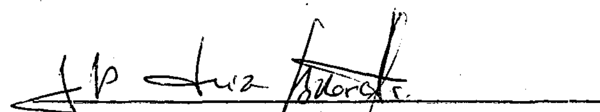
UM ESTUDO SOBRE TÉCNICAS DE
BUSCA EM GRAFOS E SUAS APLICAÇÕES

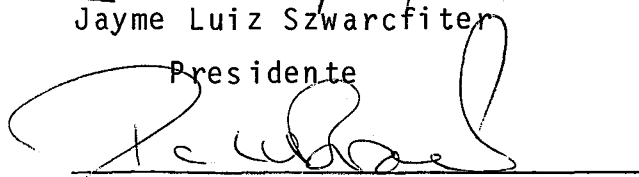
Mirian Picinini Mexas

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.)

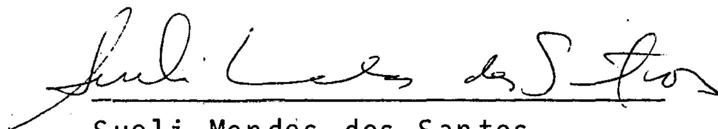
Aprovada por:



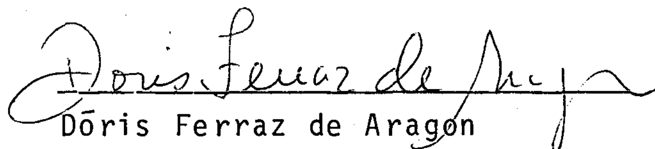
Jayme Luiz Szwarcfiter
Presidente



Paulo Oswaldo Boaventura Netto



Sueli Mendes dos Santos



Dóris Ferraz de Aragon

RIO DE JANEIRO, RJ - BRASIL
DEZEMBRO DE 1982

MÉXAS, MIRIAN PICININI

Um Estudo Sobre Técnicas de Busca em Grafos e suas Aplicações (Rio de Janeiro) 1982.

IX, 258 p. 29,7 cm (COPPE-UFRJ, M.Sc. Engenharia de Sistemas e Computação, 1982).

Tese - Univ. Fed. do Rio de Janeiro, Fac. Engenharia.

1. Técnicas de Busca em Grafos. I. COPPE/UFRJ II. Título (série).

Aos Meus Pais

AGRADECIMENTO

Agradeço ao professor Jayme Luiz Szwarcfiter a orientação dispensada durante o desenvolvimento desta tese, incentivando, apoiando e esclarecendo sempre, de boa vontade, as dúvidas que surgiram durante a elaboração da mesma.

RESUMO

A presente tese tem como objetivo fazer um estudo sobre técnicas de busca em grafos, juntamente com algumas aplicações. Neste aspecto são examinadas a busca em profundidade, em largura e casos especiais, como a busca lexicográfica. Buscas não restringidas ou parcialmente restringidas são também consideradas. As principais propriedades, algoritmos e características de cada caso são estudadas em detalhe. Como aplicações, são apresentados diversos problemas cujas soluções são fortemente baseadas nas técnicas acima descritas.

ABSTRACT

The aim of this thesis is to study the techniques of graph searching, together with some of its applications. In this aspect depth-first search, breadth-first search and some special cases as lexicographic search are examined. Non restricted or partially restricted search are also considered. The main properties, algorithms and characteristics of each case are studied in detail. As applications, many problems are presented whose solutions are heavily based on the above described techniques.

ÍNDICE

CAPÍTULO I - Introdução Geral 1

CAPÍTULO II - Definições Básicas 5

CAPÍTULO III - Busca em Profundidade 13

1) Introdução 13

2) Idéia Básica 14

3) Busca em Profundidade em Grafos Não Direcionados 15

3.1) Exemplo 15

3.2) Árvore Geradora de Busca em Profundidade 17

3.3) Algoritmos para Busca em Profundidade 18

3.4) Propriedades 24

4) Busca em Profundidade em Grafos Direcionados 26

4.1) Exemplo 28

4.2) Árvore Geradora Direcionada de Busca em Profundidade 30

4.3) Algoritmo para Busca em Profundidade em Grafos Direcionados 31

4.4) Propriedades 33

5) Conclusão 35

CAPÍTULO IV - Busca em Largura 37
1) Introdução 37
2) Idéia Básica 38
3) Busca em Largura em Grafos Não Direcionados 39
3.1) Exemplo 40
3.2) Árvore Geradora de Busca em Largura 42
3.3) Algoritmos para Busca em Largura 43
3.4) Propriedades 47
4) Busca em Largura em Grafos Direcionados .. 49
4.1) Exemplo 50
4.2) Árvore Geradora Direcionada de Busca em Largura 51
4.3) Algoritmo para Busca em Largura em Grafos Direcionados 52
4.4) Propriedades 55
5) Conclusão 57
CAPÍTULO V - Casos Especiais 59
1) Introdução 59
2) Classificação dos Algoritmos de Busca 60
2.1) Algoritmos Totalmente Restringidos ... 60
2.2) Algoritmos Não Restringidos 61
2.3) Algoritmos Parcialmente Restringidos 62
3) Busca em Largura Lexicográfica 63
3.1) Algoritmo 63
3.2) Propriedades 65

3.3) Exemplos 66
4) Busca em Profundidade Lexicográfica 70
4.1) Algoritmo 70
4.2) Propriedades 71
4.3) Exemplos 72
5) Conclusão 76
CAPÍTULO VI - Aplicações 78
1) Introdução 78
2) Componentes Conexos 79
3) Componentes Biconexos 82
4) Componentes Fortemente Conexos 87
5) Ordenação Topológica 92
6) Componentes Fracos 96
7) Todas as Ordenações Topológicas 100
8) Fechamento Transitivo 103
9) Todos os Caminhos de um Grafo 107
10) Todos os Caminhos Maximais de um Grafo ... 109
11) Ciclos Simples de um Grafo 112
12) Menor Caminho em Digrafos Acíclicos Valorados 116
13) Menor Caminho em Grafos Não Valorados 119
14) Maior Caminho em Digrafos Acíclicos Valorados 123
15) Grafos Bipartite 126
16) Dominadores 128
17) Planaridade 137
18) Representação Vetorial da Alcançabilidade em Grafos Direcionados Planares 147

19) Isomorfismo em Grafos 153

20) Digrafos Estruturados em Árvore 161

21) Isomorfismo de Busca em Profundidade 173

22) Isomorfismo de Digrafos Redutíveis em Árvore 178

23) Conjunto de Corte Mínimo em Digrafos Redutíveis 184

24) Redutibilidade de Grafos de Fluxo 195

25) Orientação de um Grafo Não Direcionado ... 201

26) Ordenação de Eliminação Minimal 209

27) Reconhecimento de Grafos Triangulares 213

28) Reconhecimento de Grafos Totais 219

29) Vértices de Realimentação 229

30) Fatores Par e Ímpar 238

31) Conclusão 244

CAPÍTULO VII - Conclusão Geral 246

REFERÊNCIAS BIBLIOGRÁFICAS 248

CAPÍTULO I

INTRODUÇÃO GERAL

O assunto a que se propõe esta Tese, "Técnicas de Busca de Grafo", é pertencente à área de Teoria Computacional dos grafos. Esta área é um ramo da Ciência da Computação que tem por objetivo resolver problemas algoritmicos em Teoria dos Grafos. Uma das razões para o crescimento do interesse nesse ramo é a grande difusão do uso dos computadores. Podemos citar alguns textos em português sobre Teoria dos grafos: LUCCHESI[52], BOAVENTURA NETTO[10], BARBOSA[4], SAVULECU[64]. FURTADO[29] é um outro texto em português, específico de algoritmos em grafos.

O objetivo deste trabalho é fazer um estudo detalhado sobre "Técnicas de Busca em Grafo", juntamente com aplicações. Uma técnica de Busca em Grafos é uma maneira de explorar um grafo dado. Serão expostos e discutidos critérios eficientes que são úteis para alcançar tal objetivo.

No Capítulo II são apresentadas as definições básicas que são relevantes para um bom entendimento dos capítulos posteriores. As nomenclaturas e notações usadas são tiradas especialmente de HARARY[35] e HARARY, NORMAN & CARTWRIGHT[36].

Um dos critérios de Busca em Grafos que será estudado no Capítulo III é a poderosa técnica conhecida como "Busca em Profundidade" ou "Backtracking". Este método foi formalizado e usado por HOPCROFT e TARJAN, em 1971, e foi estudado com detalhe por TARJAN (em TARJAN[71]). Todavia, o algoritmo para este

critério não é assim tão novo. Ele já era conhecido no século XIX como uma técnica para atravessar labirintos. Por exemplo, podemos pesquisar o relatório de LUCAS do trabalho de TRÉMAUX (em LUCAS[51]). Um outro algoritmo que foi sugerido mais tarde por TARRY (em TARRY[78]), é bom para atravessar labirintos e, de fato, a "Busca em Profundidade" é um caso especial dele. Podemos também observar o uso de "Backtraking" para resolver problemas de grafos antes da década de 70, em ROBERTS & FLORES[61]. Esta Técnica de "Busca em Profundidade" irá garantir que o grafo inteiro será examinado, e reconhecerá também quando o algoritmo irá parar, sem perder muito tempo durante a busca. Em resumo, ela faz o seguinte: visita um vértice arbitrário x; em seguida explora uma aresta (x,y) incidente em x. O próximo vértice a ser examinado é aquele que é "adjacente ao vértice mais recentemente visitado", incidente a alguma aresta não explorada. E assim continua nessa busca recursiva até visitar todos os vértices. O algoritmo para este tipo de busca terá complexidade linear.

Outra técnica que também é eficiente e será discutida no Capítulo IV, é a denominada "Busca em Largura". Este método funciona da seguinte maneira: visita um vértice arbitrário x; o próximo vértice a ser explorado será aquele que é "adjacente ao vértice menos recentemente visitado". A complexidade do algoritmo de Busca em Largura é também linear.

No Capítulo V são considerados alguns Casos Especiais. Um deles é a "Classificação dos Algoritmos de Busca" em três tipos diferentes: i) Totalmente Restringido; ii) Parcialmente Restringido e iii) Não Restringido. Cada um desses algoritmos

será explicado com detalhe, juntamente com sua complexidade. Os outros dois casos especiais são denominados: "Busca em Profundidade Lexicográfica" e "Busca em Largura Lexicográfica". O que há de novo nessas duas buscas é a maneira de como os vértices são selecionados. Eles não são escolhidos aleatoriamente como nos Capítulos III e IV, e sim obedecem uma determinada ordem lexicográfica.

Várias "aplicações" das Técnicas de Busca estudadas nos Capítulos anteriores serão apresentadas no Capítulo VI. Essas "aplicações" são importantes pois representam diferentes tipos de problemas em grafos que podem ser resolvidos utilizando os métodos de buscas aqui apresentados. Para cada aplicação serão descritos alguns novos conceitos em grafos, algoritmos com respectivas complexidades e exemplos explicativos.

Os algoritmos apresentados nesta tese serão escritos em uma linguagem de formato livre, mantendo contudo uma certa estrutura do ALGOL. É bom ressaltar que o interesse crescente em Computações de Teoria dos Grafos tem levado ao desenvolvimento de várias linguagens de programação, com o único propósito de manipular grafos. O principal objetivo de uma tal linguagem é permitir ao usuário formular operações em grafos de uma maneira compacta e natural. Como exemplo dessas linguagens podemos citar:

1) Graph-Theoretic Language (GTPL), University of West Indies, Jamaica. É uma extensão de FORTRAN. (Ver READ[59]).

2) Graph Algorithm Software Package (GASP), University of Illinois. É uma extensão de PL/1. (Ver CHASE[11]).

3) HINT, Michigan State University. É uma extensão de LISP 1.5. (Ver HART[37]).

4) Graphic Extended ALGOL (GEA), Instituto di Elettrotecnico ed Elettronica del Politecnico di Milano, Italy. É uma extensão de ALGOL. (Ver CRESPI-RECHIZZI & MORPURGO[12] e CRESPI-RECHIZZI & MORPURGO[13]).

5) FORTRAN Extended Graph Algorithmic Language (FGRAAL) University of Maryland (Ver BASILI, MESZTENYI & REINBOLDT[5]).

CAPÍTULO II

DEFINIÇÕES BÁSICAS

O que se pode observar, em Teoria dos Grafos, é que há diferenças de nomenclatura e notação de um autor para outro. Por exemplo, um determinado símbolo pode se referir a dois conceitos diferentes quando usado por dois autores diferentes. Então tornou-se necessário especificar claramente as definições básicas que são relevantes para o bom entendimento dos próximos Capítulos, e que também têm o objetivo de evitar dupla interpretação.

As nomenclaturas e notações utilizadas são baseadas principalmente nas referências: HARARY[35] e HARARY, NORMAN & CARTWRIGHT[36].

Um grafo (V,E) é constituido por um conjunto finito não vazio V, juntamente com um conjunto E de pares de elementos de V. Os elementos de V são os vértices e os elementos de E as arestas. Notaremos $n = |V|$ e $m = |E|$, ou seja, n é o número de vértices e m é o número de arestas de um grafo. Os grafos aqui considerados não têm laços nem arestas múltiplas, ou seja, não existe nenhuma aresta unindo um vértice a ele mesmo e não existe mais de uma aresta unindo dois vértices, respectivamente.

Um grafo direcionado (ou digrafo) $D = (V,E)$ é um grafo em que as arestas são pares ordenados. Um grafo não direcionado (ou simplesmente grafo) $G = (V,E)$ é um grafo em que as ares

tas são pares não ordenados.

Por abuso e para maior simplicidade, notaremos uma aresta a pelo par de vértices (v,w), tanto para grafos não direcionados como para digrafos. Nas definições que se seguem, se nada for dito explicitamente sobre o tipo de grafo utilizado, elas serão comuns a grafos e a digrafos.

Dada uma aresta a = (v,w), v e w são adjacentes, e a é incidente a ambos v e w. Também diremos que cada vértice v,w de a é incidente à aresta considerada a. O grau de um vértice v é o número de vértices que são adjacentes a v.

Em um digrafo, uma aresta (v,w) é dita sair de v e ir para w. O grau de entrada de um vértice z é o número de arestas que vão para z e o grau de saída o número de arestas que saem de z. Notaremos por grauent(v) e grausai(v) o grau de entrada e saída respecticamente. Um vértice fonte é um vértice v com grauent(v) = 0 e um vértice sumidouro v tem grausai(v) = 0. As arestas que vão para um vértice v são denominadas arestas convergentes e as que saem de v são arestas divergentes.

Um caminho de v_1 a v_k é uma sequência de vértices $v_1, v_2, \ldots, v_k$ tal que para cada i, $1 \leq i < k$, temos $(v_i, v_{i+1}) \in E$. O comprimento do caminho $v_1, v_2, \ldots, v_k$ é definido como k - 1. O vértice v_1 é dito alcançar v_k; e v_k é alcançável a partir de v_1. Obviamente cada vértice de um grafo é alcançável a ele mesmo. Um caminho trivial é composto por um único vértice. Um caminho é simples se todos os v_i são distintos.

Um ciclo é um caminho v_1, v_2, ..., v_k com $v_k = v_1$ e contendo pelo menos duas arestas diferentes. Um ciclo $v_1, v_2, \ldots, v_k - 1$,

v_k é simples se $v_1, v_2, \ldots, v_{k-1}$ é um caminho simples. Se um grafo não tem ciclos ele é chamado acíclico. Dois ciclos simples envolvendo exatamente as mesmas arestas são considerados idênticos.

Um grafo completo é um grafo que tem um número máximo de arestas, para o dado conjunto de vértices. Um digrafo acíclico completo é um digrafo acíclico em que a adição de qualquer aresta nova, entre os dois de seus vértices, cria um ciclo.

Um grafo ponderado (ou valorado) é um grafo em que existe um número natural d_{vw} associado a cada uma de suas arestas (v,w). Dado um caminho $v_1, v_2, \ldots, v_k$ em um grafo ponderado (ou valorado) definimos seu comprimento de caminho ponderado (ou valorado) como a soma dos valores das arestas que formam o caminho. Por convenção, o comprimento de caminho ponderado de um caminho trivial é zero e se não existe caminho do vértice v para w, diremos que o comprimento de caminho ponderado de v para w é ∞. Quando utilizamos grafos ponderados, podemos usar a terminologia "comprimento de caminho", como referência a "comprimento de caminho ponderado".

Dois grafos são isomorfos se existe entre seus vértices uma correspondência um a um que preserva adjacência.

Se direcionarmos as arestas de um grafo G não direcionado, obtemos um digrafo D chamado uma orientação de G. O grafo G é então chamado grafo subjacente a D.

Um conjunto é maximal (minimal) em relação a uma propriedade P quando ele possui aquela propriedade e não é subconjunto de nenhum conjunto que possua aquela propriedade.

Um grafo (V_2,E_2) é um subgrafo do grafo (V_1,E_1) quando $V_2 \subseteq V_1$ e $E_2 \subseteq E_1$. (V_1,E_1) é um supergrafo de (V_2,E_2). Se adicionalmente $V_1 = V_2$, então (V_2,E_2) é um subgrafo gerador de (V_1,E_1) e (V_1,E_1) é um supergrafo gerador de (V_2,E_2). O grafo (V_2,E_2) é chamado subgrafo induzido de (V_1,E_1) se para qualquer $v,w \in V_2$, $(v,w) \in E_1$ implica $(v,w) \in E_2$. Uma clique do grafo é um subgrafo completo.

Seja D um digrafo. Seu subgrafo gerador mínimo e supergrafo gerador máximo que preservam a alcançabilidade de D são a redução transitiva e fechamento transitivo de D, respectivamente. O fechamento transitivo de D é também chamado conjunto parcialmente ordenado (poset) induzido por D.

Um grafo não direcionado $G = (V,E)$ é conexo se existe um caminho entre cada par de vértices do grafo. Se G não é conexo ele é dito desconexo. Um grafo sem arestas é chamado totalmente desconexo. Dado $v \in V$, $a \in E$, notaremos por $G - v$ o grafo obtido de G pela retirada de G do vértice v e todas as arestas incidentes a ele, e por $G - a$ o grafo obtido de G pela retirada da aresta a. Se G é conexo e $G - v$ é desconexo, então v é um vértice de articulação. Um grafo conexo $G = (V,E)$ com $|V| > 2$ e sem vértices de articulação é dito biconexo. Os componentes conexos de G são os subgrafos conexos maximais de G. Da mesma forma, os componentes biconexos de G são os subgrafos biconexos maximais de G.

Um digrafo $D = (V,E)$ é fortemente conexo se para cada (v,w), $v,w \in V$, existe um caminho de v para w. Um digrafo D é unilateralmente conexo se para qualquer par de vértices (v,w), existe um caminho de v para w ou de w para v. Todo digrafo fortemente conexo é unilateralmente conexo.

Uma ordenação topológica de D é uma sequência $v_1, v_2, \ldots, v_n$ de todos os seus vértices, tal que se v_i alcança v_j então $i < j$.

Uma árvore é um grafo conexo e acíclico. Um conjunto de árvores disjuntas é chamado uma floresta. Uma árvore enraizada T é uma árvore em que selecionamos um vértice arbitrário chamado raiz. Se (v,w) é uma aresta de uma árvore enraizada, sendo v mais próximo da raiz do que w, então v é o pai de w e w um filho de v. Notaremos por Pai(w) e Filho(v) o pai de w e o filho de v, respectivamente. Se existe um caminho de v a w em uma árvore enraizada, tal que v está mais próximo da raiz do que w , então v é um ancestral de w e w é um descencente de v. O conjunto de ancestrais e descentendes de v será denotado por ances(v) e desc(v), respectivamente. Os vértices de ances(v) - {v} são os ancestrais próprios de v e os vértices de desc(v) - {v} os descendentes próprios de v. Um vértice diferente da raiz e de grau 1 numa árvore enraizada é chamado de folha ou vértice terminal. Um vértice que não é uma folha é chamado vértice interior. O nível de um vértice v em uma árvore enraizada é igual a um mais o comprimento do caminho da raiz para v. Portanto, o nível da raiz é um. Notaremos por NIV(v) o nível de v.

Um digrafo D é enraizado se existe um vértice, chamado raiz do digrafo, que alcança todos os outros vértices de D. Uma árvore enraizada direcionada é um digrafo acíclico em que exatamente um vértice, a raiz, tem grau de entrada zero, enquanto que os outros vértices têm grau de entrada um. Os conceitos já vistos para árvore enraizada serão também válidos para árvore enraizada direcionada.

Uma subárvore (subárvore enraizada) T_1 de uma árvore (árvore enraizada) T é um subgrafo de T, de modo que T_1 seja uma árvore (árvore enraizada).

Uma árvore geradora de um grafo não direcionado G é um subgrafo gerador de G que é uma árvore. Assim um grafo não direcionado G tem uma árvore geradora se e somente se ele é conexo, caso contrário o conjunto de árvores geradoras de seus componentes conexos define uma floresta geradora de G.

Muitos algoritmos que fazem uso de árvores enraizadas necessitam visitar os vértices da árvore em alguma ordem. As visitas comuns são: pré-ordem, pós-ordem, ordem simétrica.

Visita em Pré-Ordem:

1. Visitar a raiz r.
2. Visitar em pré-ordem as subárvores com raízes $v_1, v_2, \ldots, v_K$ (filhos de r), nessa ordem.

Visita em Pós-Ordem:

1. Visitar em pós-ordem as subárvores com raízes em $v_1, v_2, \ldots, v_k$, nessa ordem.
2. Visitar a raiz r.

Visitar em Ordem Simétrica:

1. Visitar em ordem simétrica a subárvore a esquerda.
2. Visitar a raiz r.
3. Visitar em ordem simétrica a subárvore a direita.

OBS: As visitas mais utilizadas nesta tese serão em pré-ordem.

Um _lista_ é uma sequência finita de objetos. Frequentemente os algoritmos envolvem inserção e retirada de um elemento da lista (no começo, no meio ou no fim). Para percorrer uma lista em ambas as direções usamos uma _lista duplamente ligada_ (encadeada). Uma _pilha_ é uma lista em que os objetos só podem ser inseridos ou retirados após o último elemento (TOPO). Uma _fila_ é uma lista em que os objetos são inseridos em uma extremidade e retirados da outra.

A _matriz de adjacência_ é uma matriz N x N, tal que cada elemento b_{ij} é definido com $b_{ij} = 1$ se $(v_i, v_j) \in E$ e $b_{ij}=0$ caso contrário. A _matriz de alcançabilidade_ é uma matriz N x N, onde cada elemento b_{ij} é tal que $b_{ij} = 1$ se o vértice v_i alcança v_j, e $b_{ij} = 0$ caso contrário. Para um grafo não direcionado, a _matriz de incidência_ é uma matriz N x M, com cada elemento b_{ij} definido como $b_{ij} = 1$ se a aresta a_j é incidente ao vértice v_i e $b_{ij} = 0$ caso contrário.

Dado um grafo (V,E), define-se para cada vértice $v \in V$ uma _lista de adjacência_ Adj(v) do seguinte modo: se $(v,w) \in E$ então $w \in$ Adj(v). O conjunto dessas listas de adjacência, uma para cada vértice, é chamado _estrutura de adjacência_.

Existem diferentes maneiras de representar um grafo em um computador. Por exemplo, as matrizes de adjacência ou incidência podem ser usadas para armazenar um grafo definido como uma matriz. Mas a forma que é usual e mais conveniente de representar um grafo é através de um conjunto de listas de adjacência denominado estrutura de adjacência.

A medida de eficiência de um algoritmo é denominada _Complexidade_. A _complexidade de tempo_ (_espaço_) de um algoritmo

é o tempo (espaço) requerido pelo algoritmo expresso em função do tamanho da entrada do problema em seu pior caso. O comportamento no limite da complexidade de um algoritmo quando o tamanho do problema aumenta é denominado complexidade de tempo (espaço) assintótica. As execuções dos algoritmos propostos nesta tese são avaliadas em termos de expressões em notação - O para complexidades de tempo e espaço assintóticas dos algoritmos. Seja f uma função definida para as variáveis $n_1, n_2, \ldots, n_K$. Diz-se que f é $O(h)$, denotando-se por $f = O(h)$, quando existir uma constante $c > 0$ tal que $f(n_1, n_2, \ldots, n_K) \leq c\,h$ (Ver KNUTH[47]).

Algoritmo polinomial é um que apresenta uma solução correta em um tempo T, limitado por um polinômio no comprimento da entrada, quaisquer que sejam seus dados. Existem muitos problemas para os quais os melhores algoritmos são muito vagarosos, requerendo $O(2^n)$ tempo. Alguns desses problemas têm sido realmente provados ter limites exponenciais. Outros pertencem a uma classe chamada problemas NP - completo. Apenas para ilustração, um problema NP-completo admite algoritmo exponencial para resolvê-lo, mas nada se pode afirmar, até o momento, sobre a existência de um algoritmo polinomial. Para uma definição e estudo desta classe de problema ver GAREY & JOHNSON[30].

CAPÍTULO III

BUSCA EM PROFUNDIDADE

1) INTRODUÇÃO

O objetivo deste capítulo é descrever detalhadamente sobre uma das Técnicas de Busca em Grafos, que é conhecida como BUSCA EM PROFUNDIDADE ou BACKTRACKING.

Inicialmente será explicada a Idéia Básica desta Técnica de Busca. Como um grafo pode ser não direcionado ou direcionado, será necessário dividir esta Busca em duas partes:

- Busca em Profundidade em Grafos Não Direcionados;
- Busca em Profundidade em Grafos Direcionados.

A idéia básica de Busca em Profundidade é a mesma para essas duas divisões, mas a árvore geradora resultante será diferente para cada caso. Também serão apresentados exemplos explicativos, algoritmos com respectivas complexidades e propriedades importantes.

A técnica de BACKTRACKING tem sido comumente usada e descrita como uma estratégia importante para resolver vários problemas computacionais em Teoria dos Grafos. E de fato, ela se torna assim tão útil pois além da simplicidade da idéia, o algoritmo de busca é linear em tempo e espaço.

2) IDÉIA BÁSICA

A Busca em Profundidade em um grafo qualquer, G=(V,E), é feita da seguinte maneira:

Inicialmente, seleciona-se e visita-se um vértice s ∈ V (vértice inicial). Então explora-se qualquer aresta (s,w) incidente em s, visita-se w.

Como passo geral, um vértice qualquer w a ser explorado, que seja adjacente ao vértice mais recentemente visitado v, poderá já ter sido visitado antes ou não. Se w já foi visitado, então retorna-se a v e escolhe-se uma outra aresta incidente em v, e ainda não explorada. Se w não foi visitado, visita-se w e aplica-se o processo recursivamente sobre w. Após completar a busca através de todas as arestas incidentes em w, retorna-se a v, o vértice do qual w foi alcançado primeiro.

Este processo de selecionar arestas não exploradas incidentes em v continua até que a lista dessas arestas acabe. Assim, a busca em profundidade irá terminar quando voltamos a s (vértice inicial) e não encontramos nenhuma aresta a ser explorada.

Veremos, a seguir, dois tipos de Busca em Profundidade: uma para grafos não direcionados e outra para Grafos Direcionados.

Referências para pesquisa: TARJAN[71]; BAASE[3]; GOLUMBIC[33]; AHO, HOPCROFT & ULLMAN[1]; DEO[14]; DEO[17]; REINGOLD, NIEVERGELT & DEO[60]; EVEN[26].

3) BUSCA EM PROFUNDIDADE EM GRAFOS NÃO-DIRECIONADOS

Se G = (V,E) é um grafo conexo não direcionado, então todo vértice será visitado e toda aresta será explorada uma só vez em ambas as direções (isto é, duas vezes).

Se G não é conexo, então uma busca é feita para cada componente conexo.

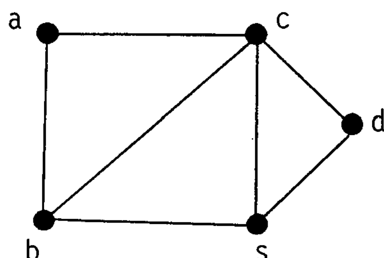
3.1) EXEMPLO

A figura (III.1) ilustra como a busca em profundidade examina um grafo não direcionado G = (V,E), representado por uma estrutura de adjacência.

Começamos com a escolha de um vértice arbitrário s. De s exploramos a primeira aresta encontrada, que é (s,b). Como b é um vértice que nunca foi visitado antes, nós ficamos em b e buscamos a primeira aresta não explorada encontrada em b, que é (b,c). Agora, no vértice c, a primeira aresta não explorada que encontramos é (c,s). Nós buscamos (c,s) e encontramos s que foi previamente visitado. Assim, retornamos a c, visitando a aresta (c,s) de alguma maneira (como uma linha tracejada na figura (III.1)) para poder distingui-la das arestas como (b,c), que levam a novos vértices. Voltando ao vértice c, observamos uma outra aresta não explorada e buscamos a primeira que encontramos que é (c,a). Novamente, como a é um vértice novo, nós ficamos em a e examinamos uma aresta não explorada, neste caso (a,b). Como b já foi previamente visitado, nós visitamos a aresta (a,b) com uma linha tracejada e retor-

namos a a. Assim, como não existe aresta não explorada em a, voltamos a c. Em c encontramos uma aresta não-explorada (c,d), que nós agora exploramos. Como d é um vértice novo, nós permanecemos em d e examinamos uma aresta não explorada em d, que é (d,s). Como s é um vértice já visitado, nós visitamos (d,s) com uma linha tracejada. Agora d está completamente examinado e assim retornamos a c. Em c não encontramos arestas não exploradas. Então, voltamos para b e finalmente voltamos para s. Todas as arestas incidentes em s já foram exploradas, o que significa que este processo aqui termina.

Grafo Dado G = (V,E)



Estrutura de Adjacência:

s → b, c, d

a → b, c

b → c, a, s

c → s, a, b, d

d → s, c

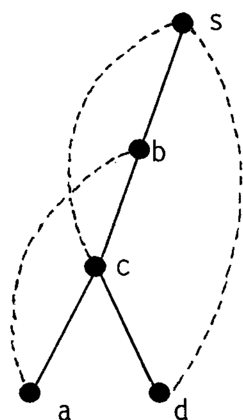


Figura III.1 - Estrutura de Busca em Profundidade.

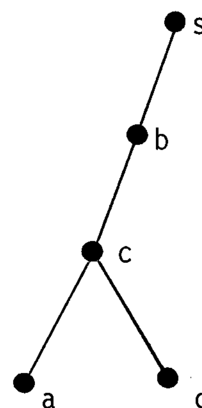


Figura III.2 - Árvore Geradora de Busca em Profundidade T

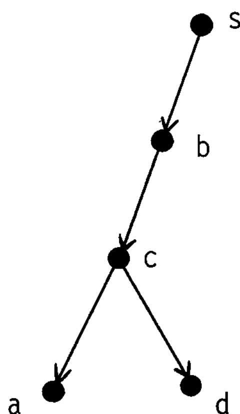


Figura III.3 - Árvore geradora direcionada de Busca em Profundidade T'.

3.2) ÁRVORE GERADORA DE BUSCA EM PROFUNDIDADE

A Busca em Profundidade gera uma árvore enraizada T, que é composta por arestas que levam a novos vértices durante a busca, e que são mostradas com linhas sólidas na figura (III.1). Essa árvore geradora enraizada T, que resulta da aplicação da técnica de Busca em Profundidade em um grafo não-direcionado, conexo G = (V,E) será classificada como Árvore Geradora de Busca em Profundidade. Caso o grafo G não seja conexo, obteremos uma Floresta Geradora de Busca em Profundidade.

As arestas de G que pertencem à Árvore Geradora de Busca em Profundidade T, são chamadas arestas de árvore; e as arestas de G que não estão em T são denominadas arestas de retorno (na figura(III.1) são representadas por linhas tracejadas). A estrutura formada por esses dois tipos de arestas é denominada Estrutura de Busca em Profundidade (Figura(III.1)).

Podemos direcionar a árvore T da raiz até as folhas, e assim obteremos uma Árvore Geradora Direcionada de Busca em Pro

fundidade T' (Ver Figura(III.3)).

3.3) ALGORITMOS PARA BUSCA EM PROFUNDIDADE

3.3.1.) O processo de busca em profundidade pode ser facilmente descrito por um simples algoritmo recursivo:

Algoritmo A:

```
        Procedimento BPR(v);
          Início
1.          Visitar e marcar v;
2.        Enquanto existir um vértice desmarcado w adjacen-
                 te a v efetuar
3.                      BPR(w);
          Fim.
```

Em um procedimento recursivo a estrutura de chamadas pode ser representada por uma árvore com raiz, onde cada vértice representa uma chamada recursiva para o procedimento. A ordem em que as chamadas são executadas corresponde a uma busca em profundidade na árvore. Seja, por exemplo, a definição recursiva da sequência de Fibonacci: $F(0) = 0$, $F(1) = 1$, e para $n \geq 2$, $F(n) = F(n-1) + F(n-2)$. A estrutura de chamadas para uma computação recursiva de F_5 é mostrada na figura (III.4). Cada vértice é rotulado com o valor corrente de n, isto é, o n para o qual a subárvore enraizada naquele vértice computa Fn. A or

dem da execução das chamadas recursivas está indicada na figura (III.4). Esta ordenação de vértices na árvore é também conhecida como pré-ordem (Obs.: Sabemos que é muito ineficiente computar os números de Fibonacci recursivamente; este exemplo é usado somente para ilustrar a conexão entre busca em profundidade e recursão).

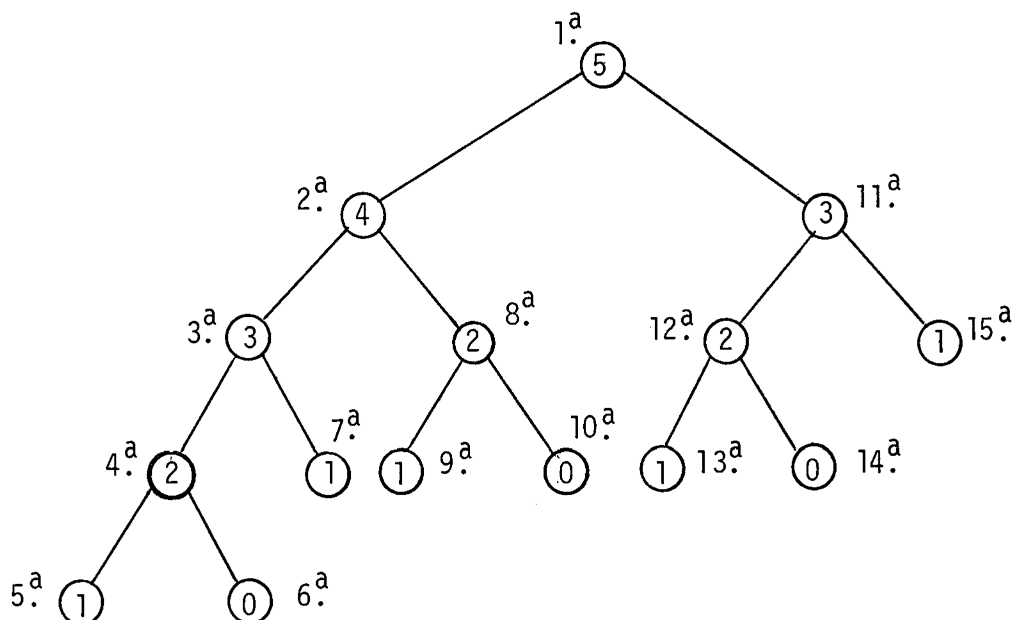


Figura (III.4)

3.3.2) Para implementar a busca em profundidade, nós precisamos fazer distinção entre vértices que ainda não foram visitados e vértices que foram visitados. Isto pode ser feito numerando sucessivamente os vértices de 1 a $|V|$ quando os visitamos. Inicialmente, fazemos num(v) = 0, para todo $v \in V$, para indicar que nenhum vértice foi visitado. Quando visitamos um vértice v pela primeira vez, fazemos num(v) receber um valor diferente de zero. Tal busca em profundidade é dada pelo algoritmo B. O procedimen-

to recursivo BP(v) realiza a busca em profundidade no grafo G = (V,E), contendo v, e constrói uma árvore geradora de Busca em Profundidade T para o grafo; u é pai de v na árvore. Se o grafo G não é conexo, T será uma floresta.

Algoritmo B: para busca em profundidade em um grafo, usando a procedure recursiva BP(v).

Entrada: Um grafo não-direcionado G = (V,E) representado por uma estrutura de adjacência, onde Adj(v) = conjunto de vértices adjacentes a v.

Saída: Uma partição de E em um conjunto T de arestas de árvore e um conjunto R de arestas de retorno.

```
        Procedimento BP(v);
        Início
1.          i ← i + 1;
            num(v) ← i;     /Marcou v "visitado"/.

2.          Para    w ∈ Adj(v)  efetuar

3.          Se  num(w) = 0   /Se w é marcado "não-visitado"/
                então  Início
4.                           [(v,w) é uma aresta de árvore]
                              T ← T ∪ {(v,w)};
5.                            BP(w);
                       Fim
                senão
6.                  Se num(w) < num(v)  e  w ≠ u
                       então  Início
7.                                  [(v,w) é uma aresta de retorno]
                                    R ← R ∪ {(v,w)};
                              Fim
            retorne
        Fim

        Início      /Programa Principal/
8.      Para    x ∈ V  efetuar num(x) ← 0;  /Marque x "não visitado"/

9.      i ← 0;
        T ← R ← ϕ;
10.     Para   x ∈ V  efetuar Se num(x) = 0  então
                                                BP(x);
        Fim
```

Nota: Se num(w) = 0, então a aresta (v,w) é uma aresta de árvore, ao passo que se num(w) ≠ 0, o teste para determinar se (v,w) é uma aresta de retorno é "num(w) < num(v) e w ≠ u". Pois, se w = u então (v,w) é necessariamente aresta de árvore. Se num(w) > num(v), então a aresta de retorno (v,w) já foi explorada, quando w era o topo da pilha.

Complexidade: Como uma chamada para BP é feita exatamente uma vez para cada vértice visitado recentemente, BP é chamada |V| vezes. Para cada chamada, a quantidade de trabalho feita é proporcional ao número de arestas incidentes em um tal vértice. Portanto o tempo e espaço requeridos para uma busca em profundidade em um grafo não direcionado é $O(|V| + |E|)$.

3.3.3) BP(v) é um exemplo de um procedimento recursivo e pode ser implementado usando uma pilha. Quando uma chamada para ela mesma é feita, os valores correntes de todas as variáveis locais para o procedimento e a linha do procedimento, que faz a chamada, são armazenados no topo da pilha. Desta maneira, quando o controle é retornado, a computação pode continuar onde havia parado. Ver algoritmo C.

Algoritmo C:

Entrada: Grafo não direcionado $G = (V,E)$; $v \in V$, o vértice do qual a busca começa.

Notação: P é uma pilha, inicialmente vazia. O vértice no topo da pilha é referido por Topo.

1. Visitar, marcar e colocar v na pilha P.

2. Enquanto P é não-vazia efetuar

3. Enquanto existir um vértice desmarcado w adjacente ao topo

4. efetuar visitar, marcar e colocar w na pilha
   Fim /w é agora topo/

5. Retirar da Pilha P.

Fim.

3.4) PROPRIEDADES

Em geral, um grafo pode ter muitas árvores geradoras de busca em profundidade. De fato existe, por exemplo, uma liberdade de escolha dos vértices da linha 2 no Algoritmo B, como também do vértice inicial.

Uma estrutura de Busca em Profundidade tem algumas importantes e úteis propriedades:

(P_1) - Se v é um ancestral próprio de w em T <u>então</u> num(v) < num(w).

(P_2) - Para cada aresta (v,w) de G, seja aresta de árvore ou aresta de retorno, v será um ancestral de w ou vice-versa.

Prova - Se (v,w) é uma aresta de árvore então a prova é imediata pela própria definição de aresta de árvore.

Seja (v,w) uma aresta de retorno.

Suponhamos que v é visitado antes de w (sem perda de generalidade), no sentido que BP(v) é chamada antes de BP(w). Assim, quando v é alcançado, w ainda está com NUM igual a zero (w não foi alcançado). Todos os vértices novos visitados por BP(v), se tornarão descendentes de v na estrutura de Busca em Profundidade. Mas BP(v) não pode terminar até que w seja alcançado, desde que w esteja na lista de adjacência de v.

(P_3) - Seja T uma árvore geradora de Busca em Profundidade de um grafo não direcionado $G=(V,E)$. Então todo caminho C em G contém um vértice p tal que todos os vértices em C são descendentes de p em T.

Prova - Sejam: C um caminho em G e p o vértice de C mais próximo da raiz s em T.

Suponhamos por absurdo que C contém um vértice que não é descendente de p. Então, necessariamente, C contém uma aresta (v,w) tal que $w \in desc(p)$ e $v \notin desc(p)$. Obviamente $v \notin desc(w)$; senão v estaria em desc(p). Se, por outro lado, $w \in desc(v)$ então p e v estão no caminho de s a w em T. Nesse caso a única possibilidade, de acordo com as hipóteses acima, seria p estar em desc(v), o que contradiz o fato de p ser o vértice de C mais próximo de s. Assim, $v \notin desc(w)$ e $w \notin desc(v)$ o que significa que (v,w) não é uma aresta de árvore, nem uma aresta de retorno, o que é uma contradição.

4) BUSCA EM PROFUNDIDADE EM GRAFOS DIRECIONADOS

A ideia básica para se fazer uma busca em profundidade em grafos direcionados é a mesma para busca em profundidade para grafos não-direcionados. Contudo, quando aplicamos tal busca a grafos direcionados (digrafos), a estrutura resultante será mais complicada do que um conjunto de "arestas de árvore" e um conjunto de "arestas de retorno" que já foram vistas para grafos não-direcionados. Esta complicação ocorre porque as arestas em um digrafo já estão orientadas e nós não temos liberdade de buscá-las na direção de nossa escolha (conforme foi feito para busca em profundidade em um grafo não-direcionado).

O que devemos observar é que na busca em profundidade em um grafo direcionado D = (V,E) as arestas serão particionadas em quatro categorias:

1º - Arestas de Árvore

São arestas que levam a novos vértices durante a busca.

2º - Arestas de Avanço

São arestas que vão de ancestrais para descendentes, mas não são arestas de árvore.

3º - Arestas de Retorno

São arestas que vão de descendentes para ancestrais.

4º - Arestas de Cruzamento

São as que estão entre vértices que não são nem ancestrais e nem descendentes um do outro.

Os algoritmos A e C, que já vimos anteriormente, também poderão ser utilizados para busca em profundidade em grafos direcionados.

Considere agora o que irá acontecer quando buscamos as arestas de um digrafo D, ao longo de suas orientações, durante uma busca em profundidade em D. Como já vimos antes, nós determinamos um número em série, num(x), para cada x, a primeira vez que o visitamos. Esses números são permanentemente designados para os vértices e indicam a ordem em que os mesmos foram visitados. Quando encontramos uma aresta (v,w) que ainda não foi buscada, w pode ter sido ou não previamente visitado. Se w não foi previamente visitado, a aresta (v,w) é marcada como uma aresta de árvore, exatamente como no grafo não-direcionado. Se w já foi previamente visitado, então w pode ser ou não um ancestral de v, em contraste com o caso não-direcionado em que w era sempre um ancestral de v. Se w é um ancestral próprio de v, então claramente num(w) < num(v) e (v,w) é uma aresta de retorno. Se w não é um ancestral de v e num(w)>num(v), então w deve ser um descendente de v e a aresta (v,w) é chamada uma aresta de avanço; ela representa um caminho alternado de v para w. Por outro lado, se num(w) < num(v) e w não é nem ancestral e nem descendente de v, então a aresta (v,w) é chamada uma aresta de cruzamento.

4.1) EXEMPLO

A figura (III.5) mostra uma busca em profundidade em um digrafo e a partição resultante de arestas nos subconjuntos já vistos anteriormente. O digrafo é representado por uma estrutura de adjacências, e a busca em profundidade é iniciada a partir de um vértice a, escolhido arbitrariamente. Os números escritos entre parenteses, ao lado da Figura (III.5), representam valores num, isto é, a ordem em que os vértices foram visitados.

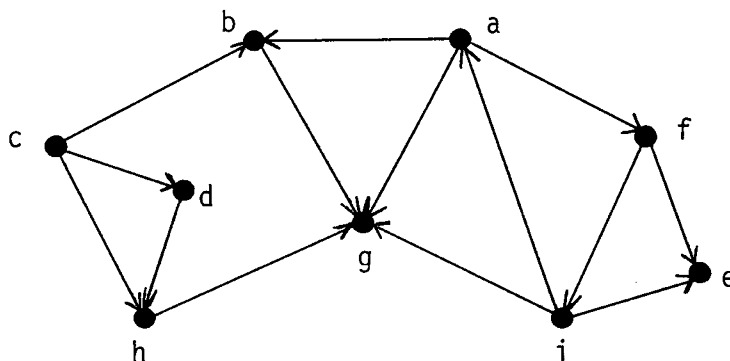
——> arestas de árvore

----> arestas de retorno

-·-·-> arestas de avanço

·····> arestas de cruzamento

Digrafo D:



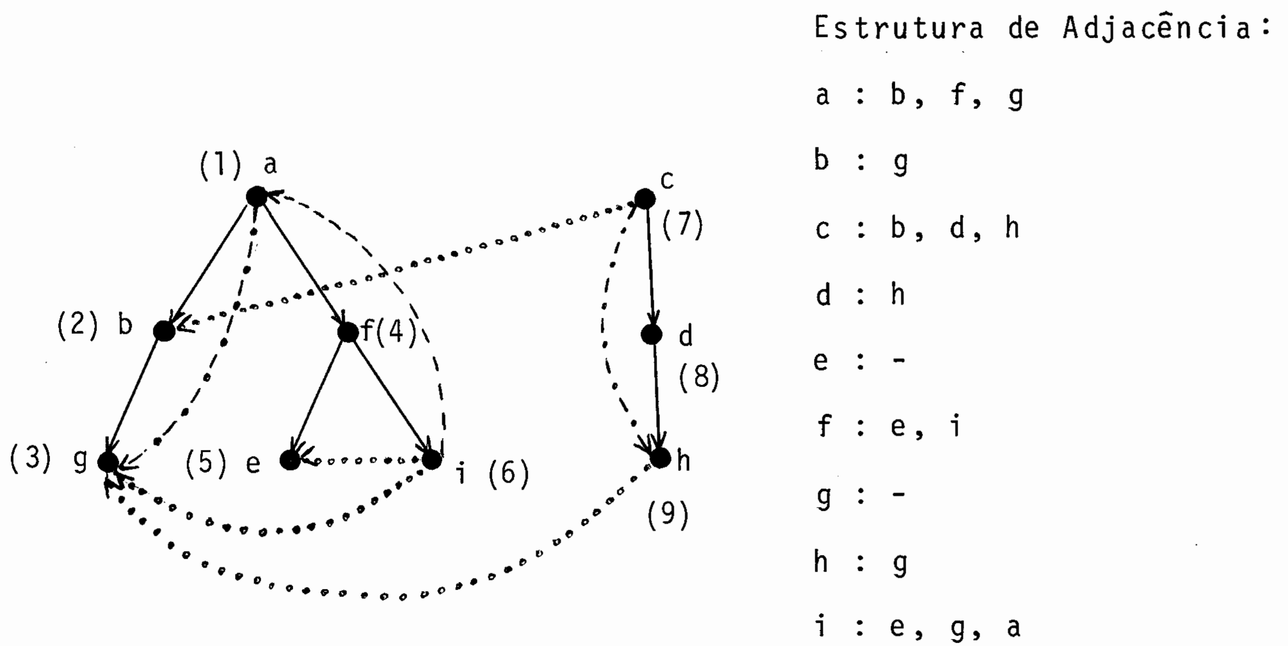


Figura III.5 - Estrutura Direcionada de Busca em Profundidade.

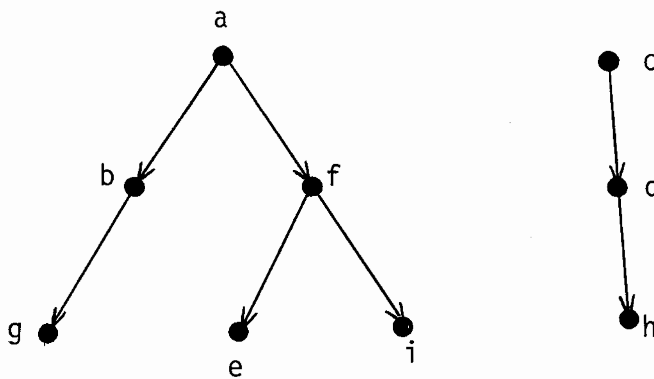


Figura III.6 - Floresta Geradora Direcionada de Busca em Profundidade.

4.2) ÁRVORE GERADORA DIRECIONADA DE BUSCA EM PROFUNDIDADE

A Busca em Profundidade em um digrafo D = (V,E) gera uma árvore (ou floresta) enraizada direcionada T que é composta por arestas de árvore. Essa árvore T é denominada Árvore (ou Floresta) Geradora Direcionada de Busca em Profundidade. (Figura (III.6)).

A estrutura obtida da aplicação da técnica de Busca em Profundidade em um digrafo D = (V,E), e que é formada por arestas de árvore, arestas de retorno, arestas de avanço e arestas de cruzamento, é denominada Estrutura Direcionada de Busca em Profundidade. (Figura (III.5)).

OBS.: A árvore (ou floresta) gerada por uma busca em profundidade não é única, e a classificação das arestas depende:

a) da ordem arbitrária na qual os vértices e arestas do digrafo original são dadas na estrutura de adjacência, e

b) da escolha do vértice inicial.

4.3) ALGORITMOS PARA BUSCA EM PROFUNDIDADE EM GRAFOS DIRECIONADOS.

Algoritmo D:

OBS.: Variável m - denota o último NUM designado para algum vértice.

Variável n - denota o último PNUM designado para algum vértice.

ND(v) - conta o número de descendentes de cada vértice v na árvore geradora T.

NUM(v) - fornece a ordem em que o vértice v é explorado durante a busca.

NUM(v)=0 sse v não foi explorado.

PNUM(v) - fornece a v um número, quando v é retirado da pilha.

PNUM(v)=0 sse v ainda não foi retirado da pilha.

Entrada: Grafo direcionado $D = (V,E)$, representado pela estrutura de adjacência ADJ(v), para $v \in V$.

Saída: Uma partição de E em um conjunto T de arestas de árvore, um conjunto R de arestas de Retorno, um conjunto A de arestas de avanço e um conjunto C de arestas de cruzamento.

```
Procedimento BPD(v);
Início
      m ← NUM(v) ← m + 1;
      ND(v) ← 1;
      Para w ∈ ADJ(v) efetuar
            Se NUM(w) = 0 então
                 Início
                       [(v,w) é uma aresta de árvore]
                       T ← T ∪ {(v,w)};
                       BPD(w);
                       ND(v) ← ND(v) + ND(w);
                 Fim

                 senão Se PNUM(w) = 0 então
                       Início    /Vértice w está na pilha/
                        [(v,w) é uma aresta de retorno]
                        R ← R ∪ {(v,w)};
                       Fim

                       senão Se NUM(v) < NUM(w) então
                               Início
                                [(v,w) é uma aresta de avanço]
                                 A ← A ∪ {(v,w)};
                               Fim
                                senão
                                Início
                                [(v,w) é uma aresta de cruzamento];
                                C ← C ∪ {(v,w)};
                                Fim
      n ← PNUM(v) ← n - 1;    /v foi retirado da pilha
Fim
```

```
Início   /Programa Principal/
    m ← 0;
    n ← |V| + 1;
    T ← R ← A ← C ← φ;
    Para cada vértice x efetuar NUM(x) ← PNUM(x) ← 0;
    Para x ∈ V efetuar Se NUM(x) = 0 então
                                             BPD(x);
Fim
```

Complexidade: O tempo e espaço requeridos para uma busca em profundidade em um grafo direcionado será também $O(|V| + |E|)$, pelo mesmo motivo já visto anteriormente para grafos não direcionados.

4.4) PROPRIEDADES

(P_1) - Suponha que todos os vértices de um grafo direcionado D são alcançáveis do vértice s, e que as arestas de D são divididas em quatro categorias usadas no Algoritmo D. Então:

i) As arestas de árvore formam uma árvore direcionada T com raiz s que contém todos os vértices em D.

ii) Se (v,w) é uma aresta de retorno, então $NUM(w) < NUM(v)$ e existe um caminho de w para v em T.

iii) Se (v,w) é uma aresta de avanço, então existe um caminho de v para w em T.

iv) Se (v,w) é uma aresta de cruzamento, então não existe um caminho de v para w em T e também não existe um caminho de w para v em T.

(P_2) - Se (v,w) é uma aresta de árvore, ou uma aresta de avanço, ou uma aresta de cruzamento então $PNUM(v) < PNUM(w)$. Se (v,w) é uma aresta de retorno então $PNUM(v) > PNUM(w)$.

(P_3) - Seja v um vértice em D. Então o número de descendentes de v na árvore geradora T é dado por

$$ND(v) = 1 + \sum_{(v,w)} ND(w)$$

(P_4) - As afirmações abaixo: (i), (ii), (iii) e (iv) são equivalentes.

i) Existe um caminho de v para w em T.

ii) $NUM(v) \leq NUM(w) < NUM(v) + ND(v)$.

iii) $PNUM(v) \leq PNUM(w) < PNUM(v) + ND(v)$.

iv) $NUM(v) \leq NUM(w)$ e $PNUM(v) \leq PNUM(w)$.

A propriedade (P_4) nos dá três métodos para identificar os descendentes de um vértice, e permite-nos identificar arestas de retorno, avanço e cruzamento, se desejarmos.

(P_5) - D é acíclico se somente se D não tem arestas de retorno.

Prova - Se D tem uma aresta de retorno (v,w), então a aresta de retorno e o conjunto de arestas de árvore unindo v e w formam um ciclo. Se D não tem arestas de retorno, todas as arestas (v,w) satisfazem: PNUM(v) < PNUM(w). Como qualquer ciclo em D deve ter pelo menos uma aresta (v,w) com PNUM(v) > PNUM(w), então D não tem ciclos.

(P_6) - Seja p um caminho de v para w.
Suponha que NUM(v) < NUM(w). Então p contém algum ancestral comum de v e w em T. (Ver prova em TARJAN[72]).

5) CONCLUSÃO

Em resumo podemos dizer que a Busca em Profundidade em um grafo não direcionado G = (V,E) irá particionar as arestas de E em dois conjuntos T e R. Uma aresta (v,w) é colocada no conjunto T se o vértice w não tiver sido previamente visitado, quando estamos no vértice v, considerando a aresta (v,w). Caso contrário a aresta (v,w) é colocada no conjunto R. As arestas que estão em T são chamadas arestas de árvore, e as que estão em R são chamadas arestas de retorno.

Por sua vez, a Busca em Profundidade em um grafo direcionado D = (V,E) irá particionar as arestas de E em quatro conjuntos: T, R, A, C, que são, respectivamente conjuntos de ares

tas de árvore, arestas de retorno, arestas de avanço e arestas de cruzamento.

Esses dois tipos de busca serão resolvidos com complexidade $O(|V| + |E|)$, o que significa que a técnica de Busca em Profundidade é eficiente.

CAPÍTULO IV

BUSCA EM LARGURA

1) INTRODUÇÃO

O método de Busca a ser estudado, neste presente capítulo, é conhecido como "Busca em Largura", que é também uma boa técnica utilizada para explorar um determinado grafo. Tal grafo poderá ser não direcionado ou direcionado, daí a necessidade de dividirmos este método em duas partes:

- Busca em Largura em Grafos Não-Direcionados;
- Busca em Largura em Grafos Direcionados.

Para cada uma dessas duas divisões, serão descritos: Algoritmos com respectivas complexidades, Exemplos que auxiliam no entendimento da técnica e Propriedades que são importantes.

A idéia básica de Busca em Largura é muito simples e o algoritmo é eficiente pois tem complexidade linear em tempo e espaço.

2) IDÉIA BÁSICA

Na *Busca em Largura* os vértices são examinados de acordo com uma ordenação dada pelo número de arestas dos menores caminhos entre eles e um vértice arbitrário v, mas fixo. São examinados todos os vértices de uma mesma distância em cada passo.

Em outras palavras, na *Busca em Largura* selecionamos um vértice e o colocamos em uma fila inicialmente vazia de vértices a serem visitados. Repetidamente removemos o vértice *x* da cabeça da fila e então inserimos na fila todos os vértices adjacentes a *x* que nunca foram colocados na fila. Este processo continua a ser executado até que a fila se torne vazia, o que significa que a Busca em Largura terminou.

Como no caso de Busca em Profundidade, a Busca em Largura (BL) é executada uma única vez para cada componente conexo do grafo. Portanto, nesta Busca, cada vértice é visitado somente uma vez.

A Busca em Largura será dividida em duas partes:

- Busca em Largura em Grafos Não-Direcionados;
- Busca em Largura em Grafos Direcionados.

Referências para pesquisa: BAASE[3], DEO[17], GOLUMBIC[33].

3) BUSCA EM LARGURA EM GRAFOS NÃO-DIRECIONADOS

Uma Busca em Largura em um grafo não-direcionado G = (V,E) particiona o conjunto de arestas em quatro categorias:

1º - Arestas de Árvore (——>)- são arestas que levam a novos vértices durante a busca.

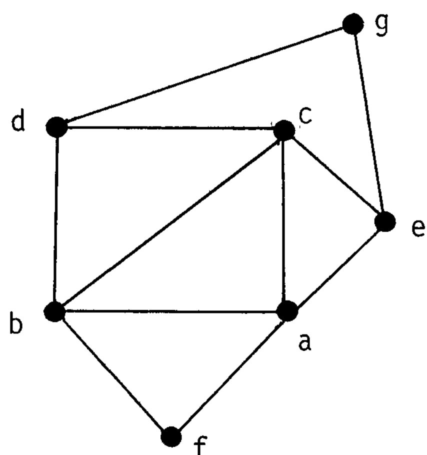
2º - Arestas entre tio e sobrinho (-.-.->) - a aresta (x,y) é uma aresta entre tio e sobrinho se |NIV(y) - NIV(x)| = 1 e (x,y) não é aresta de árvore.

3º - Arestas entre irmãos (⊙⊙⊙⊙⊙⊙ >) - a aresta (x,y) é dita ser aresta entre irmãos se Pai(x) = Pai(y) e NIV(x) = NIV(y).

4º - Arestas entre primos (*****>) - a aresta (x,y) é dita ser aresta entre primos se NIV(x) = NIV(y) e (x,y) não é aresta entre irmãos.

3.1) EXEMPLO

A figura (IV.1) ilustra como a busca em largura examina o grafo não-direcionado G = (V,E), representado por uma estrutura de adjacência:



Estrutura de Adjacência:

a : b, c, e, f

b : a, d, c, f

c : a, b, d, e

d : b, c, g

e : a, c, g

f : a, b

g : d, e

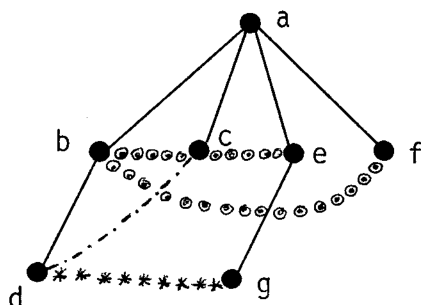


Figura IV.1 - Estrutura de Busca em Largura

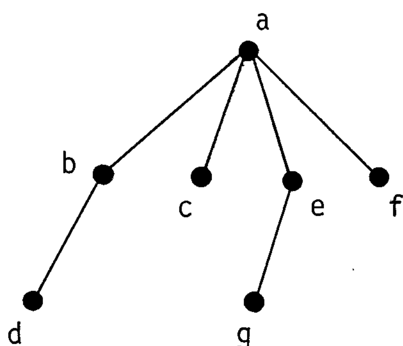


Figura IV.2 - Árvore Geradora de Busca em Largura T

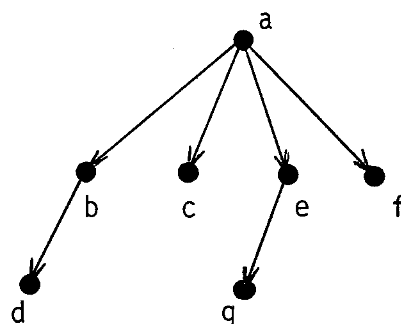


Figura IV.3 - Árvore Geradora Direcionada de Busca em Largura T'

Começamos a Busca em Largura escolhendo um vértice arbitrário a e colocando-o em uma fila inicialmente vazia. Removemos a da cabeça da fila e introduzimos na fila todos os vértices adjacentes a a, que nunca foram colocados na fila que são: b, c, e, f. Assim exploramos as arestas (a,b), (a, c), (a, e) e (a, f), com linhas sólidas (arestas de árvore). O vértice b está agora na cabeça da fila. Removemos b e colocamos na fila o vértice d, adjacente a b, (que nunca foi colocado na fila) e exploramos a aresta (b,d) (com uma linha sólida - aresta de árvore). Os vértices c e f são adjacentes a b, mas eles já estavam na fila. Como Pai(b) = Pai(c) e NIV(b)=NIV(c), a aresta (b,c) será explorada como aresta entre irmãos; Temos que NIV(b) = NIV(f) e Pai(b) = Pai(f), logo (b,f) é explorada como aresta entre irmãos. O vértice a é também é adjacente a b, mas como a já foi retirado da fila, não iremos explorar nenhuma aresta, pois nesse caso significa que a aresta (a,b) já foi explorada antes. Agora temos que c está na cabeça da fila.

Retiramos c da fila e exploramos a aresta (c,d) como aresta entre tio e sobrinho pois |NIV(d) - NIV(c)| = 1 e (c,d) não é aresta de árvore. Depois exploramos a aresta (c,e), como aresta entre irmãos pois Pai(c) = Pai(e) e NIV(c) = NIV(e). Estamos em e e o retiramos da fila. O vértice g é adjacente a e e nunca foi explorado. Assim introduzimos g na fila e exploramos a aresta (e, g) como aresta de árvore. Os demais vértices adjacentes a e, que são a e c já foram explorados e retirados da fila. Retiramos então f da cabeça da fila. Como todos os vértices adjacentes a f já foram explorados e retirados da fila, teremos para cabeça da fila o vértice d. Removendo d iremos explorar a aresta (d,g) como aresta entre primos pois NIV(d) = NIV(g) e (d, g) não é aresta entre irmãos. Os vértices b e c adjacentes a d e os vértices d e e adjacentes a g já foram explorados e retirados da fila; logo retiramos g da fila e assim a fila se torna vazia, o que significa que a Busca em Largura terminou.

3.2) ÁRVORE GERADORA DE BUSCA EM LARGURA

A Busca em Largura gera uma Árvore enraizada T, que é composta por arestas que levam a novos vértices durante a busca, e que são mostradas com linhas sólidas na figura (IV.1). Essa árvore geradora enraizada T, que resulta da aplicação da técnica de Busca em Largura em um grafo não direcionado G=(V,E), conexo, será classificada como Árvore Geradora de Busca em Largura (Figura(IV.2)). Caso o grafo G não seja conexo, obteremos uma Floresta Geradora de Busca em Largura.

As arestas de G que pertencem à Árvore Geradora de Busca em Largura T, são chamadas arestas de árvore; e as arestas de G que não estão em T são denominadas arestas entre irmãos (◎◎◎◎◎◎>), arestas entre primos (******>) e arestas entre tio e sobrinho (-.-.->). A Estrutura formada por essas quatro arestas é denominada Estrutura de Busca em Largura (Figura (IV.1)).

Podemos direcionar a árvore T da raiz até as folhas, e assim obteremos uma Árvore Geradora Direcionada de Busca em Largura (Ver Figura (IV.3)).

3.3) ALGORITMOS PARA BUSCA EM LARGURA

A técnica de Busca em Largura aplicada em um grafo não direcionado pode ser descrita pelo algoritmo abaixo.

Algoritmo E:

Entrada: Um grafo não direcionado $G = (V,E)$ representado pela estrutura de adjacência $ADJ(v)$, para $v \in V$.

Saída: Uma partição de E nos seguintes conjuntos: T de arestas de árvore; S de arestas entre tio e sobrinho; I de arestas entre irmãos; e P de arestas entre primos.

Método: Todos os vértices são inicialmente marcados "nunca enfileirado". O Procedimento BL é usado para visitar um vértice. Um Vetor NBL registra a ordem em que os vértices são enfileirados e visitados.

```
      Procedimento BL(x)
      Início

1.    Para       y ∈ Adj(x)  efetuar
2.          Se  NBL(y) = 0   /y está marcado "nunca enfileirado"/
              então  Início
3.                      Adicione a aresta (x,y) a T;
4.                      NIV(y) ← NIV(x) + 1;
5.                      Pai(y) ← x;
6.                      Adicione y a F;
7.                      i ← i + 1;
                        NBL(y) ← i;  /Marque y "enfileirado"/
                     Fim
              senão
8.               Se  NIV(y) - NIV(x) = 1
9.                 então  Adicione a aresta (x,y) a S.
                   senão
10.                     Se  NIV(y) - NIV(x) = 0
11.                       então Se  Pai(x) = Pai(y)
12.                                então Adicione (x,y) a I;
13.                                senão Adicione (x,y) a P;
                          Fim
        Retorne
      Fim
```

```
     Início     /Programa Principal/

14.  T ← S ← I ← P ← φ ; F ← Fila Vazia;
15.  i ← 0;
16.      Para    v ∈ V  efetuar
17.            NBL(v) ← 0    /Marque v "nunca enfileirado"/
18.       Enquanto F é vazia e existe v ∈ V marcado "nunca en
                   fileirado" Efetuar
19.       Início    Adicione v  a  F;
20.                 NIV(v) ← 0; Pai(v) ← 0;
21.                 i ← i + 1;
22.                 NBL(v) ← i; /Marque v "enfileirado"/
23.                 Enquanto  F  não é vazia  Efetuar
24.                 Início    x ← cabeça de F;
25.                           F ← F - x;
26.                           BL(x);
                    Fim
          Fim
     Fim  /Fim do Programa Principal/
```

<u>Complexidade</u>: Uma chamada a BL(x) é feita exatamente uma vez para cada vértice do grafo G = (V,E) , logo BL(x) é chamada $|V|$ vezes. Para cada chamada, a quantidade de trabalho é proporcional ao número de arestas incidentes em cada vértice. Portanto, o tempo requerido para fazer uma busca em largura em um grafo não direcionado é $O(|V| + |E|)$.

Um outro algoritmo para Busca em Largura, que é uma implementação linear para o Algoritmo E acima, é o seguinte:

<u>Algoritmo F</u>: Busca em Largura

<u>Entrada</u>: Grafo não direcionado $G = (V,E)$; $v \in V$ é o vértice do qual a busca começa.

<u>Notação</u>: F é uma fila, inicialmente vazia.
"$x \leftarrow F$" significa remover o primeiro elemento (x) da fila.

```
    Início
1.      Visitar e marcar v. Inserir v em F;
2.      Enquanto F é diferente de vazio efetuar
            x ← F
            Para cada vértice w desmarcado adjacente
                a x efetuar
                Visitar e marcar w;
                Inserir w em F;
            Fim
        Fim
    Fim
```

<u>Complexidade</u>: $O(|V| + |E|)$.

3.4) PROPRIEDADES

Seja T uma árvore geradora de Busca em Largura de um grafo não direcionado G = (V,E). Um grafo pode ter muitas árvores geradoras de Busca em Largura. De fato, existe, por exemplo, uma liberdade de escolha dos vértices da linha 1 no Algoritmo E, como também do vértice inicial.

OBS.: O nível de um vértice v pode ser também definido recursivamente do seguinte modo:

$$NIVEL(v) = \begin{cases} 0, \text{ se } v \text{ é uma raiz de uma árvore} \\ 1 + NIVEL(PAI(v)), \text{caso contrário} \end{cases}$$

Uma árvore geradora de busca em largura T satisfaz às seguinte propriedades:

(L_1) - Se v é um ancestral próprio de w em T, então $NBL(v) < NBL(w)$.

(L_2) - Cada aresta em G = (V,E), seja aresta de árvore, ou aresta entre tio e sobrinho, ou aresta entre irmãos, ou aresta entre primos, liga dois vértices cujo nível em T difere de no máximo 1, pois:

1) Se (x,y) é uma aresta de árvore então $NIV(y) - NIV(x) = 1$.

2) Se (x,y) é uma aresta entre tio e sobrinho, então $NIV(y) - NIV(x) = 1$.

3) Se (x,y) é uma aresta entre irmãos então $NIV(y) - NIV(x) = 0$.

4) Se (x,y) é uma aresta entre primos então $NIV(y) - NIV(x) = 0$.

(L_3) - Se v é um vértice no componente conexo de G cuja raiz em T é r, então o nível de v é igual ao comprimento do menor caminho de r para v em G.

(L_4) - Suponha que todos os vértices de um grafo não direcionado $G = (V,E)$ são alcançáveis do vértice s. Então:

i) As arestas de árvore formam uma árvore geradora T com raiz s e que contém todos os vértices de G.

ii) Se (v,w) é uma aresta de árvore então $NBL(v) < NBL(w)$.

iii) Se (v,w) é uma aresta entre irmãos então $NBL(v) < NBL(w)$ e $PAI(v) = PAI(w)$.

iv) Se (v,w) é uma aresta entre primos então $NBL(v) < NBL(w)$ e $PAI(v) \neq PAI(w)$.

v) Se (v,w) é uma aresta entre tio e sobrinho então $NBL(v) < NBL(w)$.

4) BUSCA EM LARGURA EM GRAFOS DIRECIONADOS

Uma busca em largura em um grafo direcionado conserva a mesma idéia já vista para busca em largura em um grafo não direcionado. Entretanto, agora estamos trabalhando com digrafos, onde as arestas já estão orientadas, e não temos liberdade de buscá-las na direção de nossa escolha, como já foi visto para grafos não-direcionados.

Uma Busca em Largura em um grafo direcionado D = (V,E) particiona o conjunto de arestas em seis categorias:

1º - Arestas de Árvore (———>) - são arestas que levam a novos vértices durante a busca.

2º - Arestas de Retorno (---->) - são arestas (x, y) que têm a seguinte característica: NIV(x)-NIV(y) > 1

3º - Arestas de Cruzamento (····>) - a aresta (x, y) é dita ser aresta de cruzamento se x e y estão em diferentes árvores e a direção desta aresta é sempre da direita para a esquerda.

4º - Arestas entre irmãos (⊙⊙⊙⊙⊙>) - a aresta (x,y) é dita ser aresta entre irmãos se Pai(x)= Pai(y) e NIV(x) = NIV(y).

5º - Arestas entre primos (****>) - a aresta (x,y) é dita ser aresta entre primos se NIV(x) = NIV(y) e

(x,y) não é aresta entre irmãos.

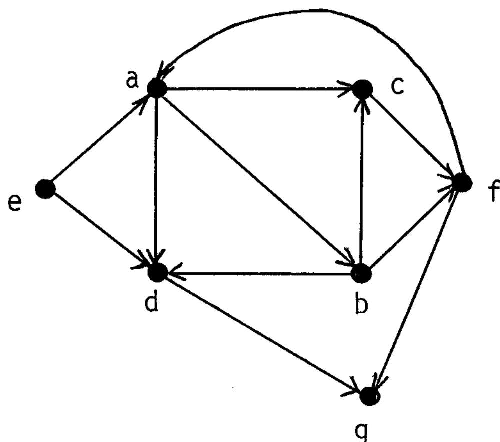
6º - <u>Arestas entre tio e sobrinho</u> (-.-.->) - a aresta (x,y) é uma aresta entre tio e sobrinho se $|NIV(y) - NIV(x)| = 1$ e (x,y) não é aresta de árvore. A direção desta aresta é da direita para a esquerda.

<u>OBS</u>.: As visitas utilizadas estão sendo feitas em pré-ordem.

4.1) EXEMPLO

A figura (IV.4) mostra como a técnica de Busca em Largura é aplicada no Digrafo D = (V,E), que é representado por uma estrutura de Adjacência. A Busca em Largura é iniciada a partir de um vértice arbitrário <u>a</u>. Os números ao lado dos vértices que pertencem à Estrutura Direcionada de Busca em Largura, representam a ordem em que os vértices foram visitados.

Digrafo D:



Estrutura de adjacência:

a : b, c, d

b : c, d, f

c : f

d : g

e : a, d

f : a, g

g : -

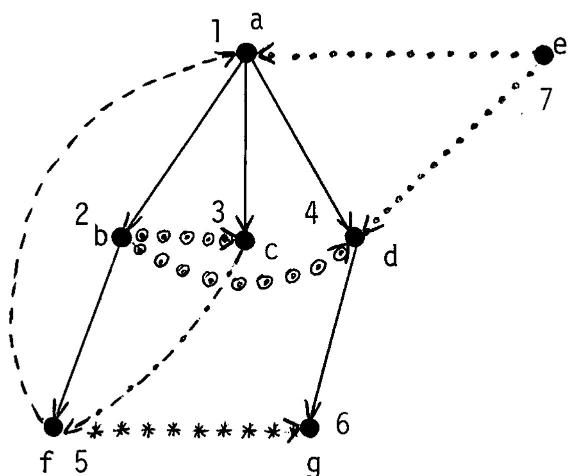


Figura IV.4 - Estrutura Direcionada de Busca em Largura.

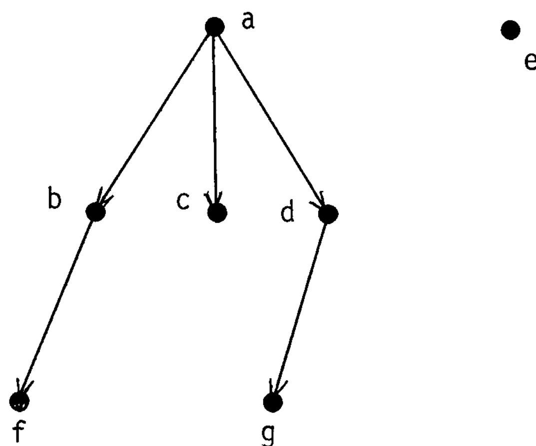


Figura IV.5 - Floresta Direcionada de Busca em Largura.

4.2) ÁRVORE GERADORA DIRECIONADA DE BUSCA EM LARGURA

A Busca em Largura em um grafo direcionado D = (V,E) gera uma árvore (ou floresta) enraizada direcionada T que é composta por arestas de árvore. Essa árvore (ou floresta) T é denominada Árvore (ou Floresta) Geradora Direcionada de Busca em Largura (Ver Figura (IV.5)).

A estrutura obtida da aplicação da técnica de Busca em Largura em um grafo direcionado D = (V,E), que é formada por arestas de árvore, arestas de retorno, arestas de cruzamento , arestas entre irmãos, arestas entre primos e arestas entre tio e sobrinho, é denominada Estrutura Direcionada de Busca em Largura (Ver Figura (IV.4)).

OBS.: A árvore (ou floresta) gerada por uma busca em largura não é única, e a classificação das arestas depende: a) da ordem arbitrária na qual os vértices e arestas do digrafo original são dadas na estrutura de adjacência e b) da escolha do vértice inicial.

4.3) ALGORITMO PARA BUSCA EM LARGURA EM GRAFOS DIRECIONADOS

O algoritmo abaixo é aplicado quando quisermos utilizar a técnica de Busca em Largura em um grafo Direcionado D = (V,E).

Algoritmo G:

Entrada: Um grafo direcionado D = (V,E), representado por uma estrutura de adjacência Adj(v), para $v \in V$.

Saída: Uma partição de E nos seguintes conjuntos: T de arestas de árvore; R de arestas de Retorno; C de arestas de Cruzamento; I de arestas entre Irmãos; P de arestas entre Primos; e S de arestas entre Tio e Sobrinho.

Método: Todos os vértices são marcados inicialmente "nunca enfileirado". A Procedure BLD é usada para visitar um vértice. Um vetor NBL registra a ordem em que os vértices são enfileirados e visitados.

```
Procedimento BLD(x);
Início
1.    Para  y ∈ Adj(x) efetuar
2.      Se  NBL(y) = 0     /y é marcado "nunca enfileirado"/
          então  Início
                  Adicione a aresta (x,y) a T;
                  NIV(y) ← NIV(x) + 1;
                  Pai(y) ← x;
                  Adicione y a F;
                  i ← i + 1;
                  NBL(y) ← i;     /Marque y "enfileirado"/
                  DISCR(y) ← m;
                 Fim

          senão
               Se NBL(x) < NBL(y)
                 então
                      Se NIV(y) - NIV(x) = 1
                         então Adicione a aresta (x,y) a S;
                         senão
                              Se NIV(y) - NIV(x) = 0
                                 então  Se Pai(x) = Pai(y)
                                           então Adicione (x,y) a I;
                                           senão Adicione (x,y) a P;
                                 Fim
```

```
                    senão
                      Se  DISCR(x) ≠ DISCR(y)
                            então  Adicione (x,y) a C;
                            senão  Se NIV(x) - NIV(y) > 1
                                      então  Adicione (x,y) a R;
                                      senão  Fim;

     Retorne
Fim           /Fim do Procedimento BLD(x)/

Início     /Programa Principal/

T ← S ← I ← P ← R ← C ← ϕ ; F ← Fila Vazia;
i ← 0 ; m ← 0 ;
    Para     v ∈ V  efetuar
          NBL(v) ← 0    /Marque v "nunca enfileirado"/
    Enquanto F é vazia e existe v ∈ V marcado "nunca en-
             fileirado"
             Adicione v a F;
             NIV(v) ← 0; Pai(v) ← 0;
             i ← i + 1; m ← m + 1;
             NBL(v) ← i;    /Marque v "enfileirado"/
             DISCR(v) ← m;  /v pertence à m-ésima árvore/

             Enquanto F não é vazia
                      x ← cabeça de F;
                      F ← F - x;
                      BLD(x);
             Fim
    Fim

Fim    /Fim de programa principal/
```

Complexidade: A execução do algoritmo G acima irá depender do número de vértices existentes no digrafo. Além disso, a quantidade de trabalho deste Algoritmo é proporcional ao número de arestas incidentes em cada vértice. Logo, o tempo requerido para fazer uma busca em Largura em um grafo direcionado é $O(|V| + |E|)$.

OBS.: O algoritmo F, já visto para Busca em Largura em grafos não direcionados, poderá ser aqui utilizado para busca em Largura em grafos direcionados.

4.4) PROPRIEDADES

Uma árvore geradora direcionada de busca em Largura T satisfaz às seguinte propriedades:

(B_1) - Se v é um ancestral próprio de w em T, então NBL(v) < NBL(w).

(B_2) - Uma aresta de árvore, ou aresta entre irmãos, ou aresta entre primos, ou aresta entre tio e sobrinho, liga dois vértices cujo nível em T difere de no máximo 1.

(B_3) - Suponha que todos os vértices de um grafo direcionado D são alcançáveis do vértice s, e que as arestas de D são divididas em seis categorias já vistas. NBL registra a ordem em que os vértices serão visitados. Então:

i) As <u>arestas de árvore</u> formam uma árvore geradora direcionada T com raiz <u>s</u> que contêm todos os vértices em D.

ii) Se (v,w) é uma <u>aresta de árvore</u> então NBL(v) < NBL(w).

iii) Se (v,w) é uma <u>aresta de retorno</u> então NBL(v) > NBL(w), e existe um caminho de w para v em T.

iv) Se (v,w) é uma <u>aresta de cruzamento</u> então não existe um caminho de v para w em T e nem de w para v, e podemos dizer também que NBL(v) > NBL(w).

v) Se (v,w) é uma <u>aresta entre irmãos</u> então NBL(v) < NBL(w) e PAI(v) = PAI(w).

vi) Se (v,w) é uma <u>aresta entre primos</u> então NBL(v) < NBL(w) é PAI(v) ≠ PAI(w).

vii) Se (v,w) é uma <u>aresta entre tio e sobrinho</u> então NBL(v) < NBL(w).

5) CONCLUSÃO

Em resumo, podemos concluir que quando aplicamos a técnica de "Busca em Largura" em um grafo não direcionado, iremos obter como resultado uma Estrutura de Busca em Largura composta por quatro diferentes conjuntos de arestas:

1) Arestas de Árvore;
2) Arestas entre Irmãos;
3) Arestas entre Primos;
4) Arestas entre Tio e Sobrinho.

Por sua vez, quando aplicamos esta técnica em um grafo Direcionado, a Estrutura resultante será constituida pelos seguintes conjuntos de arestas:

1) Arestas de Árvore;
2) Arestas de Retorno;
3) Arestas de Cruzamento;
4) Arestas entre Irmãos;
5) Arestas entre Primos;
6) Arestas entre Tio e Sobrinho.

Este critério de busca em grafos aqui descrito é também muito eficiente, pois, como podemos observar, os Algoritmos tanto para grafos não direcionados, como para digrafos, terão complexidade $O(|V| + |E|)$.

CAPÍTULO V

CASOS ESPECIAIS

1) INTRODUÇÃO

O primeiro tópico a ser descrito neste capítulo é sobre a classificação dos Algoritmos. Os algoritmos de Busca vistos até agora são classificados como TOTALMENTE RESTRINGIDOS. As outras classificações que serão aqui apresentadas são denominadas: ALGORITMOS NÃO-RESTRINGIDOS e ALGORITMOS PARCIALMENTE RESTRINGIDOS.

O segundo tópico a ser estudado é sobre "Busca em Largura Lexicográfica" em um grafo não direcionado, que é um caso especial de Busca em Largura. Neste tipo de busca os vértices serão examinados de acordo com uma prioridade que depende de seus ancestrais. Esta prioridade irá diminuir a liberdade de escolha dos vértices, mas nem sempre a eliminará, isto é, a prioridade não conduzirá a escolha necessariamente única de um vértice.

Finalmente, o terceiro tópico é sobre "Busca em Profundidade Lexicográfica" em um grafo não direcionado, que é um caso especial de Busca em Profundidade e que possui a característica de que os vértices também serão explorados de acordo com uma certa prioridade, e não aleatoriamente.

2) CLASSIFICAÇÃO DOS ALGORITMOS DE BUSCA

2.1) ALGORITMOS TOTALMENTE RESTRINGIDOS

Todos os algoritmos vistos até agora, tanto para busca em profundidade como para busca em largura, são classificados como TOTALMENTE RESTRINGIDOS, cuja característica é a seguinte: cada vértice é explorado uma única vez, logo cada aresta é explorada uma única vez.

Em ALGORITMOS TOTALMENTE RESTRINGIDOS, v está marcado se e somente se v pertence ou pertenceu à pilha (no caso de busca em profundidade) ou à fila (no caso de busca em largura).

Exemplo de um algoritmo TOTALMENTE RESTRINGIDO para busca em profundidade:

```
      Procedimento BPTR(v);
      Início
1.    Visitar, marcar e introduzir v na pilha P;
2.    Enquanto P é não vazia efetuar
          Início
3.        Enquanto existir um vértice w desmarcado adjacente
                   ao topo efetuar
                   Início
4.                 Visitar, marcar e colocar w na pilha;
                   Fim
5.        Retirar da Pilha P;
          Fim
      Fim
```

<u>Complexidade</u>: $O(|V| + |E|)$

2.2) ALGORITMOS NÃO-RESTRINGIDOS

Uma outra classificação que podemos ter é a de ALGORITMOS NÃO-RESTRINGIDOS, cuja característica é a seguinte: um vértice pode ser explorado mais de uma vez, logo uma aresta poderá ser explorada mais de uma vez.

Em ALGORITMOS NÃO-RESTRINGIDOS, v está marcado <u>se e somente se</u> v pertence à pilha (ou à fila).

Exemplo de um algoritmo NÃO-RESTRINGIDO para busca em profundidade.

<u>Procedimento</u> BPNR(v);

<u>Início</u>

1. Visitar, marcar e introduzir v na pilha P;
2. <u>Para</u> w ∈ Adj(v) <u>efetuar</u>

   <u>Início</u>

3. <u>Se</u> w desmarcado <u>então</u> BPNR(w);
4. Desmarcar e retirar v da Pilha P;

   <u>Fim</u>

<u>Fim</u>

<u>Complexidade</u>: O algoritmo acima constrói todos os caminhos (simples) do grafo, com origem no vértice raiz de busca. A complexidade dependerá do número de caminhos do grafo em ques

tão, o qual pode ser exponencial no tamanho do grafo (pior caso).

2.3) ALGORITMOS PARCIALMENTE RESTRINGIDOS

Finalmente, um algoritmo pode ser classificado como PARCIALMENTE RESTRINGIDO, onde, dependendo de uma determinada condição, alguns vértices serão desmarcados e outros não.

Este tipo de algoritmo é uma mistura do ALGORITMO TOTALMENTE RESTRINGIDO com o ALGORITMO NÃO-RESTRINGIDO.

Exemplo de um algoritmo PARCIALMENTE RESTRINGIDO para busca em profundidade.

```
   Procedimento BPPR(v);
   Início
1. Visitar, marcar e introduzir v na pilha P;
2. Para w ∈ Adj(v) efetuar
                    Início
3.                  Se w desmarcado então BPPR(w);
4.                  Se condição então
5.                  Desmarcar e retirar v da pilha P;
                    Fim
   Fim
```

Complexidade: Depende da condição. No pior caso será exponencial.

3) BUSCA EM LARGURA LEXICOGRÁFICA: (BLL)

A Busca em Largura Lexicográfica é um tipo especial de busca em largura, em que a fila usual de vértices é substituida por uma fila de subconjuntos (não ordenados) de vértices, que é algumas vezes refinada mas nunca reordenada.

Na Busca em Largura Lexicográfica, os vértices de um dado nível não são examinados na mesma ordem em que eles são enfileirados, mas sim de acordo com uma prioridade que depende de seus ancestrais.

3.1) ALGORITMO

O método é o seguinte:

Entrada: Um grafo não direcionado $G = (V,E)$ representado por uma estrutura de adjacências $Adj(v)$, $v \in V$.

Saída: Uma ordenação σ dos vértices.

Método: Os vértices são numerados de n a 1 na ordem em que eles são selecionados na linha (3) (Ver algoritmo BLL). Conforme será verificado, esta numeração fixa as posições de um esquema de eliminação σ. Para cada vértice x, o rótulo de x consistirá de um conjunto de

números listados em ordem decrescente. Os vértices podem, então, ser lexicograficamente ordenados de acordo com seus rótulos (Ordem lexicografica é justamente a ordem de dicionário, de maneira que 8651 < 874 e 532 < 5321).

<u>Algoritmo BLL</u>

<u>Início</u>

1. <u>Para</u> cada vértice $v \in V$, <u>efetuar</u> rótulo(v) = ϕ;

2. <u>Para</u> i = n <u>some</u> - 1 <u>até</u> 1 <u>efetuar</u>

   <u>Início</u>

3. Seleção: Escolher um vértice não numerado v com maior rótulo.

4. $\sigma(i) \leftarrow v$; /Isto determina para v o número i/

5. Atualização: <u>Para</u> cada vértice não numerado $w \in Adj(v)$ <u>efetuar</u> Adicione i a rótulo(w);

   <u>Fim</u>.

<u>Fim</u>.

<u>Complexidade</u>: $O(|V| + |E|)$

3.2) PROPRIEDADES

Para cada valor i, seja $R_i(x)$ o rótulo de x quando a proposição (4) do algoritmo BLL é executada, isto é, quando o i-ésimo vértice é numerado. Lembremos que o índice é decrementado em cada iteração sucessiva. Por exemplo, $Rn(x) = \phi$ para todo x e $Rn\text{-}1(x) = \{n\}$ sss $x \in Adj(\sigma(n))$.

As seguintes propriedades são importantes:

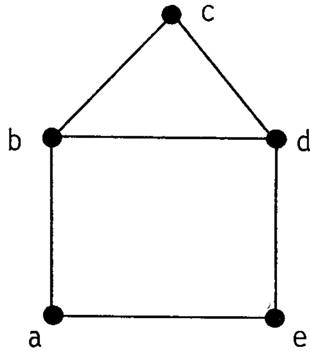
(L_1) $R_i(x) \leq R_j(x)$ $(j \leq i)$

(L_2) $R_i(x) < R_i(y) \Longrightarrow R_j(x) < R_j(y)$ $(j < i)$

(L_3) Se $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ e $c \in Adj(a) - Adj(b)$, então existe um vértice $d \in Adj(b) - Adj(a)$ com $\sigma^{-1}(c) < \sigma^{-1}(d)$.

3.3) EXEMPLOS

Exemplo 1: Aplicar o Algoritmo BLL (visto anteriormente) no grafo G = (V,E) abaixo:



Estrutura de adjacência:

a → b, e

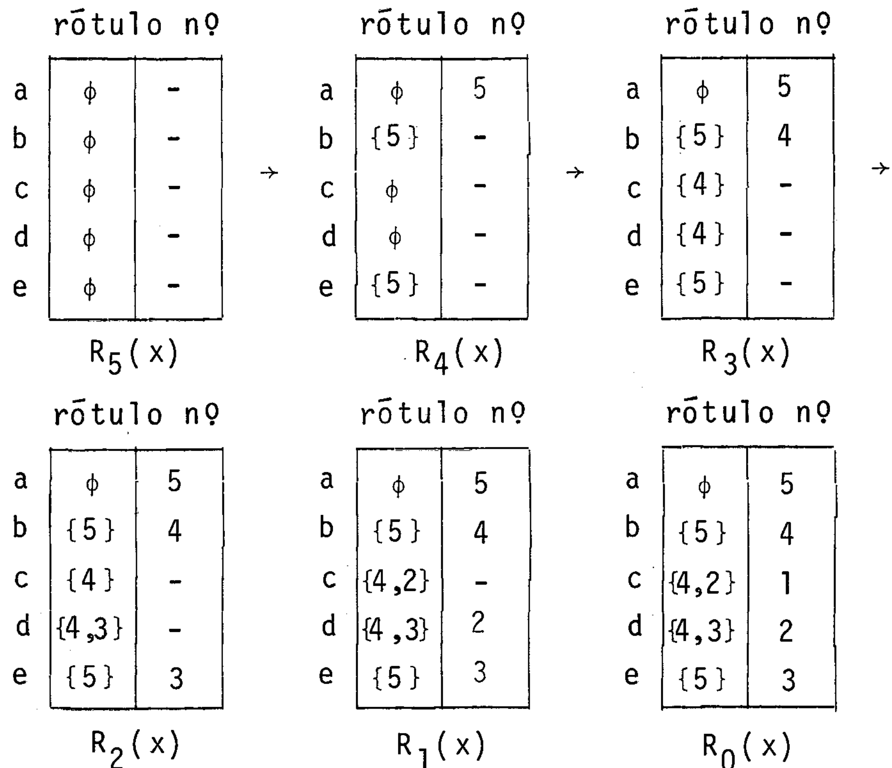
b → a, c, d

c → b, d

d → c, b, e

e → a, d

O vértice a é selecionado arbitrariamente na linha 3 durante o primeiro passo. O desenvolvimento da rotulação e numeração (nº) são descritos abaixo:



	rótulo	nº
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	ϕ	-
e	ϕ	-

$R_5(x)$ →

	rótulo	nº
a	ϕ	5
b	{5}	-
c	ϕ	-
d	ϕ	-
e	{5}	-

$R_4(x)$ →

	rótulo	nº
a	ϕ	5
b	{5}	4
c	{4}	-
d	{4}	-
e	{5}	-

$R_3(x)$ →

	rótulo	nº
a	ϕ	5
b	{5}	4
c	{4}	-
d	{4,3}	-
e	{5}	3

$R_2(x)$

	rótulo	nº
a	ϕ	5
b	{5}	4
c	{4,2}	-
d	{4,3}	2
e	{5}	3

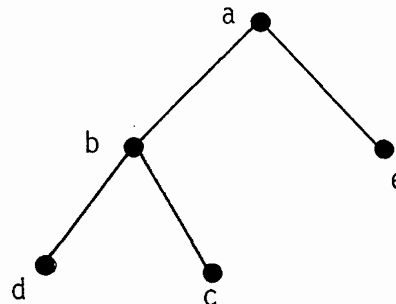
$R_1(x)$

	rótulo	nº
a	ϕ	5
b	{5}	4
c	{4,2}	1
d	{4,3}	2
e	{5}	3

$R_0(x)$

$\sigma = [c, d, e, b, a]$

Árvore geradora de Busca em Largura Lexicográfica:



Exemplo das Propriedades:

(L_1) Seja $i = 3$ e $j = 2$ $\rightarrow$ $R_3(d) \leq R_2(d)$ $(j \leq i)$

$\{4\} \leq \{4,3\}$ $2 \leq 3$

(L_2) $i = 3$, $j = 2$ $\rightarrow$ $R_3(d) < R_3(e) \Longrightarrow R_2(d) < R_2(e)$

$\{4\} < \{5\}$ $\{4,3\} < \{5\}$

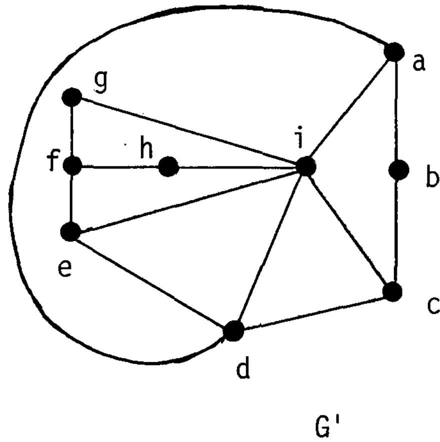
(L_3) Se $\sigma^{-1}(d) < \sigma^{-1}(e) < \sigma^{-1}(b)$ e $b \in Adj(d) - Adj(e)$

$2 < 3 < 4$

ent $\exists$ $a \in Adj(e) - Adj(d)$ com $\sigma^{-1}(b) < \sigma^{-1}(a)$

$4 < 5$

Exemplo 2: Idem para o grafo G' = (V',E') abaixo:



G'

Estrutura de adjacência:

a → d, i, b

b → a, c

c → b, i, d

d → a, e, i, c

e → f, i, d

f → g, h, e

g → f, i

h → f, i

i → a, c, d, e, h, g

	R	N
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	ϕ	-
e	ϕ	-
f	ϕ	-
g	ϕ	-
h	ϕ	-
i	ϕ	-

	R	N
a	ϕ	9
b	{9}	-
c	ϕ	-
d	{9}	-
e	ϕ	-
f	ϕ	-
g	ϕ	-
h	ϕ	-
i	{9}	-

	R	N
a	ϕ	9
b	{9}	8
c	{8}	-
d	{9}	-
e	ϕ	-
f	ϕ	-
g	ϕ	-
h	ϕ	-
i	{9}	-

	R	N
a	ϕ	9
b	{9}	8
c	{8,7}	-
d	{9}	7
e	{7}	-
f	ϕ	-
g	ϕ	-
h	ϕ	-
i	{9,7}	-

	R	N
a	ϕ	9
b	{9}	8
c	{8,7,6}	-
d	{9}	7
e	{7,6}	-
f	ϕ	-
g	{6}	-
h	{6}	-
i	{9,7}	6

	R	N
a	ϕ	9
b	{9}	8
c	{8,7,6}	5
d	{9}	7
e	{7,6}	-
f	ϕ	-
g	{6}	-
h	{6}	-
i	{9,7}	6

	R	N
a	ϕ	9
b	{9}	8
c	{8,7,6}	5
d	{9}	7
e	{7,6}	4
f	{4}	-
g	{6}	-
h	{6}	-
i	{9,7}	6

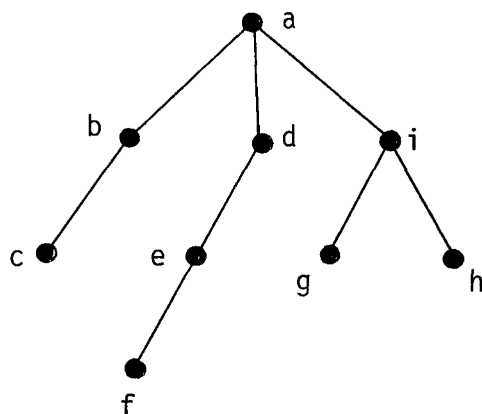
	R	N
a	ϕ	9
b	{9}	8
c	{8,7,6}	5
d	{9}	7
e	{7,6}	4
f	{4,3}	-
g	{6}	3
h	{6}	-
i	{9,7}	6

	R	N
a	ϕ	9
b	{9}	8
c	{8,7,6}	5
d	{9}	7
e	{7,6}	4
f	{4,3,2}	-
g	{6}	3
h	{6}	2
i	{9,7}	6

	R	N
a	ϕ	9
b	{9}	8
c	{8,7,6}	5
d	{9}	7
e	{7,6}	4
f	{4,3,2}	1
g	{6}	3
h	{6}	2
i	{9,7}	6

$\sigma = [f, h, g, e, c, i, d, b, a]$

Árvore Geradora de Busca em Largura Lexicográfica



4) BUSCA EM PROFUNDIDADE LEXICOGRAFICA (BPL)

A Busca em Profundidade Lexicográfica é um tipo especial de busca em Profundidade. Os vértices que serão buscados na BPL seguirão em determinado critério ou método, que será descrito abaixo.

4.1) ALGORITMO

Entrada: Os conjuntos de adjacência de um grafo não direcionado G = (V,E).

Saída: Uma ordenação σ dos vértices

Método: Os vértices são numerados de 1 a n na ordem em que eles são selecionados na linha 3 (Ver algoritmo BPL).

Para cada vértice x, o rótulo de x consistirá de um conjunto de números listados em ordem decrescente. Os vértices podem, então, ser lexicograficamente ordenados de acordo com seus rótulos.

Algoritmo BPL:

Início

1 Para cada vértice $v \in V$ efetuar rótulo $(v) = \phi$;

2 Para $i = 1$ some $+1$ até n efetuar

Início

3. Seleção: escolher um vértice não numerado v com maior rótulo;

4. $\sigma(i) \leftarrow v$

5. Atualização: Para cada vértice não numerado $w \in Adj(v)$ efetuar adicione i a rótulo(w);

Fim.

Fim.

Complexidade: $O(|V|^2)$ (Implementação direta do algoritmo).

4.2) PROPRIEDADES

Para cada valor i, seja $R_i(x)$ o rótulo de x quando a proposição 4 (do algoritmo BPL) é executada, isto é, quando o i-ésimo vértice é numerado. Lembremos que o índice é incrementado em cada iteração sucessiva.

As seguintes propriedades são importantes:

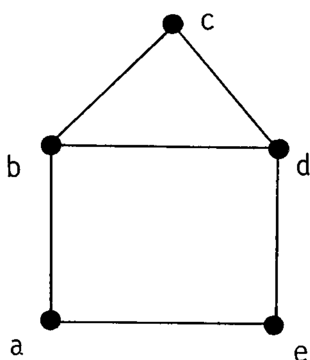
(L'_1) $R_i(x) \leq R_j(x)$ $(i \leq j)$

(L'_2) $R_i(x) < R_i(y) \Longrightarrow R_j(x) < R_j(y)$ $(i < j)$

(L'_3) $\sigma^{-1}(a) > \sigma^{-1}(b) > \sigma^{-1}(c)$ e $c \in Adj(a) - Adj(b)$, então existe um vértice $d \in Adj(b) - Adj(a)$ com $\sigma^{-1}(c) > \sigma^{-1}(d)$

4.3) EXEMPLOS

Exemplo 1: Aplicar o Algoritmo BPL no grafo G = (V,E) abaixo:



Estrutura Adjacência:

a → b, e

b → a, c, d

c → b, d

d → c, b, e

e → a, d

O vértice a será selecionado arbitrariamente no passo 3. O desenvolvimento do algoritmo para os demais vértices é mostrado a seguir:

$R_0(x)$

	R	N
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	ϕ	-
e	ϕ	-

$R_1(x)$

	R	N
a	ϕ	1
b	{1}	-
c	ϕ	-
d	ϕ	-
e	{1}	-

$R_2(x)$

	R	N
a	ϕ	1
b	{1}	2
c	{2}	-
d	{2}	-
e	{1}	-

$R_3(x)$

	R	N
a	ϕ	1
b	{1}	2
c	{2}	3
d	{2,3}	-
e	{1}	-

$R_4(x)$

	R	N
a	ϕ	1
b	{1}	2
c	{2}	3
d	{2,3}	4
e	{1,4}	-

$R_5(x)$

	R	N
a	ϕ	1
b	{1}	2
c	{2}	3
d	{2,3}	4
e	{1,4}	5

Propriedades:

(L'_1) $j = 3,\ i = 2 \rightarrow R_2(d) \leq R_3(d)$ $(i \leq j)$

$\{2\} \leq \{2,3\}$

(L'_2) $j = 3,\ i = 2 \rightarrow R_2(b) < R_2(d) \Longrightarrow R_3(b) < R_3(d)$

$\{1\} < \{2\}$ $\{1\} < \{2,3\}$

$(i < j)$

(L'_3) $\sigma^{-1}(d) > \sigma^{-1}(e) > \sigma^{-1}(b)$ e $b \in Adj(d) - Adj(e)$

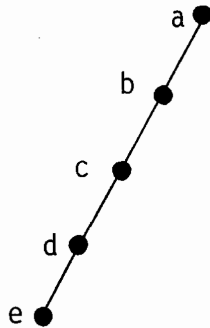
4 3 2

ent $\exists\ a \in Adj(e) - Adj(d)$ com $\sigma^{-1}(b) > \sigma^{-1}(a)$

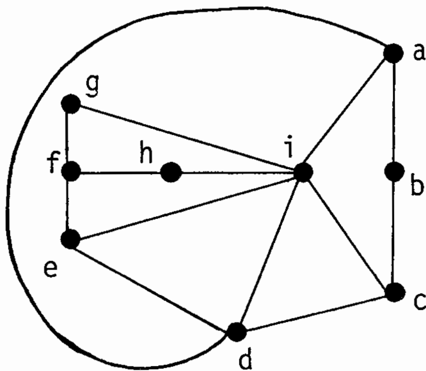
$2 > 1$

$\sigma = [a, b, c, d, e]$

Árvore Geradora de Busca em Profundidade Lexicográfica



Exemplo 2: Idem para o grafo G' = (V',E') abaixo:



Estrutura de Adjacência:

a → d, i, b

b → a, c

c → b, i, d,

d → a, e, i, c

e → f, i, d

f → g, h, e

g → f, i

h → f, i

i → a, c, d, e, h, g

Começando com o vértice f:

	R	N
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	ϕ	-
e	ϕ	-
f	ϕ	-
g	ϕ	-
h	ϕ	-
i	ϕ	-

	R	N
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	ϕ	-
e	{1}	-
f	ϕ	1
g	{1}	-
h	{1}	-
i	ϕ	-

	R	N
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	{2}	-
e	{1}	2
f	ϕ	1
g	{1}	-
h	{1}	-
i	{2}	-

	R	N
a	{3}	-
b	ϕ	-
c	{3}	-
d	{3,2}	-
e	{1}	2
f	ϕ	1
g	{3,1}	-
h	{3,1}	-
i	{2}	3

	R	N
a	{4,3}	-
b	ϕ	-
c	{4,3}	-
d	{3,2}	4
e	{1}	2
f	ϕ	1
g	{3,1}	-
h	{3,1}	-
i	{2}	3

	R	N
a	{4,3}	5
b	{5}	-
c	{4,3}	-
d	{3,2}	4
e	{1}	2
f	ϕ	1
g	{3,1}	-
h	{3,1}	-
i	{2}	3

	R	N
a	{4,3}	5
b	{5}	6
c	{6,4,3}	-
d	{3,2}	4
e	{1}	2
f	ϕ	1
g	{3,1}	-
h	{3,1}	-
i	{2}	3

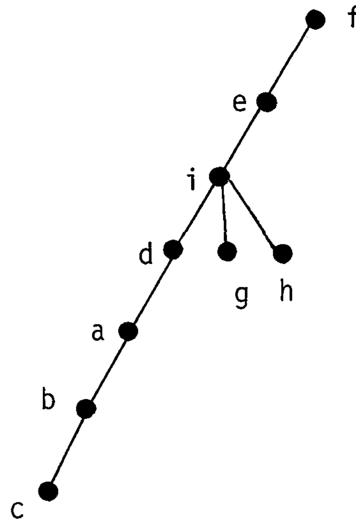
	R	N
a	{4,3}	5
b	{5}	6
c	{6,4,3}	7
d	{3,2}	4
e	{1}	2
f	ϕ	1
g	{3,1}	-
h	{3,1}	-
i	{2}	3

	R	N
a	{4,3}	5
b	{5}	6
c	{6,4,3}	7
d	{3,2}	4
e	{1}	2
f	ϕ	1
g	{3,1}	8
h	{3,1}	-
i	{2}	3

	R	N
a	{4,3}	5
b	{5}	6
c	{6,4,3}	7
d	{3,2}	4
e	{1}	2
f	ϕ	1
g	{3,1}	8
h	{3,1}	9
i	{2}	3

$\sigma = [f, e, i, d, a, b, c, g, h]$

Árvore Geradora de Busca em Profundidade Lexicográfica



5) CONCLUSÃO

Os Algoritmos de Busca podem ser classificados em três categorias diferentes: Totalmente Restringidos, Não Restringidos e Parcialmente Restringidos. Os algoritmos Totalmente Restringidos têm complexidade linear, ou seja, $O(|V| + |E|)$. A complexidade dos Não Restringidos dependerá do número de caminhos do grafo, o qual poderá ser exponencial no tamanho do grafo, no pior caso. E a complexidade dos Algoritmos Parcialmente Restringidos dependerá de uma certa condição (dada por exemplo, no passo (4) do Procedimento BPPR(v)) e poderá ser também exponencial no pior caso.

A Busca em Largura Lexicográfica é um tipo especial de Busca em Largura em que o exame de cada vértice é feito segundo certos critérios, ou seja, cada vértice a ser explorado obedece uma certa prioridade que depende de seus ancestrais. A com

plexidade para este tipo de busca é linear: $O(|V| + |E|)$.

A Busca em Profundidade Lexicográfica é também um caso especial de busca em profundidade em que cada vértice a ser explorado obedecerá a certos critérios. Este tipo de busca terá como complexidade: $O(|V|^2)$, no caso de fazermos uma implementação direta do algoritmo.

CAPÍTULO VI

APLICAÇÕES

1) INTRODUÇÃO

O presente capítulo é uma parte importante desta tese, pois ele é composto por vários problemas que serão resolvidos utilizando as técnicas de busca estudadas nos Capítulos III, IV e V.

Para cada aplicação, será descrito o problema a resolver e, em certos casos, serão introduzidos novos conceitos em grafos. Além disso serão apresentadas as soluções dos problemas, usando como ferramenta os métodos de busca, e também serão expostos exemplos explicativos que muito auxiliam na compreensão da técnica utilizada.

As referências bibliográficas aqui mencionadas nem sempre irão corresponder ao primeiro trabalho na área.

2) APLICAÇÃO - COMPONENTES CONEXOS

2.1) REFERÊNCIAS - REINGOLD, NIEVERGELT & DEO[60]; BAASE[3]; OUTRAS.

2.2) DESCRIÇÃO DO PROBLEMA

Determinar os componentes conexos de um grafo não direcionado G = (V,E).

Um <u>componente conexo</u> de G é um subgrafo conexo máximo, isto é, um subgrafo conexo que não está contido em algum maior subgrafo conexo.

<u>OBS.</u>: BAASE[3] e REINGOLD, NIEVERGELT & DEO[60] são apenas exemplos de referências para a aplicação acima. Porém, não há uma primeira referência para este problema.

2.3) MÉTODO - BUSCA EM PROFUNDIDADE OU BUSCA EM LARGURA

O problema de achar os componentes conexos de um grafo não direcionado G é muito simples. Uma das maneiras de resolvê-lo é usar a técnica de busca em profundidade, ou seja: começando em um vértice arbitrário, efetua-se uma busca em profundidade (caracterizando-se um componente); e se restarem vértices desmarcados, repete-se o procedimento (caracterizando outro(s) componente(s)).

Em REINGOLD, NIEVERGELT & DEO[60] é apresentado um Algo-

ritmo simples que determina um único número, compnum(v), para cada vértice w pertencente ao componente no qual o vértice v ocorre. A complexidade deste algoritmo é também $O(|V| + |E|)$, como o Algoritmo B.

A referência BAASE[3] apresenta um Algoritmo de Busca em Profundidade mais detalhado, que também resolve o problema aqui descrito, e cuja complexidade é $O(|V| + |E|)$.

Uma outra técnica que pode ser aplicada para determinar os componentes conexos de um grafo não direcionado G = (V,E) é a de Busca em Largura. Este método também terá complexidade igual a $O(|V| + |E|)$.

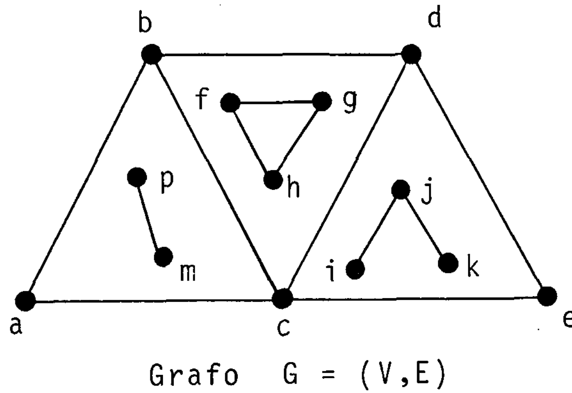
O que podemos concluir é que tanto a Busca em Profundidade, como a Busca em Largura, são duas técnicas diferentes, e ao mesmo tempo eficientes, para se resolver o problema de achar os componentes conexos. Devido à simplicidade deste problema, outros tipos de busca também poderão resolvê-lo.

2.4) EXEMPLO

Achar os componentes conexos do grafo não direcionado G = (V,E):

Estrutura de Adjacência:

a → b, c	g → f, h	m → p
b → a, c, d	h → f, g	
c → a, b, d, e	i → j	
d → b, c, e	j → i, k	
e → c, d	k → j	
f → g, h	p → m	



Grafo G = (V,E)

Aplicando a Técnica de Busca em Profundidade (O Algoritmo de REINGOLD, NIEVERGELT & DEO[60]).

Seja a o vértice inicial a ser buscado. Obteremos compnum (a) ← 1; assim como os demais vértices do 1º componente conexo terão o valor de compnum(x) ← 1 (x = a,b, d,e,c).

Similarmente: o 2º componente conexo terá

compnum(x) ← 2 (x = f, g, h)

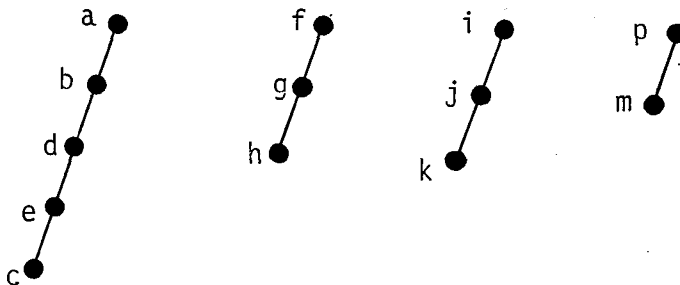
o 3º componente conexo terá

compnum(x) ← 3 (x = i, j, k)

o 4º componente conexo terá

compnum(x) ← 4 (x = p, m)

A floresta geradora de busca em profundidade será:



Logo, a Busca em Profundidade particiona o grafo G em 4 componentes conexos.

3) APLICAÇÃO - COMPONENTES BICONEXOS

3.1) REFERÊNCIA - TARJAN[71]

3.1.1) OUTRAS REFERÊNCIAS - AHO[1]; BAASE[3]; GOLUMBIC[33]; REINGOLD, NIEVERGELT & DEO[60].

3.2) DESCRIÇÃO DO PROBLEMA

Determinar os componentes biconexos de um grafo não direcionado G = (V,E).

Seja G = (V,E) um grafo não direcionado conexo. Se G - v é desconexo então v é um vértice de articulação. Um grafo conexo com $|V|>2$ e sem vértices de articulação é dito biconexo ou não separável. Um grafo que contém um vértice de articulação é chamado separável.

Propriedade (1): Se v é um vértice de articulação então existem dois vértices u,w tais que v se encontra em todo caminho de u para w.

Um Componente Biconexo ou Bloco de um grafo é um subgrafo biconexo máximo, isto é, um subgrafo biconexo que não está contido em algum maior subgrafo biconexo.

3.3) MÉTODO - BUSCA EM PROFUNDIDADE

Usamos a técnica de Busca em Profundidade para encontrar os vértices de articulação e componentes biconexos de um grafo não direcionado G = (V,E).

Em resumo, o Algoritmo para resolver o problema aqui descrito faz o seguinte:

1) Testa se um vértice na árvore geradora de busca em profundidade é um vértice de articulação. Temos o seguinte resultado:

Resultado: Em uma estrutura de busca em profundidade, um vértice v, diferente da raiz, é um vértice de articulação se e somente se existe um filho w de v, tal que não há aresta de retorno da subárvore de raiz w, para algum ancestral próprio de v. Por outro lado, a raiz da busca é vértice de articulação se e só se possuir mais de um filho.

2) Após reconhecermos os vértices de articulação, podemos também determinar os componentes biconexos aplicando a busca em profundidade e armazenando as arestas em uma pilha à medida que elas são exploradas. As arestas que estão armazenadas na pilha, quando voltamos a um vértice de articulação, formam um componente biconexo.

Com a finalidade de reconhecer um vértice de articulação, será preciso computar, durante a busca em profundidade , uma nova função, lowpt(v), para cada vértice v no grafo G (Ver

TARJAN[71] e REINGOLD, NIEVERGELT & DEO[60]). Definimos a função lowpt: V → V da seguinte forma: lowpt(v) = w onde w é o num do vértice mais próximo da raiz da árvore geradora de Busca em Profundidade (T), que pode ser atingido a partir de v descendo em T através de zero ou mais arestas, e subindo em T por apenas uma aresta de retorno.

Em outras palavras, lowpt(v) é o menor valor de num(x), onde x é um vértice do grafo que pode ser alcançado de v seguindo uma sequência de zero ou mais arestas de árvore seguidas por no máximo uma aresta de retorno.

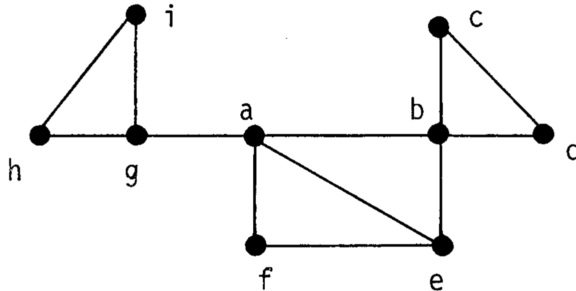
<u>Propriedade (2)</u>: v diferente da raiz é um vértice de articulação se e somente se v possui um filho w em T tal que: lowpt(w) = v ou w.

Como o algoritmo para achar os componentes biconexos de um grafo G = (V,E) é uma busca em profundidade, com uma quantidade constante de trabalho extra feita quando cada aresta é explorada, o tempo requerido é claramente $O(|V| + |E|)$.

3.4) <u>EXEMPLO</u>

Achar os componentes biconexos do grafo não direcionado G = (V,E):

Grafo G = (V,E)



Estrutura de adjacência:

a → b, e, f, g

b → a, c, d, e

c → b, d

d → b, c

e → a, b, f

f → a, e

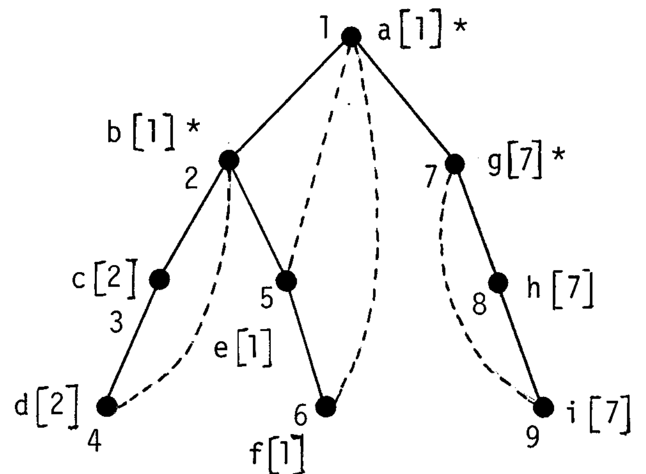
g → a, h, i

h → g, i

i → g, h

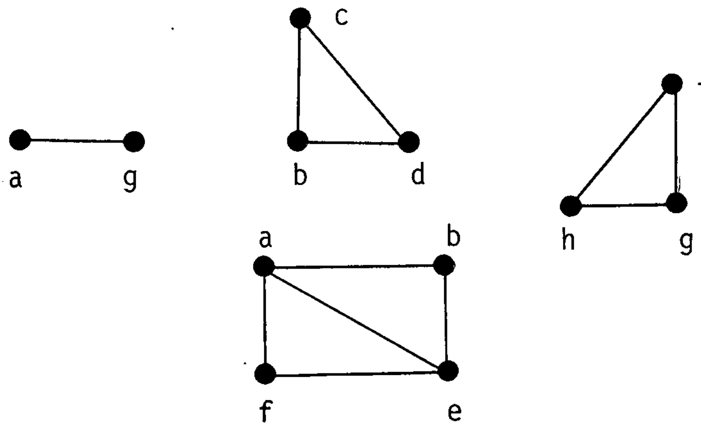
v	NUM(v)	LOWPT(v)
a	1	1
b	2	1
c	3	2
d	4	2
e	5	1
f	6	1
g	7	7
h	8	7
i	9	7

Estrutura de Busca em Profundidade



Os valores de lowpt estão ao lado dos vértices e os vértices de articulação estão marcados com * e são: a, b, e g. (Verificar pelas Propriedades (1) e (2)).

Logo, os componentes biconexos de G = (V,E) serão:
(Ver Algoritmo de TARJAN[71]).



OBS.: A identificação dos vértices de articulação e dos componentes biconexos de um dado grafo é muito importante quando um grafo é representado por uma rede de comunicação ou transporte. Suponha, por exemplo, que os vértices são estações de telefones e as arestas são linhas de telefones. Se o grafo é biconexo, o sistema ainda pode ser operado no caso em que uma estação não funcione. Portanto, o problema de determinar se um grafo é biconexo é importante , como também é o de encontrar os vértices de articulação que podem desconectá-lo.

3.5) GENERALIZAÇÕES

Informalmente falando, um grafo biconexo tem dois caminhos disjuntos entre cada par de vértices. Podemos definir triconectividade (e em geral, k-conectividade) como sendo a propriedade de ter três (em geral, K) caminhos entre algum par

de vértices. Um algoritmo eficiente para achar os componentes triconexos, que também usa busca em profundidade, foi desenvolvido por Hopcroft e Tarjan em 1973 (Ver TARJAN & HOPCROFT[77]), mas ele é muito mais complicado que o algoritmo para achar os componentes biconexos.

OBS.: A determinação eficiente dos componentes K-conexos de um grafo é um problema ainda em aberto.

4) APLICAÇÃO - COMPONENTES FORTEMENTE CONEXOS

4.1) REFERÊNCIA - TARJAN[71]

4.1.1) OUTRAS REFERÊNCIAS - AHO[1]; BAASE[3]; REINGOLD, NIEVERGELT & DEO[60].

4.2) DESCRIÇÃO DO PROBLEMA

Determinar os componentes fortemente conexos de um digrafo $D = (V,E)$.

Um digrafo D é fortemente conexo se existe pelo menos um caminho direcionado entre cada par ordenado de vértices.

Um componente fortemente conexo de um digrafo é um subgrafo fortemente conexo máximo.

4.3) MÉTODO - BUSCA EM PROFUNDIDADE

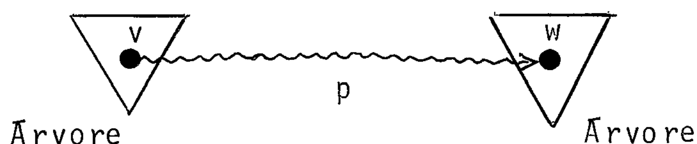
Aplicamos a técnica de busca em profundidade para determinar os componentes fortemente conexos de um dado digrafo D = (V,E).

Já vimos que na busca em profundidade em um digrafo, as arestas são particionadas em quatro categorias: arestas de árvore, arestas de retorno, arestas de avanço e arestas de cruzamento.

Nota: (1) As arestas de avanço podem ser ignoradas, visto que elas não afetam a conectividade de forte.

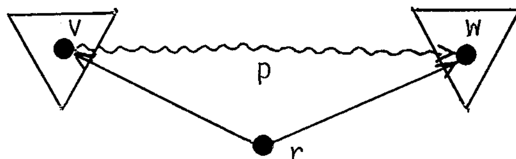
(2) As arestas de retorno e cruzamento que começam em v podem ir somente a vértices x em que num(v) > num(x).

Sejam v e w no mesmo componente fortemente conexo de D e supõe-se, sem perda de generalidade, que num(v) < num(w). Por definição, existe um caminho p em D de v para w. O caminho p não deve conter arestas de cruzamento de uma árvore da floresta geradora de busca em profundidade para uma outra árvore, desde que num(v) < num(w) e as arestas de cruzamento vão somente para os vértices com menor numeração, isto é:



é impossível.

Existe necessariamente uma subárvore em que suas arestas de cruzamento e arestas de retorno contém p; seja r a raiz da subárvore mínima. Por causa de minimalidade da subárvore, se p não passa por r, temos:



que é impossível. Então p passa por r, implicando que r está no mesmo componente fortemente conexo que v e w. Conclui-se que se S é um componente fortemente conexo de G, então os vértices de S definem uma árvore que é um subgrafo da floresta geradora. A volta desta afirmação não é verdadeira; nem toda subárvore corresponde a um componente fortemente conexo.

A identificação das raizes das subárvores, correspondendo aos componentes fortemente conexos, nos permitirá identificá-los, do mesmo modo que reconhecer os vértices de articulação nos permitiu identificar os componentes biconexos. Para reconhecer essas raízes, TARJAN (em TARJAN[71]) define a função <u>lowlink(v)</u> como sendo o número do vértice com menor numeração no mesmo componente fortemente conexo que v, que pode ser alcançado seguindo uma sequência de zero ou mais arestas de árvore seguidas de no máximo uma aresta de retorno ou uma aresta de cruzamento. Estes valores de lowlink nos dão a informação que precisamos para encontrar as raizes dos componentes fortemente conexos, pois v é uma raiz se e só se lowlink(v) = num(v). Estes valores de lowlink podem ser facilmente computados durante uma busca em profundidade.

A complexidade do Algoritmo (Ver TARJAN[71]) para determi-

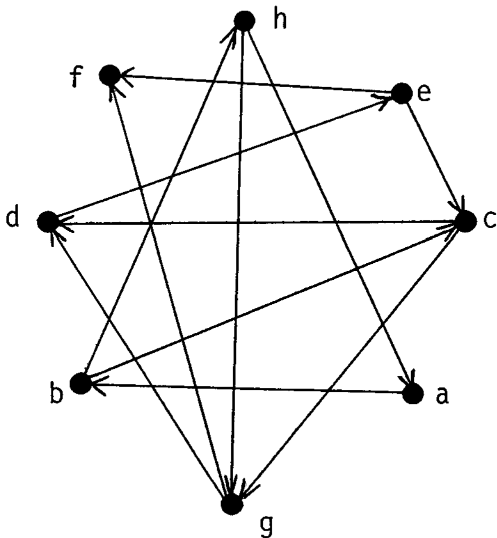
nar o problema aqui mencionado será também $O(|V| + |E|)$.

4.4) EXEMPLO

Determinar os componentes fortemente conexos do digrafo $D = (V,E)$ abaixo (Figura(VI.1)).

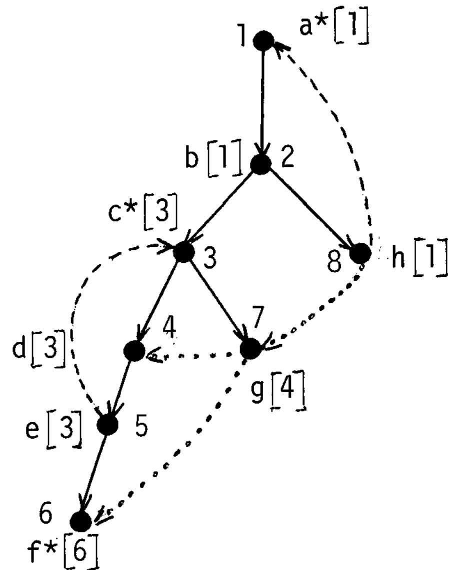
Digrafo $D = (V,E)$

Figura VI.1



Estrutura Direcionada de Busca em Profundidade.

Figura VI.2



Estrutura de Adjacência:

a → b

b → c, h

c → d, g

d → e

e → c, f

f → -

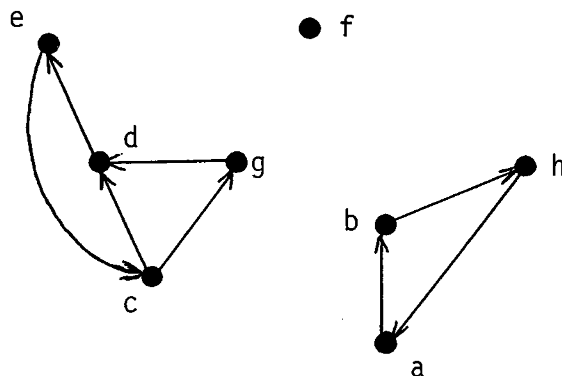
g → d, f

h → a, g

A Figura(VI.2) mostra uma árvore geradora direcionada de Busca em Profundidade com os valores de lowlink entre [ ] e as raízes dos componentes marcadas com * .

v	NUM(v)	LOWLINK	
a	1	1	←
b	2	1	
c	3	3	←
d	4	3	
e	5	3	
f	6	6	←
g	7	4	
h	8	1	

Logo, os componentes fortemente conexos de D = (V,E) são:



5) APLICAÇÃO - ORDENAÇÃO TOPOLÓGICA

5.1) REFERÊNCIA - KNUTH[47]

5.1.1) OUTRAS REFERÊNCIAS - GOLUMBIC[33]; REINGOLD, NIEVERGELT & DEO[60].

5.2) DESCRIÇÃO DO PROBLEMA

Determinar uma ordenação topológica dos vértices de um digrafo acíclico D = (V,E).

Em outras palavras, o problema acima seria: determinar uma rotulação dos vértices de um digrafo aciclico D = (V,E) com inteiros 1,2,...,|V|, tal que se existe uma aresta direcionada do vértice v_i para o vértice v_j então i < j.

Assim, dado um número n e um conjunto de pares inteiros (i,j), onde 1 ≤ i, j ≤ n, o problema de ordenação topológica é encontrar uma permutação $x_1, x_2, ..., x_n$ de {1, 2, ...,n} tal que i aparece a esquerda de j para todos os pares (i,j)que estão na entrada. É conveniente denotar os pares de entrada pela relação "i ≺ j" e lê-se "i precede j".

O problema de ordenação topológica é equivalente a se arranjar os vértices de um grafo direcionado em uma linha reta (ordem linear), de modo que todas as arestas vão da esquerda para a direita. Um tal arranjo é possível se e somente se não existem ciclos orientados no grafo, isto é, se e somente se na entrada não existem relações da forma: $i_1 \prec i_2$, $i_2 \prec i_3$, $i_3 \prec i_4$,

..., $i_k < i_1$, para $k \geq 1$.

5.3) MÉTODO - BUSCA EM PROFUNDIDADE

Uma aplicação simples da técnica de busca em profundidade em digrafos é determinar a ordenação topológica de um digrafo acíclico.

A ordenação topológica começa encontrando um vértice de $D = (V,E)$ que não tem arestas divergentes (tal vértice deve existir se D é acíclico) e marca este vértice com o maior número, a saber $|V|$. Este vértice é retirado de D, juntamente com todas as arestas convergentes a ele. Como o digrafo que restou é também acíclico, podemos repetir o processo e marcar o vértice que não tem arestas divergentes com o maior número ainda não utilizado, a saber $|V| - 1$ e assim por diante. Para conservar o algoritmo com $O(|V| + |E|)$, devemos evitar buscar o digrafo modificado por um vértice que não tem arestas divergentes.

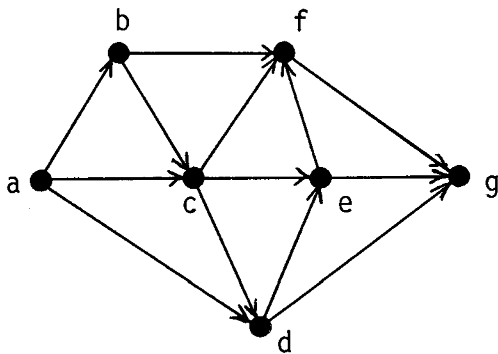
Fazemos, então, a execução de uma simples busca em profundidade em um dado digrafo acíclico. Em adição ao usual num, precisamos de um outro, que chamaremos rótulo, de tamanho $= |V|$, para conservar os rótulos dos vértices ordenados topologicamente. Logo existe uma aresta (u,v) em D, então rótulo(u) < rótulo(v).

A complexidade do algoritmo para resolver o problema aqui descrito é $O(|V| + |E|)$, visto que cada aresta é explorada uma vez.

5.4) EXEMPLO

Determinar a ordenação topológica dos vértices do digrafo acíclico D = (V,E) abaixo:

Digrafo D = (V,E)



Estrutura de adjacência:

a → b, c, d

b → c, f

c → d, e, f

d → e, g

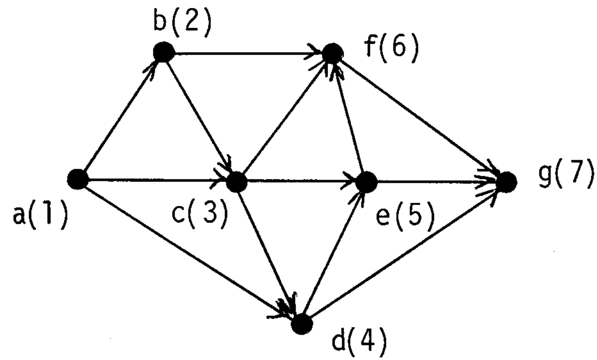
e → f, g

f → g

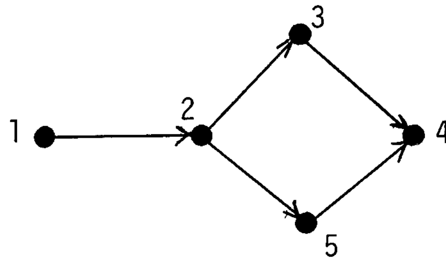
g → -

v	NUM(v)	ROTULO(v)
a	1	1
b	2	2
c	3	3
d	4	4
e	5	5
f	6	6
g	7	7

Logo, os vértices do digrafo D estão ordenados topologicamente, como pode ser visto a seguir:



OBS.: (1) O Digrafo abaixo não está ordenado topologicamente:



(2) A ordenação topológica é util na análise de redes de atividade, onde um projeto grande e complexo está representado por um digrafo em que os vértices correspondem aos objetivos no projeto e as arestas correspondem às atividades. A ordenação topológica fornece uma ordem em que os objetivos podem ser alcançados.

6) APLICAÇÃO - COMPONENTES FRACOS

6.1) REFERÊNCIAS - TARJAN[73]; PACAULT[55].

6.1.1) OUTRAS REFERÊNCIAS - TARJAN[71]; GRAHAM, KNUTH & MOTZKIN[34]

6.2) DESCRIÇÃO DO PROBLEMA

Determinar os componentes fracos de um digrafo $D = (V,E)$.

Sejam v e w dois vértices em um grafo direcionado D.

Existe um não-caminho de v para w se não existe um caminho de v para w.

Se existe um não-caminho de v_i para v_{i+1} para $1 \leq i \leq n$, então $v_1, v_2, \ldots, v_n$ é uma sequência de passos de não-caminhos de v_1 para v_n.

O conceito de componentes fracos de um grafo direcionado foi tratado por GRAHAM, KNUTH & MOTZKIN[34], da seguinte maneira: dois vértices v e w pertencem ao mesmo componente fraco se eles pertencem ao mesmo componente forte (isto é, se existe um caminho direcionado de v para w e de w para v) ou se existe uma sequência de passos de não-caminhos de v para w e de w para v.

6.3) MÉTODO - BUSCA EM PROFUNDIDADE

Em PACAULT[55] é apresentado um algoritmo para encontrar os componentes fracos de um digrafo $D = (V,E)$, com $O(|V| + |E|)$.

Na referência TARJAN[73], Tarjan apresenta um outro algoritmo (abaixo passos (1), (2) e (3)) com complexidade $O(|V|+|E|)$ que é parecido com o de Pacault, mas é mais direto. Ele utiliza o método de busca em profundidade e fornece os componentes fracos do digrafo dado.

Passos do Algoritmo de TARJAN:

(1) Encontrar os componentes fortes C_i de D e reduzir cada um a um simples vértice i, com uma aresta (i,j) no novo grafo se e somente se existe uma aresta (v,w) em D com $v \in C_i$ e $w \in C_j$. Os componentes fracos do grafo acíclico resultante correspondem aos componentes fracos de D. A redução dos componentes fortes requer $O(|V| + |E|)$ tempo usando busca em profundidade. (Já visto no algoritmo da aplicação anterior 4). Daqui por diante será suposto que D é acíclico.

(2) Numerar os vértices de D de 1 a $|V|$; então todas as arestas vão de um vértice com numeração mais baixa para um com numeração mais alta.

Esta operação é chamada ordenação topológica dos vértices de D e pode ser executada com $O(|V| + |E|)$ tempo usando busca em profundidade (Ver aplicação 5). Daqui por diante assumiremos que os vértices são identificados por número.

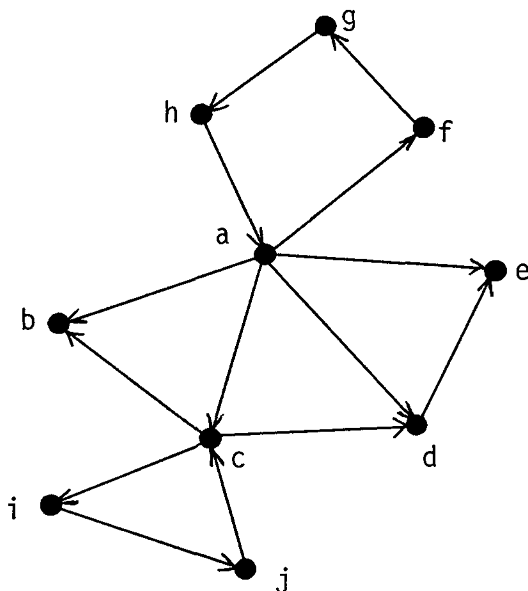
Resultado Importante: Os vértices de qualquer componente fraco de D são consecutivos (na mencionada ordenação) (Ver prova em TARJAN[73]).

(3) A referência TARJAN[73] apresenta nesse passo (3) um programa escrito em ALGOL-LIKE para calcular os componentes fracos. O tempo total de execução do passo (3) é $O(|V| + |E|)$.

Podemos então concluir que os passos (1) - (3) fornecem um algoritmo para encontrar os componentes fracos de um digrafo D e com complexidade total de $O(|V| + |E|)$.

6.4) EXEMPLO - Dado o digrafo D = (V,E) abaixo, determinar os componentes fracos de D.

Digrafo D = (V,E)



Estrutura de Adjacência:

a → b, c, d, e, f

b → -

c → b, d, i

d → e

e → -

f → g

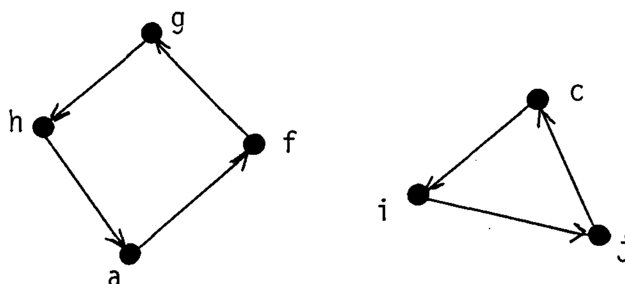
g → h

h → a

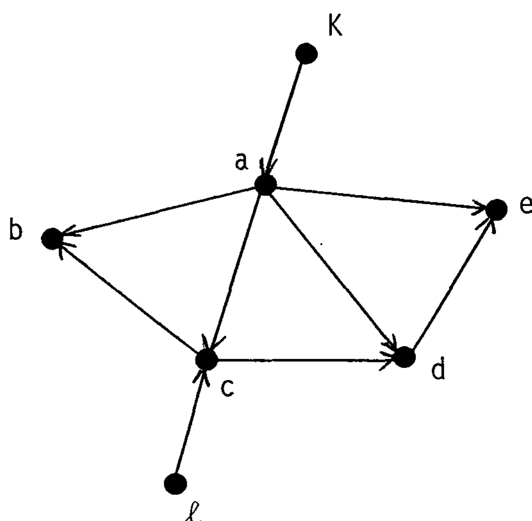
i → j

j → c

Usando a técnica de busca em profundidade, iremos encontrar os componentes fortes de D (como já foi usada na Aplicação (4) anterior). Eles são:



Aplicando o passo (1) do algoritmo aqui descrito, obtemos o seguinte grafo acíclico:



Depois de aplicar os passos (2) e (3) do algoritmo, chegamos a conclusão que os componentes fracos de D = (V,E) são:

{h, g, a, f} , {i, c, j} , {b, d, e}

7) APLICAÇÃO - TODAS AS ORDENAÇÕES TOPOLÓGICAS

7.1) REFERÊNCIAS - KNUTH & SZWARCFITER[46]

7.1.1) OUTRAS REFERÊNCIAS - REINGOLD, NIEVERGELT & DEO[60]
GOLUMBIC[33]; KNUTH[47]

7.2) DESCRIÇÃO DO PROBLEMA

Gerar todas as soluções do problema de ordenação topológica.

A solução para o problema de ordenação topológica não é única. Assim, na aplicação descrita acima, queremos um algoritmo para gerar todas as ordenações topológicas. Tal algoritmo serve como um exemplo instrutivo para vários problemas gerais importantes tais como backtracking, procedimentos para transformar recursão em iteração, manipulação de estruturas de dados e criação de programas bem estruturados.

7.3) MÉTODO - BUSCA EM PROFUNDIDADE

Uma maneira natural de resolver o problema acima é: seja x_1 um elemento que não tem predecessores; em seguida retiramos todas as relações da forma $x_1 \prec j$; seja x_2 um elemento diferente de x_1 e sem predecessores; então retiramos todas as relações da forma $x_2 \prec j$, etc. Não é difícil verificar (Ver

KNUTH[47]) que este método será sempre bem sucedido, a menos que exista um ciclo orientado na entrada. Além disso, de certa forma, esta é a única maneira de proceder, visto que x_1 deve ser um elemento sem predecessores, e x_2 deve ser sem predecessores quando todas as relações $x_1 < j$ são retiradas, etc. Esta observação leva, naturalmente, a um algoritmo que encontra todas as soluções para o problema de ordenação topológica; ele é um exemplo típico de uma procedure "backtrack" ou uma "busca em profundidade", onde em cada estágio consideramos um subproblema da forma: "Encontrar todas as maneiras de completar uma dada permutação parcial $x_1, x_2 \ldots x_k$ segundo uma ordenação topológica $x_1, x_2, \ldots x_n$". O método geral é ramificar todas as possíveis escolhas de x_{k+1}.

A referência KNUTH & SZWARCFITER[46] apresenta um algoritmo para resolver o problema de gerar todas as ordenações topológicas, e tem como Complexidade: $O(|V| + |E|)$ por ordenação gerada. Logo, se existem S ordenações, a complexidade é $O(|V| + |E|\ S)$.

OBS.: Seja $|V| = n$. Se S é máximo então o problema é equivalente a gerar todas permutações de $1, 2, \ldots, n \Longrightarrow S = n!$

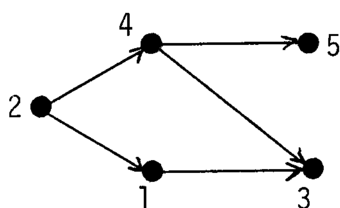
Porém, no pior caso, quando S é muito grande, a complexidade do algoritmo é melhor do que $n.n!$, pois é $e.n!$ ou seja:

$n + n(n-1) + n(n-1)(n-2) + \ldots + n! \lfloor n!\ e \rfloor - 1 = e$

constante por permutação, em média.

7.4) EXEMPLO

Gerar todas as ordenações topológicas do digrafo D' = (V,E) abaixo:



Temos como entrada:

1 < 3, 2 < 1, 2 < 4, 4 < 3, 4 < 5

Aplicando o algoritmo de KNUTH & SZWARCFITER[46], na saída teremos a impressão de cinco soluções, ou seja, todas as ordenações topológicas de D'; que são:

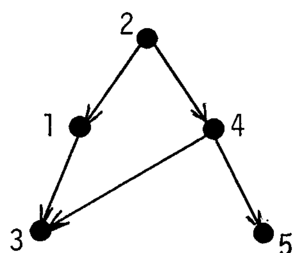
2 1 4 3 5

2 1 4 5 3

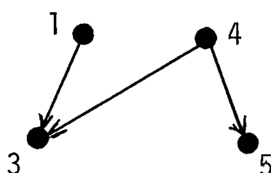
2 4 5 1 3

2 4 1 3 5

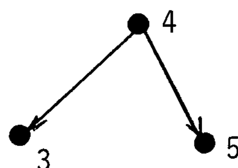
2 4 1 5 3



D = {2}

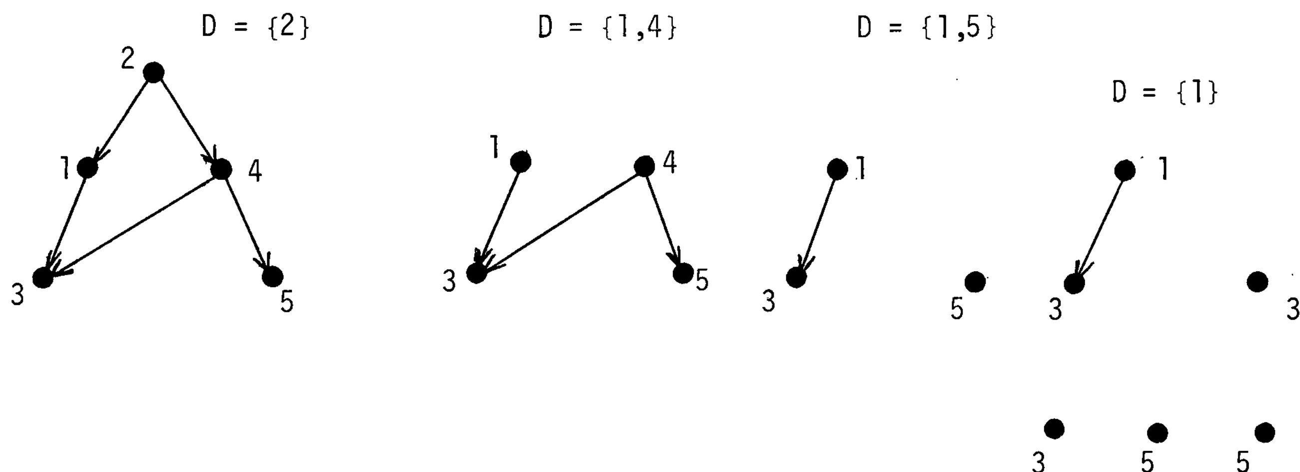


D = {1,4}



D = {4}

D = {3,5}



8) APLICAÇÃO - FECHAMENTO TRANSITIVO

8.1) REFERÊNCIA - SCHNORR[65]

8.1.1) OUTRAS REFERÊNCIAS - ROY[63]; WARSHALL[79]; FISCHER & MEYER[27]; ARLAZAROV, DINIC, KRONROD & FARADZEV[2]; EBERT[22]; BLONIARZ, FISCHER & MEYER[9].

8.2) DESCRIÇÃO DO PROBLEMA

Construir o fechamento transitivo de um digrafo D.

Seja D = (V,E) um digrafo. Seu supergrafo gerador máximo que preserva a alcançabilidade de D é denominado <u>fechamento transitivo</u>.

Suponhamos que os vértices de D são representados pelo conjunto: {1, 2, ... n}.

Sejam as listas de adjacências de D:

$L_i = \{j \mid \exists$ aresta de i para j$\}$, para $i = 1,2,\ldots, n$

Seja $m = \sum_{i=1}^{n} || L_i ||$ o número de arestas em D.

Sejam $L_i^r = \{j | \exists$ aresta de j para i$\}$ i = 1,2,..., n as listas de adjacências de arestas reversas do digrafo D. O vértice j é chamado um <u>sucessor</u> de i se existe um caminho de i para j. Neste caso, i é chamado um <u>predecessor de j</u>.

8.3) MÉTODO - BUSCA EM LARGURA OU BUSCA EM PROFUNDIDADE

Algoritmos que constroem o fechamento transitivo de um digrafo dado merecem considerável atenção. ROY[63] e mais tarde WARSHALL[79] propuseram um algoritmo que requer $O(|V|^2)$ passos. Em 1970, ARLAZAROV, DINIC, KRONROD & FARADZED[2], publicaram um algoritmo com limite de tempo $O(|V|^3 / \log |V|)$. FISCHER & MEYER[9] aplicaram a rápida matriz de multiplicação de Strassen e obtiveram um algoritmo para fechamento transitivo com limite de tempo $O(|V|^{2.81})$. BLONIARZ, FISCHER & MEYER[9] também propuseram um algoritmo com tempo médio $O(|V|^2 \log |V|)$.

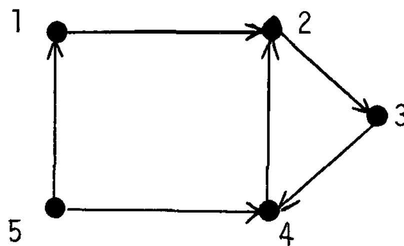
Recentemente, EBERT apresentou um algoritmo que usa busca em profundidade para computar o fechamento transitivo e que gasta $O(\max (|V| . |E|, |V|^2))$ operações.

O algoritmo que tomaremos como referência para a aplicação aqui descrita (o fechamento transitivo) é o de SCHNORR. Em SCHNORR[65] há uma descrição informal do algoritmo usado e que consiste de 5 estágios. Um desses estágios é a aplicação da técnica de busca em largura para achar uma lista S_i de sucessores, para cada vértice i. O tempo esperado de execução do algoritmo de SCHNORR é $O(n + m^*)$, onde m* é o número de

arestas no fechamento transitivo.

8.4) EXEMPLO

Achar o fechamento transitivo para o digrafo D = (V,E) abaixo:



Aplicando os 5 estágios do algoritmo de SCHNORR[65]:

Listas de adjacências:

$L_1 = \{2\}$

$L_2 = \{3\}$

$L_3 = \{4\}$

$L_4 = \{2\}$

$L_5 = \{1,4\}$

Listas de adjacências de arestas reversas:

$L_1^r = \{5\}$

$L_2^r = \{1,4\}$

$L_3^r = \{2\}$

$L_4^r = \{3,5\}$

$L_5^r = \{\ \}$

Sucessores:

$S_1 = \{2,3,4\}$

$S_2 = \{3,4\}$

$S_3 = \{2,4\}$

$S_4 = \{2,3\}$

$S_5 = \{1,2,3,4\}$

Predecessores:

$P_1 = \{5\}$

$P_2 = \{1,3,4,5\}$

$P_3 = \{1,2,4,5\}$

$P_4 = \{1,2,3,5\}$

$P_5 = \{\ \}$

Para achar os sucessores S_i, utilizamos a técnica de busca em largura. (Ver algoritmo em SCHNORR[65]).

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 = 3 + 1 = 4$$

$\| S_1 \| = 3 < 4$

$\| S_2 \| = 2 < 4$

$\| S_3 \| = 2 < 4$

$\| S_4 \| = 2 < 4$

$\| S_5 \| = 4 = 4$

As listas de adjacências do fechamento transitivo são dadas por:

L^*_i, $L^*_i = S_i \cup \{j \mid i \in P_j\} \cup \{j \mid \| S_i \| = \| P_j \| = \left\lfloor \frac{n}{2} \right\rfloor + 1\}$

$L_1^* = \{2,3,4\} \cup \{2,3,4\} \cup \{\ \} = \{2,3,4\}$

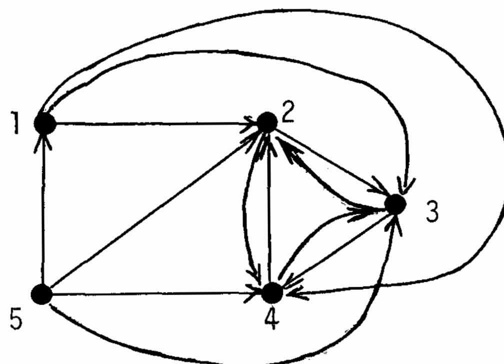
$L_2^* = \{3,4\} \cup \{3,4\} \cup \{\ \} = \{3,4\}$

$L_3^* = \{2,4\} \cup \{2,4\} \cup \{\ \} = \{2,4\}$

$L_4^* = \{2,3\} \cup \{2,3\} \cup \{\ \} = \{2,3\}$

$L_5^* = \{1,2,3,4\} \cup \{1,2,3,4\} \cup \{2,3,4\} = \{1,2,3,4\}$

Logo, o fechamento transitivo de D será:



9) APLICAÇÃO - TODOS OS CAMINHOS DE UM GRAFO

9.1) REFERÊNCIAS - —

9.2) DESCRIÇÃO DO PROBLEMA

A partir de um vértice qualquer s de um grafo G (não-direcionado ou direcionado), achar todos os caminhos simples existentes em G.

Seja G = (V,E) o grafo dado.
Seja s o vértice dado.
Caminho simples é aquele em que todos os vértices são distintos.

9.3) MÉTODO - BUSCA EM PROFUNDIDADE

O algoritmo de Busca em Profundidade para resolver o problema descrito acima é classificado como NÃO-RESTRINGIDO, ou seja, um vértice pode ser explorado mais de uma vez, logo uma aresta poderá ser explorada mais de uma vez. (Ver Capítulo V anterior).

O algoritmo será:

<u>Procedimento</u> BP(v);

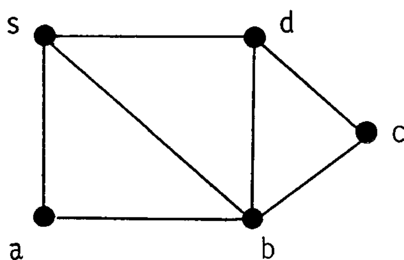
1. Visitar e marcar v; Introduzir v na Pilha;
2. Listar pilha;
3. <u>Para</u> w ∈ Adj(v) <u>Faça</u> <u>Se</u> w desmarcado
   <u>então</u> BP(w);
4. Desmarcar v; Retirar v da pilha

<u>Fim</u>.

A COMPLEXIDADE deste algoritmo será: $O(|V| + |E|)$ por caminho.

9.4) <u>EXEMPLO</u>

A partir do vértice s, achar todos os caminhos simples existentes no grafo G = (V,E) abaixo:



Estrutura de adjacência:

s → a, b, d

a → s, b

b → a, s, d, c

c → b, d

d → c, b, s

Aplicaremos o algoritmo aqui descrito, sendo <u>s</u> o primeiro vértice a ser explorado. E assim, recursivamente, iremos explorar os demais vértices, listando todos os caminhos simples existentes em G que são:

s	s b d c
s a	s b c
s a b	→ s b c d
s a b d	s d
s a b d c	s d c
s a b c	s d c b
→ s a b c d	s d c b a
s b	s d b
s b a	s d b a
s b d	s d b c

10) APLICAÇÃO - TODOS OS CAMINHOS MAXIMAIS DE UM GRAFO

10.1) REFERÊNCIAS - —

10.2) DESCRIÇÃO DO PROBLEMA

A partir de um vértice qualquer s de um grafo G (não-direcionado ou direcionado), achar todos os caminhos simples maximais existentes em G.

Sejam: G = (V,E) o grafo dado e s o vértice dado.
Caminho Simples: é aquele em que todos os vértices são distintos.
Caminho Maximal: o caminho $v_1, v_2, \ldots, v_k$ é dito ser um caminho maximal de um grafo G quando ele for simples e não existir nenhum caminho do tipo: $v_1, v_2 \ldots, v_k, z$, onde z é um vértice de G diferente de $v_1, \ldots v_k$.

10.3) MÉTODO - BUSCA EM PROFUNDIDADE

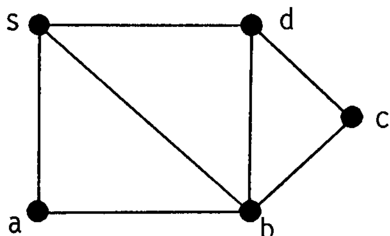
O algoritmo de Busca em Profundidade para resolver o problema aqui descrito , é classificado como NÃO-RESTRINGIDO. (Ver Capítulo V anterior). Ele será:

```
     Procedimento BPM (v);
1.   FLAG(v):= FALSO;
     u:= PAI(v);
2.   Visitar e marcar v; Inserir v na Pilha;
3.   Para w ∈ Adj(v) Faça Se w desmarcado então FLAG(v):= VERD
                                                BPM(w);
4.   Se FLAG(v) = FALSO e u ≠ w
5.            então  Listar Pilha.
6.             Desmarcar v; Retirar v da pilha;
     Fim
```

A COMPLEXIDADE deste algoritmo será $O(|V| + |E|)$ por caminho.

10.4) EXEMPLO

A partir do vértice s, achar todos os caminhos simples maximais existentes no grafo G = (V,E) a seguir:



Estrutura de adjacência:

s → a, b, d

a → b, s

b → d, c, a, s

c → b, d

d → c, b, s

Inicialmente aplicaremos o algoritmo BPM(s) para o vértice s; Depois: BPM(a) , BPM(b) , BPM(d) ,
FLAG(a)=Verd , FLAG(b)=Verd , FLAG(d)=Verd,
BPM(c)
FLAG(c)=Falso

Aqui chegamos no passo (4) e entraremos no <u>então</u> do <u>Se</u> listando o caminho maximal: s a b d c.

Continuando a aplicar o algoritmo, teremos como resultado final os seguintes caminhos maximais:

s a b d c

s a b c d

s b a

s b d c

s b c d

s d c b a

s d b a

s d b c

OBS.: A árvore que encontra os caminhos maximais será exponencial. No pior caso (por exemplo, quando o grafo é completo), o número de vértices será (n-1)!

11) APLICAÇÃO - CICLOS SIMPLES DE UM GRAFO

11.1) REFERÊNCIAS - SZWARCFITER & LAUER[67]

11.1.1) OUTRAS REFERÊNCIAS - MATETI & DEO[53]; JOHNSON[42]; WEINBLATT[80]; TARJAN[74]; KARAYIANNIS & LOIZOU[44].

11.2) DESCRIÇÃO DO PROBLEMA

Enumerar todos os ciclos simples de um grafo (direcionado ou não direcionado).

O problema de enumerar todos os ciclos simples de um grafo não direcionado tem muito em comum com o problema de enumerar todos os ciclos direcionados de um digrafo, e podemos considerá-los em paralelo.

De fato, qualquer algoritmo que enumera todos os ciclos de um digrafo pode ser usado para enumerar todos os ciclos de um grafo não direcionado, com uma pequena alteração.

Seja: C → Número de ciclos simples de um digrafo D = (V,E)

11.3) MÉTODO - BUSCA EM PROFUNDIDADE

A maioria dos melhores algoritmos conhecidos para enumerar os ciclos simples de um digrafo são baseados na técnica de busca em profundidade.

Entre o grande número de algoritmos de ciclos analisados por Prabhaker e Deo (Ver MATETI & DEO[53]), o algoritmo de JOHNSON apresenta um melhor limite de tempo e espaço, particularmente um limite linear com o tamanho do grafo, por ciclo.

Szwarcfiter e Lauer também apresentam um algoritmo que tem um limite de tempo e espaço de pior caso semelhante ao de Johnson. Contudo o limite do algoritmo de Szwarcfiter é sempre menor ou igual ao de Johnson. (Complexidade no pior caso: de tempo = $O(|V| + |E|)C$ e de espaço = $O(|V| + |E|)$, em ambos os algoritmos).

Um dos primeiros algoritmos que utilizava busca em profundidade para enumerar todos os ciclos simples usava um algoritmo BACKTRACK não restringido. Ele empregava a técnica de construir caminhos simples a partir de um vértice inicial s. (Chamado vértice de partida em uma pilha). Um vértice v é marcado e inserido na pilha. Depois que v é retirado da pilha (só no caso em que v = s), é realizada uma operação "DESMARCAR" que para tal algoritmo significa "Desmarcar Vértice v".

Para Tarjan (Ref. TARJAN[74]) "DESMARCAR" seria: "Se v estava envolvido em um ciclo então desmarque v e todos os vértices de um conjunto Z, que consiste dos vértices que são marcados e entraram na pilha por último, depois de v". Devido a tais restrições, o algoritmo de TARJAN passa a conter um algo

ritmo BACKTRACK parcialmente restringido.

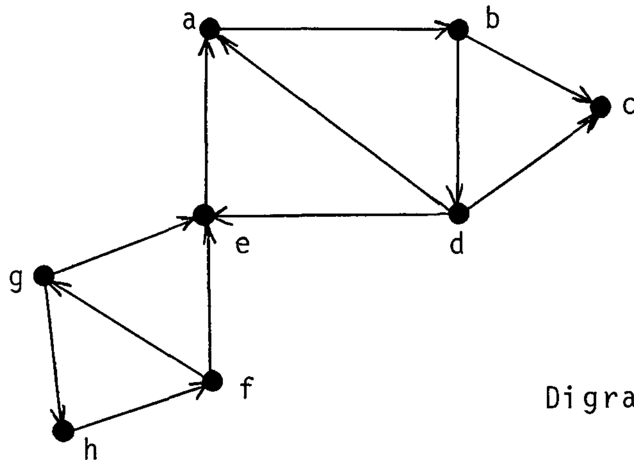
Para Johnson (Ref. JOHNSON[42]), "DESMARCAR" seria: "Se v estava envolvido em um ciclo então desmarque v e todos os vértices de um conjunto Z_1, $Z_1 \subseteq Z$. Z_1 consiste de vértices $x \in Z$, para os quais existe um caminho de x para v, envolvendo somente vértices de Z". O algoritmo de Johnson é também parcialmente restringido. Observação: tal algoritmo acha primeiro os componentes fortemente conexos do grafo, e a partir de cada componente, acha os ciclos.

Para Szwarcfiter e Lauer (Ref. SZWARCFITER & LAUER[67]) "DESMARCAR" tem o mesmo significado que o algoritmo de Johnson, exceto no caso em que as arestas que levam a vértices x desmarcados recentemente são inseridas de novo nas listas de adjacência.

A principal diferença entre o algoritmo de Johnson e o de Szwarcfiter e Lauer, é que este último descobre um ciclo simples logo que ele é gerado, ou seja, logo que ele apareça nas posições do topo da pilha.

11.4) EXEMPLO

Dado o digrafo D = (V,E), enumerar todos os ciclos de D.



Digrado D = (V,E)

Aplicando o algoritmo de SZWARCFITER & LAUER:

Estrutura de adjacência:

a → b

b → c, d

c → -

d → a, c, e

e → a

f → e, g

g → e, h

h → f

Componentes fortemente conexos do digrafo D acima:

1º) a b d e

2º) c

3º) f, g, h

Para cada componente conexo serão achados os ciclos nele contidos.

Por exemplo, para o componente - 1º) a b d e

Estrutura de Adjacência do 1º componente

a → b

b → d

d → a, e

e → a

Seja s = a (inicial)

	v = d; w = a = s		v = e; w = a = s
	listar o ciclo:	e	listar o ciclo:
d	a b d a	d	a b d e a
b		b	
a		a	

Logo:

Para o 1º componente, temos os ciclos: (a b d a) e (a b d e a)

Para o 3º componente, temos o ciclo: (f g h f)

12) APLICAÇÃO - MENOR CAMINHO EM DIGRAFOS ACÍCLICOS VALORADOS

12.1) REFERÊNCIA - SZWARCFITER[68]

12.2) DESCRIÇÃO DO PROBLEMA

Encontrar o menor caminho em um digrafo acíclico valorado.

Problemas de menores caminhos constituem uma área importante em algoritmos de grafos, principalmente porque existem diferentes aplicações que utilizam tais algoritmos.

Digrafos acíclicos constituem, também, uma classe importante de digrafos, com muitas aplicações específicas.

A referência SZWARCFITER[68] apresenta diferentes problemas para encontrar menores caminhos em digrafos acíclicos, tais como: problema de menor caminho entre dois vértices; menores caminhos de todos os vértices a um vértice fixo, e de um vértice fixo a todos os vértices; menores caminhos entre cada par de vértices, etc.

Seja D = (V,E) um digrafo acíclico, cujas arestas possuem valores.

12.3) MÉTODO - BUSCA EM PROFUNDIDADE

OBS.: Apesar de antigo, é desconhecido o autor do primeiro algoritmo linear para o problema acima descrito.

Szwarcfiter apresenta um algoritmo para achar o menor caminho entre dois vértices, que utiliza uma procedure backtracking recursiva (PATH), que executa uma busca em profundidade no digrafo. No final desta busca, o menor caminho de um vértice a a um vértice b é determinado. Tal algoritmo requer $O(|V| + |E|)$ espaço e tempo.

O algoritmo de SZWARCFITER[68] para encontrar os menores caminhos de todos os vértices a um dado vértice, faz algumas modificações no algoritmo citado acima, mas continua a manter a mesma procedure backtracking (PATH) que utiliza busca em profundidade. Este algoritmo também requer $O(|V| + |E|)$ espaço e tempo. Também, os algoritmos que determinam menores caminhos de um dado vértice a todos os vértices, e menores caminhos entre cada par de vértices, continuam a utilizar a procedure re

cursiva de busca em profundidade.

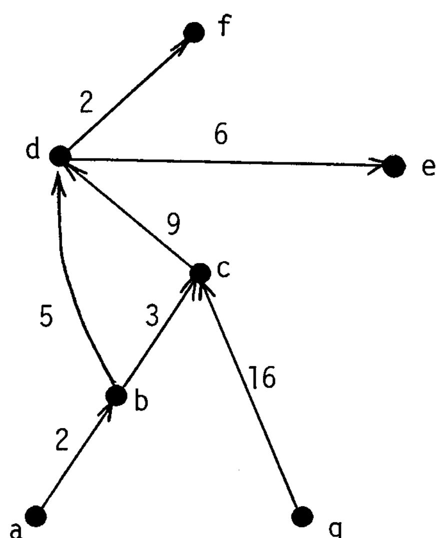
OBS.: Uma solução eficiente para o problema de menor caminho entre dois vértices de um digrafo geral (onde ciclos podem estar presentes) foi proposto por Dijkstra em 1959 (Ver DIJKSTRA[18]). O método de Dijkstra tem um limite de tempo de $O(|V|^2)$ e foi projetado para digrafos com valores não negativos. Ver também DREYFUS[21].

12.4) EXEMPLO

Achar o comprimento do menor caminho entre o vértice a e o vértice f do digrafo acíclico abaixo:

OBS.: Será aplicado o algoritmo da Ref. SZWARCFITER[68].

Digrafo D:



Estrutura de adjacência:

a → b

b → c, d

c → d

d → e, f

e → -

f → -

g → c

Comprimentos: lenght(v):

a → ∞	9
b → ∞	7
c → ∞	11
d → ∞	2
e → ∞	
f → 0	
g → ∞	

Logo, o menor caminho entre a e f é 9.

13) APLICAÇÃO - MENOR CAMINHO EM GRAFOS NÃO VALORADOS

13.1) REFERÊNCIA - EVEN[25]

13.2) DESCRIÇÃO DO PROBLEMA

Encontrar o menor caminho entre um vértice qualquer s e um outro t que pertencem a um grafo G não direcionado ou direcionado, cujas arestas não possuem nenhum valor.

Seja G = (V,E) um grafo não direcionado, e s e t dois de seus vértices.

O problema aqui descrito é muito simples, e uma das referências onde podemos encontrá-lo é em EVEN[25]. Apesar de antigo, é desconhecido o autor do primeiro algoritmo linear para este problema.

13.3) MÉTODO - BUSCA EM LARGURA

Para resolver o problema aqui descrito, a referência EVEN[25] apresenta um algoritmo (que encontra o comprimento de um menor caminho) composto dos seguintes passos:

(1) Rotule o vértice s com 0 (zero). Seja i = 0.

(2) Encontre todos os vértices não rotulados que estão ligados por uma aresta a vértices rotulados i. Se não existem tais vértices, t não é alcançado de s; pare. Se existem, rotule-os i + 1.

(3) Se t é rotulado, vá para o passo (4). Se não, incremente i de 1 e vá para o passo (2).

(4) O comprimento de um menor caminho de s para t é i + 1; pare.

Comentários:

a) Se não existe caminho de s para t, o algoritmo termina depois que todos os vértices que são alcançados de s tenham sido rotulados.

b) Se t é alcançado de s, o algoritmo fornece os graus dos vértices; ele pára uma vez que t é rotulado, e deixa todos os vértices, que estão mais distantes de s, não rotulados.

c) Se o grafo é direcionado, e o caminho é solicitado ser direcionado, o algoritmo ainda é aplicado de modo que o passo (2) é trocado como se segue: Encontre todos os vértices não rotulados que são alcançados por uma aresta que procede de um vértice rotulado i; e assim por diante.

Uma vez que o algoritmo aqui citado termine como sucesso (passo (4)), a referência EVEN[25] apresenta também um outro algoritmo (de volta) para encontrar <u>um</u> menor caminho.

A complexidade do algoritmo aqui descrito será $O(|V|+|E|)$.

<u>OBS.</u>: Esta técnica de Busca em Largura, aplicada no algoritmo aqui descrito, irá encontrar também o menor caminho de <u>s</u> para todos os demais vértices do grafo que foram alcançados durante a busca.

13.4) <u>EXEMPLO</u>

Dado o grafo não direcionado G a seguir, encontrar o menor caminho de <u>s</u> a <u>t</u>.

Estrutura de Adjacência:

s → a, f

a → s, b, d

b → a, c, e

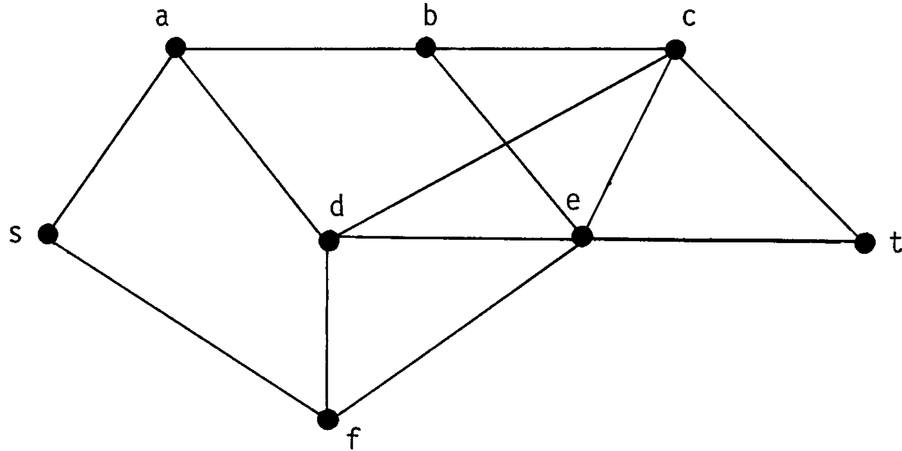
c → b, d, e, t

d → a, c, e, f

e → b, c, d, f, t

f → s, d, e

t → c, e

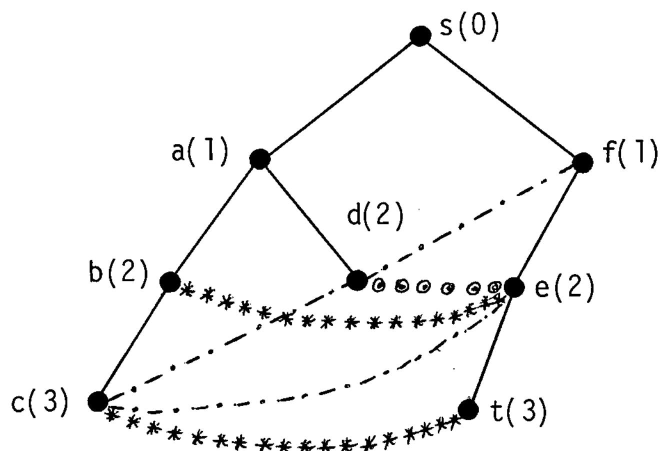


Grafo G

Aplicando o algoritmo visto anteriormente, que utiliza a técnica de busca em largura, temos:

Primeiro s é rotulado 0; então a e f são rotulados 1; b, d, e são rotulados 2; c e t são rotulados 3. Visto que t está rotulado com 3, o comprimento de um menor caminho de s para t é 3. (Ver Estrutura de Busca em Largura abaixo). Aplicando o algoritmo (de volta) da ref. EVEN[25] podemos obter um menor caminho como sendo a seguinte sequência de vértices: t e f s.

Estrutura de Busca em Largura:



OBS.: ——> Aresta de árvore

-.-.-> Aresta entre tio e sobrinho

⊙ ⊙ ⊙ > Aresta entre irmãos

***** > Aresta entre primos

Além de achar o comprimento do menor caminho de s a t , encontramos também o comprimento do menor caminho de s a todos os demais vértices de G (que são os rótulos entre parenteses)

14) APLICAÇÃO - MAIOR CAMINHO EM DIGRAFOS ACÍCLICOS VALORADOS

14.1) REFERÊNCIA - SZWARCFITER[68]

14.1.2) OUTRAS REFERÊNCIAS - KLEIN[45]; LASS[48]; ELMAGHRABY[24]; FURTADO[29]; EVEN[25]; PRICE[56].

14.2) DESCRIÇÃO DO PROBLEMA

Encontrar o maior caminho em um digrafo acíclico com valores.

Seja D = (V,E) um digrafo acíclico, cujas arestas possuem valores.

O problema acima descrito está diretamente relacionado com redes de PERT (Project Evaluation and Review Technique), para um caminho crítico em que uma rede é necessariamente o maior caminho no digrafo correspondente. Portanto o presente

problema é também estudado na vasta literatura de PERT, CPM (Critical Path Method) e redes de projetos de escalonamento.

Klein (em KLEIN[45]), Lass (em LASS[48]), Elmaghraby (em ELMAGHRABY[24]), Furtado (em FURTADO[29]), Even (em EVEN[25]), Price (em PRICE[56]), e outros estudaram o problema de caminho crítico (ou maior caminho) e soluções foram apresentadas, que geram algoritmos eficientes. Porém, não há uma primeira referência para este tipo de problema.

14.3) MÉTODO - BUSCA EM PROFUNDIDADE

O método descrito em SZWARCFITER[68] utiliza uma procedure backtracking recursiva PATHL para resolver o problema aqui descrito. Esta procedure PATHL é semelhante a procedure PATH (do algoritmo para achar o menor caminho, vista na aplicação 12), com algumas diferenças.

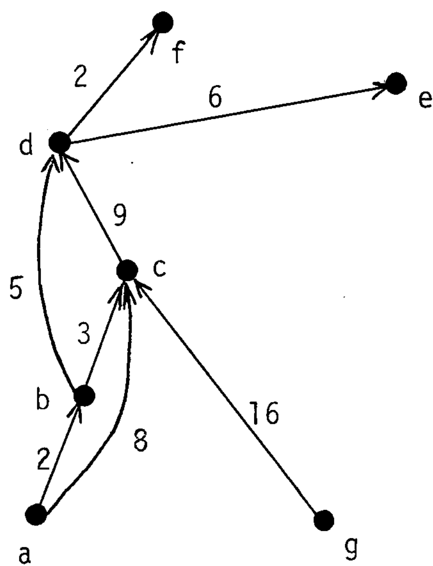
A procedure PATHL executa uma busca em profundidade no digrafo acíclico, e no final da computação de uma chamada a PATHL(v), o maior caminho, começando no vértice v, foi calculado.

A complexidade deste algoritmo será $O(|V| + |E|)$.

OBS.: O problema de encontrar o maior caminho num digrafo geral é NP-completo.

14.4) EXEMPLO

Encontrar o maior caminho no digrafo acíclio abaixo:



Seguindo o algoritmo da Ref. SZWARCFITER[68] , que usa busca em profundidade, obteremos:

Comprimentos (lenght(v) dos vértices alcançados durante a Busca em Profundidade):

a →	$-\infty$	20	23
b →	$-\infty$	18	
c →	$-\infty$	15	
d →	∞	2	6
e →	0		
f →	0		
g →	$-\infty$	31	

Logo, o maior caminho tem comprimento igual a 31.

15) APLICAÇÃO - GRAFOS BIPARTITE

15.1) REFERÊNCIAS - REINGOLD, NIEVERGELT & DEO[60]; BAASE[3]; GOLUMBIC[33].

15.2) DESCRIÇÃO DO PROBLEMA

Determinar se um dado grafo não direcionado é bipartite.

Um grafo não direcionado $G = (V,E)$ é bipartite se V pode ser particionado em dois conjuntos V_1 e V_2, de modo que cada aresta em E une um vértice em V_1 a um vértice em V_2.

OBS.: O problema acima descrito é muito simples, não tendo, portanto, uma primeira referência. Como consulta, podemos utilizar REINGOLD, NIEVERGELT & DEO[60], BAASE[3] OU GOLUMBIC[33].

15.3) MÉTODO - BUSCA EM LARGURA

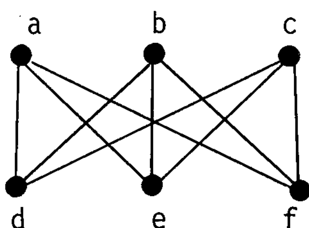
Se ao aplicarmos a técnica de busca em largura em um determinado grafo não direcionado, obtermos uma estrutura de busca em largura somente formada por arestas de árvore ou arestas entre tio e sobrinho, então o grafo será bipartite. Isto será equivalente a não haver ciclos de comprimento ímpar.

COMPLEXIDADE: $O(|V| + |E|)$

15.4) EXEMPLO

Determinar se os seguintes grafos são bipartites (utilizar a técnica de busca em largura):

1º)



Estrut. Adjacen.

a → d, e, f

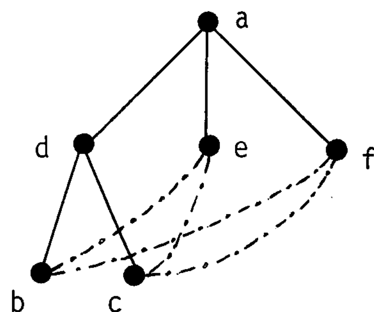
b → d, e, f

c → d, e, f

d → a, b, c

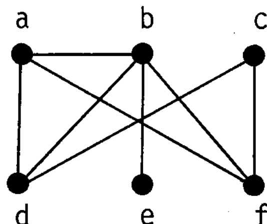
e → a, b, c

f → a, b, c



É bipartite

2º)



Estrut. Adjacen.

a → b, d, f

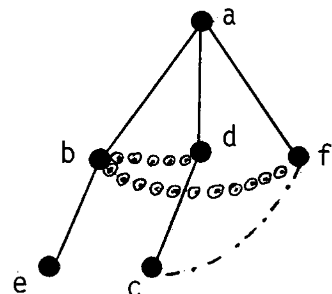
b → a, d, e, f

c → d, f

d → a, b, c

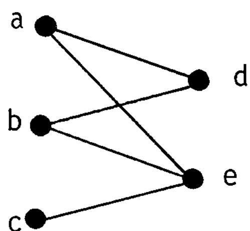
e → b

f → a, b, c



Não é bipartite (pois existe arestas entre irmãos)

3º)



Estrut. Adjacen.

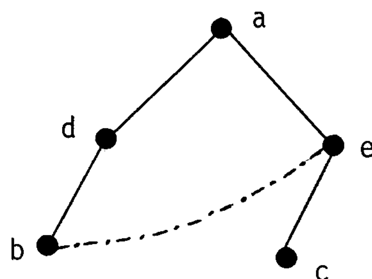
a → d, e

b → d, e

c → e

d → a, b

e → a, b, c



É bipartite

16) APLICAÇÃO - DOMINADORES

16.1) REFERÊNCIA - TARJAN[71]; TARJAN[72].

16.1.1) OUTRAS REFERÊNCIAS - AHO[1]; PURDON & MOORE[57]; BELL & ABEL[7].

16.2) DESCRIÇÃO DO PROBLEMA

Encontrar o dominador imediato (IDOM(v)) de cada vértice (v) de um grafo direcionado.

Suponha que D = (V,E) seja um grafo direcionado, com |V| vértices e |E| arestas, e um vértice inicial s.

Se um vértice d está em todo caminho do vértice s a um vértice i , então d é chamado um dominador de i.

Se d é um dominador de i e todos os outros dominadores d' de i também dominam d , então d é chamado em dominador imediato de i.

OBS.: O dominador imediato se existir, é único.

16.3) MÉTODO - BUSCA EM PROFUNDIDADE

Inicialmente aplicamos a técnica de busca em profundidade (BP) no digrafo D, obtendo uma Estrutura de Busca em Profundidade, composta por arestas de árvore (T) arestas de retorno (R), arestas de avanço (A) e arestas de cruzamento (C).

Depois aplicamos quatro transformações na Estrutura de Busca em Profundidade, que irão preservar os dominadores. Elas são denominadas:

1) "Substituição de arestas de retorno"
2) "Substituição de arestas de cruzamento"
3) "Retirada de arestas de avanço"
4) "Retirada de arestas de retorno"

1) "Substituição de arestas de retorno"

Suponha que uma busca em profundidade (BP) em um grafo direcionado D seja executada, e que todos os vértices de D são alcançáveis do vértice inicial s.

Para algum vértice v, seja $F(v) = \{w \mid w \neq v$ e $\exists u$, tal que existe um caminho de w para v, e de v para u em T, e (u,w) é uma aresta de retorno de D$\}$. Seja HIGHPT(v) o vértice numerado mais alto em F(v), se F(v) não é vazio (Ver algoritmo para calcular HIGHPT(v) em TARJAN[72]). Como cada elemento em F(v) é um ancestral de v em T, é claro que $w \in F(v)$ implica que existe um caminho de w para HIGHPT(v).

A transformação "Substituição de arestas de retorno" consiste em retirar todas as arestas de retorno e adicionar uma nova aresta de retorno (v, HIGHPT(v)) para cada vértice v para o qual HIGHPT(v) está definido.

2) "Substituição de arestas de cruzamento"

Seja D um grafo que foi explorado usando busca em profundidade (BP) e cujas arestas de retorno foram substituidas usando a 1ª transformação aqui descrita.

Seja D(i) o subgrafo de D que contém todas as arestas de árvore em D mais todas as arestas que vão para vértices v, tal que NUM(v) > i.

O i-ésimo semidominador do vértice v (SDOM(i,v)) será definido como o dominador imediato de v em D(i). Será assumido que v ≠ 1 (isto é, v não é o vértice inicial). Os valores de SDOM nos dirão como converter arestas de cruzamento em arestas de avanço.

Seja (u,v) uma aresta de cruzamento na estrutura de BP.

A transformação "substituição de arestas de cruzamento" consiste em retirar (u,v) e adicionar a aresta (SDOM(v,u), v).

Se a aresta resultante dessa transformação ainda for aresta de cruzamento, aplicamos novamente a transformação aqui mencionada, e continuamos a agir desta maneira, até que encontremos uma aresta que seja de avanço.

3) "<u>Retirada de arestas de avanço</u>"

Sejam (u,v) e (u_1,v) duas arestas de avanço, com $u_1 > u$.

A transformação "Retirada de arestas de avanço" consiste simplesmente em retirar a aresta (u_1,v).

4) "<u>Retirada de arestas de retorno</u>"

Seja v um vértice em D tal que uma aresta de retorno (v, HIGHPT(v)) deixa v, no máximo uma aresta de avanço (dita (u,v)) entra em v, e nenhuma aresta de cruzamento ou aresta de retorno entram em v.

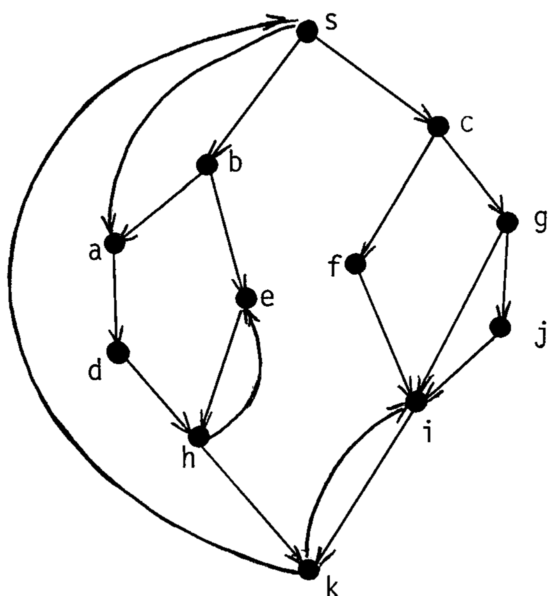
A transformação "Retirada de arestas de retorno" consiste em retirar a aresta de retorno (v, HIGHPT(v)) e adicionar (u, HIGHPT(v)) caso (u,v) seja definina e (u, HIGHPT(v)) seja uma aresta de avanço (isto é, u < HIGHPT(v)).

<u>OBS.</u>: O algoritmo de TARJAN[72] para achar dominadores tem complexidade: $O(|V| + |E|)$ de espaço e $O(|V| \log |V| + |E|)$ de tempo.

16.4) EXEMPLO

Dado o grafo direcionado abaixo, encontrar o dominador imediato de cada vértice do grafo:

Grafo Direcionado D:

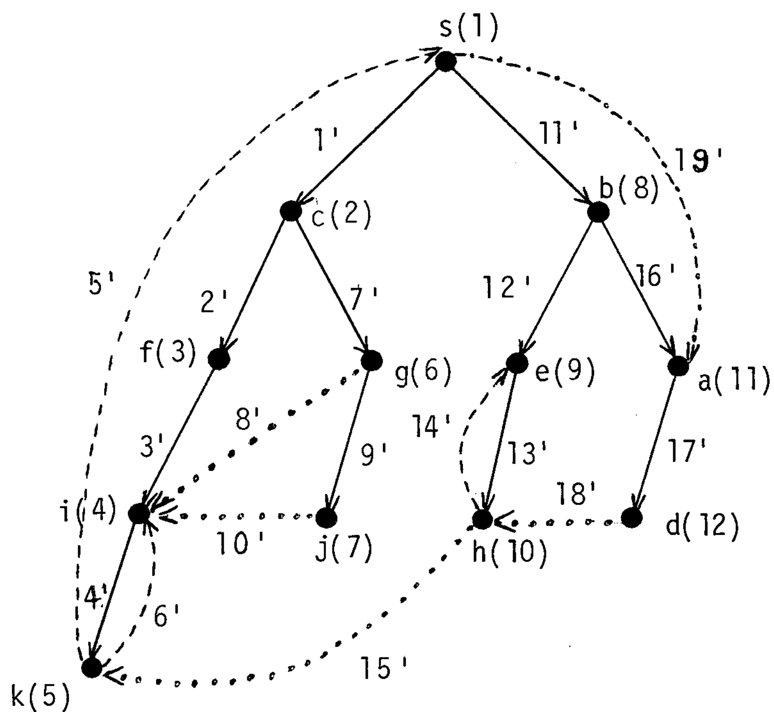


Estrutura de adjacência:

s → c, b, a
a → d
b → e, a
c → f, g
d → h
e → h
f → i
g → i, j
h → g, k
i → k
j → i
k → s, i

Aplicando a técnica de Busca em Profundidade (BP) em D:

Estrutura de Busca em Profundidade:



———> Aresta de árvore

-----> Aresta de retorno

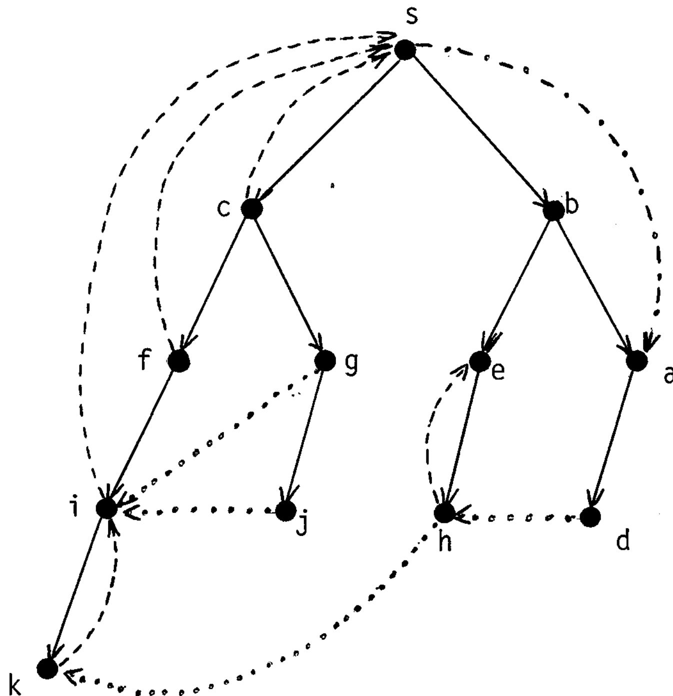
.....> Aresta de cruzamento

-.-.-> Aresta de avanço

OBS.: Ao lado dos vértices, entre parenteses, estão os valores de NUM(v) → indica a ordem em que os vértices foram visitados.

1ª Tranformação - Substituição de arestas de retorno"

Será retirada a aresta de retorno (k,s) e serão adicionadas as arestas de retorno: (c,s), (f,s) e (i,s).



2ª Transformação - "Substituição de arestas de cruzamento".

Arestas de Cruzamento: (g,i), (j,i), (h,k) e (d,h)

1) Para (g,i), achar (SDOM(v,u), v).

SDOM(v,u) = SDOM(i,g) = c

Logo, (SDOM(i,g), i) = (c,i)

2) (j,i)

$(\underbrace{SDOM(i,j)}_{g}, i) = (g,i)$ que ainda é aresta de cruzamento. Terá que aplicar novamente a 2ª transformação.

Logo, (SDOM(i,g), i) = (c, i)

3) (h,k)

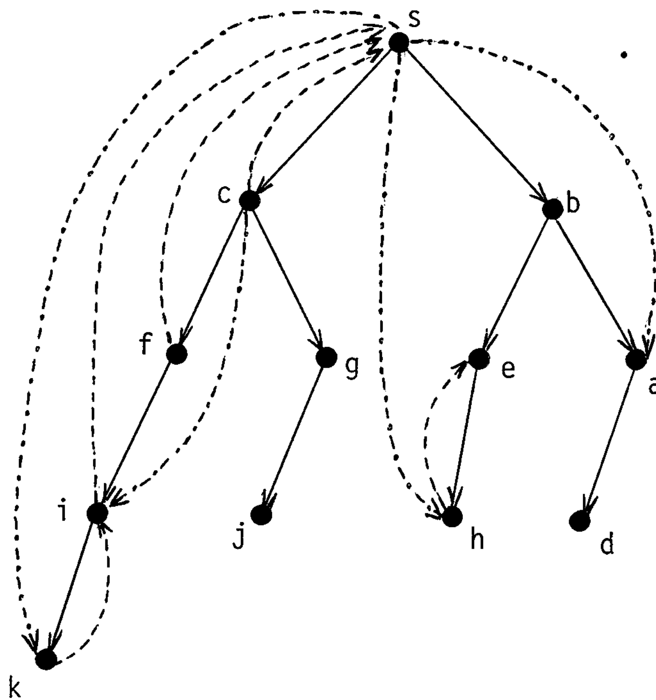
$(\underbrace{SDOM(k,h)}_{s}, k) = (s,k)$

4) (d,h)

$(\underbrace{SDOM(h,d)}_{a}, h) = (a,h)$ que ainda é aresta de cruzamento. Terá que aplicar novamente a 2ª transformação:

Assim: (SDOM(h,a), h) = (s,h)

Logo, serão retiradas as arestas de cruzamento: (g,i), (j,i), (h,k) e (d,h). Serão adicionadas as arestas de avanço: (c,i), (s,k) e (s,h).



<u>3ª Transformação</u> - "Retirada de arestas de avanço"

Arestas de avanço: (s,a), (s,h), (s,k) e (c,i)

Aplicando a 3ª transformação, somente a aresta de avanço (c,i) será retirada.

<u>4ª Transformação</u> - "Retirada de arestas de retorno"

Arestas de retorno: (c,s), (f,s), (i,s), (k,i) e (h,e)

1) (c,s) → retira (c,s)

2) (f,s) → retira (f,s)

3) (i,s) → retira (i,s)

4) (k,i) → tem a aresta de avanço (s,k)

   Retira (k,i) e adiciona a aresta de avanço (s,i) =

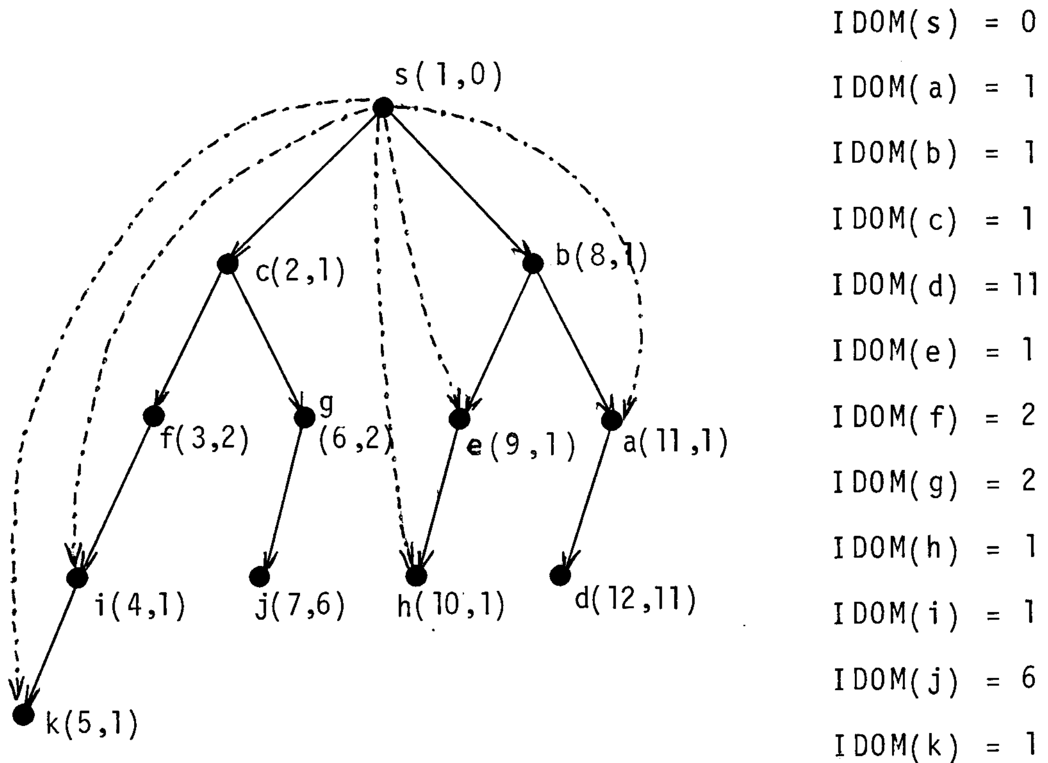
   = (u, HIGHPT(v))

5) (h,e) → tem a aresta de avanço (s,h)

Retira (h,e) e adiciona a aresta de avanço (s,e) =

= (u, HIGHPT(v))

Aplicando: 3ª e 4ª transformações:



Entre parenteses estão os valores dos números dos vértices (NUM(v)) e dos dominadores imediatos (IDOM(v)).

Logo, podemos concluir que: a árvore resultante da aplicação das quatro transformações conterá somente arestas de árvore e arestas de avanço, e fornecerá os dominadores imediatos de cada vértice do grafo direcionado D.

17) APLICAÇÃO - PLANARIDADE

17.1) REFERÊNCIA - HOPCROFT & TARJAN[41]

17.1.1) OUTRAS REFERÊNCIAS - REINGOLD, NIEVERGELT & DEO[60]; TARJAN[71].

17.2) DESCRIÇÃO DO PROBLEMA

Verificar se um grafo não direcionado é ou não planar.

Um grafo G=(V,E) é dito planar se existe um mapeamento de seus vértices e arestas em um plano tal que: a) cada vértice v está mapeado a um vértice distinto v' no plano, e b) cada aresta (v,w) está mapeada em uma curva simples tendo vértices terminais (v',w'), de modo que c) os cruzamentos entre arestas se encontrem somente em vértices terminais comuns. Em outras palavras, um grafo é planar se for possível desenhá-lo num plano onde duas arestas só se cruzam num vértice.

OBS.: a) Um digrafo é planar se e somente se seu grafo subjacente for planar; daí nós necessitamos apenas considerar grafos não direcionados.

b) Um grafo não direcionado é planar se e somente se todos os seus componentes conexos são planares; daí nós precisamos apenas considerar grafos não direcionados, conexos.

c) É fácil ver que um grafo não direcionado é pla-

nar se e somente seus componentes biconexos são planares. Então, se um grafo não direcionado não for biconexo, nós podemos decompo-lo em seus componentes biconexos e considerá-los individualmente.

Não é difícil desenhar o grafo completo de quatro vértices em um plano sem cruzar as arestas; consequentemente, um grafo não planar deve ter no mínimo cinco vértices. Similarmente, pode-se mostrar que um grafo não planar tem pelo menos nove arestas. Em geral, podemos usar a fórmula de Euler, relacionando o número de regiões, vértices e arestas de um grafo planar:

regiões + vértices = arestas + 2

para derivar o fato de que um grafo com $n > 2$ vértices e mais do que $3n - 6$ arestas nunca pode ser planar.

17.3) MÉTODO - BUSCA EM PROFUNDIDADE

O algoritmo da referência REINGOLD, NIEVERGELT & DEO[60], começa usando um algoritmo já visto (Aplicação 3 anterior), para decompor um grafo em seus componentes biconexos e determinar que tais componentes satisfazem $|V| \geq 5$ e $9 \leq |E| \leq 3|V|-6$. O algoritmo de planaridade deve então considerar somente estes grafos biconexos com $O(|V|)$ arestas. O algoritmo para determinar planaridade, neste caso, é bastante envolvente. Segue a sua descrição.

A estratégia básica é primeiro encontrar um ciclo C em G; imergir C no plano como uma curva fechada simples; decompor o grafo restante G-C em caminhos disjuntos em arestas, e então tentar imergir cada um desses caminhos inteiramente ou do lado de dentro de C ou do lado de fora de C. Se conseguirmos, com sucesso, imergir o grafo inteiro G, o grafo é planar; caso contrário ele é não planar. A dificuldade deste método é que, na imersão dos caminhos, nós podemos escolher ou o lado de dentro de C, ou o lado de fora de C. Devemos nos certificar que uma escolha inicial errada não irá evitar a imersão de um caminho mais tarde e, desse modo, forçando-nos a afirmar incorretamente que um grafo planar é não planar.

Será necessário gerar caminhos sistematicamente, para escolher áreas apropriadas para imergi-los, e, talvez, reorganizar os caminhos já imergidos, para acomodar novos caminhos. A fim de gerar os caminhos em uma certa ordem desejada, nós ordenaremos de novo os vértices e as listas de adjacência. Primeiramente nós executamos uma busca em profundidade no grafo dado

G (que é não direcionado e biconexo), obtendo uma Estrutura de Busca em Profundidade P (que consiste de $|V|-1$ arestas de árvore e $|E|-|V|+1$ arestas de retorno). De agora em diante nós iremos ignorar os rótulos originais dos vértices e identificaremos os vértices por seus valores NUM. Em seguida aplicaremos as funções: lowpt(v) e nexlopt(v) para cada vértice v do grafo. Essas funções são assim definidas: Seja lowpt(v) o vértice com menor numeração alcançável do vértice v ou de algum de seus descendentes (na árvore geradora de BP), por meio de no máximo uma aresta de retorno. Quando não é possível alcançar v por meio de uma simples aresta de retorno, v é o próprio lowpt(v). Similarmente, seja nexlopt(v) o próximo vértice com menor numeração abaixo de v, excluindo lowpt(v), que pode ser alcançado desta maneira. Se não existe tal vértice, nexlopt(v) será igual a v. Note que: como G é biconexo, pode-se garantir que $v \geq$ nexlopt(v) > lowpt(v), exceto quando v é a raiz, isto é, quando $v=1$. Quando v é a raiz, nós temos: lowpt(v) = nexlopt(v) = v = 1.

Direcionando as arestas de P, da raiz em direção às folhas, obtemos uma Estrutura Direcionada de Busca em Profundidade $\vec{P}$.

O próximo passo é ordenar de novo as listas de adjacência de $\vec{P}$ (Estrutura Direcionada de BP), de modo que, durante uma busca em profundidade em $\vec{P}$, os caminhos em $\vec{P}$ serão gerados em uma certa ordem desejada. Com esse propósito, será computada uma função ϕ com valor inteiro para cada aresta (v,w) em $\vec{P}$:

$$\phi[(v,w)] = \begin{cases} 2w & \text{se } (v,w) \text{ é uma aresta de retorno.} \\ 2\,\text{lowpt}(w) & \text{se } (v,w) \text{ é uma aresta de árvore e nexlopt}(w) \geq v. \\ 2\,\text{lowpt}(w)+1 & \text{se } (v,w) \text{ é uma aresta de árvore e nexlopt}(w) < v. \end{cases}$$

Assim, para cada vértice v, nós ordenaremos todas as arestas (v,w) em ordem crescente de acordo com os valores de ϕ e usamos esta ordem nas listas de adjacência. As novas listas de adjacência para $\vec{P}$ serão chamadas Padj(v), <u>listas de adjacência propriamente ordenadas</u>.

Tendo obtido as listas de adjacência propriamente ordenadas representando a Estrutura $\vec{P}$, nós agora aplicamos uma <u>busca em profundidade</u> para decompor $\vec{P}$ em um ciclo C e um número de caminhos de arestas disjuntas p_i. A ref. REINGOLD, NIEVERGELT & DEO[60] apresenta um algoritmo (procedure PATH) que decompõem um digrafo representado por listas de adjacência propriamente ordenadas em um ciclo C = p_0 e caminho $p_1, p_2, \ldots$.

Um <u>segmento</u> (de $\vec{P}$ com respeito ao ciclo C) é ou a) uma simples aresta de retorno (v_i, w) não em C, mas onde $v_i, w \in C$, ou b) um subgrafo consistindo de uma aresta de árvore (v_i, w), $v_i \in C$, $w \notin C$, e a subárvore direcionada enraizada em w, junto com todas as arestas de retorno desta subárvore. O vértice v_i em C, em que um segmento é originado, é chamado <u>vértice base</u> do segmento.

Claramente, todos os caminhos pertencentes a um segmento devem ser imergidos juntos ou todos no lado de dentro de C ou todos no lado de fora de C. Esta é a razão de agrupar os

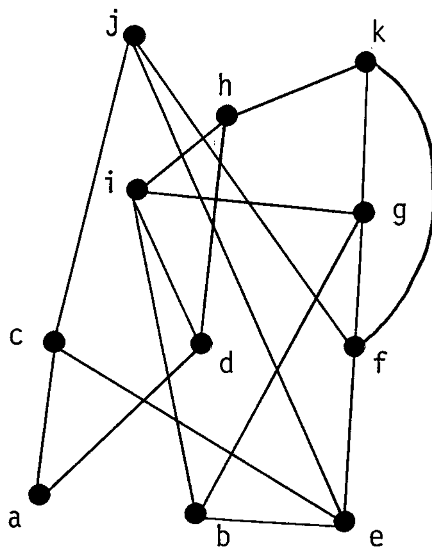
caminhos em segmentos.

Imersão: o ciclo C é imergido no plano como uma simples curva fechada e depois os segmentos serão imergidos. Para imergir um segmento S, nós consideramos o primeiro caminho p em S e escolhemos um lado de C, por exemplo o lado de dentro, em que imergimos p. Examinando os caminhos imergidos previamente, nós determinamos se p pode ser imergido no lado de dentro de C. Se p pode ser imergido, nós o imergimos; caso contrário todos aqueles segmentos previamente imergidos no lado de dentro de C, que estão blocando a imersão de p, são movidos para o lado de fora de C. Trazendo estes segmentos para o lado de fora, podemos forçar alguns outros segmentos do lado de fora para o lado de dentro e assim por diante. Se p ainda não puder ser imergido no lado de dentro de C, mesmo depois desta reorganização dos segmentos, então G é não-planar. Se p pode ser imergido no lado de dentro de C, nós o imergimos e então tentamos imergir o resto do segmento S aplicando o algoritmo de imersão recursivamente. Se conseguirmos obter sucesso, continuamos a imersão com o próximo segmento.

Complexidade: O algoritmo inteiro para testar se um grafo é planar requer somente $O(|V| + |E|)$ operações (Ver REINGOLD, NIEVERGELT & DEO[60]). Contudo, como $|E| \leq 3|V| - 6$ (caso contrário o grafo não deve ser planar), o algoritmo requer $O(|V|)$ operações.

17.4) EXEMPLO

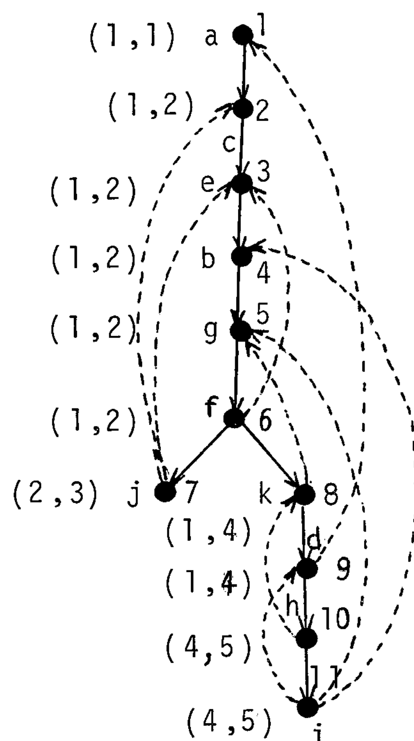
Verificar se o grafo G = (V,E) abaixo é planar:



Estrutura de Adjacência:

a : c, d
b : e, g, i
c : a, e, j
d : a, h, i, k
e : b, c, f, j
f : e, g, j, k
g : b, f, i, k
h : d, i, k
i : b, d, g, h
j : c, e, f
k : d, f, g, h

Aplicando a técnica de busca em profundidade no grafo G acima, obtemos uma Estrutura de Busca em Profundidade P. Direcionando P, obtemos a Estrutura Direcionada de Busca em Profundidade $\vec{P}$ abaixo:



- Os números nos vértices em $\vec{P}$ são os valores de NUM dos vértices.
- Os valores de lowpt(v) e nexlopt(v) para os vértices do grafo G são dados entre parenteses ao lado de NUM : (lowpt(v), nexlopt(v)).

Valores da função ϕ para as arestas da Estrutura $\vec{P}$:

	Aresta (v,w)	$\phi[(v,w)]$
Arestas de Árvore	(1,2)	2
	(2,3)	2
	(3,4)	3
	(4,5)	3
	(5,6)	3
	(6,7)	5
	(6,8)	3
	(8,9)	3
	(9,10)	9
	(10,11)	9
Arestas de Retorno	(9,1)	2
	(7.2)	4
	(6,3)	6
	(7,3)	6
	(11,4)	8
	(8,5)	10
	(11,5)	10
	(10,8)	16
	(11,9)	18

Agora obtemos as listas de adjacência propriamente ordenadas (Padj(v)) da Estrutura $\vec{P}$.

Padj(v)

1 : 2

2 : 3

3 : 4

4 : 5

5 : 6

6 : 8, 7, 3

7 : 2, 3

8 : 9, 5

9 : 1, 10

10 : 11, 8

11 : 4, 5, 9

Então decompomos a estrutura $\vec{P}$ em um ciclo p_0 e caminhos p_1, p_2, p_3,...,p_8, (aplicando a técnica de busca em profundidade, utilizando Padj(v)).

p_0 = C = (1, 2, 3, 4, 5, 6, 8, 9, 1)

p_1 = (9, 10, 11, 4)

p_2 = (11, 5)

p_3 = (11, 9)

p_4 = (10, 8)

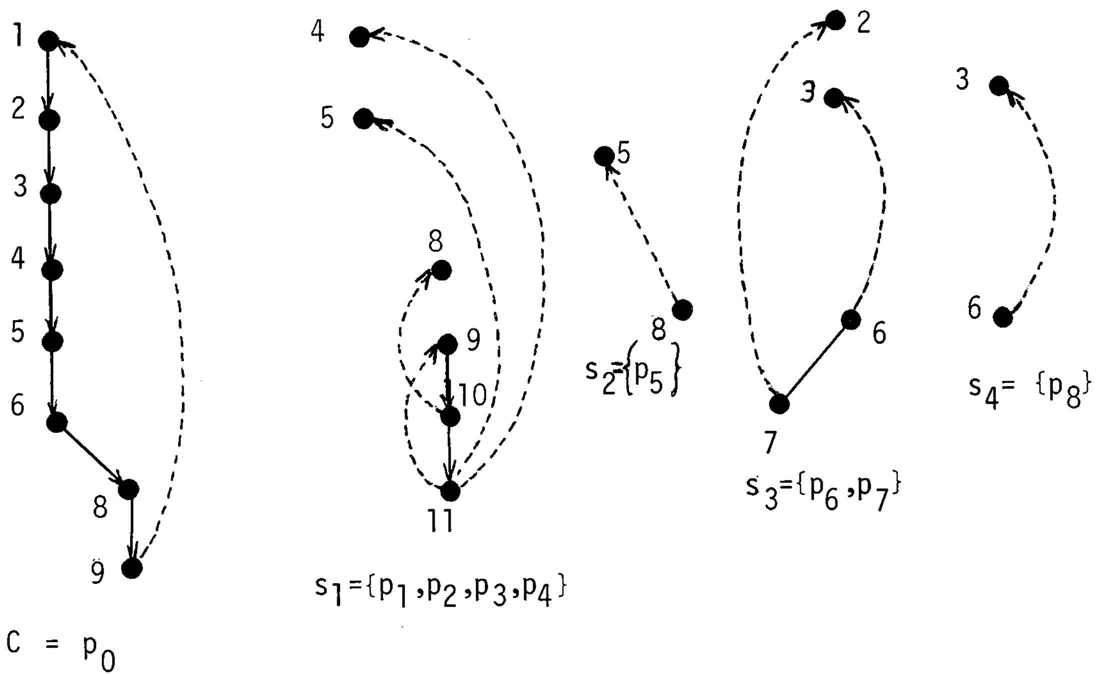
p_5 = (8, 5)

p_6 = (6, 7, 2)

p_7 = (7, 3)

p_8 = (6, 3)

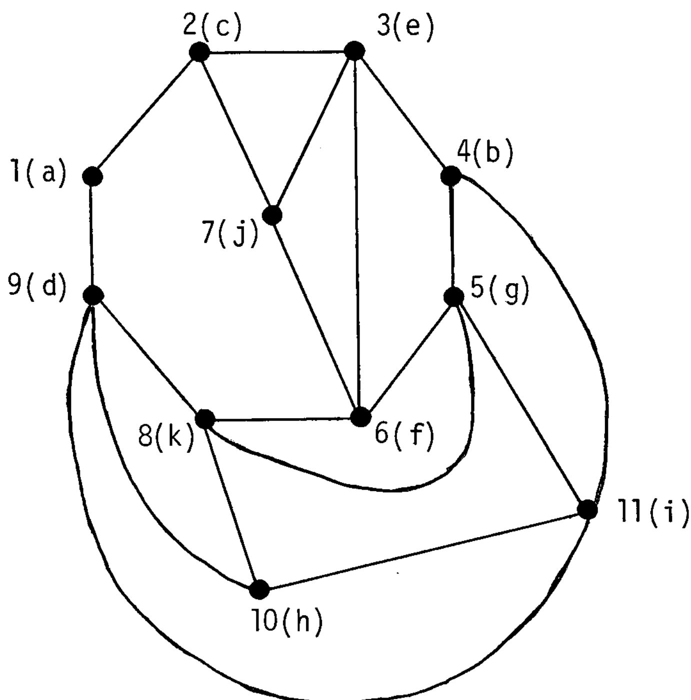
<u>Segmentos</u> de $\vec{P}$ com respeito ao ciclo C:



<u>Imersão</u>: no plano do ciclo C e dos segmentos acima:

Segmentos imergidos no lado de dentro de C = S_3, S_4

Segmentos imergidos no lado de fora de C = S_1, S_2



Conclusão:
O grafo G é planar.

18) APLICAÇÃO - REPRESENTAÇÃO VETORIAL DA ALCANÇABILIDADE EM GRAFOS DIRECIONADOS PLANARES

18.1) REFERÊNCIA - KAMEDA[43]

18.1.1) OUTRAS REFERÊNCIAS - TARJAN[71]; HARARY[35]; BAASE[3].

18.2) DESCRIÇÃO DO PROBLEMA

Representar, por meio de vetores, a alcançabilidade de um dado grafo direcionado planar acíclico.

Seja D = (V,E) um grafo direcionado planar acíclico, onde V é o conjunto de vértives e E o conjunto de arestas.

Notação: $v < u$ significa que v alcança u.

O objetivo do problema descrito acima é rotular os vértices por vetores de tal maneira que $u < v$ se e somente se $L(u) < L(v)$, onde $L(u)$ é um rótulo do vetor d-dimensional determinado para o vértice u. Para dois vetores $(a_1,a_2,\ldots,a_d)$ e $(b_1,b_2,\ldots,b_d)$, definimos $(a_1,a_2,\ldots,a_d) < (b_1,b_2,\ldots,b_d)$ se e somente se $a_i \leq b_i$ para todo i, $1 \leq i \leq d$, e existe j, $1 \leq j \leq d$, tal que $a_j < b_j$.

Um grafo é dito ter a propriedade P se e somente se ele é um grafo direcionado acíclico planar, tal que todos os vértices com grau de entrada igual a zero e todos os vértices com grau de saída igual a zero estão na fronteira da mesma face, e tal que a fronteira pode ser dividida em duas seções contiguas contendo os vértices do primeiro tipo (grau de entrada

= 0) e vértices do último tipo (grau de saída = 0), respectivamente. Sem perda de generalidade, podemos assumir que esta será a face exterior (Ver HARARY[35]) que contém o infinito.

OBS.: Todo grafo planar pode ser imergido num plano de tal forma que uma certa face seja a exterior.

18.3) MÉTODO - BUSCA EM PROFUNDIDADE

Aqui será descrito um algoritmo para resolver o problema acima, que rotula os vértices de qualquer grafo com propriedade P, tal que:

a) Os rótulos usados são de vetores 2-dimensional (ou 2-vetores);

b) Cada componente de vetores varia entre 1 e $n = |V|$, inclusive;

c) O algoritmo trabalha em tempo igual a: $k_1 n + k_2$, onde k_1 e k_2 são constantes (Ver KAMEDA[43]).

É dado um grafo com propriedade P. Acrescenta-se a este grafo um vértice fonte s e um vértice sumidouro t. Então qualquer vértice será alcançável de s, e todo vértice alcançará t .

A Busca em Profundidade à Esquerda é definida ser a mesma Busca em Profundidade que já definimos anteriormente (Ver TARJAN[71]). Como já vimos, na execução de uma Busca em Profundidade é conveniente usar um pilha (Ver BAASE[3]). Um vértice é empilhado quando ele é alcançado pela 1ª vez, e desempilhado quando todos os vértices adjacentes a ele já foram

examinados.

A Busca em Profundidade a Direita é semelhante a Busca em Profundidade a Esquerda, exceto que a lista de adjacência é examinada da direita para a esquerda, fazendo uso de uma estrutura de lista duplamente ligada.

Agora será apresentado um algoritmo para computar os primeiros e segundos componentes dos rótulos do vetor (2-dimensional).

Algoritmo

Dado um digrafo D com a propriedade P, seja D' o grafo obtido de D introduzindo o vértice fonte s e o vértice sumidouro t. Seja n = |V|.

1. Executar a Busca em Profundidade a Esquerda em D', com o valor inicial de i = n+1. Quando um vértice é empilhado obtém-se i como o primeiro componente ($L_1(v)$) de seu rótulo vetor e i é decrementado de 1. A árvore geradora assim criada é chamada Árvore Geradora a Esquerda.

2. Os segundos componentes ($L_2(v)$) dos rótulos vetor são similarmente computados usando Busca em Profundidade a Direita e a árvore geradora resultante é chamada Árvore Geradora a Direita.

Note que se s e t são agora retirados, então temos os componentes variando entre 1 e n.

Resultado

Seja $L(v) = (L_1(v), L_2(v))$ o rótulo de um vetor 2-dimensional para $v \in V$ computado como descrito anteriormente. Então para dois vértices distintos u e v: $u < v$ se e somente se $L(u) < L(v)$. (Ver prova em KAMEDA[43]).

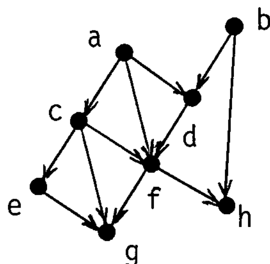
Análise do Algoritmo

A Busca em Profundidade, em geral, requer tempo proporcional à soma do número de vértices e o número de arestas. Contudo, em um grafo planar, o número de arestas é limitado por uma função linear do número de vértices (ver aplicação anterior. Assim o algoritmo de rotulação aqui descrito trabalha em tempo proporcional a n, ou seja, a complexidade é $O(|V|)$.

OBS.: Tentamos aqui encontrar uma condição necessária e suficiente para a alcançabilidade de um grafo ser representado por vetores 2-dimensionais. O problema de encontrar o d mínimo, tal que rótulos de vetor d-dimensional possam ser usados para representar a alcançabilidade de um dado grafo, é ainda um problema em aberto.

18.4) EXEMPLO

Fazer a representação vetorial da alcançabilidade do grafo direcionado acíclico planar D abaixo:



Estrutura de Adjacência de D':

s → a, b

a → c, f, d

b → d, h

c → e, g, f

d → f

e → g

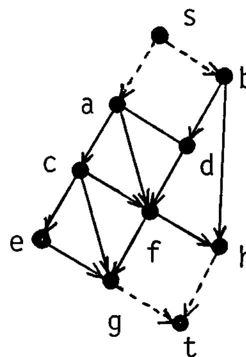
f → g, h

g → t

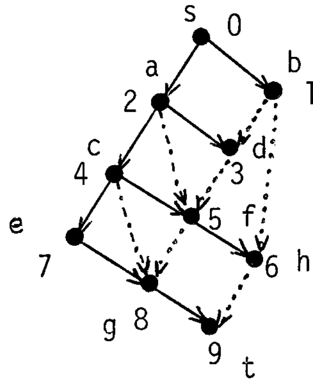
h → t

t → -

O grafo D' é obtido de D introduzindo o vértice fonte s e o vértice sumidouro t.

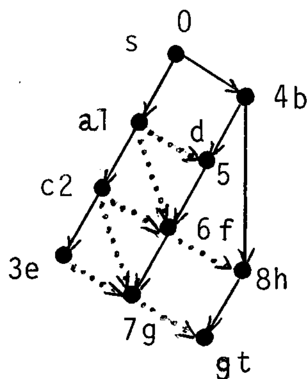


Aplicando a Busca em Profundidade a Esquerda em D', e rotulando os vértices à medida que eles são desempilhados, obtemos os primeiros componentes dos rótulos do vetor (ou seja $L_1(v)$).



Ãrvore Geradora a Esquerda

Similarmente, aplicamos a Busca em Profundidade a Direita em D', obtemos a Ãrvore Geradora a Direita rotulada com os <u>segundos componentes</u> dos rõtulos do vetor (ou seja $L_2(v)$).



Logo,

$L_1(a) = 2$	$L_2(a) = 1$
$L_1(b) = 1$	$L_2(b) = 4$
$L_1(c) = 4$	$L_2(c) = 2$
$L_1(d) = 3$	$L_2(d) = 5$
$L_1(e) = 7$	$L_2(e) = 3$
$L_1(f) = 5$	$L_2(f) = 6$
$L_1(g) = 8$	$L_2(g) = 7$
$L_1(h) = 6$	$L_2(h) = 8$

Aplicando o Resultado, por exemplo, temos:

$u \prec v$ sss $L(u) < L(v)$

$u \prec v$ sss $(L_1(u), L_2(u)) < (L_1(v), L_2(v))$

1) $a \prec g$ sss $(L_1(a),\ L_2(a)) < (L_1(g),\ L_2(g))$

$a \prec g$ sss $(2,\ 1) < (8,\ 7)$ é Verd pois $2 < 8$ e $1 < 7$

2) $c \prec h$ sss $(L_1(c),\ L_2(c)) < (L_1(h),\ L_2(h))$

$c \prec h$ sss $(4,\ 2) < (6,\ 8)$ é Verd pois $4 < 6$ e $2 < 8$

19) APLICAÇÃO - ISOMORFISMO EM GRAFOS

19.1) REFERÊNCIA - BERZTISS[8]

19.1.1) OUTRAS REFERÊNCIAS - DEO[15]; AHO[1]; REINGOLD, NIEVERGELT & DEO[60]; READ & CORNEIL[58].

19.2) DESCRIÇÃO DO PROBLEMA

Determinar se dois grafos direcionados são isomorfos.

Dois grafos $D_A = (V_A,\ E_A)$ e $D_B = (V_B,\ E_B)$ são isomorfos, escritos $D_A \cong D_B$, se existe uma correpondência um a um, sobrejetiva $f : V_A \rightarrow V_B$ tal que $(v,w) \in E_A$ se e somente se $(f(v),\ f(w)) \in E_B$, isto é, se existe uma correpondência entre os vértices de D_A e os vértices de D_B, que preserva as relações de adjacência.

Grafos isomorfos diferem somente na rotulação dos vértices. Então o problema de determinar isomorfismo ocorre em um número de situações práticas, tais como recuperação de

informação e identificação de componentes químicos.

OBSERVAÇÃO

1) Qualquer grafo não direcionado pode ser transformado em um grafo direcionado, substituindo cada aresta (do grafo não direcionado) por duas arestas direcionadas opostas. Dois digrafos assim obtidos são isomorfos se e somente se os grafos originais são isomorfos.

2) O problema de isomorfismo em grafos é conhecido estar em NP, mas ainda não foi possível mostrar que ele pertence à classe dos problemas de NP-completo, nem foi possível encontrar um algoritmo geral de isomorfismo com um limite de tempo polinomial (ou melhor, não é conhecido que ele seja NP-completo, nem é conhecido que ele esteja em P).

19.3) MÉTODO - BUSCA EM PROFUNDIDADE

Suponha que são dados dois digrafos: $D_x(V_x, E_x)$ e $D_y = (V_y, E_y)$, e queremos testar se existe isomorfismo entre eles, ou seja $D_x \cong D_y$. Assumiremos $V_x = V_y = \{1,2,...,n\}$; se $|V_x| \neq |V_y|$, os digrafos não são isomorfos. Seja D_x o digrafo selecionado como digrafo de referência. Seja $D_x(k)$ um subgrafo de D_x induzido pelos vértices $\{1,2,..., k\}$, $0 \leq k \leq n$. Claramente, $D_x(0)$ é o subgrafo vazio e $D_x(1)$ é o subgrafo consistindo de um vértice simples 1 e sem arestas.

O método mais direto para determinar isomorfismo de

grafos é usar backtrack para examinar todas as n! correspondências possíveis entre os vértices de dois digrafos de n vértices. Quando n é grande este backtrack não restringido não é bom por causa do tamanho de n! Contudo, com um corte adequado da árvore de busca, este método forma a base de um algoritmo razoável, em certos casos, para isomorfismo de digrafos (Ver REINGOLD, NIEVERGELT & DEO[60] e DEO[15]).

Assumimos que D_x e D_y são dados como estruturas de adjacência Xadj e Yadj, respectivamente. Seja Xdeg(i) e Ydeg(i) os números compostos por i-ésimos vértices em D_x e D_y, respectivamente. Note que Xdeg(i) = Ydeg(i) se e somente se o vértice i em D_x e o vértice j em D_y tem o mesmo grau de entrada e de saída (OBS.: Note que os conjuntos mútiplos $\{Xdeg(i) | 1 \leq i \leq n\}$ e $\{Ydeg(i) | 1 \leq i \leq n\}$ devem ser iguais se $D_x \cong D_y$).

O primeiro passo no algoritmo para achar isomorfismo em digrafos, é aplicar a técnica de busca em profundidade em um dos digrafos (supondo que seja D_x). Ao invés de começar a busca em profundidade em D_x de algum vértice arbitrário como raiz, nós selecionamos o vértice inicial em D_x, como sendo o que pertence ao menor subconjunto de vértices tendo número Xdeg idênticos (ou seja, começando em um vértice $x \in V_x$ para os quais existem poucos $v \in V_x$, tal que Xdeg(x) = Xdeg(v)), a fim de manter a árvore de backtrack tão reduzida no topo quanto possível. Começando desta raiz, nós executamos uma busca em profundidade em D_x, e obtemos deste processo inteiros distintos de 1 a n, (valor NUM), para cada um dos vértices em D_x. De agora em diante somente os valores NUM são usados como nomes dos vértices de D_x e os rótulos antigos são abandonados.

Como as arestas são encontradas durante a busca em profundidade em D_x, nós as arranjaremos em uma tabela, indicando o número do vértice do qual a aresta particular origina ($From_i$) e o vértice em que a aresta termina (To_i). Em adição às arestas originalmente em D_x, esta tabela também conterá arestas "imaginárias" indo para as raízes das árvores na floresta de busca em profundidade de um vértice imaginário zero. As arestas são, então, ordenadas de modo que todas as arestas pertencentes ao subgrafo $D_x(k)$ aparecem antes das arestas incidentes em vértices $i > k$.

Depois de executar a busca em profundidade em D_x, é usada uma estratégia backtracking com alguns cortes, para comparar os dois digrafos, ou seja, determinar se $D_x \cong D_y$. Evidentemente, $D_x(0)$ é isomorfo ao subgrafo vazio de D_y. Suponha que, em um determinado momento, nós encontramos um subgrafo de D_y consistindo de vértices $s \subseteq V_y$ que é isomorfo a $D_x(k)$. Nós tentamos estender o isomorfismo para $D_x(k+1)$ escolhendo um vértice $v \in V_y - s$ para corresponder a $k + 1 \in V_x$. Se encontramos um tal v, nós registramos a correspondência fazendo $f_{k+1} \leftarrow v$ e tentamos estender o isomorfismo para $D_x(k+2)$. Se não existe um tal v, nós voltamos a $D_x(k - 1)$ e tentamos escolher um vértice diferente para corresponder a $k \in V_x$. Este processo continua até que ou encontramos um isomorfismo entre $D_x(n) = D_x$ e D_y ou voltamos a $D_x(0)$ e concluimos que $D_x \ncong D_y$ (Tal processo é mostrado no algoritmo 8.13 da ref. REINGOLD, NIEVERGELT & DEO[60]).

O valor esperado da complexidade do algoritmo é:

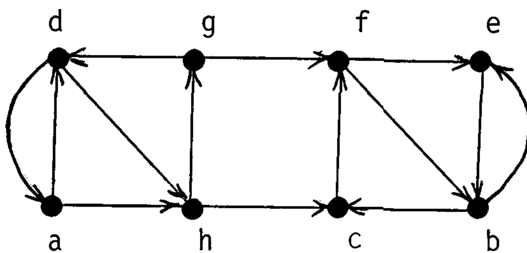
$$O(n^{\log n}) \quad \text{(ver DEO[15])} .$$

A ref. READ & CORNEIL[58] sugere que quando os grafos não são isomorfos, é mais vantagem utilizar a técnica de busca em largura, pois pode-se descobrir com mais antecedência que os grafos possuem partições incompatíveis. Ao passo que, se os grafos são isomorfos, é melhor utilizar a técnica de busca em profundidade.

19.4) EXEMPLO

Verificar se D_x e D_y são isomorfos.

Considere os seguintes digrafos D_x e Dy e suas estruturas de adjacência.



D_x

Estrutura Adj. Xadj(v)

a → d, h

b → c, e

c → f

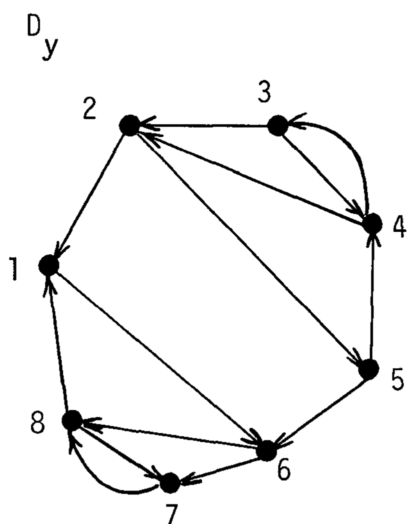
d → a, h

e → b

f → b, e

g → d, f

h → c, g



Estrutura de Adj. Yadj(v)

1 → 6

2 → 1, 5

3 → 2, 4

4 → 2, 3

5 → 4, 6

6 → 7, 8

7 → 8

8 → 1, 7

Os graus de entrada, de saída e os graus compostos dos vértices são:

D_x:

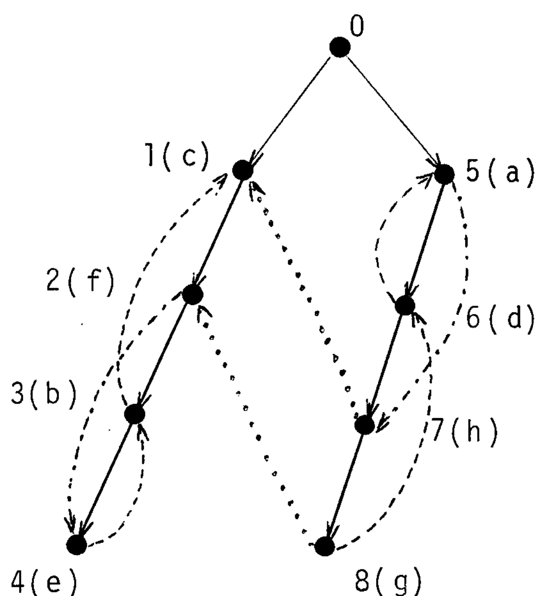
Vértice	a	b	c	d	e	f	g	h
Grau de saída	2	2	1	2	1	2	2	2
Grau de entrada	1	2	2	2	2	2	1	2
Xdeg	21	22	12	22	12	22	21	22

D_y:

Vértice	1	2	3	4	5	6	7	8
Grau de saída	1	2	2	2	2	2	1	2
Grau de entrada	2	2	1	2	1	2	2	2
Ydeg	12	22	21	22	21	22	12	22

Como os conjuntos mútiplos consistindo de graus compostos são os mesmos, os digrafos podem ser isomorfos.
Fazemos, então, uma busca em profundidade em D_x, e rotulamos os vértices novamente. Existem dois vértices com Xdeg = 12, dois com Xdeg = 21, e quatro vértices com

Xdeg = 22. Então, podemos começar a busca em profundidade com um dos seguintes vértices: c, e, a ou g; (fazendo isto, teremos como resultado poucas correspondências entre $D_x(1)$ e subgrafos de D_y). Arbitrariamente escolhemos c, resultando a seguinte estrutura direcionada de busca em profundidade, com os rótulos antigos entre parênteses:



A figura acima mostra arestas de árvore (———>), arestas de retorno (---->), arestas de cruzamento (····>) e arestas de avanço (-.-.->). As duas arestas que ligam as raízes de árvores na floresta de busca em profundidade são arestas imaginárias ((0,1) e (0,5)).

À medida que as arestas são encontradas durante a busca em profundidade, elas são adicionadas a tabela from e to. A i-ésima aresta encontrada vai do vértice $From_i$ para o vértice to_i, $0 \leq i \leq n$. Por exemplo, no exemplo anterior

temos:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$From_i$	0	1	2	3	3	4	2	0	5	6	6	7	7	8	8	5
to_i	1	2	3	1	4	3	4	5	6	5	7	1	8	6	2	7

Estas arestas são então ordenadas em ordem crescente para $graph_i$, onde $graph_i = \max (From_i, to_i)$ é o menor k tal que a aresta $(From_i, to_i)$ está em $D_x(k)$. Temos então:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$From_i$	0	1	2	3	3	4	2	0	5	6	6	5	7	7	8	8
to_i	1	2	3	1	4	3	4	5	6	5	7	7	1	8	6	2
$Graph_i$	1	2	3	3	4	4	4	5	6	6	7	7	7	8	8	8

Agora já podemos utilizar o algoritmo da Ref. REINGOLD, NIEVERGELT & DEO[60], que determina se o vértice $v \in V_y$ pode corresponder ao vértice $k \in V_x$. Se não encontramos uma cópia isomorfa de $D_x(k-1)$ em D_y, sabemos que, para todo i, $j < k$, (i,j) é uma aresta de $D_x(k-1)$ se e somente se (f_i, f_j) é uma aresta de D_y. O vértice v em D_y pode corresponder ao vértice k em D_x, estabelecido que (i,k) é uma aresta de $D_x(k)$ se e somente se (f_i, v) é uma aresta de D_y, e similarmente, (k,i) é uma aresta de $D_x(k)$ se e somente se (v,f_i) é uma aresta de D_y. Isto é fácil de verificar, desde que nós temos os arrays From, to, graph.

Assim, aplicando o algoritmo da ref. REINGOLD, NIEVERGELT & DEO[60] ou DEO[15] saberemos que os dois digrafos do exemplo dado são isomorfos, ou seja: $D_x \cong D_y$, e temos as seguintes correspondências:

D_x		D_y
a	$\leftrightarrow$	3
b	$\leftrightarrow$	8
c	$\leftrightarrow$	1
d	$\leftrightarrow$	4
e	$\leftrightarrow$	7
f	$\leftrightarrow$	6
g	$\leftrightarrow$	5
h	$\leftrightarrow$	2

20) APLICAÇÃO - DIGRAFOS ESTRUTURADOS EM ÁRVORE

20.1) REFERÊNCIA - SZWARCFITER[69]

20.1.1) OUTRAS REFERÊNCIAS - HOPCROFT E TARJAN[40]; DILWORTH[19]; GAREY & JOHNSON[30].

20.2) DESCRIÇÃO DO PROBLEMA

Resolver os seguintes problemas para a classe de Digrafos Estruturados em Árvore:

1) Reconhecimento

2) Redução e Fechamento Transitivo

3) Isomorfismo

4) Decomposição Mínima por Cadeia

5) Dimensão do poset (conjunto parcialmente ordenado) induzido.

Um digrafo acíclico é <u>estruturado em árvore</u> quando sua redução transitiva é uma árvore enraizada direcionada.

O fechamento transitivo de D é também chamado conjunto parcialmente ordenado (poset) induzido por D. Se v alcança w, $v \neq w$, diremos que v e w são <u>comparáveis</u>. Se v não alcança w, nem w alcança v então v e w são <u>incomparáveis</u>.

20.3) MÉTODO - BUSCA EM PROFUNDIDADE

1) RECONHECIMENTO DE DIGRAFOS ESTRUTURADOS EM ÁRVORE

O reconhecimento de digrafos estruturados em árvore pode ser feito da seguinte maneira:

Dado um digrafo $D = (V,E)$ (com $|V|$ vértices e $|E|$ arestas) enraizado no vértice r, primeiramente ordenamos topologicamente suas listas de adjacência. Depois, começando do vértice r, executamos uma <u>busca em profundidade</u> em D. O digrafo D irá pertencer à classe dos digrafos estruturados em árvore se a estrutura de busca em profundidade não tiver aresta de cruzamento.

Para decidir se uma dada aresta é ou não uma aresta de cruzamento, dentro de uma busca em profundidade, podemos gastar um tempo constante. Portanto, é imediato verificar que o processo inteiro descrito acima é completado em $O(|V| + |E|)$ tempo.

(Ver SZWARCFITER[69]).

2) REDUÇÃO E FECHAMENTO TRANSITIVO

Aqui o problema consiste em obter a redução e o fechamento transitivo de um digrafo estruturado em árvore: $D = (V, E)$.

Podemos facilmente verificar que se D é de fato um digrafo estruturado em árvore, então a estrutura de busca em profundidade sem arestas de cruzamento obtida de D é precisamente a redução transitiva de D. Isto significa que este problema pode ser também resolvido em $O(|V| + |E|)$ tempo.

Para encontrar o fechamento transitivo de D, nós precisamos observar que, para cada vértice $v \in V$, a lista de adjacência A(v) do fechamento transitivo de D é formada exatamente pelo conjunto de descendentes de v, na redução transitiva (árvore enraizada direcionada) de D. Consequentemente, o fechamento transitivo pode ser obtido em tempo proporcional ao seu tamanho.

3) ISOMORFISMO DE DIGRAFOS ESTRUTURADOS EM ÁRVORE

Dado dois digrafos estruturados em árvore: $D_1 = (V_1, E_1)$ e $D_2 = (V_2, E_2)$, queremos saber se eles são isomorfos.

Primeiramente obtemos suas árvores de redução transitiva T_1 e T_2, respectivamente.

Para i = 1,2,.., n, seja $h(w_i)$ o nível do vértice w_i na árvore T_i. Suponha que um conjunto de rótulos é determinado para cada vértice w_i de T_i, tal que:

Se w_i é a raiz de T_i então $s(w_i) = \phi$, caso contrário $s(w_i) = \{h(v_i) | (v_i, w_i) \in E_i\}$.

Para cada $w_1 \in V_1$ nós encontramos o conjunto $s(w_1)$ como já foi descrito. Similarmente, encontramos os conjuntos de rótulos para todo w_2 em T_2. Observe que cada conjunto de rótulos $s(w_i)$ é composto por elementos $h(v_i)$ (níveis), dentro do limite $1 < h(v_i) < n$.

Aplicamos, por exemplo, um algoritmo feito por Hopcroft e Tarjan (Ref. HOPCROFT & TARJAN[40]) para isomorfismo de árvore rotulada e decidimos se T_1 e T_2 são árvores enraizadas rotuladas isomorfas.

Assim temos por lema já provado em SZWARCFITER[69] que D_1 e D_2 são isomorfos se e somente se T_1 e T_2 são árvores enraizadas rotuladas isomorfas.

O tempo de execução do presente problema será fácil de ser obtido: já sabemos que T_1 e T_2 podem ser construidas em $O(|V| + |E|)$; a computação de todos os conjuntos de rótulos requer também $O(|V| + |E|)$ operações. O algoritmo de Hopcroft e Tarjan para isomorfismo de árvores rotuladas é limitado por $O(|V| + R)$, onde $|V|$ é o número de vértices de cada árvore e R é a soma dos comprimentos de todos os conjuntos de rótulos. Claramente, $L = |E|$ para cada árvore. Portanto, o processo inteiro é limitado por $O(|V| + |E|)$.

4) DECOMPOSIÇÃO MÍNIMA POR CADEIAS DE DIGRAFOS ESTRUTURADOS EM ÁRVORE

Seja D = (V,E) um digrafo acíclico.

OBSERVAÇÃO

a) Uma cadeia de D é uma sequência de vértices que é um caminho no fechamento transitivo de D.

b) O problema de decomposição mínima por cadeias consiste em encontrar um conjunto mínimo de cadeias que cubra os vértices de D.

c) Teorema de Dilworth: O número mínimo de cadeias que cobrem D é igual ao número máximo de vértices incomparáveis de D.

Seja D = (V,E) um digrafo estruturado em árvore. Para encontrar um conjunto de decomposição mínima por cadeia $\{c_1, c_2, \ldots, c_k\}$ de D, determinamos, inicialmente, a árvore de redução transitiva T de D. A primeira cadeia c_1 no conjunto é um caminho da raiz de T para algum de seus vértices sumidouros (grausaída (v) = 0). Depois retiramos de T os vértices de c_1. Desta maneira transformamos T - c_1 em uma floresta. Escolhemos alguma árvore T' de T - c_1 e a cadeia c_2 é um caminho da raiz de T' para algum de seus vértices sumidouros. Retiramos c_2 de T - c_1 e aplicamos a mesma operação para (T - c_1) - c_2. Procedemos assim até que a floresta se torne vazia. Como a retirada destas cadeias não pode criar novos sumidouros, o número total de ca

deias obtidas pelo processo aqui descrito é igual ao número de vértices sumidouros de D, o que significa que $\{c_1, \ldots c_k\}$ é de fato mínimo.

É também imediato verificar que o conjunto de cadeias pode ser obtido de T em $O(|V|)$ tempo e assim o processo inteiro é limitado por $O(|V| + |E|)$.

OBSERVAÇÃO

As cadeias obtidas pelo método acima são de fato caminhos de D. Isto significa que para digrafos estruturados em árvore D, o problema de obter um conjunto mínimo de cadeias que cubra os vértices de D é equivalente ao problema de encontrar um conjunto mínimo de caminhos disjuntos, cujos vértices cubram D. Estes dois problemas são claramente os mesmos quando D é seu próprio fechamento transitivo, mas, caso contrário, podem ter diferentes soluções.

5) DIMENSÃO DO POSET INDUZIDO

OBSERVAÇÃO

a) Seja D = (V,E) um digrafo acíclico. Podemos caracterizar conjunto parcialmente ordenado (poset) induzido de D, (isto é, o fechamento transitivo de D) através de um conjunto de k ordenações topológicas de D, tal que v alcança w se e somente se v precede w em todas as k ordenações, para todo v, $w \in V$. Em outras palavras, v e w são incomparáveis em D se e

somente se existem duas ordenações t_1, t_2, tal que v precede w em t_1 e w precede v em t_2.

b) O valor mínimo de k é a <u>dimensão do poset</u>. Existe um algoritmo polinomial para verificar se a dimensão de um poset é ≤ 2 (Ver GAREY & JOHNSON[30]). Contudo, é NP-completo o problema de determinar a dimensão k de um poset, quando $k \geq 3$.

c) Seja T uma árvore enraizada. Em uma busca em pré-ordem de T, nós visitamos a raiz de T e depois, recursivamente visitamos os filhos de T. Se os filhos são visitados na ordem da esquerda para a direita, nós a chamaremos de <u>pré-ordem esquerda</u>; se as visitas são da direita para a esquerda, nós temos uma <u>pré-ordem direita</u>.

Aqui queremos considerar o problema de encontrar um conjunto mínimo de k ordenações topológicas que define um poset induzido por um digrafo estruturado em árvore D' = (V,E), cujo valor mínimo de k é dito a dimensão do poset.

Seja T' a árvore de redução transitiva do digrafo estruturado em árvore D'. Se p_1, p_2 são as buscas de pré-ordem esquerda e direita de T' respectivamente, então $\{p_1, p_2\}$ é um conjunto de ordenações topológicas que completamente caracteriza o poset induzido por D'. Uma consequência deste fato é que um poset especificado por um digrafo estruturado em árvore é necessariamente bi-dimensional (exceto, naturalmente, o caso em que $p_1 = p_2$ que corresponde ao poset sendo uma ordenação total).

Logo, as ordenações topológicas $\{p_1, p_2\}$ podem ser obtidas em tempo linear.

20.4) <u>EXEMPLO</u>

Dado um digrafo $D = (V,E)$ abaixo:

Estrutura de Adjacência:

v_1 : v_2, v_3, v_4, v_7, v_8

v_2 : v_4, v_5, v_7

v_3 : —

v_4 : —

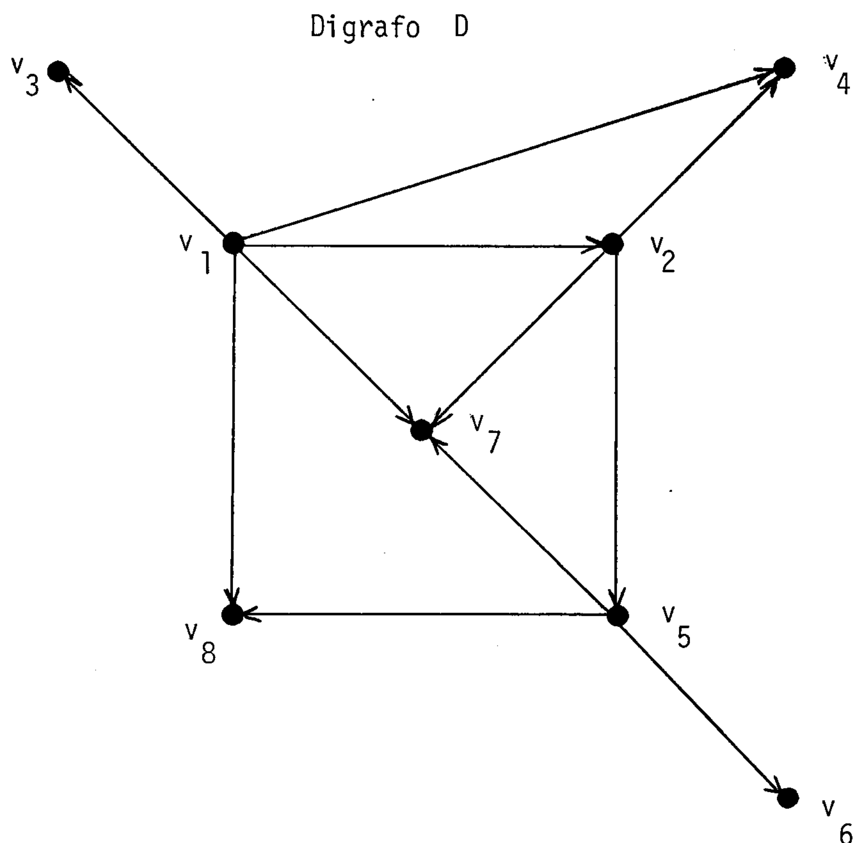
v_5 : v_6, v_7, v_8

v_6 : —

v_7 : —

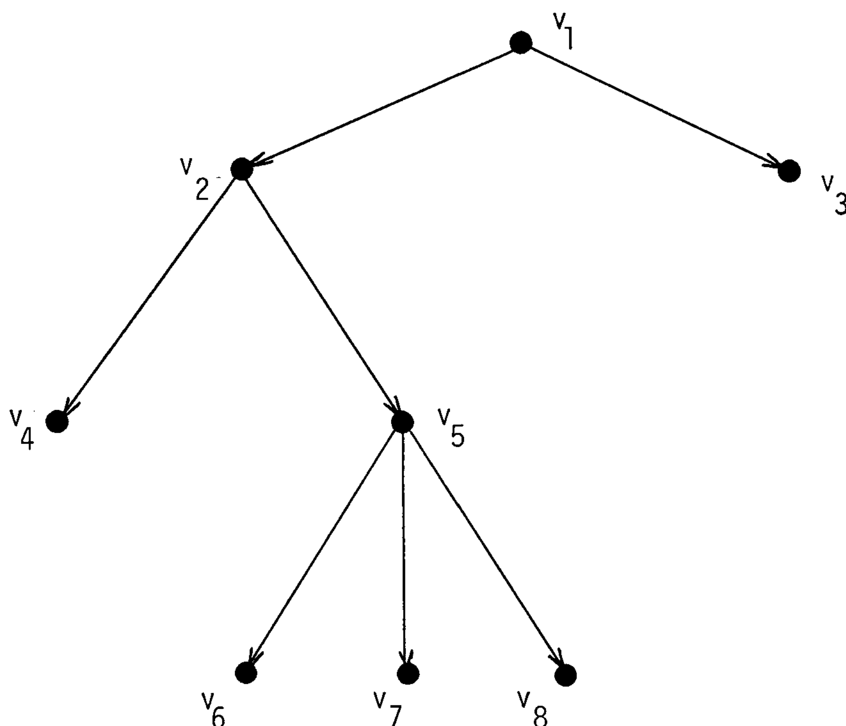
v_8 : —

Digrafo D



1) Reconhecer se D é digrafo estruturado em árvore:

As listas de adjacência já estão ordenadas topologicamente. Aplicando a técnica de busca em profundidade em D, teremos:



O digrafo D é estruturado em árvore, pois a estrutura acima obtida da busca em profundidade de D não possui arestas de cruzamento.

2) Obter a redução transitiva e o fechamento transitivo de D.

A <u>redução transitiva</u> é a estrutura de busca em profundidade sem arestas de cruzamento obtida de D e está descrita acima.

Lista de adjacência A(v) do <u>fechamento transitivo</u>:

$A(v_1) \rightarrow v_2, v_3, v_4, v_5, v_6, v_7, v_8$

$A(v_2) \rightarrow v_4, v_5, v_6, v_7, v_8$

$A(v_3) \rightarrow$ —

$A(v_4) \rightarrow$ —

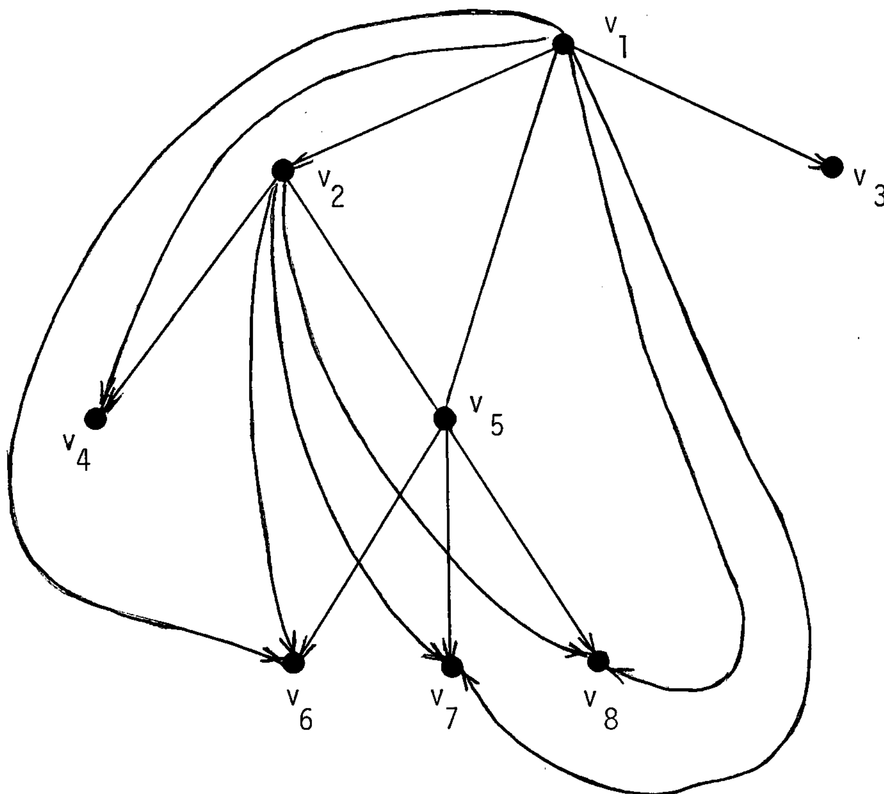
$A(v_5) \rightarrow v_6, v_7, v_8$

$A(v_6) \rightarrow$ —

$A(v_7) \rightarrow$ —

$A(v_8) \rightarrow$ —

Logo, o fechamento transitivo será:



3) Verificar se o digrafo D' abaixo é isomorfo ao digrafo D dado:

Estrutura de Adjacência

v'_1 : v'_2, v'_3, v'_4, v'_7, v'_8

v'_2 : v'_4, v'_5, v'_7

v'_3 : —

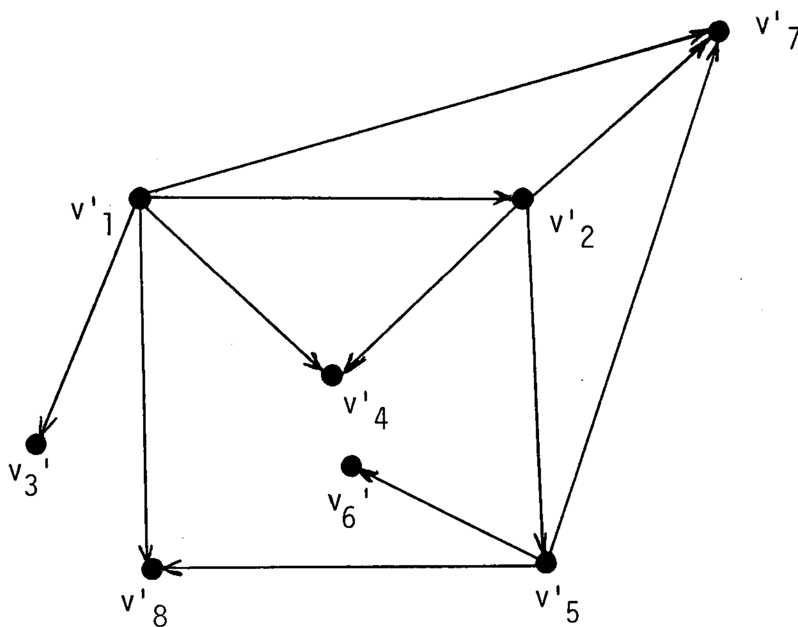
v'_4 : —

v'_5 : v'_6, v'_7, v'_8

v'_6 : —

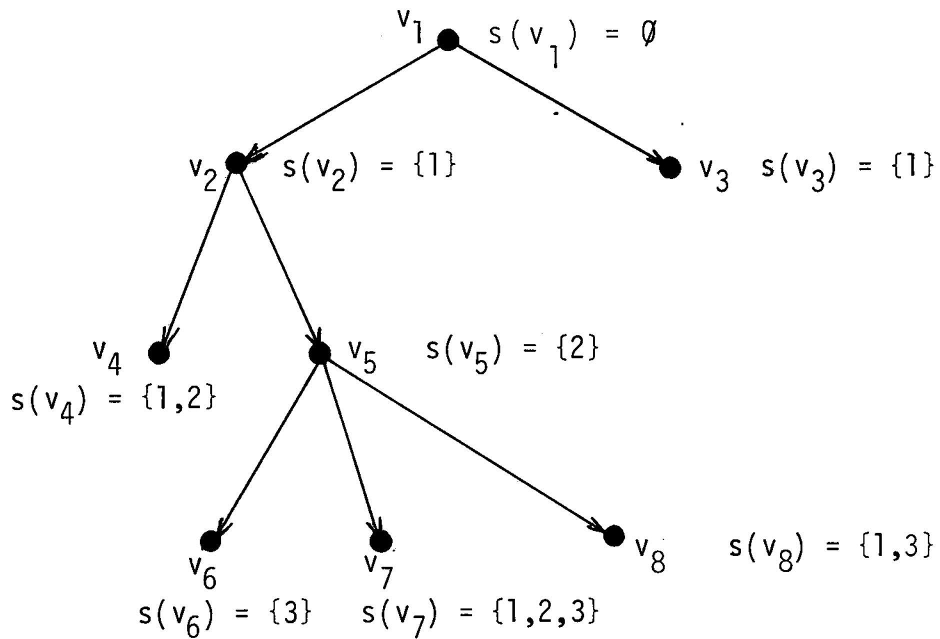
v'_7 : —

v'_8 : —

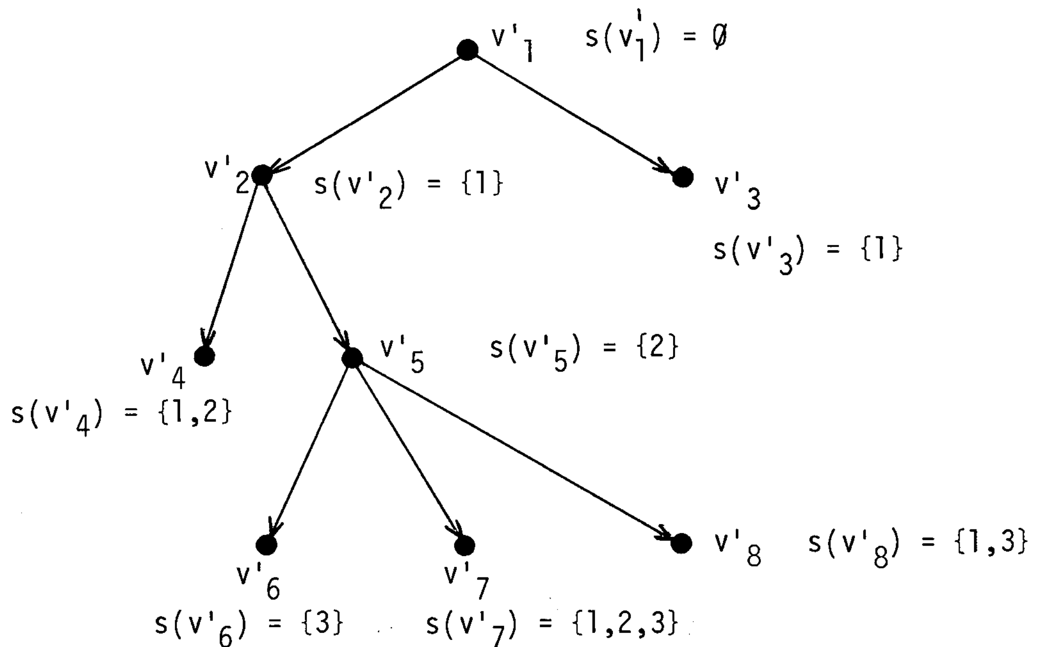


Digrafo D'

Árvore rotulada T representando o digrafo D:



Árvore rotulada T' representando o digrafo D':



T e T' são árvores enraizadas rotuladas isomorfas então D e D' são digrafos isomorfos.

4) Encontrar um conjunto de decomposição mínima por cadeias do digrafo estruturado em árvore D.

Ele será por exemplo:

$\{(v_1, v_2, v_4), (v_3), (v_5, v_6), (v_7), (v_8)\}$

5) Encontrar um conjunto mínimo de ordenações topológicas que define um poset induzido pelo digrafo estruturado em árvore D.

Pré-ordem esquerda: $(v_1, v_2, v_4, v_5, v_6, v_7, v_8, v_3) = p_1$

Pré-ordem direita : $(v_1, v_3, v_2, v_5, v_8, v_7, v_6, v_4) = p_2$

$\{p_1, p_2\} \rightarrow$ conjunto de ordenaçõs topológicas que caracteriza o poset induzido por D.

Dimensão do poset = 2

21) APLICAÇÃO - ISOMORFISMO DE BUSCA EM PROFUNDIDADE

21.1) REFERÊNCIA - SZWARCFITER[69]

21.1.1) OUTRA REFERÊNCIA - HOPCROFT & TARJAN[40]

21.2) DESCRIÇÃO DO PROBLEMA

Verificar se duas Buscas em Profundidade (BP), executadas em um grafo não direcionado, são isomorfas.

Seja $G = (V,E)$ um grafo não direcionado.

Sejam S_1 e S_2 duas buscas em profundidade executadas em G.

Duas buscas em profundidade S_1 e S_2 começando respectivamente de vértices r_1 e r_2 de um grafo não direcionado $G = (V,E)$ são <u>isomorfas</u> quando existe uma permutação f tal que:

(i) $r_2 = f(r_1)$ e

(ii) Para cada aresta $(v,w) \in E$

(v,w) é uma aresta de árvore em S_1 se e somente se $(f(v), f(w))$ é uma aresta de árvore em S_2,

(v,w) é uma aresta de retorno em S_1 se e somente se $(f(v), f(w))$ é uma aresta de retorno em S_2.

21.3) <u>MÉTODO - BUSCA EM PROFUNDIDADE (EM DIGRAFOS ESTRUTURADOS EM ÁRVORE)</u>

Dado $G = (V,E)$ e duas buscas em profundidade S_1 e S_2, primeiramente obtemos os digrafos estruturados em árvore D_1 e D_2, direcionando cada aresta de G dos menores para os maiores números de busca em profundidade (NUM(v)), respectivamente. Assim podemos notar que as arestas de árvore de S_1 são mapeadas em arestas que pertencem à redução transitiva de D_1, enquanto as arestas de retorno de S_1 são mapeadas em arestas que são redundantes sob a redução transitiva. Isto significa que a <u>busca em profundidade de S_1 corresponde</u> precisamente à <u>redução</u>

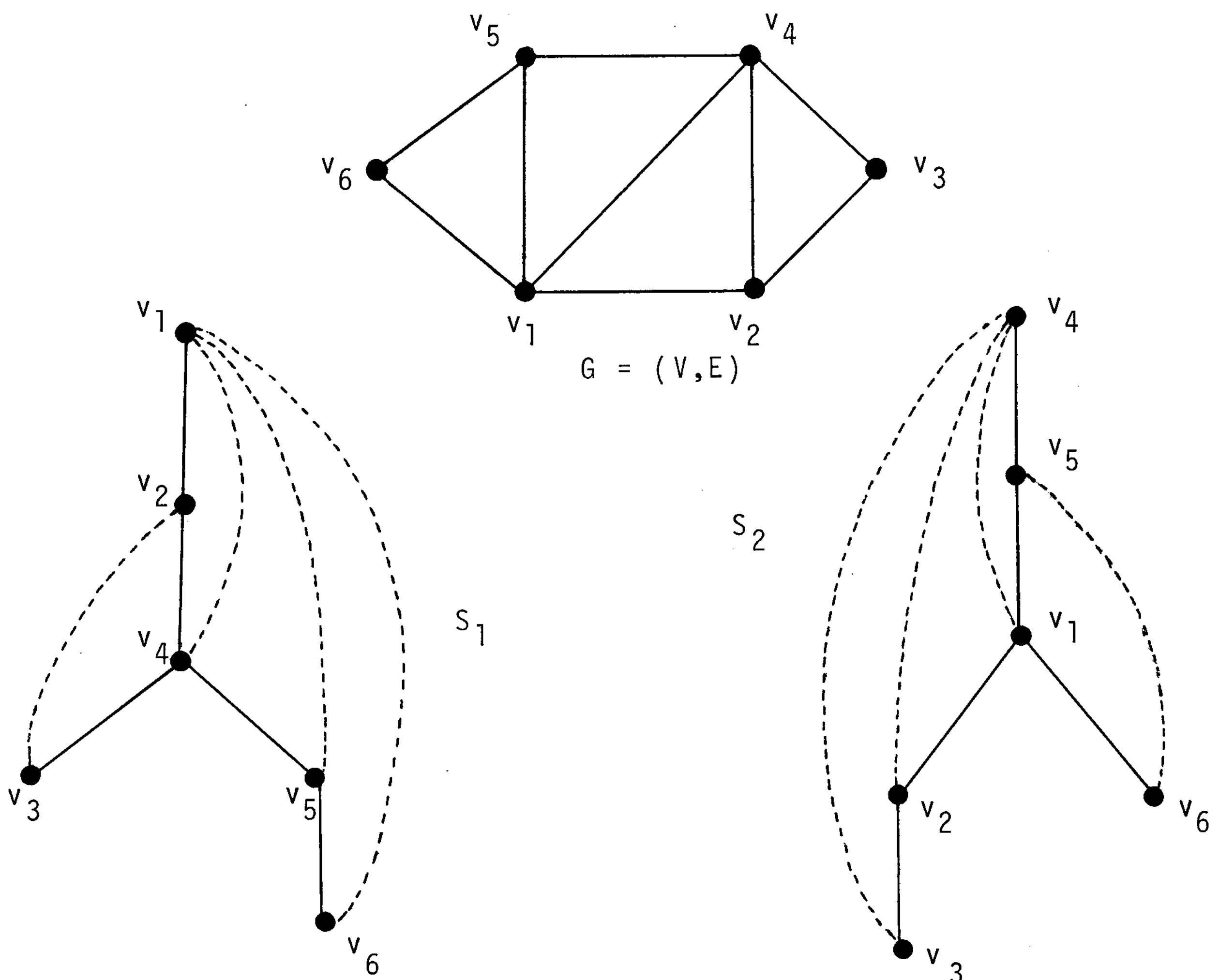
transitiva de D_1. Aplicamos o mesmo processo para S_2 e D_2, respectivamente.

Depois de obter D_1 e D_2 aplicamos o método já visto para decidir o isomorfismo desses digrafos estruturados em árvore (Ver HOPCROFT & TARJAN[40]). Finalmente, S_1 e S_2 são isomorfas se e somente se D_1 e D_2 são isomorfos.

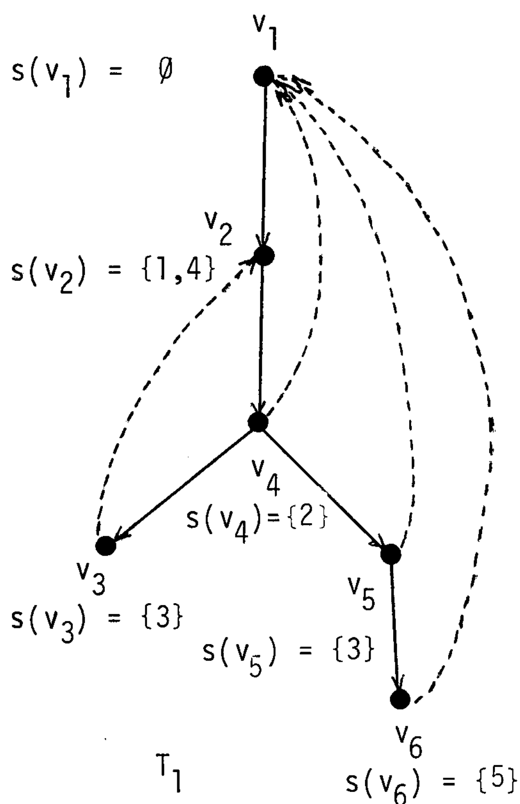
O tempo de execução desse processo inteiro pode ser claramente mostrado ser também limitado por $O(|V| + |E|)$

21.4) EXEMPLO

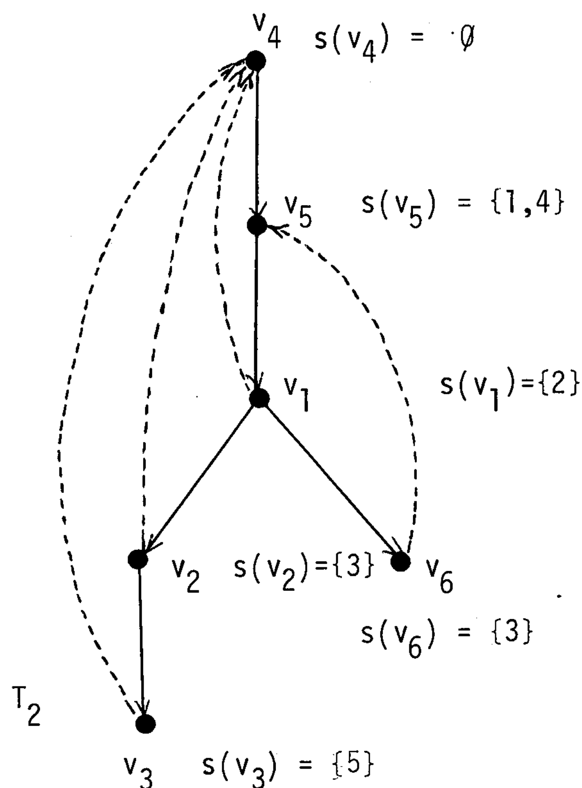
Dado o grafo não direcionado G, e duas buscas em profundidade S_1 e S_2, verificar se S_1 e S_2, são isomorfas:



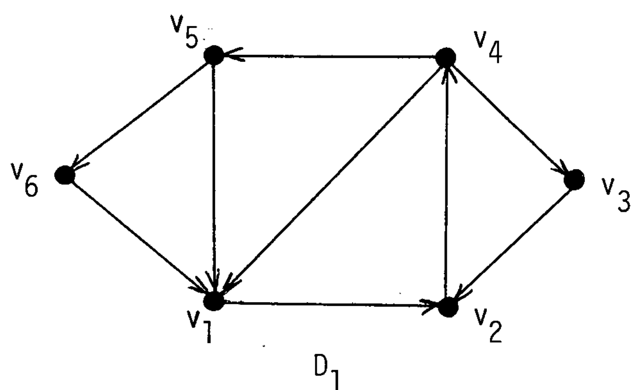
Redução Transitiva de D_1

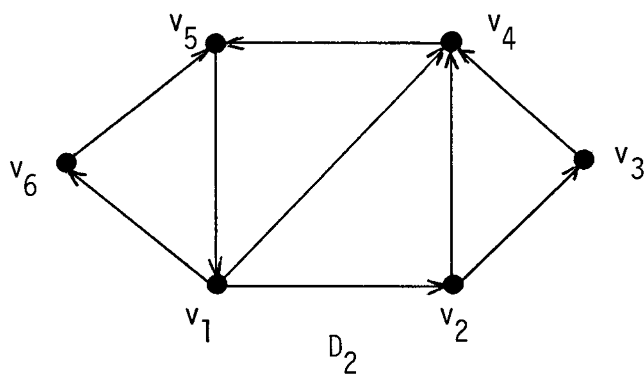


Redução Transitiva de D_2



Obter os digrafos estruturados em árvore D_1 e D_2:





As reduções transitivas de D_1 e D_2 são respectivamente T_1 e T_2 (que correspondem às Buscas em Profundidade S_1 e S_2, respectivamente, só que T_1 e T_2 estão direcionadas).

Achamos então os conjuntos de rótulos de todos os vértices em T_1 e T_2

Conjuntos de rótulos de T_1

$s(v_1) = \emptyset$

$s(v_2) = \{1, 4\}$

$s(v_3) = \{3\}$

$s(v_4) = \{2\}$

$s(v_5) = \{3\}$

$s(v_6) = \{5\}$

Conjuntos de rótulos de T_2

$s(v_4) = \emptyset$

$s(v_5) = \{1, 4\}$

$s(v_6) = \{3\}$

$s(v_1) = \{2\}$

$s(v_2) = \{3\}$

$s(v_3) = \{5\}$

Aplicando o algoritmo de Hopcroft e Tarjan, para isomorfismo de árvores enraizadas rotuladas, podemos ver claramente que T_1 e T_2 são isomorfas. Logo D_1 e D_2 são digrafos isomorfos, o que implica em S_1 e S_2 são isomorfas.

22) APLICAÇÃO - ISOMORFISMO DE DIGRAFOS REDUTÍVEIS EM ÁRVORE

22.1) REFERÊNCIA - SZWARCFITER[69]

22.1.1) OUTRAS REFERÊNCIAS - HOPCROFT & TARJAN[40]; HECHT & ULLMAN[38]; HECHT & ULLMAN[39]; TARJAN[75].

22.2) DESCRIÇÃO DO PROBLEMA

Verificar se dois digrafos redutíveis em árvore são isomorfos.

Seja $D = (V,E)$ um digrafo (possivelmente com ciclos) enraizado em r. Seja S uma Busca em Profundidade (BP) de D, começando de r. Denote por R_S o conjunto de arestas de retorno obtidas de S. O Digrafo D é dito redutível quando R_S é sempre o mesmo, independente de S. Digrafos redutíveis foram caracterizados em HECHT & ULLMAN[38] e HECHT & ULLMAN[39], e eles podem ser reconhecidos em tempo ligeiramente mais do que linear por Tarjan (TARJAN[75]).

Seja D(S) o digrafo acíclico obtido de D, retirando as arestas de R_S, isto é, D(S) é o digrafo $(V,E - R_S)$. Um digrafo redutível D será dito redutível em árvore quando D(S) é um digrafo estruturado em árvore.

Se D é um digrafo acíclico então claramente $R_S = \phi$ e $D(S) = D$, para qualquer Busca em Profundidade S. Portanto, a classe de digrafos redutíveis contém todos os digrafos acícli

cos. Isto significa que não podemos resolver o problema de isomorfismo para digrafos redutíveis, sem resolver também o isomorfismo de grafos gerais. Porém, quando D é redutível em árvore, existe uma maneira simples de verificar isomorfismo.

22.3) MÉTODO - BUSCA EM PROFUNDIDADE: (BP)

O isomorfismo de digrafos estruturados em árvore descrito em SZWARCFITER[69], foi baseado essencialmente em duas idéias:

(i) Se D é um digrafo estruturado em árvore, existe uma maneira eficiente de unicamente encontrar uma árvore geradora especial T (a redução transitiva de D).

(ii) Todas as arestas (v,w) de D são tais que w é um descendente de v em T.

Neste caso, contudo, precisamos distinguir entre arestas de avanço e arestas de retorno. Uma possível maneira de realizar isto é considerar a árvore de subdivisão T' de T, isto é, T' é a árvore enraizada direcionada obtida de T substituindo cada aresta (v,w) de T por um novo vértice α e adicionando arestas (v,α) e (α,w). A árvore T' quando propriamente rotulada pode representar unicamente um digrafo redutível em árvore.

O seguinte algoritmo encontra esta representação:

Dado o digrafo $D = (V,E)$:

1. Reconhecer se D é de fato um digrafo redutível (ver TARJAN[75]). Pare, no caso de resposta negativa.

2. Encontrar o conjunto R de arestas de retorno. Pare se $(V, E - R)$ não é um digrafo estruturado em árvore.

3. Encontre a árvore de redução transitiva $T(V, E_T)$ do digrafo estruturado em árvore $(V, E - R)$.

4. Encontre a árvore de subdivisão $T'(V', E_{T'})$ de T.

5. Seja $h(z)$ o nível do vértice $z \in V'$ em T'. Agora, para cada vértice $z \in V'$, determine um conjunto de rótulos $s(z)$, da seguinte maneira:
   5.1. Inicialize: $s(v) = \phi$, para todo $v \in V'$.
   5.2. Para cada aresta de avanço $(v,w) \in (E - E_T)-R$, adicione o rótulo $h(v)$ a $s(w)$.
   5.3. Para cada aresta de retorno $(v,w) \in R$, adicione o rótulo $h(w) + 1$ a $s(v)$.

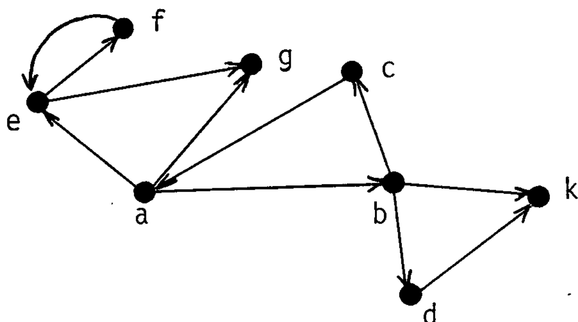
Se este algoritmo não parar nos passos 1 ou 2, então ele encontra uma árvore enraizada direcionada rotulada que unicamente representa o digrafo D. A verificação de isomorfismo agora é simples. Dado os digrafos redutíveis em árvore D_1 e D_2, encontramos suas representações de árvore T'_1 e T'_2, respectivamente. Assim, D_1, D_2 são isomorfos se e somente se T'_1,T'_2

são isomorfas como árvores enraizadas direcionadas rotuladas. (Ver HOPCROFT & TARJAN[40]). A complexidade do passo 1 é um pouco mais do que linear. Todos os outros passos podem ser executados em tempo linear.

22.4) EXEMPLO

Verificar se os digrafos redutíveis em árvore D_1 e D_2 são isomorfos.

Digrafo D_1



Estrutura de Adjacência:

a → b, e, g

b → c, d, k

c → a

d → k

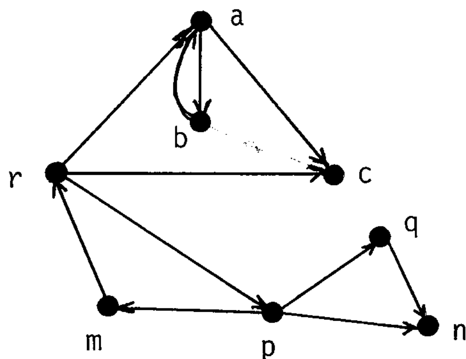
e → f, g

f → e

g → -

k → -

Digrafo D_2



Estrutura de Adjacência:

r → a, c, p

a → b, c

b → a

c → -

p → q, m, n

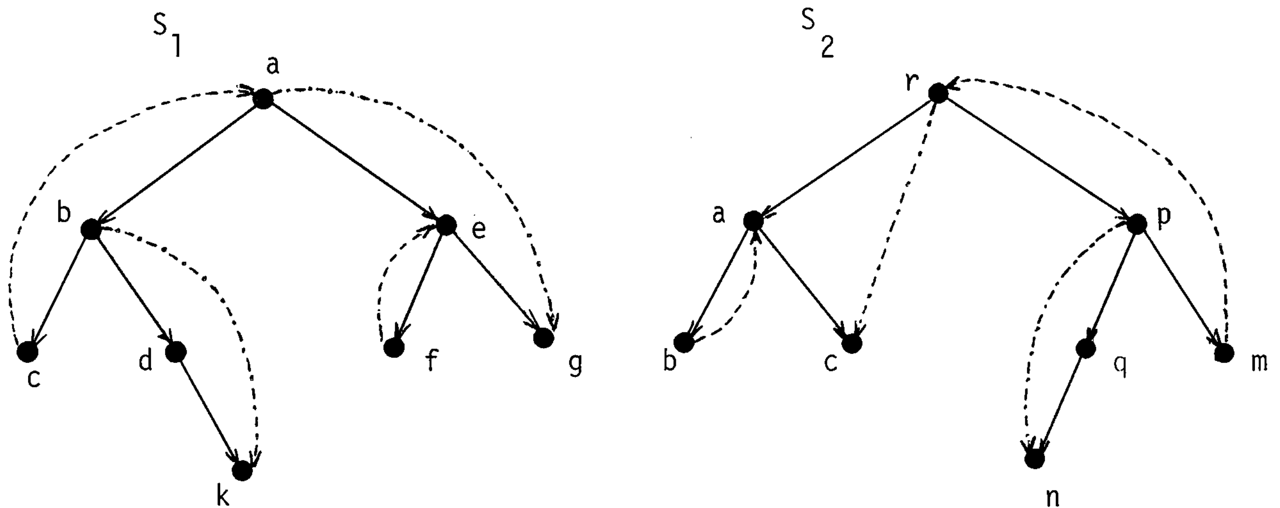
q → n

m → r

n → -

1) D_1 e D_2 são digrafos redutíveis (Ver TARJAN[75])

Busca em Profundidade



$R_1 = \{(c,a),\ (f,e)\}$ $\qquad$ $R_2\ \{(b,a),\ (m,r)\}$

2) $(V,\ E - R_1)$ é um digrafo estruturado em árvore

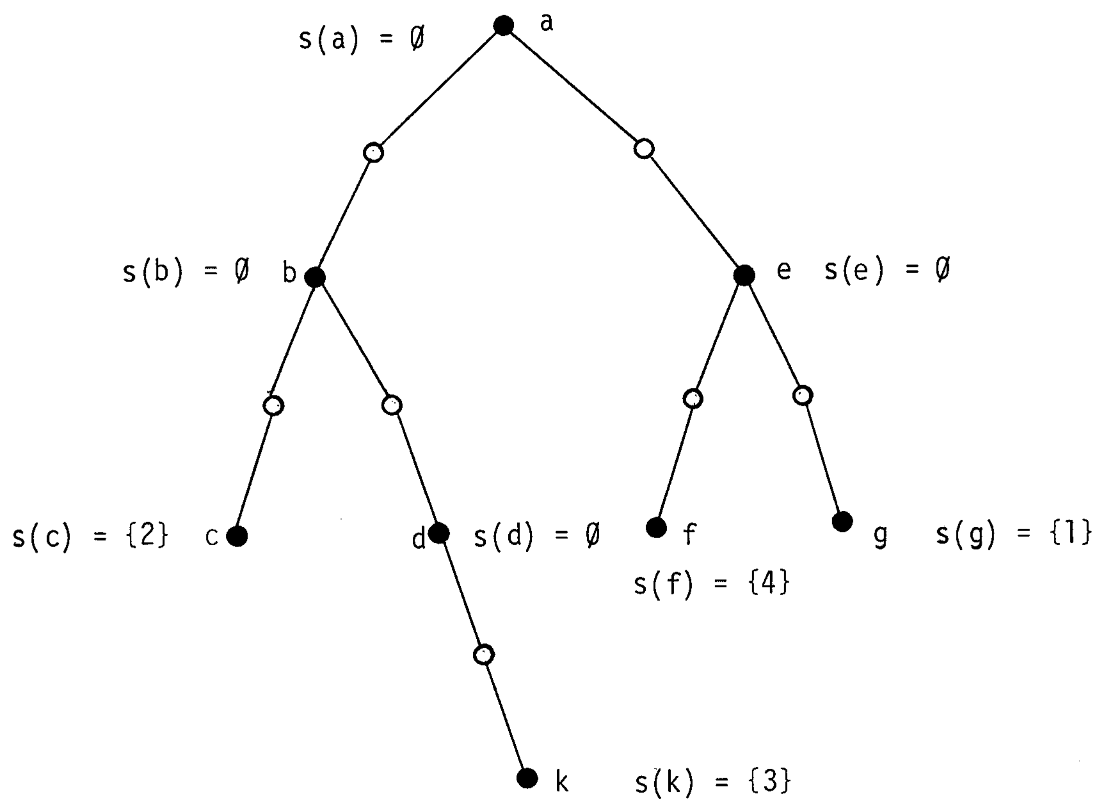
$(V,\ E - R_2)$ é um digrafo estruturado em árvore

3) Árvore de redução transitiva = $T_1(V,\ E_{T_1})$

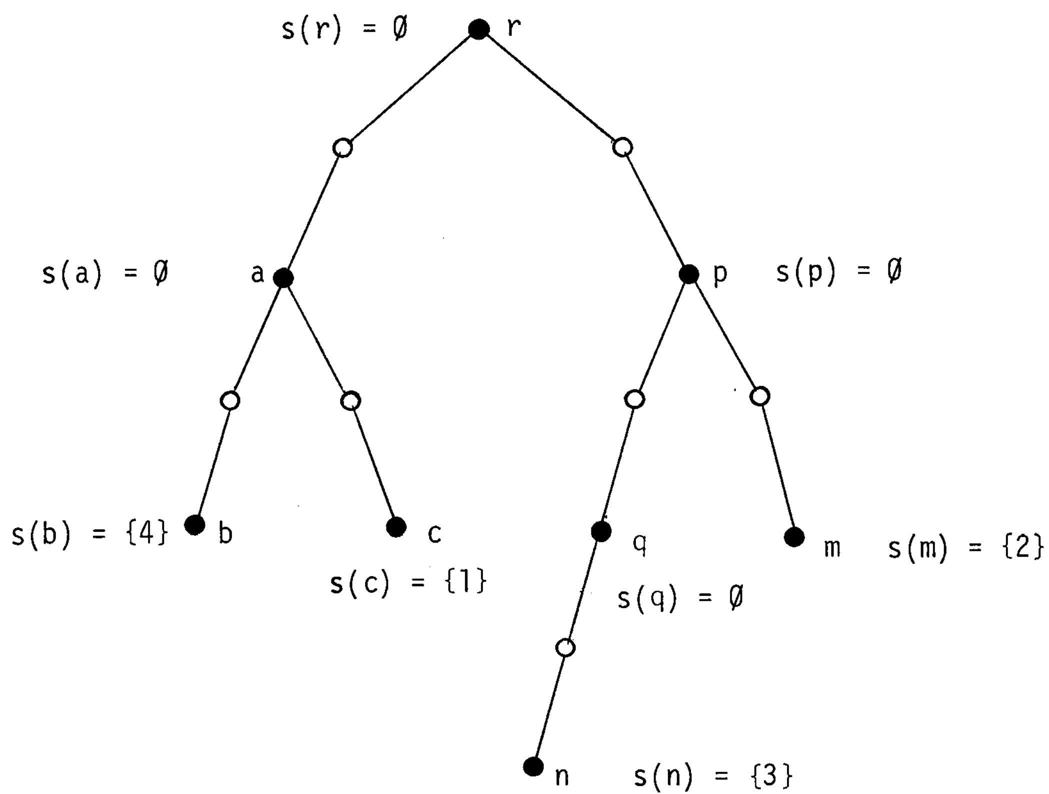
Árvore de redução transitiva = $T_2(V,\ E_{T_2})$

onde E_{T_1} e E_{T_2} são as arestas de árvore de S_1 e S_2, respectivamente.

4) Árvore de subdivisão $T'_1(V', E_{T_1})$ de T_1



Árvore de subdivisão $T'_2(V', E_{T_2})$ de T_2



5) $s(a) = \phi \longrightarrow s(r) = \phi$

$s(b) = \phi \longrightarrow s(p) = \phi$

$s(c) = \{2\} \longrightarrow s(m) = \{2\}$

$s(d) = \phi \longrightarrow s(q) = \phi$

$s(e) = \phi \longrightarrow s(a) = \phi$

$s(f) = \{4\} \longrightarrow s(b) = \{4\}$

$s(g) = \{1\} \longrightarrow s(c) = \{1\}$

$s(k) = \{3\} \longrightarrow s(n) = \{3\}$

T'_1 e T'_2 são isomorfas (Ver HOPCROFT & TARJAN[40]), então podemos afirmar que D_1 e D_2 são isomorfos.

23) APLICAÇÃO - CONJUNTO DE CORTE MÍNIMO EM DIGRAFOS REDUTÍVEIS

23.1) REFERÊNCIA - SHAMIR[66]

23.1.1) OUTRAS REFERÊNCIAS - TARJAN[71]; TARJAN[75]; HECHT & ULLMAN[39].

23.2) DESCRIÇÃO DO PROBLEMA

Encontrar os conjuntos de corte mínimo em digrafos redutíveis.

A análise de muitos processos modelados por grafos direcionados requer a seleção de um subconjunto de vértices que cortam todos os ciclos no grafo. A redução do tamanho de um

tal conjunto de corte usualmente leva a uma análise mais simples e mais eficiente, mas o problema de encontrar conjuntos de corte mínimo em grafos direcionados gerais é conhecido ser NP-completo. Aqui, porém, estamos interessados em encontrar conjuntos de corte mínimo em digrafos redutíveis, e isto poderá ser feito em tempo linear (Ver SHAMIR[66]).

Veremos agora, algumas definições:

Def.: Um vértice v corta um caminho P se ele pertence a P.

Def.: Um conjunto S de vértices em um digrafo D é um conjunto de corte se qualquer ciclo em D é cortado por pelo menos um vértice de S.
Um conjunto de corte S é mínimo se para quaisquer outros conjuntos de corte S', $|S| \leq |S'|$. Os vértices em um conjunto de corte são chamados vértices de corte.

Nota: Um conjunto de corte mínimo de D não é necessário ser único. Por exemplo, quando D é um ciclo, qualquer vértice de D é um conjunto de corte. Contudo, todos os conjuntos de corte mínimo têm o mesmo tamanho.

Def.: O conjunto de todos os ciclos em um digrafo D é denotado por C_D.

Def.: Dado um conjunto S de vértices, o conjunto de todos os ciclos em D que não são cortados por vértices em S é denotado por $C^S{}_D$.
Claramente, $C^\phi{}_D$ é o conjunto inicial de ciclos C_D, e um conjunto S é um conjunto de corte se e somente se $C^S{}_D = \phi$.

Def.: Dado um digrafo enraizado D = (V, E, r) e uma Busca em Profundidade (BP) α, definimos o grafo direcionado acíclico (g d a) de D definido por α ser $D^\alpha{}_d = (V, E^\alpha{}_d, r)$ onde $E^\alpha{}_d$ é o conjunto de arestas g d a (arestas de árvore, avanço e cruzamento) em E.
$D^\alpha{}_d$ é sempre um g d a enraizado, e se alguma aresta em $E - E^\alpha{}_d$ é adicionada nele, um ciclo é gerado.

Def.: Um digrafo enraizado D = (V, E, r) é redutível se o grafo direcionado acíclico (g d a) de D definido por α, $D^\alpha{}_d$ é o mesmo para qualquer BP (Busca em Profundidade) α de D.

Def.: Um vértice v' domina um outro vértice v em um digrafo enraizado D = (V, E, r) se v' corta qualquer caminho de r para v.

<u>Resultado 1</u>: Se (v,v') é uma aresta de retorno em um digrafo redutível D, então v' domina v.

<u>Resultado 2</u>: Se D é um digrafo redutível, então qualquer ciclo simples em D contém exatamente uma aresta de retorno.

<u>OBS.</u>: Qualquer ciclo em um digrafo contém pelo menos um ciclo simples. O <u>Res. 2</u> fornece uma partição útil do conjunto de ciclos simples em D em termos de suas arestas de retorno. Assim, para verificar se um dado conjunto <u>S</u> é um <u>conjunto de corte</u>, é suficiente verificar, para qualquer aresta de retorno (v,v') em D, que todos os caminhos do grafo direcionado acíclico (g d a) de v' para v contém vértices de S.

<u>Def.</u>: Se (v,v') é uma aresta de retorno em um digrafo redutível D, então v' é dito uma <u>cabeça</u> e v é dito um <u>final</u> em D (também diremos que v' e v são cabeça e final correspondentes).

<u>Def.</u>: Seja D um grafo redutível que é parcialmente cortado por um conjunto S de vértices. Então uma <u>cabeça</u> v é <u>ativa</u> se existe algum caminho no g d a de v para um final correspondente, que não é cortado por vértices de S.

Uma cabeça ativa é máxima se nenhum de seus descendentes do g d a em D é uma cabeça ativa.

Nota: Se uma cabeça v é ativa, então existe pelo menos um ciclo em C^S_D que contém v, mas não necessariamente vice-versa. Portanto, a não ser que S seja um conjunto de corte, o digrafo D contém pelo menos uma cabeça ativa, e também pelo menos uma cabeça ativa máxima.

Resultado 3: Seja D um digrafo redutível, e seja S um subconjunto de um conjunto de corte mínimo em D. Se v_1 é uma cabeça ativa máxima em D, então $S \cup \{v_1\}$ é também um subconjunto de um conjunto de corte mínimo em D.

23.3) MÉTODO - BUSCA EM PROFUNDIDADE

O algoritmo básico para resolver o problema de achar conjuntos de corte mínimo em digrafos redutíveis, seria o seguinte:

Algoritmo A:

1. Comece com $S = \phi$
2. Selecione uma cabeça ativa máxima v em D com respeito ao conjunto corrente S. Se não existe nenhuma, pare ; caso contrário faça $S \leftarrow S \cup \{v\}$ e repita o passo 2.

Quando for implementar este algoritmo, é conveniente enumerar as cabeças em D por uma Busca em Profundidade, e considerá-las em pós-ordem.

OBS.: Uma BP define duas possíveis ordens nos vértices:

i) Pré-ordem → a ordem em que os vértices são colocados na pilha durante a BP.

ii) Pós-ordem → a ordem em que os vértices são retirados da pilha durante a BP.

Por propriedades de BP, todos os descendentes do gda de um vértice v (e em particular as cabeças ativas entre eles) ocorrem antes de v em pós-ordem. Qualquer cabeça que é ativa em algum estágio intermediário no algoritmo é imediatamente adicionada a S, e assim parando de ser ativa com respeito ao novo S. Além disso, o conjunto S pode ser expandido, e então uma cabeça não-ativa não pode se tornar ativa novamente em um estágio mais tarde. Consequentemente, se cabeças são consideradas em pós-ordem, então qualquer cabeça que ainda é ativa quando sua volta acontece, será uma cabeça ativa máxima.

A maneira mais simples de verificar se uma dada cabeça v é ativa, é pesquisar caminhos de g d a não cortados entre v e seus finais correspondentes. Isto pode ser feito em tempo linear propagando os rótulos de v através de arestas do g d a (que são arestas de árvore, avanço e cruzamento), mas o processo de rotulação tem que ser repetido para qualquer cabeça máxima, assim dando um algoritmo de $O(|V|.|E|)$.

A fim de desenvolver um algoritmo mais eficiente, nós estruturamos (Ver SHAMIR[66]) a busca de uma tal maneira que cada aresta é usada somente uma vez, mesmo que ela possa pertencer ao caminho do g d a entre muitos pares de cabeça - final em D. Como a busca deve transmitir informação suficiente a fim de determinar quais detes caminhos são cortados e quais ainda não foram cortados, parece que nós necessitamos rotular aqueles que são conjuntos de cabeças. Infelizmente, este método leva a algoritmos não-lineares.

A estrutura especial de digrafos redutíveis nos permite usar rótulos muito mais econômicos. Para cada vértice v nós associamos um número simples $\ell_S(v)$, que pode ser trocado quando novos vértices de corte são adicionados ao conjunto corrente S. Denotamos por n(v) (um inteiro entre 1 e $|V|$) a posição de v na sequência pré-ordem de vértices em D.

Resultado

Seja D um digrafo redutível e seja S um conjunto arbitrário de vértices. Então para qualquer vértice v em D, a seguinte equação é válida:

$$\ell_S(v) = \begin{cases} 0, & \text{se } v \in S \text{ ou } v \text{ não tem descendentes em D,} \\ \max[\ell_S(v_1), \ldots, \ell_S(v_i), n(v_{i+1}), \ldots, n(v_k)], & \text{caso contrário.} \end{cases}$$

onde $(v,v_1), (v,v_2), \ldots, (v,v_i)$ são todas as arestas do g d a saindo de v, e $(v, v_{i+1}), \ldots, (v, v_k)$ são todas as arestas de retorno saindo de v.

<u>OBS.</u>: O Algoritmo Final (abaixo) usa então uma simples Busca em Profundidade de D a fim de numerar os vértices de D em pré-ordem, rotulá-los em pós-ordem, considerar cabeças sucessivas em pós-ordem, e adicionar novos vértices a S (trocando seus rótulos por zero) quando seus rótulos de pós-ordem e números de pré-orden coincidem. As complexidades em tempo e espaço são claramente lineares em relação ao tamanho de D, ou seja $O(|V| + |E|)$.

<u>Algoritmo Final</u>

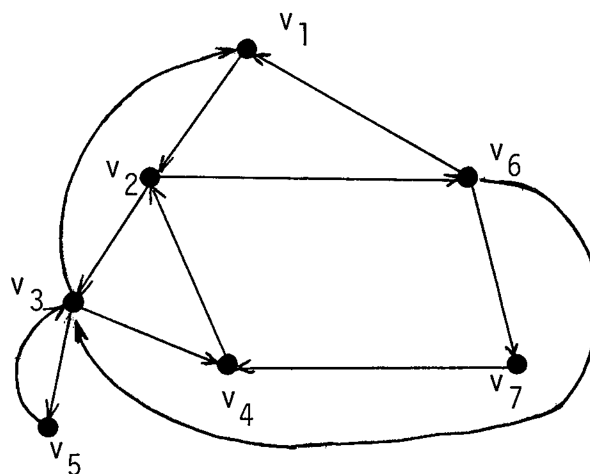
<u>Nota</u>: Rótulos são denotados por $\ell(v)$, sem um conjunto subscrito explicito; o algoritmo mantém somente um sistema de tais rótulos, que correspondem a algum estágio para o conjunto corrente S. A variável top indica o topo da pilha.

1. Seja $S \leftarrow \phi$; vértice contador $c \leftarrow 1$; coloque r na pilha (vazia).
2. $n(top) \leftarrow c$; $c \leftarrow c+1$; $\ell(top) \leftarrow 0$.
3. Se existe alguma aresta desmarcada (top, v), então marque-a e continue; caso contrário vá para o passo 7.
4. Se v não foi visitado até agora, coloque v na pilha e vá para o passo 2.
5. Se (top, v) é uma aresta de retorno, faça $\ell(top) \leftarrow \max(\ell(top), n(v))$ e vá para o passo 3.

6. Faça $\ell(top) \leftarrow max(\ell(top), \ell(v))$ e vá para o passo 3.
7. Se $\ell(top) = n(top)$, faça $S \leftarrow S \cup \{top\}$ e $\ell(top) \leftarrow 0$.
8. Guarde o top da pilha em v'; retire da pilha; se a pilha é vazia, pare, caso contrário faça $\ell(top) \leftarrow max(\ell(top), \ell(v'))$; vá para o passo 3.

23.4) EXEMPLO

Encontrar os conjuntos de corte mínimo do digrafo redutível abaixo:



OBS.: a) O digrafo acima é redutível, pois é fácil verificar que qualquer Busca em Profundidade (BP) reconhece (v_3,v_1), (v_4,v_2), (v_5,v_3) e (v_6,v_1) como arestas de retorno e todas as outras arestas como arestas do g d a (Ver BP a seguir).

Aplicando a técnica de Busca em Profundidade:

Estrutura da Adjacência:

$v_1 \rightarrow v_2$

$v_2 \rightarrow v_3, v_6$

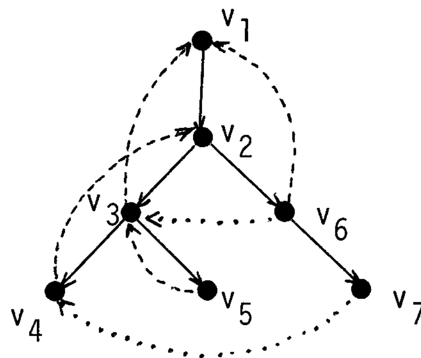
$v_3 \rightarrow v_1, v_4, v_5$

$v_4 \rightarrow v_2$

$v_5 \rightarrow v_3$

$v_6 \rightarrow v_1, v_3, v_7$

$v_7 \rightarrow v_4$



v	n(v)	ℓ(v)
v_1	1	0
v_2	2	~~2~~ 0
v_3	3	~~2~~ ~~3~~ 0
v_4	4	2
v_5	5	3
v_6	6	2
v_7	7	2

$S = \{v_3, v_2\}$

Note que a BP em que filhos da esquerda são considerados antes de filhos da direita, dá origem a números de pré-ordem satisfazendo $n(v_i) = i$, para todo i.

Aplicando o Algoritmo Final, começaremos fazendo $S \leftarrow 0$ e colocando a raiz v_1 na pilha. Então continuamos ao longo do caminho do g d a $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$, colocando v_2, v_3, e v_4 na pilha. A aresta (v_4, v_2) é então encontrada como sendo uma aresta de retorno, e assim $\ell(v_4) \leftarrow max(0, n(v_2)) = 2$. Desde que v_4 não tem outros filhos, nós verificamos que $\ell(v_4) = 2 \neq 4 = n(v_4)$ (passo 7), retire v_4 da pilha, e faça $\ell(v_3) \leftarrow max(0, \ell(v_4)) = 2$. O vértice v_5 é então colocado na pilha, e a aresta de retorno (v_5, v_3) é encontrada e então $\ell(v_5) \leftarrow max(0, n(v_3)) = 3$, Retirando v_5 da pilha, atualizamos $\ell(v_3) \leftarrow max(2, \ell(v_5)) = 3$.

Considerando agora a aresta de retorno (v_3, v_1), faremos $\ell(v_3) \leftarrow max(3, n(v_1)) = 3$. Antes de retirar v_3 da pilha, descobrimos que $\ell(v_3) = 3 = n(v_3)$, e assim adicionamos v_3 a S e trocamos seu rótulo para 0 $(\ell(v_3) \leftarrow 0)$.

Durante o resto do processo, o vértice v_7 obtém o rótulo 2 $(\ell(v_7) \leftarrow 2)$ calculado por v_4; o vértice v_6 obtém o rótulo 2 $(\ell(v_6) \leftarrow max(0, \ell(v_3), \ell(v_7), n(v_1)) = 2)$, e o vértice v_2 obtém o rótulo 2 $(\ell(v_2) \leftarrow max(0, \ell(v_3), \ell(v_6)) = 2)$. Antes de fazer o backtrack de v_2 para v_1, nós novamente descobrimos que $\ell(v_2) = 2 = n(v_2)$, e então adicionamos v_2 a S, trocando seu rótulo por $0(\ell(v_2) = 0)$. Isto leva a $\ell(v_1) \leftarrow max(0, \ell(v_2)) = 0$ e o algoritmo pára depois que v_1 é retirado da pilha.

O conjunto de corte mínimo encontrado pelo algoritmo será $S = \{v_3, v_2\}$. Ele não é único, pois $\{v_5, v_2\}$ e $\{v_3, v_6\}$ são também conjuntos de corte. Todos os outros conjuntos de corte contêm três os mais vértices.

24) APLICAÇÃO - REDUTIBILIDADE DE GRAFO DE FLUXO

24.1) REFERÊNCIA - TARJAN[75]

24.1.1) OUTRAS REFERÊNCIAS - HECHT & ULLMAN[38]; HECHT & ULLMAN[39]

24.2) DESCRIÇÃO DO PROBLEMA

Verificar se um grafo de fluxo é redutível.

Muitos métodos de otimização de código modelam o fluxo de controle em um programa de computador por um grafo direcionado, chamado grafo de fluxo. Para que esses métodos funcionem, o grafo de fluxo deve ter uma propriedade especial chamada redutibilidade. Tais métodos incluem algoritmos para encontrar dominadores, para encontrar subexpressões comuns, encontrar variáveis ativas, encontrar definições inúteis, etc.

Seja $D = (V,E)$ um grafo direcionado onde V é o conjunto de vértices e E o conjunto de arestas.

Um grafo de fluxo (D,s) é um grafo com um vértice distinguido s tal que cada vértice é alcançável de s.

O vértice v "domina" o vértice w no grafo de fluxo

(D,s) se v ≠ w e todo caminho de s para w contém v.

Sejam v e w ≠ s dois vértices em um grafo de fluxo (D,s). "Redução de w em v", significa formar um novo grafo D' de D retirando o vértice w e suas arestas incidentes, adicionando uma aresta (v,x) para cada aresta retirada (w,x) com x ≠ v e (v,x) não sendo uma aresta já existente, e adicionando uma aresta (x,v) para cada aresta retirada (x,w) com x ≠ v e (x,v) não sendo uma aresta já existente. D' é obviamente um grafo de fluxo.

Se D' é formado de D por várias operações de redução, cada vértice v' em D' corresponde a um conjunto de vértices em D, especialmente aqueles reduzidos em v'. Se (v,w) é uma aresta de D tal que v está reduzido em v' em D' e w está reduzido em w', então ou v' = w' ou (v',w') é uma aresta de D'. Neste caso (v',w') em D' corresponde a (v,w) em D.

Considere a seguinte transformação:

TR_1: Se (v,w) é a única aresta convergente em w e w ≠ s, "reduza" w em v.

Um grafo de fluxo é redutível se e somente se ele pode ser transformado em um grafo consistindo somente do vértice s, pela repetida aplicação de TR_1. Esta definição difere da de Hecht e Ullman (Ver HECHT & ULLMAN[38]), em que eles permitem laços (arestas da forma (v,v)) em seus grafos de fluxo, e permite uma outra transformação que retira laços.

Se D' é obtido de D pela repetida aplicação de TR_1, D' é uma redução de D. Se aplicarmos TR_1 repetidas vezes, até o ponto em que não pudermos mais aplicá-la, teremos como re

sultado um único grafo. Assim, a ordem das transformações não é relevante em um teste para redutibilidade.

24.3) <u>MÉTODO - BUSCA EM PROFUNDIDADE</u>

Seja T uma árvore geradora de Busca em Profundidade do grafo de fluxo (D,s).

<u>Teorema 1</u>: D é redutível se somente se w domina v para cada aresta de retorno (v,w)(relativo a T) (Ver HECHT & ULLMAN[39]).

Para cada vértice w em D, seja C(w) = {v | (v,w) é uma aresta de retorno} e seja P(w) = {v | ∃ z ∈ C(w) tal que existe um caminho de v para z que não passa por w}. Se não existem arestas de retorno (v,w), ambos: C(w) e P(w) são vazios.

<u>Lema 2</u>: D é redutível se e somente se para todo <u>w</u> e para todo <u>v ∈ P(w)</u>, existe um caminho de w para v em T. (Ver prova em TARJAN[75]).

Seja w o vértice com numeração mais alta em D com uma aresta de retorno entrando em w. Suponha que para todo v∈P(w) existe um caminho de w para v em T. Seja D' formado de D reduzindo todos os vértices de P(w) em w.

Lema 3: Toda aresta (v',w') em D' corresponde a uma aresta (v,w') de D com um caminho de v' para v em T (Ver prova em TARJAN[75]).

Seja T' o subgrafo de D' cujas arestas correspondem às arestas de T.

Lema 4: T', com a mesma numeração de T, é uma árvore geradora de busca em profundidade de D'. Arestas de retorno de D' correspondem a arestas de retorno de D, arestas de avanço de D' correspondem a arestas de avanço ou arestas de cruzamento de D, e arestas de cruzamento de D' correspondem a arestas de cruzamento de D. (Ver prova em TARJAN[75]).

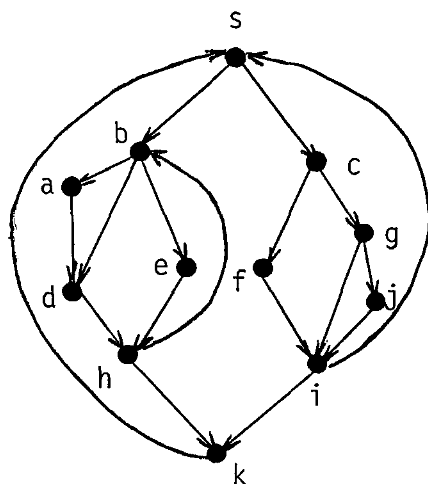
Lema 5: Para qualquer vértice $x < w$, sejam P'(x) e C'(x) definidos em D', relativos a T', assim como P(x) e C(x) o são em relação a T. Então existe um caminho de x para y em T' para todo $y \in$ P'(x) se e somente se existe um caminho de x para y para todo $y \in$ P(x). (Ver prova em TARJAN[75]).

Os lemas 2 - 5 levam a um algoritmo eficiente para testar a redutibilidade de um grafo de fluxo (D,s). (Ver TARJAN[75]).

Em TARJAN[75], Tarjan apresenta um algoritmo com um limite de tempo quase linear para determinar se um grafo de fluxo é redutível. O método usa busca em profundidade e um bom algoritmo para computar uniões de conjuntos disjuntos. O tempo de execução exato do algoritmo depende do tempo de execução exato do algoritmo de união de conjuntos, que é desconhecido. Contudo, um bom limite neste tempo de execução é conhecido, e o algoritmo de redutibilidade requer $O(\min\{E \log^* E, V \log V + E\})$ tempo para testar um grafo com V vértices e E arestas, onde $\log^* x = \min\{i \mid \log^{(i)} x \leq 1\}$. Se $E > V \log V$, o algoritmo requer $O(E)$ tempo e é ótimo dentro de um fator constante, desde que toda aresta deve ser examinada para determinar redutibilidade.

24.4) EXEMPLO

Verificar se o grafo de fluxo abaixo é redutível:

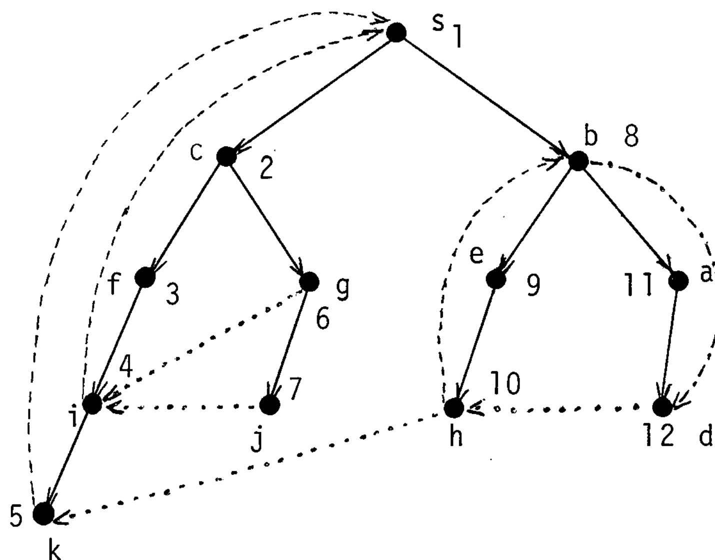


Grafo de Fluxo (D,s)

Estrutura de Adj:

s → c, b
a → d
b → e, a, d
c → f, g
d → h
e → h
f → i
g → i, j
h → b, k
i → s, k
j → i
k → s

Aplicando a técnica de busca em profundidade em (D,s), obtemos a estrutura de busca em profundidade abaixo:

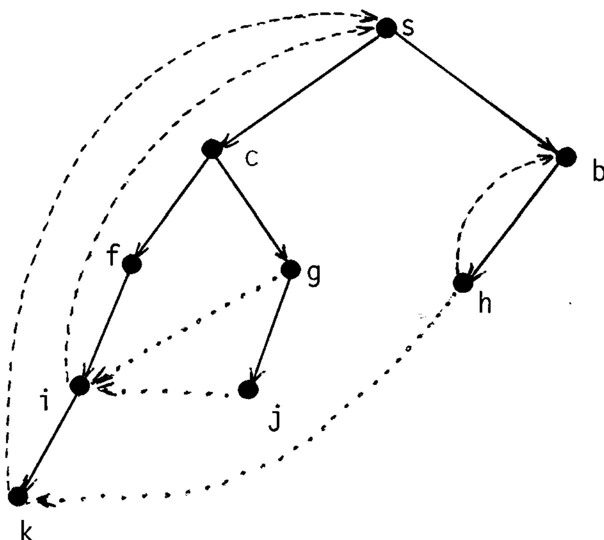


Agora iremos aplicar várias vezes a transformação TR_1.

Seja $w_1 = b \rightarrow C(b) = \{h\}$; $P(b) = \{a,d,e\}$

$w_2 = s \rightarrow C(s) = \{i,k\}$; $P(s) = \{c,f,g,j,b,a,d,e,h\}$

Primeiramente reduzimos a em b, depois reduzimos d em b, e em b, obtendo a seguinte árvore:



Reduzindo h em b, b em s, c em s, g em s, f em s, f em s, i em s, obtemos:

Finalmente reduzindo k em s, obtemos:

e chegamos à conclusão que o grafo de fluxo (D,s) é redutível.

OBS.: Aplicando o lema 2 podemos ver claramente que (D,s) é redutível, pois para todo w e para todo $v \in P(w)$, existe um caminho de w para v em T.

25) APLICAÇÃO - ORIENTAÇÃO DE UM GRAFO NÃO DIRECIONADO

25.1) REFERÊNCIA - SZWARCFITER, PERSIANO & OLIVEIRA[70].

25.1.1) OUTRAS REFERÊNCIAS - LOVÁSZ[50]; TARJAN[71]

25.2) DESCRIÇÃO DO PROBLEMA

Dado um grafo não direcionado G = (V,E) e um par s, t de vértices distintos de G, determinar uma orientação acíclica D para G tendo s como única fonte e t como único sumidouro.

Encontramos o problema aqui descrito em SZWARCFITER, PERSIANO & OLIVEIRA[70], como também as seguintes variações deste problema básico:

(i) O par s, t não é dado, e se quer determinar a única fonte e o único sumidouro da orientação.

(ii) O par s, t é substituido por dois subconjuntos de vértices S e T, e a propriedade que a orientação deve satisfazer passa a ser que os vértices de S sejam as fontes e os vértices de T os sumidouros.

(iii) O par de subconjuntos S, T da variação (ii) é substituido por um par de inteiros positivos k_1, k_2, que devem corresponder, respectivamente, as cardinalidades do conjunto de fontes e sumidouros da orientação.

OBS.: 1) Se direcionamos as arestas de um grafo não direcionado G, obtemos um digrafo D chamado uma orientação de G. O grafo G é então chamado o grafo subjacente a D.

25.3) <u>MÉTODO - BUSCA EM PROFUNDIDADE (BP)</u>

Para cada Busca em Profundidade, definimos a função lowpt: $V \to V$, como se segue: lowpt(v) = w, onde w é o vértice mais próximo da raiz da árvore de BP T que pode ser alcançado começando de v, descendo em T através de zero ou mais arestas de árvore e subindo em T por no máximo uma aresta de retorno.

<u>OBS.</u>: Esta função lowpt é a mesma que já foi definida nas aplicações sobre componentes biconexos e planaridade.

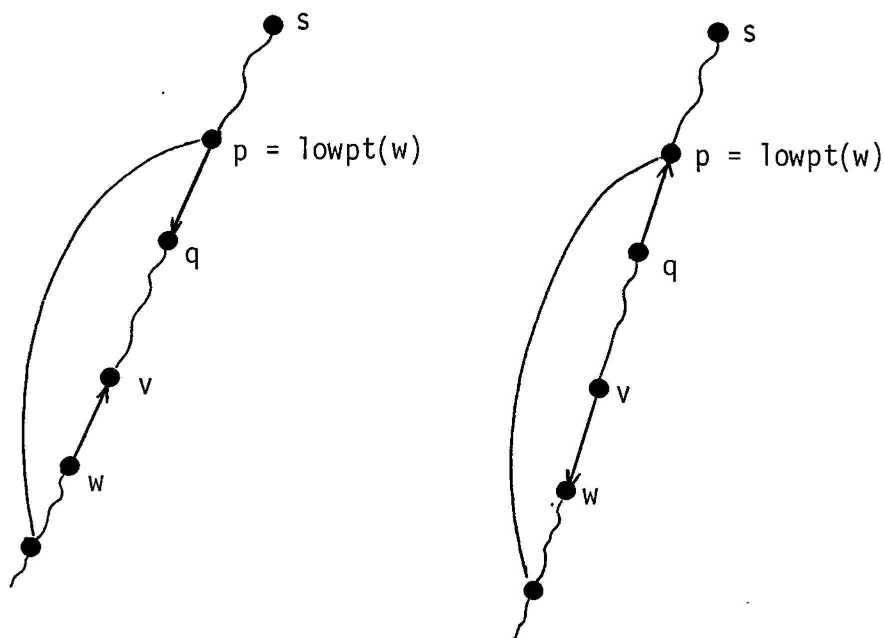
- <u>Problema Básico</u>:

Suponha que sejam dados um grafo não direcionado conexo G = (V,E) e um par de vértices distintos s, t $\in$ V. Desejamos encontrar uma orientação acíclica D de G, tendo uma única fonte s e um único sumidouro t. Assumiremos que o par (s,t) é uma aresta do grafo não direcionado. Caso não seja podemos, evidentemente, adicioná-la a E e tendo encontrado o digrafo D pretendido, podemos obter a solução final de nosso problema simplesmente retirando $(\overrightarrow{s,t})$ de D. Por outro lado, se o problema não tiver solução para o grafo não direcionado $(V, E \cup \{(s,t)\})$ então necessariamente ele não tem solução para (V,E). Então esta suposição é válida e portanto o grafo de entrada será $G(V, E \cup \{(s,t)\}$.

O algoritmo descrito em SZWARCFITER, PERSIANO & OLIVEIRA[70] é uma aplicação da técnica de Busca em Profundidade e faz uso da função lowpt. Inicialmente encontramos uma árvore geradora de BP: $T(V, E_T)$ de G, enraizado em s e contendo a aresta (s,t). Depois computamos o valor de lowpt(v) para ca

da $v \in V$. O processo de direcionar as arestas é simples: Para direcionar as arestas de árvore, nós percorremos a árvore T de sua raiz s, até suas folhas. Inicialmente a aresta (s,t) é direcionada como $\overrightarrow{(s,t)}$. Quando visitamos um vértice v, direcionamos as arestas de árvore incidentes a v e que ainda não foram direcionadas. Suponha que estamos visitando o vértice $v \neq s$. A essa altura todas as arestas no caminho de v para s em T já foram direcionadas. Desejamos direcionar a aresta $(v,w) \in E_T$, onde w é um filho de v. Seja p = lowpt(w) e q um filho de p, no caminho de p para w em T. Ver Figura (VI.1) abaixo. A direção a ser dada para a aresta (v,w) depende da direção da aresta (p,q) suposta já dada, como se segue:

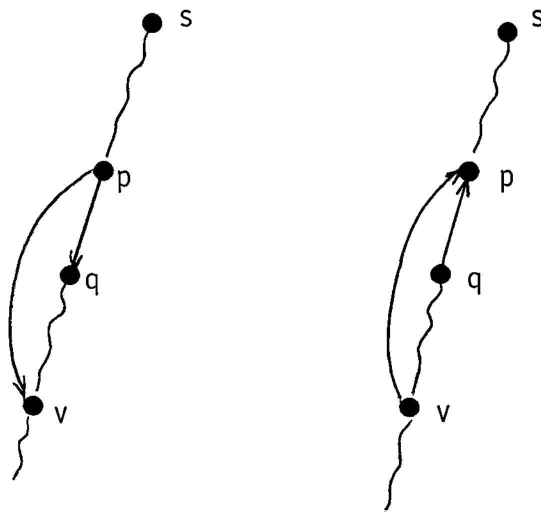
Regra 1 (arestas de árvore): $\overrightarrow{(w,v)} \iff \overrightarrow{(p,q)}$



Figura(VI.1): Direcionando arestas de árvore.

Agora iremos <u>direcionar as arestas de retorno</u>, tomadas em qualquer ordem. Suponha que (p,v) é uma aresta de retorno e q o filho de p, no caminho de p para v em T. Ver figura (VI.2). A direção a ser dada à aresta de retorno (p,v) depende da direção da aresta de árvore (p,q) suposta já dada, como se segue:

Regra 2 (arestas de retorno) : $(\overrightarrow{p,v}) \Longleftrightarrow (\overrightarrow{p,q})$



Figura(VI.2): Direcionando arestas de retorno.

O algoritmo descrito falha se na regra 1, lowpt(w)=v ou lowpt(w) = w. Mas neste caso v é um vértice de articulação e o problema não tem solução (Ver prova em SZWARCFITER, PERSIANO & OLIVEIRA[70]).

Para calcular a complexidade do algoritmo, primeiro verificamos que encontrar a árvore T e computar a função lowpt,

pode ser feito em tempo $O(|V| + |E|)$, usando o algoritmo de biconectividade de Tarjan (Ver TARJAN[71]). Para implementar as regras 1 e 2, tudo que precisamos conhecer é a direção das arestas de árvore, já visitadas. Observe então que podemos obter mais facilmente essa informação se acrescentarmos um rótulo + ou - a cada vértice q indicando a direção de arestas (p,q) de árvore, onde p é o pai de q em T. O processo inteiro pode portanto ser completado num tempo $O(|V| + |E|)$.

Para confirmar a validade do algoritmo aqui descrito, será necessário provar essencialmente os seguinte fatos:

(1) Se D é um digrafo que pode ser construido a partir de seu grafo subjacente $G(V, E \cup \{(s,t)\})$ pela aplicação das regras 1 e 2, então D é acíclico, com uma única fonte s e um único sumidouro t.

(2) Se o grafo não direcionado $G(V, E \cup \{(s,t)\})$ não é biconexo então não existe orientação acíclica para G, tendo uma única fonte s e um único sumidouro t (ou seja, existe uma orientação acíclica de G tendo uma única fonte s e um único sumidouro t, se e somente se o grafo $(V, E \cup \{(s,t)\})$ é biconexo).

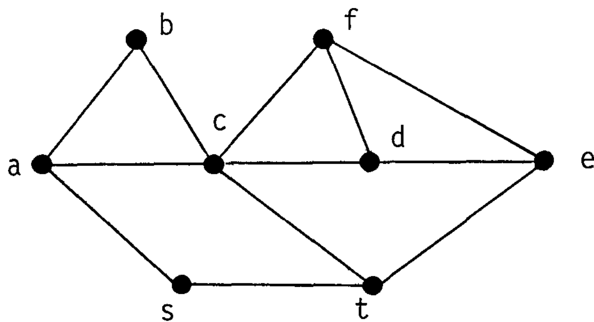
A prova de tais fatos acima estão descritas detalhadamente em SZWARCFITER, PERSIANO & OLIVEIRA[70].

OBS.: LOVÁSZ apresenta também um algoritmo para resolver o problema aqui descrito, que tem como complexidade $O(|V| \cdot |E|)$. (Ver LOVÁSZ[50]).

25.4) EXEMPLO

Dado o grafo não direcionado G(V,E) e um par de vértices distintos s, t $\in$ V, encontrar uma orientação acíclica de G.

Grado Não-Direcionado G



Estrutura de Adjacência:

s → t, a

t → s, c, e

a → s, b, c

b → a, c

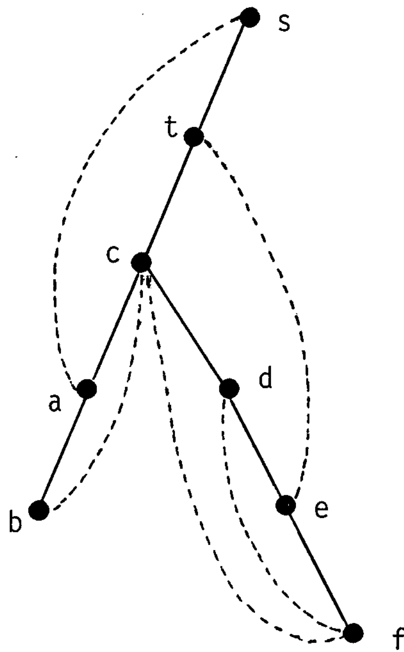
c → a, b, d, f, t

d → c, e, f

e → d, f, t

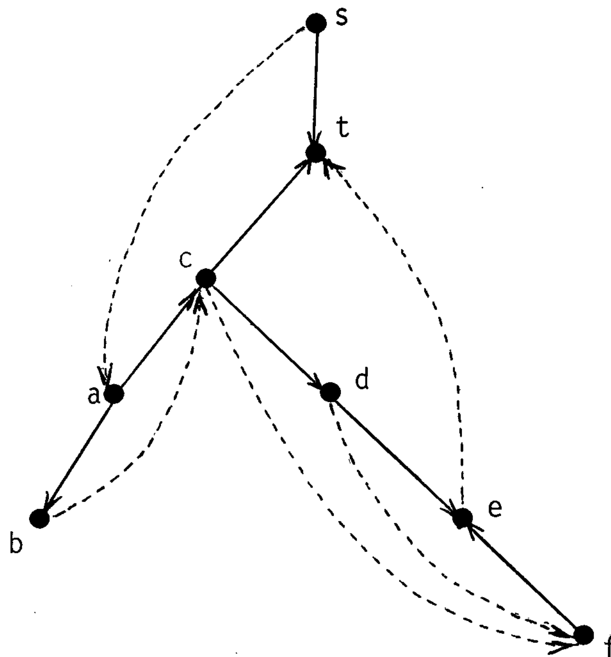
f → c, d, e

Estrutura de Busca em Profundidade:



vértice w	p=lowpt(w)
s	s
t	s
a	s
b	c
c	s
d	t
e	t
f	c

Agora iremos direcionar as arestas de árvore e as arestas de retorno segundo as REGRAS 1 e 2, respectivamente, e iremos obter a orientação acíclica D de G.



Orientação Acíclica D de G.

26) APLICAÇÃO - ORDENAÇÃO DE ELIMINAÇÃO MINIMAL

26.1) REFERÊNCIA - ROSE, TARJAN & LEUKER[62].

26.1.1) OUTRA REFERÊNCIA - OHTSUKI[54]

26.2) DESCRIÇÃO DO PROBLEMA

Encontrar uma ordenação de eliminação minimal de um grafo G.

Seja G = (V,E) um grafo não direcionado, conexo, tal que $|V| = n$.

Uma ordenação de V é uma bijeção $\alpha : \{1,2,\ldots, n\} \leftrightarrow V$. Algumas vezes denotaremos uma ordenação por $V = \{x_i\}_{i=1}^{n}$.

$G_\alpha = (V,E,\alpha)$ é um grafo ordenado. Em G_α, o conjunto de vértices monotonamente adjacente a um vértice v é definido por:

$$madj(v) = adj(v) \cap \{w \in V \mid \alpha^{-1}(v) < \alpha^{-1}(w)\}$$

Para um vértice v, a deficiência D(v) é o conjunto de arestas definido por:

$$D(v) = \{(x,y) \mid x \in Adj(v),\ y \in Adj(v),\ y \notin Adj(x),\ x \neq y\}$$

O grafo G_v obtido de G da seguinte forma: (i) adicionando arestas de modo que todos os vértice em Adj(v) são pares adjacentes, e (ii) retirando v e suas arestas indicentes;

é dito grafo de v - eliminação de G. Isto é:

$$G_v = (V - \{v\},\ E(V - \{v\}) \cup D(v))$$

Para um grafo ordenado $G_\alpha = (V, E, \alpha)$, o processo de eliminação: $P(G_\alpha) = [G = G_0, G_1, G_2, \ldots, G_{n-1}]$ é a sequência de grafos de eliminação definidos recursivamente por $G_0 = G$, $G_i = (G_{i-1})x_i$, para $i = 1,2, \ldots, n-1$. Se $G_i = (V_i, E_i)$ para $i = 0,1, \ldots, n-1$, o conteúdo $F(G_\alpha)$ é definido por:

$$F(G_\alpha) = \bigcup_{i=1}^{n-1} \tau_i ,$$

onde $\tau_i = D(x_i)$ em G_{i-1}, e o grafo de eliminação G^*_α é definido por $G^*_\alpha = (V, E \cup F(G_\alpha))$.

Uma ordenação α é uma ordenação de eliminação minimal ou ordenação minimal de G se nenhuma outra ordenação β satisfaz: $F(G_\beta) \subset F(G_\alpha)$.

Uma ordenação α é uma ordenação de eliminação mínima de G se nenhuma outra ordenação β satisfaz: $|F(G_\beta)| < |F(G_\alpha)|$

OBS.: Qualquer ordenação mínima de um grafo é minimal.

26.3) MÉTODO - BUSCA EM LARGURA LEXICOGRÁFICA

O problema de encontrar uma ordenação de eliminação mínima é NP-completo. (Ver ROSE, TARJAN & LEUKER[62]). Portanto, estamos mais interessados em encontrar uma ordenação de eliminação minimal.

Em OHTSUKI[54], Ohtsuki, obtém um algoritmo com $O(|V| \cdot |E|)$, que encontra ordenações minimais.

Usaremos aqui, um esquema de ordenação lexicográfica (que é um tipo especial de <u>busca em largura</u>) para encontrar ordenações minimais (Ver ROSE, TARJAN & LEUKER[62]). Os vértices do grafo são numerados de n a 1. Durante a busca, cada vértice v tem um <u>rótulo</u> associado consistindo de um conjunto de números selecionados de {1,2,...,n}, ordenados em ordem decrescente. Dado dois rótulos: $L_1 = [p_1, p_2, \ldots, p_k]$ e $L_2 = [q_1, q_2, \ldots, q_\ell]$, definimos $L_1 < L_2$ se, para algum j, $p_i = q_i$ para $i = 1, 2, \ldots, j-1$ e $p_j < q_j$, ou se $p_i = q_i$ para $i = 1,2,\ldots, k$ e $k < \ell$. $L_1 = L_2$ se $k = \ell$ e $p_i = q_i$, $1 \le i \le k$.

<u>Algoritmo</u> LEX M: Considere o seguinte esquema de ordenação:

<u>Início</u>

<u>Para</u> todo vértice $v \in V$, faça rótulo (v) = ϕ;

<u>Para</u> i = n <u>adicione</u> -1 até 1 <u>Faça</u>

<u>Início</u>

<u>Seleção</u>: Escolha um vértice não numerado v com maior rótulo;

<u>comentário</u> determine v, o número de i;

$\alpha(i) := v$;

<u>Atualização</u>: <u>Para</u> cada vértice não numerado w, tal que exista um caminho $[v = v_1, v_2, v_3, \ldots, v_{k+1} = w]$ com v_j não numerado e rótulo $(v_j) <$ rótulo(w), para j=2,3,...,k <u>Faça</u> adicione i a rótulo(w),

<u>Fim</u>

<u>Fim</u> LEX M.

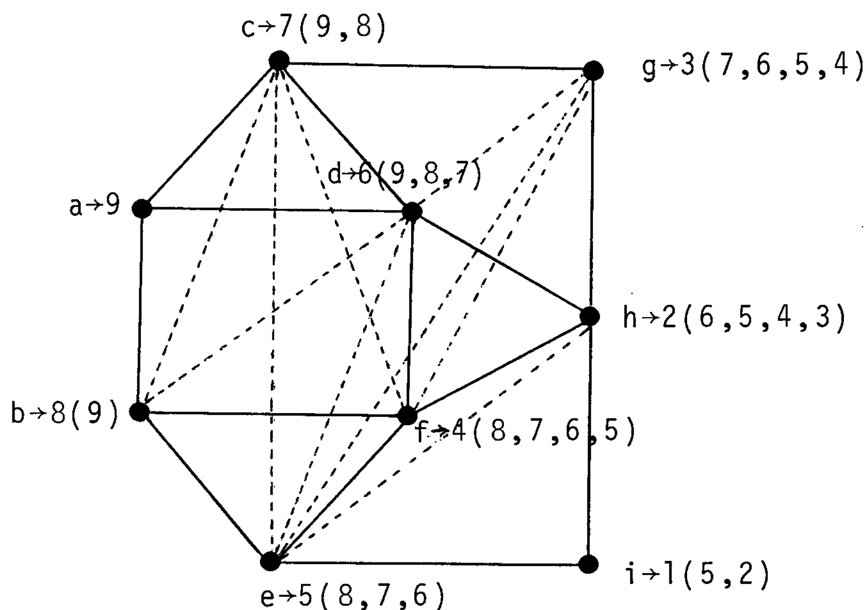
Este algoritmo constrói uma ordenação α para um grafo inicialmente não-ordenado $G = (V,E)$ e constrói um rótulo $L(v)$, dado pelo valor final de rótulo (v), para cada $v \in V$.

Em ROSE, TARJAN & LEUKER[62] é apresentada uma implementação de LEX M com complexidade $O(|V| \cdot |E|)$

<u>OBS</u>.: A condição em <u>Atualização</u>, para atualizar rótulos, é necessária por causa da existência de arestas fill-in.

26.4) <u>EXEMPLO</u>

Temos, abaixo, uma ordenação minimal de um grafo G, gerada pelo algoritmo LEX M:



Os rótulos finais estão entre parênteses, e nove arestas fill-in estão tracejadas.

V	NO	RÓTULO
a	9	ϕ
b	8	{9}
c	7	{9,8}
d	6	{9,8,7}
e	5	{8,7,6}
f	4	{8,7,6,5}
g	3	{7,6,5,4}
h	2	{6,5,4,3}
i	1	{5,2}

Logo, a ordenação minimal será:

$\alpha = [i,h,g,f,e,d,c,b,a]$

OBS.: Esta ordenação α não é mínima, pois existe uma outra ordenação com somente cinco arestas fill in.

27) APLICAÇÃO - RECONHECIMENTO DE GRAFOS TRIANGULARES

27.1) REFERÊNCIA - ROSE, TARJAN & LEUKER[62]

27.1.1) OUTRAS REFERÊNCIAS - DIRAC[20]; LEKKERKERKER & BOLAND[49]; GOLUMBIC[33]; TARJAN[76]; FULKERSON & GROSS[28].

27.2) DESCRIÇÃO DO PROBLEMA

Reconhecer se um grafo não direcionado G é um grafo triangular.

Um grafo não direcionado G = (V,E) é triangular se todo ciclo C de comprimento estritamente maior que 3 possui

uma corda, isto é, uma aresta unindo dois vértices não consecutivos do ciclo. Equivalentemente, G não contém um subgrafo induzido isomorfo a C_n para $n > 3$. Ser triangular é uma propriedade hereditária inerente em todos os subgrafos induzidos de G.

Um vértice x de G é dito simplicial se seu conjunto de adjacência Adj(x) induz um subgrafo completo de G, isto é, Adj(x) é uma clique (não necessariamente maximal).

OBS.: Dirac (Ref. DIRAC[20]) e mais tarde Lekkerkerker e Boland (Ref. LEKKERKERKER & BOLAND[49]), provaram que um grafo triangular sempre tem um vértice simplicial (de fato, pelo menos dois deles).

Seja G = (V,E) um grafo não direcionado e seja $\tau=\{v_1, v_2, \ldots, v_n\}$ uma ordenação dos vértices. Diremos que τ é uma ordenação de eliminação perfeita ou ordenação perfeita se cada v_i é um vértice simplicial do subgrafo induzido $G\{v_i, \ldots, v_n\}$. Em outras palavras, cada conjunto:

$$X_i = \{v_j \in Adj(v_i) \mid j > i\}$$

é completo. (Ver ROSE, TARJAN & LEUKER[62])

Um subconjunto $S \subset V$ é um separador de vértice para vértices não adjacentes a e b (ou separador de a-b) se a remoção de S do grafo separa a e b em componentes conexos distintos. Se nenhum subconjunto de S é um separador de a-b, então S é um separador mínimo de vértice para a e b.

<u>Resultado 1</u>: Seja G um grafo não direcionado. As seguintes afirmações são equivalentes:

i) G é triangular;

ii) G tem uma ordenação perfeita.
Além disso, qualquer vértice simplicial pode iniciar uma ordenação perfeita.

iii) Todo separador mínimo de vértice produz um grafo completo de G.

<u>Resultado 2</u>: Todo grafo triangular G = (V,E) tem um vértice simplicial. Além disso, se G não é uma clique, então ele tem dois vértices simpliciais não adjacentes. (Ver prova em GOLUMBIC[33]).

27.3) <u>MÉTODO - BUSCA EM LARGURA LEXICOGRÁFICA</u>

Do Resultado 2 acima, Fulkerson e Gross (Ref. FULKERSON & GROSS[28]) obtêm um procedimento de reconhecimento que fornece-nos uma escolha de pelo menos dois vértices para cada posição na construção de uma ordenação perfeita para um grafo triangular. Portanto, nós podemos aleatoriamente escolher um vértice v_n e guardá-lo na última posição de uma ordenação perfeita.

Similarmente podemos escolher qualquer vértice v_{n-1} adjacente a v_n e guardá-lo na (n-1)-ésima posição. Se continuarmos desta maneira, será construido um esquema de backwards.

Iremos utilizar a Busca em Largura Lexicográfica (BLL) para reconhecer grafos triangulares. Temos então o seguinte resultado:

Resultado 3: Um grafo não direcionado G = (V,E) é triangular se e somente se a ordenação τ produzida pelo Algoritmo BLL é uma ordenação perfeita (Ver prova em GOLUMBIC[33]).

OBS.: 1) Em um trabalho não publicado, TARJAN (Ref. TARJAN[76]) apresenta um outro método de explorar um grafo, que pode ser usado para reconhecer grafos triangulares. Ele é denominado "Busca de Cardinalidade Máxima (BCM)".

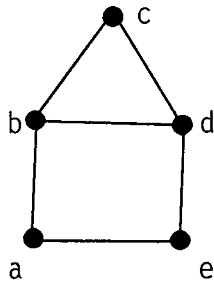
2) O algoritmo de Busca em Largura Lexicográfica que constrói uma ordenação perfeita é parecido com o que encontra ordenação minimal (já visto na Aplicação 26 anterior). A única diferença é que na Atualização da ordenação perfeita teremos:

Atualização: Para cada vértice não numerado $w \in Adj(v)$ Faça Adicione i a rótulo(w).

27.4) EXEMPLO

Verificar se os grafos não direcionados abaixo são triangulares:

1) Grafo G



Estrutura de Adjacência:

a → b, e

b → a, d, c

c → b, d

d → c, b, e

e → a, d

Aplicaremos o algoritmo BLL para o grafo G acima. O vértice a é selecionado arbitrariamente.

A evolução da rotulação e a numeração são ilustradas abaixo:

	ROTULO	Nº
a	ϕ	-
b	ϕ	-
c	ϕ	-
d	ϕ	-
e	ϕ	-

→

	ROTULO	Nº
a	ϕ	5
b	{5}	-
c	ϕ	-
d	ϕ	-
e	{5}	-

→

	ROTULO	Nº
a	ϕ	5
b	{5}	4
c	{4}	-
d	{4}	-
e	{5}	-

→

	ROTULO	Nº
a	ϕ	5
b	{5}	4
c	{4}	-
d	{4,3}	-
e	{5}	3

→

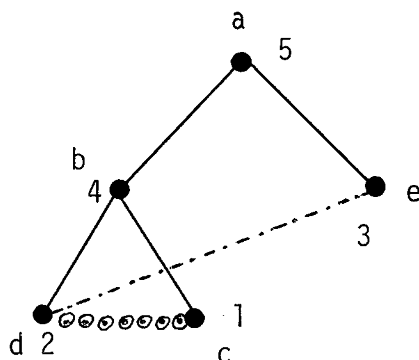
	ROTULO	Nº
a	ϕ	5
b	{5}	4
c	{4,2}	-
d	{4,3}	2
e	{5}	3

→

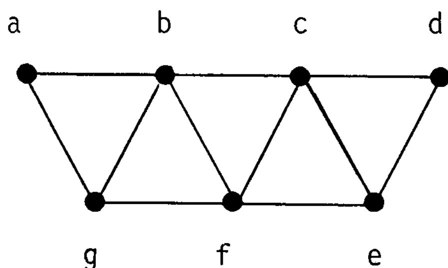
	ROTULO	Nº
a	ϕ	5
b	{5}	4
c	{4,2}	1
d	{4,3}	2
e	{5}	3

Note que a numeração final $\tau = [c,d,e,b,a]$ é uma ordenação perfeita e assim G é um grafo triangular.

Estrutura de Busca em Largura:



2) Grafo M



Estrutura de Adjacência:

a → b, g

b → c, f, g, a

c → d, e, f, b

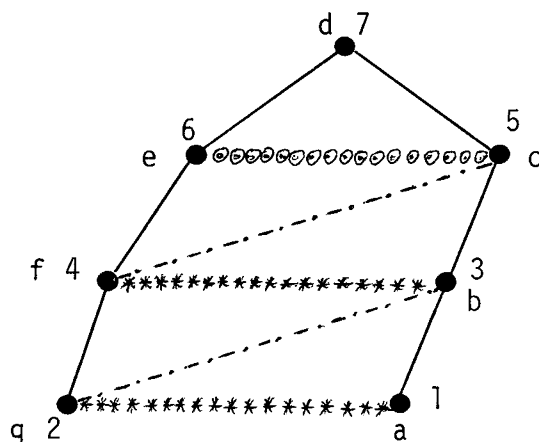
d → e, c

e → d, c, f

f → e, c, b, g

g → f, b, a

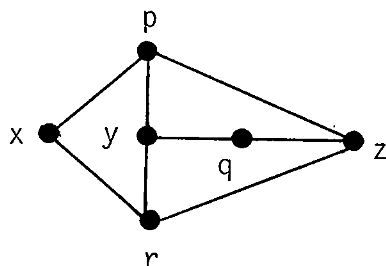
Estrutura de Busca em Largura:



A numeração final é τ = (a,g,b,f,c,e,d) que é uma ordenação perfeita, logo M é triangular.
Note que τ não é única. De fato, M tem 96 diferentes ordenações perfeitas.

OBS.: Em M, todo separador de vértice mínimo tem cardinalidade 2.

3) Grafo N



Estrutura de Adjacência:

x → p, r
y → p, q, r
z → p, q, r
p → x, y, z
q → y, z
r → x, y, z

Podemos ver claramente que N não tem nenhum vértice simplicial, assim não podemos nem começar a construir uma ordenação perfeita (não tem nenhuma).
Logo, N não é triangular: OBS.: Em N, o conjunto {y,z} é um separador de vértice mínimo para p e q, enquanto {x,y,z} é separador de vértice mínimo para p e r.

28) APLICAÇÃO - RECONHECIMENTO DE GRAFOS TOTAIS

28.1) REFERÊNCIA - GAVRIL[32]

28.1.1) OUTRA REFERÊNCIA - BEHZAD[6]

28.2) DESCRIÇÃO DO PROBLEMA

Reconhecer se um grafo conexo H é total.

Seja H = (V,E) um grafo conexo. O número de vértices de H será denotado por $|V| = n$, e o número de arestas por $|E| = m$.

Um conjunto completamente conexo (conjunto c.c) de H é um conjunto de vértices em que cada dois vértices são adjacentes.

Seja Adj(v) o conjunto de vértices adjacentes a um vértice v em H, e grau (v) = |Adj(v)| o grau de v.

Seja $Adj^{+}(v) = Adj(v) + \{v\}$, e max grau (H) será denotado como o grau máximo entre os vértices de H.

Um grafo H é chamado total se existe um grafo G de modo que há uma correspondência um a um entre os vértices de H e os vértices e arestas de G, tal que dois vértices de H são adjacentes se e somente se os vértices correspondentes em G são adjacentes ou incidentes. Neste caso diremos que H é o grafo total de G.

Para um vértice v de H, denotaremos por $\bar{v}$ seu elemento correspondente em G, e v será chamado um u-vértice se $\bar{v}$ é um vértice de G, e v será chamado um a-vértice se $\bar{v}$ é uma aresta de G. É fácil ver que para todo u-vértice v de H, grau(v) é exatamente duas vezes o grau de $\bar{v}$ em G e para todo a-vérti w de H, grau(w) é a soma dos graus dos vértices de G adjacentes à aresta $\bar{w}$.

Um caminho ou um ciclo serão considerados como um caminho simples e um ciclo simples. Um caminho (um ciclo) de um grafo será chamado sem corda se no grafo não existem arestas ligando dois vértices não consecutivos do caminho (do ciclo).

28.3) MÉTODO - BUSCA EM LARGURA

Em BEHZAD[6], Behzad descreve, de modo geral, um algoritmo interessante para o reconhecimento de grafos totais. A ref. GAVRIL[32] descreve uma versão do algoritmo de Behzad e que requer $O(|E|)$ passos e utilizará a técnica de Busca em Largura.

Assumiremos que o grafo $H = (V,E)$, a ser testado, é conexo e tem mais do que 3 vértices.

Seja $V = \{v_1, v_2, \ldots, v_n\}$.

Assumiremos que $H(V)$ é dado pelas listas de adjacência de seus vértices.

Considere um grafo H, e um subconjunto $U \subseteq V$. Podemos verificar se H(V) é o grafo total de H(U) da seguinte maneira:

Seja $U = \{u_1, u_2, \ldots, u_p\}$ e $A = V - U = \{a_1, a_2, \ldots, a_t\}$

Para todo $a_j \in A$ será determinado um conjunto $L(a_j)$. Inicialmente $L(a_j) = \phi$, para todo $1 \leq j \leq t$. Para todo $u_i \in U$ adicionamos u_i ao conjunto $L(a_j)$ de todo $a_j \in Adj(u_i) \cap A$.

Seja $L = \{L(a_1), \ldots, L(a_t)\}$

Lema: H é o grafo total de H(U) se e somente se as seguintes condições são satisfeitas:

i) Todo $L(a_j) \in L$ tem exatamente dois elementos;

ii) Para cada dois elementos adjacentes a_j, $a_k \in A$, temos que $|L(a_j) \cap L(a_k)| = 1$;

iii) Para todo $u_i \in U$, o conjunto $Adj(u_i) \cap A$ é completamente conexo (c.c);

iv) O número de arestas de $H(U)$ é igual a $|A|$;

v) Para todo $L(a_j) \in L$, os dois elementos de $L(a_j)$ são adjacentes em H.

(O processo de i a v requer $O(|E|)$ passos) (Ver GAVRIL[32]).

Boa Configuração

Seja H o grafo total de um grafo G tal que max grau (H) > 4. Adj(v) pode ser particionado em dois subconjuntos A_v e B_v, sendo A_v o conjunto de u-vértices e B_v o conjunto de a-vértices de Adj(v). Claramente, B_v é c.c, $|A_v| = |B_v|$ = grau (v)/2, e cada elemento de $B_v(A_v)$ é adjacente a exatamente um elemento de $A_v(B_v)$.

Adj(w) pode ser particionado em dois subconjuntos A_w e B_w, sendo que A_w representa um u-vértice e seus a-vértices adjacentes, e B_w representa o outro u-vértice e seus a-vértices adjacentes. Neste caso ambos: A_w e B_w são completamente conexos(A_w, B_w não tem que ser do mesmo tamanho), e cada elemento de $B_w(A_w)$ é adjacente a no máximo um elemento $A_w(B_w)$ e cada vértice $s \in V - Adj^+(w)$ é adjacente a no máximo quatro elementos de Adj(w).

Para um vértice v de H diremos que H(Adj(v)) tem uma boa configuração se H(Adj(v)) é conexo e Adj(v) pode ser particionado em dois subconjuntos A_v, B_v tal que:

i) B_v é completamente conexo e $|B_v| \geq grau(v)/2$;

ii) Cada elemento de $B_v(A_v)$ é adjacente a no máximo um elemento de $A_v(B_v)$;

iii) Se A_v não é completamente conexo então cada elemento de $A_v(B_v)$ é adjacente a exatamente um elemento de $B_v(A_v)$.

É fácil ver que se grau(v) > 4 então a partição em A_v, B_v é única exceto para uma possível troca entre A_v e B_v quando $|A_v| = |B_v|$. Assim, se um grafo H(V) é total, então para cada $v \in V$, H(Adj(v)) tem uma boa configuração.

Algoritmo

Considere um grafo conexo H(V). Em linhas gerais, o algoritmo faz o seguinte: Primeiro trata das situações em que H pode ser o grafo total de um caminho sem corda, um ciclo sem corda ou uma clique. Se H não é nenhum desses casos, o algoritmo trata da situação restante (que será quando max grau(H) > 4). Em cada caso tentamos encontrar um a-vértice v_1 e dois u-vértices v_2, v_3 adjacentes a v_1 e aplicamos a técnica de Busca em Largura (Ver GAVRIL[32]) → BL(H, v_1, v_2, v_3, U). Então verificamos que H é o grafo total de H(U).

O algoritmo para testar se o grafo H(V) é ou não total funciona da seguinte maneira:

1. Se max grau(H) $\leq$ 4 então H somente pode ser o grafo total de um caminho sem corda ou de um ciclo sem corda.

1.a. Seja H o grafo total de um caminho sem corda. Então deve haver um vértice v_1 de grau 2 que é um u-vértice, um vértice v_2 de grau 3 adjacente a v_1 que é um a-vértice e um vértice v_3 de grau 4 adjacente a v_1, v_2, que é um u-vértice. Em seguida, testamos que um grafo H tendo max grau(H) ≤ 4 é o grafo total de um caminho sem corda como se segue:
Verificamos se em H existe um vértice v_1 de grau 2, um vértice v_2 adjacente a v_1, de grau 3 e um vértice v_3 adjacente a v_1, v_2 de grau 4. Se v_1, v_2, v_3 não existem então H não é o grafo total de um caminho sem corda. Caso contrário, executamos uma BL(H,v_2,v_1,v_3,U) e verificamos se H é ou não o grafo total de H(U). Tudo isto requer $O(|E|)$ passos.

1.b. Seja H o grafo total de um ciclo sem corda. Então, todos os seus vértices são de grau 4. Seja U o conjunto de seus u-vértices e A o conjunto de seus a-vértices. É fácil ver que H é também o grafo total de H(A), e de fato podemos trocar o conjunto de u-vértices com o conjunto de a-vértices. Assim podemos considerar qualquer vértice v_1 de H um a-vértice e então, dois vértices adjacentes v_2, v_3, ambos adjacentes a v_1, serão u-vértices. Portanto, testamos se um grafo H tendo max grau

(H) $\leq$ 4 é o grafo total de um ciclo sem corda como se segue:

Primeiro verificamos se todos os vértices de H são de grau 4. Então procuramos três vértices v_1, v_2, v_3, mutuamente adjacentes. Se não existem tais vértices, então H não é o grafo total de um ciclo sem corda. Caso contrário, executamos uma BL(H, v_1, v_2, v_3, U) e verificamos se H é ou não o grafo total de H(U). Tudo isto requer $O(|E|)$ passos.

2. Se max grau(H) > 4, seja v um vértice de H tendo grau máximo. Se H(Adj(v)) não tem uma boa configuração então H não é total. Caso contrário, seja A_v, B_v a boa configuração de H(Adj(v)). Se H é o grafo total de H($B_v \cup \{v\}$) então nós terminamos. Caso contrário nós sabemos que H não é o grafo total de uma clique, e temos os seguinte casos:

   i) Se A_v não é completamente conexo, então v deve ser um u-vértice. Tomamos um vértice $v_1 \in B_v$ e o vértice $v_2 \in A_v$ adjacente a v_1. Assim V_1 deve ser um a-vértice e v, v_2 devem ser seus dois u-vértices adjacentes. Executamos uma BL(H, v_1, v, v_2, U). Então, H é total sss H é o grafo total de H(U).

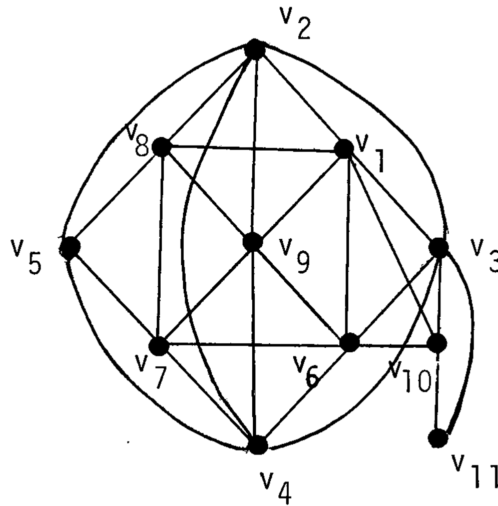
ii) Se A_v é completamente conexo, então v deve ser um a-vértice. Se existe um vértice $z \in V - Adj^+(v)$ adjacente a mais do que quatro elementos de Adj(v) então H não é total. Caso contrário, procuramos dois elementos adjacentes $w_1 \in A_v$, $w_2 \in B_v$ tal que $\{Adj(w_1) - Adj^+(v)\} \cup \{w_2\}$ e $\{Adj(w_2) - Adj^+(v)\} \cup \{w_1\}$ não são completamente conexos (isto requer $O(|E|)$ passos desde que todo $z \in V - Adj^+(v)$ é adjacente a no máximo quatro elementos de Adj(v)). Se w_1, w_2 não existem, então H não é total. Caso contrário, w_1, w_2 devem ser dois u-vértices adjacentes a v. Executamos uma BL(H,v,w_1,w_2,U). Então H é total sss H é o grafo total de H(U).

O algoritmo aqui descrito requer $O(|E|)$ passos. (Ver em GAVRIL[32]).

28.4) <u>EXEMPLO</u>

Verificar se o grafo conexo H a seguir é total.

Grafo H(V)



Estrutura de Adjacência:

$v_1 \rightarrow v_2, v_3, v_6, v_8, v_9, v_{10}$

$v_2 \rightarrow v_1, v_3, v_4, v_5, v_8, v_9$

$v_3 \rightarrow v_1, v_2, v_4, v_6, v_{10}, v_{11}$

$v_4 \rightarrow v_2, v_3, v_5, v_6, v_7, v_9$

$v_5 \rightarrow v_2, v_4, v_7, v_8$

$v_6 \rightarrow v_1, v_3, v_4, v_7, v_9, v_{10}$

$v_7 \rightarrow v_4, v_5, v_6, v_8, v_9$

$v_8 \rightarrow v_1, v_2, v_5, v_7, v_9$

$v_9 \rightarrow v_1, v_2, v_4, v_6, v_7, v_8$

$v_{10} \rightarrow v_1, v_3, v_6, v_{11}$

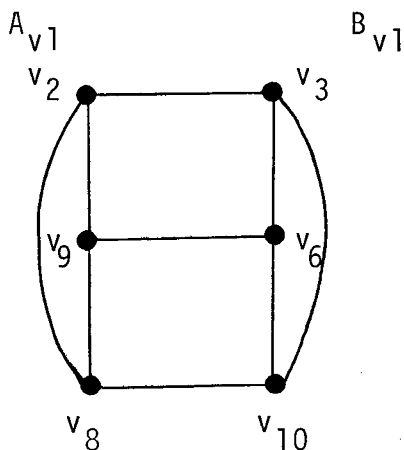
$v_{11} \rightarrow v_3, v_{10}$

Aplicando o algoritmo aqui descrito no grafo H(V),

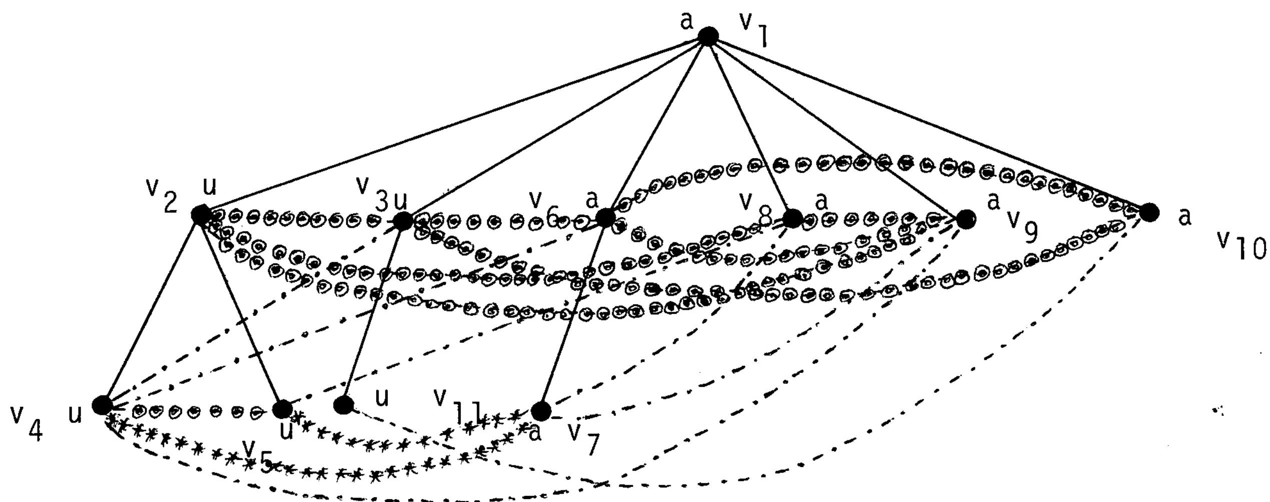
Seja $V = \{v_1, v_2, \ldots, v_{11}\}$

Neste caso max grau(H) > 4; seja v_1 o vértice de H tendo o grau máximo.

$Adj(v_1) = \{v_2, v_3, v_6, v_8, v_9, v_{10}\}$ tem uma boa configuração, que é a seguinte:



Cada elemento de $V - Adj^+(v_1) = \{v_4, v_5, v_7, v_{11}\}$ é adjacente a no máximo quatro elementos de $Adj(v_1)$ e $\{Adj(v_2) - Adj^+(v_1)\} \cup \{v_3\}$, $\{Adj(v_3) - Adj^+(v_1)\} \cup \{v_2\}$ não são completamente conexos. Logo, v_1 é um a-vértice e v_2, v_3 seus dois u-vértices. Aplicando a técnica de Busca em Largura $BL(H, v_1, v_2, v_3, U)$, (Ver procedure BL descrita em GAVRIL[32]), obtemos a seguinte estrutura de árvore BL:



——> aresta de árvore

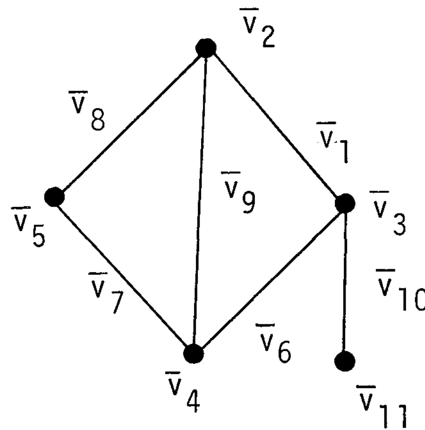
@@@@@> aresta entre irmãos

****> aresta entre primos

-.-.> aresta entre tio e sobrinho

e $U = \{v_2, v_3, v_4, v_5, v_{11}\}$, obtido quando aplicamos $BL(H, v_1, v_2, v_3, U)$.

E H(U) será:



Podemos ver claramente, pela definição ou pelo Lema, que H é grafo total de H(U), logo concluimos que H é total.

29) APLICAÇÃO - VÉRTICE DE REALIMANETAÇÃO

29.1) REFERÊNCIA - GAREY & TARJAN[31]

29.1.1) OUTRAS REFERÊNCIAS - TARJAN[71]; TARJAN[72]; AHO[1]; KNUTH[47]

29.2) DESCRIÇÃO DO PROBLEMA

Encontrar todos os vértices de realimentação em um digrafo arbitrário fortemente conexo.

Seja D = (V,E) um grafo direcionado com n = |V| vértices e m = |E| arestas.

Um <u>vértice de realimentação</u> de D é um vértice contido em todo ciclo de D.

Um <u>conjunto de realimentação</u> de D é um conjunto $S \subseteq V$ tal que todo ciclo de D contém pelo menos um vértice em S.

<u>OBS</u>.: O problema de determinar um conjunto de realimentação de cardinalidade mínima aparece em vários contextos, mas, infelizmente, é classificado com NP-difícil.

29.3) <u>MÉTODO - BUSCA EM PROFUNDIDADE</u>

Em GAREY & TARJAN[31] é apresentado um algoritmo que tem como complexidade $O(|E|)$ tempo e que usa busca em profundidade para encontrar todos os vértices de realimentação em um digrafo arbitrário fortemente conexo. Se o grafo não é fortemente conexo, podemos resolver o problema aqui descrito encontrando os componentes fortemente conexos usando o algoritmo da Ref. TARJAN[71], e achando conjuntos de realimentação separadamente em cada componente.

Suponha que os vértices de D são numerados de 1 a n conforme eles são alcançados durante a busca em profundidade. Assumiremos que cada vértice de D é identificado por seu número.

Seja <u>s</u> o menor vértice sem nenhuma aresta de árvore saindo dele. Então (1, 2, 3, ..., s-1, s) é um caminho da ár-

vore (isto é, caminho na árvore geradora) de 1 a s. Como D é fortemente conexo, deve existir alguma aresta (s,t) deixando s em D, e como esta aresta não foi incluida na árvore, nós devemos ter $1 \leq t \leq s-1$. Assim (s,t) é uma aresta de retorno, e os únicos vértices de realimentação possíveis são aqueles no conjunto $\{t, t+1, \ldots s-1, s\}$.

Podemos aperfeiçoar este conjunto de possibilidades como se segue:

Seja y o maior vértice com uma aresta de retorno convergindo nele, isto é:

$$y = \max \{v: \text{existe uma aresta de retorno } (u,v)\}$$

Seja z o maior vértice que é um ancestral de todos os vértices que têm arestas de retorno, isto é:

$$z = \max \{w: \text{para cada aresta de retorno } (u,v), w \text{ é um ancestral de } u\}$$

Note que z deve ser um ancestral de s e assim $z \leq s$.

Como cada aresta de retorno (u,v) define um ciclo em D consistindo de (u,v) e o caminho na árvore de v para u, e como qualquer vértice de realimentação deve depender de todos esses ciclos, <u>todo vértice de realimentação deve ser um descendente de y e um ancestral de z</u>. Segue que os únicos possíveis vértices de realimentação são aqueles pertencentes ao conjunto $\{v: y \leq v \leq z\}$. Se $y > z$ ou se o subgrafo de D, obtido retirando todos esses vértices, não é acíclico, então não existem vértices de realimentação.

A fim de determinar quais destes candidatos são vértices de realimentação, nós computamos dois valores para cada

vértice $v \in V$. Eles são definidos como se segue:

maxi(v) = max {$w \neq v$: $w \leq z$ e existe um caminho de v para w que não usa arestas de retorno e que passa através de somente vértices u satisfazendo $u > z$}

<u>OBS</u>.: Um caminho p é dito passar através de u se u pertence a p e u não é um vértice final de p.

loop(v) =
- verdade, se existe um caminho de v para algum descendente w de z que não usa arestas de retorno e que passa através de somente vértices u satisfazendo $u > z$;
- falso, caso contrário.

Por convenção: $\max(\phi) = 0$

A busca em profundidade mantém uma pilha de todos os vértices encontrados até aqui, que têm arestas divergentes que ainda não foram examinadas. Os vértices são removidos desta pilha em ordem topológica invertida (Ver KNUTH[47]), com respeito ao grafo acíclico, obtido retirando todas as arestas de retorno. Se (u,v) não é uma aresta de retorno, então v é desempilhado antes de u durante a busca. Assim, se z é conhecido antes da hora, as seguintes equações podem ser usadas para computar maxi e loop durante a busca, vértice por vértice na or-

dem de desempilhar:

maxi(v) = max({w: w $\leq$ z e (v,w) não é uma aresta de retorno} $\cup$ {maxi(w): w > z e (v, w) não é uma aresta de retorno}).

loop(v) = verdade, se e somente se ou v é um descendente de z ou não existe uma aresta de retorno (v,w) tal que w > z e loop(w) = verdade.

O teorema abaixo caracteriza os vértices de realimentação.

<u>Teorema</u>: Um vértice v $\in$ V é um vértice de <u>realimentação</u> se e somente se:

i) y $\leq$ v $\leq$ z;

ii) O conjunto {x: y $\leq$ x $\leq$ z} é um conjunto de realimentação;

iii) maxi(u) $\leq$ v para 1 $\leq$ u < v;

iv) loop(u) = falso para y $\leq$ u < v.

Este teorema sugere o seguinte algoritmo para encontrar todos os vértices de realimentação em um grafo fortemente conexo D.

Algoritmo FB

Passo 1: Execute uma busca em profundidade em D, numerando os vértices de 1 a n na ordem de busca e computando y e z;

Passo 2: Testar se $y \leq z$ e se D torna-se acíclico quando todos os vértices x satisfazendo $y \leq x \leq z$ são retirados. Se o teste falhar, pare (D não tem vértices de realimentação).

Passo 3: Repita a busca em profundidade, computando maxi(v) e loop(v) para todos os vértices $v \in V$ usando as equações já dadas.

Passo 4: Se y = raiz então maxitest = 0. Senão maxitest $\leftarrow \max\{maxi(u): 1 \leq u < y\}$. Inicialize o conjunto de vértices de realimentação como conjunto vazio, e $v \leftarrow y$. Então execute os seguintes passos:

Passo 4a: Se maxitest $\leq$ v, adicione v ao conjunto de vértices de realimentação.

Passo 4b: Faça maxitest $\leftarrow \max\{maxitest, maxi(v)\}$.

Passo 4c: Se loop(v) = falso e $v < z$, faça $v \leftarrow v+1$ e retorne ao Passo 4a. Caso contrário, pare.

Não é difícil verificar que a quantidade total de tempo requerido por este algoritmo é $O(|E|)$, onde $|E|$ é o número de arestas em D.

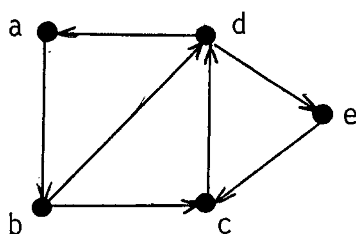
OBS.: O algoritmo acima encontra-se em GAREY & TARJAN[31]. Para que ele funcione corretamente é necessário acrescentar no Passo 4 o seguinte:

Se y = raiz então maxitest = 0.

29.4) EXEMPLO

Encontrar todos os vértices de realimentação para os seguintes grafos:

1) Grafo D



Estrutura de Adj.:

a → b

b → c, d

c → d

d → a, e

e → c

Ciclos:

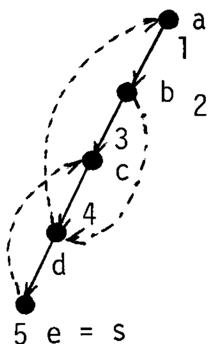
a b c d a

a b d a

c d e c

Aplicando o algoritmo FB aqui descrito:

Passo 1: Seja T a estrutura de Busca em Profundidade de D:



y = c

z = d

<u>Passo 2</u>: $\{v: y \leq v \leq z\}$ = {c,d} → possíveis vértices de realimentação. Retirando {c,d}, D torna-se acíclico.

<u>Passo 3</u>:

maxi(a) = b	loop(a) = falso
maxi(b) = d	loop(b) = falso
maxi(c) = d	loop(c) = falso
maxi(d) = 0	loop(d) = verdade
maxi(e) = 0	loop(e) = verdade

<u>Passo 4</u>: maxitest = 0

maxitest = d

Conjunto de Realimentação:

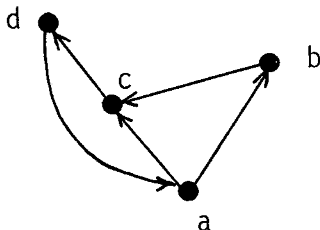
F = { } v ← c

F = {c} v ← d

F = {c,d} loop(d) = verdade

Logo, os vértices de realimentação de D são <u>c</u> e <u>d</u>.

2) Grafo L



Estrutura de Adj.:

a → b, c

b → c

c → d

d → a

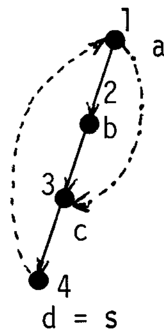
Ciclos:

a c d a

a b c d a

Aplicando o Algoritmo FB:

<u>Passo 1</u>: Seja T' a estrutura de Busca em Profundidade de L:



$y = a$

$z = d$

<u>Passo 2</u>: $\{v: y \leq v \leq z\} = \{a, b, c, d\}$ → possíveis vér tices de realimentação. Retirando {a,b,c,d} , L torna-se acíclico.

<u>Passo 3</u>:

maxi(a) = c	loop(a) = falso
maxi(b) = c	loop(b) = falso
maxi(c) = d	loop(c) = falso
maxi(d) = 0	loop(d) = falso

<u>Passo 4</u>: maxitest = 0

maxitest = c

maxitest = d

Conjunto de realimentação:

F = { }	v ← a
F = {a}	v ← b
F = {a,c}	v ← c
F = {a,c,d}	v ← d

Logo, os vértices de realimentação são a, c, d.

30) APLICAÇÃO - FATORES PAR E IMPAR

30.1.) REFERÊNCIA - EBERT[23]

30.1.1) OUTRAS REFERÊNCIAS - TARJAN[71]; HARAY[35]; DEO, KRISHNAMOORTHY & PAI[16].

30.2) DESCRIÇÃO DO PROBLEMA

Encontrar um fator ímpar (ou par) maximal em um grafo não direcionado conexo de ordem par (ou ímpar).

Seja G = (V,E) um grafo não direcionado, onde V é o conjunto de vértices e E é o conjunto de arestas.

Seja $n = |V|$ o número de vértices; e $m = |E|$ o número de arestas.

Um fator ímpar (par) F de G é um subgrafo gerador não nulo de G, onde o grau de cada vértice em F é ímpar (par).

Res.1: O número de vértices de grau ímpar é par em todo grafo (pois $2m = \Sigma$ grau(v)).

Por causa deste Res. 1 somente existem fatores ímpares em grafos com número par de vértices. O algoritmo de EBERT[23] mostra que existe pelo menos um tal fator em cada gra

fo conexo com número par de vértices.

Res. 2: Todo fator par F contém pelo menos um ciclo. (Se F não possui ciclos, ele conterá uma árvore e portanto uma folha com um grau ímpar).

OBS.: Em DEO, KRISHNAMOORTH & PAI[16] é mostrado que o fator ímpar de uma árvore com número par de vértices é único. (Eles também fornecem um algoritmo para encontrar esse tal fator).

30.3) MÉTODO - BUSCA EM PROFUNDIDADE

Usaremos a técnica de busca em Profundidade (BP) para encontrar um fator ímpar maximal (em relação ao número de arestas) em um grafo não direcionado conexo com número par de vértices. O algoritmo abaixo irá utilizar dois vetores:

NUM → é obtido à medida que os vértices são explorados e controla a execução da busca (é o número de BP).

NOVO-GRAU → controla o grau dos vértices quando o fator é encontrado.

ALGORITMO FAT-IMPAR:

```
Procedimento BP(v,u)
Início
    i ← i + 1;
    NUM(v) ← i;
    Para w ∈ Adj(v) efetuar
        Se NUM(w) = 0
          então
              Início      /(v,w) é uma aresta de árvore/
                 BP(w,v);
                 Se  NOVO-GRAU(w) é par
                    então ADIC(v,w);
                 Fim
              Senão
                Início
                Se NUM(w) ≤ NUM(v) e w ≠ u
                   então      /(v,w) é uma aresta de
                                 retorno)
                      ADIC (v,w);
                Fim
Fim
```

```
Procedimento ADIC (v,w);
   Início
         /Adicione (v,w) ao fator/
          NOVO-GRAU (v) = NOVO GRAU (v) + 1;
          NOVO-GRAU (w) = NOVO-GRAU (w) + 1;
   Fim

Procedimento Principal:
   Início
         p ← |V|;
         Se  p  é ímpar
             então     'Não existe fator ímpar'
             Senão
               Início
                 Para j = 1  Some 1 até p efetuar
                        NUM(j) = NOVO-GRAU(j) = 0;
                        i = 0;
                        BP(1,0);
               Fim
   Fim
```

O algoritmo acima descrito tem complexidade $O(|E|)$. (Ver EBERT[23]).

OBSERVAÇÃO

1) Seja $w \neq 1$ um vértice arbitrário. Quando a chamada ao procedimento BP(w , v) termina, todas as arestas de retorno terminando em w e algumas arestas de árvore começando em w , já foram adicionadas ao fator. Se NOVO-GRAU(w) não é ímpar, a aresta (v,w) é incluida no fator. O NOVO-GRAU(1) será ímpar por causa do Res. 1.

2) Se a condição, descrita no Procedimento BP(v,w) sobre a aresta de árvore: NOVO-GRAU(w) é par for substituida por: NOVO-GRAU(w) é ímpar e se o teste descrito no Procedimento Principal: p é ímpar, for removido, obteremos um algoritmo FAT-PAR que irá encontrar um fator par maximal, e também terá como complexidade $O(|E|)$.

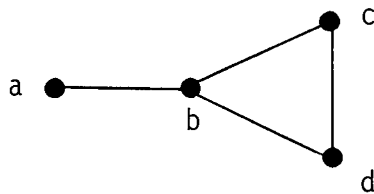
3) Se os algoritmos são aplicados para árvores, não existem arestas de retorno. A indução sobre os vértices na ordem em que as chamadas a BP(v,u) terminam, mostra que o algoritmo citado (EBERT[23]) fornece um único fator ímpar em árvores com número par de vértices. O mesmo processo de indução mostra que não existem fatores pares em qualquer árvore, que é também deduzido pelo Res. 2.
Para qualquer outro grafo conexo (exceto no caso de fatores ímpares em grafos de ordem ímpar), os

algoritmos fornecem um fator, desde que exista pelo menos uma aresta de retorno incluida. O número de arestas neste fator difere do número de arestas no grafo original por no máximo n-1.

30.4) EXEMPLO

Dado o grafo não direcionado G abaixo, encontrar um fator ímpar maximal.

Grafo G (com número par de vértices)



Estrutura de adjacência:

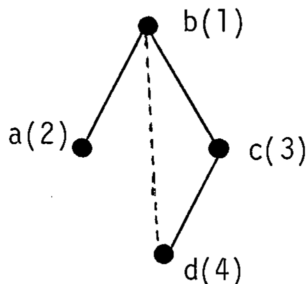
a → b

b → a, c, d

c → b, d

d → b, c

Estrutura de BP:



NUM(b) = 1

NUM(a) = 2

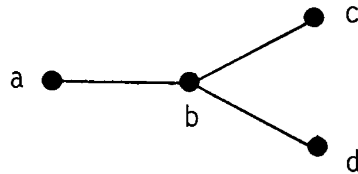
NUM(c) = 3

NUM(d) = 4

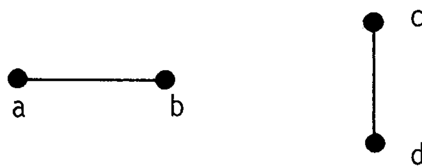
Fatores adicionais:

ADIC(b,a)	ADIC(d,b)	ADIC(b,c)
NOVO-GRAU(a) = 1	NOVO-GRAU(b) = 2	NOVO-GRAU(b) = 3
NOVO-GRAU(b) = 1	NOVO-GRAU(d) = 1	NOVO-GRAU(c) = 1

Assim, quando acabamos de aplicar o algoritmo FAT-IMPAR obtemos o seguinte fator ímpar maximal:



Um outro fator ímpar maximal que podemos obter do grafo G dado será:



OBS.: O exemplo acima mostra que o fator encontrado não é necessariamente máximo, visto que o grafo G pode ter fatores maximais com diferentes números de arestas.

31) CONCLUSÃO

Neste capítulo foram descritos alguns problemas, com suas respectivas soluções, que utilizaram um ou mais métodos de buscas em grafos.

Podemos citar, por exemplo, a técnica de busca em Profundidade usada para encontrar componentes biconexos, determinar ordenação topológica, encontrar maior caminho, determinar planaridade, etc. Outra técnica, a busca em largura, foi utili

zada para reconhecer grafos totais, reconhecer grafos bipartites, encontrar menor caminho em grafos não ponderados, etc. Outras técnicas foram também utilizadas como importantes ferramentas para resolver problemas em grafos.

Logo, podemos concluir que os métodos de busca em grafos muito auxiliam na resolução de vários problemas práticos, e geram algoritmos com eficientes complexidades de tempo e espaço.

CAPÍTULO VII

CONCLUSÃO GERAL

Esta tese abordou diversos problemas importantes em Teoria de Grafos. Muitos desses problemas requerem processos automáticos para resolvê-los, pois quase sempre conduzem a grafos grandes, ou seja, grafos que são impossíveis analisar sem a ajuda do computador. Como grafos são largamente usados como modelos de fenômenos reais, é importante descobrir algoritmos eficientes para responder questões teóricas sobre grafos.

Busca em Profundidade ou Backtracking foi uma ferramenta aqui estudada e que proporciona um algoritmo muito eficiente para resolver vários problemas. Tal algoritmo é um procedimento recursivo, que pode ser implementado utilizando uma pilha e que tem complexidade $O(|V| + |E|)$, tanto para grafos não direcionados como para grafos direcionados.

Outra poderosa técnica é a Busca em Largura. Seu algoritmo correspondente pode ser implementado usando uma fila, e tem complexidade linear $O(|V| + |E|)$ para grafos não direcionados ou para digrafos.

A Busca em Largura Lexicográfica e a Busca em Profundidade Lexicográfica são usadas só para grafos não direcionados. As complexidades para esses dois tipos de buscas são respectivamente $O(|V| + |E|)$ e $O(|V|^2)$. Tais buscas impõem uma determinada prioridade na escolha dos vértices a serem explora-

dos.

Nesta tese foi também apresentada a classificação dos algoritmos em três tipos: 1º) Totalmente Restringido; 2º) Não Restringido; e 3º) Parcialmente Restringido. O primeiro gasta $O(|V| + |E|)$; o segundo tem complexidade que depende do número de caminhos do grafo em questão e que pode ser exponencial no tamanho do grafo (pior caso);e o terceiro tem complexidade que depende da condição nele contida, podendo ser, no pior caso, exponencial.

Vários problemas em Teoria dos Grafos são resolvidos com mais eficiência utilizando as técnicas de busca aqui estudadas. A coletânea das aplicações apresentadas no Capítulo VI é composta de problemas considerados importantes em Teoria dos Grafos, havendo, é claro, outras aplicações que também utilizam tais técnicas de busca.

Finalmente, o que se pode observar é que todo algoritmo para resolver um problema em grafos requer sempre o exame de vértices e arestas. Ao se examinar um grafo de uma maneira estruturada, alguns algoritmos tornam-se mais fáceis de entender e mais rápidos durante a execução. A escolha do método a ser usado frequentemente afetará a eficiência do algoritmo. Porém, selecionar simplesmente uma boa estrutura de dados não é suficiente para garantir uma boa implementação. Uma escolha cuidadosa da Técnica de Busca também se faz necessária.

REFERÊNCIAS

|1| AHO, A.V., HOPCROFT, J.E. & ULLMAN, J.D., The Design and Analysis of Computer. Addison Wesley, Reading, Ma., 1974.

|2| ARLAZAROV, V.L., DINIC, E.A., KRONROD, M.A. & FARADZEV, I. A., On Economical Construction of the Transitive Closure of an Oriented Graph. Soviet. Math. Dokl., 11: 1209-1210, 1970.

|3| BAASE, S., Computer Algorithms: Introduction to Design and Analysis. Reading, Addisson-Wesley Publishing, 1978.

|4| BARBOSA, R.M., Combinatória e Grafos. Nobel, São Paulo, S.P., 1974.

|5| BASILI, V.R., MESZTENYI, C.K. & REINBOLDT, W.C., FGRAAL: Fortran Extended Graph Algorithmic Language. Computer Science Center Report, Univ. of Maryland, 1972.

|6| BEHZAD, M., A Characterization of Total Graphs. Proc. Amer. Math. Soc., 26: 383-389, 1970.

|7| BELL, J.R. & ABEL, N.E., An Algorithm for Finding Immediate Predominators in Optimizing Compilers. Digital Equipament Corp. Maynard, Mass, (Unpublished), 1972.

|8| BERZTISS, A.T., A Backtrack Procedure For Isomorphism of Directed Graphs. J. ACM, 20(3): 365-377, Jul., 1973.

|9| BLONIARZ, P.A., FISCHER, M.J. & MEYER, A. R., A Note on the Average Time to Compute Transitive Closures: Automata Languages and Programming. Michaelson and Milner, eds., Edinburgh Univ. Press, Edinburgh,1976.

|10| BOAVENTURA NETO, P. O., Teoria e Modelos de Grafos. Edgar Blücher, São Paulo, S.P., 1979.

|11| CHASE, S.M., GASP - Graph Algorithm Software Package. Quartely Tech. Progress. Rep., Department of Computer Science, Univ. of Illinōis, Urbana, 1969.

|12| CRESPI-RECHIZZI, S. & MORPURGO, R., A Graph Theory Oriented Extension to Algol. Calcolo, 5(4): 1-43. 1968.

|13| CRESPI-RECHIZZI, S. & MORPURGO, R., A Language for Treating Graphs. Comm ACM, 13(5): 319-323, Maio, 1970.

|14| DEO, N., Graph Theory With Applications to Engineering and Computer Science. Englewood Cliffs Prentice-Hall, Series in Automatic Computation, 1974.

|15| DEO, N., DAVIS, J.M. & LORD, R.E., A New Algorithm for Digraph Isomorphism. BIT, 17: 16-30, 1977.

|16| DEO, N., KRISHNAMOORTHY, M.S. & PAI, A.B., Generalizations of Line Graphs and Applications. Inf. Process. Lett, 6(1): 14-17, FEV., 1977.

|17| DEO, N., Breadth and Depth-First Searches in Graph Theoretic Algorithms. J. Computer and System Science, 4(2): 31-37, 1974.

|18| DIJKSTRA, E.W., A Note on Two Problems in Connexion With Graphs. Numer. Math, 1: 269-271, 1959.

|19| DILWORTH, R.P., A Decomposition Theorem for Partially Ordered Sets. Ann. Math., 51: 161-166, 1950.

|20| DIRAC, G.A., On Rigid Circuit Graphs. Abh. Math. Sem. Univ. Hamburg, 25: 71-76, 1961.

|21| DREYFUS, S.E., An Appraisal of Some Shortest Path Algorithms. Oper. Res., 17: 395-412, 1969.

|22| EBERT, J., A Sensitive Transitive Closure Algorithm. Inf. Process.Lett., 12(5): 255-258, OUT., 1981.

|23| EBERT, J., A Note on Odd and Even Factors of Undirected Graphs. Inf. Process. Lett., 11(2): 70-72, OUT., 1980.

|24| ELMAGHRABY , S.E., The Theory of Networks and Management Science, (Part I). Manage. Sci., 17: 1-34, 1970.

|25| EVEN, S., Algorithmic Combinatorics. Macmillan Co., New York, N.Y., 1973.

|26| EVEN, S., Graph Algorithms. Computer Science Press, Potomac, Mar., 1979.

|27| FISCHER, M.J, & MEYER, A.R., Boolean Matrix Multiplication and Transitive Closure. Twelfth Annual IEEE Symp. on Switching and Automata Theory, East Lansing, MI, 1971.

|28| FULKERSON, D.R. & GROSS, O.A., Incidence Matrices and Interval Graphs. Pacific J. Math., 15: 835-855, 1965.

|29| FURTADO, A.L., Teoria dos Grafos - Algoritmos. Livros Téc nicos e Científicos Editora S.A., Rio de Janeiro, R.J, 1973.

|30| GAREY, M.R. & JOHNSON, D.S., Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, C.A., 1979.

|31| GAREY, M.R., & TARJAN, R.E., A Linear Time Algorithm for Finding all Feedback Vertices. Inf. Process. Lett, 7(6): 274-276, OUT., 1978.

|32| GAVRIL, F., A Recognition Algorithm for the Total Graphs. NETWORKS, 8: 121-133, 1978.

|33| GOLUMBIC, M.C., Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York, N.Y., 1980.

|34| GRAHAM, R.L., KNUTH, D.E. & MOTZKIN, T.S., Complements and Transitive Closures. Discrete Math, 2: 17-30, 1972.

|35| HARARY, F., Graph Theory. Addison-Wesley, Publ. Co., Reading, Mass, 1969.

|36| HARARY, F., NORMAN. R.Z. & CARTWRIGHT, D., Structural Models: An Introduction to the Theory of Directed Graphs. John Wiley & Sons Inc., New York, N.Y., 1965.

|37| HART, R., HINT: A Graph Processing Language. Research Report, Computer Institute for Social Science Research, Michigan State Univ., East Lansing, FEV., 1969.

|38| HECHT, M.S. & ULLMAN, J.D., Flow Graph Reducibility. SIAM. J. COMPUT., 1(2): 188-202, JUN, 1972.

|39| HECHT, M.S. & ULLMAN, J.D., Characterizations of Reducible Flow Graphs., J.ACM, 21(3): 367-375, JUL, 1974.

|40| HOPCROFT, J.E. & TARJAN, R.E., Isomorphism of Planar Graphs, in Complexity of Computer Computations. MILLER, R.E., & THATCHER, J.W., eds., Plenum Press, New York, N.Y., 1972.

|41| HOPCROFT, J.E. & TARJAN, R.E.,Efficient Planarity Testing. J. ACM, 21: 549-568, 1974.

|42| JOHNSON, D.B., Finding all the Elementary Circuits of a Directed Graph. SIAM J. COMPUT., 4: 77-84, 1975.

|43| KAMEDA, T., On the Vetor Representation of the Reachability in Planar Directed Graphs. Inf. Process. Lett., 3(3): 75-77, JAN., 1975.

|44| KARAYIANNIS, A. & LOIZOU, G., Cycle Detection in Critical Path Networks. Inf. Process. Lett, 7(1): 15-19, JAN, 1978.

|45| KLEIN, M.M., Scheduling Project Networks. Comm. ACM, 10: 225-231, 1971.

|46| KNUTH, D.E. & SZWARCFITER, J.L., A Structured Program to Generate all Topological Sorting Arrangements. Inf. Process. Lett., 2(6): 153-157, ABR., 1974.

|47| KNUTH, D.E., Fundamental Algorithms - The Art of Computer Programming 1. Addison-Wesley, Publ. Co., Reading Mass., 1968 (segunda edição em 1973).

|48| LASS, S.E., PERT Time Calculation Without Topological Ordering. Comm. ACM. 8: 172-174, 1965.

|49| LEKKERKERKER, C.G. & BOLAND, J.C., Representation of a Finite Graph by a Set of Intervals on the Real Line. Fund. Math., 51: 45-64, 1962.

|50| LOVÁSZ. Combinatorial Problems and Exercices. North-Holland. Amsterdam, 1979.

|51| LUCAS, E., Récreations Mathématiques. Paris, 1882.

|52| LUCCHESI, C.L., Introdução à Teoria de Grafos. 12º Colóquio Brasileiro de Matemática, Poços de Caldas, M.G., 1979.

|53| MATETI, P. & DEO, N., On Algorithms for Enumeration all Circuits of a Graph. SIAM J. COMPUT., 5(1): 90-99, MAR, 1976.

|54| OHTSUKI, T., A Fast Algorithm for Finding an Optimal Ordering for Vertex Elimination on a Graph. J. Math. Anal. Appl., 5: 133-145, 1976.

|55| PACAULT,J.F., Computing the Weak Components of a Directed Graph. SIAM. J. COMPUT, 3(1): 56-61, MAR., 1974.

|56| PRICE, W.L., Graphs and Networks - an Introduction. Butterworth & Co., London, 1971.

|57| PURDOM, P.W. & MOORE, C. W., Algorithm 430: Immediate Predominators in a Directed Graph . Ibid, 15: 777-778, 1972.

|58| READ, R.C. & CORNEIL, D.G., The Graph Isomorphism Disease. J. Graph Theory, 1 (4): 339-363, 1974.

|59| READ, R.C., Teaching Graph Theory to a Computer, Recent Progress in Combinatorics. W.T. Tutte edc, Academic Press, Inic. NEW YORK, 1969.

|60| REINGOLD, E.M., NIEVERGELT, J., & DEO, N., Combinatorial Algorithms: Theory and Practice. Englewood Cliffs, Prentice-Hall, 1977.

|61| ROBERTS, S.M. & FLORES, B., Systematic Generation of Hamiltoniam Circuits. Comm. ACM, 9: 690-694, 1964.

|62| ROSE, D.J., TARJAN, R.E. & LEUKER, G.S., Algorithmic Aspects of Vertex Elimination on Graphs. SIAM J. COMPUT. 5(2): 266-283, JUN., 1976.

|63| ROY, M.B., Transitivité et Connexité. Comptes Rendus, 249: 216-218, 1959.

|64| SAVULECU, S.C., Grafos, Digrafos e Redes Elétricas, Inst. Bras. Ed. Cient., São Paulo, S.P., 1980.

|65| SCHNORR, C.P., An Algorithm for Transitive Closure With Linear Expected Time. SIAM J. COMPUT., 7(2): 127-133, MAIO, 1978.

|66| SHAMIR, A., A Linear Time Algorithm for Finding Minimum Cutsets in Reducible Graphs. SIAM J. COMPUT., 8(4): 645-655, NOV., 1979.

|67| SZWARCFITER, J.L. & LAUER, P.E., A Search Strategy for the Elementary Cycles of a Directed Graph. BIT, 16: 192-204, FEV., 1976.

|68| SZWARCFITER, J.L., On Optimal and Near Optimal Algorithms for Some Computational Graph Problems. Ph.D. Thesis, Comp. Lab., Univ. of Newcastle upon Tyne Newcastle upon Tyne, 1975.

|69| SZWARCFITER, J.L., On a Class of Acyclic Directed Graphs. Mem, UCB/ERL M80/6, Elec. Res. Lab., Univ. of Calif., Berkeley, Calif., 1980.

|70| SZWARCFITER, J.L., PERSIANO, R.C.M. & OLIVEIRA, A.A.F., Um Problema em Orientação de Grafos. Relatórios Técnicos do Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, 1981.

|71| TARJAN, R.E., Depth-First Search and Linear Graph Algorithms. SIAM J. COMPUT., 1(2): 146-160, JUN, 1972.

|72| TARJAN, R.E., Finding Dominators in Directed Graphs. SIAM J. COMPUT., 3(1): 62-89, MAR, 1974.

|73| TARJAN, R.E., A New Algorithm for Finding Weak Components. Inf. Process. Lett., 3(1): 13-61, JUL., 1974.

|74| TARJAN, R.E., Enumeration of the Elementary Circuits of a Directed Graph. SIAM J. COMPUT., 2(3): 211-216, SET., 1973.

|75| TARJAN, R.E., Testing Flow Graph Reducibility. J. Computer and System Sciences, 9(3): 355-365, DEZ., 1974.

|76| TARJAN, R.E., Maximum Cardinality Search and Chordal Graphs, Comp. Sc. Dept., Stanford Univ., Unpublished Lecture, 1976.

|77| TARJAN, R.E. & HOPCROFT, J.E., Dividing a Graph into Triconnected Components. SIAM J. COMPUT, 2(3); 135-158, SET., 1973.

|78| TARRY, G., Le Problème des Labyrinthes. Nouvells Ann. de Math, 14, 1875.

|79| WARSHALL, S., A Theorem on Boolean Matrices. J. ACM., 9: 11-12, 1962.

|80| WEINBLATT, H., A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph. J. ACM, 19(1): 43-56, JAN., 1972.