UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MÁRCIO MUCCILLO SKLAR

Algoritmo Distribuído Detector de Ciclos Baseado em Busca e Difusão

Trabalho de Graduação

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, dezembro de 2008

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Sklar, Márcio Muccillo

Algoritmo Distribuído Detector de Ciclos Baseado em Busca e Difusão / Márcio Muccillo Sklar. – Porto Alegre: Instituto de Informática da UFRGS, 2008.

56 f.: il.

Trabalho de Graduação – Universidade Federal do Rio Grande do Sul. Curso de Graduação em Ciência da Computação, Porto Alegre, BR–RS, 2008. Orientador: Cláudio Fernando Resin Geyer.

1. Algoritmo Distribuído. 2. Detecção de Ciclo. 3. Difusão. 4. Busca em Grafos. 5. Dígrafo. 6. Redes. I. Geyer, Cláudio Fernando Resin. II. Título.

“The important thing is not to stop questioning.
Curiosity has its own reason for existing.
(...) the mysteries of eternity, of life, of the marvelous structure of reality.
(...) to comprehend a little of this mystery every day.
Never lose a holy curiosity.”
— Albert Einstein

AGRADECIMENTOS

À minha mãe, pela educação e disponibilidade.
À minha namorada, pelo carinho e compreensão.
Aos meus amigos e colegas, pela descontração.
Ao meu orientador, por confiar no Trabalho.
Ao Instituto de Informática da UFRGS, pelo conhecimento e pela estrutura.
A Donald Knuth, pelo TEX.
A Leslie Lamport, pelo LATEX e pelas diversas contribuições à área de Sistemas Distribuídos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADDC	Algoritmo Distribuído Detector de Ciclos
AG	Arborescência Geradora
BFS	*Breadth-First Search*
BGP	*Border Gateway Protocol*
DAG	*Directed Acyclic Digraph*
DFS	*Depth-First Search*
FIFO	*First-In First-Out*
id	Identificador
MST	*Minimal Spanning Tree*
OSPF	*Open Shortest Path First*
RIP	*Routing Information Protocol*
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE SÍMBOLOS

$\textcircled{a}$	Nodo com rótulo a
$\leftrightarrow$ ou $-$	*Link* Bidirecional
$a \rightarrow b$ ou a, b	Caminho, de comprimento um, de a até b
$a \rightarrow b \rightarrow c$ ou a, b, c	Caminho de comprimento dois, de a até c, passando por b

LISTA DE FIGURAS

LISTA DE CÓDIGOS

RESUMO

O presente trabalho tem como objetivo principal a modelagem de um algoritmo distribuído baseado em busca e difusão para detectar ciclos simples em uma rede de topologia qualquer. Existem, na literatura, uma série de algoritmos de busca em grafos. De acordo com o tipo de estrutura utilizada para armazenamento, a ordem em que o grafo é percorrido é alterada, caracterizando um tipo de busca com aplicações diferentes. Trabalhos prévios definem, em geral, métodos cujo objetivo é a descoberta de rotas acíclicas, caracterizada por cobrir todos os nodos de uma rede conexa, através da formação de uma *spanning tree* do grafo. Inversamente, este trabalho centra-se na modelagem de um algoritmo distribuído capaz de capturar rotas cíclicas a partir de nodos previamente escolhidos. Para tanto, um modelo simples de difusão, com busca exaustiva de caminhos, é utilizado sobre uma rede, devidamente abstraída por meio de um dígrafo.

Palavras-chave: Algoritmo Distribuído, Detecção de Ciclo, Difusão, Busca em Grafos, Dígrafo, Redes.

Cycle Finder Algorithm based in Search and Diffusing Computations

ABSTRACT

The main goal of this paper is the modelling of a distributed algorithm based in search and diffusion computations in order to detect cycles in any network topology. There are, in the literature, many graph-search algorithms. Each storage structure defines a order in which the graph is traversed and different applications of this search. Previous works define, in general, methods whose goal is the discovery of acyclic routes, carachterized by covering all nodes of a connected network by discovering a spanning tree of the graph. Instead of it, this paper is centered in the modelling of a distributed algorithm able to detect cyclic routes starting of initial nodes previously chosen. In order to perform it, a simple diffusing computation model that searches exaustively paths is utilized in a network, properly represented by a digraph.

Keywords: Distributed Algorithm, Cycle Detection, Diffusing Computation, Graph Search, Digraph, Computer Network.

1 INTRODUÇÃO

1.1 Dígrafos e Ciclos Diretos

Segundo (SEDGEWICK, 2002, p. 209, tradução nossa):

> Grafos são um dos tópicos unificadores em Ciência da Computação - uma representação abstrata que pode descrever a organização de sistemas de transporte, circuitos elétricos, interações humanas e redes de telecomunicações. Todas essas diferentes estruturas que podem ser modeladas usando um formalismo simples é uma fonte de poder para o programador disciplinado.

"Um grafo $G = (V,E)$ é definido por um conjunto de vértices V, e um conjunto de arestas E consistindo de **pares ordenados ou não-ordenados** de vértices de V." (SKIENA, 2007a, p. 3, tradução e grifo nossos). Dessa definição de SKIENA, conclui-se que grafos podem ter seu conjunto E consistindo de **pares ordenados** de vértices de V, o que se constitui em **grafos direcionados** (ou **dígrafos**), cujas elementos do conjunto E (pares com ordem) são chamados, neste trabalho, de **arcos**, pois possuem um sentido determinado.

Dígrafos são melhores aplicáveis à prática, em que uma rede, a Internet por exemplo, freqüentemente possui *links*, que por diversos motivos, funcionam em um só sentido. Segundo (BROIDO; CLAFFY, 2001, pp. 1,2, tradução e grifo nossos), "grafos direcionados refletem mais acuradamente a conectividade presente na Internet. Uma vez que as rotas são baseadas em políticas, o *link* reverso, mesmo quando presente fisicamente, nem sempre é capaz de transmitir tráfego de resposta [no sentido contrário]".

Para refletir este intento, também a implementação dos algoritmos, neste trabalho, é feita utilizando dígrafos. Outrossim, **dígrafos simétricos** poderão ser utilizados quando se quiser ter comunicação bidirecional, ou mesmo simular arestas de grafos não-direcionados, as quais podem assumir um ou outro sentido.[1]

Além do que, grafos direcionados são úteis para representar: fluxogramas, estados de uma rede de petri, máquinas de estados em geral, categorias, etc. "Ao modelar uma rede de ruas, os vértices podem representar as cidades ou junções, em que alguns pares estão conectados por ruas/arestas. [...] Em um circuito eletrônico, as junções são os vértices, e os componentes, as arestas." (SKIENA, 2007a, pp. 4,5, tradução nossa).

[Sedgewick, pp. 14,141, tradução nossa] também ressalta o potencial dos dígrafos:

> Muitas aplicações [...] são naturalmente expressas em termos de dígrafos. [...] Interpreta-se a direção dos arcos em dígrafos de várias ma-

[1] A utilização de dígrafos simétricos para "simular" estruturas de grafos não-dirigidos em implementações de algoritmos não só é muito comum como também é a regra.

> neiras. Por exemplo, em um dígrafo de chamadas telefônicas, pode-se considerar um arco como estando direcionado de quem fez a chamada para quem a recebeu. Em uma dígrafo de transações, pode-se ter uma relação similar em que se interpreta um arco como dinheiro, coisas ou informações fluindo de uma entidade à outra. Pode-se encontrar situações mais atuais nas quais tal modelo serve para modelar a Internet, com vértices representando páginas da *Web*, e os arcos, os *links* entre as páginas.

Não obstante, muitos problemas facilmente resolvidos sobre grafos não-orientados tornam-se complicados quando transpostos para dígrafos. (SEDGEWICK, 2002, p. 66 e 67) cita, como exemplo dessa dificuldade, o problema da conectividade, o problema da existência de ciclo de comprimento par, entre outros.

Em computação, ciclos são, em geral, modelados como caminhos fechados **simples** em grafos. Dessa forma, ciclos são constituídos por uma seqüência "circular" de **vértices distintos**. Em dígrafos, ciclos são também chamados de **ciclos diretos**, já que possuem um sentido bem definido. Ciclos, portanto, não possuem nodos repetido (exceto o primeiro, que é igual ao último). Neste trabalho, quando um nodo é encontrado (visitado) pela segunda vez, ocorre justamente a detecção desse ciclo, terminando especificamente esse processamento, por tê-lo encontrado.[2] [3]

1.2 Detecção de Ciclos

Trabalhos prévios, no nível de implementação de protocolos, têm-se preocupado, sobretudo, na detecção de rotas livres de ciclos. Em (BROIDO; CLAFFY, 2001, p. 4), a remoção tanto de *loops* estáticos quanto a de *pseudoloops* formados quando um rota rapidamente alterna entre dois caminhos permite evitar o barulho de conectividade causado pela propagação de pacotes em multicaminhos de tamanhos variados entre a fonte e o *responsive node*.

Existem vastamente na literatura protocolos que implementam diversos tipos de algoritmos de roteamento. Uma das classes existentes é a que faz uso de **vetor de distância (*distance-vector routing*)**. O método é baseado no algoritmo de *BellmanFord*, que calcula caminhos mínimos em um dígrafo com pesos nos arcos. O algoritmo funciona de forma que cada nodo distribua a seus vizinhos as informações sobre distâncias na rede (compartilhamento de tabelas de roteamento), de forma que todos saibam a melhor rota para chegar a um destino qualquer. Entretanto, possui a limitação de não prevenir a formação de *loops* (ciclos) e sofrer com o problema da "contagem ao infinito". Como exemplo desse tipo de modelo, tem-se o RIP (*Routing Information Protocol*), que ameniza os problemas citados e o OSPF (*Open Shortest Path First*), que trabalha bem em grandes redes. Muitas outras implementações evitam a formação de *loops*, mas sofrem com o aumento da complexidade.

Outra classe de algoritmos é chamada de roteamento por **vetor de caminho (*path vector routing*)**. Tal modelo mantém o caminho que atualiza informações levadas pela rede por meio de difusão. Atualizações que contém *loops* na rede são detectadas e descartadas.[4] Um exemplo de implementação dessa classe é o BGP (*Border Gateway Protocol*),

[2] Quando for mencionado o termo **ciclo**, estará implícito **ciclo simples direcionado**.

[3] Pode-se conferir uma explicação mais detalhada no exemplo 1.5.33 e na explicação conseguinte, constante na página 21.

[4] Este modelo é muito semelhante ao proposto neste trabalho - o ADDC. A diferença é que aqui vai-se propor a difusão em um nível mais alto de abstração, detendo-se mais no nível do dígrafo representado e

protocolo base da camada de rede da Internet, projetado para evitar *loops* de roteamento em topologias arbitrárias.

Para (TEL, 2000) os principais critérios para um bom método de roteamento em redes são a corretude, a eficiência, a complexidade, a robustez, a adaptatividade e a justiça no uso por diferentes usuários. Entretanto a maioria dos algoritmos leva em consideração apenas um subconjunto destes critérios.

O problema da detecção de ciclos envolve uma série de outras questões teóricas e práticas da computação e da matemática. Como exemplo, podem-se citar:

- O correto e eficiente roteamento, disseminação e coleta de informações em redes se utilizam de algoritmos para construção de estruturas do tipo *spanning* (tais como *spanning trees*, r-subárvores maximais), os quais, por sua vez, fazem uso de propriedades bem conhecidas dos ciclos.

- A detecção de ciclos pode outrossim ser utilizada como forma de se identificar *loops* infinitos em alguns tipos de programas, tal como apresentado em (GELDER, 1987) apud (WIKIPEDIA, 2008a).

- Conforme mostrado em (NIVASCH, 2004) apud (WIKIPEDIA, 2008a), configurações periódicas em simulações de autômatos celulares podem ser encontradas aplicando a detecção de ciclos sobre a seqüência de estados assumidos pelo autômato.

- Em (AUGUSTON; HON, 1997) apud (WIKIPEDIA, 2008a), a detecção de ciclos em listas encadeadas acusa uma incorreção no algoritmo que está utilizando tal estrutura de dados. Também se pode garantir que ao percorrer uma lista encadeada em sentido contrário não se introduza ciclos nessa lista.

- Algoritmos de detecção de ciclos ajudam a encontrar e a melhorar "fraquezas" em funções criptográficas, tais como na ajuda ao estabelecer a resistência de colisão em funções de *hash* criptográficas.

- Em (KNUTH, 1969) apud (WIKIPEDIA, 2008a), medir o quão forte é um "gerador numérico pseudorandômico" através do comprimento do ciclo gerado por tal seqüência numérica é uma das aplicações citadas por KNUTH ao adaptar o método de FLOYD utilizando grafos funcionais, que ficou também conhecido como "Algoritmo da Tartaruga e da Lebre" (*Tortoise and Hare Algorithm*), pois se utiliza de dois ponteiros que se movem na seqüência de vértices descrita pelas funções do grafo a velocidades diferentes. [5] O Algoritmo de Floyd para detecção de ciclos, criado por FLOYD na década de sessenta, conforme (FLOYD, 1967) apud (WIKIPEDIA, 2008a), é utilizado para listagem de todos os ciclos simples em um grafo direto.

- Em (POLLARD, 1975) apud (WIKIPEDIA, 2008a), destacam-se os algoritmos de POLLARD, baseados no método de FLOYD para detecção de ciclos, para fatoração de números inteiros.

em suas propriedades. Outra diferença é que os ciclos detectados serão armazenados.

[5] Grafos funcionais são dígrafos nos quais todo vértice tem grau de saída igual a um, simulando o comportamento de uma função.

Para saber, simplesmente, se um grafo possui ou não algum ciclo (é ou não um **DAG [Directed Acyclic Digraph]**), pode-se utilizar um **algoritmo de busca em profundidade (*Depth Search First [DFS]*)**. A idéia é percorrer uma **diárvore** que seja **subdígrafo** do dígrafo em questão, cujos **arcos de retorno** indicam a existência de ciclo no dígrafo, e, simetricamente, todo dígrafo que possua ciclo possui **arco de retorno**.

Não obstante, a simples transposição de uma busca DFS para o ambiente distribuído a ser usado neste trabalho é inviável, já que DFS supõe a utilização de uma pilha em memória centralizada. Em uma rede genérica, aqui tratada, pode não haver a existência de memória compartilhada, para armazenamento e compartilhamento de tal estrutura. De qualquer forma, a implementação e gerenciamento desse tipo de memória pode se tornar custosa. Além disso, DFS não é capaz de listar todos os ciclos existentes em um dígrafo.

Além das buscas, existe, na literatura, descrições de algoritmos do tipo "força-bruta" para encontrar ciclos em um dígrafo. (FEOFILOFF, 2008a) descreve um método simples que parte do fato que, dado um arco $v \rightarrow w$ de um dígrafo D, todos os caminhos simples distintos partindo de w e chegando em v definem um ciclo direto. Para tanto, a busca por tais caminhos é feita percorrendo-se todas as possibilidades, para cada um dos arcos de D. Nesse caso, ciclos triviais são considerados. Ciclos triviais são ciclos com comprimento dois. Em grafos não-direcionados, os ciclos triviais comumente são ignorados, pois se utilizam da mesma aresta (em ambos os sentidos) para compor o ciclo trivial.

Uma das formas de percorrer uma rede é através de métodos de difusão. Algoritmos de difusão são utilizados para, por exemplo, determinar caminhos ótimos entre dois pontos de uma rede. Tal classe de algoritmo é também utilizada, de forma eficiente, na descoberta de topologia de redes.

"Dijkstra e Scholten [(DIJKSTRA; SCHOLTEN, 1980)] criaram a noção de difusão para prover uma base genérica para uma grande e importante classe de problemas [...] os resultados de difusões podem ser aplicados a uma grande variedade de problemas." (CHANDY; MISRA, 1980, tradução nossa).

Um algoritmo nesses moldes, que dê conta de percorrer, de forma distribuída, uma rede em busca de ciclos, é de grande interesse, sobretudo se for facilmente definido e de pouca complexidade. Entretanto, tal algoritmo certamente terá um comportamento mais complexo do que uma busca seqüencial, pois precisa dar conta de explorar, de forma concorrente e distribuída, o ambiente.

Neste sentido, o objetivo principal do trabalho é propor um algoritmo - chamado de ADDC - com memória distribuída que possa detectar ciclos em uma rede, modelada segundo um dígrafo. Dessa forma, os ciclos detectados serão dirigidos (diretos), além de simples (sem vértices repetidos). É utilizado, para tanto, um modelo de multidifusão em que determinados nodos, previamente marcados, iniciam uma difusão de informações, para posterior coleta dos ciclos, dentro do mesmo fluxo inicialmente gerado.

1.3 Objetivos

- Estudar primeiramente as alternativas existentes para representação de um grafo em ambiente seqüencial, e esboçar uma forma de representação que naturalmente suporte a fácil visualização, por pessoas, de ciclos em um dígrafo.

- Estudar algoritmos seqüenciais de busca para um melhor entendimento sobre seu comportamento. Descrever algumas implementações existentes, comparando-as e propor outras implementações, visando a uma compreensão mais profunda sobre

tais métodos.

- Formular um modelo de algoritmo distribuído baseado em difusão e busca de ciclos em um ambiente assíncrono de rede, com memória distribuída.
- Descrever o ADDC em pseudocódigo, com vistas a facilitar o desenvolvimento de um futuro protótipo.
- Calcular a complexidade do algoritmo proposto, comparando-a com a de outros algoritmos.
- Provar a corretude e a terminação do Algoritmo proposto.

1.4 Organização do Trabalho

O trabalho em questão está dividido da seguinte forma:

1. Primeiramente, na seção 1.5, algumas definições essenciais de Teoria dos Grafos assumidas no trabalho, portanto fundamentais para o entendimento do que segue, são postas.
2. No capítulo 2, são estudados a estruturação e travessia de dígrafos em ambiente seqüencial.
3. No capítulo 3, O ADDC é descrito e formalizado, juntamente com o cálculo de sua complexidade e prova de corretude e terminação; após, exemplos de funcionamento do algoritmo são apresentados.
4. Na conclusão, capitulo 4, os resultados e conceitos obtidos são discutidos, bem como são enumerados alguns tópicos de trabalhos posteriores.

1.5 Definições

Neste capítulo constam uma série de definições e conceitos essenciais para o entendimento do trabalho. Em sua maioria, são conceitos provindos da Teoria dos Grafos.

1.5.1 Definições Básicas de Teoria dos Grafos

Definição 1.5.1 (Vértice ou Nodo) *Representa uma das pontas de uma aresta ou de um arco.*

Definição 1.5.2 (Arco) *Um **arco dirigido** (ou somente **arco**) (u,v) é uma ligação unidirecional (com sentido) entre dois nodos u e v e escreve-se $u \rightarrow v$ ou simplesmente $u - v$ se o contexto estiver claro. O nodo u representa a origem do arco, e o nodo v representa seu destino.*

Definição 1.5.3 (Origem ou Entrada do Arco) *Ponta inicial do arco $u - v$, incidente no nodo u.*

Definição 1.5.4 (Destino ou Saída do Arco) *Ponta final do arco $u - v$, incidente no nodo v.*

Definição 1.5.5 (Arcos Paralelos) *Diz-se de dois ou mais arcos com a mesma origem e o mesmo destino.*

Definição 1.5.6 (Arco Antiparalelo) *Diz-se de **arco inverso** ($v \rightarrow u$) a um dado arco $u \rightarrow v$.*

Definição 1.5.7 (Aresta) *Uma aresta u,v é uma ligação, sem sentido definido, entre dois nodos u e v e escreve-se uv, $u - v$ ou u,v.*

Definição 1.5.8 (Adjacência e Incidência) *Dois vertices v, w são ditos adjacentes se existe um arco (ou aresta) entre eles. Tais vértices são ditos incidentes a este(a) arco (aresta), bem como este(a) arco (aresta) é dito(a) incidente a tais nodos.*

Definição 1.5.9 (Vizinhança) *Em **grafos não-direcionados**, um nodo v **é vizinho de** um nodo w, e vice-versa, se e somente se existe uma aresta ligando esses dois nodos. Já em **dígrafos**, o conceito é um pouco diferente, uma vez que **ser vizinho** é, neste caso, uma propriedade não-comutativa. Ou seja: Um nodo v **é vizinho de** um nodo w se e somente se existe um arco com saída em w e entrada em v ($w \rightarrow v$); por outro lado, v **possui** w **como vizinho** se e somente se existe um arco com saída em v e entrada em w ($v \rightarrow w$).*

Definição 1.5.10 (Grafo Simples Não-Dirigido) *Um **grafo simples** G [daqui para frente, somente **grafo**; ou equivalentemente, uma **rede bidirecional** [ou **não-orientada**]] é um par $(V(G),A(G))$, em que $V(G)$ é um conjunto finito, não-vazio de elementos chamados vértices (ou nodos ou nós), e $A(G)$ é um conjunto finito de pares não-ordenados de elementos distintos de $V(G)$, chamados de arestas.* (LEONEL, 2001, p.1).

Definição 1.5.11 (Grafos Conexos) *Um grafo é dito **grafo conexo** se, dado qualquer par de vértices u, v de G, existe uma rota de a para b [intuitiva e equivalentemente, se não existe um nodo isolado na rede, ao qual não se pode chegar por meio de um canal de comunicação.]* (LEONEL, 2001, p.4).

Definição 1.5.12 (Dígrafo (Dirigido) Simples) *Um **dígrafo simples** D [daqui para frente, somente **dígrafo**; ou equivalentemente, uma **rede orientada** {ou **unidirecional**}] é definido como um par $(V(D), A(D))$, em que $V(D)$ é um conjunto finito, não-vazio de vértices, e $A(D)$ é um conjunto finito de pares ordenados de elementos distintos de $V(D)$, chamados [...] arcos dirigidos [ou somente arcos].*[6] (LEONEL, 2001, p.1).

Definição 1.5.13 (Dígrafo Simétrico) *Dígrafo em que todo arco possui um **arco antiparalelo**.*

Definição 1.5.14 (Bidirecionalidade ou Simetria) *Uma ligação entre quaisquer dois nodos u, v é dita **bidirecional** (ou **simétrica**) se e somente se v é vizinho de u ($u \to v$) e u é vizinho de v ($v \to u$), notando-se $u \leftrightarrow v$ ou somente $u - v$.*

Definição 1.5.15 (Grau) *O **grau** de um vértice v de um **grafo não-dirigido** G é o numero de arcos incidentes em v.*

Definição 1.5.16 (Grau de Entrada) *O **grau de entrada** de um vértice v de um **dígrafo** D é o numero de arcos com destino (entrando) em v.*

Definição 1.5.17 (Grau de Saída) *O **grau de saída** de um vértice v de um **dígrafo** D é o numero de arcos com origem (saindo) de v.*

Definição 1.5.18 (Caminhos e Circuitos) [7] *Dado um grafo [ou dígrafo] G, um **caminho** [ou **passeio**] em G é uma seqüência finita de arcos na forma: $v_0 \to v_1 \to v_2 \to v_3 \to \ldots \to v_{f-1} \to v_f$ (ou $v_0, v_1, v_2, v_3, \ldots, v_{f-1}, v_f$).*
v_0 é chamado de vértice inicial e v_f é o vértice final; o caminho é chamado de v_0 até v_f.
*O **comprimento** do caminho é o número de arcos do caminho.*
*Se [...] todos os vértices são distintos (exceto talvez v_0 e v_f), o chamamos de **rota** [ou **caminho simples**].*
*Uma **rota** é **fechada** se $v_0 = v_f$.*
*Uma rota fechada é um **circuito** [ou **rota cíclica** ou somente **ciclo**].*
(LEONEL, 2001, p.2-3, grifos nossos).

Definição 1.5.19 (Comprimento) *O **Comprimento** (ou **distância**) de um caminho refere-se ao número de arcos (ou arestas) do caminho.*

---

[6] Um grafo pode ser pensado como um dígrafo com os arcos nos dois sentidos.

[7] Em (WIKIPEDIA, 2008b, tradução e grifos nossos) e (WIKIPEDIA, 2008c), assim como neste trabalho, algumas ressalvas são feitas a respeito de definições controversas em Teoria dos Grafos:

> Tradicionalmente [por exemplo em (FEOFILOFF, 2008b)], um **caminho** refere-se ao que atualmente é chamado de **passeio aberto**. [...] Alguns autores [...] usam o termo **passeio** para caminhos nos quais vértices ou arestas podem ser repetidas e reservam o termo **caminho** para o que aqui é chamado de **caminho simples**. [...] Assim como o conceito de caminho, tradicionalmente [coerentemente, em (FEOFILOFF, 2008c)] um **ciclo** refere-se a qualquer **passeio fechado** (ou seja, em um **ciclo** podem haver **vértices repetidos**, além do primeiro e último), mas nos dias de hoje [e também neste trabalho] um **ciclo** deve ser entendido como **simples** por definição.

Definição 1.5.20 (Ciclo Trivial) *Ciclo sobre um grafo não-dirigido de comprimento igual a* 2 *(dois).*

Definição 1.5.21 (Caminho Cíclico) *Equivale ao conceito de* ***caminho fechado****. Um* ***caminho*** *pode ser* ***não-simples****, ou seja, pode conter vértices repetidos, além do inicial e do final no caso de ser fechado.*

Definição 1.5.22 (Caminho Direto) *Quando caminhos são definidos sobre dígrafos, são chamados de* ***caminho direto****. Se tal caminho for fechado, é chamado de* ***ciclo direto****.*

Definição 1.5.23 (Fracamente Conexo) *Um dígrafo é dito* ***fracamente conexo*** *(ou simplesmente* ***conexo****) se para todos os pares de nodos v e w do dígrafo, existe um caminho de v para w* ***ou*** *existe um caminho de w para v. Equivalentemente, um dígrafo é fracamente conexo se retirando o sentido dos arcos, isto é, transformando todos os arcos em arestas, obtém-se um grafo conexo.*

Definição 1.5.24 (Fortemente Conexo) *Um dígrafo é dito* ***fortemente conexo*** *se entre quaisquer dois nodos v e w do dígrafo, existe um caminho de v para w* ***e*** *existe um caminho de w para v*

Definição 1.5.25 (Território ou Alcançabilidade) *Um vértice v é* ***alcançável*** *a partir de um vértice r se existe um* ***caminho*** *de r a v. É interessante notar que se existe tal caminho, também existe um* ***caminho simples*** *de r a v. O* ***território*** *de um vértice r é o conjunto de todos os vértices alcançáveis a partir de r.*

Definição 1.5.26 (DAG) ***DAG (directed acyclic graph)*** *é um dígrafo que não contém ciclos (****acíclico****).*

Definição 1.5.27 (Subgrafo) *Um* ***subgrafo*** *(equivalentemente um* ***subdígrafo****) de um (dí)grafo G (D) é um (dí)grafo cujo conjunto formado por seus nodos é um* ***subconjunto*** *do conjunto formado pelos nodos de G (D) e cuja relação de* ***adjacência*** *é um subconjunto daquele restringido pelo subconjunto de nodos.* (WIKIPEDIA, 2008d).

Definição 1.5.28 (Arborescência) *Uma* ***arborescência*** *(ou arborescência divergente) é um* ***dígrafo*** *em que*

- *não existem vértices com grau de entrada maior que* 1 *(um)*
- *existe exatamente um vértice com grau de entrada zero* 0 *(zero)*
- *cada um dos vértices é término de um caminho com origem no único vértice que tem grau de entrada nulo.*

*O único vértice com grau de entrada nulo é a raiz da arborescência. É claro que todos os vértices diferentes da raiz têm grau de entrada igual a um (*1*). Arborescências são também conhecidas por* ***diárvore****,* ***árvore radicada (rooted tree)****,* ***árvore enraizada****,* ***árvore dirigida****,* ***árvore de busca (search tree)****, etc.* (FEOFILOFF, 2008d).

Definição 1.5.29 (Arborescência Geradora) *Uma* ***arborescência geradora (AG)*** *"A" de um dígrafo D é uma arborescência que é subdígrafo de D, tal que A possui todos os nodos de D.*

Definição 1.5.30 (Árvore) *Uma **árvore** é um **grafo** conexo que não contém ciclos. Toda árvore com V vértices tem exatamente $V - 1$ arestas. Qualquer adição de aresta forma um ciclo no grafo. Os nodos de grau igual a* 1 *(um) são conhecidos como folhas da árvore.*

Definição 1.5.31 (Árvore Geradora) *Uma **árvore geradora** A de um grafo G é uma árvore que é subdígrafo de G, tal que A possui todos os nodos de G.*

Definição 1.5.32 (Pai-Filho) *Em uma arborescência, todo arco define uma relação **pai-filho**, em que o nodo incidente na origem do arco é pai do nodo incidente no destino do arco, o qual é chamado filho daquele nodo.*

1.5.1.1 Ciclo

Exemplo 1.5.33 (Exemplo — ciclo não-simples) *A figura 1.1 mostra um exemplo de um ciclo que não é simples.*

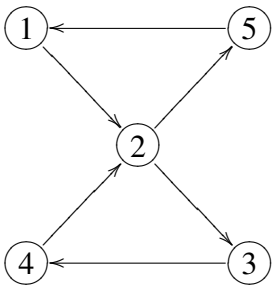

Figura 1.1: Exemplo de um ciclo não-simples

Na figura 1.1, está visível que o nodo ② é utilizado duas vezes na formação do que tradicionalmente se chama de **ciclo não-simples**, representado pelo caminho fechado (não-simples) seguinte: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 1$. Pelo viés de sua construção, o ciclo inicia no nodo ①, passa pelo nodo ②, pelo ③, pelo ④ e novamente pelo ②, quando poderia encerrar um ciclo ($2 \rightarrow 3 \rightarrow 4 \rightarrow 2$). Ao contrário, continua percorrendo seu caminho (daí a caracterização de ser não-simples), até encontrar o nodo ①, finalizando. Não obstante, o viés adotado pelo presente trabalho, indica, neste "ciclo não-simples", a existência de dois **ciclos simples**, a saber: o ciclo $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ e o ciclo $1 \rightarrow 2 \rightarrow 5 \rightarrow 1$.

Tais definições coadunam com a de [Sedgewick, p. 10, tradução nossa]: "Um caminho em um grafo é uma seqüência de vértices na qual cada vértice sucessivo (depois do primeiro) é adjacente ao seu predecessor no caminho. Em um caminho simples, os vértices e arestas são distintos. Um ciclo é um caminho que é simples exceto pelo fato de o primeiro e o último vértices serem o mesmo."

Como neste trabalho, se dará prioridade a uma visão mais construtivista, e seu objetivo primário é o de encontrar (construir) ciclos, os dígrafos serão sempre encarados como possuidores de **ciclos simples**. Assim sendo, no primeiro reencontro de um nodo qualquer, um ciclo simples deve ser detectado.

Na figura 1.2, tem-se um exemplo de um ciclo simples. Perceba-se que se pode encarar que o ciclo começa no nodo ①, percorre os demais e finaliza no nodo ①. Ou seja, apenas o nodo inicial e final coincidem. Todavia, pode-se visualizar o dígrafo como "um todo" em que todos os nodos cumprem a mesma função. Ou seja, qualquer um deles

poderia assumir o papel de “inicial-final”, fazendo com que tal conceito se torne irrelevante por esse ponto de vista. Conforme bem observa (WIKIPEDIA, 2008c, tradução e grifos nossos), “um ciclo é um caminho (**simples**) em que o vértice inicial e o final são os mesmos. Note-se, contudo, que ao contrário dos caminhos, qualquer vértice de um ciclo pode ser identificado como sendo o início do ciclo, logo o início é freqüentemente não especificado”.

Exemplo 1.5.34 (Exemplo — ciclo simples) *A figura 1.2 mostra um exemplo de um ciclo simples.*

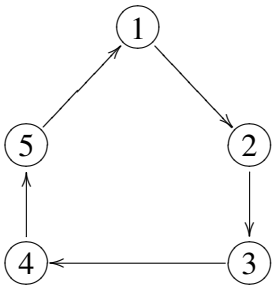

Figura 1.2: Exemplo de um ciclo simples.

2 AMBIENTE SEQÜENCIAL

Grafos são a base para um grande número de problemas e aplicações em Ciência da Computação. (CORMEN et al., 2001, p. 525, tradução nossa) afirma o seguinte a respeito do assunto: "Grafos são estruturas de dados pervasivas em Ciência da Computação, e os algoritmos que os utilizam são fundamentais para a área. Há centenas de problemas computacionais interessantes definidos em termos de grafos.".

Neste capítulo, primeiramente, na seção 2.1, são revistos conceitos de estrutura de dígrafos em ambiente seqüencial e, após, na seção 2.2, são descritos algoritmos de travessia, a fim de se ter uma base importante para os fundamentos do objetivo do trabalho.

2.1 Estrutura

Abaixo, explicam-se duas formas clássicas de se caracterizar um dígrafo qualquer: representação por **incidência** e por **adjacência**.

- Incidência: Esta forma de representação destaca em quais arcos, e de que forma, cada um dos nodos do dígrafos é incidente. Ela, comumente, se utiliza de uma matriz *nxm* (**matriz de incidência**), em que n é o número de nodos (rotulados de v_1 a v_n) e m é o número de arcos (rotulados a_1 a a_m), de forma que a posição e_{ij} da matriz é preenchida com o valor 1 (respectivamente -1) se a saída (respectivamente a entrada) do arco a_j é incidente ao nodo n_i, e preechida com 0 em caso contrário.

  Esta forma de representação não é muito utilizada na prática, já que algumas tarefas simples, tais como descobrir ou listar os vizinhos (adjacências) de um nodo, possuem alta complexidade.

- Adjacência: Para suprir algumas deficiências da representação por incidência, esta forma foi criada para possibilitar a listagem dos vizinhos da forma mais otimizada possível, pois sua representação é baseada justamente neles. Essa forma é a mais natural e simples de se representar um dígrafo.

  Pode-se utilizar, para tanto, uma matriz *nxn* (n é o número de nodos), chamada de **matriz de adjacência**, em que a posição e_{ij} matriz é preenchida com 1 se n_j é vizinho de n_i ($n_i \rightarrow n_j$); com -1 se n_i é vizinho de n_j ($n_j \rightarrow n_i$); ou ainda com 0 se tais nodos não são adjacentes nem de uma forma nem de outra. Sua principal vantagem é possibilitar que se saiba se um dado nodo é vizinho de outro, ou vice-versa, em tempo constante.

  Pode ser conveniente utilizar matrizes quando o dígrafo for denso, ou seja, quando possuir muitos arcos relativamente ao número de vértices. Todavia, para fins de

economia de espaço, sobretudo em dígrafos cujo número de arcos não é muito maior que o número de vértices (dígrafos esparsos), utiliza-se a forma **listas de adjacência**, que é, comumente, composta de várias listas encadeadas, estruturadas em um vetor.

"A ordem na qual os nodos aparecem nas listas depende do método usado para construir as listas." (SEDGEWICK, 2002, p. 17, tradução nossa). A luz disso, o vetor de listas, se tiver uma ordem pré-determinada de ocupação dos seus índices, facilita a procura por um determinado vértice. Por exemplo: um dígrafo com vértices rotulados de 0 a $n-1$, pode ter em um vetor v, a seguinte estrutura: na posição $v[0]$, estará a lista dos vértices adjacentes ao vértice de rótulo 0; na posição $v[1]$, estará a lista dos vértices adjacentes ao vértice de rótulo 1; e assim por diante até que, na posição $v[n-1]$, estará a lista dos vértices adjacentes ao vértice de rótulo $n-1$. Dessa forma, percorre-se com facilidade o dígrafo, cujo custo para tanto é relativo apenas ao número de vértices mais o número de arcos. A desvantagem se encontra no fato de as posições ocupadas pelos nodos no vetor de listas serem dependentes dos rótulos dos nodos. Ou seja, os rótulos dos nodos acabam influenciando na estrutura do dígrafo, quando, por motivos de independência de nomes, poderiam ser informações secundárias. A seção 2.1, seguinte, propõe uma solução para este problema.

A representação utilizando adjacência possui a desvantagem de exigir um algoritmo com complexidade de ordem n^2 para saber de quais nodos n um dado nodo x é vizinho ($n \rightarrow x$). Já para saber quem são os vizinhos n de um dado nodo x ($x \rightarrow n$), o custo é linear. Para se ter ambas as informações, o custo é da ordem de n^2, portanto.

Por sua vez, a implementação em listas é, no geral, mais complexa do que se utilizadas estruturas estáticas, como matrizes, devido, muitas vezes, à utilização de ponteiros. Adicionalmente, o custo para a alocação dinâmica de memória pode ser alto se comparado com o tempo gasto para se percorrer o dígrafo representado, dependendo da aplicação.

- Ainda, uma terceira forma de representar um dígrafo seria através do seu conjunto de arestas simplesmente (também chamada de **vetor-de-arcos**), em que apenas as arestas são representadas, estruturando-as como pares ordenados em matrizes ou listas encadeadas. No entanto, essa forma, por ser pouco flexível em relação a operações e consultas no dígrafo, não é muito utilizada. De fato, esta forma é, algumas vezes, utilizada como base para a implementação de estruturas de dados mais complexas e maleáveis.

Nesta seção, é proposta uma estruturação de dígrafos, com independência de rótulos, em um ambiente seqüencial. Supõe-se que o dígrafo está totalmente acessível em qualquer momento que se queira, após a entrada de dados. Neste trabalho, a implementação dos algoritmos, sobre dígrafos, utiliza tal conceito de independência.

Além disso, a estrutura de dados adotada é do tipo **listas de adjacência em matrizes** (que junta algumas vantagens do modelo **listas de adjacência**, com a facilidade da implementação em matrizes). Tal estrutura foi obtida da implementação disponível em (SKIENA; REVILLA, 2003a).

Segundo (SKIENA; REVILLA, 2003b, tradução e grifo nossos):

> Esta estrutura de dados parece combinar as piores propriedades da **matriz de adjacência** (alto gasto de memória) com as piores propriedades da **listas de adjacência** (a necessidade de procurar pela existência de uma dada aresta). Entretanto, [...] esta forma é a mais simples estrutura de dados de se programar, particularmente para grafos estáticos que não mudam após serem construídos.

2.1.1 Independência de Rótulos

Apesar de se basear na implementação de SKIENA; REVILLA, a estrutura aqui utilizada possui uma diferença em relação a ela. Fez-se com que o algoritmo fosse independente dos nomes dos rótulos. Os rótulos são apenas utilizados em operações de entrada e saída de dados. Note-se, no Anexo, nos códigos 5.2.1 e 5.2.3, que a função **get_label**, responsável por obter o rótulo de um determinado nodo, é utilizada apenas para impressão dos resultados na tela (saída de dados). Já a entrada de dados é perfeitamente inteligível apenas como um conjunto de pares-ordenados de arcos, os quais são informados por meio de rótulos, mas são armazenados em posições consecutivas do *array* de arcos (**edges[][]**), independente de seus rótulos, importando apenas a sua estrutura.

O fato de tais rótulos não influenciarem na posição ocupada pelos nodos na matriz faz com que as linhas da matriz sejam ocupadas sem "buracos" por tais nodos. Tal fato permite que se continue a utilizar a estrutura de listas de adjacência em **matrizes**, pela facilidade de acessá-la e de programá-la, sem perder sua dinamicidade e economia de memória.

Portanto, propõe-se uma representação mais estrutural, em que os rótulos não são armazenados junto da estrutura do grafo, mas sim como informação adicional, em um *array* separado chamado **lvertices[]**, cuja declaração e uso se encontram no código 5.1.1 do Anexo. Dessa forma, as relações entre os nodos são destacadas, sem dar importância aos nomes. Além disso, os rótulos dos nodos não precisam, necessariamente, serem nomeados de 0 a $N-1$ (em que N é o número de nodos do grafo), podendo-se rotulá-los com números naturais quaisquer, independente da observância de ordem.

2.1.1.1 Modelo de Dígrafo

A entrada de dados que define o dígrafo pretendido é totalmente definida somente através de arcos (ligações unidirecionais). Dessa forma, nodos isolados são dispensáveis, ainda mais que se está a modelar redes de computadores, que têm aqui no trabalho especial interesse de aplicação. Um nodo isolado em uma rede não tem razão de ser; ele não se comunica com nenhuma outra parte da rede, logo não pertence à rede.

Um dígrafo, quando visto como uma rede (ver capítulo 3), só existe enquanto entidade que relaciona, unidirecionalmente, nodos uns aos outros através de arcos. Não existem nodos isolados, destacados um dos outros e independentes, mas como um todo unido, coerente, que se vai descobrindo em etapas, durante a descoberta da rede.

(SEDGEWICK, 2002, p. 09, tradução e grifo nossos), ao aprofundar seu pensamento em relação à representação de grafos não-orientados, admite a possibilidade de se conceber um grafo apenas enquanto arestas, dispensando os nodos:

> Podemos desenhar um grafo marcando pontos para os vértices e desenhando linhas que conectam-nos às arestas. Um desenho nos dá uma intuição sobre a estrutura do grafo; mas essa intuição pode ser **enganatória**, porque o grafo é definido independente de sua representação. [...] Embora seja **suficiente considerar um grafo simplesmente como**

> **um conjunto de arestas**, examinaremos outras representações que são particularmente adequadas como base para estruturas de dados.

Mais adiante, SEDGEWICK ainda afirma: "**Um grafo não é nada mais nada menos do que seu conjunto de arestas**, e freqüentemente é necessário um método para recuperar o grafo nessa forma, a despeito da sua representação interna. Pode-se considerar uma representação do tipo **vetor-de-arestas** como base para uma implementação." (SEDGEWICK, 2002, p. 18, tradução e grifo nossos)

Adicionalmente, os algoritmos aqui descritos ignoram **arcos paralelos**. Arcos ligando os mesmos nodos, no mesmo sentido, são exatamente os mesmos, o que não faz sentido na generalidade em que se quer propor o conceito de rede, em que *links* paralelos não possuiriam nenhum tipo de diferenciação, já que aqui não se considera nenhum tipo de peso (ou velocidade) para eles. Logo, opta-se por não se trabalhar com **multidígrafos**, mas sim com **dígrafos simples**, por definição. Também para (FEOFILOFF, 2008e), "arcos paralelos podem ser desconsiderados. Poderíamos tentar dizer que dois arcos são paralelos se têm a mesma ponta inicial e a mesma ponta final. Mas esse conceito não faz sentido, uma vez que arcos são meros pares de vértices e portanto dois arcos diferentes não podem ter a mesma ponta inicial e a mesma ponta final." Por sua vez, SEDGEWICK "[...] permite arcos paralelos, tratando-os como 'cópias' distintas de um mesmo arco" (SEDGEWICK, 2002) apud (FEOFILOFF, 2008e).

Entretanto, para fins de robustez e coerência com a aplicação em redes, as implementações aqui desenvolvidas reconhecem os **laços** do dígrafo. Laços são *links* que ligam um nodo a ele mesmo, podendo ser entendido como uma ligação do tipo *loopback*, cujo papel é receber todo tráfego enviado por si mesmo. De acordo com (WIKIPEDIA, 2008e), "O Internet Protocol (Endereço IP) define uma rede loopback. No IPv4, esta deve ser a rede 0 ('a própria rede') e um endereço de loopback para o computador atual — 0.0.0.0 — ('este computador nesta rede')".

2.2 Travessia

Conforme (CORMEN et al., 2001, p. 527, tradução e grifo nossos),

> Busca (travessia ou pesquisa) em grafo significa sistematicamente seguir suas arestas a fim de visitar seus vértices. Um algoritmo de busca em grafo pode descobrir muito sobre a estrutura do grafo. Muitos algoritmos percorrem grafos para obterem suas estruturas. Outros algoritmos de grafos são organizados como simples elaborações de algoritmos básicos para busca em grafos.

A ação principal da maioria das buscas em um grafo, ao atravessar suas arestas, é descobrir cada vértice do grafo exatamente uma vez. Dessa forma, garante-se a construção de uma **árvore geradora** (***spanning tree***), que se caracteriza por conter todos os vértices do grafo (não-orientado) sem conter ciclos.

As descrições e definições feitas neste capítulo, entretanto, serão direcionadas a dígrafos, já que estas são as estruturas prioritárias de atuação do trabalho. Neste caso, ao invés de uma árvore geradora, tem-se a definição de uma **arborescência geradora** (abreviadamente **AG**).

Nesta seção, serão vistos basicamente dois tipos mais conhecidos de busca seqüencial em grafos: **Busca em Profundidade (DFS)** e **Busca em Largura (BFS)**. O intento, neste trabalho, em estudar tais métodos consagrados é que serão de grande valia para um entendimento posterior de como percorrer uma rede, de forma distribuída, à procura de

ciclos, além de serem comparados ao ADDC.

2.2.1 Definições

Muitos algoritmos, tais como descoberta de caminho mínimo, construção de *spanning tree*, detecção de componentes conexos e fortemente conexos e, destacadamente, detecção de ciclos, utilizam, ou podem utilizar, como base dois tipos fundamentais de algoritmos de travessia em grafos: o *Breadth-Fist Search* ou BFS ("Busca primeiro em largura") e o *Depth-First Search* ou DFS ("Busca primeiro em profundidade").

Conforme escreve (SKIENA; REVILLA, 2003b, p. 194), "A operação básica na maioria dos algoritmos em grafos é percorrer o grafo completamente e sistematicamente. Nós queremos visitar todo vértice e toda aresta exatamente uma vez em uma ordem bem-definida." (SKIENA, 2007b, p. 9, traduçao nossa).

Tanto BFS como DFS são aplicados em estruturas de grafo que, no geral, são definidos em função de seus nodos. Cada nodo, durante a execução do algoritmo, assume, em cada instante, um determinado estado. Quando um nodo é pela primeira vez visitado (descoberto) diz-se que ele está no estado **descoberto**. O estado **processado** geralmente é atribuído ao nodo somente após todos os seus vizinhos (no caso de BFS) ou todo o seu território (no caso de DFS) ter(em) sido processado(s). Já os arcos, como são definidos e processados em função dos nodos, não possuem, no geral, estados. De fato, os arcos são processados no mesmo instante em que são encontrados.

Nas próximas seções, têm-se algumas características e diferenças entre os dois algoritmos citados.

2.2.1.1 DFS

Consiste na varredura do dígrafo utilizando busca em profundidade. Para cada novo nodo encontrado, uma chamada recursiva é feita, ordenando-se os nodos explorados em uma pilha. Cada novo nodo encontrado é agregado à arborescência geradora do dígrafo a se formar. [1] Possui, assim como BFS, muitas aplicações, que serão citadas e utilizadas neste trabalho, especialmente a que se refere ao seu talento natural em detectar se um dígrafo possui ciclo ou, caso contrário, é um **DAG (Dígrafo Acíclico)**.

SEDGEWICK compara estratégias de travessia em um grafo com a "exploração de um labirinto". Ele o faz quando da introdução da busca DFS em grafos não-orientados: "Nosso interesse na exploração de *Tremaux* (exploração de labirinto) é que esta técnica nos leva imediatamente à clássica função recursiva para travessia de grafos: [...] DFS. [...] DFS é, ilusoriamente, simples porque é baseado em um conceito familiar e é fácil de implementar; de fato, ele é um sutil e poderoso algoritmo que pode ser usado para numerosas tarefas difíceis de processamento de grafo." (SEDGEWICK, 2002, p. 81, grifo e tradução nossa)

DFS, enquanto algoritmo de busca seqüencial, possui complexidade igual a $O(E)$ para detectar ciclos em grafos não-orientados, ou seja, é linear em relação ao número de arestas do grafo, desde que a estrutura de dados utilizada seja listas de adjacência.

DFS, tipicamente, escolhe aleatoriamente um nodo inicial para iniciar a busca. Por isso é também muito utilizado para detecção de compontentes conexas, bastando que se reinicie o algoritmo para nodos não alcançados nas execuções anteriores. Tal fato faz com que cada reaplicação do algortimo defina uma componente conexa.

[1] Alguns autores (CORMEN et al. lhe dá o adjetivo "DFS", chamando-a, por exemplo, de árvore-DFS. Aqui, chamar-se-ia arborescência-DFS.

2.2.1.2 Classificação de Arcos

Segue a classificação de arcos gerada por uma execução DFS:

- **Arco Descendente**: conecta um vértice a outro vértice descendente na arborescência, ou seja, desce na arborescência.

  Em dígrafos, são dois os arcos descendentes:

  Arco de Arborescência: sai do pai e entra no filho. Como arcos de arborescência têm seu destino em um nodo recém descoberto, o nodo que o descobre é chamado **pai**, e o nodo descoberto é chamado **filho**.

  Arco *Forward*: qualquer arco descendente que não seja arco de arborescência.

- **Arco Ascendente**: conecta um vértice a outro vértice ancestral na arborescência, ou seja, sobe na arborescência.

  Em dígrafos, são dois os arcos descendentes:

  Arco-Pai: tem origem no pai e destino no filho.

  Arco de Retorno: qualquer arco ascendente que não seja arco-pai.

- **Arco Cruzado**: diz-se que $v - w$ é um arco cruzado, se w não é ancestral nem descendente de v. Equivalentemente, v não é ancestral nem descendente de w.

Aqui também, um arco ascendente encerra um ciclo. Um arco-pai sempre define um ciclo direto de comprimento dois; já um arco de retorno sempre define um ciclo direto de comprimento maior que dois.[2]

Teorema 2.2.1 (DFS — Detecção de Ciclo) *Em uma busca DFS em um dígrafo qualquer, um arco ascendente $y \rightarrow x$ sempre define um ciclo direto formado pelo arco em ascendência $y \rightarrow x$, mais o caminho direto que começa no nodo x e vai até o nodo y, constituído pelos arcos da arborescência, que se encontram enumerados na pilha, no momento do processamento do arco de retorno.*

Entretanto, DFS possui a limitação de não conseguir enumerar exatamente os ciclos existentes. O teorema 2.2.1 acima pode incluir (significar), implicitamente, a existência de outro(s) ciclo(s), não-detectável(is), além do ciclo explicitado pelo Teorema. De fato, após a detecção de tal ciclo, qualquer rota descendente alternativa, que vá de um ponto a outro da arborescência que compôs o ciclo detectado (x a y) define também um ciclo.

[2] SEDGEWICK considera, em dígrafos, arcos de retorno como todos os arcos que formam ciclos, logo incluem arcos de retorno propriamente ditos (entendidos como os considerados neste trabalho: qualquer arco de retorno que não seja arco-pai) mais os arcos-pai. Aqui neste trabalho, estes dois tipos de arcos constituem um superconjunto chamado de **arcos ascendentes**. Já em grafos não-orientados, SEDGEWICK faz a discriminação de praxe, ou seja, arcos de retorno referem-se somente a arcos ascendentes que não sejam arcos-pai, devido ao fato de, em grafos não-orientados, arcos-pai não definirem ciclos (de fato, definem **ciclos triviais**, desconsiderados por SEDGEWICK enquanto ciclos). Ele utiliza este tipo de classificação para poder afirmar que "todo arco de retorno (***back link***) forma um ciclo, e vice-versa" (independente do tipo de grafo que esteja tratando).

2.2.1.3 BFS

BFS consiste em varrer todo o dígrafo através de uma busca em largura. Tal fato faz com que esteja relacionado com o conceito de distância entre vértices. Tipicamente, um nodo inicial é escolhido pelo usuário e então se inicia o algoritmo, descobrindo primeiramente todos os vizinhos de tal nodo. Os nodos que vão sendo descobertos são colocados em uma fila. Um nodo é dito descoberto (ou visitado) quando ele é encontrado pela primeira vez. O arco que tem como destino um nodo recém-encontrado é chamado de arco de arborescência. O nodo origem deste arco é tido como o pai do nodo destino do arco. Tal informação é, geralmente, registrada em um **vetor de pais**. Tais tipos de arcos são assim chamados pois formam uma arborescência (conhecida também como árvore radicada ou árvore direcionada). [3]

(CORMEN et al., 2001, p. 531) indica a complexidade do método BFS para se percorrer um grafo completamente: $O(V+E)$ (linear em relação à lista de adjacência representativa do grafo G)".

BFS tem como principal característica o fato de que quaisquer dois nodos pertencentes à arborescência construída pelo algoritmo possuem um caminho mínimo entre si. "A busca BFS é apropriada se se está interessado em caminhos mínimos em dígrafos sem pesos nos arcos." (SKIENA, 2007b, p. 9, tradução nossa).

Segundo (LYNCH, 1996), a pesquisa em largura (BFS) minimiza o custo de comunicação em uma rede, desde que se assuma que todo canal tem custo constante (tempo constante).

(FEOFILOFF, 2008f) resume bem o comportamento da fila em relação ao estado de um vértice ao concluir que todo vértice que está na fila já foi descoberto; e se um vértice v já foi descoberto, mas algum de seus vizinhos ainda não foi descoberto, então v está na fila.

Da mesma forma que em DFS, é possível classificar os arcos de um dígrafo percorrido por uma busca BFS, já que tais buscas se diferenciam essencialmente na ordem em que os arcos são adicionados à arborescência. Na literatura consultada, a classificação de arcos em busca BFS não foi encontrada. Portanto, supõe-se serem inéditas as relações feitas a seguir entre DFS e BFS, em termos de classificação de arcos.

Os tipos de arcos em BFS possuem o mesmo conceito de DFS. Contudo, é interessante notar que BFS, em vista da sua própria definição, não aceita a existência de arcos *forward*. Um arco forward é do tipo descendente, que conecta nodos "avós" x com nodos "netos" z (ou "bisavós" com "bisnetos" ou assim por diante). Logo um arco $x-z$ deste tipo conectaria o nodo x ao nodo z através de um caminho de comprimento igual a um, o qual é menor do que o caminho (necessariamente maior que um) de x a z através dos arcos da arborescência, funcionando como uma espécie de "atalho" para alcançar um nodo, o que é uma contradição, pois todos os arcos da arborescência em BFS representam ou são parte de um caminho mínimo entre os nodos.

A classificação dos arcos em BFS é idêntica a feita em DFS, com a diferença de não existir, conforme explicado, arcos *forward*. Portanto, tal classificação é aqui omitida. Note-se, ainda, que, em BFS, arcos de retorno existem, desde que não possuam arco antiparalelo, uma vez que, caso possua, tal par de arcos, é, na verdade, formado por um arco de arborescência juntamente com um arco-pai antiparalelo. Pode-se entender intuitivamente este fenômeno pelo fato de não existir, em BFS, o dual de um arco de

[3] (CORMEN et al., 2001, p. 532) lhe dá o adjetivo "BFS", chamando-a, por exemplo, de árvore-BFS. Aqui, chamar-se-ia arborescência-BFS.

retorno: o arco *forward* (ver subseção 2.2.2 abaixo). Logo, um arco de retorno só existe se não estiver acompanhado de seu arco antiparalelo (dual), fato só concebível em dígrafos não-simétricos, em que todo arco não precisa necessariamente ter "seu par" antiparalelo.

2.2.2 Dualidade

Independente de se estar ou não tratando de dígrafos simétricos, nos quais o conceito de **antiparalelo** ou **dual** é naturalmente percebido, pode-se pensar os diferentes tipos de arcos enquanto conceitos inversos (duais). Visualize-se as relações:

Arco-pai é o **dual** de **arco de arborescência**;

Arco de retorno é o **dual** de **arco *forward***;

O **dual** de **arco cruzado** é **ele próprio**. (Entretanto, em DFS, arcos cruzados não possuem arcos duais.)

Na maioria das vezes, a idéia de dualidade facilita uma visão mais universal de conceitos aparentemente sem relação. Como exemplo, a implementação aqui desenvolvida para classificação de arcos durante a pesquisa BFS, primou por um raciocínio dual, o que certamente facilitou a criação do bloco de código responsável por esta classificação, conforme se pode constatar no Anexo, código 5.2.1.

2.2.2.1 Comparações

A diferença básica entre a busca em largura e a busca em profundidade está na estrutura de dados auxiliar empregada pelas diferentes estratégias de pesquisa. A busca em largura usa uma fila FIFO (de vértices), enquanto a busca em profundidade usa uma pilha. Na versão recursiva da busca em profundidade, a pilha não aparece explicitamente pois é administrada pelo mecanismo de recursão.

Há várias outras diferenças, mais superficiais, entre os dois algoritmos:

- Na busca em profundidade, o próprio algoritmo escolhe o vértice inicial, enquanto a busca em largura começa tipicamente num vértice especificado pelo usuário.
- A busca em profundidade pode visitar, se várias chamadas forem feitas ao método, todos os vértices do dígrafo, mesmo aqueles não "alcançáveis" a partir do vértice inicial. Por sua vez, a busca em largura visita apenas os vértices que podem ser alcançados a partir do vértice inicial.
- BFS geralmente é descrito por meio de um algoritmo iterativo, enquanto DFS é, usualmente, descrito de forma recursiva.

3 ADDC

Neste capítulo é formulado um algoritmo distribuído que dê conta de detectar ciclos em uma rede, representada por um dígrafo. Primeiramente, são enumerados alguns desafios a serem enfrentados; após o ambiente esperado pelo algoritmo é descrito; na seção seguinte, as estruturas de dados e os procedimentos executados sobre tais estruturas são descritos em pseudo-C; posteriormente, tais procedimentos são divididos em tarefas concorrentes, para finalmente o algoritmo completo ser formalizado em pseudocódigo, juntamente com o cálculo de sua complexidade e prova de sua corretude e terminação.

3.1 Desafios

Um dos desafios da modelagem do algoritmo pretendido é que ele é aplicado sobre redes estruturadas como dígrafos, os quais representam, de uma melhor forma, *links* que podem estar disponíveis em apenas um sentido. (SEDGEWICK, 2002, pp. 31,142) ressalta que a assimetria faz dos dígrafos objetos combinatórios mais complexos quando comparados a grafos não-orientados, citando a dificuldade de, em dígrafos, se encontrar diretamente todos os arcos que têm destino em um determinado vértice.

A questão da conectividade (ou alcançabilidade) complica sobremaneira em dígrafos, e é um assunto essencial em redes. "Ter tal entendimento é mais complicado para dígrafos do que para grafos não-orientados. Por exemplo, deveria ser possível, em uma olhada, dizer se um pequeno grafo não-orientado é conexo ou se contém um ciclo; essas propriedades não são tão fáceis de serem reconhecidas em dígrafos." (SEDGEWICK, 2002, p. 145, tradução nossa)

Na literatura existem algoritmos seqüenciais bem estudados, de fácil implementação e pouca complexidade, para detecção de ciclos em dígrafos. Entretanto, em um ambiente distribuído de rede, não se conhece, de vista, a rede por completo, diferentemente do que ocorre com algoritmos de busca seqüencial em grafos, em que, geralmente, se conhece a estrutura completa do grafo antecipadamente, tão logo os dados de entrada tenham sido informados.

Sabe-se também que em um ambiente com memória distribuída, a informação relevante final (no caso, a listagem de ciclos), ou parte dela, pode estar presente em vários pontos da rede. O algoritmo, por conseguinte, deve dar conta de repassar tal informação ao nodo interessado nela. No ADDC, é, tipicamente, o nodo inicial, que a requisitou, o nodo que, ao final da execução, deve possuir as informações sobre os ciclos. Constata-se que, pela natureza do algoritmo de difusão e da estrutura do dígrafo, naturalmente essa característica é obtida.

3.2 Ambiente e Modelagem

Nesta seção, são descritos o ambiente distribuído esperado, o modelo de comunicação e programação do ADDC e sua descrição informal, juntamente com seus objetivos.

3.2.1 Definição do Ambiente

ADDC é desenvolvido para funcionar em um ambiente distribuído, tipicamente uma rede modelada como um dígrafo. Como em dígrafos os *links* são direcionados, a rede é modelada através de canais de comunicação unidirecionais entre os nodos. Supõe-se que cada nodo x que possui y como vizinho ($x \rightarrow y$), possui um canal de comunicação que vai de x a y. Caso y possua x como vizinho ($y \rightarrow x$), deve existir um canal em sentido contrário. O acesso aos canais de comunicação com os vizinhos é possível através de um id. Portanto, a topologia da rede é desconhecida dos nodos, pois cada nodo é capaz, apenas, de se comunicar com seus vizinhos.

Os canais de comunicação entre os nodos são assumidos como confiáveis. Ou seja, não duplicam nem perdem mensagens. Além disso, são, por simplicidade, assumidos como FIFO. A comunicação de dados, pertencente ao nível de rede, é abstraída, bem como a camada de hardware é desconsiderada.

Adicionalmente, um ambiente real de rede, em que *links* possuem velocidades diferentes, e nodos possuem latências variadas, deve ser suportado. A rede deve ser estática; logo, mudanças em sua topologia, no decorrer do algoritmo, não são suportadas.

3.2.2 Modelo de Comunicação e Programação

A comunicação entre os diferentes nodos da rede é feita, explicitamente, através de **troca de mensagens**. Tanto o envio quanto o recebimento de mensagens é **assíncrono e não-bloqueante**, além de preservar a ordem FIFO dos canais de comunicação. A programação do algoritmo, distribuída nos diferentes nodos da rede, é do tipo ***multitask***. Cada *task* é considerada **atômica**, ou seja, não pode ser interrompida pela CPU até ser finalizada.

Cada *task* é responsável por uma tarefa independente do ADDC. Resumidamente, uma *task* é responsável pelo envio de mensagens, outra pelo recebimento e processamento de mensagens, e outra pelo encaminhamento delas. A comunicação entre as diferentes *tasks* dentro de um mesmo nodo, quando necessária, é feita através da escrita e leitura a um **buffer de sistema**, preservando, por simplicidade, a ordem FIFO. Tal acesso compartilhado é assumido como sincronizado e consistente.

Tasks que precisam ler ou receber mensagens, por serem não-bloqueantes, são implementados através de um *loop* que é executado até que um dado seja lido ou uma mensagem recebida.

3.2.3 Descrição do ADDC

A principal característica de funcionamento do ADDC é a difusão de informações. Um nodo, previamente marcado como inicial, interessado em descobrir, e armazenar, todos os ciclos que lhe incluem, dá início a uma difusão. Tal nodo difunde, a todos os seus nodos vizinhos, uma mensagem com a informação de seu rótulo. Uma mensagem gerada inicialmente por um nodo inicial X é chamada de **mensagem iniciada por** X. Cada nodo que receber essa mensagem adiciona seu rótulo a ela e a encaminha também para cada um de seus vizinhos. Após a adicionar seu rótulo, um nodo **não** encaminha a mensagem recebida (interrompe-a) caso ele já a tenha enviado anteriormente. Nesse

momento, tem-se a detecção de uma **mensagem cíclica**. Ou seja, uma mensagem cíclica é detectada se ela já passou anteriormente por este nodo (foi por ele encaminhada) ou foi iniciada anteriormente por ele.

3.2.3.1 Detecção de Ciclos no ADDC

Uma mensagem cíclica, quando detectada (e, conseqüentemente, interrompida), é analisada, pelo nodo que a detectou, de forma a saber se ela deve ser armazenada ou descartada. Ela é armazenada (indicando a detecção de um ciclo) caso tenha sido iniciada pelo mesmo nodo que a detectou; em caso contrário, é descartada. A ação de armazenamento ou descarte, a ser tomada, por um nodo y detector de uma mensagem cíclica, está de acordo com as definições 3.2.1 e 3.2.2.

Definição 3.2.1 *Uma mensagem cíclica iniciada por x, e detectada por y, é **armazenada** por y se e somente se* $y = x$.

Definição 3.2.2 *Uma mensagem cíclica iniciada x, detectado por y, é **descartado** por y se e somente se* $y \neq x$.

A figura 3.1 mostra a formação de duas mensagens cíclicas iniciadas por A. A mensagem cíclica "A,B,C,D,A" (que é também um ciclo[1], à esquerda, é detectada pelo nodo A ($A = A$), e, portanto, armazenada por ele. Já a mensagem cíclica "A,B,C,D,B" (que não é um ciclo), à direita, é detectada pelo nodo B ($B \neq A$), e, portanto, descartada por ele.

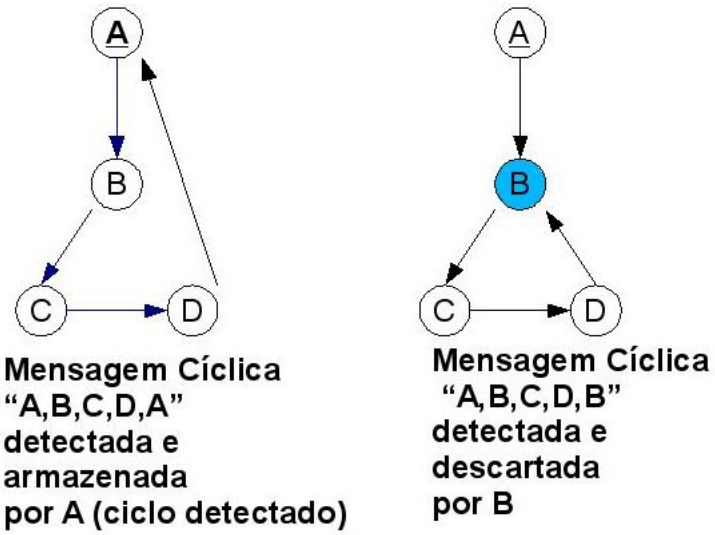

Figura 3.1: armazenamento e descarte de mensagens cíclicas

Note-se que uma mensagem cíclica não possui o seu último nodo coincidente, necessariamente, com o primeiro, pois pode coincidir com outro qualquer. No caso de coincidir com o primeiro, tem-se o caso de um ciclo, que é sempre simples (os únicos nodos repetidos são o inicial e o final).

Pode-se provar que toda mensagem armazenada é um ciclo simples, e que tal armazenamento ocorre apenas no nodo inicial. Simetricamente, todos os ciclos simples que

[1] o termo **ciclo** não precisa vir acompanhado do adjetivo **simples** para ser entendido como tal, já que, em ADDC, todo ciclo detectado é sempre simples.

incluem tal nodo inicial, e somente estes, são por ele armazenados. A prova da corretude do ADDC, juntamente com a prova de que ele termina, estão disponíveis na seção 3.7.1.

Além disso, por estar se tratando de dígrafos, os ciclos formados são sempre **dirigidos**, ou seja, possuem um sentido bem-determinado. O armazenamento de um ciclo em um sentido não informa nada a respeito da existência de outro ciclo no sentido contrário.

3.2.3.2 Multidifusão

ADDC suporta a existência de mais de um nodo inicial interessado na detecção de ciclos na mesma rede. Assim sendo, um nodo inicial pode estar a difundir sua mensagem no mesmo momento que outro, de forma que existam, na mesma rede, várias difusões simultâneas. Por isso, pode-se dizer que ADDC trabalha com multidifusões.

A figura 3.2 mostra uma rede com vários pontos de disparo de difusão.

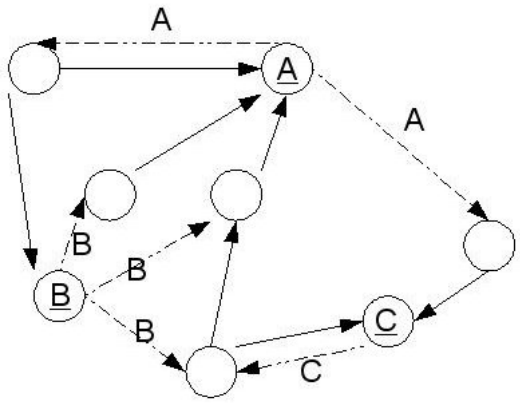

Figura 3.2: Nodos iniciais A, B e C iniciando a busca por ciclos

3.3 Estruturas e Procedimentos

Esta seção define, em pseudocódigo (pseudo-C), as estruturas de dados presentes no ADDC bem como uma biblioteca de procedimentos padrão, além de primitivas executados sobre tais estruturas. Tais procedimentos são, posteriormente, utilizados pelas *tasks* que implementam o ADDC.

3.3.1 Estruturas de Dados

3.3.1.1 Mensagem

As mensagens, no ADDC, são representadas por meio de um vetor de rótulos (*char* $m[1 : N+1]$), em que cada posição é ocupada, ordenadamente, por um rótulo. A figura 3.3 ilustra a adição de rótulos ao vetor m ao longo do tempo.

O vetor m, idealmente, deveria possuir tamanho igual a $N+1$ tal que N é o número de nodos da rede. Tal tamanho garantiria o armazenamento, sem estouro, do maior ciclo simples possível de ser formado. Ou seja, uma mensagem cíclica que inclui todos os nodos da rede, mais a repetição do inicial. Entretanto, o número de nodos da rede é um dado não conhecido, portanto é assumido que o vetor m seja "tão grande quanto se queira".

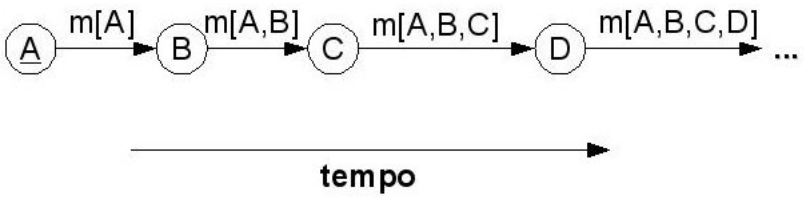

Figura 3.3: Nodos encaminhando mensagem enviada pelo nodo inicial A

3.3.1.2 Ciclo

Os ciclos, representados por vetores *m*, detectados pelos nodos iniciais são armazenados em uma matriz bidimensional (vetor de vetores de *m*). Portanto, cada nodo inicial deve conter uma estrutura da seguinte forma: $char\ ciclos[1:M][1:N+1]$, em que cada linha armazena a informação de um ciclo. O número máximo de ciclos (limitado por M linhas) e o comprimento máximo de cada ciclo (limitado por $N+1$ colunas) depende, respectivamente, da topologia e do número de nodos da rede. Logo, assim como na estrutura de mensagem *m*, a matriz *ciclos* deve, na prática, ser "tão grande quanto se queira".

3.3.2 Procedimentos

3.3.2.1 Troca de Mensagens

A troca de mensagens é implementada através de primitivas do tipo *send* e *receive* não-bloqueantes. Uma primitiva *send* tem como argumento uma mensagem a ser enviada e o id do canal de comunicação com o nodo destino. Uma primitiva *receive* tem apenas como o argumento uma mensagem a ser recebida. Note-se que o ADDC precisa das açoes "enviar para todos os vizinhos" e "receber todas as mensagens", as quais são, respectivamente, implementadas, simplesmente, por meio de um laço de *sends* e um de *receives*. A diferença consiste no fato de que, em um *send*, deve-se informar o destino da mensagem a ser enviada, enquanto que um receive recebe, indistintamente, qualquer mensagem disponível no canal.

3.3.2.2 Adicionar Rótulo

Este procedimento implementa a adição de um rótulo à próxima posição livre da mensagem. Serve tanto para o nodo inicial adicionar seu rótulo a uma mensagem inicial, recém criada, quanto para que cada nodo subseqüente que receba tal mensagem adicione seu rótulo a ela.

A implementação deste procedimento está disponível no codigo 3.3.1.

```
1 char[] adicionar_rotulo(char mylabel, char[] m) {
2   m[sizeof(m)+1] := mylabel;
3   return m;
4 }
```

Código 3.3.1: *Implementação do Procedimento* ***Adicionar Rótulo***

Ele recebe como argumentos um rótulo e uma mensagem e retorna uma mensagem cuja última posição livre foi preenchida pelo rótulo informado.

3.3.2.3 Enviar Mensagem Inicial

Este procedimento consiste no envio de uma mensagem que contém, em sua primeira posição, o rótulo do nodo que a está enviando. Ele será usado pelo procedimento, disponível em nodos iniciais, responsável por implementar o início da difusão de mensagens. A implementação do procedimento **Enviar Mensagem Inicial** está disponível no codigo 3.3.2.

```
1 void enviar_mensagem_inicial(int id, char mylabel) {
2   var char mensagem[] := null;
3
4   mensagem := adicionar_rotulo(mylabel, mensagem);
5   send(id, mensagem);
6 }
```

Código 3.3.2: *Implementação do Procedimento* ***Enviar Mensagem Inicial***

Ele recebe como argumento o id do canal em que a mensagem deve ser enviada e o rótulo do nodo que a quer enviar; cria uma mensagem, adiciona o rótulo recebido a ela (fazendo uso do procedimento **Adicionar Rótulo**) e envia uma mensagem através da chamada da primitiva *send*, passando como parâmetros o id recebido e a mensagem retornada por **Adicionar Rótulo**.

3.3.2.4 Receber Mensagem

Tal procedimento consiste, simplesmente, em um laço a espera de que uma mensagem seja recebida. Sua implementação se encontra no código 3.3.3

```
1 void receber_mensagem(char[] m) {
2   do {
3     receive(m);
4   } while (m == null);
5 }
6
```

Código 3.3.3: *Implementação do Procedimento* ***Receber Mensagem***

3.3.2.5 Detectar Mensagem Cíclica

Este procedimento tem como objetivo detectar se uma mensagem recebida por um nodo *y* qualquer é uma mensagem cíclica, ou seja, se *y* se repete na mensagem. O código 3.3.4 mostra sua implementação:

```
1 bool detectar_mensagem_ciclica(char[] m) {
2   for (i=1; i<sizeof(m)-1; i++) {
3      if (m[i] == m[sizeof(m)])
4        return true;
5     }
6     return false;
7 }
```

Código 3.3.4: *Implementação do Procedimento* ***Detectar Mensagem Cíclica***

Ele recebe como argumento uma mensagem (em que a última posição já contém o rótulo do nodo que a recebeu); verifica se a última posição da mensagem se repete em alguma outra posição dela, retornando verdadeiro se se repete, e falso em caso contrário.

3.3.2.6 Armazenamento e Descarte

As operações de armazenamento ou descarte de uma mensagem cíclica detectada são feitas pelos procedimentos descritos abaixo:

O procedimento **Armazenar Ciclo**, código 3.3.5, simplesmente, recebe como parâmetro uma mensagem, e a armazena na próxima linha livre da matriz *ciclos*.

```
1 void armazenar_ciclo(char[]  m) {
2   ciclos[sizeof(ciclos[][])] := m;
3 }
```

Código 3.3.5: *Implementação do Procedimento* ***Armazenar Ciclo***

O procedimento **Descartar Mensagem**, código 3.3.6, basta "não-fazer nada" com a mensagem recebida como parâmetro, já que, ao não armazená-la nem encaminhá-la, estará, virtualmente, descartando-a.

```
1 void descartar_mensagem(char[] m) {
2   // basta não fazer nada
3 }
```

Código 3.3.6: *Implementação do Procedimento* ***Descartar Mensagem***

3.3.2.7 Encaminhamento de Mensagem

A ação de encaminhar uma mensagem recebida é implementada por meio de dois procedimentos e de uma primitiva *send*. O procedimento **Escrever no Buffer** é encarregado pela escrita de mensagens em um ***buffer* de sistema**; o procedimento **Ler do *Buffer*** é encarregado pela leitura dessas mensagens; a primitiva *send* faz o envio das mensagens lidas.

O procedimento **Escrever no Buffer**, disponível no código 3.3.7, recebe como parâmetro uma mensagem, e a escreve no *buffer* de sistema.

```
1 void escrever_buffer(char[] m) {
2  /* system_buffer possui acesso compartilhado
3   * sincronizado e consistente */
4   write(m, system_buffer);
5 }
```

Código 3.3.7: *Implementação do Procedimento* ***Escrever no Buffer***

O procedimento **Ler do Buffer**, disponível no código 3.3.8 faz um acesso de leitura ao *buffer* de sistema. Se não há nenhuma mensagem no *buffer*, é retornado valor nulo; caso contrário, a mensagem lida é retornada.

```
1  char[] ler_buffer() {
2    var char mensagem[] := null;
3
4    /* system_buffer possui acesso compartilhado
5    * sincronizado e consistente */
6    mensagem := read(system_buffer);
7    return mensagem;
8
9  }
```

Código 3.3.8: *Implementação do Procedimento* ***Ler do Buffer***

3.4 Tasks

Os procedimentos listados na seção 3.3 são agora utilizados na implementação, em pseudo-C, de *tasks* atômicas, as quais são as tarefas concorrentes constituintes do ADDC. São três as *tasks* implementadas: *task* 1, responsável por iniciar o envio de mensagens (início de difusão); *task* 2, responsável por receber e processar mensagens; e *task* 3, responsável por encaminhar mensagens.

3.4.1 *Task* 1: Iniciar Difusão

Esta *task* faz uso do seguinte procedimento:

- Enviar Mensagem Inicial

Essa *task* implementa o início de uma difusão. Consiste, simplesmente, em um "*loop* de *sends*" para cada um dos vizinhos. Portanto, o procedimento **Enviar Mensagem Inicial** é chamado tantas vezes quantas forem o número de vizinhos, passando como parâmetro o id do canal de comunicação do nodo para o qual a mensagem é, a cada chamada, enviada.

O código 3.4.1 descreve esta *task*.

```
1  task iniciar_difusao() {
2    const char mylabel := "x";
3    var int id;
4
5    forall id in channels[] {
6      enviar_mensagem_inicial(id, mylabel) {
7    }
8  }
```

Código 3.4.1: *Implementação da Task 1*

3.4.2 *Task* 2: Processar Mensagem Recebida

Esta *task* faz uso dos seguintes procedimentos:

- Receber Mensagem
- Detectar Mensagem Cíclica
- Armazenar Ciclo
- Descartar Mensagem
- Escrever no Buffer

Portanto, ela, além do recebimento, é também responsável pelo processamento das mensagens (detecção e armazenamento ou descarte) e pela escrita no *buffer* de sistema. O encaminhamento de fato (leitura do *buffer* e envio) é feito pela *task* 3.

Na figura 3.4, tem-se o fluxograma desta *task*; no código 3.4.2, sua implementação.

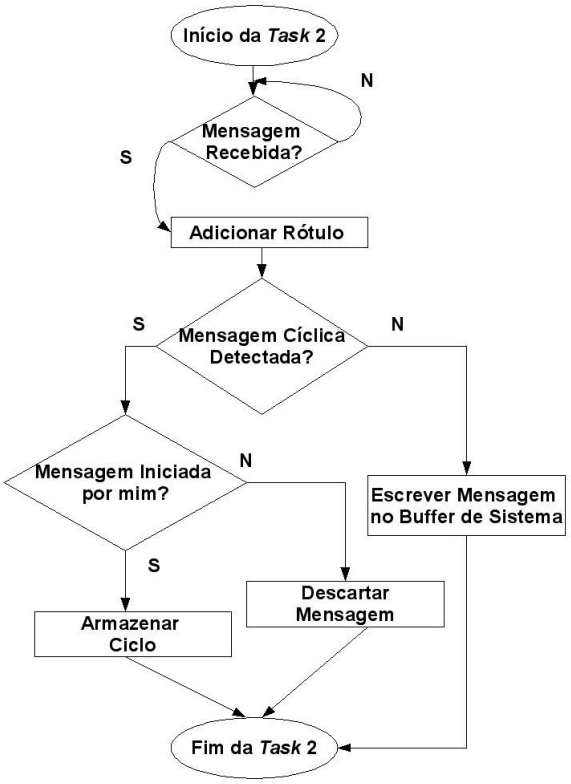

Figura 3.4: Fluxograma da *Task* 2

```
1  task processar_mensagem_recebida() {
2    const char mylabel := "x";
3    var char mensagem[] := null
4
5    receber_mensagem(mensagem);
6    mensagem := adicionar_rotulo(mylabel, mensagem);
7    if (detectar_mensagem_ciclica(mensagem) == true) {
8      if (mensagem[1] == mylabel) {
9        // mensagem iniciada por mim
10       armazenar_ciclo(mensagem);
11     }
12     else {
13       descartar_mensagem(mensagem);
14     }
15   }
16   else {
17    // não foi detectada mensagem cíclica
18    // colocá-la no buffer de sistema para ser encaminhada
19    escrever_buffer(mensagem);
20   }
21 }
```

Código 3.4.2: *Implementação da Task 2*

Visualize-se, no código 3.4.2, que após detectar uma mensagem cíclica, um nodo precisa verificar se tal mensagem foi ou não por ele iniciada. Para fazer essa verificação,

o nodo compara seu rótulo com o rótulo presente na primeira posição da mensagem. Se for igual, armazena a mensagem; se não for, descarta-a. No caso da mensagem recebida não ser cíclica, ela é escrita no *buffer* de sistema, para que a *task* 3 possa encaminhá-la.

3.4.3 *Task* 3: Encaminhar Mensagem

Esta *task*, além da utilização de uma primitiva *send*, faz uso do seguinte procedimento:

- Ler do Buffer

Tal *task* consiste em um laço que é executado até que alguma mensagem seja lida do *buffer* de sistema. Quando isso ocorre, tal mensagem é enviada para cada um de seus vizinhos. O pseudocódigo 3.4.3 descreve-a.

```
1  task encaminhar_mensagem() {
2    var char mensagem[];
3    var int id;
4
5    do {
6      mensagem := ler_buffer();
7    } while (mensagem == null) {
8
9    forall id in channels[] {
10     send(id, mensagem);
11   }
12 }
13
14
```

Código 3.4.3: *Implementação da Task 3*

3.5 Formalização

O código 3.5.1 representa a implementação completa do ADDC em pseudocódigo. São utilizadas as notações de (TEL, 2000) para formalização de algoritmos distribuídos.

O primeiro par *begin-end* representa a implementação da ***task* 1**, a qual deve ser executada somente uma vez, e apenas por nodos iniciais. Um nodo inicial a executa com o objetivo de dar início à difusão, ou seja, é feito um *send* da mensagem inicial para cada um de seus vizinhos.

O segundo par *begin-end* representa a implementação da ***task* 2**, a qual deve ser executada por todos os nodos. Quando executada, tal *task* aguarda o recebimento de uma mensagem. Quando recebida, essa mensagem é processada, armazenada ou descartada, ou, ainda, escrita em um *buffer* (para posterior leitura pela *task* 3). Então, tal *task* termina sua execução.

O terceiro par *begin-end* representa a implementação da ***task* 3**, que deve ser executada por todos os nodos. Quando executada, essa *task* faz acesso de leitura ao *buffer* de sistema até que consiga ler uma mensagem (finalmente escrita pela *task* anterior). Por fim, tal mensagem é enviada a cada um dos vizinhos do nodo executor da *task*.

Conforme já explicitado, as *tasks* são **atômicas**, ou seja, quando uma delas inicia sua execução, em um nodo, é necessário que ela finalize para que outra *task* assuma o controle. Adicionalmente, o modelo *multitask* permite concorrência entre elas, gerando uma maior eficiência na execução do algoritmo em cada nodo. Por exemplo, enquanto um nodo inicial está iniciando uma difusão (*task* 1), ele pode, concorrentemente, estar esperando o recebimento de uma mensagem (início da *task* 2), e/ou enviando outra anteriormente recebida (*task* 3).

```
1  var system_buf          : buffer
2      channels[1..N]      : channel
3      mylabel             : character
4      m[1..N]             : vector
5      ciclos[1..M][1..N]  : character
6
7  begin if p is initiator then only once
8           m := adicionar_rotulo(mylabel, m);
9           forall q E Neigh_p do send <m> to q ;
10 end;
11
12 begin receive <m>;
13       m := adicionar_rotulo(mylabel, m);
14       if detectar_mensagem_ciclica(m) then
15          if m[1] = mylabel then
16             armazenar_ciclo(m);
17          else descartar_mensagem(m);
18       else escrever_buffer(m);
19 end;
20
21 begin
22   repeat
23          m := ler_buffer();
24   until m not null;
25   forall r E Neigh_p do send <m> to r ;
26 end
```

Código 3.5.1: *Implementação do ADDC*

Esquematicamente, ADDC possui a seguinte estrutura:

Entrada: rede, representada por um dígrafo sem pesos nos *links*, com os nodos iniciais marcados

Se nodo inicial, **Inicie uma Difusão**, uma única vez

concorrentemente **Receba e Processe Mensagem**

concorrentemente **Encaminhe Mensagem**

Saída: matriz de ciclos em cada nodo inicial

O grafo de *tasks* (ou **grafo de processos**), indicando a dependência entre elas, é mostrado na figura 3.5. Note-se que as três *tasks* são independentes entre si.

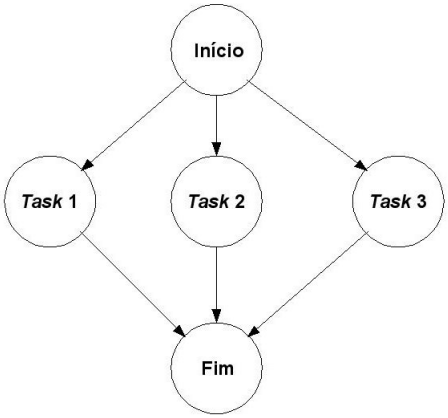

Figura 3.5: Grafo de *Tasks* do ADDC

3.6 Complexidade do ADDC

Nesta seção, são calculadas as complexidades em relação ao número de passos necessários à terminação do ADDC, e em relação ao número total de mensagens trocadas durante sua execução.

3.6.1 Número de Passos

O número de passos do ADDC é igual ao número de encadeamentos *send/receive* desde o início da difusão até sua interrupção, quando da detecção de uma mensagem cíclica. Logo, o número de passos está limitado ao comprimento da maior mensagem cíclica possível de ser formada. No pior caso, uma mensagem cíclica inclui todos os nodos da rede uma única vez, e retorna ao nodo inicial, caso em que se tem a formação de um ciclo. Portanto, tal ciclo possui todos os N nodos da rede e N arcos (*links*), os quais geram N encadeamentos *send/receive*. Tem-se, então, que o número de passos do ADDC, supondo-se que todos os nodos iniciais iniciem suas difusões ao mesmo tempo, é limitado a N, tal que N é o número de nodos da rede. A complexidade é, pois, igual a $O(N)$.

Na prática, em uma rede relativamente "bem conectada" (densa), o número de passos é, de fato, igual, ou um pouco menor, ao número de nodos da rede. Tal fato pode ser constatado nos testes que fazem uso de difusões simultâneas, disponíveis na seção 3.8.

3.6.2 Número de Mensagens Totais

O número de mensagens totais enviadas durante a difusão de um nodo inicial pode ser obtida multiplicando-se a quantidade de passos pela quantidade de mensagens enviadas em cada passo. O número de passos, conforme já calculado, é, no máximo, igual ao número N de nodos da rede. Já o número de mensagens enviadas em cada passo é, no máximo, igual ao número L de *links* da rede. Portanto, o número de mensagens trocadas, considerando-se um único nodo inicial, é, no pior caso, igual a NxL.

Adicionalmente, para se obter o número total de mensagens, a equação anterior precisa ser multiplicada pelo número de nodos iniciais da rede, já que cada um deles é responsável por uma difusão independente. No máximo, o número de nodos inciais é igual ao número N de nodos da rede.

Portanto, tem-se:

- número total de mensagens = $O(N(NxL))$

Entretanto, em geral, nem todos os nodos da rede são iniciais. De fato, apenas alguns poucos nodos são, previamente, marcados como iniciais. Tais nodos podem refletir a vontade, de um humano, em detectar todos os ciclos da rede. Neste caso, basta que se escolha nodos iniciais que dêem conta de alcançar, por suas difusões, todos os nodos da rede. O cálculo deste número é complicado de ser obtido, e varia com os nodos iniciais escolhidos e com a topologia da rede. Entretanto pode-se se estimar que, em média, ele é bem menor que o número de nodos da rede, e, portanto, pode ser definido por uma constante I.

Dessa forma, reduz-se a complexidade no pior caso à $I(NxL)$. Como I é uma constante, a ordem da complexidade se transforma em:

$$O(NxL)$$

Adicionalmente, sobretudo em redes com pouca conectividade (esparsas), o número de mensagens em trânsito, em cada passo, é bem menor que L, já que, em um passo, apenas alguns nodos estão enviando mensagens. Logo, as mensagens enviadas em um passo ficarão limitadas ao número de vizinhos de tais nodos. Nesses casos, portanto, o ADDC se torna, na prática, mais eficiente do que indica a complexidade no pior caso obtida.

3.6.3 Comparações

Em ambiente seqüencial, a detecção de ciclos possui complexidade linear ao número de arcos se implementada por DFS. Entretanto, tal método não é capaz de listar todos os ciclos de um dígrafo. Apesar de ADDC não garantir a detectação de todos os ciclos de um dígrafo, pode-se ajustar adequadamente quais nodos iniciais são escolhidos, de forma a cobrir todos os ciclos da rede.

A detecção seqüencial de ciclos em dígrafos por "força-bruta" (esgotamento de caminhos) possui, segundo FEOFILOFF, complexidade igual a $O(L(N+L))$, a qual, apesar de ser inferior à complexidade do ADDC no pior caso, pode comportar-se de forma mais ineficiente em uma rede bastante densa em que o número L de *links* é superior ao número N de nodos. Isso ocorre porque tal algoritmo de ambiente seqüencial possui, em sua complexidade, um fator L^2, enquanto que o ADDC possui um fator linear de L. Apesar disso, em vez de comparar ao número de mensagens, o mais correto seria comparar a complexidade de um algoritmo seqüencial à complexidade do número de passos do ADDC, já que o número de passos indica "fatias de tempo" encadeadas (seqüenciais) em um algoritmo distribuído. Neste caso, o ADDC possui complexidade inferior, pois é linear em relação ao número de nodos da rede.

3.7 Corretude e Terminação

A prova da corretude do ADDC é, primeiramente, feita apenas para um nodo inicial, da seguinte forma: primeiro prova-se que um nodo inicial armazena todos os seus ciclos simples; então prova-se que somente ciclos são armazenados pelo nodo inicial; após, por contradição, é provado que todos os ciclos armazenados pelo nodo são, de fato, simples. Finalmente, é provada a corretude na presença de um número qualquer de nodos iniciais simultâneos. Na seção seguinte, é provado que o algoritmo termina, ou seja, que as mensagens geradas pelos nodos iniciais não ficam, eternamente, circulando na rede.

3.7.1 Corretude

Proposição 3.7.1 *Em qualquer rede, uma difusão gerada por um nodo inicial X causa o armazenamento de todos os ciclos simples que incluem o nodo X.*

Para provar a proposição 3.7.1, basta constatar que todo ciclo que inclui X possui um *link* chegando em X e outro saindo de X. Ao iniciar uma difusão, um nodo X envia uma mensagem m a todos os *links* que saem dele. Portanto, cada uma dessas mensagens iniciadas em X percorre um ciclo da seguinte forma: cada um dos nodos participantes do ciclo, em cada passo, recebe a mensagem, e, caso tal nodo não se repita no ciclo, encaminha-a ao próximo nodo vizinho. Como o ciclo é assumido como simples, o único nodo que se repete no ciclo é o inicial, momento em que a mensagem retorna a ele. Logo, todos os nodos do ciclo encaminham a mensagem, exceto o inicial, que a detectará. Como a mensagem m foi iniciada por X, ela é, por ele, armazenada (conforme definição 3.2.1).

Proposição 3.7.2 *Um nodo inicial X possui armazenado apenas ciclos.*

A prova da proposição 3.7.2, de que somente ciclos são armazenados, é obtida ao constatar-se que uma mensagem iniciada em X só é armazenada por X se ela for detectada por este nodo (definição 3.2.1). E uma mensagem é detectada por um nodo X, se, e somente se, X se repete na mensagem. Logo, a mensagem é um ciclo. Note-se que isso não prova que tal ciclo é simples.

Proposição 3.7.3 *Todos os ciclos armazenados por um nodo inicial X são, de fato, simples.*

A prova da proposição 3.7.3 é por contradição: suponha que tal ciclo não seja simples. Então, existe um nodo y ($y \neq X$) que se repete no ciclo. Se y se repete no ciclo, tal mensagem deveria ter sido detectada e descartada por ele (definição 3.2.1), interrompendo seu encaminhamento. Logo, tal mensagem não teria chegado em X (a detecção e o armazenamento não teria ocorrido em X), o que é uma contradição com a proposição 3.7.2.

Proposição 3.7.4 *Nodos iniciais podem difundir simultaneamente de forma independente.*

Para verificar a proposição 3.7.4, basta provar que um nodo inicial não armazena ciclos de outro nodo inicial. Isso, de fato, ocorre, pois o armazenamento de uma mensagem é feito com a verificação do nodo que iniciou esta mensagem (definição 3.2.1), e não com a simples verificação de que um nodo é inicial. Dessa forma, um nodo inicial cumpre, para outros nodos iniciais, o papel de "não-inicial" , não armazenando, portanto, mensagens que não iniciam nele.

3.7.2 Terminação

Teorema 3.7.5 *O ADDC sempre termina.*

Para se ter a certeza de que, em algum momento, o ADDC termina, basta provar que uma mensagem m qualquer é, em algum momento, interrompida por um nodo x_n, formando a mensagem $m = x_1, x_2, .., x_n$.

Sabe-se, ao contrário, que uma mensagem m não é interrompida (é encaminhada adiante) por x_n se, e somente se, ambas as condições abaixo forem verdadeiras:

(c1) x_n não se repete m ($\forall i \in [1; n-1] \mid x_i \neq x_n$); e

(c2) x_n possui ao menos um vizinho; (se não possuir vizinho, não há para onde encaminhar m.)

A prova do teorema 3.7.5 é por contradição: suponha que uma mensagem m não seja interrompida em x_n (é encaminhada por ele). Portanto as condições (c1) e (c2) devem ser verdadeiras.

Sabe-se, por outro lado, que uma mensagem m qualquer só pode assumir uma das duas hipóteses seguintes:

(h1) x_n se repete em m ($\exists i \in [1; n-1] \mid x_i = x_n$); ou

(h2) x_n não se repete em m.

Se (h1) é verdadeira, então (c1) é falsa (pois uma é o inverso da outra), o que é uma contradição (já que (c1) deveria ser verdadeira).

Se (h2) é verdadeira, x_n assume, em algum momento, uma das duas hipóteses abaixo (supondo-se um número finito de nodos na rede):

- x_n não terá mais para onde encaminhar a mensagem m, por **não ter vizinhos**, logo (c2) é falsa, o que é uma **contradição**, já que **(c2)** deveria ser verdadeira; ou

- x_n (se transformando em x_{n-1}) encaminhará a mensagem para um vizinho x_n, que interromperá a mensagem, pois qualquer vizinho x_n de x_{n-1} é, necessariamente, um **nodo que se repete** na mensagem m, logo (c1) é falsa, o que é uma **contradição**, já que **(c1)** deveria ser verdadeira.

3.8 Testes ADDC

Nesta seção, serão mostrados exemplos de execuções do ADDC sobre redes variadas. Cada exemplo tem o intuito de destacar um possível comportamento do ADDC em determinada situação. Dessa forma, pretende-se reforçar o seu funcionamento.

Serão utilizados, para melhor entendimento, a idéia de "passos" (1^o, 2^o, 3^o, etc), que correspondem a uma fatia de tempo qualquer. Nos exemplos, supõe-se que, em cada nodo, as tarefas são executadas, sincronamente, da seguinte forma: as *tasks* 1 e 2, nessa ordem (**iniciar difusão** e **receber mensagem e processá-la**), são executadas em um passo. Igualmente, as *tasks* 3 e 2, nessa ordem (**encaminhar mensagem** e **recebê-la e processá-la**) podem ser executadas também em um passo. No exemplo 3.8.5, será exemplificado um caso em que atrasos diferenciados serão considerados.

Adicionalmente, os elementos do vetor mensagem m, a ser difundido, serão, por economia de espaço, representados separados por vírgula. Assim, um vetor $m[A,B,C]$ será representado, simplesmente, por "A,B,C".

Abaixo, constam as conveções (estilo dos nodos) utilizadas nas figuras para representar a execução de alguns procedimentos das *tasks* em cada passo:

- **Difusão Iniciada (*task* 1)** → nodo com rótulo sublinhado e seta pontilhada.

- **Ciclo Detectado (Armazenado)**[2] **(*task* 2)** → nodo com rótulo em negrito (e ciclo informado na figura).

[2]Reforçando: uma mensagem cíclica quando detectada e armazenada é chamada de **ciclo**, logo pode-se dizer que um **ciclo foi detectado**.

- **Mensagem Descartada**[3] **(*task* 2)** → nodo preenchido.
- **Mensagem Encaminhada (*task* 3)** → seta pontilhada.

3.8.1 Descarte de Mensagens Cíclicas

O exemplo 3.8.1, além de mostrar a detecção de ciclos obtidos por A (único nodo inicial), deixa claro o descarte de mensagens por outros nodos.

Exemplo 3.8.1 (ADDC ao descartar mensagem cíclica iniciada por A) *Na figura 3.6, estão representados a rede inicial e a execução do ADDC, formada, neste caso, por quatro passos.*

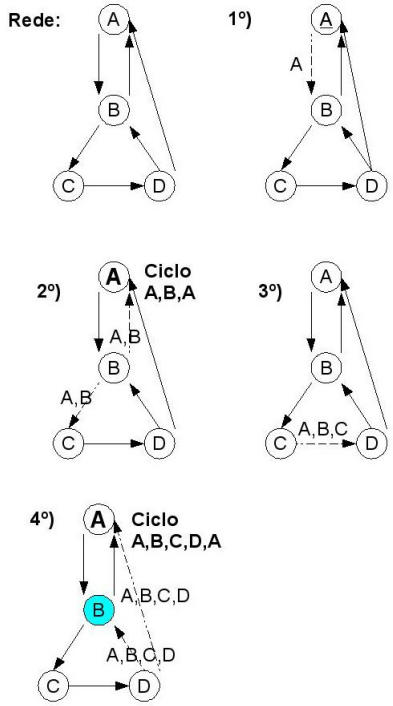

Figura 3.6: Descarte pelo nodo B da mensagem cíclica iniciada por A (“A,B,C,D,B”), no 4º passo.

Note-se que o único nodo marcado como inicial é o nodo A, que inicia difusão, no 1º passo. Logo, apenas ele está interessado na detecção de ciclos simples (que lhe incluam). O nodo A detecta ciclos no 2º e 4º passo. No 4º passo, o nodo B interrompe o encaminhamento da mensagem cíclica “A,B,C,D,B”, descartando-a, já que tal não foi por ele iniciada.

O exemplo 3.8.2 seguinte tem o objetivo de mostrar o descarte de mensagens cíclicas em um contexto um pouco diferente.

Exemplo 3.8.2 (ADDC ao descartar mensagem cíclica nos nodos B e C) *Na figura 3.7, constam a rede inicial juntamente da execução do ADDC, formada, neste caso, por quatro passos.*

[3]Reforçando: mensagem descartada é uma mensagem cíclica detectada e descartada (pois não formou um **ciclo**, mas, sim, apenas uma mensagem cíclica).

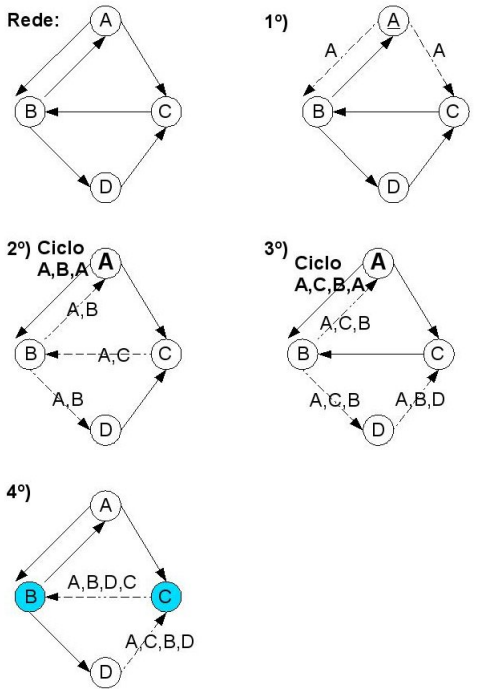

Figura 3.7: Descarte de mensagem por dois nodos da rede

Agora, com a modificação da rede, houve descarte de mensagens cíclicas por dois nodos diferentes, no 4º passo. O nodo B descartou a mensagem “A,B,D,C,B”, e o nodo C descartou a mensagem “A,C,B,D,C”.

3.8.2 Multidifusão

O exemplo 3.8.3 mostra o comportamento do ADDC na presença de multidifusões. Nele, o nodo B, também marcado como inicial, dá início à sua difusão no mesmo instante que o nodo A.

Exemplo 3.8.3 (ADDC iniciado em dois pontos da rede) *Na figura 3.8, têm-se a rede inicial juntamente da execução do ADDC, formada, neste caso, por quatro passos.*

Note-se, na figura 3.8, que as diferentes difusões não interferem uma na outra. A execução é igual a da figura 3.6, com a adição da difusão gerada por B. Tal fato fez coincidir, no 4º passo, o descarte, pelo nodo B, da mensagem “A,B,C,D,B”, iniciada por A, juntamente da detecção, por B, do ciclo B,C,D,A,B. Tal concomitância de fatos é representada pela sobreposição de “preenchido com negrito” em B. Adicionalmente, no 2º e 3º passos, o nodo B detectou, respectivamente, os ciclos B,A,B e B,C,D,B.

O disparo das difusões, de cada nodo inicial, não precisa ocorrer ao mesmo tempo. No exemplo 3.8.4, tem-se a ilustração do comportamento obtido quando o nodo B inicia sua difusão no 2º passo (ou seja, um passo após o nodo A ter iniciado a sua).

Exemplo 3.8.4 (ADDC iniciado em dois pontos da rede, em momentos diferentes) *Na figura 3.9, estão ilustradas a rede inicial juntamente da execução do ADDC, formada, neste caso, por cinco passos.*

Obteve-se, neste teste, um resultado bem semelhante ao do exemplo 3.8, em que os fluxos foram disparados ao mesmo tempo, no 1º passo. A diferença, agora, é que um passo a mais foi requerido para que a difusão de B chegasse ao fim.

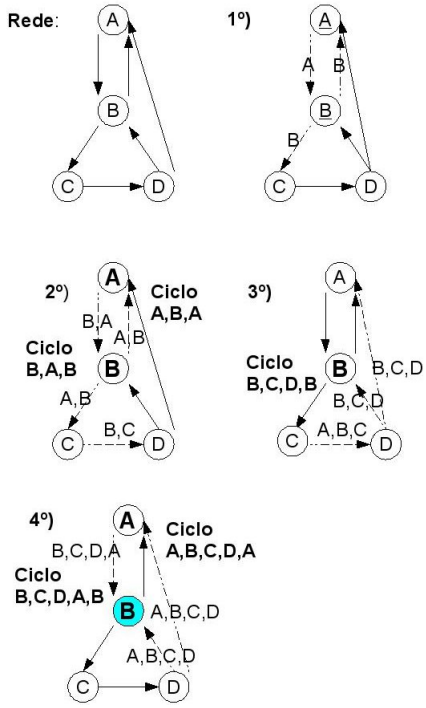

Figura 3.8: Execução com multidifusões, geradas por A e B

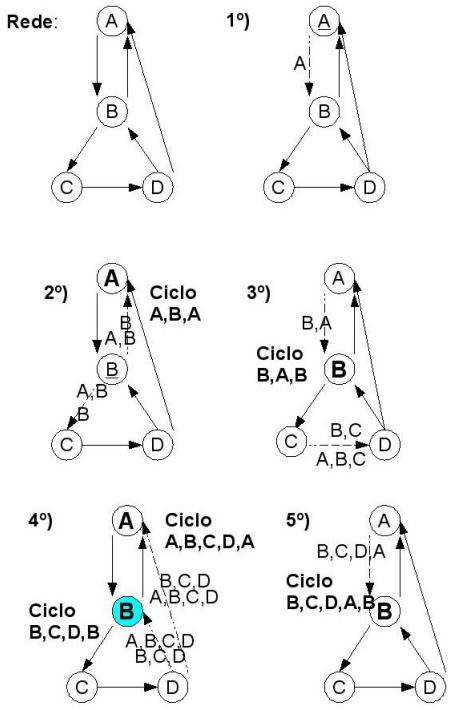

Figura 3.9: Execução com multidifusões, geradas, em momentos diferentes, por A e B

Note-se, outrossim, que, no 2º passo, está representado o envio de duas mensagens de B para A e também de B para C. Em ambos os *links*, no mesmo passo, trafegaram uma mensagem com informação “A,B” e outra com informação “B”. Algo semelhante ocorreu no 3º passo (mensagens “A,B,C” e “B,C”, de C para D) e também no 4º passo em mensagens enviadas de D para A e de D para B.

Perceba-se, ainda que, no 4º passo, houve, por B, tanto a detecção do ciclo B,C,D,B quanto o descarte da mensagem "A,B,C,D,B".

3.8.3 Atrasos Diferenciados

No exemplo 3.8.5 seguinte, será suposto que o nodo B demora dois passos, ao invés de um, para executar a *task* 3, ou seja, possui um atraso que é igual ao dobro do tempo levado pelos outros nodos para realizar a tarefa de encaminhamento de mensagens. As outras *tasks* continuam levando o mesmo número de passos anteriormente estabelecido para serem completadas. Adicionalmente, este exemplo considera a existência de dois nodos inciais, A e B, com difusões concomitantes.

Exemplo 3.8.5 (ADDC sobre uma rede em que nodo B completa a *task* 3 em dois passos) *Na figura 3.10, estão representadas a rede inicial juntamente da execução do ADDC, formada, neste caso, por cinco passos.*

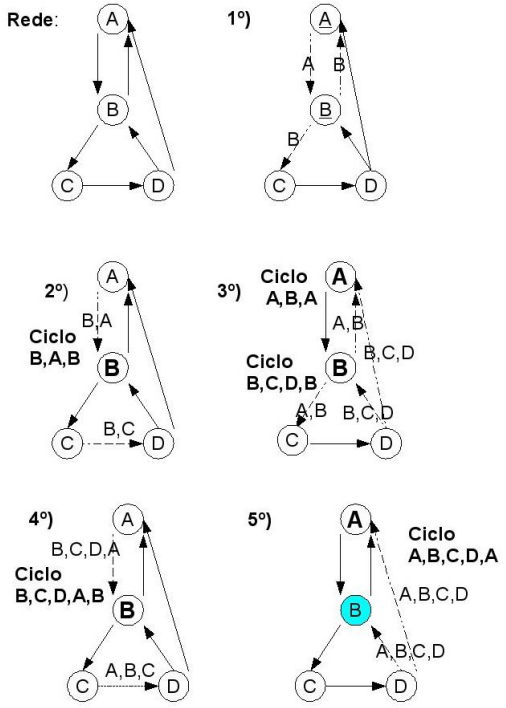

Figura 3.10: Execução com atraso maior na *task* 3 executada por B

Note-se que em relação ao exemplo 3.8.3, sem atrasos diferenciados, houve a necessidade de um passo extra para a terminação do algoritmo. As mensagens cíclicas que necessitaram da execução da *task* 3 pelo nodo B demoraram um passo a mais para serem detectadas. Tais mensagens são aquelas encaminhadas em algum momento por B, mas iniciadas por qualquer nodo diferente de B: "A,B,A" e "A,B,C,D,A" (detectadas como ciclos por A no 3º e 5º passo) e "A,B,C,D,B" (descartada por B no 5º passo).

Por sua vez, as mensagens cíclicas iniciadas em B são, justamente, as que não precisam da execução da *task* 3 (encaminhamento) em B. Tais ciclos foram detectados, por B, no mesmo número de passos do que no exemplo 3.8.3, o qual não possuía atrasos diferenciados. São eles: B,A,B e B,C,D,B e B,C,D,A,B (detectados, respectivamente, no 2º, 3º e 4º passos).

4 CONCLUSÃO

Um grafo pode ser totalmente estruturado independente de seus nodos. Assim sendo, os rótulos são apenas informados na entrada e saída de dados do programa, possibilitando uma economia de memória com o armazenamento de estruturas independentes de rótulos.

Adicionalmente, a forma de estruturar e representar tais grafos tem relevância no desempenho e no entendimento de algoritmos executados sobre tais estruturas de dados. Em geral, um grafo é representado através da relação de adjacência entre seus nodos.

"Problemas difíceis que envolvem processamento de grafos podem ter soluções simples, além de as abstrações que estávamos usando (DFS e listas de adjacência) serem mais poderosas do que se pode perceber." (SEDGEWICK, 2002, p. 198 tradução nossa)

Métodos de busca em grafos são modelos de algoritmos muito utilizados para resolução de diversos problemas em computação. Destacam-se modelos de buscas em profundidade (DFS) e em largura (BFS). DFS, por exemplo, é útil em muitas aplicações, entre elas a detecção de ciclos. Mas não serve, por exemplo, para a enumeração, explícita, dos ciclos de um dígrafo. Durante uma busca, os arcos do dígrafo sofrem uma classificação de acordo com o papel que cumprem na arborescência em formação.

Nesse sentido, um algoritmo distribuído de difusão de mensagens, chamado ADDC, foi modelado. Seu objetivo principal é a detecção de ciclos através do armazenamento de ciclos simples em nodos previamente marcados como iniciais. Tais nodos são aqueles interessados na descoberta de ciclos que lhe incluam. Portanto, o problema da conectividade da rede foi resolvida através da limitação de que o ADDC apenas detecte ciclos dos quais os nodos iniciais façam parte. Cada nodo inicial r é responsável por uma execução independente de difusão. Dessa forma, garante-se que cada execução consegue explorar todos os nodos alcançáveis a partir do nodo inicial, dividindo a rede em componentes r-alcançáveis.

Foi possível um bom detalhamento, em nível de pseucódigo, do ADDC. Logo, uma possível implementação de um protótipo não deve ser algo muito distante do que foi detalhado neste trabalho. Entretanto, em uma implementação real, será possível constatar aspectos de desempenho e de arquitetura impossíveis de serem verificados só no papel. Em (GARG, 2004), um *framework* para troca de mensagens entre processos distribuídos, implementado em Java, está disponível. Tal ambiente, juntamente com a utilização de *Java Threads*, seria a primeira opção de escolha para a implementação do modelo *multitasking* sobre o qual ADDC foi modelado. Com isso, a complexidade do ADDC, apesar de já ter sido calculada, poderá ser testada na prática, pois freqüentemente ocorre de a complexidade no caso médio ser bem mais aceitável do que o limite superior admitido no cálculo da complexidade no pior caso. Pelo contrário, uma complexidade teoricamente pequena pode se tornar inviável na prática.

4.1 Trabalhos Futuros

Abaixo, tem-se algumas idéias para trabalhos futuros:

- Propor a otimização do ADDC em ambientes específicos, tais como quando um tipo de topologia de rede é previamente conhecida, ou se, por exemplo, todos as comunicações da rede são bidirecionais. Neste caso, a rede pode ser modelada como um dígrafo simétrico. Tais otimizações devem visar sobretudo à diminuição do número de mensagens necessárias à detecção de ciclos.

- Implementar um protótipo do ADDC, utilizando *Java Threads* e o framework proposto por GARG e disponível, com funções adicionais, em (SKLAR, 2007).

- Verificar de que forma implementar a escolha automática de um conjunto de nodos iniciais capazes de cobrir todos os ciclos da rede.

- Formular uma detecção de término para o ADDC, de forma que um dado nodo inicial saiba o momento em que a matriz *ciclos* esteja completamente populada com todos os ciclos que lhe incluem.

5 ANEXO

Neste anexo, encontram-se partes de códigos fontes referenciadas durante o trabalho. Para ter acesso ao código completo e comentado das implementações, acesse (SKLAR, 2008).

5.1 Estrutura

```
1  typedef struct {
2    int edges[MAXV+1][MAXDEGREE]; /* adjacency info */
3    int degree[MAXV+1]; /* outdegree of each vertex */
4    int nvertices; /* number of vertices in the graph */
5    int lvertices[MAXV]; /* label of vertices */
6    int nedges; /* number of edges in the graph */
7  } graph;
8
9  initialize_graph(graph *g) {
10   int i;  /* counter */
11   g -> nvertices = 0;
12   g -> nedges = 0;
13   for (i=0; i<MAXV; i++) {
14     g->degree[i] = 0;
15     g->lvertices[i] = -1;
16   }
17 }
18 /*  função para obter, a partir do seu rótulo, a posição
19 *   em que o nodo está armazenado/será armazenado) */
20 int transf_label_to_struct(graph *g, int lv) {
21   int l;
22   int position = -1;
23   for (l=0; l<MAXV; l++) {
24    if (g->lvertices[l] == lv) {
25       position = l;
26     }
27   }
28   if (position == -1) {
29     g->lvertices[g->nvertices] = lv;
30     position = g->nvertices++;
31   }
32   return position;
33 }
34 /* função para se obter o rótulo de um nodo
35 *  (utilizada apenas na saída de dados na tela) */
36 int get_label(graph *g, int v) {
37   if (v == -1) {
38     return -1;
39   }
40   return g->lvertices[v];
41 }
```

Código 5.1.1: *Declaração e manipulação dos rótulos do dígrafo, com uso do vetor* ***lvertices[]****, que armazena, separadamente da estrutura do dígrafo, os rótulos dos nodos.*

5.2 BFS

```
bfs_process_edge(graph *g, int x, int y) {
  if (x == y) {
    // laço processado
    printf("laço (%d,%d)\n",get_label(g,x),get_label(g,y));
  }
  else if (is_descendant(y,x,parent)) {
    if (parent[y] == x)
      // arco de arborescência processado
      printf("arc.arb. (%d,%d)\n",get_label(g,x),get_label(g,y));
    else // arco forward inexiste em BFS
  }
  else if (is_descendant(x,y,parent)) {
    if (parent[x] == y)
      // arco-pai processado
      printf("arco-pai (%d,%d)\n",get_label(g,x),get_label(g,y));
    else
      // arco de retorno processado
      printf("arc.ret.(%d,%d)\n",get_label(g,x),get_label(g,y));
  } // nao é nem descendente nem ascendente (é cruzado)
  else if (distance[x] == distance[parent[y]])
    // arco cruzado mínimo processado
    printf("a.cruz.mín.(%d,%d)\n",get_label(g,x),get_label(g,y));
  else
    // arco cruzado processado
    printf("arc.cruz.(%d,%d)\n",get_label(g,x),get_label(g,y));
}
```

Código 5.2.1: *Função de classificação de arcos pertencente ao código BFS para classificação completa de arcos. Tal classificação contempla o conceito de* ***arco cruzado mínimo****, além de estar* ***dualmente estruturada****.*

```
const int graph_type = SIMETRIC;

bfs(graph *g, int root)
{
  queue q;      /* queue of vertices to visit */
  int v;        /* current vertex */
  int i;        /* counter */
  init_queue(&q);
  enqueue(&q,root);
  parent[root] = root;
  distance[root] = 0;

  while (empty(&q) == FALSE) {
    v = dequeue(&q);
    for (i=0; i<g->degree[v]; i++) {
      if (parent[g->edges[v][i]] == -1) {
        enqueue(&q,g->edges[v][i]);
        parent[g->edges[v][i]] = v;
        distance[g->edges[v][i]] = distance[v]+1;
      }
      bfs_process_edge(g,v,g->edges[v][i]);
    }
    process_vertex(g,v);
  }
}
```

Código 5.2.2: *Função principal do código BFS*

```
1  enum states {undiscovered, discovered, processed} state[MAXV];
2  int entry_time[MAXV];
3  int exit_time[MAXV];
4  int time = 0;
5  int parent[MAXV]; /* discovery relation */
6  /* distância mínima da raiz até os outros vértices */
7  int distance[MAXV];
8
9  /* if true, cut off search immediately */
10 bool finished = FALSE;
11
12 initialize_search(graph *g) {
13   int i;  /* counter */
14
15   for (i=0; i<MAXV; i++) {
16     state[i] = undiscovered;
17     parent[i] = -1;
18     distance[i] = -1;
19   }
20 }
21
22 print_path(graph *g, int start, int end, int parents[]) {
23   if ((start == end) || (end == -1))
24     printf("%d",get_label(g,start));
25   else {
26     print_path(g,start,parents[end],parents);
27     printf(" %d",get_label(g,end));
28   }
29 }
30
31 bool is_descendant(int descendant,int ascendant,int parents[])
32 {
33   int v = descendant;
34   do {
35     v = parents[v];
36     if (v == ascendant) {
37       return TRUE;
38     }
39   } while (parents[v] != v);
40   return FALSE;
41 }
```

Código 5.2.3: *Funções e estruturas auxiliares criadas para a implementação do BFS*

REFERÊNCIAS

AUGUSTON, M.; HON, M. H. Assertions for Dynamic Shape Analysis of List Data Structures. In: INTERNATIONAL WORKSHOP ON AUTOMATED AND ALGORITHMIC DEBUGGING, AADEBUG'97, LINKOPING, 3., 1997. **Proceedings...** [S.l.: s.n.], 1997. p.37–42.

BROIDO, A.; CLAFFY k. **Internet topology**: connectivity of ip graphs. 2001.

CHANDY, K. M.; MISRA, J. **On Diffusing Computations**. Austin, TX, USA: [s.n.], 1980.

CORMEN, T. H.; STEIN, C.; RIVEST, R. L.; LEISERSON, C. E. **Introduction to Algorithms**. [S.l.]: McGraw-Hill Higher Education, 2001.

DIJKSTRA, E. W.; SCHOLTEN, C. S. Termination detection for diffusing computations. **Information Processing Letters**, [S.l.], v.11, n.1, p.1–4, August 1980.

FEOFILOFF, P. **Ciclos em grafos e digrafos — Algoritmos para Grafos em C via Sedgewick**. Disponível em <http://www.ime.usp.br/∼pf/algoritmos_para_grafos/aulas/cycles.html>. Acessado em outubro de 2008.

FEOFILOFF, P. **Grafos - Passeios e Caminhos**. Disponível em <http://www.ime.usp.br/∼pf/analise_de_algoritmos/aulas/grafos.html#caminho>. Acessado em setembro de 2008.

FEOFILOFF, P. **Ciclos e grafos acíclicos - Ciclos**. Disponível em <http://www.ime.usp.br/∼pf/analise_de_algoritmos/aulas/cycles-and-dags.html#cycle>. Acessado em setembro de 2008.

FEOFILOFF, P. **Arborescências**. <http://www.ime.usp.br/∼pf/algoritmos_para_grafos/aulas/arborescences.html>. Acessado em setembro de 2008.

FEOFILOFF, P. **Digrafos — Algoritmos para Grafos em C via Sedgewick**. Disponível em <http://www.ime.usp.br/∼pf/algoritmos_para_grafos/aulas/digraphs.html>. Acessado em outubro de 2008.

FEOFILOFF, P. **Busca em Largura — Algoritmos para Grafos em C via Sedgewick**. Disponível em <http://www.ime.usp.br/∼pf/algoritmos_para_grafos/aulas/bfs.html>. Acessado em outubro de 2008.

FLOYD, R. W. Nondeterministic Algorithms. **J. ACM**, New York, NY, USA, v.14, n.4, p.636–644, 1967.

GARG, V. K. **Concurrent and Distributed Computing in Java**. [S.l.]: Wiley-Interscience, 2004. 309p.

GELDER, A. V. Efficient loop detection in Prolog using the tortoise-and-hare technique. **J. Log. Program.**, New York, NY, USA, v.4, n.1, p.23–31, 1987.

KNUTH, D. E. **The Art of Computer Programming.** [S.l.]: Addison-Wesley Publishing Co., Menlo Park, 1969. v.Vol. II: Seminumerical algorithms.

LEONEL, N. A. **Teoria dos Grafos e Análise Combinatória**. Instituto de Informática, UFRGS, Porto Alegre: Editora Sagra Luzzatto, 2001. 200p.

LYNCH, N. A. **Distributed Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

NIVASCH, G. Cycle detection using a stack. **Inf. Process. Lett.**, Amsterdam, The Netherlands, The Netherlands, v.90, n.3, p.135–140, 2004.

POLLARD, J. M. A Monte Carlo method for Factorization. **BIT**, [S.l.], v.15, n.3, p.331–334, 1975.

SEDGEWICK, R. **Algorithms in C**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. v.Part 5: Graph Algorithms.

SKIENA, S. **Graph Data Structures — Skiena's Algorithms Lectures (Lecture 10)**. Disponível em <http://www.cs.sunysb.edu/∼algorith/video-lectures/2007/lecture10.pdf>. Acessado em outubro de 2008.

SKIENA, S. **Breadth-First Search — Skiena's Algorithms Lectures (Lecture 11)**. Disponível em <http://www.cs.sunysb.edu/∼algorith/video-lectures/2007/lecture11.pdf>. Acessado em outubro de 2008.

SKIENA, S. S.; REVILLA, M. **Programs from Programming Challenges — Programming Challenges**: the programming contest training manual. Disponível em <http://www.cs.sunysb.edu/∼skiena/392/programs/>. Acessado em setembro de 2008.

SKIENA, S. S.; REVILLA, M. **Programming Challenges**: the programming contest training manual. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

SKLAR, M. M. **Terminação em Grafo - Framework para Ambiente Distribuído**. Disponível em <http://www.inf.ufrgs.br/∼marcios/TerminacaoEmGrafo>. Acessado em outubro de 2008.

SKLAR, M. M. **Implementações do Trabalho de Graduação**. Disponível em <http://www.inf.ufrgs.br/∼marcios/bfs_dfs>. Acessado em novembro de 2008.

TEL, G. **Introduction to Distributed Algorithms**. 2th.ed. Cambridge, USA: CAMBRIDGE, 2000.

WIKIPEDIA. **Cycle detection — Wikipédia, a enciclopédia livre**. Disponível em <http://en.wikipedia.org/wiki/Cycle_detection>. Acessado em setembro de 2008.

WIKIPEDIA. **Glossary of graph theory (Walks) — Wikipédia, a enciclopédia livre**. Disponível em <http://en.wikipedia.org/wiki/Glossary_of_graph_theory#Walks>. Acessado em setembro de 2008.

WIKIPEDIA. **Path (graph teory) — Wikipédia, a enciclopédia livre**. Disponível em <http://en.wikipedia.org/wiki/Path_(graph_theory)>. Acessado em setembro de 2008.

WIKIPEDIA. **Subgraphs (Subgraph) — Wikipédia, a enciclopédia livre**. Disponível em <http://en.wikipedia.org/wiki/Subgraph#Subgraphs>. Acessado em novembro de 2008.

WIKIPEDIA. **Loopback — Wikipédia, a enciclopédia livre**. Disponível em <http://pt.wikipedia.org/wiki/Loopback>. Acessado em outubro de 2008.