = list[index1];
243         list[index1] = list[index2];
244         list[index2] = temp;
245     }
246 }
```