EDUARDO SANT'ANA DA SILVA

TÉCNICAS DE CAMINHOS DISJUNTOS PARA ROTEAMENTO EM SYSTEMS-ON-CHIP

Tese apresentada ao Programa de Pós-Graduação em Informática do Setor de Ciências Exatas da Universidade Federal do Paraná, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.
Orientador: Prof. Eduardo Todt

CURITIBA

2013

S586t Silva, Eduardo Sant'Ana da
Técnicas de caminhos disjuntos para roteamento em *systems-on-chip* / Eduardo Sant'Ana da Silva. – Curitiba, 2013.
153f. : il. color. ; 30 cm.

Tese (doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2013.

Orientador: Eduardo Todt.
Bibliografia: p. 103-115.

1. Sistemas programáveis em chip. 2. Redes em chips. 3. Árvores (Teoria dos grafos). I. Universidade Federal do Paraná. II. Todt, Eduardo del. III. Título.

CDD: 004.64

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa do aluno de Doutorado em Ciência da Computação, Eduardo Sant'Ana da Silva, avaliamos a tese de doutorado intitulada *"Aplicação de técnicas de caminhos disjuntos para roteamento em systems-on-chip"*, cuja defesa pública foi realizada no dia 03 de outubro de 2013, às 15:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após avaliação, decidimos pela (X)**aprovação** ( )**reprovação** do candidato.

Curitiba, 03 de outubro de 2013.

Prof. Dr. Eduardo Todt
DINF/UFPR – Orientador

Prof. Dr. Murilo Vicente Gonçalves da Silva
UTFPR – Membro Externo

Prof. Dr. Djones Vinicius Lettnin
UFSC – Membro Externo

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR – Membro Interno

Prof. Dr. Aldri Luiz dos Santos
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO · UNIVERSIDADE FEDERAL DO PARANÁ · Setor de Ciências Exatas · Coordenação de Pós-Graduação em Informática

RESUMO

Systems-on-chip (*SoCs*) são sistemas compostos contidos em um único substrato de silício. Os *SoCs* foram introduzidos nas metodologias de projeto para atender à crescente demanda de aplicações complexas que requerem um grande poder computacional para sua execução. A utilização de *SoCs* contribui para uma diminuição de consumo de potência, pela ausência de um *clock* global, e para uma diminuição da área utilizada, visto que os componentes contidos em blocos são resultantes de projetos otimizados. As aplicações são compostas por subsistemas presentes em blocos distintos de lógica, cuja interação requer meios de comunicação eficientes para seu adequado funcionamento. Devido à demanda por uma maneira eficiente de comunicação interna aos *SoCs*, surgiram as chamadas *Networks-on-chip* (*NoCs*). Assim como nas redes tradicionais, as *NoCs* possuem problemas a serem resolvidos, dentre eles, a criação de técnicas eficientes de roteamento. Apesar da maioria das *NoCs* implementadas comercialmente utilizarem uma técnica conhecida como *wormhole*, na qual um nó (ou vértice) estabelece um caminho direto até o nó alvo, ainda faz-se necessário evitar a competição por rotas já utilizadas. Dessa forma generalizamos o problema em se obter caminhos disjuntos internos aos *SoCs* em um problema de grafos conhecido como árvores geradoras independentes, que pode ser descrito resumidamente como: Dado um grafo G, um conjunto de árvores geradoras enraizadas em um vértice r em G é dito vértice/aresta independente se, para cada vértice v em G, $v \neq r$, os caminhos de r a v em qualquer par de árvores são vértice/aresta disjuntos. Se a conectividade de G é k, o problema resume-se à construção de k árvores geradoras independentes com cada vértice do grafo como raiz de tais árvores. Esse problema permanece em aberto para grafos em geral com conectividade $k \geq 4$. No entanto, foi demonstrado que, para um hipercubo de dimensão k, denotado por Q_k, existem k árvores geradoras enraizadas em um vértice arbitrário de G. Neste trabalho é proposto um algoritmo para gerar k árvores geradoras independentes sobre hipercubos com o intuito de utilizá-lo na elaboração de técnicas de roteamento eficientes entre os núcleos distintos dos *SoCs* , assim como em processadores de múltiplos núcleos. Dentre as contribuições deste trabalho enfatizamos o consumo reduzido de recursos computacionais utilizados pelo algoritmo proposto, como: memória e processamento; assim como a modificação do algoritmo *ECUBE* para se construir rotas disjuntas sem que seja necessária a construção completa das árvores geradoras independentes. O algoritmo proposto se comporta de forma similar aos algoritmos comumente utilizados em roteamentos em *NoCs*, conforme mostrado pela utilização de um simulador de *NoCs*: *Noxim*. A construção de árvores geradoras disjuntas tende a seguir um dos dois objetivos: construção eficiente e altura ótima. Este trabalho tem o foco na construção eficiente, sendo que a altura ótima foi um resultado inerente ao método de construção proposto.

ABSTRACT

Systems-on-chip (SoCs) are compound systems contained on a single silicon substrate. The SoCs were introduced in design methodologies to meet the growing demand for complex applications that require massive computational power for their execution. The use of SoCs contributes to reduced power consumption, due to the absence of a global clock, and a decrease in the used area, since the components are contained in optimized design blocks. Such applications are composed of subsystems present in distinct logic blocks , whose interaction requires an efficient communication system for their proper functioning. Due to demand for an efficient internal communication to SoCs, the Networks-on-chip (NoCs) have emerged. Just as in traditional networks, the NoCs have problems to be solved, among them we have the creation of efficient routing techniques. Although most commercially NoCs implemented use a technique known as *wormhole* in which a node establishes a path straight to the target node, it is still necessary to avoid competition for routes already used. Thus, we generalize the problem of obtaining disjoint paths internal to SoCs in a problem known as independent spanning trees, that can be briefly described as: Given a graph G, a set of spanning trees rooted at a vertex r in G is said vertex/edge independent if for each vertex v in G, $v \neq r$, the paths from r to v in any pair of trees are vertex/edge disjoint. If the connectivity of G is k, the problem comes down to construct k independent spanning trees with each vertex of the graph as the root of such trees. This issue remains open for general graphs with connectivity $k \geq 4$. However, it has been shown that for a hypercube of dimension k, denoted by Q_k, there are k spanning trees rooted at any arbitrary vertex of G. In this thesis we proposed an algorithm to generate k independent spanning trees on hypercubes in order to use them in developing efficient techniques for routing between distinct cores of SoCs, as well as *multi-core* processors. Among the contributions of this thesis, one can be emphasized, the reduced consumption of computational resources used by the proposed algorithm: memory and processing time; as well as the *ECUBE* modified algorithm to build disjoint routes without requiring the entire independent spanning trees construction. The proposed algorithm behaves similarly to the algorithms commonly used in routing in *NoCs*, as shown by the use of the *NoCs* simulator: *Noxim*. The construction of disjoint spanning trees tend to follow one of two objectives: efficient construction and optimum height. This work is focused on efficient construction, and the optimum height was an inherited result of the proposed construction method.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC	*Application Specific Integrated Circuit*	circuito integrado de aplicação específica
CAD	*Computer Aided Design*	projeto assistido por computador
DSP	*Digital Signal Processing*	processamento digital de sinais
FPGA	*Field Programmable Gate Array*	arranjo de portas programável em campo
GALS	*Globally Asynchronous Locally Synchronous*	globalmente assíncrono localmente síncrono
GOPS	*Giga Operations Per Second*	bilhões de operações por segundo
HDL	*Hardware Description Language*	linguagem de descrição de hardware
I/O	*Input/Output*	entrada/saída
IP	*Intelectual Property*	propriedade intelectual
IST	*Independent Spanning Tree*	árvore geradora independente
NIS	*Network Information Service*	serviço de informação de rede
NOC	*Network On Chip*	rede em chip
RAM	*Random Access Memory*	memória de acesso aleatório
ROM	*Read Only Memory*	memória somente de leitura
SOC	*System On Chip*	sistema em chip
VHDL	*VHSIC Description Language*	linguagem de descrição VHSIC
VHSIC	*Very-High-Speed Integrated Circuit*	circuito integrado de velocidade muito alta

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

1.1 Delimitação do Tema

Systems-on-chip (*SoCs*) são sistemas compostos contidos em um único substrato de silício. Por exemplo, um *SoC* para uma aplicação de telecomunicações pode conter um microprocessador, um processador de sinal digital (*DSP*), memória de acesso aleatório (*RAM*) e memória somente de leitura (*ROM*). Os *SoCs* foram introduzidos nas metodologias de projeto para atender à crescente demanda por aplicações complexas que requerem um grande poder computacional para sua execução. Tais aplicações se tornaram comuns devido, principalmente, à evolução da indústria de semicondutores e à capacidade de integração prevista pela lei de Moore. A lei de Moore não é uma lei da física e sim uma observação empírica a qual descreve tendências no progresso da tecnologia de produção de semicondutores. Em seu artigo original[82], Moore afirma que a complexidade dos circuitos integrados dobra a cada ano. Entenda-se por complexidade o número de componentes que podem ser implementados em uma área definida de circuito integrado, não somente transistores. Em 1975 Moore modificou sua observação para a ocorrência do dobro da complexidade a cada dezoito meses e não a cada ano, como afirmado anteriormente.

Apesar de não ser uma lei da física e sim descrever uma tendência, a lei de Moore pode se extinguir abruptamente por aspectos físicos. Em 2002 quando a capacidade de miniaturização de transistores ficava em torno de 100 nanômetros previa-se que os problemas ocorreriam quando se atingisse o patamar de 40 nanômetros [73] devido em parte a um efeito termodinâmico conhecido como ruído de Johnson-Nyquist. O efeito é ocasionado devido à dissipação de potência e ao teorema de equipartição de energia dos sistemas termodinâmicos [73]. Tal efeito não impõe limites quanto ao tamanho possível dos transistores, mas sim à densidade de

integração respeitando-se os limites de dissipação máxima em tais circuitos.

Uma maneira de contornar esse problema seria parar de aumentar a densidade de integração, representada pela Lei de Moore, ou parar de aumentar a frequência de *clock*.

Em vista disso, os projetistas passaram de uma visão computacional centrada unicamente no projeto de processadores, para uma visão voltada à comunicação, devido à relação entre o atraso na interconexão *versus* o atraso das tecnologias de portas lógicas. Pela utilização de uma rede estruturada pode-se obter parâmetros elétricos bem controlados que eliminam iterações de tempo e permitem a utilização de circuitos de alto desempenho para reduzir a latência e aumentar a largura de banda[31]. Problemas como integridade do sinal, devido à interferência de ruído, assim como flutuações no fornecimento de energia podem ocasionar erros na transmissão de sinais.

Em eletrônica um núcleo de propriedade intelectual, núcleo *IP* ou bloco *IP* é uma unidade reutilizável de lógica, célula ou projeto de circuito integrado cuja propriedade intelectual é detida por uma parte [1]. Núcleos *IP* podem ser licenciados para outras partes ou podem ser de uso e propriedade exclusiva de uma única. O termo é derivado do licenciamento de patentes e direitos autorais de código fonte, assim como direitos de propriedade intelectual existentes na área de projetos de circuitos. Núcleos *IP* podem ser utilizados como blocos de construção dentro de projetos de circuitos ASIC (*Application Specific Integrated Circuit*) ou projetos de lógica FPGA (*Field Programmable Gate Array*).

Até então, eram utilizados meios compartilhados, barramentos arbitrários e ligações ponto a ponto para se conectar blocos distintos em um único *SoC* . Tais abordagens tendem a ser adotadas de maneira descentralizada sem a utilização de padrões definidos, prejudicando assim, a produtividade quando o número de blocos *IP* aumenta.

A demanda de uma maneira eficiente de comunicação dentro de um *SoC* fez com que surgissem as chamadas *Networks-on-Chip* (NoCs) [17, 18, 46, 75, 2, 101]. Tais redes baseiam-se em um tipo de comunicação mais estruturada como a utilizada na maioria das redes existentes atualmente, sendo a Internet um dos exemplos mais conhecidos [54].

Assim como na Internet, a comunicação em uma *NoC* é usualmente baseada em pacotes.

Dessa forma, os pacotes trafegam pelos roteadores, os quais conectam dois ou mais núcleos de forma a prover um meio de comunicação dentre eles. Embora similar à Internet, as *NoCs* são simplificadas e otimizadas de modo a atender uma série de restrições quando implementadas em um *SoC*, tais como: espaço limitado e baixo consumo de energia.

Assim como os diferentes blocos *IPs* utilizam-se das *NoCs* para sua comunicação, núcleos distintos também o fazem. Os blocos *IPs* são considerados heterogêneos enquanto a maioria dos processadores *multi-core* (múltiplos núcleos) utiliza núcleos idênticos, homogêneos. No nicho de sistemas embarcados, *multi-cores* são representados por várias arquiteturas e dispositivos diferentes, como por exemplo: aparelhos celulares, computadores de bordo, etc. Destes processadores alguns dos mais conhecidos possuem dois, quatro e oito núcleos, baseados em multi-processamento simétrico ou arquiteturas de cache compartilhada. Nesse segmento também se incluem processadores de vários núcleos que implementam uma forma de memória compartilhada, memória distribuída ou uma combinação dos dois tipos.

Embora o x86 seja provavelmente a arquitetura mais popular, ela representa somente uma ínfima parte dos processadores destinados ao setor de sistemas embarcados. Outros exemplos nessa área incluem processadores como o Freescale 8641D (dois núcleos), o MPCore Cortex-A9 ARM (com 2 a 4 núcleos), Plurality's Hypercore (capaz de suportar entre 16 e 256 núcleos) e o processador Tilera's TilePro64 (64 núcleos) [16, 60, 79].

Os projetistas atuais estão enfrentando problemas como queda de desempenho resultante da transição em seus projetos embarcados com processadores de núcleo único para um processador de múltiplos núcleos. Independentemente do número de núcleos, os projetistas devem resolver os problemas associados à memória compartilhada como também a outros recursos compartilhados pelo sistema. Inúmeros esforços têm sido empregados na compreensão e resolução de problemas de desempenho relacionados aos *multi-cores*. Problemas estes que incluem a alta taxa de sincronização e comunicação entre os núcleos [87, 104].

A tecnologia *multi-core* embarcada inclui *SoCs* com uma vasta combinação de processadores homogêneos e heterogêneos destinada a aplicações específicas. Muitos *SoCs* são proprietários e estão profundamente intrínsecos em dispositivos de usuário final. No entanto, pro-

dutos *SoC* como Freescale's QorIQ P4080, Texas Instrument's OMAP3503 e Cavium Network's Octeon CN38XX estão comercialmente disponíveis para uma ampla variedade de aplicações embarcadas [79].

Aplicando o corolário da Lei de Moore para *multi-core*, estima-se a existência de processadores com mil núcleos dentro de uma década [6]. Sistemas com milhares de núcleos apresentam desafios significativos na área de arquitetura de computadores como latência, dissipação de potência, comunicação eficiente, dentre outros. Em vista disso, tornam-se necessárias novas abordagens para solucionar tais desafios.

1.2 Problemática

Algumas definições são necessárias para o entendimento do que é descrito a seguir: Um grafo G é um par ordenado de conjuntos disjuntos (V, E) tal que E é um subconjunto do conjunto $V^{(2)}$ de pares não ordenados de V. O conjunto V é o conjunto de vértices e E o conjunto de arestas. Se G é um grafo, então $V = V(G)$ é o conjunto de vértices de G e $E = E(G)$ é o conjunto de arestas. Uma aresta x, y une os vértices x e y e é denotada por xy. Então xy e yx são a mesma aresta. Se $xy \in E(G)$, então x e y são vértices adjacentes ou vizinhos de G, e os vértices x e y são incidentes à aresta xy. Duas arestas são adjacentes se elas tem exatamente um vértice extremo em comum, ou seja, são adjacentes se incidem no mesmo vértice.

Um caminho é uma sequência de vértices $v_0, v_1, v_2, ..., v_n$ que descreve o percurso do vértice i para o vértice j. Um grafo G é dito conexo, se dois vértices quaisquer podem ser unidos por um caminho, caso contrário é dito desconexo. Um grafo é dito completo se para todo $i, j \in V$, com $i \neq j$, $ij \in E$.

Um ciclo em um grafo é um subconjunto do conjunto de arestas que representam um caminho de tal forma que o primeiro vértice do caminho corresponde ao último vértice.

Um componente de um grafo G é o subgrafo maximal conexo. Se G é conexo e para algum conjunto de vértices e arestas W, $G - W$ é desconexo, então é dito que W separa G.

Se em $G - W$ dois vértices s e t pertencem a dois componentes diferentes, então W separa s de t.

Para $k \geq 2$, é dito que um grafo G é k-conexo se G é um grafo completo K_{k+1} ou tem ao menos $k + 2$ vértices e nenhum conjunto de $k - 1$ vértices os separa. De forma similar, para $k \geq 2$, um grafo G é k-arco-conexo se ele tem ao menos dois vértices e nenhum conjunto de no máximo $k-1$ arestas o separa. Um grafo conexo é também dito ser 1-conexo e 1-arco-conexo. O máximo valor de k para qual um grafo conexo G é dito k-conexo é a conectividade de G, denotada por $\kappa(G)$. Se G é desconexo então $\kappa(G) = 0$. A arco conectividade $\lambda(G)$ é definida de forma análoga [20].

Qualquer sistema multi-processado deve permitir que os processadores troquem dados entre todos os vértices que compõem o conjunto. Dado o seguinte modelo: Sejam A e B vértices distintos quaisquer do hipercubo, considere o problema de envio de dados do vértice A ao vértice B. A forma pela qual isso é feito, é movendo-se os dados através de um caminho de A para B passando por uma quantidade, de preferência pequena, de vértices intermediários. No hipercubo os vértices são rotulados utilizando-se a representação binária de seus índices no seguinte intervalo: $(0, ..., n - 1)$, onde n é o número de vértices do hipercubo. Por exemplo, no hipercubo de oito vértices Q_3, o vértice 2 tem como rótulo os *bits* 010.

Por definição, o comprimento de um caminho entre dois vértices é o número de arestas que compõem o caminho. No caso do hipercubo, para se chegar a A partindo-se de B, basta atravessar sucessivamente os vértices cujos rótulos são obtidos modificando-se os *bits* de A, um a um, a fim de transformar A em B, supondo-se que A e B diferem apenas em i *bits*, ou seja, sua distância de *Hamming* tem comprimento máximo i. A distância de *Hamming* entre duas palavras, de mesmo comprimento, é o número de lugares onde elas diferem, ou seja, o número de posições onde uma tem o valor 0 e outra o valor 1 em uma sequência binária.

Para otimizar a comunicação entre os vértices que compõem o sistema, deve-se utilizar uma topologia que minimize a distância entre eles. Dentre as topologias utilizadas e implementadas comercialmente tem-se o hipercubo [61, 85]. A topologia hipercubo possui várias características atraentes. Primeiramente, ela é homogênea ou vértice-simétrica no sentido de

que o sistema parece o mesmo visto de qualquer vértice. Dessa forma não há bordas ou limites onde os vértices precisem ser tratados como casos especiais.

Nessa topologia são empregadas $n.N/2$ ligações para se conectar $N = 2^n$ vértices e cada vértice possui n interligações para gerenciar. Vértices não conectados diretamente devem se comunicar por meio de mensagens enviadas através de vértices intermediários no modo *store-and-forward* (armazena e encaminha). Além disso, outras estruturas computacionais úteis, principalmente malhas de dimensões arbitrárias, podem ser embutidas em um hipercubo de tal maneira que vértices adjacentes na estrutura original também sejam adjacentes no hipercubo. Vários algoritmos têm sido propostos para embutir topologias importantes dentro de hipercubos como malhas retangulares [23, 68], árvores [33, 55, 68] e pirâmides [76, 126].

A ideia de se utilizar a topologia hipercubo como uma forma de organização de multiprocessadores não é nova. Pelo uso de centenas de microprocessadores de baixo custo é possível se ter um super computador ao alcance de um único usuário. Em vista disso, o fabricante do iPSC chamou seu produto de "supercomputador pessoal" [62].

Implementações comerciais de super computadores utilizando a topologia hipercubo surgiram na década de 1980, assim como o iPSC pode-se citar o Cosmic Cube [98]. Entretanto, conectar de forma eficiente vários processadores sempre representou um problema.

As *NoCs* provêem uma possível solução para esse problema, pela utilização de barramentos conectados por roteadores que trocam pacotes de uma forma similar às redes tradicionais [17, 31, 47, 53].

FPGAs (*Field Programmable Gate Arrays*) evoluíram o bastante para serem capazes de implementar *NoCs* sofisticadas para interconexão de blocos *IPs* ainda mais sofisticados. Dentre as vantagens da utilização de FPGAs para *NoCs* pode-se apontar o fato do atraso na transmissão de sinais por fios e barramentos comuns. Utilizando a mesma tecnologia de portas lógicas tanto para implementação de blocos *IPs* quanto para a comunicação entre eles, pode-se diminuir consideravelmente a latência do sistema como um todo. No projeto com FPGAs, diferentemente de ASICs, deve se levar em conta a quantidade de portas lógicas utilizadas. Nos ASICs, cujo projeto é feito para atender uma necessidade específica, a quantidade

de fios pode ser mensurada de modo a se beneficiar na fase de projeto de características como dissipação de potência e interferência eletromagnética. Os FPGAs, de forma contrária, não são customizáveis apenas configuráveis. Logo, o que não for utilizado de lógica no projeto continuará na estrutura do FPGA sem uso, caracterizando desperdício. Dessa forma, topologias aparentemente atraentes como malhas, podem ocupar tanta área como topologias mais complexas como hipercubos [95].

Apesar da maioria das *NoCs* implementadas comercialmente utilizar uma técnica conhecida como *wormhole*, na qual um vértice estabelece uma rota direta até o vértice alvo, ainda faz-se necessário evitar a competição por caminhos já utilizados. Dessa forma generalizamos o problema em se obter caminhos disjuntos internos aos *SoCs* , em um problema de grafos conhecido como árvores geradoras independentes, que pode ser descrito resumidamente como: Dado um grafo G, um conjunto de árvores geradoras enraizadas em um vértice r em G é dito vértice/aresta independente se, para cada vértice v em G, $v \neq r$, os caminhos de r a v em qualquer par de árvores são internamente vértice/aresta disjuntos. Foi provado que a conjectura de vértices implica na conjectura de arestas. Se a conectividade de G é k, o problema resume-se à construção de k árvores geradoras independentes com cada vértice do grafo como raiz de tais árvores. Esse problema permanece em aberto para grafos em geral com conectividade superior a quatro [72].

1.3 Objetivos e Contribuições

O principal objetivo deste trabalho é apresentar um método eficiente para a geração de caminhos disjuntos que necessite de poucos recursos computacionais, como memória e capacidade de processamento, visto que sua finalidade é a aplicação em sistemas embarcados.

O problema em se encontrar caminhos disjuntos em grafos k-conexos, principalmente em hipercubos, o qual é o foco do trabalho, não é novidade. Entretanto, os algoritmos já apresentados para a solução do problema exigem uma quantidade de memória superior ao apresentado neste trabalho, assim como uma computação mais complexa do que a requerida

pelo método proposto.

A construção de árvores geradoras disjuntas tende a seguir um dos dois objetivos: construção eficiente e altura ótima [109]. Este trabalho tem o foco na construção eficiente, sendo que a altura ótima foi um resultado inerente ao método de construção proposto.

Em relação ao roteamento, para se diminuir o impacto de falhas é proposta neste trabalho a utilização de um algoritmo de roteamento semi-adaptativo. Tal algoritmo utiliza k caminhos disjuntos alternadamente, sendo k tolerante a falhas. Sua implementação é simples não requerendo uma lógica complexa e portanto consumindo uma menor área no projeto de *Networks-on-Chip* (apêndice D e E). Através da utilização de um simulador de *NoCs* mostramos a eficiência do algoritmo proposto, mesmo na sua utilização em uma topologia de malha. O algoritmo é tão eficiente quanto os demais algoritmos frequentemente utilizados em *NoCs*, além de apresentar uma menor taxa de colisão quando utilizado na topologia hipercubo.

1.4 Organização do Texto

Este trabalho é organizado como descrito a seguir. O capítulo 2 apresenta uma descrição formal do problema do ponto de vista da área de grafos, juntamente com uma introdução mais detalhada de *SoCs* e *NoCs*, além dos principais trabalhos relacionados à geração de árvores geradoras independentes. No capítulo 3 são descritos os algoritmos propostos, seguido pelo capítulo 4 no qual são apresentados os resultados da aplicação do algoritmo em simulações. O trabalho é finalizado pelo capítulo 5, onde as conclusões e trabalhos futuros são apresentados.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta os principais trabalhos da literatura relacionados à construção de ávores geradoras disjuntas, precedidos pela fundamentação teórica necessária para seu entendimento. São apresentados também os *Systems-on-chip* (SoCs) e as *Networks-on-chip* (NoCs), os quais representam uma aplicação prática do trabalho proposto.

2.1 Fundamentação Teórica

Antes de discutir as principais abordagens encontradas na literatura para geração de caminhos disjuntos sobre grafos $k-$conexos, faz-se necessária a definição de alguns termos utilizados ao longo deste trabalho. São também utilizadas algumas definições básicas de grafos, para o leitor que não possua familiaridade com o assunto sugerem-se as seguintes referências [20, 22, 34, 50].

2.1.1 Hipercubo

O conjunto de arestas incidentes a um vértice $x \in V(G)$, a vizinhança de x, é denotado por $\Gamma(x)$. O grau de x, $d(x)$, é o número de vértices contidos na vizinhança de x, logo $d(x) = |\Gamma(x)|$. O grau mínimo dos vértices de um grafo G é denotado por $\delta(G)$ e o grau máximo por $\Delta(G)$. Se todos os vértices de G tem o mesmo grau, então G é dito k-regular ou regular de grau k. Um grafo é dito regular se ele é k-regular para algum valor de k.

O hipercubo, usualmente denotado por Q_k ou 2^k é um grafo k-regular em que cada vértice é denotado por k símbolos $\epsilon_1, ..., \epsilon_k$ no qual $\epsilon_i = 0$ ou 1 e dois vértices são adjacentes se os identificadores diferem exatamente em um único símbolo (exemplo ilustrado na figura 2.1). Há muito tempo hipercubos têm sido utilizados na área de redes e arquitetura de computadores,

devido a sua simplicidade para o desenvolvimento de algoritmos [4, 11, 28, 49, 66, 70, 94, 107, 118] e por suas atrativas características como a distância logarítmica de seus vértices.

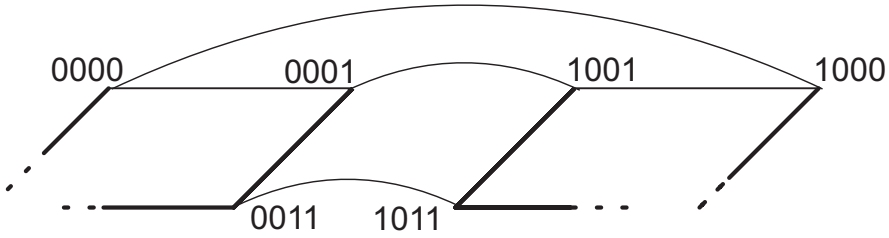


Figura 2.1: Dois vértices são adjacentes no hipercubo se os símbolos diferem exatamente em uma única coordenada.

Um grafo G possui um caminho Hamiltoniano se, e somente se, existe um caminho em G pelo qual cada vértice é visitado uma única vez. Um ciclo Hamiltoniano é um ciclo que forma um caminho Hamiltoniano.

Todo hipercubo é um grafo bipartido. É fato conhecido que todo hipercubo Q_n é Hamiltoniano para $n > 1$. Além disso, existe um caminho Hamiltoniano entre os vértices u e v, se e somente se u e v fazem parte de partições diferentes do grafo bipartido do hipercubo.

A propriedade Hamiltoniana do hipercubo está fortemente relacionada à teoria do código de Gray[119]. Um código Gray de *m-bits*, denotado por G_m, designa uma sequência entre todos os números binários de *m-bits*. G_1 é definida como (0, 1) e para $m > 1$, G_m é definido recursivamente em termos de G_{m-1} como $(0G_{m-1}, 1{G_{m-1}}^r)$ onde ${G_{m-1}}^r$ é a ordem reversa de G_{m-1} e $0G_{m-1}$ $(1{G_{m-1}}^r)$ representa a prefixação de cada número binário em G_{m-1} $({G_{m-1}}^r)$ com 0 (1) [56].

Por exemplo, a sequência de números binários: $(000, 001, 011, 010, 110, 111, 101, 100)$ é um código Gray de 3 *bits* representando um caminho Hamiltoniano em um hipercubo de 2^3 vértices (3-cubo), começando com o vértice 000 e terminando com o vértice 100.

2.1.2 Matriz de Adjacências

Seja $G = (V, E)$ um grafo com um conjunto de vértices $V = \{v_1, \ldots, v_n\}$ e um conjunto de arestas E, a matriz de adjacências $M_G = (m_{ij})$, de mesma ordem do grafo G, possui valor não nulo na posição i, j (geralmente 1 quando o grafo não for valorado) se e somente se v_i e v_j possuem uma aresta entre eles, caso contrário o valor da coordenada i, j é nulo. Todas as posições m_{ij} da matriz que satisfazem a condição $i = j$ possuem valor nulo. A figura 2.2 $b)$ ilustra a matriz de adjacências correspondente ao hipercubo Q_3 com 8 vértices, ilustrado na figura 2.2 $a)$. Na figura 2.2 $c)$ é apresentada uma representação da matriz de adjacências de mais fácil visualização que é comumente utilizada neste trabalho para facilitar a distinção de padrões aqui apresentados. A representação decimal, equivalente aos identificadores binários dos vértices nos hipercubos e árvores deste trabalho, também é utilizada para facilitar a visualização das matrizes.

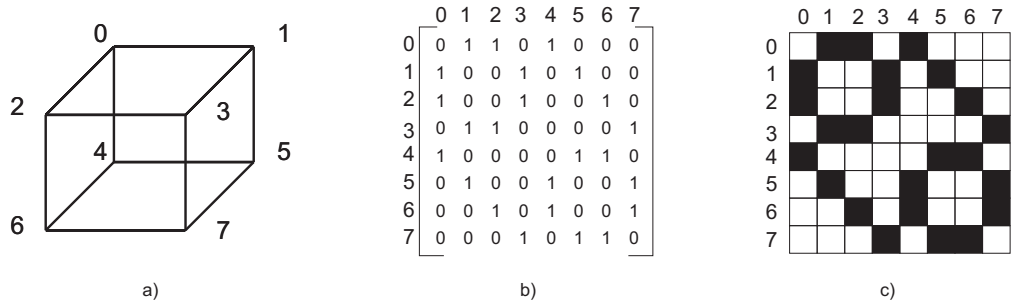


Figura 2.2: Matriz de adjacências do hipercubo Q_3

O hipercubo n-dimensional Q_n pode ser definido recursivamente como:

$$Q_1 = K_2 \tag{2.1}$$

$$Q_n = K_2 \times Q_{n-1} \tag{2.2}$$

Sendo que K_2 é grafo completo com dois vértices.

Como a matriz de adjacência de K_2 é dada por:

$$R = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.3}$$

A matriz de adjacências de Q_n pode ser expressada como:

$$A = \sum_{i=0}^{k} I^i \otimes R \otimes I^{k-i} \tag{2.4}$$

onde I denota a matriz identidade e $\otimes$ é o Produto Tensorial que é apresentado a seguir. R é a matriz de adjacências do grafo completo K_2, equivalente à matriz do hipercubo Q_2 e também é referenciado na literatura de mecânica quântica, em se tratando de hipercubos, como a matriz de Pauli σ_x [27, 81].

2.1.3 Produto Tensorial

Sejam V, W dois espaços vetoriais finitos sobre $\mathbb{R}$, o produto tensorial define um novo espaço vetorial, denotado por $V \otimes W$, que segue duas propriedades:

— se $v \in V$ e $w \in W$ então existe um produto $v \otimes w \in V \otimes W$

— o produto formado é bilinear:

$$(\lambda v_1 + \mu v_2) \otimes w = \lambda v_1 \otimes w + \mu v_2 \otimes w$$

$$v \otimes (\lambda w_1 + \mu w_2) = \lambda v \otimes w_1 + \mu v \otimes w_2$$

Ou seja, o produto tensorial de dois espaços vetoriais V e W é uma maneira de criar um novo espaço vetorial, de forma análoga à multiplicação de inteiros. Por exemplo:

$$\mathbb{R}^n \otimes \mathbb{R}^k \approxeq \mathbb{R}^{nk} \tag{2.6}$$

Em particular:

$$r \otimes \mathbb{R}^n \approxeq \mathbb{R}^n \tag{2.7}$$

sendo r um escalar.

O produto tensorial é uma ferramenta útil que permite juntar espaços vetoriais para formar espaços vetoriais maiores. Especificamente para este trabalho, o produto tensorial auxilia a análise de padrões de grafos n-dimensionais reduzindo o escopo analisado a uma forma planar de duas dimensões, a matriz de adjacências. A fórmula 2.3 mostra a maneira pela qual o produto tensorial é calculado, tal representação é também conhecida como produto de *Kronecker* [42].

$$U \otimes V = \begin{bmatrix} u_{1,1}V & u_{1,2}V & \dots \\ u_{2,1}V & u_{2,2}V & \\ \vdots & & \ddots \end{bmatrix} = \begin{bmatrix} u_{1,1}v_{1,1} & u_{1,1}v_{1,2} & \dots & u_{1,2}v_{1,1} & u_{1,2}v_{1,2} \\ u_{1,1}v_{2,1} & u_{1,1}v_{2,2} & & u_{1,2}v_{2,1} & u_{1,2}v_{2,2} \\ \vdots & & \ddots & & \\ u_{2,1}v_{1,1} & u_{2,1}v_{1,2} \dots & & & \\ u_{2,1}v_{2,1} & u_{2,1}v_{2,2} & & & \\ \vdots & & & & \end{bmatrix}$$

Figura 2.3: Cálculo do produto tensorial.

A figura 2.4 mostra a construção da matriz de adjacências de Q_3 utilizando-se produto tensorial.

$$A_{Q_k} = \sum_{i=0}^{k} I^i \otimes R \otimes I^{k-i}$$

$$A_{Q_3} = \sum_{i=0}^{3} I^i \otimes R \otimes I^{3-i}$$

$$A_{Q_3} = R \otimes I \otimes I + I \otimes R \otimes I + I \otimes I \otimes R$$

$$R = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$R \otimes I = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$R \otimes I \otimes I = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad I \otimes R \otimes I = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad I \otimes I \otimes R = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$Q_3 = R \otimes I \otimes I + I \otimes R \otimes I + I \otimes I \otimes R = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Figura 2.4: Cálculo da matriz de adjacências de Q_3 utilizando produto tensorial

2.1.4 Caminhos Disjuntos

Um grafo $G' = (V', E')$ é um subgrafo de $G = (V, E)$ se e somente se, $V' \subset V$ e $E' \subset E$. Nesse caso, pode-se escrever $G' \subset G$. Se G' contém todas as arestas de G que unem dois vértices em V', então G' é dito ser um subgrafo induzido por V' e é denotado por $G[V']$.

Uma árvore é um grafo conexo e sem ciclos. Uma árvore geradora ou espalhadora do grafo $G = (V, E)$ é uma árvore da forma $T = (V, E')$ com $E' \subset E$, ou seja, um subconjunto das arestas de G que cobrem todos os vértices do grafo, é conexo e não forma ciclos.

Dado um grafo G, um conjunto de árvores geradoras enraizadas em um vértice r em G é chamado independente se, para cada vértice $v \in G$, $v \neq r$, os caminhos de r a v em qualquer

par de árvores são disjuntos.

O problema de árvores geradoras independentes é caracterizado pela tentativa de se construir pares de árvores vértice/aresta independentes e tem aplicações como transmissão de dados e protocolos de comunicação confiável. Por exemplo, uma árvore geradora abrangendo um grafo subjacente de uma rede pode ser visto como um esquema de transmissão para comunicação de dados e tolerância a falhas. Tal esquema pode ser implementado pelo envio de k cópias da mensagem ao longo das k árvores geradoras independentes enraizadas no vértice de origem [13](figura 2.5).

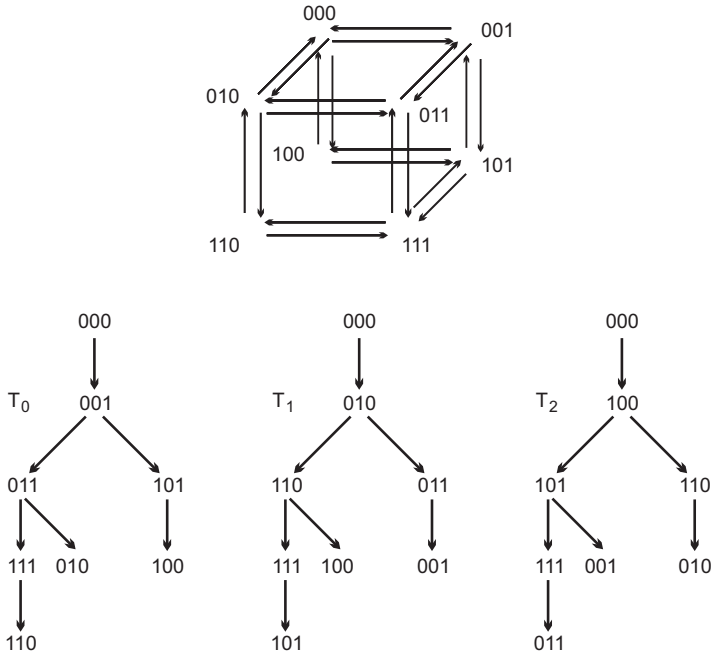


Figura 2.5: Envio de mensagens utilizando as k árvores geradoras sobre o hipercubo Q_3.

Para outras aplicações consulte [24] para o problema de transmissão multi-vértice, [112] para difusão um-para-todos e [14] para difusão confiável e segura de mensagens.

O problema de construção de múltiplas árvores geradoras independentes é considerado um problema complexo para grafos arbitrários. Entretanto, existe uma conjectura sobre árvores geradoras independentes que é: para qualquer grafo k-conexo G e cada vértice v de G como raiz, existem k árvores geradoras independentes (k-IST Independent Spanning Trees) sobre G. A conjectura foi provada apenas para grafos k-conexos com $k \leq 4$ e continua em aberto para

grafos arbitrários nos quais $k > 4$ [26, 63, 125].

Itai e Rodeh [63] provaram para k=2. Para $k = 3$, foi provado independentemente por Zehavi e Itai [125], e por Cheriyan e Maheshwari [26].

Em [36] Edmonds provou o teorema 2.1.1 postulado a seguir.

Teorema 2.1.1. *Seja G um grafo direcionado e r um vértice de G. Suponha que para qualquer vértice $v(\neq r)$ de G, existam k caminhos arco-disjuntos de r a v em G. Então existem k árvores geradoras arco-disjuntas com raiz em r em G.*

Em [97] Frank colocou a seguinte conjectura (2.1.2), a qual é a versão com vértices do teorema de Edmonds.

Conjectura 2.1.2. *Seja G um grafo direcionado e r um vértice de G. Suponha que para qualquer vértice $v(\neq r)$ existam k caminhos vértice-disjuntos de r para G. Então existem k árvores geradoras independentes com raiz r em G.*

Whitty apresentou uma generalização da conjectura 2.1.2 para grafos direcionados, sendo que para grafos direcionados gerais as conjecturas 2.1.2 e 2.1.3 são equivalentes [58].

Conjectura 2.1.3. *Seja G um grafo direcionado k-vértice conexo, então existem k-árvores geradoras independentes com raiz em qualquer vértice de G.*

Para $k = 2$, Whitty [116] provou esta conjectura. Entretanto Huck [57] refutou a conjectura 2.1.3 (por consequência a 2.1.2) para grafos gerais com $k \geq 3$, embora tenha provado que a conjectura 2.1.2 permanece para grafos acíclicos.

Entretanto, vários resultados são conhecidos para algumas classes de grafos, além do hipercubo, tais como: grafos produto [86], grafos planares [59], anéis cordais [63], grafos de De Bruijn e Kautz [45, 51], hiper-estrelas dobradas [93], tórus multi dimensionais [110] e grafos circulantes recursivos [123].

2.1.5 Algoritmos de Roteamento Avaliados

Um algoritmo de roteamento define qual caminho ou caminhos de rede são permitidos para cada pacote. Idealmente, o algoritmo de roteamento fornece caminhos mais curtos para todos

os pacotes de tal maneira que o tráfego seja uniformemente distribuído de modo a minimizar a contenção da rede.

Caminhos que tenham um número ilimitado de saltos a partir do vértice fonte, podem resultar na situação na qual os pacotes nunca atinjam seu destino. Esta situação é conhecida como *livelock*. De forma similar, caminhos que façam com que um conjunto de pacotes fiquem esperando indefinidamente pela liberação de recursos (*links* ou *buffers*) reservados para outros pacotes, podem prevenir que os pacotes cheguem a seus destinos. Esta situação é conhecida como *deadlock*.

Deadlock surge devido ao fato dos recursos serem finitos e a probabilidade de sua ocorrência aumenta com o aumento do tráfego de rede e a diminuição da disponibilidade dos recursos (figura 2.6).

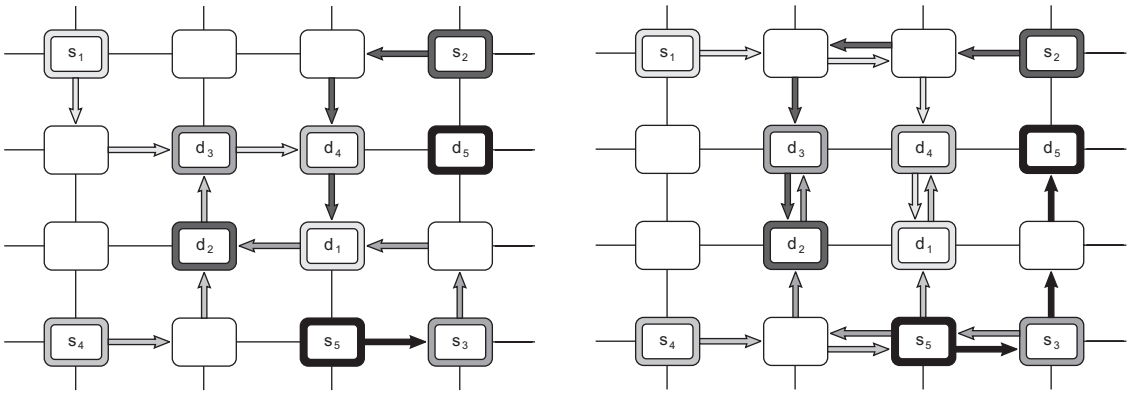


Figura 2.6: (a)*Deadlock* é formado a partir dos pacotes destinados a d_1 passando por d_4 bloqueando outros no mesmo conjunto que ocuparam completamente seus *buffers* de recursos requeridos a um salto de distância de seus destinos. (b)*Deadlock* é evitado usando *dimension-ordered routing (DOR). Neste caso os pacotes utilizam primeiro todo o deslocamento na dimensão* X *antes de percorrer a dimensão* Y *[54].*

Uma maneira simples de se proteger contra *livelock* é restringir o encaminhamento de pacotes de tal forma que somente caminhos mínimos sejam permitidos. Outra maneira, menos restritiva, é permitir que o caminho seja composto por um número limitado de saltos. A forma menos restrita tem a vantagem de consumir o mínimo de largura de banda, mas evita que os pacotes possam usar caminhos não mínimos alternativos em caso de contenção ou falhas.

Entretanto, *deadlocks* são mais difíceis de serem evitados. Dentre as estratégias utilizadas duas são comuns na prática: prevenção e detecção.

Na prevenção de *deadlocks* o algoritmo de roteamento restringe os caminhos permitidos àqueles no qual o estado global da rede é livre de *deadlocks*. Uma maneira comum para isso é estabelecer uma ordem entre os recursos e conceder tais recursos a pacotes em alguma ordem parcial ou total de modo que não sejam formadas dependências cíclicas sob esses recursos. Com isso, permite-se que seja fornecido um caminho de fuga para os pacotes não importando onde eles estejam na rede, evitando assim um estado de *deadlock* [30, 48].

Na recuperação de *deadlock*, recursos são concedidos a pacotes sem levar em conta a prevenção de *deadlock*. Caso um *deadlock* seja detectado, um ou mais pacotes são removidos dos recursos em *deadlock*, quer sendo redirecionados para recursos especiais de recuperação de *deadlocks* ou simplesmente descartando os pacotes.

Os tipos de roteamento são classificados como adaptativos e determinísticos. Em um roteamento determinístico, um pacote atravessa a rede seguindo um caminho pré-determinado fixo entre a origem e o destino, enquanto que em um roteamento adaptativo o pacote pode atravessar uma série de caminhos alternativos.

Dentre os algoritmos determinísticos dois deles são utilizados neste trabalho, o *ECUBE* [77, 106] e o *DOR*(*dimension-ordered routing*) [30].

No roteamento *ECUBE* o cabeçalho do pacote carrega o endereço do vértice destino d. Quando o vértice v do hipercubo recebe um pacote, o algoritmo *ECUBE* faz o seguinte cálculo $c = d \oplus v$. Se $c = 0$, o pacote está em seu destino final e é encaminhado para o processador local. Caso contrário, o pacote é encaminhado para o canal de saída na k -ésima dimensão, onde k é a posição do 1 mais a direita (alternativamente, mais a esquerda) em c.

No roteamento *DOR*(*dimension-order routing*), cada pacote é encaminhado em uma dimensão de cada vez, chegando à coordenada correta em cada dimensão antes de prosseguir para a próxima.

Algoritmos adaptativos são mais complexos demandando mais lógica e portanto ocupando uma maior área em um circuito integrado. Portanto, não são adequados para *Networks-on-*

Chip. Por esta razão algoritmos determinísticos como *ECUBE* e *DOR* são utilizados [54]. A figura 2.6 $a)$ mostra um exemplo de *deadlock* e a figura 2.6 $b)$ como evitá-lo utilizando o algoritmo *DOR*.

Porém, algoritmos determinísticos, os quais sempre utilizam as mesmas rotas, são suscetíveis a falhas pois não se adaptam a novas rotas no caso de vértices problemáticos (falhos ou com grande latência).

2.1.6 System-on-Chip e Network-on-Chip

Durante as últimas duas décadas, projetistas de circuitos integrados têm aumentado a capacidade de integração e o consumo de energia em seus projetos, que tem permitido alavancar rapidamente o desempenho dos circuitos produzidos. Entretanto, a maioria dos circuitos integrados atuais possuem severas restrições quanto ao consumo de energia e portanto são limitados em sua escala de integração [99]. Moore previu o crescimento exponencial do número de transistores em um circuito integrado, porém, explicar como tal escala de integração afetaria as características físicas dos dispositivos só seria feito na década seguinte. Em 1974 Dennard [32] publicou um artigo endereçando tais questões.

Seguindo-se a teoria de escala de integração de Dennard, os dispositivos se tornariam menores permitindo mais transistores em uma mesma área. Com isso, a capacitância C seria reduzida, pois é calculada pelo produto das dimensões pelo coeficiente dielétrico dividido pela espessura do óxido de porta. Logo a carga Q a ser removida para mudar o estado de uma porta seria diminuída visto que $Q = CV$. A intensidade de corrente I também seria reduzida, logo o atraso de porta diminuiria visto que o atraso D é dado por $D = Q/I$. Por consequência a energia necessária para a transição de porta, dada por CV^2 também seria decrescida. Sendo assim, diminuindo-se o tamanho dos transistores, proporcionando uma maior escala de integração com mais transistores por mm^2, teria-se uma diminuição do atraso de porta e da energia necessária para troca de estado. Seguindo-se a escala de Dennard, mantendo-se a densidade de potência constante, a área de lógica ocupada seria reduzida e com isso a energia de transição seria reduzida com a frequência, por outro lado, aumentada, resultando em uma diminuição de potência por porta. Então somente aumentando-se a escala de integração, obteriam-se mais transições de portas por segundo, porém mantendo-se a mesma área e consumo de potência anteriores. Logo, a integração por si só, é capaz de trazer um significante aumento de desempenho computacional para um mesmo consumo de potência.

No entanto, a densidade de potência tem aumentado continuamente como mostra a figura 2.7. A razão é uma combinação de projetistas não seguindo exatamente a escala constante

de integração além de criar modelos mais agressivos, aumentando assim o desempenho mais rapidamente do que Dennard havia previsto. A figura 2.7 mostra o crescimento da frequência de clock até o início dos anos 2000, dez vezes mais rápida do que o previsto pelas regras de Dennard. Tais estratégias de projeto não eram um problema na época, visto que o consumo de potência não era uma restrição. Com o aumento do uso de computadores portáteis e telefones inteligentes, os quais utilizam baterias, essa premissa mudou[99].

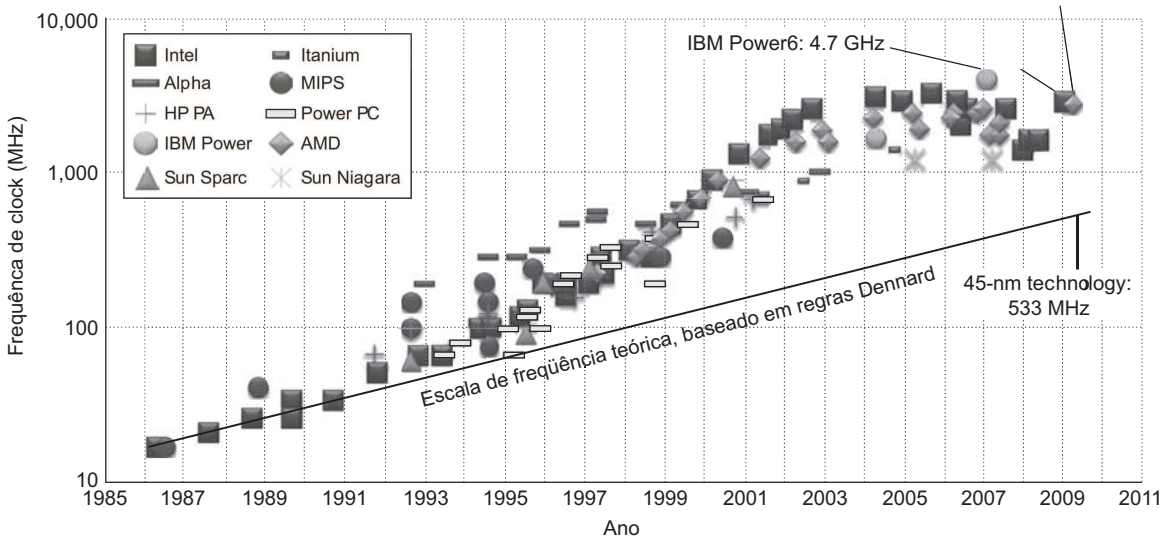


Figura 2.7: Estatísticas de frequência de clock de microprocessadores. A linha teórica de escala de frequência representa a aplicação das diretrizes de escala de Robert Dennard para o processador cronologicamente em primeiro lugar no gráfico, o que teria resultado em uma frequência de clock de 533MHz para 45 *nm* tecnologia. A indústria excedeu as previsões de Dennard em uma ordem de magnitude [91].

Árduas pesquisas na área de circuitos integrados têm mostrado que a melhor e talvez única alternativa para economizar energia seria reduzir o desperdício. Porém, a potência também é desperdiçada quando há desperdício de desempenho. Entende-se por desperdício de desempenho executar uma tarefa com um esforço desnecessário, sendo que, de outra forma a mesma tarefa poderia ser realizada com um esforço bem menor.

Devido às melhorias contínuas na tecnologia de silício pode-se integrar cada vez mais dispositivos em um único circuito integrado, porém com o revés de resultar em um vão de produtividade dos projetistas como mostrado na figura 2.8. Nessa figura, o eixo x expressa a progressão no tempo, enquanto o eixo y possui duas linhas. A linha contínua mostra a taxa de

crescimento da complexidade dos circuitos integrados, medida pelo número de transistores por circuito lógico, enquanto que a linha pontilhada mostra a produtividade do projetista, medida pelo número médio de transistores projetados por um engenheiro em um mês. O progresso da tecnologia de silício supera de longe a capacidade em se utilizar eficazmente esses transistores em projetos de curto prazo [91] como mostra a figura 2.8. Como o tempo de vida de um produto tende a continuar encolhendo, o tempo de comercialização torna-se uma restrição de projeto.

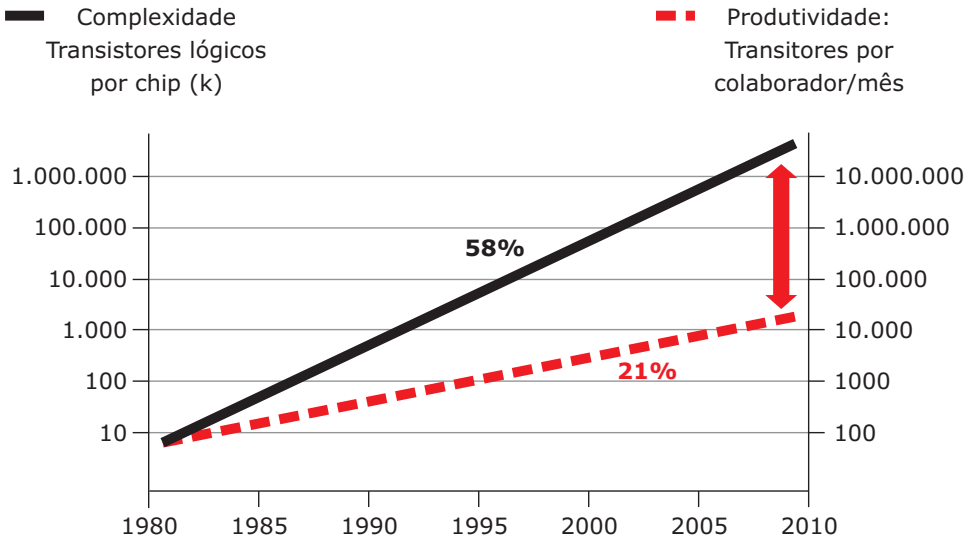


Figura 2.8: Gráfico de quantidade de transistores por circuito integrado versus número médio de transistores projetados por colaborador/mês. [91]

A figura 2.9 mostra a evolução dos requisitos em bilhões de operações por segundo (GOPS) de aplicações separadas em quatro categorias: vídeo, áudio/voz, gráficos e comunicação/reconhecimento.

Requisitos de alto desempenho refletem um alto consumo de potência, por isso, cada vez mais têm-se optado pela utilização de vários circuitos integrados dedicados, mais simples, do que um único circuito integrado de propósito geral.

Dessa forma, utilizando-se diferentes circuitos integrados, conhecidos como blocos *IP* (Intelectual Property) componentes de um *SoC* (System-on-chip), ao invés de um único circuito integrado de propósito geral, é possível se obter soluções integradas para os mais diversos problemas de projeto nas áreas de telecomunicações e multimídia. Tal sucesso deve-se não

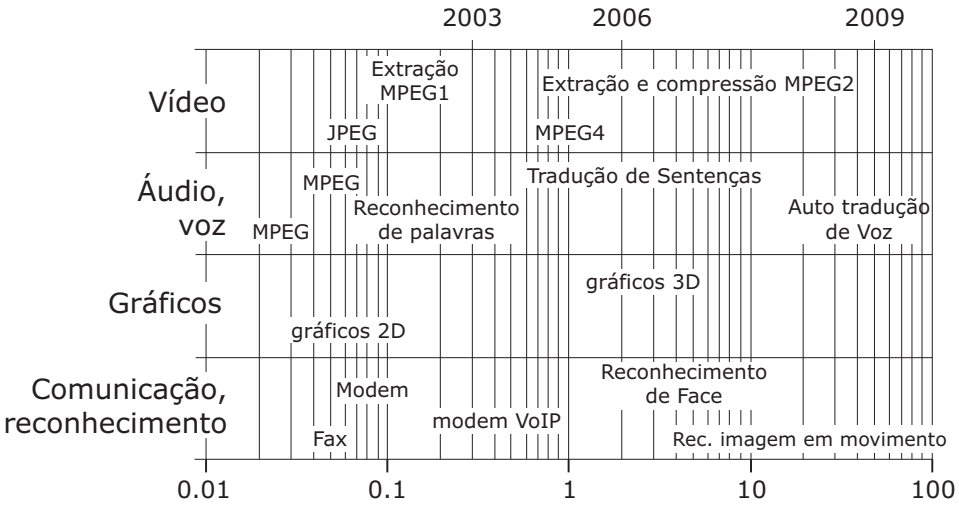


Figura 2.9: Gráfico de bilhões de operações por segundo (GOPS), requeridos por aplicação subdivididas em quatro categorias. [91]

só à tecnologia de fabricação, mas também, à habilidade em se conectar componentes já existentes incluindo processadores, memórias e controladores. A reutilização de componentes já existentes não é só atrativa no aspecto de consumo de energia, pela utilização de *clocks* distribuídos, mas também em relação aos custos de projeto e testes. Com a utilização de componentes separados, pode-se testá-los individualmente de forma desacoplada [9, 115].

Em relação ao acoplamento dos componentes individuais, o método de sincronização mais provável para os futuros circuitos integrados envolve a utilização de *clocks* distintos. Tal ausência de uma referência única de tempo torna os *SoCs* sistemas distribuídos, porém presentes em um único substrato de silício. Nesta abordagem é pouco provável que um sistema de controle global centralizado tenha sucesso em manter o sincronismo de cada componente do sistema. O modo síncrono requer que cada flip-flop, cada roteador componente da *NoC* utilize o mesmo sinal de *clock*. No entanto tal mecanismo propicia um grande consumo de energia, podendo requerer até 30% do total consumo de potência do sistema como um todo [90].

No modo mesócrono, cada *clock* local é derivado de um *clock* global distribuído por todo circuito integrado. Todos os módulos síncronos em um sistema mesócrono usam o mesmo *clock* fonte, porém a fase nos módulos distintos pode diferir devido a um *clock* global não balanceado da rede. Se uma *NoC* tem uma topologia regular, como malha por exemplo, a diferença de fase é determinística. Para topologias irregulares, no entanto, a diferença de fase

pode não ser determinística e sincronizadores precisam ser usados entre os domínios de *clock*.

No caso pleisócrono, os sinais de *clock* são gerados localmente. O *clock* local tem quase a mesma frequência que os *clocks* gerados em outras partes do sistema, causando um desvio de frequência de pequeno porte. Entretanto, uma vez que as frequências de *clock* entre os componentes de rede não são exatamente as mesmas, o receptor requer um mecanismo para resolver a margem de fase variável. A necessidade de tal mecanismo pode criar uma sobrecarga nas implementações das *NoCs* desse tipo.

No caso assíncrono, não há necessidade de *clocks*. Sinais chegam de forma arbitrária e protocolos assíncronos, tais como *handshaking* de duas ou quatro fases são usados para a sincronização entre os componentes de comunicação. O esquema globalmente assíncrono localmente síncrono (GALS) [52] é uma extensão do paradigma assíncrono, onde os protocolos de comunicação assíncrona são utilizados para a comunicação dentre as regiões distintas com *clock* local.

Essa abordagem explora as vantagens de ambos os sistemas, síncrono e assíncrono. No entanto, o uso de um protocolo assíncrono pode degradar o desempenho, pois cada sinal deve viajar por todo o sistema a cada transição de sinal. Uma solução é usar *pipelining* assíncrono [113] que pode melhorar o rendimento em ligações de maior extensão.

A utilização de junções espalhadas pelos diferentes blocos *IP*, causa problemas de comunicação e sincronização por meio de ruído elétrico, interferência eletromagnética, cargas induzidas por radiação, dentre outros, agravados quando altas frequências são utilizadas. Neste contexto, para se transmitir dados digitais por fios de forma confiável é necessária a utilização de mecanismos de verificação e correção de erros, os quais introduzem além de uma sobrecarga no sistema, um consumo de potência excessivo em tarefas secundárias dentro de tal cenário. Lembrando que o cenário em questão é o da integração de componentes que já fazem parte das necessidades atuais, sendo que tanto a quantidade de blocos *IPs* quanto a integração de tais blocos tende a aumentar.

Cada bloco pertencente ao *SoC* tem sua função específica e pode ter um *clock* diferente de operação. DSPs (processadores digitais de sinal) tendem a trabalhar de forma paralela

junto a outros componentes de propósito similar, tais como codificadores e decodificadores de áudio e vídeo. Diferentemente dos DSPs, os processadores de propósito geral operam de forma sequencial e quando acoplados a ambientes heterogêneos há uma necessidade tanto de abstração do modelo de tráfego de dados, quanto de uma padronização dos meios de comunicação dentre os diferentes blocos. Dessa demanda em se ter uma maneira eficiente e padronizada de comunicação entre os blocos *IPs*, componentes de um *SoC*, surgiram as Networks-on-Chip [10, 17, 75, 100].

As *NoCs* surgiram em meados dos anos 1990 em uma tentativa de integrar de modo eficiente a grande quantidade de processadores e componentes reutilizáveis em um único substrato de silício, os *Systems on Chip*. Para se resolver o problema de interconexão, atraso, consumo de energia e escalabilidade que surgiram pela utilização dos barramentos convencionais em *SoCs*, cada vez maiores e mais complexos, as *NoCs* foram propostas como um paradigma evolucionário [17, 31].

Network-on-Chip é vista como uma abordagem para prover alto desempenho e escalabilidade, além de uma infraestrutura robusta para comunicação *on-chip*. As arquiteturas de interconexão utilizadas nos *SoCs*, devem atender aos requerimentos destes sistemas oferecendo concomitantemente escalabilidade, reusabilidade e paralelismo na comunicação. Além disso, cobrir questões como restrições de consumo de energia e utilização de *clock* distribuído [52, 87, 103].

Nas *NoCs* recursos computacionais como memória, *I/O* e unidades lógicas são interconectados por roteadores. Em tais redes os recursos comunicam-se entre si utilizando pacotes de dados direcionados por roteadores implementados no mesmo circuito integrado.

Os problemas críticos endereçados pelas *NoCs* são:

— Interconexão global: atraso, consumo de energia, ruído, escalabilidade e confiabilidade.

— Circuitos integrados com multi-processadores.

— Produtividade na integração de sistemas.

Suganya e Nagarajan [105] apresentam um sistema *multi-core* embarcado e seus resultados

apontam o roteamento como o maior problema do projeto de circuitos integrados na área de multi-processadores embarcados.

Quanto à preocupação com produtividade no desenvolvimento de *Systems on Chip* têm-se muitas iniciativas que abordam as metodologias CAD (*Computer Aided Design*). Dentre elas, podemos citar o SUNMAP [83], que mapeia a estrutura de núcleos para uma topologia específica conforme as necessidades da aplicação em questão. A ferramenta também suporta diferentes métodos de roteamento como caminho mínimo e divisão de tráfego. Em [15] os autores vão além, propondo um sistema reconfigurável que possa se adaptar as diferentes necessidades.

Em [65] o Xpipes é apresentado. Ele fornece um sistema de desenvolvimento completo e flexível de *NoC* com uma biblioteca de componentes personalizáveis, tais como: Open Core Protocol (OCP), interfaces de rede (NIS), roteadores e interligações, além do compilador Xpipes. Além do Xpipes várias outras arquiteturas têm sido propostas na literatura como: 2D Torus [31], AEthereal [46], Butterfly Fat Tree (BFT) [89], CHAIN [12], CLICHÉ [75], HERMES [2], MANGO [19], Nostrum [75], Octagon [71], Proteo [100], QNoC [21], SOCBUS [117] e SPIN [5]. Uma arquitetura de *NoC* define uma topologia e um conjunto de protocolos que determinam esquemas para comutação, roteamento, interfaces e controle de fluxo.

A adoção de *NoCs* é um processo evolutivo, cujo sucesso será determinado por fatores como a complexidade dos projetos e como rapidamente o processo de fabricação irá avançar nos próximos anos. A figura 2.10 mostra a evolução das tecnologias de interconexão *SoCs*.

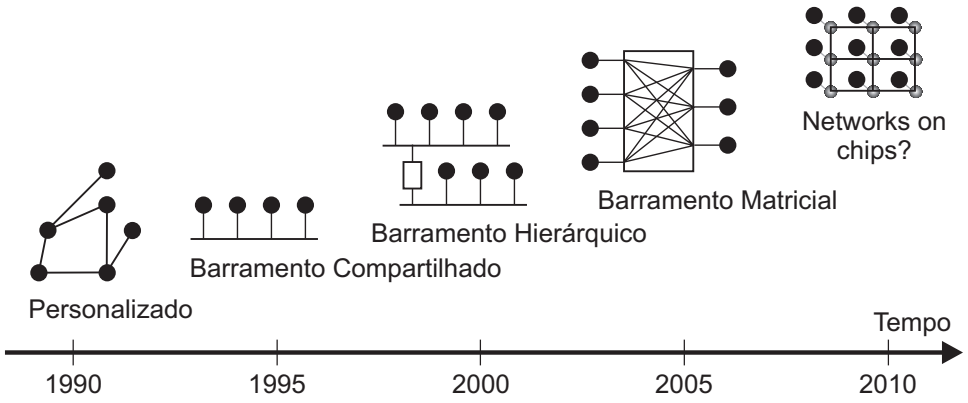


Figura 2.10: Evolução das tecnologias de interconexão de Systems on Chip [91]

O conhecimento provindo da área de redes de computadores, sistemas distribuídos e da área de telecomunicações, fornece uma ampla gama de resultados a serem mapeados para o domínio das *Networks on Chip*. Este mapeamento não é nada simples devido principalmente às restrições impostas à implementação das infraestruturas de rede em silício.

Desta limitação se faz necessário a utilização de novas abordagens para antigos problemas como o de roteamento. Técnicas de caminhos disjuntos representam uma ampla gama de pesquisa na área de grafos cuja utilização prática se dá na área de redes, por exemplo.

Devido às restrições de espaço das *NoCs* seus componentes, roteadores e interfaces de rede, devem ser simples para minimizar requisitos de memória e capacidade de processamento. Nas redes tradicionais, uma maneira usual para se evitar congestionamento é o simples descarte de pacotes [29, 108]. Uma implementação de uma *NoC* que não descarte pacotes pode ser uma solução de baixo custo, porém, com um revés de introduzir a possibilidade de *deadlocks*. *Deadlock* é uma situação onde dois ou mais processos são incapazes de prosseguir por que cada um deles está bloqueado esperando por outro para fazer algo.

Deadlocks podem ser evitados em implementações que não descartam pacotes pelo uso de restrições em relação tanto à topologia quanto ao roteamento. Topologias como *fat-tree* têm sido consideradas onde, no caso de estouro de buffer, pacotes são devolvidos aos vértices remetentes pelo recuo nas árvores [47]. Neste trabalho, é apresentada uma abordagem de árvores geradoras disjuntas a serem utilizadas alternadamente para o envio de pacotes no intuito de se evitar o congestionamento de buffers e consequentemente deadlocks quando na espera da liberação de tais recursos.

Várias abordagens têm sido propostas para o problema de se encontrar árvores disjuntas sobre topologias diversas. Dentre os algoritmos para a construção de árvores geradoras arco-disjuntas em G, tem-se [43, 111]. Em [8] Annexstein e Berman apresentaram um modelo matemático para roteamento de redes baseado na geração de caminhos em uma direção consistente, como descrito pelos autores. Tal direção consistente é obtida através de um *framework* algébrico e geométrico, pela definição de um sistema de coordenadas direcional para espaços vetoriais reais. Este modelo faz o mapeamento de vértices de uma rede para pontos

de um espaço multi-dimensional e garante que os caminhos gerados nas diferentes direções, partindo-se de um mesmo nó fonte, sejam disjuntos.

Árvores geradoras arco-disjuntas têm sido estudadas não só do ponto de vista teórico mas do prático também. Em computação paralela massiva, o processo de envio de uma mensagem originada em um processador para todos os outros processadores (*broadcasting*) é um processo fundamental. É de conhecida importância a construção de um grande número de árvores geradoras independentes de baixa profundidade na interconexão de redes, para possibilitar um esquema eficiente de *broadcast* [35, 41]. Tais árvores também são utilizadas em esquemas de tolerância a falhas [38, 45, 63, 86, 92], apresentados a seguir.

2.2 Trabalhos Relacionados

Esta seção apresenta e discute as principais abordagens encontradas na literatura para geração de caminhos disjuntos sobre grafos $k-$conexos. Isso é importante devido à aplicação de tais caminhos no roteamento de *NoCs*, como proposto neste trabalho. A utilização da topologia hipercubo foi devida a seu histórico em computação paralela [61, 85], assim como a simplicidade dos algoritmos propostos aqui, que somente se aplicam ao hipercubo.

2.2.1 Árvores Geradoras Independentes sobre Grafos de Produto

A construção de árvores geradoras independentes em uma família particular de redes tem recebido grande atenção da comunidade científica [25, 37, 40, 70]. Sendo que os principais trabalhos abordando hipercubos são: [86, 109, 124].

Em [86], Obokata et al. observaram que um hipercubo k-dimensional possui k árvores geradoras independentes enraizadas em qualquer vértice. De acordo com os autores, um hipercubo k-dimensional Q_k pode ser visto como um grafo de produto de Q_{k-1} e K_2 (grafo completo com dois vértices), e k árvores geradoras independentes em Q_k podem ser construídas recursivamente a partir de $k-1$ árvores geradoras independentes em $Q_k - 1$. A figura 2.11 mostra quatro árvores geradoras independentes sobre o hipercubo Q_4 utilizando o algoritmo

apresentado em [86].

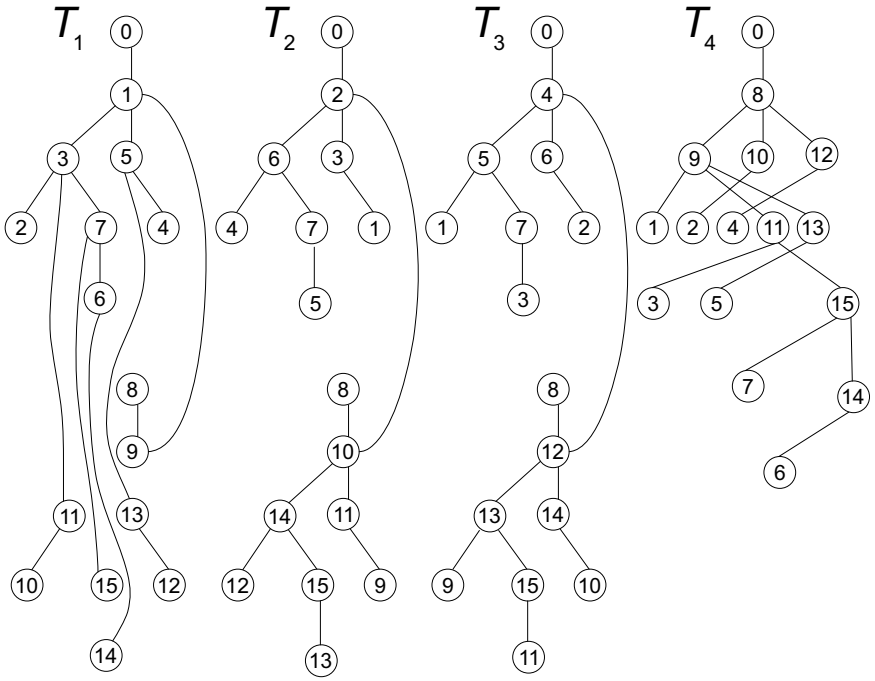


Figura 2.11: Quatro árvores geradoras independentes sobre o hipercubo Q_4 [109].

Entretanto o algoritmo de Obokata não produz árvores geradoras ótimas em termos de comprimento médio dos caminhos para $k > 3$. Utilizando a definição 1 de árvores geradoras independentes ótimas [109]:

Definição 1: Seja $d(G; u, v)$ a distância, ou seja, o número de arestas, entre os vértices u e v em G. A *altura* da árvore T, enraizada no vértice r, é a distância máxima dos caminhos de r para qualquer outro vértice em T.

Em [74], o *comprimento do caminho* de uma árvore é definido como a soma das distâncias de cada vértice até a raiz da árvore. O comprimento do caminho é um conceito que pode ser utilizado para analisar o custo de busca de uma árvore. Seja G um grafo k-conexo, caso exista $S = \{T_1, T_2, ..., T_k\}$, um conjunto de árvores geradoras independentes, enraizadas em r em G, $D(v)$ denota a *distância média* de v a r em S; i.e,

$$D(v) = \sum_{i=1}^{k} d(T_i; r, v)/k \tag{2.9}$$

Então, o comprimento médio dos caminhos de S pode ser definido como a soma de $D(V)$ para todo $v \in V$ e $v \neq r$. Ou seja, comprimento médio é dado por:

$$\sum_{v \in V \setminus r} D(v) \; ou \sum_{i=1}^{k} \sum_{v \in V \setminus r} d(T_i; r, v)/k \tag{2.10}$$

Um conjunto S de árvores geradoras independentes é dito ser ótimo se o comprimento médio dos caminhos é mínimo. Por exemplo, o comprimento médio dos caminhos do conjunto de árvores mostrado na figura 2.11 é: (46 + 46 + 46 + 50) / 4 = 47, enquanto o comprimento do médio dos caminhos do conjunto de árvores mostrado na figura 2.12 é (46 + 46 + 46 + 46) / 4 = 46. Um dos requisitos para que as árvores sejam ótimas é que o comprimento máximo dos caminhos do raiz r, para qualquer outro vértice $v \in V(T)$ deve ser $O(logN) + 1$, sendo N o número de vértices.

2.2.2 Árvores Geradoras Independentes Ótimas sobre Hipercubos

Tang, Wang e Leu [109] apresentaram um algoritmo $O(kn)$ para construir k árvores geradoras independentes ótimas em um hipercubo k dimensional. O algoritmo baseia-se na ideia de que k árvores geradoras independentes podem ser construídas recursivamente a partir de $k-1$ árvores independentes sobre $Q^{(k-1)}$. A principal melhoria em comparação ao algoritmo de Obokata et al. [86] é que as árvores são ótimas em relação ao comprimento médio dos caminhos. A figura 2.12 mostra as quatro árvores geradoras ótimas para Q_4.

Os autores desse trabalho consideram somente o vértice 0 como raiz porque o hipercubo é um grafo vértice simétrico [50].

Primeiramente Tang et al. reescrevem o algoritmo proposto por Obokata et al. [86] da seguinte forma, figura 2.13. Baseado no algoritmo IST, k árvores geradoras independentes sobre Q_k são construídas recursivamente pela combinação de $T_{A,i}$ com $T_{B,i}$ ($i = 1,\ 2,\ ...,\ k-1$) e então transformando T_1 a fim de criar T_k.

Um método denominado troca benéfica de pai é utilizado para reduzir a altura das árvores de Obokata et al. tornando-as, assim, ótimas em relação à altura. Para esta finalidade, o

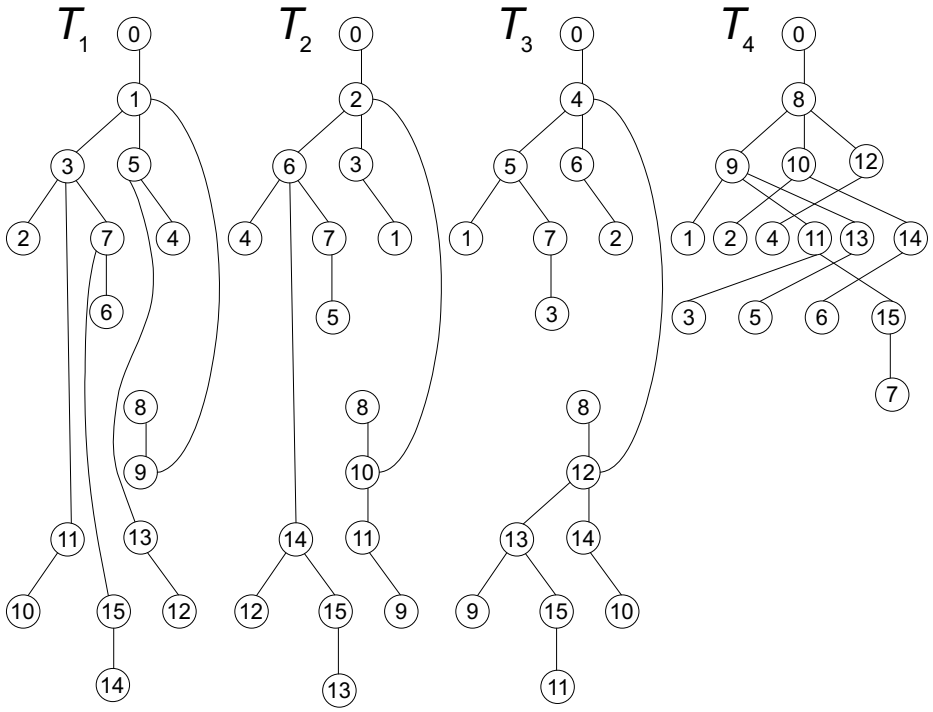


Figura 2.12: Quatro árvores geradoras independentes ótimas sobre o hipercubo Q_4 [109].

algoritmo apresentado na figura 2.14 é proposto por Tang et al.

Apesar de produzir árvores geradoras independentes ótimas, o algoritmo de Tang et al. [109] é recursivo e por essa razão requer uma grande quantidade de memória para a construção das árvores.

2.2.3 Árvores Geradoras Independentes Ótimas sobre Hipercubos (sem recursão)

Em [124] Yang et al apresentam um algoritmo utilizando o conceito de quadrado latino com distância de *Hamming* é apresentado (figura 2.15). Em relação ao trabalho de Tang et al. [109] a geração das árvores é mais simples e facilmente paralelizável. A quantidade de memória utilizada é da ordem de $O(kN)$, o que aproxima-se da metade da quantidade requerida em [109] cuja construção recursiva requer $\sum_{i=1}^{k} i \cdot 2^i = (k-1)2^{k+1} + 2 = (2k-2)N + 2$.

A figura 2.16 mostra as árvores geradoras independentes ótimas para o hipercubo Q_4,

Algoritmo *IST*
Entrada: k.
Saída: Um conjunto de k árvores geradoras independentes sobre Q_k .
Método:
Passo 1: **Se** k for igual a 2, **então** retorne caminho <0, 1, 3, 2> e caminho <0, 2, 3, 1>
como T_1 e T_2, respectivamente,
senão execute *IST* com $k = k - 1$.
Passo 2: Sejam $T_{A,1}, T_{A,2}, ..., T_{A,k-1}$ as $k - 1$ árvores geradoras independentes sobre Q_{k-1}.
Construa $T_{B,1}, T_{B,2}, ..., T_{B,k-1}$ pela adição de 2^{k-1} ao rótulo de cada vértice em
$T_{A,1}, T_{A,2}, ..., T_{A,k-1}$
Passo 3: (Construção de $T_1, T_2, ..., T_{k-1}$)
Para $i = 1$ **até** $k-1$ **faça**
Construa T_i conectando o único filho da raiz em $T_{B,i}$ (i.e., vértice $2^{i-1} + 2^{k-1}$)
com o vértice correspondente em $T_{A,i}$ (i.e., vértice 2^{i-1}).
Passo 4: (Construção de T_k)
Subpasso 4.1 (Crie o único filho do raiz)
Conecte o vértice 2^{k-1} com o vértice 0.
Subpasso 4.2 (Crie $k-1$ netos do raiz)
Para $i = 0$ **até** $k-2$ **faça**
Conecte o vértice $2^{k-1} + 2^i$ com o vértice 2^{k-1}
Subpasso 4.3 (Crie $2^{k-1} - 1$ folhas)
Para todo vértice $v \in Q_A$ e $v \neq 0$ **faça**
Conecte o vértice v com o vértice $v + 2^{k-1}$.
Subpasso 4.4 (Crie $2^{k-1} - k$ arestas em T_k transformando T_1)
Para todo vértice $v \in Q_B$ e $v \notin \{2^{k-1}\} \cup N(2^{k-1})$
faça
pai de (v,k) = pai (v,1).
pai de (v,1) = $v - 2^{k-1}$.
fim do faça
Fim do Algoritmo *IST*

Figura 2.13: Algoritmo IST (Independent Spanning Tree) [109], baseado no algoritmo de Obokata et al. [86].

obtidas utilizando o algoritmo apresentado neste trabalho.

2.3 Discussão

Neste capítulo foram apresentados os principais trabalhos relacionados ao tema da tese, além de algumas das motivações práticas para o trabalho proposto nesta tese. A crescente demanda por dispositivos mais complexos para atender aplicações cada vez mais sofisticadas, assim como as limitações físicas impostas pela densidade de integração, faz com que se tornem necessárias novas abordagens para a construção de *SoCs*. Tais abordagens podem se valer de soluções já propostas para problemas antigos, mas com o revés de que estas soluções devem ser adaptadas para as limitações dos dispositivos atuais. Limitações estas como:

— Consumo de potência: visto que uma grande parte dos dispositivos móveis funciona a

Algoritmo *OIST*
Entrada: *k*.
Saída: *k* árvores geradoras ótimas sobre Q_k.
Método:
Passo 1: **Se** *k* for igual a 2, **então** retorne caminho <0, 1, 3, 2> e caminho <0, 2, 3, 1> como T_1 e T_2, respectively.
senão execute *OIST* com *k = k - 1*.
Passo 2: Construa $T_1, T_2, \ldots, T_{k-1}$ usando o mesmo método descrito nos Passos 2 e 3 do Algoritmo IST.
Passo 3: (Construção de T_k)
Subpassos 3.1, 3.2 e **3.3** são os mesmos Subpassos 4.1, 4.2 e 4.3 do Algoritmo IST, respectivamente.
Subpassos 3.4 (Crie $2^{k-1} - k$ arestas em T_k transformando T_x)
Para todo vértice $v \in Q_B$ e $v \notin \{2^{k-1}\} \cup N(2^{k-1})$
faça
Escolha T_x $(1 \leq x \leq k-1)$ como a árvore transformada na qual
v **– pai** (*v*, *x) é máximo.*
pai de (v,k) = pai (v,x).
pai de (v,x) = $v - 2^{k-1}$.
fim do faça
Fim do Algoritmo *OIST*

Figura 2.14: Algoritmo OIST (Optimal Independent Spanning Trees) [109]

Algoritmo GEN-PARENT
início
para cada vértice $x(\neq 0) \in Q_k$ e com string binária $x = x_{k-1}\, x_{k-2} \ldots x_0$ **faça**
para $i = 0$ até $k - 1$ **faça**
se $(x_i = 1)$ $parent(T_i, x) = x - 2^{succ(i)}$; // onde $succ(i)$ se refere a $I_k(x)$
se $(x_i = 0)$ $parent(T_i, x) = x - 2^i$;
fim do faça
fim do faça
fim GEN-PARENTS

Figura 2.15: Algoritmo GEN PARENT [124]

bateria;

— Área: visto que os dispositivos móveis tem tamanho reduzido, implicando que os circuitos devem ocupar a menor área possível;

— Desempenho: tarefas secundárias, como roteamento, devem utilizar algoritmos otimizados para que tais tarefas executem de forma transparente para o usuário final;

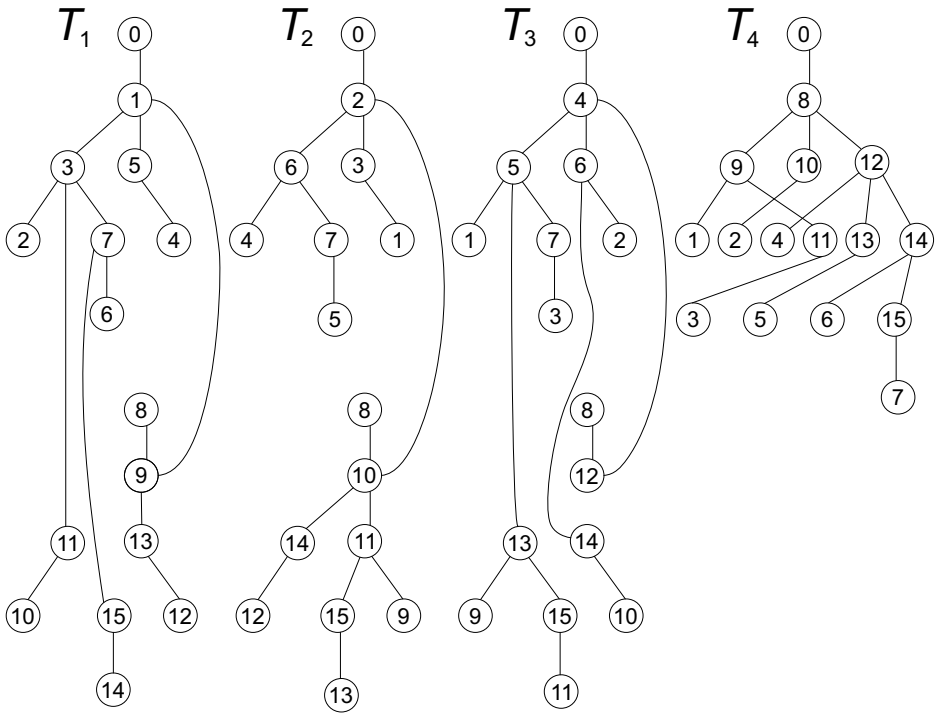


Figura 2.16: Quatro árvores geradoras independentes ótimas sobre Q_4 utilizando o algoritmo apresentado neste trabalho.

CAPÍTULO 3

METODOLOGIA

Este capítulo descreve a metodologia desenvolvida para a construção de árvores geradoras independentes sobre hipercubos. Dois algoritmos foram elaborados ao longo deste trabalho, sendo que cada um deles tem sua devida importância conforme a aplicação e ao número de blocos *IPs*. O primeiro algoritmo é um método de otimização, referenciado de agora em diante como *OptimIST* (*Optimization for Independent Spanning Trees*), o segundo algoritmo foi denominado *MinimalIST* (*Minimalistic Independent Spanning Trees*) devido às suas caraterísticas em relação a consumo de recursos computacionais.

3.1 Algoritmo de Otimização - OptimIST

O primeiro algoritmo é um método de otimização que tem como objetivo minimizar o número de erros nas árvores geradas, o que entende-se por erro é explicado a seguir.

Pela definição apresentada anteriormente, dado um grafo G, um conjunto de árvores geradoras enraizadas em um vértice r em G é dito independente se, para cada vértice v em G, $v \neq r$, os caminhos de r a v em qualquer par de árvores são vértice disjuntos. Caso contrário, cada vértice comum aos caminhos de r a v é considerado um erro. O algoritmo *OptimIST* é executado uma única vez e seu objetivo é gerar um conjunto de arestas que, caso evitadas, produzem k árvores geradoras independentes para Q_k. As árvores produzidas são ótimas em relação ao comprimento médio dos caminhos.

O algoritmo de otimização pode ser executado em uma máquina com grande capacidade computacional, visto que só é executado uma única vez. Por fim, o resultado é uma lista de arestas a serem evitadas as quais são utilizadas no algoritmo simplificado, cujo destino final é a implementação em um dispositivo com capacidade computacional limitada como, por exemplo, um roteador de um sistema embarcado.

O desempenho de um determinado computador paralelo é fortemente dependente da topologia de grafo escolhida e dos algoritmos por ele executados. Diversas topologias têm sido propostas na literatura tais como: árvores, tórus, hipercubos, grafos estrela, *fat tree*, etc [3, 114, 120, 121].

O estudo de árvores geradoras independentes tem sido recorrente na literatura. Dentre suas aplicações tem-se protocolos tolerantes a falhas e utilização em redes de computação distribuída [13, 37, 14]. Em uma rede, por exemplo, *broadcasting* é o envio de uma mensagem de um dado vértice a todos os outros vértices da rede. Em particular, ao transmitir a partir de uma fonte, pode-se querer encontrar tantas árvores geradoras arco-disjuntas quanto for possível para se evitar congestionamento.

No entanto, tal exigência pode ser muito rigorosa e pode não permitir tantas árvores geradoras quanto a conectividade do grafo. Importante notar que se quer enviar k mensagens originadas no vértice fonte para outro vértice v em k diferentes rotas, com o objetivo de se evitar congestionamentos. Logo, não é necessário para isso se ter k árvores geradoras arco-disjuntas. É suficiente que as k rotas não se cruzem. Na verdade, esta definição é um relaxamento dos requisitos das árvores geradoras arco-disjuntas. Por exemplo, um hipercubo Q_k não tem arestas suficientes para as k árvores arco-disjuntas, no entanto, ele possui k árvores geradoras independentes. De fato, para se ter k árvores arco-disjuntas sobre Q_n cada aresta do hipercubo que conecta dois vértices $u, v \in V$ deve ser duplicada, com cada uma delas assumindo um sentido, conforme mostrado na figura 3.1 para as arestas de Q_3.

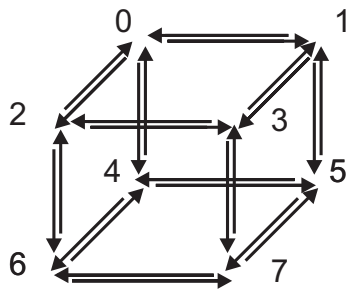


Figura 3.1: Arestas duplicadas de Q_3 a serem distribuídas dentre as 3 árvores geradoras de Q_3.

Conjectura 3.1.1. *Para todo grafo k-conexo existem k árvores geradoras independentes, provado para $k \leq 4$ (com raiz em qualquer vértice do grafo)*

A quantidade necessária de arestas para formar uma árvore cobrindo todos os vértices do grafo G é igual ao número de vértices menos um, visto que um grafo minimamente conectado requer uma aresta a menos que o número de vértices para ser conexo [20].

$$|E| = |V| - 1 \tag{3.1}$$

Logo a quantidade de arestas necessárias para as k árvores geradoras é dada por:

$$|E_{T_k}| = k \times (|V| - 1) \tag{3.2}$$

Onde $|E_{T_k}|$ é a quantidade de arestas de todas as k árvores geradoras e $|V|$ o número de vértices do grafo.

O hipercubo não possui arestas distintas suficientes para se obter a quantidade de árvores geradoras arco-disjuntas correspondente a sua conectividade k. No entanto, se considerarmos as arestas em cada direção como distintas, pode-se obter as arestas necessárias para as k árvores, sendo esse o ponto de partida do algoritmo proposto.

Considerando cada aresta do hipercubo como duas arestas distintas em direções opostas, ligando os mesmos vértices da aresta original, temos o novo número de arestas do hipercubo dado por:

$$|E_{Q_k}| = 2 \times (2^{(k-1)} \times k) \tag{3.3}$$

Como o vértice raiz de cada árvore deve aparecer em cada árvore somente naquela posição, ou seja, não há nenhum caminho nas árvores da forma $..., v, r, ...$, com r presente no meio do caminho, e sendo que todos os caminhos são iniciados em r, as arestas incidentes em r devem ser consideradas em uma única direção. Como o hipercubo Q_k é k regular, tem-se k arestas incidentes ao vértice raiz, que não são duplicadas, sendo assim, a nova equação do número de

arestas de Q_k passa a ser:

$$|E_{Q_k}| = 2 \times (2^{(k-1)} \times k) - k \tag{3.4}$$

Logo a quantidade de arestas em Q_k é igual a quantidade de arestas necessárias para as k árvores E_{T_k}:

$$\begin{aligned}
|E_{T_k}| &= |E_{Q_k}| \\
k \times (|V| - 1) &= 2 \times (2^{(k-1)} \times k) - k \\
k \times 2^k - k &= 2 \times (\frac{2^k \times k)}{2}) - k \\
k \times 2^k - k &= 2^k \times k - k \\
k \times 2^k - k &= k \times 2^k - k
\end{aligned}$$

De fato, pode-se construir árvores geradoras sob qualquer grafo hipercubo Q_n partindo-se do grafo completo K_{2^n}, eliminando-se as arestas das árvores de K_{2^n} que não estão contidas em Q_n e redistribuindo as restantes dentre as n árvores de Q_n (apêndice A e apêndice B). O problema está em como distribuir de maneira eficaz essas arestas de modo a manter as árvores independentes. Por exemplo, a figura 3.2 mostra as 21 arestas de Q_3 distribuídas dentre as 3 árvores geradoras independentes.

No apêndice C um algoritmo para distribuição das arestas é apresentado, denominado *EdgeDistAdj*. Este algoritmo opera linha a linha da matriz de adjacências, sucessivas vezes, para obter as árvores geradoras. O algoritmo por si só não é suficiente para construir as árvores geradoras independentes, somente garante que cada vértice tenha um pai diferente em cada árvore.

Por exemplo, as árvores de Q_4 geradas por este algoritmo, mostradas na figura 3.3, não são independentes, pois o caminho do vértice 0 para o vértice 15 nas árvores T_1 e T_3 tem o

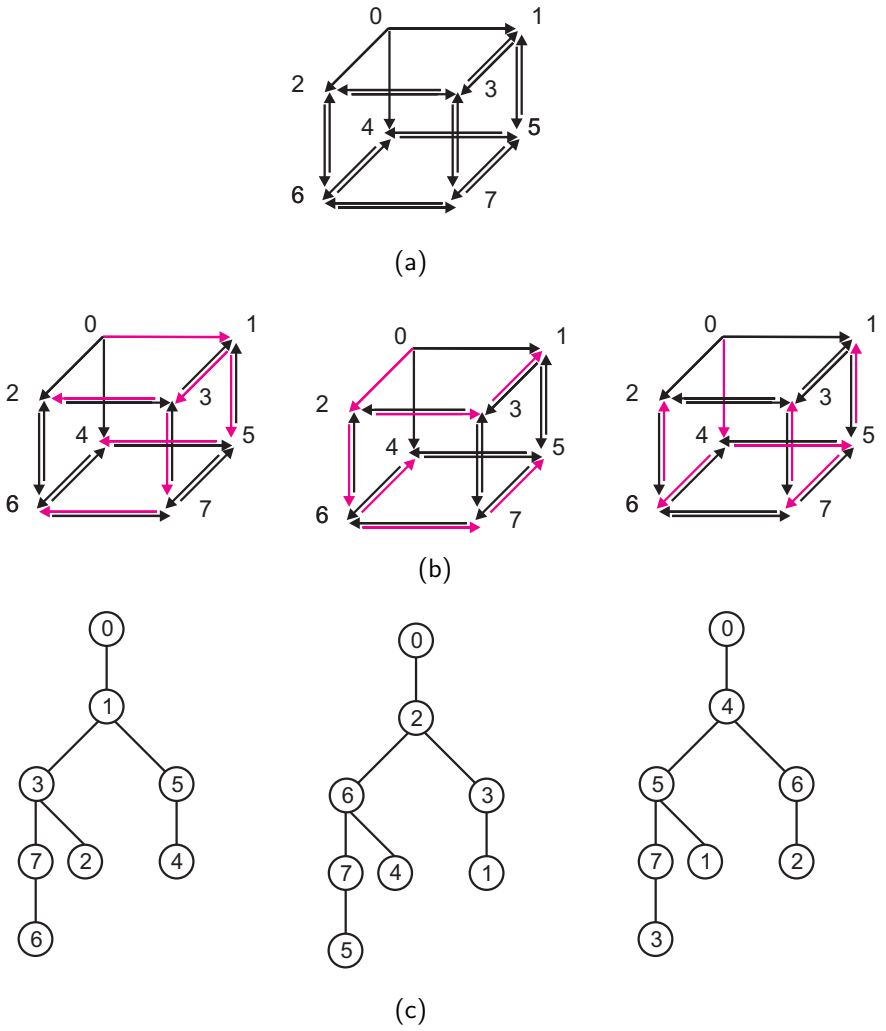


Figura 3.2: (a) As arestas de Q_3 a serem distribuídas, considerando o vértice 0 como raiz. (b) Uma distribuição possível gerando as respectivas árvores abaixo. (c) 3 árvores de Q_3 geradas pela distribuição das arestas.

vértice 10 em comum.

Entretanto, com poucas operações, como mostrado na figura 3.4, trocando-se o pai do vértice 15 na árvore T_0 de 7 para 11 e na árvore T_1 de 11 para 7 obtém-se as 4 árvores geradoras independentes sobre Q_4.

As quatro árvores geradoras independentes mostradas na figura 3.4 são obtidas por uma única aresta evitada. Na figura em questão cada cor representa uma árvore, azul escuro, azul claro, amarelo e verde representam as árvores T_0, T_1, T_2 e T_3, respectivamente.

Para tornar esse processo automatizado, os passos descritos a seguir foram implementados.

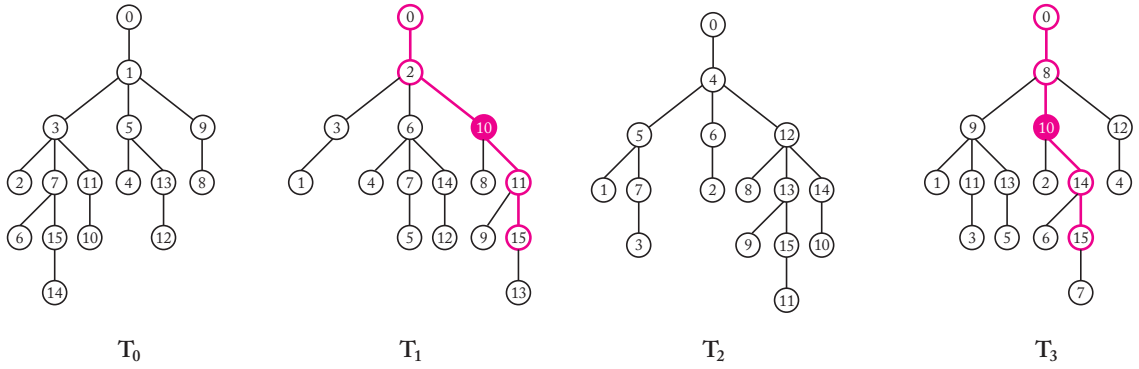


Figura 3.3: Os caminhos do vértice 0 ao vértice 15 nas árvores T_1 e T_3 compartilham o vértice 10.

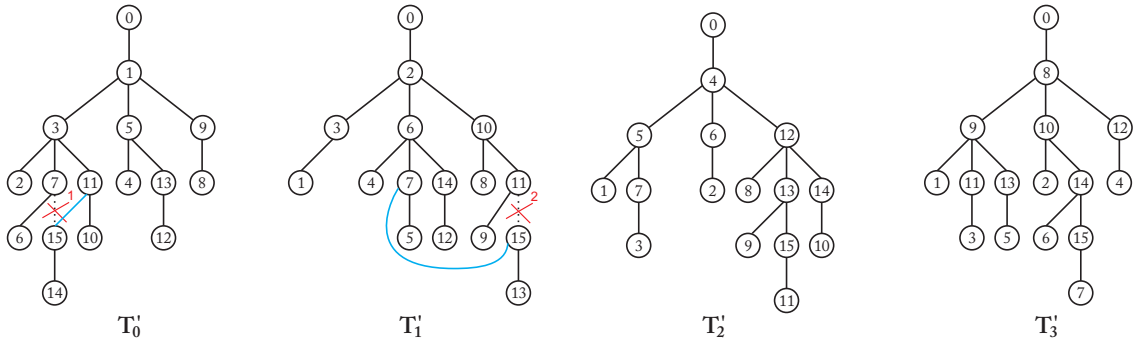


Figura 3.4: Com algumas mudanças pode-se obter 4 árvores geradoras independentes sobre Q_4.

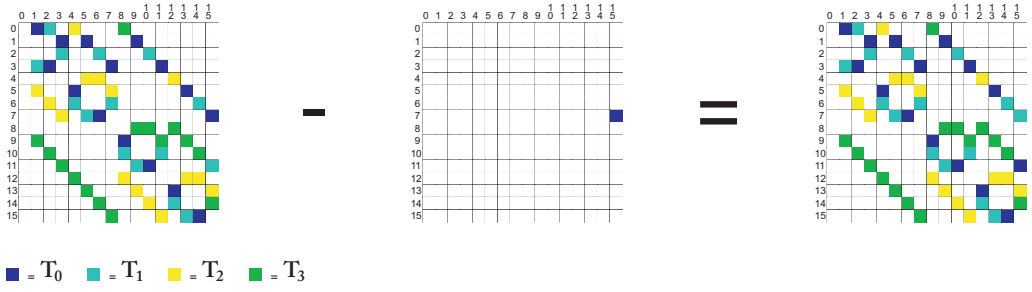


Figura 3.5: Evitando-se a aresta 7-15 na árvore T_0 para formar as 4 ISTs de Q_4

Ainda partindo-se da distribuição de arestas mencionada anteriormente, escolhe-se de forma incremental uma aresta que seria atribuída a uma árvore específica e a remove dessa atribuição. Calcula-se então a quantidade de vértices em comum entre todos os caminhos do vértice raiz da árvore para os demais. Caso a remoção de tal aresta atribuída a árvore em questão cause a

diminuição do número de vértices em comum, esta aresta é adicionada a uma lista de arestas a serem evitadas e o processo continua sucessivamente pegando-se a próxima aresta. Caso a eliminação de tal aresta cause a criação de um grafo desconexo e não das k árvores geradoras, a contabilização dos vértices com problemas não é executada, partindo-se então para o próximo passo do algoritmo, escolhendo-se outra aresta para ser evitada. Em outras palavras, percorre-se a lista de arestas evitando-se uma aresta por vez e contabilizando o número de vértices com problemas. Caso ocorra a diminuição do número de vértices problemáticos, a aresta que causou tal diminuição é colocada em uma lista de arestas a serem evitadas até que o número de erros converja para 0, quando possível. O fluxograma do algoritmo é apresentado na figura 3.6.

É importante notar que esse algoritmo é executado uma única vez para se calcular a lista de arestas a serem evitadas, as quais serão adicionadas de forma estática no algoritmo final.

Com a aplicação do método de otimização, o algoritmo para criar árvores geradoras independentes foi validado até Q_8, hipercubo com 256 vértices. A partir deste número, a próxima potência de dois, com 512 vértices (Q_9), apresentou problemas. Das árvores geradas para Q_9 tem-se 33 pares de árvores completamente independentes e 3 pares apresentando problemas em 3 caminhos com 1 vértice em comum em cada um deles. Uma nova abordagem se fez necessária para resolver esse problema.

Observando-se a matriz de arestas a serem evitadas, notou-se que certas arestas se repetiam em intervalos regulares. O padrão de arestas a serem evitadas na árvore 0 se repetia em um intervalo regular, os da árvore 1 também, com a intercalação de um quadro, os da árvore 2 com dois quadros e assim por diante, sempre em intervalos de 2^t onde $t \in \{0, ..., k\}$ representa o índice da árvore (figura 3.7).

Além de apresentar padrões bem definidos na matriz, observou-se que estes padrões apareciam em locais específicos. Para mostrar o significado das localidades específicas no grafo do hipercubo, uma representação gráfica do produto de Kronecker é ilustrada na figura 3.8. O produto de Kronecker ou produto tensorial é a operação base para a geração de hipercubos, conforme mostrado no capítulo 2. Nesta figura pode-se notar que o produto de Kronecker duplica o grafo original e as arestas em destaque ligam o grafo original, de dimensão k, ao

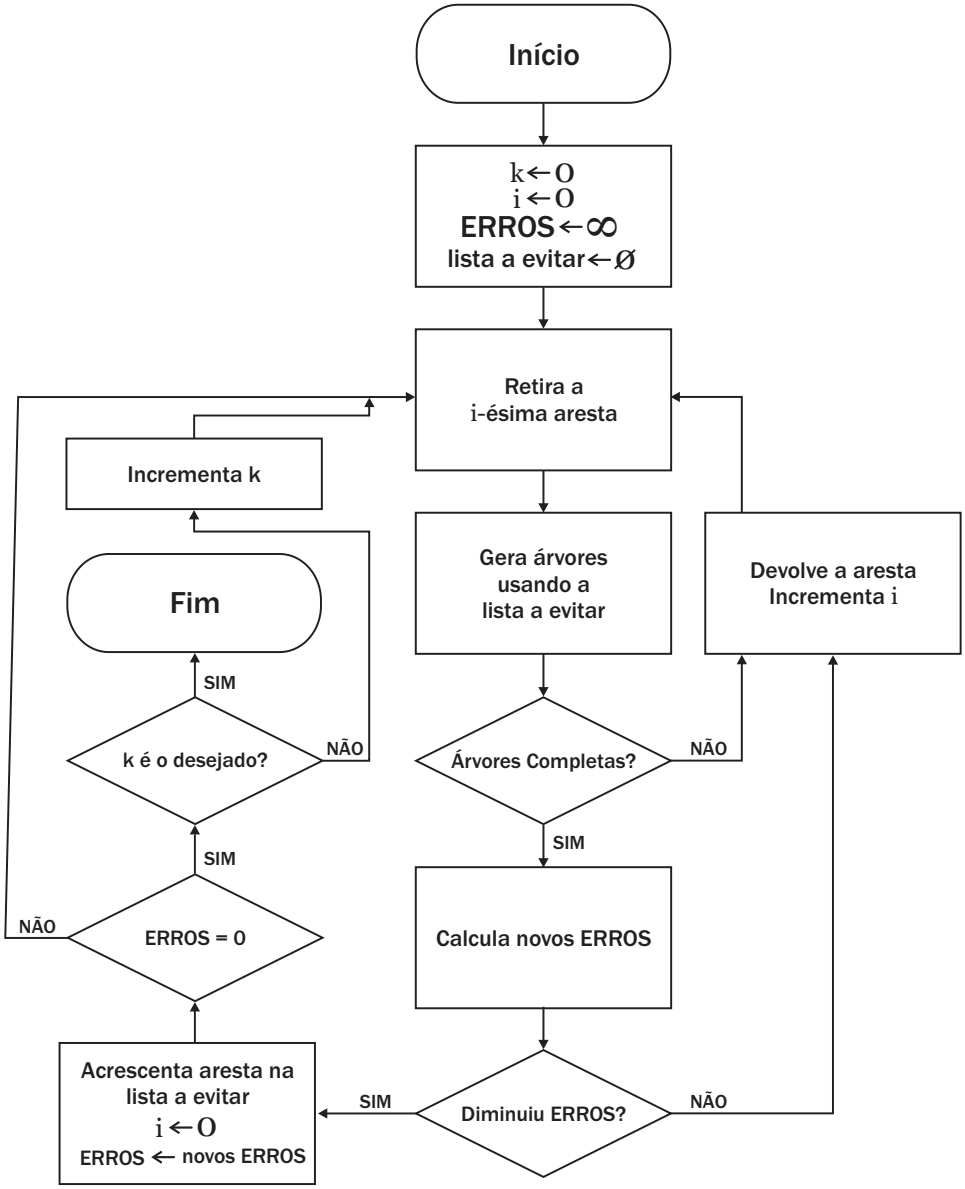


Figura 3.6: Fluxograma do algoritmo de *OptimIST*

grafo duplicado em uma dimensão $k+1$. O produto de Kronecker da matriz da figura 3.8 $d)$ é o resultado do produto de Kronecker das matrizes das figuras $a)$ com $a)$. Os passos $b)$ e $c)$ são mostrados somente para ilustrar as arestas equivalentes na figura. Assim como a figura 3.8 $i)$ é o resultado do produto de Kronecker com a matriz da figura $f)$ com ela mesma.

Os padrões de intervalos regulares de arestas a evitar aparecem justamente sobre as arestas que ligam os grafos na dimensão $k+1$, na parte superior à diagonal principal, visto que a parte inferior à diagonal principal possui sempre o mesmo padrão. Esse padrão é definido da

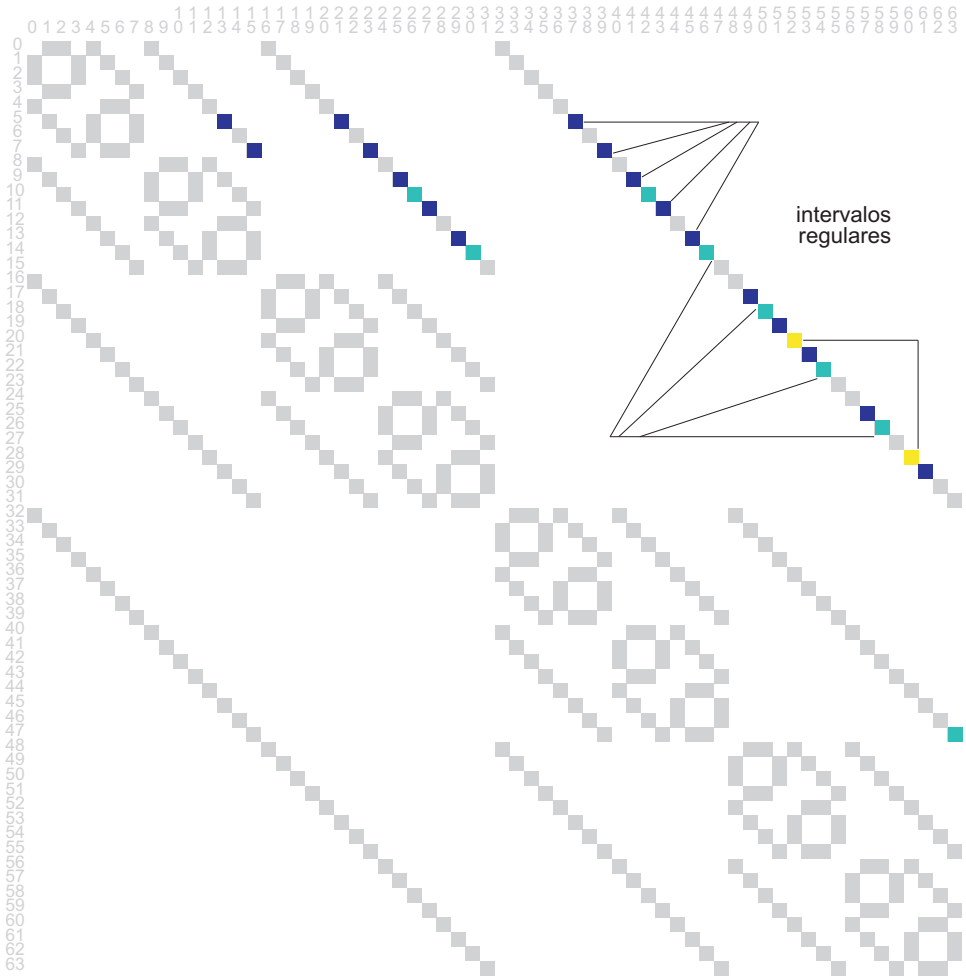


Figura 3.7: Intervalos regulares identificados no conjunto de arestas a serem evitadas (fig. do autor).

seguinte forma: todas as diagonais da parte inferior possuem uma única cor, ou seja, todos os elementos de cada diagonal pertencem à mesma árvore. Além disso todos os elementos destas diagonais são vértices folhas das k árvores, ou seja, vértices que não possuem filhos, figura 3.9.

É importante salientar que o fato dos padrões de arestas se concentrarem acima da diagonal principal deve-se à característica dos vértices abaixo da diagonal principal serem folhas. De fato, os caminhos de r a v em quaisquer pares de árvores devem ser disjuntos, desconsiderando r e v, logo, não importando onde as folhas estão localizadas, desde que seus caminhos até o vértice raiz sejam disjuntos.

Definindo-se o padrão de arestas a ser seguido antes de tentar convergir o algoritmo para

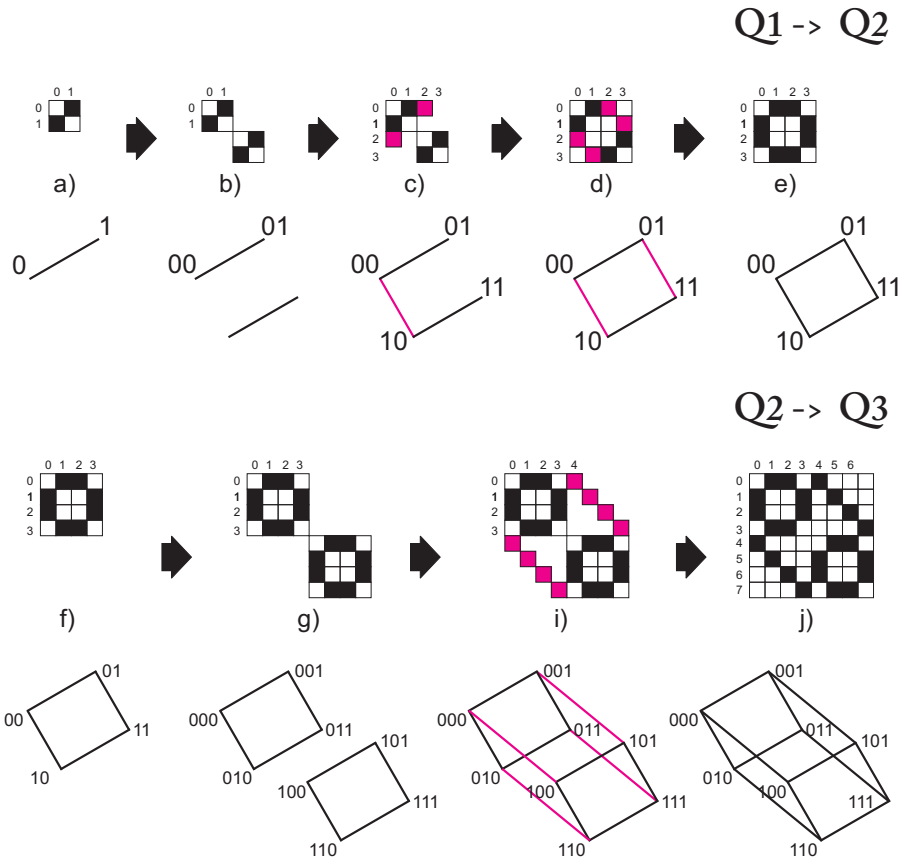


Figura 3.8: Produto de Kronecker representado graficamente (fig. do autor).

0 erros, somente retirando-se uma aresta a ser atribuída a uma árvore específica, resolveu o problema para Q_9 (512 vértices), e o método convergiu para 0 erros.

Realizando-se experimentos, foi também observado que a ordem das arestas retiradas influencia no resultado final. Fato o qual confirmou a dependência da escolha de uma aresta estar fortemente relacionada às arestas anteriores.

Outros experimentos, porém, apresentaram bons resultados. Um deles foi replicar na mesma árvore as arestas cuja remoção causou a diminuição de erros. Isso foi testado devido ao padrão encontrado nas arestas evitadas.

O objetivo principal deste trabalho é desenvolver um método capaz de produzir k árvores geradoras independentes utilizando-se um algoritmo simples que necessite de poucos recursos computacionais como processamento e memória. Partir do grafo completo e tentar construir a árvore com algum algoritmo ganancioso não é uma alternativa, devido à explosão combinatória

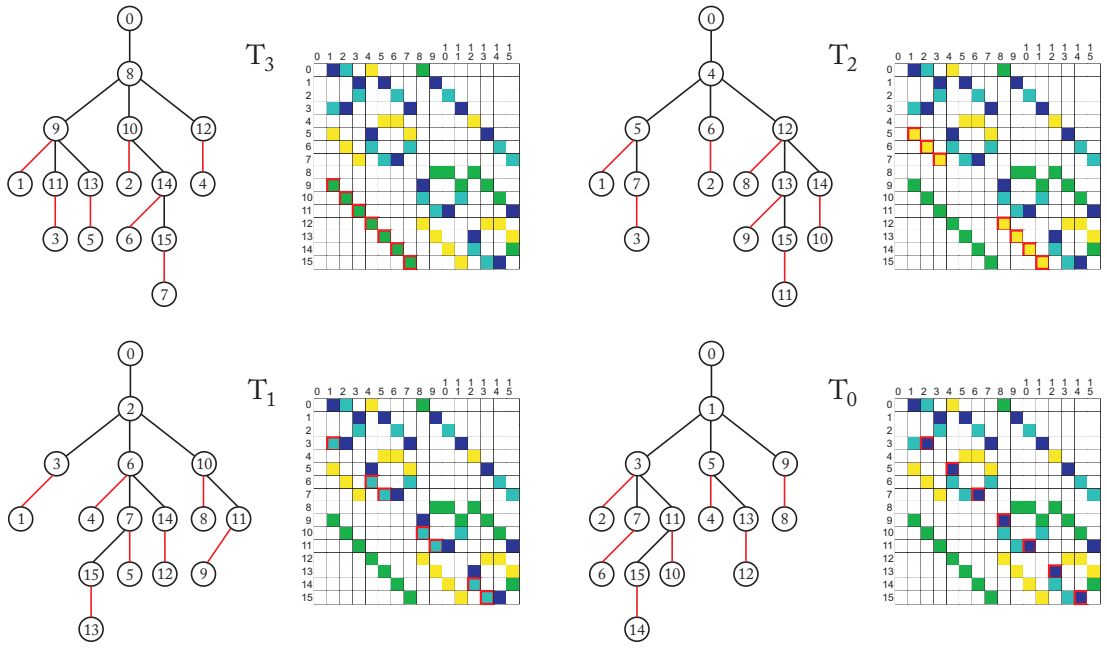


Figura 3.9: A parte inferior à diagonal principal representa todos os vértices folhas das k árvores.

de árvores possíveis.

O algoritmo apresentado no apêndice C, modificado para não utilizar as arestas a serem evitadas, apresenta um bom desempenho. Entretanto, o consumo de memória não é adequado para dispositivos mais limitados, principalmente pela necessidade em se manter grandes matrizes de adjacências em memória. Lembrando que, apesar do método de otimização ser executado uma única vez, ele somente gera as arestas a serem evitadas. A distribuição das arestas ainda se faz necessária no algoritmo a ser executado no dispositivo final.

No intuito de minimizar a quantidade de memória despendida pelo algoritmo, um novo processo foi desenvolvido de modo a utilizar a menor representação possível para informar se uma aresta foi utilizada em uma árvore, empregando um único *bit* por aresta. Para isso, as arestas do hipercubo são geradas sempre em uma ordem específica. Desse modo, para saber se uma aresta foi utilizada, basta verificar em um vetor de *bits* a posição específica da aresta. As arestas da forma (x, y), são consideradas diferentes das (y, x) quanto à distribuição.

A seguir são descritos os passos do algoritmo o qual gera as arestas em duas fases. Na primeira fase as arestas são geradas respeitando-se a regra na qual o índice do primeiro vértice

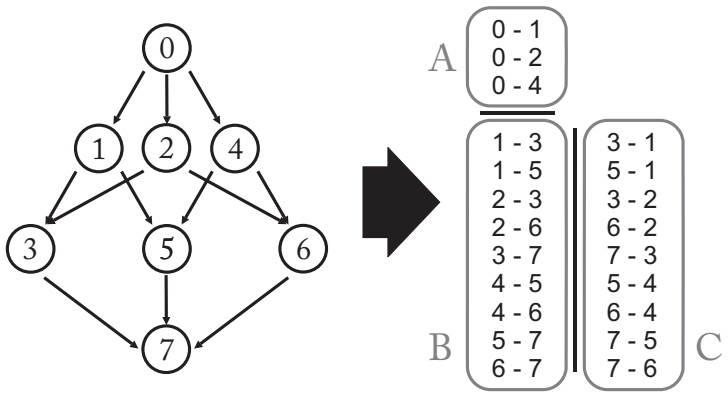


Figura 3.10: Geração da lista de arestas. O conjunto A é formado pelas arestas cujos vértices são o raiz e seus únicos filhos em cada árvore. O conjunto B contém as arestas que serão distribuídas primeiramente, seguidas pelo conjunto C.

da aresta não é maior do que o segundo. Seguindo essa regra nenhuma aresta repetida é gerada e todas as arestas criadas nesta fase são direcionadas no grafo, tal como mostrado na figura 3.10, conjuntos A e B.

A segunda fase é necessária para tratar as arestas que não foram geradas na primeira fase. Isso é feito para cobrir todas as arestas do grafo em ambos os sentidos, com exceção das que são incidentes ao vértice raiz. Para isso, estende-se a lista de arestas anterior acrescentando-se as arestas com a orientação inversa. As arestas que fazem par com o vértice raiz são ignoradas nessa etapa. O novo conjunto de arestas é representado pelo conjunto C na figura 3.10.

Para se construir k-ISTs (*Independent Spanning Trees*) sobre o hipercubo são necessárias $k(n-1)$ arestas. Isso é obtido após as duas etapas anteriores, em seguida, o problema passa a ser a distribuição das arestas dentre as k-ISTs. Cada árvore deve possuir $n-1$ arestas para cobrir n vértices do grafo.

O algoritmo percorre as arestas ao mesmo tempo que as gera e as arestas são geradas na mesma ordem para cada árvore. Embora eficiente quanto aos recursos computacionais e gerar k árvores arco-disjuntas (usando a notação na qual cada aresta do hipercubo é duplicada), o resultado inicial foi uma única árvore geradora vértice-independente das demais. Quanto às demais árvores, foi constatado que alguns vértices apresentavam problemas, ou seja, estes vértices eram comuns a alguns caminhos em árvores distintas.

```
input  : k
output: lista de arestas de cada árvore

t: índice da árvore  k: número de árvores, ordem do hipercubo ex. para Q_3 k=3
edgesOrder: ordem das arestas 0 ou 1 (normal e inversa)  vertices: número de
vértices n = 2^k  operadores bit a bit OR: ∨, e XOR: ⊕

for t ← 1 to k do
    for edgesOrder ← 0 to 1 do
        for n ← 0 to vertices do
            for b ← 1 ; i ← 1 to k do
                n2 ← n ∨ b;
                if n ≠ n2 then
                    if edgesOrder = 0 then
                        edge ←Edge(n,n ⊕ b);
                    else
                        edge ←Edge(n ⊕ b, n);
                    end
                    if ! UsedEdge(edge) then
                        tratamento especial da primeira aresta da árvore t;
                        if ! TailListContainsNodesOfEdge(edge) then
                            if ! BelongToAvoidSet(t, edge) then
                                UseEdge(edge);
                                AddToTree(t, edge);
                            end
                        end
                    end
                end
            end
        end
    end
end
```

Algoritmo 1: Algoritmo *EdgeDist* de distribuição das arestas dentre as k árvores, proposto neste trabalho.

Com o objetivo de se resolver esse problema um novo método foi desenvolvido. Este método consiste na observação de alguns padrões na matriz de adjacências que, caso evitados, produzem todas as árvores independentes a partir dos hipercubos de baixa ordem analisados, entre oito e trinta e dois vértices, como exemplo. Entretanto a observação destes padrões somente pode ser realizada para pequenas grandezas.

Importante salientar que a observação de tais padrões tem como resultado final uma lista de arestas a serem evitadas em árvores específicas. Tais arestas são obtidas por um método de otimização, descrito a seguir e, a partir desse resultado, um conjunto de tuplas (aresta e árvore da qual deve ser excluída) é gerado. Esse conjunto é acrescentado de forma estática ao algoritmo final que faz uso dele para gerar as arestas das árvores.

O algoritmo *OptimIST* (figura 3.6) obteve sucesso até $Q_8(256$ vértices). Para $Q_9(512$ vértices), entretanto, os primeiros problemas apareceram, o algoritmo não convergia e os erros pararam em 3 vértices problemáticos, ou seja, que se repetiram em mais de um caminho, portanto, não sendo independentes.

Como mencionado anteriormente, identificou-se alguns padrões de arestas a serem evitadas, sendo que esses padrões se repetiam nos quadrantes dos hipercubos de menor ordem, figura 3.11. Esses padrões foram então explorados na lista de arestas a serem evitadas sendo que, além de se repetirem nos quadrantes, eles representam um intervalo regular.

O fato de definir esse padrão de arestas a ser seguido pelo algoritmo, antes de tentar convergir somente retirando uma aresta a ser atribuída a uma árvore específica, resolveu o problema para Q_9 $(512$ vértices), e o método convergiu para 0 erros. O método também convergiu para 0 erros para Q_{10} $(1024$ vértices), e os testes pararam neste ponto devido a busca de novas alternativas que produzissem resultados mais rapidamente.

O processo de otimização, o qual é executado uma única vez, tem sido custoso, para Q_{11} (2048 vértices) cada erro é eliminado após cerca de quatorze horas de execução no ambiente de testes atual (Mac OSX, 2.4GHx Intel Core 2 Duo, 4GB RAM). A eliminação de um erro ocorre pela escolha de caminhos alternativos que façam com que este vértice não seja compartilhado por mais de um caminho. Algumas remoções de arestas causam uma maior redução de erros,

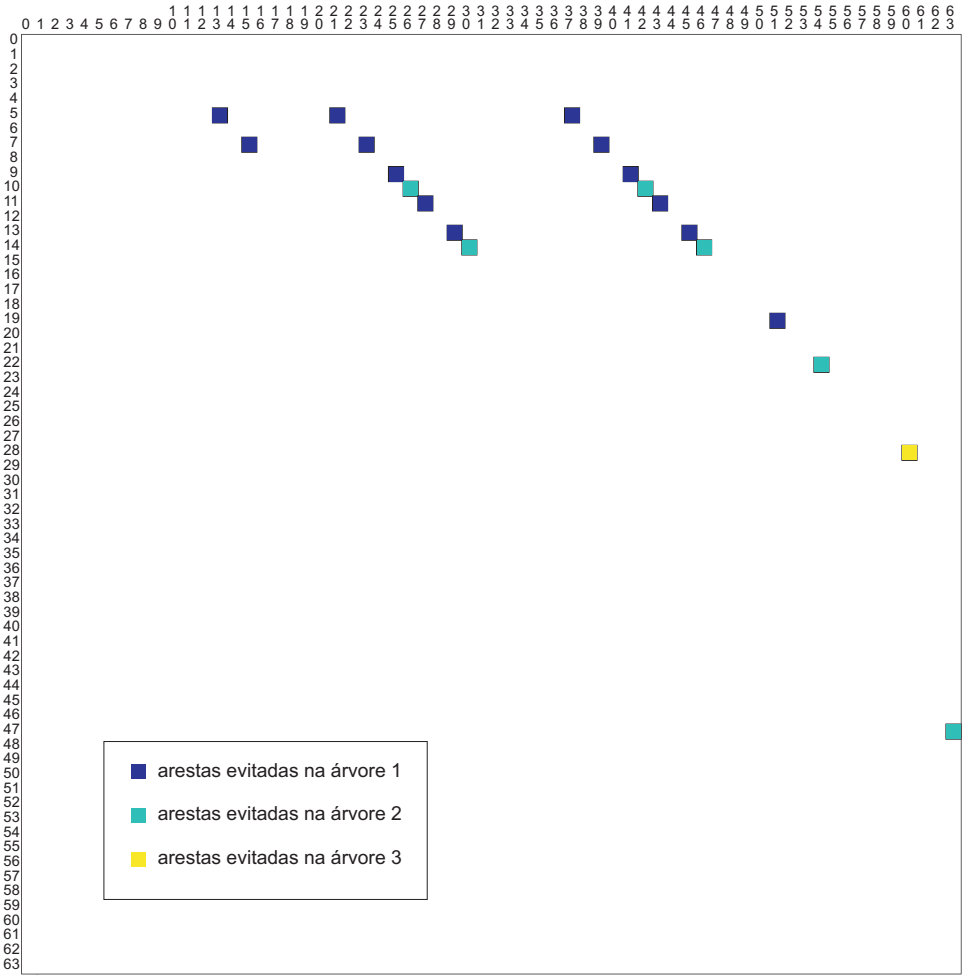


Figura 3.11: Ilustração da lista de arestas a serem evitadas para Q_6

mas em média, um erro é removido por vez.

O algoritmo de otimização foi então modificado para se valer dos padrões identificados, os quais são pré-calculados para até $k+1$. A nova versão do algoritmo de otimização é mostrada na figura 3.12.

Quanto à distribuição de arestas tem-se dois algoritmos, como mencionado anteriormente, eles não garantem árvores geradoras independentes por si só, são somente utilizados para distribuir as arestas. O primeiro algoritmo é o *EdgeDist* (algoritmo 1), e o segundo, *EdgeDistAdj* é descrito em detalhes no apêndice C.

Apesar do algoritmo *EdgeDist* apresentado para geração das árvores ser simples e eficaz quanto ao uso de recursos devido ao fim planejado, para *NoCs*, para o método de otimização o cálculo das árvores foi feito por um algoritmo com uma característica diferente do algoritmo

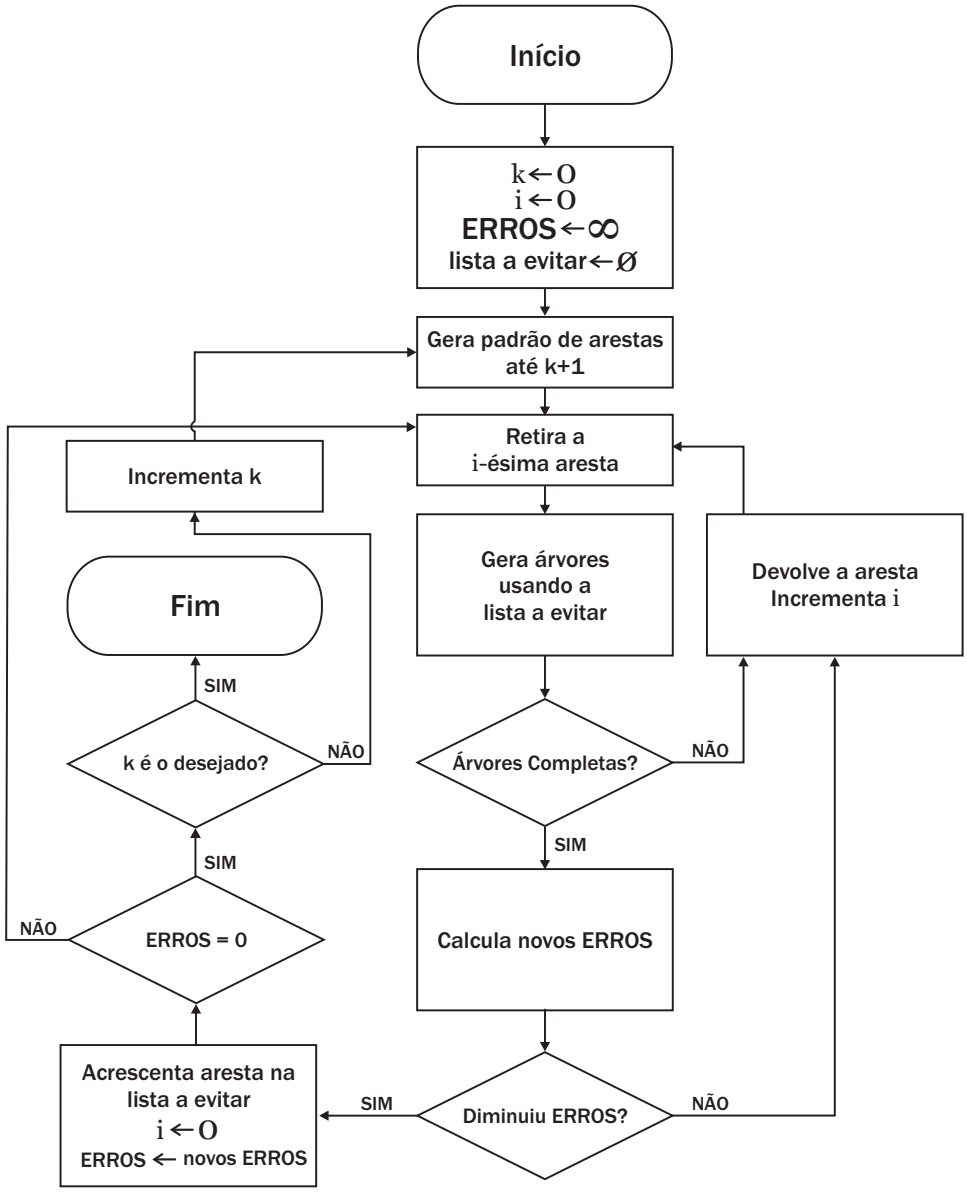


Figura 3.12: Fluxograma do algoritmo de otimização utilizando padrão pré-calculado

anterior. O algoritmo de distribuição de arestas utilizado na fase de otimização faz uso somente da matriz de adjacências para computar as árvores e funciona como descrito no apêndice C. Tal algoritmo foi denominado *EdgeDistAdj*.

O algoritmo *EdgeDistAdj* apresentou resultados melhores quanto ao tempo de criação das árvores, no método de otimização, entretanto consumindo mais memória que o algoritmo *EdgeDist*, $O(n^2)$ para *EdgeDistAdj* e $O(n)$ para *EdgeDist*. O consumo de memória de $O(n^2)$ deve-se ao fato de ser necessário manter a matriz de adjacências $n \times n$ em memória.

Sendo assim, optou-se por continuar utilizando-se o algoritmo *EdgeDist* como o algoritmo de distribuição a ser executado no dispositivo final, pelo fato de consumir menos recursos.

3.2 Algoritmo Minimalista - MinimalIST

Apesar do algoritmo de distribuição de arestas *EdgeDist* (algoritmo 1) ser eficaz quanto à utilização de recursos, o algoritmo de otimização *OptimIST* utilizado para computar a lista de arestas a serem evitadas é extremamente custoso em relação ao tempo consumido. Os resultados obtidos foram computados até 1024 (2^{10}) vértices devido ao cronograma da tese, sendo que o processamento para grandezas maiores foi estimado em meses na plataforma de testes utilizada.

Antes de continuar, faz-se necessário a apresentação de alguns teoremas propostos:

Seja um grafo G um par (V, E) onde E é um subconjunto de $\{\{x, y\} : x, y \in V; x \neq y\}$.

Definição 1: [69] A rotação de um vértice é uma rotação cíclica a direita do seu endereço, $R_o(i) = (i_0 i_{n-1} ... i_2 i_1)$. A rotação de um grafo G é definida como $R_o(G) = G(R_o(V), R_o(E))$, onde $R_o(V) = \{R_o(i) | \forall i \in V\}$ e $R_o(E) = \{(R_o(i), R_o(j)) | \forall (i, j) \in V\}$. Além disso, R_o^{-1} é a rotação a esquerda e $R_o^k = R_o \circ R_o^{k-1}$ para qualquer k.

Definição 2: [69] Translação de um vértice i por s é a operação ou-exclusivo $(\oplus)$ bit a bit dos endereços, $T_r(s, i) = c$. A translação de um grafo G em relação a s é definida por $T_r(s, G) = G(T_r(s, V), T_r(s, E))$, onde $T_r(s, V) = \{T_r(s, i) | \forall i \in V\}$ e $T_r(s, E) = \{(T_r(s, i), T_r(s, j)) | \forall (i, j) \in E\}$.

Definição 3: [69] O período de um número binário i denotado por $P(i)$ é definido como $P(i) = min_{m>0} R_o^m(i) = i$. Se $P(i) < n$ o número binário é *cíclico* caso contrário ele é *não-cíclico*. O período do número binário (010101) é 2. Dado um vértice fonte o número cíclico é definido com base nele, por exemplo, o vértice (01000) é *cíclico* em relação ao vértice fonte (00001).

Definição 4: [69] Uma árvore geradora binomial 0-nível tem um vértice. Uma árvore geradora binomial n-níveis é construída pela adição de uma aresta entre as raízes das duas árvores de $(n-1)$-níveis fazendo de uma delas a nova raiz.

A árvore geradora binomial com raiz em um vértice s, $T^{SBT}(s)$, é definida como: Seja p tal que $c_m = 0, \forall m \in \{p+1, p+2, ..., n-1\} \equiv M^{SBT}(c)$ e seja $p = -1$ se $c = 0$. O conjunto $M^{SBT}(c)$ é o conjunto de zeros à esquerda de c. Então,

$$children^{SBT}(i,s) = \{(i_{n-1}, i_{n-2}, ..., i_m, ..., i_0)\}, \forall m \in M^{SBT}(c), \tag{3.6}$$

$$parent^{SBT}(i,s) = \begin{cases} \phi, & se \quad i = s; \\ (i_{n-1}, i_{n-1}, ..., i_p, ..., i_0), & se \quad i \neq s. \end{cases} \tag{3.7}$$

Teorema 3.2.1. *Em relação ao hipercubo Q_n, qualquer árvore geradora independente enraizada em qualquer vértice $r \in V$ pode ser obtida pela operação ou-exclusivo* bit *a* bit *com todos os vértices das árvores com raiz em 0.*

Teorema 3.2.2. *Dada uma árvore inicial, T_0, enraizada no vértice 0, k árvores $T_1, T_2, ..., T_k$, todas com raiz no vértice 0, podem ser geradas a partir de permutações de vértices. Seja π_i a permutação que gera T_i. Dado um vértice x de T_0, $\pi_i(x)$ é o vértice equivalente em T_i. E π_i é definida como:*

$$\pi_i(x) = \begin{cases} 2^i \times x \mod 2^k - 1, & se \quad x \neq 2^k - 1 \\ x, & se \quad x = 2^k - 1 \end{cases} \tag{3.8}$$

Se T_0 for construída pelo algoritmo MinimalIST*, e as árvores $T_1, T_2, ..., T_k$ construídas pela operação π acima então o conjunto $T_0, T_1, ..., T_k$ é um conjunto de árvores geradoras independentes sobre Q_k.*

Prova: A permutação π_i é equivalente à rotação à esquerda da árvore 0 ($R_o^{-1}(T_0)$).

Lema 3.2.3. *[69] Rotações, reflexões e translações do grafo preservam a distância* Hamming *entre os vértices. A operação de rotação R_o^k faz o mapeamento de cada aresta na dimensão d para a dimensão $(d-k) \mod n$. Já a operação de reflexão mapeia cada aresta na dimensão d para a dimensão $n-1-d$. A translação, por sua vez, preserva a dimensão de cada aresta.*

A rotação e a reflexão preservam a direção de cada aresta. A translação reverte a direção de todas as arestas na dimensão na qual $s_m = 1, m \in D$

Corolário 3.2.4. *Como a topologia do grafo permanece inalterada sob as operações de rotação, reflexão e translação [69] e o hipercubo é um grafo simétrico, a operação de permutação de* T_0*, nesse caso, equivale à operação de rotação,* $R_o(T) = T(R_o(V))$*, na qual* $R_o(V) = \{R_o(i) | \forall i \in V\}$ *mapeia o conjunto original de vértices para uma nova dimensão preservando a distância de* Hamming *entre os vértices. Desse modo, nenhum ancestral de um vértice será repetido e portanto, os caminhos serão disjuntos.*

□

Teorema 3.2.5. *Todas as árvores geradoras independentes, construídas pelo algoritmo* MinimalIST *são ótimas em termos de tamanho médio dos caminhos [109].*

Prova: O algoritmo produz nada menos que uma árvore geradora binomial, com o vértice $k/2$ aplicado como termo s na translação $T_r(s, G)$. Como a árvore geradora binomial tem a menor distância entre o vértice raiz e todos os outros vértices da árvore e devido ao fato da transformação para uma *IST* acrescentar somente um nível à árvore, então o comprimento médio dos caminhos é ótimo. □

Teorema 3.2.6. *Seja* i *um número binário qualquer com* k *bits, se* k *é primo não existe nenhum vértice cíclico de modo que alguma rotação produza o mesmo vértice dentre todos os* 2^k *vértices.*

Prova: Pela definição 3, se $P(i) < n$ o número binário i é *cíclico* caso contrário ele é dito *não-cíclico*. A prova vem por contradição. Seja n o número de bits em i. Assuma que sendo n primo existe ao menos uma rotação de modo que $R_o^m(i) = i$. Qualquer vértice *cíclico* é caracterizado por uma repetição regular de um padrão de tamanho p. Logo, o padrão deve aparecer no número binário q vezes. Então:

$$p \times q = n$$
$$p = \frac{n}{q}$$

Como n é primo, os únicos valores permitidos para q são 1 e n.

$$p = \begin{cases} 1, & se \quad q = n; \\ n, & se \quad q = 1. \end{cases} \tag{3.10}$$

Logo $P(i) = n$ e se $P(i) < n$ o número binário é *cíclico* caso contrário ele é *não-cíclico*, então qualquer número binário com n bits, sendo n primo é *não-cíclico* em relação a ele mesmo. Ou seja, não possui nenhum número cíclico a ele de modo que qualquer rotação produza o mesmo número binário. □

Entretanto, durante o desenvolvimento da demonstração do teorema anterior chegou-se a outro teorema, o qual simplifica a construção das k árvores geradoras:

Teorema 3.2.7. *Todas as árvores geradoras independentes enraizadas em um vértice $r \in V$ podem ser obtidas a partir de uma única árvore com vértice raiz r.*

Deste teorema concluímos que somente é necessário gerar uma única árvore para se obter todas as k árvores geradoras sobre o hipercubo. Sendo que para $r = 0$ a seguinte função com $t = (1, ..., k)$ pode ser utilizada para gerar as árvores:

$$T_t(x) = \begin{cases} 2^t \times x \mod 2^k - 1, & x \neq 2^k - 1 \\ x, & x = 2^k - 1 \end{cases} \tag{3.11}$$

Para simplificar a geração das árvores, em relação aos recursos computacionais utilizados, evitando a necessidade da presença de circuitos complexos para efetuar operações de multiplicação e potenciação, a função acima foi substituída por um simples deslocamento de *bits* à esquerda com o índice da árvore usado como a quantidade de *bits* a ser deslocada.

$$T(x,t) = \left\{ \; x \ll t \right. \tag{3.12}$$

O algoritmo de distribuição de arestas pôde ser então alterado da seguinte forma: a iteração até k foi removida, visto que, somente é necessário gerar uma única árvore. A verificação de arestas já utilizadas também foi removida, visto que, para a primeira árvore não é necessária (algoritmo 2).

```
input  : k
output: lista de arestas da primeira árvore, 0 como raiz

 k: 2^k é a ordem do hipercubo
 edgesOrder: ordem das arestas (normal=0 or inversa=1)
 vertices: número de vértices (2^k)
 operadores bit a bit OR: ∨, e XOR: ⊕

for edgesOrder ← 0 to 1 do
    for n ← 0 to vertices do
        for b ← 1 ; i ← 1 to k do
            n2 ← n ∨ b;
            if n ≠ n2 then
                if edgesOrder = 0 then
                    edge ←Edge(n,n ⊕ b);
                else
                    edge ←Edge(n ⊕ b, n);
                end
                if IsTheFirstEdge(edge) then
                    AddToTree(0, edge);
                end
                if ParentIsInTheTree(edge) then
                    AddToTree(0, edge);
                end
            end
            b ← b(rll)1;
        end
    end
end
```

Algoritmo 2: A geração de todas as k árvores é removida do algoritmo *EdgeDist*, assim como a verificação das arestas já utilizadas e das arestas a serem evitadas. Este algoritmo modificado é denominado *MinimalIST*, utilizado para criar a árvore T_0. Basta criar T_0 visto que todas as outras árvores são derivadas dela e obtidas pela operação de deslocamento de *bits* à esquerda.

Com a utilização desta nova versão do algoritmo de distribuição de arestas, *MinimalIST*, o método de otimização apresentado anteriormente foi removido e o problema resolvido para qualquer n.

Uma das vantagens do algoritmo *MinimalIST* é a utilização de um único *bit* para marcar a presença de um pai na árvore. Outra vantagem é a utilização da operação de deslocamento de *bits* à esquerda, como exemplificado na figura 3.13, aplicada ao algoritmo de Obokata et al. [86], para se obter árvores geradoras ótimas.

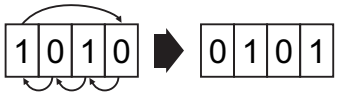


Figura 3.13: Rotação de *bits* à esquerda.

No exemplo de Obokata, a árvore geradora T_4 é responsável pelo problema, figura 3.14, visto que a soma das distâncias do vértice raiz a todos os outros é a maior dentre as árvores.

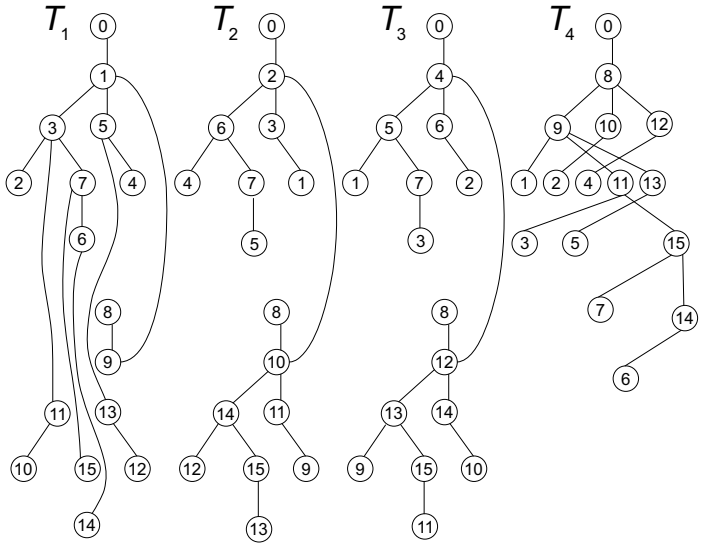


Figura 3.14: Árvore T_4 não ótima em relação à altura utilizando algoritmo de Obokata et al. [86]

Entretanto, se a árvore T_4 for recriada a partir da árvore T_3 pela operação de rotação dos *bits* de seus índices, as árvores de Obokata tornam-se ótimas.

O artigo de Yang et al. [124] também menciona a possibilidade da rotação dos *bits* ser

utilizada para a geração dos vértices das outras árvores. Entretanto, neste artigo é mencionado o fato como uma característica das árvores produzidas sendo que as árvores resultantes são isomórficas, uma vantagem para os esquemas de *broadcast*. Aqui, porém, utilizou-se esta característica para a construção das árvores e foi mostrado também como gerar as árvores a partir de qualquer árvore geradora (pelo deslocamento de *bits* à esquerda), sendo que todas as árvores produzidas são isomórficas para qualquer algoritmo que produza as árvores. Ou seja, qualquer algoritmo que produza pelo menos uma árvore ótima, em relação a altura, terá todas as características das árvores produzidas pelo método de Yang et al.

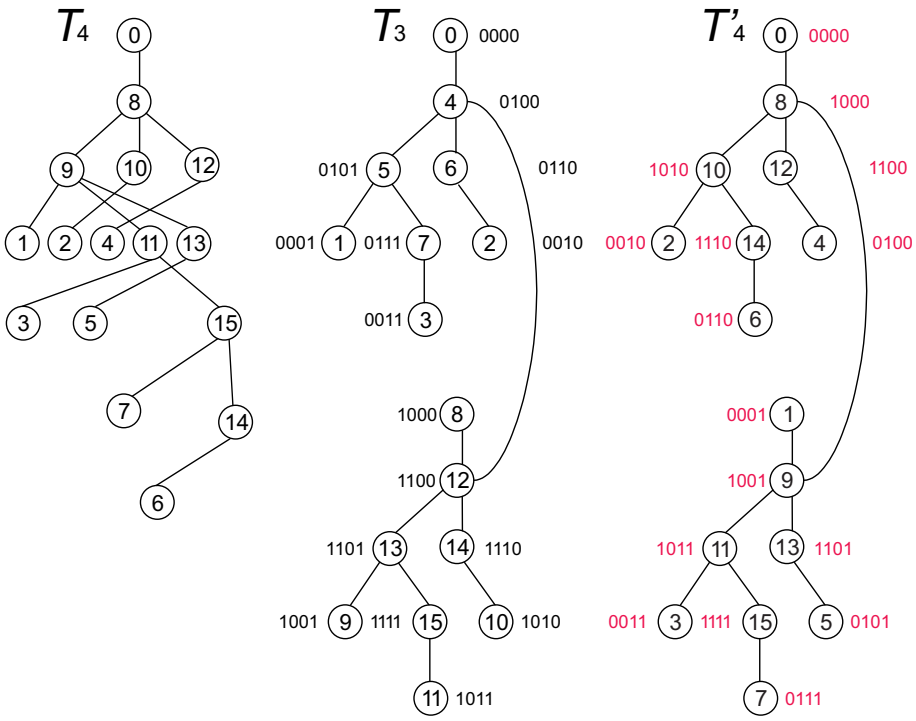


Figura 3.15: Aplicando-se a rotação de *bits* sobre as árvores de Obokata.

3.3 Estrutura de Dados

Embora o processo de cálculo das árvores geradoras independentes apresentado neste trabalho seja simples, pode haver necessidade de uma estrutura compacta que armazene as árvores de forma eficiente, principalmente para sua utilização no roteamento das *NoCs*.

Nesta seção é apresentado o modelo de persistência das árvores geradoras cuja principal vantagem é o espaço requerido para seu armazenamento. Em Yang et al. a quantidade de memória necessária para persistir as árvores é da ordem $O(k.n)$.

O modelo sugerido aqui apresenta a mesma ordem $O(k.n)$. Em cada linha da matriz têm-se os filhos, que podem ser calculados da seguinte forma: para cada *bit* ativo na linha i o filho é calculado pela fórmula $i \oplus 2^j$, onde i representa a linha e j a coluna. Então para o vértice 0 temos o vértice 4 como único filho.

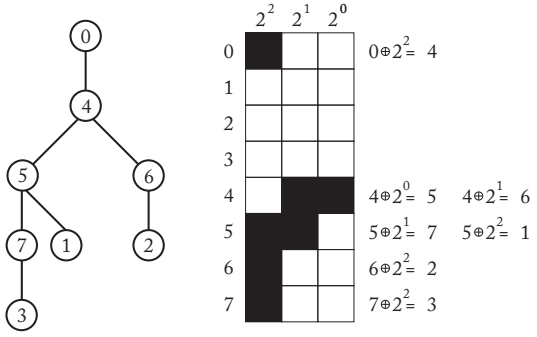


Figura 3.16: Modelo sugerido para persistência das árvores

O número de *bits* ativos em cada matriz representa o número de arestas da árvore, ou seja $n-1$. O modelo de armazenamento aqui apresentado se vale da propriedade do hipercubo na qual as strings binárias, que representam os vértices, diferem em uma única coordenada. Logo, o modelo de persistência apresentado pode ser utilizado para quaisquer árvores que respeitem tal restrição.

O espaço requerido para árvores pode ser reduzido para $(k-1).n$, pela utilização da seguinte fórmula proposta neste trabalho:

$$T_k = g(f(\sum_{i=1}^{k-1} T_i)) \tag{3.13}$$

$$f(x) = \begin{cases} 1, & i = 2^j \\ x, & i \neq 2^j. \end{cases} \tag{3.14}$$

$$g(x) = \begin{cases} 1, & i, j = 0 \\ 0, & i, j = 1 \end{cases} \tag{3.15}$$

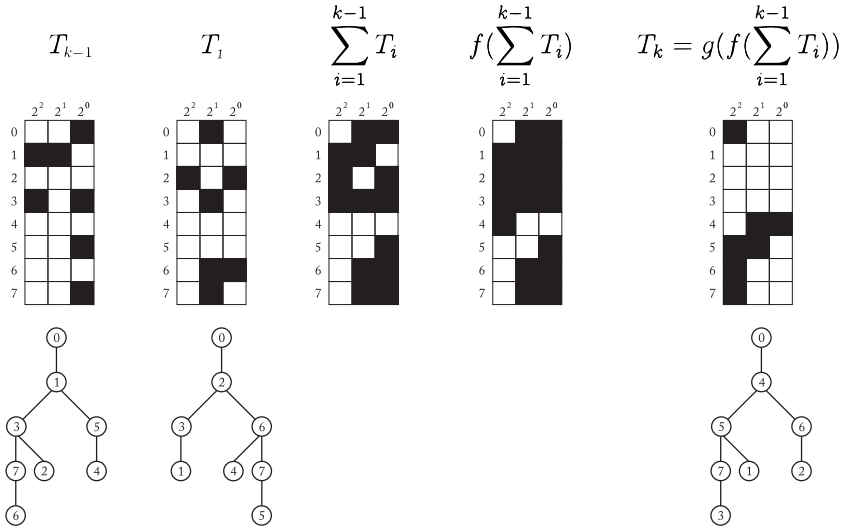


Figura 3.17: Cálculo de T_3 de Q_3

A equação acima é resultado do fato que a superposição das informações de adjacências das k árvores de Q_k resulta em uma matriz sem nenhum valor 0. Sendo que a última árvore pode ser obtida pela negação da soma de todas as outras. Isso se deve ao fato de que a somatória de todas as árvores preenche a matriz de adjacências em todas as potências de 2 com exceção das diagonais, onde $i = j$.

3.4 Discussão

Neste capítulo foram apresentados dois algoritmos para a construção de árvores geradoras independentes sobre hipercubos, o *OptimIST* e o *MinimalIST*. O algoritmo *OptimIST* se

Algoritmo	Autores	Tempo	Memória (bits)
IST	Obokata et al.	$O(kN)$	$O(k^2N)$
OIST	Tang et. al.	$O(kN)$	$O(k^2N)$
GEN-PARENTS	Yang et. al.	$O(kN)$	$O(k^2N)$
MinimalIST	Silva et. al.	$O(kN)$	$O(N)$

Tabela 3.1: Complexidade dos algoritmos para construção de árvores geradoras independentes.

vale de um método de otimização que incrementalmente reduz o número de erros causados por atribuições de arestas específicas a certas árvores. O algoritmo *MinimalIST* se vale da construção de uma única árvore geradora, sendo que as árvores restantes são dela derivadas pela operação de deslocamento de *bits*. Também foi apresentado um modelo compacto de persistência das árvores que pode ser utilizado quando necessário.

A tabela 3.1 contém a complexidade dos algoritmos de construção de árvores geradoras independentes analisados, sendo N o número de vértices do hipercubo e $k = log(N)$. O algoritmo *OptimIST* não é determinístico, não há como estimar a complexidade em relação ao tamanho da entrada. As arestas geradas pelo algoritmo *OptimIST* são, posteriormente, utilizadas de forma estática no algoritmo *EdgeDist*. Como o algoritmo funciona comprovadamente somente para $k \leq 10$, ele não foi colocado na tabela comparativa, visto que não cobre todos os valores de k.

Outro fato importante é que algoritmos de roteamento em *NoCs* devem ser compactos, o uso de tabelas de roteamento mesmo que compactas, deve ser evitado quando possível. O algoritmo deve também ser eficaz ao processar informações relacionadas ao roteamento, pois em alguns casos, a decodificação das informações de roteamento pode ser mais demorada do que a consulta a uma tabela de roteamento convencional [39].

Como nos outros algoritmos de roteamento analisados: *xy*, *oddeven*, *westfirst*, etc., o roteamento *IST*, baseado na utilização das árvores geradoras independentes, deve ser capaz de fazer o roteamento com o mínimo de informação possível. Como o algoritmo alterna entre árvores e cada árvore possui um caminho distinto, a informação relativa a árvore deve estar presente em pelo menos um vértice, no vértice fonte da mensagem. Idealmente, qualquer vértice subsequente deveria ser calculado somente em relação ao vértice corrente, e o vértice

de destino final, como no roteamento *ECUBE*, por exemplo. Ou seja, idealmente deve existir um mapeamento dos caminhos disjuntos para um algoritmo de roteamento que possa calcular o próximo salto a partir de uma única informação de destino. Obviamente a quantidade mínima de memória necessária é o endereço do nó destino da rede, O(log(N)) *bits*.

Em [96, 78] um método de roteamento compacto conhecido como roteamento de intervalo foi apresentado, onde é possível fazer o roteamento em qualquer rede utilizando $O(log(N))$ *bits* por *link* de comunicação. Entretanto, no caso de caminhos mínimos o algoritmo pode falhar e requerer tantos *bits* quanto as tabelas de roteamento convencionais [44].

O algoritmo *MinimalIST* constrói as árvores por inteiro, além de ser $O(N)$ *bits*. No roteamento ponto a ponto não é necessária a construção total da árvore, bastando somente a construção do caminho de roteamento do vértice fonte ao vértice destino. Devido a isso, apresentamos uma técnica de construção de rotas disjuntas, que geram os mesmos caminhos presentes nas árvores geradoras independentes, cuja entrada é composta pelo vértice fonte, índice da árvore a ser utilizada e vértice destino final.

As árvores geradoras independentes podem ser construídas utilizando-se o roteamento *eCube*. Em tal roteamento cada caminho pode ser computado sem qualquer conhecimento prévio das outras árvores, ou mesmo de outros caminhos da árvore no qual o vértice fonte se encontra. Para mostrar como isso pode ser feito, primeiramente é necessário explicar como gerar a $(k-1)$-ésima árvore de Q_k a partir da árvore geradora binomial T^{SBT}. A figura 3.18 ilustra a operação de translação $T_r(n/2, T^{SBT})$ utilizando-se Q_4 como exemplo.

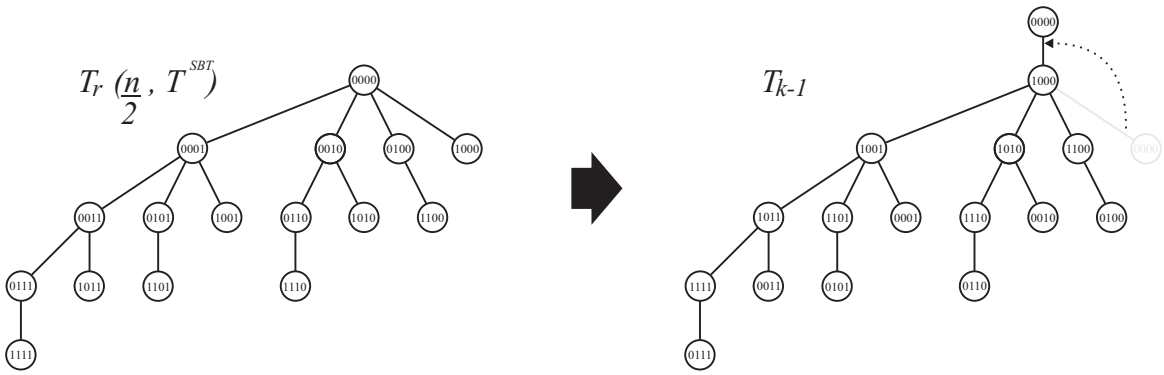


Figura 3.18: Obtendo-se $T3$ a partir de T_{SBT} sobre Q_4

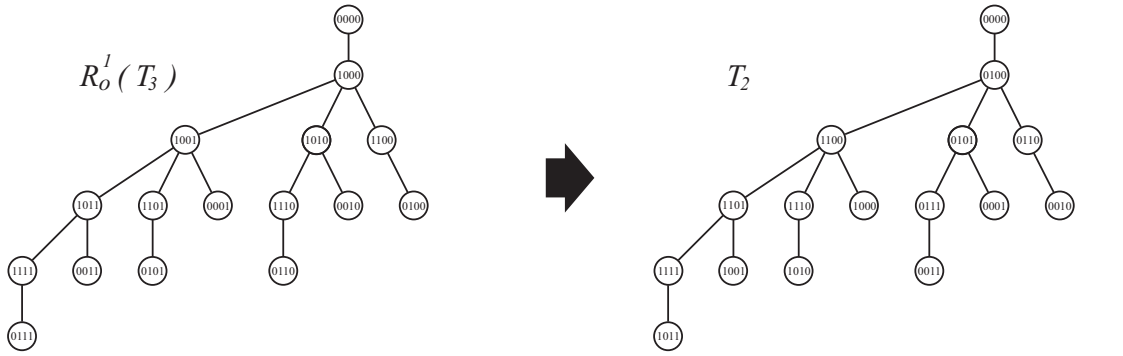
Algoritmo	Autores	Tempo	Memória (bits)
IST	Obokata et al.	$O(kN)$	$O(k^2N)$
OIST	Tang et. al.	$O(kN)$	$O(k^2N)$
GEN-PARENTS	Yang et. al.	$O(kN)$	$O(k^2N)$
MinimalIST	Silva et. al.	$O(kN)$	$O(N)$
EcubeIST	Silva et. al.	$O(kN)$	$O(k)$

Tabela 3.2: Complexidade dos algoritmos para construção de árvores geradoras independentes. O algoritmo *EcubeIST* necessita somente de $k = log(N)$ *bits* para gerar as árvores geradoras independentes.

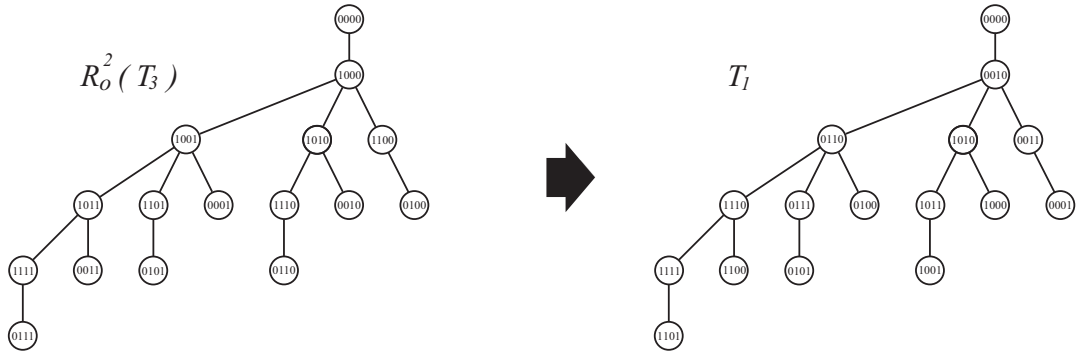
De posse da $(k-1)$-ésima árvore, pode-se construir as outras árvores geradoras independentes restantes utilizando-se a operação de rotação $R_o(T_{k-1})$, a rotação de *bits* à esquerda ou à direita pode ser utilizada. A operação de rotação da árvore pode ser definida de forma similar como a rotação de um grafo, porém mais restritiva, visto que a rotação da árvore produz arestas que não pertencem à árvore original. Logo $R_o(T,(V,E)) = T(R_o(V))$, onde $R_o(V) = \{R_o(i)|\forall i \in V\}$. Além disso, deve-se levar em conta que operação de rotação de *bits* além de alterar o caminho a ser utilizado, altera os vértices fonte e destino, com exceção dos vértices com todos os valores em 0 ou 1 na representação binária dos índices.

Na figura 3.19 pode-se ver a construção das outras *ISTs* a partir da rotação de T_{k-1}.

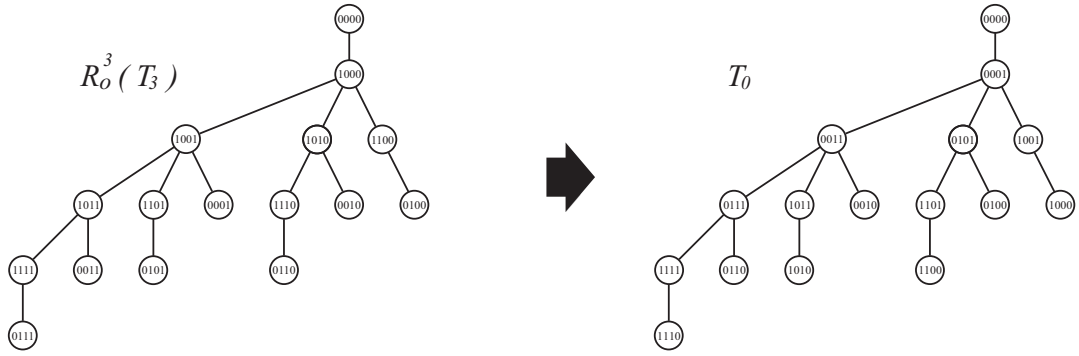
Por exemplo, o caminho do vértice 13 ao vértice 14 na árvore T_3, figura 3.20 pode ser calculado por $source \oplus (0 \,||\, eCube(2^{k-1}, source \oplus target))$, sendo || a operação de concatenação e $\oplus$ a operação binária ou-exclusivo. Como somente é acrescentado um salto à rota convencional que seria utilizada pelo algoritmo *eCube*, sendo que o *eCube* utiliza somente $log(N)$ *bits* de memória de roteamento, que é o endereço do destino final, pode-se construir as k-árvores geradoras independentes sobre o hipercubo utilizando-se $O(log(N))$ *bits*. O tempo requerido para a construção da árvore é $2Nlog(N)$ visto que deve-se calcular o caminho na árvore T_{k-1} primeiramente, antes de converter o caminho para a árvore desejada através da operação de rotação de *bits*. Logo, o tempo requerido para execução do algoritmo é $O(Nlog(N))$. Retomando a tabela de complexidade, acrescentando o algoritmo que calcula as árvores se valendo do *eCube* e denominando tal algoritmo como *EcubeIST*, lembrando que $k = log(N)$, temos (tabela 3.2):



(a)



(b)



(c)

Figura 3.19: Transformando T_3 nas outras ISTs de Q_4. (a) $T_3 \to T_2$ (b) $T_3 \to T_1$ (c) $T_3 \to T_0$.

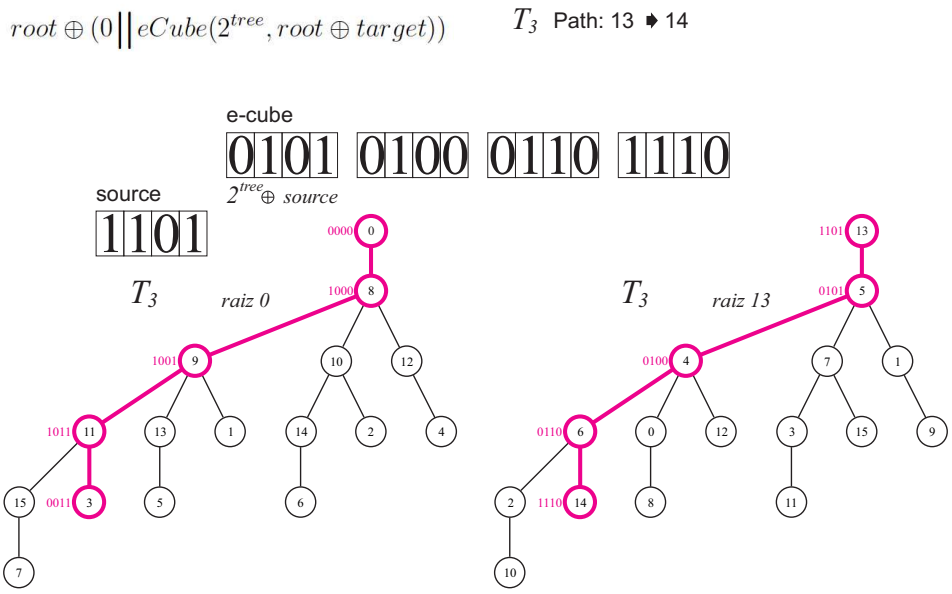


Figura 3.20: Rota do vértice 13 ao 14 na árvore T_3

CAPÍTULO 4

RESULTADOS EXPERIMENTAIS

Neste capítulo são apresentados os resultados obtidos com a realização dos experimentos, juntamente com a discussão das contribuições advindas da aplicação das técnicas que compõem o método desenvolvido, assim como dificuldades e problemas encontrados.

4.1 Algoritmo MinimalIST

Um comparativo com o principal trabalho [124] relacionado à geração de árvores geradoras independentes sobre hipercubos é descrito a seguir. Tal trabalho foi considerado principal, por seus resultados em relação às alturas das árvores e sua complexidade $O(kN)$, sendo N o número de vértices. $O(kN)$ é a mesma complexidade $O(N \log N)$ visto que $k = \log N$, preferiu-se utilizar $O(kN)$ por simplicidade.

A tabela 4.1 apresenta os dados sobre o consumo de memória do algoritmo proposto comparado com o algoritmo apresentado por Yang et al. [124]. O espaço em memória requerido pelo algoritmo de Yang et al., que é:

$$O(kN) \tag{4.1}$$

Como nosso espaço de memória é medido em *bits*, para comparar com o algoritmo apresentado por Yang et al. é necessário normalizar os resultados. Para isso deve-se multiplicar a complexidade de Yang et al. por k, sendo que são necessários k *bits* para representar N. Logo, a nova equação do consumo de memória do algoritmo de Yang passa a ser:

$$O(k^2N) \tag{4.2}$$

Comparado com o método desenvolvido neste trabalho, que requer N *bits*, nosso algoritmo

consome 0.34% da memória requerida por Yang et al. para construir as árvores geradoras independentes sobre Q_{17} (131.070 vértices).

Tabela 4.1: Memória despendida(em *bits*) durante o processo de criação das árvores

k	Método Yang	MinimalIST	MinimalIST/Yang %
3	72	8	11.11
4	256	16	6.25
5	800	32	4.0
6	2,304	64	2.77
7	6,272	128	2.04
8	16,384	256	1.56
9	41,472	512	1.23
10	102,400	1,024	1.0
11	247,808	2,048	0.82
12	589,824	4,096	0.69
13	1,384,448	8,192	0.59
14	3,211,264	16,384	0.51
15	7,372,800	32,768	0.44
16	16,777,216	65,536	0.39
17	37,879,808	131,072	0.34

Foram feitos testes até Q_{17} até o presente momento, o processo de verificação dos caminhos independentes foi feito computacionalmente embora o algoritmo funcione para qualquer N.

Para mostrar a simplicidade do algoritmo *MinmalIST* foi avaliada uma implementação do algoritmo utilizando a linguagem de programação C. A tabela 4.3 mostra os resultados obtidos avaliando-se três ambientes diferenciados, tabela 4.2.

4.2 Algoritmo OptimIST

O algoritmo *OptimIST* foi executado em dois ambientes similares, sendo que somente os resultados referentes a um ambiente (*AMD Opteron(TM) Processor 6136*) são mostrados.

Processador	Clock	Cache L2	RAM	Sist. Operacional
Intel Core 2 Duo	2,4 GHz	3 MB	6 GB	OS X 10.8
AMD Opteron(TM) Processor 6136	2,4 GHz	512 KB	128 GB	Debian 7.1
ARM 1176JZF-S	0,7 GHz	128 KB	0,5 GB	Debian 7.1

Tabela 4.2: Ambientes de teste utilizados.

Grafo	Vértices	Memória (bytes)	Tempo para gerar 1 árvore (segundos)		
			ARM	Intel	AMD
Q_1	2	0,25	0,017	0,004	0,005
Q_2	4	0,5	0,017	0,004	0,007
Q_3	8	1	0,018	0,004	0,010
Q_4	16	2	0,017	0,004	0,012
Q_5	32	4	0,017	0,004	0,015
Q_6	64	8	0,017	0,004	0,017
Q_7	128	16	0,018	0.005	0,019
Q_8	256	32	0,018	0,005	0,021
Q_9	512	64	0,020	0.005	0,025
Q_{10}	1.024	128	0,023	0,006	0,029
Q_{11}	2.048	256	0,029	0,007	0,031
Q_{12}	4.096	512	0,041	0,008	0,043
Q_{13}	8.192	1.024	0,068	0,009	0,047
Q_{14}	16.384	2.048	0,119	0,015	0,054
Q_{15}	32.768	4.096	0,226	0,028	0,060
Q_{16}	65.536	8.192	0,454	0,051	0,074
Q_{17}	131.072	16.384	0,882	0,095	0,093
Q_{18}	262.144	32.768	1,878	0,181	0,184
Q_{19}	524.288	65.536	3,904	0,361	0,380
Q_{20}	1.048.576	131.072	8,112	0,736	0,790
Q_{21}	2.097.152	262.144	16,676	1,561	1,634
Q_{22}	4.194.304	524.288	35,432	3,202	3,388
Q_{23}	8.388.608	1.048.576	75,565	6,547	6,935
Q_{24}	16.777.216	2.097.152	155,167	13,537	14,347
Q_{25}	33.554.432	4.194.304	329,428	27,512	29,600
Q_{26}	67.108.864	8.388.608	679,989	59,690	61,184
Q_{27}	134.217.728	16.777.216	1.401,671	118,039	125,931
Q_{28}	268.435.456	33.554.432	2.936,366	244,764	259,918
Q_{29}	536.870.912	67.108.864	6.564,441	509,110	537,133
Q_{30}	1.073.741.824	134.217.728	13.507,901	1.181,033	1.107,093

Tabela 4.3: Tempo requerido para execução do algoritmo *MinimalIST* nos 3 ambientes distintos: ARM 0.7 GHz, Intel 2.4 GHz e AMD 2.4 GHz.

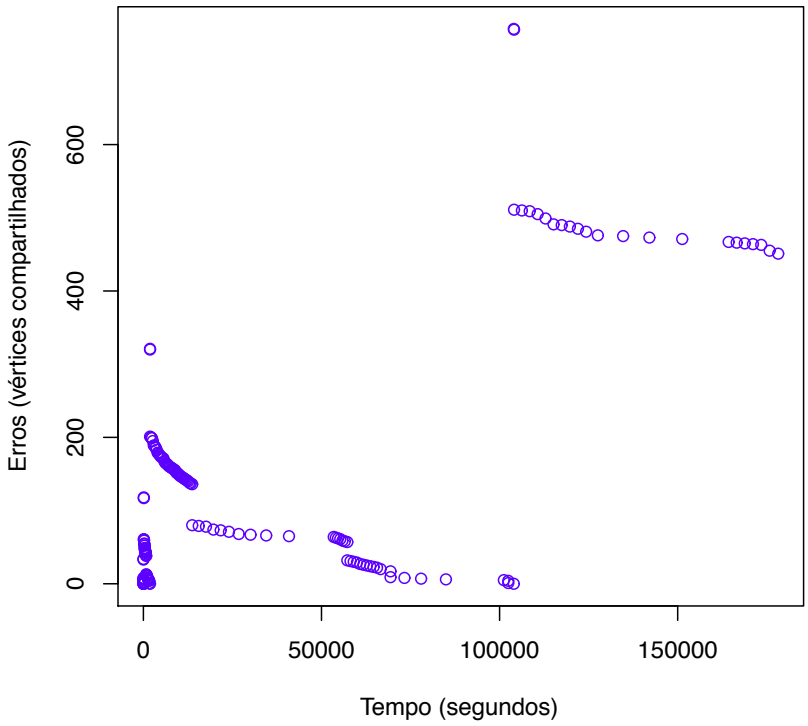
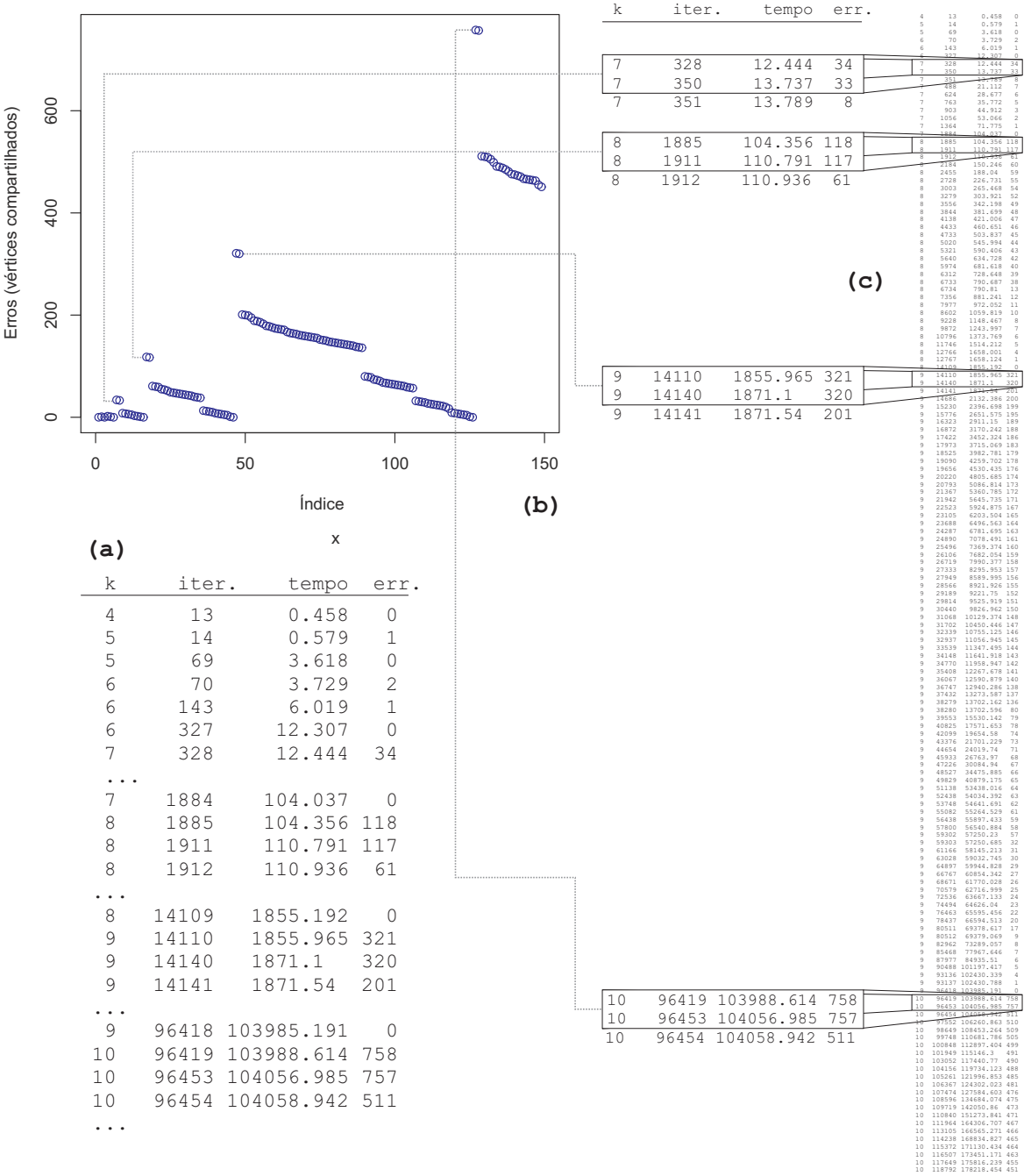


Figura 4.1: Gráfico do tempo necessário para minimização dos erros pelo algoritmo *OptimIST*. Somente o intervalo até Q_9 com 450 erros é mostrado para melhor visualização dos valores, embora o algoritmo tenha sido executado com sucesso até Q_{10} com 0 erros. Lembrando que erros são vértices compartilhados em mais de um caminho nas árvores geradoras.

Os resultados obtidos durante os testes são descritos a seguir. Na figura 4.1 pode-se verificar o tempo necessário para a convergência do método *OptimIST*. Na figura 4.2 podem-se notar alguns padrões quanto a convergência dos erros. Para cada k a amplitude dos erros diminui drasticamente após a correção de 2 erros. Isso parece indicar mais um padrão de arestas a serem evitadas para que o algoritmo *OptimIST* converja mais rapidamente.

4.3 Roteamento Networks on Chip

Nesta seção é descrita a aplicação do algoritmo *MinimalIST* para a geração de caminhos disjuntos em roteamento de *Networks-on-Chip*. Os algoritmos utilizados na comparação foram o *ECUBE*, *DOR* e o *IST*. Alguns experimentos foram realizados com o objetivo de testar a



k	iter.	tempo	err.
4	13	0.458	0
5	14	0.579	1
5	69	3.618	0
6	70	3.729	2
6	143	6.019	1
6	327	12.307	0
7	328	12.444	34
...			
7	1884	104.037	0
8	1885	104.356	118
8	1911	110.791	117
8	1912	110.936	61
...			
8	14109	1855.192	0
9	14110	1855.965	321
9	14140	1871.1	320
9	14141	1871.54	201
...			
9	96418	103985.191	0
10	96419	103988.614	758
10	96453	104056.985	757
10	96454	104058.942	511
...			

Figura 4.2: (a) Tabela resumida destacando-se os intervalos principais referentes a cada valor de k, com k variando de 4 a 10 (tempo medido em segundos). (b) Gráfico de erros. (c) Tabela de dados referentes ao gráfico de erros. Em destaque o padrão de erros, dois intervalos próximos seguidos por uma queda abrupta.

utilização de árvores geradoras independentes para o roteamento. Para se ter um comparativo dos algoritmos de roteamento *IST* (algoritmo proposto) e *ECUBE* com o algoritmo *DOR*

é necessário se ter um mapeamento do hipercubo para uma malha de dimensões X,Y. Na figura 4.3 tem-se um exemplo do mapeamento de uma malha de 32 vértices, 8×4 para o hipercubo Q_5, 2^5 vértices. Para se mapear uma malha para um hipercubo, a quantidade de vértices deve ser uma potência de 2 para que se possa ter o mesmo número de vértices do hipercubo. O mapeamento ocorre da seguinte maneira: Cada coordenada do hipercubo é obtida pela concatenação dos *bits* da representação das coordenadas x, y da malha utilizando-se seu código $gray$ equivalente, resultando na representação binária do vértice do hipercubo.

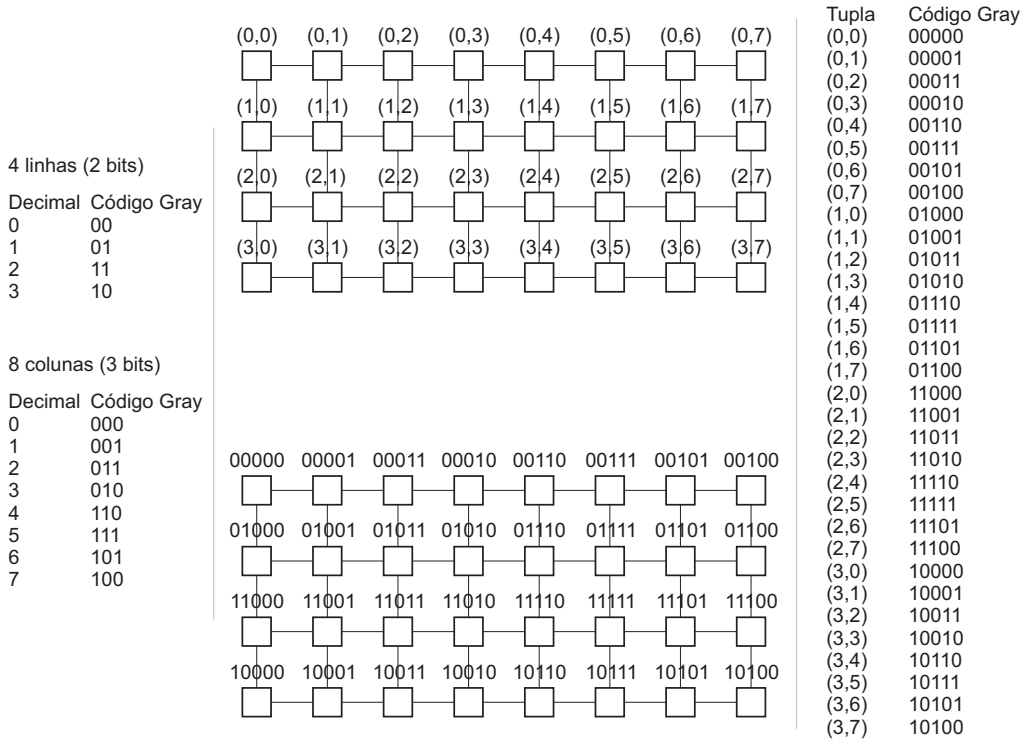


Figura 4.3: Mapeamento de uma malha de 8×4 vértices para o hipercubo Q_5.

Neste sentido, foram realizados testes considerando os algoritmos *DOR*, *ECUBE* e o algoritmo proposto neste trabalho, o *MinimalIST* (chamado de *IST* nos testes). O objetivo é medir a quantidade de vértices concorrentes em rotas aleatórias. Para o primeiro conjunto de testes foram utilizadas 2000 execuções para cada quantidade de vértices transmitindo concorrentemente entre eles, no caso em questão de 1 a 256 vértices. Aleatoriamente foram escolhidos um vértice fonte e um destino da mensagem, em seguida foram calculadas as rotas do vértice fonte ao vértice destino utilizando-se os três algoritmos (*DOR*, *ECUBE* e *IST*). O número de

execuções escolhido mostrou-se mais do que suficiente para retratar graficamente o número de colisões *versus* número de vértices transmitindo concorrentemente. Apesar do *ECUBE* ser somente um *DOR* com dimensões a mais, o *DOR* considerado aqui só utiliza as dimensões (x,y), não utilizando então todos os *links* disponíveis na topologia hipercubo (veja figura 4.4 para os *links* utilizados). Isto não deve ser visto como uma desvantagem para o *DOR* visto que seu equivalente com mais dimensões, *ECUBE*, é utilizado no comparativo, e sim como uma tentativa de mostrar as vantagens em se utilizar mais *links* em relação a concorrência na transmissão de mensagens dentre os vértices.

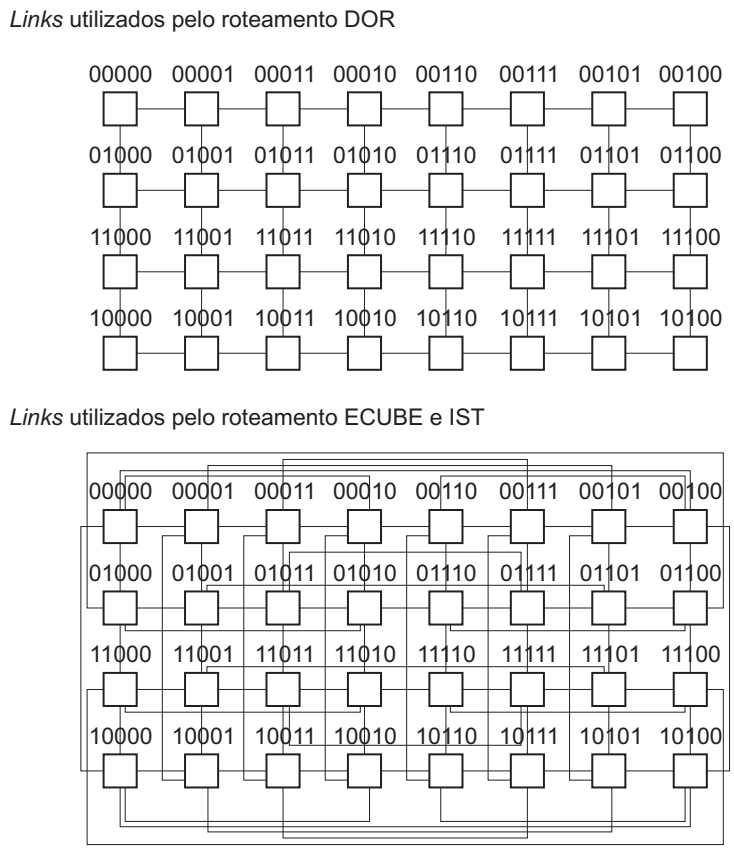


Figura 4.4: Links utilizados em cada roteamento considerado.

Primeiramente foram considerados como colisões somente vértices que apareciam em mais de um caminho dentre as rotas analisadas. Os resultados desta análise podem ser vistos na figura 4.6 (a) e (b) . Em seguida, foram consideradas a utilização de *links*, cujo exemplo do que é considerado colisão pode ser visto na figura 4.5.

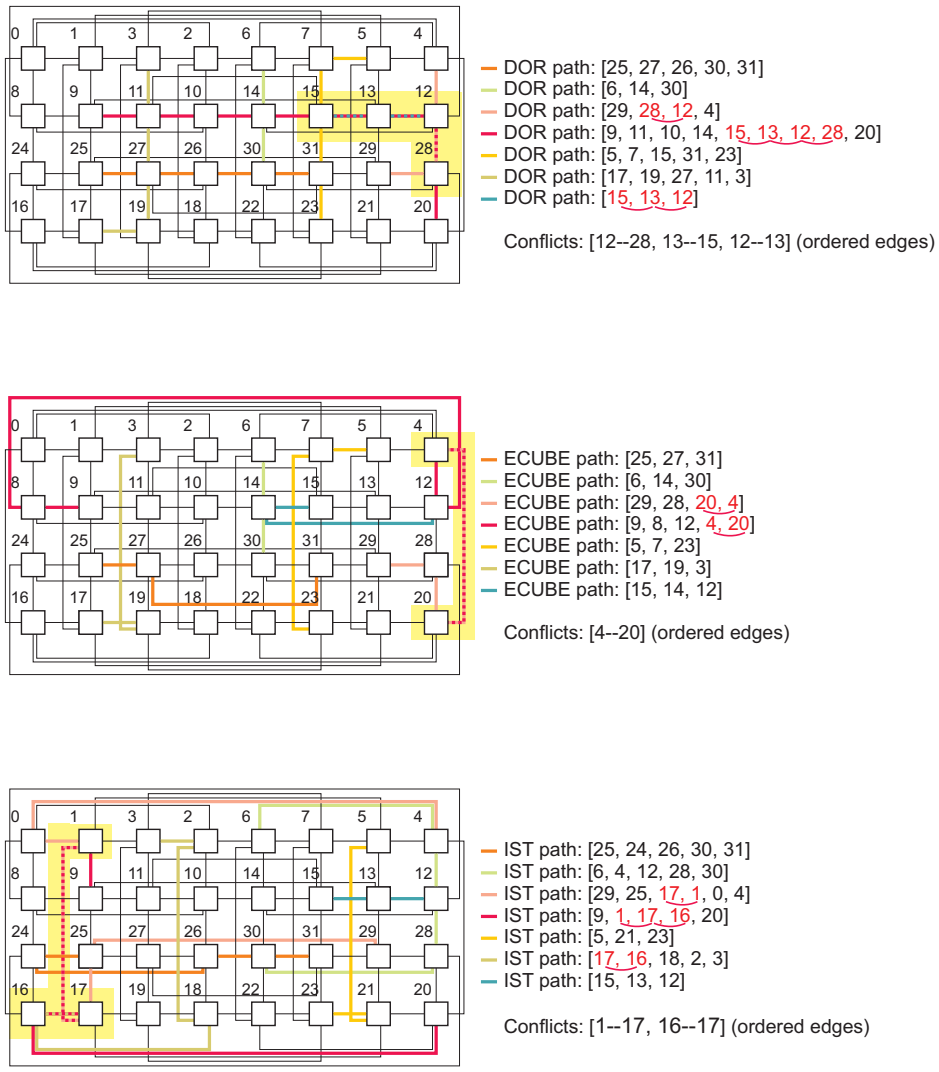


Figura 4.5: Exemplos de colisões considerando *links* com os três algoritmos utilizados (*DOR, ECUBE, IST*).

Para o segundo conjunto de testes um outro modelo de tráfego foi utilizado. Cerca de 75% do tráfego é feito entre vizinhos e 25% distribuído uniformemente entre os demais vértices de acordo com o princípio de localidade de comunicação descrito em [7] . Cerca de 14% do tráfego foi direcionado a um único vértice escolhido aleatoriamente para caracterizar *hotspots* (Em um tráfego caracterizado como *hotspot* poucos nós da rede recebem mais tráfego se comparado aos nós restantes), figura 4.7 (c), (d), (e) e (f).

Foram também feitos testes levando-se em conta a utilização de caminhos alternativos em

caso de falhas, figura 4.8. No entanto, não se mostrou vantajoso a utilização de tal abordagem não havendo ganho quanto ao número de caminhos prejudicados por falhas e, principalmente, ao fato de tal abordagem requerer a implementação de uma lógica mais complexa.

Para validar a utilização das árvores geradoras independentes em uma *NoC*, o software de simulação Noxim [88] foi utilizado. Este software foi elaborado utilizando-se a plataforma *SystemC* como base, a qual provê uma simulação precisa em relação a ciclos, consumo de potência, etc.

O mapeamento dos índices do hipercubo para os índices padrões usados em uma malha do *Noxim* é mostrado na figura 4.9, assim como os nós utilizados como *hotspots* em destaque. Para os testes, foi utilizada uma malha de dimensões 8×4. Os algoritmos de roteamento utilizados nos testes, além do *IST*, foram os fornecidos pelo *Noxim*. Somente as arestas presentes na malha foram utilizadas no roteamento. O objetivo aqui, foi mostrar que, para uma topologia de malha, comumente utilizada em *NoCs*, o algoritmo apresenta desempenho similar aos algoritmos de roteamento mais utilizados. Em uma malha de 2 dimensões os algoritmos *DOR* e *ECUBE* se equivalem, sendo aqui representados pelo algoritmo *xy*, nome utilizado no simulador *Noxim*.

Os resultados são descritos da seguinte forma, a tabela 4.4 mostra os resultados obtidos pelo simulador com a execução de 10 milhões de ciclos, utilizando-se um trafego aleatório. As tabelas 4.5 e 4.6 mostram um tráfego aleatório com 14% do tráfego direcionado aos *hotspots* 7 (canto superior esquerdo) e 11 (central à malha) respectivamente. Tais nós foram escolhidos para caracterizar um modelo diferenciado de tráfego, sendo que quaisquer outros nós posicionados em um canto e em uma posição central à malha poderiam ter sido utilizados.

Tabela 4.4: Resultado da simulação de uma malha 8×4.

Algoritmo	Pctes. Rec.	Flits Rec.	Atrs. Médio	Atrs. Máx.	Vazão /IP	Energia Tot.
ist	3197719	19189594	157	14.3871	0.0599735	0.00325296
xy	3199638	19202152	168	14.3870	0.0600127	0.00325410
negat.first	1781440	10694170	289	15.5048	0.0334226	0.00237962
w.first	3196712	19182913	193	14.5486	0.0599526	0.00325308
n.last	3199406	19195222	262	15.2956	0.0599911	0.00325453
oddeven	3196349	19186843	213	14.4579	0.0599649	0.00325423
fully ad.	3199212	19193530	136	14.3608	0.0599858	0.00325529

Tabela 4.5: Resultado da simulação de uma malha 8×4. Hotspot IP 7, 14% do tráfego.

Algoritmo	Pctes. Rec.	Flits Rec.	Atrs. Médio	Atrs. Máx.	Vazão /IP	Energia Tot.
ist	933918	5601659	2.457e+06	9.624e+06	0.0175069	0.00194531
xy	933226	5599035	2.457e+06	9.621e+06	0.0174987	0.00194498
negat.first	933350	5599959	1.412e+06	9.959e+06	0.0175016	0.00185284
w.first	932858	5598818	1.421e+06,	9.945e+06	0.0174981	0.00185270
n.last	933121	5598700	2.464e+06	9.611e+06	0.0174977	0.00194532
oddeven	932702	5597947	1.634e+06	9.943e+06	0.0174953	0.00188821
fully ad.	933571	5601766	1.425e+06	9.972e+06	0.0175073	0.00185287

Tabela 4.6: Resultado da simulação de uma malha 8×4. Hotspot IP 11, 14% do tráfego.

Algoritmo	Pctes. Rec.	Flits Rec.	Atrs. Médio	Atrs. Máx.	Vazão /IP	Energia Tot.
ist	932605	5598878	1.892e+06	0.0174982	9.925e+06	0.00168392
xy	933894	5601561	1.898e+06	0.0175066	9.918e+06	0.00168421
negat.first	932848	5596531	2.083e+06	0.0174909	9.931e+06	0.00170629
w.first	933500	5602600	1.995e+06	0.0175099	9.901e+06	0.00167645
n.last	933682	5599515	1.764e+06	0.0175002	9.925e+06	0.00167374
oddeven	933915	5602934	2.222e+06	0.0175109	9.716e+06	0.00169801
fully ad.	933276	5600717	2.111e+06	0.0175040	9.900e+06	0.00170714

Os gráficos das figuras 4.10, 4.11 e 4.12 correspondem aos dados das tabelas 4.4, 4.5 e 4.6 respectivamente.

4.4 Discussão dos resultados

O conhecimento adquirido durante a elaboração do algoritmo *OptimIST* forneceu uma série de ideias a serem seguidas para sua utilização em outras topologias de grafos k-regulares, visto que o *MinimalIST* só funciona para hipercubos. A abordagem da escolha de arestas a serem evitadas foi inovadora, não tendo em nosso conhecimento similar abordagem na área.

O algoritmo *MinimalIST* mostrou-se extremamente eficiente na geração de árvores geradoras, sua implementação é simples e não necessita de paralelização em sua execução. Em 2 dos 3 ambientes utilizados conseguiu-se uma taxa de geração de arestas de mais de $1.10^6 arestas/segundo$.

Analisando-se os experimentos realizados utilizando-se *DOR*, *ECUBE* e *IST* conclui-se que o *IST* tem mais chances de colisões que o *ECUBE*. Isso deve-se ao fato de que todas as

rotas utilizadas pelo algoritmo de roteamento *IST* serem maiores em 1 unidade que as rotas utilizadas pelo algoritmo *ECUBE*. Esta diferença de uma unidade pode parecer pequena, mas vale lembrar que o comprimento máximo das rotas do algoritmo *ECUBE* é $log(n)$, sendo n o número de vértices. Entretanto, mesmo com uma maior chance de colisões, as rotas produzidas pelo algoritmo *IST* são menos suscetíveis a vértices falhos, compensando sua utilização, do nosso ponto de vista. Mesmo porque, a adaptação do algoritmo *ECUBE* para o *IST* é simples, requerendo somente a adição de um salto à rota do algoritmo *ECUBE*, como mostrado no apêndice E. Importante notar que, mesmo na falta de *links* extras da topologia hipercubo, o algoritmo *MinimalIST*, quando aplicado a malhas, apresenta resultados tão bons quanto os algoritmos normalmente utilizados para roteamento em tal topologia.

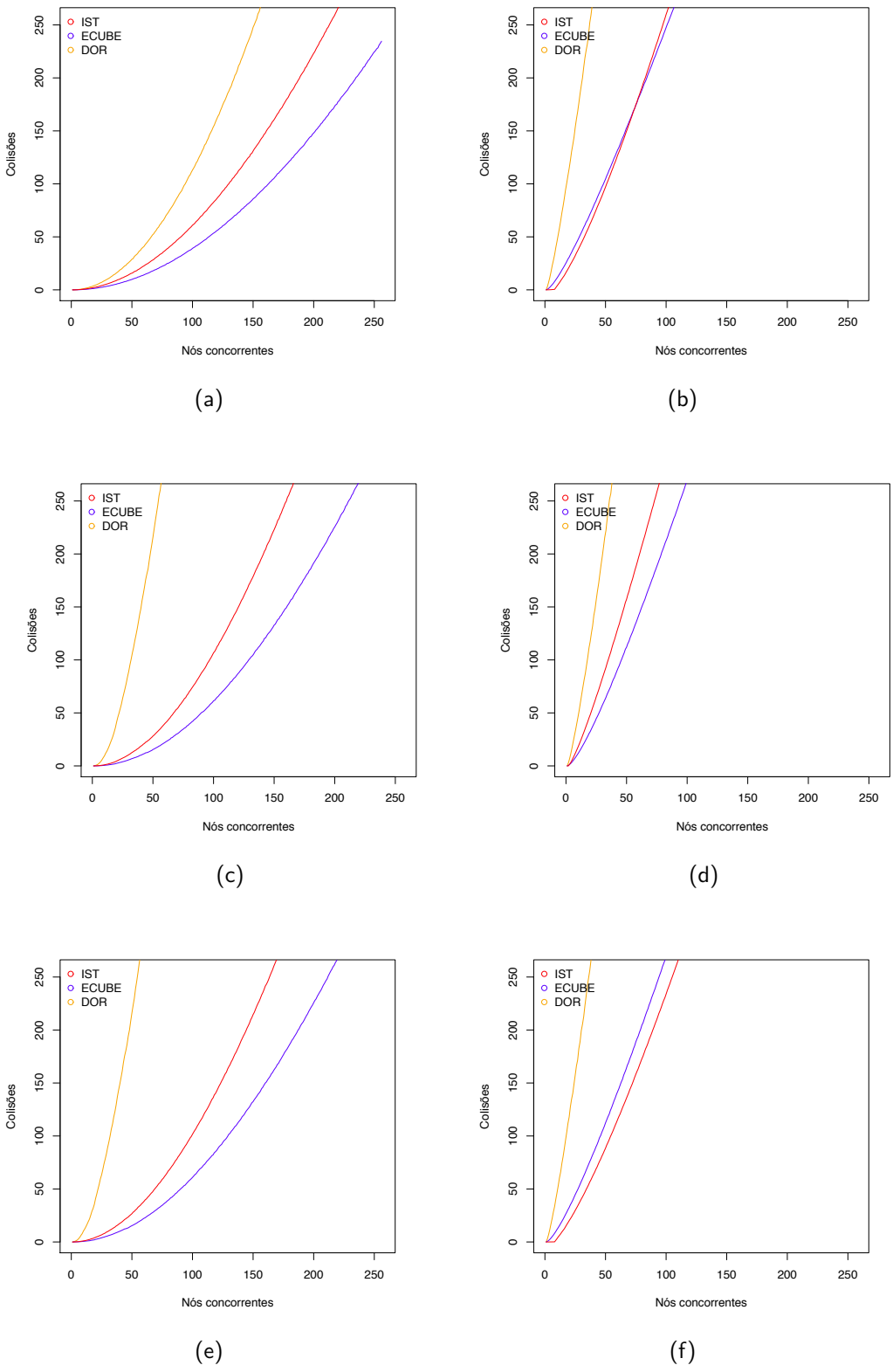


Figura 4.6: Os gráficos do lado esquerdo representam testes utilizando vértices aleatórios como fonte das mensagens, escolhidos arbitrariamente. O lado direito com a mesma fonte sempre. (a)(b) *Considerando somente vértices como concorrentes.*; (c)(d) *Links concorrentes*; (e)(f) *Links concorrentes com IST alternando entre árvores.*

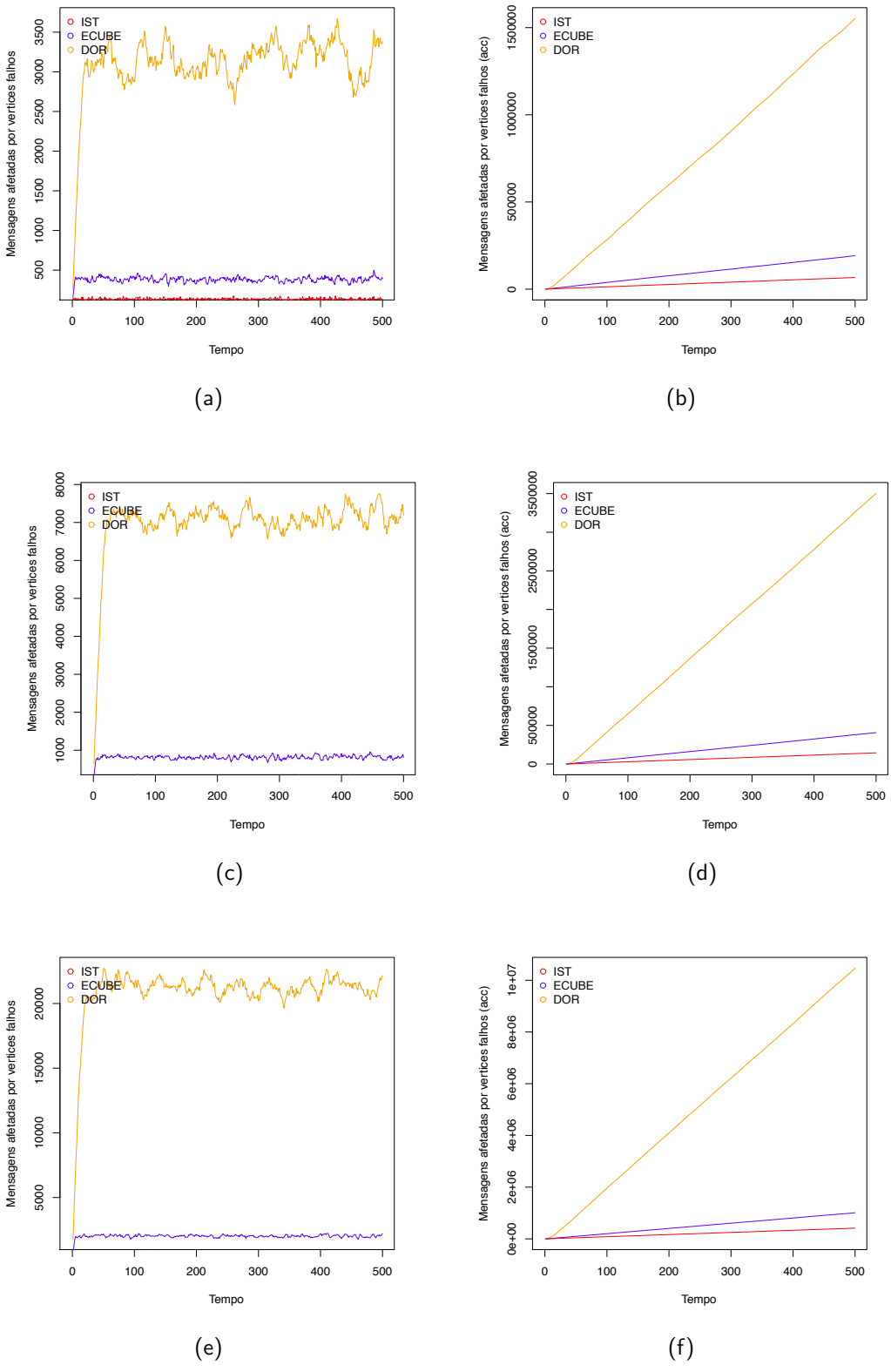


Figura 4.7: Os gráficos do lado esquerdo representam testes utilizando vértices falhos nas seguintes percentagens. (a)(b) *10% de vértices falhos*; (c)(d) *20% de vértices falhos*; (e)(f) *50% vértices falhos*; Os gráficos do lado direito representam os valores acumulados de seus respectivos gráficos a esquerda. Tempo em iterações da simulação.

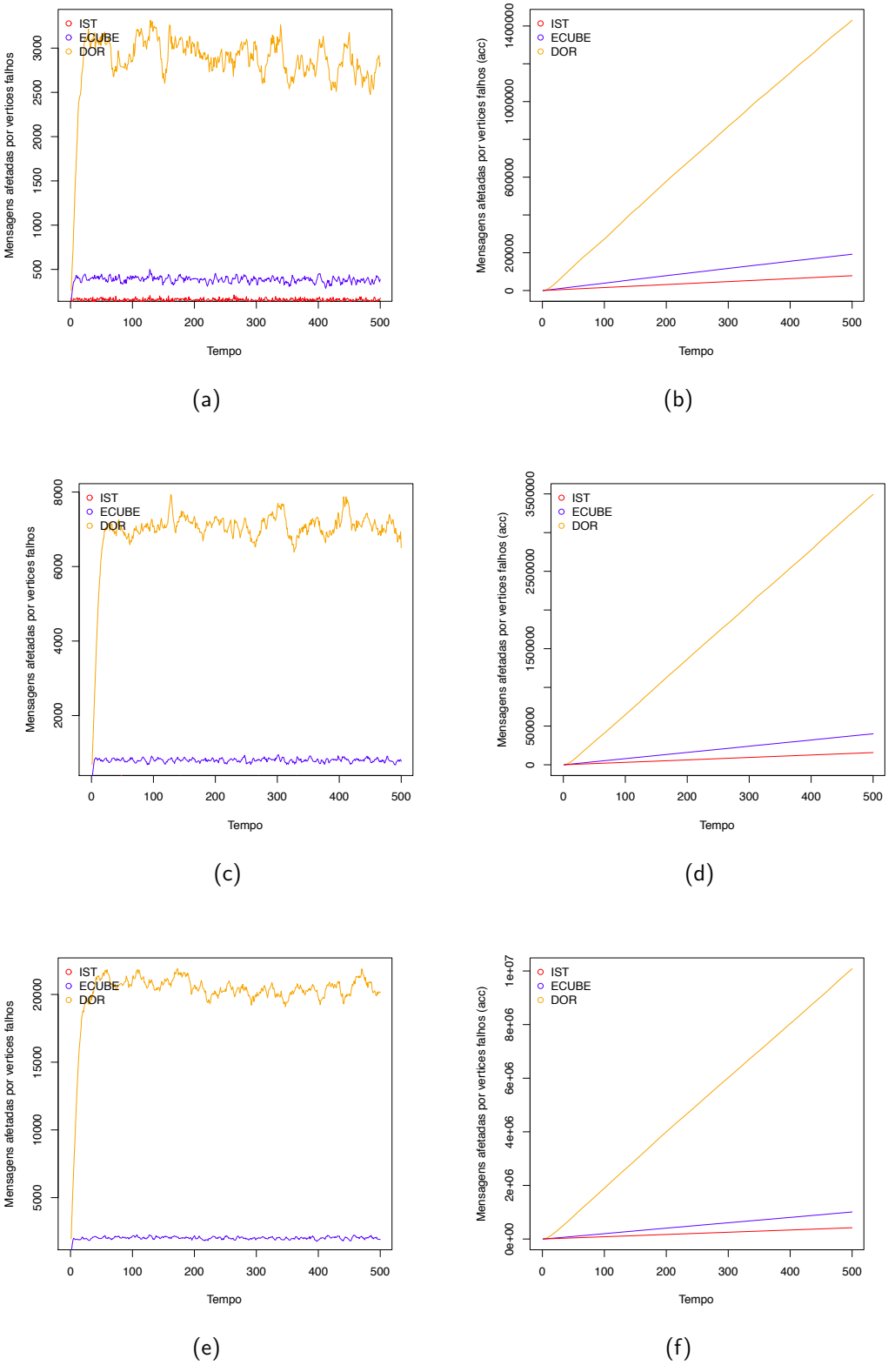


Figura 4.8: Os gráficos do lado esquerdo representam testes utilizando vértices falhos nas seguintes percentagens. (a)(b) *10% de vértices falhos*; (c)(d) *20% de vértices falhos*; (e)(f) *50% vértices falhos*; Os gráficos do lado direito representam os valores acumulados de seus respectivos gráficos a esquerda. Tempo em iterações da simulação.

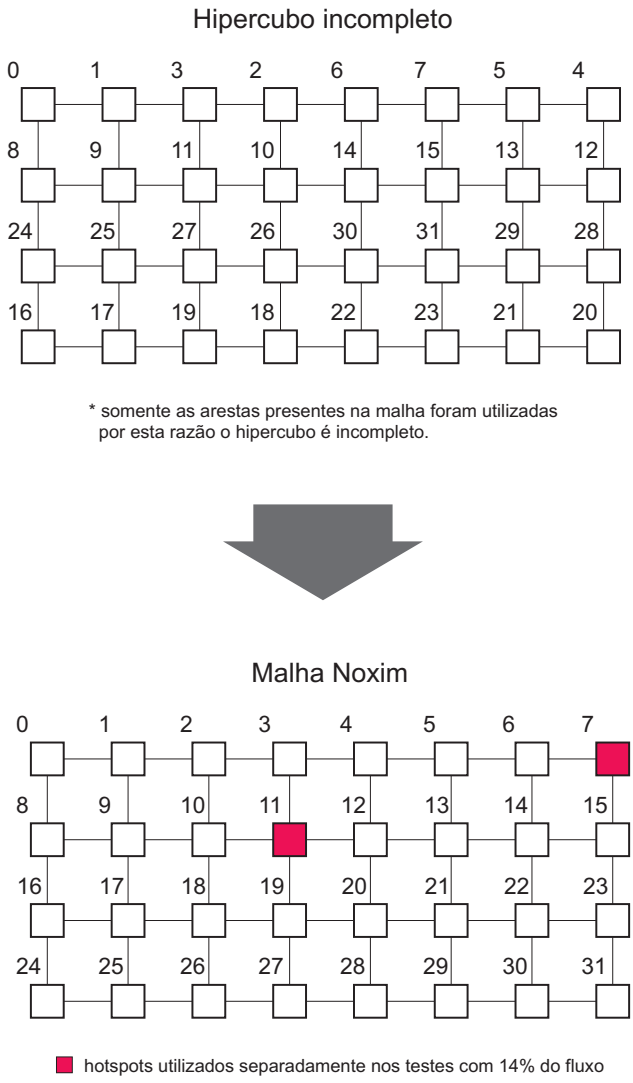


Figura 4.9: Topologia utilizada para as simulações de *Networks on Chip com o simulador* Noxim. *Malha* 8×4. *Somente as arestas presentes na malha foram utilizadas no roteamento.*

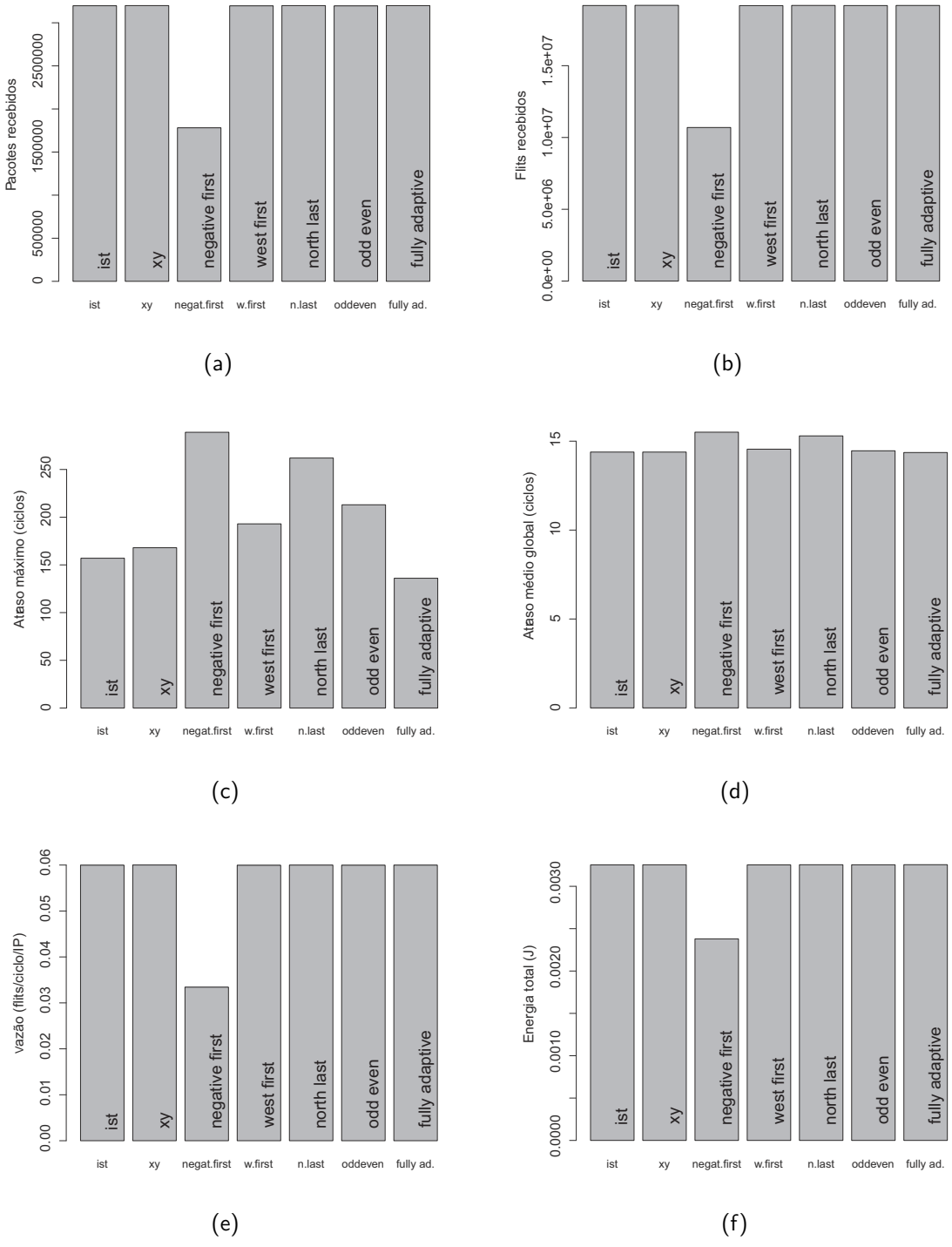


Figura 4.10: Simulação de 7 algoritmos de roteamento utilizando o simulador Noxim (SystemC). Roteamento aleatório para uma malha de 8×4 blocos IPs. Os gráficos representam: (a) *Pacotes recebidos*; (b) *Flits recebidos*; (c) *Atraso máximo (em ciclos)*; (d) *Atraso médio global (em ciclos)*; (e) *Vazão (em flits/ciclos/IP)*; (f) *Energia total consumida (em Joules)*;

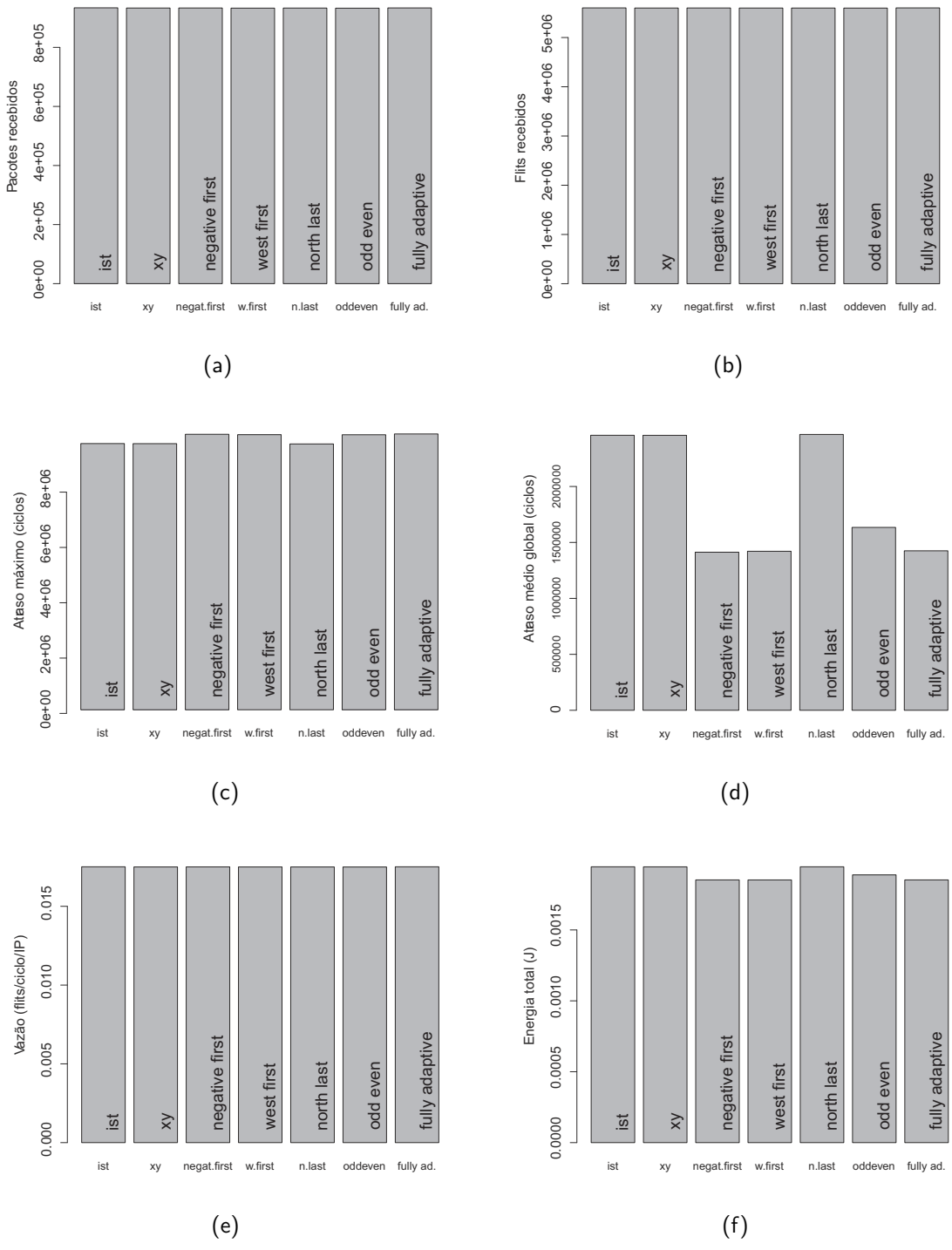


Figura 4.11: Simulação de 7 algoritmos de roteamento utilizando o simulador Noxim (SystemC). Roteamento aleatório para uma malha de 8×4 blocos IPs. *Hotspot*:7. Os gráficos representam: (a) *Pacotes recebidos*; (b) *Flits recebidos*; (c) *Atraso máximo (em ciclos)*; (d) *Atraso médio global (em ciclos)*; (e) *Vazão (em flits/ciclos/IP)*; (f) *Energia total consumida (em Joules)*;

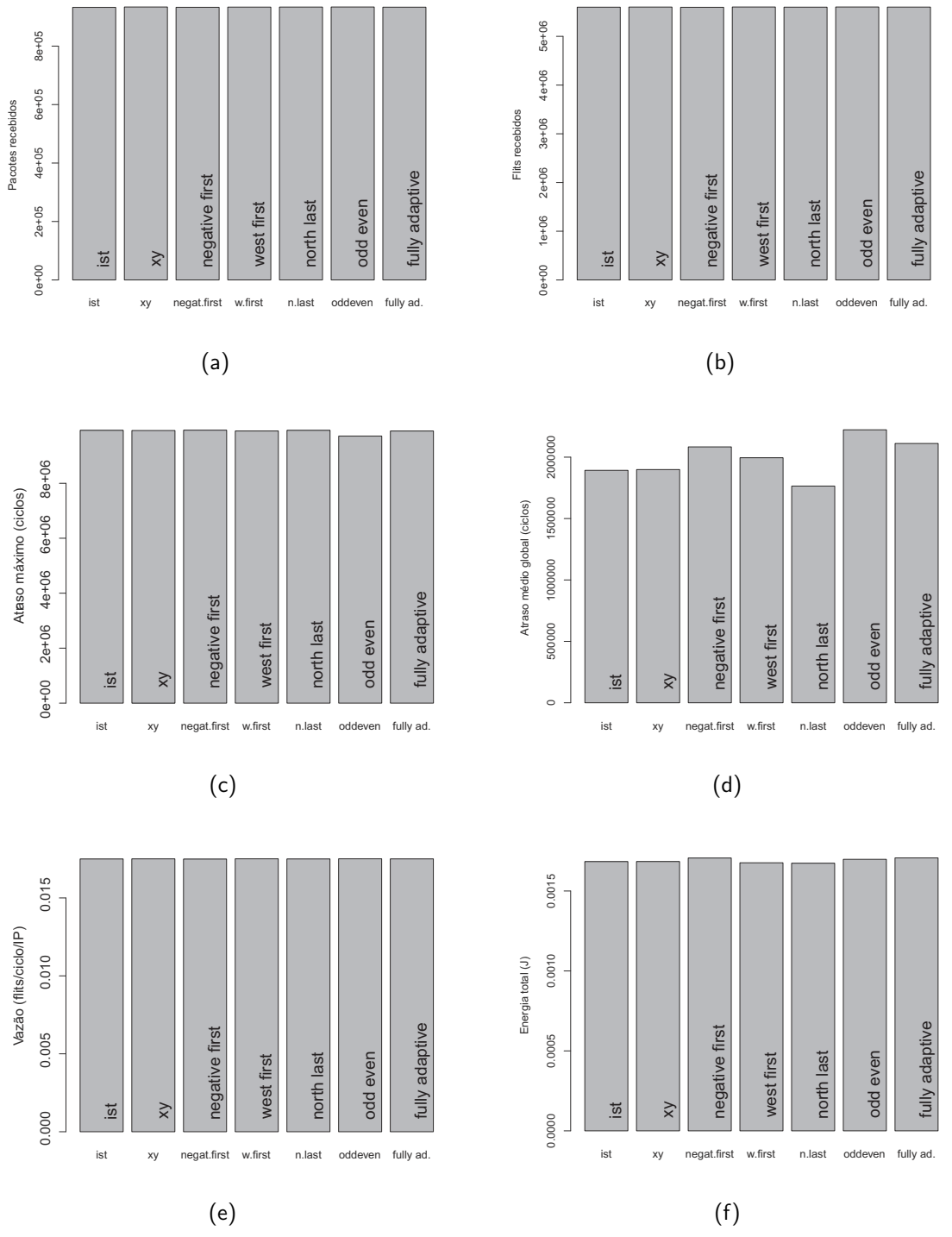


Figura 4.12: Simulação de 7 algoritmos de roteamento utilizando o simulador Noxim (SystemC). Roteamento aleatório para uma malha de 8×4 blocos IPs. *Hotspot*:11. Os gráficos representam: (a) *Pacotes recebidos*; (b) *Flits recebidos*; (c) *Atraso máximo (em ciclos)*; (d) *Atraso médio global (em ciclos)*; (e) *Vazão (em flits/ciclos/IP)*; (f) *Energia total consumida (em Joules)*;

CAPÍTULO 5

CONCLUSÕES

As *NoCs* representam uma possível solução para o problema da utilização de barramentos arbitrários, conectados por roteadores que trocam pacotes de uma forma similar às redes tradicionais. Os circuitos atuais evoluíram o bastante para serem capazes de implementar *NoCs* sofisticadas para interconexão de blocos IPs. Para se transmitir uma grande quantidade de dados, geralmente presentes nos dispositivos atuais, é necessário um processo eficiente de roteamento. A grande disponibilidade de recursos computacionais presentes nos dias de hoje, faz com que algumas abordagens utilizadas para se resolver os problemas encontrados façam uso abusivo de computação, que outrora não seria possível devido aos recursos limitados impostos em projetos embarcados.

Este trabalho descreve as principais técnicas para a construção de árvores geradoras independentes sobre hipercubos, além de propor um método para geração de todas as k árvores de Q_k baseado na manipulação de uma única árvore. Tal árvore é obtida a partir de um algoritmo simples, que consome pouca memória e pode ser implementado em um circuito relativamente simples (apêndice E).

As características do algoritmo desenvolvido proporcionam uma significativa melhora em relação ao principal trabalho da área, *Parallel construction of Optimal Independent Spanning Trees on Hypercubes* de Yang et al. [124].

Outro fato importante a salientar é que o desenvolvimento de algoritmos para a construção de árvores geradoras independentes tende a buscar dois objetivos de pesquisa, um é o projeto de esquemas de construção eficientes [64, 80, 84] sendo que o outro é a redução das alturas das árvores [51, 109, 122]. Neste trabalho conseguimos atingir os dois objetivos de pesquisa focando em um esquema de construção eficiente, o qual mostrou-se ótimo em relação as alturas das árvores.

5.1 Restrições

Dois algoritmos foram apresentados neste trabalho, o *OptimIST* e o *MinimalIST*. O algoritmo *OptimIST* foi utilizado para hipercubos de pequena ordem (até 1024 nós). Visto que o método de redução de erros poderia ou não convergir a 0 com a análise de novos padrões. O algoritmo *MinimalIST* necessita de poucos recursos computacionais para sua implementação. Porém consideramos como encerrada a pesquisa em relação a evolução deste algoritmo, e passamos a analisar somente suas aplicações. Já o algoritmo *OptimIST* que se vale basicamente de análise de certos padrões para a construção das árvores mostrou-se promissor para a análise de outros grafos k conexos além do hipercubo.

5.2 Trabalhos Futuros

Um dos trabalhos futuros pode ser a evolução do algoritmo *OptimIST* para sua utilização em outras famílias de grafos k-conexos e sua aplicação, em uma versão simplificada, em algoritmos adaptativos de roteamento.

Além disso, o algoritmo *MinimalIST* pode ser implementado em um *FPGA* por completo, utilizando-se vários blocos *IPs* para estudar as situações no qual seria melhor utilizado.

Porém, uma das grandes contribuições deste trabalho foi mostrar que há um algoritmo rápido, simples e eficaz para a geração de árvores geradoras independentes sobre hipercubos. Até então o problema era considerado complexo, visto que a proposta do principal trabalho da área foi junto a um congresso de computação paralela, Yang et al. [124].

Dentre as possíveis aplicações temos:

— Sistemas P2P (Peer to Peer), seja pela aplicação do hipercubo a topologia virtual da rede, distribuição ou recuperação de chaves;

— Compactação, sequenciamento, indexação de código genético. (visto que o DNA possui a estrutura de um hipercubo Q_6 [67]. Pesquisa já iniciada com alguns resultados promissores);

— Estudo de caminhos disjuntos em qualquer topologia que possa ser embutida em um hipercubo;

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Guest Editor's Introduction: What is Infrastructure IP? *IEEE Design and Test of Computers*, 19:5–7, 2002.

[2] HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, 2004.

[3] Long paths and cycles in hypercubes with faulty vertices. *Information Sciences*, 179(20):3634–3644, 2009.

[4] B. Abali, F. Ozguner, and A. Bataineh. Balanced parallel sort on hypercube multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 4(5):572–581, May 1993.

[5] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. Zeferino. SPIN: a scalable, packet switched, on-chip micro-network. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 70–73 suppl., 2003.

[6] A. Agarwal and M. Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 750–753, New York, NY, USA, 2007. ACM.

[7] M. Ali, M. Welzl, A. Adnan, and F. Nadeem. Using the ns-2 network simulator for evaluating network on chips (noc). In *Emerging Technologies, 2006. ICET '06. International Conference on*, pages 506–512, Nov. 2006.

[8] F. S. Annexstein and K. A. Berman. Directional Routing via Generalized st-Numberings. *SIAM J. Discret. Math.*, 13:268–279, Apr. 2000.

[9] S. Arasu, C. Ravikumar, and S. Nandy. A low power and low cost scan test architecture for multi-clock domain SoCs using virtual divide and conquer. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 9 pp.–377, Nov. 2005.

[10] D. Atienza, F. Angiolini, S. Murali, A. Pullini, L. Benini, and G. D. Micheli. Network-on-Chip design and synthesis outlook. *Integration, the VLSI Journal*, 41(3):340–359, 2008.

[11] C. Aykanat, F. Özgüner, F. Ercal, and P. Sadayappan. Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes. *IEEE Trans. Comput.*, 37:1554–1568, Dec. 1988.

[12] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *Micro, IEEE*, 22(5):16–23, Sep/Oct 2002.

[13] F. Bao, Y. Funyu, Y. Hamada, and Y. Igarashi. Reliable broadcasting and secure distributing in channel networks. In *Parallel Architectures, Algorithms, and Networks, 1997. (I-SPAN '97) Proceedings., Third International Symposium on*, pages 472–478, Dec. 1997.

[14] F. Bao, Y. Igarashi, and S. R. Öhring. Reliable broadcasting in product networks. *Discrete Applied Mathematics*, 83(1-3):3–20, 1998.

[15] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology adaptive network-on-chip design and implementation. *Computers and Digital Techniques, IEE Proceedings* -, 152(4):467–472, July 2005.

[16] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, 2008.

[17] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35:70–78, 2002.

[18] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[19] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. Supervised by Assoc. Prof. Jens Sparsø, IMM.

[20] B. Bollobás. *Modern graph theory*. Graduate texts in mathematics. Springer, 1998.

[21] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2-3):105 – 128, 2004. Special issue on networks on chip.

[22] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier, New York, 1976.

[23] T. Chan and Y. Saad. Multigrid algorithms on the hypercube multiprocessor. *Computers, IEEE Transactions on*, C-35(11):969–977, Nov. 1986.

[24] Y.-S. Chen, C.-Y. Chiang, and C.-Y. Chen. Multi-node broadcasting in all-ported 3-D wormhole-routed torus using an aggregation-then-distribution strategy. *J. Syst. Archit.*, 50:575–589, Sept. 2004.

[25] Y.-S. Chen, T.-Y. Juang, and Y.-Y. Shen. Congestion-free embedding of 2(n-k) spanning trees in an arrangement graph. *Journal of Systems Architecture*, 47(1):73–86, 2001.

[26] J. Cheriyan and S. N. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *J. Algorithms*, 9(4):507–537, 1988.

[27] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 59–68, New York, NY, USA, 2003. ACM.

[28] G.-M. Chiu. A fault-tolerant broadcasting algorithm for hypercubes. *Inf. Process. Lett.*, 66:93–99, Apr. 1998.

[29] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/-Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[30] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *Computers, IEEE Transactions on*, C-36(5):547–553, May 1987.

[31] W. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689, 2001.

[32] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct. 1974.

[33] S. Deshpande and R. M. Jenevin. Scalability of a binary tree on a hypercube. *Proc. Int. Conf. Parallel Processing*, pages 661–668, 1986.

[34] R. Diestel. *Graph Theory*. Number 173 in Graduate Texts in Mathematics. Springer, Heidelberg, 3rd edition, 2005.

[35] D. Du, D. Hsu, and Y. Lyuu. On the diameter vulnerability of Kautz digraphs. *Discrete Mathematics*, 151(1-3):81–85, 1996.

[36] J. Edmonds. Submodular Functions, Matroids, and Certain Polyhedra. In M. Junger, G. Reinelt, and G. Rinaldi, editors, *Combinatorial Optimization - Eureka, You Shrink!*,

volume 2570 of *Lecture Notes in Computer Science*, pages 11–26. Springer Berlin / Heidelberg, 2003.

[37] P. Fragopoulou and S. Akl. Edge-disjoint spanning trees on the star network with applications to fault tolerance. *Computers, IEEE Transactions on*, 45(2):174–185, Feb. 1996.

[38] P. Fraigniaud. Fault-tolerant gossiping on hypercube multicomputers. In A. Bode, editor, *Distributed Memory Computing*, volume 487 of *Lecture Notes in Computer Science*, pages 463–472. Springer Berlin / Heidelberg, 1991.

[39] P. Fraigniaud and C. Gavoille. A theoretical model for routing complexity. In L. Gargano and D. Peleg, editors, *SIROCCO*, pages 98–113. Carleton Scientific, 1998.

[40] P. Fraigniaud and C.-T. Ho. Arc-Disjoint Spanning Trees on Cube-Connected Cycles Networks. In *ICPP (1)'91*, pages 225–229, 1991.

[41] P. Fraigniaud and E. Lazard. *Methods and problems of communication in usual networks*. Rapports de recherche. Université de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique, 1991.

[42] C. F.Van and Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1-2):85–100, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[43] H. N. Gabow. A Matroid Approach to Finding Edge Connectivity and Packing Arborescences. *Journal of Computer and System Sciences*, 50:259–273, 1995.

[44] C. Gavoille and E. Guevremont. Worst case bounds for shortest path interval routing. *Journal of Algorithms*, 27(1):1–25, 1998.

[45] Z. Ge and S. L. Hakimi. Disjoint rooted spanning trees with small depths in De Bruijn and Kautz graphs. *SIAM Journal on Computing*, 26(1):79–92, 1997.

[46] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, 2005.

[47] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, pages 250–256, New York, NY, USA, 2000. ACM.

[48] K. Gunther. Prevention of deadlocks in packet-switched data transport systems. *Communications, IEEE Transactions on*, 29(4):512–524, 1981.

[49] A. K. Gupta and S. E. Hambrusch. Multiple network embeddings into hypercubes. *J. Parallel Distrib. Comput.*, 19:73–82, Oct. 1993.

[50] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[51] T. Hasunuma and H. Nagamochi. Independent spanning trees with small depths in iterated line digraphs. *Discrete Applied Mathematics*, 110(2-3):189–211, 2001.

[52] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 873–878, 1999.

[53] J. Henkel, W. Wolf, and S. Chakradhar. On-chip networks: a scalable, communication-centric embedded system design paradigm. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 845–851, 2004.

[54] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, Sept. 2006.

[55] C. Ho and S. Johnsson. Spanning balanced trees in boolean cubes. *SIAM Journal on Scientific and Statistical Computing*, 10(4):607–630, 1989.

[56] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms/C++*. Silicon Press, 2007.

[57] A. Huck. Disproof of a conjecture about independent branchings in k-connected directed graphs. *Journal of Graph Theory*, 20(2):235–239, 1995.

[58] A. Huck. Independent branchings in acyclic diagraphs. *Discrete Mathematics*, 199(1-3):245–249, 1999.

[59] A. Huck. Independent Trees in Planar Graphs Independent trees. *Graphs and Combinatorics*, 15:29–77, 1999.

[60] HyperCore Software Developer's Handbook. Plurality, 2009. `www.plurality.com`.

[61] Intel 17455-03. *iPSC User's guide*. Portland, OR, USA, Oct. 1985.

[62] Intel Corporation. iPSC System Overview, Jan. 1986.

[63] A. Itai and M. Rodeh. The Multi-Tree Approach to Reliability in Distributed Networks. *Information and Computation*, 79:43–59, 1984.

[64] Y. Iwasaki, Y. Kajiwara, K. Obokata, and Y. Igarashi. Independent spanning trees of chordal rings. *Information Processing Letters*, 69(3):155–160, 1999.

[65] A. Jalabert, S. Murali, L. Benini, and G. De Micheli. XpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. In *Proceedings of the conference on Design, automation and test in Europe*, volume 2, pages 884–889, Washington, DC, USA, 2004. IEEE Computer Society.

[66] J.-F. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor, 1987.

[67] M. A. Jimenez-Montano, C. R. de la Mora-Basanez, and T. Poschel. The hypercube structure of the genetic code explains conservative and non-conservative aminoacid substitutions in vivo and in vitro. *Biosystems*, 39(2):117 – 125, 1996.

[68] S. Johnsson. Communication efficient basic linear algebra computation on hypercube architectures. *J. Parallel Distrib. Computing*, 4(2):133–172, Apr. 1987.

[69] S. Johnsson and C. Ho. Optimum broadcasting and personalized communication in hypercubes. *Computers, IEEE Transactions on*, 38(9):1249–1268, Sept. 1989.

[70] S. Johnsson and C.-T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers*, 38:1249–1268, 1989.

[71] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *Micro, IEEE*, 22(5):36–45, 2002.

[72] S. Khuller and B. Schieber. On independent spanning trees. *Information Processing Letters*, 42(6):321–323, 1992.

[73] L. B. Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[74] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[75] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.

[76] T.-H. Lai and W. White. Mapping pyramid algorithms into hypercubes. *Journal of Parallel and Distributed Computing*, 9(1):42–54, 1990.

[77] C. R. Lang. The extension of object-oriented languages to a homogeneous concurrent architecture. Dept. of Computer Science, California Institute of Technology. Technical report, 5014: TR: 82 118-124, 1982.

[78] J. Leeuwen and R. Tan. Computer networks with compact routing tables. In *The Book of L*, pages 259–273. Springer Berlin Heidelberg, 1986.

[79] M. Levy and T. M. Conte. Embedded multicore processors and systems. *Micro, IEEE*, 29(3):7–9, 2009.

[80] K. Miura, D. Takahashi, S.-i. Nakano, and T. Nishizeki. A linear-time algorithm to find four independent spanning trees in four-connected planar graphs. In J. Hromkovic and O. Sykora, editors, *Graph-Theoretic Concepts in Computer Science*, volume 1517 of *Lecture Notes in Computer Science*, pages 115–132. Springer Berlin / Heidelberg, 1998.

[81] C. Moore and A. Russell. Quantum Walks on the Hypercube. In *Randomization and Approximation Techniques in Computer Science*, volume 2483 of *Lecture Notes in Computer Science*, pages 952–952. Springer Berlin / Heidelberg, 2002.

[82] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, Apr. 1965.

[83] S. Murali and G. De Micheli. SUNMAP: a tool for automatic topology selection and generation for NoCs. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 914–919, New York, NY, USA, 2004. ACM.

[84] S. Nagai and S.-i. Nakano. A linear-time algorithm to find independent spanning trees in maximal planar graphs. In U. Brandes and D. Wagner, editors, *Graph-Theoretic Concepts in Computer Science*, volume 1928 of *Lecture Notes in Computer Science*, pages 15–29. Springer Berlin / Heidelberg, 2000.

[85] Ncube V1.0. *Ncube 6400 processor manual*. Beaverton, OR, USA, 1990.

[86] K. Obokata, Y. Iwasaki, F. Bao, and Y. Igarashi. Independent spanning trees of product graphs. In F. d'Amore, P. Franciosa, and A. Marchetti-Spaccamela, editors, *Graph-*

Theoretic Concepts in Computer Science, volume 1197 of *Lecture Notes in Computer Science*, pages 338–351. Springer Berlin / Heidelberg, 1997.

[87] J. Owens, W. Dally, R. Ho, D. Jayasimha, S. Keckler, and L.-S. Peh. Research Challenges for On-Chip Interconnection Networks. *Micro, IEEE*, 27(5):96–108, sept.-oct. 2007.

[88] M. Palesi, D. Patti, and F. Fazzino. Noxim Simulator. `http://noxim.sourceforge.net/`.

[89] P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Design of a switch for network on chip applications. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 5, pages V–217 – V–220 vol.5, May 2003.

[90] S. Pasricha and N. Dutt. Chapter 3 - On-Chip Communication Architecture Standards. In *On-Chip Communication Architectures*, pages 43–100. Morgan Kaufmann, Burlington, 2008.

[91] S. Pasricha and N. Dutt. Introduction. In *On-Chip Communication Architectures*, pages 1–15. Morgan Kaufmann, Burlington, 2008.

[92] P. Ramanathan and K. Shin. Reliable broadcast in hypercube multicomputers. *IEEE Transactions on Computers*, 37:1654–1657, 1988.

[93] A. A. Rescigno. Vertex-disjoint spanning trees of the star network with applications to fault-tolerance and security. *Information Sciences*, 137(1-4):259–276, 2001.

[94] Y. Saad and M. H. Schultz. Topological Properties of Hypercubes. *IEEE Trans. Comput.*, 37:867–872, July 1988.

[95] M. Saldaña, L. Shannon, and P. Chow. The routability of multiprocessor network topologies in FPGAs. In *Proceedings of the 2006 international workshop on System-level interconnect prediction*, SLIP '06, pages 49–56, New York, NY, USA, 2006. ACM.

[96] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28(1):5–8, 1985.

[97] A. Schrijver. Fractional Packing And Covering. *Stichting Mathematisch Centrum*, pages 201–274, 1979.

[98] C. L. Seitz. The cosmic cube. *Commun. ACM*, 28:22–33, Jan. 1985.

[99] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian. Rethinking Digital Design: Why Design Must Change. *Micro, IEEE*, 30(6):9–24, nov.-dec. 2010.

[100] D. Siguenza-Tortosa, T. Ahonen, and J. Nurmi. Issues in the development of a practical NoC: the Proteo concept. *Integration, the VLSI Journal*, 38(1):95–105, 2004.

[101] D. Siguenza-Tortosa and J. Nurmi. VHDL-based simulation environment for Proteo NoC. *High-Level Design, Validation, and Test Workshop, IEEE International*, 0:1–6, 2002.

[102] E. Silva, A. Guedes, and E. Todt. Independent Spanning Trees on Systems-on-chip Hypercubes Routing. In *Electronics, Circuits and Systems (ICECS), IEEE International Conference on (To be Appeared.)*, Mar. 2013.

[103] K. Srinivasan, K. Chatha, and G. Konjevod. Linear-programming-based techniques for synthesis of network-on-chip architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(4):407–420, Apr. 2006.

[104] P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *VLSI Technology, Systems, and Applications, 2001. Proceedings of Technical Papers. 2001 International Symposium on*, pages 184–187, 2001.

[105] K. Suganya and V. Nagarajan. Efficient run-time task allocation in reconfigurable multiprocessor System-on-Chip with Network-on-Chip. In *Computer, Communication and*

Electrical Technology (ICCCET), 2011 International Conference on, pages 12–17, Mar. 2011.

[106] H. Sullivan and T. R. Bashkow. A large scale, homogeneous, fully distributed parallel machine, i. *SIGARCH Comput. Archit. News*, 5(7):105–117, Mar. 1977.

[107] T.-Y. Sung, T.-Y. Ho, and M.-Y. Lin. Multiple-Edge-Fault Tolerance with Respect to Hypercubes. *IEEE Trans. Parallel Distrib. Syst.*, 8:187–192, Feb. 1997.

[108] A. S. Tanenbaum. *Computer networks (4. ed.)*. Prentice Hall, 2002.

[109] S.-M. Tang, Y.-L. Wang, and Y.-H. Leu. Optimal Independent Spanning Trees on Hypercubes. *J. Inf. Sci. Eng.*, 20(1):143–156, 2004.

[110] S.-M. Tang, J.-S. Yang, Y.-L. Wang, and J.-M. Chang. Independent Spanning Trees on Multidimensional Torus Networks. *IEEE Transactions on Computers*, 59:93–102, 2010.

[111] P. Tong and E. Lawler. A faster algorithm for finding edge-disjoint branchings. *Information Processing Letters*, 17(2):73–76, 1983.

[112] Y.-C. Tseng, S.-Y. Wang, and C.-W. Ho. Efficient broadcasting in wormhole-routed multicomputers: a network-partitioning approach. *Parallel and Distributed Systems, IEEE Transactions on*, 10(1):44–61, Jan. 1999.

[113] T. Villiger, H. Kaslin, F. Gurkaynak, S. Oetiker, and W. Fichtner. Self-timed ring for globally-asynchronous locally-synchronous systems. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 141–150, May 2003.

[114] D. Walker and S. Latifi. Improving bounds on link failure tolerance of the star graph. *Information Sciences*, 180(13):2571–2575, 2010.

[115] S. Watanabe, K. Seto, Y. Ishikawa, S. Komatsu, and M. Fujita. Protocol Transducer Synthesis using Divide and Conquer approach. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 280–285, Jan. 2007.

[116] R. W. Whitty. Vertex-disjoint paths and edge-disjoint branchings in directed graphs. *Journal of Graph Theory*, 11(3):349–358, 1987.

[117] P. Wolkotte, G. Smit, G. Rauwerda, and L. Smit. An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 155a, Apr. 2005.

[118] J. Wu and E. B. Fernandez. Broadcasting in faulty hypercubes. *Microprocessing and Microprogramming*, 39(1):43–53, 1993.

[119] R.-Y. Wu, G.-H. Chen, J.-S. Fu, and G. J. Chang. Finding cycles in hierarchical hypercube networks. *Information Processing Letters*, 109(2):112–115, 2008.

[120] X. Wu and S. Latifi. Substar Reliability Analysis in Star Networks. *Information Sciences*, 178(10):2337–2348, 2008.

[121] Xie-Bin and Chen. Many-to-many disjoint paths in faulty hypercubes. *Information Sciences*, 179(18):3110–3115, 2009.

[122] J.-S. Yang, J.-M. Chang, S.-M. Tang, and Y.-L. Wang. Reducing the height of independent spanning trees in chordal rings. *IEEE Trans. Parallel Distrib. Syst.*, 18(5):644–657, May 2007.

[123] J.-S. Yang, J.-M. Chang, S.-M. Tang, and Y.-L. Wang. On the independent spanning trees of recursive circulant graphs G(cdm,d) with $d > 2$. *Theoretical Computer Science*, 410(21-23):2001–2010, 2009.

[124] J.-S. Yang, S.-M. Tang, J.-M. Chang, and Y.-L. Wang. Parallel construction of optimal independent spanning trees on hypercubes. *Parallel Comput.*, 33(1):73–79, 2007.

[125] A. Zehavi and A. Itai. Three tree-paths. *Journal of Graph Theory*, 13:175–188, 1988.

[126] S. G. Ziavras and M. A. Siddiqui. Pyramid mappings onto hypercubes for computer vision: Connection machine comparative study. *Concurrency: Practice and Experience*, 5(6):471–489, 1993.

APÊNDICE A

OBTENDO-SE ÁRVORES GERADORAS INDEPENDENTES A PARTIR DE GRAFOS COMPLETOS

Todo grafo k-conexo é um subgrafo do grafo completo, denotado por K_n. Primeiramente será mostrado que o grafo completo satisfaz a conjectura 3.1.1, pois qualquer grafo é um subgrafo do grafo completo. A prova será feita por indução utilizando-se grafos de ordem K_i e K_{ii} sendo que $ii = i + 1$. Como exemplo utilizaremos K_6 como K_i.

Enumerando os vértices de K_6 temos a seguinte matriz de adjacências A.1:

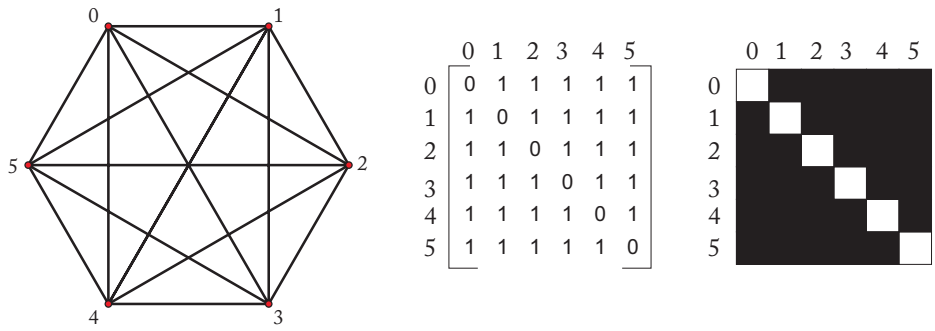


Figura A.1: K_6 grafo 5-conexo

A matriz de adjacências do grafo é a base para a matriz transformada a qual contém todas as arestas necessárias para a formação das k árvores.

Para demonstrar como tal distribuição é feita, apresentamos uma convenção de leitura da matriz de adjacências quando se tratando da identificação das árvores. Sendo assim a matriz de adjacências representada na figura A.2 $b)$ deve ser lida da seguinte forma: Os valores 1 da linha i da matriz indicam que os vértices de índice correspondente à coluna j são filhos do vértice com índice i. Desta maneira pode-se obter a árvore geradora da figura A.2 pela leitura da matriz de adjacências da forma descrita anteriormente.

Uma das consequências disso, é que todos os vértices folhas possuem a linha i corres-

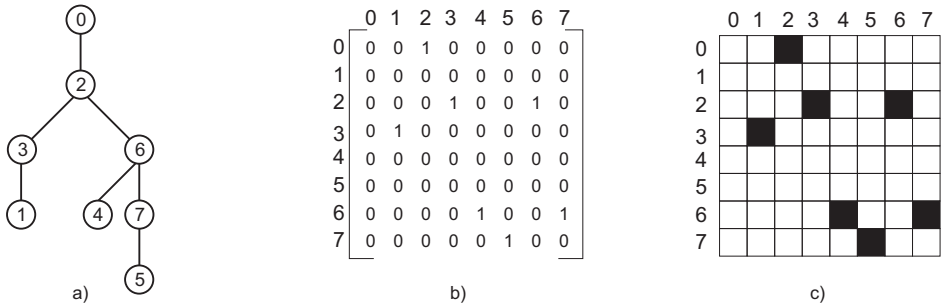


Figura A.2: Representação de uma árvore geradora e sua matriz correspondente

pondente ao índice do vértice zerada (vértices $1, 4, 5$ da figura A.2). Um fato importante a salientar, nessa convenção em se ler a matriz de adjacências, é que a aresta $x-y$ é considerada distinta da aresta $y-x$. Dessa forma o grafo torna-se direcionado e cada aresta orientada em uma direção específica deve pertencer a uma árvore distinta.

Seguindo essa forma de ler a matriz transformada, é necessário satisfazer um requisito das k árvores independentes: o vértice raiz deve possuir grau 1 nas k árvores distintas. Para isso, deve-se zerar a coluna dos valores correspondentes ao vértice raiz na matriz transformada. Para o vértice raiz 0 tem-se a matriz da figura A.3.

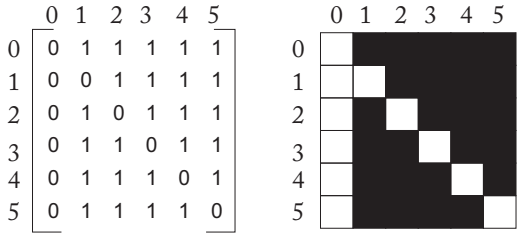


Figura A.3: Zerando a coluna 0 raiz

Para K_6, um grafo 5-conexo, a quantidade de arestas distintas necessárias para as k árvores é obtida pela fórmula citada anteriormente:

$$k \times (|V| - 1) = 5 \times (6 - 1) = 25 \; arestas \tag{A.1}$$

É trivial ver que não importando o vértice raiz, ou seja, não importando qual coluna zerar

na matriz de adjacências, teremos as 25 arestas necessárias para as 5 árvores geradoras de K_6, figura A.4.

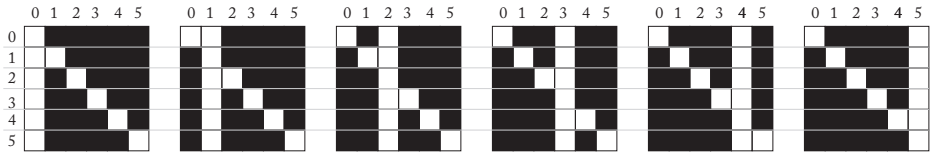


Figura A.4: Matriz transformada de K_6 com qualquer vértice como raiz

Agora é necessário demonstrar que para qualquer grafo completo K_n tem-se as arestas necessárias a serem distribuídas dentre as k árvores geradoras.

Para obtermos a matriz de K_7 temos que acrescentar mais uma linha e uma coluna à matriz de K_6. Cada linha acrescentada adiciona $|V| - 2$ arestas, visto que os valores da diagonal principal, além dos valores da coluna equivalente ao vértice raiz, são sempre zerados. Por sua vez cada coluna acrescentada adiciona $|V| - 1$ arestas.

Logo o número de arestas adicionadas, E_{add}, é dada pela equação:

$$\begin{aligned} E_{add} &= |V| - 2 + |V| - 1 \\ &= 2(|V|) - 3 \\ &= 2(k+1) - 2 \\ &= 2k + 2 - 3 \\ &= 2k + 1 \end{aligned}$$

Lembrando que o número de arestas necessárias para formar as k árvores, $|E_{kt}|$ é dada pela equação:

$$|E_{kt}| = k \times (|V| - 1) \tag{A.2}$$

Sendo que o grafo completo K_n é $n-1$ conexo, temos que o número de vértices do grafo K_n é dado pela equação:

$$|V| = k + 1 \tag{A.3}$$

Substituindo-se $|V|$ na equação anterior pelo seu valor $k + 1$ temos:

$$\begin{aligned} |E_{kt}| &= k \times (|V| - 1) \\ &= k \times ((k + 1) - 1) \\ &= k^2 \end{aligned}$$

Seja K_i o grafo K antes da adição das arestas, acrescentando-se a linha e coluna nova à matriz de adjacências. K_{ii} denota o valor após esta adição, por exemplo, para K_5 e K_6, K_5 seria K_i e K_6 seria K_{ii}.

Como:

$$|E_{K_i}| = k_i^2 \tag{A.5}$$

Queremos provar que o número de arestas de K_i mais as arestas adicionadas pela linha e coluna acrescentadas é igual ao número de arestas de K_{ii}

Prova:

$$\begin{aligned} |E_{K_i}| + |E_{add}| &= |E_{K_{ii}}| \\ k_i^2 + (2k_i + 1) &= (k_i + 1)^2 \\ k_i^2 + 2k_i + 1 &= k_i^2 + 2k_i + 1 \end{aligned}$$

□

Com isso demonstra-se que qualquer grafo completo K_n tem arestas suficientes para as k árvores geradoras independentes.

A seguir é explicado como as arestas são distribuídas entre as diferentes árvores e o que representa zerar a coluna correspondente ao vértice raiz na matriz de adjacências.

A figura A.5 mostra a matriz de adjacências de K_6 e as arestas direcionadas correspondentes a cada célula da matriz, as quais devem ser distribuídas entre as k árvores geradoras independentes.

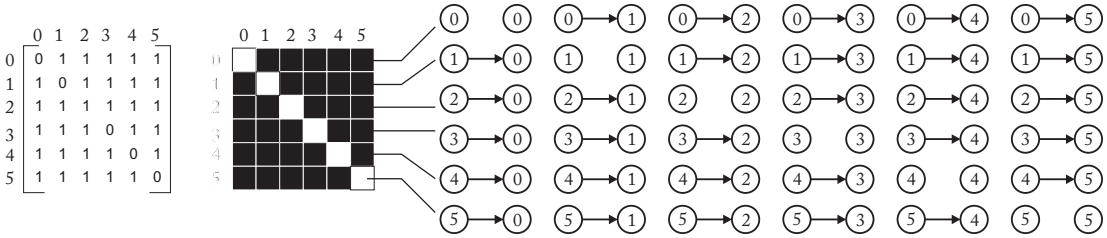


Figura A.5: Arestas de K_6

Na figura A.6 pode-se notar a razão pela qual as arestas correspondentes à coluna do vértice raiz em questão são removidas. Isso é feito para se evitar ciclos no grafo, os quais devem ser removidos para se obter árvores como grafos resultantes. A figura mostra k árvores geradoras independentes para K_6.

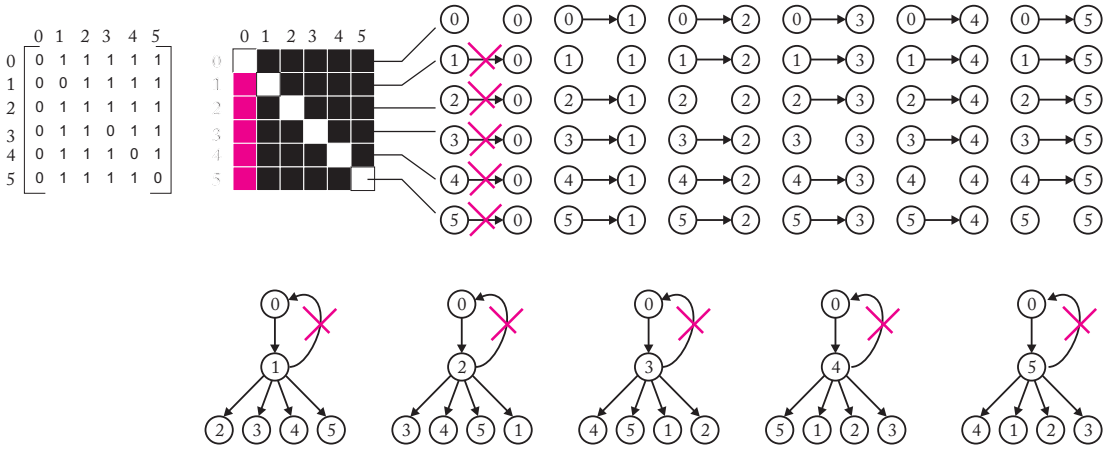


Figura A.6: Removendo-se os ciclos

Na figura A.7 são mostradas as arestas utilizadas para cada árvore. As árvores são facilmente formadas utilizando-se a aresta na linha zero, como raiz da árvore, e o restante das

arestas da linha correspondente, ex.: a linha 1 contém as arestas utilizadas na árvore 1, a linha 2 a lista de arestas da árvore 2 e assim por diante.

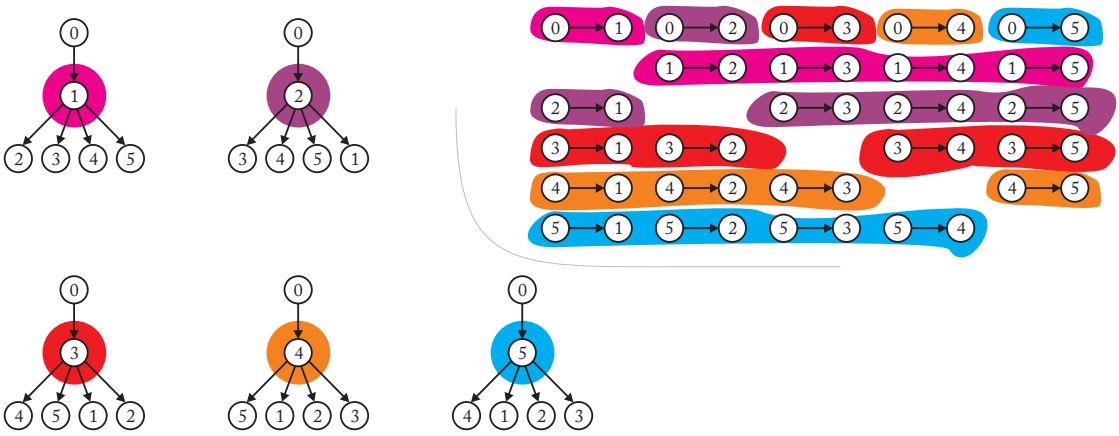


Figura A.7: Árvores e arestas utilizadas por elas em destaque

A figura A.8 mostra as 5 árvores geradoras de K_6 e suas matrizes correspondentes com raiz no vértice 0. As árvores cujo vértice raiz escolhido é o de índice 1 são mostradas na figura A.9.

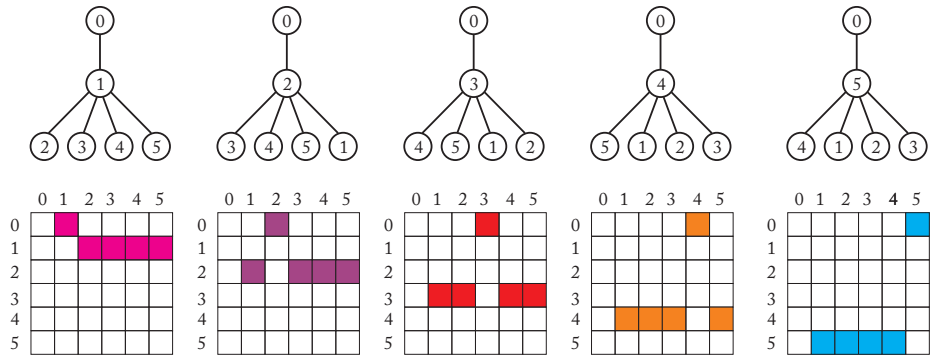


Figura A.8: Árvores com raiz 0 e matrizes equivalentes a sua representação

Com isso, observa-se que não importando o vértice escolhido como raiz da árvore do grafo completo, todas as árvores geradoras independentes são facilmente construídas.

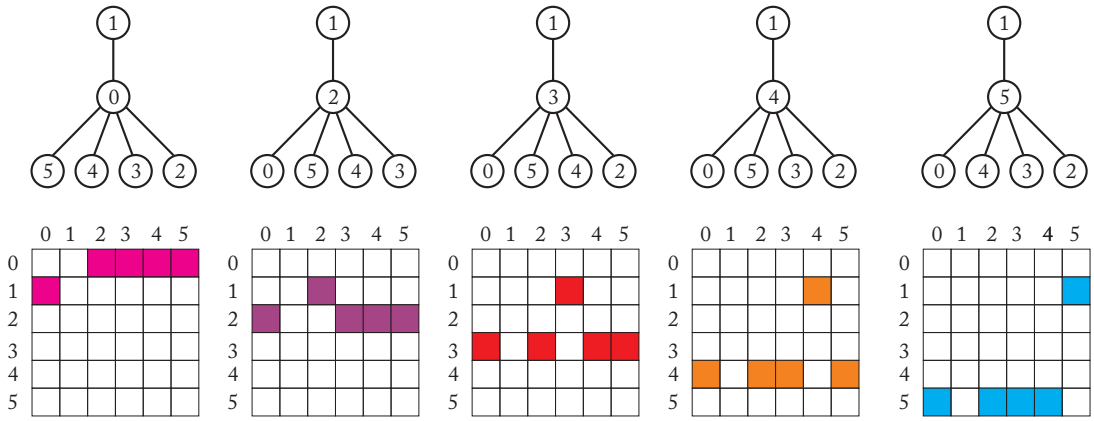
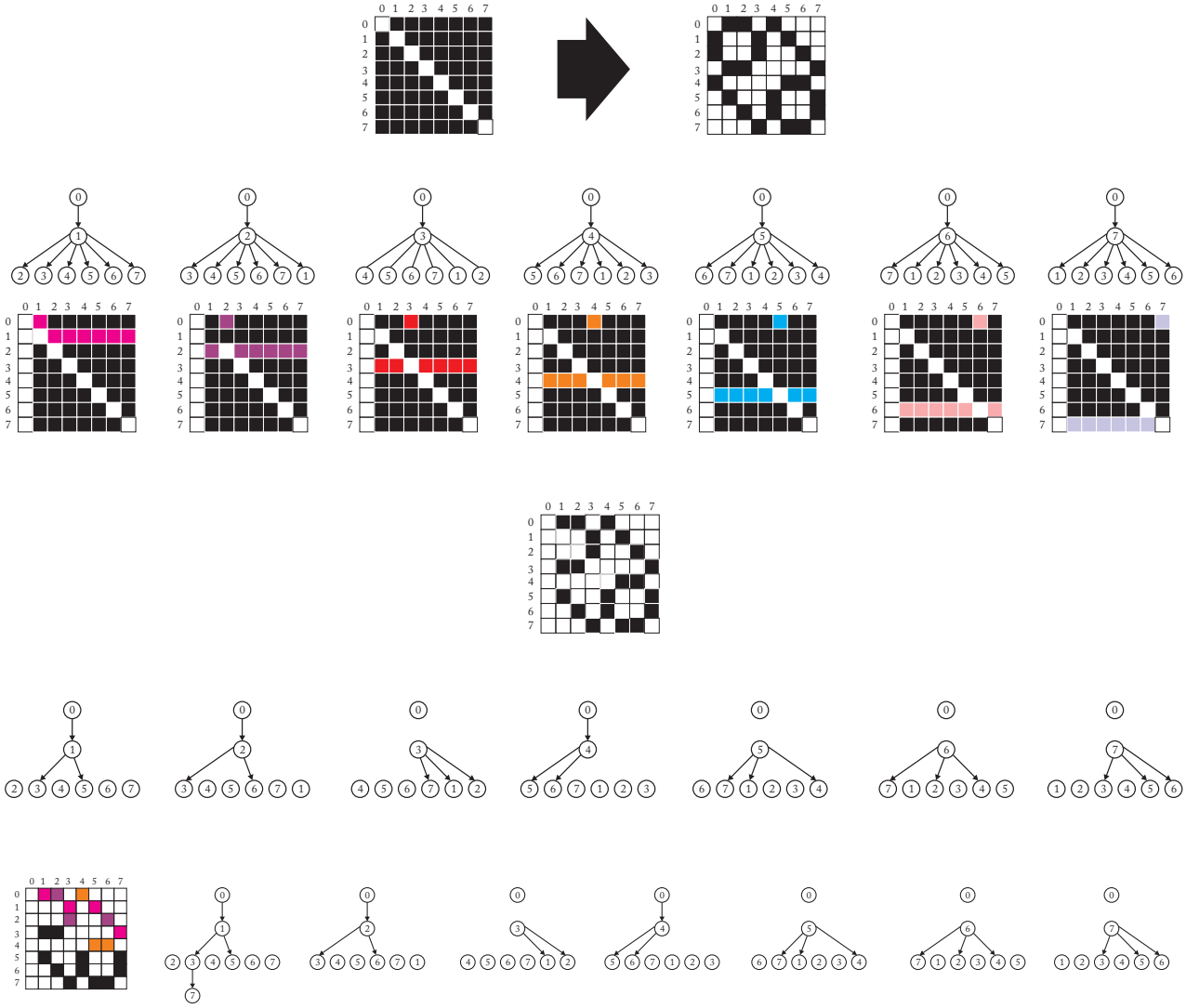


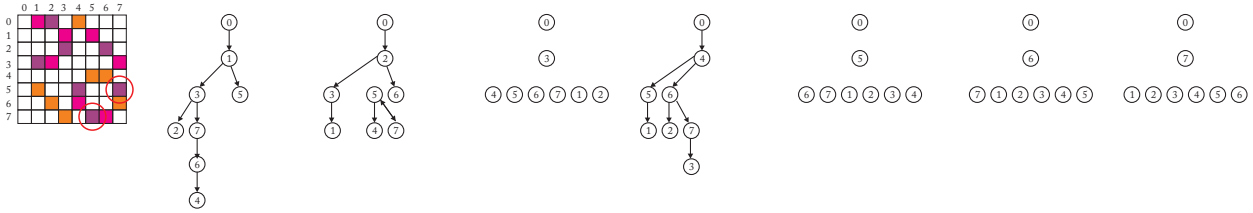
Figura A.9: Árvores com raiz 1 e matrizes equivalentes a sua representação

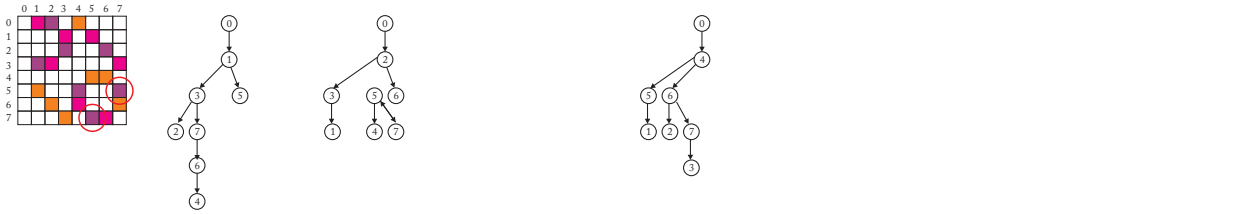
APÊNDICE B

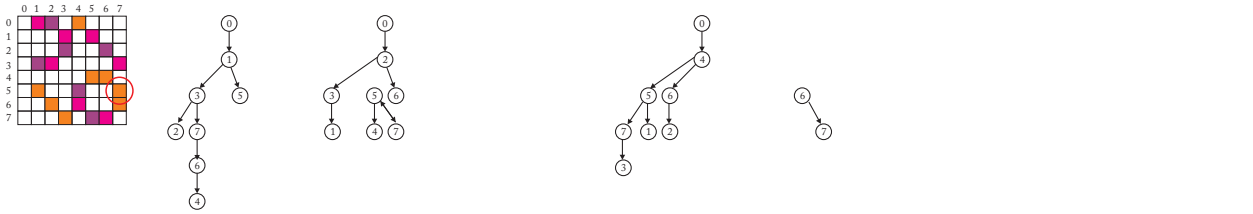
CALCULANDO Q_3 A PARTIR DE K_8

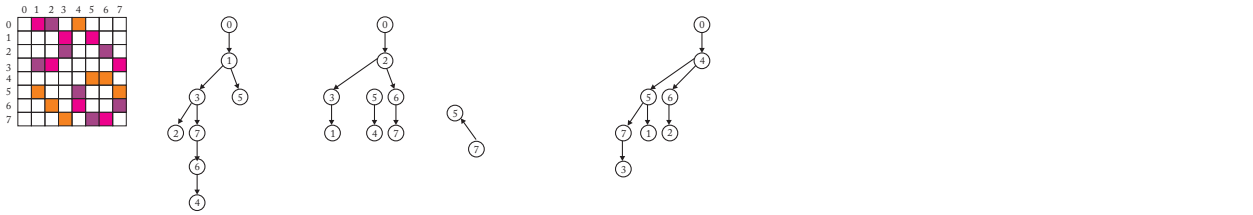
Este apêndice contém um exemplo de árvores geradoras obtidas pela remoção de arestas do grafo completo de mesma ordem. A sequência de passos a seguir mostra uma maneira de gerar as árvores de Q_3 a partir de K_8.

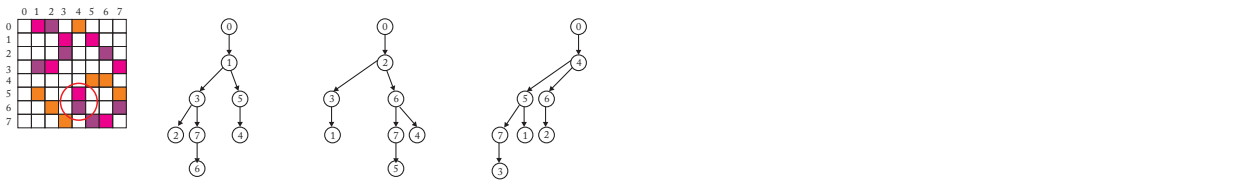


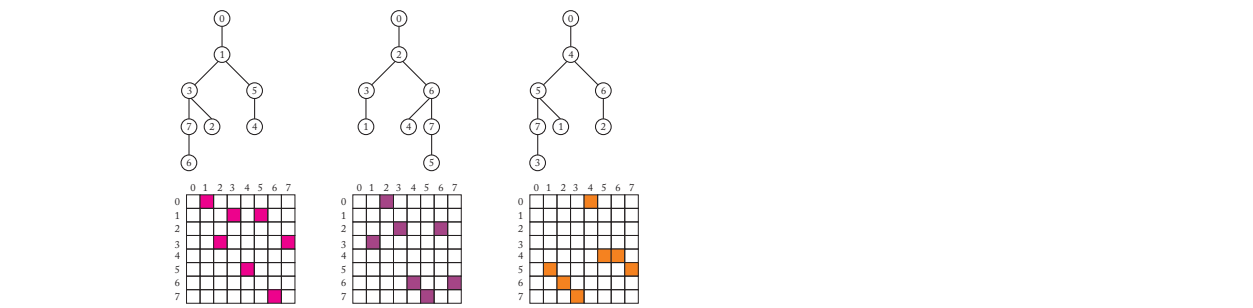










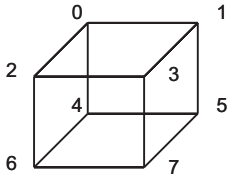


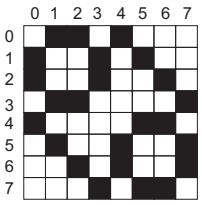
APÊNDICE C

DISTRIBUIÇÃO DAS ARESTAS

A seguir o algoritmo de distribuição de arestas *EdgeDistAdj*, proposto neste trabalho, é apresentado. O algoritmo utiliza como entrada a matriz de adjacências de Q_k para produzir uma distribuição de arestas dentre as k árvores. O algoritmo serve como ponto de partida para o algoritmo de otimização *OptimIST*. Porém, o algoritmo por si só, não garante que as árvores geradoras sejam independentes.

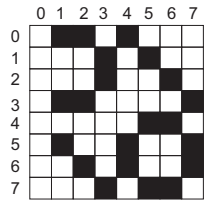
Considere o problema: Dado o hipercubo Q_3,representado por sua matriz de adjacências, ache as 3 árvores geradoras independentes com o vértice 0 como raiz.
Para facilitar a visualização a matriz de adjacências será também descrita por uma matriz cujo valor zero é representado pela cor branca, valor 1 pela cor preta e cada valor diferente de 0 e 1 por uma cor arbitrária.

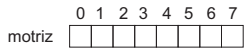




0	1	1	0	1	0	0	0
1	0	0	1	0	1	0	0
1	0	0	1	0	0	1	0
0	1	1	0	0	0	0	1
1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1
0	0	1	0	1	0	0	1
0	0	0	1	0	1	1	0

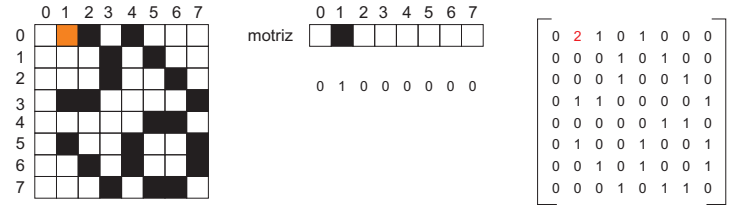
Primeiramente deve-se zerar os valores da coluna correspondente ao vértice raiz. Um vetor inicialmente vazio o qual chamaremos de vetor motriz será utilizado para auxiliar a computação.





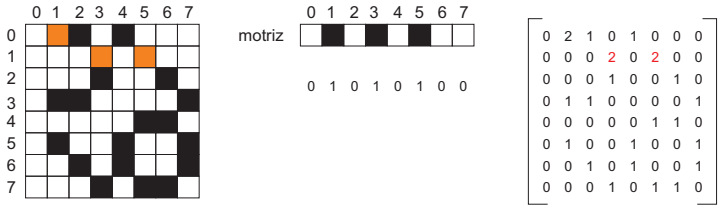
0	1	1	0	1	0	0	0
0	0	0	1	0	1	0	0
0	0	0	1	0	0	1	0
0	1	1	0	0	0	0	1
0	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1
0	0	1	0	1	0	0	1
0	0	0	1	0	1	1	0

Da linha zero pega-se o primeiro valor correspondente ao valor 1 e atribui-se uma cor a ele (na matriz de adjacência, o valor 2)

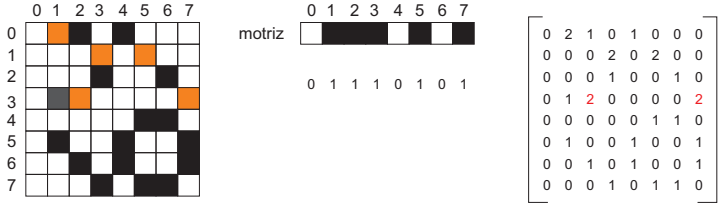


Na próxima linha, linha 1, verifica-se se na linha motriz o valor 1 está presente. Se o valor 1 estiver presente significa que o nó 1 está na árvore e pode ser utilizado como pai. Como a coluna 1 do vetor motriz possui a linha 1, busca-se então todos os valores em 1 da linha. Os valores são então trocados pelo valor 2 na matriz de adjacências e a linha motriz recebe os dois valores em 1.

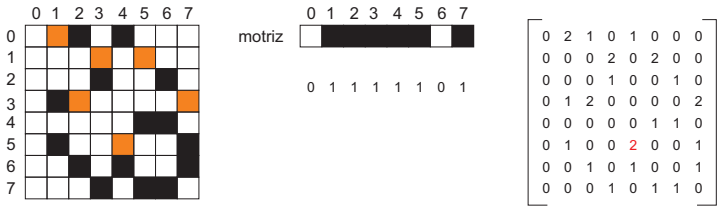
A próxima linha analisada é a linha 2. Como o nó 2 não está no vetor motriz, ignora-se a linha 2.



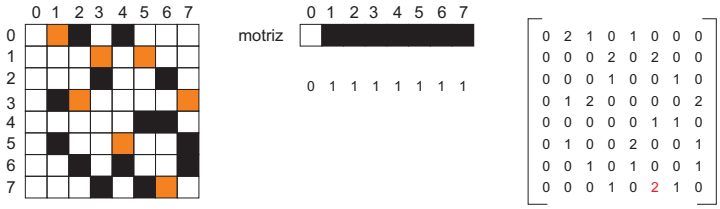
A próxima linha analisada é a linha 3. Como o nó 3 está na linha motriz, consideramos todos os valores em 1. Como a coluna 1 já está presente na linha motriz, ela é ignorada, as colunas 2 e 7 são adicionadas então a linha motriz e modificadas na matriz de adjacências.



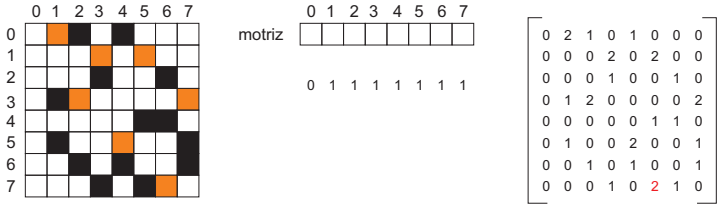
A próxima linha analisada é a linha 5, já que a linha 4 é ignorada visto que a coluna 4 no vetor motriz está zerada.



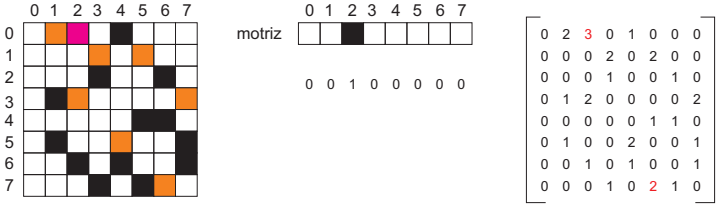
A próxima linha analisada é a linha 7, já que a linha 6 é ignorada visto que a coluna 6 no vetor motriz está zerada. Encerra-se então o processo para a primeira árvore, visto que o vetor motriz está completo com exceção da coluna 0.



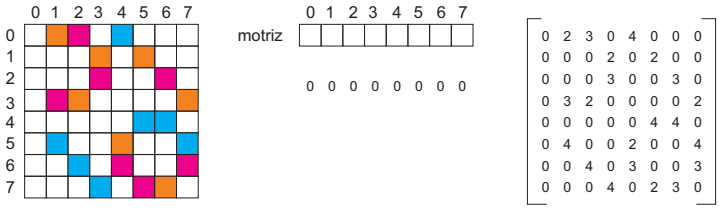
Para a próxima árvore reinicia-se o processo com a linha motriz zerada. Lembre-se que os valores analisados são somente aqueles em 1, assim os valores trocados no ciclo anterior são ignorados.



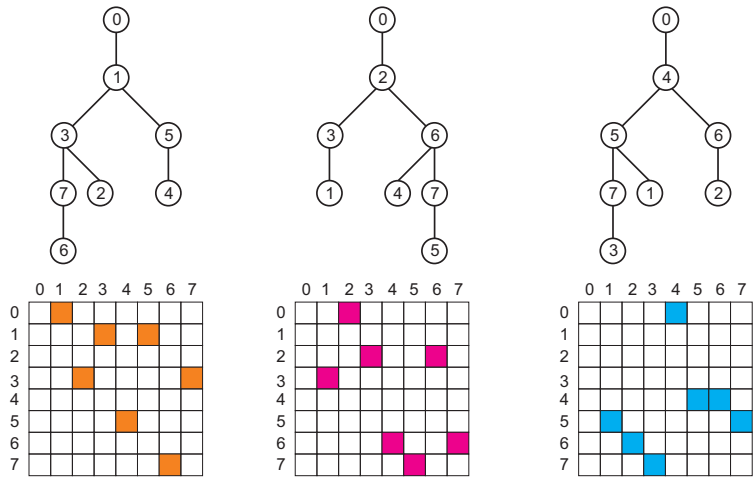
Para a primeira linha consideramos somente o primeiro valor em 1.



Como o processo é repetitivo para as k-árvores apenas a matriz final resultante é mostrada:



No final do processo as seguintes árvores são formadas pela nova matriz de adjacências.



APÊNDICE D

MODELO SIMULINK DE GERAÇÃO DE ARESTAS PARA Q_4

Com o objetivo de mostrar a simplicidade do algoritmo proposto *MinimalIST*, uma simulação do algoritmo em circuito foi feita utilizando-se o software *Matlab Simulink* para Q_4 (figura D.1). Cada bloco customizado é descrito com exceção da implementação do *bit-array*, visto que esta é trivial. Para Q_4 é necessário um *bit-array* com 16 bits (2^4). Na figura D.2 pode-se notar que as arestas geradas para a primeira árvore são finalizadas em aproximadamente 120 ciclos. O circuito, juntamente com o algoritmo *MinimalIST*, é parte do artigo apresentado oralmente na conferência *IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2013)* [102].

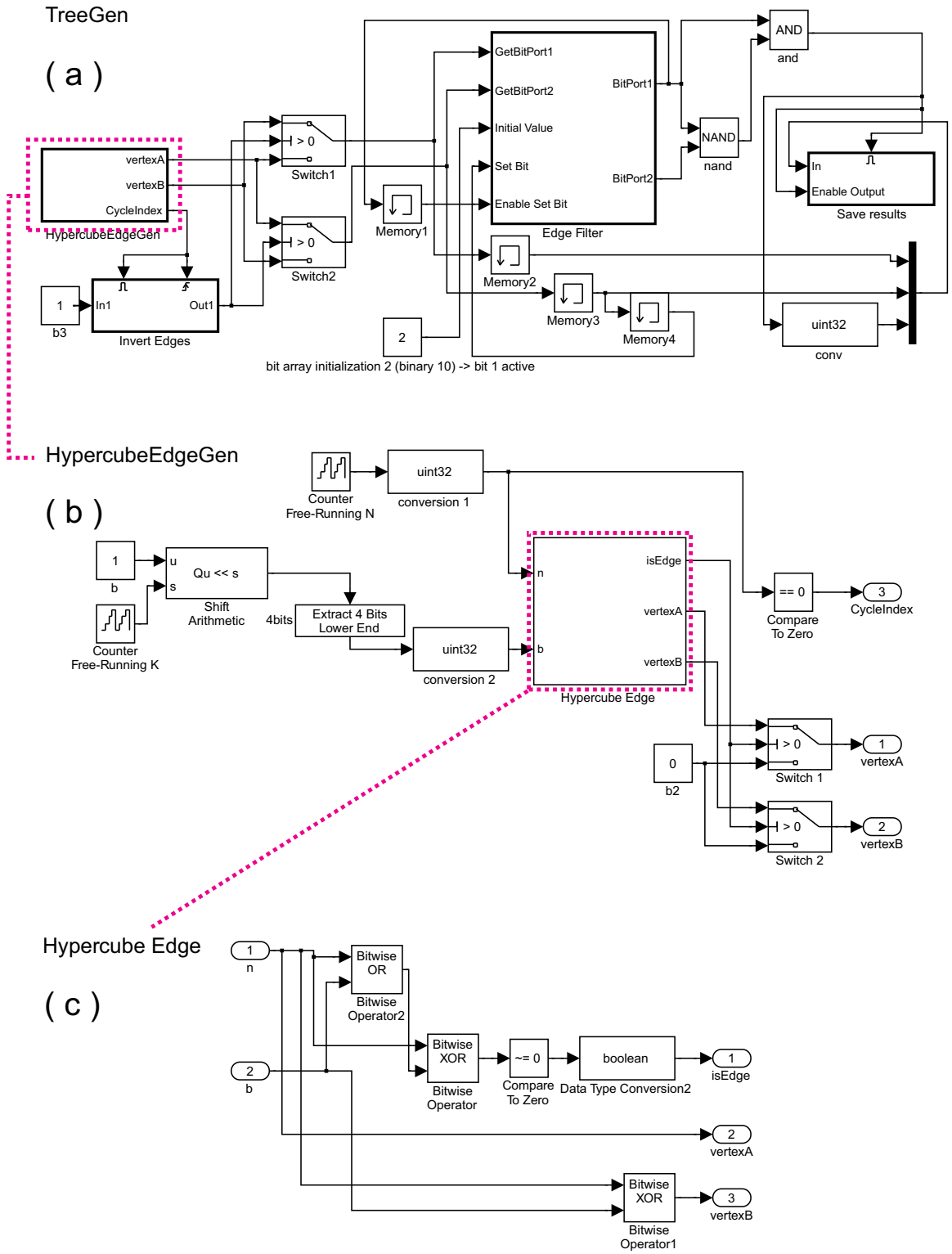


Figura D.1: Bloco utilizado para calcular a primeira árvore geradora sobre Q_4. (a) Circuito completo. (b) *HypercubeEdgeGen* em detalhes. (c) *HypercubeEdge* em detalhes.

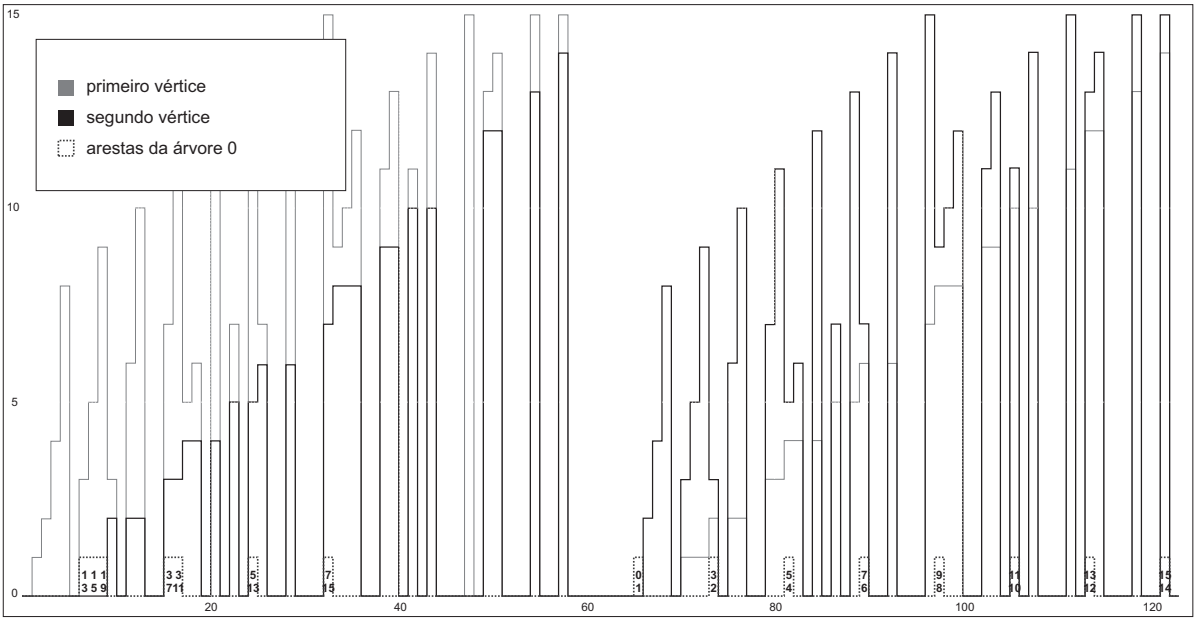


Figura D.2: Diagrama de onda MatLab Simulink. Eixo X representa os ciclos, eixo Y a entrada representando a aresta composta de dois vértices. Caso a aresta pertença a árvore T_0 ela é destacada pela linha pontilhada. Somente T_0 é construída, pois as outras árvores podem ser derivadas dela pelo deslocamento de *bits*.

APÊNDICE E

CÓDIGO VHDL PARA TRANSFORMAR ALGORITMO ECUBE EM IST

Neste apêndice é apresentada a aplicação do algoritmo proposto para o cálculo das rotas entre os vértices utilizando as árvores geradoras independentes. A implementação é feita utilizando-se VHDL (*VHSIC Description Language*) e contém somente a parte modificada do algoritmo *ECUBE*, sendo que qualquer implementação do *ECUBE* já existente pode continuar sendo utilizada. A implementação retrata a equação E.1, a árvore utilizada deve ser T_k, sendo que as outras árvores devem ser derivadas via operação de deslocamento de *bits* à esquerda, logo $tree = k$ para $T_1, T_2, ..., T_k$, ou $tree = k - 1$ para $T_0, T_1, ..., T_{k-1}$, dependendo da convenção utilizada para se definir a primeira árvore.

(E.1)

$$source \oplus (0 \Big|\Big| eCube(2^{tree}, source \oplus target))$$

E.1 Código VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY ecube2ist IS
PORT (
source: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
shift: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
outp: BUFFER STD_LOGIC_VECTOR (7 DOWNTO 0);
nsource: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
);
END ecube2ist;
```

```
ARCHITECTURE behavior OF ecube2ist IS
BEGIN
PROCESS (shift)
VARIABLE temp1: STD_LOGIC_VECTOR (7 DOWNTO 0);
VARIABLE temp2: STD_LOGIC_VECTOR (7 DOWNTO 0);
VARIABLE inp: STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN

inp := "00000001";

IF (shift(0)='0') THEN
        temp1 := inp;
ELSE
        temp1(0) := '0';
        FOR i IN 1 TO inp'HIGH LOOP
        temp1(i) := inp(i-1);
        END LOOP;
END IF;

IF (shift(1)='0') THEN
        temp2 := temp1;
ELSE
        FOR i IN 0 TO 1 LOOP
                temp2(i) := '0';
        END LOOP;

        FOR i IN 2 TO inp'HIGH LOOP
                temp2(i) := temp1(i-2);
        END LOOP;
END IF;
```

```
----- 3rd shifter -----
IF (shift(2)='0') THEN
        outp <= temp2;
ELSE
        FOR i IN 0 TO 3 LOOP
                outp(i) <= '0';
        END LOOP;
        FOR i IN 4 TO inp'HIGH LOOP
                outp(i) <= temp2(i-4);
        END LOOP;

END IF;

END PROCESS;

nsource <= source XOR outp;

END behavior;
```

E.2 Estimativa de consumo de recursos da placa FPGA

A listagem a seguir representa o consumo esperado de recursos da placa FPGA.

```
 Build: PlanAhead v14.4 by xbuild
  Report: by esantana on host snowball, pid 14981

Report for pblock: ROOT


Physical Resource Estimates
===========================

Site Type       Available  Required  % Util
-------------------------------------------
LUT             46560      16        1
FD_LD           93120      0         0
SLICEL          7460       3         1
SLICEM          4180       2         1
BSCAN           4          0         0
BUFGCTRL        32         0         0
BUFHCE          72         0         0
BUFIODQS        36         0         0
BUFR            18         0         0
```

CAPTURE	1	0	0
CFG_IO_ACCESS	1	0	0
DCI	9	0	0
DCIRESET	1	0	0
DNA_PORT	1	0	0
DSP48E1	288	0	0
EFUSE_USR	1	0	0
FRAME_ECC	1	0	0
GTXE1	8	0	0
IBUFDS_GTXE1	6	0	0
ICAP	2	0	0
IDELAYCTRL	9	0	0
ILOGICE1	240	0	0
IODELAYE1	360	0	0
MMCM_ADV	6	0	0
OLOGICE1	240	0	0
PCIE_2_0	1	0	0
PMVBRAM	21	0	0
PMVIOB	2	0	0
RAMBFIFO36E1	156	0	0
STARTUP	1	0	0
SYSMON	1	0	0
TEMAC_SINGLE	4	0	0
USR_ACCESS	1	0	0

IO Statistics

IO Ports	Bonded IOs	Utilization
27	240	11.2 %

Primitive Statistics

Primitive type	Count
LUT	16
IO	27

A figura E.1 descreve o bloco esquemático gerado pelo software ISE da Xilinx.

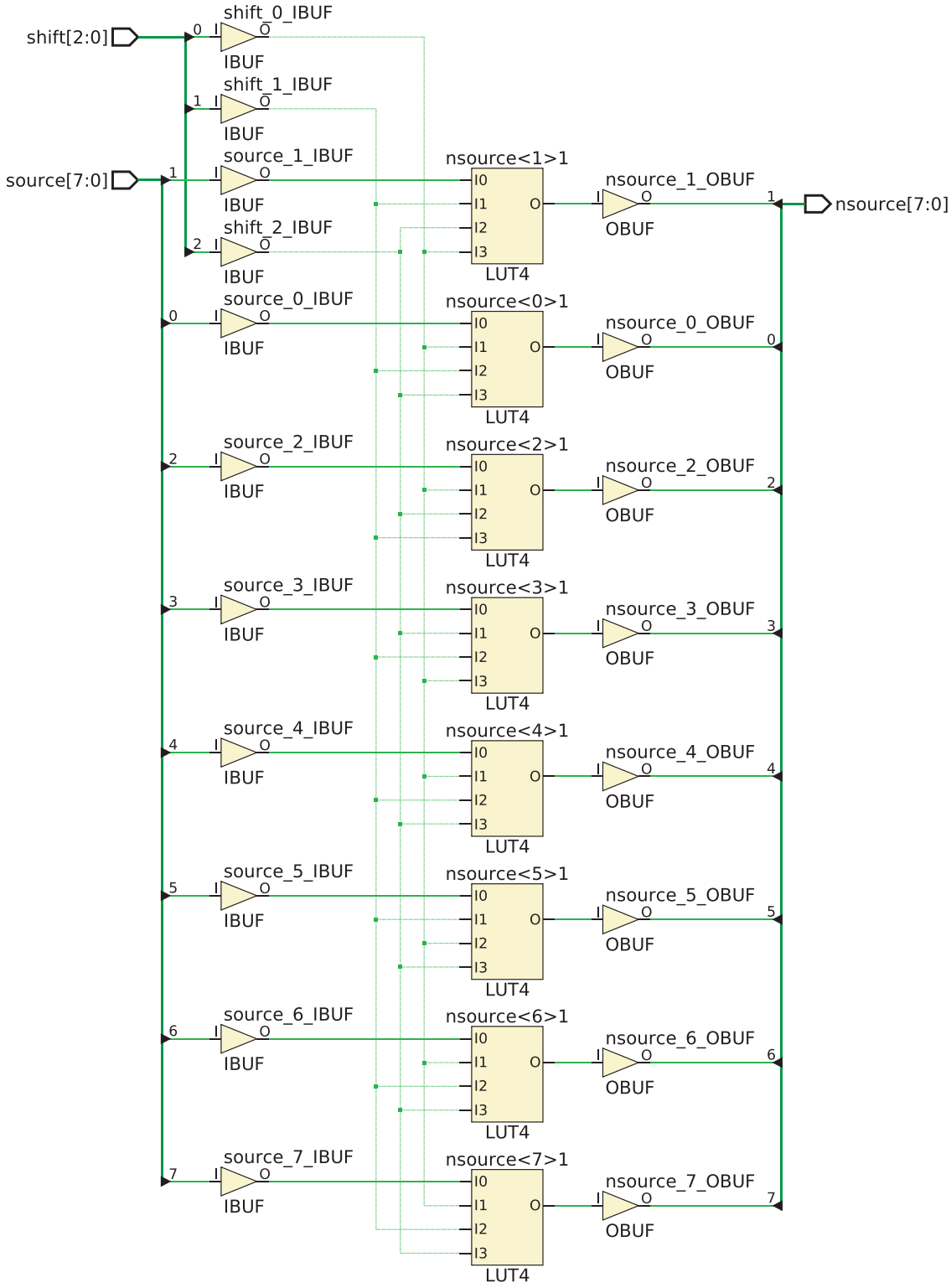


Figura E.1: Diagrama esquemático gerado pela ferramenta ISE da Xilinx.

E.3 Simulação utilizando o software ISE da Xilinx

Na figura E.2 o resultado da simulação do código VHDL é mostrado, utilizando-se o vértice 13 como vértice fonte da mensagem.

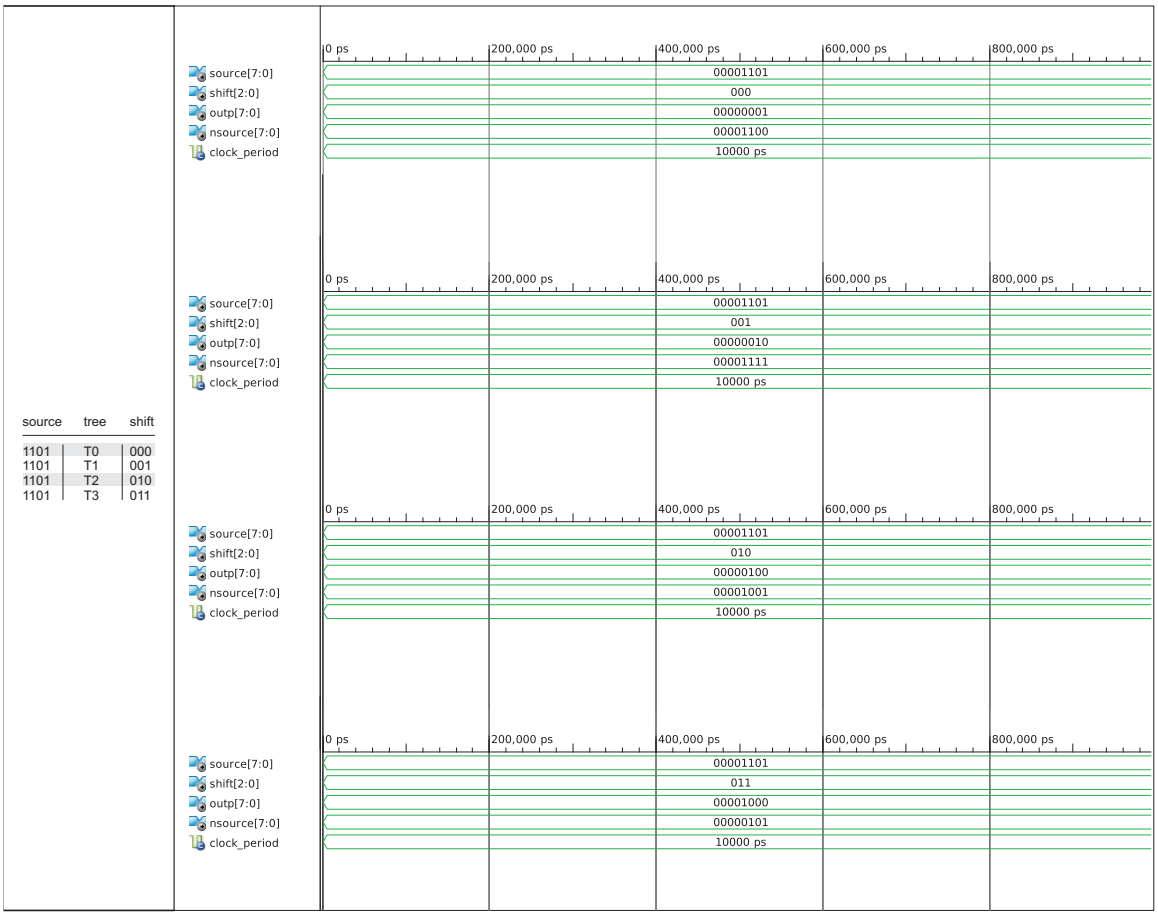


Figura E.2: Simulação do algoritmo em VHDL utilizando-se ISim da Xilinx

APÊNDICE F

DADOS ESTATÍSTICOS DOS TESTES

Neste apêndice são apresentadas as análises estatísticas dos dados gerados durante a simulação de rotas. Cada variável é explicada a seguir:

— ROUND: índice de execução;

— FAILURES: quantidade de vértices falhos simulados;

— CN: número de vértices concorrentes, transmitindo ao mesmo tempo;

— ACC_MESSAGES: valor acumulado que representa quantidade de mensagens;

— F_ECUBE: número de rotas afetadas por vértices falhos, utilizando ECUBE;

— F_DOR: número de rotas afetadas por vértices falhos, utilizando DOR;

— F_IST: número de rotas afetadas por vértices falhos, utilizando IST;

— ACCF_ECUBE: valor acumulado de falhas utilizando ECUBE;

— ACCF_DOR: valor acumulado de falhas utilizando DOR;

— ACCF_IST: valor acumulado de falhas utilizando IST;

— C_ECUBE: quantidade de vértices suscetíveis a congestionamento, utilizando ECUBE;

— C_DOR: quantidade de vértices suscetíveis a congestionamento, utilizando DOR;

— C_IST: quantidade de vértices suscetíveis a congestionamento, utilizando IST;

— BACKLOG_ECUBE: quantidade de mensagens presentes no sistema, para ECUBE;

— BACKLOG_DOR: quantidade de mensagens presentes no sistema, para DOR;

— BACKLOG_IST: quantidade de mensagens presentes no sistema, para IST;

A simulação funciona da seguinte forma, a cada iteração (*ROUND*) da simulação são injetadas no sistema 512 mensagens. Cada rota percorrida por uma mensagem é comparada com as rotas dos vértices concorrentes. Cada nó comum a mais de um caminho é contabilizado como suscetível a congestionamento. Essa análise, apesar de ser mais pessimista, descreve um modelo mais preciso. Pessimista devido ao fato de que, mesmo tendo vértices comuns a rotas concorrentes, o que importa é em que intevalo de tempo cada trecho estaria sendo utilizado. Somente trechos utilizados ao mesmo tempo teriam problemas em relação à concorrência.

Os dados a seguir descrevem experimentos, com 500 execuções cada, para cada uma das seguintes configurações:

— Primeira simulação, 1024 vértices, 102 (10%) nos falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação (figura F.1);

— Segunda simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação (figura F.2);

— Primeira simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação (figura F.3);

— Segunda simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação (figura F.4);

— Primeira simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação (figura F.5);

— Segunda simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação (figura F.6);

— Primeira simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação (figura F.7);

— Segunda simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação (figura F.8);

— Primeira simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação (figura F.9);

— Segunda simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação (figura F.10);

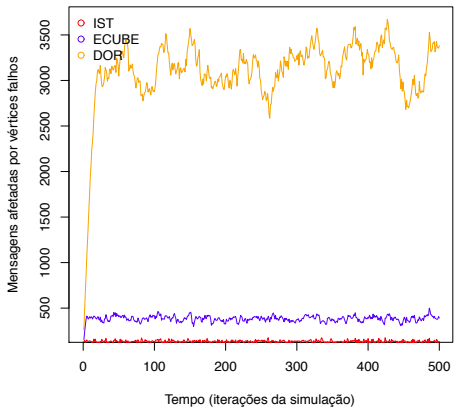
— Primeira simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação (figura F.11);

— Segunda simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação (figura F.12);

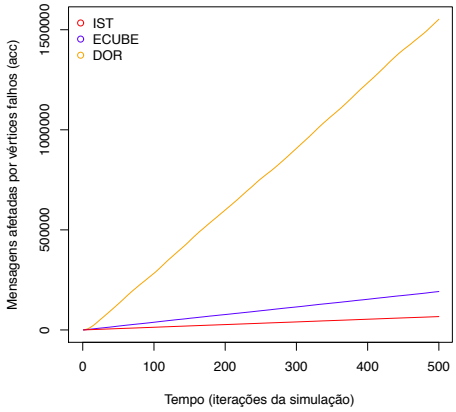
O termo *com recuperação* significa que caso a rota escolhida contenha um vértice falho, uma nova rota é utilizada. Caso um vértice falho ainda esteja presente na nova rota escolhida, outra rota é sucessivamente escolhida até que as k rotas se esgotem. No caso do termo *sem recuperação* o vértice não faz nada caso a rota contenha um vértice falho e a falha é contabilizada.

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :102   Min.   :512   Min.   :   512   Min.   :155.0
 1st Qu.:124.8   1st Qu.:102   1st Qu.:512   1st Qu.: 64384   1st Qu.:365.0
 Median :249.5   Median :102   Median :512   Median :128256   Median :385.0
 Mean   :249.5   Mean   :102   Mean   :512   Mean   :128256   Mean   :383.9
 3rd Qu.:374.2   3rd Qu.:102   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:405.0
 Max.   :499.0   Max.   :102   Max.   :512   Max.   :256000   Max.   :501.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   : 89.0   Min.   : 261   Min.   :   155   Min.   :  135
 1st Qu.:123.8   1st Qu.:2989   1st Qu.: 48806   1st Qu.:16946
 Median :133.0   Median :3147   Median : 96094   Median :33530
 Mean   :133.5   Mean   :3105   Mean   : 96120   Mean   :33527
 3rd Qu.:143.0   3rd Qu.:3305   3rd Qu.:143699   3rd Qu.:50162
 Max.   :175.0   Max.   :3672   Max.   :191956   Max.   :66763
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    261   Min.   :   133   Min.   :    162   Min.   :    1855
 1st Qu.: 367241   1st Qu.:112293   1st Qu.: 485664   1st Qu.: 4749588
 Median : 756553   Median :224402   Median : 982528   Median : 9738521
 Mean   : 760500   Mean   :223706   Mean   : 980637   Mean   : 9718612
 3rd Qu.:1151291   3rd Qu.:334839   3rd Qu.:1475262   3rd Qu.:14686350
 Max.   :1552352   Max.   :448300   Max.   :1977570   Max.   :19682701
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1284   1st Qu.:3442   1st Qu.:1566
 Median :1304   Median :3495   Median :1588
 Mean   :1302   Mean   :3449   Mean   :1576
 3rd Qu.:1326   3rd Qu.:3534   3rd Qu.:1607
 Max.   :1396   Max.   :3642   Max.   :1650
```

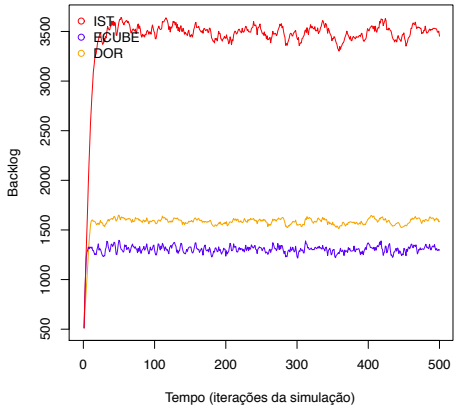
(a)



(b)



(c)

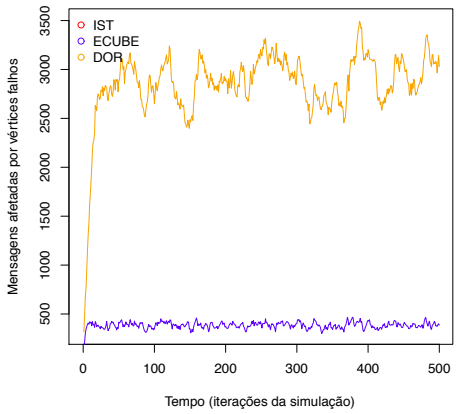


(d)

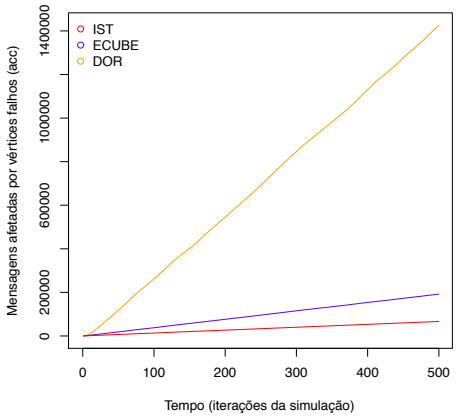
Figura F.1: Primeira simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação. (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
Min.   :  0.0   Min.   :102   Min.   :512   Min.   :   512   Min.   :156.0
1st Qu.:124.8   1st Qu.:102   1st Qu.:512   1st Qu.: 64384   1st Qu.:364.0
Median :249.5   Median :102   Median :512   Median :128256   Median :386.0
Mean   :249.5   Mean   :102   Mean   :512   Mean   :128256   Mean   :384.3
3rd Qu.:374.2   3rd Qu.:102   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:406.0
Max.   :499.0   Max.   :102   Max.   :512   Max.   :256000   Max.   :467.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
Min.   : 99.0   Min.   : 317   Min.   :   156   Min.   :  128
1st Qu.:124.0   1st Qu.:2730   1st Qu.: 48239   1st Qu.:16965
Median :133.0   Median :2880   Median : 95905   Median :33425
Mean   :133.3   Mean   :2852   Mean   : 95993   Mean   :33443
3rd Qu.:141.0   3rd Qu.:3060   3rd Qu.:143739   3rd Qu.:50024
Max.   :178.0   Max.   :3493   Max.   :192143   Max.   :66632
    ACCF_DOR          C_ECUBE           C_IST             C_DOR
Min.   :    317   Min.   :   142   Min.   :    165   Min.   :    1736
1st Qu.: 338784   1st Qu.:110035   1st Qu.: 474667   1st Qu.: 4647254
Median : 693301   Median :218272   Median : 961004   Median : 9568932
Mean   : 697438   Mean   :218122   Mean   : 959929   Mean   : 9574664
3rd Qu.:1051266   3rd Qu.:326656   3rd Qu.:1444856   3rd Qu.:14488449
Max.   :1425956   Max.   :436802   Max.   :1937423   Max.   :19410920
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
Min.   : 512   Min.   : 512   Min.   : 512
1st Qu.:1272   1st Qu.:3406   1st Qu.:1554
Median :1296   Median :3466   Median :1576
Mean   :1292   Mean   :3418   Mean   :1565
3rd Qu.:1320   3rd Qu.:3510   3rd Qu.:1595
Max.   :1406   Max.   :3648   Max.   :1656
```

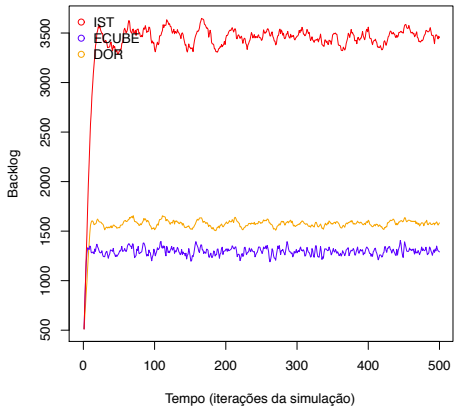
(a)



(b)



(c)

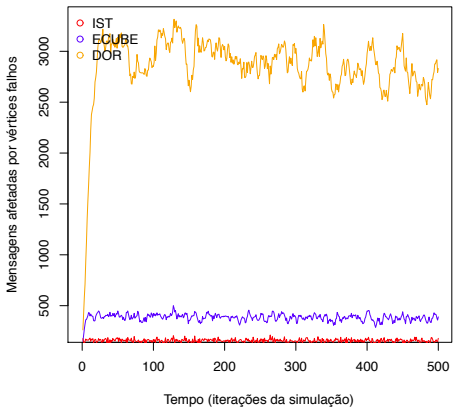


(d)

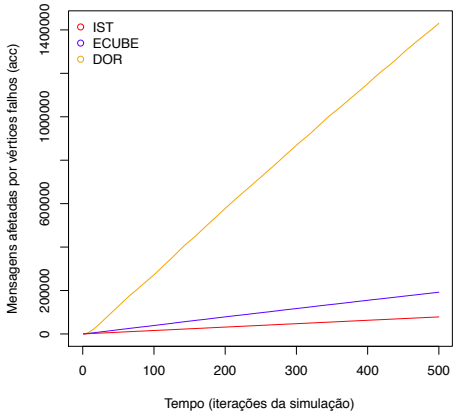
Figura F.2: Segunda simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação. (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :102   Min.   :512   Min.   :   512   Min.   :150.0
 1st Qu.:124.8   1st Qu.:102   1st Qu.:512   1st Qu.: 64384   1st Qu.:364.8
 Median :249.5   Median :102   Median :512   Median :128256   Median :388.0
 Mean   :249.5   Mean   :102   Mean   :512   Mean   :128256   Mean   :384.5
 3rd Qu.:374.2   3rd Qu.:102   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:404.0
 Max.   :499.0   Max.   :102   Max.   :512   Max.   :256000   Max.   :500.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   :108.0   Min.   : 262   Min.   :   150   Min.   :  176
 1st Qu.:146.0   1st Qu.:2765   1st Qu.: 49161   1st Qu.:19844
 Median :157.0   Median :2894   Median : 97804   Median :39460
 Mean   :156.9   Mean   :2861   Mean   : 97261   Mean   :39459
 3rd Qu.:168.0   3rd Qu.:3049   3rd Qu.:145443   3rd Qu.:59072
 Max.   :212.0   Max.   :3315   Max.   :192244   Max.   :78433
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    262   Min.   :   120   Min.   :    124   Min.   :    1839
 1st Qu.: 352240   1st Qu.:111180   1st Qu.: 479414   1st Qu.: 4436748
 Median : 723400   Median :222497   Median : 970074   Median : 9111944
 Mean   : 717603   Mean   :221388   Mean   : 965384   Mean   : 9091722
 3rd Qu.:1082970   3rd Qu.:331350   3rd Qu.:1453313   3rd Qu.:13744090
 Max.   :1430551   Max.   :440550   Max.   :1930838   Max.   :18336686
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1274   1st Qu.:3338   1st Qu.:1560
 Median :1298   Median :3393   Median :1579
 Mean   :1293   Mean   :3350   Mean   :1568
 3rd Qu.:1322   3rd Qu.:3434   3rd Qu.:1597
 Max.   :1392   Max.   :3570   Max.   :1653
```

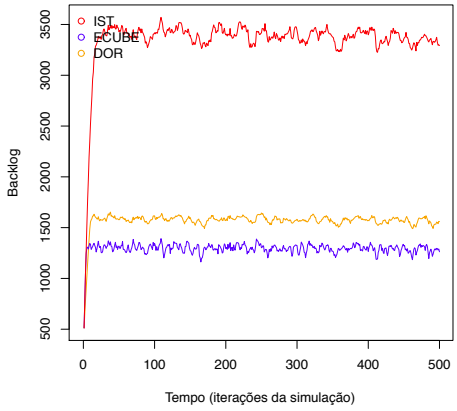
(a)



(b)



(c)

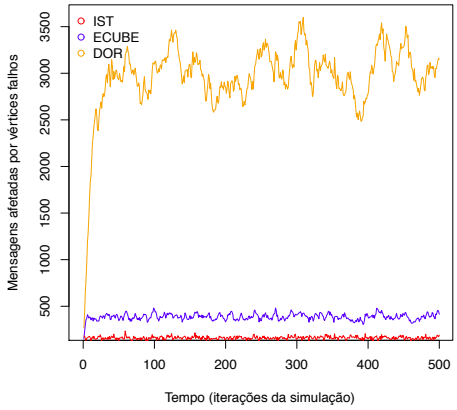


(d)

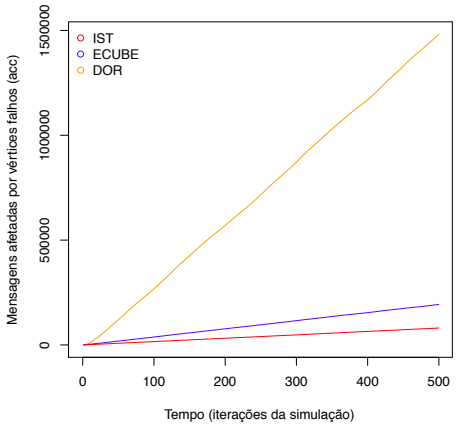
Figura F.3: Primeira Simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
Min.   :  0.0   Min.   :102   Min.   :512   Min.   :   512   Min.   :152.0
1st Qu.:124.8   1st Qu.:102   1st Qu.:512   1st Qu.: 64384   1st Qu.:365.0
Median :249.5   Median :102   Median :512   Median :128256   Median :386.0
Mean   :249.5   Mean   :102   Mean   :512   Mean   :128256   Mean   :386.5
3rd Qu.:374.2   3rd Qu.:102   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:409.0
Max.   :499.0   Max.   :102   Max.   :512   Max.   :256000   Max.   :482.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
Min.   :115.0   Min.   : 268   Min.   :   152   Min.   :  180
1st Qu.:150.8   1st Qu.:2843   1st Qu.: 48275   1st Qu.:20135
Median :161.0   Median :3000   Median : 96366   Median :40114
Mean   :161.3   Mean   :2964   Mean   : 96537   Mean   :40290
3rd Qu.:172.0   3rd Qu.:3139   3rd Qu.:145397   3rd Qu.:60525
Max.   :234.0   Max.   :3601   Max.   :193260   Max.   :80674
    ACCF_DOR         C_ECUBE          C_IST             C_DOR
Min.   :    268   Min.   :   125   Min.   :    138   Min.   :    1681
1st Qu.: 349566   1st Qu.:109384   1st Qu.: 473794   1st Qu.: 4691089
Median : 718031   Median :220950   Median : 969240   Median : 9771744
Mean   : 725433   Mean   :221362   Mean   : 969874   Mean   : 9810970
3rd Qu.:1103350   3rd Qu.:333141   3rd Qu.:1465429   3rd Qu.:14909890
Max.   :1481842   Max.   :443486   Max.   :1958968   Max.   :20013341
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
Min.   : 512   Min.   : 512   Min.   : 512
1st Qu.:1282   1st Qu.:3470   1st Qu.:1570
Median :1302   Median :3517   Median :1590
Mean   :1301   Mean   :3472   Mean   :1579
3rd Qu.:1326   3rd Qu.:3558   3rd Qu.:1606
Max.   :1410   Max.   :3706   Max.   :1656
```

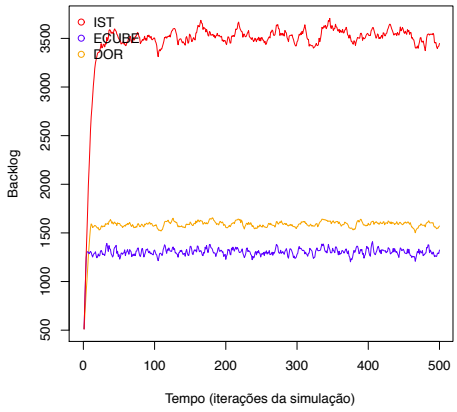
(a)



(b)



(c)

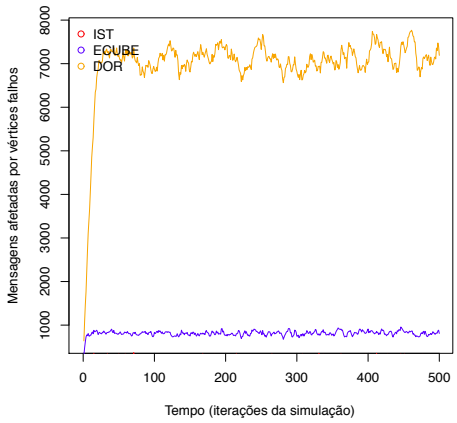


(d)

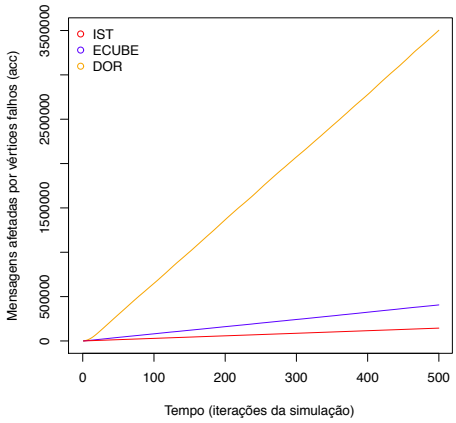
Figura F.4: Segunda simulação, 1024 vértices, 102 (10%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação. (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :204   Min.   :512   Min.   :   512   Min.   :308.0
 1st Qu.:124.8   1st Qu.:204   1st Qu.:512   1st Qu.: 64384   1st Qu.:782.8
 Median :249.5   Median :204   Median :512   Median :128256   Median :814.0
 Mean   :249.5   Mean   :204   Mean   :512   Mean   :128256   Mean   :812.5
 3rd Qu.:374.2   3rd Qu.:204   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:845.0
 Max.   :499.0   Max.   :204   Max.   :512   Max.   :256000   Max.   :957.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   :234.0   Min.   : 638   Min.   :   308   Min.   :   263
 1st Qu.:274.0   1st Qu.:6930   1st Qu.:101624   1st Qu.: 36413
 Median :289.5   Median :7118   Median :202000   Median : 72288
 Mean   :289.3   Mean   :7006   Mean   :202403   Mean   : 72356
 3rd Qu.:305.0   3rd Qu.:7270   3rd Qu.:303361   3rd Qu.:108389
 Max.   :367.0   Max.   :7767   Max.   :406244   Max.   :144674
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    638   Min.   :   107   Min.   :    117   Min.   :    1686
 1st Qu.: 834630   1st Qu.:111676   1st Qu.: 477719   1st Qu.: 4777265
 Median :1725754   Median :221246   Median : 963350   Median : 9801846
 Mean   :1722520   Mean   :221149   Mean   : 963486   Mean   : 9775937
 3rd Qu.:2603454   3rd Qu.:330856   3rd Qu.:1450195   3rd Qu.:14761540
 Max.   :3503040   Max.   :443793   Max.   :1944278   Max.   :19775414
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1278   1st Qu.:3448   1st Qu.:1557
 Median :1298   Median :3484   Median :1577
 Mean   :1298   Mean   :3450   Mean   :1567
 3rd Qu.:1324   3rd Qu.:3530   3rd Qu.:1599
 Max.   :1420   Max.   :3684   Max.   :1653
```

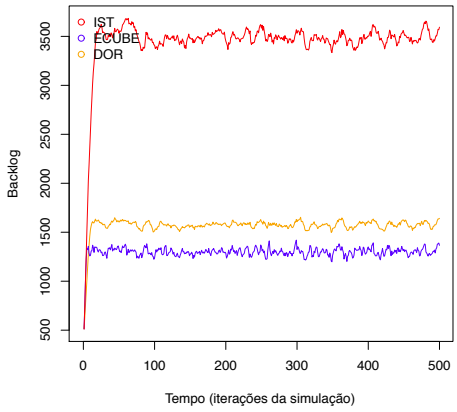
(a)



(b)



(c)

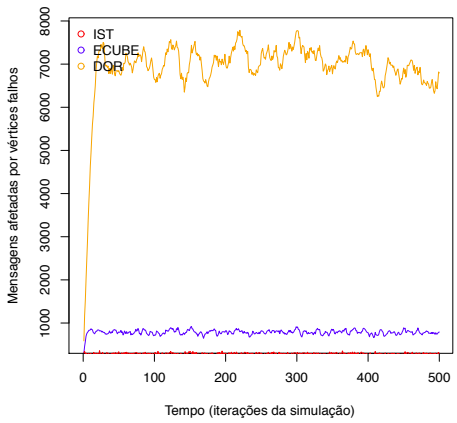


(d)

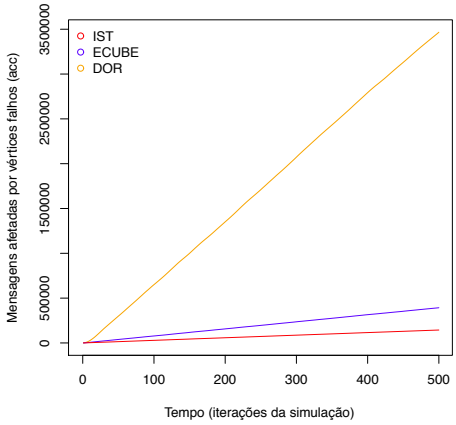
Figura F.5: Primeira simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :204   Min.   :512   Min.   :   512   Min.   :310.0
 1st Qu.:124.8   1st Qu.:204   1st Qu.:512   1st Qu.: 64384   1st Qu.:753.0
 Median :249.5   Median :204   Median :512   Median :128256   Median :786.0
 Mean   :249.5   Mean   :204   Mean   :512   Mean   :128256   Mean   :785.4
 3rd Qu.:374.2   3rd Qu.:204   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:820.2
 Max.   :499.0   Max.   :204   Max.   :512   Max.   :256000   Max.   :919.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   :237.0   Min.   : 586   Min.   :   310   Min.   :   307
 1st Qu.:274.0   1st Qu.:6800   1st Qu.: 98248   1st Qu.: 36550
 Median :288.0   Median :7025   Median :196366   Median : 72548
 Mean   :288.2   Mean   :6933   Mean   :196697   Mean   : 72414
 3rd Qu.:302.0   3rd Qu.:7258   3rd Qu.:295206   3rd Qu.:108450
 Max.   :359.0   Max.   :7785   Max.   :392687   Max.   :144122
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    586   Min.   :   107   Min.   :    146   Min.   :    1693
 1st Qu.: 831625   1st Qu.:112203   1st Qu.: 481826   1st Qu.: 4559195
 Median :1713820   Median :223968   Median : 967662   Median : 9389788
 Mean   :1719393   Mean   :223361   Mean   : 967783   Mean   : 9401509
 3rd Qu.:2611752   3rd Qu.:334517   3rd Qu.:1455635   3rd Qu.:14249570
 Max.   :3466360   Max.   :444854   Max.   :1938229   Max.   :18995695
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1278   1st Qu.:3390   1st Qu.:1559
 Median :1302   Median :3444   Median :1577
 Mean   :1300   Mean   :3406   Mean   :1570
 3rd Qu.:1326   3rd Qu.:3498   3rd Qu.:1599
 Max.   :1426   Max.   :3648   Max.   :1681
```

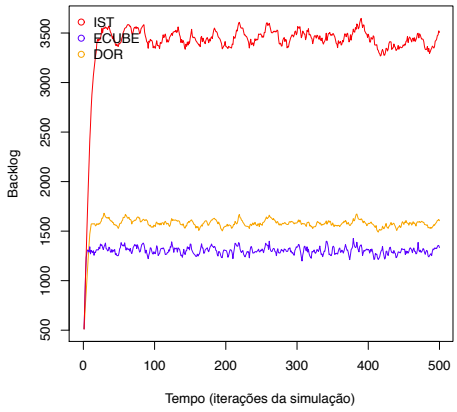
(a)



(b)



(c)

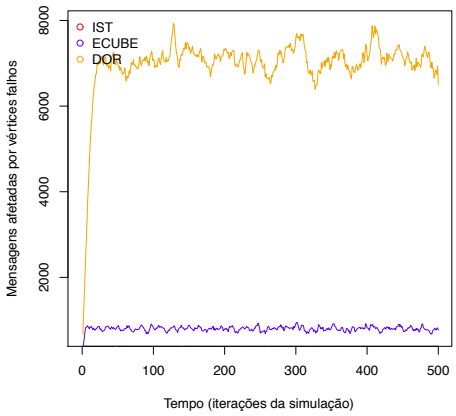


(d)

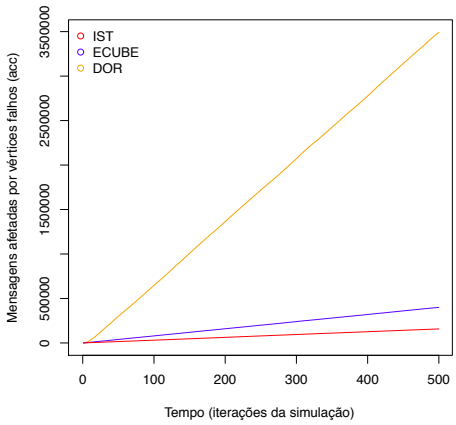
Figura F.6: Segunda simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :204   Min.   :512   Min.   :   512   Min.   :293.0
 1st Qu.:124.8   1st Qu.:204   1st Qu.:512   1st Qu.: 64384   1st Qu.:769.0
 Median :249.5   Median :204   Median :512   Median :128256   Median :804.0
 Mean   :249.5   Mean   :204   Mean   :512   Mean   :128256   Mean   :800.0
 3rd Qu.:374.2   3rd Qu.:204   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:833.2
 Max.   :499.0   Max.   :204   Max.   :512   Max.   :256000   Max.   :947.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   :247.0   Min.   : 680   Min.   :   293   Min.   :   313
 1st Qu.:298.0   1st Qu.:6913   1st Qu.: 99902   1st Qu.: 39337
 Median :315.0   Median :7068   Median :200180   Median : 79241
 Mean   :315.2   Mean   :6986   Mean   :200201   Mean   : 79059
 3rd Qu.:330.0   3rd Qu.:7233   3rd Qu.:300054   3rd Qu.:118508
 Max.   :387.0   Max.   :7935   Max.   :399987   Max.   :157591
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    680   Min.   :    98   Min.   :    129   Min.   :    1649
 1st Qu.: 829642   1st Qu.:109732   1st Qu.: 476930   1st Qu.: 4555136
 Median :1722788   Median :221346   Median : 976831   Median : 9366706
 Mean   :1721359   Mean   :220898   Mean   : 970414   Mean   : 9327968
 3rd Qu.:2603918   3rd Qu.:332204   3rd Qu.:1464294   3rd Qu.:14097100
 Max.   :3492889   Max.   :440684   Max.   :1940255   Max.   :18783885
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1276   1st Qu.:3386   1st Qu.:1564
 Median :1300   Median :3434   Median :1584
 Mean   :1296   Mean   :3389   Mean   :1573
 3rd Qu.:1324   3rd Qu.:3476   3rd Qu.:1603
 Max.   :1420   Max.   :3596   Max.   :1675
```

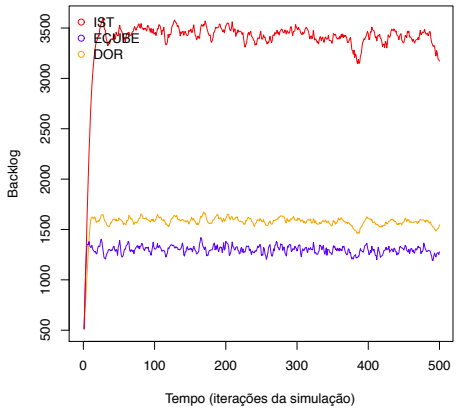
(a)



(b)



(c)

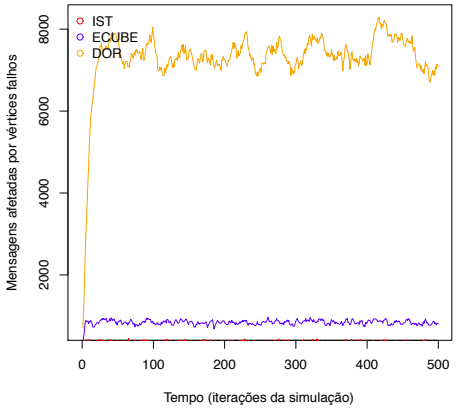


(d)

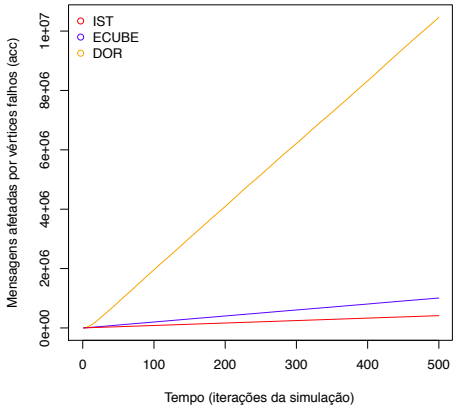
Figura F.7: Primeira simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
Min.   :  0.0   Min.   :204   Min.   :512   Min.   :   512   Min.   :349.0
1st Qu.:124.8   1st Qu.:204   1st Qu.:512   1st Qu.: 64384   1st Qu.:803.8
Median :249.5   Median :204   Median :512   Median :128256   Median :841.0
Mean   :249.5   Mean   :204   Mean   :512   Mean   :128256   Mean   :839.5
3rd Qu.:374.2   3rd Qu.:204   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:877.0
Max.   :499.0   Max.   :204   Max.   :512   Max.   :256000   Max.   :967.0
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
Min.   :283.0   Min.   : 707   Min.   :   349   Min.   :   363
1st Qu.:343.8   1st Qu.:7159   1st Qu.:105362   1st Qu.: 46058
Median :364.0   Median :7357   Median :208799   Median : 91572
Mean   :364.2   Mean   :7299   Mean   :210011   Mean   : 91429
3rd Qu.:381.0   3rd Qu.:7603   3rd Qu.:315050   3rd Qu.:136647
Max.   :450.0   Max.   :8291   Max.   :419753   Max.   :182085
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
Min.   :    707   Min.   :   107   Min.   :    131   Min.   :    1652
1st Qu.: 877284   1st Qu.:110135   1st Qu.: 473636   1st Qu.: 4654642
Median :1789668   Median :220226   Median : 962154   Median : 9568700
Mean   :1798068   Mean   :220720   Mean   : 964252   Mean   : 9567853
3rd Qu.:2715612   3rd Qu.:331372   3rd Qu.:1455232   3rd Qu.:14470345
Max.   :3649566   Max.   :442124   Max.   :1948090   Max.   :19448037
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
Min.   : 512   Min.   : 512   Min.   : 512
1st Qu.:1276   1st Qu.:3430   1st Qu.:1568
Median :1300   Median :3478   Median :1584
Mean   :1297   Mean   :3430   Mean   :1574
3rd Qu.:1320   3rd Qu.:3518   3rd Qu.:1601
Max.   :1396   Max.   :3638   Max.   :1659
```

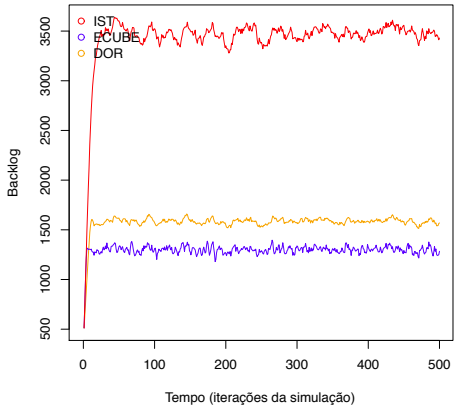
(a)



(b)



(c)

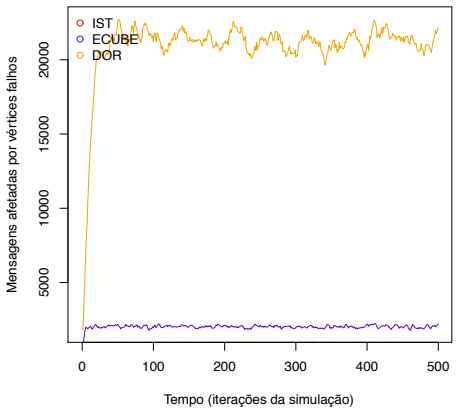


(d)

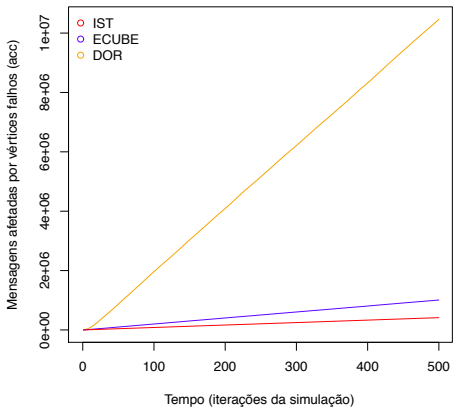
Figura F.8: Segunda simulação, 1024 vértices, 204 (20%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :512   Min.   :512   Min.   :   512   Min.   : 761
 1st Qu.:124.8   1st Qu.:512   1st Qu.:512   1st Qu.: 64384   1st Qu.:1958
 Median :249.5   Median :512   Median :512   Median :128256   Median :2024
 Mean   :249.5   Mean   :512   Mean   :512   Mean   :128256   Mean   :2018
 3rd Qu.:374.2   3rd Qu.:512   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:2082
 Max.   :499.0   Max.   :512   Max.   :512   Max.   :256000   Max.   :2231
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   :698.0   Min.   : 1809   Min.   :    761   Min.   :   845
 1st Qu.:801.0   1st Qu.:20766   1st Qu.: 252646   1st Qu.:104410
 Median :826.5   Median :21296   Median : 505187   Median :207442
 Mean   :828.0   Mean   :20932   Mean   : 504529   Mean   :207500
 3rd Qu.:855.0   3rd Qu.:21656   3rd Qu.: 756070   3rd Qu.:310593
 Max.   :943.0   Max.   :22730   Max.   :1008902   Max.   :413984
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    1809   Min.   :   118   Min.   :    134   Min.   :    1647
 1st Qu.: 2504455   1st Qu.:111504   1st Qu.: 470188   1st Qu.: 4542929
 Median : 5168887   Median :220768   Median : 952573   Median : 9380202
 Mean   : 5160690   Mean   :220202   Mean   : 950288   Mean   : 9349256
 3rd Qu.: 7795596   3rd Qu.:329000   3rd Qu.:1427025   3rd Qu.:14133415
 Max.   :10465988   Max.   :439578   Max.   :1915333   Max.   :18932012
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1272   1st Qu.:3392   1st Qu.:1557
 Median :1298   Median :3433   Median :1577
 Mean   :1295   Mean   :3395   Mean   :1568
 3rd Qu.:1320   3rd Qu.:3474   3rd Qu.:1595
 Max.   :1392   Max.   :3646   Max.   :1663
```

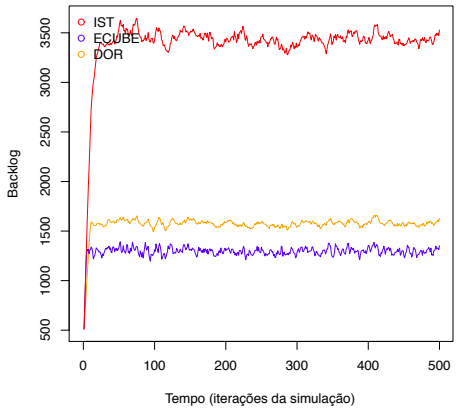
(a)



(b)



(c)

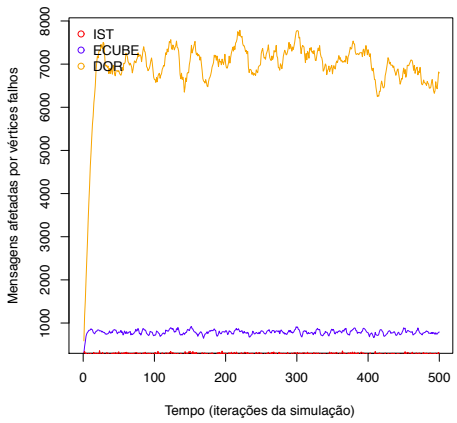


(d)

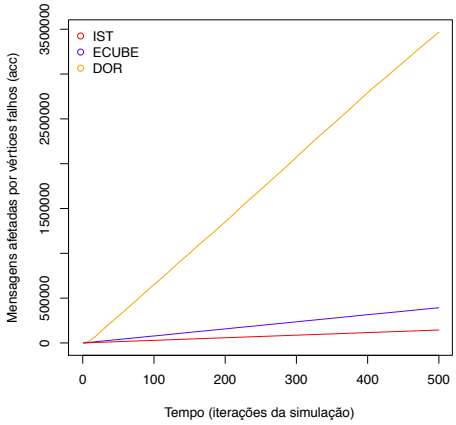
Figura F.9: Primeira simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :512   Min.   :512   Min.   :   512   Min.   : 741
 1st Qu.:124.8   1st Qu.:512   1st Qu.:512   1st Qu.: 64384   1st Qu.:1956
 Median :249.5   Median :512   Median :512   Median :128256   Median :2024
 Mean   :249.5   Mean   :512   Mean   :512   Mean   :128256   Mean   :2017
 3rd Qu.:374.2   3rd Qu.:512   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:2088
 Max.   :499.0   Max.   :512   Max.   :512   Max.   :256000   Max.   :2311
     F_IST           F_DOR         ACCF_ECUBE        ACCF_IST
 Min.   :678.0   Min.   : 1633   Min.   :    741   Min.   :   772
 1st Qu.:751.0   1st Qu.:20504   1st Qu.: 249753   1st Qu.: 97568
 Median :776.0   Median :20850   Median : 503150   Median :194370
 Mean   :776.6   Mean   :20564   Mean   : 503489   Mean   :194388
 3rd Qu.:801.0   3rd Qu.:21241   3rd Qu.: 756134   3rd Qu.:291273
 Max.   :892.0   Max.   :22824   Max.   :1008360   Max.   :388285
    ACCF_DOR          C_ECUBE          C_IST             C_DOR
 Min.   :    1633   Min.   :   133   Min.   :    135   Min.   :    1472
 1st Qu.: 2454457   1st Qu.:109130   1st Qu.: 467347   1st Qu.: 4412028
 Median : 5046342   Median :219898   Median : 954848   Median : 9128490
 Mean   : 5070482   Mean   :220359   Mean   : 957670   Mean   : 9142118
 3rd Qu.: 7682288   3rd Qu.:331036   3rd Qu.:1444629   3rd Qu.:13859096
 Max.   :10282210   Max.   :443324   Max.   :1936406   Max.   :18551591
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1280   1st Qu.:3376   1st Qu.:1564
 Median :1302   Median :3422   Median :1585
 Mean   :1300   Mean   :3382   Mean   :1575
 3rd Qu.:1326   3rd Qu.:3472   3rd Qu.:1603
 Max.   :1426   Max.   :3612   Max.   :1659
```

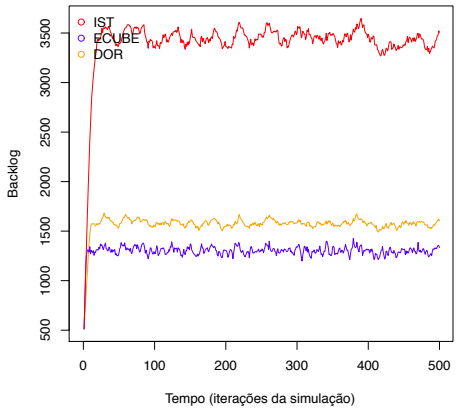
(a)



(b)



(c)

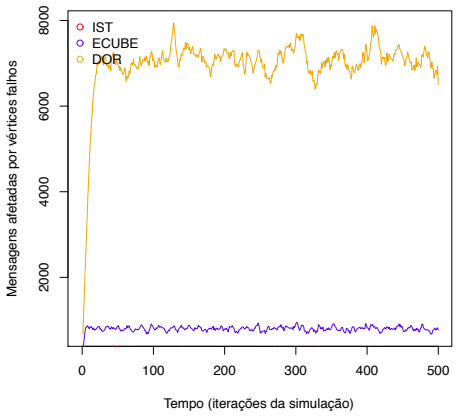


(d)

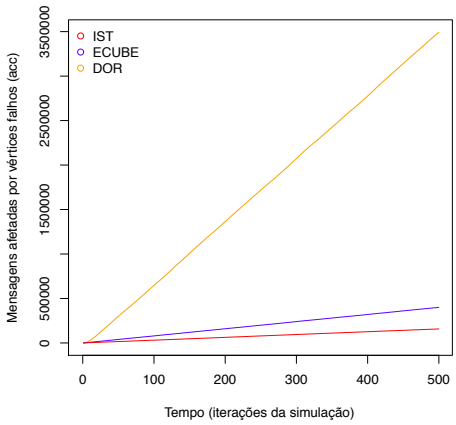
Figura F.10: Segunda simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, com recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
 Min.   :  0.0   Min.   :512   Min.   :512   Min.   :   512   Min.   : 821
 1st Qu.:124.8   1st Qu.:512   1st Qu.:512   1st Qu.: 64384   1st Qu.:1967
 Median :249.5   Median :512   Median :512   Median :128256   Median :2022
 Mean   :249.5   Mean   :512   Mean   :512   Mean   :128256   Mean   :2018
 3rd Qu.:374.2   3rd Qu.:512   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:2084
 Max.   :499.0   Max.   :512   Max.   :512   Max.   :256000   Max.   :2259
     F_IST           F_DOR        ACCF_ECUBE        ACCF_IST
 Min.   :716.0   Min.   : 1949   Min.   :    821   Min.   :   910
 1st Qu.:815.8   1st Qu.:20019   1st Qu.: 252401   1st Qu.:106902
 Median :842.0   Median :20470   Median : 506901   Median :212049
 Mean   :842.4   Mean   :20168   Mean   : 505000   Mean   :211572
 3rd Qu.:868.0   3rd Qu.:20881   3rd Qu.: 756510   3rd Qu.:316318
 Max.   :974.0   Max.   :21901   Max.   :1009085   Max.   :421212
    ACCF_DOR          C_ECUBE          C_IST            C_DOR
 Min.   :    1949   Min.   :   148   Min.   :    170   Min.   :    1768
 1st Qu.: 2439246   1st Qu.:112759   1st Qu.: 479982   1st Qu.: 4421950
 Median : 5016042   Median :224745   Median : 974722   Median : 9064979
 Mean   : 4990153   Mean   :223130   Mean   : 970174   Mean   : 9033576
 3rd Qu.: 7533043   3rd Qu.:333355   3rd Qu.:1457721   3rd Qu.:13630631
 Max.   :10084042   Max.   :444455   Max.   :1953740   Max.   :18301583
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
 Min.   : 512   Min.   : 512   Min.   : 512
 1st Qu.:1276   1st Qu.:3346   1st Qu.:1563
 Median :1296   Median :3398   Median :1586
 Mean   :1297   Mean   :3358   Mean   :1574
 3rd Qu.:1324   3rd Qu.:3458   3rd Qu.:1606
 Max.   :1434   Max.   :3564   Max.   :1676
```

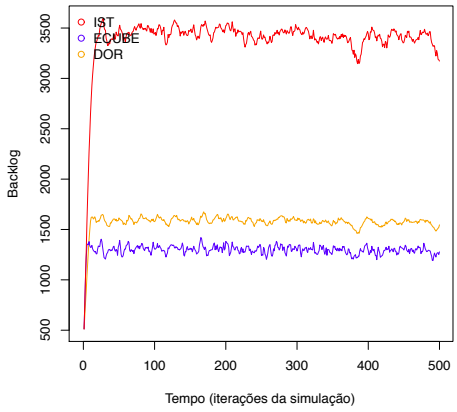
(a)



(b)



(c)

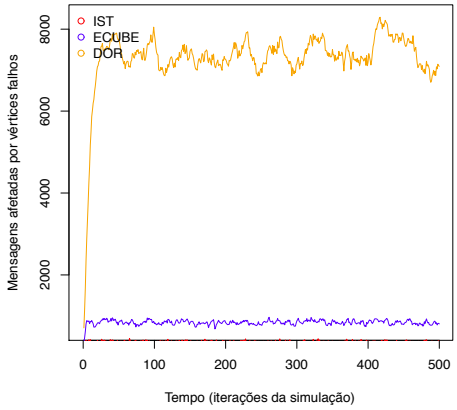


(d)

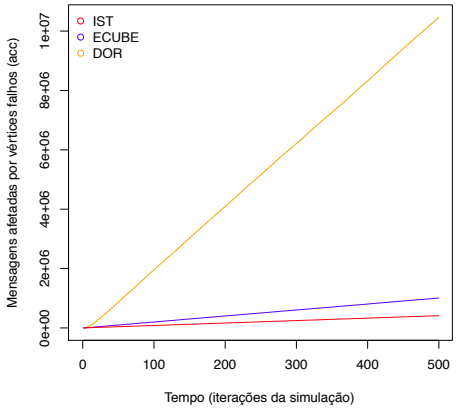
Figura F.11: Primeira simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;

```
     ROUND          FAILURES         CN       ACC_MESSAGES      F_ECUBE
Min.   :  0.0   Min.   :512   Min.   :512   Min.   :   512   Min.   : 770
1st Qu.:124.8   1st Qu.:512   1st Qu.:512   1st Qu.: 64384   1st Qu.:1952
Median :249.5   Median :512   Median :512   Median :128256   Median :2012
Mean   :249.5   Mean   :512   Mean   :512   Mean   :128256   Mean   :2007
3rd Qu.:374.2   3rd Qu.:512   3rd Qu.:512   3rd Qu.:192128   3rd Qu.:2065
Max.   :499.0   Max.   :512   Max.   :512   Max.   :256000   Max.   :2243
     F_IST           F_DOR         ACCF_ECUBE        ACCF_IST
Min.   :700.0   Min.   : 1853   Min.   :    770   Min.   :   802
1st Qu.:768.0   1st Qu.:20679   1st Qu.: 250828   1st Qu.:100092
Median :797.0   Median :21072   Median : 501945   Median :199780
Mean   :797.7   Mean   :20784   Mean   : 501376   Mean   :199675
3rd Qu.:824.0   3rd Qu.:21522   3rd Qu.: 751652   3rd Qu.:299040
Max.   :933.0   Max.   :22731   Max.   :1003581   Max.   :398855
    ACCF_DOR          C_ECUBE           C_IST            C_DOR
Min.   :    1853   Min.   :   131   Min.   :    135   Min.   :    1576
1st Qu.: 2487702   1st Qu.:109492   1st Qu.: 464880   1st Qu.: 4537970
Median : 5118718   Median :220485   Median : 959368   Median : 9356435
Mean   : 5119711   Mean   :220767   Mean   : 959450   Mean   : 9374593
3rd Qu.: 7753980   3rd Qu.:332066   3rd Qu.:1452174   3rd Qu.:14217015
Max.   :10392075   Max.   :443625   Max.   :1948193   Max.   :19072169
 BACKLOG_ECUBE    BACKLOG_IST     BACKLOG_DOR
Min.   : 512   Min.   : 512   Min.   : 512
1st Qu.:1278   1st Qu.:3414   1st Qu.:1567
Median :1300   Median :3453   Median :1584
Mean   :1299   Mean   :3414   Mean   :1576
3rd Qu.:1324   3rd Qu.:3494   3rd Qu.:1603
Max.   :1444   Max.   :3678   Max.   :1682
```

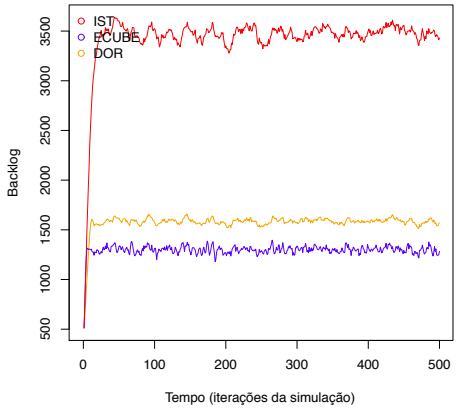
(a)



(b)



(c)



(d)

Figura F.12: Segunda simulação, 1024 vértices, 512 (50%) vértices falhos arbitrários, 512 mensagens injetadas por iteração, sem recuperação; (a) *Análise estatística*; (b) *Gráfico de mensagens afetadas por vértices falhos*; (c) *Gráfico acumulado das mensagens afetadas por vértices falhos*; (d) *Gráfico das mensagens presentes no sistema (backlog)*;