JOSÉ MARCELO ALMEIDA PRADO CESTARI

Uma Abordagem Distribuída Baseada no Algoritmo do Carteiro Chinês para Diagnóstico de Redes de Topologia Arbitrária

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Mestrado em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Elias Procópio Duarte Jr.

CURITIBA

Março de 2001

UFPR
Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática do aluno ***José Marcelo Almeida Prado Cestari***, avaliamos o trabalho intitulado **"Uma Estratégia Baseada no Algoritmo do Carteiro Chinês para Diagnóstico Distribuído de Redes de Topologia Arbitrária"**, cuja defesa foi realizada no dia 23 de março de 2001. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 23 de março de 2001.

Prof. Dr. Elias Procópio Duarte Júnior
Presidente - Orientador

Prof. Dr. Antônio Alfredo Loureiro
Membro Externo - DCC/UFMG

Profª. Dra. Cristina Duarte Murta
DINF/UFPR

Agradecimentos

Primeiro de tudo gostaria de agradecer a Deus pela força, apoio e oportunidade de poder estudar em uma instituição como a Universidade Federal do Paraná desde 1994, cuja fama é justificada pela qualidade do ensino.

Agradeço também ao meu orientador, professor Elias Procópio Duarte Jr, o melhor orientador que um aluno poderia ter, e sem o qual esta dissertação não seria a mesma.

Agradecimentos especiais aos meus pais Isidoro José Cestari e Maria Aparecida A. P. Cestari e a todas as seguintes pessoas que de uma maneira ou de outra me apoiaram nesta caminhada: os professores do departamento de informática da UFPR pelas dicas e conselhos, Luiz Adriano A. P. Cestari e Laura Fabiana A. P. Cestari pela existência, a Renata B. Bley pela paciência, amor e companhia, a todo pessoal da C.M.F.A (Daniel, Junior, Maurício, Oliver, Renato e Rodrigo), ao CITS (Centro Internacional de Tecnologia de Software) e especialmente ao pessoal do CNTS, a todos da Voodoo Blues (Daniel, Guilherme e Adriano), aos meus parentes, ao pessoal da Scudengal pelos muitos momentos de diversão, aos irmãos da A.·.R.·.L.·.S.·.Luz da Verdade 94 e a todas as outras pessoas que fazem parte da minha vida.

Obrigado
José Marcelo A. P. Cestari

Índice

Resumo 4

Abstract 5

1 Introdução 6

1.1 Diagnóstico em Nível de Sistema 7

1.2 O Agente Chinês 9

2 Algoritmos de *Diagnóstico em* Redes de Topologia Arbitrária 11

2.1 Modelo PMC 12

2.2 O Algoritmo de Bagchi & Hakimi 13

2.3 O Algoritmo Adapt 14

2.4 O Algoritmo RDZ 17

2.5 O Algoritmo NBND 24

2.6 Um Algoritmo Baseado em Inundação de Mensagens 29

3 O Agente Chinês 32

3.1 Caminhamento em Grafos 32

3.2 O Algoritmo do Carteiro Chinês 38

3.3 O Agente Chinês 44

3.4 Latência do Algoritmo 48

4 Resultados de Simulação 52

4.1 Um Grafo Exemplo 53

4.2 Grafo $D_{1,2}$ 56

4.3 Hipercubo (16 nodos) 59

4.4 Um grafo Randômico 62

4.5 Rede Nacional de Pesquisa (RNP) 64

4.6 Comparações 66

4.7 Conclusão 73

5 Conclusão 75

6 Referências 77

Apêndice A 81

Resumo

Neste trabalho é apresentado um novo algoritmo para o diagnóstico distribuído de redes de topologia arbitrária baseado no algoritmo do Carteiro Chinês. Um agente móvel, isto é, um processo que é executado e transmitido entre os nodos da rede, percorre seqüencialmente todos os enlaces, de acordo com o caminho determinado pelo algoritmo do Carteiro Chinês. O agente, chamado Agente Chinês, vai testando os enlaces detectando novos eventos e disseminando as informações obtidas para os demais nodos da rede. Quando todos os nodos do sistema recebem a informação sobre um evento, o diagnóstico se completa. Neste trabalho assume-se que falhas não particionam a rede, e que um novo evento só ocorre após o diagnóstico do evento anterior. São apresentadas provas rigorosas do melhor e pior caso da latência do algoritmo, isto é, o tempo necessário para completar um diagnóstico. São apresentados também resultados experimentais obtidos através da simulação do algoritmo em vários tipos diferentes de topologia, dentre eles, hipercubos de 16, 64 e 128 vértices, grafo $D_{1,2}$ com 9 vértices, além de um grafo randômico com 50 vértices e probabilidade de aresta igual a 10%, e a topologia da Rede Nacional de Pesquisa (RNP). São simuladas falhas de um enlace em cada grafo, são medidos o número de mensagens geradas e o tempo necessário para que o diagnóstico se complete. Os resultados indicam que o tempo necessário para realizar o diagnóstico é, na média, menor que o pior caso apresentado, e que o número de mensagens disseminadas é freqüentemente menor que o requerido por outros algoritmos semelhantes.

Abstract

This work presents a new algorithm for distributed diagnosis of general topology networks. A mobile agent visits all links sequentially, following the path generated by the Chinese Postman algorithm. The agent, called Chinese Agent, tests the links detecting new events and disseminates event information to the rest of the network. When all nodes of the system receive the information about the event, the diagnosis is complete. This work assumes that faults do not partition the network, and that a new event only occurs after the previous event has been diagnosed. Rigorous proofs of the best and the worst case of the latency of the algorithm are presented. Experimental results are also presented which were obtained from the simulation of the algorithm on different types of topologies, like hypercubes with 16, 64 and 128 nodes, the $D_{1,2}$ graph with 9 nodes, a random graph with 50 nodes and link probability equal to 10%, and the Brazilian National Research Network (RNP) topology. One link fault is simulated in each graph, both the number of messages and the algorithm's latency were measured. The results show that the time necessary to complete the diagnosis is, in average, smaller than the worst case. A comparison with other algorithms shows that the number of messages generated by the proposed algorithm is frequently smaller than the number of messages required by others similar algorithms.

Capítulo 1

Introdução

Devido à grande expansão das redes de computadores, vem crescendo a necessidade de sistemas de monitoração que possam detectar falhas de forma confiável a fim de se melhorar a confiabilidade do sistema como um todo.

Em grande parte dos sistemas de monitoração de redes atuais, apenas uma máquina é responsável por supervisionar a rede, o que é chamado de monitoração centralizada [1]. A monitoração centralizada não é perfeita, pois a máquina que faz a supervisão pode falhar e assim a rede não é mais monitorada; e pode ocorrer também uma grande concentração de mensagens, ou seja, a máquina responsável pela monitoração da rede irá receber um número excessivo de mensagens. Um sistema distribuído de monitoração é mais eficiente e confiável.

Nas próximas seções são apresentados conceitos básicos de diagnóstico em nível de sistema e uma visão geral de um novo algoritmo para diagnóstico de redes de topologia arbitrária proposto neste trabalho.

1.1 Diagnóstico em Nível de Sistema

A monitoração de uma rede pode ser realizada através da execução de um algoritmo distribuído de diagnóstico nesta rede (neste trabalho os termos sistema e rede são usados como sinônimos). Diversos modelos e algoritmos já foram propostos para diagnóstico em nível de sistema, e o modelo PMC [10] é o mais usado e o que desencadeou a idéia de diagnóstico em nível de sistema. Neste modelo o sistema é representado por um grafo, onde os nodos são os vértices e as arestas são os enlaces que interligam os nodos, sendo que as arestas são os caminhos por onde os testes serão executados. A asserção mais importante neste modelo é que os nodos *sem-falha* são totalmente confiáveis, ou seja, irão realizar os testes corretamente [11] e enviar informações corretas sobre os testes que eles executam.

Um algoritmo distribuído de diagnóstico permite que todos os nodos que estão operacionais obtenham informação sobre o estado de todos os nodos e enlaces, ou alternativamente sobre a conectividade do sistema. Algoritmos de diagnóstico têm sido classificados em duas categorias [3]: algoritmos que assumem sistemas representáveis por um grafo completo e algoritmos que assumem sistemas de topologia arbitrária. Este trabalho trata de algoritmos para sistemas de topologia arbitrária, isto é, entre um par de nodos do sistema pode ou não haver um enlace de comunicação.

Para se realizar o diagnóstico de um sistema de topologia arbitrária deve-se tratar de uma questão conhecida como ambigüidade de falhas, ou seja, quando é impossível distinguir se um nodo está *falho* ou se todos os enlaces de comunicação para tal nodo é que estão *falhos*. Por exemplo, na figura 1.1 é impossível, para qualquer outro nodo da rede, distinguir qual é o estado correto dos nodos representados em A e B, pois estas

configurações são ambíguas. Assim, ao invés de calcular quais nodos estão *falhos* e quais nodos estão *sem-falha*, o algoritmo de diagnóstico calcula quais nodos estão atingíveis e quais nodos estão inatingíveis.

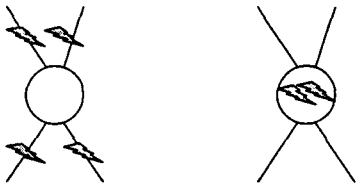


Figura 1.1: Configuração ambígua de falha.

Um algoritmo distribuído de diagnóstico de redes de topologia arbitrária tem sua execução normalmente dividida em três etapas: teste, disseminação e diagnóstico. Na etapa de testes, cada nodo executa um procedimento adequado de testes para determinar se um enlace está *sem-falha* ou não. A composição de cada teste deve ser decidida de acordo com a tecnologia do sistema específico. Na etapa de disseminação ocorre o envio de mensagens para todos os nodos da rede, informando sobre um novo evento (um enlace *sem-falha* se torna *falho*, ou um enlace *falho* se torna *sem-falha*). Na etapa de diagnóstico todos os nodos *sem-falha* identificam a porção da rede à qual estão conectados.

A eficiência de um algoritmo de diagnóstico é medida pelo número de mensagens que são utilizadas na etapa de disseminação e pelo intervalo de tempo necessário para se realizar o diagnóstico. O número de mensagens utilizadas na etapa de disseminação indica quantas mensagens são disseminadas pela rede até a conclusão do diagnóstico, nessa medida também são consideradas mensagens redundantes, ou seja, quando um determinado nodo recebe mensagens com a mesma informação mais de uma vez.

Entre os algoritmos para diagnóstico de redes de topologia arbitrária previamente publicados encontram-se: Bagchi & Hakimi [8], Adapt [14], Adapt2 [9], RDZ [2] e NBND [1,3]. Neste trabalho é apresentado um novo algoritmo para diagnóstico distribuído em redes de topologia arbitrária apresentado a seguir.

1.2 O Agente Chinês

O algoritmo do Agente Chinês, apresentado neste trabalho, realiza o diagnóstico do sistema, testando os enlaces, detectando novos eventos, e disseminando as informações obtidas para os demais nodos da rede. O Agente Chinês percorre um caminho determinado pelo algoritmo do *Carteiro Chinês* [26,27], que busca encontrar o menor percurso partindo de um nodo qualquer e retornando ao mesmo nodo passando por todos os vértices pelo menos uma vez, e *percorrendo cada aresta ao menos uma vez*. Um exemplo do funcionamento do algoritmo do Agente Chinês é ilustrado na figura 1.2 e descrito a seguir.

Considere que uma falha no enlace que liga o nodo 2 ao nodo 1 ocorre em algum instante de tempo. O percurso da rede começa no nodo 1, e o percurso então será de 1→3→4→2→1 e assim sucessivamente, até que ocorra o evento no enlace (2,1), e o nodo 2 detecte a falha. Nesse momento o Agente Chinês deve divulgar a informação sobre esse novo evento, enlace (2,1) *falho*, a todos os nodos, ou seja, deve ser iniciado um novo Agente Chinês que irá disseminar a mensagem. Porém, tem-se que o grafo remanescente da falha não é mais *euleriano* (figura 1.2-B) e, portanto deve ser transformado em tal (figura 1.2-C). Feito isso, o novo Agente Chinês continua seu caminho informando a todos os nodos sobre a falha do enlace (2,1), tendo como nodo inicial do percurso o nodo que detectou o evento. Portanto, o percurso agora será 2→4→3→1, e assim, todos os nodos da

Capítulo 2

Algoritmos de Diagnóstico em Redes de Topologia Arbitrária

Diversos modelos e algoritmos já foram propostos para diagnóstico em nível de sistema. Existem duas categorias principais de algoritmos de diagnóstico. Uma categoria realiza diagnóstico em redes representáveis por um grafo completo, ou seja, existe um canal de comunicação entre quaisquer dois nodos da rede [17]. A outra categoria, foco deste trabalho, considera redes de topologia arbitrária, ou seja, entre dois nodos do sistema pode ou não haver um canal de comunicação.

Nas próximas seções é apresentado o modelo mais estudado de diagnóstico: o modelo PMC [10]. São apresentados os algoritmos para diagnóstico em redes de topologia arbitrária propostos por Bagchi & Hakimi [8], seguidos pelos algoritmos Adapt [14], RDZ [2], NBND [3] e um algoritmo baseado em inundação de mensagens [12].

2.1 O modelo PMC

O modelo PMC [10] foi o modelo que desencadeou a idéia de diagnóstico em nível de sistema. Neste modelo o sistema é representado por um grafo, onde os nodos são os vértices e as arestas são os enlaces que interligam os nodos, sendo que as arestas são os caminhos por onde os testes serão executados. Nesse modelo cada nodo deve estar ligado com todos os demais, ou seja, o grafo que representa a topologia é um grafo completo e também não são consideradas falhas entre os enlaces, somente os nodos ficam *falhos*.

A asserção mais importante neste modelo é que os nodos *sem-falha* são totalmente confiáveis, ou seja, irão responder aos testes corretamente [11] e enviar informações corretas sobre os testes que eles executam. Os nodos que estão *falhos* podem ou não responder aos testes, porém suas respostas são arbitrárias. Por outro lado, os nodos *sem-falha* testam os outros nodos em um determinado intervalo de tempo pré-definido, enviando um sinal e esperando por uma resposta, se a resposta não chegar neste determinado intervalo de tempo já definido, o nodo testado é considerado *falho*.

O grafo de testes para esse sistema é um grafo direcionado, ou seja, as arestas são direcionadas exatamente como os testes são executados. Na figura 2.1 a aresta direcionada do nodo 1 para o nodo 2 indica que o nodo 1 testa o nodo 2, o mesmo ocorre para as outras arestas representadas na figura.

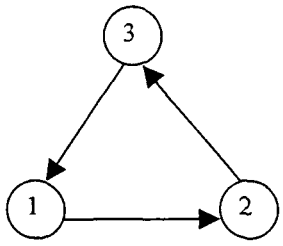


Figura 2.1: grafo de testes.

Nos primeiros algoritmos de diagnóstico em nível de sistema apenas um monitor central é responsável por realizar o diagnóstico da rede. O modelo distribuído surgiu mais tarde, um algoritmo distribuído de diagnóstico permite que todos os nodos que realizam os testes façam, eles próprios, o diagnóstico do sistema.

2.2 O Algoritmo de Bagchi & Kakimi

Em 1991, Bagchi & Hakimi [8] apresentaram um algoritmo para diagnóstico em redes de topologia arbitrária. O algoritmo funciona da seguinte maneira: no início cada nodo *sem-falha* conhece apenas seu próprio estado e o dos nodos vizinhos. Nodos *sem-falha* iniciam o algoritmo formando um grafo de testes através do qual as mensagens de diagnóstico são disseminadas. Cada nodo cria uma árvore por onde as informações serão disseminadas. Vários nodos podem iniciar ao mesmo tempo, portanto, vários grafos de testes podem ser montados, e se isso ocorrer, os grafos se unem formando apenas um. Os autores mostraram que o algoritmo requer no máximo $3n \log(p) + O(n+pt)$ mensagens, onde p é o número de nodos *sem-falha* e t é o número de nodos *falhos*.

O algoritmo requer que não haja mudança de estado dos nodos durante sua execução, ou seja, um nodo *sem-falha* não pode se tornar *falho* e um nodo *falho* não pode se tornar *sem-falha* durante a execução do algoritmo, portanto, é um algoritmo que é executado *off-line*.

Algumas características do algoritmo, além das citadas anteriormente, são: a disseminação das mensagens de diagnóstico ocorre seqüencialmente, usando a topologia representada pelo grafo, o tamanho dos pacotes é muito grande devido a quantidade de

informações que são armazenadas, várias árvores podem ser criadas e depois são unidas em apenas uma, podendo causar uma sobrecarga no processamento das informações.

2.3 O Algoritmo Adapt

O algoritmo apresentado em [14] e descrito formalmente em [15] permite o diagnóstico adaptativo e distribuído de redes de topologia arbitrária, sendo executado on-line detectando continuamente a ocorrência de falhas nos nodos. O algoritmo permite também que ocorram falhas dinâmicas e recuperação de nodos (nodos que passam do estado *falho* para *sem-falha*). A fim de evitar que mensagens redundantes sejam processadas, o Adapt utiliza um mecanismo para validação que descarta tais mensagens.

O algoritmo Adapt é dividido em três fases, denominadas pelos autores de: *Search*, *Destroy* e *Inform*, que no decorrer da descrição do algoritmo serão mais bem detalhadas.

2.3.1 Estrutura de dados

Cada nodo possui um único identificador chamado n_i e também um vetor denominado *Syndromes*. Este vetor contém em cada uma de suas posições uma lista dos nodos que foram testados, ou seja, o *Syndromes*[i] contém uma lista dos nodos que foram testados pelo nodo n_i com seus respectivos resultados, e um *timestamp* local que serve para identificar a idade da informação dos resultados dos testes.

Um nodo n_i recebe informações dos *Syndromes* de outros nodos através de mensagens de diagnóstico ou pacotes. Pacotes são usados para distribuir informações entre nodos e para indicar quando os nodos devem adaptar seus testes.

Os pacotes são divididos conforme as fases do algoritmo, e são de três tipos: *Search*, *Destroy* e *Inform*. Todos os pacotes contêm as seguintes informações: o identificador do nodo que originou o teste, uma cópia do vetor *Syndromes* que está armazenada no nodo e uma lista de nodos que já receberam e repassaram o pacote.

2.3.2 Descrição do Algoritmo

O algoritmo Adapt constrói uma sessão de testes adaptativa nas fases de *Search* e *Destroy*. Durante a fase de Search, testes são adicionados localmente em cada nodo para que se estabeleça uma sessão de testes.

2.3.3 A Fase de ***Search***

Durante esta operação, testes são realizados de maneira periódica, nenhum pacote é transmitido e todos os nodos contém o vetor *Syndromes* mais recente. A fase *Search* é executada da seguinte maneira: todos os nodos vizinhos são examinados e o teste de um vizinho é adicionado localmente se não houver nenhum caminho para aquele nodo na sessão de testes atual (é executada uma versão do algoritmo de Dijkstra para se determinar o menor caminho [16] no vetor *Syndromes* corrente). Um pacote *Search* contendo o novo vetor *Syndromes* é gerado e enviado para todos os nodos *sem-falha*.

Os nodos restantes executam o procedimento *Search* quando receberem o pacote *Search*, e se um nodo adicionar mais testes, um novo pacote *Search* é iniciado refletindo as atualizações do novo vetor *Syndromes*.

O diagnóstico correto é obtido quando a fase de *Search* estiver completa. A sessão de testes é fortemente conexa já que o procedimento *Search* é executado em todos os nodos *sem-falha* (contanto que a rede formada pelos nodos *sem-falha* seja conexa). Caso a rede

formada pelos nodos *sem-falha* não seja conexa, a sessão de testes é fortemente conexa em cada componente.

2.3.4 A Fase de ***Destroy***

É um complemento que serve para remover possíveis testes redundantes que foram introduzidos na fase de *Search*. O procedimento *Destroy* é executado em cada nodo e tem o seguinte funcionamento. O teste de um vizinho é removido se existem outros caminhos para aquele nodo no mesmo grafo de testes. *Destroy* deve ser executado seqüencialmente e em um nodo por vez, a fim de garantir que o grafo de testes continue fortemente conexo.

Os autores mostram que o número de testes é $N \leq |T| \leq 2(N-1)$, onde N é o número de nodos e T é o número de testes que são realizados.

2.3.5 Pacotes

A fim de que o algoritmo seja executado de maneira correta, o método *Search* deve ser executado em todos os nodos, possivelmente em paralelo, seguido pela execução, em seqüencial, do método *Destroy*. O algoritmo Adapt utiliza pacotes para garantir que essa seqüência seja respeitada. Pacotes de *Search* são iniciados por nodos que detectam uma falha, e uma vez que chegam a um outro nodo, fazem com que o método *Search* seja iniciado. O fato de existirem muitos pacotes *Search* faz com que o procedimento de *Search* seja executado em paralelo.

Quando a fase *Search* termina, um único pacote *Destroy* é iniciado e assim o método *Destroy* será executado seqüencialmente. Todos os pacotes possuem o vetor *Syndromes* e dessa maneira as informações mais recentes sobre os testes são distribuídas

para todos os nodos *sem-falha*. Os pacotes são enviados usando uma árvore de busca em profundidade calculada de forma distribuída no sistema.

2.3.6 A Fase de ***Inform***

Embora todos os nodos *sem-falha* possuam um diagnóstico correto, eles podem conter diferentes vetores *Syndromes* que são armazenados localmente durante a fase de *Destroy*. Um pacote de *Inform* serve para atualizar todas informações contidas em cada nodo, dentro do vetor *Syndromes*, para o estado atual dos testes. Um único pacote *Inform* contendo as informações mais recentes é enviado para todos os nodos *sem-falha*.

Algumas características do algoritmo são: alguns enlaces nunca são monitorados, impossibilitando o uso deste algoritmo para gerência de redes; um grande número de mensagens é armazenado, além do tamanho das mesmas também serem grandes.

.4 O Algoritmo RDZ

O algoritmo RDZ [2] funciona da seguinte maneira: os nodos testam seus vizinhos a intervalos de testes predeterminados, e propagam informações sobre novos eventos para restante da rede. Estas etapas são denominadas *detecção* e *disseminação.* A informação e é propagada é sobre *eventos*. Um evento é a transição do estado de um nodo de *sem-ha* para *falho* ou de *falho* para *sem-falha*.

A principal vantagem do algoritmo RDZ sobre os anteriores é que a disseminação é lizada de forma paralela, assim, a latência do algoritmo RDZ é proporcional ao diâmetro rede e é a melhor possível. No caso dos algoritmos Adapt [14] e Bagchi e Hakimi [8] a seminação é seqüencial. O algoritmo também permite que nodos que estavam *falhos* e

for a mesma, então o nodo *j* tem a mesma informação que o nodo *i* sobre o estado do nodo cujo evento está sendo reportado. Se a informação é mais antiga, significa que o nodo *j* tem informação mais recente que o nodo *i* sobre, pelo menos, um nodo. Se a informação for mais nova, então, o nodo *j* possui informação mais antiga do que o nodo *i* sobre o estado de pelo menos um dos nodos da rede e tem a mesma informação, ou mais recente, sobre o restante dos nodos. Desta forma, contém informação igual, mais nova ou mais antiga em relação a informação que o nodo já possui.

Se a informação recebida já é conhecida, então não é propagada adiante pelo nodo que a recebeu. Se a informação é mais velha, então a informação contida na mensagem é atualizada e enviada de volta somente para o nodo que enviou esta mensagem. Se a informação for mais nova, então a informação local do nodo receptor é atualizada com a informação recebida na mensagem. Em outras palavras, as informações locais são atualizadas e uma nova mensagem é disseminada.

Se um nodo *j* falhar ao enviar uma mensagem de confirmação quando este recebe uma mensagem de um nodo *i*, o nodo *i* irá detectar o evento de falha e iniciar a sua disseminação (uma transição do nodo *j* de *sem-falha* para *falho*). O nodo *i* assume que o nodo *j* está *falho*, mesmo que o nodo *j* já tenha se tornado *sem-falha* mas não respondeu com uma mensagem de confirmação porque estava *falho* quando recebeu a primeira mensagem vinda do nodo *i*. O caso descrito a seguir ilustra o uso das transações de validação como testes.

2.4.3 A Falha ***Jellyfish***

Um nodo que executa o algoritmo RDZ é chamado órfão se nenhum outro nodo está testando-o. Um nodo pode se tornar órfão *falho* ou órfão *sem-falha*. Se um nodo *sem-falha* se torna *falho*, seu testador detecta esse evento de falha devido a ocorrência de um *time-out* na resposta do teste, então o testador dissemina a informação deste evento e pára de testar o nodo. Portanto, todos os nodos *falhos*, depois de terem sido testados por seus respectivos testadores, tornam-se nodos órfãos *falhos*.

Quando um nodo *falho* é recuperado, ele procura um a um entre seus vizinhos por um nodo que o teste, até encontrar um vizinho que esteja *sem-falha*. Esse nodo vizinho *sem-falha* vai confirmar a requisição do nodo que acabou de ser recuperado e irá começar testá-lo. Dessa maneira, o nodo que foi recuperado não é mais um nodo órfão e o nodo que confirmou o teste inicia a disseminação da informação sobre o evento (órfão *falho* se torna *sem-falha*) para o resto dos nodos da rede, enviando uma transação de validação para todos os seus vizinhos *sem-falha*, que por sua vez enviam para seus vizinhos *sem-falha* e assim por diante.

Se o nodo testador de um nodo *sem-falha* tornar-se *falho*, o nodo que está sendo testado irá se tornar um órfão *sem-falha*, já que cada nodo é testado por apenas um único outro nodo. Nesse caso, o nodo órfão não irá necessariamente detectar de forma imediata que não há nenhum outro nodo testando-o. Eventualmente, um de seus vizinhos *sem-falha*, se houver algum, envia para o órfão novas informações sobre o estado de algum nodo da rede. Uma vez que isso acontece, o órfão percebe que não está sendo testado por nenhum outro nodo, e requisita a cada um de seus vizinhos que o teste, até que seja encontrado um vizinho *sem-falha* que confirme essa requisição. Um órfão *sem-falha* pode se tornar *falho*

antes que informações sobre a falha de seu testador sejam recebidas. Se isso ocorrer, o evento de falha (órfão *sem-falha* se torna órfão *falho*) não é detectado imediatamente. Eventualmente, o vizinho que envia informação sobre o estado da rede para o órfão *falho* percebe esse evento de falha quando não recebe uma confirmação do nodo órfão *falho* e, então, irá começar a disseminar a mensagem sobre o evento de falha.

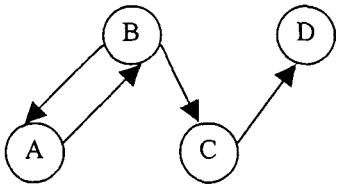


Figura 2.2: Configuração de falha *jellyfish*.

Tais eventos ocorrem com determinadas configurações chamadas de *jellyfish*. Neste tipo de configuração que ocorre a falha, existe, entre dois componentes conectados por testes, um conjunto de nodos tal que parte destes nodos realiza testes entre si de maneira cíclica, e ocorrem também outros testes que têm como origem o ciclo formado.

Considere o exemplo de configuração *jellyfish* da figura 2.2. Todos os nodos formam uma configuração *jellyfish* onde existe um ciclo (nodo A e nodo B) e testes que partem deste ciclo (nodo B para nodo C para nodo D). Um problema ocorre quando todos os nodos do ciclo da *jellyfish* se tornam *falhos* simultaneamente. Assim, os nodos que pertencem aos componentes conectados vizinhos ao ciclo não irão diagnosticar esta situação, pois não estão realizando testes nos nodos que compõem a *jellyfish*. Uma configuração *jellyfish* pode envolver apenas um nodo, ou um número arbitrário deles. Na figura 2.2, se ambos os nodos A e B se tornarem *falhos* ao mesmo tempo, os nodos C e D

não irão diagnosticar a falha. O mesmo ocorre se os nodos A, B e C se tornarem *falhos*, ou seja, o nodo D não irá detectar o evento.

O comportamento do algoritmo na ocorrência de configurações *jellyfish* é um dos pontos negativos do algoritmo RDZ, ou seja, o fato de alguns eventos serem detectados apenas eventualmente não permite que este algoritmo seja usado para uma aplicação mais prática de monitoração de rede.

2.4.4 Estrutura de Dados

Os dados necessários para a execução do algoritmo são mantidos tanto em estruturas de dados locais, encontradas em cada um dos nodos, como em estruturas de dados que são transportadas junto com as mensagens que circulam pela rede. O algoritmo faz referência a essas estruturas de dados como dados locais e dados de mensagem, respectivamente. A estrutura de dados descrita a seguir serve para ilustrar de maneira mais completa o funcionamento do algoritmo RDZ.

Considere que N é o número total de nodos da rede. As estruturas de dados locais de um nodo j são as seguintes:

- Um vetor de contadores de evento (*event-counters*), definido como $event_j$ [1..N], onde $event_j$ [i] contém o valor do contador, em relação ao evento mais recente que ocorreu no nodo i detectado pelo nodo que o estava testando naquele momento. Tais contadores de evento medem o número de ocorrências de eventos. Os contadores são inicializados em zero; quando um nodo *falho* fica *sem-falha* ou quando um nodo *sem-falha* fica *falho* há um incremento no valor do contador. Por exemplo, se um nodo k detecta um evento de falha em um nodo i, ocorre um incremento de uma unidade no $event_k$ [i]. Os contadores são incrementados de uma maneira que, após sua atualização, se o contador $event_j$ [i] possuir

um valor par, então o nodo *i* é considerado *sem-falha* pelo nodo *j*, e se o *event-counter* $event_j[i]$ for ímpar, então o nodo *i* é considerado *falho* pelo nodo *j*.

- Existe também um vetor que guarda informações sobre os vizinhos testados (*test/neighbor*), que é representado por $testnb_j[1..N]$, onde $testnb_j[i]$ pode ser 0, 1, 2, 3 ou 4. Se o nodo *i* não for vizinho do nodo *j* então o valor de $testnb_j[i]$ é 0, se o nodo *i* é testado por *j* o valor de $testnb_j[i]$ é 1. Quando o nodo *i* está testando o nodo *j* o valor de $testnb_j[i]$ é 2. Quando nem o nodo *i*, nem o nodo *j* estão testando-se um ao outro, porém são nodos vizinhos, o valor de $testnb_j[i]$ é 3. Finalmente, $testnb_j[i]$ vale 4 quando o nodo *i* testa o nodo *j* e vice-versa. Assumindo que apenas nodos vizinhos podem testar-se uns aos outros, se $testnb_j[i]$ for 1, 2, 3 ou 4 então *i* é um nodo vizinho de *j*, e se $testnb_j[i]$ for 0, o nodo *i* não é um vizinho de *j*.

Os dados que constam nas mensagens podem ser divididos em:

- Um vetor de contadores de evento (*event-counters*) chamado de $msg\_event[1..N]$, onde $msg\_event[i]$ contém o valor do contador, em relação ao evento mais recente que ocorreu no nodo *i*, detectado por algum outro nodo na rede. De forma similar ao *event-counter* que existe localmente em cada nodo, o valor de um *event-counter* na mensagem pode ser par ou ímpar. Se for par, significa que representa um nodo *sem-falha*, e um *event-counter* ímpar denota um nodo *falho*.

- Um vetor de bits chamado $intrapath[1..N]$, onde $intrapath[i]$ vale 1 se a mensagem já passou pelo nodo *i* anteriormente. Tal recurso é usado para reduzir o número de mensagens redundantes enviadas. O vetor *intrapath* é inicializado em 0 quando um nodo monta uma mensagem contendo as informações mais recentes que serão passadas adiante para outro nodo.

Tanto nas informações locais como nas informações que serão enviadas com a mensagem, não é necessário, além do que seria redundante, transportar dados sobre o estado dos nodos (*falho* ou *sem-falha*), pois é possível deduzir tais informações pelo *event-counter* de cada nodo, como foi descrito anteriormente.

As vantagens do algoritmo são a melhor latência possível e o menor número de testes realizados. A maior desvantagem é o comportamento do algoritmo na presença das falhas *jellyfish*. Os autores apresentam provas formais e resultados de simulação sobre a latência, número de testes realizados, dentre outras.

2.5 O Algoritmo NBND

O algoritmo NBND ("Non-Broadcast Network Diagnosis") permite o diagnóstico de falhas nos canais de comunicação da rede e o cálculo da conectividade sob o ponto de vista de qualquer nodo *sem-falha*. O algoritmo é executado em três fases: testes, disseminação e cálculo da conectividade. O menor número de testes possível é utilizado, cada canal de comunicação é testado por apenas um dos nodos por ele interligados. Assim como o algoritmo RDZ, a latência do algoritmo NBND é proporcional ao diâmetro da rede e é a melhor possível, pois os nodos disseminam informações sobre os eventos (transição do estado de um nodo de *sem-falha* para *falho* ou de *falho* para *sem-falha*) em paralelo.

2.5.1 Ambigüidade de Falhas

Antes de apresentar a descrição do algoritmo, é necessário fazer uma breve descrição das situações apresentadas na figura 2.3. Na situação A, está representada a

ocorrência de uma falha em que o estado do nodo é *sem-falha*, porém todos os enlaces deste nodo estão *falhos*. Na situação B, ocorre uma situação de falha onde o próprio nodo está *falho*.

O algoritmo NBND se baseia no fato de que é possível determinar se um determinado enlace está em *time-out* (não houve resposta dentro de período pré-definido) ou se está *sem-falha*. Entretanto, é impossível distinguir se um nodo está *falho* ou se todos os enlaces de comunicação para tal nodo é que estão *falhos*. Por exemplo, na figura 2.3 é impossível, para qualquer outro nodo da rede, determinar qual é o estado correto dos nodos representados em A ou B, pois estas configurações são ambíguas. Assim, ao invés de calcular quais nodos estão *falhos* e quais nodos estão *sem-falha*, o algoritmo calcula quais nodos estão atingíveis e quais nodos estão inatingíveis. Ou seja, os estados de um enlace são *sem-falha* e *timed-out* e os estados de um nodo são *sem-falha* ou *inatingível*.

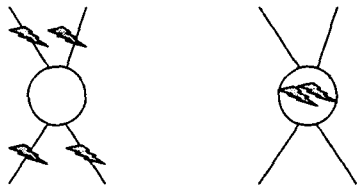


Figura 2.3: Configuração ambígua de falha.

Ao receber uma informação sobre o estado de um enlace, um nodo pode calcular qual porção da rede se tornou inacessível. Esta abordagem para realizar o diagnóstico da rede, baseada em *time-outs* dos enlaces e na inacessibilidade dos nodos, é mais próxima da realidade do que as abordagens apresentadas em algoritmos anteriores ao NBND.

único, o nodo com maior identificador será o testador. No caso deste nodo, ou do próprio enlace se tornar *falho*, o nodo de menor identificador, após detectar a inatividade do outro nodo, passa a executar testes. Este tipo de distribuição dos testes pode fazer com que alguns nodos realizem diversos testes, enquanto outros executam poucos. Uma estratégia mais justa é os nodos se alternarem como testadores e testados a cada intervalo de testes [13].

2.5.5 Disseminação de Informações Sobre Novos Eventos

Quando um novo evento é descoberto, o nodo testador dissemina a informação para todos seus vizinhos, em paralelo, e estes, por sua vez, repetem o processo. Para reduzir o número total de mensagens e de mensagens redundantes, estas contêm um campo no qual os identificadores de nodos que já sabem do novo evento são armazenados.

De maneira semelhante ao algoritmo RDZ, cada nodo mantém uma tabela com contadores que representam o estado de todos os nodos da rede. Estes valores são inicializados em zero, e incrementados de uma unidade cada vez que o nodo sofrer um novo evento. Assim, valores pares indicam nodos *sem-falha* e valores ímpares indicam nodos *falhos*.

2.5.6 Diagnóstico: Cálculo da Conectividade

Basta executar um algoritmo de conectividade no sistema representado como um grafo com as informações sobre o estado dos nodos, para que um nodo *sem-falha* determine a porção do sistema à qual está conectado.

2.5.7 Exemplo de Diagnóstico

Considere a rede representada pela figura 2.4. As arestas estão direcionadas do nodo testador para o nodo testado, e inicialmente todos os nodos estão *sem-falhas*.

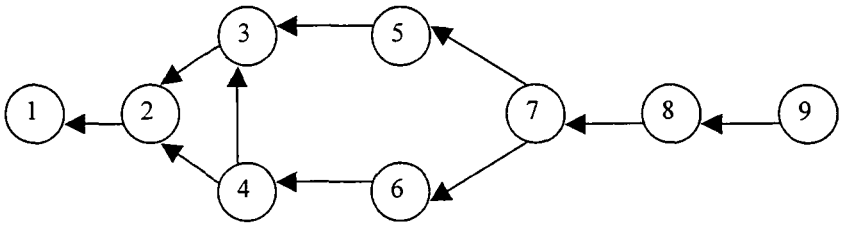


Figura 2.4: Grafo de testes da rede.

Considere que no instante de tempo t=0, os nodos começam a execução do algoritmo, e o intervalo de testes é de 30 unidades de tempo. O enlace entre o nodo 3 e o nodo 5 fica *falho* no tempo t=100. No próximo intervalo de testes, o nodo 5 detecta o evento, é o instante de tempo t=120. Ele dissemina mensagens para seu vizinho *sem-falha*, o nodo 7. Este processa a informação em t=150, e dissemina para seus vizinhos, os nodos 6 e 8. Neste intervalo, como o nodo 3 não recebeu informações sobre o canal que o liga ao nodo 5, ele o testa, e descobre o evento. A informação é disseminada para os nodos 2 e 4. O nodo 4 recebe mensagem também do nodo 6, referente ao mesmo evento. Neste momento, o nodo 9 recebe mensagem de diagnóstico do nodo 8. Os nodos 4 e 9 não disseminam mensagens, pois sabem que tanto o nodo 8, como o 2 e 6 já sabem do evento. Em t=180, o nodo 1 recebe a mensagem do nodo 2, completando o diagnóstico. São necessários 3

intervalos de teste para o diagnóstico completo, e são disseminadas 8 mensagens sobre o evento. Apenas o nodo 4 recebe duas mensagens sobre o mesmo evento.

O algoritmo NBND foi aplicado para gerência de falhas de redes, pois sendo distribuído e tolerante a falhas, permite que a contínua monitoração do sistema.

2.6 Um Algoritmo Baseado em Inundação de Mensagens

Outro algoritmo que permite que os nodos *sem-falha* de uma determinada rede realizem o diagnóstico da rede é o algoritmo baseado em inundação de mensagens [12].

Assim como o algoritmo NBND, o algoritmo de inundação é divido em três etapas: testes, disseminação e diagnóstico. Um nodo testa seus vizinhos e caso seja detectado um novo evento (enlace *sem-falha* se torna *falho*, ou enlace *falho* se torna *sem-falha*), uma mensagem com informações sobre esse evento é disseminada em paralelo para todos seus vizinhos. Cada nodo que recebe a mensagem também dissemina a mensagem aos seus vizinhos, exceto para o nodo que enviou a mensagem, e dessa maneira o processo se repete.

2.6.1 Descrição do Algoritmo

O algoritmo baseado em inundação de mensagens considera falha nos enlaces e exige que todos os nodos conheçam a topologia da rede. Em um intervalo de testes, todos os nodos testam todos os enlaces para seus vizinhos. Se todos os testadores não detectarem um novo evento, nenhuma mensagem é disseminada. No próximo intervalo de testes, os nodos executam os testes novamente, e assim por diante.

Quando um novo evento é detectado, mensagens são disseminadas notificando os demais nodos da rede. O nodo que detectou o evento envia a mensagem para todos os seus

vizinhos, os quais, por sua vez, também a enviam para todos os seus vizinhos, exceto para o(s) nodo(s) de onde veio a mensagem, e assim sucessivamente. Este processo de disseminação de mensagens utilizado neste algoritmo é chamado de inundação ou *flooding*.

Assim como no algoritmo RDZ e no NBND, cada nodo possui um contador de eventos, e tal contador é inicializado em zero, indicando o estado inicial *sem-falha*. A cada novo evento detectado, este contador é incrementado de uma unidade. Desta maneira, contadores com valores ímpares indicam que o enlace está *falho*, e contadores com valores pares indicam um estado *sem-falha*.

A estratégia de disseminar mensagens para todos os nodos vizinhos pode ocasionar um número grande de mensagens redundantes (mensagens que um nodo já recebeu anteriormente).

Considere uma topologia de rede representada na figura 2.5, onde o enlace entre o nodo A e B está *falho*, e as setas indicam o fluxo das mensagens entre os nodos. Quando o nodo A detecta o evento no enlace (A,B), começa a disseminação das mensagens. O nodo A envia mensagem para os nodos C e D, o nodo C envia mensagens para o nodo B e D, e ao mesmo tempo, o nodo D envia mensagens aos nodos C e B. Desta maneira, os nodos B, C e D recebem a mesma mensagem duas vezes, sendo disseminadas três mensagens redundantes.

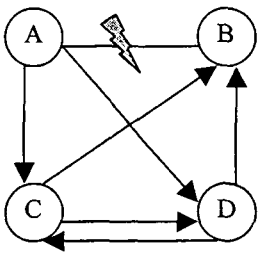


Figura 2.5: Fluxo das mensagens.

A fim de minimizar o impacto do recebimento de mensagens redundantes, o algoritmo utiliza um mecanismo para descartar tais mensagens. Quando um nodo recebe uma mensagem, ele compara as informações contidas na mensagem com seus próprios dados locais. Se as informações recebidas forem as mesmas ou mais antigas do que as informações que o nodo possui, então o nodo ignora a mensagem. Caso contrário, ele atualiza as informações locais e dissemina a mensagem para seus vizinhos.

O diagnóstico de um evento ocorre quando todos os nodos *sem-falha* recebem as mensagens referentes a um novo estado de um enlace. Assim como no NBND, a latência do algoritmo é a melhor possível, sendo proporcional ao diâmetro da rede.

Simulações e resultados experimentais são apresentados pelos autores, e os resultados mostram que o número de mensagens redundantes é, em média, menor que o número máximo possível, que é de duas vezes o número de enlaces.

No próximo capítulo um novo algoritmo para o diagnóstico distribuído de redes de topologia arbitrária é apresentado.

Capítulo 3

O Agente Chinês

Neste capítulo será apresentado o novo algoritmo para diagnóstico de redes de topologia arbitrária baseado no algoritmo do Carteiro Chinês, descrito inicialmente em [25]. As seções estão divididas da seguinte maneira: inicialmente é apresentada uma breve revisão sobre caminhamento em grafos, incluindo a apresentação do algoritmo clássico do Carteiro Chinês. Em seguida, o novo algoritmo distribuído é especificado. Provas sobre a latência do algoritmo no melhor e no pior caso são apresentadas no final do capítulo.

3.1 Caminhamento em Grafos

Antes de serem abordados os caminhamentos em grafos, é necessária uma breve explicação sobre algumas definições básicas de grafos.

Um grafo é um conjunto não vazio V de vértices e uma lista E de arestas que são pares não ordenados de elementos de V. Usa-se $G=(V,E)$ para denotar um grafo G com um conjunto de vértices V e uma lista de arestas E. Um grafo orientado é um grafo cujas arestas são direcionadas. Formalmente, um grafo orientado é um conjunto de vértices V e um

conjunto de pares ordenados (a,b) (onde $a, b \in V$) chamados arestas. O vértice a é o vértice inicial da aresta, e o b é o vértice final. Um *caminho* é uma seqüência de vértices $v_1v_2...v_k$ onde v_iv_{i+1} são arestas para todos $1 \leq i \leq k-1$. Uma *trilha* é um caminho sem arestas repetidas. Um *percurso* é um caminho sem vértices repetidos. Um *caminho fechado* é um caminho $v_1v_2...v_k$ onde $v_1=v_k$. Uma *trilha fechada* é um caminho fechado sem arestas repetidas. Se existe um percurso entre quaisquer pares de nodos em um grafo G então G é *conexo*. Caso contrário, G é *desconexo* e pode ser particionado em subgrafos conexos chamados componentes. A nomenclatura pode variar conforme o autor, e neste capítulo foi adotada a abordagem de [6, 20].

A primeira referência que se conhece sobre caminhamento em grafos vem do problema das sete pontes de Königsberg [5], hoje Kaliningrad, ilustradas na figura 3.1. O problema questionava se seria possível passar por todas as pontes somente uma vez visitando-se todos os pontos terrestres pelo menos uma vez. O matemático suíço Leonhard Euler se dispôs a tentar achar uma solução para o problema e mostrou que não havia uma solução para esse caso em particular.

Figura 3.1: Ilustração das pontes de Königsberg [4].

O grafo da figura 3.2 representa o problema, com as arestas representando as pontes, e os vértices representando os pontos terrestres. Do problema enfrentado por Euler, surgiu a noção de grafo *euleriano*: diz-se que um grafo *G conexo* e não-orientado é dito *unicursal* ou *euleriano* se existe uma trilha fechada ou um caminho *euleriano* em *G* contendo cada aresta apenas uma vez e cada vértice pelo menos uma vez. A condição necessária e suficiente para que um grafo *conexo* seja *euleriano* é dada pelo Teorema de Euler: um grafo *G* conexo é *euleriano* se e somente se todos os vértices possuírem *grau* par, ou seja, um número par de arestas deve incidir sobre todos os vértices. Uma prova formal desse teorema pode ser encontrada em [22].

A figura 3.3 é um exemplo de grafo *euleriano* chamado octaedro.

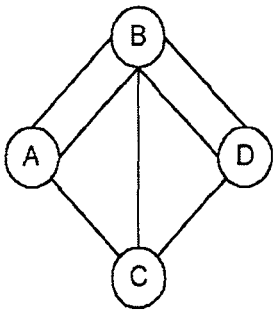


Figura 3.2: Representação do problema das sete pontes de Königsberg.

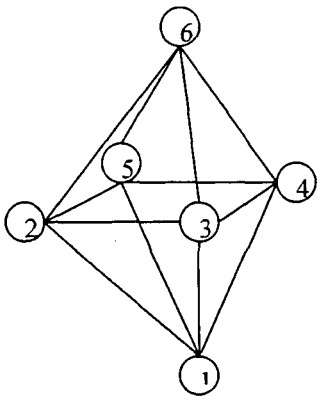


Figura 3.3: O octaedro é um grafo *euleriano*.

O problema das pontes de Königsberg foi de grande importância para o início dos estudos dos grafos e suas aplicações práticas como rotas de viagens, telefonia, entre outras. Muitas outras questões que são mapeadas em grafos recaem em um caso semelhante ao de determinar um caminho fechado sobre as pontes.

A solução para se encontrar caminhos fechados em grafos não *eulerianos* e outros relacionados é transformar grafos não *eulerianos* em *eulerianos*, ou seja, fazer com que os graus dos vértices ímpares se tornem par. Isto é possível para qualquer grafo finito e conexo, devido a um dos mais importantes lemas do estudo dos grafos, o chamado Handshake-Lemma [18, 19], que afirma que a soma dos graus dos vértices é par e o número de vértices de grau ímpar também é par. Dessa maneira, sempre haverá pares de vértices com grau ímpar para serem ligados uns aos outros.

3.1.1 O Algoritmo de Fleury

Um dos algoritmos para se achar um caminho *euleriano* (uma trilha fechada) possível, em um grafo *euleriano*, é chamado algoritmo de Fleury [6, 24].

Após verificar se o grafo é *euleriano*, o algoritmo funciona da seguinte maneira: partindo de qualquer nodo inicial, o algoritmo escolhe uma aresta ligada a esse nodo e a percorre chegando assim a um outro nodo. O próximo nodo realiza o mesmo passo e assim sucessivamente todos os nodos, até que todas as arestas tenham sido percorridas e o algoritmo volte ao nodo inicial. Quando o algoritmo percorre uma aresta, ela é removida, e esse fato garante que o algoritmo não irá percorrer a mesma aresta novamente, mantendo assim a característica de um caminho *euleriano*.

O algoritmo de Fleury não pode simplesmente ir percorrendo arestas, e as removendo, de maneira aleatória. Antes de ser removida, o algoritmo verifica se a aresta é

uma ponte ou não (uma aresta é uma ponte quando, se for removida, o grafo se particiona, não sendo possível chegar a qualquer nodo destino a partir de qualquer nodo de origem). Tal etapa é uma das mais importantes do algoritmo, já que se uma aresta ponte for removida, o grafo se particiona não sendo mais possível encontrar um caminho que retorne ao nodo inicial. Desta forma, uma aresta que é ponte só é percorrida caso seja a única aresta que um determinado nodo possui.

O algoritmo de Fleury pode ser resumido da seguinte maneira:

Begin
Escolher um vértice inicial *V*;
Do {
- Escolher uma aresta incidente em *V* (escolher ponte apenas se não houver outra alternativa);
- Remover aresta;
- Número_de_arestas = Número_de_arestas –1;
- *V* ← outro vértice da aresta escolhida;

} *While* (Número_de_arestas > 0);
End

Um exemplo de execução do algoritmo de Fleury está na figura 3.4. Na parte B da figura, as arestas direcionadas indicam um dos possíveis caminhos que o algoritmo percorre, podendo partir de qualquer nodo inicial.

Na figura 3.4-B, seja o nodo 3 o nodo inicial. O algoritmo de Fleury irá escolher uma aresta incidente ao nodo 3, no caso a aresta (3,4), pois a estratégia escolhida para o exemplo, foi a de se realizar um percurso escolhendo-se o nodo que possui o maior índice. As estratégias podem variar. A aresta (3,4) não é uma ponte e, portanto o algoritmo irá percorrê-la chegando ao nodo 4. O nodo 4 possui apenas uma aresta ligada a ele, já que a aresta (3,4) já foi visitada, e assim o algoritmo irá continuar sua execução percorrendo a aresta (4,5). Seguindo a mesma estratégia do percurso, o enlace (5,6) não é ponte e será

percorrido, seguido pelos enlaces (6,1) e (1,2), que são as únicas opções dos nodos 6 e 1 respectivamente. Caso a estratégia do percurso não fosse priorizada pelo nodo de maior índice, o algoritmo de Fleury poderia tentar percorrer o enlace (2,3), porém essa aresta é uma ponte (não seria mais possível chegar ao nodo 3 a partir de qualquer outro nodo) e assim, o algoritmo iria verificar se o nodo 2 possui outras arestas que não sejam pontes. Neste caso, o nodo 2 também está ligado ao nodo 5, e como essa aresta não é ponte, o algoritmo de Fleury iria escolher o enlace (2,5). Neste exemplo representado pela figura 3.4-B o algoritmo de Fleury irá testar primeiro o enlace (2,5) e irá percorrê-lo pois o mesmo não é ponte. Continuando sua execução, as arestas (5,2) e (2,3) são percorridas, e dessa maneira todos os enlaces são percorridos somente uma vez e todos os nodos são visitados pelo menos uma vez, completando-se assim o caminho *euleriano* no grafo. O caminho completo do algoritmo de Fleury neste caso é 3→ 4 → 5 → 6 → 1 → 2 → 5 → 2 → 3.

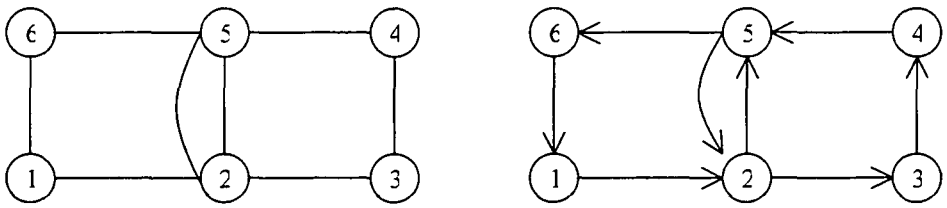


Figura 3.4: Grafo inicial (A) e o percurso do algoritmo de Fleury (B).

3.2 O Algoritmo do Carteiro Chinês

O algoritmo do Carteiro Chinês [6,26,27], cujo problema foi formulado por Meigu Guan em 1962, busca encontrar o menor percurso partindo de um nodo qualquer e retornando ao mesmo nodo passando por todos os vértices pelo menos uma vez, e *percorrendo cada aresta ao menos uma vez*. O algoritmo do Carteiro Chinês trata de um problema mais geral que o tratado pelo algoritmo de Fleury, já que o grafo não precisa ser *euleriano* e opcionalmente pesos podem ser adicionados às arestas, a fim de se encontrar efetivamente o menor percurso no grafo. No algoritmo do Carteiro Chinês, a idéia principal é minimizar a soma das distâncias passando ao menos uma vez por cada aresta e ao menos uma vez por cada nodo.

Um exemplo prático é a idéia de um carteiro que precisa entregar suas cartas e não deseja passar pela mesma rua mais de uma vez, ou seja, ele deseja otimizar a distância percorrida. Eventualmente, o carteiro pode perceber que, para sair do correio passando por todas as ruas e casas e retornando a sede do correio, será necessário repetir a passagem sobre uma ou mais ruas para que seu percurso seja completo e ele volte à agência do correio, mesmo assim ele deseja otimizar o caminho percorrido. No algoritmo de Fleury, cada aresta pode ser visitada apenas uma vez enquanto que no Carteiro Chinês, pode-se passar mais de uma vez pelas arestas. Essa repetição de "ruas" pode ser abstraída, apenas para fins de representação, como a criação de uma rua paralela à rua original que liga as mesmas duas casas.

Quando o grafo não é *euleriano*, deve haver uma etapa de transformação que torne possível a realização de um caminho *euleriano*, conforme o Teorema de Euler. A

transformação de um grafo não *euleriano* para um *euleriano* ocorre com a adição de novas arestas ao grafo original, tais arestas são referenciadas neste trabalho como arestas virtuais.

O exemplo ilustrado pela figura 3.5 indica a transformação de um grafo que não é *euleriano* para um grafo *euleriano*, através da adição das arestas virtuais, pois é impossível visitar todas as arestas somente uma vez com a topologia representada em 3.5-A. No exemplo da figura, os nodos 2, 3, 4 e 5 possuem grau ímpar e apenas uma aresta ligando-os ao nodo 1, e caso qualquer uma das arestas seja visitada, o grafo irá se particionar e não haverá outra aresta para se continuar o caminho. Após a transformação do grafo, representada na figura 3.5-B, todos os nodos possuem grau par e portanto existe um caminho *euleriano*.

Caso o grafo já seja *euleriano*, nenhuma transformação é necessária e o algoritmo do Carteiro Chinês irá realizar seu percurso visitando todas as arestas apenas uma vez e todos os nodos pelo menos uma vez. Caso o grafo não seja *euleriano*, existem diversas maneiras para se adicionar arestas virtuais, porém o algoritmo do Carteiro Chinês garante que sejam adicionadas o menor número possível e que o percurso será otimizado. As arestas virtuais são criadas apenas para fins de representação, pois o verdadeiro significado é que a aresta original é percorrida mais de uma vez, tantas quantas são as arestas virtuais que foram adicionadas.

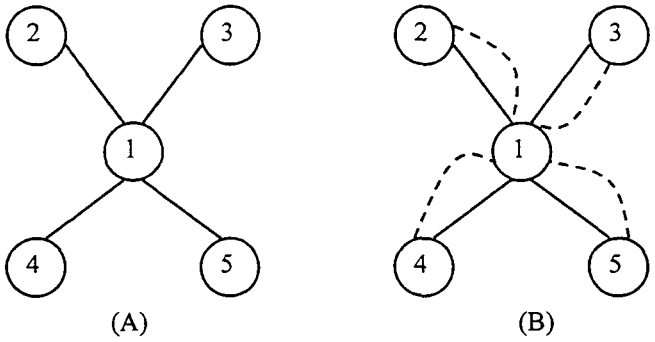


Figura 3.5: Grafo original (A) e grafo transformado (B).

Arestas virtuais não podem ser adicionadas entre dois nodos que já não possuam arestas originais ligando-os, pois caso isso ocorresse, uma nova ligação que não existe diretamente entre dois nodos estaria sendo criada. A figura 3.6 ilustra uma transformação incorreta de um grafo não *euleriano* para um grafo *euleriano*. Em 3.6-A é possível visualizar o grafo original com dois nodos que possuem grau ímpar e, portanto, o grafo deve ser transformado em *euleriano*. Como já existem arestas entre os nodos 1 e 2, e 2 e 3, as arestas virtuais devem ser adicionadas entre esses nodos, conforme mostra 3.6-B. Adicionando-se uma aresta entre os nodos 1 e 3, como ilustrado em 3.6-C, todos os nodos também possuem grau par e o grafo é *euleriano*, sendo que foram adicionadas menos arestas virtuais. Porém, a figura 3.6-C não respeita a topologia original do grafo, criando uma aresta que não existia originalmente entre os nodos 1 e 3.

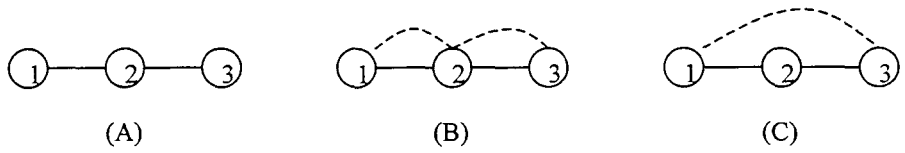


Figura 3.6: Grafo original (A), grafo transformado corretamente (B) e incorretamente (C).

3.2.1 Especificação do Algoritmo

O algoritmo do Carteiro Chinês trabalha com pesos adicionados as arestas. Segue abaixo a especificação do algoritmo:

1. Primeiro, calcular o valor do menor percurso entre quaisquer dois vértices podendo ser usado o algoritmo de Dijkstra quantas vezes forem necessárias.
2. Criar uma matriz M = $[m_{ij}]$ de tamanho $|V|$ x $|V|$ (onde $|V|$ é o número de vértices de grau ímpar). A matriz M_{ij} é preenchida com os valores do menor

percurso entre um vértice $v_i \in V$ e outro vértice $v_j \in V$, (onde V é o conjunto dos vértices de grau ímpar).

3. Encontrar a melhor combinação de quaisquer dois vértices que estejam em V. Ou seja, encontrar a menor soma dos valores dos percursos que foram calculados nos passos 1 e 2.
4. Usando Dijkstra, é calculado o menor caminho entre as combinações de vértices obtidas no passo anterior, e são adicionadas arestas virtuais nesses caminhos.
5. O grafo já está transformando em *euleriano* e resta descobrir os caminhos *eulerianos*.

A figura 3.7 representa um grafo não *euleriano* com peso nas arestas e cujos vértices 1,3,6 e 11 possuem grau ímpar.

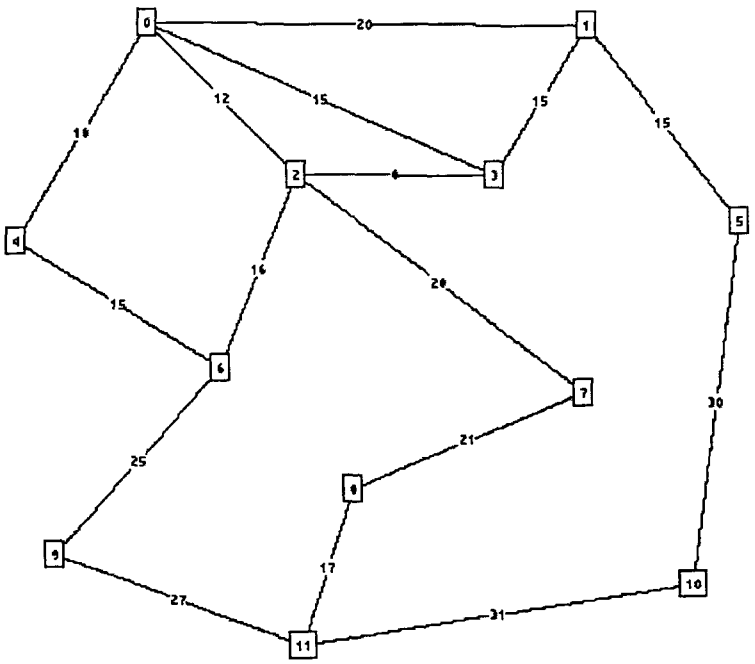


Figura 3.7: Grafo inicial não *euleriano* com peso nas arestas.

Após realizar os passos 1 e 2, a menor distância entre esses vértices pode ser mapeada na seguinte tabela:

V	1	3	6	11
1	-	15	39	76
3	15	-	24	76
6	39	24	-	52
11	76	76	52	-

Ao se realizar o passo 3, encontra-se a menor soma de tamanho dos percursos possíveis, que está representada na tabela a seguir:

V	1	3	6	11
1	-	15	39	76
3	15	-	24	76
6	39	24	-	52
11	76	76	52	-

Como o enlace (1, 3) é igual ao enlace (3,1), apenas a parte superior direita da tabela é relevante para os resultados, como mostra a tabela seguinte:

V	1	3	6	11
1	-	15	39	76
3	15	-	24	76
6	39	24	-	52
11	76	76	52	-

É necessário encontrar o menor percurso entre os vértices 1 e 3 e 6 e 11, que são as arestas (1, 3), (6,9) e (9, 11), e assim, três arestas virtuais serão adicionadas, ligando os nodos (1,3), (6,9) e (9,11), como mostra a figura 3.8. Um possível caminho *euleriano* é 0→1→5→10→11→8→7→2→3→1→3→2→6→9→11→9→6→4→0, que também está indicado na figura 3.8 na forma de setas direcionadas.

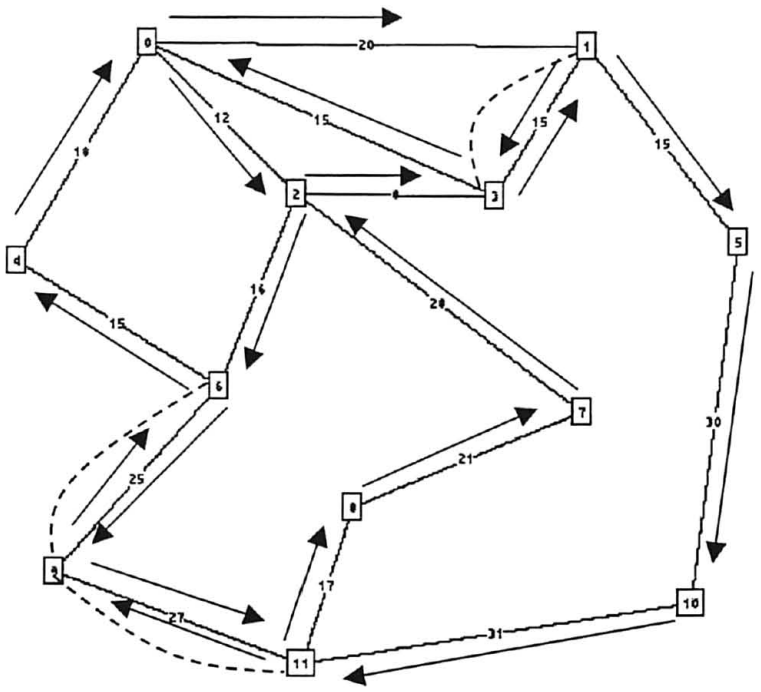


Figura 3.8: Grafo com arestas virtuais e um possível caminho *euleriano*.

Na próxima seção é descrita uma versão distribuída do algoritmo do Carteiro Chinês, aplicada para diagnóstico distribuído de redes de topologia arbitrária.

3.3 O Agente Chinês

Nesta seção é apresentado um novo algoritmo baseado em um agente distribuído e tolerante a falhas, o Agente Chinês, que percorre o caminho determinado pelo algoritmo do Carteiro Chinês, para diagnóstico em nível de sistema de redes de topologia arbitrária. O algoritmo do Agente Chinês utiliza o modelo PMC e realiza diagnóstico sobre eventos que ocorrem nos enlaces, ou seja, um enlace pode estar em um estado de *falha* ou *sem-falha*. Assume-se que um evento não particiona a rede e que o diagnóstico de um evento é completado antes da ocorrência do próximo evento.

3.3.1 Descrição do Algoritmo

O Agente Chinês, que é iniciado em um nodo qualquer, fica percorrendo todos os enlaces da rede indefinidamente até que seja encontrado um evento. Existe um intervalo de tempo pré-definido para que o Agente Chinês seja transmitido sobre um enlace, por exemplo, caso o intervalo seja um, a cada uma unidade de tempo o Agente Chinês percorre um enlace. Ou seja, em uma topologia com 3 enlaces, o Agente Chinês irá gastar 3 intervalos para percorrer todos os enlaces.

O caminho percorrido é determinado pelo algoritmo do Carteiro Chinês. O teste de um enlace é feito através do próprio envio do Agente Chinês de um nodo ao próximo nodo no caminho.

O Agente Chinês leva mensagens para disseminar as informações de diagnóstico para todos os nodos, e tais mensagens são formadas por três campos: o primeiro é o identificador do nodo de origem, o qual detectou um evento. O segundo campo é o identificador de um nodo, e junto com o primeiro campo, identificam o enlace no qual ocorreu o evento, e o terceiro campo é o contador de eventos. Quando o Agente Chinês percorre os nodos, ele transporta consigo essa mensagem. A figura 3.9 ilustra o conteúdo da mensagem.

Identificador do nodo que detectou o evento	*Identificador do nodo da outra ponta do enlace que ocorreu o evento.*	*Contador de eventos*

Figura 3.9: Mensagem que é enviada.

O campo contador de eventos é inicializado com 0 (zero), indicando que ainda não houve nenhum evento. A cada detecção de um evento o valor do contador de eventos é incrementado de uma unidade.

A figura 3.10 representa o percurso de um Agente Chinês sendo iniciado no nodo 1, sendo que inicialmente não há informações sobre eventos na mensagem. O nodo 1 atualiza o valor do campo identificador do nodo de origem, contido na mensagem, para o valor 1 e envia o Agente Chinês para o nodo 2. O nodo 2 atualiza o campo identificador para valor 2 e envia o Agente Chinês para o nodo 3, e assim sucessivamente conforme mostra o fluxo da figura. O segundo campo da mensagem não é preenchido com nenhum valor e o campo contador de eventos é igual a 0 (zero), pois nenhum evento foi detectado.

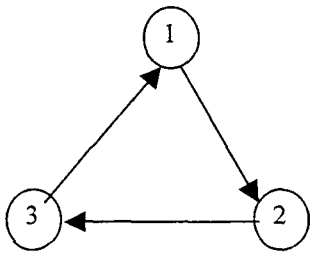


Figura 3.10: Agente Chinês e suas mensagens.

A falha de um enlace é detectada quando o Agente Chinês não consegue percorrer um determinado enlace e ir para o próximo nodo. Nesse caso, o Agente Chinês, que está no nodo que detectou o evento de falha, irá recalcular a nova topologia da rede (conforme o algoritmo do Carteiro Chinês), atualizar os campos de sua mensagem e começar a disseminação da informação sobre o evento.

A figura 3.11 ilustra a ocasião que um Agente Chinês detecta uma falha. Supõe-se que em algum momento o nodo 3 tenha detectado uma falha no enlace (3,1) e, portanto, o Agente Chinês não irá conseguir ir para o nodo 1. Neste momento, a topologia da rede é recalculada, e são adicionadas duas arestas virtuais: uma entre os nodos 1 e 2, e outra entre os nodos 2 e 3. O nodo 3 atualiza o primeiro campo da mensagem com o valor 3, isto é, o identificador do nodo que detectou o evento, preenche o segundo campo da mensagem com o identificador do nodo da outra ponta do enlace onde o evento ocorreu, incrementa de uma unidade o valor do campo contador de eventos da sua mensagem, e inicia a disseminação da informação sobre o evento para os outros nodos da rede.

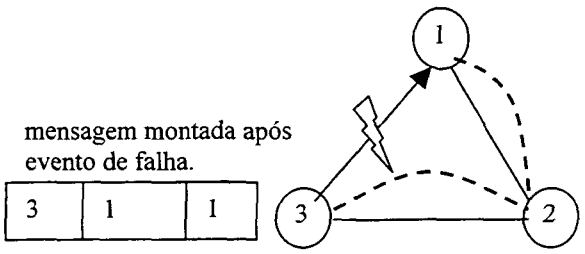


Figura 3.11: Agente Chinês detecta a falha.

Quando está ocorrendo a disseminação das mensagens, o primeiro campo da mensagem não é alterado conforme o identificador dos nodos que estão sendo visitados, ou seja, o primeiro campo permanece com o valor do identificador do nodo que detectou o evento. Seja o grafo original o exemplo da figura 3.11, a figura 3.12 ilustra o processo de diagnóstico e a disseminação das mensagens. O nodo 3 envia a mensagem ao nodo 2 avisando sobre o evento que ocorreu no enlace (3,1), e a seguir, o nodo 2 envia a mensagem para o nodo 1. Dessa maneira, todos os nodos já receberam a informação sobre o evento. Foram necessárias 2 mensagens disseminadas e 2 intervalos de tempo para se completar o diagnóstico. O Agente Chinês irá continuar realizando seu caminho, ou seja, partindo do nodo 1 e indo para o nodo 2, e posteriormente ao nodo 3, e assim sucessivamente de acordo com o caminho determinado pelo Carteiro Chinês, porém o diagnóstico já foi realizado.

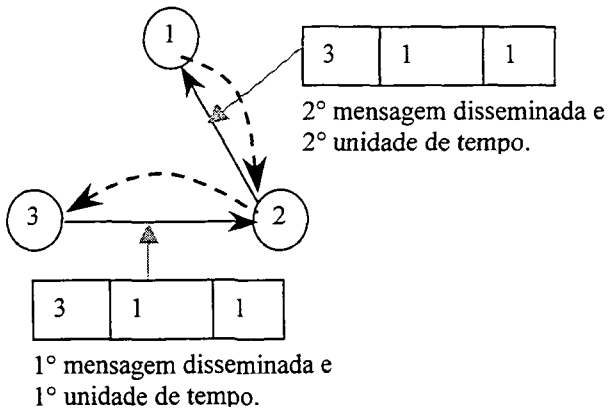


Figura 3.12: Disseminação da mensagem após evento.

3.3.2 Critérios para Avaliação do Algoritmo

O desempenho do algoritmo do Agente Chinês é medido de acordo com três critérios: tempo para diagnóstico de um evento, ou seja, tempo necessário para que todos os

nodos da rede saibam da ocorrência de um evento; número de mensagens que são transmitidas para se realizar o diagnóstico, e o número de mensagens redundantes que são disseminadas até que se realize o diagnóstico. Quando um nodo recebe uma mensagem que contém informação sobre um evento já conhecido por ele, tal mensagem é redundante, ou seja, o nodo já havia recebido uma outra mensagem com a mesma informação.

3.4 Latência do Algoritmo

O intervalo de tempo necessário para que seja realizado o diagnóstico do sistema, ou seja, quando todos os nodos *sem-falha* recebem informação sobre um evento, é denominado latência. Nesta seção é abordada a latência no melhor e no pior caso.

3.4.1 Melhor Caso da Latência

Teorema 1: no melhor caso, o Agente Chinês necessita de N-1 intervalos para que todos os nodos recebam informação sobre um evento e o diagnóstico seja completo, onde N é o número de nodos do sistema.

Considere que o sistema está conectado e existe um caminho partindo do nodo que detectou o evento e passando pelos demais nodos do sistema sem utilizar arestas virtuais. Neste caso, a distância do primeiro ao último nodo é igual a N-1 arestas.

Como o diagnóstico só se completa quando todos os nodos receberem o Agente Chinês, e um dos nodos é o que detecta o evento, o Agente Chinês deverá passar por no mínimo N-1 nodos para que o evento seja completamente diagnosticado. Desta forma, o resultado acima indica a latência do algoritmo no melhor caso. ■

Para exemplificar o resultado deste teorema considere os grafos da figura 3.13. Os grafos representam a topologia do sistema após a ocorrência de algum evento. Na figura 3.13-A, considere que o nodo 1 detectou algum evento e irá começar a disseminação da mensagem. O Agente Chinês será enviado ao nodo 2, posteriormente ao nodo 3, ao nodo 4 e finalmente ao nodo 5, seguindo o trajeto 1→2→3→4→5. Ou seja, foram necessários 4 intervalos, N-1 intervalos, para o diagnóstico completo do sistema. Observar que as arestas virtuais não são usadas neste exemplo para que o diagnóstico se complete. Na figura 3.13-B, considere que o nodo 3 detectou um evento e irá iniciar a disseminação da mensagem. O Agente Chinês irá realizar o percurso 3→4→1→2, e assim, todos os nodos receberam informações sobre o evento ocorrido no sistema. Neste caso foram necessários 3 intervalos, N-1, para o diagnóstico completo.

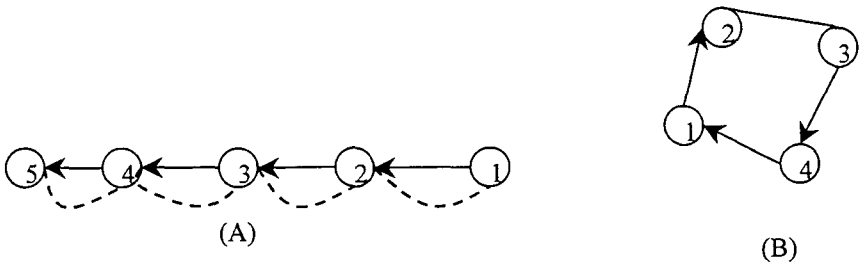


Figura 3.13: Grafos dos sistemas após a ocorrência de algum evento.

3.4.2 Pior Caso da Latência

Teorema 2: No pior caso, o Agente Chinês necessita de 2*L-1 intervalos para que todos os nodos recebam informação sobre um evento e o diagnóstico seja completo, onde L é o número de enlaces do sistema original (antes da transformação para *euleriano*).

Como o diagnóstico só se completa quando todos os nodos receberem o Agente Chinês, e um dos nodos é o que detecta o evento, o pior caso do algoritmo ocorre quando o

agente passa por um enlace virtual, além do enlace real, para alcançar cada nodo antes de completar seu caminho. Desta forma, se para cada enlace existe um enlace virtual correspondente, o número total de arestas no caminho do Agente Chinês será $2*L$. Entretanto, quando o Agente Chinês alcança o último nodo, o diagnóstico já se completou e não é necessário contabilizar o intervalo referente à passagem pelo enlace virtual que liga este último nodo a um outro nodo. Portanto, o número de intervalos necessários para completar o diagnóstico é $2*L$-1. ■

Para exemplificar o resultado deste teorema considere os grafos da figura 3.14. Os grafos representam a topologia do sistema após a ocorrência de algum evento. Na figura 3.14-A, considere que o nodo 2 detectou algum evento e irá começar a disseminação da mensagem. O Agente Chinês será enviado ao nodo 3, posteriormente ao nodo 4, e finalmente ao nodo 5, porém o agente deve retornar passando pelos nodos 4, 3 e 2 novamente, pois o nodo 1 ainda não recebeu a mensagem sobre o evento. Dessa maneira, o trajeto seguido será: 2→3→4→5→4→3→2→1. Ou seja, foram necessários 7 intervalos, $2*L$-1, para o diagnóstico completo do sistema. Na figura 3.14-B, considere que o nodo 1 detectou um evento e irá iniciar a disseminação da mensagem. O Agente Chinês irá realizar o percurso 1→6→1→5→1→4→1→3→1→2, e assim, todos os nodos receberam informações sobre o evento ocorrido no sistema. Assim, foram necessários 9 intervalos, $2*L$-1, para o diagnóstico completo.

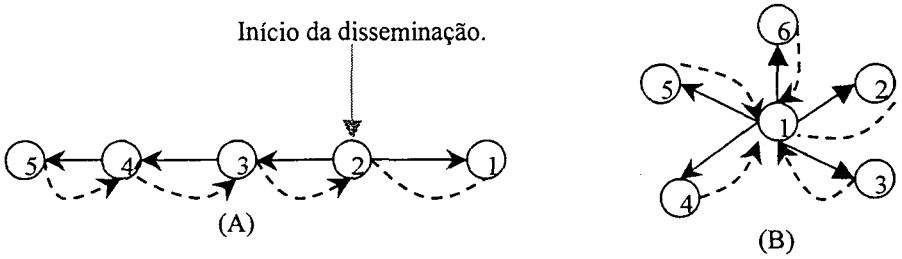


Figura 3.14: Grafos dos sistemas após a ocorrência de algum evento.

No capítulo seguinte são apresentados resultados de simulações utilizando diversas topologias, além de comparações do algoritmo do Agente Chinês com um algoritmo baseado em inundação de mensagens e com um algoritmo baseado no NBND [3]. Os resultados experimentais obtidos através de simulação apresentados no próximo capítulo indicam que a latência é, na média, muito menor que o pior caso apresentado acima.

Capítulo 4

Resultados de Simulação

Neste capítulo são apresentados resultados experimentais obtidos através da simulação do algoritmo de diagnóstico em redes de topologia arbitrária especificado no capítulo 3. O algoritmo foi executado em diversos tipos de topologias, dentre elas um grafo exemplo com 7 vértices, um grafo $D_{1,2}$ com 9 vértices, grafos hipercubos de 16, 64 e 128 vértices, além de um grafo randômico e a topologia da Rede Nacional de Pesquisa (RNP) [23]. Estas topologias representam um conjunto significativo que permite a avaliação do algoritmo e sua comparação com outros previamente publicados. Foram simuladas falhas de um enlace em cada grafo; e um dos nodos cujo enlace está conectado detecta o evento e inicia a disseminação das mensagens.

A linguagem de simulação de eventos discretos *SiMulation Programming Language – SMPL* [7] foi utilizada, e nestas simulações, o intervalo de testes convencionado e o intervalo de disseminação foi de 1 (uma) unidade de tempo. A seguir são apresentadas as descrições das simulações e os resultados obtidos, assim como uma comparação com um algoritmo baseado em inundação [12] e um algoritmo baseado no NBND [3]. Os grafos foram implementados e representados usando listas de adjacências [21].

4.1 Um Grafo Exemplo

Considere o grafo exemplo apresentado na figura 4.1-A. O algoritmo inicia a sua execução em t=0, e o enlace (1,3) se torna *falho* em um determinado instante de tempo t. Antes do Agente Chinês iniciar sua execução o algoritmo transforma o grafo em *euleriano*. A figura 4.1 ilustra a transformação do grafo com uma aresta virtual sendo adicionada entre os nodos 6 e 3.

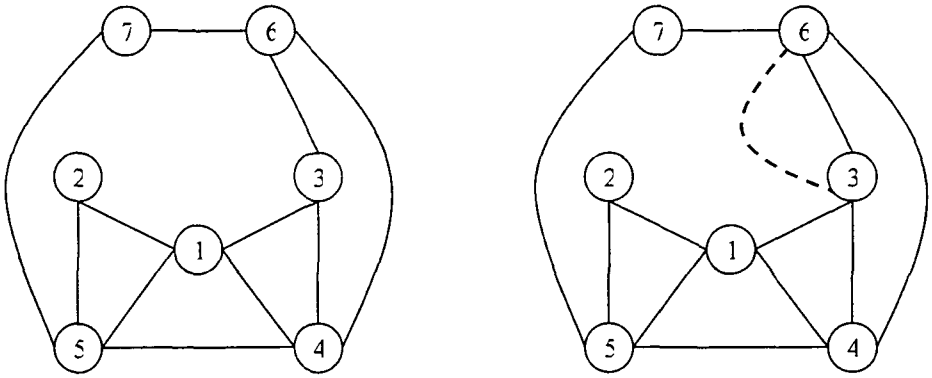


Figura 4.1: Grafo inicial (A) e grafo transformado em *euleriano* (B).

A execução do Agente Chinês começa no nodo 1, e assim, o nodo 1 tenta enviar o Agente Chinês para o nodo 2, como não há falha, o Agente Chinês é enviado com sucesso. Dessa maneira, o Agente Chinês irá percorrer o caminho determinado pelo algoritmo do Carteiro Chinês, ou seja, depois do nodo 2, o Agente Chinês irá para o nodo 5, e realizar o caminho 5→1→4→3→6→4→5→7→6→3 e finalmente irá retornar ao nodo de origem 1, conforme ilustra a figura 4.2. Tal caminho é realizado indefinidamente, ou até que seja detectado um evento. Como não houve evento, a mensagem que o Agente Chinês envia contém somente o identificador do nodo testador, e o valor do contador de eventos igual a 0

(zero), conforme também ilustra a figura 4.2. Quando o Agente Chinês está prestes a ser enviado para o próximo nodo, o valor do campo nodo testador, que está contido na mensagem, é substituído pelo identificador do respectivo nodo que está enviando a mensagem. Por exemplo, quando o nodo 1 envia o Agente Chinês para o nodo 2, o valor do campo nodo testador é 1 (um), e quando o nodo 2 envia o Agente Chinês para o nodo 5, o valor do campo nodo testador é alterado para 2, e assim sucessivamente.

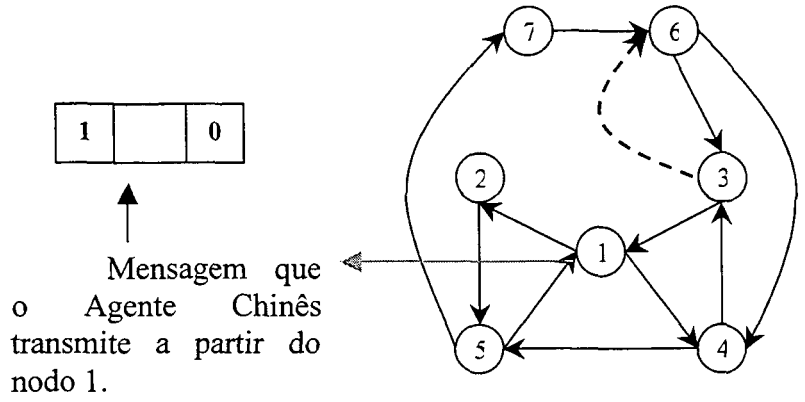


Figura 4.2: Caminho do Agente Chinês.

Em determinado instante de tempo *t*, o Agente Chinês que está no nodo 3 irá detectar um evento no enlace (3,1) e não irá conseguir continuar seu caminho que o levaria ao nodo 1, conforme representado na figura 4.3.

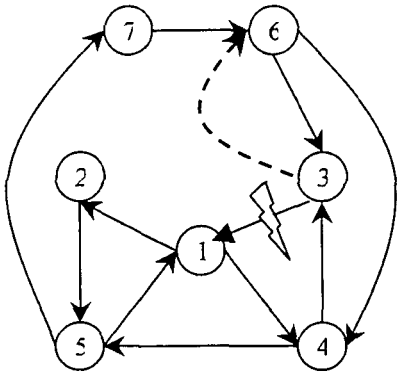


Figura 4.3: Falha no enlace (3,1).

Neste momento, o nodo 3 prepara-se para iniciar o processo de disseminação de mensagens sobre o novo evento detectado. A mensagem de disseminação contém o identificador do nodo que detectou o evento, que é igual a 3, o identificador do nodo da outra ponta do enlace que sofreu o evento, que é 1, e o contador de eventos é incrementado para 1, conforme ilustrado na figura 4.4-B. A topologia do grafo também é recalculada a fim de transformar o grafo, após a detecção de um evento, para um grafo que seja *euleriano*, e assim o Agente Chinês pode continuar seguindo seu caminho. É importante salientar novamente que a nova topologia do grafo é calculada conforme a topologia do grafo original, e não conforme a topologia do grafo após ser transformado pela primeira vez em *euleriano* (caso tenha havido essa transformação). O grafo após a transformação irá conter uma aresta virtual entre os nodos 1 e 4, e 6 e 4, conforme ilustra a figura 4.4-A.

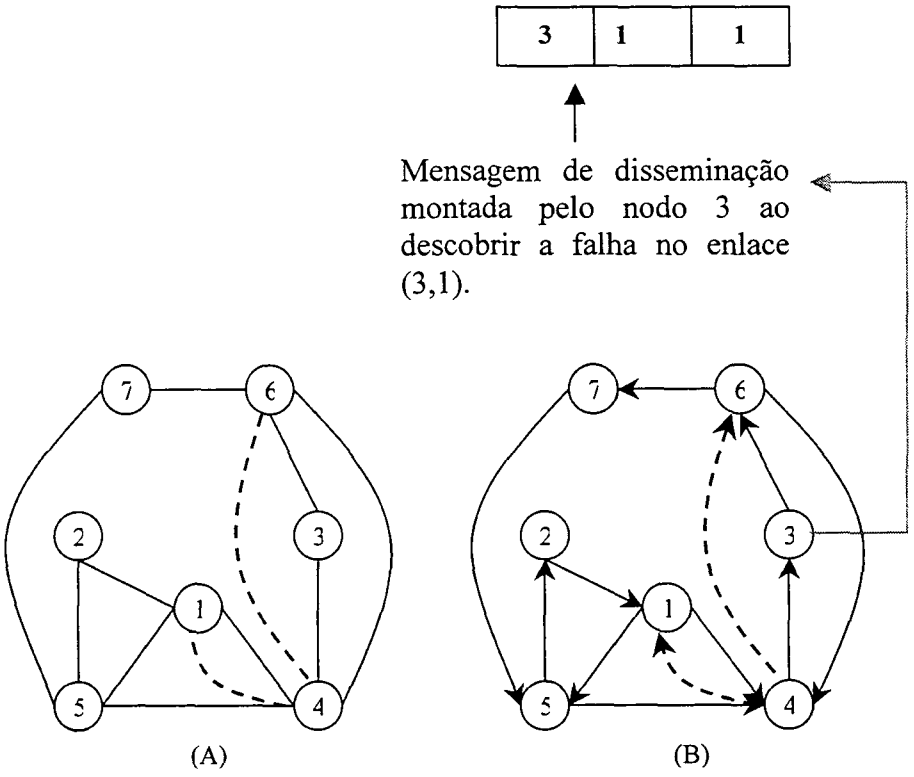


Figura 4.4: (A) Grafo *euleriano* após falha, (B) caminho de disseminação.

O nodo 3 inicia o evento de disseminação da mensagem, e assim, todos os nodos *sem-falha* do sistema irão receber a informação sobre o evento no enlace (3,1). Conforme foi definido, o intervalo de disseminação é de 1 (uma) unidade de tempo e, portanto, o nodo 3 inicia a disseminação em um tempo $t=0$, e a cada 1 (uma) unidade de tempo o Agente Chinês leva a mensagem para outro nodo. O Agente Chinês irá percorrer o seguinte caminho que se inicia no nodo 3: 3→6→4→6→7→5→2→1, e quando a mensagem chegar ao nodo 1, o diagnóstico está completo, ou seja, todos os nodos *sem-falha* foram informados sobre o evento. O nodo 6 recebe a mesma mensagem 2 vezes, ou seja, o diagnóstico é realizado com a ocorrência de 1 (uma) mensagem redundante.

Em resumo os resultados são apresentados na tabela 4.1 abaixo:

Total de mensagens	**Mensagens redundantes**	**Latência (caso atual)**	**Latência (melhor caso)**	**Latência (pior caso)**
7	1	7	6	21

Tabela 4.1: Resultados da simulação para o grafo exemplo com 7 nodos.

Neste exemplo o número total de mensagens na rede foi 7, sendo que uma dessas mensagens foi redundante. O tempo total para completar o diagnóstico foi de 7 unidades de tempo.

4.2 Grafo $D_{1,2}$

A figura 4.5 apresenta um grafo *euleriano* que representa uma rede de topologia $D_{1,2}$ com nove vértices. A execução do algoritmo inicia em $t=0$ e o enlace entre o nodo 6 e o nodo 8 torna-se *falho* em um determinado instante de tempo. O Agente Chinês inicia seu caminho a partir do nodo 1, e segue percorrendo os seguintes nodos:

1→9→2→4→3→5→4→6→5→7→8→6→7→9→8→1→3→2 e retornando ao nodo 1, dando início ao processo novamente até que seja detectado um evento. A figura 4.6 representa o caminho que o Agente Chinês percorre a partir do nodo 1.

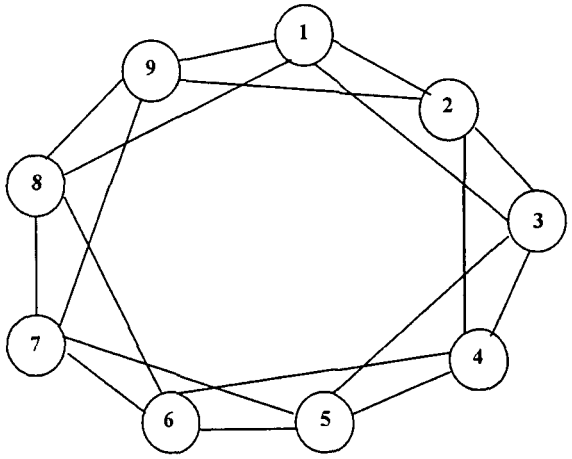


Figura 4.5: Grafo $D_{1,2}$ representando a topologia da rede.

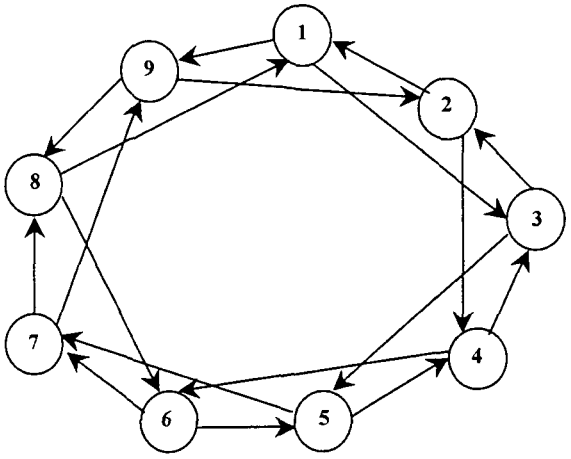


Figura 4.6: Caminho do Agente Chinês.

Em determinado instante de tempo *t*, o Agente Chinês, que está no nodo 8, irá detectar um evento no enlace (8,6) e não irá conseguir continuar seu caminho que o levaria ao nodo 6, conforme representado na figura 4.7.

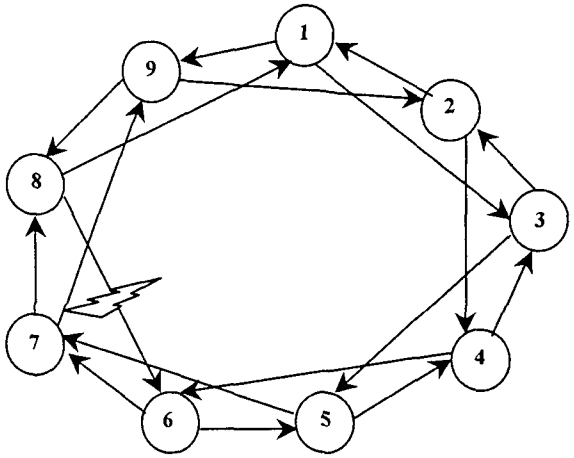


Figura 4.7: Falha no enlace (8,6).

Neste momento a topologia do grafo também é recalculada a fim de transformar o grafo após o evento para um grafo que seja *euleriano*, e assim o Agente Chinês pode continuar seguindo seu caminho, como ilustra a figura 4.8.

O nodo 8 se prepara para iniciar o processo de disseminação de mensagens, de acordo com o novo evento detectado, formando então a mensagem de disseminação. A mensagem contém o identificador do nodo testador, que é igual a 8, o identificador do nodo da outra ponta do enlace que sofreu o evento, que é 6, e o contador de eventos é incrementado de uma unidade. O grafo após a transformação irá conter uma aresta virtual entre os nodos 8 e 7, e 6 e 7, conforme ilustra a figura 4.8.

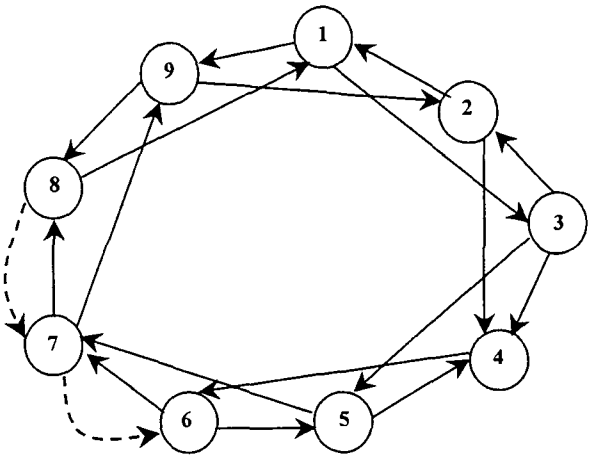


Figura 4.8: Disseminação do evento.

O nodo 8 inicia a disseminação e posteriormente segue o seguinte caminho: 8→7→9→2→1→9→8→1→3→4→6→7→6→5, e assim, quando a mensagem chegar ao nodo 5 o diagnóstico estará completo.

Em resumo os resultados são apresentados na tabela 4.2 abaixo:

Total de mensagens	**Mensagens redundantes**	**Latência (caso atual)**	**Latência (melhor caso)**	**Latência (pior caso)**
13	5	13	8	35

Tabela 4.2: Resultados da simulação para o grafo $D_{1,2}$ com 9 nodos.

Neste exemplo o número total de mensagens na rede foi 13, sendo 5 mensagens redundantes. O tempo total para completar o diagnóstico foi de 13 unidades de tempo.

4.3 Hipercubo (16 nodos)

A topologia hipercubo com 16 nodos é apresentada na figura 4.9. O algoritmo inicia sua execução em t=0, e o enlace entre o nodo 5 e o nodo 7 torna-se *falho* em um determinado instante de tempo. O Agente Chinês inicia seu caminho a partir do nodo 1, e segue percorrendo os seguintes nodos: 1 → 2 → 6 → 8 → 4 → 2 → 10 → 14 → 16 → 15 →7 → 3 → 1 → 5 → 7 → 8 → 16 → 12 → 4 → 3 → 11 → 12 → 10 → 9 → 11 → 15 → 13 → 5 → 6 → 14 → 13 → 9 e retornando ao nodo 1, dando início ao processo novamente até que seja detectado um evento. A figura 4.9 representa o caminho que o Agente Chinês percorre a partir do nodo 1.

Em determinado instante de tempo t, o Agente Chinês, que está no nodo 5, irá detectar um evento no enlace (5,7) e não irá conseguir continuar seu caminho que o levaria ao nodo 7, conforme representado na figura 4.9.

Neste momento a topologia do grafo também é recalculada a fim de transformar o grafo após o evento para um grafo que seja *euleriano*, e assim o Agente Chinês pode continuar seguindo seu caminho, como ilustra a figura 4.10.

O nodo 5 se prepara para iniciar o processo de disseminação de mensagens, de acordo com o novo evento detectado, formando então a mensagem de disseminação. A mensagem contém o identificador do nodo testador, que é igual a 5, o identificador do nodo da outra ponta do enlace que sofreu o evento, que é 7, e o contador de eventos é incrementado de uma unidade. O grafo após a transformação irá conter uma aresta virtual entre os nodos 5 e 3, 3 e 1, e 1 e 5 conforme ilustra a figura 4.10.

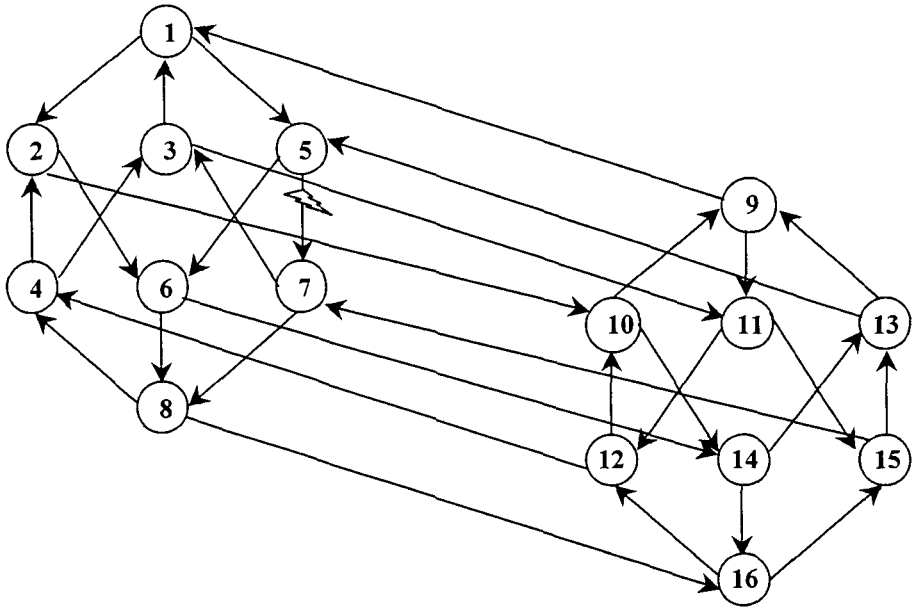


Figura 4.9: Hipercubo com 16 nodos.

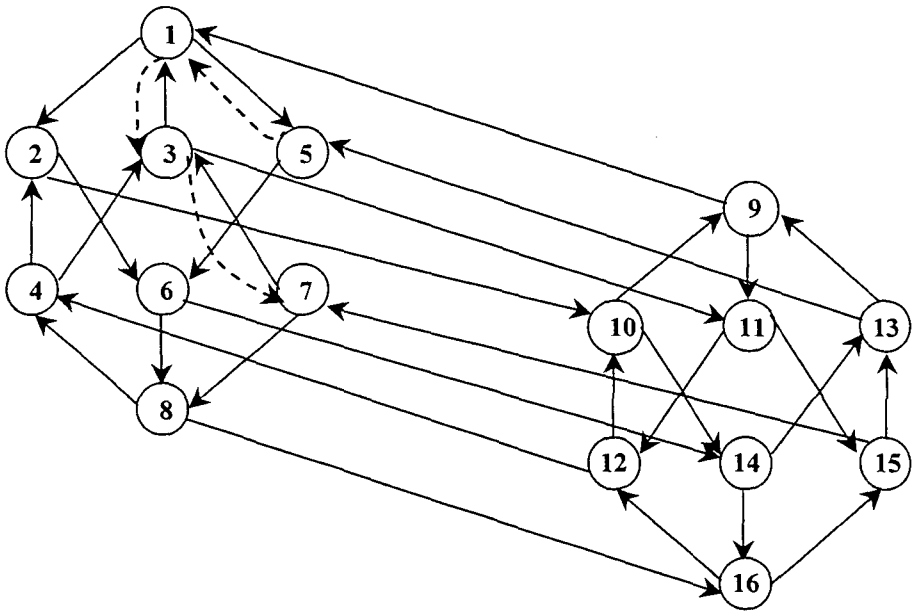


Figura 4.10: Topologia após o evento com o respectivo caminho do Agente Chinês.

O nodo 5 inicia a disseminação e posteriormente segue o seguinte caminho: 1→3→7→8→16→12→4→3→11→12→10→9→11→15→13→5→6→14→13→9→1→2, e assim, quando a mensagem chegar ao nodo 2 o diagnóstico estará completo.

Em resumo os resultados são apresentados na tabela 4.3 abaixo:

Total de mensagens	**Mensagens redundantes**	**Latência (caso atual)**	**Latência (melhor caso)**	**Latência (pior caso)**
22	7	22	15	63

Tabela 4.3: Resultados da simulação para o grafo hipercubo com 16 nodos.

Neste exemplo o número total de mensagens na rede foi 22, sendo 7 mensagens redundantes. O tempo total para completar o diagnóstico foi de 22 unidades de tempo.

4.4 Um Grafo Randômico

Este experimento foi realizado através da criação de um programa para gerar um grafo randômico. Padronizou-se o número de vértices igual a 50 e a probabilidade de haver uma aresta entre um par de vértices de 10%. Desta forma, para cada par de vértices possível um número aleatório entre 1 e 10 é gerado. Se o número gerado for igual a 1 existe uma aresta entre o par de vértices; caso contrário não há aresta entre eles.

A tabela a seguir contém a configuração do grafo simulado.

No vértice	**Incidem arestas para os seguintes nodos**
1	36 26 25 18 17 2
2	18 17 15 13 1
3	38 14 4
4	50 44 43 20 17 3
5	37 34 32 11
6	49 48 35 23
7	31 26
8	46 41 33 31 11
9	49 48 24
10	44 33 21 20 19 12
11	36 33 29 18 15 13 8 5
12	29 23 15 10
13	48 25 22 19 11 2
14	48 47 46 45 3
15	48 42 39 29 27 19 12 11 2
16	46 35 34 31 25
17	22 4 2 1
18	50 37 33 22 11 2 1
19	43 37 29 23 22 15 13 10
20	50 47 45 39 38 35 10 4
21	30 25 10
22	19 18 17 13
23	42 38 34 32 19 12 6
24	31 9
25	31 28 21 16 13 1
26	49 40 36 7 1
27	44 40 30 15
28	50 40 38 33 31 25
29	50 37 19 15 12 11
30	27 21
31	48 45 34 28 25 24 16 8 7
32	40 23 5

33	38 28 18 11 10 8
34	39 31 23 16 5
35	48 46 40 39 20 16 6
36	48 26 11 1
37	48 40 29 19 18 5
38	45 33 28 23 20 3
39	46 42 35 34 20 15
40	37 35 32 28 27 26
41	48 46 8
42	49 48 39 23 15
43	19 4
44	27 10 4
45	38 31 20 14
46	41 39 35 16 14 8
47	20 14
48	50 49 42 41 37 36 35 31 15 14 13 9 6
49	48 42 26 9 6
50	48 29 28 20 18 4

O algoritmo inicia sua execução em t=0, e o enlace entre o nodo 2 e o nodo 18 torna-se *falho* em um determinado instante de tempo. O Agente Chinês é iniciado a partir do nodo 1 e realiza seu caminho indefinidamente até que seja detectado um evento.

Em determinado instante de tempo t, o Agente Chinês que está no nodo 2 irá detectar um evento no enlace (2,18) e não irá conseguir continuar seu caminho que o levaria ao nodo 18. A topologia do grafo é recalculada a fim de transformar o grafo após o evento para um grafo que seja *euleriano*, e assim o Agente Chinês pode continuar seguindo seu caminho. O nodo 2 se prepara para iniciar o processo de disseminação de mensagens, de acordo com o novo evento detectado, formando então a mensagem de disseminação. A mensagem contém o identificador do nodo testador, que é igual a 2, o identificador do nodo da outra ponta do enlace que sofreu o evento, que é 18, e o contador de eventos é incrementado de uma unidade.

Em resumo os resultados são apresentados na tabela 4.4 abaixo:

Total de mensagens	Mensagens redundantes	Latência (caso atual)	Latência (melhor caso)	Latência (pior caso)
109	60	109	49	259

Tabela 4.4: Resultados da simulação para o grafo randômico com 50 nodos.

Neste exemplo o número total de mensagens na rede foi 109, sendo 60 mensagens redundantes. O tempo total para completar o diagnóstico foi de 109 unidades de tempo.

4.5 Rede Nacional de Pesquisa (RNP)

A topologia da RNP é apresentada na figura 4.11. A tabela a seguir contém a numeração dos nodos correspondentes a cada cidade brasileira, além dos Estados Unidos, de acordo com a simulação realizada.

Número do nodo	Cidade brasileira correspondente
1	Brasília
2	Goiânia
3	Campo Grande
4	Rio Branco
5	Porto Velho
6	Cuiabá
7	Manaus
8	Palmas
9	Boa Vista
10	Macapá
11	Belém
12	E.U.A.
13	Rio de Janeiro
14	São Paulo

Número do nodo	Cidade brasileira correspondente
15	Belo Horizonte
16	Vitória
17	Fortaleza
18	Salvador
19	Recife
20	Porto Alegre
21	Florianópolis
22	Curitiba
23	São Luís
24	Teresina
25	Maceió
26	Aracaju
27	Natal
28	Campina Grande

O algoritmo inicia sua execução em t=0, e o enlace entre o nodo 1 (Brasília) e o nodo 14 (São Paulo) torna-se *falho* em um determinado instante de tempo. O Agente Chinês percorre seu caminho entre os nodos indefinidamente ou até que ocorra um evento.

Em determinado instante de tempo t, o enlace entre o nodo 1 e o nodo 14 tornar-se *falho*. O nodo 14 ao tentar enviar o Agente Chinês para o nodo 1 detecta que o enlace (14,1) está *falho*. Neste momento o nodo 14 prepara-se para iniciar a disseminação da mensagem que será enviada aos nodos da rede. A mensagem de disseminação montada pelo nodo 14 é ilustrada na figura 4.12.

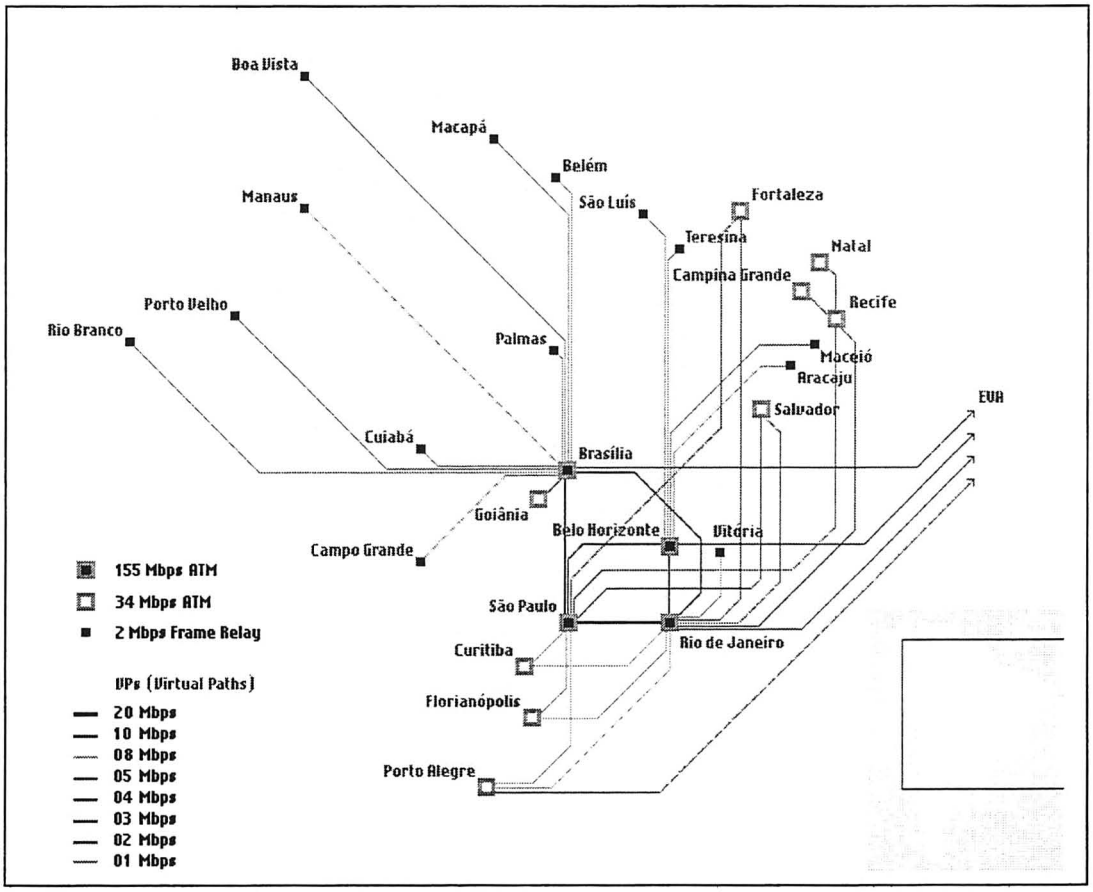


Figura 4.11: Configuração RNP [23].

Mensagem de disseminação montada pelo nodo 14 ao descobrir a falha no enlace (14,1).

Figura 4.12: Mensagens de disseminação montada pelo nodo 14.

O nodo 14 inicia a disseminação e posteriormente segue o seguinte caminho: 20→12→13→20→13→22→14→19→27→19→28→19→13→18→14→17→13→16→13→15→13→21→14→13→1→11→1→10→1→9→1→8→1→7→1→6→1→5→1→4→1→3→1→2→1→12→15→26→15→25→15→24→15→23, e assim, quando a mensagem chegar ao nodo 23 o diagnóstico estará completo.

Em resumo os resultados são apresentados na tabela 4.5 abaixo:

Total de mensagens	**Mensagens redundantes**	**Latência (caso atual)**	**Latência (melhor caso)**	**Latência (pior caso)**
54	27	54	27	75

Tabela 4.5: Resultados da simulação para o grafo da RNP com 28 nodos.

Neste exemplo o número total de mensagens na rede foi 54, sendo 27 mensagens redundantes. O tempo total para completar o diagnóstico foi de 54 unidades de tempo. O elevado número de mensagens redundantes se deve ao elevado número de vértices de grau um.

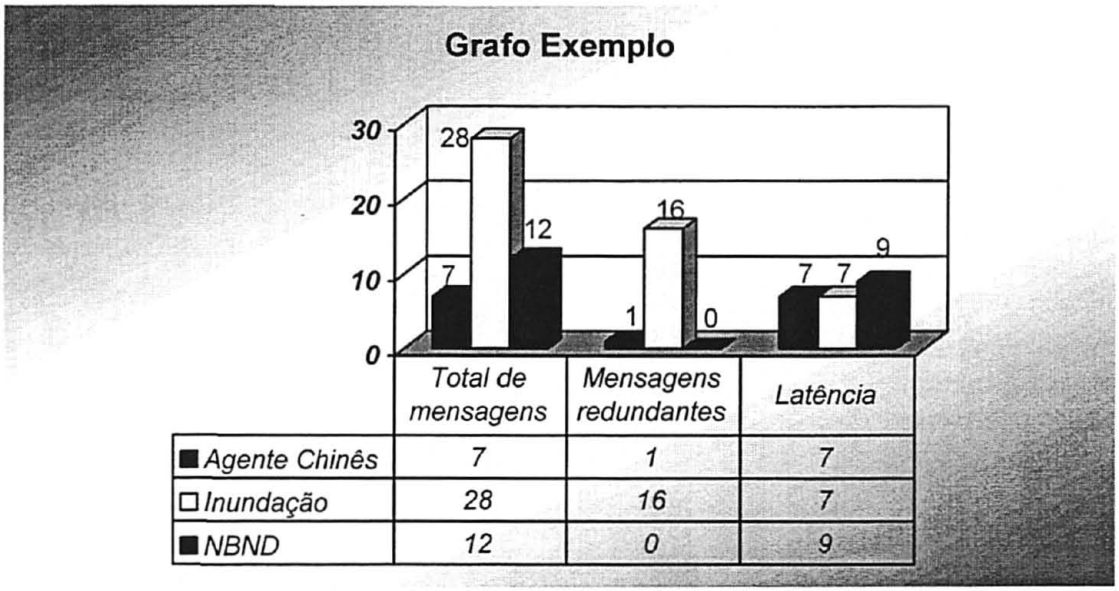
4.6 Comparações

Esta seção contém alguns resultados de simulação do algoritmo Agente Chinês em comparação ao algoritmo baseado em inundação [12] e ao algoritmo baseado no NBND

[3]. Os parâmetros considerados são o número de mensagens de disseminação, o número de mensagens redundantes e o tempo para realizar o diagnóstico.

4.6.1 Um Grafo Exemplo

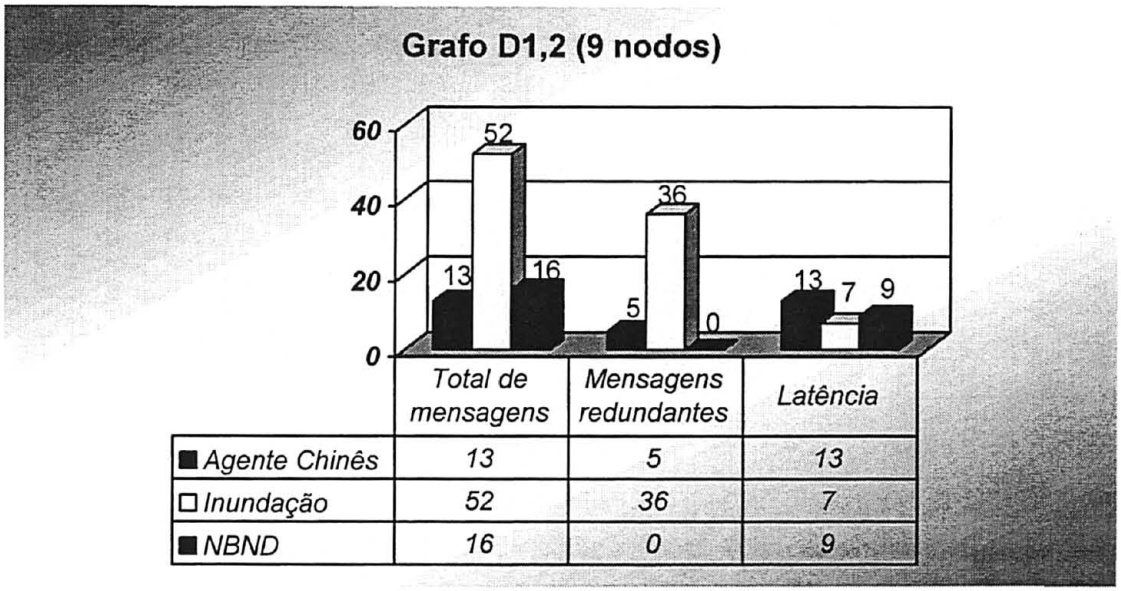
O gráfico a seguir contém resultados comparativos da execução dos algoritmos com a configuração do grafo exemplo descrito na seção 4.1.



Nesta topologia o Agente Chinês possui o menor número de mensagens de disseminação, o segundo menor número de mensagens redundantes e um tempo de diagnóstico igual ao algoritmo baseado em inundação, e menor que o algoritmo baseado no NBND. O total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é bem maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

4.6.2 Grafo $D_{1,2}$

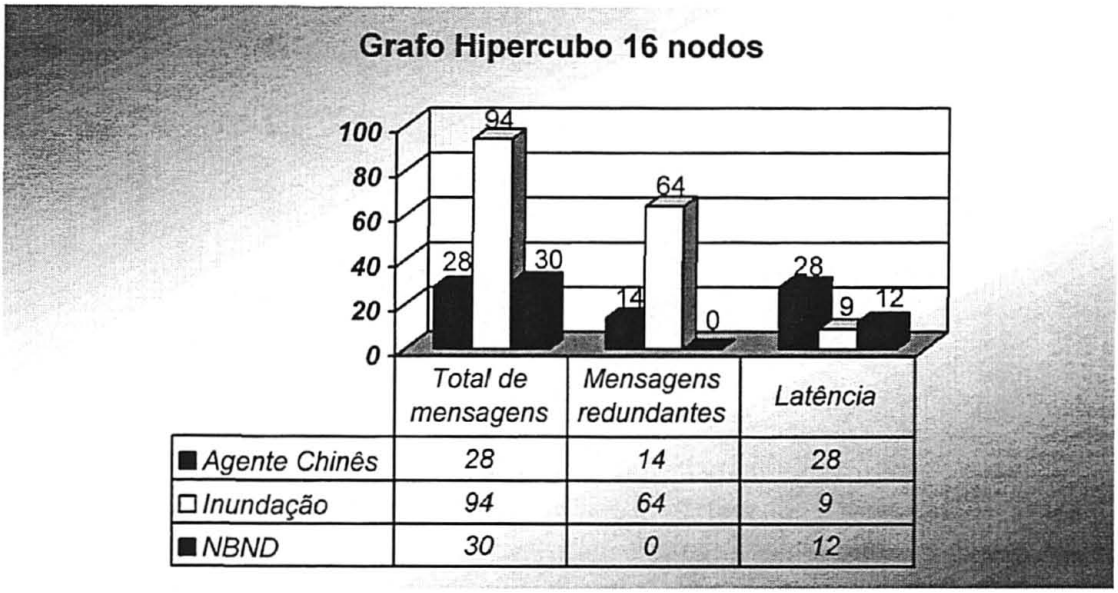
O gráfico a seguir contém resultados comparativos da execução dos algoritmos com a configuração do grafo $D_{1,2}$ com 9 vértices descrito na seção 4.2.



Nesta topologia o Agente Chinês possui o menor número de mensagens de disseminação, o segundo menor número de mensagens redundantes e apresenta um tempo de diagnóstico maior que os outros algoritmos. Novamente, o total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é bem maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

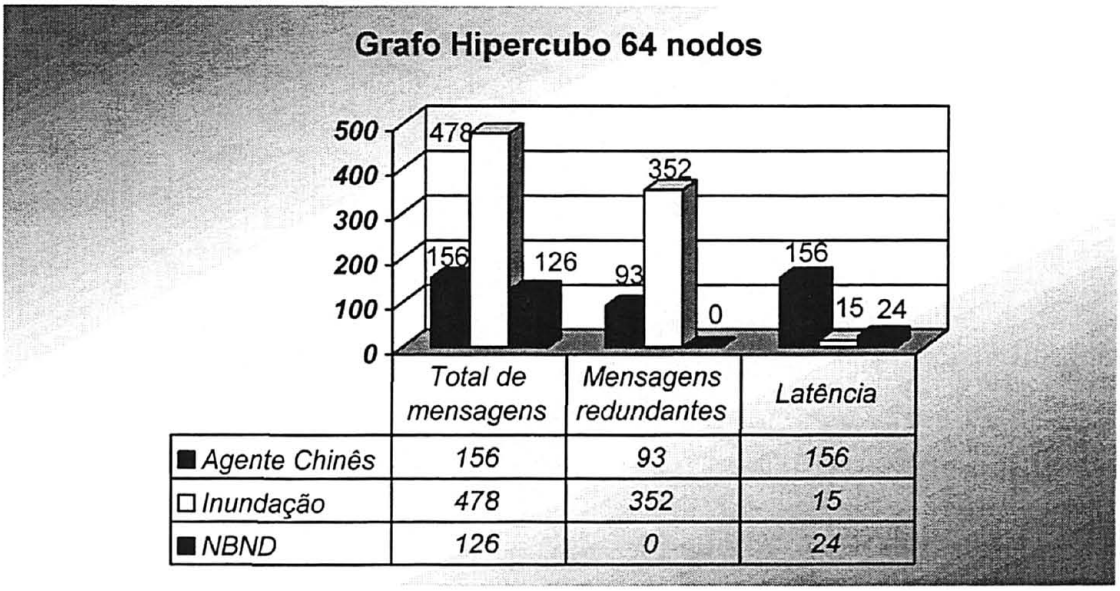
4.6.3 Hipercubo

O gráfico a seguir contém resultados comparativos da execução dos algoritmos com a configuração do grafo hipercubo com 16 vértices descrito na seção 4.3.



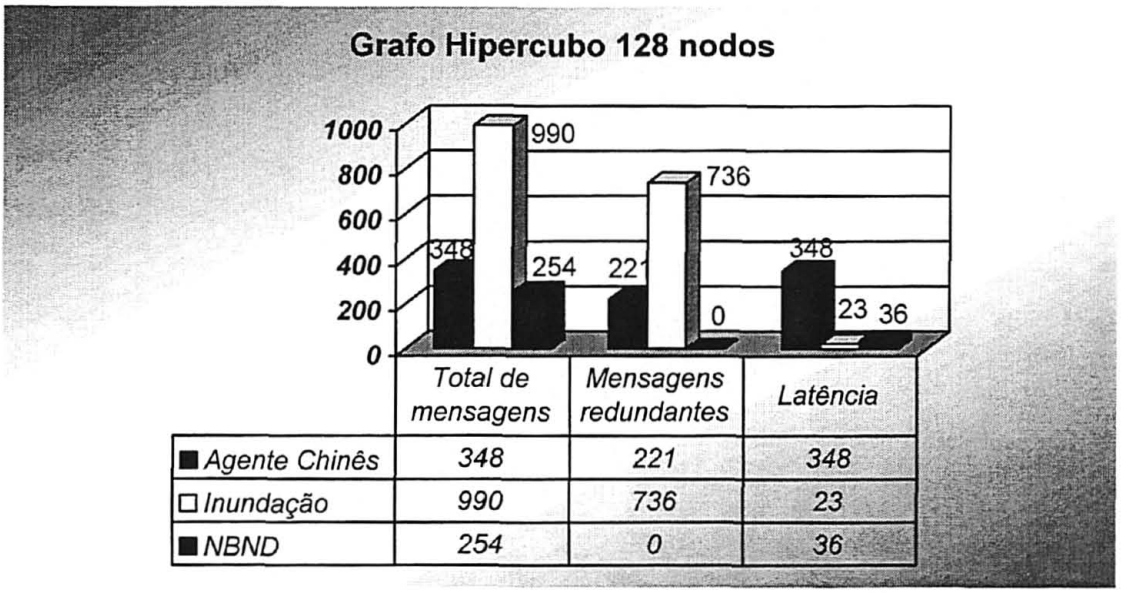
Mais uma vez, o Agente Chinês possui o menor número de mensagens de disseminação, o segundo menor número de mensagens redundantes e apresenta um tempo de diagnóstico maior que os outros algoritmos. Novamente, o total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é bem maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

Para um hipercubo com 64 vértices os resultados são:



Nesta topologia, o Agente Chinês possui o segundo menor número de mensagens de disseminação, o segundo menor número de mensagens redundantes e apresenta um tempo de diagnóstico bem maior que os outros algoritmos. Mais uma vez, o total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é bem maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

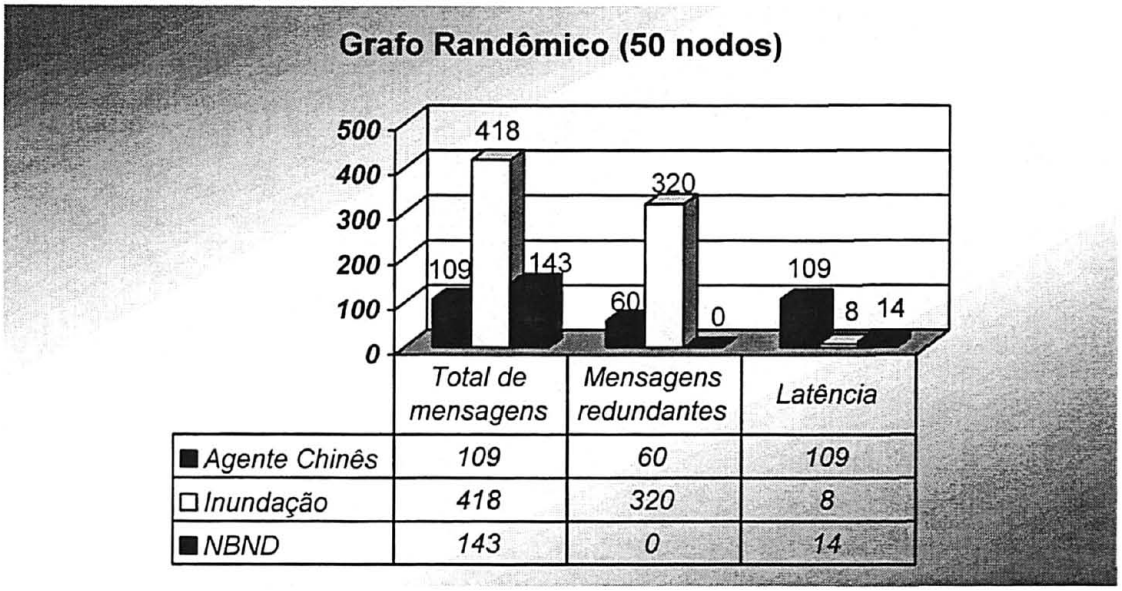
Para um hipercubo com 128 vértices os resultados são:



Novamente, o Agente Chinês possui o segundo menor número de mensagens de disseminação, o segundo menor número de mensagens redundantes e apresenta um tempo de diagnóstico bem maior que os outros algoritmos. Mais uma vez, o total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é quase 4 vezes maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

4.6.4 Um Grafo Randômico

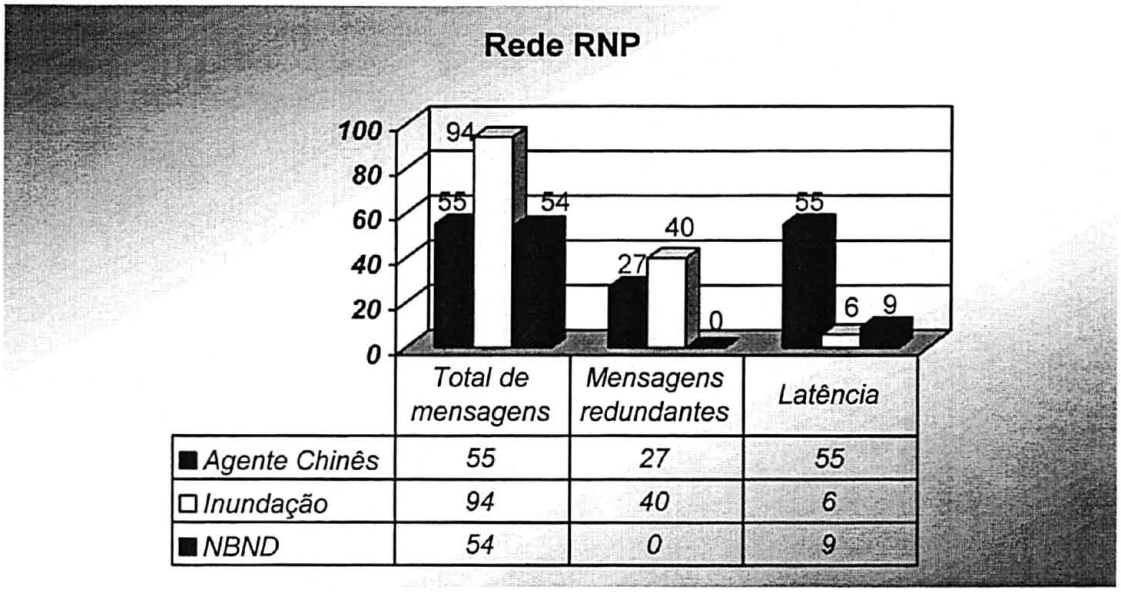
O gráfico a seguir contém resultados comparativos da execução dos algoritmos com a configuração do grafo randômico descrito na seção 4.4.



Nesta topologia, o Agente Chinês possui o menor número de mensagens de disseminação, o segundo menor número de mensagens redundantes e apresenta um tempo de diagnóstico bem maior que os outros algoritmos. O total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é bem maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

4.6.5 Rede Nacional de Pesquisa (RNP)

O gráfico a seguir contém resultados comparativos da execução dos algoritmos com a configuração da topologia RNP descrita na seção 4.5.



Nesta topologia, o Agente Chinês possui apenas uma mensagem a mais que o algoritmo baseado no NBND, possui o segundo menor número de mensagens redundantes e um tempo de diagnóstico maior que os outros algoritmos. O total de mensagens e o número de mensagens redundantes do algoritmo baseado em inundação é quase duas vezes maior que o total de mensagens e o número de mensagens redundantes no Agente Chinês.

4.7 Conclusão

Os resultados apresentados acima foram obtidos a partir de simulações realizadas comparando-se o algoritmo Agente Chinês, proposto neste trabalho, com os dados referentes à simulação de um algoritmo baseado em inundação de mensagens e com os dados de um algoritmo baseado no NBND.

Cada algoritmo usado nas simulações possui vantagens e desvantagens em relação a sua aplicação. O algoritmo apresentado neste trabalho dissemina suas mensagens de

maneira seqüencial, seguindo o caminho do Carteiro Chinês. Dessa maneira, irá existir apenas uma mensagem circulando pelo sistema em um determinado instante de tempo, e por outro lado, o tempo necessário para que todos os nodos recebam informação sobre um evento, e assim realizar o diagnóstico, é maior do que em um algoritmo que utiliza uma estratégia de disseminação paralela.

Capítulo 5

Conclusão

Neste trabalho foi apresentado um novo algoritmo, chamado Agente Chinês, baseado no algoritmo do Carteiro Chinês para o diagnóstico distribuído em redes de topologia arbitrária. Um agente de diagnóstico inicia seu percurso na rede percorrendo os nodos e testando os enlaces, e caso o agente detecte a ocorrência de um evento, o algoritmo recalcula a topologia da rede e reinicia o Agente Chinês, fazendo-o percorrer a nova topologia resultante do evento. O Agente Chinês inicia a disseminação da informação sobre o evento ocorrido avisando aos outros nodos sobre o estado da rede. Dessa maneira, quando todos os nodos do sistema receberem a informação sobre o evento, o diagnóstico estará completo.

Uma análise dos resultados mostra que a latência do algoritmo no melhor caso é de N-1 intervalos para que todos os nodos recebam informação sobre um evento e o diagnóstico seja completo, onde N é o número de nodos do sistema. No pior caso, a latência do algoritmo é de $2*L$-1 intervalos para que todos os nodos recebam informação sobre um evento e o diagnóstico seja completo, onde L é o número de enlaces do sistema original (antes da transformação para *euleriano*).

Foram realizadas simulações com vários tipos diferentes de topologia, dentre elas um grafo exemplo com 7 vértices, um grafo $D_{1,2}$ com 9 vértices, grafos hipercubos de 16, 64 e 128 vértices, além de um grafo randômico e a topologia da Rede Nacional de Pesquisa (RNP). Foram simuladas falhas de um enlace em cada grafo; e um dos nodos cujo enlace está conectado detecta o evento e inicia a disseminação das mensagens. Os resultados são comparados aos obtidos através da simulação de outros algoritmos de diagnóstico de redes de topologia arbitrária. Tais comparações utilizaram dados referentes à simulação de um algoritmo baseado em inundação de mensagens, e dados referentes à simulação de um algoritmo baseado no NBND.

Cada algoritmo usado nas simulações possui vantagens e desvantagens em relação a sua aplicação. O algoritmo Agente Chinês utiliza uma estratégia de disseminação seqüencial de suas mensagens, seguindo o caminho do Carteiro Chinês. Dessa maneira, irá existir apenas uma mensagem circulando pelo sistema em um determinado instante de tempo, porém, o tempo necessário para que todos os nodos recebam informação sobre um evento, e assim realizar o diagnóstico, é maior do que em um algoritmo que utiliza uma estratégia de disseminação paralela.

Trabalhos futuros incluem a modificação do algoritmo para permitir recuperações de enlaces *falhos* e nodos *falhos*. Além disso, deve-se também permitir falhas que particionem a rede, e a ocorrência de eventos dinâmicos, ou seja, a detecção e recuperação de diversos eventos simultâneos. Um estudo sobre o impacto de existirem diversos *Agentes Chineses* realizando o diagnóstico de um sistema ao mesmo tempo, e como tais agentes seriam criados, eliminados e como iriam trocar informações entre si é objetivo de trabalho futuro. A produção de uma ferramenta prática para o diagnóstico de conectividade de redes de longa distância é também objetivo de trabalho futuro.

Referências

[1] E.P. Duarte Jr., **"Um Algoritmo para Diagnóstico de Redes de Topologia Arbitrária"**, I Workshop de Tolerância a Falhas da SBC, Porto Alegre, 1998.

[2] S. Rangarajan, A. T. Dahbura, and E.A. Ziegler, **"A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies,"** *IEEE Transactions on Computers*, Vol. 44, pp. 312-333, 1995.

[3] E.P. Duarte Jr., F. Mansfield, T. Nanya, and S. Noguchi, **"Non-Broadcast Network Fault-Monitoring Based on System-Level Diagnosis,"** *Proc.IFIP/IEEE IM'97*, pp. 597-609, 1997.

[4] F. J. N. Gomes, B. F. Rezende, G. C. Barcellos, L. W. L. Pereira, E. F. Coutinho, G. A. de Castro, M. J. N. Gomes, A. W. C. Palhano, **"Xnês: Um Ambiente Visual para Geração de Soluções Ótimas de Instâncias do Problema do Carteiro Chinês,"** http://gemeos.uece.br/~negreiro/, 1999.

[5] L. Euler, **"Solutio problematis ad geometriam situs pertinentis,"** *Comentarii Academiae Scientiarum Imperialis Petropolitanae* 8, pp.128-140, 1736.

[6] R. J. Wilson, J. J. Watkins, **"Graphs- An Introductory Approach,"** John Wiley & Sons, 1990.

[7] M. H. MacDougall, **"Simulating Computer Systems: Techniques and Tools,"** The MIT Press, Cambridge, MA, 1987.

[8] A. Bagchi, and S.L. Hakimi, **"An Optimal Algorithm for Distributed System-Level Diagnosis,"** *Proc.* 21st *Fault Tolerant Computing Symp.*, June, 1991.

[9] R. Bianchini, M. Stahl, and R. Buskens, **"The Adapt2 on-line diagnosis algorithm for general topology networks,"** in *Proc. Globecom*, pp.610-614, 1992

[10] F. Preparata, G. Metze, and R.T. Chien, **"On the Connection Assignment Problem of Diagnosable Systems,"** *IEEE Transactions on Eletronic Computers*, Vol. 16, pp. 848-854, 1968

[11] S. L. Hakimi, and A. T. Amin, **"Characterization of Connection Assignments of Diagnosable Systems,"** *IEEE Transactions on Computers*, Vol. 23, pp. 86-88, 1974.

[12] E.P. Duarte Jr.and G.O. Mattos, **"Diagnóstico em Redes de Topologia Arbitrária: Um Algoritmo Baseado em Inundação de Mensagens"**, *in portuguese* In *Proceedings of the 2nd SBC Test and Fault Tolerance Workshop, SBC-WTF'2*, pp. 82-87, Curitiba, Brazil, 2000.

[13] J.I. Siqueira, and E.P. Duarte Jr. **"Uma Ferramenta para Diagnostico Distribuído de Redes de Alta Velocidade",** In *Proceedings of the 2nd SBC Brazilian National Research Network (RNP) Workshop, SBC-WRNP'2*, pp. 106-116, BH, Brasil, 2000.

[14] M. Stahl, R. Buskens, and R. P. Bianchini, **"Simulation of the Adapt On-Line Diagnosis Algorithm for General Topology Networks,"** *IEEE Transactions on Computers*, Vol. 44, pp. 180-187, 1992.

[15] M. Stahl, R. Buskens, and R. P. Bianchini, **"On-Line Diagnosis in General Topology Networks,"** *Workshop on Fault Tolerant Parallel and Distributed Systems,* July 1992.

[16] J. A. Bondy and U. S. R. Murty, **"Graph Theory and Applications,"** *Elsevier North Holland*, Inc., New York, 1976.

[17] E.P. Duarte Jr., L.C.P. Albini, and A. Brawerman **"An Algorithm for Distributed Diagnosis of Dynamic Fault and Repair Events"** In *Proceedings of the 7th IEEE International Conference on Parallel and Distributed Systems, IEEE/ICPADS'00*, pp. 299-306, Iwate, Japan, 2000.

[18] Dominic Battré, 1998; http://www.geocities.com/SiliconValley/Peaks/1667/

[19] Dr. Chris Mawata, 1997; http://www.utc.edu/~cpmawata/petersen/

[20] Chris Caldwell, 1995; http://www.utm.edu/departments/math/graph/glossary.html

[21] R. Sedgewick, **"Algorithms,"** Addison-Wesley, 1988.

[22] Kenneth A. Ross, Charles R. B. Wright, **"Discrete Mathematics,"** Prentice Hall International.

[23] Rede Nacional de Pesquisa, 2001; http://www.rnp.br

[24] Dr. Larry Bowen, 1999; http://www.sa.ua.edu/ctl/math103/

[25] E.P. Duarte Jr., and J.M.A.P. Cestari **"O Agente Chinês para Diagnóstico de Redes de Topologia Arbitrária"** *in portuguese* In *Proceedings of the 2nd SBC Test and Fault Tolerance Workshop, SBC-WTF'2*, pp. 88-93, Curitiba, Brazil, 2000.

[26] J. Edmonds, and E. L. Johnson, **"Matching, Euler Tours and the Chinese Postman,"** *Mathematical Programming*, 5:88-124, 1973.

[27] H. A. Eiselt, M. Gendreau, and G. Laporte, **"Arc-Routing Problems, Part 1: The Chinese Postman Problem,"** *Operations Research*, 43:231-242, 1995.

Apêndice A

Neste apêndice é apresentado o código fonte do algoritmo do Agente Chinês descrito neste trabalho. O algoritmo do Carteiro Chinês [18] é utilizado sobre o grafo original, a fim de transformá-lo em *euleriano* caso seja necessário.

```
#include "smpl.c"
#include "bmeans.c"
#include "rand.c"
#include <stdio.h>
#include "smpl.h"
#include "fila.c"

#define maxV 100      /* numero maximo de vertices */
#define true 1
#define false 0
#define fault 2
#define recebe_msg 1 /* unico evento e' receber msg*/


struct node {
  int v;
  int timestamp;      //contador de eventos
  struct node *next;
};
struct node *t, *z;

struct adjacencia {
  int grau;
  struct node *prim;
};
struct adjacencia adj[maxV], adj_org[maxV]; /* o grafo original, com
                                               timestamps atualizados
                                               e' mantido em adj_org[] */

typedef struct
{
  int id,
    change,
    t_round,
    cont_tests,
    cluster;
  struct adjacencia adj[maxV];
}tnodo;
tnodo nodo[maxV];
```

```
int j, x, y, aux=0 ,V,E;
int val[maxV], caminho[maxV]; //caminho guarda os vertices visitados
int vetor_original[2*maxV],fim=1;
int graus[maxV], adicionadas=0;
int primeiro_e_impar=false;
int ja_passou[maxV];
int cont,c;
int lfalho[2]={0,0} ; /*define qual e' o link falho*/


static int prox,prox_aux, sabem=0, new_event=0, prev_round=0;

int adjlist(void);
int pode_tirar(int origem, int destino);
int listbfs (void);
void retira_aresta(int org, int dest);
void procura(void);
int impar(int valor);
int precisa(void);
void adiciona(int destino);
void imprime_adj(struct adjacencia lista[], int codigo);
void copia_listas(struct adjacencia adj[], struct adjacencia adj_org[], int tam);
int tsmp_impar(int pos1, int pos2);

////////////////////////////////////////////////////////////////////
void copia_listas(struct adjacencia origem[], struct adjacencia destino[], int
tam)
{

  int cont=1;
  struct node *t_origem, *t_destino;

  for (cont=1; cont<=vetor_original[0]; cont++)
    {
      destino[cont].prim=z;
    }


  for (cont=1; cont<=vetor_original[0]; cont++)
    {
      t_origem=origem[cont].prim;
      while (t_origem!=z)
       {
         if ((impar(t_origem->timestamp)==0)){
           t_destino = (struct node *) malloc (sizeof *t_destino);
           t_destino->next=destino[cont].prim;
           destino[cont].prim=t_destino;
           t_origem=t_origem->next;
           t_destino=t_destino->next;
         }
         else
           {
             t_origem=t_origem->next;
           }
       }
    }


  for (cont=1; cont<=vetor_original[0]; cont++)
   {
     t_origem=origem[cont].prim;
     t_destino=destino[cont].prim;
```

```
      while (t_origem!=z)
        {
        if (impar(t_origem->timestamp)==0)
          {
            t_destino->v=t_origem->v;
            t_destino->timestamp=t_origem->timestamp;
            t_origem=t_origem->next;
            t_destino=t_destino->next;
          }
        else
          {
            t_origem=t_origem->next;
          }
        }
    }
}
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
int lista_vazia(struct adjacencia lista[], int tam)
{
       int cont;

       for (cont=1;cont<=tam;cont++)
              if (lista[cont].prim!=z)
                  return 0; //o nodo possui links

       return 1; //nenhum nodo possui links
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
visit (int k)
{
  int id=0;
  struct node *t;

  t = (struct node *) malloc(sizeof *t);
  put (k);
  while (!queueempty())
    {
      k = get(); val[k] = ++id;
      for (t = adj[k].prim; t != z; t = t->next)
       if (val[t->v] == 0)
         { put (t->v); val[t->v] = -1; }
    }
  return;
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
listbfs ()
{
  int k;
  queueinit();

  for (k=1; k<=vetor_original[0]; k++) val[k]=0;
  for (k=1; k<=vetor_original[0]; k++)
    if (val[k] == 0) visit(k);
  return;
}
////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////
adjlist()
{
  z=(struct node *) malloc (sizeof *z);
  z->next=z;
  z->v=-1;
  for (j=1;j<=vetor_original[0];j++)
    {
      adj[j].prim=z;
      adj[j].grau=0;
      adj_org[j].prim=z;
      adj_org[j].grau=0;
    }
  for (j=1;j<=vetor_original[1];j++)
    {
      x=vetor_original[2*j];
      y=vetor_original[2*j+1];
      t= (struct node *) malloc (sizeof *t);
      t->v =x;
      //inicializa o timestamp verificando se os nodos sao falhos ou nao
      t->timestamp=0;
      t->next=adj[y].prim;
      adj[y].prim=t;
      adj[y].grau+=1;
      graus[y]=adj[y].grau;
      t= (struct node *) malloc (sizeof *t);
      t->v =y;
      t->timestamp=0; //inicializa o timestamp
      t->next=adj[x].prim;
      adj[x].prim=t;
      adj[x].grau+=1;
      graus[x]=adj[x].grau;
    }
  return;
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void impr_vetor()
{
  int i;
  printf("\n");
  for (i=0;i<=(2*vetor_original[0]);i++)
    printf("\nVetor[%d]=%d",i,vetor_original[i]);
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void impr_graus()
{
  int i;
  printf("\n");
  for (i=1;i<=vetor_original[0];i++)
    printf("--G[%d]=%d",i,graus[i]);
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void  retira_aresta(int org, int dest)
{
  struct node *antes,*atual,*antes1, *atual1;
  int i;
```

```
    antes = adj[dest].prim;
    atual = adj[dest].prim;

   if ((pode_tirar(org,dest))==0)
     {
       antes = adj[dest].prim;
       atual = adj[dest].prim;
       antes1=adj[org].prim;
       atual1=adj[org].prim;

       if ((atual1->v)==dest)
        adj[org].prim=adj[org].prim->next; //retira a aresta da origem->destino
       else
        while(atual1->v!=dest)
          {
            atual1=atual1->next;
            //printf("\nATUAL1=%d",atual1->v);
            if ((atual1->v)==dest)
              antes1->next=atual1->next;
            else
              antes1=antes1->next;
          }

       if ((atual->v)==org) //caso seja o primeiro
        adj[dest].prim=adj[dest].prim->next;
       else
        while (atual->v!=org)
          {
            atual=atual->next;
            if ((atual->v)==org)
              antes->next=atual->next;
            else
              antes=antes->next;
          }


       graus[org]--;
       graus[dest]--;
       prox=dest;
     }
   else
     retira_aresta(org,adj[org].prim->next->v);
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
int pode_tirar(int origem, int destino)
{
  struct node *apt;
  apt = adj[origem].prim;
  if( (((impar(graus[origem])==1 && impar(graus[destino])==1)) && (apt->next==z))
||
        ((impar(graus[origem])==0) || (impar(graus[destino])==0)))
    return 0;
  else
    return 1;
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
void le(FILE *arq)
{
  int ch;       /* Caracter lido do disco */
```

```
  int temp;     /* Contem o numero enquanto ele vai sendo montado */
  int cont=0;   /* Contem o numero de elementos que foi lido do arquivo */
  ch = fgetc (arq);
  while (ch != EOF) {
    temp = 0;
    while (ch != 32 && ch != EOF && ch!=13 && ch!=10)
      //caracteres decimais para funcionar no Linux
      {
      temp = (ch - 48) + (temp*10);
      ch = fgetc (arq);
      }
    if (vetor_original[0] > maxV) {
      printf ("\n O arquivo contem mais vertices que o definido");
      exit(0);
    }
    else {
      vetor_original[cont] = temp;
      cont ++;
    }
    ch = fgetc (arq);
    if (ch==10)  //para funcionar no linux de casa, nao precisa no AIX
      ch=fgetc(arq);
  }
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
int impar(int valor)
{
  int r;

  r= valor%2;
  return r;  //se r=1 e' impar, se r=0 e' par
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
void procura()
{
  int i;

  for(i=vetor_original[0]; i>=1; i--)
    {
      if (i==1) //caso seja o primeiro vertice
       primeiro_e_impar=true;

      if (impar(graus[i])==1 && (graus[i]!=0) && (adj[i].prim->next==z)
        && (aparece_uma(i)==1))
       adiciona(i+1);
      else
       if ((impar(graus[i])==1) && (graus[i]!=0))
        adiciona(i);
    }
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
int aparece_uma(int valor)
{
      int i, aparece=0;
      for (i=2;i<=(2*vetor_original[0])+1; i++)
      {
            if (vetor_original[i]==valor)
```

```
                aparece++;
            i++;
            if (i>(2*vetor_original[0])+1)
                break;
      }
      return aparece;
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
int precisa()
{
  int i;
  for (i=vetor_original[0]; i>=1; i--)
    if (impar(graus[i])==1)
      return 1;
  return -1;
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
void adiciona(int destino)
{
 int c;
 struct node *t, *taux;


  for (c=(2*vetor_original[1]+1); c>=2; c--)
    if (vetor_original[c]==destino)
      {
      t = (struct node *) malloc (sizeof *t);
      taux = (struct node *) malloc (sizeof *taux);
      t=adj[vetor_original[c-1]].prim;
      while(t->next!=z)
        t = t->next;
      taux->v=destino;
      taux->next=t->next;
      t->next=taux;
      graus[vetor_original[c]]+=1;
      graus[vetor_original[c-1]]+=1;


      t = (struct node *) malloc (sizeof *t);
      taux = (struct node *) malloc (sizeof *taux);
      t=adj[vetor_original[c]].prim;
      while(t->next!=z)
        t = t->next;
      taux->v=vetor_original[c-1];
      taux->next=t->next;
      t->next=taux;
      adicionadas+=1;
      c--;
      }
    else
      c--;
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
void imprime_adj(struct adjacencia lista[], int codigo)
{
  int cont=1;
  struct node *ptr;
```

```
  ptr=lista[cont].prim;
  for(cont=1; cont<=vetor_original[0]; cont++)
    {
      if (codigo==1)
       printf("\nadj[%d]",cont);
      else
       printf("\nadj_org[%d]",cont);
      while (ptr!=z){
       if (impar(ptr->timestamp)==0) //so imprime se o timestamp for par
         {                                                    //nao ha´ falha no link
           printf("->%d(%d)",ptr->v,ptr->timestamp);
           ptr=ptr->next;
         }
       else
         ptr=ptr->next;
       }
      ptr=lista[cont+1].prim;
    }
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void atualiza_graus(int vertice1, int vertice2)
{
  graus[vertice1]--;
  graus[vertice2]--;
}
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
void atualiza_timestamp(struct adjacencia lista[])
{
 int cont=1;

 struct node *ptr;
 ptr=lista[cont].prim;

 for (cont=1; cont<=vetor_original[0]; cont++)
   {
     while (ptr!=z)
       {
       if ( ((ptr->v==lfalho[0]) || (ptr->v==lfalho[1]))
            && ((cont==lfalho[0]) || (cont==lfalho[1])))
         {
           ptr->timestamp++;
           //ptr->grau--; no futuro
           graus[ptr->v]--;
         }
       ptr=ptr->next;
       }
     ptr=lista[cont+1].prim;
   }
}
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
int tsmp_impar(int pos1, int pos2)
{

 int cont=1,num_impar=0;
 struct node *ptr;
 ptr=adj_org[cont].prim;
```

```
  for (cont=1; cont<=vetor_original[0]; cont++)
    {
      while (ptr!=z)
      {
         if ((pos1==cont) || (pos2==cont))
           if ((pos1==ptr->v) || (pos2==ptr->v))
             if (impar(ptr->timestamp)==1)
               num_impar++;

      ptr=ptr->next;
      }
      ptr=adj_org[cont+1].prim;
    }
  if (impar(num_impar)==0)
    return 0;
  else
    return 1;

}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
void atualiza_vetor(int org, int dest)
{
  int cont,cont1;
  for (cont=2; cont<=(2*vetor_original[1]+2); cont++)
    {
      if  ((org==vetor_original[cont]) && (dest==vetor_original[cont+1]))
       for (cont1=cont; cont1<=(2*vetor_original[1]+2); cont1++)
         vetor_original[cont1]=vetor_original[cont1+2];
    }
  /* abaixo quando perde apenas um vertice. Fazer uma geral*/
   if ((adj_org[dest].prim->next==z) || (adj_org[org].prim->next==z))
     vetor_original[0]--;
  vetor_original[1]--;
  vetor_original[(2*vetor_original[1])+2]=org;
}
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
void zera_lista(struct adjacencia lista[])
{
  int i;
  for (i=1;i<=vetor_original[0];i++)
    lista[i].prim=z;
}
////////////////////////////////////////////////////////////////////

// PROGRAMA PRINCIPAL ////////////////////////////////////
////////////////////////////////////////////////////////////////////
int main(int argc,char *argv[])
{
  static int N,
    event,
    i,j,k=0,
    token;
  FILE *arq;     /* Ponteiro para o arquivo que vai ser lido */
  static char fa_name[5];
  int x,num_falhos=0;
  struct node *t, *taux;
  int primeira_vez=true;
```

```
if (argc != 2) {
  printf ("Modo de usar : trab5 <nome do arquivo> \n");
  exit (-1);
}
if ((arq =fopen (argv[1],"r+")) == NULL) {
  printf ("Erro ao abrir o arquivo! Abortando. \n");
  exit(-1);
}

le (arq);
V=vetor_original[0];
E=vetor_original[1];
N=V;

adjlist();
copia_listas(adj,adj_org,maxV);
   //copia a adj para adj_org, mantendo a lista original intacta
imprime_adj(adj,1);
impr_graus();
j=vetor_original[2*vetor_original[1]+2];
cont=1;
prox=j;

if (precisa()==1)
  {
    printf("\nO grafo nao eh Euleriano, e necessario transforma-lo...");
    procura();
  }
else
  {
    printf("\nO grafo e' Euleriano, use o outro programa...\n");
    exit(0);
  }

if (primeiro_e_impar==true)
  adicionadas--;

smpl(0,"Agente Chines Distribuido Com Um Evento - Jose Marcelo");

for(i=1;i<=V;i++)
  {
    nodo[i].cluster=1;
    nodo[i].change=0;
    nodo[i].cont_tests=0;
    memset(fa_name,'\0',5);
    sprintf(fa_name,"%d",i);
    nodo[i].id=facility(fa_name,1);
    for(c=1;c<=V;c++)
     {
       nodo[i].adj[c].grau=adj[c].grau;
       t= (struct node *) malloc (sizeof *t);
       taux=(struct node *) malloc (sizeof *taux);
       nodo[i].adj[c].prim=t;
       t->v =adj[c].prim->v;
       t->next=adj[c].prim->next;
       while (t->next!=z)
         {
           taux=t;
           t= (struct node *) malloc (sizeof *t);
           t->v =taux->next->v;
           t->next=taux->next->next;
           taux->next=t;
         }
```

```
        }
    }

  schedule(recebe_msg,30.0,j);   /* schedule no nodo inicial */

  printf("\nNo inicial: %d\n",j);
  while (time()<=1500/*event<2*/)
    {
      cause(&event,&token);
      switch(event)
       {
       case recebe_msg:
         {
           printf("\nProximo=%d Cont=%d Tempo=%4.1f",prox,cont,time());
           prox_aux=adj[prox].prim->v;
              if ((lista_vazia(adj,vetor_original[0])==1) || (prox_aux==-1))
              {
                     copia_listas(&adj_org,&adj,maxV);
                     zera_lista(&adj);
                     adjlist();
                     prox_aux=adj[prox].prim->v;

              if (precisa()==1) //precisa de arestar virtual
                {
                  printf("\nO grafo nao eh Euleriano, e necessario transforma-
lo...");
                  procura();

                }
              }


           if ((lfalho[0]==prox) && (lfalho[1]==prox_aux) ||
              (lfalho[0]==prox_aux && lfalho[1]==prox))
              {
              printf("\nO link %d-%d esta falho\n",prox,prox_aux);
              if (primeira_vez==true) //primeira atualizacao do timestamp
                {
                  if ((adj[prox].prim->next==z))
                    {
                           printf("\nNao ha' para onde ir. Saindo...\n");
                           break;
                           exit(1);
                    }
                  printf("\nAtualizando o timestamp...\n");
                  atualiza_timestamp(adj_org);
                  primeira_vez=false;
                }
              else
                if (tsmp_impar(prox,prox_aux)==1)
                 {
                    printf("\nAinda nao pode cair aqui...\n");
                    printf("\nAtualizando o timestamp...\n");
                    atualiza_timestamp(adj_org);

                 }


              copia_listas(adj_org,adj,maxV);
              atualiza_vetor(prox,prox_aux);
              zera_lista(adj);
              adjlist();
```

```
            if (precisa()==1) //precisa de arestar virtual
               {
                 printf("\nO grafo nao eh Euleriano, e necessario transforma-
lo...");
                 procura();

               }

            }
          else
            {
            retira_aresta(prox,prox_aux);
            ja_passou[cont]=prox;
            cont+=1;
            }


            if(time()<=1500)
                  schedule(recebe_msg,30.0,prox);
            else
                  {
                        event=2;
                  }
          break;
        }
      }
    }


  fclose(arq);
  printf("\nAcabou com sucesso\n");

  return;
}
```