Cálculo de Confiabilidade de Redes *Mesh* Utilizando o Método da Tabela *Booleana*

Luís Filipe Meyer (PPGEP/UFRGS)
Flávio Sanson Fogliatto (PPGEP/UFRGS)

Resumo
As redes mesh constituem-se num paradigma para a próxima geração de redes industriais sem fio. Apesar dessas redes serem mais baratas e de prática instalação, ainda são pouco utilizadas. A causa reside na falta confiança para transmissão dos dados entre origem e destino. As redes mesh são consideradas sistemas complexos no estudo de sua confiabilidade. Seus valores não são obtidos de forma rápida e simples e para seu cálculo existem diversos métodos. Um dos métodos é o da tabela booleana. Este artigo apresenta um software para a obtensão da confiabilidade de redes mesh utilizando o método da tabela booleana.
Palavras chave: Redes mesh. Confiabilidade. Sistemas complexos. Tabela booleana.

1 Introdução

Existe uma demanda crescente por informações em ambientes industriais, gerando a adição de novos dispositivos, cada vez mais rápidos e mais inteligentes. Essa demanda resulta na elevação dos custos de instalação e de implantação de novos aparelhos em uma planta industrial (LAFRAIA 2001).

Como exemplo dessa situação de aumento de custos, pode-se citar o cabeamento das instalações industriais para a realização de medições e controle (POOR; HODGES 2004). Como alternativa de menor custo, as redes *wireless* têm sido utilizadas em um número crescente de aplicações (MORING 2004).

Entretanto, a utilização de tecnologias *wireless* em ambientes industriais vem enfrentando dificuldades, grande parte motivada pela falta de confiabilidade na transmissão e na recepção dos dados (MOORE 2004). Situações indesejáveis podem acarretar problemas sérios, tanto do ponto de vista financeiro, como do ponto de vista humano (LAFRAIA 2001). Este fato torna-se ainda mais determinante ao considerar-se a indústria de Petróleo e Gás Natural, uma vez que uma falha de comunicação pode gerar acidentes de conseqüências catastróficas (CHÉRIT *et al.*, 2011).

Para incrementar o uso de tecnologias *wireless* em ambientes industriais, as redes devem ser concebidas de acordo com o ambiente no qual serão instaladas, atendendo requisitos específicos de cada situação. Ao invés do foco estar na velocidade de transmissão dos dados, as redes industriais devem ter seu foco na confiabilidade da entrega dos dados. Na medida que as redes tornam-se cada vez mais complexas, a confiabilidade torna-se peça chave para sua utilização (TANG 1999). Nesse contexto, as redes *mesh* apresentam vantagens sobre as demais topologias de rede (POOR; HODGES 2004). As redes *mesh* constituem-se em paradigma para as próximas gerações de redes sem fio, tendo como principais vantagens as capacidades de auto organização e auto configuração (HOUSSAIN; LEUNG 2008).

Contudo, para a determinação da confiabilidade, são necessários cálculos trabalhosos, pelo fato do sistema que utiliza esta topologia ser do tipo complexo. Diversos métodos podem ser utilizados para o cálculo desses sistemas, entre eles, pode-se citar o método da tabela booleana, que se destaca pela sua simplicidade. Entretanto, na literatura não encontram-se muitas aplicações práticas reportadas desse método de cálculo de confiabilidade. Apesar de ser de baixa complexidade do ponto de vista matemático, sua implementação pode tornar-se bastante trabalhosa, na medida em que o número de componentes do sistema aumenta (FOGLIATTO; RIBEIRO 2009).

O presente artigo traz como principal contribuição um algoritmo para o cálculo de confiabilidade para sistemas complexos, baseado no uso do método da tabela booleana. Torna-se possível, assim, a utilização da informação de confiabilidade do sistema para a tomada de decisões ainda na fase de projeto, referente ao uso da topologia mais adequada em função dos resultados de confiabilidade obtidos. O método da tabela boolena é capaz de prever todas as combinações possíveis para uma determinada topologia.

A estrutura deste artigo obedece a seguinte ordem. A seção 2 apresenta a fundamentação teórica sobre confiabilidade de sistemas complexos e sobre topologias de redes do tipo *mesh*. A seção 3 apresenta a descrição da metodologia proposta para o cálculo da confiabilidade em redes *mesh*. A seção 4 apresenta uma aplicação no

cálculo de uma topologia de rede do tipo *mesh*. Na seção 5 encontra-se a conclusão.

2 Revisão bibliográfica

Esta seção tem como objetivo a apresentação da fundamentação teórica relativa a dois assuntos abordados neste artigo: cálculo de confiabilidade de sistemas complexos e redes *mesh*. Será realizada uma breve revisão dos principais métodos de obtenção de confiabilidade. Após, os conceitos referentes às redes mesh serão apresentados.

2.1 Métodos de análise de confiabilidade de sistemas complexos

Um sistema é considerado complexo quando não pode ser modelado ou decomposto unicamente em sistemas do tipo série, paralelo, ou combinação destes sistemas (FOGLIATTO; RIBEIRO 2009). A obtenção de equações que expressem os eventos de interesse, na maioria das vezes, é uma tarefa extremamente complicada (NELSON *et al.*, 1970). A Figura 1 mostra um exemplo de sistema complexo.

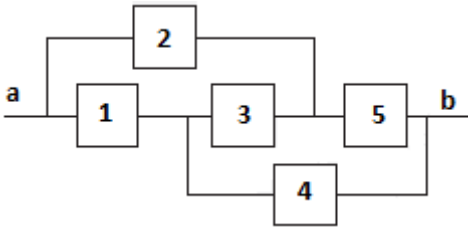

Figura 1: Exemplo de sistema complexo

A literatura apresenta diversas ferramentas para a obtenção da confiabilidade de sistemas complexos, dentre as quais pode-se destacar o método da decomposição, métodos de *tie set* e *cut set*, método da tabela boolena e método da tabela de redução. A aplicabilidade de cada uma das técnicas depende muito da complexidade do sistema analisado (FOTUHI-FIRUZABAD *et al.*, 2004).

O método da decomposição calcula a confiabilidade do sistema complexo a partir da identificação de um elemento considerado chave para o sistema. A escolha é baseada no número de caminhos em que o elemento chave participa. Em seguida, são geradas duas hipóteses para o sistema. A primeira baseia-se na probabilidade do sistema estar funcionando, supondo que o elemento chave esteja operante; a segunda considera a probabilidade do sistema estar em modo de falha, para o caso do elemento chave não operar. A confiabilidade total do sistema é obtida pela soma das duas hipóteses (FOGLIATTO; RIBEIRO 2009).

Os métodos de *tie set* e *cut set* são métodos tambem utilizados na obtenção de confiabilidades de sistemas complexos (AHLUWALIA; LI 2011). Nelson *et al*. (1970) definem *tie set* como o caminho formado pelo menor número possível de elementos operantes que ligam a entrada e a saída do sistema, e *cut set* como o conjunto de elementos que, ao serem inutilizados, interrompem o caminho entre a entrada e a saída.

O objetivo dos métodos *tie set* e *cut set* é encontrar o conjunto mínimo de elementos que satisfaçam suas definições. Entretanto, a identificação dos *tie sets* e *cut sets* mínimos nem sempre é uma tarefa simples. Na medida que o número de elementos aumenta, a tarefa se torna impossível de ser realizada apenas inspecionando visualmente o sistema para verificar as interconexões dos elementos (AHLUWALIA; LI 2011).

Fotuhi-Firuzabad *et al.* (2004) propõem um algoritmo chamado "*Path tracing Algorithm*" para calcular os *tie sets* de um sistema, utilizando como ponto de partida a matriz de conexão da rede. A matriz de conexão da rede é uma matriz quadrada que representa os *n* elementos do sistema. Cada linha da matriz representa o nó inicial e cada coluna representa o nó final. A matriz resultante normalmente é do tipo esparsa, uma vez que um elemento do sistema se conecta não com todos os elementos, mas com apenas alguns (FOTUHI-FIRUZABAD *et al.*, 2004). A partir dessa matriz, Ahlluwalia e Li (2011) elaboraram um algoritmo que realiza simplificações no sistema e, aliado aos métodos de *tie set* e *cut set*, reduzem drasticamente o tempo de processamento dos resultados.

Outro método utilizado é o da tabela boolena. O método consiste na construção de uma tabela que contenha todas as hipóteses possíveis que o sistema venha a assumir. O número de combinações é função do número de elementos do sistema (FOGLIATTO; RIBEIRO 2009). A equação (1) informa o número possível de combinações, onde *n* representa o número de elementos do sistema.

$$\text{Combinações} = 2^n \qquad (1)$$

Na medida em que o número de elementos cresce, a complexidade da resolução por este método também aumenta (FOGLIATO; RIBEIRO 2009). A Figura 2 ilustra parcialmente a tabela boolena gerada.

1	2	3	4	5	SISTEMA	PROBABILIDADE
1	0	0	1	0	0	P(1)[1- P(2)] [1- P(3)] P(4) [1- P(5)]
1	0	1	0	1	1	P(1)[1- P(2)]P(3)[1- P(4)] P(5)
1	1	0	0	1	0	P(1) P(2)[1- P(3)][1- P(4)] P(5)
1	1	1	0	0	0	P(1) P(2) P(3)[1- P(4)] [1- P(5)]
0	1	1	0	1	1	[1- P(1)] P(2) P(3)[1- P(4)] P(5)
1	1	1	1	1	1	P(1)P(2)P(3)P(4) P(5)

Figura 2: Tabela booleana de parte do sistema complexo da Figura 1

A confiabilidade total do sistema é obtida através do somatório das confiabilidades das combinações que deixam o sistema funcional (FOGLIATTO; RIBEIRO 2009).

O método da tabela de redução é uma variação do método da tabela booleana. Para a utilização do método, deve-se montar a tabela booleana do sistema. Busca-se na tabela todas as combinações de possibilidades que levem o sistema a um estado operacional (FOGLIATTO; RIBEIRO 2009).

Sempre que o elemento estiver operante na hipótese considerada válida, representa-se seu valor por P(x); já quando o elemento não estiver operante, deve-se utiliza-se o estado negado de P(x). A partir das equações encontradas, aplicam-se técnicas de simplificações de álgebra booleana até chegar na combinação de menor número de elementos possíveis (IDOETA; CAPUANO 1998).

2.2 Redes *Mesh*

As redes *mesh* foram criadas a partir da mistura de elementos fixos com elementos móveis interconectados e sem a utilização de fios, tendo como premissa a flexibilidade e o baixo custo (BRUNO *et al.*, 2005).

Poor e Hodges (2004) definem redes *mesh* a partir de suas características (uma rede *ad-hoc*, *multi-hope*) e as classificam como um subset das redes *ad-hoc*. Já Akyildiz *et al.* (2005) define redes *mesh* como uma rede auto-organizada, auto-configurada, tendo seus elementos a capacidade de estabelecer e manter uma conexão entre si. Entretanto, Akyildiz *et al.* (2005) consideram que as redes *ad-hoc* são um subset das redes *mesh* e não o contrário, baseado principalmente no fato de que as redes *mesh* necessitam de algoritmos muito mais sofisticados para realizar as tarefas de controle e roteamento.

Tipicamente, existem dois elementos na hierarquia das redes *mesh*: os roteadores *mesh* e os clientes *mesh*. Os clientes são estruturas que podem ter alguma mobilidade e normalmente possuem apenas uma interface de rede (MUOGLIM *et al., 2011).*

Os roteadores normalmente são estruturas fixas e provêem o acesso dos clientes à rede. Alguns ainda possuem capacidades adicionais e funcionam como *gateways* para interconexão com outras redes *mesh* e outros protocolos de rede. Para executar a função de *gateway*, normalmente possuem mais de uma interface de rede (AKYILDIZ *et al.*, 2005).

As redes *mesh* podem ser colocadas de 3 formas em relação a sua arquitetura: arquitetura *backbone*, arquitetura *client* e arquitetura híbrida. Na arquitetura do tipo *backbone*, os roteadores formam a infraestrutura para que os clientes se conectem não somente às redes *mesh*, mas também a outros protocolos de rede. Normalmente esta arquitetura é utilizada com diversos tipos de tecnologia de rádio (AKYILDIZ *et al.*, 2005).

A arquitetura do tipo *client* promove uma conexão ponto a ponto dos elementos da rede. Nesta arquitetura os elementos precisam ter funções adicionais de roteamento e configuração. O pacote destinado a um nó passa por múltiplos elementos da rede até chegar ao nó de destino. A vantagem desta arquitetura está na mobilidade dos seus elementos (AKYILDIZ *et al.*, 2005).

A arquitetura híbrida é formada pela composição das duas arquiteturas anteriores. Os clientes podem acessar outras redes através dos roteadores, e sua mobilidade faz com que a cobertura da área aumente significativamente (AKYILDIZ *et al.*, 2005). A Figura 3 ilustra um exemplo de uma rede *mesh* com arquitetura híbrida.

As redes *mesh* têm capacidade de auto-configuração, auto-organização e auto-diagnóstico. Não é necessária a configuração manual de cada elemento adicionado à rede e, em caso de falha, a rede *mesh* se adapta à perda do nó e procura uma nova rota até que o destino final seja alcançado (POOR; HODGES 2004).

Algumas questões devem ser consideradas nos projetos de redes *mesh*, como por exemplo a interferência e perda de eficiência provocada pelo uso de antenas omnidirecionais (HSU; RUBIN 2008). Ramanathan (2001) realizou um estudo comparativo entre os tipos de antenas empregadas e concluiu que as antenas direcionais do tipo *beamforming* ofereceram um aumento significativo no atravessamento e uma redução no atraso dos pacotes.

Contudo, o uso de antenas direcionais provoca uma redução no número de nós visíveis por elementos da rede *mesh* (AKYILDIZ *et al.*, 2005). Para contornar esta limitação, os protocolos da camada de enlace de rede, roteamento e transporte utilizam técnicas mais avançadas de controle de fluxo, roteamento e conexão (JAIN *et al.*, 2003).

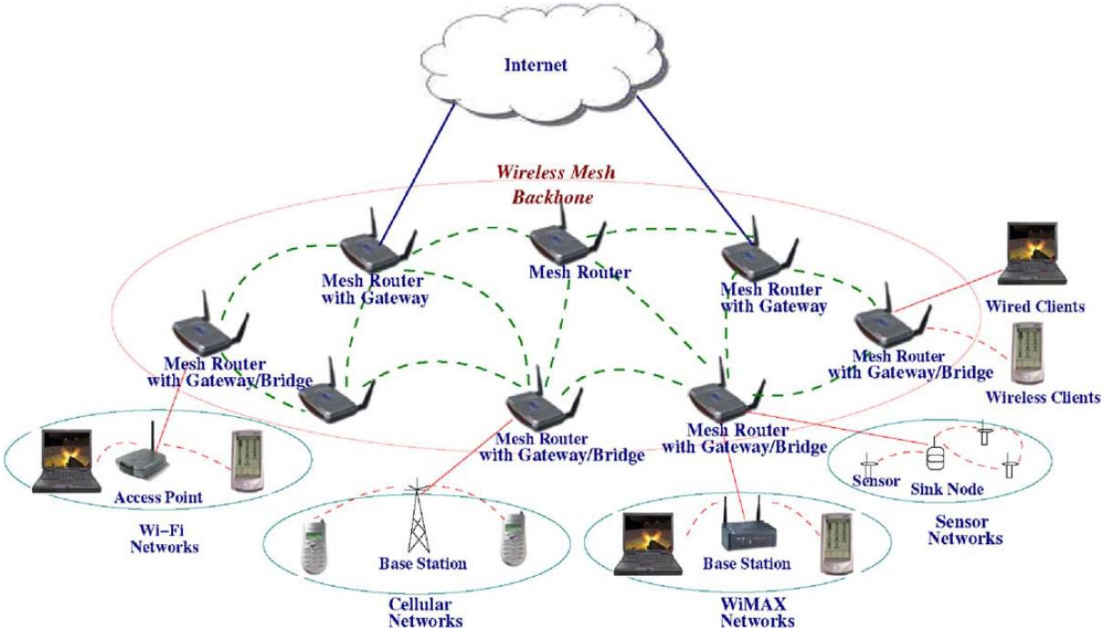

Figura 3: Exemplo de uma rede *mesh* híbrida

Akyldiz *et al.* (2005) acreditam que a questão de segurança tambem deve ser levada em consideração nos projetos de redes *mesh*. Martignon *et al.* (2009) identificam duas áreas: uma relacionada aos clientes finais e outra relacionadas ao *backbone*. Salem e Hubaux (2006) exemplificam ataques às redes *mesh* e citam como os fatores mais críticos na questão de segurança à detecção de elementos corrompidos na rede, a falta de um elemento central que realize a autenticação dos elementos, e um protocolo seguro de roteamento.

3 Método

Esta seção descreve os passos propostos para a obtenção da confiabilidade de um sistema complexo utilizando o método da tabela boolena. Serão apresentados fluxogramas para facilitar o entendimento das etapas do algoritmo implementado. O método proposto pode ser subdivido em 4 etapas, conforme apresentado na Figura 4. Cada etapa será explicada detalhadamente.

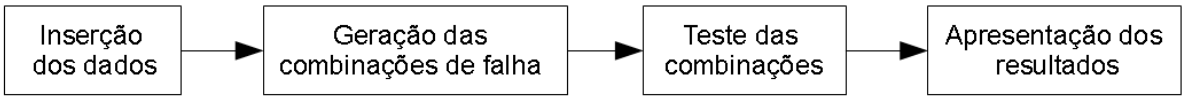

Figura 4: Diagrama geral do método

Passo 1 – Inserção dos dados

O projeto de uma rede *mesh* em ambiente industrial deve garantir um alto valor de confiabilidade aos dados que trafegam. Fica a critério do projetista o posicionamento dos elementos no ambiente fisíco, para atender demandas de segurança e critérios específcos para cada ambiente.

Com o dimensionamento e a topologia da rede em mãos, deve-se inserir os dados necessários ao *software*. A Figura 5 mostra o fluxograma para a inserção dos dados. Deve-se fornecer a quantidade de elementos contidos na rede. O sistema calculará a quantidade necessária de testes para a rede.

A próxima tarefa é a montagem da matriz de conexão para informar os links existentes entre os elementos da rede. A linha da matriz representa o ponto inicial do link, e a coluna seu ponto final. Ao final de cada linha, o *software* solicita que seja informado se o elemento faz parte do início ou do final do sistema. Em caso afirmativo para cada uma das informações, coloca-se o valor '1'. Em caso negativo, o valor '0'.

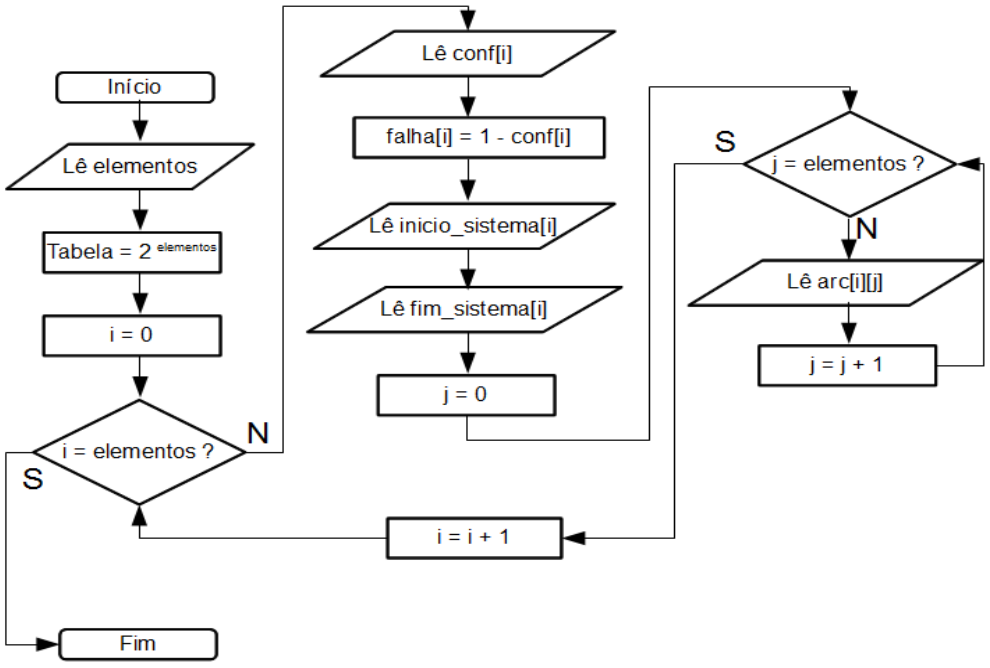

Figura 5: Fluxograma de inserção dos dados

Passo 2 – Geração das combinações de Falha

As combinações geradas consideram as falhas em cada elemento como independentes. Além disso, cada elemento pode assumir dois estados: funcional ou não funcional. O número de combinações geradas varia em função do número de elementos do sistema.

Deve-se montar uma matriz de combinações para testes de 2^n combinações por n elementos. A Figura 6 fornece um fluxograma da geração das combinações possíveis para os sistemas calculados.

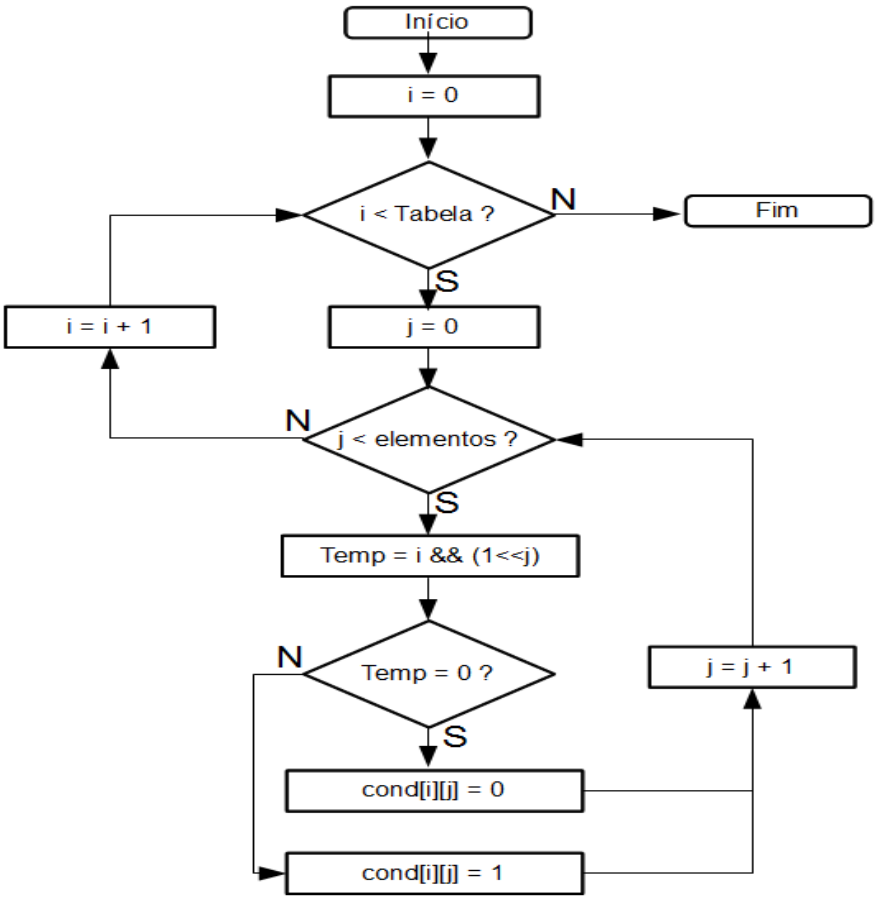

Figura 6: Fluxograma da geração das combinações de falha

Passo 3 – Teste das combinações

O teste das combinações geradas constitui-se na parte central do softwate. Nem todas as hipóteses necessitam passar por esse teste; descartam-se aquelas que contenham todos os elementos iniciais ou finais em estado de falha.

Para cada combinação, o algoritmo percorrerá o sistema até encontrar um caminho válido entre os pontos inicial e final. Basta encontrar um caminho possível para considerar a combinação válida. É possível que exista mais de um caminho, mas esse será desconsiderado no cálculo da confiabilidade. A Figura 7 mostra a máquina de estados para a realização dos testes de cada combinação.

- **PREPARA_TESTE:** contabiliza o número de elementos que possuem um link com o início do sistema, indicando o número de caminhos iniciais possíveis e servindo de controle para reconhecimento de uma condição inválida.

- **ATUALIZA_ELEMENTO_INICIAL:** encontra e armazena os elementos iniciais do sistema. Assegura que o primeiro elemento do caminho seja um elemento inicial. A Figura 8 fornece mais detalhes desse estado.

- **VERIFICA_ULTIMO_ELEMENTO:** verifica se o elemento testado é um elemento final do sistema. A Figura 9 fornece mais detalhes desse estado.

- **ENCONTRA_PROXIMO_ELEMENTO:** encontra o próximo elemento válido para o caminho. Caso não exista, informa ao estado RETORNA_ELEMENTO que o elemento já foi testado na hipótese analisada. A Figura 10 e a Figura 11 fornecem mais detalhes sobre esse estado.

- **RETORNA_ELEMENTO_ANTERIOR:** retorna ao elemento anterior com o objetivo de tentar um novo caminho. A Figura 12 fornece mais detalhes sobre esse estado.

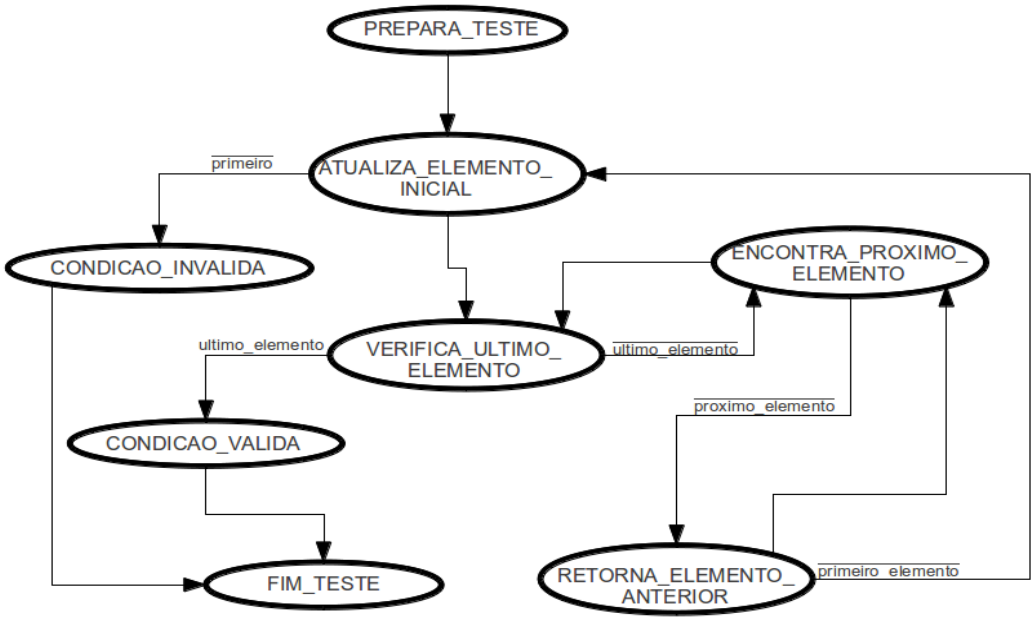

Figura 7: Diagrama de estados dos testes

As indicações de CONDICAO_VALIDA e CONDICAO_INVALIDA foram adicionadas na Figura 7 para facilitar o entendimento sobre os resultados alcançados em cada combinação.

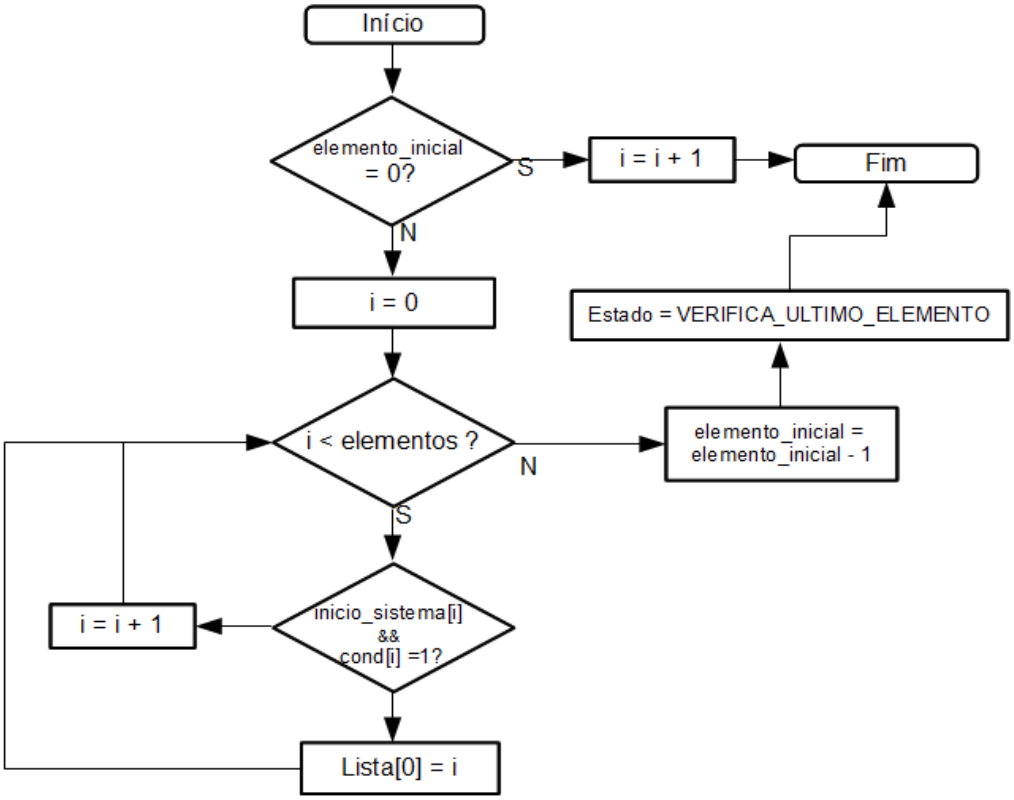

Figura 8: Fluxograma do estado ATUALIZA_ELEMENTO_INICIAL

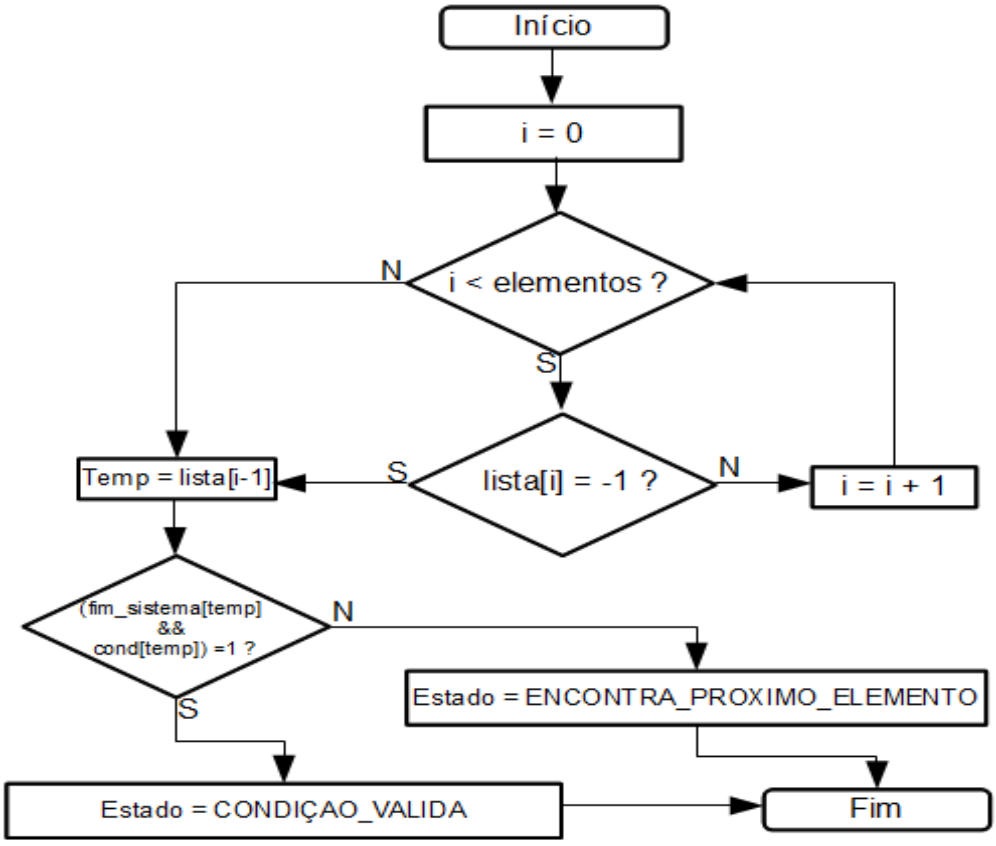

Figura 9: Fluxograma do estado VERIFICA_ULTIMO_ELEMENTO

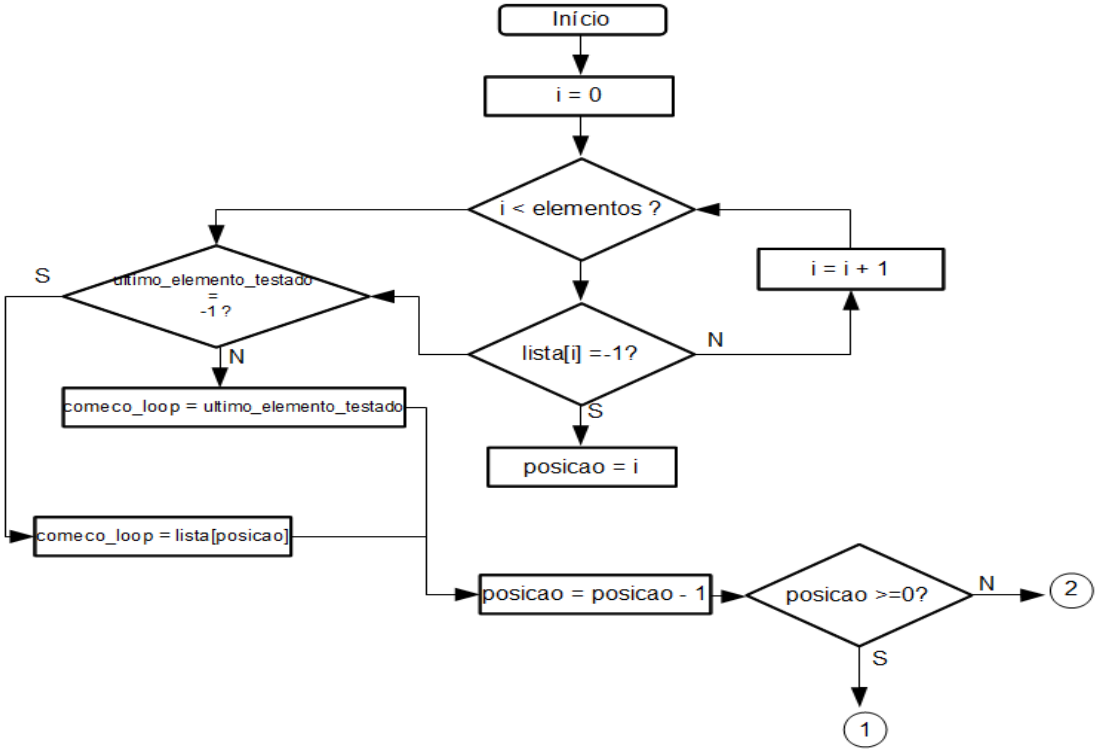

Figura 10: Parte inicial do fluxograma do estado ENCONTRA_PROXIMO_ELEMENTO

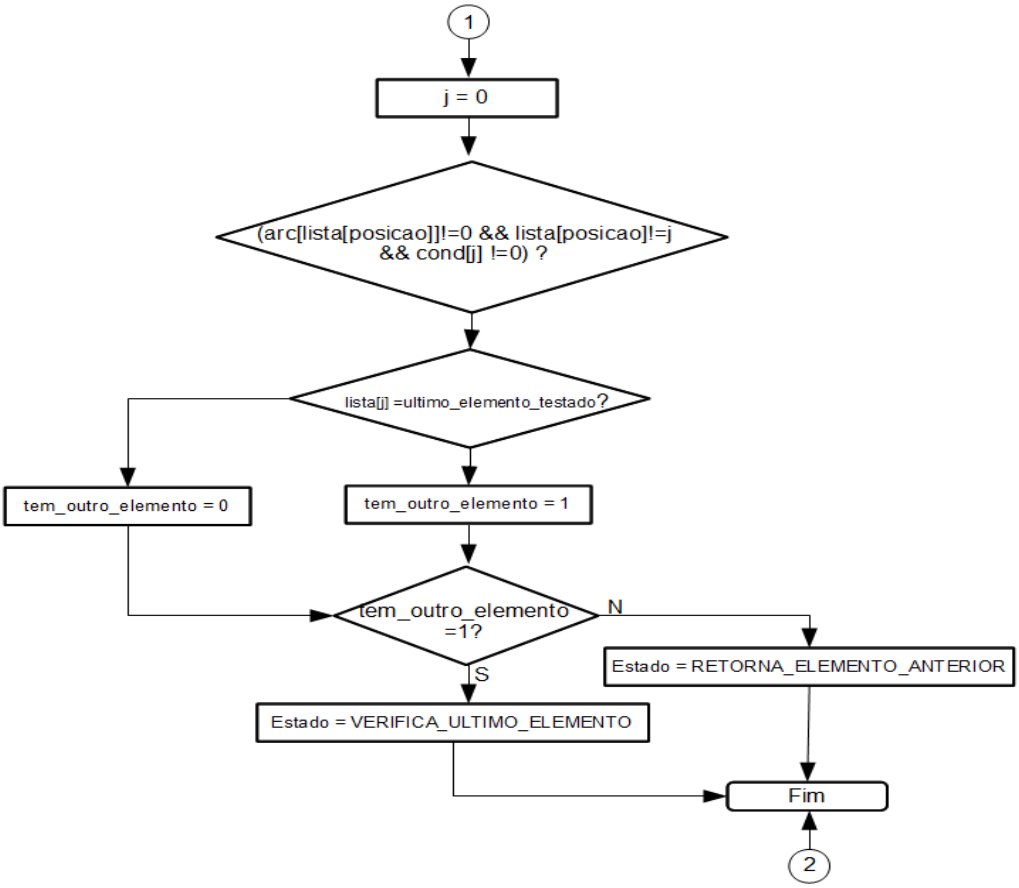

Figura 11: Parte final do fluxograma do estado ENCONTRA_PROXIMO_ELEMENTO

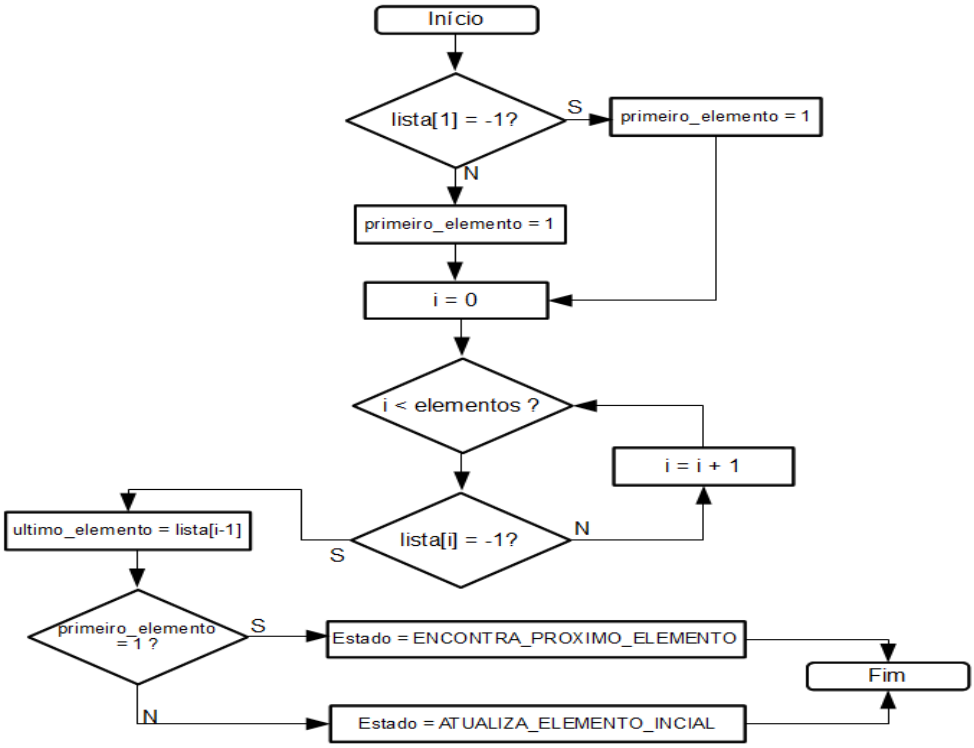

Figura 12: Fluxograma do estado RETORNA_ELEMENTO_ANTERIOR

Passo 4 – Apresentação dos resultados

Os dados são apresentados mostrando os testes de cada combinação e a condição encontrada. Caso a condição seja válida, a confiabilidade desta hipótese é calculada e somada à confiabilidade total do sistema.

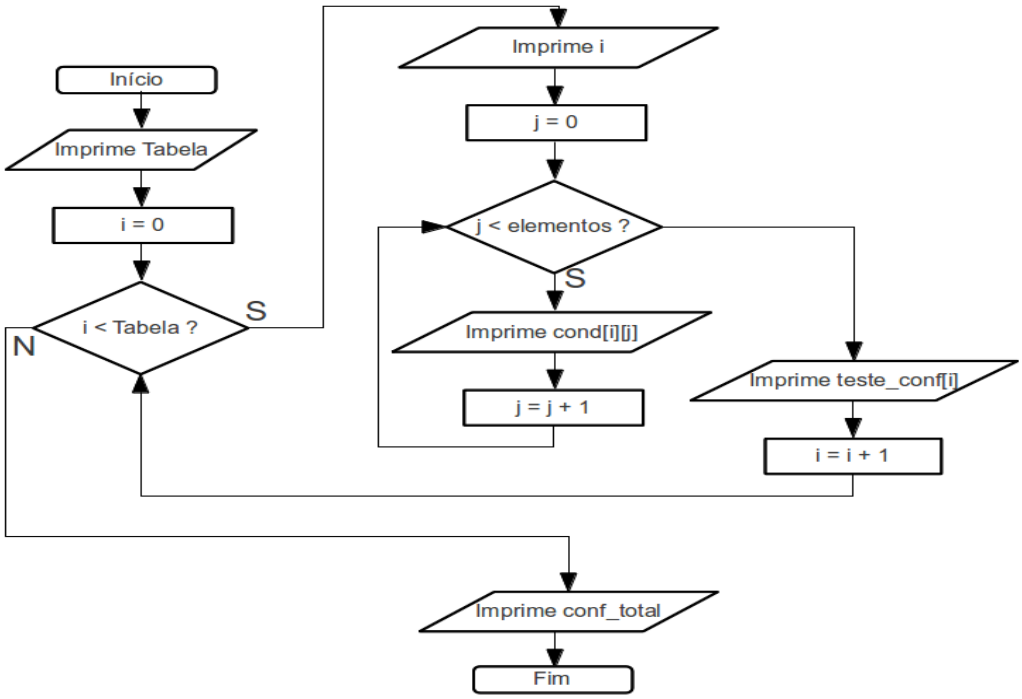

Figura 13: Fluxograma de apresentação dos resultados

4 Estudo de caso

O método proposto será demostrado no sistema apresentado na Figura 14. Trata-se de uma topologia hipotética de um projeto de uma rede *mesh*. A confiabilidade de todos os elementos tem o mesmo valor.

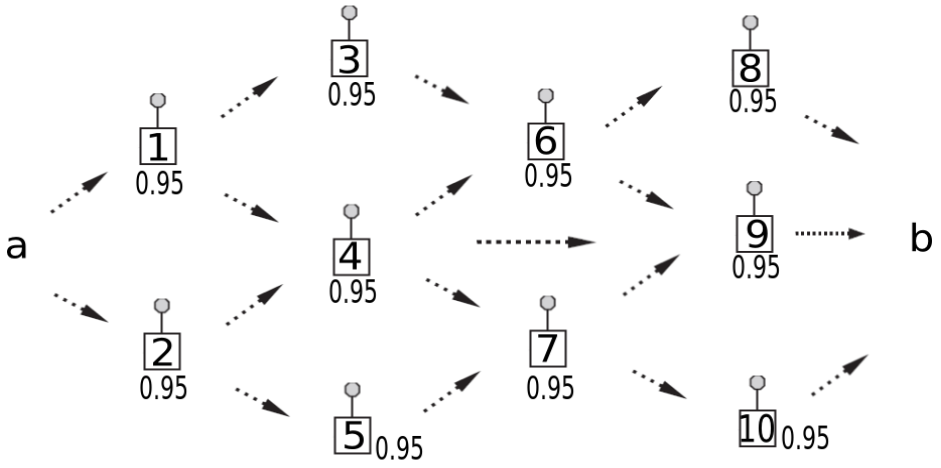

Figura 14: Exemplo de aplicação

A partir da Figura 14 monta-se a matriz de conexão dos elementos, mostrada na Figura 15. Os dados serão informados ao *software* da esquerda para a direita e de cima para baixo. Analisando visualmente a Figura 147 vemos que os elementos iniciais do sistema são elementos ‘1’ e ‘2’ e os elementos finais são os elementos ‘8’, ‘9’ e ‘10’. A confiabilidade de todos os elementos terá o valor de 0,95.

	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	0	0	0	0	0	0
2	0	0	0	1	1	0	0	0	0	0
3	0	0	0	0	0	1	0	1	0	0
4	0	0	0	0	0	1	1	0	0	0
5	0	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Figura 15: Matriz de conexão dos elementos

A primeira informação inserida será o número de elementos da rede. Neste caso, o número informado será 10. O sistema gerará 1024 combinações possíveis de testes.

Os resultados apresentados na Figura 16 mostram algumas das hipóteses que foram testadas. Todas as combinações foram testadas, entretanto apenas algumas delas foram apresentadas.

1	2	3	4	5	6	7	8	9	10	Condição	Confiabilidade
1	1	0	1	1	0	0	1	0	0	inválida	0
0	0	1	1	1	0	0	1	0	0	inválida	0
1	0	1	1	1	0	0	1	0	0	inválida	0
0	1	1	1	1	0	0	1	0	0	inválida	0
0	0	1	0	1	0	0	0	1	0	inválida	0
1	0	1	0	1	0	0	0	1	0	inválida	0
0	0	0	1	1	0	0	0	1	0	inválida	0
1	0	0	1	1	0	0	0	1	0	válida	1,27E-02
0	1	0	1	1	0	0	0	1	0	válida	1,27E-02
1	1	0	1	1	0	0	0	1	0	válida	2,42E-01
0	0	1	1	1	0	0	0	1	0	inválida	0
1	0	1	1	1	0	0	0	1	0	válida	2,42E-01
0	1	1	1	1	0	0	0	1	0	válida	2,42E-01
1	1	1	1	1	0	0	0	1	0	válida	4,59E+00
0	0	0	0	0	1	0	0	1	0	inválida	0
1	0	0	0	0	1	0	0	1	0	inválida	0
0	1	0	0	0	1	0	0	1	0	inválida	0
1	1	0	0	0	1	0	0	1	0	inválida	0
1	0	0	1	0	0	0	1	1	0	válida	1,27E-02
0	0	1	0	1	1	0	1	1	0	inválida	0
1	0	1	0	1	1	0	1	1	0	válida	4,59E+00
0	1	1	0	1	1	0	1	1	0	inválida	0
1	1	1	0	1	1	0	1	1	0	válida	8,73E+01
0	0	0	1	1	1	0	1	1	0	inválida	0
1	0	0	1	1	1	0	1	1	0	válida	4,59E+00
0	1	0	0	1	0	1	1	1	0	válida	2,42E-01

1	1	0	0	1	0	1	1	1	0	inválida	0
0	0	1	0	1	0	1	1	1	0	inválida	0
0	1	0	1	0	1	0	0	1	1	válida	2,42E-01
1	1	0	1	0	1	0	0	1	1	válida	4,59E+00
0	0	1	1	0	1	0	0	1	1	inválida	0
1	0	1	1	0	1	0	0	1	1	válida	4,59E+00
0	1	1	1	0	1	0	0	1	1	válida	4,59E+00
1	1	1	1	0	1	0	0	1	1	válida	8,73E+01
1	0	1	0	1	1	0	0	1	1	válida	4,59E+00
0	1	1	0	1	1	0	0	1	1	inválida	0
1	1	0	0	0	1	0	1	1	1	inválida	0
0	0	1	0	0	1	0	1	1	1	inválida	0
1	0	1	0	0	1	0	1	1	1	válida	4,59E+00
0	1	1	0	0	1	0	1	1	1	inválida	0
1	1	1	0	0	1	0	1	1	1	válida	8,73E+01
0	0	0	1	0	1	0	1	1	1	inválida	0
1	0	0	1	0	1	0	1	1	1	válida	4,59E+00
0	1	0	1	0	1	0	1	1	1	válida	4,59E+00
0	1	1	0	1	0	1	1	1	1	válida	8,73E+01
1	1	1	0	1	0	1	1	1	1	inválida	0
0	0	0	1	1	0	1	1	1	1	inválida	0
1	0	0	1	1	0	1	1	1	1	válida	8,73E+01
0	1	0	1	1	0	1	1	1	1	válida	8,73E+01
1	1	0	1	1	0	1	1	1	1	válida	1,66E+03
0	0	1	1	1	0	1	1	1	1	inválida	0
1	0	1	1	1	0	1	1	1	1	válida	1,66E+03
0	1	1	1	1	0	1	1	1	1	válida	1,66E+03
1	1	1	1	1	0	1	1	1	1	válida	3,15E+04
0	0	0	0	0	1	1	1	1	1	inválida	0
1	0	0	0	0	1	1	1	1	1	inválida	0
0	1	0	0	0	1	1	1	1	1	inválida	0
1	0	1	1	1	1	1	1	1	1	válida	3,15E+04
0	1	1	1	1	1	1	1	1	1	válida	3,15E+04
1	1	1	1	1	1	1	1	1	1	válida	5,99E+05

Figura 16: Apresentação parcial dos resultados obtidos

A confiabilidade calculada para o sistema tem o valor de 0.992074.

5 Conclusões

O método proposto apresentou um algoritmo e um *software* para o cálculo de confiabilidade em redes *mesh*, baseado no método da tabela *booleana*. O cálculo baseia-se na coleta de informações sobre a disposição dos elementos na rede. Deve-se informar ao *software* os valores da matriz de conexão, a confiabilidade de cada elemento e qual a sua relação com o ponto inicial e final do sistema. O *software* calculará a viabilidade e a confiabilidade de cada combinação, informando a confiabilidade total do sistema.

O *software* apresentou resultados satisfatórios para o cálculo de confiabilidade de sistemas complexos, principalmente para sistemas que contenham muitos elementos. A principal desvantagem da utilização desse método reside no fato de que para cada combinação da tabela é necessário analisar novamente o sistema. Nesse contexto, a ferramenta apresentou um considerável ganho, uma vez que o tempo demandado para testes da tabela *boolena* é praticamente nulo.

Como sugestão de trabalhos futuros, sugere-se a criação de uma interface gráfica que facilite a montagem da matriz adjacente, pois esta tarefa demanda a maior quantidade de tempo para o teste. Outra possibilidade de melhoria seria a realização de uma análise automática sobre os resultados, identificando possíveis elementos cuja falha resultaria em danos críticos ao sistema.

Referências Bibliográficas

AHLUWALIA R., LI C. An efficient approach to representation and simplification of complex networks Computers & Industrial Engineering. Vol 61 p. 525-528, 2011.

AKYILDIZ I. F., WANG X., WANG W. *Wireless mesh networks: a survey, Computer Networks* 47 (4) p.445–487, 2005.

BRUNO R., CONTI M., GREGORI E. Mesh networks: commodity multihop ad hoc networks, IEEE Communications Magazine 43 (3) 123–131, 2005.

CHÉRIT A. & BAUQUIS, P. R. & SIGONNEY P. *Health, Safety, the Enviroment, Ethics* . In: BRET-ROUZAUT N. & FAVENNEC J. P. (Coordinators). *Oil and Gas Explorations and Prodution: Reserves, Costs, Contracts*. Technip. p. 285-299, 2011.

FOGLIATO F. S. & RIBEIRO J. L. D. *Confiabilidade e manutenção Industrial*. Elsevier, 2009.

FOTUHI-FIRUZABAD M., BILLINTON R., MUNIAN T. S., VINAYAGAM B. *A Novel Approach to Determine Minimal Tie-Sets of Complex Network IEEE Transactions on Reliability*, vol. 53 NO 1, p. 61–70, 2004.

HOSSAIN E.; LEUNG K. *Wireless Mesh Networks: Architectures and Protocols*. Springer, 2008.

HSU J., RUBIN I. *Directional random access scheme for mobile ad hoc networking using beamforming antenna*, Ad Hoc Networks 6, p. 127–153, 2008.

IDOETA A. V., CAPUANO F. G. *Elementos de Eletrônica Digital.* São Paulo: Érica 1998.

JAIN K., PADHYE, J. PADMANABHAN, QIU L. *Impact of interference on multi-hop wireless network performance, in: ACM Annual International Conference on Mobile Computing and Networking (MOBICOM)*, p. 66–80, 2003.

LAFRAIA J. R. B. *Manual de confiabilidade, Mantenibilidade e Disponibilidade*. Rio de Janeiro: Qualimark, 2001

MARTIGNON F., PARIS S., CAPONE A. *Design and implementation of MobiSEC: A complete security architecture for wireless mesh networks, Computer Networks* 53 p. 2192–2207, 2009.

MOORE, M. R. *Wireless Sensor Networks*. In: Dowla F. (Editor). *Handbook of RF and Wireless Technologies*. Elsevier. p. 355-373, 2004.

MORING, J. T. *A Survey of RF and Wireless Technology*. In: Dowla F (Editor). *Handbook of RF and Wireless Technologies*. Elsevier. p.1-22, 2004.

MUOGLIM E., LOO K. K., COMLEY R. *Wireless mesh networksecurity: A traffic engineering management approach*, Journal of Network and Computer Applications 34, p. 478–491, 2011.

NELSON A. C., BATTS J. R., BEADLES R. L. *A computer program for approximating system reliability*. *IEEE Transactions on Reliability*, vol. R-19, p. 61–65, 1970.

POOR, R. D. & HODGES B. *Reliable Wireless Networks for Industrial Applications*. In: Dowla. F. (Editor). *Handbook of RF and Wireless Technologies*. Elsevier. p.401-416, 2004.

RAMANATHAN R. *On the performance of ad hoc networks with beamforming antennas*, in: ACM/SIGMOBILE Mobi-Hoc2001, 2001.

SALEM N. B., HUBAUX J.-P. *Securing wireless mesh networks, IEEE Wireless Communications* 13 (2) p.50–55, 2006.

TANG J. *Mechanical system reliability analysis using a combination of graph theory and Boolean function, Reliability Engineering & System Safety* 72 (1) p.21-30, 2001.

Anexos

Em anexo está o códico fonte do método apresentado no artigo. A linguagem de computação utilizada é a linguagem C.

```
/*
 * main.c
 *
 *  Created on: Jul 23, 2012
 *      Author: lmeyer
 */
#include "main.h"
#include "nodes.h"
#include "table.h"


int main()
{
	int answer;
	int ret_scanf;
	struct _graph g;

	do{
		printf("Teste de confiabilidade baseados na tabela verdade \n\n");
		do{
			printf("Escolha uma opção abaixo:\n");
			printf("1 - Inserir sistema\n");
			printf("2 - Mostrar a matriz do sistema\n");
			printf("3 - Calcular confiabilidade\n");
			printf("4 - Sair do programa\n\n");
			printf("Escolha uma opção: ");
			ret_scanf = scanf("%d", &answer);
		}while((answer < 1) && (answer > 4));

		switch(answer){
		case 1:
			insert_system(&g);
			break;
		case 2:
			print_matriz(&g);
			break;
		case 3:
			calc_conf(&g);
			print_conf(&g);
			break;
		}
	}while(answer != 4);

	return 0;
}


/*
```

```
 * main.h
 *
 *  Created on: Jul 23, 2012
 *      Author: lmeyer
 */

#ifndef MAIN_H_
#define MAIN_H_

#include <stdio.h>
#include <stdlib.h>

#define MAX_NODES 20

struct _node{
	float conf;
	float fail;
	int begin;
	int end;
	int condition;
};

struct _arc{
	int adj;
};

struct _graph{
	struct _node node[MAX_NODES];
	struct _arc arc[MAX_NODES][MAX_NODES];
	int number_nodes;
	int size_test_table;
	float system_conf;
};

#endif /* MAIN_H_ */
```

Figura 17: Código fonte main.c e main.h

```
/*
 * list.c
 *
 *  Created on: Sep 7, 2012
 *      Author: lmeyer
 */

#include "list.h"
#include "operation_list.h"
#include <stdlib.h>
#include <string.h>

CONDITION_STATE test_valid_condition(struct _graph *g)
{
	int *list;
	STATE state;
	CONDITION_STATE condition;
	int first_elements;
	int last_element;
	int has_next_element;
```

```
	int last_element_tested = -1;
	int is_first_element = -1;

	state = FIND_FIRST_STATE;
	condition = NOT_TESTED_CONDITION;

	list =(int *)malloc(sizeof(int)*g->number_nodes);

	if(list==NULL){
		return condition = ERROR_CONDITION;
	}

	memset_list(list, g->number_nodes);
	first_elements = discover_first_elements(g);

	do{
		switch(state){
		case FIND_FIRST_STATE:
			if(first_elements){
				find_initial_node(g, list);
				first_elements--;
				state = VERIFY_LAST_ELEMENT;
			}else{
				condition = INVALID_CONDITION;
			}
			break;
		case FIND_NEXT_ELEMENT:
			has_next_element = find_next_node(g, list,
last_element_tested);
			if(has_next_element){
				state = VERIFY_LAST_ELEMENT;
			}else{
				state = RETURN_PREVIOUS_ELEMENT;
			}
			break;
		case VERIFY_LAST_ELEMENT:
			last_element = verify_last_element(g, list);
			if(last_element){
				condition = VALID_CONDITION;
			}else{
				state = FIND_NEXT_ELEMENT;
			}
			break;
		case RETURN_PREVIOUS_ELEMENT:
			is_first_element = test_first_element(list);
			last_element_tested = return_previous_element(list, g-
>number_nodes);
			if(is_first_element){
				state = FIND_FIRST_STATE;
			}else{
				state = FIND_NEXT_ELEMENT;
			}
			break;
		}
	}while((condition!=VALID_CONDITION)&&(condition!=INVALID_CONDITION));

	free(list);
	return condition;
```

```
}

void print_list(int *list, int len)
{
	int i;

	printf("\nlist: ");
	for(i=0;i<len;i++){
		printf("%d   ", list[i]);
	}
	printf("\n");
}



/*
 * list.h
 *
 *  Created on: Sep 7, 2012
 *      Author: lmeyer
 */

#ifndef LIST_H_
#define LIST_H_

#include "main.h"

typedef enum _STATE{
	FIND_FIRST_STATE,
	FIND_NEXT_ELEMENT,
	VERIFY_LAST_ELEMENT,
	RETURN_PREVIOUS_ELEMENT,
	INVALID_NODE_LIST,
}STATE;

typedef enum _CONDITION_STATE{
	VALID_CONDITION = 1,
	INVALID_CONDITION = 2,
	ERROR_CONDITION = 3,
	NOT_TESTED_CONDITION = 4
}CONDITION_STATE;

struct _node_list{
	int node;
	int last_node_verified;
	struct _node_list *next;
	struct _node_list *prev;
};

CONDITION_STATE test_valid_condition(struct _graph *g);
void debug_state(STATE state);
void print_list(int *list, int len);
void print_node(int *node);
#endif /* LIST_H_ */
```

Figura 18: Código fonte list.c e list.h

```c
/*
 * nodes.c
 *
 *  Created on: Jul 31, 2012
 *      Author: lmeyer
 */

#include "nodes.h"
#include <math.h>

void insert_system(struct _graph *g)
{
	int ret_scanf;
	int i,j;

	printf("Insira o número de elementos: ");
	ret_scanf = scanf("%d", &g->number_nodes);

	printf ("num_states : %d\n", g->number_nodes);
	printf("Insira a matriz do sistema \n");

	g->size_test_table = pow(2,g->number_nodes);

	for(i=0;i<g->number_nodes;i++){
		printf("Insira o estado %d \n", i+1);
		printf("Confiabilidade do sistema :");
		ret_scanf = scanf("%f", &g->node[i].conf);
		g->node[i].fail = 1 - g->node[i].conf;
		printf("Inicia o sistema: ");
		ret_scanf = scanf("%d", &g->node[i].begin);
		for(j=0;j<g->number_nodes;j++){
			if(i!=j){
				printf("Adjacente ao sistema %d :",j+1);
				ret_scanf = scanf("%d", &g->arc[i][j].adj);
			}else{
				g->arc[i][j].adj = 1;
			}
		}
		printf("Finaliza o sistema: ");
		ret_scanf = scanf("%d", &g->node[i].end);
	}
}

void print_matriz(struct _graph *g)
{
	unsigned int i, j;
	unsigned char c = 'A';

	printf ("Matriz Adjacente do sistema\n\n");
	for(i=0;i<g->number_nodes;i++){
		printf("     %c", c++);
	}
	printf("\n");
	c = 'A';
	for(i=0;i<g->number_nodes;i++){
		printf("%c", c++);
		for(j=0;j<g->number_nodes;j++){
```

```
                    printf("     %d",g->arc[i][j].adj);
            }
            printf("\n");
      }
}

void calc_conf(struct _graph *g)
{
      printf ("Tamanho ta tabela booleana: %d\n", g->size_test_table);

      printf("Matriz de teste \n\n");
      print_states(g);
      test_table(g);
}

void print_conf(struct _graph *g)
{
      printf("Confiabilidade do sistema : %f \n", g->system_conf);
}

void print_states(struct _graph *g)
{
      int i;
      char c = 'A';

      for(i=0;i<g->number_nodes;i++){
            printf("     %c", c++);
      }
      printf("\n");
}

/*
 * nodes.h
 *
 *  Created on: Jul 31, 2012
 *      Author: lmeyer
 */

#ifndef NODES_H_
#define NODES_H_

#include "main.h"

void insert_system(struct _graph *g);
void print_matriz(struct _graph *g);
void calc_conf(struct _graph *g);
void print_conf(struct _graph *g);
void print_states(struct _graph *g);
int find_valid_condition(struct _graph *g);
#endif /* NODES_H_ */
```

Figura 19: Código fonte nodes.c e nodes.h

```
/*
 * operation_list.c
 *
 *  Created on: Oct 20, 2012
 *      Author: lmeyer
```

```
 */

#include "operation_list.h"
/**
 * @brief discover number of first elements that are valid
 * @param g
 * @return number of first elements
 */
int discover_first_elements(struct _graph *g)
{
	int number_elements = 0;
	int i;

	for(i=0;i<g->number_nodes;i++){
		if((g->node[i].begin)&&(g->node[i].condition)){
			number_elements++;
		}
	}
	return number_elements;
}

void find_initial_node(struct _graph *g, int *list)
{
	int i;

	for(i=0;i<g->number_nodes;i++){
		if((g->node[i].begin)&&(g->node[i].condition)){
			list[0] = i;
#ifdef DEBUG
			print_node(&i);
			break_program();
#endif
			break;
		}
	}
}

int find_next_node(struct _graph *g, int *list, int last_element_tested)
{
	int position;
	int j;
	int has_next_element = 1;
	int start_loop;

	position = find_first_free_position(list, g->number_nodes);
	if(last_element_tested!=-1){
		start_loop = last_element_tested;
	}else{
		start_loop = list[position];
	}

	position--;
	if(position>=0){
		for(j=start_loop;j<g->number_nodes;j++){
			if((g->arc[list[position]][j].adj)&&(list[position]!=j)&&(g-
>node[j].condition)){
				if(verify_element(list, j, g->number_nodes,
last_element_tested)){
```

```
                                    list[++position] = j;
                                    has_next_element = 1;
                                    return has_next_element;
                              }

                        }
                  }
            }
            has_next_element = 0;
            return has_next_element;
}

int find_first_free_position(int *list, int len)
{
      int i;
      int position;

      for(i=0;i<len;i++){
            if(list[i]==-1)
                  return i;
      }
      return -1;
}

int verify_element(int *list, int value, int len, int last_element_tested)
{
      int ret;
      int i;

      if(value==last_element_tested){
      return 0;
      }
      ret = 1;
      return ret;
}

int return_previous_element(int *list, int len)
{
      int i;
      int last_element;

      for(i=0;i<len;i++){
            if(list[i]==-1){
                  break;
            }
      }
      last_element = list[i-1];
      list[i-1] = -1;
      return last_element;
}

int test_first_element(int *list)
{
      int ret;

      if(list[1]== -1){
      ret = 1;
      }else{
```

```
                ret = 0;
        }
        return ret;
}

int verify_last_element(struct _graph *g, int *list)
{
        int last_element;
        int i;
        int element;

        for(i=0;i<g->number_nodes;i++){
                if(list[i]== -1)
                        break;
        }
        element = list[i-1];
        if((g->node[element].end)&&(g->node[element].condition)){
                last_element = 1;
        }else{
                last_element = 0;
        }
        return last_element;
}

void print_node(int *node)
{
        char c = 'A';

        printf("\nNODE : %c\n", c+(*node));
}

void memset_list(int *list, int len)
{
        int i;

        for(i=0;i<len;i++){
                list[i] = -1;
        }
}

void break_program()
{
        char c;
        int ret;

        ret = scanf("%c",&c);
}
/*
 * operation_list.h
 *
 *  Created on: Oct 20, 2012
 *      Author: lmeyer
 */

#ifndef OPERATION_LIST_H_
#define OPERATION_LIST_H_

#include "main.h"
```

```
int discover_first_elements(struct _graph *g);
void find_initial_node(struct _graph *g, int *list);
int find_next_node(struct _graph *g, int *list, int last_element_tested);
int find_first_free_position(int *list, int len);
int return_previous_element(int *list, int len);
int test_first_element(int *list);
int verify_last_element(struct _graph *g, int *list);
void print_node(int *node);
void memset_list(int *list, int len);
void break_program(void);
#endif /* OPERATION_LIST_H_ */
```

Figura 20: Código fonte operation_list.c operation_list.h

```
/*
 * table.c
 *
 *  Created on: Aug 9, 2012
 *      Author: lmeyer lmeyer@pd3.com.br
 */

#include <stdio.h>
#include "table.h"
#include "list.h"

//#define DEBUG
int find_valid_condition(struct _graph *g)
{
	CONDITION_STATE condition;

	condition = test_valid_condition(g);
#ifdef DEBUG
	printf("condition = %d\n", condition);
#endif
	if(condition==VALID_CONDITION){
		return 1;
	}else{
		return 0;
	}
}

void find_conf_value(struct _graph *g)
{
	int i;
	float state_conf = 1;

	for(i=0;i<g->number_nodes;i++){
		if(g->node[i].condition){
			state_conf *= g->node[i].conf;
		}else{
			state_conf *= g->node[i].fail;
		}
	}
	printf("%e ", state_conf);
	g->system_conf += state_conf;
}
```

```
void set_table(struct _graph *g, int value)
{
	int i;
	unsigned int mask;
	int temp;

	for(i=0;i<g->number_nodes;i++){
		temp = value & (1<<i);
		if(temp){
			g->node[i].condition = 1;
		}else{
			g->node[i].condition = 0;
		}
	}
}

void test_table(struct _graph *g)
{
	int i;

	for(i=0;i<g->size_test_table;i++){
		set_table(g,i);
		printf("%d   ", i+1);
		make_test(g);
	}
}

void make_test(struct _graph *g)
{
	int i;
	int test_start = 0;
	int test_end = 0;
	int valid_condition = 0;

	for(i=0;i<g->number_nodes;i++){
		printf("%d   ", g->node[i].condition);
	}

	test_start = test_condition_start(g);
	test_end = test_condition_end(g);

	if((!test_start) || (!test_end)){
		printf("inválida   ");
		printf("0");
	}else{
		valid_condition = find_valid_condition(g);
		if(valid_condition){
			printf(" válida    ");
			find_conf_value(g);
		}else{
			printf("inválida   ");
			printf("0");
		}
	}
	printf("\n");
}
```

```
int test_condition_start(struct _graph *g)
{
	int i;

	for(i=0;i<g->number_nodes;i++){
		if((g->node[i].begin) && (g->node[i].condition)){
			return 1;
		}
	}
	return 0;
}

int test_condition_end(struct _graph *g)
{
	int i;

	for(i=0;i<g->number_nodes;i++){
		if((g->node[i].end) && (g->node[i].condition))
			return 1;
	}
	return 0;
}


/*
 * table.h
 *
 *  Created on: Aug 9, 2012
 *      Author: lmeyer lmeyer@pd3.com.br
 */

#ifndef TABLE_H_
#define TABLE_H_

#include "main.h"

void set_table(struct _graph *, int);
void test_table(struct _graph *);
void make_test(struct _graph *);
int test_condition_start(struct _graph *);
int test_condition_end(struct _graph *);
void find_conf_value(struct _graph *g);
#endif /* TABLE_H_ */
```

Figura 21: Código fonte operation_list.c operation_list.h