TIAGO APARECIDO TEIXEIRA

HEURÍSTICAS PARA A GERAÇÃO DE ARQUITETURAS RECONFIGURÁVEIS EM ARRANJOS BIDIMENSIONAIS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS – BRASIL
2009

Dedico este título a todos os que me apoiaram e acreditaram que este dia chegaria, em especial, à minha tia Ana Teixeira Matias, pela motivação e, no momento em que eu mais precisei, ela foi minha luz no m do túnel.

Agradeço aos meus pais José Mauricio e Maria Perpetua, pela paciência e dedicação durante todo esse tempo.

E ao meu irmão Leonardo, pelo apoio sempre que necessário.

Cabe aqui uma dedicação especial ao professor Dr. Roberto dos Santos Barbieri, pois só ele sabe o quão especial é esse título para mim, nenhuma outra pessoa, fora da minha família acompanhou tão de perto minha luta para formar a base que me possibilitou chegar até aqui. A ele, meu eterno muito obrigado.

... O Mestre na arte da vida faz pouca distinção entre o seu trabalho e o seu lazer, entre a sua mente e o seu corpo, entre a sua educação e a sua recreação, entre o seu amor e a sua religião. Ele dificilmente sabe distinguir um corpo do outro. Ele simplesmente persegue sua visão de excelência em tudo que faz, deixando para os outros a decisão de saber se está trabalhando ou se divertindo. Ele acha que está sempre fazendo as duas coisas simultaneamente.

(Texto Zen Budista)

Agradecimentos

Ao meu orientador, Professor Dr. Ricardo dos Santos Ferreira, pelo grande apoio e pela dedicação.
Esse título é fruto de um trabalho que começou logo no meu primeiro trabalho de iniciação científica, por isso não poderia deixar de agradecer ao responsável por esse começo, esse título também é seu, professor Msc. Luciano Eugênio de Castro Barbosa.
A João Manuel Paiva Cardoso do Departamento de Informática da Faculdade de Engenharia da Universidade do Porto em Portugal que, mesmo de longe, sempre forneceu toda a ajuda possível contribuindo muito para a qualidade do material que se encontra nesse trabalho.
Ao professor Vladimir Oliveira Di Iorio, cuja base da manipulação de XML utilizando java serviu para o avanço desse trabalho.
Em especial ao professor José Elias Cláudio Arroyo, que, em poucos minutos e uma folha de papel, me forneceu o que eu precisava para a implementação do Simulated Annealing e Path Relinking.
Às pessoas do Departamento de Informática da Universidade Federal de Viçosa que direta ou indiretamente apoiaram meu trabalho, especialmente a Altino Alves, sempre solícito e disposto a resolver os maiores problemas no menor tempo possível.
Ao senhor Generoso Carneiro Neto, que sempre me incentivou, garantindo meu emprego, possibilitando a realização do mestrado.
A José Carlos de Azevedo, que me mostrou o que é ser um verdadeiro líder, apontando sempre o caminho do desenvolvimento profissional.
A todas as pessoas do grupo Apolo.
À Universidade Federal de Viçosa.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Siglas

ADRES *Architecture for Dynamically Reconfigurable Embedded Systems*..8

AG Algoritmo Genético.....4

ALAP O Mais Tarde Possível – *As Late As Possible*.....27

ASAP O Mais Cedo Possível – *As Soon As Possible*.....27

ASIC Circuito Integrado de Aplicação Específica – *Application Specific Integrate Circuit*.....3

CAD Projeto Auxiliado por Computador – *Computer Aided Design*.....10

DSP Encaminhamento de Dados – *DataSource Path*.....11

PE Elemento de Processamento – *Process Element*.....5

FIR Filtro de Resposta a Impulsos Finitos – *Filter Infinit Response*.....13

FPGA *Field Programmable Gate Array*.....3

GPP Processador de Propósito Geral – *General Pourpose Processor*.....3

ID Identificação.....15

ILP Paralelismo a Nível de Instrução – *Instruction Level Paralelism*....1

JWS *Java Web Start*.....76

MUX Multiplexador.....15

PADDI *Programmable Arithmetic Device for DSP*.....11

PR *Path Relinking*.....4

RAM Memória de Acesso Randômico – *Random Access Memory*.....9

RAMP *Research Accelerator for Multiple Processors*.....3

RaPiD *Reconfigurable Pipelined Datapath*.....9

RAW *Reconfigurable Architecture Workstation*.....8

RISC Conjunto Reduzido de Instruções de Computação – *Reduced Instruction Set Computer*.....8

SA *Simulated Annealing*.....31

SoC Sistemas em um *Chip* – *System on Chip*.....10

SRAM Memória de Acesso Randômico Estático – *Static Random Access Memory*.....8

TGFF *Task Graphs for Free*.....26

UEA Unidade de Execução Aritmética.....11

ALU Unidade Lógica Aritmética – *Aritmetical Logic Unit*.....8

VLSI Integração em Escala Muito Grande – *Very Large Scale Integration*.....10

XDS *eXtreme Processing Platform* (XPP) *Development Suite*.....9

XML *eXtensible Markup Language*.....12

XPP *eXtreme Processing Platform*.....xix

Resumo

TEIXEIRA, Tiago Aparecido, M.Sc., Universidade Federal de Viçosa, Abril de 2009. **Heurísticas para a geração de arquiteturas reconfiguráveis em arranjos bidimensionais**. Orientador: Ricardo dos Santos Ferreira. Co-orientadores: Vladimir Oliveira Di Iorio e José Elias Cláudio Arroyo.

Arquiteturas reconfiguráveis de grão grosso são alternativas para a redução do tempo de projeto, a complexidade do posicionamento e roteamento, o tempo de configuração e a memória de configuração para projetos de sistemas embarcados com demanda de alto desempenho e baixo consumo de energia. Porém o espaço de projeto é amplo e necessita de ferramentas flexíveis para sua exploração. Este trabalho propõe uma ferramenta de exploração automática do espaço de projeto das topologias, sendo a abordagem baseada em heurísticas (Algoritmos Genéticos, Simulated Annealing, Path Relinking) e algoritmos de escalonamento (ALAP e ASAP) visando as arquiteturas reconfiguráveis em arranjos com padrões regulares e escaláveis de interconexão. Para validar a ferramenta, um conjunto de aplicações multimídia, derivadas do conjunto de MediaBench e de núcleos de algoritmos para processamento de sinais (FIR, DCT etc.) foi utilizado na avaliação das arquiteturas geradas. Os critérios de custo levaram em conta os números de conexões após o mapeamento, o caminho crítico e o tempo de busca das soluções. Os resultados experimentais mostraram que as arquiteturas geradas podem reduzir em quase 20% o custo de conexões quando comparados à topologia 0_1_hop, apontada por outros trabalhos como a mais adequada.

Abstract

TEIXEIRA, Tiago Aparecido, M.Sc., Universidade Federal de Viçosa, April of 2009. **Heuristics for the generation of reconfigurable architectures in bidimensional arrays**. Adiviser: Ricardo dos Santos Ferreira. Co-Adivisers: Vladimir Oliveira Di Iorio and José Elias Cláudio Arroyo.

Coarse-grained reconfigurable architecture appears as an alternative solution to reduce the design time, the routing and placement complexity, the reconfiguration time, and the reconfiguration memory, to design high performance and low power embedded system. However, the design space is too wide and it needs new explorations tools. This work proposes an tool of the automatic exploration of design space of the topologies, to be the foccus based in heuristics (Genetic Algoritms, Simulated Annealing and Path Relinking) and schedule algorithms (ASAP and ALAP) to see reconfigurable architecture in arrays with regular patterns and scalabre of the interconection. To validate this tool, a set of multimedia applications, from the set of clusters MediaBench and algorithms for signal processing (FIR, DCT etc.) was used in the evaluation of the generated architectures. The criteria for cost take into account the number of connections after the mapping, the critical path and the time to search for solutions. The experimental results showed that the generated architecture can reduce by almost 20% the cost of connections when compared to the topology 0_1_hop, identified by other studies as the most appropriate.

Capítulo 1

Introdução

A demanda por desempenho computacional para sistemas embarcados (celulares, PDAs, etc.) vem crescendo nos últimos anos. Entretanto, hoje existem novas restrições para atender ao mercado, com os dispositivos móveis cada vez mais presentes em nossa vida e em nossa economia.

Seguindo essa tendência mundial, os consumidores de hoje buscam um dispositivo portátil que combine as funcionalidades de um telefone, de uma agenda, do acesso à internet e de uma câmera digital (Mei et al., 2005). Isto requer dos novos dispositivos uma capacidade de processamento que possa combinar as diferentes exigências de desempenho destas funcionalidades com as restrições de energia (devido ao uso de bateria) e a potência (aquecimento).

Além disso, os sistemas devem procurar sempre que possível fazer uso da área disponível em silício, seguindo a lei de Moore (Moore, 1965). Segundo Moore, a densidade de integração dobra a cada 18 meses. Atualmente, devido às complexidades das arquiteturas atuais, o desempenho e a produtividade dos projetos não acompanham mais a lei de Moore. A Figura 1.1, extraída de Austin et al. (2004), mostra a diferença entre a lei de Moore e a performance atual, utilizando para isso as métricas de desempenho do conjunto de *benchmarks* SPECInt2000, que utiliza o Paralelismo a Nível de Instrução – *Instruction Level Paralelism* (ILP), aplicado a geração de processadores Intel. Pode-se notar que a partir do Pentium III, que já incorpora processamento superscalar, o ganho em desempenho deixou de acompanhar a lei de Moore, estando uma ordem de grandeza abaixo, motivando a busca de novas alternativas.

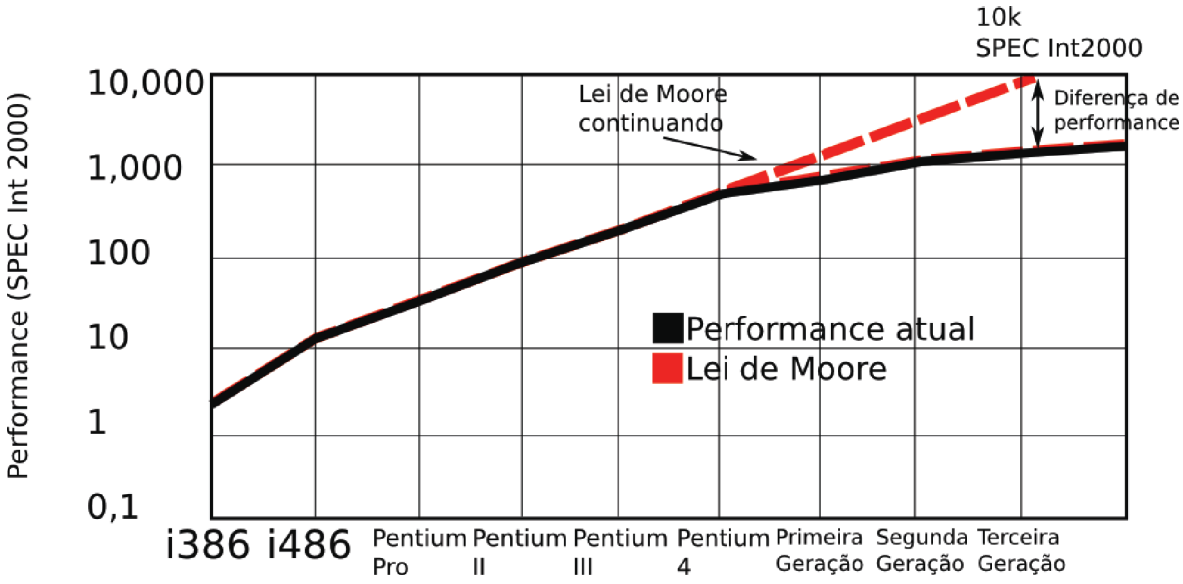


Figura 1.1. Desempenho, extraído de Austin et al. (2004)

Outro fator importante é aliar flexibilidade e reduzir o *time-to-market* para atender a um mercado consumidor cada vez mais dinâmico. Dentro deste cenário, as arquiteturas regulares com múltiplas unidades e com tecnologia reconfigurável, vêm surgindo como uma alternativa promissora (Budiu et al., 2005).

Para garantir a escalabilidade das arquiteturas dos sistemas embarcados, tirando proveito dos avanços tecnológicos, alguns pontos devem ser priorizados, como conexões curtas e limitação de potência (Patterson, 2006). Por exemplo, deve se evitar os fios de conexão global (Budiu et al., 2005), descentralizando o controle e utilizando estruturas regulares para capturar o paralelismo das aplicações.

Com a evolução das tecnologias de silício, e as nanotecnologias, a barreira de um trilhão de transistores está próxima (Robinett et al., 2007). Entretanto, os chips maiores são mais suscetíveis a falhas, o que motiva a pesquisa de novas arquiteturas que possam se adaptar dinamicamente às falhas, possuindo estruturas paralelas e também alguma redundância.

De acordo com o exposto em Patterson (2006), três fatores contribuíram para a redução do ritmo da evolução dos processadores: potência dissipada, o atraso das memórias em relação aos registradores, e a exploração do paralelismo em nível de instrução. Nas décadas de 70/80 a evolução era da ordem de 25% ao ano, depois aumentou para 52% nas décadas 80/90. E atualmente reduziu-se novamente caindo para 20% em 2002 (Patterson, 2006). Se o projetista tem o dobro da área, ao invés de desenvolver um novo processador mais potente, e consequentemente mais complexo,

prefere usar a área para colocar dois processadores da geração atual, simplificando o projeto e tirando proveito da tecnologia, por isso várias empresas passaram a fabricar chips com vários núcleos processadores.

O projeto de um Processador de Propósito Geral – *General Purpose Processor* (GPP) em hardware é uma tarefa complexa, com um longo tempo de projeto e um alto custo. O custo será amortecido se for um sucesso de vendas. Se o projeto for alterado, um alto custo decorrente da fabricação das primeiras unidades é gerado. O GPP oferece muita flexibilidade. Entretanto, a execução de softwares específicos pode não atender as restrições de energia, potência e desempenho, quando comparados aos obtidos por um Circuito Integrado de Aplicação Específica – *Application Specific Integrate Circuit* (ASIC) projetado para a aplicação alvo (Ribeiro, 2002).

Agora se considerarmos os ASICs (Wu & Tsai, 2004), que requerem máscaras específicas para cada projeto, com um alto custo de projeto e um tempo de desenvolvimento longo, para aplicações com uma alta demanda e um grande volume de produção, o alto custo do projeto também é amortizado (Ribeiro, 2002). Porém alterações no projeto ou nas aplicações podem causar um grande prejuízo, e o projeto deve ser refeito. Ou seja, esta solução tem desempenho porém não tem flexibilidade.

Como alternativa para as soluções fixas em hardware, aparecem às soluções em hardware reconfigurável já que o hardware reconfigurável pode ser alterado depois da fabricação, e alterações de projeto podem ser facilmente inseridas sem provocar prejuízos, além de poder ser utilizado para novas aplicações.

Dentro dessa perspectiva as Arquiteturas Reconfiguráveis (Bossuet et al., 2003) vêm surgindo como uma solução atrativa em termos de capacidade, desempenho, baixo consumo de energia e flexibilidade (pelas possibilidades de reconfiguração em tempo de execução e recursos de multigranularidade). Ao contrário dos processadores que possuem alta complexidade de projeto e não conseguem mais acompanhar a lei de Moore (Moore, 1965; DeBenedictis, 2004; Intel, 2008b), os circuitos reconfiguráveis são regulares, como as memórias, e vem dobrando de densidade a cada 18 meses.

Além disso, as indústrias de reconfiguráveis possuem um faturamento da ordem de um bilhão de dólares ao ano (Patterson, 2006). Entretanto faltam ferramentas para projeto e simulação de arquiteturas multinúcleos com um alto volume de unidades.

O grupo de pesquisa *Research Accelerator for Multiple Processors* (RAMP) planeja o desenvolvimento de uma arquitetura com 1024 processadores, com o esforço conjunto do MIT, CMU, Intel e universidades de Berkeley, Stanford e Washington (Wawrzynek et al., 2007).

Arquiteturas reconfiguráveis de grão grosso aparecem como uma solução mais escalável que arquiteturas de grão fino [ex.: *Field Programmable Gate Array* (FPGA)], e

se tornaram cada vez mais importantes nos últimos anos (Hartenstein, 2001b; Bossuet et al., 2003; Bansal et al., 2004; Huang et al., 2004). Esta abordagem reduz o tempo e a complexidade de configuração, bem como problemas de tamanho, posicionamento e roteamento, possibilitando uma otimização do tempo de projeto e síntese. No entanto, muitas arquiteturas parecem ser concebidas sem fortes evidências do porque algumas decisões de projeto foram tomadas.

1.1 Motivação

A busca por arquiteturas mais apropriadas para processamento intensivo de dados baseadas nas arquiteturas de arranjos e na utilização de fluxo de dados no lugar do fluxo de instruções, vem se tornando uma importante opção para se obter desempenho adequado, sem consumo elevado de energia em comparação com a geração atual de processadores.

Ambientes para investigar o espaço de projeto vêm sendo propostos (Hartenstein et al., 2000a), buscando uma melhor exploração das diversas características presentes nessas arquiteturas. Nossa abordagem propõe um ambiente para gerar e avaliar um grande número de padrões locais de interligação.

Estes padrões locais são escaláveis, o que é fundamental para o aproveitamento das vantagens dos avanços das novas tecnologias baseadas em silício. Um dos problemas principais de uma arquitetura paralela é a rede de comunicação entre as diversas unidades. Este trabalho aborda as arquiteturas reconfiguráveis e suas redes de conexões.

Uma característica fundamental a ser estudada será o formato dos padrões de conexão dos elementos de processamento. Atualmente, o atraso das conexões vem sendo dominante em relação ao atraso das unidades de cálculo. Além disso, a complexidade de geração das conexões dificulta a estimação do atraso e da energia consumida nas etapas iniciais de alguns projetos.

1.2 Objetivos

Este trabalho propõe uma ferramenta para geração e exploração de diversos padrões de interconexão local. Nossa abordagem inicial é baseada nas heurísticas Algoritmo Genético (AG) e *Path Relinking* (PR) para gerar novos padrões de conexão a partir de um conjunto inicial, os melhores padrões obtidos servem de entrada para outras heurísticas a fim de se conseguir uma otimização ainda maior.

Para avaliar as arquiteturas reconfiguráveis de grão grosso, algoritmos da área de processamento digital de sinal são usados. Estes algoritmos possuem diversas aplicações

em multimídia, como codificação, compactação, processamento, etc. A ideia central para tais arquiteturas é a distribuição das computações de um algoritmo para várias unidades de processamento, ou *Elemento de Processamento – Process Element (PE)*, de forma que sejam executadas em paralelo.

1.3 Organização do texto

Os próximos capítulos estão organizados de forma: o Capítulo 2 apresenta a revisão bibliográfica e a contextualização das arquiteturas de elementos de alta granularidade, com suas vantagens e desvantagens. O Capítulo 3 descreve o processo de extração da representação em fluxo de dados a partir do código fonte, e o modelo gráfico utilizado durante o mapeamento. Esse capítulo descreve, ainda, cada fase envolvida no mapeamento, assim como, as métricas utilizadas durante a avaliação da melhor solução pelo processo de mapeamento.

No Capítulo 4, é apresentado o gerador de arquiteturas desenvolvido neste trabalho e como as heurísticas avaliadas são utilizadas de forma isolada e em conjunto, possibilitando a obtenção de um melhor resultado heurístico das arquiteturas geradas. A Figura 4.1 mostra um diagrama do gerador. Nesse capítulo são descritas ainda duas metodologias utilizadas para a extração de arquiteturas a partir de um, ou de um conjunto de fluxos de dados, e essas duas metodologias em conjunto ainda podem ser utilizadas para encontrar o caminho crítico do fluxo de dados e mostrá-lo na arquitetura gerada.

Já no Capítulo 5, são apresentados alguns dos principais resultados obtidos com o gerador de arquiteturas, descrito no Capítulo 4, e a análise dos resultados obtidos em cada fase do projeto. E finalmente no Capítulo 6 são apresentadas as principais conclusões e os trabahlos futuros.

Capítulo 2

Arquiteturas Reconfiguráveis

Como visto no capítulo anterior, as soluções baseadas em processadores superescalares estão chegando ao limite. Como uma tentativa para conseguir uma sobrevida maior desse modelo, o programa de pesquisa computacional Intel Tera-Scale (Intel, 2008a) busca técnicas de escalonamento multinúcleos para arquiteturas de 10 a 100 núcleos, em que as instruções serão executadas na escala de teraflops em contraste com os gigaflops das gerações atuais.

As soluções apresentadas tentam modificar o modelo atual, baseado em execuções sequenciais, em um modelo de execução paralela. No entanto, ter um hardware paralelo não significa que as operações executadas pelos softwares atuais passem a ser automaticamente paralelas, justificando trabalhos de pesquisa no projeto, modelagem e mapeamento das aplicações para as arquiteturas paralelas. O universo de pesquisa de arquiteturas paralelas é muito amplo, sendo estudado há décadas. O escopo desta dissertação é o formato dos padrões de conexão dos elementos de processamento para as arquiteturas reconfiguráveis em arranjo. Atualmente, o atraso das conexões vem sendo dominante em relação ao atraso das unidades de cálculo. Além disso, a complexidade de geração das conexões dificulta a estimação do atraso e da energia consumida nas etapas iniciais de alguns projetos. Para tentar achar respostas para alguns desses problemas, propomos um ambiente de geração de arquiteturas, que será detalhado no Capítulo 4. Agora neste capítulo, será apresentado o estado da arte das soluções com arquiteturas reconfiguráveis.

2.1 Arquiteturas Reconfiguráveis de Baixa Granularidade

O primeiro FPGA comercial foi desenvolvido pela Xilinx Inc., surgindo em 1984. Um circuito FPGA é um arranjo de células lógicas associado a uma infraestrutura de interconexões, em que ambas podem ter funcionalidade alterada após a fabricação. Nos FPGA, a célula básica implementa uma função a nível de bit.

Segundo Trimberger (1994); Ribeiro (2002), nos dispositivos com granulosidade baixa há um grande número de blocos lógicos simples. Quando aplicado a operações a nível de palavras de 16 ou 32 bits, essa baixa granularidade acaba gerando uma sobrecarga durante o processo de simulação, síntese, posicionamento das unidades e roteamento das interconexões entre os elementos. Além disso, aumentam o consumo de energia, a complexidade e o tempo de reconfiguração (Nageldinger, 2001).

2.2 Arquiteturas Reconfiguráveis de Alta Granularidade

Arquiteturas com elementos reconfiguráveis de alta granularidade tentam superar as desvantagens das soluções baseadas em FPGA (Nageldinger, 2001), fornecendo uma largura de bits compatível com os datapaths. Ao contrário de FPGAs, o datapath a nível de palavra permite o mapeamento eficiente de operadores em silício, evitando assim a sobrecarga de roteamento em relação à solução por bit.

A lógica de configuração de elementos de alta granularidade, segundo Huang et al. (2004) tem a vantagem de proporcionar uma reconfiguração mais rápida, usando um número menor de bits de configuração e com um tempo de clock menor para a execução da reconfiguração. As arquiteturas reconfiguráveis de alta granularidade têm se mostrado mais adequadas para aplicações de utilização intensiva de dados como as de multimídia e as do domínio das comunicações e telecomunicações (Hartenstein et al., 2000a), enquanto as arquiteturas de baixa granularidade são melhores para computação de nível de bit.

Huang & Malik (2001) propõem o uso de um co-processador de elementos de alta granularidade para reduzir o tamanho do fluxo de bits de configuração.

Segundo Hartenstein (2001a), as arquiteturas reconfiguráveis de alta granularidade podem ser divididas em três grandes grupos: arquiteturas baseadas em malha, arquiteturas baseadas em vetores e arquiteturas baseadas em barramento. Exemplos de cada um desses tipos serão apresentados nas próximas subseções.

2.2.1 Arquiteturas Baseadas em Malha

Nesse tipo de arquitetura, os PEs são organizados de forma retangular como em uma matriz com conexões horizontais e verticais, fornecendo recursos para um paralelismo mais eficiente, pois permite ligações entre elementos adjacentes em até oito direções com uma distância de apenas uma unidade.

Esse tipo de arquitetura foi explorado nesse trabalho. Algumas arquiteturas que seguem esse modelo estão descritas abaixo. Vale ressaltar que a granularidade dos PEs pode variar de simples unidades funcionais (como um multiplicador) até pequenos processadores.

***Reconfigurable Architecture Workstation* (RAW)** Waingold et al. (1997) proveem uma arquitetura de multiprocessadores de Conjunto Reduzido de Instruções de Computação – *Reduced Instruction Set Computer* (RISC) composta por microprocessadores de 32 bits MIPS R2000 modificados, conectados em blocos no formato de uma matriz N x N. Cada processador contém: Unidade Lógica Aritmética – *Aritmetical Logic Unit* (ALU), pipeline de 6 estágios, unidade de ponto flutuante, controladores, 32 registradores de propósito geral e 16 registradores de ponto flutuante, contador de programas, uma memória cache de dados local e 32 Kilobytes de memória de instruções do tipo Memória de Acesso Randômico Estático – *Static Random Access Memory* (SRAM).

Um esquema para explorar as principais características do microprocessador RAW foi proposto por Moritz et al. (1998). Eles revelam uma abordagem utilizando métodos analíticos (sem depender de simulações) e um processo de otimização para sugerir uma boa relação custo-eficácia na utilização da arquitetura RAW para a execução de um conjunto de aplicações. Com base em um espaço fixo para o chip, o esquema explora a granulosidade (área de cada bloco) e o percentual de blocos dedicados à memória, processamento, comunicação e entrada e saída. No entanto, esta abordagem é totalmente adaptada à topologia presente no RAW e não é capaz de explorar diferentes topologias.

KressArray, os trabalhos apresentados por Hartenstein et al. (2000b,a); Nageldinger (2001) foram algumas das poucas exceções a explorar as características arquitetônicas [nesse caso, um número de propriedades do KressArray (Hartenstein et al., 1994)]. O *KressArray Xplorer* é capaz de gerar sugestões de projetos usando um mecanismo de inferência por lógica *fuzzy*. Os autores vêm mostrando sugestões indicando melhorias na base do *KressArray* podendo ser acoplado pela adição de interconexões de vizinhos mais próximos (Nageldinger, 2001).

***Architecture for Dynamically Reconfigurable Embedded Systems* (ADRES)** segundo Mei et al. (2005), é uma abordagem que usa um template

para modelar diferentes matrizes de elementos reconfiguráveis de alta granularidade baseadas em malha. A abordagem inclui um compilador capaz de gerar código para várias plataformas sendo facilmente adaptável e um simulador. Estes usam uma linguagem de descrição de arquiteturas específica para cada instância diferente do ADRES. Os parâmetros que podem ser avaliados são as topologias de comunicação, o conjunto de operações suportadas, os recursos de alocação e a latência na arquitetura alvo. Porém não faz uso de ferramentas automáticas para geração dos parâmetros a serem explorados.

Um outro trabalho que aborda a exploração de topologias para arquiteturas de alta granularidade em arranjos bi-dimensionais foi proposto por Bansal et al. (2004). Este trabalho faz a exploração de alguns modelos de topologia como grid, 0-hop e 1-hop, onde cada PE é conectado aos vizinhos diretos, aos vizinhos diretos e com um salto, aos vizinhos diretos e com um e dois saltos, respectivamente. O trabalho avalia também a arquitetura com barramento em linha e coluna, mas é restrito a arranjos 4x4 e 8x8, e não faz análise de arranjos maiores. Outro foco do trabalho é a política de varredura do arranjo no momento do posicionamento e escalonamento da aplicação na arquitetura. Os resultados avaliam que uma arquitetura com duas ligações na direção linha e coluna são satisfatórias para os casos avaliados.

Algumas arquiteturas comerciais de grão grosso são geradas por ferramentas de simulação proprietárias como o *Development Suite* (XDS) da PACT (Baumgarte et al., 2003). O usuário pode especificar, antes de fabricar, o número de barramentos, o tamanho das palavras, o número de células etc. Porém, esse tipo de abordagem é dependente da arquitetura.

2.2.2 Arquiteturas Baseadas em Vetores

Algumas arquiteturas baseadas em uma distribuição linear das unidades e uma estruturada bi-dimensional para conexão visam ao mapeamento de segmentos internos. Se os segmentos têm bifurcações, podem ser necessários recursos adicionais para o roteamento, como mais linhas abrangendo a totalidade ou uma parte da matriz. A arquitetura *Reconfigurable Pipelined Datapath* (RaPiD) (Ebeling et al., 1996) fornece diferentes recursos computacionais, como ALUs, memórias do tipo Memória de Acesso Randômico – *Random Access Memory* (RAM), multiplicadores e registradores, mas, irregularmente, distribuídos.

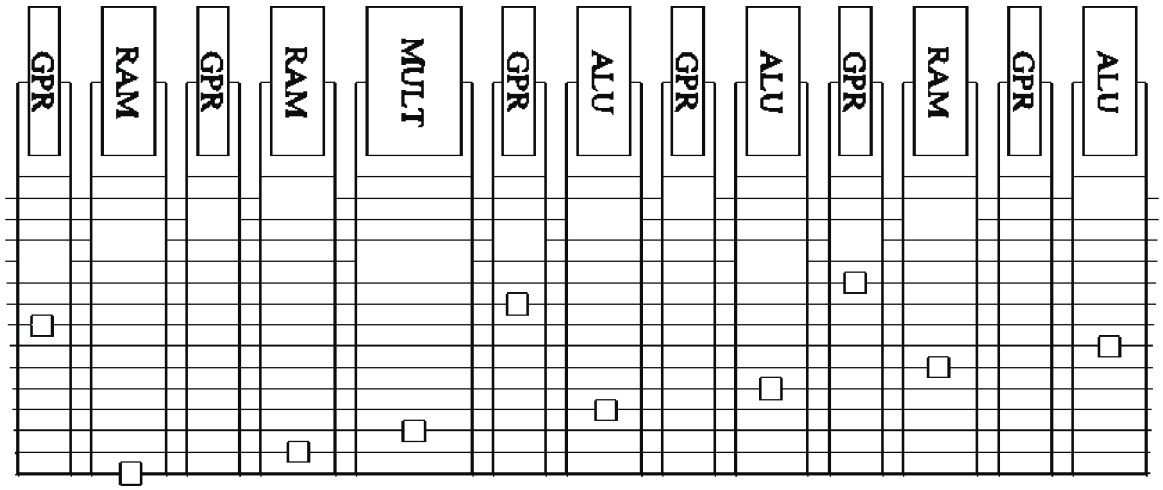


Figura 2.1. RaPiD, extraído de Cronquist et al. (1999).

A arquitetura do ***Reconfigurable Pipelined Datapath***, segundo Ebeling et al. (1996); Cronquist et al. (1999) é composta por elementos computacionais de alta granulosidade do tamanho de palavras, como ALU, multiplicadores, e células de memória organizadas ao longo de um eixo unidimensional. O roteamento entre os elementos é feito por um barramento com largura igual ao tamanho de uma palavra, a Figura 2.1 representa o esquema de distribuição dos elementos na arquitetura RaPiD.

O projeto **Totem** (Hauck et al., 2008), semelhante à abordagem dessa dissertação, propõe um gerador automático de arquiteturas. Porém o **Totem** é específico para a arquitetura RaPiD baseada em um vetor de unidades, enquanto esta dissertação aborda as topologias para um modelo em malha ou bi-dimensional. O projeto **Totem** visa ainda proporcionar um caminho completamente automático para a criação de hardware reconfigurável customizado, orientado para o uso em Sistemas em um Chip – *System on Chip* (SoC). Para tal o sistema é dividido em três partes: (a) gerador de arquiteturas, que gera a descrição utilizada para o bloco lógico e para as interconexões programáveis; (b) Integração em Escala Muito Grande – *Very Large Scale Integration* (VLSI) layout generator que traduz a descrição da arquitetura para transistores e máscaras de *layout*; e (c) ferramenta de geração de posicionamento e roteamento que cria a suíte de Projeto Auxiliado por Computador – *Computer Aided Design* (CAD) para a arquitetura. O fluxo do sistema Totem pode ser visualizado na Figura 2.2.

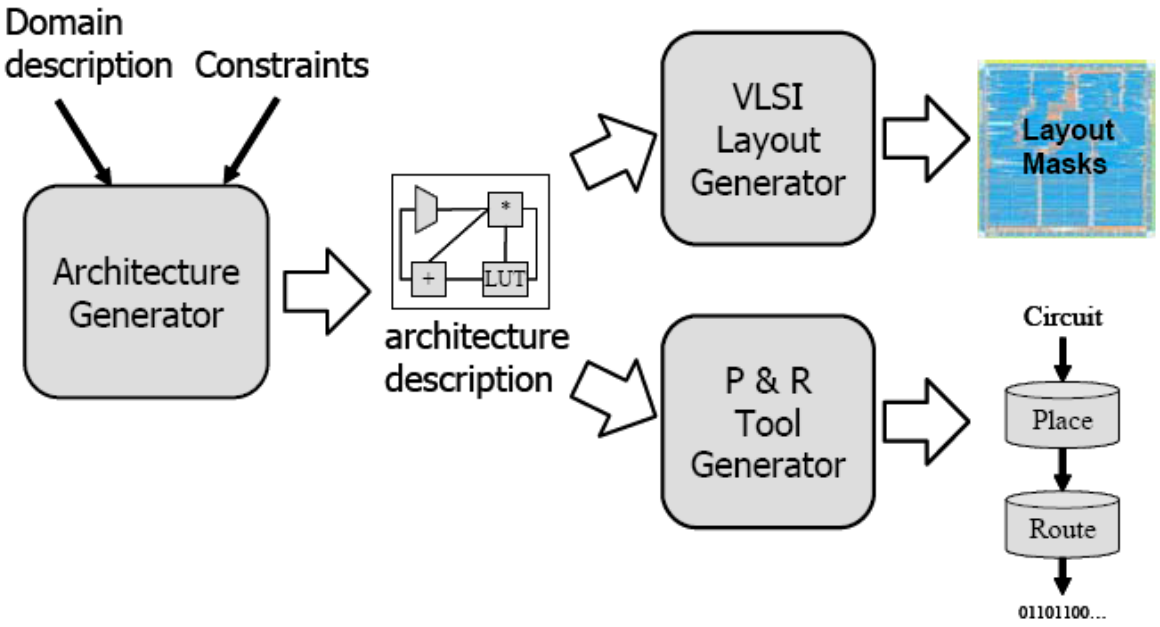


Figura 2.2. Totem, fluxo do sistema (Hauck et al., 2008).

2.2.3 Arquiteturas Baseadas em Barramentos

Um barramento de troca completo é a rede de comunicação mais poderosa e mais fácil de rotear. Porém as arquiteturas dessa categoria costumam usar apenas um barramento reduzido. ***Programmable Arithmetic Device for DSP* (PADDI)** (Chen & Rabaey, 1990, 1992) foi desenvolvido para a prototipagem rápida de funcionalidades de Encaminhamento de Dados – *DataSource Path* (DSP) com oito PEs, todos ligados por um barramento multicamadas. **PADDI-2** tem 48 PEs (Yeung & Rabaey, 1993), mas reserva uma área para um barramento restrito com uma estrutura hierárquica de interligação de vetores de PEs formando "clusters". Esta fabricação sofisticada tem um impacto sobre os novos roteamentos.

A **Arquitetura Pleiades** (Rabaey, 1997) é uma generalização de baixa energia do "PADDI-3" com microprocessadores programáveis e arquiteturas de Unidade de Execução Aritmética (UEA) heterogêneas, que permitem a integração entre UEA de granularidade baixa e alta e memórias no lugar das UEAs.

Capítulo 3

Mapeamento

O universo de arquiteturas reconfiguráveis é muito grande. Neste capítulo, iremos apresentar o processo de mapeamento de uma aplicação em uma arquitetura. As arquiteturas seguem o modelo em malha. O processo de mapeamento escolhido foi proposto por Ferreira et al. (2007), que é genérico e pode ser adaptado para outros modelos. Como tanto a aplicação quanto a arquitetura são modeladas por grafos, esta abordagem de mapeamento permite criar uma estrutura de geração dos grafos na busca automática de uma arquitetura mais adequada para um conjunto de aplicações.

Inicialmente, apresentamos como o grafo de fluxo de dados representa o núcleo extraído do código de uma aplicação. Mostramos também como um arquitetura bi-dimensional pode ser modelada de forma a automatizar o processo de geração de novas arquiteturas a partir de modificações em um conjunto inicial de arquiteturas. O processo de geração, foco principal desse trabalho, é abordado no próximo capítulo. Finalmente, iremos apresentar o processo de mapeamento utilizado por este trabalho, o qual nos permite avaliar se uma aplicação será executada de forma eficiente em uma arquitetura.

Vale ressaltar que o escopo de geração de arquiteturas é amplo, e este trabalho aborda a geração de topologias em duas dimensões para arquiteturas reconfiguráveis, em que os elementos de processamento implementam funções de fluxo de dados.

3.1 Grafo de Fluxo de Dados

Para facilitar a portabilidade da descrição da aplicação que queremos implementar, o grafo de fluxo de dados que representa as aplicações faz uso de um padrão desenvolvido por Ferreira et al. (2004) utilizando *eXtensible Markup Language* (XML). Como um exemplo da geração de um grafo de fluxo de dados, apresentaremos cada fase necessária para a transformação da parte passível de execução em paralelo do núcleo da aplicação

de um Filtro de Resposta a Impulsos Finitos – *Filter Infint Response* (FIR) de ordem 4.

A primeira fase que consideramos foi o modelo matemático que indica o paralelismo intrínseco de parte da execução do núcleo do Algoritmo da aplicação FIR. Esse modelo matemático pode ser visualizado nas equações 3.1 a 3.4, a seguir.

$$y0 = x0 * h0 + x1 * h1 + x2 * h2 + x3 * h3 \tag{3.1}$$

$$y1 = x1 * h0 + x2 * h1 + x3 * h2 + x4 * h3 \tag{3.2}$$

$$y2 = x2 * h0 + x3 * h1 + x4 * h2 + x5 * h3 \tag{3.3}$$

$$y3 = x3 * h0 + x4 * h1 + x5 * h2 + x6 * h3 \tag{3.4}$$

Equações para o FIR 4

O modelo matemático traduzido em forma de Algoritmo codificado na linguagem C pode ser visualizado no Algoritmo 3.1.

Algoritmo 3.1: FIR 4

```
1 /* TEXAS INSTRUMENTS, INC.
2 * USAGE This routine is C Callable and can be called as:
3 * void fir(short *x, short *h, short *y, int N, int M)
4 * x = input array
5 * h = coefficient array
6 * y = output array
7 * N = number of coefficients (MULTIPLE of 4 >= 4)
8 * M = number of output samples (EVEN >= 2)
9 */
10 void fir(short x[], short h[], short y[], int N, int M) {
11     int i, j, sum;
12     for (j = 0; j < M; j++) {
13         sum = 0;
14         for (i = 0; i < N; i++)
15             sum += x[i + j] * h[i];
16         y[j] = sum » 15;
17     }
18 }
```

A partir do Algoritmo 3.1, é possível a extração do modelo de fluxo de dados. Este trabalho pode ser realizado por um compilador. Não é escopo dessa dissertação

a geração dos grafos de fluxo de dados. Iremos utilizar um conjunto de aplicações já mapeado na forma de grafos de fluxo de dados. A Figura 3.1 ilustra o grafo gerado para a aplicação do FIR4. Nosso objetivo é gerar uma arquitetura que seja adequada para um conjunto de aplicações, tendo como ponto de partida a descrição das aplicações na forma de grafo de fluxo de dados.

A vantagem do modelo em fluxo de dados é o paralelismo explícito e o funcionamento em pipeline. Cada elemento do vetor é aplicado à entrada, e a cada ciclo um novo valor de y é calculado. Para este exemplo, todas as operações são implementadas em paralelo, eliminando a sobrecarga de controle do loop da versão codificada em C no Algoritmo 3.1.

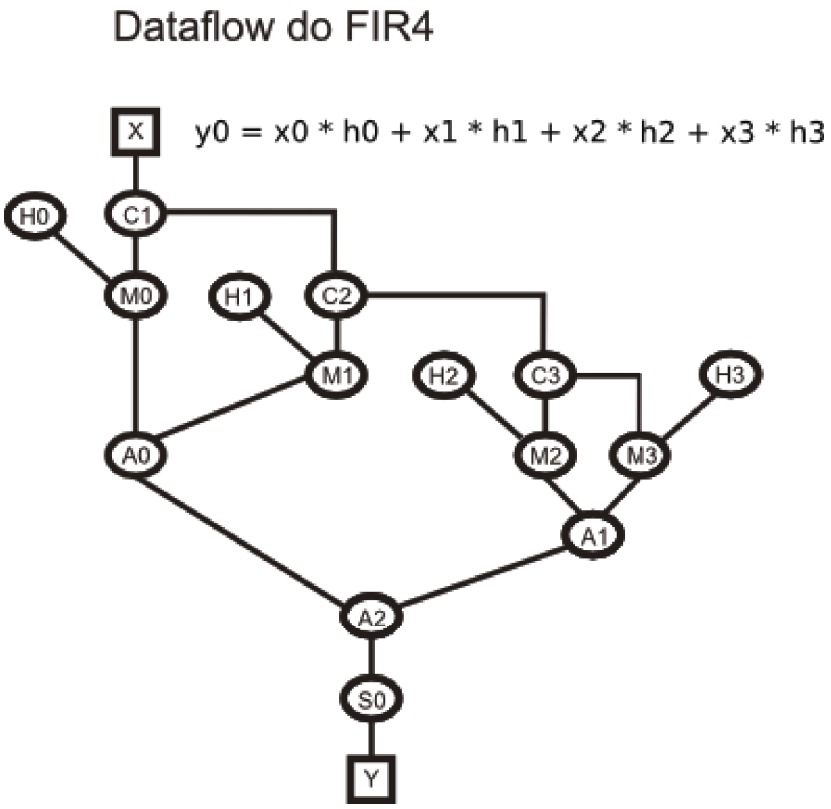


Figura 3.1. Grafo FIR de ordem 4.

3.2 Definição da Arquitetura

Para este trabalho, consideramos apenas matrizes de arquiteturas 2D. No entanto, o nosso modelo de arquitetura é um grafo e pode ser utilizado na representação de arquiteturas n-dimensionais.

Cada arquitetura é composta por um conjunto de PEs, por exemplo, ALU, Multiplexador (MUX), contadores, células de memória etc. e uma interligação de rede. A interligação de rede é composta por um conjunto de conexões diretas entre dois PEs. Nosso objetivo é encontrar a melhor forma de interligar os PEs para um dado conjunto de aplicações. Vamos considerar que cada PE dispõe de um número exclusivo de Identificação (ID). Este número poderia ser um endereço em uma matriz como linha e coluna, ou i, j, sendo i = linha e j = coluna. Por exemplo, um $PE_{i,j}$, tem uma ligação direta a um $PE_{i+1,j}$ em uma rede de interconexão em grade. A ligação pode ser uma função de i, j e diversas arquiteturas podem ser representadas. As arquiteturas de alta granularidade mais utilizadas são as compostas por padrões de interconexão local tais como aquelas em grade, hexagonal, octal e n_hop.

Neste trabalho, buscamos explorar o espaço de solução de diferentes padrões de conexão. Para ampliar a busca, pode-se gerar uma arquitetura aleatória, pela geração aleatória de conexões locais entre cada PE. Outro espaço que pode ser explorado é o conjunto das arquiteturas baseadas em operações de bits, que são criadas usando padrões locais agregados a padrões baseados em bits tais como: *perfect shuffler, inverse perfect shuffler, bit reverse, butterfly, baseline, cube* etc. Tal agregação ocorre ora utilizando-se um padrão local em linha com um padrão baseado em bit em coluna, ora usando um padrão baseado em bit em linha e um padrão local em coluna.

Os padrões de conexão foram divididos em dois grupos: os padrões de saltos e os padrões de inversão de bit. Para o mapeamento desses grupos em arquivos XML foi seguido o seguinte script:

1. O arquivo XML deve fornecer um meio escalável e dinâmico para que as definições dos padrões pudessem evoluir com o tempo.

2. A definição do arquivo XML deve fornecer um meio para identificar a qual grupo pertence o padrão inserido, para poder haver um controle posterior.

3. O arquivo XML deve possuir apenas ligações de destino, pois as ligações de origem serão sempre a dupla $PE_{\{i,j\}}$, ou seja $PE_{\{linha,coluna\}}$.

4. O Algoritmo de leitura do arquivo XML deve estar preparado para tal evolução, sendo que para isso têm que ser robusto o suficiente para aceitar qualquer formato de ligação e qualquer quantidade de conexões por elementos, ao mesmo tempo em que não permita violações ao XML-schema definido.

5. Cada elemento pode possuir um padrão diferente, tanto em cada uma das conexões, quanto em uma mesma conexão, ou seja, um padrão de saltos na linha

da conexão 1 e um padrão de inversão de bit na coluna da conexão 1, e/ou um padrão de saltos na linha e na coluna da conexão 2 etc.

Para um padrão de saltos igual a 0_n_hop com 8 conexões, por exemplo, a origem será sempre $PE_{i,j}$ e os destinos serão: primeira conexão $PE_{i,j+1}$, segunda conexão $PE_{i+1,j}$, terceira conexão $PE_{i,j-1}$, quarta conexão $PE_{i-1,j}$, quinta conexão $PE_{i,j+1+n}$, sexta conexão $PE_{i+1+n,j}$, sétima conexão $PE_{i,j-1-n}$ e oitava conexão $PE_{i-1-n,j}$. Como cada conexão é composta por uma dupla linha, coluna, o elemento XML para esse exemplo seria o presente no Algoritmo 3.2. Já a Figura 3.2 representa graficamente essa descrição para uma arquitetura 0_3_hop. Ou seja, a descrição de uma arquitetura em XML é feita pelo conjunto vizinhos de um $PE_{i,j}$.

Algoritmo 3.2: Elemento em XML

```
<element id="0">
    <link id="0">
        <line type="HOP" family="Base">
            <hop>0</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>1</hop>
        </colun>
    </link>
    <link id="1">
        <line type="HOP" family="Base">
            <hop>1</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>0</hop>
        </colun>
    </link>
    <link id="2">
        <line type="HOP" family="Base">
            <hop>0</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>-1</hop>
        </colun>
    </link>
    <link id="3">
        <line type="HOP" family="Base">
            <hop>-1</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>0</hop>
        </colun>
    </link>
    <link id="4">
        <line type="HOP" family="Base">
            <hop>0</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>1+N</hop>
        </colun>
    </link>
    <link id="5">
        <line type="HOP" family="Base">
            <hop>1+N</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>0</hop>
        </colun>
    </link>
    <link id="6">
        <line type="HOP" family="Base">
            <hop>0</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>-1-N</hop>
        </colun>
    </link>
    <link id="7">
        <line type="HOP" family="Base">
            <hop>-1-N</hop>
        </line>
        <colun type="HOP" family="Base">
            <hop>0</hop>
        </colun>
    </link>
</element>
```

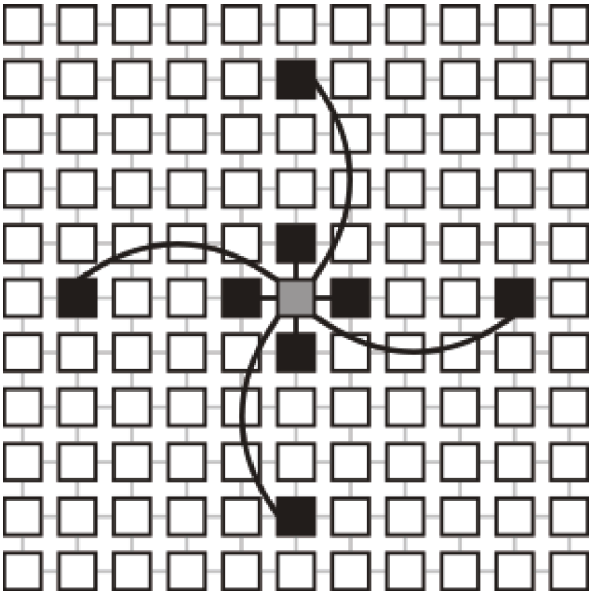


Figura 3.2. Saltos 0_3_hop.

Com base no XML-schema definido e com o exemplo da Figura 3.2, podemos concluir que se pode usar qualquer número positivo ou negativo inteiro para representar o número de saltos entre as conexões.

Já para um padrão de inversão de bit, por exemplo, da aplicação do padrão *perfect shuffler* em linha, pode ser gerado uma conexão local para o $PE_{i,j}$, utilizando uma rotação à esquerda de um bit no endereço fonte para obter o destino na linha. Vamos considerar o $PE_{4,0}$, ou seja, o PE localizado na linha quatro e coluna zero, em binário $PE_{100,000}$. Ao aplicar a inversão de bit *perfect shuffler* teriamos a ligação ao $PE_{1,0}$, em binário $PE_{001,000}$, sugerindo a ligação $PE_{4,0} \rightarrow PE_{1,0}$, mas como aplicamos a inversão somente em linha, o sugerido por esse trabalho seria aplicar o padrão de saltos em coluna. Caso aplicássemos o padrão octal, teríamos a ligação $PE_{4,0} \rightarrow PE_{1,1}$. A representação gráfica desse exemplo está na Figura 3.3a. Na Figura 3.3b, é possível a verificação do padrão de inversão de bit em conjunto com o padrão de saltos.

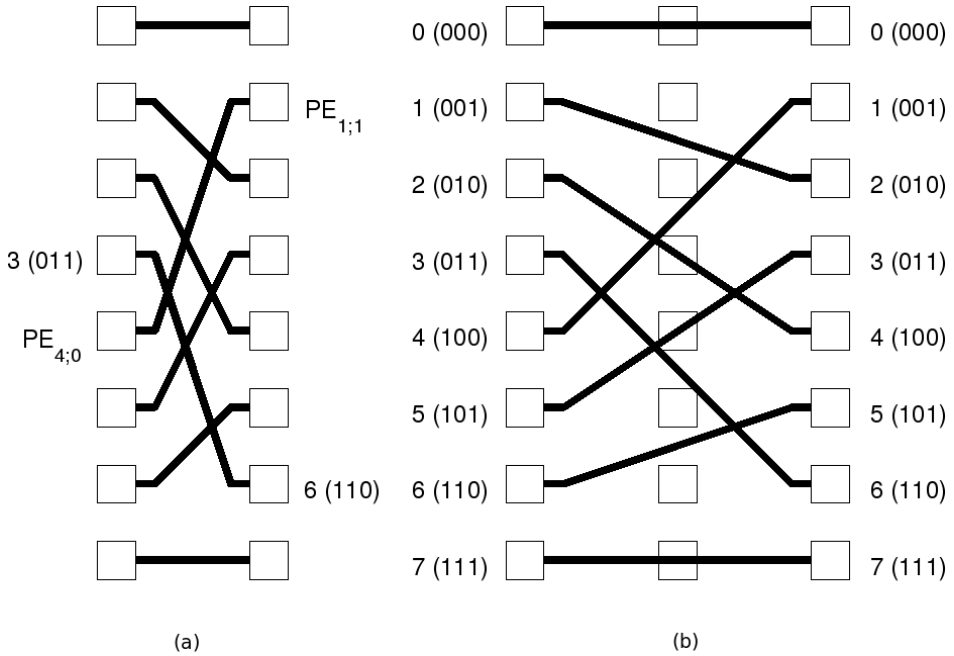


Figura 3.3. Exemplo de inversão de bits com os exemplos das nomenclaturas em decimal, em binário e em relação ao posicionamento do PE no arranjo bidimensional.

3.3 Abordagens para o Problema de Mapeamento

O mapeamento normalmente é dividido em duas fases: posicionamento e roteamento. Como o posicionamento e o roteamento são problemas NP-Completos bem conhecidos (Tessier & Burleson, 2001), há um grande espaço para permutação, por exemplo, para a matriz N x M, o espaço de solução é $(NM)!$. Com isso, até mesmo para pequenos arranjos, é necessária uma exaustiva pesquisa tornando-a proibitiva.

As próximas seções apresentarão a solução adotada para o mapeamento (Ferreira et al., 2007) do grafo de fluxo de dados nas arquiteturas estudadas e usarão como exemplo o posicionamento e o roteamento do FIR 4 em duas arquiteturas distintas. O fluxo de dados do FIR 4 e as duas arquiteturas podem ser observados na Figura 3.4.

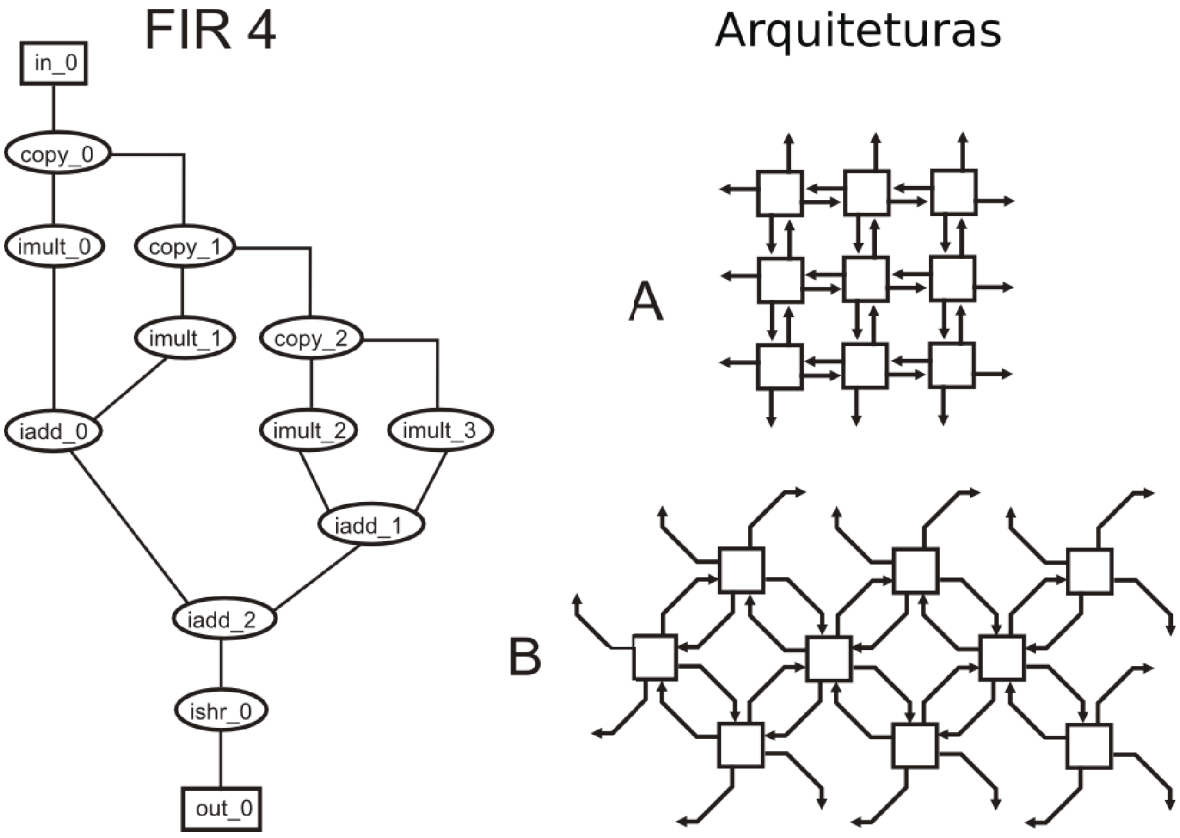


Figura 3.4. Fluxo de dados do FIR 4 e arquiteturas.

3.3.1 Posicionamento

O posicionamento tem um impacto significativo sobre a próxima fase, o roteamento, (Nageldinger, 2001). Assim, este processo é conduzido por uma função objetivo que normalmente reflete a rotabilidade do posicionamento. O posicionamento adotado é baseado em uma busca em profundidade no grafo de fluxo de dados e da arquitetura, e foi proposto por Ferreira et al. (2007). Além da flexibilidade para representar a arquitetura, esta implementação tem um custo polinomial que a permite ser executada várias vezes durante a avaliação de diversas arquiteturas. A Figura 3.5 ilustra o posicionamento em profundidade para o FIR4 em uma arquitetura em malha. A cada passo, um caminho em profundidade é mapeado. Maiores detalhes do algoritmo estão descritos por Ferreira et al. (2007).

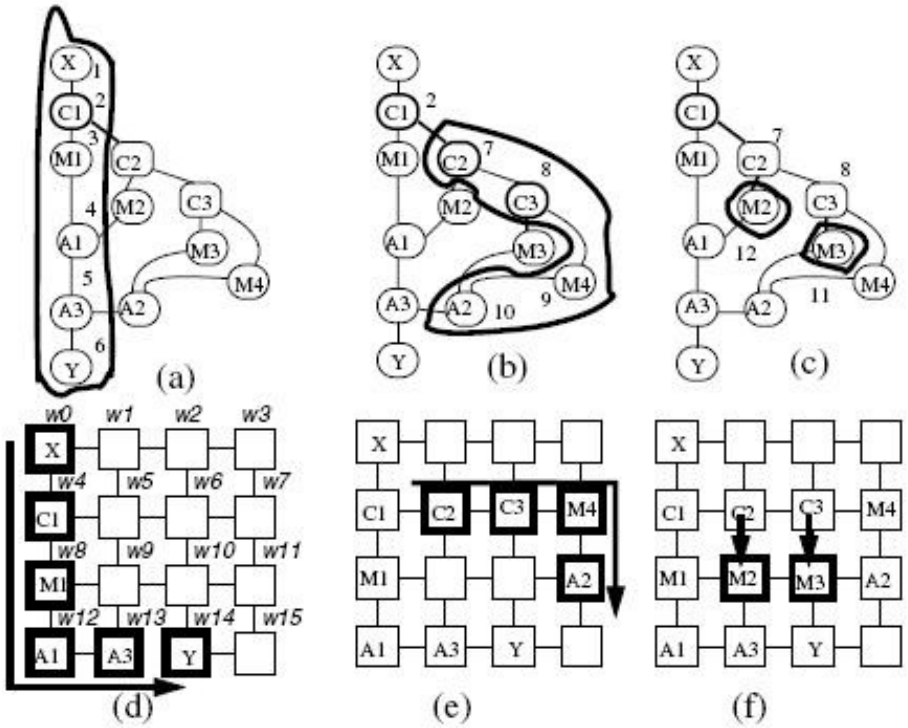


Figura 3.5. Gráfico do fluxo de dados e posicionamento: (a), (d) Primeiro caminho; (b), (e) Segundo caminho; (c), (f) Últimos dois caminhos Ferreira et al. (2007).

Um exemplo envolvendo o posicionamento do grafo de fluxo de dados do FIR 4 demonstradona Figura 3.1 pode ser observado na Figura 3.6 em que o mesmo grafo foi posicionado em duas arquiteturas distintas.

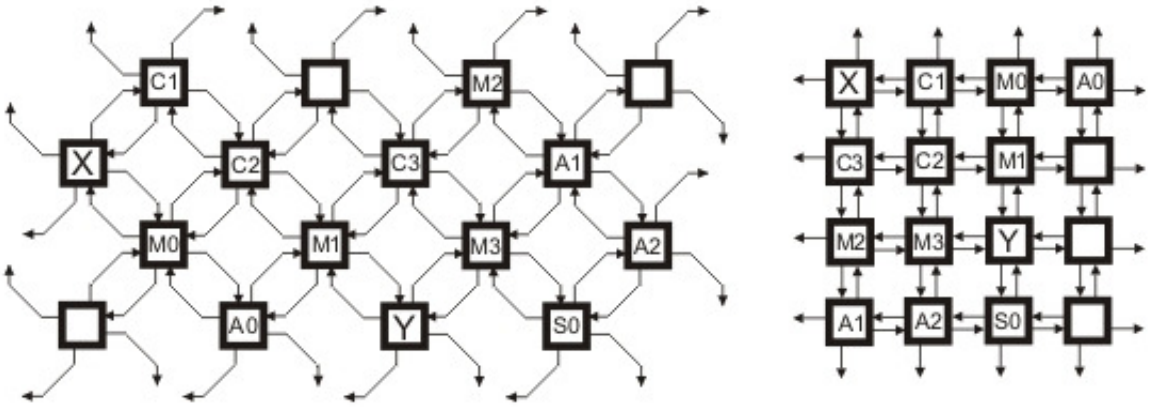


Figura 3.6. Representando o posicionamento do fluxo de dados do FIR 4 em duas arquiteturas distintas.

3.3.2 Roteamento

Uma vez que o posicionamento das células em uma arquitetura reconfigurável é feito, o que a fase de roteamento determina é quais dos interruptores reconfiguráveis deverão ser ativados para formar as ligações entre as células.

Um dos algorimos mais usados é o Algoritmo PathFinder (McMurchie & Ebeling, 1995). Esse algoritmo faz um balanceamento entre o desempenho e a capacidade de roteamento e tenta chegar na solução ótima que atenda aos dois quesitos. Uma estratégia de negociação, em que sinais mais críticos têm prioridade, é utilizada para reduzir o atraso. Após o posicionamento em profundidade descrito na seção anterior, o roteamento com PathFinder é aplicado para finalizar o mapeamento.

Um exemplo envolvendo o roteamento do FIR 4 demonstrado na Figura 3.1, pode ser observado na Figura 3.7 em que o mesmo foi roteado em duas arquiteturas distintas, de acordo com as possibilidades de roteamento de cada arquitetura. Nota-se nesse exemplo que, dependendo da forma como os elementos fossem posicionados, o roteamento poderia ter sido facilitado ou impossibilitado, uma vez que, se todos os multiplicadores, por exemplo, tivessem ficado em um mesmo lado de uma ou outra arquitetura, não haveria como rotear todas as ligações entre os elementos.

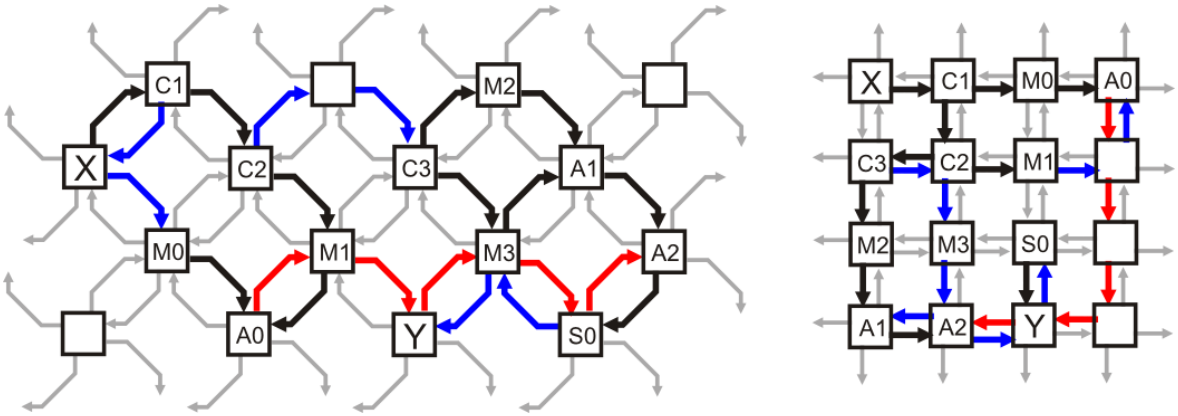


Figura 3.7. Representando o roteamento do fluxo de dados do FIR 4 em duas arquiteturas distintas.

3.3.3 Métricas de Avaliação

Como métrica principal para o mapeamento, utilizou-se o custo de interconexão. O cálculo do custo pode ser feito de formas distintas: uma baseada no número médio

de segmentos utilizados nas conexões entre os elementos e a outra baseada no número total dos segmentos utilizados, que leva em conta o tamanho dos grafos de fluxo de dados.

Para o cálculo da primeira opção, é utilizada a Fórmula 3.5. Com essa fórmula, para cada grafo de fluxo de dados mapeado em uma arquitetura tem-se o número médio dos segmentos necessários para sua implementação em hardware. Para o grafo da Figura 3.4, depois de mapeado na arquitetura B da Figura 3.4, suas ligações foram satisfeitas utilizando-se $\frac{(11*1)+(3*2)+(1*5)}{22} = 1,4666$, em que 11 ligações gastaram 1 segmento, 3 ligações gastaram 2 segmentos e 1 ligação gastou 5 segmentos. A soma do número de segmentos dividida pelo número de ligações, dá uma média de 1,4666 segmentos utilizados na arquitetura para mapear todas as ligações do fluxo de dados.

$$comprimento\ médio\ dos\ fios\ = \frac{\sum(ligacões\ *\ segmentos)}{total\ ligacões} \tag{3.5}$$

Fórmula Média

Já para o cálculo da segunda opção, é utilizada a fórmula 3.6 , ou seja, é considerado o total de segmentos utilizados para mapear um fluxo de dados em uma arquitetura. Para o exemplo anterior, o valor seria 22, significando que foram necessários 22 segmentos na arquitetura para o mapeamento de 15 ligações do grafo de fluxo de dados.

$$comprimento\ total\ dos\ fios\ = \sum(ligacões\ *\ segmentos) \tag{3.6}$$

Fórmula Total

Capítulo 4

Geração Automática de Arquiteturas

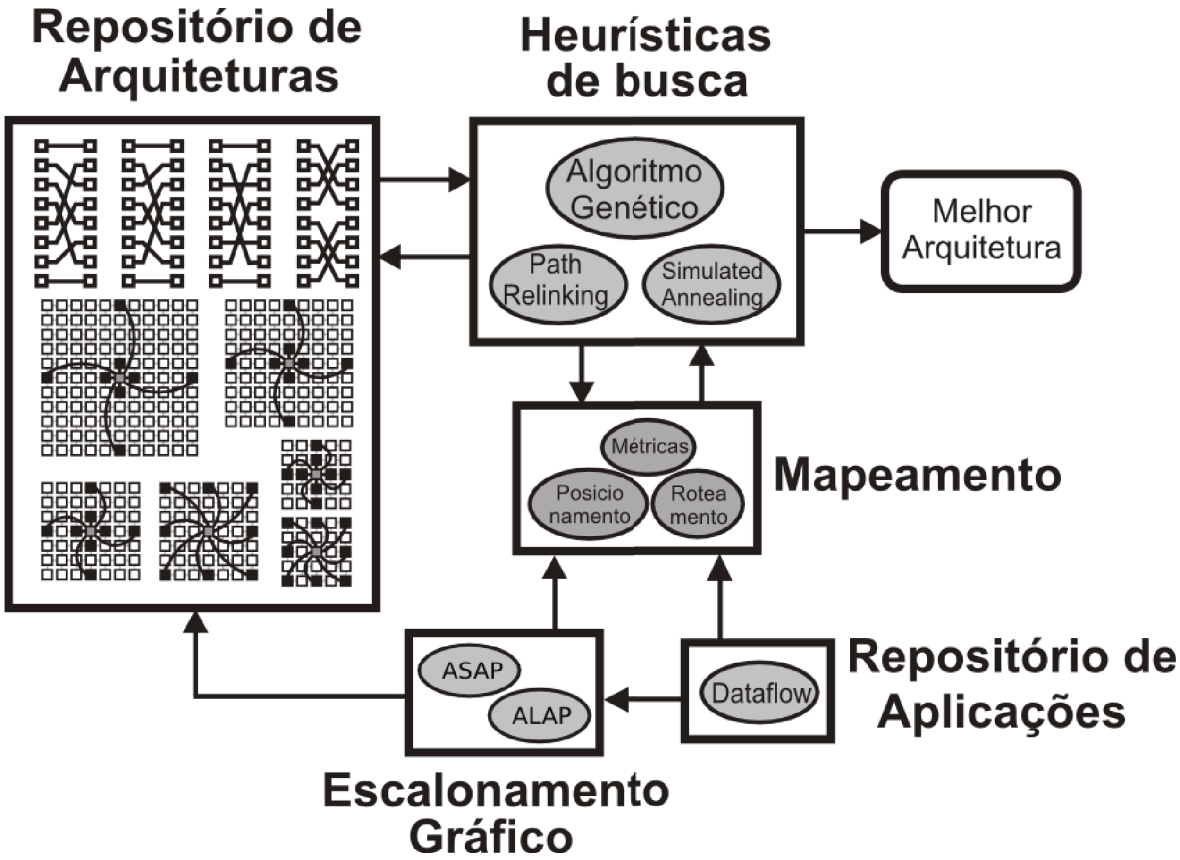


Figura 4.1. Gerador de Arquiteturas

O foco principal deste trabalho é o desenvolvimento de um gerador de arquiteturas para explorar o espaço de projeto das arquiteturas de grão grosso, gerando automaticamente uma arquitetura adequada para um conjunto de aplicações. O módulo de

heurísticas de busca, ilustrado na Figura 4.1, interage com um conjunto de módulos: Compilador, Escalonamento Gráfico, Repositório de Arquiteturas, e Mapeamento.

Nossa abordagem é flexível, suportando vários fluxos de geração. Neste trabalho, apresentamos dois fluxos: (1) baseado em geração de arquiteturas, utilizando heurísticas de busca e (2) baseado em extração de arquiteturas a partir de aplicações, utilizando algoritmos de escalonamento gráfico. Inicialmente, as aplicações são compiladas, e o núcleo principal que domina o tempo de execução é extraído. Nosso objetivo é encontrar uma arquitetura com um conjunto de segmentos eficiente para executar os núcleos das aplicações.

O primeiro fluxo, baseado na geração de arquiteturas, pode ser visualizado na Figura 4.2 e seus passos estão descritos a seguir:

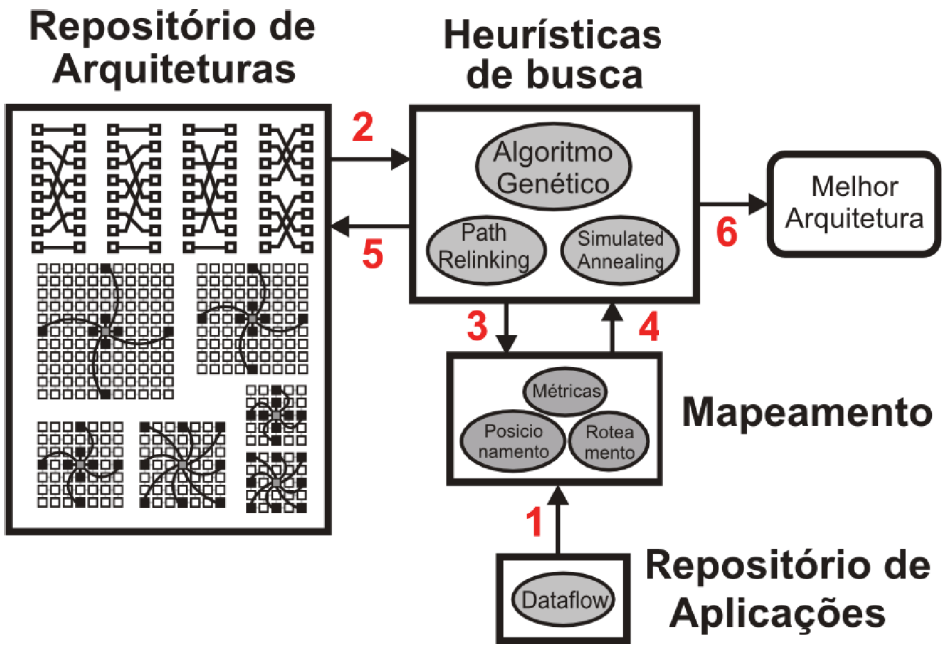


Figura 4.2. Representação do primeiro ciclo analisado.

1. Em uma entrada, o módulo de mapeamento seleciona um grafo de fluxo de dados de uma aplicação, tendo o processo de geração de um grafo sido detalhado na Seção 3.1. Para cada arquitetura, todas as aplicações do repositório são mapeadas e avaliadas.

2. Em outra entrada, o módulo de heurísticas de busca seleciona uma arquitetura do repositório, a estrutura da arquitetura foi definida na Seção 3.2.

3. A arquitetura selecionada, ou melhorada, pela heurística é entregue ao módulo de mapeamento.

4. O módulo de mapeamento aplica o posicionamento e o roteamento gerando um valor de acordo com a métrica selecionada pelo módulo de geração, retornando esse valor para o módulo de heurísticas de busca.

5. As melhores arquiteturas retornam ao repositório, e o ciclo do passo 3 até o 5 repete-se até que um ponto de parada seja atingido.

6. A melhor arquitetura é identificada e armazenada.

O segundo fluxo, baseado na extração de arquiteturas a partir de aplicações, pode ser visualizado na Figura 4.3, e os seus passos estão descritos a seguir:

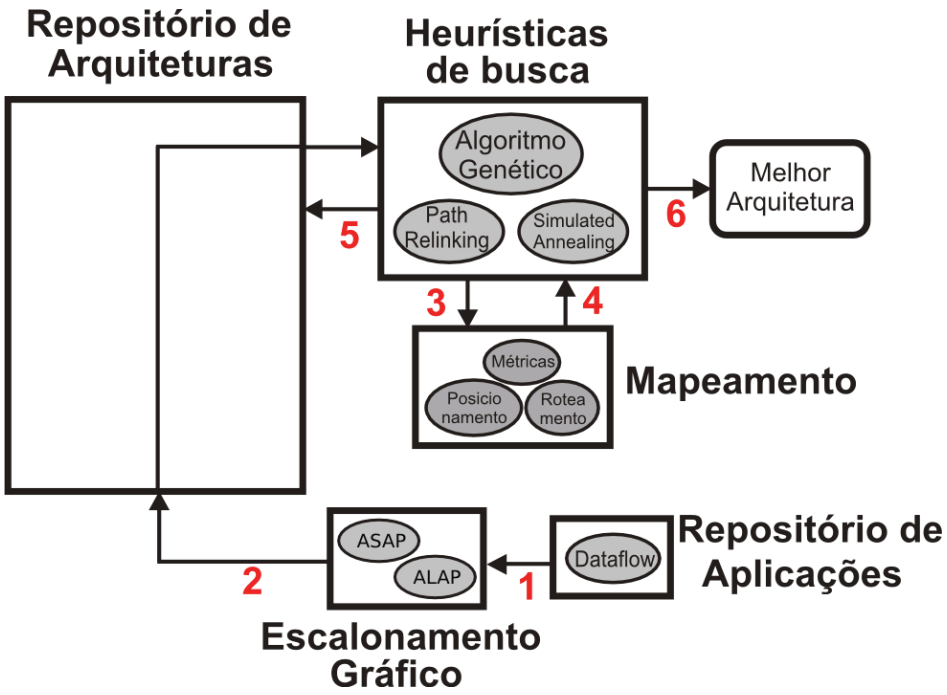


Figura 4.3. Representação do segundo ciclo analisado.

1. Todas as aplicações são repassadas para o módulo de escalonamento gráfico.

2. Uma arquitetura é gerada utilizando informações extraídas pelos algoritmos de escalonamento gráfico e utilizando a estrutura definida na Seção 3.2.

3. A arquitetura selecionada, ou melhorada, pela heurística é entregue ao módulo de mapeamento.

Tabela 4.1. Composição dos fluxos de dados avaliados

Fonte	Vértices	Arcos
Manual para processamento de sinais	6 a 249	6 a 295
Aleatório	64 a 124	89 a 169
MediaBench	11 a 333	8 a 354

4. O módulo de mapeamento aplica o posicionamento e o roteamento gerando um valor de acordo com a métrica selecionada pelo módulo de heurísticas de busca, retornando esse valor para o módulo de heurísticas de busca.

5. As melhores arquiteturas retornam ao repositório e o ciclo do passo 3 até o 5 repete-se até que um ponto de parada seja atingido.

6. A melhor arquitetura é identificada e armazenada

4.1 Repositório de Aplicações

Como já mencionado anteriormente, esse trabalho não visa a geração dos grafos de fluxo dedados a partir de seus respectivos códigos fonte. Para se ter aplicações reais sendo avaliadas, este trabalho optou por três origens distintas, sendo:

1. Extração manual a partir de algoritmos da área de processamento digital de sinais feita por Ferreira et al. (2004) e a partir do microprocessador MIPS realizada por Guerra (2008).

2. Geração aleatória de fluxos de dados utilizando o software *Task Graphs for Free* (TGFF) desenvolvido por Dick et al. (1998), sendo os parâmetros utilizados: 2 conexões de entrada e 2 conexões de saída por elemento e os vértices folhas unidos em pares até restar apenas um único vértice folha.

3. Fluxos de dados de *benchmarks* selecionados pelo aplicativo *MediaBench* disponibilizados pelo grupo de pesquisa ExPRESS (Extensible & Group, 2008). Estes grafos foram extraídos de um conjunto de 1400 grafos presentes em aplicações multimídia.

Algumas das características dos grafos utilizados estão disponíveis na Tabela 4.1.

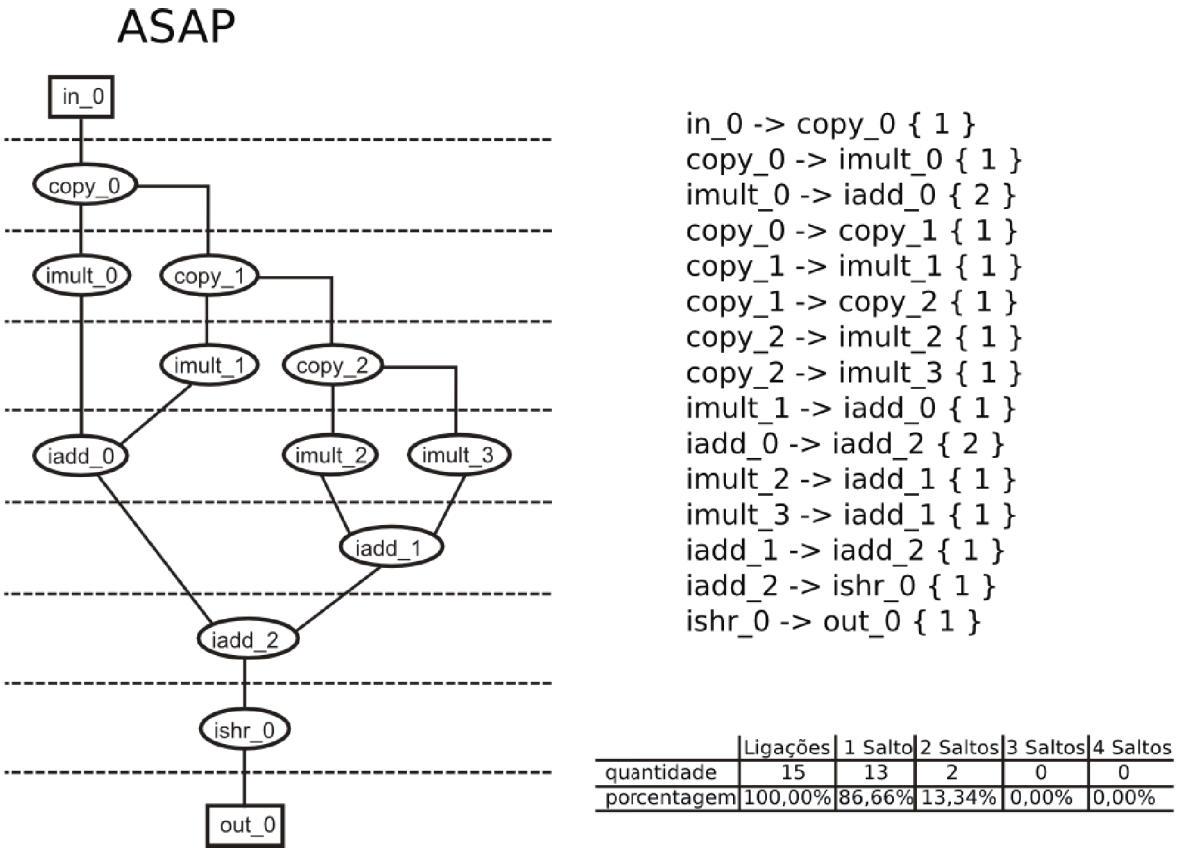
4.2 Escalonamento Gráfico

O escalonamento gráfico foi utilizado para tentar extrair uma arquitetura a partir dos grafos de fluxo de dados, sendo esse processo detalhado a seguir. O processo foi implementado com a utilização dos algoritmos O Mais Cedo Possível – *As Soon As Possible* (ASAP) e O Mais Tarde Possível – *As Late As Possible* (ALAP) para estabelecer uma métrica de distância nos grafos de fluxo de dados que são independentes da arquitetura.

O algoritmo ASAP parte de um vértice que não contenha antecessores e percorre o grafo marcando os vértices percorridos, sendo que só avança para um próximo vértice quando o vértice em que estiver posicionado não possuir nenhum antecessor não percorrido. Caso ainda haja algum antecessor não percorrido, o algoritmo retorna pelo caminho de onde veio e verifica se não há nenhum outro caminho ainda não percorrido, repetindo esses passos até ser atingido um vértice que não contenha nenhum sucessor, ou seja, o final do grafo. Um exemplo da aplicação do Algoritmo ASAP para o grafo do FIR 4 pode ser visualizado na Figura 4.4, onde cada nível representa um avanço na profunidade e a diferença entre níveis determina qual é a distância entre dois elementos, ou qual deverá ser o tamanho do segmento necessário para ligar esses dois elementos e o maior número de segmentos necessários para que o elemento raiz atinja ao elemento folha é considerado o caminho crítico do gráfico.

Já o algoritmo ALAP percorre o caminho inverso ao percorrido pelo algoritmo ASAP, partindo do ultimo vértice, ou seja, do vértice que não possui nenhum sucessor até atingir o vértice que não possua nenhum antecessor. Um exemplo do algoritmo ALAP aplicado ao grafo do FIR 4 pode ser visualizado na Figura 4.5, onde cada nível representa um avanço na profunidade e a diferença entre níveis determina qual é a distância entre dois elementos, ou qual deverá ser o tamanho do segmento necessário para ligar esses dois elementos.

Ao atingir o final do grafo, todos os vértices possuem um número que os identifica em relação à sua posição no grafo. A partir desse número é possível quantificar a distância entre os vértices que formam uma ligação, definindo assim o provável número de saltos necessários para um vértice se ligar a outro. A partir desse número de saltos são extraídos os padrões dos arcos dos grafos e formadas as arquiteturas. Na Figura 4.5, todos os arcos têm distância 1, exceto os arcos $copy_0, mult_0$ e $copy_1, mult_1$ que

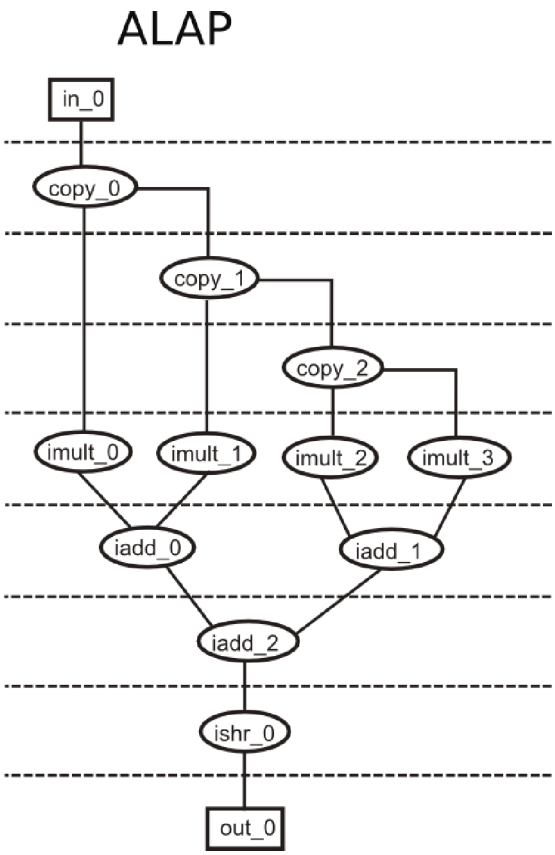


	Ligações	1 Salto	2 Saltos	3 Saltos	4 Saltos
quantidade	15	13	2	0	0
porcentagem	100,00%	86,66%	13,34%	0,00%	0,00%

Figura 4.4. Algoritmo ASAP aplicado ao fluxo de dados do FIR 4.

têm distâncias 3 e 2, respectivamente. Um histograma das distância é construído onde temos 13 ligações de custo 1, 1 de custo 2 e 1 de custo 3. Todos os grafos das aplicações são escalonados, e um histograma final de todas as arestas de todos os grafos é gerado.

Foi desenvolvido um módulo que encontra o caminho crítico da arquitetura que estiver sendo avaliada, de acordo com a quantidade de segmentos utilizados no roteamento dos elementos. Esse módulo recebe o número de segmentos utilizado pelo roteamento para satisfazer cada arco do fluxo de dados e a partir de então utiliza o algoritmo ASAP para percorrer o grafo do fluxo de dados com as informações dos números de segmentos gastos por arco e retorna a soma dos segmentos do maior caminho percorrido como sendo o caminho crítico. Um exemplo da aplicação desse módulo pode ser observado na Figura 4.6, em que um grafo de um fluxo de dados tem os seus arcos marcados de acordo com o número de segmentos gastos para satisfazê-los.




```
in_0 -> copy_0 { 1 }
copy_0 -> imult_0 { 3 }
imult_0 -> iadd_0 { 1 }
copy_0 -> copy_1 { 1 }
copy_1 -> imult_1 { 2 }
copy_1 -> copy_2 { 1 }
copy_2 -> imult_2 { 1 }
copy_2 -> imult_3 { 1 }
imult_1 -> iadd_0 { 1 }
iadd_0 -> iadd_2 { 1 }
imult_2 -> iadd_1 { 1 }
imult_3 -> iadd_1 { 1 }
iadd_1 -> iadd_2 { 1 }
iadd_2 -> ishr_0 { 1 }
ishr_0 -> out_0 { 1 }
```

	Ligações	1 Salto	2 Saltos	3 Saltos	4 Saltos
quantidade	15	13	1	1	0
porcentagem	100,00%	86,66%	6,67%	6,67%	0,00%

Figura 4.5. Algoritmo ALAP aplicado ao fluxo de dados do FIR 4.

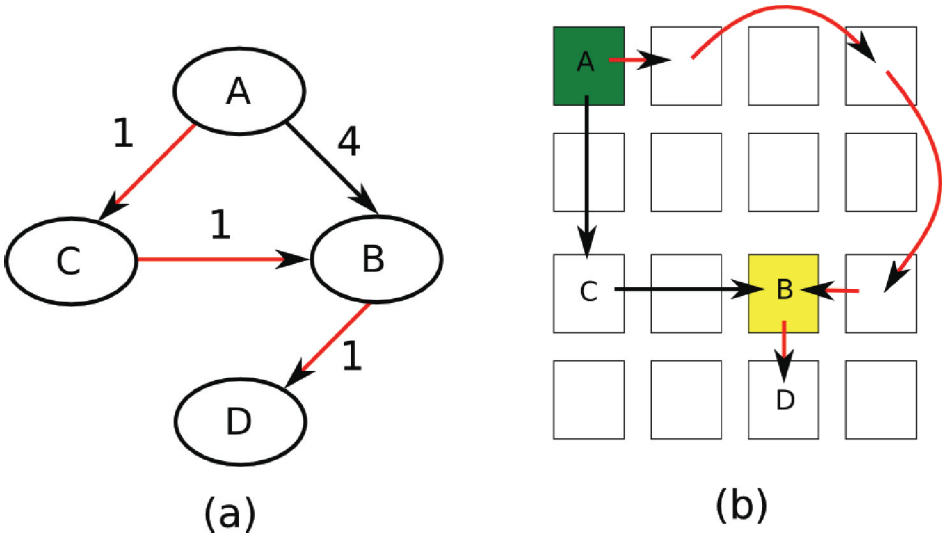


Figura 4.6. Exemplo de um caminho crítico:(a) caminho crítico ideal composto por *A->C->D* o grafo foi marcado com o número de segmentos gastos em cada arco, (b) caminho crítico alterado pelo mapeamento, sendo agora o maior caminho composto por *A->B->D*.

4.3 Repositório de Arquiteturas e Mapeamento

No módulo Repositório de Arquiteturas se encontram as descrições em XML das arquiteturas que servirão de entrada para os algoritmos presentes no módulo de heurísticas de busca. É nesse módulo também que são armazenadas as arquiteturas modificadas pelas heurísticas aplicadas pelo módulo de geração.

O módulo de mapeamento é o responsável pela tradução do modelo de fluxo de dados das aplicações avaliadas em um modelo implementável em hardware, chamado nesse trabalho de arquitetura, onde os elementos necessários para a execução do software, como ALU, multiplicadores, somadores, etc., são posicionados em um modelo de estrutura física e posteriormente as sequências de uso desses elementos de software são traduzidas como fios que as ligam no modelo físico.

O módulo de mapeamento recebe como entrada os grafos e fluxo de dados vindos do repositório de aplicações, e as arquiteturas vindas do gerador de arquiteturas e/ou informações vindas do módulo de escalonamento gráfico, gerando informações para as heurísticas utilizadas no módulo gerador de arquiteturas.

Todos os detalhes envolvidos do módulo de mapeamento podem ser encontrados nas Seções 3.3.1 e 3.3.2 do Capítulo 3.

4.4 Gerador de Arquiteturas

As heurísticas apresentadas nos próximos tópicos partem de um padrão de segmento (ou de um conjunto de padrões) e geram variações nesses padrões a fim de obter um padrão que, ao ser mapeado e avaliado, apresente resultados melhores que os obtidos com os padrões de origem. No entanto se o padrão de origem já for um padrão considerado "ótimo " corre-se o risco de rodar determinada heurística por horas (ou até mesmo dias) sem, no final do processo, obter um resultado melhor que o obtido com o padrão de origem, para evitar tal risco cada heurística possui um ponto de parada independente dos resultados obtidos e do conjunto de *benchmark* avaliado, com isso para um grupo maior de *benchmark* a heurística não será interrompida em um determinado tempo que a impessa de percorrer o mesmo espaço de solução percorrido para um grupo menor de *benchmarks*.

Como tentativa de sempre obter o melhor padrão possível, as arquiteturas geradas seguem o mesmo padrão na entrada e na saída da heurística utilizada, o que permite que a saída do resultado de uma heurística sirva de entrada para outra heurística.

Além disso, para este trabalho foi desenvolvido um sistema de avaliação dos resultados obtidos com o mapeamento que funciona independentemente da heurística aplicada, facilitando a coerência entre os diversos resultados obtidos com as diferen-

tes combinações heurísticas possíveis. O método de avaliação desenvolvido consegue armazenar o melhor indivíduo gerado independentemente da fase onde ele foi obtido, permitindo ainda a captura de um número X qualquer de indivíduos considerados os melhores.

Outra função de uso comum identificada diz respeito ao que é considerado uma mutação no AG, uma perturbação no *Simulated Annealing* (SA) ou simplesmente a transformação aleatória de um indivíduo em outro com o PR. Para esses três pontos, foi utilizada uma única pertubação simples, que será descrita na Seção 4.4.2.4.

4.4.1 Path Relinking

O Algoritmo *Path Relinking* é uma estratégia de intensificação proposta por Glover (1997), vinculada ao método de busca Tabu. Essa estratégia consiste na transformação de um indivíduo S1 em um indivíduo S2. O processo de transformação seleciona, a cada iteração, uma característica de S2 e a repassa para S*, substituindo a uma das características de S1, até que o indivíduo S* seja igual ao indivíduo S2. A Figura 4.7 representa essa transformação.

A implementação do Algoritmo PR utilizada nesse trabalho armazena o melhor indivíduo obtido durante a fase de transformação de um indivíduo em outro, o que pode ser observado na Figura 4.7, em que o indivíduo considerado o melhor aparece em destaque. Esse processo de escolha do melhor indivíduo se repete até a conclusão do algoritmo, quando se tem o melhor indivíduo global, assim como o padrão de ligação desse indivíduo.

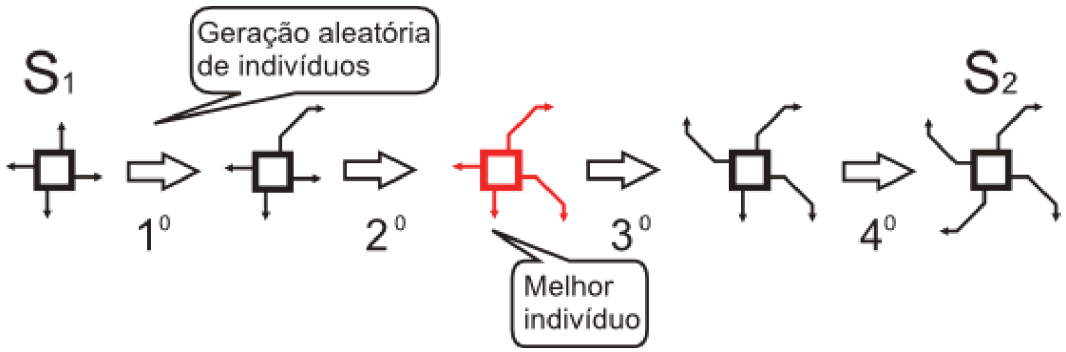


Figura 4.7. Processo de transformação do Path Relinking.

Para esse trabalho, o algoritmo poderia ser alimentado a partir de uma população inicial formada por parte dos mesmos indivíduos utilizados pelo AG (descrito na Seção 4.4.2), normalmente escolhidos de forma aleatória, ou ainda por uma determinada

quantidade de indivíduos que formassem os melhores indivíduos obtidos com o AG ou com SA.

O pseudocódigo do PR desenvolvido pode ser visto no Algoritmo 4.1. Nele pode ser visualizada uma condição de parada que garante o fim do algoritmo. Essa inserção foi necessária, uma vez que o processo de transformação de um indivíduo em outro é um processo aleatório, podendo levar o algoritmo a um estado infinito (ou quase infinito) de geração de caracteríticas, porém notou-se que essa condição de parada é atingida em apenas 0.10% dos casos.

Algoritmo 4.1: Path Relinking

```
1 pathRelinkig (S1, S2){
2     S* = S1; cont = 0;
3     enquanto(S* != S2){
4         característica(S*) = característicaAleatória(S2);
5         se (cont > númeroCaracterísticas(S2) ^ 2){
6             S* = S2;
7         }
8         ++cont;
9         avaliar(S*);
10    }
11 }
```

4.4.2 Algoritmos Genéticos

O Algoritmo Genético utilizado neste trabalho foi uma implementação voltada para a manipulação de grafos de fluxo de dados, e a base teórica para a implementação desse algoritmo pode ser encontrada em Russell & Norvig (2004) e um pseudocódigo de um AG pode ser visualizado no Algoritmo 4.2. Porém, nessa variação implementada durante a fase de classificação da população inicial, cada indivíduo ou arquitetura, é mapeado de acordo com o descrito na Seção 3.3 e, de acordo com a métrica utilizada, um valor numérico será gerado e posteriormente utilizado para a inclusão do indivíduo no grupo de indivíduos que serão cruzados ou descartados.

Durante o cruzamento, 20% dos indivíduos considerados os melhores são cruzados em pares gerando outros 20% de novos indivíduos que substituem os 20% dos indivíduos considerados os piores. E de acordo com a probabilidade de mutação definida, os indivíduos depois de cruzados sofrem uma alteração, que no caso deste trabalho é a alteração em um valor numérico correspondente à alteração de um segmento da arquitetura.

Após essas alterações nos indivíduos, os novos indivíduos obtidos são avaliados pelo mapeamento descrito na Seção 3.3, e um novo valor numérico é obtido e então toda a população renovada é reclassificada para a determinação dos 20% melhores e dos 20% piores, esse processo se repete até que a condição de parada seja atingida.

Algoritmo 4.2: Algoritmo Genético

```
1 algoritmoGenético(populaçãoInicial, númeroDeGerações){
2     indivíduo[] = populaçãoInicial;
3     para(indivíduo[0] até indivíduo[n-1]) faça{
4         calculaFitness(indivíduo[++n]);
5         avaliar(indivíduo[++n]);
6     }
7     ordena(indivíduo);
8     geraçãoAtual = 0;
9     enquanto(geraçãoAtual < numeroDeGerações){
10        umPassoDoGenético(indivíduo);
11        ordena(indivíduo);
12        ++geraçãoAtual;
13    }
14 }
15 umPassoDoGenético(indivíduo){
16    para(indivíduo[0] até individuo[n-2])
17        cruzar(indivíduo[n], indivíduo[++n], resultFilho1,
   resultFilho2);
18        calcularFitness(resultFilho1);
19        se(númeroRandômico = taxaMutação){
20            resultFilho1.mutação;
21        }
22        avaliar(resultFilho1);
23        indivíduo[n] = resultFilho1;
24        calcularFitness(resultFilho2);
25        se(númeroRandômico = taxaMutação){
26            resultFilho2.mutação;
27        }
28        avaliar(resultFilho2);
29        indivíduo[++n] = resultFilho2;
30    }
31 }
```

4.4.2.1 População Inicial

Para as populações iniciais utilizadas neste trabalho, procuramos utilizar sempre 100 indivíduos diferentes. Para obter tais indivíduos, utilizamos as arquiteturas baseadas em malha e fizemos testes com arquiteturas baseadas em inversão de bit, ambas descritas na Seção 3.2 do Capítulo 3.

Esses 100 indivíduos diferentes são obtidos por cruzamentos manuais entre as arquiteturas em que características de duas arquiteturas são previamente misturadas seguindo algum tipo de padrão lógico, como, por exemplo, os segmentos pares seriam 0_2_hop e os segmentos ímpares seriam 0_4_hop, ou a primeira metade dos segmentos seriam octal_0_hop e a outra metade octal_3_hop, além de uma mesma população poder conter indivíduos octal e 0_n_hop.

Os testes realizados com arquiteturas baseadas em inversão de bit não conseguiram gerar indivíduos regulares, apesar de conseguir resultados ligeiramente melhores, da ordem de 1% no número de segmentos, como um dos objetivos deste trabalho era gerar arquiteturas escálaveis os resultados obtidos com as arquiteturas baseadas em inversão de bit foram descartados dos resultados apresentados no Capítulo 5.

4.4.2.2 Cruzamento

Propomos três possibilidades de cruzamento de indivíduos para uso com o AG, no entanto, todo o ponto de corte continua sendo aleatório, o que muda é o que é considerado durante o corte. A primeira abordagem é baseada em conjuntos de PEs ou nós. A segunda abordagem se baseia em um conjunto de interligações locais entre cada PE (Teixeira et al., 2007). E a terceira possibilidade utiliza o algoritmo PR para fazer o cruzamento entre dois indivíduos (Teixeira et al., 2009). Vamos considerar uma matriz de arquiteturas no formato N x M, em que N e M são os números de linhas e colunas, respectivamente.

4.4.2.2.1 Abordagem de conjunto de elementos de processamento: A arquitetura pode ser modelada como um conjunto de PEs. Cada PE é definido pelo seu ID e um conjunto definido de segmentos locais. Vamos examinar as duas arquiteturas conforme mostrado na Figura 4.8, em que cada PE tem quatro segmentos de saída e quatro de entrada.

Na primeira arquitetura, cada PE tem seus segmentos em linha e em coluna formando linhas horizontais e verticais. Para a segunda arquitetura, cada PE tem seus segmentos formando diagonais entre linhas e colunas.

Cada arquitetura pode ser representada como um vetor unidimensional, em que a posição "i" armazena o conjunto de locais de saída das ligações do PE_i. Vamos

considerar os vetores A e B na Figura 4.9, que representam as duas arquiteturas já mostradas na Figura 4.8.

O AG irá criar novas arquiteturas pela aplicação da operação de cruzamento nos indivíduos A e B. Suponha o ponto de corte do vetor entre a posição 2 e a posição 3. Duas novas arquiteturas serão geradas como mostra a Figura 4.10.

O primeiro filho terá os PEs de 0 a 2 do indivíduo A e os PEs de 3 a 8 de B. Cada PE traz seus segmentos de saída para a nova arquitetura e recebe os segmentos de entrada vindos dos outros PEs que formarão assim uma nova arquitetura. O segundo filho receberá as outras partes dos vetores, ou seja, PEs de 0 a 2 do indivíduo B e os PEs de 3 a 8 de A.

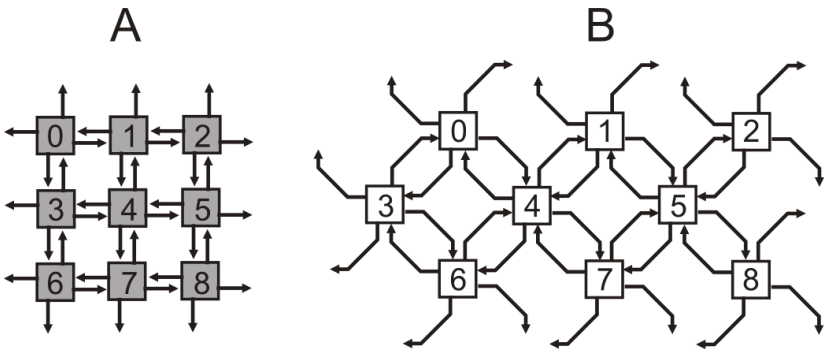


Figura 4.8. Arquiteturas iniciais.

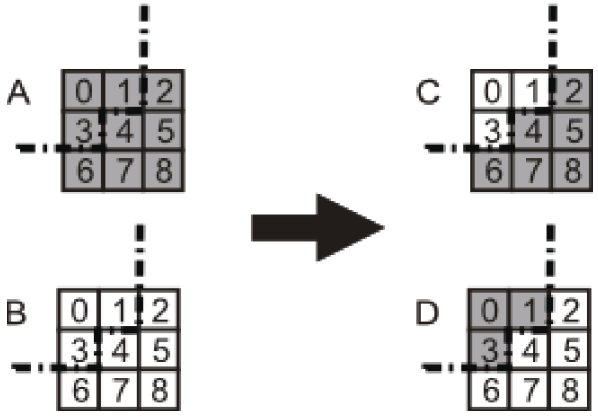


Figura 4.9. Vetores onde o cruzamento ocorre.

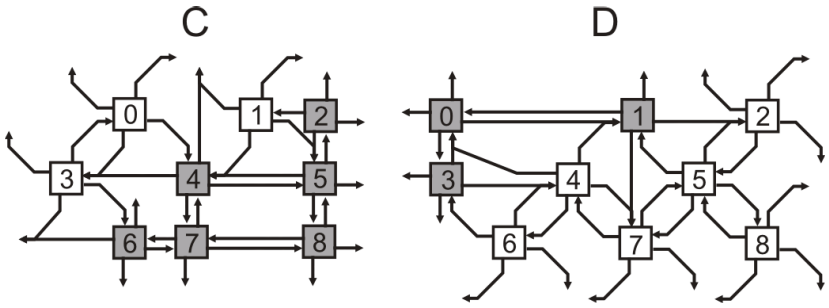


Figura 4.10. Novas arquiteturas formadas.

O grande problema dessa abordagem é a formação de arquiteturas irregulares que acabaram gerando um maior custo de implementação em hardware. Um exemplo de uma arquitetura que poderia ser gerada pode ser visualizada na Figura 4.11.

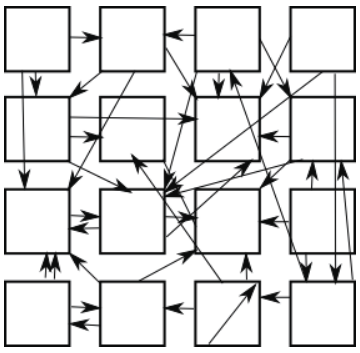


Figura 4.11. Arquitetura irregular gerada pelo crossover utilizando conjunto de elementos de processamento.

4.4.2.2.2 Abordagem de interconexões locais: Vamos considerar arquiteturas homogêneas, em que cada PE tem o mesmo conjunto de segmentos. Como exemplo, podemos verificar as arquiteturas mostradas na Figura 4.12, em que cada PE tem quatro segmentos de saída nas direções em linha e coluna. Esses conjuntos de segmentos são modelados pelos vetores mostrados na Figura 4.13.

Cada saída tem um padrão definido em função do seu endereço linha e coluna, onde usamos "i" como o número da linha e "j" como o número da coluna. Por exemplo, a saída 1 da arquitetura B da Figura 4.12 é uma função de i-1, j+1, ou seja, a saída é conectada à linha anterior e a próxima coluna.

O operador de cruzamento seleciona um ponto de corte aleatório, em um local do vetor de conjuntos de segmentos de um elemento de cada uma das arquiteturas, estabelecendo a geração de dois novos conjuntos de segmentos. Esses novos conjuntos podem ser vistos na Figura 4.13 (c) e (d).

A Figura 4.14 mostra as novas arquiteturas geradas a partir do ponto de corte entre os segmentos 0 e 1 de cada um dos elementos que formam as arquiteturas A e B.

Essa abordagem produz arquiteturas escaláveis, em que havendo necessidade de aumento do número de elementos basta produzirem novos a partir do modelo de conjuntos de segmentos utilizado, ou seja, a partir de uma única matriz podem-se produzir arquiteturas de qualquer tamanho.

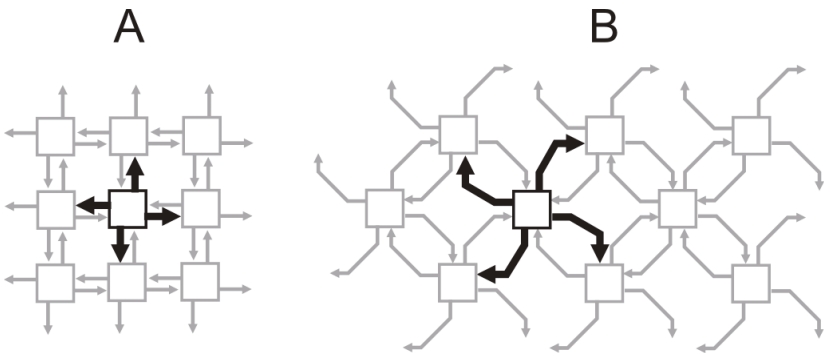


Figura 4.12. Arquiteturas iniciais com um elemento em destaque.

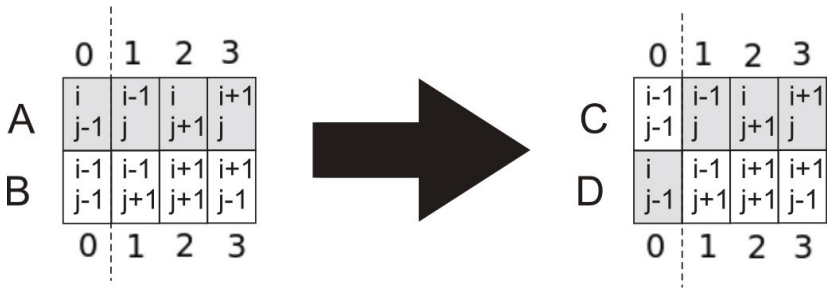


Figura 4.13. Vetores que representam os elementos destacados sendo cruzados.

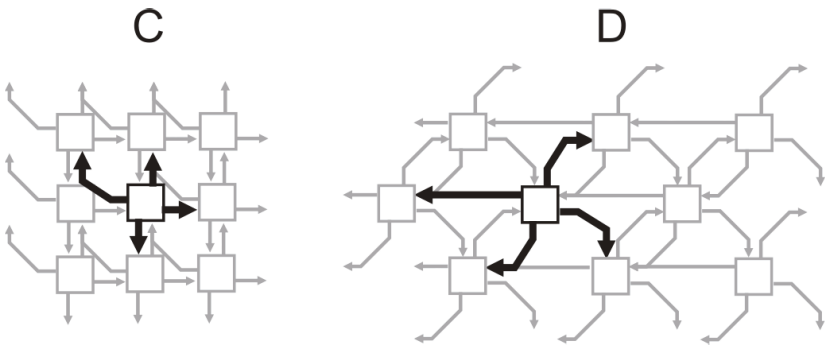


Figura 4.14. Novas arquiteturas geradas a partir do cruzamento dos elementos.

4.4.2.2.3 Algoritmo Genético com cruzamento baseado em Path Relinking

O AG desenvolvido nesse trabalho possui ainda um terceiro método de cruzamento que é baseado no Algoritmo PR, em que o primeiro filho é gerado pela aplicação do PR do Pai_1 para o Pai_2, e o segundo filho do Pai_2 para o Pai_1. Em ambos os casos, o indivíduo com melhor resultado de *fitness* obtido, entre um pai e o outro, é selecionado como sendo o filho resultante do cruzamento. Como a geração de indivíduos intermediários é um processo aleatório, cada filho tem uma grande probabilidade de ser diferente. Um exemplo dessa transformação pode ser visualisado na Figura 4.15.

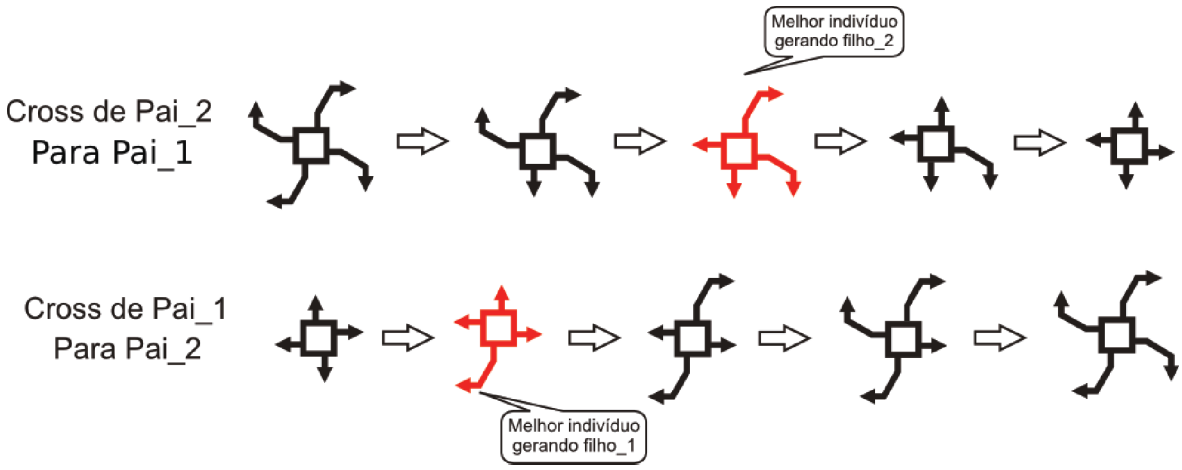


Figura 4.15. Processo de cruzamento utilizando Path Relinking.

Para este trabalho, a aplicação dessa variação de cruzamento se deu entre os elementos formadores das arquiteturas, como proposto no tópico anterior, gerando arquiteturas regulares, porém o tempo gasto por essa variação do algoritmo foi maior que

as outras opções de cruzamento gerando no final da execução do algoritmo resultados piores que os obtidos nas outras variações do algorítmo. O detalhamento dos resultados obtidos estão expostos no Capítulo 5.

4.4.2.3 Fitness

A função de *fitness* é a mesma utilizada pelas três heurísticas implementas neste trabalho. Ela faz parte do módulo de avaliação do gerador de arquiteturas e está inserida na parte de métricas do mapeamento, pois por meio dos cruzamentos e mutações, o AG manipula as arquiteturas e essas arquiteturas modificadas são passadas para o módulo de mapeamento, em que o algoritmo descrito na Seção 3.3.3 do mapeamento é aplicado, e o valor obtido é retornado ao AG, ou PR ou SA, enfim, ao algoritmo que o chamou, em forma de um valor que serve de fator de descarte ou aproveitamento da arquitetura que estiver sendo avaliada.

O cálculo do *fitness* pode considerar ainda a média das médias obtidas com a avaliação de cada grafo de fluxo de dados do conjunto que estiver sendo avaliado, ou a soma das somas dos segmentos obtidos.

4.4.2.4 Mutação

Esse ponto do AG é chamado com uma determinada probabilidade durante sua execução, causando uma mutação em um determinado ponto, que pode ser um aumento ou diminuição do número de saltos de uma linha ou coluna, o bit a ser invertido em uma arquitetura de inversão de bits, enfim, qualquer ponto que contenha algum valor numérico pode sofrer uma mutação.

Como ocorre na função utilizada como *fitness*, essa função é a mesma utilizada na fase de perturbação do SA e na fase de transição do PR. Um exemplo da aplicação desse Algoritmo pode ser visualizado na Figura 4.16.

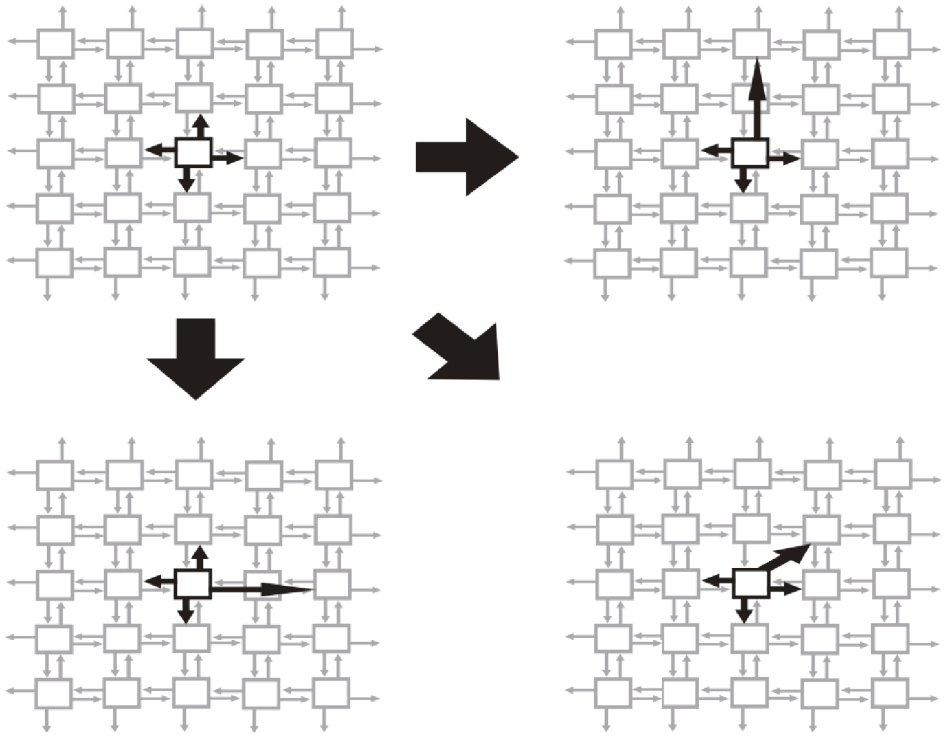


Figura 4.16. Possibilidades de mutação.

4.4.3 Simulated Annealing

A metaheurística *Simulated Annealing* é um algoritmo derivado de processos de recozimento de metais sólidos desenvolvido por Metropolis, em 1953 (Youssef et al., 2001). A analogia com a otimização combinatória foi introduzida por Kirkpatrick et al. (1983) e aperfeiçoada por Cerný (1985). O termo *annealing* refere-se a um processo de resfriamento térmico que começa pela liquidificação de um cristal, a alta temperatura, seguido pela lenta e gradativa diminuição de sua temperatura, até que o ponto de solidificação seja alcançado, quando o sistema atinge um estado de energia mínima. O SA pode ser utilizada na solução de diversos problemas de otimização combinatória.

Diferentemente do foco dado aos algoritmos PR e AG, que neste trabalho podem ser utilizados para gerar indivíduos a partir de uma população inicial, o algoritmo SA é utilizado para tentar obter um indivíduo ainda melhor, partindo do melhor indivíduo obtido com o AG ou com o PR.

Para tal ele utiliza o mesmo método de perturbação e avaliação utilizado pelos demais Algoritmos dentro da estrutura algorítmica apresentada no Algoritmo 4.3.

Algoritmo 4.3: Simulated Annealing

```
1  simulatedAnnealing(indivíduo, tempInícial, tempFinal, maxIteração){
2      temperatura = tempInicial;
3      iAtual = avaliar(indivíduo);
4      enquanto(temperatura < tempFinal){
5          enquanto(iteração < maxIteração){
6              indNovo = pertubação(indAtual);
7              iNovo = avaliar(indNovo);
8              delta = iNovo - iAtual;
9              se(delta < 0){
10                 indAtual = indNovo;
11             }
12             senão se (randômico < e^-(delta/temperatura)){
13                 indAtual = indNovo
14             }
15         }
16         temperatura = variação(temperatura);
17     }
18 }
```

Capítulo 5

Resultados

Este capítulo apresenta os resultados obtidos com o Gerador Automático de Arquiteturas usando as duas técnicas de geração de arquiteturas: baseada nas arquiteturas e baseada nas aplicações. O objetivo é encontrar uma arquitetura mais adequada para um dado conjunto de aplicações. Todas as arquiteturas são bi-dimensionais, representadas por uma matriz, e os elementos de processamento implementam operadores de cálculo (soma, multiplicação etc.) e fluxo de dados (copy, merge etc.). O parâmetro avaliado é o padrão de ligação de um PE com os seus vizinhos, na busca de uma melhor topologia de interconexões para a arquitetura alvo. O critério de custo é minimizar o número total de segmentos para o roteamento de todo o conjunto de grafos das aplicações que foi mapeado na arquitetura alvo. Ao aplicar uma técnica de busca, a(s) melhor(es) arquitetura(s) é(são) armazenadas e podem ser reutilizadas através de uma realimentação na ferramenta para uma otimização usando outra técnica. Por exemplo, podemos aplicar o AG e depois aplicar o SA sobre o melhor resultado do primeiro.

Inicialmente, iremos apresentar as possibilidades de representação e vizualização dos resultados.

5.1 Possibilidades de Representações

Os resultados obtidos podem ser representados de várias formas, tais como: gráficos de concentração, em que cada resultado não nulo aparece como um ponto possibilitando a interpretação de locais de maior ou menor concentração de resultados; gráficos de variações, em que apenas os resultados de maior e menor valores são mostrados; e o intervalo entre eles aparece totalmente preenchido fornecendo uma visão mais nítida da variação dos resultados. Exemplos desses dois tipos de gráficos podem ser visualizados nas Figuras 5.1a e b. Esses gráficos foram gerados a partir dos resultados obtidos da

aplicação do AG sobre uma população inicial, onde os PE possuiam 8 segmentos por nó no formato 0_n_hop variando de 0_1_hop até 0_4_hop.

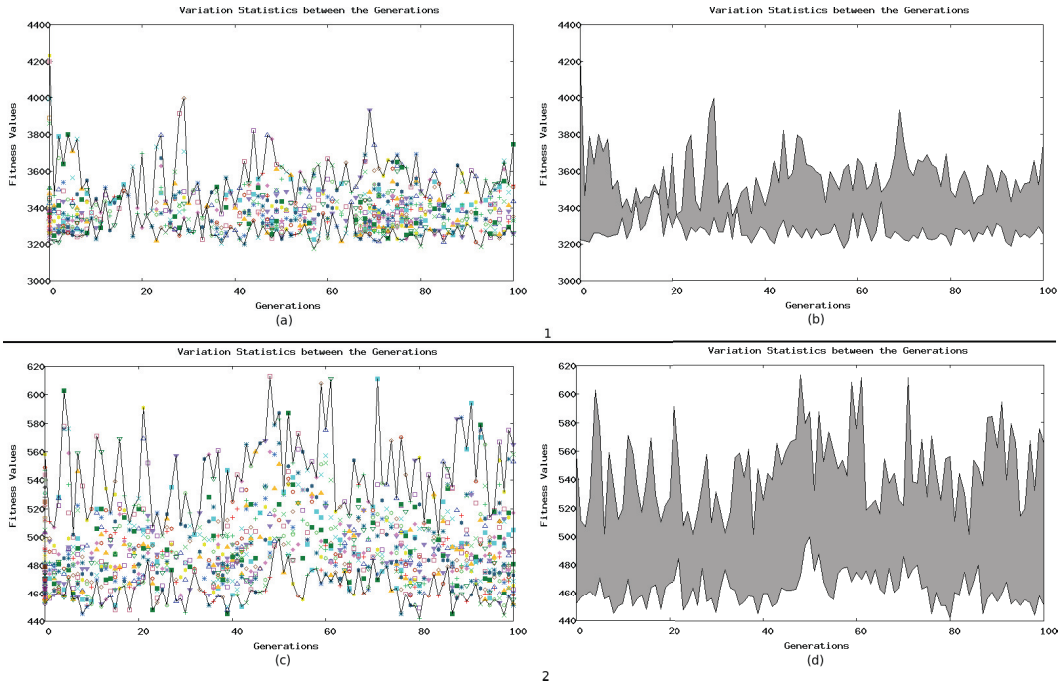


Figura 5.1. Exemplo – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno

Uma terceira forma de representação gráfica é obtida pela descrição em XML da arquitetura considerada a melhor para o *benchmark* no qual ela esteja sendo avaliada, em que a descrição em XML é utilizada para gerar um modelo gráfico dos segmentos. Um exemplo desse modelo gráfico pode ser visualizado na Figura 5.2.

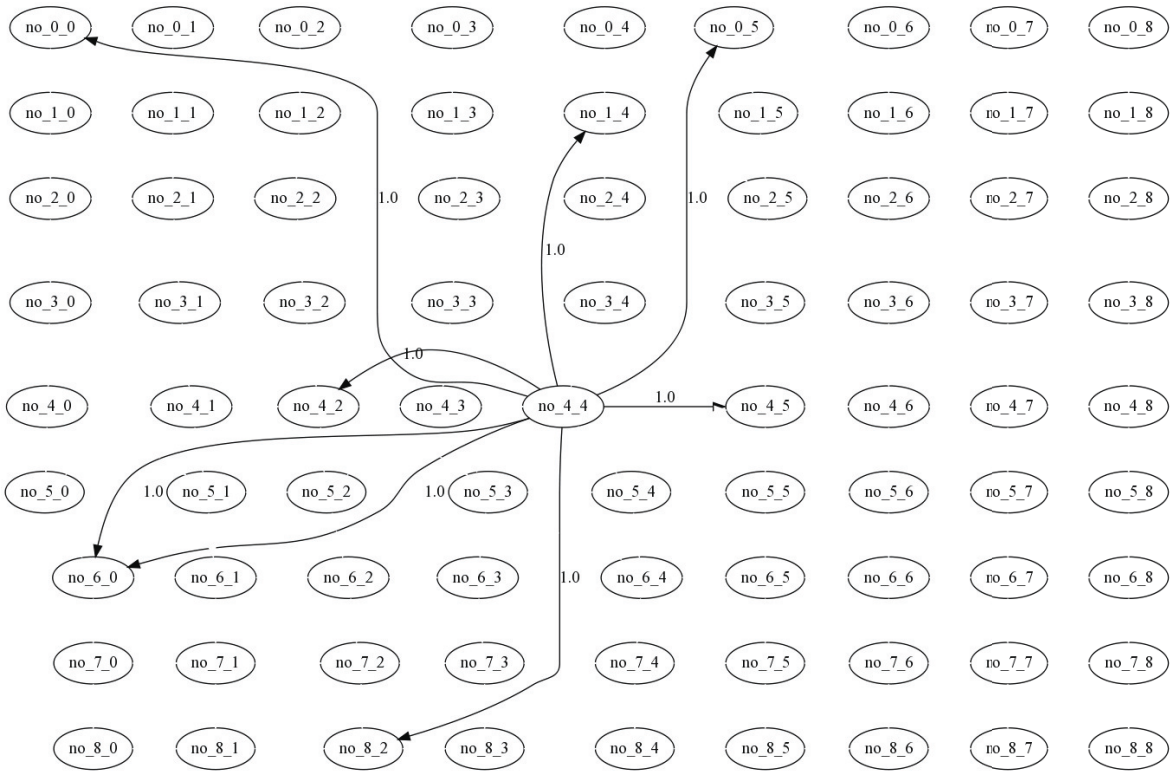


Figura 5.2. Conexões – para esse exemplo uma matriz de 9 x 9 foi usada para ilustrar as saídas de cada um dos nós, sendo representadas as saídas do no 4_4.

Em uma quarta forma de representação gráfica, pode-se mapear um conjunto de *benchmarks* na melhor arquitetura encontrada para um dado benchmark, ou subconjunto chamado de grupo de treinamento, comparando o resultado obtido com uma arquitetura de referência. Com essa forma de representação, podemos verificar se uma arquitetura encontrada para o grupo de treinamento apresenta um bom resultado para os *benchmarks* de um grupo maior. Um exemplo dessa representação pode ser visualizado na Figura 5.3, em que a arquitetura de referência utilizada foi a 0_1_hop. O conjunto de *benchmarks* utilizado nessa comparação pode ser visualizado na Tabela 5.3 e na Figura 5.7.

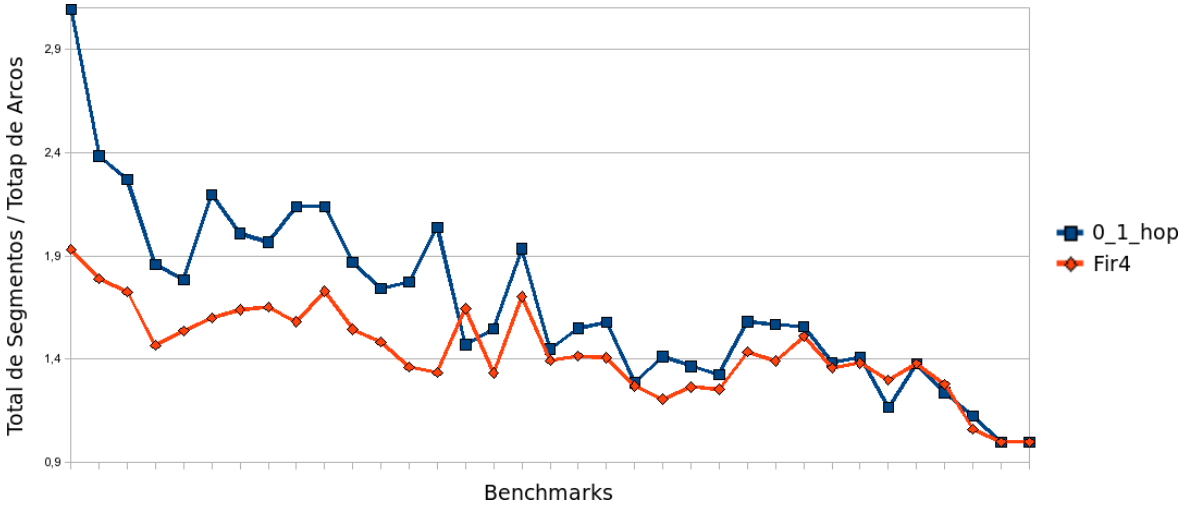


Figura 5.3. Gráfico de comparação entre 0_1_hop e a melhor arquitetura obtida para o FIR 4.

Os formatos de apresentação de resultados descritos até agora servem tanto para resultados individuais, quanto para resultados coletivos, em que um conjunto de *benchmarks* é mapeado. Já as apresentações em tabelas têm como objetivo principal fornecer uma comparação entre *benchmarks* de um conjunto ou entre conjuntos de *benchmarks*, como por exemplo, a Tabela 5.1, que apresenta os resultados obtidos com o uso do Algoritmo de PathFinder para o roteamento dos *benchmarks* extraídos de Extensible & Group (2008) na arquitetura 0_1_hop.

A coluna arcos representa o número de arcos presentes em cada benchmark, a coluna segmentos representa o total de segmentos utilizados na arquitetura para possibilitar a satisfação de todos os arcos presentes em cada *benchmark* e a coluna segmentos / arcos representa a quantidade média de segmentos utilizados para o mapeamento dos arcos.

Uma última forma de apresentação de resultados é a exposta na Figura 5.4, em que os elementos aparecem posicionados em uma matriz, que seria a matriz de elementos de alta granularidade, na qual os elementos extraídos dos grafos de fluxo de dados dos *benchmarks* são posicionados.

Tabela 5.1. Resultado do PathFinder de 0_1_hop sobre *benchmarks* (Extensible & Group, 2008)

Benchmark	Arcos	Segmentos	Segmentos / Arcos
arf	30	33	1,1
collapse_pyr	89	132	1,48
cosine1	76	119	1,57
cosine2	91	208	2,29
ewf	55	77	1,4
feedback_points	51	80	1,57
FIR1	43	67	1,56
FIR2	39	57	1,46
h2v2_smooth_downsample	55	86	1,56
hal	8	10	1,25
horner_bezier_surf	16	18	1,13
idctcol	236	563	2,39
interpolate_aux	104	208	2
invert_matrix_general	380	915	2,41
jpeg_fdct_islow	210	458	2,18
jpeg_idct_ifast	210	468	2,23
matmul	124	227	1,83
motion_vectors	29	36	1,24
smooth_color_z_triangle	196	440	2,24
write_bmp_header	93	154	1,66

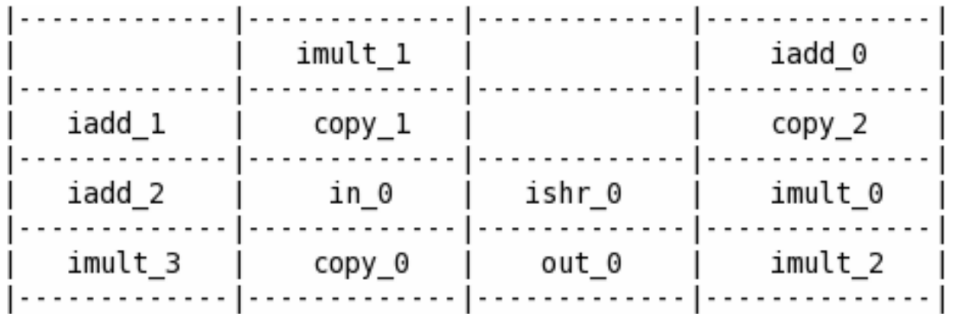


Figura 5.4. Posicionamento do FIR 4 na melhor arquitetura obtida pelo Algoritmo Genético.

5.2 Ferramenta de Geração Desenvolvida

Com base no definido no Capítulo 4, foi desenvolvida uma ferramenta de geração automática de arquiteturas. Para os estudos de caso, foram seguidos dois fluxos, conforme

descrito no Capítulo 4. O relacionamento das heurísticas envolvidas está representado na Figura 5.5. Na parte superior da Figura 5.5a, mostramos o primeiro ciclo de experimentos baseados na otimização de um conjunto inicial de arquiteturas, sobre as quais são aplicadas as técnicas de busca, gerando novas arquiteturas mais adequadas. O ponto de partida é uma população inicial composta por 100 indivíduos formados por padrões de ligação variando de 0_1_hop até 0_4_hop. A ferramenta alimenta a heurística AG usando para o cálculo do *fitness* o resultado do mapeamento dos *benchmarks* de cada grupo, e conforme o estudo de caso, são aplicadas duas variações do AG, com cruzamento por nó e com cruzamento por PR, gerando uma arquitetura considerada a melhor para cada variação, gerando também os 10 melhores indivíduos para cada variação.

Além disso, os 10 melhores indivíduos encontrados pelo AG em cada uma das variações são utilizados como entrada para a heurística de PR. Essa heurística é aplicada também a 10 indivíduos escolhidos de forma aleatória da mesma população usada pelo AG.

A partir das melhores arquiteturas obtidas com as variações do AG e das aplicações do PR, a heurística SA é aplicada a tais arquiteturas com a finalidade de se tentar chegar a uma arquitetura ainda melhor, chegando a um terceiro nível de otimização.

No segundo ciclo de experimentos ilustrado na parte inferior da Figura 5.5b, usa-se a segunda abordagem de geração de arquiteturas. A partir dos grafos das aplicações, os algoritmos de escalonamento são aplicados derivando um histograma de distâncias entre os vértices, que é usado para gerar uma arquitetura. Após a geração, o Algoritmo de SA também é aplicado.

A ferramenta foi utilizada por todos os estudos de caso para a geração dos resultados, sendo que o processo básico pode ser visualizado na Figura 5.6 em que durante o treinamento um grupo de *benchmarks* é utilizado para se chegar a uma, ou várias, arquitetura(s) consideradas as melhores para aquele grupo, de acordo com a heurística utilizada. Após essa fase, a melhor arquitetura encontrada é então mapeada em um conjunto de *benchmarks* chamado conjunto total, e o resultado obtido é considerado durante a comparação entre os estudos de caso. O detalhamento de cada estudo de caso será feito na Seção 5.4.

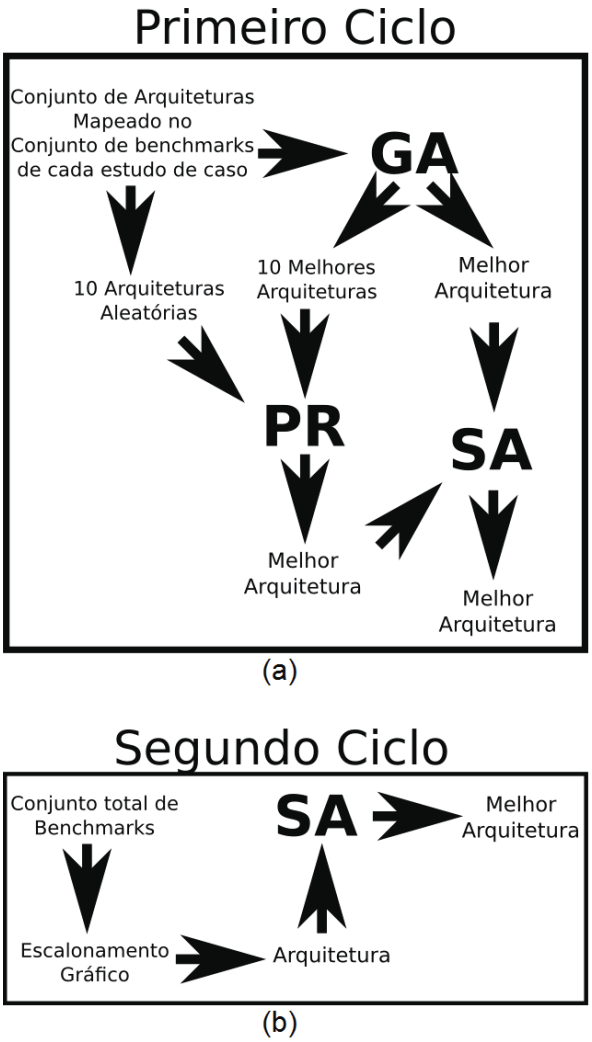


Figura 5.5. Relação das heurísticas com os ciclos.

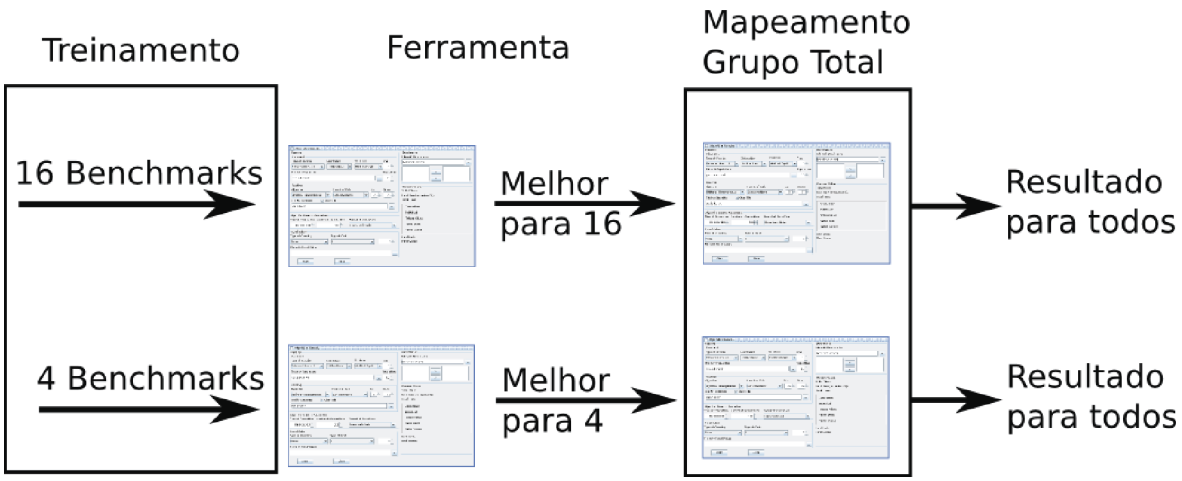


Figura 5.6. Esquema básico utilizado por todos os estudos de caso.

5.3 Organização dos Estudos de caso

A criação dos estudos de caso apresentados teve dois objetivos: procurar um conjunto de aplicações que fosse representativo, fornecendo resultados que pudessem ser válidos para um grupo mais amplo de aplicações, e verificar quais heurísticas poderiam ser mais eficientes na busca por estes resultados, considerando a qualidade dos resultados. Além disso, o menor aumento possível do caminho crítico após o mapeamento em relação ao caminho crítico ideal também foi avaliado. Para auxiliar nessa busca, a ferramenta desenvolvida possui uma interface gráfica que permite a configuração de vários parâmetros. Entre esses parâmetros, os principais podem ser visualizados na Tabela 5.2. Tais parâmetros estão descritos em detalhes no anexo A, tanto os gerais quanto os específicos para cada heurística. A utilização de tais parâmetros se deu a fim de poder ser feita uma comparação mais objetiva dos resultados obtidos por ambas as heurísticas.

Primeiro selecionamos dois subgrupos de aplicações e geramos a(s) melhor(es) arquitetura(s) para eles. Posteriomente, o conjunto de todas as aplicações foi mapeado na melhor arquitetura para uma segunda fase de avaliação do resultado gerado pela ferramenta. O número de vértices e arestas para o conjunto total de aplicações está descrito na Tabela 5.3 e Figura 5.7.

Tabela 5.2. Parâmetros globais utilizados em todos os estudos de caso

Parâmetro	Opção utilizada
Tipo de toróide	entre a próxima linha e a última coluna da linha
Direção dos arcos	unidirecional
Algoritmo de posicionamento	profundidade modificada
População de posicionamento	11
Tempos de posicionamento	1 ms
Fórmula para o custo dos segmentos	fixo em 1
Algoritmo de roteamento	Dijsktra
Limite inferior	25
Limite superior	25
Formato dos resultados	total de segmentos

Tabela 5.3. Conjunto Total com 35 *benchmarks*

Fonte	Ordem	Benchmarks	Arcos	Vértices
Selecionados via mediabench ExPRESS (Extensible & Group, 2008)	1	arf	30	28
	2	Collapse pyr	89	72
	3	cosine1	76	66
	4	cosine2	91	82
	5	ewf	55	42
	6	feedback points	51	54
	7	FIR1	43	44
	8	FIR2	39	40
	9	h2v2 smooth downsample	55	52
	10	hal	8	11
	11	horner bezier surf	16	17
	12	idctcol	236	186
	13	interpolate aux	104	108
	14	invert matrix general	380	359
	15	jpeg fdct islow	210	173
	16	jpeg idct ifast	210	167
	17	matmul	124	116
	18	motion vectors	29	32
	19	smooth color z triangle	196	196
	20	write bmp header	93	111
Códificados manualmente por Ferreira et al. (2004)	21	FIR16	63	49
	22	Cplx8	60	46
	23	FilterRGB	70	57
	24	FDCTa	186	139
	25	FDCT	124	92
	26	FilterRGB+Paeth	104	82
	27	SNN	295	249
	28	TreeFIR64	255	193
	29	FIR64	255	193
	30	FilterRGB+Paeth+FDCT	290	221
Gerados aleatóriamente via TGFF disponibilizado por Dick et al. (1998)	31	graph50_2x2	89	64
	32	graph100_2x2	169	124
Codificados manualmente por Guerra (2008) com base no processador mips	33	Conf10	34	21
	34	Conf11	25	20
	35	Conf12	6	6

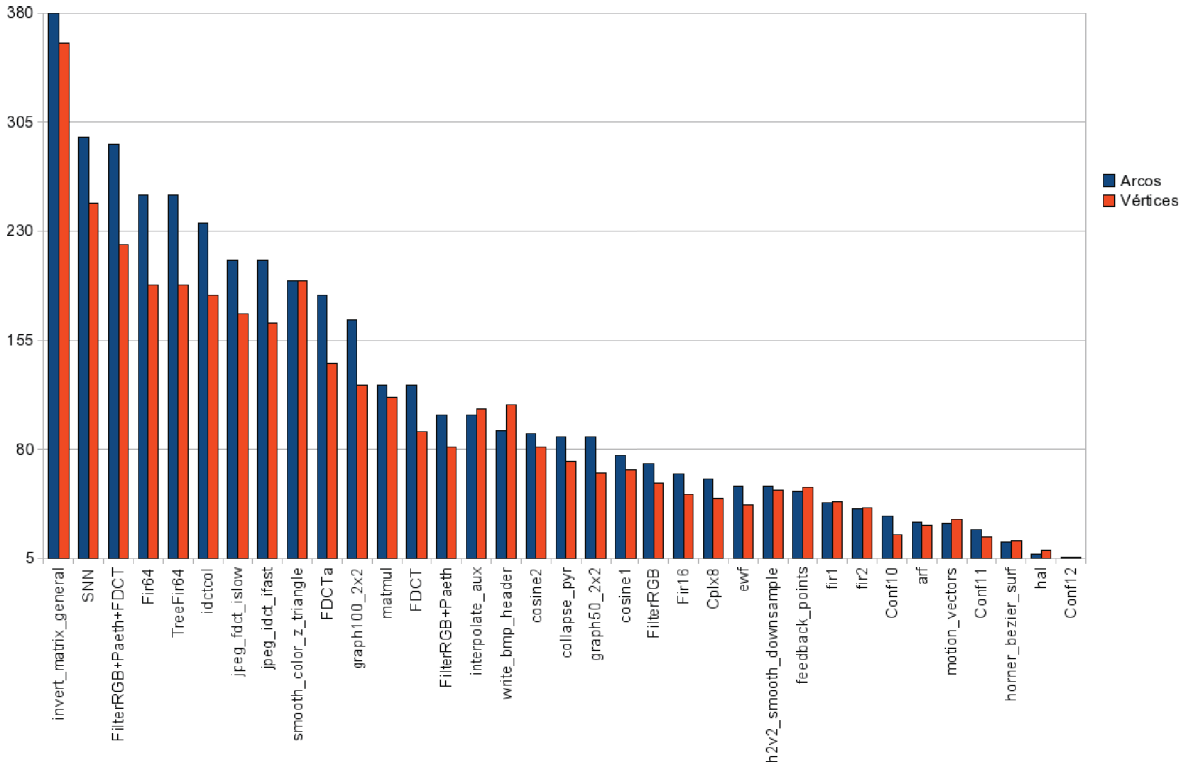


Figura 5.7. Conjunto Total com 35 *benchmarks*.

Para termos uma referência para a comparação dos resultados, a arquitetura com topologia 0_1_hop foi selecionada. Esta topologia foi apontada por Mei et al. (2005) e Bansal et al. (2004) como a topologia mais adequada para grafos de aplicações semelhantes aos estudados nesta dissertação. Vale ressaltar que o espaço de busca de variações de arquitetura proposta aqui é bem mais amplo do que o que foi explorado pelos trabalhos anteriores dos autores supracitados.

Após a apresentação e interpretação de cada estudo de caso, é feita uma comparação entre ambos. Os algoritmos utilizados no primeiro e no segundo estudo são diferentes dos algoritmos utilizados no terceiro. Por essas diferenças, optou-se por fazer uma comparação mais detalhada entre o primeiro e o segundo estudo de caso comparando os melhores resultados de número total de segmentos e menor aumento do caminho crítico com esses mesmos valores obtidos da melhor arquitetura do terceiro estudo de caso.

5.4 1º e 2º Estudos de Caso

Para a formulação do primeiro estudo de caso, foi utilizado um subgrupo baseado no conjunto total, chamado grupo grande contendo 16 *benchmarks*. Para ser representativo, foram escolhidos *benchmarks* das três fontes presentes na Tabela 5.3. Extraídos manualmente dos algoritmos de processamento de sinais, os gerados pela ferramenta TGFF e os extraídos da base de aplicações MediaBench, os *benchmarks* desse grupo podem ser visualizados na Tabela 5.4 e Figura 5.8, em que cada *benchmark* é mapeado na mesma arquitetura, e o número total de segmentos na arquitetura necessários para satisfazer a todos os arcos dos *benchmarks* é somado ao total de segmentos dos demais *benchmarks* e a arquitetura que obtiver o menor número total de segmentos, que consiga satisfazer a todos os *benchmarks*, é selecionada como sendo a melhor arquitetura. Avalia-se ainda a arquitetura que consiga aumentar o mínimo possível o caminho crítico em relação ao caminho crítico original do *benchmark* avaliado e destaca-se a heurística que consiga tais arquiteturas no menor tempo possível.

A mesma organização lógica utilizada para a geração do primeiro estudo de caso foi utilizada para a geração do segundo estudo de caso, o grande diferencial entre ambos é o número de *benchmarks* presentes em cada grupo, sendo que para o grupo pequeno apenas 4 *benchmarks* foram selecionados, os quais podem ser vistos em destaque na Tabela 5.4, que contém todos os *benchmarks* que fazem parte dos grupos grande e pequeno.

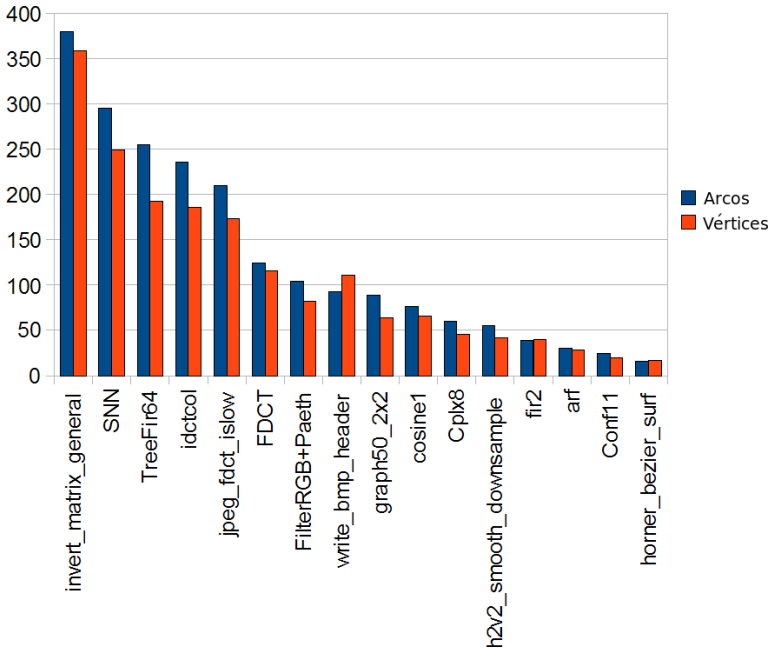


Figura 5.8. Grupo Grande com 16 *benchmarks*.

Tabela 5.4. Conjunto dos *benchmarks* utilizados nos estudos de caso (os *benchmarks* em negrito fazem parte do segundo estudo, apenas o graph50_2x2 faz parte dos dois estudos de caso)

Benchmarks	Arcos	Vértices
arf	30	28
cosine1	76	66
feedback points	**51**	**54**
FIR2	39	40
h2v2 smooth downsample	55	52
horner bezier surf	16	17
idctcol	236	186
interpolate aux	**104**	**108**
invert matrix general	380	359
jpeg fdct islow	210	173
write bmp header	93	111
FIR16	**63**	**49**
Cplx8	60	46
FDCT	124	92
FilterRGB+Paeth	104	82
SNN	295	249
TreeFIR64	255	193
graph50_2x2	***89***	***64***
Conf11	25	20

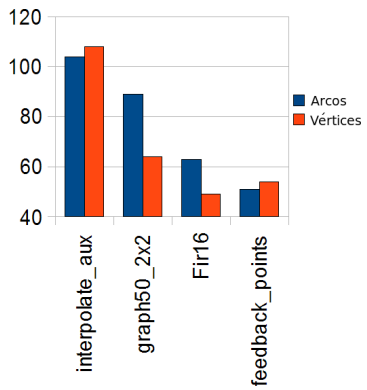


Figura 5.9. Grupo Pequeno com 4 *benchmarks*.

O processo de geração dos resultados seguiu a lógica definida na Figura 5.6, ou seja, para cada heurística avaliada a ferramenta foi treinada para um grupo de *benchmarks*, no caso do grupo grande a ferramenta foi treinada mapeando cada uma das arquiteturas em cada um dos 16 *benchmarks* e para o grupo pequeno em cada um dos 4 *benchmarks*, fornecendo no final do processo a melhor arquitetura para o conjunto avaliado. Posteriormente essa arquitetura obtida foi utilizada para o mapeamento do conjunto total de *benchmarks*, sendo os valores obtidos desse mapeamento no grupo total utilizados nas comparações a seguir.

Os gráficos que representam essa comparação foram obtidos por meio do número total de segmentos necessários ao mapeamento de cada um dos 35 *benchmarks* do grupo total (Tabela 5.3). Cada gráfico está dividio em três subgráficos para facilitar a comparação visual entre os totais de segmentos obtidos pelos *benchmarks* (que variaram de 6 a 897 segmentos).

5.4.1 Algoritmo Genético

Duas variações de cruzamento do Algoritmo Genético foram aplicadas a fim de obter uma comparação qualitativa entre ambas. Os AGs tiveram os mesmos valores para os parâmetros descritos na Tabela 5.5.

Tabela 5.5. Parâmetros utilizados pelo Algoritmo Genético

Parâmetro	Valor
Conexões por PE	8
Indivíduos na população inicial	100
Total de gerações	100
Taxa de renovação da população [1]	20%
Probabilidade de mutação	0.5%
Tipo de fitness	total de segmentos
Tipo de crossover	variável [2]

Os três subgráficos da Figura 5.10a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.10b representam os resultados obtidos do treinamento no grupo pequeno, sendo que para o grupo grande a heurística AG com

[1] Sendo a geração de 20% de novos indivíduos a partir dos 20% melhores, com o descarte dos 20% piores

[2] Podendo ser por nó ou por PR

crossover por PR ficou 14,31% melhor que 0_1_hop e para o grupo pequeno a heurística AG com *crossover* por nó ficou 18,31% melhor que 0_1_hop.

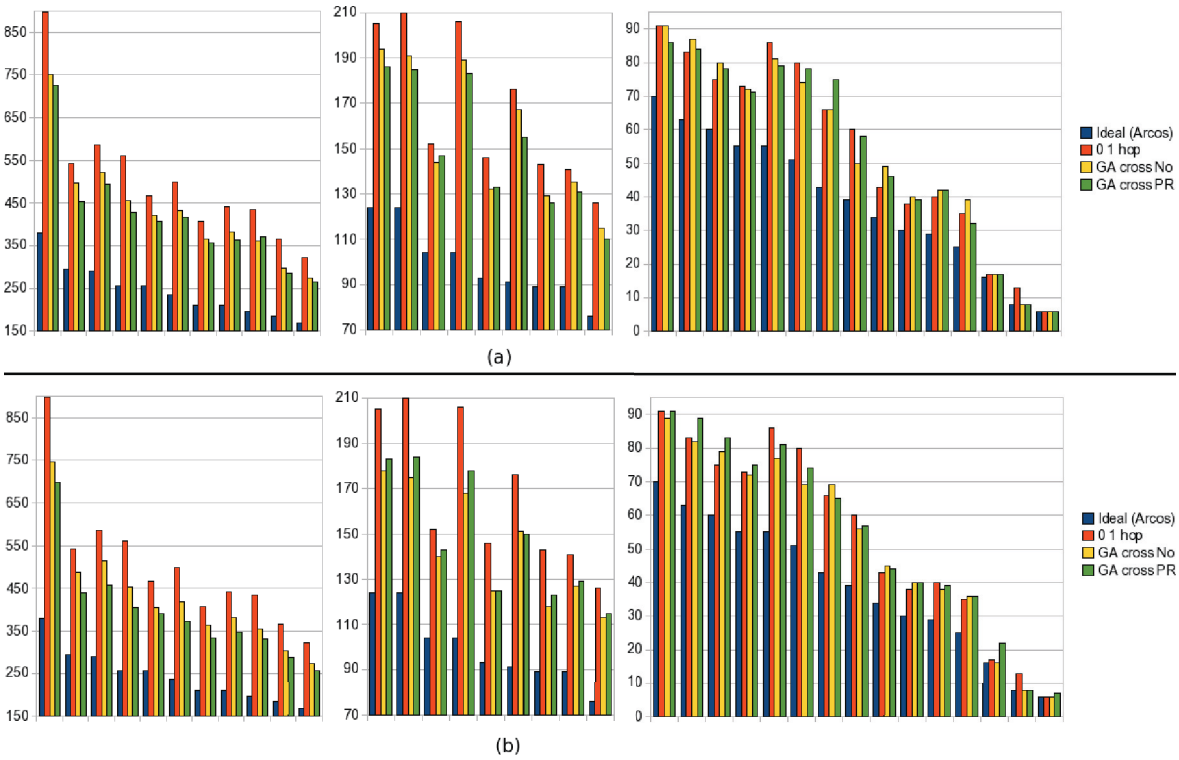


Figura 5.10. Algoritmo Genético: (a) Grupo Grande, a heurística GA *crossover* por PR ficou 61,54% acima do número mínimo possível de arcos; (b) Grupo Pequeno, a heurística GA cross nó ficou 53,99% acima do número mínimo possível de arcos.

5.4.2 Path Relinking

O Algoritmo *Path Relinking* foi aplicado aos dois melhores conjuntos de indivíduos obtidos da aplicação do AG sobre uma população inicial, sendo o primeiro conjunto composto pelos 10 melhores indivíduos obtidos da aplicação do AG com crossover por nó e o segundo pelos 10 melhores indivíduos obtidos da aplicação do AG com crossover utilizando PR.

Os três subgráficos da Figura 5.11a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.11b, representam os resultados obtidos do treinamento no grupo pequeno, sendo que para o grupo grande, a heurística PR AG *crossover* PR ficou 15,30% melhor que 0_1_hop e para o grupo pequeno, a heurística PR AG *crossover* PR ficou 18,20% melhor que 0_1_hop.

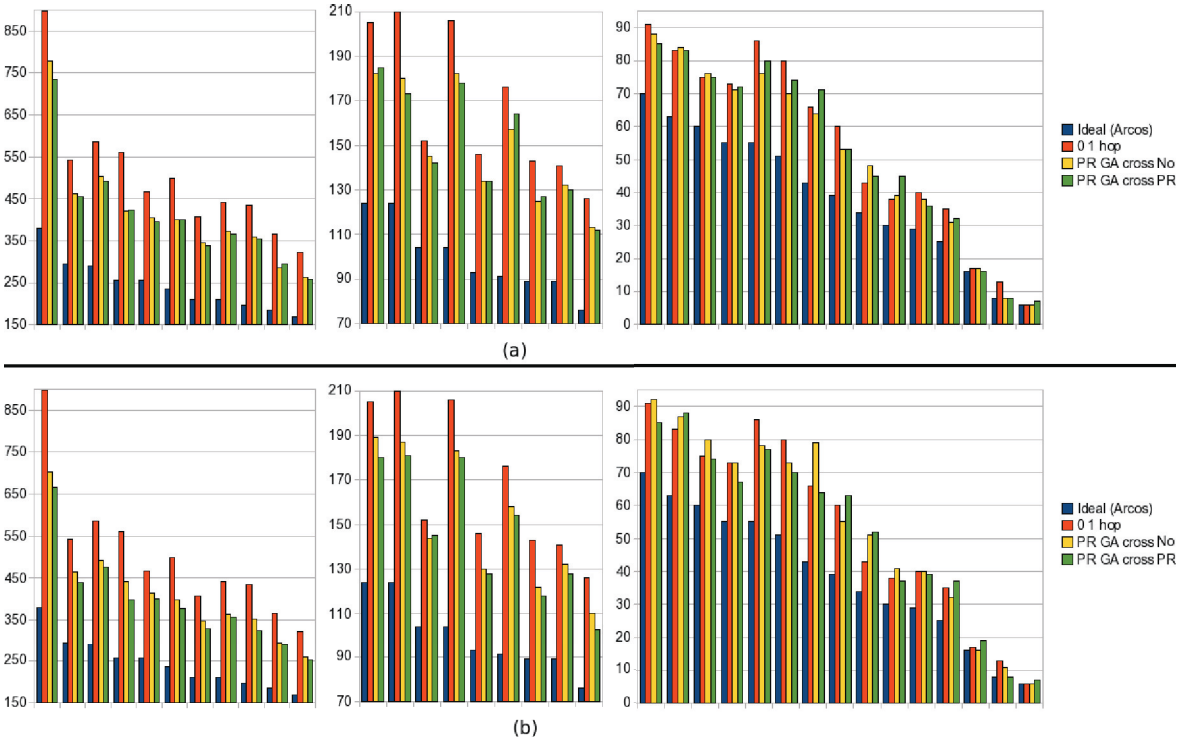


Figura 5.11. Path Relinking: (a) Grupo Grande, a heurística PR GA *crossover* PR ficou 59,66% acima do número mínimo possível de arcos; (b) Grupo Pequeno, a heurística PR GA *crossover* PR ficou 54,21% acima do número mínimo possível de arcos.

5.4.3 Simulated Annealing

O Algoritmo de *Simulated Annealing* foi aplicado a duas arquiteturas obtidas pela aplicação de outras heurísticas. A primeira arquitetura escolhida foi obtida pela aplicação do AG com *crossover* por nó a uma população inicial, a segunda arquitetura é o resultado da aplicação do PR aos 10 melhores indivíduos obtidos com a aplicação do AG com *crossover* por nó sobre uma população inicial.

Os três subgráficos da Figura 5.12a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.12b representam os resultados obtidos do treinamento no grupo pequeno, sendo que, para o grupo grande, a heurística SA AG *crossover* por nó ficou 18,30% melhor que 0_1_hop e para o grupo pequeno, a heurística SA AG *crossover* por nó ficou 17,29% melhor que 0_1_hop.

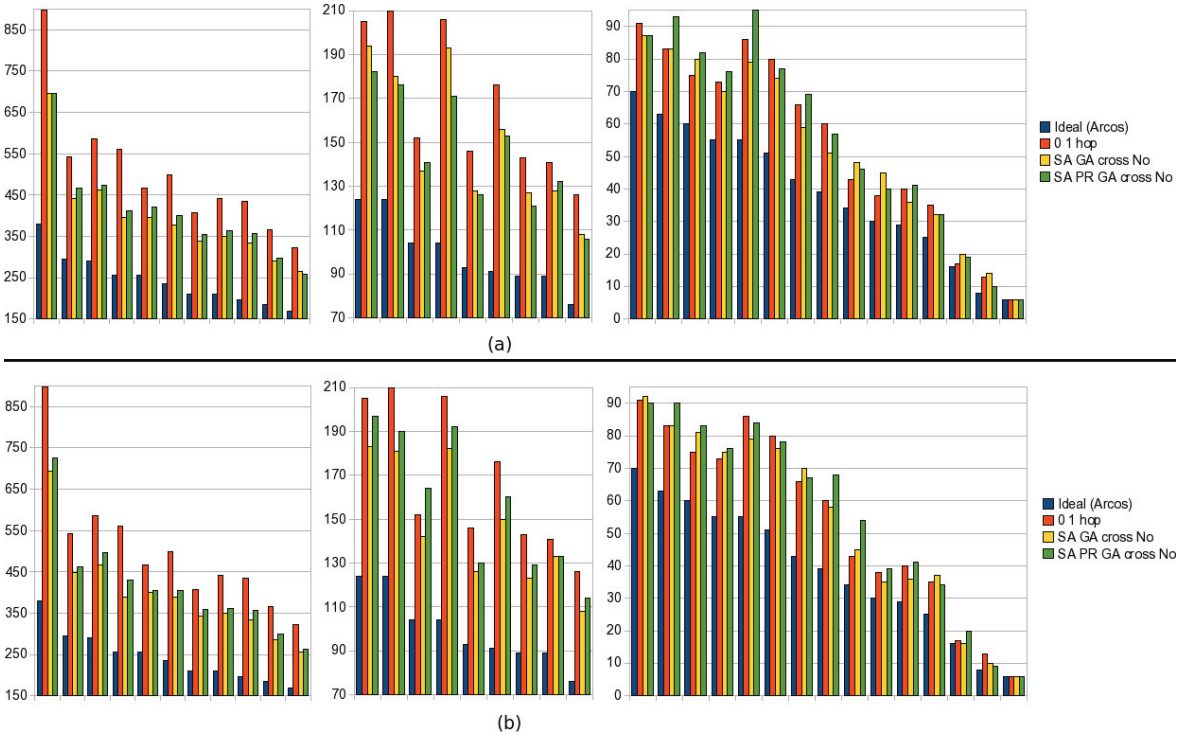


Figura 5.12. Simulated Annealing: (a) Grupo Grande, a heurística SA GA *crossover* nó ficou 54,01% acima do número mínimo possível de arcos; (b) Grupo Pequeno, a heurística SA GA *crossover* nó ficou 55,91% acima do número mínimo possível de arcos.

O mesmo Algoritmo de SA foi aplicado a duas arquiteturas obtidas pela aplicação do algoritmo de escalonamento gráfico, sendo que em uma variação foi considerado o percentual real obtido pelo histograma do algoritmo de escalonamento e em uma segunda variação foi limitado a 50% o número total de segmentos com o mesmo número de saltos.

Essa implementação foi desenvolvida considerando a tendência apontada na Seção 5.5.

A aplicação do Algoritmo de SA nas arquiteturas extraídas com os algoritmos de escalonamento gráfico se deu seguindo o mesmo processo descrito na seção 5.4.

Os três subgráficos da Figura 5.13a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.13b representam os resultados obtidos do treinamento no grupo pequeno, sendo que, para o grupo grande, a heurística SA ASAP ficou 10,57% melhor que 0_1_hop e para o grupo pequeno, a heurística SA ALAP limitado ficou 16,44% melhor que 0_1_hop.

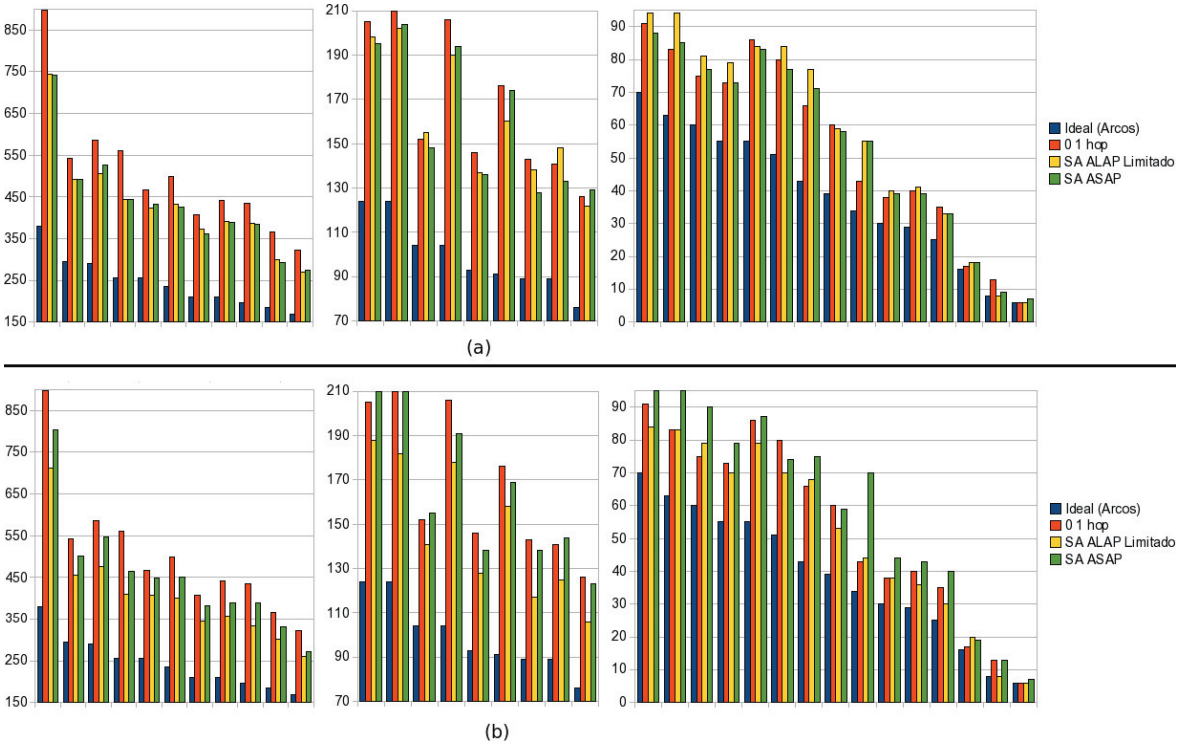


Figura 5.13. Simulated Annealing: (a) Grupo Grande, a heurística SA ASAP ficou 68,58% acima do número mínimo possível de arcos; (b) Grupo Pequeno, a heurística SA ALAP limitado ficou 57,52% acima do número mínimo possível de arcos.

5.4.4 Comparativo entre as heurísticas no 1º estudo de caso

Para os resultados a seguir as melhores topologias obtidas com as heurísticas treinadas para o grupo grande foram mapeadas em todos os *benchmarks* do grupo total, porém no Anexo B é demonstrado como cada uma das topologias encontradas surgiram.

O total de segmentos obtidos no mapeamento de cada uma das topologias encontradas é comparado com o total de segmentos obtidos com o mapeamento na topologia 0_1_hop e em relação ao número total de arcos dos *benchmarks* do grupo total, esse comparativo pode ser visualizado na Tabela 5.6, assim como o número total de segmentos presentes nos caminhos críticos de todos os *benchmarks* do grupo total. E na Figura 5.14 pode ser visualizado o gráfico comparativo com esses dois parâmetros.

Tabela 5.6. Total de Segmentos e Segmentos de caminho crítico obtidos, com as heurísticas de busca implementadas, em relação ao grupo grande

Topologia Obtida com:	Total de Segmentos	Relativo 0_1_hop	Relativo Arcos	Caminho Crítico	Relativo 0_1_hop	Relativo Arcos
Ideal (total arcos)	4160	-46,95%	0,00%	598	0,00%	0,00%
0 1 hop	7842	0,00%	88,51%	891	0,00%	0,00%
GA cross No	6957	-11,29%	67,24%	821	-7,86%	37,29%
GA cross PR	6720	-14,31%	61,54%	799	-10,33%	33,61%
PR GA cross No	6718	-14,33%	61,49%	805	-9,65%	34,62%
PR GA cross PR	6642	-15,30%	59,66%	808	-9,32%	35,12%
SA ALAP Limitado	7066	-9,90%	69,86%	856	-3,93%	43,14%
SA ASAP	7013	-10,57%	68,58%	816	-8,42%	36,45%
SA GA cross No	**6407**	**-18,30%**	**54,01%**	**797**	**-10,55%**	**33,28%**
SA PR GA cross No	6672	-14,92%	60,38%	823	-7,63%	37,63%

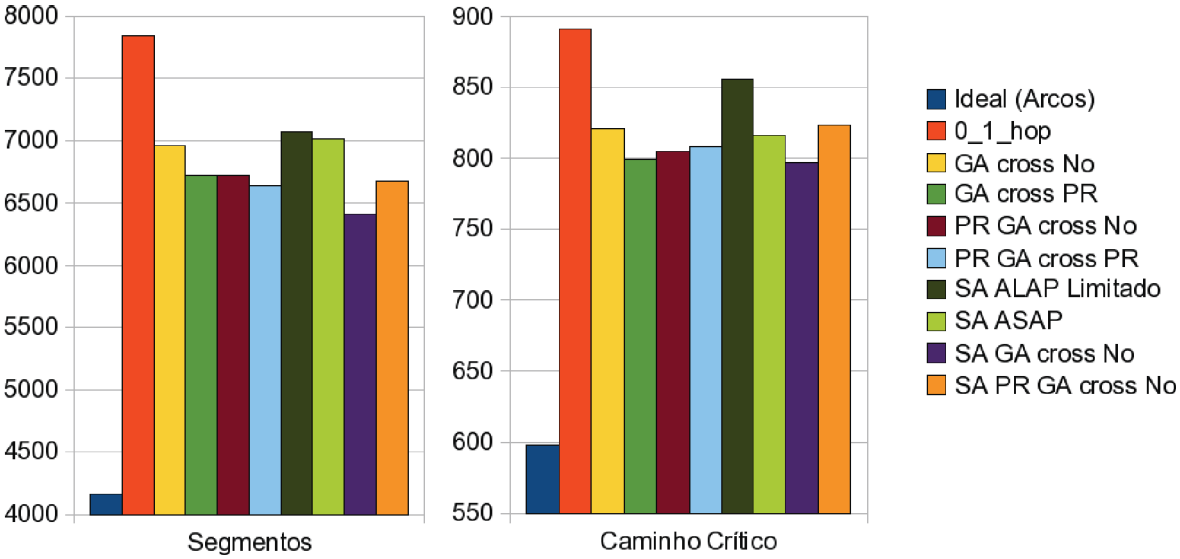


Figura 5.14. Gráficos com as variações do número total de segmentos e do número de segmentos dos caminhos críticos.

A partir dos dados da Tabela 5.6 e Figura 5.14, podemos conferir que a heurística SA aplicada a melhor topologia extraída com o algoritmo AG com *crossover* por nó obteve o melhor resultado em relação ao número total de segmentos e em relação ao aumento do caminho crítico, com isso essa combinação heurística obteve o melhor resultado geral.

5.4.5 Comparativo entre as heurísticas no 2º estudo de caso

Para os resultados a seguir as melhores topologias obtidas com as heurísticas treinadas para o grupo pequeno foram mapeadas em todos os *benchmarks* do grupo total, porém no Anexo B é demonstrado como cada uma das topologias encontradas surgiram.

Tabela 5.7. Total de Segmentos e Segmentos de caminho crítico obtidos, com as heurísticas de busca implementadas, em relação ao grupo pequeno

Topologia Obtida com:	Total de Segmentos	Relativo 0_1_hop	Relativo Arcos	Caminho Crítico	Relativo 0_1_hop	Relativo Arcos
Ideal (total arcos)	4160	-46,95%	0,00%	598	-32,88%	0,00%
0 1 hop	7842	0,00%	88,51%	891	0,00%	49,00%
GA cross No	**6406**	**-18,31%**	**53,99%**	809	-9,20%	35,28%
GA cross PR	6563	-16,31%	57,76%	805	-9,65%	34,62%
PR GA cross No	6699	-14,58%	61,03%	837	-6,06%	39,97%
PR GA cross PR	6415	-18,20%	54,21%	798	-10,44%	33,44%
SA ALAP Limitado	6553	-16,44%	57,52%	**789**	**-11,45%**	**31,94%**
SA ASAP	7369	-6,03%	77,14%	853	-4,26%	42,64%
SA GA cross No	6486	-17,29%	55,91%	794	-10,89%	32,78%
SA PR GA cross No	6811	-13,15%	63,73%	837	-6,06%	39,97%

O total de segmentos obtidos no mapeamento de cada uma das topologias encontradas é comparado com o total de segmentos obtidos com o mapeamento na topologia 0_1_hop e em relação ao número total de arcos dos *benchmarks* do grupo total, esse comparativo pode ser visualizado na Tabela 5.7, assim como o número total de segmentos presentes nos caminhos críticos de todos os *benchmarks* do grupo total. E na Figura 5.15 pode ser visualizado o gráfico comparativo com esses dois parâmetros.

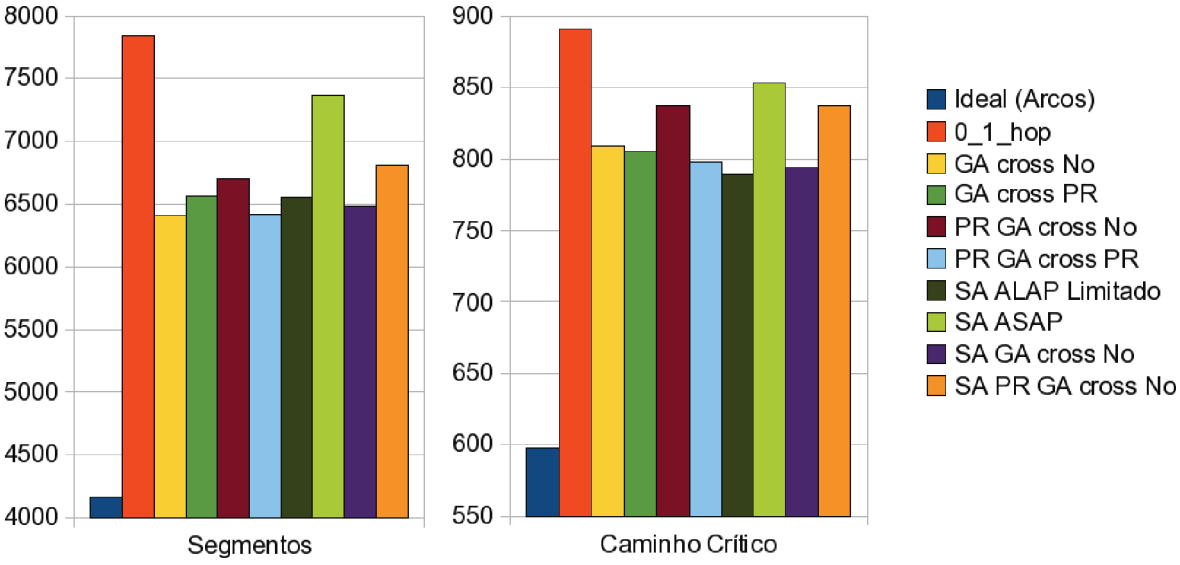


Figura 5.15. Gráficos com as variações do número total de segmentos e do número de segmentos dos caminhos críticos.

A partir dos dados da Tabela 5.7 e Figura 5.15, podemos observar que a heurística AG com *crossover* por nó obteve o melhor resultado em relação ao menor número de segmentos obtidos, quanto ao menor aumento do caminho crítico, porém, para

esse estudo de caso, a heurística SA aplicada a topologia obtida com a aplicação do Algoritmo ALAP limitado foi melhor.

5.5 3º Estudo de Caso – Escalonamento

Para a formulação desse estudo de caso, os algoritmos de escalonamento gráfico foram aplicados aos *benchmarks* do grupo total (Tabela 5.3) obtendo-se um histograma com as distâncias de cada um dos vértices de cada *benchmark*. A partir desse histograma foram geradas topologias que continham em sua composição segmentos com os tamanhos relativos a tais distâncias. Essas topologias foram construídas com 8 segmentos por nó, no entanto, observou-se uma tendência a distâncias iguais a 1 (0_hop), que geraram arquiteturas com 6 ou 7 segmentos com comprimento 1.

Devido a essa tendência optou-se por fixar um limitador à quantidade de segmentos com o mesmo comprimento, chegando ao limite de 50%, favorecendo assim a construção de topologias com segmentos maiores e, como pode ser observado adiante, essa decisão acabou por gerar arquiteturas melhores.

Como exemplo, podemos citar o caso em que para a primeira variação o escalonamento apontasse para 80% dos segmentos sendo 0_hop, 15%, sendo 1_hop, 3% 2_hop, 1% 3_hop e 1% 4_hop só seria possível a construção de uma arquitetura contendo 0_hop e 1_hop, porém limitando-se a 50% o número máximo de segmentos. Com o mesmo número de saltos seria possível a construção de uma arquitetura com 0, 1 e 2_hops, notando-se que o número de segmentos com 0_hop variava de 75% a 100% nos menores *benchmarks*, porém conforme será demonstrado a seguir as melhores arquiteturas eram aquelas que possuíam o melhor equilíbrio entre segmentos entre 0_hop e 4_hop.

5.5.1 ASAP

Duas variações do algoritmo ASAP (sem limitação e com limitação), foram aplicadas a fim de se obter uma comparação qualitativa entre ambas. Elas extraíram topologias de um mesmo conjunto de *benchmarks*, o conjunto total. A topologia da arquitetura extraída com a primeira variação pode ser visualizada na Tabela 5.8, já a Tabela 5.9 contém a topologia da arquitetura extraída com a segunda variação do algoritmo ASAP.

Os três subgráficos da Figura 5.16 representam os resultados obtidos com o mapeamento da arquitetura extraída com as variações do algortimo ASAP e mapeadas no grupo total, sendo que a variação que obteve o melhor resultado foi ASAP Limitado que ficou 18,45%, melhor que 0_1_hop.

Tabela 5.8. Distribuição dos comprimentos dos segmentos obtidos com ASAP

	Ligações	1 hop	2 hops	3 hops	4 hops	5 hops
percentual	100.00%	81,66%	7,475%	6,279%	1,816%	2,759%
quantidade [3]	8	6,533	0,598	0,502	0,145	0,220
real [4]	8	6	1	1	0	0

Tabela 5.9. Distribuição dos comprimentos dos segmentos obtidos com ASAP Limitado

	Ligações	1 hop	2 hops	3 hops	4 hops	5 hops
percentual	100.00%	50,00%	21,79%	16,91%	4,717%	6,566%
quantidade [5]	8	4	1,743	1,353	0,377	0,525
real [6]	8	4	2	2	0	0

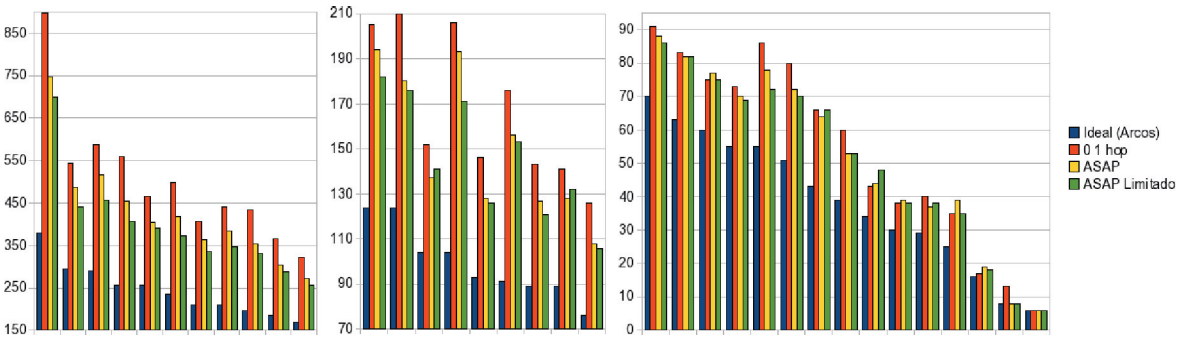


Figura 5.16. ASAP: a heurística ASAP Limitado ficou 53,73% acima do número mínimo possível de arcos.

5.5.2 ALAP

Duas variações do algoritmo ALAP (sem limitação e com limitação), foram aplicadas a fim de se obter uma comparação qualitativa entre ambas. Elas extraíram arquiteturas de um mesmo conjunto de *benchmarks*, o conjunto total. A topologia da arquitetura

[3] Sugerida pelo histograma
[4] Adotada pelo Algoritmo de arredondamento
[5] Idem a nota 3
[6] Idem a nota 4

extraída com a primeira variação pode ser visualizada na Tabela 5.10, já a Tabela 5.11 contém a topologia da arquitetura extraída com a segunda variação do Algoritmo ALAP.

Os três subgráficos da Figura 5.17 representam os resultados obtidos com o mapeamento da arquitetura extraída com as variações do Algortimo ALAP e mapeadas no grupo total, sendo que a variação que obteve o melhor resultado foi ALAP Limitado que ficou 18,38%, melhor que 0_1_hop.

Tabela 5.10. Distribuição dos comprimentos dos segmentos obtidos com ALAP

	Ligações	1 hop	2 hops	3 hops	4 hops	5 hops
percentual	100.00%	88,78%	1,79%	6,066%	2,008%	1,349%
quantidade [7]	8	7,102	0,143	0,485	0,160	0,107
real [8]	8	7	1	0	0	0

Tabela 5.11. Distribuição dos comprimentos dos segmentos obtidos com ALAP Limitado

	Ligações	1 hop	2 hops	3 hops	4 hops	5 hops
percentual	100.00%	50,00%	6,912%	27,70%	9,525%	5,86%
quantidade [9]	8	4	0,553	2,216	0,762	0,468
real [10]	8	4	1	3	0	0

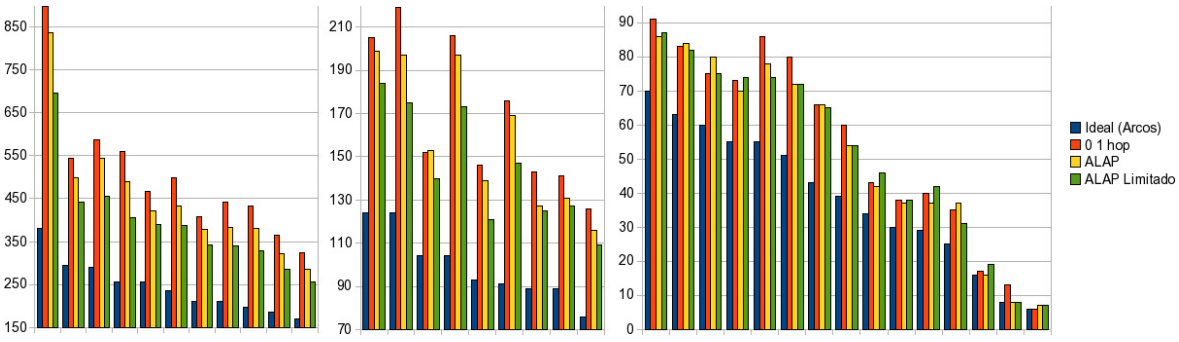


Figura 5.17. ALAP: a heurística ALAP Limitado ficou 53,87% acima do número mínimo possível de arcos.

[7] Idem a nota 3
[8] Idem a nota 4
[9] Idem a nota 3
[10] Idem a nota 4

Tabela 5.12. Total de Segmentos e Segmentos de caminho crítico obtidos, com os Algoritmos de Escalonamento

Topologia Obtida com:	Total de Segmentos	Relativo 0_1_hop	Relativo Arcos	Caminho Crítico	Relativo 0_1_hop	Relativo Arcos
Ideal (total arcos)	4160	-46,95%	0,00%	598	-32,88%	0,00%
0 1 hop	7842	0,00%	88,51%	891	0,00%	49,00%
ALAP	7176	-8,49%	72,50%	849	-4,71%	41,97%
ALAP Limitado	6401	-18,38%	53,87%	796	-10,66%	33,11%
ASAP	6830	-12,90%	64,18%	804	-9,76%	34,45%
ASAP Limitado	**6395**	**-18,45%**	**53,73%**	**779**	**-12,57%**	**30,27%**

5.5.3 Comparativo entre os Algoritmos no 3º estudo de caso

Na Tabela 5.12 e na Figura 5.18, todas as arquiteturas aparecem lado a lado com suas melhoras relativas à arquitetura 0_1_hop, quando consideramos o total de segmentos obtidos e o aumento do tamanho do caminho crítico em relação aos caminhos críticos extraídos dos *benchmarks* avaliados, com isso podemos ter uma visão mais precisa da melhor arquitetura gerada.

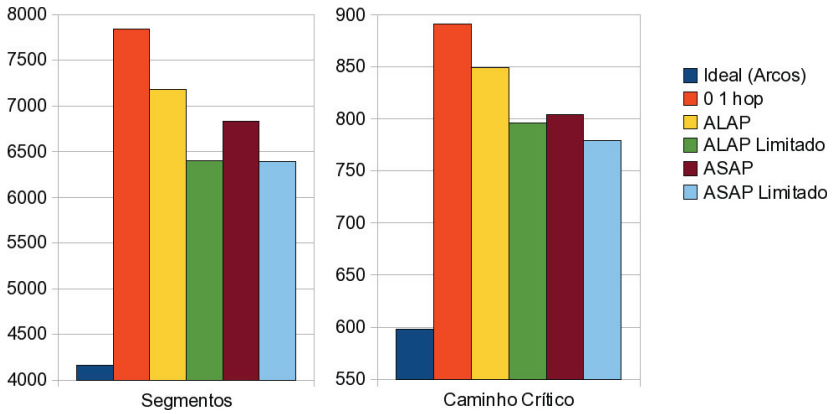


Figura 5.18. Gráficos com as variações do número total de segmentos e do número de segmentos dos caminhos críticos.

Dentre os algoritmos de escalonamento gráfico avaliados, o algoritmo ASAP Limitado conseguiu extrair a melhor arquitetura. Chegou-se a esse resultado depois de mapear a arquitetura extraída nos *benchmarks* da Tabela 5.3.

Tabela 5.13. Comparativo entre os melhores resultados do Total de Segmentos e Segmentos de caminho crítico.

Topologia Obtida com:	Estudo de Caso	Total de Segmentos	Relativo 0_1_hop	Relativo Arcos	Caminho Crítico	Relativo 0_1_hop	Relativo Arcos
Ideal (total arcos)	–	4160	-46,95%	0,00%	598	-32,88%	0,00%
0 1 hop	–	7842	0,00%	88,51%	891	0,00%	49,00%
GA cross No	Grande	6407	-18,30%	54,01%	797	-10,55%	33,28%
GA cross No	Pequeno	6406	-18,31%	53,99%	809	-9,20%	35,28%
SA ALAP Limitado	Pequeno	6553	-16,44%	57,52%	789	-11,45%	31,94%
ASAP Limitado	Escalonamento	**6395**	**-18,45%**	**53,73%**	**779**	**-12,57%**	**30,27%**

5.6 Comparativo entre os estudos de caso

Essa seção apresenta um comparativo entre todos os estudos de caso implementados, esse comparativo pode ser observado na Tabela 5.13, onde os melhores resultados obtidos com cada estudo de caso são colocados lado a lado, e na Figura 5.19 é apresentada a melhor topologia obtida entre todas as topologias de todos os estudos de caso analisados. Já na Figura 5.20 é feito a comparação entre os melhores resultados obtidos em relação aos caminhos críticos.

Como pudemos observar, a arquitetura gerada a partir do histograma obtido da execução do Algoritmo ASAP Limitado obteve o melhor resultado em relação ao número mínimo de segmentos e ao aumento mínimo no caminho crítico sendo, por tanto, a melhor topologia obtida.

A Figura 5.19 representa tal arquitetura, porém temos que considerar que não há uma regra clara que dita onde deve estar cada segmento de determinado tamanho. Essa arquitetura foi gerada a partir dos dados do histograma fornecido pelo algoritmo, porém a disposição de seus segmentos quanto à direção de determinado comprimento é um processo aleátorio.

Na Figura 5.20, podemos observar o gráfico com os melhores caminhos críticos obtidos pelos dois melhores resultados alcançados pelas heurísticas avaliadas e pelo melhor resultado obtido com os algoritmos de escalonamento, que é também o melhor caminho crítico obtido.

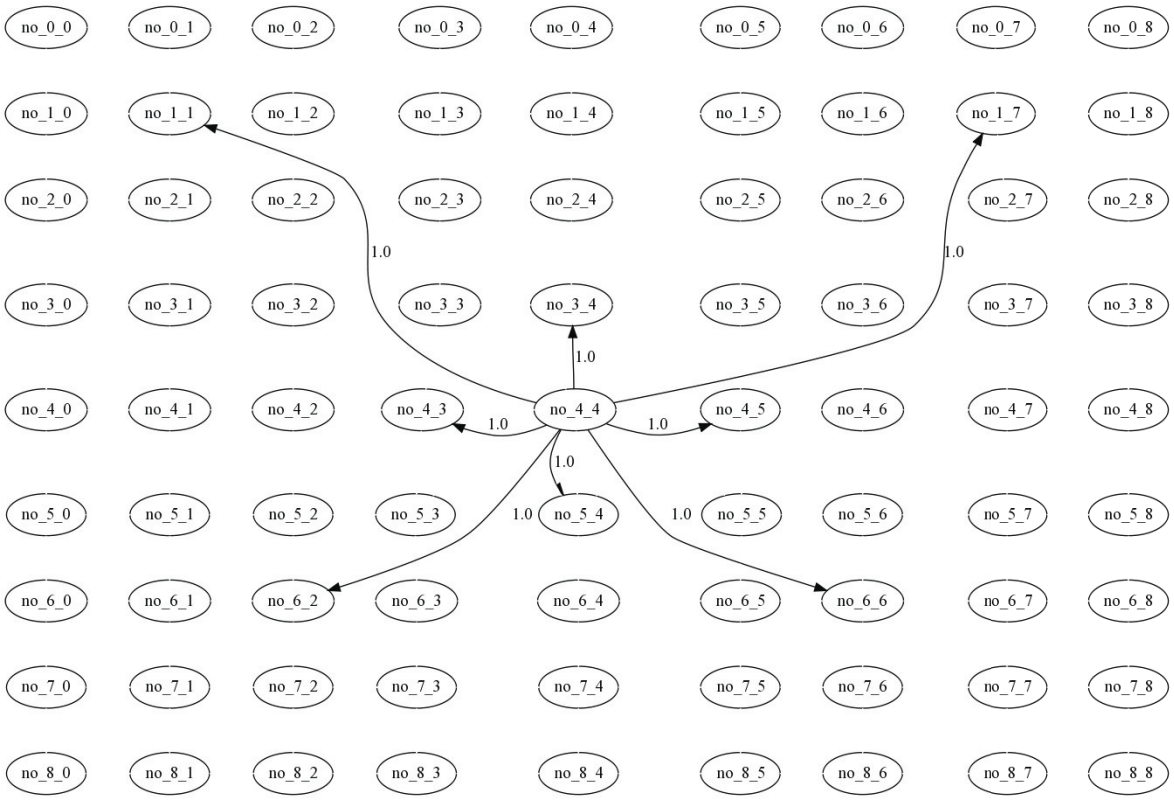


Figura 5.19. Topologia da melhor arquitetura obtida.

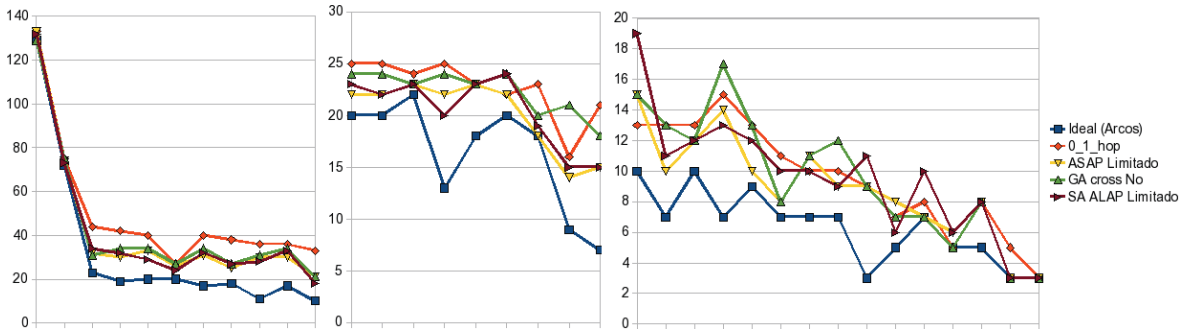


Figura 5.20. Menores aumentos do caminho crítico obtidos.

Capítulo 6

Conclusão

Apesar da tecnologia de integração estar seguindo a lei de Moore, dobrando a cada dezoito meses, o desempenho e a produtividade dos projetos de novas arquiteturas não vem acompanhando. O principal motivo, para tal, é a alta complexidade dos processadores superescalares atuais. A solução avaliada neste trabalho, busca tirar proveito dos avanços tecnológicos e propõe uma solução baseada em arquiteturas paralelas escaláveis compostas por vários elementos de processamento simples. Esta abordagem visa também reduzir a potência dissipada e a energia consumida.

Porém o espaço de projeto de arquiteturas é muito amplo, por isso, neste trabalho o foco foram as aplicações de processamento de sinais e multimídia, sobretudo os algoritmos de compactação de imagens/vídeo, por serem, aplicações que tem uma alta demanda de uso de processamento e ao mesmo tempo um alto paralelismo implícito, alguns exemplos de uso desses algoritmos incluem TV digital, celulares, internet, etc. Este paralelismo pode ser explorado com as arquiteturas paralelas baseadas no modelo de fluxo de dados. Esta abordagem, descentraliza o controle, reduzindo a complexidade das interconexões. Nas implementações atuais, as interconexões consomem uma parcela significativa da área física total utilizada. Neste trabalho, o espaço de solução foi restrito aos modelos bidimensionais com operadores de fluxo de dados, buscando uma topologia de interconexão eficiente.

O principal resultado foi o desenvolvimento de uma ferramenta de busca, que foi utilizada para encontrar soluções que reduzissem o custo das conexões entre os elementos, através da geração automática de topologias. A ferramenta procura gerar automaticamente uma topologia que possa servir de base na construção de uma arquitetura escalável facilitando a implementação em *hardware* de cada aplicação alvo.

Para validar a ferramenta na busca de uma melhor topologia, foram selecionadas aplicações reais na área de multimídia, cujos algoritmos apresentam núcleos com grande demanda computacional e potencial de paralelismo. O grafo do fluxo de dados de cada

núcleo foi extraído, formando um conjunto de grafos que foram usados para avaliar o potencial de cada topologia. Com base em cada topologia obtida a ferramenta gera uma arquitetura, replicando o padrão de conexão obtido da topologia nas ligações entre os elementos, a partir dessa arquitetura, faz o mapeamento do conjunto de grafos das aplicações e avaliada o custo da topologia gerada. O critério de custo foi o número de segmentos necessários para mapear todos os arcos dos grafos de cada aplicação. O caminho crítico e o tempo de processamento das várias técnicas de busca também foi estudado.

Para possibilitar a busca de uma topologia que atendesse aos critérios de escalabilidade das arquiteturas, menor número total de segmentos, e menor aumento possível no caminho crítico, a abordagem deste trabalho foi baseada no uso de heurísticas de busca, como Algoritmos Genéticos, *Simulated Annealing* e *Path Relinking*, tanto de forma isolada quanto em conjunto, e de algoritmos de escalonamento, como ASAP e ALAP. Com essas implementações a ferramenta possibilita várias abordagens para a geração de topologias. O trabalho dessa dissertação foi concentrado em duas abordagens: topologias e aplicações. A abordagem baseada em topologias, gera novas topologias a partir de um conjunto inicial, enquanto a abordagem baseada em aplicações, deriva uma topologia de um conjunto de aplicações.

A primeira abordagem fez uso das heurísticas de busca (Algoritmos Genéticos, *Simulated Annealing* e *Path Relinking*) para a geração automática de topologias a partir de um conjunto inicial de topologias, ou seja, novas topologias são derivadas utilizando a aplicação de heurísticas de busca sobre um conjunto inicial. A seleção da melhor topologia é baseada na minimização do total de segmentos.

Como o foco principal desse trabalho foi a forma como os elementos eram conectados entre si. Se fez necessário selecionar um algoritmo de mapeamento. Os critérios para escolha foram o tempo de execução e a flexibilidade. O mapeamento é composto por duas etapas: posicionamento e roteamento. Ambos os problemas são NP-completos e a maioria dos trabalhos propõe soluções baseadas em heurísticas. Neste trabalho, a abordagem proposta em Ferreira et al. (2007) foi escolhida por tratar a descrição da arquitetura como um grafo, permitindo flexibilidade na criação de novas topologias a partir das heurísticas de busca. O algoritmo de roteamento é genérico e oferece duas possibilidades muito utilizadas: *Dijsktra* e *Pathfinder* (Teixeira et al., 2009). Entretanto, vale observar, que a ferramenta de busca irá encontrar a melhor topologia para o algoritmo de mapeado adotado. Outros algoritmos podem ser adicionados futuramente, aumentando o espaço de solução e adicionando novas funções de custo.

A topologia 0_1_hop foi apontada por dois trabalhos correlatos de exploração como a mais apropriada para mapeamento de aplicações semelhante as abordadas

nestes trabalhos Teixeira et al. (2007, 2009). Para validar a abordagem descrita aqui, a arquitetura 0_1_hop foi escolhida como referência.

Dois experimentos foram desenvolvidos dentro dessa abordagem, no primeiro um grupo contendo 16 *benchmarks* foi mapeado em cada arquitetura e a partir do resultado obtido a melhor arquitetura era selecionada, o mesmo processo foi utilizado na seleção da melhor arquitetura do segundo grupo contendo 4 *benchmarks*.

Vale ressaltar que o conjunto de aplicações é muito regular, apesar de representativo, os núcleos de algoritmos de processamento de sinais digitais e multimídia possuem grafos com muita localidade, que favorece arquiteturas com ligações locais, como podemos observar na topologia que obteve o melhor resultado, dentro dessa abordagem, que apresentou 62,5% de suas ligações seguindo o padrão 0_1_hop e apenas 37,5% seguindo o padrão 3_hop, ficando 18,30% melhor que 0_1_hop em relação ao total de segmentos e com um aumento de 33,28% em relação ao caminho crítico mínimo.

A segunda abordagem derivou uma topologia diretamente do conjunto total de aplicações avaliados, usando para isso os algoritmos de escalonamento estudados. Para se ter uma métrica de distância, entre os vértices dos fluxos de dados dos *benchmarks*, e a partir dessa distância determinar o comprimento dos segmentos necessários para interliga-los na arquitetura, os algoritmos de escalonamento ASAP e ALAP foram utilizados. Essa abordagem permitiu uma rápida caracterização de uma topologia e gerou os melhores resultados, reduzindo o número total de segmentos para implementação das aplicações, possibilitando um menor aumento do caminho crítico e um tempo de geração irrelevante, se comparado aos tempos obtidos com a primeira abordagem.

Outro ponto importante é a localidade das ligações presentes nos grafos das aplicações. A maioria possui grafos onde os vértices são conectados localmente e poucas arestas fazem as ligações mais longas, como podemos observar pelos resultados dos histogramas dos algoritmos ASAP e ALAP. Esse fato fica mais evidente na melhor topologia encontrada que apesar de ter 75% das conexões consideradas 0_1_hop e os outros 25% poderem ser considerados 2_hop apresentou um resultado 18,45% melhor que a topologia 0_1_hop em relação ao total de segmentos e com um aumento de 30,27% em relação ao caminho crítico mínimo.

Devido a limitação encontrada no algoritmo de posicionamento utilizado, sugerimos para os próximos trabalhos a utilização de um algoritmo de posicionamento que levasse em consideração fatores como custo variável para a alocação dos elementos de processamento e um posicionamento heterogêneo, onde alguns elementos tem um lugar pré-determinado na arquitetura, como células de memória, multiplexadores entre outros.

Além da sugestão do uso de outros algoritmos de posicionamento e roteamento, sugeriríamos a utilização de outros critérios de custo, como a medição do consumo de energia ou a melhoria alcançada no desempenho da aplicação relacionando com a diminuição do número de segmentos ou a redução do caminho crítico.

Referências Bibliográficas

Austin, T.; Blaauw, D.; Mahlke, S.; Mudge, T.; Chakrabarti, C. & Wolf, W. (2004). Mobile supercomputers. *Computer*, 37(5):81–83.

Bansal, N.; Gupta, S.; Dutt, N.; Nicolau, A. & Gupta, R. (2004). Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, p. 10474, Washington, DC, USA. IEEE Computer Society.

Baumgarte, V.; Ehlers, G.; May, F.; Nückel, A.; Vorbach, M. & Weinhardt, M. (2003). Pact xpp-a self-reconfigurable data processing architecture. *J. Supercomput.*, 26(2):167–184.

Bossuet, L.; Gogniat, G. & Philippe, J. L. (2003). Fast design space exploration method for reconfigurable architectures. In *Engineering of Reconfigurable Systems and Algorithms*, pp. 65–71.

Budiu, M.; Artigas, P. & Goldstein, S. (2005). Dataflow: A complement to superscalar. *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pp. 177–186.

Cerný, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation. *Journal of Optimization Theory and Applications*, 45(1):41–51.

Chen, D. & Rabaey, J. (1990). Paddi: Programmable arithmetic devices for digital signal processing. In *VLSI Signal Processing IV*. IEEE Press.

Chen, D. C. & Rabaey, J. M. (1992). A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. In *IEEE J. Solid-State Circuits*, volume 27.

Cronquist, D. C.; Fisher, C.; Figueroa, M.; Franklin, P. & Ebeling, C. (1999). Architecture design of reconfigurable pipelined datapaths. In *ARVLSI '99: Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, p. 23, Washington, DC, USA. IEEE Computer Society.

DeBenedictis, E. P. (2004). Will moore's law be sufficient? In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 45, Washington, DC, USA. IEEE Computer Society.

Dick, R. P.; Rhodes, D. L. & Wolf, W. (1998). Tgff: task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pp. 97–101, Washington, DC, USA. IEEE Computer Society.

Ebeling, C.; Cronquist, D. C. & Franklin, P. (1996). Rapid - reconfigurable pipelined datapath. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126–135, London, UK. Springer-Verlag.

Extensible, P. & Group, R. E. S. (2008). Benchmarks.

Ferreira, R.; Cardoso, J. M. P. & Neto, H. C. (2004). Field programmable logic and application. volume 3203/2004 of *Lecture Notes in Computer Science*, pp. 1022–1026. Springer, Berlin / Heidelberg.

Ferreira, R.; Garcia, A.; Teixeira, T. & Cardoso, J. M. P. (2007). A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures. volume 00, pp. 61–66, Los Alamitos, CA, USA. IEEE Computer Society.

Gansner, E. R.; Koutsofios, E.; North, S. C. & Vo, K.-P. (1993). A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230.

Glover, F. W. (1997). Tabu search and adaptive memory programing - advances, applications and challenges. In *Interfaces in computer science and operations research: advances in metaheuristics, optimization, and stochastic modeling technologies*. Springer.

Guerra, L. D. (2008). Array de processadores com grão grosso e fluxo de dados. Technical report, Universidade Federal de Viçosa, Centro de Ciências Exatas e Tecnológicas, Departamento de Informática.

Hartenstein, R. (2001a). Coarse grain reconfigurable architecture (embedded tutorial). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pp. 564–570, New York, NY, USA. ACM.

Hartenstein, R. (2001b). A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pp. 642–649, Piscataway, NJ, USA. IEEE Press.

Hartenstein, R.; Herz, M.; Hoffmann, T. & Nageldinger, U. (2000a). Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, pp. 163–168, New York, NY, USA. ACM Press.

Hartenstein, R.; Kress, R. & Reinig, H. (1994). A dynamically reconfigurable wavefront array architecture for evaluation of expressions. *Application Specific Array Processors, 1994. Proceedings., International Conference on*, pp. 404–414.

Hartenstein, R. W.; Herz, M.; Hoffmann, T. & Nageldinger, U. (2000b). Generation of design suggestions for coarse-grain reconfigurable architectures. In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pp. 389–399, London, UK. Springer-Verlag.

Hauck, S.; Compton, K.; Eguro, K.; Holland, M.; Phillips, S. & Sharma, A. (2008). Totem: Domain-specific reconfigurable logic.

Huang, Z. & Malik, S. (2001). Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *DATE 2001*.

Huang, Z.; Malik, S.; Moreano, N. & Araujo, G. (2004). The design of dynamically reconfigurable datapath coprocessors. *Trans. on Embedded Computing Sys.*, 3(2):361–384.

Intel (2008a). From a few cores to many: A tera-scale computing research overview. Technical report, Intel.

Intel (2008b). Moore's law: Made real by intel innovation. Technical report, Intel.

Kirkpatrick, S.; Gelatt, C. D. & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220, 4598(4598):671–680.

McMurchie, L. & Ebeling, C. (1995). Pathfinder: a negotiation-based performance-driven router for fpgas. In *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pp. 111–117, New York, NY, USA. ACM.

Mei, B.; Lambrechts, A.; Verkest, D.; Mignolet, J.-Y. & Lauwereins, R. (2005). Architecture exploration for a reconfigurable architecture template. *IEEE Des. Test*, 22(2):90–101.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Moritz, C. A.; Yeung, D. & Agarwal, A. (1998). Exploring optimal cost-performance designs for raw microprocessors. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, p. 12, Washington, DC, USA. IEEE Computer Society.

Nageldinger, U. (2001). *Coarse-Grained Reconfigurable Architecture Design Space Exploration*. PhD thesis, University of Kaiserslautern CS department (Informatik).

Patterson, D. A. (2006). New directions for cacm? *Commun. ACM*, 49(1):33–35.

Pimentel, B. L. (2007). Array de processadores com grão grosso e fluxo de dados. Technical report, Universidade Federal de Viçosa, Centro de Ciências Exatas e Tecnológicas, Departamento de Informática.

Rabaey, J. (1997). Reconfigurable processing: The solution to low-power programmable dsp. In *ICASSP '97: Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97)*, volume 1, p. 275, Washington, DC, USA. IEEE Computer Society.

Ribeiro, A. A. L. (2002). Reconfigurabilidade dinâmica e remota de fpgas. Master's thesis, Universidade de São Paulo - Instituto de Ciências Matemáticas e de Computação.

Robinett, W.; Snider, G. S.; Kuekes, P. J. & Williams, R. S. (2007). Computing with a trillion crummy components. *Commun. ACM*, 50(9):35–39.

Russell, S. & Norvig, P. (2004). *Inteligência Artificial*. Elsevier Editora Ltda, Rio de Janeiro - Brasil, 2 edição.

Taghavi, T.; Ghiasi, S.; Ranjan, A.; Raje, S. & Sarrafzadeh, M. (2004). Innovate or perish: Fpga physical design. In *ISPD '04: Proceedings of the 2004 international symposium on Physical design*, pp. 148–155, New York, NY, USA. ACM.

Teixeira, T.; Ferreira, R. & Cardoso, J. M. P. (2009). Explorando com meta-heurísticas o espaço de projeto de arquiteturas reconfiguráveis de grão grosso. In *V Jornadas sobre Sistemas Reconfiguráveis*, pp. 111 – 118. Actas das V Jornadas sobre Sistemas Reconfiguráveis, Lisboa.

Teixeira, T.; Luppi, B. & Ferreira, R. (2007). Automatic architecture generation for coarse-grained reconfigurable array. In *7º Microeletronics Student Forum*. Rio de Janeiro.

Tessier, R. & Burleson, W. (2001). Reconfigurable computing for digital signal processing: A survey. *J. VLSI Signal Process. Syst.*, 28(1-2):7–27.

Trimberger, S. M. (1994). *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, MA, USA.

Waingold, E.; Taylor, M.; Sarkar, V.; Lee, V.; Lee, W.; Kim, J.; Frank, M.; Finch, P.; Devabhaktumi, S.; Barua, R.; Babb, J.; Amarsinghe, S. & Agarwal, A. (1997). Baring it all to software: The raw machine. Technical report, Cambridge, MA, USA.

Wawrzynek, J.; Patterson, D.; Oskin, M.; Lu, S.-L.; Kozyrakis, C.; Hoe, J. C.; Chiou, D. & Asanovic, K. (2007). Ramp: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57.

Wu, K.-C. & Tsai, Y.-W. (2004). Structured asic, evolution or revolution? In *ISPD '04: Proceedings of the 2004 international symposium on Physical design*, pp. 103–106, New York, NY, USA. ACM.

Yeung, A. K. W. & Rabaey, J. (1993). A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput dsp algorithms. In *Proc. HICSS-26*, Kauai, Hawaii.

Youssef, H.; Sait, S. M. & Adiche, H. (2001). Evolutionary algorithms, simulated annealing and tabu search: a comparative study. *Engineering Applications of Artificial Intelligence*, 14(2):167–181.

Anexo A

Implementação do Gerador de Arquiteturas

Como no gerador automático de arquiteturas, definido no capítulo 4, essa ferramenta possibilita a entrada dos padrões de conexões das arquiteturas no formato XML e gera as arquiteturas de saída, consideradas as melhores, também em XML, possibilitando que a saída de uma heurística, ou de um algoritmo de escalonamento, seja a entrada de uma outra heurística, ou a realimentação da mesma heurística. Para tal foi implementado um módulo que possibilita que o usuário defina a ordem de execução das heurísticas, que serão executadas seguindo os parâmetros armazenados anteriormente, definidos pelo próprio usuário.

Como a implementação da ferramenta foi feita na linguagem de programação java, ela funciona independentemente da plataforma, podendo ser executada tanto em sistemas operacionais distintos quanto em ambientes distintos, como desktops ou browsers, via *Java Web Start* (JWS), porém, quando executada em ambiente Linux, tem a possibilidade de criação automática de gráficos e grafos, desde que no sistema haja alguns softwares de terceiros, como gnuplot, graphviz (Gansner et al., 1993) e TGFF (Dick et al., 1998). Será criada uma página na internet para a ferramenta com links para esses softwares (todos distribuídos de forma livre).

A seguir, apresentamos duas divisões dos tipos de parâmetros disponíveis: os parâmetros em comum a todas as heurísticas e os parâmetros específicos a cada heurística, assim como os parâmetros dos algoritmos de escalonamento gráfico.

A.1 Parâmetros em Comum

Os parâmetros em comum a todas as heurísticas são preenchidos com os mesmos valores para possibilitar um resultado final que represente melhor o alcançado com cada

heurística ou combinação heurística. No entanto, cada possibilidade de configuração de cada parâmetro será descrita a seguir para se ter uma noção das opções de execução disponíveis no Gerador Automático de Arquiteturas. As descrições dos parâmetros específicos de cada heurística serão detalhadas na próxima seção.

Foi implementada uma tela contendo todos os parâmetros em comum, as heurísticas, a fim de possibilitar o mapeamento de uma determinada arquitetura em *benchmarks* para o qual ela não foi treinada. Essa tela pode ser visualizada na Figura A.1.

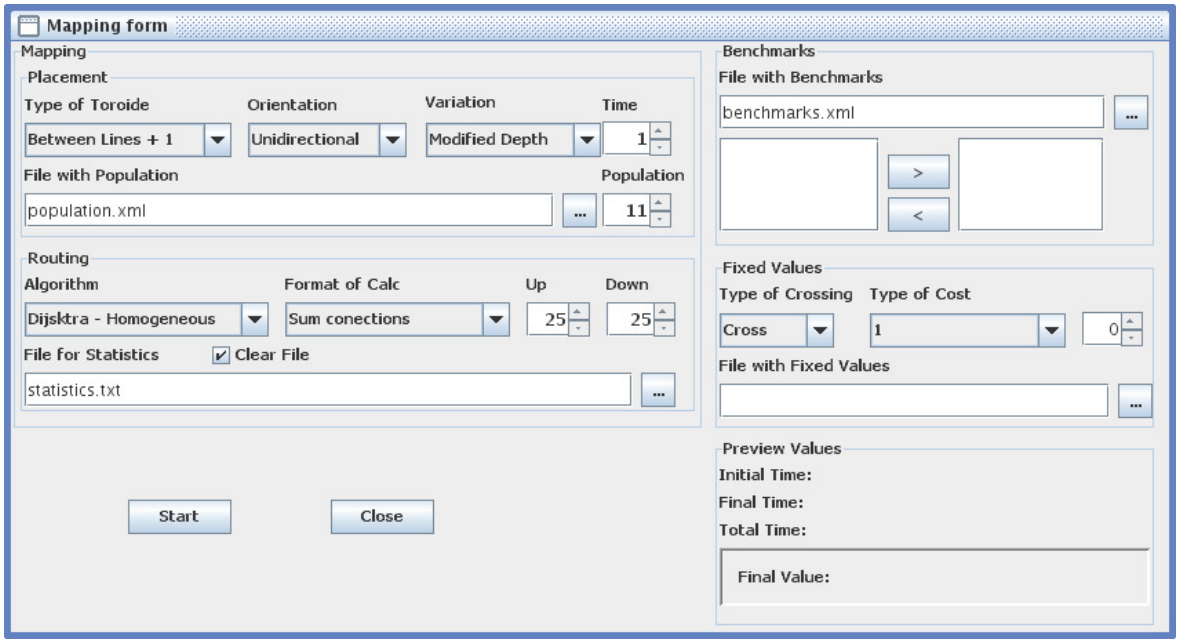


Figura A.1. Tela de Mapeamento.

Tipo de toróide – É a ligação de um elemento de uma matriz a um outro em outra extremidade. Para esse trabalho, foram implementadas seis possibilidades de toróide, sendo: (a) entre linhas, (b) entre a próxima linha e a última coluna da linha, (c) entre colunas, (d) entre a próxima coluna e a última linha da coluna, (e) entre a próxima linha, (f) entre a próxima coluna, além da possibilidade de não se ter toróide. Conforme Nageldinger (2001) e Hartenstein et al. (2000a) o uso de toróides é encorajado, e mesmo em nossas experiências práticas constatamos uma melhora nos resultados quanto utilizamos principalmente a toróide do tipo (b) que é também o valor padrão desse parâmetro.

Direção dos arcos – Define como serão as ligações entre os arcos, se em um único sentido (unidirecional) ou em ambos os sentidos (bidirecional), sendo que para a

Tabela A.1. Parâmetros disponiveis

Parâmetro / Opções	a	b	c	d	e	f	g
Tipo de toróide	entre linhas	entre a próxima linha e a última coluna da linha	entre colunas	entre a próxima coluna e a última linha da coluna	entre a próxima linha	entre a próxima coluna	sem toróide
Direção dos arcos	unidirecional	bidirecional					
Algoritmo de posicionamento	profundidade	randômica	profunidade modificada	todas as anteriores			
População de posicionamento	minímo 11						
Tempos de posicionamento	minímo 1 ms						
Fórmula para o custo dos segmentos	total de saltos	fixo em 1	proporcional ao comprimento				
Algoritmo de roteamento	Dijsktra	PathFinder					
Limite inferior	minímo 2						
Limite superior	minímo 3						
Formato dos resultados	média segmentos	total segmentos					
Arquivo de entrada com os indivíduos	arquivo XML						
Conjunto de *benchmarks*	arquivo XML						
Arquivo de armazenamento das estatísticas	arquivo texto						
Arquivo com as conexões fixas	arquivo XML						
Tipo de conexões fixas	diagonais	verticais e horizontais					

implementação física do sentido bidirecional será necessário o dobro do número de fios, por isso o valor padrão para esse parâmetro é unidirecional.

Algoritmo de posicionamento – O Algoritmo de posicionamento utilizado foi desenvolvido por Ferreira et al. (2007) e possui como possibilidades: (a) busca em profundidade, (b) busca randômico, (c) busca em profundidade modificada (uma parte em profundidade outra em largura), (d) todas as anteriores de acordo com uma divisão aleatória. Sendo que a que vem apresentando os melhores resultados é a busca em profundidade modificada e por isso essa opção aparece como padrão para esse parâmetro.

População de posicionamento – Representa o número de tentativas de obtenção de um melhor valor de posicionamento que será tentado pelo Algoritmo de posicionamento. Assim como o próximo parâmetro, constatou-se que quanto menor o valor melhor os resultados gerados, por isso esse parâmetro recebe como padrão um valor

que corresponde a 11 tentativas de melhora de posicionamento, correspondendo esse valor ao ao mínimo exigido para que o algoritmo funcione. Quanto menos tentativas de posicionamento, menos complexos serão esses posicionamentos, facilitando assim o roteamento.

Tempo de posicionamento – Serve para impedir que o número de tentativas acabe prejudicando todo o sistema devido a uma demora desnecessária. No entanto, notou-se que um tempo mínimo, como um milésimo de segundo, proporciona apenas uma única tentativa de posicionamento para a maioria dos *benchmarks* analisados, tendendo a gerar uma organização dos elementos na matriz que facilita o roteamento, produzindo assim melhores resultados, por isso o padrão para esse parâmetro passou a ser um milésimo de segundo.

Fórmula para o custo dos segmentos – A partir desse parâmetro, é possível definir como o Algoritmo de roteamento utilizado, descrito a seguir, irá interpretar o tamanho dos segmentos de cada arquitetura, se independentemente de seu tamanho (número de saltos) serão considerados apenas um (custo 1), se será aplicada alguma fórmula para calcular a proporcionalidade de seu tamanho em relação ao número de segmentos gastos, como o apresentado por Taghavi et al. (2004), ou alguma outra fórmula. O padrão adotado na geração dos resultados presentes nos estudos de caso a seguir foi o custo 1, o mesmo utilizado pelo algoritmo de posicionamento.

Algoritmo de roteamento – Há duas opções de algoritmos de roteamento disponíveis no Gerador de Arquiteturas: o Algoritmo de Dijkstra implementado por Ferreira et al. (2007) e o Algoritmo de PathFinder implementado por Pimentel (2007). O Algoritmo de Dijkstra não necessita de nenhum outro parâmetro extra para funcionar e aparece como padrão por apresentar resultados de uma forma mais rápida e satisfatória, já o Algoritmo e PathFinder necessita de dois outros parâmetros descritos a seguir.

Limite inferior – É o número mínimo de conexões, por nó na arquitetura, consideradas durante a execução do Algoritmo de PathFinder. Quanto menor, mais nós participam das conexões, porém maior a possibilidade de congestionamentos. Para evitar a possibilidade de congestionamento e possibilitar o roteamento de um número maior de *benchmarks*, optou-se por usar um valor simbólico igual a 25 como padrão para esse parâmetro.

Limite superior – É o número máximo de conexões, por nó na arquitetura, consideradas durante a execução do Algoritmo de PathFinder. Observou-se que, a partir de um determinado número, há uma perda da influência desse parâmetro no valor final obtido. Assim como o anterior, optou-se por um valor simbólico igual a 25 como

padrão para esse parâmetro, para que não houvesse uma interferência desse parâmetro no valor final obtido.

Formato dos resultados – Os resultados do roteamento são utilizados para a avaliação das arquiteturas, porém os algoritmos de roteamento utilizados permitem dois formatos de resultados: (a) a média do número de segmentos da arquitetura utilizados para o roteamento das conexões dos *benchmarks* na arquitetura e, (b) o total de segmentos da arquitetura utilizados para o roteamento das conexões dos *benchmarks* na arquitetura avaliada. Com isso pôde-se optar por qual dos dois formatos as heurísticas irão selecionar a melhor arquitetura. Para os estudos de caso apresentados, optou-se por usar o total de segmentos como fator de avaliação das arquiteturas pelas heurísticas.

Arquivo de entrada com os indivíduos – Dependendo da heurística esse arquivo pode conter de um a infinitos indivíduos. Para o AG, esse arquivo representa a população inicial e nesse trabalho ele continha normalmente 100 indivíduos. Para o PR, ele continha por volta de 10 indivíduos e para o SA ele continha um único indivíduo que normalmente era alguma arquitetura considerada a melhor por uma das outras duas heurísticas. Os indivíduos são definidos em XML conforme detalhado na seção 3.2.

Conjunto de *benchmarks* – A entrada desse parâmetro é uma descrição em XML da localização do benchmark, o nome do *benchmark* e a quantidade de vértices e conexões que formam o benchmark. À esquerda ficam os *benchmarks* disponíveis e à direita os que serão usados no mapeamento. Caso nenhum seja selecionado todos os bechmarks da esquerda serão utilizados.

Arquivo de armazenamento das estatísticas – Nesse arquivo, serão armazenados todos os valores gerados pelo Algoritmo de roteamento. Como descrito na seção anterior, o conteúdo desse arquivo servirá para a geração de dois gráficos, além disso, o nome dado a esse arquivo servirá de base para os nomes de outros arquivos gerados pelas heurísticas, dentre eles, os arquivos em XML contendo o melhor e o pior indivíduo, os dez melhores indivíduos, e quando for o caso, a última população inteira.

Caso se opte por usar algumas ligações de forma que as heurísticas não as possa manipular, o que chamamos de conexões fixas, esse uso deve ser definido de acordo com os seguintes parâmetros:

Arquivo com as conexões fixas – Durante grande parte dos testes, usou-se esse parâmetro contendo uma lista de conexões que seriam inseridas apenas no mapeamento das arquiteturas, porém não seriam passadas para as partes das heurísticas que manipulam as conexões, como os cruzamentos, mutações, perturbações, transformações em

geral etc. Esse parâmetro pôde ser abandonado com a correção de vários problemas nos códigos, mas ainda está disponível na ferramenta.

Tipo de conexões fixas – Quando se usavam as conexões fixas, foram testados dois tipos de conexões, as em diagonais e as verticais e horizontais, e o padrão que melhor apresentava resultados era o que unia as verticais e horizontais por isso esse valor aparece como padrão do parâmetro.

A.2 Parâmetros Específicos

Continuando o fluxo definido na Figura 5.5, as próximas seções apresentarão uma breve descrição dos parâmetros específicos de cada heurística implementada para esse trabalho.

A.2.1 Algoritmo Genético

Tempo de geração – Esse parâmetro serve para definir um tempo máximo de execução para o AG, porém para esse trabalho seu uso é desencorajado, pois, dependendo de outros parâmetros, o AG pode demorar de alguns minutos a alguns dias para atingir 100 gerações, por exemplo, mas seu valor padrão é o equivalente a 24 horas.

Número de gerações – É o principal ponto de parada do AG utilizado nesse trabalho para as gerações de resultados, isso se deve à tentativa de obtenção de resultados mais precisos, e o padrão para esse parâmetro tem sido 100 gerações.

Opções de cruzamentos – Para esse trabalho, foram implementadas três opções de cruzamento, conforme descrito na seção 4.4.2.2, porém por apresentar como resultado final uma arquitetura muito irregular, o cruzamento entre topologias foi excluído dos resultados presentes nos estudos de caso analisados, e por apresentar um tempo de convergência inferior, o cruzamento entre nos aparece como padrão desse parâmetro.

Outros parâmetros – Parâmetros como taxa de renovação, taxa de mutação, número de gerações sem melhoria e número mínimo de indivíduos não estão disponíveis à customização do usuário do Gerador de Arquiteturas. Esses parâmetros são definidos internamente no código fonte da ferramenta e seus valores padrão são: taxa de renovação de 20% do número total de indivíduos; taxa de mutação, que tem uma probabilidade igual a 0,5% de ocorrer durante cada passo no AG; e o número de gerações sem melhoria antes que o AG aborte é de 200. O número mínimo de indivíduos exigidos para o AG é de 10 indivíduos para que a taxa de renovação seja de pelo menos 2 indivíduos a cada geração.

A tela do Gerador de Arquiteturas que possibilita a interação do usuário com a heurística AG pode ser visualizada na Figura A.2.

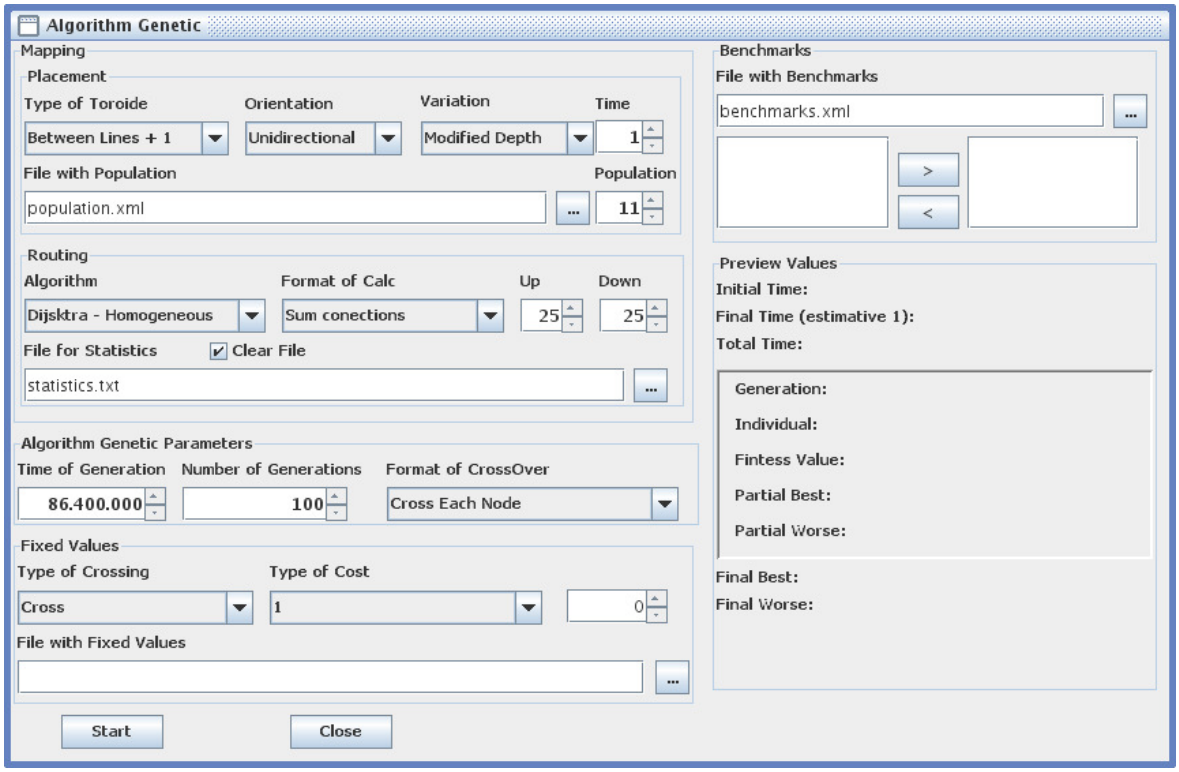


Figura A.2. Tela do Gerador de Arquiteturas que possibilita a interação com o Algoritmo Genético.

A.2.2 Path Relinking

A partir do arquivo de entrada dos indivíduos definido na seção anterior, o PR pode ser aplicado de três formas:

PR entre todos os indivíduos do arquivo, normalmente aplicado ao conjunto dos X melhores indivíduos obtidos com o AG, podendo ser aplicado a qualquer quantidade de indivíduos, em que todos são transformados em todos, obtendo-se o melhor indivíduo.

PR entre os X primeiros indivíduos do arquivo, nesse caso recomenda-se que o arquivo contenha em sua ordenação do melhor para o pior indivíduo.

PR entre um número aleatório X de indivíduos, recomenda-se o seu uso quando o número de indivíduos do arquivo for muito grande e eles não estejam ordenados, ou mesmo para fins de amostra estatística.

A princípio, o ponto de parada do Algoritmo é a transformação de todos os indivíduos um no outro, porém a cada transformação de um indivíduo em outro há um tempo máximo permitido de 30 segundos, esse tempo corresponde a quase 30.000 tentativas de transformações de um indivíduo em outro. Isso foi necessário devido às características aleatórias das transformações entre os indivíduos, tendo-se notado a possibilidade de

um tempo quase infinito entre tais transformações, apesar de isso ocorrer em menos de 1% dos casos, e quando ocorria, perdia-se toda a geração.

A tela do Gerador de Arquiteturas que possibilita a interação do usuário com a heurística PR pode ser visualizada na Figura A.3.

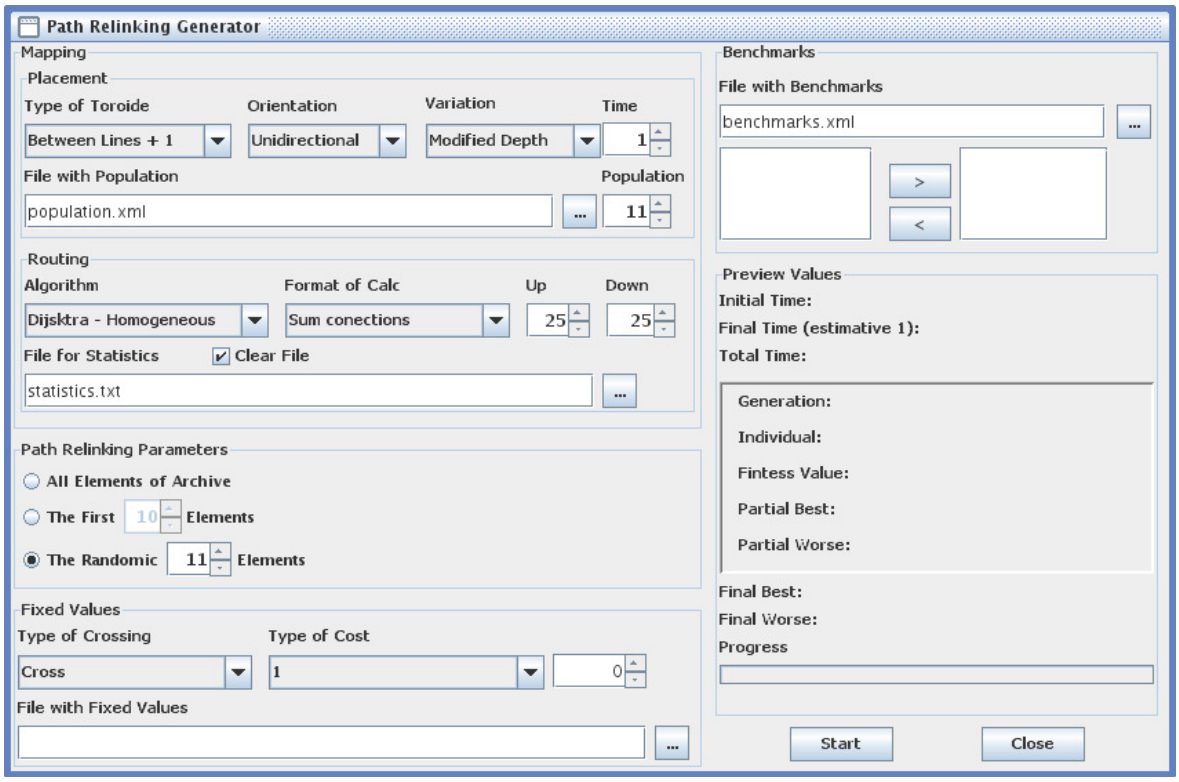


Figura A.3. Tela do Gerador de Arquiteturas que possibilita a interação com o Path Relinking.

A.2.3 Simulated Annealing

Temperaturas inicial e final – A aplicação da taxa de redução sobre a temperatura inicial gera uma temperatura atual em que será reaplicada a taxa de redução até que a temperatura atual passe a ser menor ou igual ao valor definido na temperatura final. Os valores padrão para essas temperaturas são 1000 e 0,001, respectivamente, e se considerarmos o valor padrão definido para a taxa de redução que é de 0,9, teremos 132 reduções de temperatura.

Iterações máximas – Entre cada redução de temperatura, será tentado um número máximo de iterações, e a cada iteração haverá uma perturbação. Após essa perturbação será calculado o custo como Δ = (valor novo) – (valor atual), e se esse

valor for menor que zero, o valor atual passa a ser o novo valor obtido (ou seja houve uma redução) senão a probabilidade de aceitação de um valor pior é: dado um valor randômico, se esse for menor que $e^{-\Delta/T}$ em que Δ é o valor do custo obtido e T = temperatura corrente, o pior valor será aceito. Isso normalmente ocorre quando a temperatura é maior.

Fórmulas para cálculo da taxa de redução da temperatura – Para esse trabalho, foram implementadas quatro fórmulas:

1. $T_k = \alpha T_{k-1} \; \forall k \geq 1$ onde $0 < \alpha < 1$
2. $T_k = \frac{T_{k-1}}{1+\gamma\sqrt{T_{k-1}}} \; \forall k \geq 1$ onde $0 < \gamma < 1$
3. $T_k = \begin{cases} \beta T_{k-1} \; se \; k{=}1 \\ \frac{T_{k-1}}{1+\gamma T_{k-1}} \; se \; k{\geq}2 \end{cases}$ onde $\gamma = \frac{T_0 - T_{k-1}}{(k-1)T_0 T_{k-1}}$ e $0 < \beta < 1$
4. $T_k = 0,9T_{k-1} \; \forall k \geq 1$

A tela do Gerador de Arquiteturas que possibilita a interação do usuário com a heurística SA pode ser visualizada na Figura A.4.

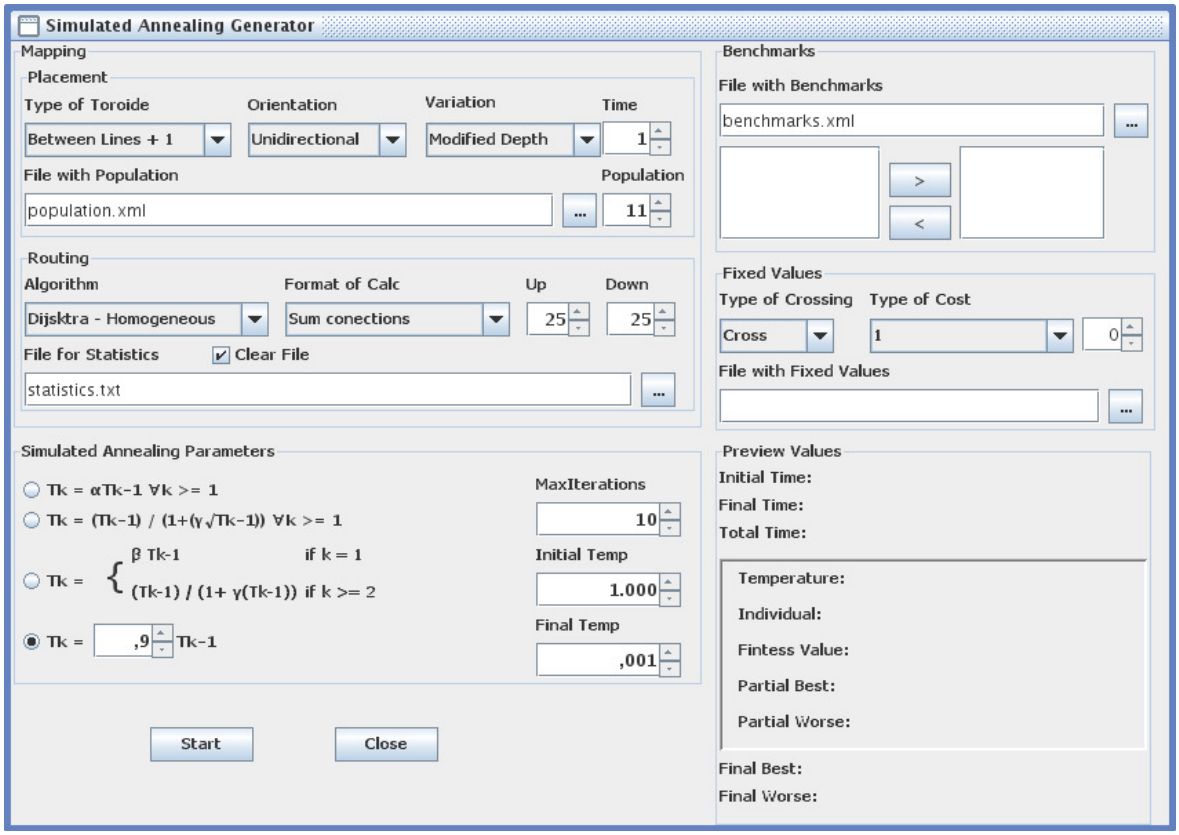


Figura A.4. Tela do Gerador de Arquiteturas que possibilita a interação com o Simulated Annealing.

A.2.4 Escalonamento

Arquivo com *benchmarks* – É o arquivo que contém as informações dos *benchmarks* que terão a topologia extraída para a formação da arquitetura.

Arquivo com a topologia inicial – O uso desse parâmetro é encorajado quando se desejar obter o caminho critico dos *benchmarks* em relação a uma topologia específica ou caso queira obter uma topologia baseada no balanceamento dos arcos dos fluxo de dadoss dos *benchmarks* em relação a uma topologia específica.

Orientação – Define se os segmentos da topologia serão direcionados para uma única direção ou se serão bidirecionais.

Segmentos – Utilizado para a definição da quantidade de segmentos que farão parte da topologia, esse parâmetro é utilizado pelo histograma na definição da distribuição dos tamanhos dos segmentos obtidos do escalonamento.

Limpar destino – Utilizado caso se queira limpar qualquer resultado anterior obtido com os mesmos parâmetros selecionados, porém aconselha o seu uso sempre que se desejar gerar uma nova topologia.

Máximo de conexões – Limita o número de segmentos com o mesmo comprimento em 50%. Esse parâmetro surgiu da necessidade de limitar o número de segmentos com comprimento igual a um, que correspondem a mais de 80%, na maioria dos casos.

Tipo de custo – Segue a mesma lógica do parâmetro global de mesmo nome, utilizado aqui para a obtenção de um gráfico com segmentos que levem em consideração o peso das distâncias obtidas.

Exibição dos resultados das extrações – Para tal, existem abas com os nomes dos algoritmos de escalonamento e com o resumo dos algoritmos contendo a profundidade máxima atingida por cada segmento para cada *benchmarks* escalonado.

Na Figura A.5 são exibidos os parâmetros descritos e o espaço reservado para a visualização do gráfico da topologia extraída para o Algoritmo ASAP. Na Figura A.6, é exibida a aba utilizada na apresentação dos resultados: à esquerda o comprimento encontrado entre cada arco do fluxo de dados e à direita o histograma com o resumo das quantidades de cada comprimento em relação ao total de comprimentos e a distribuição em relação à quantidade de segmentos da topologia.

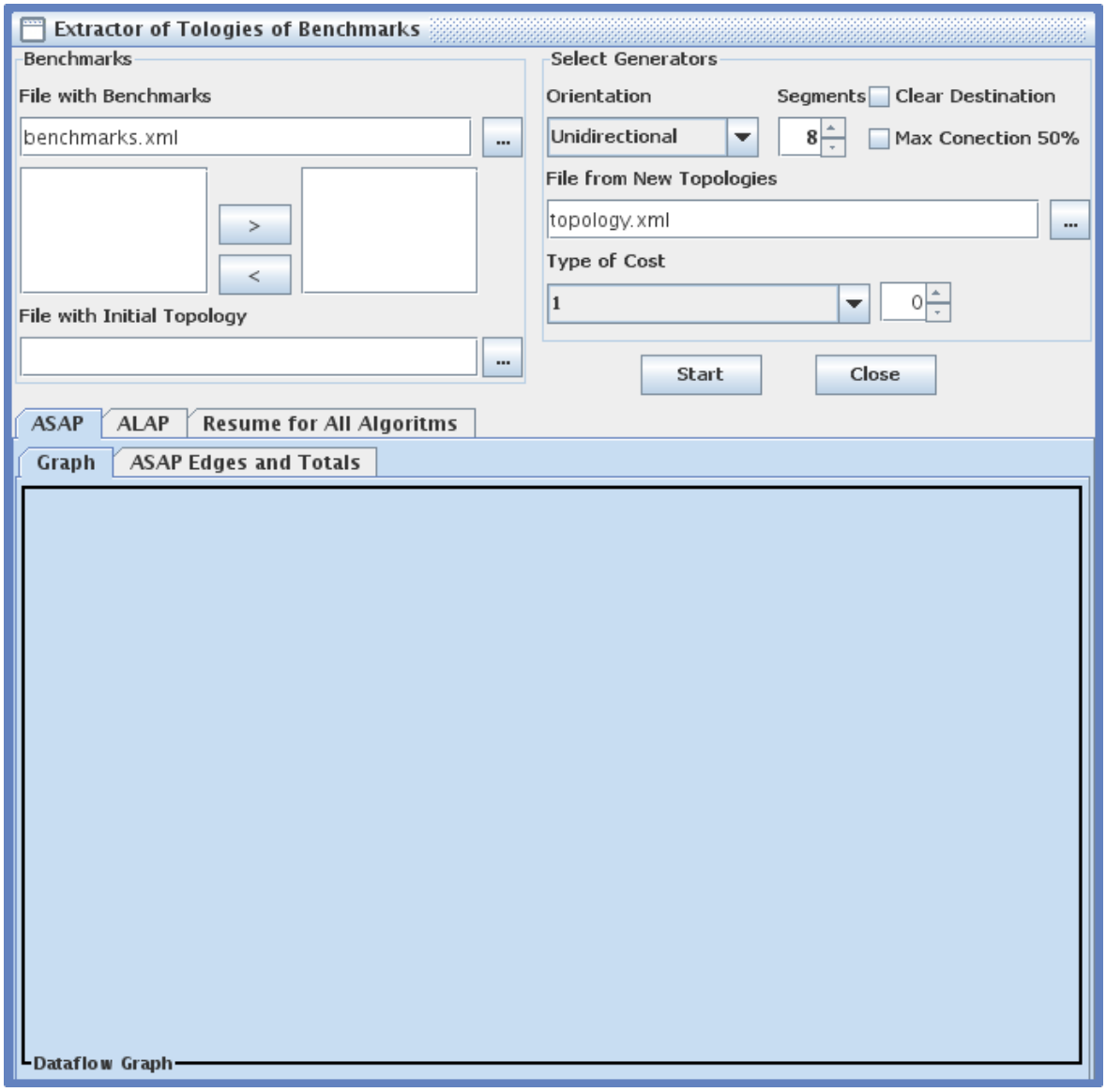


Figura A.5. Tela do Extrator de Topologias de Benchmarks, com o detalhe do visualizador de gráficos a mostra.

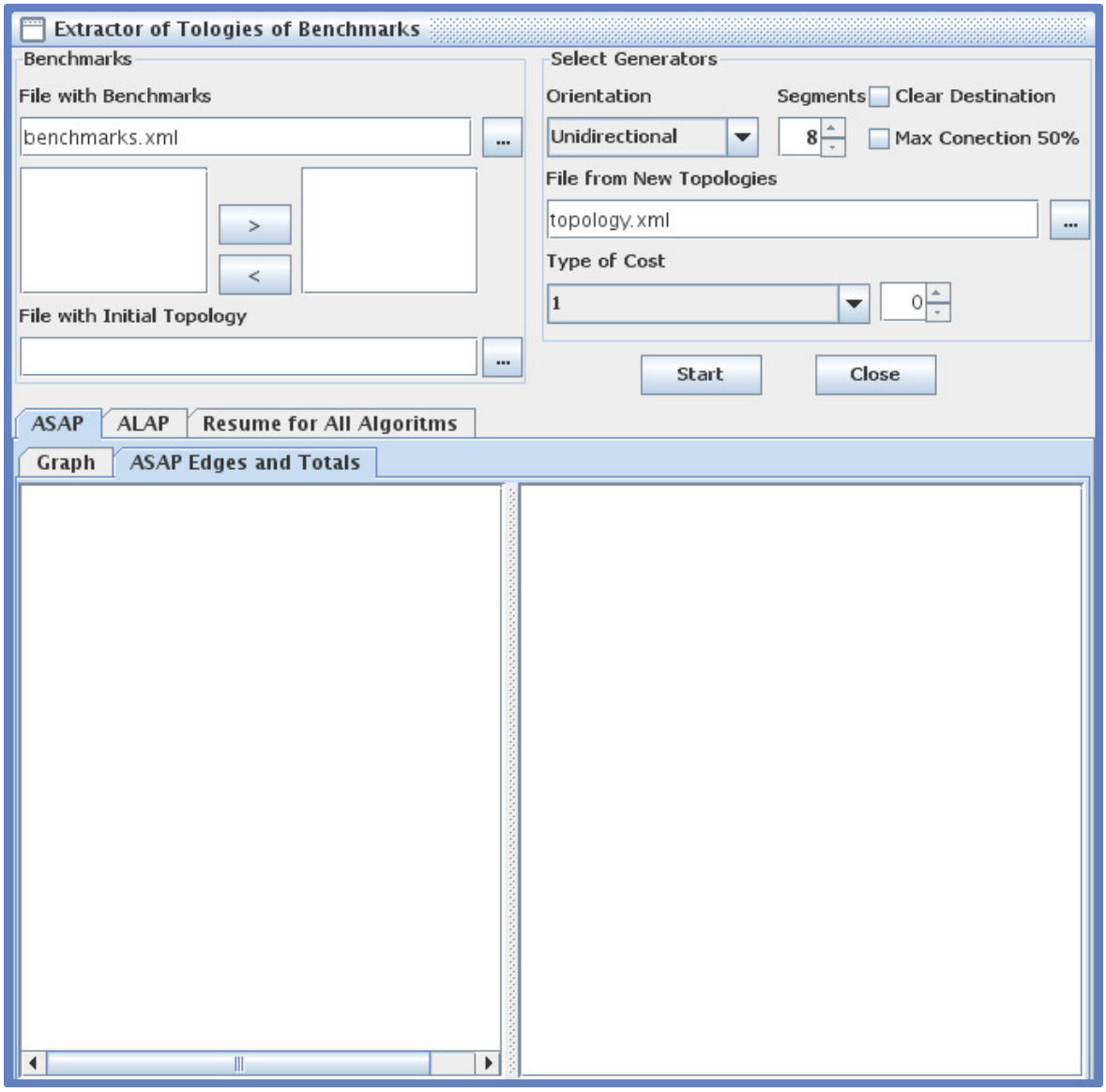


Figura A.6. Tela do Extrator de Topologias de Benchmarks com o detalhe da saída de resultados à mostra: à esquerda onde é mostrado o comprimento de cada arco do fluxo de dados, à direita onde ficam o histograma e a distribuição dos comprimentos dos segmentos.

Anexo B

1º e 2º Estudos de Caso

Esse anexo tem por objetivo expor os resultados obtidos com cada heurística e variação heurística analisada durante a execução desse trabalho, expondo o caminho percorrido até a obtenção dos resultados presentes no Capítulo 5, para tal em cada heurística analisada são demonstrados os resultados obtidos com o treinamento das heurísticas no grupo grande, composto por 16 *benchmarks*, e no grupo pequeno, composto por 4 *benchmarks*.

Entre os resultados demonstrados estão os gráficos de concentração e estatístico, o gráfico da topologia obtida com a heurística, o gráfico com os caminhos críticos totais por heurística e as tabelas com a distribuição dos comprimentos dos segmentos das melhores arquiteturas encontradas com cada heurística.

B.1 Algoritmo Genético

Duas variações de cruzamento do Algoritmo Genético foram aplicadas a fim de obter uma comparação qualitativa entre ambas. Os AGs tiveram os mesmos valores para os parâmetros descritos na Tabela 5.5.

Os três subgráficos da Figura 5.10a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.10b representam os resultados obtidos do treinamento no grupo pequeno, sendo que para o grupo grande a heurística AG com *crossover* por PR ficou 14,31% melhor que 0_1_hop e para o grupo pequeno a heurística AG com *crossover* por nó ficou 18,31% melhor que 0_1_hop.

Os três subgráficos da Figura B.1a representam os resultados obtidos do caminho crítico para o grupo grande e os da Figura B.1b os obtidos do caminho crítico para o grupo pequeno, sendo que para o grupo grande a heurística AG com *crossover* por PR ficou 10,33% melhor que 0_1_hop e para o grupo pequeno a heurística AG com *crossover* por PR ficou 9,65% melhor que 0_1_hop.

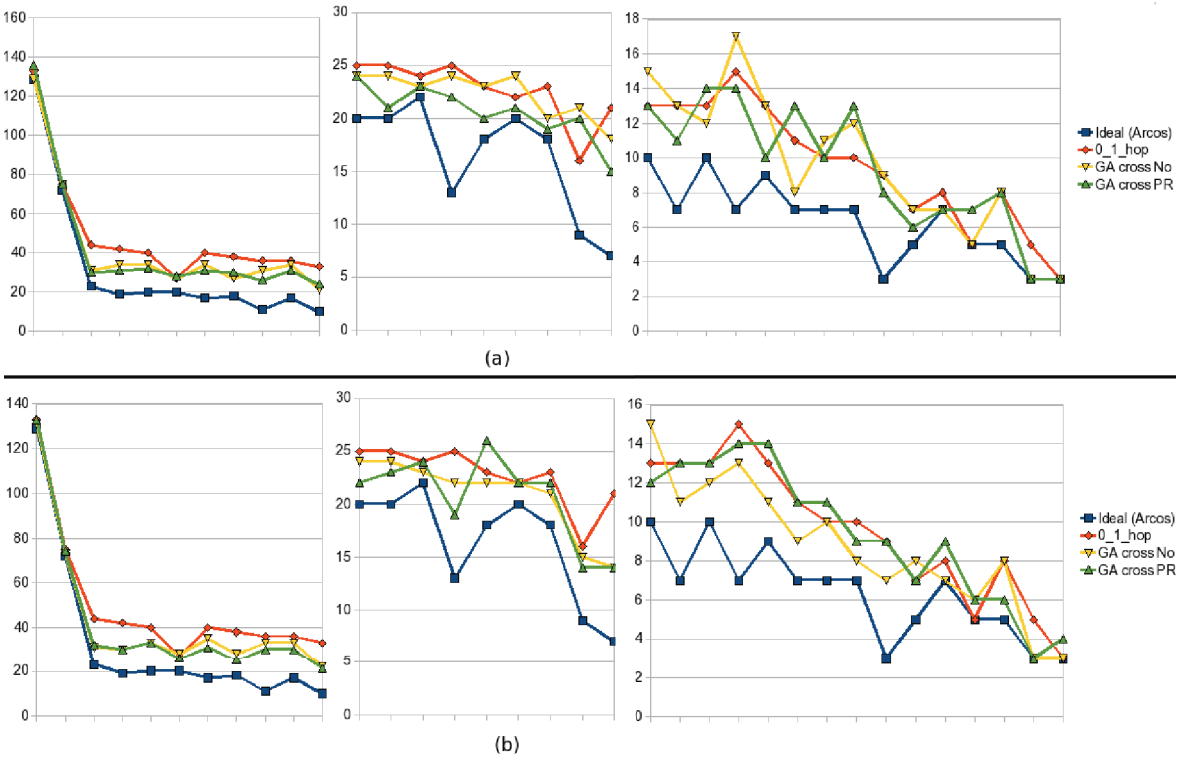


Figura B.1. Algoritmo Genético: (a) Grupo Grande, a heurística GA *crossover* PR ficou 33,61% acima do número mínimo possível de arcos do caminho crítico; (b) Grupo Pequeno, a heurística GA *crossover* PR ficou 34,62% acima do número mínimo possível de arcos do caminho crítico.

B.1.1 Algoritmo Genético com *crossover* por nó

Com a observação do gráfico de concentração, Figura B.2-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.2-2c podemos verificar uma maior concentração de resultados variando entre 460 e 500 segmentos.

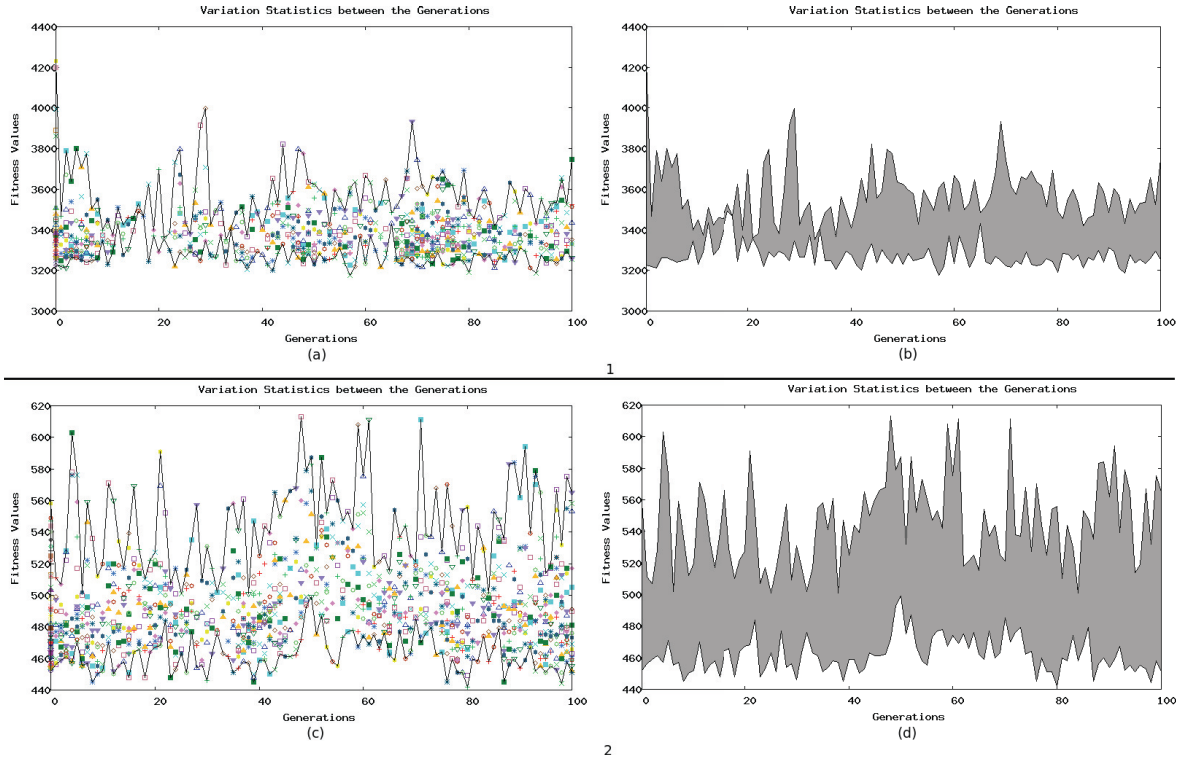


Figura B.2. Algoritmo Genético com *crossover* por nó – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.2-1b podemos verificar que não houveram valores inválidos gerados a ponto de ficar alguma geração sem algum valor válido, e aparentemente o melhor resultado se deu por volta da geração 60. Com a observação do gráfico estatístico, Figura B.2-2d podemos verificar que não houveram valores inválidos gerados a ponto de ficar alguma geração sem algum valor válido, e aparentemente o melhor resultado se deu por volta da geração 80.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.3 e B.4, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.1, sendo que para o grupo grande a maioria dos segmentos possuem dois ou quatro saltos (cerca de 75% de um total de oito conexões) e para o grupo pequeno a maioria dos segmentos possuem quatro saltos (cerca de 62,5% de um total de oito conexões) e o restante três saltos.

Tabela B.1. Distribuição dos segmentos da melhor arquitetura obtida com AG com *crossover* por nó

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	2	3	0	3
Pequeno	0	0	0	3	5

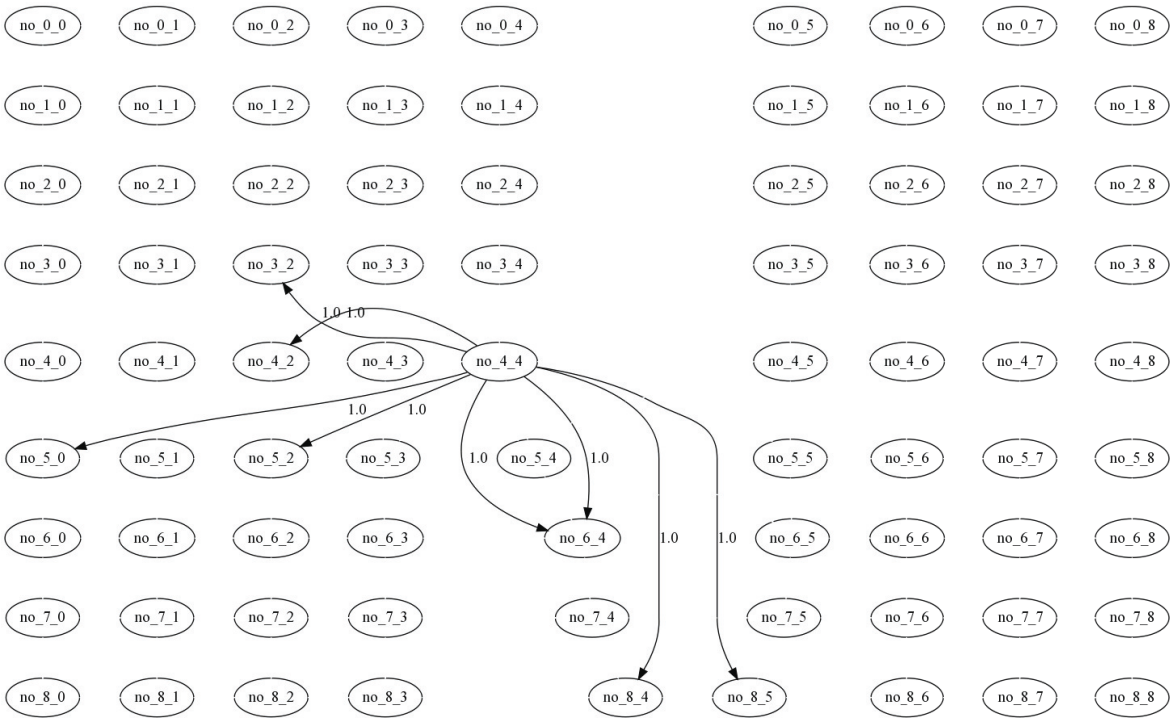


Figura B.3. Grupo Grande: Topologia da melhor arquitetura para o Algoritmo Genético com *crossover* por nó.

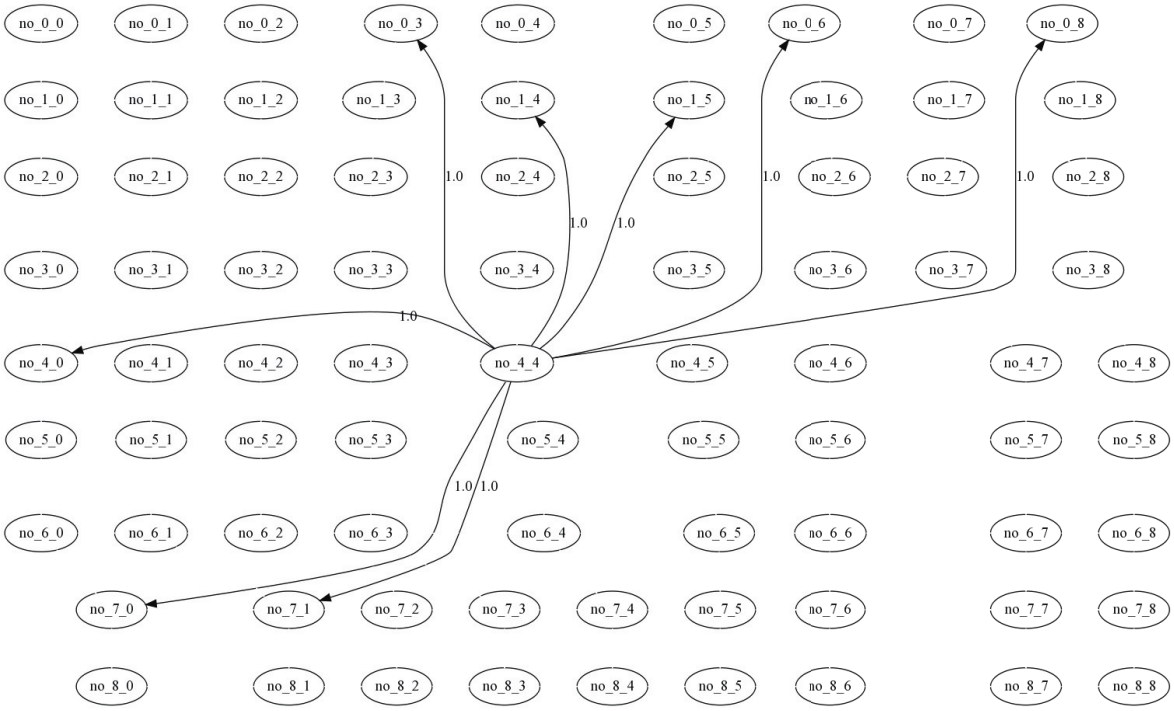


Figura B.4. Grupo Pequeno: Topologia da melhor arquitetura para o Algoritmo Genético com *crossover* por nó.

B.1.2 Algoritmo Genético com *crossover* por Path Relinking

Com a observação do gráfico de concentração, Figura B.5-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.5-2c podemos verificar uma maior concentração de resultados variando entre 450 e 500 segmentos.

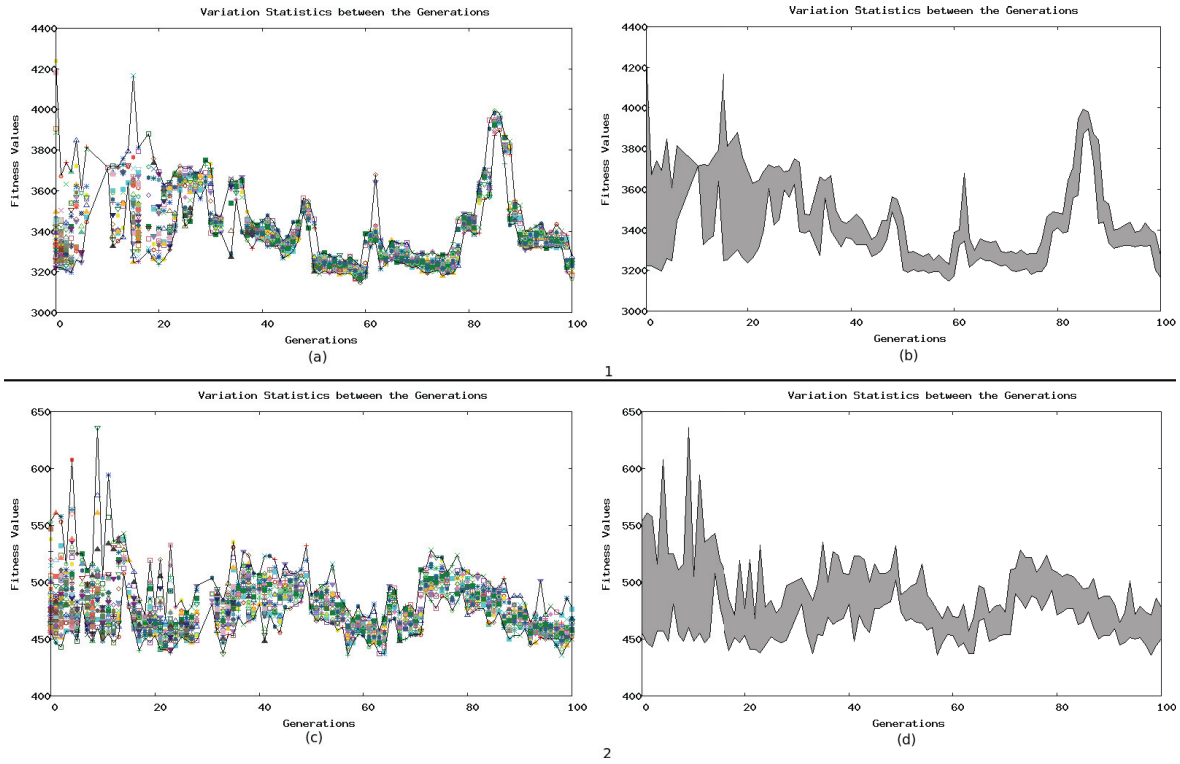


Figura B.5. Algoritmo Genético com *crossover* por Path Relinking – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.5-1b podemos verificar que não houveram valores válidos por volta da geração 10, e aparentemente o melhor resultado se deu por volta da geração 60. Com a observação do gráfico estatístico, Figura B.5-2d podemos verificar que aparentemente o melhor resultado se deu por volta da geração 30.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.6 e B.7, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.2, sendo que tanto para o grupo grande, quanto para o grupo pequeno a maioria dos segmentos possuem dois ou quatro saltos, sendo ambos com cerca de 87,5%, de um total de oito conexões.

Tabela B.2. Distribuição dos segmentos da melhor arquitetura obtida com Algoritmo Genético com *crossover* por Path Relinking

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	0	3	1	4
Pequeno	0	0	3	1	4

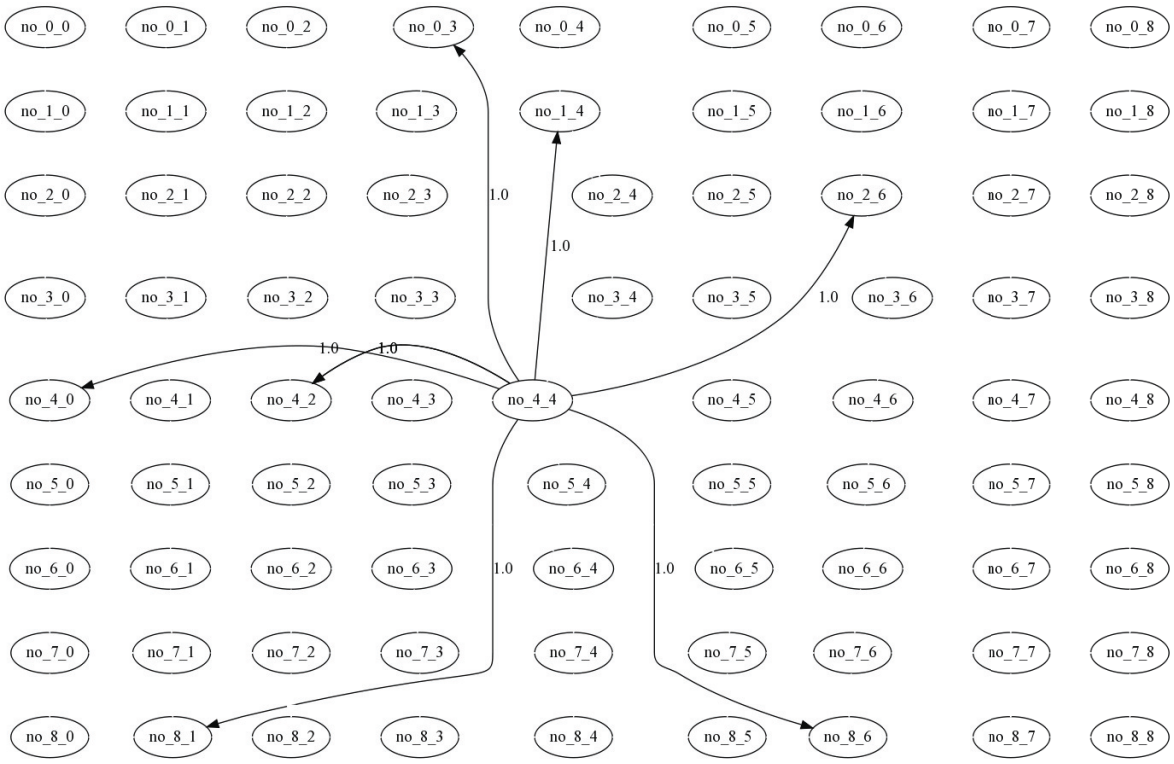


Figura B.6. Grupo Grande: Topologia da melhor arquitetura para o Algoritmo Genético com *crossover* por Path Relinking.

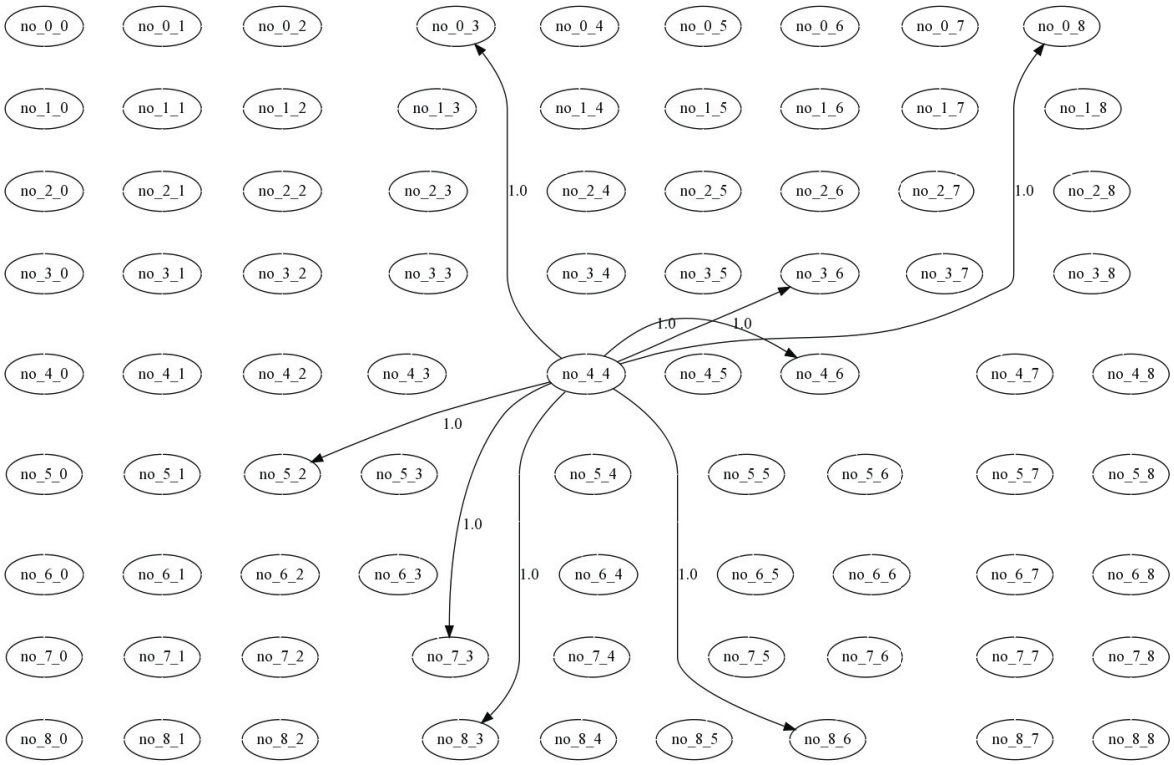


Figura B.7. Grupo Pequeno: Topologia da melhor arquitetura para o Algoritmo Genético com *crossover* por Path Relinking.

B.2 Path Relinking

O Algoritmo *Path Relinking* foi aplicado aos dois melhores conjuntos de indivíduos obtidos da aplicação do AG sobre uma população inicial, sendo o primeiro conjunto composto pelos 10 melhores indivíduos obtidos da aplicação do AG com *crossover* por nó e o segundo pelos 10 melhores indivíduos obtidos da aplicação do AG com *crossover* utilizando PR.

Os três subgráficos da Figura 5.11a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.11b, representam os resultados obtidos do treinamento no grupo pequeno, sendo que para o grupo grande, a heurística PR AG *crossover* PR ficou 15,30% melhor que 0_1_hop e para o grupo pequeno, a heurística PR AG *crossover* PR ficou 18,20% melhor que 0_1_hop.

Os três subgráficos da Figura B.8a representam os resultados obtidos do caminho crítico para o grupo grande e os da Figura B.8b os obtidos do caminho crítico para o grupo pequeno, sendo que para o grupo grande, a heurística PR AG *crossover* nó ficou

9,65% melhor que 0_1_hop e para o grupo pequeno, a heurística PR AG *crossover* PR ficou 10,44% melhor que 0_1_hop.

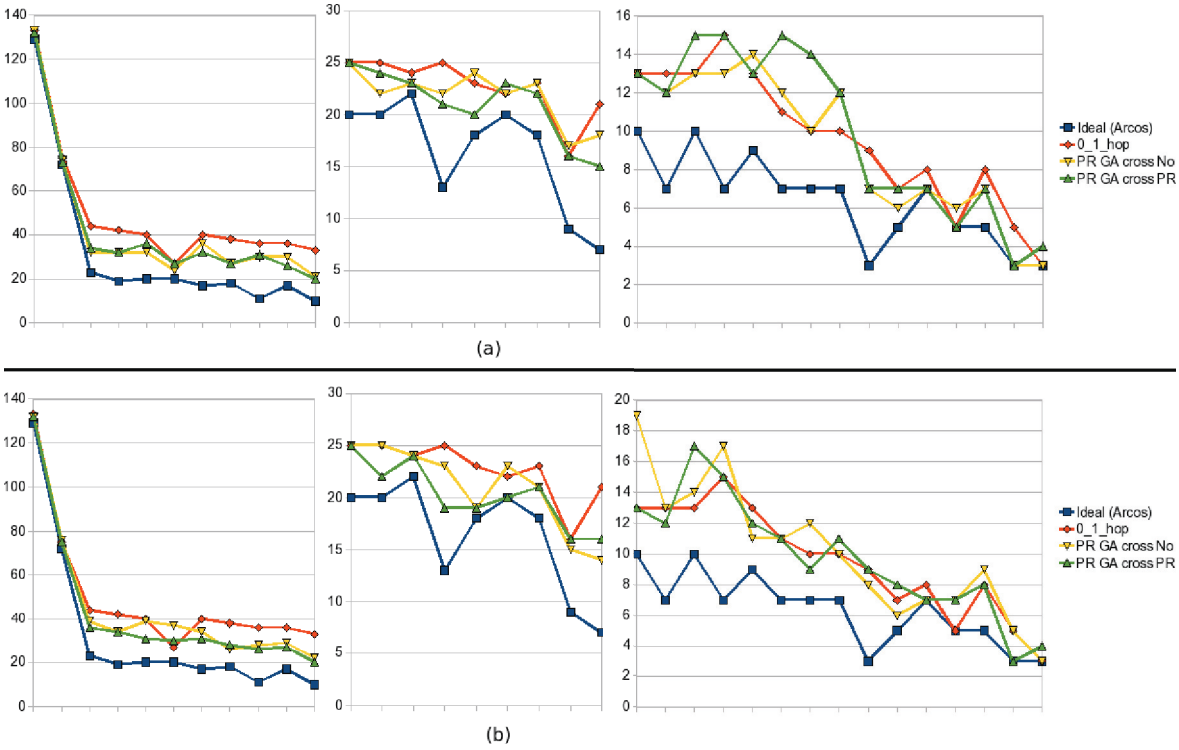


Figura B.8. Path Relinking: (a) Grupo Grande, a heurística PR GA *crossover* nó ficou 34,62% acima do número mínimo possível de arcos do caminho crítico; (b) Grupo Pequeno, a heurśtica PR GA *crossover* PR ficou 33,44% acima do número mínimo possível de arcos do caminho crítico.

B.2.1 Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó

Com a observação do gráfico de concentração, Figura B.9-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3400 segmentos. Com a observação do gráfico de concentração, Figura B.9-2c podemos verificar uma maior concentração de resultados variando entre 460 e 500 segmentos.

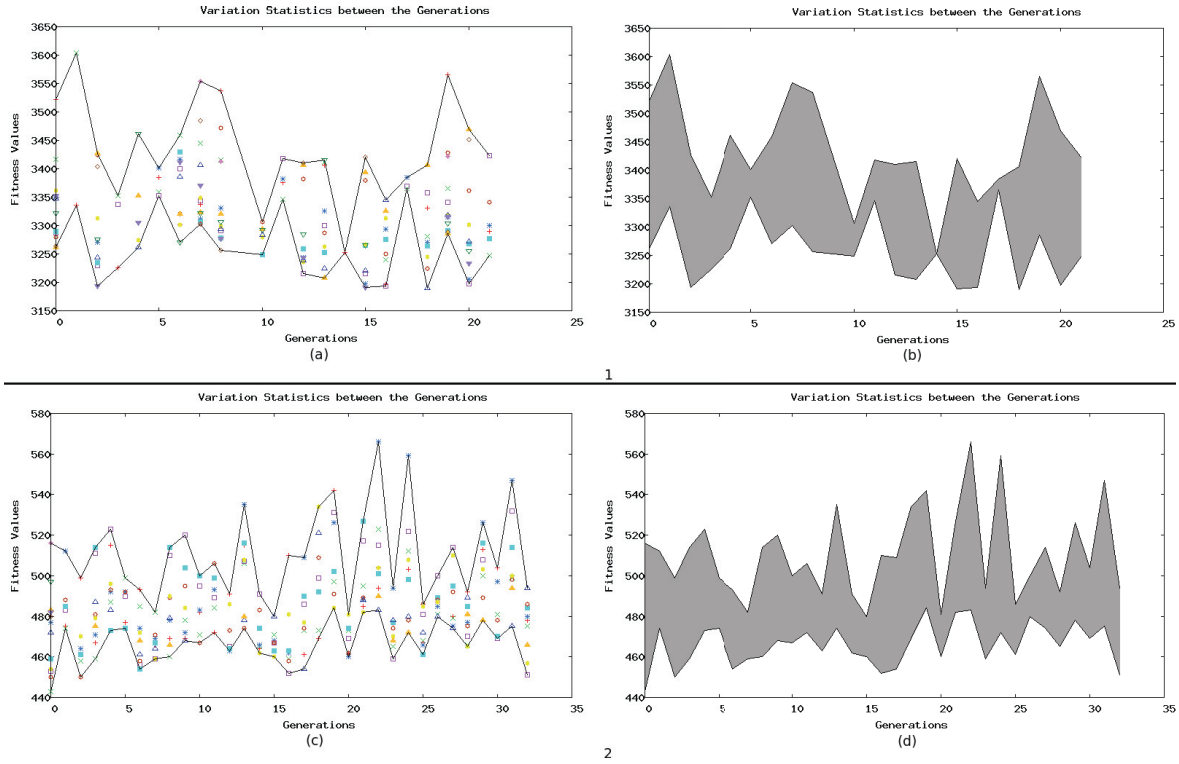


Figura B.9. Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó – A Gráficos de Concentração; B Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.9-1b podemos verificar que aparentemente o melhor resultado aparece logo próximo a quinta comutação de indivíduos. Com a observação do gráfico estatístico, Figura B.9-2d podemos verificar que aparentemente o melhor resultado aparece logo próximo a primeira comutação de indivíduos.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.10 e B.11, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.3, sendo que para o grupo grande a maioria dos segmentos possuem três ou quatro saltos (cerca de 75% de um total de oito conexões) e para o grupo pequeno a maioria dos segmentos possuem dois ou três saltos (cerca de 75% de um total de oito conexões).

Tabela B.3. Distribuição dos segmentos da melhor arquitetura obtida com Path Relinking de Algoritmo Genético com *crossover* por nó

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	1	1	2	4
Pequeno	0	0	3	3	2

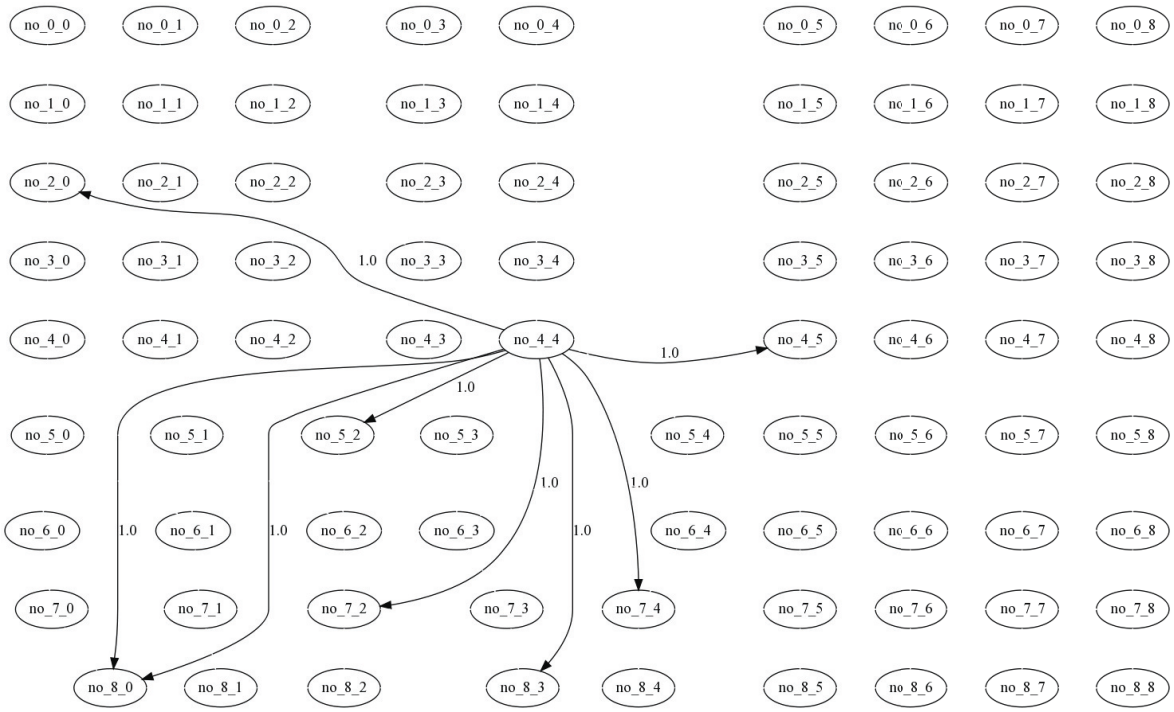


Figura B.10. Grupo Grande: Topologia da melhor arquitetura para o Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó.

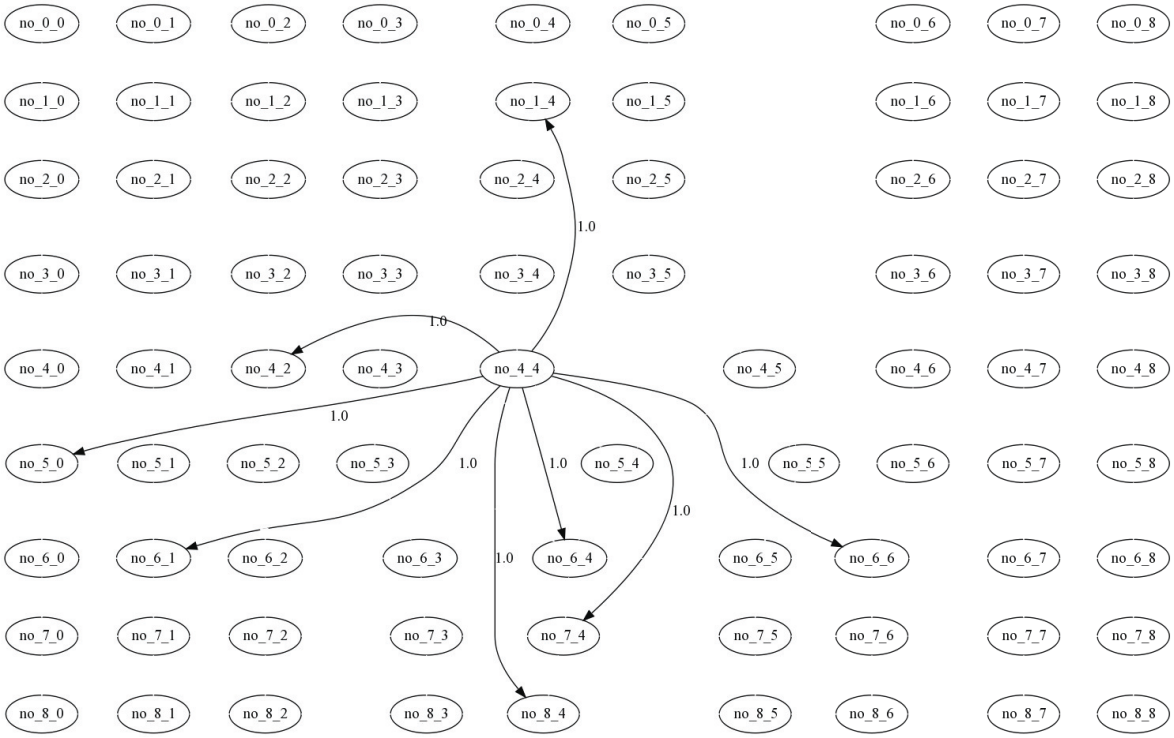


Figura B.11. Grupo Pequeno: Topologia da melhor arquitetura para o Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó.

B.2.2 Path Relinking aplicado ao Algoritmo Genético com *crossover* por Path Relinking

Com a observação do gráfico de concentração, Figura B.12-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.12-2c podemos verificar uma maior concentração de resultados variando entre 450 e 480 segmentos.

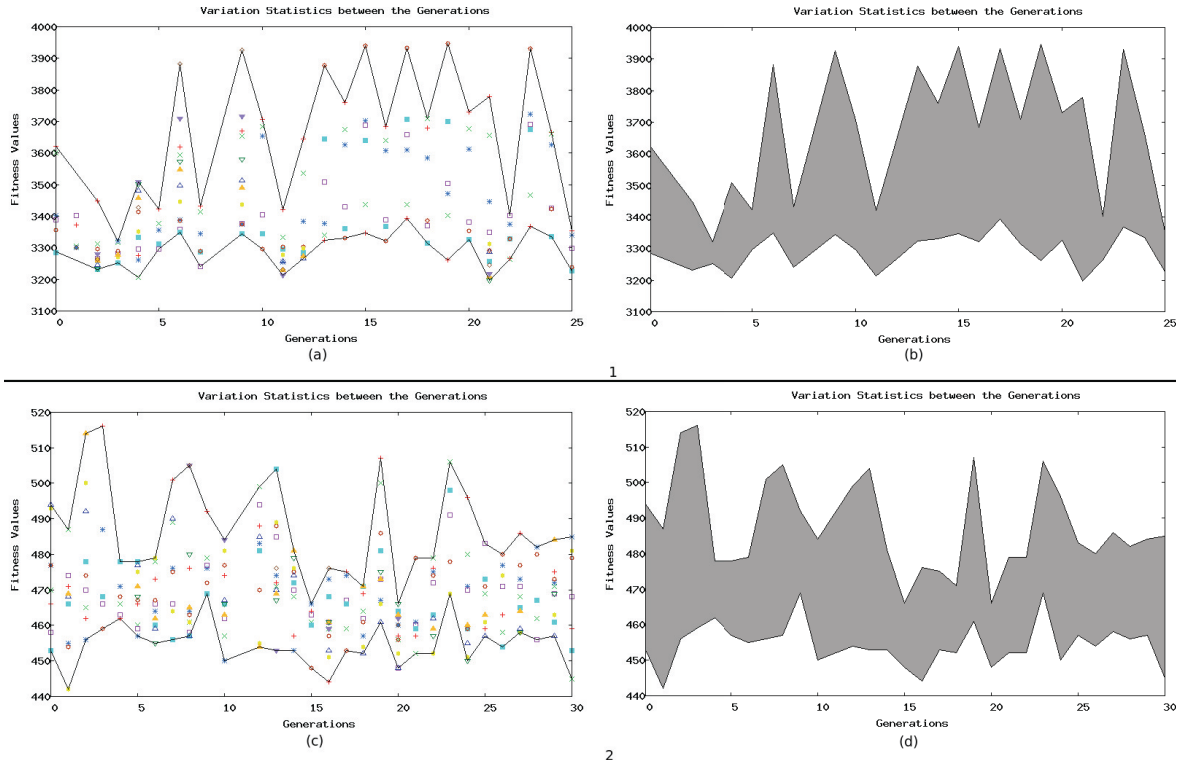


Figura B.12. Path Relinking aplicado ao Algoritmo Genético com *crossover* por Path Relinking – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.12-1b podemos verificar que aparentemente o melhor resultado se deu por volta da comutação de indivíduos número 20. Com a observação do gráfico estatístico, Figura B.12-2d podemos verificar que aparentemente o melhor resultado se deu antes da quinta comutação de indivíduos.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.13 e B.14, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.4, sendo que para o grupo grande todos os segmentos possuem um ou três saltos, ou seja, todas as oito conexões e para o grupo pequeno todos os segmentos possuem dois ou três saltos, ou seja, todas as oito conexões.

Tabela B.4. Distribuição dos segmentos da melhor arquitetura obtida com Path Relinking de Algoritmo Genético com *crossover* por Path Relinking

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	2	0	6	0
Pequeno	0	0	3	5	0

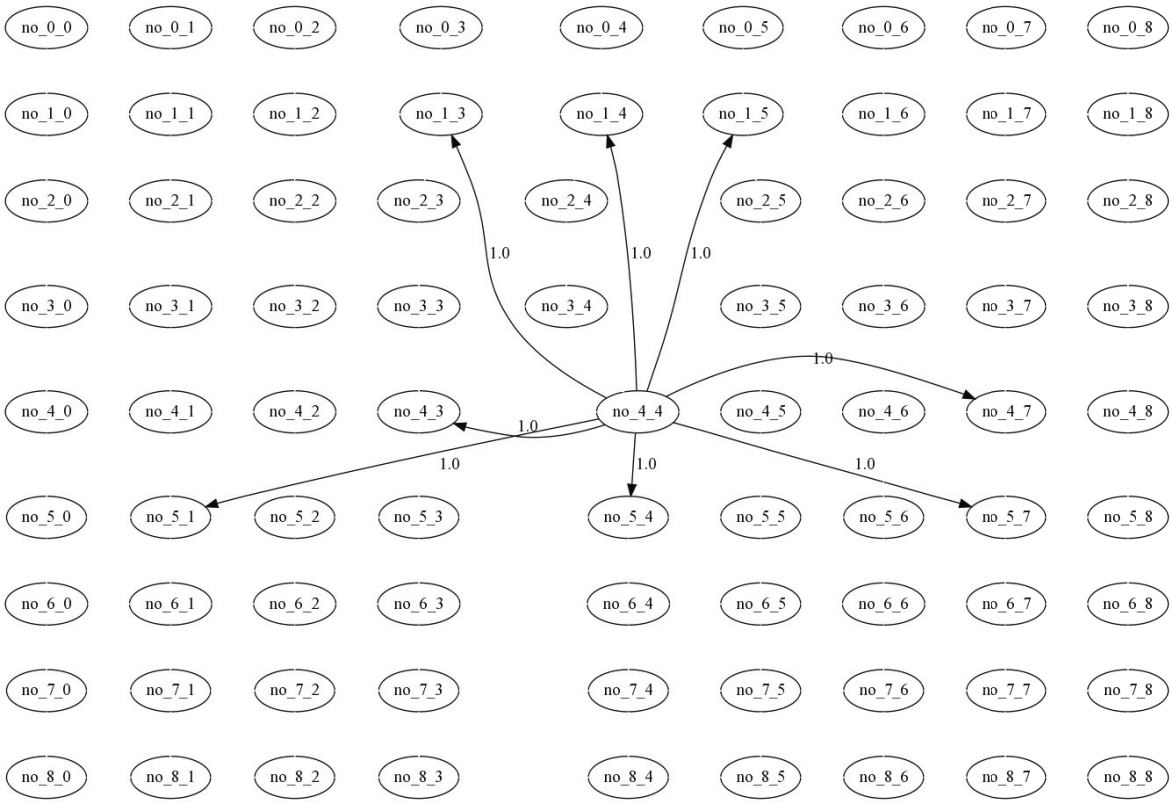


Figura B.13. Grupo Grande: Topologia da melhor arquitetura para o Path Relinking aplicado ao Algoritmo Genético com *crossover* por Path Relinking.

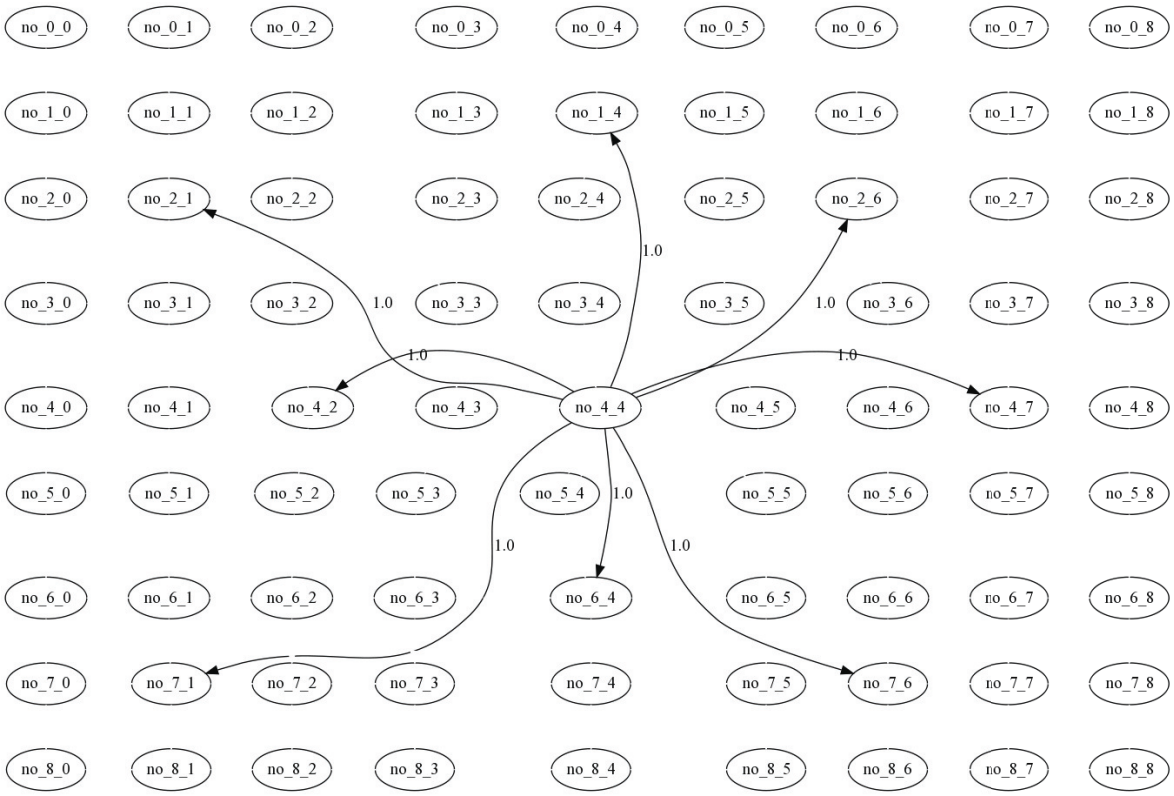


Figura B.14. Grupo Pequeno: Topologia da melhor arquitetura para o Path Relinking aplicado ao Algoritmo Genético com *crossover* por Path Relinking.

B.3 Simulated Annealing

O Algoritmo de *Simulated Annealing* foi aplicado a duas arquiteturas obtidas pela aplicação de outras heurísticas. A primeira arquitetura escolhida foi obtida pela aplicação do AG com *crossover* por nó a uma população inicial, a segunda arquitetura é o resultado da aplicação do PR aos 10 melhores indivíduos obtidos com a aplicação do AG com *crossover* por nó sobre uma população inicial.

Os três subgráficos da Figura 5.12a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.12b representam os resultados obtidos do treinamento no grupo pequeno, sendo que, para o grupo grande, a heurística SA AG *crossover* por nó ficou 18,30% melhor que 0_1_hop e para o grupo pequeno, a heurística SA AG *crossover* por nó ficou 17,29% melhor que 0_1_hop.

Os três subgráficos da Figura B.15a representam os resultados obtidos do caminho crítico para o grupo grande e os da Figura B.15b os obtidos do caminho crítico para o grupo pequeno, sendo que para o grupo grande, a heurística SA AG *crossover* por

nó ficou 10,55% melhor que 0_1_hop e para o grupo pequeno, a heurística SA AG *crossover* por nó ficou 10,89% melhor que 0_1_hop

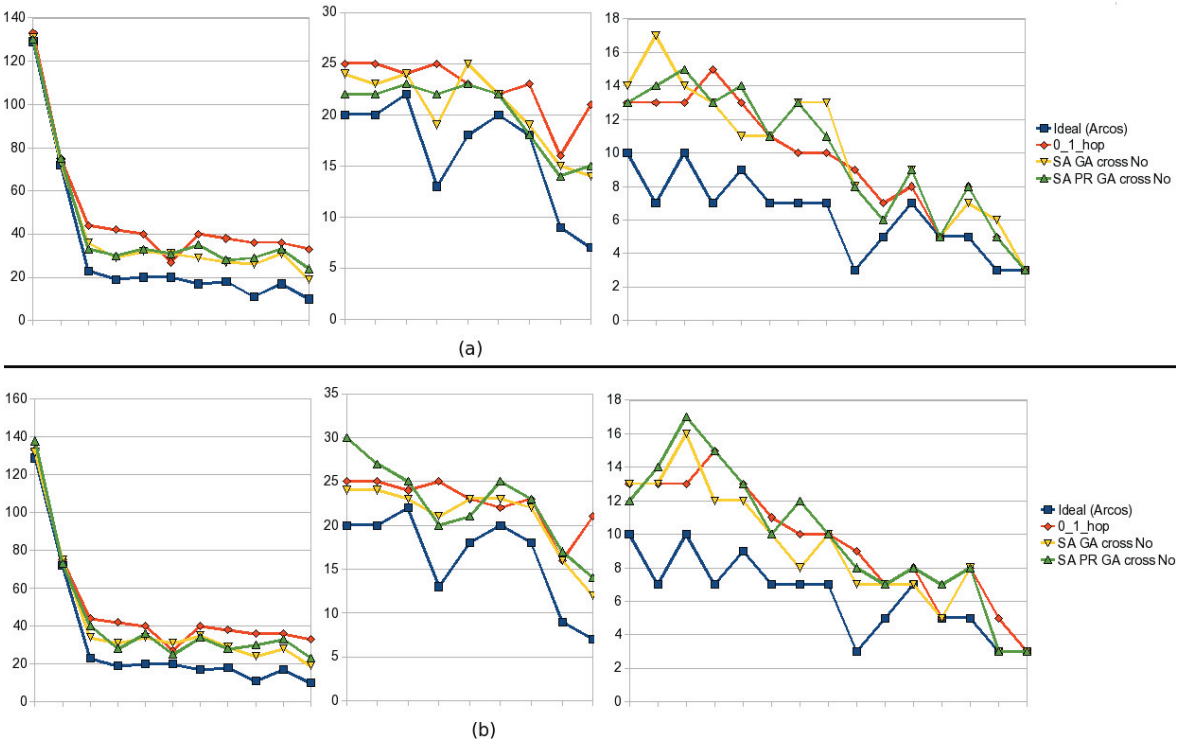


Figura B.15. Simulated Annealing: (a) Grupo Grande, a heurística SA GA *crossover* nó ficou 33,28% acima do número mínimo possível de arcos do caminho crítico; (b) Grupo Pequeno, a heurística SA GA *crossover* nó ficou 32,78% acima do número mínimo possível de arcos do caminho crítico.

O mesmo Algoritmo de SA foi aplicado a duas arquiteturas obtidas pela aplicação do algoritmo de escalonamento gráfico, sendo que em uma variação foi considerado o percentual real obtido pelo histograma do algoritmo de escalonamento e em uma segunda variação foi limitado a 50% o número total de segmentos com o mesmo número de saltos.

Essa implementação foi desenvolvida considerando a tendência apontada na Seção 5.5.

A aplicação do Algoritmo de SA nas arquiteturas extraídas com os algoritmos de escalonamento gráfico se deu seguindo o mesmo processo descrito na seção 5.4.

Os três subgráficos da Figura 5.13a representam os resultados obtidos do treinamento no grupo grande e os da Figura 5.13b representam os resultados obtidos do treinamento no grupo pequeno, sendo que, para o grupo grande, a heurística SA ASAP

ficou 10,57% melhor que 0_1_hop e para o grupo pequeno, a heurística SA ALAP limitado ficou 16,44% melhor que 0_1_hop.

Os três subgráficos da Figura B.16a representam os resultados obtidos do caminho crítico para o grupo grande e os da Figura B.16b os obtidos do caminho crítico para o grupo pequeno, sendo que para o grupo grande, a heurística SA ASAP ficou 8,42% melhor que 0_1_hop e para o grupo pequeno, a heurística SA ALAP limitado ficou 11,45% melhor que 0_1_hop.

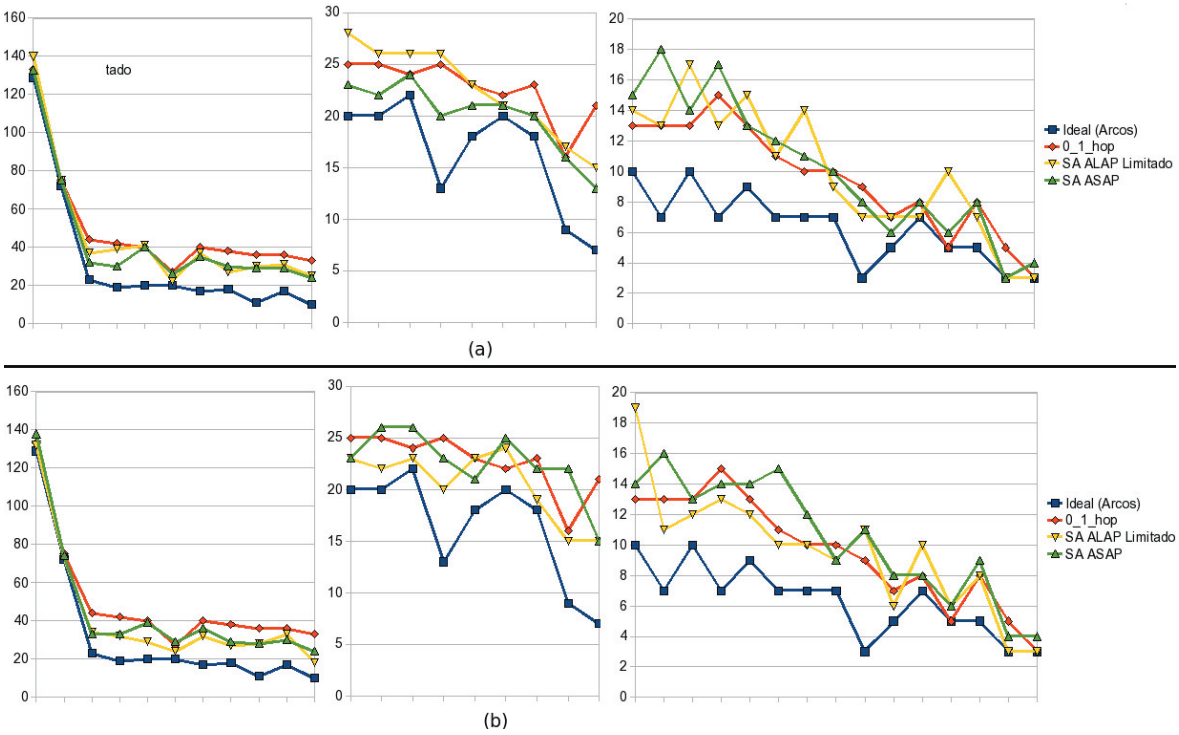


Figura B.16. Simulated Annealing: (a) Grupo Grande, a heurística SA ASAP ficou 36,45% acima do número mínimo possível de arcos do caminho crítico; (b) Grupo Pequeno, a heurística SA ALAP limitado ficou 31,94% acima do número mínimo possível de arcos do caminho crítico.

B.3.1 Simulated Annealing aplicado ao Algoritmo Genético com *crossover* por nó

Com a observação do gráfico de concentração, Figura B.17-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.17-2c podemos verificar uma maior concentração de resultados variando entre 460 e 480 segmentos.

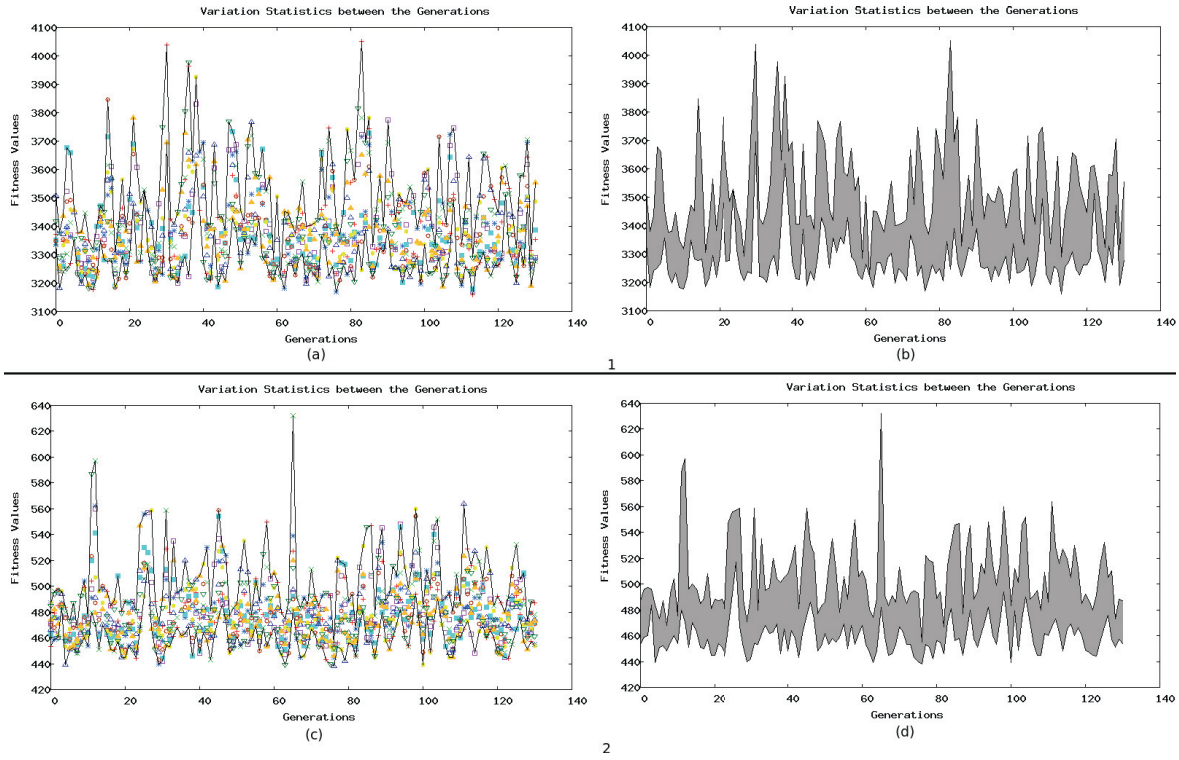


Figura B.17. Simulated Annealing aplicado ao Algortimo Genético com *crossover* por nó – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.17-1b podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 110. Com a observação do gráfico estatístico, Figura B.17-2d podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 80.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.18 e B.19, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.5, sendo que tanto para o grupo grande, quanto para o grupo pequeno a maioria das conexões possuem três ou quatro saltos (cerca de 75% para o grupo grande e 62,5% para o grupo pequeno, de um total de oito conexões).

Tabela B.5. Distribuição dos segmentos da melhor arquitetura obtida com Simulated Annealing de Algoritmo Genético com *crossover* por nó

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	0	2	3	3
Pequeno	0	1	1	2	4

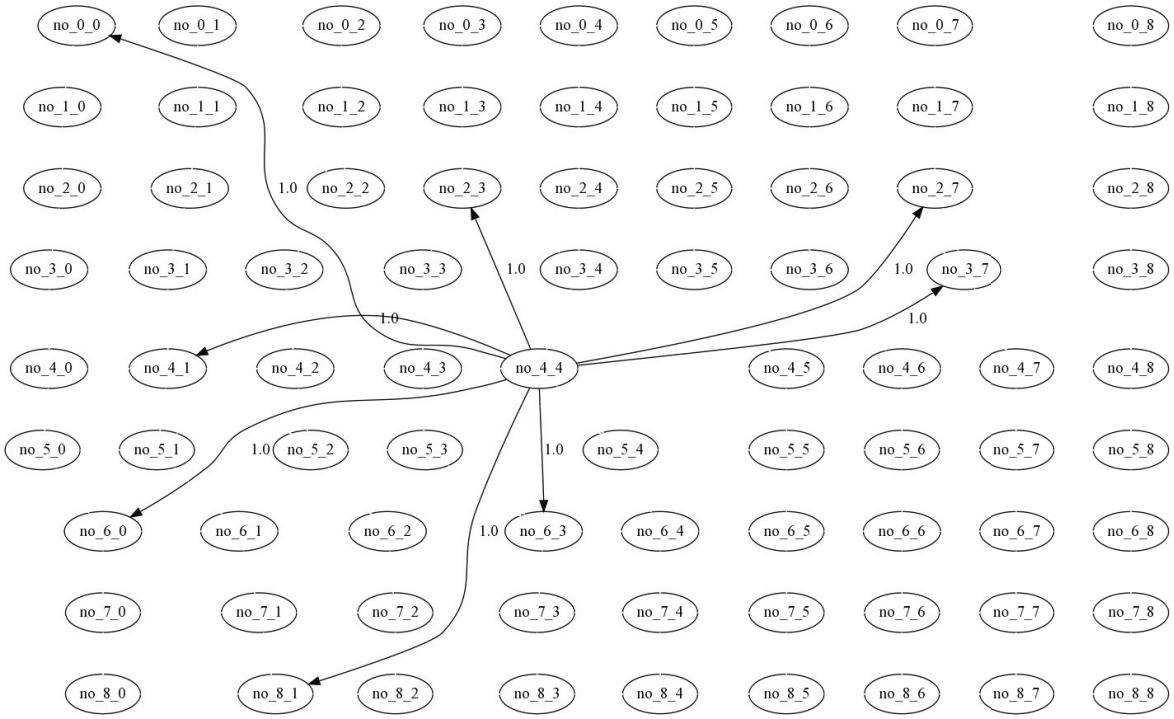


Figura B.18. Grupo Grande: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Algoritmo Genético com *crossover* por nó.

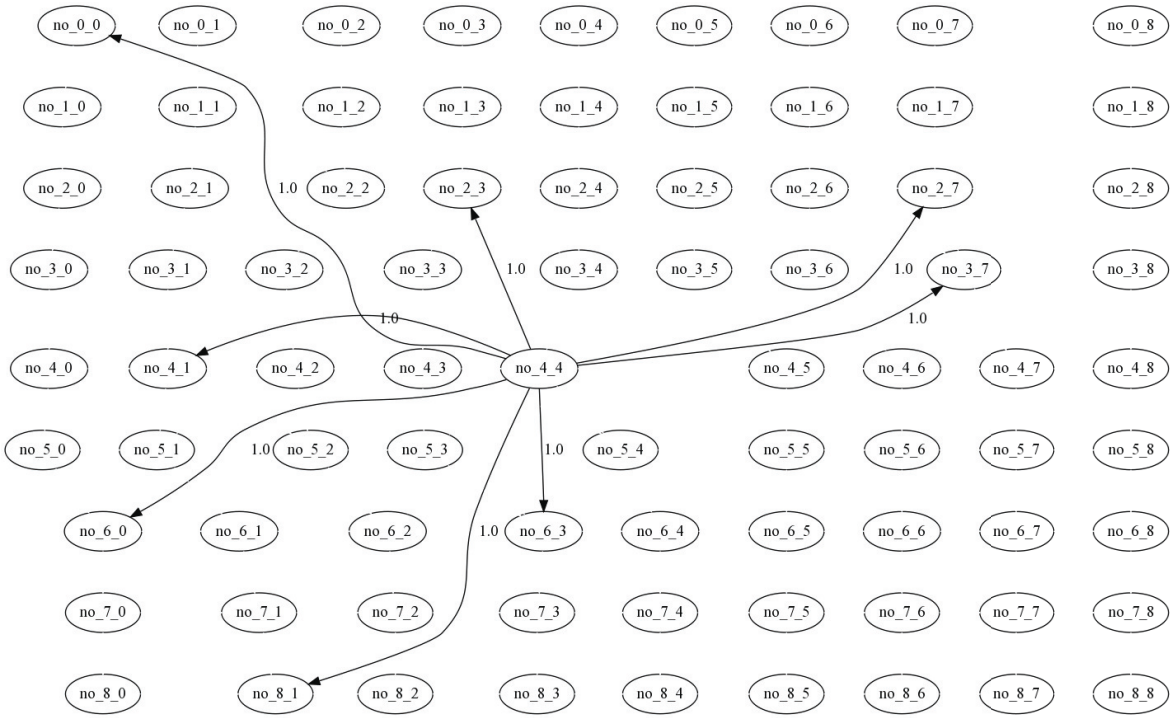


Figura B.19. Grupo Pequeno: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Algoritmo Genético com *crossover* por nó.

B.3.2 Simulated Annealing aplicado ao Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó

Com a observação do gráfico de concentração, Figura B.20-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.20-2c podemos verificar uma maior concentração de resultados variando entre 460 e 490 segmentos.

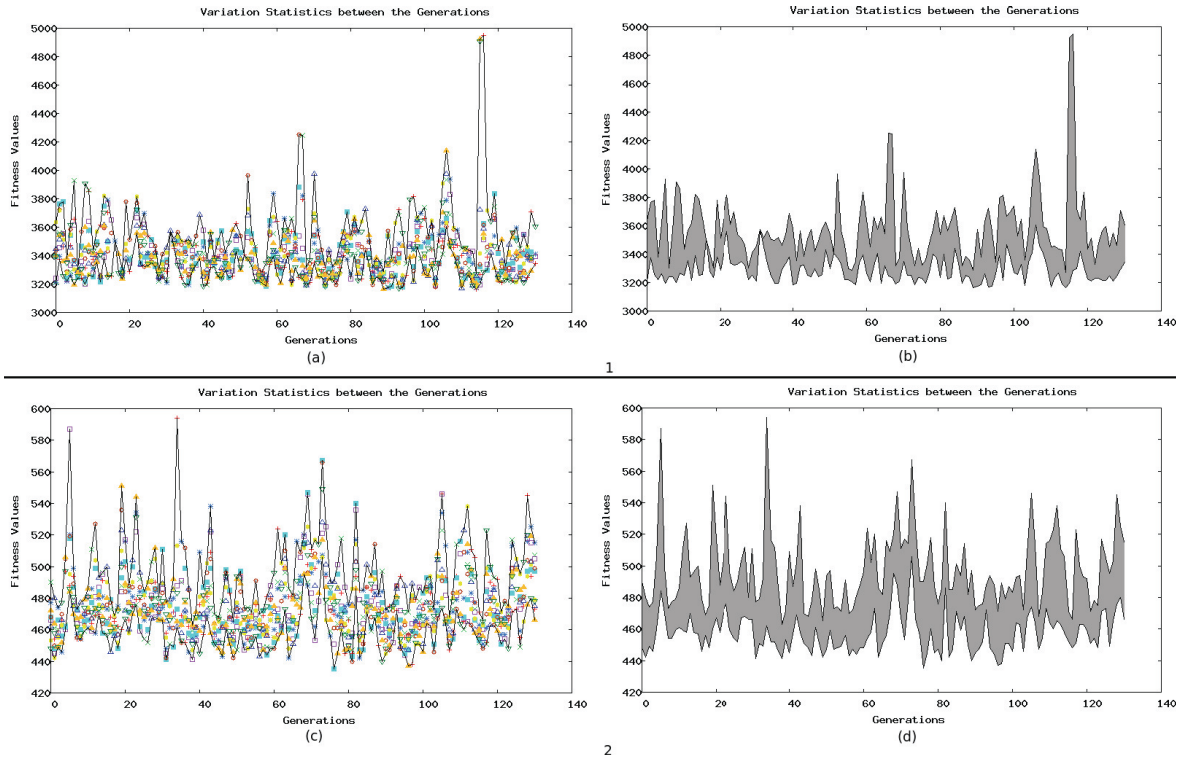


Figura B.20. Simulated Annealing aplicado ao Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.20-1b podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 90. Com a observação do gráfico estatístico, Figura B.20-2d podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 80.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.21 e B.22, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.6, sendo que tanto para o grupo grande, quanto para o grupo pequeno a maioria das conexões possuem três ou quatro saltos (cerca de 75%, de um total de oito conexões).

Tabela B.6. Distribuição dos segmentos da melhor arquitetura obtida com Simulated Annealing de Path Relinking de Algoritmo Genético com *crossover* por nó

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	2	0	2	4
Pequeno	0	0	2	4	2

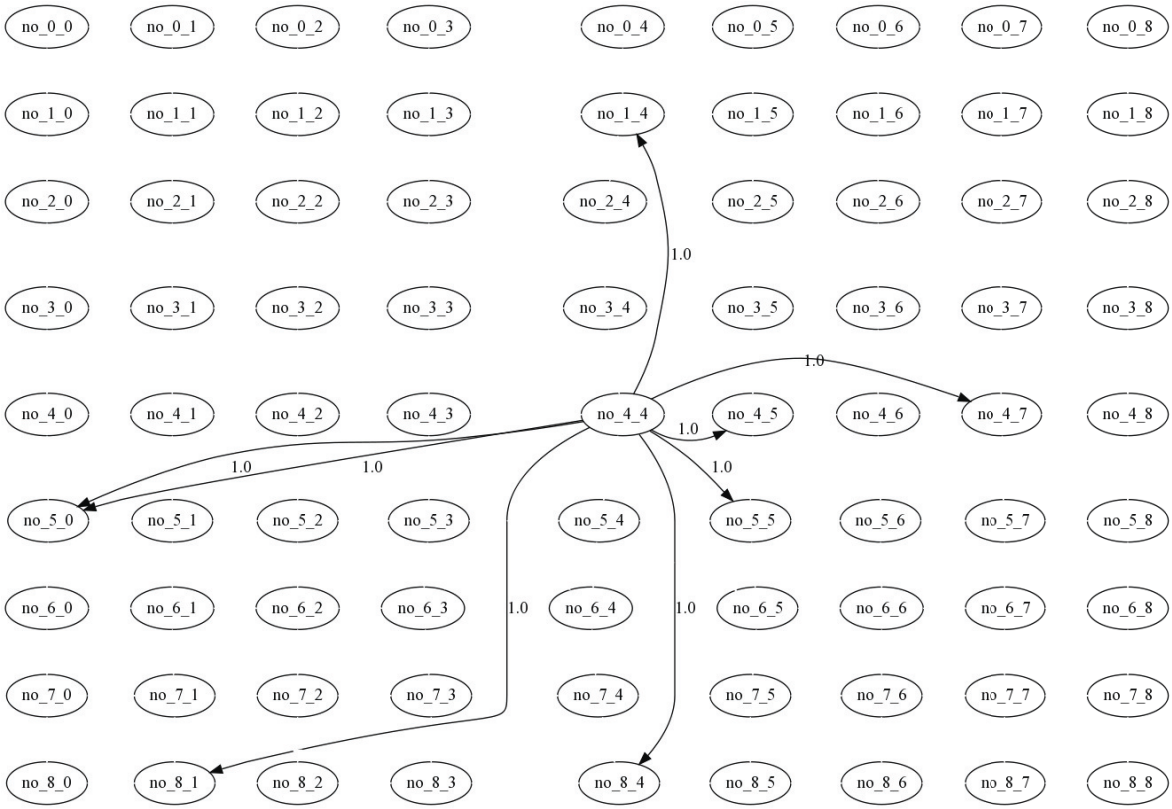


Figura B.21. Grupo Grande: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó.

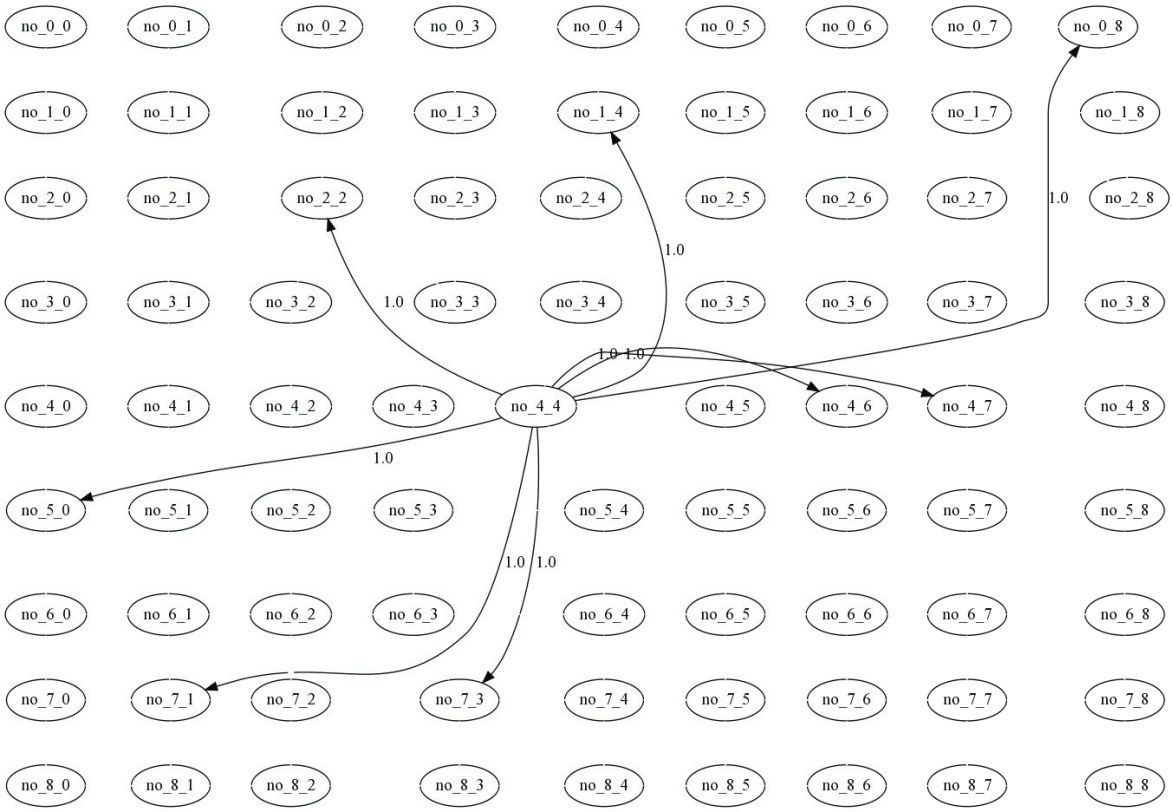


Figura B.22. Grupo Pequeno: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Path Relinking aplicado ao Algoritmo Genético com *crossover* por nó.

B.3.3 Simulated Annealing aplicado ao Algoritmo ALAP Limitado

Com a observação do gráfico de concentração, Figura B.23-1a podemos verificar uma maior concentração de resultados variando entre 3300 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.23-2c podemos verificar uma maior concentração de resultados variando entre 450 e 500 segmentos.

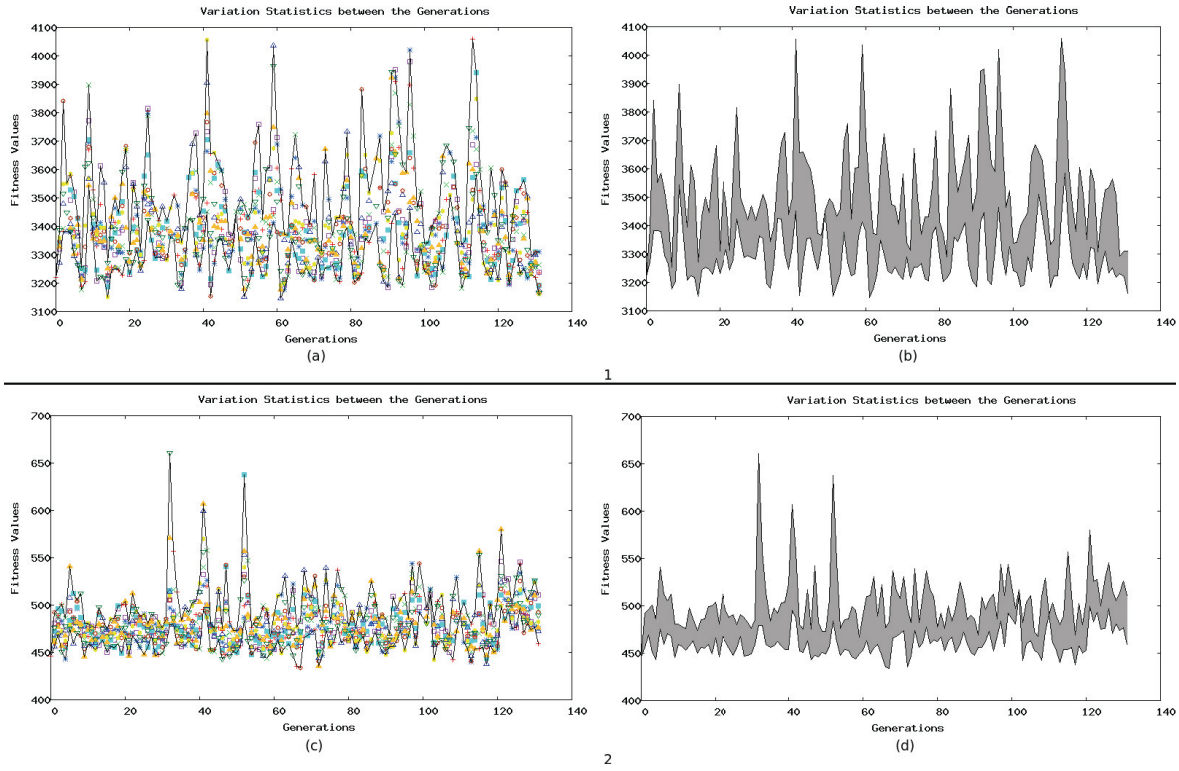


Figura B.23. Simulated Annealing aplicado a topologia extraída com o Algoritmo ALAP Limitado – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.23-1b podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 20. Com a observação do gráfico estatístico, Figura B.23-2d podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 60.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.24 e B.25, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.7, sendo que para o grupo grande a maioria dos segmentos possuem três ou quatro saltos (cerca de 87,5% de um total de oito conexões) e para o grupo pequeno a maioria dos segmentos possuem dois ou quatro saltos (cerca de 87,5% de um total de oito conexões).

Tabela B.7. Distribuição dos segmentos da melhor arquitetura obtida com Simulated Annealing de ALAP Limitado

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	0	0	1	2	5
Pequeno	0	0	4	1	3

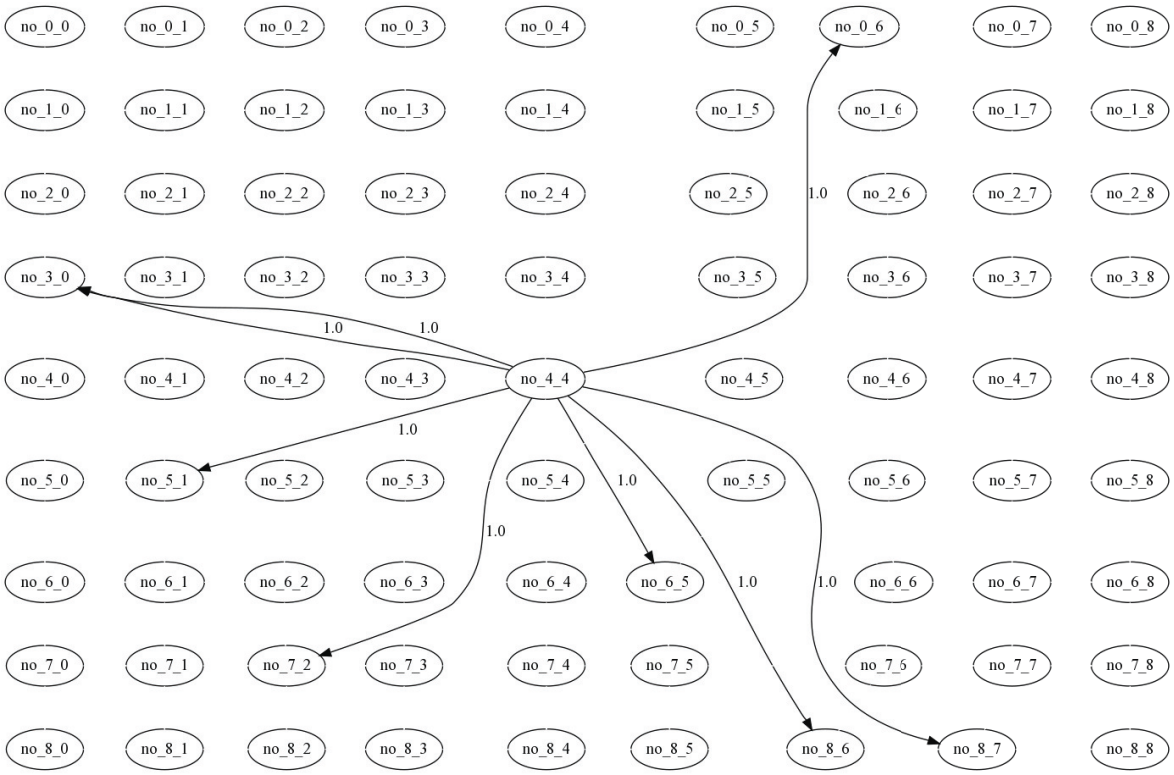


Figura B.24. Grupo Grande: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Algoritmo ALAP Limitado.

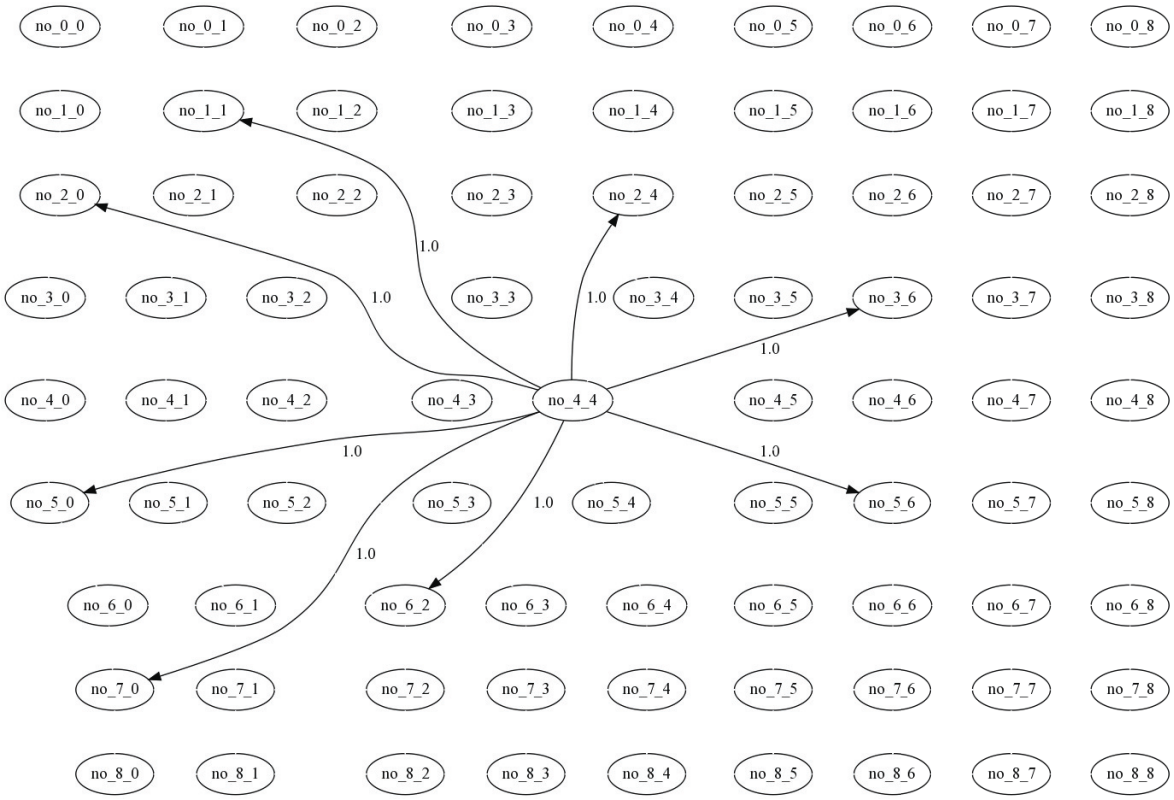


Figura B.25. Grupo Pequeno: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Algoritmo ALAP Limitado.

B.3.4 Simulated Annealing aplicado ao Algoritmo ASAP

Com a observação do gráfico de concentração, Figura B.26-1a podemos verificar uma maior concentração de resultados variando entre 3200 e 3500 segmentos. Com a observação do gráfico de concentração, Figura B.26-2c podemos verificar uma maior concentração de resultados variando entre 450 e 490 segmentos.

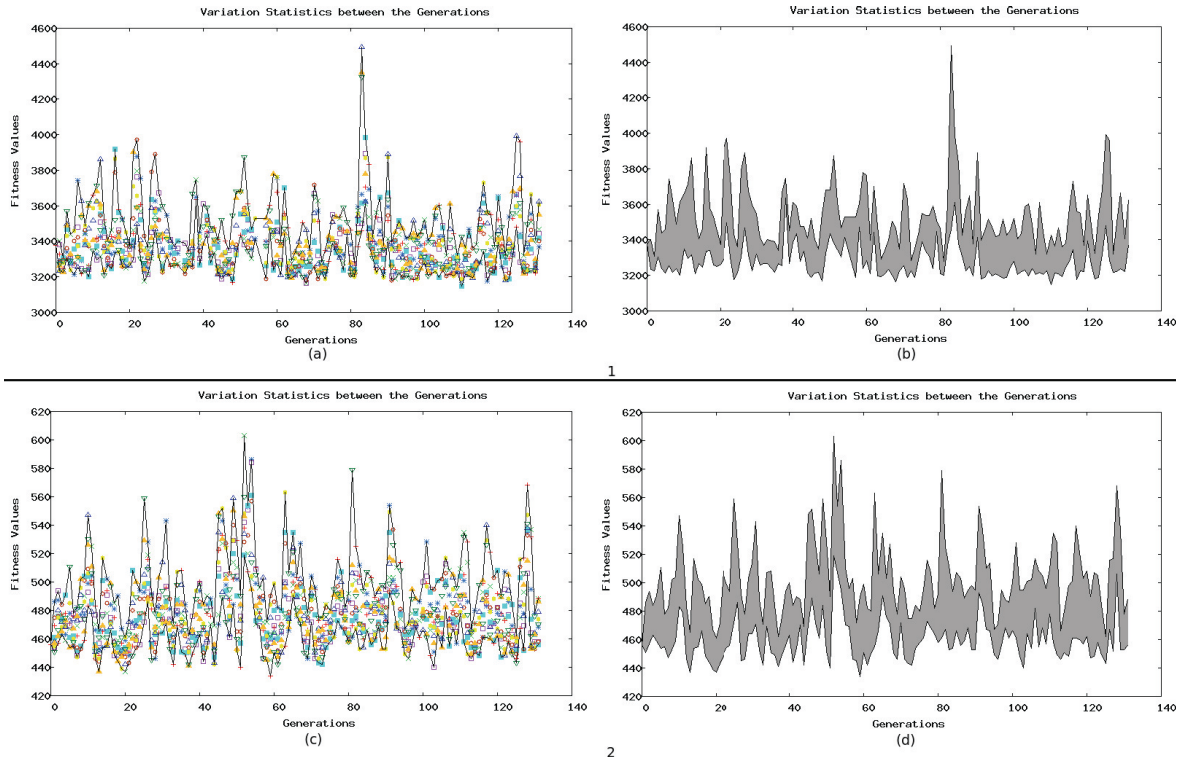


Figura B.26. Simulated Annealing aplicado a topologia extraída com o Algoritmo ASAP – (a) e (c) Gráficos de Concentração; (b) e (d) Gráficos Estatísticos: 1 Grupo Grande; 2 Grupo Pequeno.

Já com a observação do gráfico estatístico, Figura B.26-1b podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 110. Com a observação do gráfico estatístico, Figura B.26-2d podemos verificar que aparentemente o melhor resultado se deu por volta da mudança de temperatura número 60.

O formato das melhores topologias das arquiteturas encontradas para os grupos grande e pequeno podem ser visualizadas nas Figuras B.27 e B.28, respectivamente e a distribuição do tamanho de cada segmento está na Tabela B.8, sendo que para o grupo grande a maioria das conexões possuem dois ou três saltos (cerca de 62,5% de um total de oito conexões) e para o grupo pequeno a maioria das conexões possuem três ou quatro saltos (cerca de 75% de um total de oito conexões).

Tabela B.8. Distribuição dos segmentos da melhor arquitetura obtida com Simulated Annealing de ASAP.

	Saltos				
Segmentos por grupo	0	1	2	3	4
Grande	1	0	3	2	2
Pequeno	0	0	2	3	3

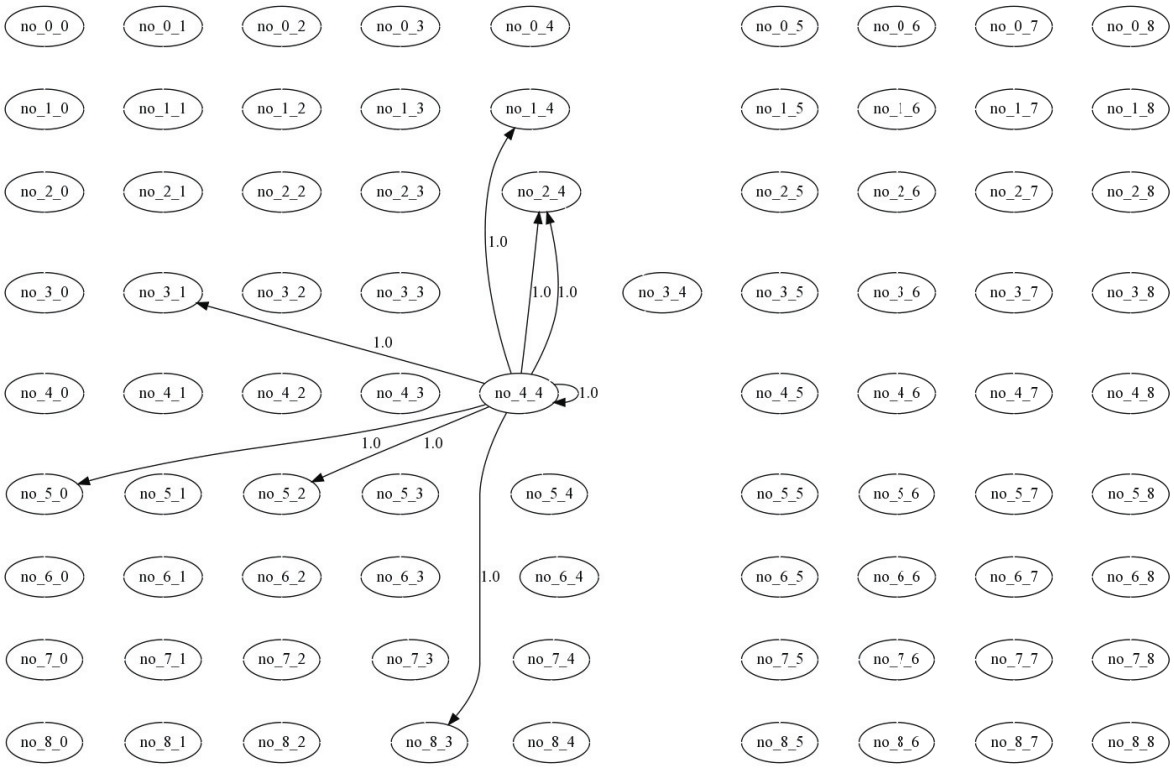


Figura B.27. Grupo Grande: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Algoritmo ASAP.

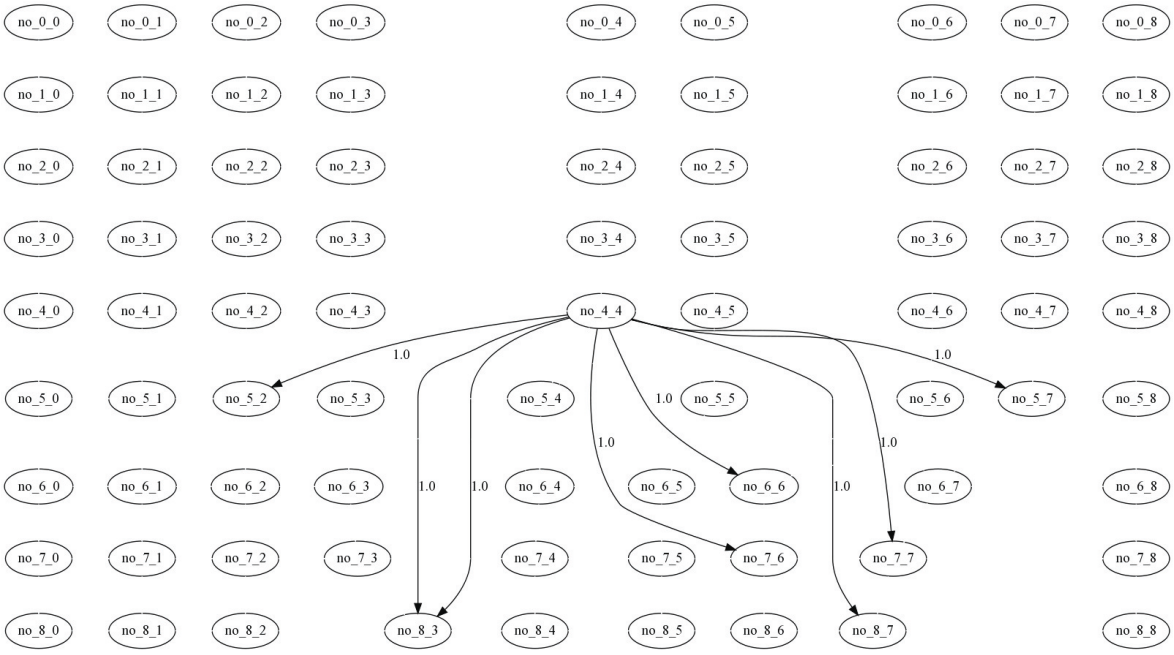


Figura B.28. Grupo Pequeno: Topologia da melhor arquitetura para o Simulated Annealing aplicado ao Algoritmo ASAP.

Anexo C

Dataflows

Nesse anexo é apresentado o código completo do processo de transformação de um algoritmo da área de processamento de sinais digitais em um dataflow a ordem seguida é a mesma detalhada na seção 3.1, ou seja, o código em C transformado em uma especificação XML que através de um XML-Schema é transformado em código java, seguindo um padrão de grafo de fluxo, que é utilizado através de carregamento dinâmico de classe para o mapeamento em uma arquitetura.

O componente java denominado jaxb é utilizado para transformar o XML original em um arquivo DOT (Gansner et al., 1993) para possibilitar a visualização do fluxo do dataflow gerado.

Nas próximas seções são apresentadas as quatro visões dos algoritmos utilizados nesse trabalho, sendo: o código em linguagem C, o dataflow em XML, o dataflow em código java e a representação gráfica do dataflow, essas seções representam, ainda, as duas formas de obtenção de dataflows utilizadas, a extração manual (Ferreira et al., 2004; Guerra, 2008) e a extração automática (Extensible & Group, 2008; Dick et al., 1998).

Na primeira seção apresentamos os dataflows que foram codificados de forma manual por Ferreira et al. (2004), ou seja, a partir do algoritmo na linguagem C Ferreira criou o dataflow correspondente. Nessa seção

Na segunda seção apresentamos os dataflows que foram extraídos de forma automática pelo aplicativo mediabench (Extensible & Group, 2008) e disponibilizados de forma gratuita na internet.

C.1 Codificados manualmente (Ferreira et al., 2004)

Algoritmo C.1: SNN

```
#include <ToolsImage.h>
namespace tools{
    template <class ImageType, class AccessRGB>
    void FilterSNNcore(ImageType &Input, ImageType &Output, const tools::CROI &ROI,
      std::vector<tools::CROI>& coord){
        int sz = coord.size();
        for (unsigned int y = ROI.y1(); y < ROI.y2(); ++y){
            for (unsigned int x = ROI.x1(); x < ROI.x2(); ++x){
                int mr = 0, mg = 0, mb = 0;
                for (char a = 0; a < sz; ++a) {
                    unsigned char cr = AccessRGB::GetR(Input(x,y)),
                    cg = AccessRGB::GetG(Input(x,y)), cb = AccessRGB::GetB(Input(x,y)),
                    r1 = AccessRGB::GetR(Input(x+coord[a].x1(),y+coord[a].y1())),
                    g1 = AccessRGB::GetG(Input(x+coord[a].x1(),y+coord[a].y1())),
                    b1 = AccessRGB::GetB(Input(x+coord[a].x1(),y+coord[a].y1())),
                    r2 = AccessRGB::GetR(Input(x+coord[a].x2(),y+coord[a].y2())),
                    g2 = AccessRGB::GetG(Input(x+coord[a].x2(),y+coord[a].y2())),
                    b2 = AccessRGB::GetB(Input(x+coord[a].x2(),y+coord[a].y2()));
                    if (tools::DistRGBEuklidPow2(cr, cg, cb, r1, g1, b1) <
                      tools::DistRGBEuklidPow2(cr, cg, cb, r2, g2, b2)){
                        mr += r1; mg += g1; mb += b1;
                    } else {
                        mr += r2; mg += g2; mb += b2;
                    }
                }
                Output(x,y) = AccessRGB::PackRGB(mr/4, mg/4, mb/4);
            }
        }
    }
    template <class ImageType, class AccessRGB>
    void FilterSNN(ImageType &Input, ImageType &Output) {
        int w = Input.GetWidth(), h = Input.GetHeight();
        if (Output.GetWidth() != w || Output.GetHeight() != h) {
            Output.SetSize(w,h);
        }
        std::vector<tools::CROI> coord;
        coord.resize(4);
        coord[0] = tools::CROI(-1, -1, +1, +1);
        coord[1] = tools::CROI( 0, -1, 0, +1);
        coord[2] = tools::CROI(+1, -1, -1, +1);
        coord[3] = tools::CROI(-1, 0, +1, 0);
        FilterSNNcore<ImageType, AccessRGB>(Input, Output, tools::CROI(1, 1, w-1, h-1), coord);
        for (int y = 0; y < h; y++) {
            Output(0,y) = Input(0,y);
            Output(w-1,y) = Input(w-1,y);
        }
        for (int x = 0; x < w; x++) {
            Output(x,0) = Input(x,0);
            Output(x,h-1) = Input(x,h-1);
        }
    }
};
```

Algoritmo C.2: Dataflow XML do SNN

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="snn">
    <COMPONENT unit="cap10" operation="cap10" name="cap10"/>
    <COMPONENT unit="cbp3mb" operation="cbp3mb" name="cbp3mb"/>
    <COMPONENT unit="andrdp1" operation="andrdp1" name="andrdp1"/>
    <COMPONENT unit="cap1b2" operation="cap1b2" name="cap1b2"/>
    <COMPONENT unit="sbcp3" operation="sbcp3" name="sbcp3"/>
    <COMPONENT unit="mdp3mg2" operation="mdp3mg2" name="mdp3mg2"/>
    <COMPONENT unit="sbdp3" operation="sbdp3" name="sbdp3"/>
    <COMPONENT unit="mcp3mb2" operation="mcp3mb2" name="mcp3mb2"/>
    <COMPONENT unit="map1mg" operation="map1mg" name="map1mg"/>
    <COMPONENT unit="addp11" operation="addp11" name="addp11"/>
    <COMPONENT unit="out" operation="out" name="out_0"/>
    ⋮
    <COMPONENT unit="cap1g2" operation="cap1g2" name="cap1g2"/>
    <COMPONENT unit="cbp1b2" operation="cbp1b2" name="cbp1b2"/>
    <COMPONENT unit="ccp11" operation="ccp11" name="ccp11"/>
    <COMPONENT unit="adbp30" operation="adbp30" name="adbp30"/>
    <COMPONENT unit="cap1g2" operation="cap1g2" name="cap1g2"/>
    <COMPONENT unit="adcp11" operation="adcp11" name="adcp11"/>
    <COMPONENT unit="mcp3mr2" operation="mcp3mr2" name="mcp3mr2"/>
    <SIGNAL name="edge_0">
        <SOURCE name="map1mg"/>
        <SINK name="sap1mg"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="andbap1"/>
        <SINK name="sbap1"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="andgbp3"/>
        <SINK name="sgbp3"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="andrpix2"/>
        <SINK name="c_px2r"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="cap3g2"/>
        <SINK name="map3mg2"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="pix_0"/>
        <SINK name="cpix2_0"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="sbcp1"/>
        <SINK name="ccp1mb"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="mbp1mg"/>
        <SINK name="sbp1mg"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="cdp11"/>
        <SINK name="andgdp1"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="cbpx2r"/>
        <SINK name="cbp1r2"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE name="cap3mb"/>
        <SINK name="map3mb"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_293">
        <SOURCE name="ccp3mb"/>
        <SINK name="mcp3mb"/>
    </SIGNAL>
    <SIGNAL name="edge_294">
        <SOURCE name="cbp12"/>
        <SINK name="andrbp1"/>
    </SIGNAL>
</DESIGN>
```

Algoritmo C.3: Dataflow java do SNN

```
package architectures.dataflows.javagraphs;
import architectures.dataflows.graph.Node;
import architectures.dataflows.graph.RGraph;
public class snn extends DataflowGraph {
 RGraph graph = new RGraph();
 @Override
 public RGraph getGraph() {
  return graph;
 }
 public snn() {
  try {
    Node cap11 = new Node("cap11", "COPY", "ALU");
    graph.addVertex(cap11);
    Node cap12 = new Node("cap12", "COPY", "ALU");
    graph.addVertex(cap12);
    Node andrap1 = new Node("andrap1", "IAND", "ALU");
    graph.addVertex(andrap1);
andrap1.setPortValue("B", "255");
    Node andgap1 = new Node("andgap1", "IAND", "ALU");
    graph.addVertex(andgap1);
andgap1.setPortValue("B", "65280");
    Node andbap1 = new Node("andbap1", "IAND", "ALU");
    graph.addVertex(andbap1);
andbap1.setPortValue("B", "16711680");
    Node sgap1 = new Node("sgap1", "ISHR", "ALU");
    graph.addVertex(sgap1);
sgap1.setPortValue("B", "8");
    Node sbap1 = new Node("sbap1", "ISHR", "ALU");
    graph.addVertex(sbap1);
sbap1.setPortValue("B", "16");
    Node cap1mr = new Node("cap1mr", "COPY", "ALU");
    graph.addVertex(cap1mr);
    Node cap1mg = new Node("cap1mg", "COPY", "ALU");
    graph.addVertex(cap1mg);
    Node cap1mb = new Node("cap1mb", "COPY", "ALU");
    graph.addVertex(cap1mb);
    Node map1mr = new Node("map1mr", "IMUL", "MULT");
    graph.addVertex(map1mr);
    Node map1mg = new Node("map1mg", "IMUL", "MULT");
    graph.addVertex(map1mg);
    Node map1mb = new Node("map1mb", "IMUL", "MULT");
    graph.addVertex(map1mb);
    Node cap1r2 = new Node("cap1r2", "COPY", "ALU");
    graph.addVertex(cap1r2);
    Node cap1g2 = new Node("cap1g2", "COPY", "ALU");
    graph.addVertex(cap1g2);
    Node cap1b2 = new Node("cap1b2", "COPY", "ALU");
    graph.addVertex(cap1b2);
    Node map1mr2 = new Node("map1mr2", "IMUL", "MULT");
    graph.addVertex(map1mr2);
    Node map1mg2 = new Node("map1mg2", "IMUL", "MULT");
    graph.addVertex(map1mg2);
    Node map1mb2 = new Node("map1mb2", "IMUL", "MULT");
    graph.addVertex(map1mb2);
    Node sap1mr = new Node("sap1mr", "ISUB", "ALU");
    graph.addVertex(sap1mr);
    Node sap1mg = new Node("sap1mg", "ISUB", "ALU");
    graph.addVertex(sap1mg);
    Node sap1mb = new Node("sap1mb", "ISUB", "ALU");
    graph.addVertex(sap1mb);
    Node adap10 = new Node("adap10", "IADD", "ALU");
    graph.addVertex(adap10);
    .
    .
    .
    graph.addEdge(pix_1, cap10);
    graph.addEdge(pix_2, cap30);
    graph.addEdge(pix_3, cbp10);
    graph.addEdge(pix_4, cbp30);
    graph.addEdge(pix_5, ccp10);
    graph.addEdge(pix_6, ccp30);
    graph.addEdge(pix_7, cdp10);
    graph.addEdge(pix_8, cdp30);
    graph.addEdge(cap10, cap11);
    graph.addEdge(cap11, cap12);
    graph.addEdge(cap11, andgap1);
    graph.addEdge(cap12, andrap1);
    graph.addEdge(cap12, andbap1);
    graph.addEdge(andgap1, sgap1);
    graph.addEdge(andbap1, sbap1);
    graph.addEdge(andrap1, cap1mr);
    graph.addEdge(sgap1, cap1mg);
    graph.addEdge(sbap1, cap1mb);
    graph.addEdge(cap1mr, map1mr);
    graph.addEdge(cap1mr, map1mr);
    graph.addEdge(cap1mg, map1mg);
    graph.addEdge(cap1mg, map1mg);
    graph.addEdge(cap1mb, map1mb);
    graph.addEdge(cap1mb, map1mb);
    graph.addEdge(capx2r, cap1r2);
    graph.addEdge(capx2g, cap1g2);
    graph.addEdge(capx2b, cap1b2);
    graph.addEdge(cap1r2, map1mr2);
    graph.addEdge(cap1r2, map1mr2);
    graph.addEdge(cap1g2, map1mg2);
    graph.addEdge(cap1g2, map1mg2);
    graph.addEdge(cap1b2, map1mb2);
    graph.addEdge(cap1b2, map1mb2);
    graph.addEdge(map1mr, sap1mr);
    graph.addEdge(map1mg, sap1mg);
    graph.addEdge(map1mb, sap1mb);
    graph.addEdge(map1mr2, sap1mr);
    graph.addEdge(map1mg2, sap1mg);
    graph.addEdge(map1mb2, sap1mb);
    graph.addEdge(sap1mr, adap10);
    graph.addEdge(sap1mb, adap10);
    graph.addEdge(sap1mg, adap11);
    graph.addEdge(adap10, adap11);
    graph.addEdge(cap10, muxa);
    graph.addEdge(cap30, cap31);
    .
    .
    .
  } catch (Exception e) { }
 }
}
```

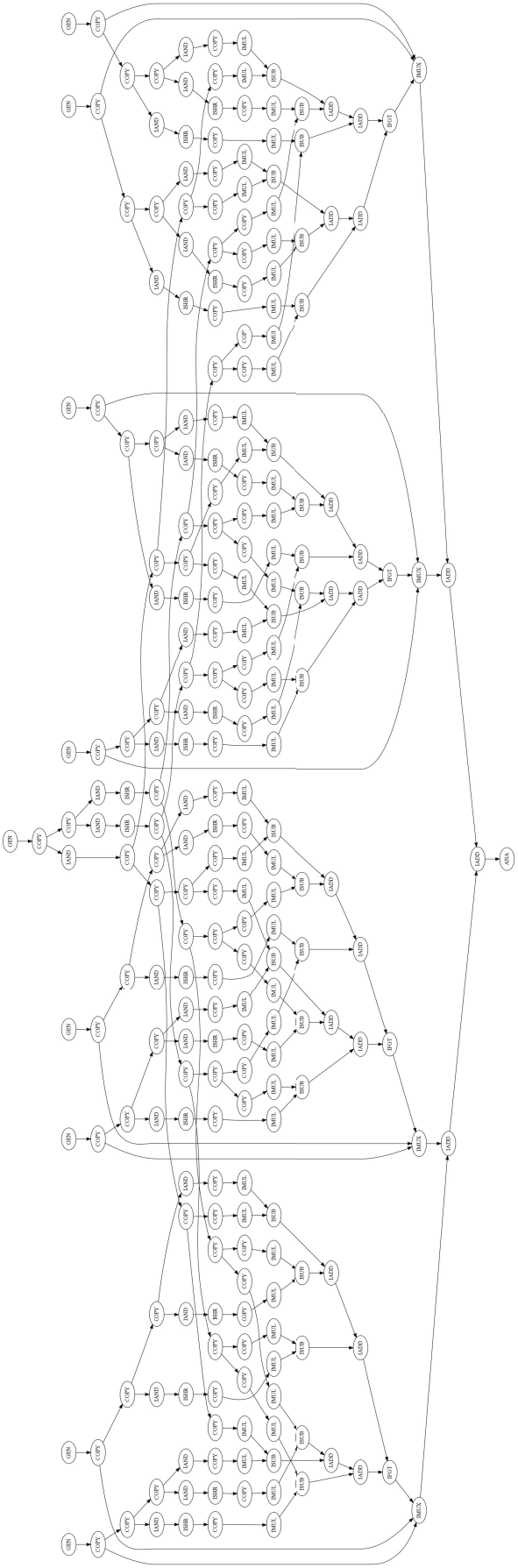


Figura C.1. Gráfico do dataflow do SNN

Algoritmo C.4: FDCT

```
#ifndef SW
#include "XPP.h"
#define TP
#else
#include <sys/types.h>
#include <time.h>
#endif
#define N 8
#define M N*N
#define num_fdcts 4
#define SIZE num_fdcts*M
short dct_io_ptr[SIZE];
#ifdef EXT_ALLOC
#pragma extern dct_io_ptr 1
#endif short dct_io_tmp[SIZE];
#ifdef EXT_ALLOC
#pragma extern dct_io_tmp 3
#endif short dct_o[SIZE];
#ifdef EXT_ALLOC
#pragma extern dct_o 2
#endif
main() {
 int tmp;
 const unsigned short c1 = 0x2C62, c3 = 0x25A0;
 const unsigned short c5 = 0x1924, c7 = 0x08D4;
 const unsigned short c0 = 0xB505, c2 = 0x29CF;
 const unsigned short c6 = 0x1151;
 short f0, f1, f2, f3,f4, f5;
 short f6, f7, Q0, Q1, S0, S1;
 int g0, g1, h0, h1,p0, p1;
 int q0a, s0a, q0, q1,s0, s1;
 short r0, r1; int P0, P1, R0;
 short R1, g2, g3, h2, h3;
 int F0, F1, F2, F3, F4, F5, F6, F7;
 int F0r, F1r, F2r, F3r, F4r, F5r, F6r, F7r;
 unsigned i, j, i_1;
 #ifdef SW time_t t0, t1;
 clock_t c01, c11;
 for (i = 0; i < num_fdcts*M; i++) {
  dct_io_ptr[i] = i;
 }
 t0 = time(NULL); c01 = clock();
 #endif i_1 = 0;
 for (i = 0; i < num_fdcts; i++) {
  for (j = 0; j < N; j++) {
   f0 = dct_io_ptr[ 0+i_1];
   f1 = dct_io_ptr[ 8+i_1];
   f2 = dct_io_ptr[16+i_1];
   f3 = dct_io_ptr[24+i_1];
   f4 = dct_io_ptr[32+i_1];
   f5 = dct_io_ptr[40+i_1];
   f6 = dct_io_ptr[48+i_1];
   f7 = dct_io_ptr[56+i_1];
   g0 = f0 + f7; h2 = f0 - f7; g1 = f1 + f6;
   h3 = f1 - f6; h1 = f2 + f5; g3 = f2 - f5;
   h0 = f3 + f4; g2 = f3 - f4; p0 = g0 + h0;
   r0 = g0 - h0; p1 = g1 + h1; r1 = g1 - h1;
   q1 = g2; s1 = h2; s0a= h3 + g3; q0a= h3 - g3;
   s0 = (s0a * c0 + 0x7FFF) >> 16;
   q0 = (q0a * c0 + 0x7FFF) >> 16;
   P0 = p0 + p1; P1 = p0 - p1;
   R1 = c6 * r1 + c2 * r0;
   R0 = c6 * r0 - c2 * r1; Q1 = q1 + q0;
   Q0 = q1 - q0; S1 = s1 + s0; S0 = s1 - s0;
   F0 = P0; F4 = P1; F2 = R1; F6 = R0;
   F1 = c7 * Q1 + c1 * S1; F7 = c7 * S1 - c1 * Q1;
   F5 = c3 * Q0 + c5 * S0; F3 = c3 * S0 - c5 * Q0;
   dct_io_tmp[ 0+i_1] = F0;
   dct_io_tmp[ 8+i_1] = F1 >> 13;
   dct_io_tmp[16+i_1] = F2 >> 13;
   dct_io_tmp[24+i_1] = F3 >> 13;
   dct_io_tmp[32+i_1] = F4;
   dct_io_tmp[40+i_1] = F5 >> 13;
   dct_io_tmp[48+i_1] = F6 >> 13;
   dct_io_tmp[56+i_1] = F7 >> 13;
   i_1++;
  }
  i_1 += 56;
 }
 #ifdef TP XPP_next_conf();
 #endif i_1 = 0;
 for (i = 0; i < N*num_fdcts; i++) {
  f0 = dct_io_tmp[0+i_1]; f1 = dct_io_tmp[1+i_1];
  f2 = dct_io_tmp[2+i_1]; f3 = dct_io_tmp[3+i_1];
  f4 = dct_io_tmp[4+i_1]; f5 = dct_io_tmp[5+i_1];
  f6 = dct_io_tmp[6+i_1]; f7 = dct_io_tmp[7+i_1];
  g0 = f0 + f7; h2 = f0 - f7; g1 = f1 + f6;
  h3 = f1 - f6; h1 = f2 + f5; g3 = f2 - f5;
  h0 = f3 + f4; g2 = f3 - f4; p0 = g0 + h0;
  r0 = g0 - h0; p1 = g1 + h1; r1 = g1 - h1;
  q1 = g2; s1 = h2; s0a= h3 + g3; q0a= h3 - g3;
  q0 = (q0a * c0 + 0x7FFF) >> 16;
  s0 = (s0a * c0 + 0x7FFF) >> 16;
  P0 = p0 + p1; P1 = p0 - p1; F6 = R0;
  R1 = c6 * r1 + c2 * r0; R0 = c6 * r0 - c2 * r1;
  Q1 = q1 + q0; Q0 = q1 - q0; S1 = s1 + s0;
  S0 = s1 - s0; F0 = P0; F4 = P1; F2 = R1;
  F1 = c7 * Q1 + c1 * S1; F7 = c7 * S1 - c1 * Q1;
  F5 = c3 * Q0 + c5 * S0; F3 = c3 * S0 - c5 * Q0;
  F0r = (F0 + 0x0006) >> 3;
  F1r = (F1 + 0x7FFF) >> 16;
  F2r = (F2 + 0x7FFF) >> 16;
  F3r = (F3 + 0x7FFF) >> 16;
  F4r = (F4 + 0x0004) >> 3;
  F5r = (F5 + 0x7FFF) >> 16;
  F6r = (F6 + 0x7FFF) >> 16;
  F7r = (F7 + 0x7FFF) >> 16;
  dct_o[0+i_1] = F0r; dct_o[1+i_1] = F1r;
  dct_o[2+i_1] = F2r; dct_o[3+i_1] = F3r;
  dct_o[4+i_1] = F4r; dct_o[5+i_1] = F5r;
  dct_o[6+i_1] = F6r; dct_o[7+i_1] = F7r;
  i_1 += 8;
 }
 #ifdef SW t1 = time(NULL);
 c11 = clock();
 #endif
}
```

Algoritmo C.5: Dataflow XML do FDCT

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="fdct">
    <COMPONENT unit="sep" operation="sep" name="sep_4"/>
    <COMPONENT unit="sub" operation="sub" name="sub_q0a"/>
    <COMPONENT unit="shr" operation="shr" name="shr_F6"/>
    <COMPONENT unit="add" operation="add" name="add_q0"/>
    <COMPONENT unit="copy" operation="copy" name="copy_f_5"/>
    <COMPONENT unit="sub" operation="sub" name="sub_g3"/>
    <COMPONENT unit="add" operation="add" name="add_h0"/>
    <COMPONENT unit="copy" operation="copy" name="copy_S1"/>
    <COMPONENT unit="copy" operation="copy" name="copy_h0"/>
    <COMPONENT unit="sub" operation="sub" name="sub_h3"/>
    <COMPONENT unit="copy" operation="copy" name="copy_f_6"/>
    <COMPONENT unit="copy" operation="copy" name="copy_f_3"/>
    <COMPONENT unit="copy" operation="copy" name="copy_g2"/>
    <COMPONENT unit="add" operation="add" name="add_g0"/>
    ⋮
    <COMPONENT unit="copy" operation="copy" name="copy_g1"/>
    <COMPONENT unit="copy" operation="copy" name="copy_g2"/>
    <COMPONENT unit="shr" operation="shr" name="shr_s0"/>
    <COMPONENT unit="pas" operation="pas" name="pas_6"/>
    <SIGNAL name="edge_0">
  <SOURCE name="sep_2"/>
        <SINK name="sep_6"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="copy_r0"/>
        <SINK name="mul_r0F6"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="shr_s0"/>
        <SINK name="copy_s0"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="copy_g0"/>
        <SINK name="add_p0"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="shr_F2"/>
        <SINK name="pas_1"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="copy_q0"/>
        <SINK name="sub_Q0"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="sub_F3"/>
        <SINK name="shr_F3"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="copy_f_7"/>
        <SINK name="sub_h2"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="copy_f_4"/>
        <SINK name="add_h0"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="sep_6"/>
        <SINK name="copy_f_7"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE name="sub_Q0"/>
        <SINK name="copy_Q0"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_122">
        <SOURCE name="sub_h3"/>
        <SINK name="copy_h3"/>
    </SIGNAL>
    <SIGNAL name="edge_123">
        <SOURCE name="copy_q0"/>
        <SINK name="add_Q1"/>
    </SIGNAL>
</DESIGN>
```

Algoritmo C.6: Dataflow java do FDCT

```
package architectures.dataflows.javagraphs;
import architectures.dataflows.graph.Node;
import architectures.dataflows.graph.RGraph;
public class DCT extends DataflowGraph {
 RGraph graph = new RGraph();
  @Override
  public RGraph getGraph() {
  return graph;
 }
 public FDCT() {
  try {
   Node X = new Node("X", "GEN", "IO");
   graph.addVertex(X);
   Node out_0 = new Node("out_0", "ANA", "IO");
   graph.addVertex(out_0);
   Node sep_0 = new Node("sep_0", "SEP", "ALU");
   graph.addVertex(sep_0);
   Node copy_f_0 = new Node("copy_f_0", "COPY",
"ALU");
   graph.addVertex(copy_f_0);
   Node sep_1 = new Node("sep_1", "SEP", "ALU");
   graph.addVertex(sep_1);
   Node copy_f_1 = new Node("copy_f_1", "COPY",
"ALU");
   graph.addVertex(copy_f_1);
   Node sep_2 = new Node("sep_2", "SEP", "ALU");
   graph.addVertex(sep_2);
   Node copy_f_2 = new Node("copy_f_2", "COPY",
"ALU");
   graph.addVertex(copy_f_2);
   Node sep_3 = new Node("sep_3", "SEP", "ALU");
   graph.addVertex(sep_3);
   Node copy_f_3 = new Node("copy_f_3", "COPY",
"ALU");
   graph.addVertex(copy_f_3);
   Node sep_4 = new Node("sep_4", "SEP", "ALU");
   graph.addVertex(sep_4);
   Node copy_f_4 = new Node("copy_f_4", "COPY",
"ALU");
   graph.addVertex(copy_f_4);
   Node sep_5 = new Node("sep_5", "SEP", "ALU");
   graph.addVertex(sep_5);
   Node copy_f_5 = new Node("copy_f_5", "COPY",
"ALU");
   graph.addVertex(copy_f_5);
   Node sep_6 = new Node("sep_6", "SEP", "ALU");
   graph.addVertex(sep_6);
   Node copy_f_6 = new Node("copy_f_6", "COPY",
"ALU");
   graph.addVertex(copy_f_6);
   Node copy_f_7 = new Node("copy_f_7", "COPY",
"ALU");
   graph.addVertex(copy_f_7);
   Node add_g0 = new Node("add_g0", "IADD",
"ALU");
   graph.addVertex(add_g0);
   Node sub_h2 = new Node("sub_h2", "ISUB",
"ALU");
   graph.addVertex(sub_h2);
   Node copy_g0 = new Node("copy_g0", "COPY",
"ALU");
   graph.addVertex(copy_g0);
   Node copy_h2 = new Node("copy_h2", "COPY",
"ALU");
   graph.addVertex(copy_h2);
   Node add_g1 = new Node("add_g1", "IADD",
"ALU");
   graph.addVertex(add_g1);
   Node sub_h3 = new Node("sub_h3", "ISUB",
"ALU");
   graph.addVertex(sub_h3);
  } catch (Exception e) { }
 }
}
```

```
   Node copy_g1 = new Node("copy_g1", "COPY",
"ALU");
   graph.addVertex(copy_g1);
   .
   .
   .
   graph.addVertex(copy_h3);
   Node add_h1 = new Node("add_h1", "IADD",
"ALU");
   graph.addVertex(add_h1);
   Node sub_g3 = new Node("sub_g3", "ISUB",
"ALU");
   graph.addVertex(sub_g3);
   Node copy_h1 = new Node("copy_h1", "COPY",
"ALU");
   graph.addVertex(copy_h1);
   Node copy_g3 = new Node("copy_g3", "COPY",
"ALU");
   graph.addVertex(copy_g3);
   Node add_h0 = new Node("add_h0", "IADD",
"ALU");
   graph.addVertex(add_h0);
   Node sub_g2 = new Node("sub_g2", "ISUB",
"ALU");
   graph.addVertex(sub_g2);
   Node copy_h0 = new Node("copy_h0", "COPY",
"ALU");
   graph.addVertex(copy_h0);
   .
   .
   .
   graph.addEdge(X, sep_0);
   graph.addEdge(sep_0, sep_1);
   graph.addEdge(sep_0, sep_2);
   graph.addEdge(sep_1, sep_3);
   graph.addEdge(sep_1, sep_4);
   graph.addEdge(sep_2, sep_5);
   graph.addEdge(sep_2, sep_6);
   graph.addEdge(sep_3, copy_f_0);
   graph.addEdge(sep_3, copy_f_4);
   graph.addEdge(sep_4, copy_f_2);
   graph.addEdge(sep_4, copy_f_6);
   graph.addEdge(sep_5, copy_f_1);
   graph.addEdge(sep_5, copy_f_5);
   graph.addEdge(sep_6, copy_f_3);
   graph.addEdge(sep_6, copy_f_7);
   graph.addEdge(copy_f_0, add_g0);
   graph.addEdge(copy_f_0, sub_h2);
   graph.addEdge(copy_f_7, add_g0);
   graph.addEdge(copy_f_7, sub_h2);
   graph.addEdge(add_g0, copy_g0);
   graph.addEdge(sub_h2, copy_h2);
   graph.addEdge(copy_f_1, add_g1);
   graph.addEdge(copy_f_1, sub_h3);
   graph.addEdge(copy_f_6, add_g1);
   graph.addEdge(copy_f_6, sub_h3);
   graph.addEdge(add_g1, copy_g1);
   graph.addEdge(sub_h3, copy_h3);
   graph.addEdge(copy_f_2, add_h1);
   .
   .
   .
```

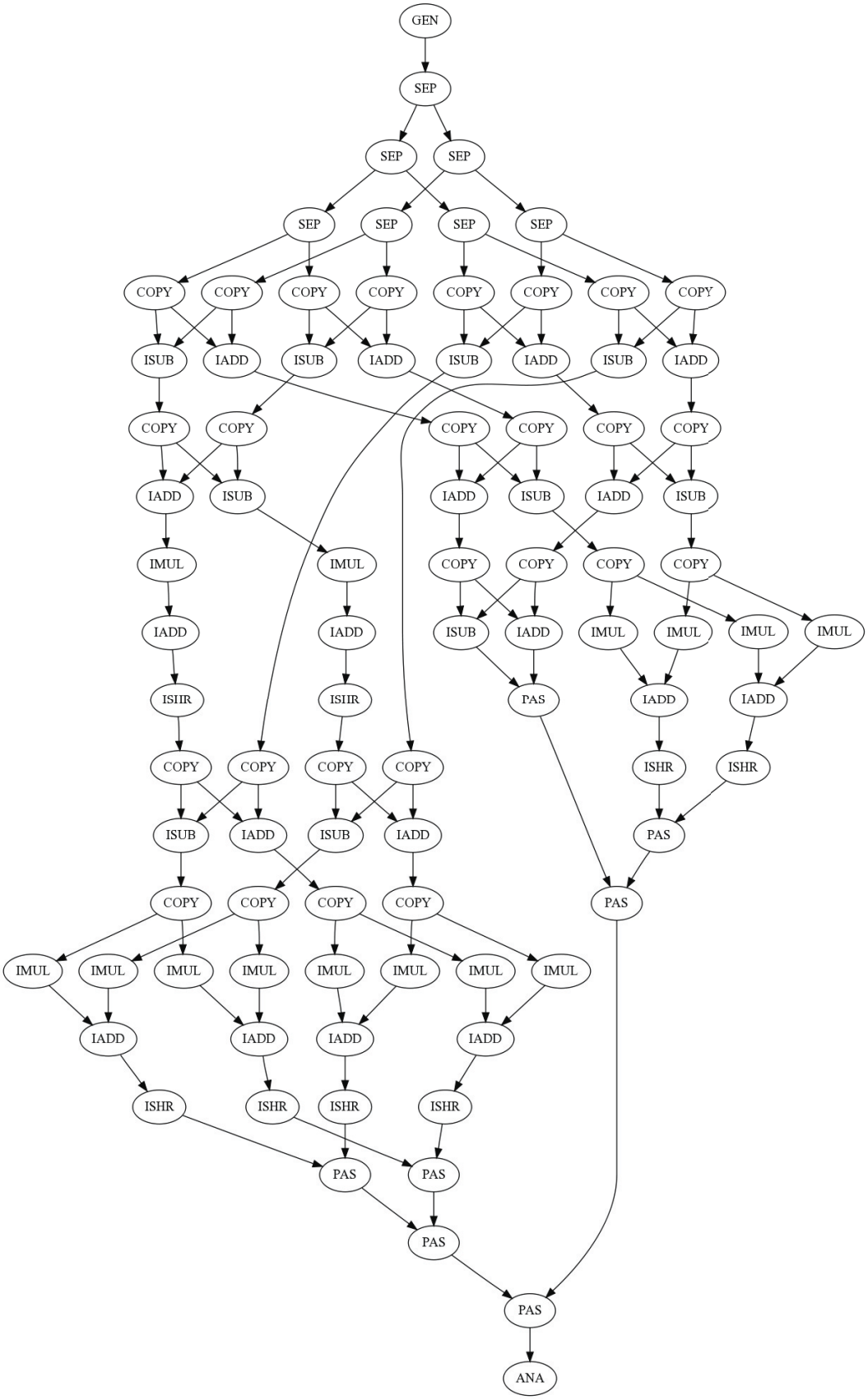


Figura C.2. Gráfico do dataflow do FDCT

C.2 Obtidos via internet (Extensible & Group, 2008)

Algoritmo C.7: interpolate_aux

```
 1 #define LINTERP( T, A, B ) ( (A) + (T) * ( (B) - (A) ) )
 2 #define EYE_SPACE 1
 3 #define CLIP_SPACE 2
 4 static GLuint Space;
 5 static void interpolate_aux( GLcontext* ctx, GLuint dst, GLfloat t, GLuint in, GLuint
   out ) {
 6     struct vertex_buffer* VB = ctx->VB;
 7     if (ctx->ClipMask & CLIP_FCOLOR_BIT) {
 8         VB->Fcolor[dst][0] = LINTERP( t, VB->Fcolor[in][0], VB->Fcolor[out][0] );
 9         VB->Fcolor[dst][1] = LINTERP( t, VB->Fcolor[in][1], VB->Fcolor[out][1] );
10         VB->Fcolor[dst][2] = LINTERP( t, VB->Fcolor[in][2], VB->Fcolor[out][2] );
11         VB->Fcolor[dst][3] = LINTERP( t, VB->Fcolor[in][3], VB->Fcolor[out][3] );
12     } else if (ctx->ClipMask & CLIP_FINDEX_BIT){
13         VB->Findex[dst] = (GLuint) (GLint) LINTERP( t, (GLfloat) VB->Findex[in],
   (GLfloat) VB->Findex[out] );
14     }
15     if (ctx->ClipMask & CLIP_BCOLOR_BIT){
16         VB->Bcolor[dst][0] = LINTERP( t, VB->Bcolor[in][0], VB->Bcolor[out][0] );
17         VB->Bcolor[dst][1] = LINTERP( t, VB->Bcolor[in][1], VB->Bcolor[out][1] );
18         VB->Bcolor[dst][2] = LINTERP( t, VB->Bcolor[in][2], VB->Bcolor[out][2] );
19         VB->Bcolor[dst][3] = LINTERP( t, VB->Bcolor[in][3], VB->Bcolor[out][3] );
20     } else if (ctx->ClipMask & CLIP_BINDEX_BIT) {
21         VB->Bindex[dst] = (GLuint) (GLint) LINTERP( t, (GLfloat) VB->Bindex[in],
   (GLfloat) VB->Bindex[out] );
22     }
23     if (ctx->ClipMask & CLIP_TEXTURE_BIT) {
24         if (Space==CLIP_SPACE) {
25             VB->Eye[dst][2] = LINTERP( t, VB->Eye[in][2], VB->Eye[out][2] );
26         }
27         VB->TexCoord[dst][0] = LINTERP(t,VB->TexCoord[in][0],VB->TexCoord[out][0]);
28         VB->TexCoord[dst][1] = LINTERP(t,VB->TexCoord[in][1],VB->TexCoord[out][1]);
29         VB->TexCoord[dst][2] = LINTERP(t,VB->TexCoord[in][2],VB->TexCoord[out][2]);
30         VB->TexCoord[dst][3] = LINTERP(t,VB->TexCoord[in][3],VB->TexCoord[out][3]);
31     }
32 }
```

Algoritmo C.8: Dataflow XML do interpolate_aux

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="interpolate_aux">
    <COMPONENT unit="ADD" operation="ADD" name="ADD_55"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_67"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_147"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_131"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_84"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_49"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_7"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_78"/>
    ⋮
    <COMPONENT unit="SUB" operation="SUB" name="SUB_80"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_25"/>
    <COMPONENT unit="STR" operation="STR" name="STR_92"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_108"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_27"/>
    <SIGNAL name="edge_0">
        <SOURCE name="MUL_85"/>
        <SINK name="ADD_89"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="MUL_127"/>
        <SINK name="ADD_128"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="ADD_97"/>
        <SINK name="ADD_99"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="ADD_55"/>
        <SINK name="LOD_56"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="MUL_13"/>
        <SINK name="ADD_20"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="ADD_128"/>
        <SINK name="STR_138"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="ADD_112"/>
        <SINK name="LOD_113"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="SUB_80"/>
        <SINK name="MUL_81"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="ADD_143"/>
        <SINK name="ADD_145"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="SUB_126"/>
        <SINK name="MUL_127"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE name="MUL_2"/>
        <SINK name="ADD_9"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_102">
        <SOURCE name="ADD_7"/>
        <SINK name="ADD_9"/>
    </SIGNAL>
    <SIGNAL name="edge_103">
        <SOURCE name="ADD_51"/>
        <SINK name="ADD_53"/>
    </SIGNAL>
</DESIGN>
```

Algoritmo C.9: Dataflow java interpolate_aux

```
package files.dataflow;
import architectures.dataflows.graph.Node;
import architectures.dataflows.graph.RGraph;
import architectures.dataflows.javagraphs.DataflowGraph;
public class interpolate_aux extends DataflowGraph {
 RGraph graph = new RGraph();
 @Override
 public RGraph getGraph() {
  return graph;
 }
 public interpolate_aux() {
  try {
    Node ADD_55 = new Node("ADD_55", "ADD", "ADD");
    graph.addVertex(ADD_55);
    Node LOD_67 = new Node("LOD_67", "LOD", "LOD");
    graph.addVertex(LOD_67);
    Node ADD_147 = new Node("ADD_147", "ADD",
"ADD");
    graph.addVertex(ADD_147);
    Node MUL_131 = new Node("MUL_131", "MULT",
"MULT");
    graph.addVertex(MUL_131);
    Node MUL_84 = new Node("MUL_84", "MULT",
"MULT");
    graph.addVertex(MUL_84);
    Node MUL_49 = new Node("MUL_49", "MULT",
"MULT");
    graph.addVertex(MUL_49);
    Node ADD_7 = new Node("ADD_7", "ADD", "ADD");
    graph.addVertex(ADD_7);
    Node LOD_78 = new Node("LOD_78", "LOD", "LOD");
    graph.addVertex(LOD_78);
    Node ADD_73 = new Node("ADD_73", "ADD", "ADD");
    graph.addVertex(ADD_73);
    Node ADD_36 = new Node("ADD_36", "ADD", "ADD");
    graph.addVertex(ADD_36);
    Node MUL_173 = new Node("MUL_173", "MULT",
"MULT");
    graph.addVertex(MUL_173);
    Node ADD_66 = new Node("ADD_66", "ADD", "ADD");
    graph.addVertex(ADD_66);
    Node MUL_71 = new Node("MUL_71", "MULT",
"MULT");
    graph.addVertex(MUL_71);
    Node ADD_31 = new Node("ADD_31", "ADD", "ADD");
    graph.addVertex(ADD_31);
    Node MUL_117 = new Node("MUL_117", "MULT",
"MULT");
    graph.addVertex(MUL_117);
    Node ADD_29 = new Node("ADD_29", "ADD", "ADD");
    graph.addVertex(ADD_29);
    Node MUL_141 = new Node("MUL_141", "MULT",
"MULT");
    graph.addVertex(MUL_141);
    Node MUL_105 = new Node("MUL_105", "MULT",
"MULT");
    graph.addVertex(MUL_105);
    Node MUL_151 = new Node("MUL_151", "MULT",
"MULT");
    graph.addVertex(MUL_151);
    Node ADD_143 = new Node("ADD_143", "ADD",
"ADD");
    graph.addVertex(ADD_143);
    Node ADD_167 = new Node("ADD_167", "ADD",
"ADD");
    graph.addVertex(ADD_167);
    Node MUL_116 = new Node("MUL_116", "MULT",
"MULT");
    graph.addVertex(MUL_116);
    Node ADD_112 = new Node("ADD_112", "ADD",
"ADD");
    graph.addVertex(ADD_112);
    Node STR_138 = new Node("STR_138", "STR",
"STR");
    graph.addVertex(STR_138);
    .
    .
    .
    graph.addEdge(MUL_85, ADD_89);
    graph.addEdge(MUL_127, ADD_128);
    graph.addEdge(ADD_97, ADD_99);
    graph.addEdge(ADD_55, LOD_56);
    graph.addEdge(MUL_13, ADD_20);
    graph.addEdge(ADD_128, STR_138);
    graph.addEdge(ADD_112, LOD_113);
    graph.addEdge(SUB_80, MUL_81);
    graph.addEdge(ADD_143, ADD_145);
    graph.addEdge(SUB_126, MUL_127);
    graph.addEdge(MUL_2, ADD_9);
    graph.addEdge(MUL_106, ADD_110);
    graph.addEdge(ADD_174, STR_184);
    graph.addEdge(SUB_172, MUL_173);
    graph.addEdge(ADD_183, STR_184);
    graph.addEdge(ADD_154, ADD_156);
    graph.addEdge(ADD_5, ADD_7);
    graph.addEdge(MUL_39, ADD_43);
    graph.addEdge(MUL_81, ADD_82);
    graph.addEdge(MUL_176, ADD_183);
    graph.addEdge(ADD_62, ADD_64);
    graph.addEdge(MUL_131, ADD_135);
    graph.addEdge(LOD_170, SUB_172);
    graph.addEdge(MUL_162, ADD_169);
    graph.addEdge(MUL_151, ADD_158);
    graph.addEdge(MUL_49, ADD_53);
    graph.addEdge(ADD_75, ADD_77);
    graph.addEdge(ADD_29, ADD_31);
    graph.addEdge(ADD_179, ADD_181);
    graph.addEdge(ADD_87, ADD_89);
    graph.addEdge(ADD_147, LOD_148);
    graph.addEdge(LOD_113, SUB_126);
    graph.addEdge(ADD_64, ADD_66);
    graph.addEdge(ADD_16, ADD_18);
    graph.addEdge(MUL_60, ADD_64);
    graph.addEdge(ADD_167, ADD_169);
    graph.addEdge(MUL_163, ADD_167);
    graph.addEdge(ADD_119, ADD_121);
    graph.addEdge(ADD_20, LOD_21);
    graph.addEdge(ADD_158, LOD_159);
    graph.addEdge(MUL_59, ADD_66);
    graph.addEdge(ADD_123, LOD_124);
    graph.addEdge(MUL_117, ADD_121);
    graph.addEdge(MUL_24, ADD_31);
    graph.addEdge(ADD_31, LOD_32);
    graph.addEdge(ADD_108, ADD_110);
    graph.addEdge(LOD_148, ADD_174);
    .
    .
    .
  } catch (Exception e) { }
 }
}
```

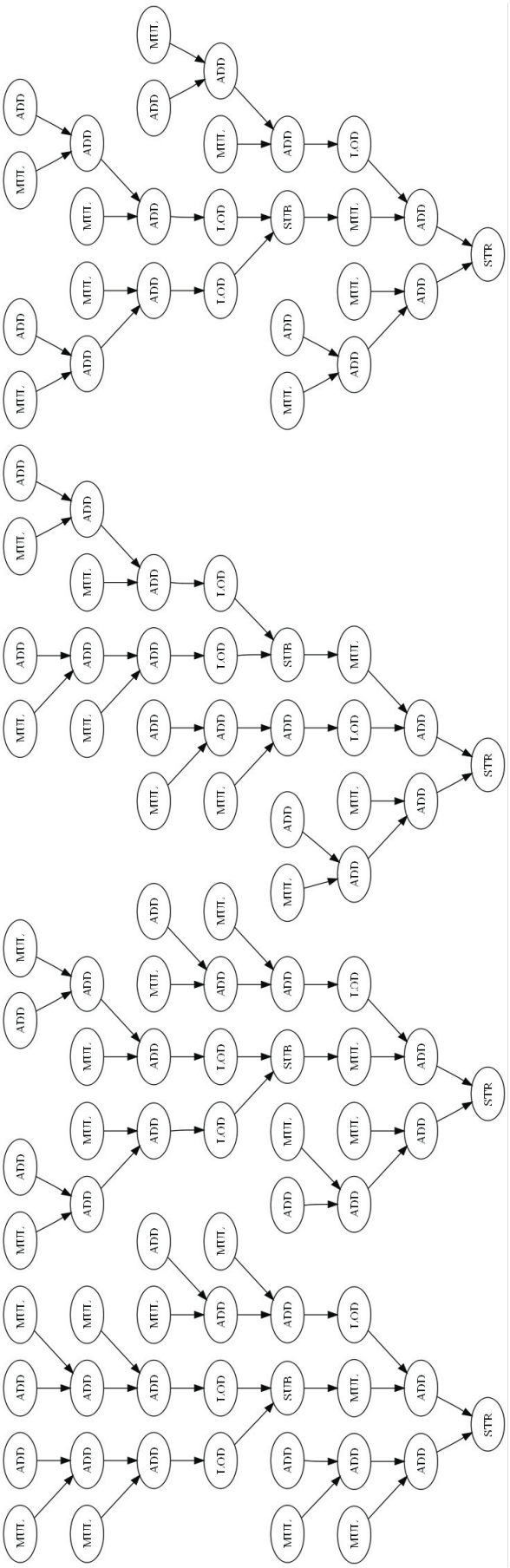


Figura C.3. Gráfico do dataflow do interpolate_aux

Algoritmo C.10: horner_bezier_surf

```
1 static void horner_bezier_surf(GLfloat *cn, GLfloat *out, GLfloat u,
  GLfloat v, GLuint dim, GLuint uorder, GLuint vorder){
2     GLfloat *cp = cn + uorder*vorder*dim;
3     GLuint i, uinc = vorder*dim;
4     if(vorder > uorder){
5         if(uorder >= 2){
6             GLfloat s, poweru;
7             GLuint j, k, bincoeff;
8             for(j=0; j<vorder; j++){
9                 GLfloat *ucp = &cn[j*dim];
10                bincoeff = uorder-1;
11                s = 1.0-u;
12                for(k=0; k<dim; k++){
13                    cp[j*dim+k] = s*ucp[k] + bincoeff*u*ucp[uinc+k];
14                }
15                for(i=2, ucp+=2*uinc, poweru=u*u; i<uorder; i++,
16                  poweru*=u, ucp +=uinc){
17                    bincoeff *= uorder-i;
18                    bincoeff /= i;
19                    for(k=0; k<dim; k++){
20                        cp[j*dim+k] = s*cp[j*dim+k] +
21                        bincoeff*poweru*ucp[k];
22                    }
23                }
24            }
25            horner_bezier_curve(cp, out, v, dim, vorder);
26        } else {
27            horner_bezier_curve(cn, out, v, dim, vorder);
28        }
29    } else {
30        if(vorder > 1){
31            GLuint i;
32            for(i=0; i<uorder; i++, cn += uinc){
33                horner_bezier_curve(cn, &cp[i*dim], v, dim, vorder);
34            }
35            horner_bezier_curve(cp, out, u, dim, uorder);
36        } else {
37            horner_bezier_curve(cn, out, u, dim, uorder);
38        }
39    }
40 }
```

Algoritmo C.11: Dataflow XML do horner_bezier_surf

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="horner_bezier_surf">
    <COMPONENT unit="STR" operation="STR" name="STR_25"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_0"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_19"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_29"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_20"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_14"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_21"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_2"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_5"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_8"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_1"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_18"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_17"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_24"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_10"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_6"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_15"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_11"/>
    <SIGNAL name="edge_0">
        <SOURCE name="MUL_10"/>
        <SINK name="MUL_17"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="LOD_15"/>
        <SINK name="MUL_17"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="ADD_20"/>
        <SINK name="MUL_21"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="LOD_6"/>
        <SINK name="MUL_8"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="MUL_21"/>
        <SINK name="ADD_24"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="ADD_14"/>
        <SINK name="LOD_15"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="ADD_24"/>
        <SINK name="STR_25"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="MUL_8"/>
        <SINK name="ADD_18"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="MUL_19"/>
        <SINK name="ADD_20"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="MUL_17"/>
        <SINK name="ADD_18"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE name="MUL_2"/>
        <SINK name="ADD_5"/>
    </SIGNAL>
    <SIGNAL name="edge_11">
        <SOURCE name="MUL_11"/>
        <SINK name="ADD_14"/>
    </SIGNAL>
    <SIGNAL name="edge_12">
        <SOURCE name="ADD_18"/>
        <SINK name="STR_25"/>
    </SIGNAL>
    <SIGNAL name="edge_13">
        <SOURCE name="ADD_1"/>
        <SINK name="MUL_2"/>
    </SIGNAL>
    <SIGNAL name="edge_14">
        <SOURCE name="ADD_5"/>
        <SINK name="LOD_6"/>
    </SIGNAL>
    <SIGNAL name="edge_15">
        <SOURCE name="MUL_0"/>
        <SINK name="ADD_1"/>
    </SIGNAL>
</DESIGN>
```

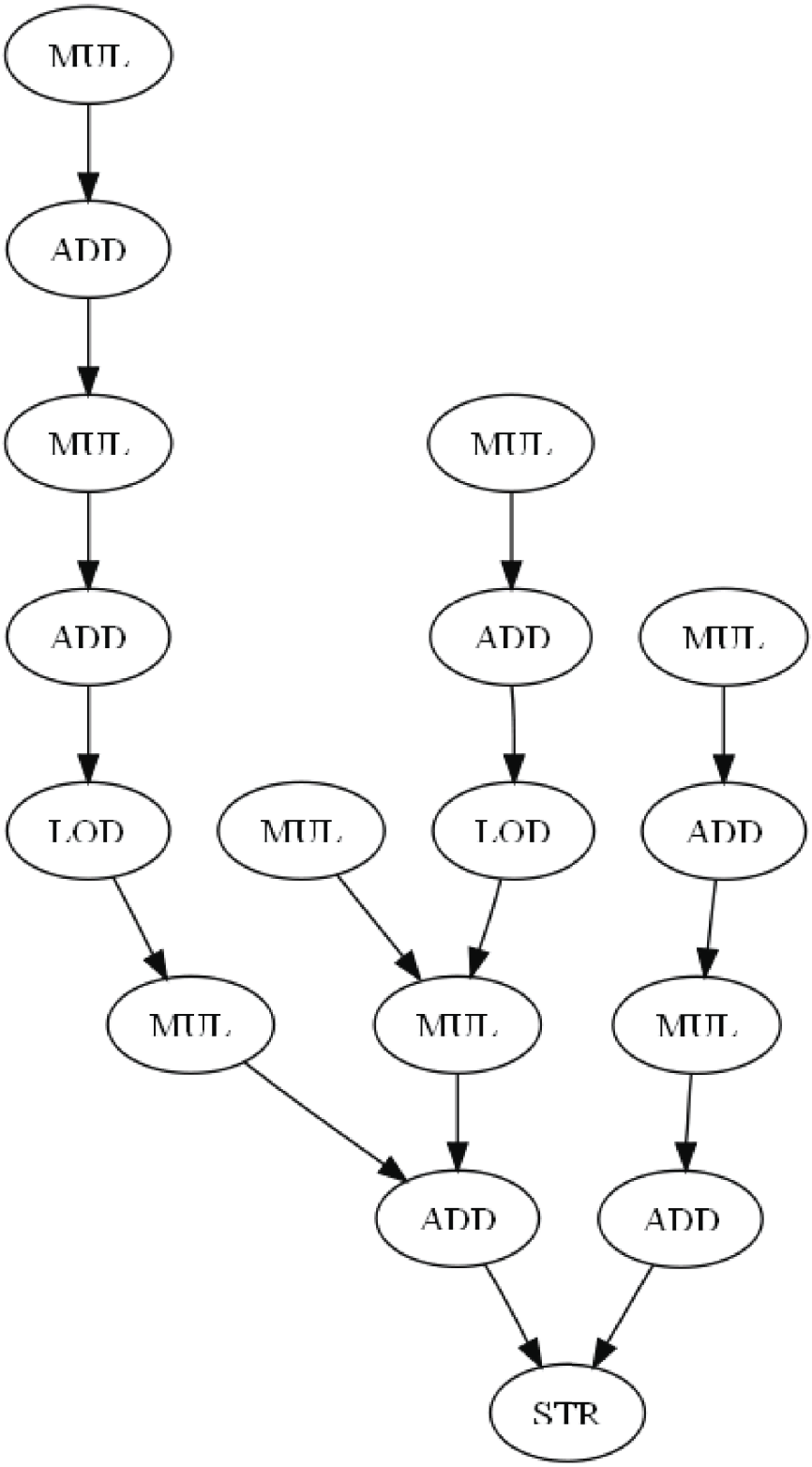


Figura C.4. Gráfico do dataflow do horner_bezier_surf

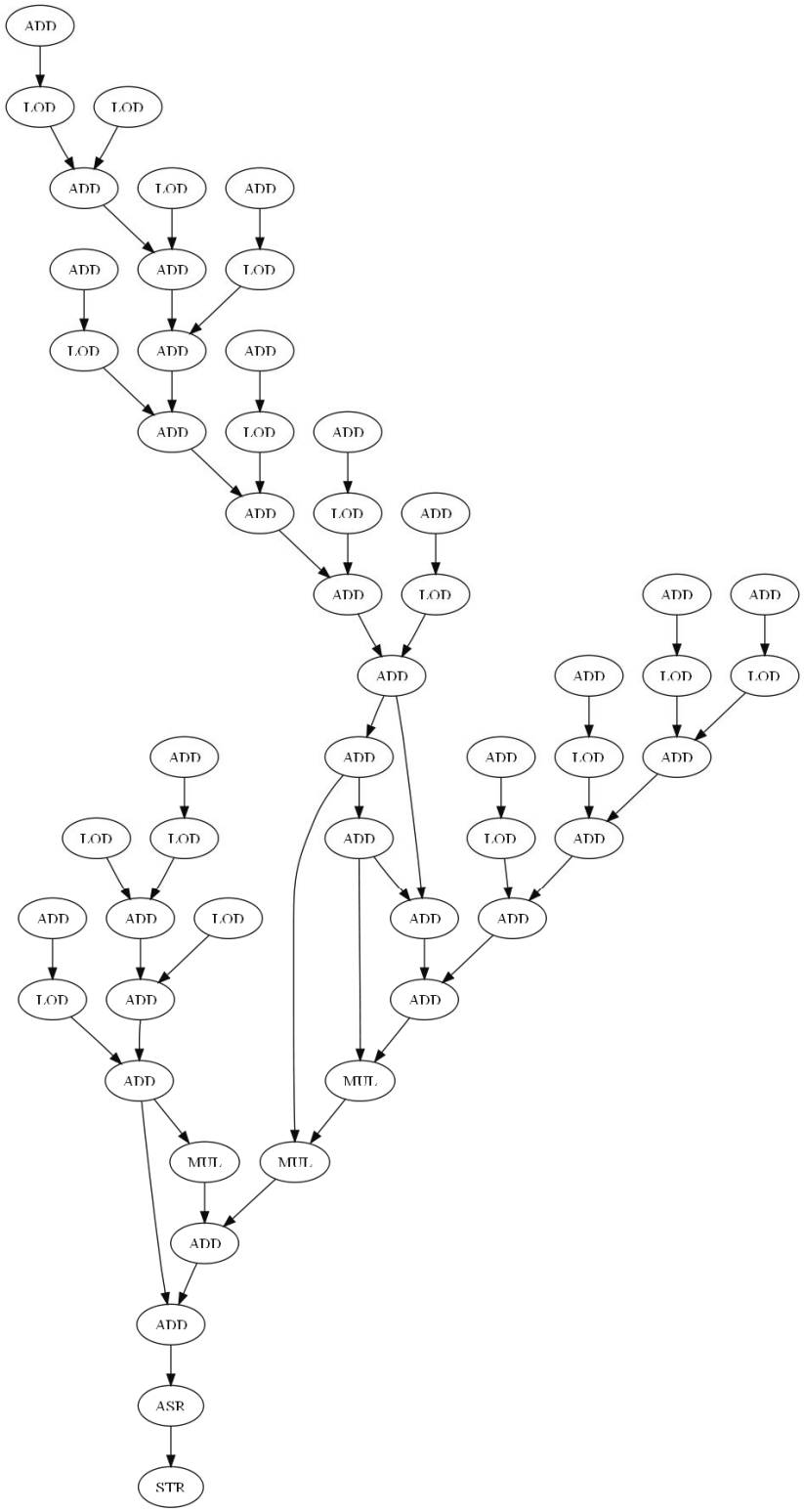


Figura C.5. Gráfico do dataflow do h2v2_smooth_downsample

Algoritmo C.12: Dataflow java horner_bezier_surf

```
package files.dataflow;
import architectures.dataflows.graph.Node;
import architectures.dataflows.graph.RGraph;
import architectures.dataflows.javagraphs.DataflowGraph;
public class horner_bezier_surf extends DataflowGraph {
 RGraph graph = new RGraph();
 @Override
 public RGraph getGraph() {
  return graph;
 }
 public horner_bezier_surf() {
  try {
   Node STR_25 = new Node("STR_25", "STR", "STR");
   graph.addVertex(STR_25);
   Node MUL_0 = new Node("MUL_0", "MULT", "MULT");
   graph.addVertex(MUL_0);
   Node MUL_19 = new Node("MUL_19", "MULT", "MULT");
   graph.addVertex(MUL_19);
   Node ADD_29 = new Node("ADD_29", "ADD", "ADD");
   graph.addVertex(ADD_29);
   Node ADD_20 = new Node("ADD_20", "ADD", "ADD");
   graph.addVertex(ADD_20);
   Node ADD_14 = new Node("ADD_14", "ADD", "ADD");
   graph.addVertex(ADD_14);
   Node MUL_21 = new Node("MUL_21", "MULT", "MULT");
   graph.addVertex(MUL_21);
   Node MUL_2 = new Node("MUL_2", "MULT", "MULT");
   graph.addVertex(MUL_2);
   Node ADD_5 = new Node("ADD_5", "ADD", "ADD");
   graph.addVertex(ADD_5);
   Node MUL_8 = new Node("MUL_8", "MULT", "MULT");
   graph.addVertex(MUL_8);
   Node ADD_1 = new Node("ADD_1", "ADD", "ADD");
   graph.addVertex(ADD_1);
   Node ADD_18 = new Node("ADD_18", "ADD", "ADD");
   graph.addVertex(ADD_18);
   Node MUL_17 = new Node("MUL_17", "MULT", "MULT");
   graph.addVertex(MUL_17);
   Node ADD_24 = new Node("ADD_24", "ADD", "ADD");
   graph.addVertex(ADD_24);
   Node MUL_10 = new Node("MUL_10", "MULT", "MULT");
   graph.addVertex(MUL_10);
   Node LOD_6 = new Node("LOD_6", "LOD", "LOD");
   graph.addVertex(LOD_6);
   Node LOD_15 = new Node("LOD_15", "LOD", "LOD");
   graph.addVertex(LOD_15);
   Node MUL_11 = new Node("MUL_11", "MULT", "MULT");
   graph.addVertex(MUL_11);
   graph.addEdge(MUL_10, MUL_17);
   graph.addEdge(LOD_15, MUL_17);
   graph.addEdge(ADD_20, MUL_21);
   graph.addEdge(LOD_6, MUL_8);
   graph.addEdge(MUL_21, ADD_24);
   graph.addEdge(ADD_14, LOD_15);
   graph.addEdge(ADD_24, STR_25);
   graph.addEdge(MUL_8, ADD_18);
   graph.addEdge(MUL_19, ADD_20);
   graph.addEdge(MUL_17, ADD_18);
   graph.addEdge(MUL_2, ADD_5);
   graph.addEdge(MUL_11, ADD_14);
   graph.addEdge(ADD_18, STR_25);
   graph.addEdge(ADD_1, MUL_2);
   graph.addEdge(ADD_5, LOD_6);
   graph.addEdge(MUL_0, ADD_1);
  } catch (Exception e) { }
 }
} labelalgo:datajhorner
```

Algoritmo C.13: h2v2_smooth_downsample

```
1  METHODDEF(void)
2  h2v2_smooth_downsample (j_compress_ptr cinfo, jpeg_component_info * compptr, JSAMPARRAY
   input_data, JSAMPARRAY output_data){
3      int inrow, outrow;
4      JDIMENSION colctr;
5      JDIMENSION output_cols = compptr->width_in_blocks * DCTSIZE;
6      register JSAMPROW inptr0, inptr1, above_ptr, below_ptr, outptr;
7      INT32 membersum, neighsum, memberscale, neighscale;
8      expand_right_edge(input_data - 1, cinfo->max_v_samp_factor + 2, cinfo->image_width,
   output_cols * 2);
9      memberscale = 16384 - cinfo->smoothing_factor * 80;
10     /* scaled (1-5*SF)/4 */
11     neighscale = cinfo->smoothing_factor * 16;
12     /* scaled SF/4 */
13     inrow = 0;
14     for (outrow = 0; outrow < compptr->v_samp_factor; outrow++){
15         outptr = output_data[outrow];
16         inptr0 = input_data[inrow];
17         inptr1 = input_data[inrow+1];
18         above_ptr = input_data[inrow-1];
19         below_ptr = input_data[inrow+2];
20         /* Special case for first column:  pretend column -1 is same as column 0 */
21         membersum = GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) + GETJSAMPLE(*inptr1) +
   GETJSAMPLE(inptr1[1]);
22         neighsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[1]) + GETJSAMPLE(*below_ptr) +
   GETJSAMPLE(below_ptr[1]) + GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[2]) + GETJSAMPLE(*inptr1) +
   GETJSAMPLE(inptr1[2]);
23         neighsum += neighsum;
24         neighsum += GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[2]) + GETJSAMPLE(*below_ptr) +
   GETJSAMPLE(below_ptr[2]);
25         membersum = membersum * memberscale + neighsum * neighscale;
26         *outptr++ = (JSAMPLE) ((membersum + 32768) >> 16);
27         inptr0 += 2; inptr1 += 2; above_ptr += 2; below_ptr += 2;
28         for (colctr = output_cols - 2; colctr > 0; colctr-){
29             /* sum of pixels directly mapped to this output element */
30             membersum = GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) + GETJSAMPLE(*inptr1) +
   GETJSAMPLE(inptr1[1]);
31             /* sum of edge-neighbor pixels */
32             neighsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[1]) + GETJSAMPLE(*below_ptr)
   + GETJSAMPLE(below_ptr[1]) + GETJSAMPLE(inptr0[-1]) + GETJSAMPLE(inptr0[2]) +
   GETJSAMPLE(inptr1[-1]) + GETJSAMPLE(inptr1[2]);
33             /* The edge-neighbors count twice as much as corner-neighbors */
34             neighsum += neighsum;
35             /* Add in the corner-neighbors */
36             neighsum += GETJSAMPLE(above_ptr[-1]) + GETJSAMPLE(above_ptr[2]) +
   GETJSAMPLE(below_ptr[-1]) + GETJSAMPLE(below_ptr[2]);
37             /* form final output scaled up by 2^16 */
38             membersum = membersum * memberscale + neighsum * neighscale;
39             /* round, descale and output it */
40             *outptr++ = (JSAMPLE) ((membersum + 32768) >> 16);
41             inptr0 += 2;
42             inptr1 += 2;
43             above_ptr += 2;
44             below_ptr += 2;
45         }
46         /* Special case for last column */
47         membersum = GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) + GETJSAMPLE(*inptr1) +
   GETJSAMPLE(inptr1[1]);
48         neighsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[1]) + GETJSAMPLE(*below_ptr) +
   GETJSAMPLE(below_ptr[1]) + GETJSAMPLE(inptr0[-1]) + GETJSAMPLE(inptr0[1]) +
   GETJSAMPLE(inptr1[-1]) + GETJSAMPLE(inptr1[1]);
49         neighsum += neighsum;
50         neighsum += GETJSAMPLE(above_ptr[-1]) + GETJSAMPLE(above_ptr[1]) +
   GETJSAMPLE(below_ptr[-1]) + GETJSAMPLE(below_ptr[1]);
51         membersum = membersum * memberscale + neighsum * neighscale;
52         *outptr = (JSAMPLE) ((membersum + 32768) >> 16);
53         inrow += 2;
54     }
55 }
```

Algoritmo C.14: Dataflow XML do h2v2_smooth_downsample

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="h2v2_smooth_downsample">
    <COMPONENT unit="ADD" operation="ADD" name="ADD_118"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_44"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_72"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_83"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_45"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_96"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_33"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_100"/>
    ⋮
    <COMPONENT unit="ADD" operation="ADD" name="ADD_32"/>
    ⋮
    <COMPONENT unit="MULT" operation="MULT" name="MUL_121"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_20"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_61"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_122"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_89"/>
    <SIGNAL name="edge_0">
        <SOURCE name="LOD_33"/>
        <SINK name="ADD_36"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="ADD_123"/>
        <SINK name="ADD_126"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="ADD_72"/>
        <SINK name="ADD_80"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="ADD_112"/>
        <SINK name="LOD_113"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="ADD_118"/>
        <SINK name="MUL_122_AUX_2"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="ADD_56"/>
        <SINK name="ADD_64"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="ADD_126"/>
        <SINK name="ASR_128"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="ADD_104"/>
        <SINK name="LOD_105"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="LOD_8"/>
        <SINK name="ADD_11"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="ADD_96"/>
        <SINK name="LOD_97"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE
   name="MUL_122_AUX_2"/>
        <SINK name="MUL_122"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_30">
        <SOURCE name="LOD_113"/>
        <SINK name="ADD_116"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_54">
        <SOURCE name="LOD_97"/>
        <SINK name="ADD_100"/>
    </SIGNAL>
</DESIGN>
```

Algoritmo C.15: Dataflow java h2v2_smooth_downsample

```
package files.dataflow;
import architectures.dataflows.graph.Node;
import architectures.dataflows.graph.RGraph;
import architectures.dataflows.javagraphs.DataflowGraph;
public class h2v2_smooth_downsample extends DataflowGraph {
 RGraph graph = new RGraph();
 @Override
 public RGraph getGraph() {
  return graph;
 }
 public h2v2_smooth_downsample() {
  try {
    Node ADD_118 = new Node("ADD_118", "ADD", "ADD");
    graph.addVertex(ADD_118);
    Node ADD_44 = new Node("ADD_44", "ADD", "ADD");
    graph.addVertex(ADD_44);
    Node ADD_72 = new Node("ADD_72", "ADD", "ADD");
    graph.addVertex(ADD_72);
    Node ADD_83 = new Node("ADD_83", "ADD", "ADD");
    graph.addVertex(ADD_83);
    Node LOD_45 = new Node("LOD_45", "LOD", "LOD");
    graph.addVertex(LOD_45);
    Node ADD_96 = new Node("ADD_96", "ADD", "ADD");
    graph.addVertex(ADD_96);
    Node LOD_33 = new Node("LOD_33", "LOD", "LOD");
    graph.addVertex(LOD_33);
    Node ADD_100 = new Node("ADD_100", "ADD", "ADD");
    graph.addVertex(ADD_100);
    Node LOD_12 = new Node("LOD_12", "LOD", "LOD");
    graph.addVertex(LOD_12);
    Node ADD_132 = new Node("ADD_132", "ADD", "ADD");
    graph.addVertex(ADD_132);
    Node ADD_104 = new Node("ADD_104", "ADD", "ADD");
    graph.addVertex(ADD_104);
    Node ADD_36 = new Node("ADD_36", "ADD", "ADD");
    graph.addVertex(ADD_36);
    Node ADD_56 = new Node("ADD_56", "ADD", "ADD");
    graph.addVertex(ADD_56);
    Node ADD_32 = new Node("ADD_32", "ADD", "ADD");
    graph.addVertex(ADD_32);
    Node ADD_76 = new Node("ADD_76", "ADD", "ADD");
    graph.addVertex(ADD_76);
    Node LOD_105 = new Node("LOD_105", "LOD", "LOD");
    graph.addVertex(LOD_105);
    Node ADD_80 = new Node("ADD_80", "ADD", "ADD");
    graph.addVertex(ADD_80);
    Node ADD_7 = new Node("ADD_7", "ADD", "ADD");
    graph.addVertex(ADD_7);
    Node ADD_15 = new Node("ADD_15", "ADD", "ADD");
    graph.addVertex(ADD_15);
    Node ADD_48 = new Node("ADD_48", "ADD", "ADD");
    graph.addVertex(ADD_48);
    Node LOD_20 = new Node("LOD_20", "LOD", "LOD");
    graph.addVertex(LOD_20);
    Node LOD_61 = new Node("LOD_61", "LOD", "LOD");
    graph.addVertex(LOD_61);
    Node MUL_122 = new Node("MUL_122", "MULT", "MULT");
    graph.addVertex(MUL_122);
    Node LOD_37 = new Node("LOD_37", "LOD", "LOD");
    graph.addVertex(LOD_37);
    Node ADD_116 = new Node("ADD_116", "ADD", "ADD");
    graph.addVertex(ADD_116);
    Node ADD_126 = new Node("ADD_126", "ADD", "ADD");
    graph.addVertex(ADD_126);
    Node ADD_40 = new Node("ADD_40", "ADD", "ADD");
    graph.addVertex(ADD_40);
    Node LOD_69 = new Node("LOD_69", "LOD", "LOD");
    graph.addVertex(LOD_69);
    Node ADD_68 = new Node("ADD_68", "ADD", "ADD");
    graph.addVertex(ADD_68);
    Node ADD_23 = new Node("ADD_23", "ADD", "ADD");
    graph.addVertex(ADD_23);
    Node ADD_112 = new Node("ADD_112", "ADD", "ADD");
    graph.addVertex(ADD_112);
    Node LOD_90 = new Node("LOD_90", "LOD", "LOD");
    graph.addVertex(LOD_90);
    Node LOD_97 = new Node("LOD_97", "LOD", "LOD");
    graph.addVertex(LOD_97);
    ⋮
    graph.addEdge(LOD_37, ADD_40);
    graph.addEdge(LOD_1, ADD_11);
    graph.addEdge(MUL_122, ADD_123);
    graph.addEdge(ADD_108, ADD_116);
    graph.addEdge(ADD_7, LOD_8);
    graph.addEdge(LOD_90, ADD_100);
    graph.addEdge(LOD_53, ADD_56);
    graph.addEdge(ASR_128, STR_130);
    graph.addEdge(LOD_61, ADD_64);
    graph.addEdge(LOD_113, ADD_116);
    graph.addEdge(MUL_121, ADD_123);
    graph.addEdge(LOD_12, ADD_15);
    graph.addEdge(ADD_52, LOD_53);
    graph.addEdge(ADD_19, LOD_20);
    graph.addEdge(ADD_68, LOD_69);
    graph.addEdge(ADD_76, LOD_77);
    graph.addEdge(LOD_45, ADD_48);
    graph.addEdge(ADD_40, ADD_48);
    graph.addEdge(ADD_60, LOD_61);
    graph.addEdge(ADD_116, ADD_118);
    graph.addEdge(ADD_11, ADD_15);
    graph.addEdge(LOD_77, ADD_80);
    graph.addEdge(LOD_105, ADD_108);
    graph.addEdge(LOD_69, ADD_72);
    graph.addEdge(ADD_64, ADD_72);
    ⋮
  } catch (Exception e) { }
 }
}
```

Algoritmo C.16: write_bmp_header

```
1  LOCAL(void)
2  write_bmp_header (j_decompress_ptr cinfo, bmp_dest_ptr dest){
3      char bmpfileheader[14];
4      char bmpinfoheader[40];
5      #define PUT_2B(array,offset,value) \
6      (array[offset] = (char) ((value) & 0xFF), \
7      array[offset+1] = (char) (((value) >> 8) & 0xFF))
8      #define PUT_4B(array,offset,value) \
9      (array[offset] = (char) ((value) & 0xFF), \
10     array[offset+1] = (char) (((value) >> 8) & 0xFF), \
11     array[offset+2] = (char) (((value) >> 16) & 0xFF), \
12     array[offset+3] = (char) (((value) >> 24) & 0xFF))
13     INT32 headersize, bfSize;
14     int bits_per_pixel, cmap_entries;
15     if (cinfo->out_color_space == JCS_RGB){
16         if (cinfo->quantize_colors){
17             /* Colormapped RGB */
18             bits_per_pixel = 8;
19             cmap_entries = 256;
20         } else {
21             bits_per_pixel = 24;
22             cmap_entries = 0;
23         }
24     } else {
25         bits_per_pixel = 8;
26         cmap_entries = 256;
27     headersize = 14 + 40 + cmap_entries * 4;
28     bfSize = headersize + (INT32) dest->row_width * (INT32) cinfo->output_height;
29     MEMZERO(bmpfileheader, SIZEOF(bmpfileheader));
30     MEMZERO(bmpinfoheader, SIZEOF(bmpinfoheader));
31     bmpfileheader[0] = 0x42;
32     bmpfileheader[1] = 0x4D; PUT_4B(bmpfileheader, 2, bfSize);
33     PUT_4B(bmpfileheader, 10, headersize);
34     PUT_2B(bmpinfoheader, 0, 40);
35     PUT_4B(bmpinfoheader, 4, cinfo->output_width);
36     PUT_4B(bmpinfoheader, 8, cinfo->output_height);
37     PUT_2B(bmpinfoheader, 12, 1);
38     PUT_2B(bmpinfoheader, 14, bits_per_pixel);
39     if (cinfo->density_unit == 2) {
40         PUT_4B(bmpinfoheader, 24, (INT32) (cinfo->X_density*100));
41         PUT_4B(bmpinfoheader, 28, (INT32) (cinfo->Y_density*100));
42     } PUT_2B(bmpinfoheader, 32, cmap_entries);
43     if (JFWRITE(dest->pub.output_file, bmpfileheader, 14) != (size_t) 14){
44         ERREXIT(cinfo, JERR_FILE_WRITE);
45     }
46     if (JFWRITE(dest->pub.output_file, bmpinfoheader, 40) != (size_t) 40){
47         ERREXIT(cinfo, JERR_FILE_WRITE);
48     }
49     if (cmap_entries > 0){
50         write_colormap(cinfo, dest, cmap_entries, 4);
51     }
52 }
```

Algoritmo C.17: Dataflow XML do write_bmp

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="write_bmp_header">
    <COMPONENT unit="STR" operation="STR" name="STR_275"/>
    <COMPONENT unit="AND" operation="AND" name="AND_182"/>
    <COMPONENT unit="AND" operation="AND" name="AND_93"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_133"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_88"/>
    <COMPONENT unit="STR" operation="STR" name="STR_188"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_3"/>
    ⋮
    <COMPONENT unit="AND" operation="AND" name="AND_83"/>
    <COMPONENT unit="ASR" operation="ASR" name="ASR_101"/>
    <COMPONENT unit="STR" operation="STR" name="STR_81"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_134"/>
    ⋮
    <SIGNAL name="edge_0">
        <SOURCE name="ADD_4_AUX_2"/>
        <SINK name="ADD_4_AUX_0"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="ADD_202"/>
        <SINK name="LOD_203"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="LOD_176"/>
        <SINK name="LSR_180"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="LOD_233"/>
        <SINK name="LSR_237"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="ADD_133"/>
        <SINK name="LOD_134"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="ADD_18_AUX_3"/>
        <SINK name="ASR_63"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="ADD_4"/>
        <SINK name="ADD_4_AUX_2"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="ADD_80"/>
        <SINK name="STR_81"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="ADD_98"/>
        <SINK name="STR_99"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="ADD_274"/>
        <SINK name="STR_275"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE name="ASR_53"/>
        <SINK name="AND_55"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_91">
        <SOURCE name="ADD_130"/>
        <SINK name="STR_131"/>
    </SIGNAL>
    <SIGNAL name="edge_92">
        <SOURCE name="ADD_88"/>
        <SINK name="STR_89"/>
    </SIGNAL>
</DESIGN>
```

Algoritmo C.18: Dataflow java writer_bmp_reader

```
 1 package files.dataflow;
 2 import architectures.dataflows.graph.Node;
 3 import architectures.dataflows.graph.RGraph;
 4 import architectures.dataflows.javagraphs.DataflowGraph;
 5 public class write_bmp_header extends DataflowGraph {
 6  RGraph graph = new RGraph();
 7  @Override
 8  public RGraph getGraph() {
 9   return graph;
10  }
11  public write_bmp_header() {
12   try {
13    Node STR_275 = new Node("STR_275", "STR",
   "STR");
14    graph.addVertex(STR_275);
15    Node AND_182 = new Node("AND_182", "AND",
   "AND");
16    graph.addVertex(AND_182);
17    Node AND_93 = new Node("AND_93", "AND", "AND");
18    graph.addVertex(AND_93);
19    Node ADD_133 = new Node("ADD_133", "ADD",
   "ADD");
20    graph.addVertex(ADD_133);
21    Node ADD_88 = new Node("ADD_88", "ADD", "ADD");
22    graph.addVertex(ADD_88);
23    Node STR_188 = new Node("STR_188", "STR",
   "STR");
24    graph.addVertex(STR_188);
25    Node MUL_3 = new Node("MUL_3", "MULT", "MULT");
26    graph.addVertex(MUL_3);
27    Node ADD_187 = new Node("ADD_187", "ADD",
   "ADD");
28    graph.addVertex(ADD_187);
29    Node ADD_60 = new Node("ADD_60", "ADD", "ADD");
30    graph.addVertex(ADD_60);
31    Node LOD_146 = new Node("LOD_146", "LOD",
   "LOD");
32    graph.addVertex(LOD_146);
33    Node LSR_150 = new Node("LSR_150", "LSR",
   "LSR");
34    graph.addVertex(LSR_150);
35    Node ADD_98 = new Node("ADD_98", "ADD", "ADD");
36    graph.addVertex(ADD_98);
37    Node STR_230 = new Node("STR_230", "STR",
   "STR");
38    graph.addVertex(STR_230);
39    Node STR_257 = new Node("STR_257", "STR",
   "STR");
40    graph.addVertex(STR_257);
41    Node STR_131 = new Node("STR_131", "STR",
   "STR");
42    graph.addVertex(STR_131);
43    Node ADD_70 = new Node("ADD_70", "ADD", "ADD");
44    graph.addVertex(ADD_70);
45    Node ADD_190 = new Node("ADD_190", "ADD",
   "ADD");
46    graph.addVertex(ADD_190);
47    Node ADD_157 = new Node("ADD_157", "ADD",
   "ADD");
48    graph.addVertex(ADD_157);
      ⋮
49    graph.addVertex(LOD_9);
50    graph.addEdge(ASR_53, AND_55);
51    graph.addEdge(LOD_191, AND_194);
52    graph.addEdge(AND_83, STR_89);
53    graph.addEdge(MUL_3, ADD_4);
54    graph.addEdge(ADD_157, STR_158);
55    graph.addEdge(AND_93, STR_99);
56    graph.addEdge(LOD_134, AND_137);
57    graph.addEdge(ADD_232, LOD_233);
58    graph.addEdge(LOD_278, BNE_282);
59    graph.addEdge(ADD_160, LOD_161);
60    graph.addEdge(AND_239, STR_245);
61    graph.addEdge(LOD_161, LSR_165);
62    graph.addEdge(LOD_9, MUL_17);
63    graph.addEdge(LOD_218, LSR_222);
64    graph.addEdge(ADD_199, STR_200);
65    graph.addEdge(LSR_180, AND_182);
66    graph.addEdge(AND_45, STR_51);
67    graph.addEdge(AND_113, STR_119);
68    graph.addEdge(AND_65, STR_71);
69    graph.addEdge(LOD_14, MUL_17);
70    graph.addEdge(ADD_277, LOD_278);
71    graph.addEdge(AND_103, STR_109);
72    graph.addEdge(ADD_250, STR_251);
73    graph.addEdge(ADD_217, LOD_218);
74    graph.addEdge(ADD_70, STR_71);
75    graph.addEdge(ADD_256, STR_257);
76    graph.addEdge(ASR_267, AND_269);
77    graph.addEdge(ADD_145, LOD_146);
78    graph.addEdge(ADD_42, STR_43);
79    graph.addEdge(ADD_244, STR_245);
80    graph.addEdge(ADD_108, STR_109);
81    graph.addEdge(AND_194, STR_200);
82    graph.addEdge(ADD_264, STR_265);
83    graph.addEdge(ASR_63, AND_65);
84    graph.addEdge(LSR_165, AND_167);
85    graph.addEdge(AND_152, STR_158);
86    graph.addEdge(LOD_203, LSR_207);
87    graph.addEdge(MUL_17, ADD_18);
88    graph.addEdge(LSR_150, AND_152);
89    graph.addEdge(AND_259, STR_265);
90    graph.addEdge(AND_269, STR_275);
91    graph.addEdge(ADD_60, STR_61);
92    graph.addEdge(ADD_190, LOD_191);
93    graph.addEdge(ADD_13, LOD_14);
94    graph.addEdge(LSR_222, AND_224);
95    graph.addEdge(AND_182, STR_188);
96    graph.addEdge(AND_55, STR_61);
97    graph.addEdge(ASR_91, AND_93);
98    graph.addEdge(LSR_237, AND_239);
99    graph.addEdge(AND_209, STR_215);
100   graph.addEdge(AND_167, STR_173);
101   graph.addEdge(ADD_124, STR_125);
102   graph.addEdge(AND_75, STR_81);
103   graph.addEdge(ADD_118, STR_119);
104   graph.addEdge(AND_224, STR_230);
105   graph.addEdge(ADD_175, LOD_176);
106   graph.addEdge(ADD_50, STR_51);
107   graph.addEdge(ADD_229, STR_230);
108   graph.addEdge(ASR_101, AND_103);
      ⋮
109
110  } catch (Exception e) { }
111 }
112 }
```

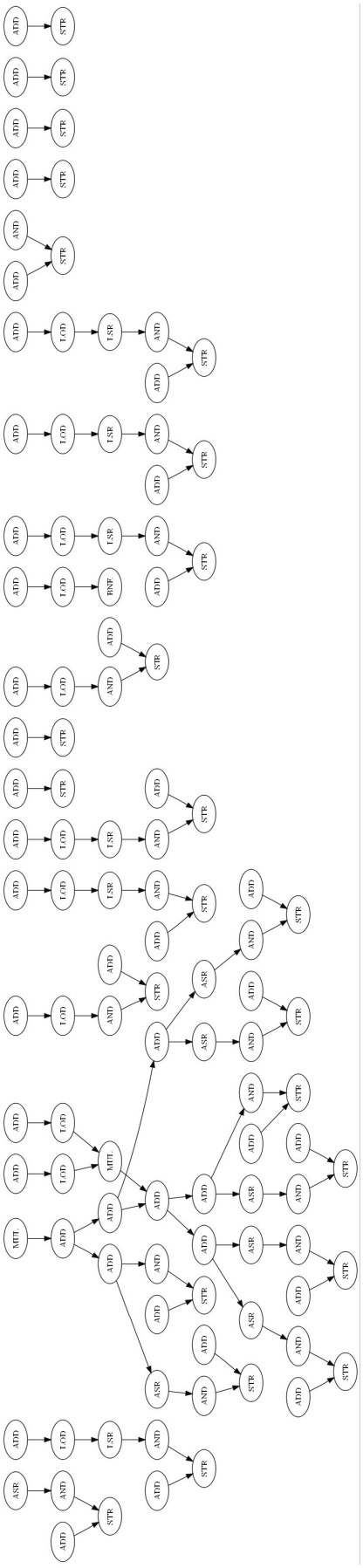


Figura C.6. Gráfico do dataflow do write_bmp_header

Algoritmo C.19: Matmul

```
 1 static void matmul( GLfloat *product, const GLfloat *a, const GLfloat *b ){
 2     /* This matmul was contributed by Thomas Malik */
 3     GLint i;
 4     #define A(row,col) a[(col<<2)+row]
 5     #define B(row,col) b[(col<<2)+row]
 6     #define P(row,col) product[(col<<2)+row]
 7     /* i-te Zeile */
 8     for (i = 0; i < 4; i++) {
 9         GLfloat ai0=A(i,0), ai1=A(i,1), ai2=A(i,2), ai3=A(i,3);
10         P(i,0) = ai0 * B(0,0) + ai1 * B(1,0) + ai2 * B(2,0) + ai3 * B(3,0);
11         P(i,1) = ai0 * B(0,1) + ai1 * B(1,1) + ai2 * B(2,1) + ai3 * B(3,1);
12         P(i,2) = ai0 * B(0,2) + ai1 * B(1,2) + ai2 * B(2,2) + ai3 * B(3,2);
13         P(i,3) = ai0 * B(0,3) + ai1 * B(1,3) + ai2 * B(2,3) + ai3 * B(3,3);
14     }
15     #undef A
16     #undef B
17     #undef P
18 }
```

Algoritmo C.20: Dataflow XML do matmul

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DESIGN trace="true" name="matmul">
    <COMPONENT unit="MULT" operation="MULT" name="MUL_102"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_43"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_157"/>
    <COMPONENT unit="ADD" operation="ADD" name="ADD_52"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_69"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_33_AUX_3"/>
    <COMPONENT unit="STR" operation="STR" name="STR_203"/>
    ⋮
    <COMPONENT unit="ADD" operation="ADD" name="ADD_72"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_33_AUX_2"/>
    <COMPONENT unit="LOD" operation="LOD" name="LOD_24"/>
    <COMPONENT unit="MULT" operation="MULT" name="MUL_195"/>
    <SIGNAL name="edge_0">
        <SOURCE name="ADD_183"/>
        <SINK name="LOD_184"/>
    </SIGNAL>
    <SIGNAL name="edge_1">
        <SOURCE name="MUL_138"/>
        <SINK name="ADD_141"/>
    </SIGNAL>
    <SIGNAL name="edge_2">
        <SOURCE name="MUL_180"/>
        <SINK name="ADD_183"/>
    </SIGNAL>
    <SIGNAL name="edge_3">
        <SOURCE name="MUL_171"/>
        <SINK name="ADD_174"/>
    </SIGNAL>
    <SIGNAL name="edge_4">
        <SOURCE name="MUL_73"/>
        <SINK name="ADD_76"/>
    </SIGNAL>
    <SIGNAL name="edge_5">
        <SOURCE name="ADD_108"/>
        <SINK name="LOD_109"/>
    </SIGNAL>
    <SIGNAL name="edge_6">
        <SOURCE name="MUL_129"/>
        <SINK name="ADD_132"/>
    </SIGNAL>
    <SIGNAL name="edge_7">
        <SOURCE name="MUL_93"/>
        <SINK name="ADD_94"/>
    </SIGNAL>
    <SIGNAL name="edge_8">
        <SOURCE name="ADD_112"/>
        <SINK name="STR_119"/>
    </SIGNAL>
    <SIGNAL name="edge_9">
        <SOURCE name="LOD_24_AUX_6"/>
        <SINK name="MUL_60"/>
    </SIGNAL>
    <SIGNAL name="edge_10">
        <SOURCE name="MUL_153"/>
        <SINK name="ADD_154"/>
    </SIGNAL>
    ⋮
    <SIGNAL name="edge_122">
        <SOURCE name="MUL_163"/>
        <SINK name="ADD_166"/>
    </SIGNAL>
    <SIGNAL name="edge_123">
        <SOURCE name="LOD_15_AUX_0"/>
        <SINK name="MUL_177"/>
    </SIGNAL>
</DESIGN>
```

Algoritmo C.21: Dataflow java matmul

```
package files.dataflow;
import architectures.dataflows.graph.Node;
import architectures.dataflows.graph.RGraph;
import architectures.dataflows.javagraphs.DataflowGraph;
public class matmul extends DataflowGraph {
 RGraph graph = new RGraph();
 @Override
  public RGraph getGraph() {
 return graph;
 }
 public matmul() {
  try {
    Node MUL_102 = new Node("MUL_102", "MULT", "MULT");
    graph.addVertex(MUL_102);
    Node MUL_43 = new Node("MUL_43", "MULT", "MULT");
    graph.addVertex(MUL_43);
    Node MUL_157 = new Node("MUL_157", "MULT", "MULT");
    graph.addVertex(MUL_157);
    Node ADD_52 = new Node("ADD_52", "ADD", "ADD");
    graph.addVertex(ADD_52);
    Node MUL_69 = new Node("MUL_69", "MULT", "MULT");
    graph.addVertex(MUL_69);
    Node STR_203 = new Node("STR_203", "STR", "STR");
    graph.addVertex(STR_203);
    Node ADD_10 = new Node("ADD_10", "ADD", "ADD");
    graph.addVertex(ADD_10);
    Node ADD_156 = new Node("ADD_156", "ADD", "ADD");
    graph.addVertex(ADD_156);
    Node ADD_82 = new Node("ADD_82", "ADD", "ADD");
    graph.addVertex(ADD_82);
    Node ADD_66 = new Node("ADD_66", "ADD", "ADD");
    graph.addVertex(ADD_66);
    Node MUL_180 = new Node("MUL_180", "MULT", "MULT");
    graph.addVertex(MUL_180);
    Node ADD_198 = new Node("ADD_198", "ADD", "ADD");
    graph.addVertex(ADD_198);
    Node LOD_125 = new Node("LOD_125", "LOD", "LOD");
    graph.addVertex(LOD_125);
    Node LOD_151 = new Node("LOD_151", "LOD", "LOD");
    graph.addVertex(LOD_151);
    Node MUL_147 = new Node("MUL_147", "MULT", "MULT");
    graph.addVertex(MUL_147);
    Node ADD_48 = new Node("ADD_48", "ADD", "ADD");
    graph.addVertex(ADD_48);
    Node MUL_87 = new Node("MUL_87", "MULT", "MULT");
    graph.addVertex(MUL_87);
    Node MUL_2 = new Node("MUL_2", "MULT", "MULT");
    graph.addVertex(MUL_2);
    Node ADD_160 = new Node("ADD_160", "ADD", "ADD");
    graph.addVertex(ADD_160);
    Node ADD_99 = new Node("ADD_99", "ADD", "ADD");
    graph.addVertex(ADD_99);
    Node ADD_141 = new Node("ADD_141", "ADD", "ADD");
    graph.addVertex(ADD_141);
    Node MUL_11 = new Node("MUL_11", "MULT", "MULT");
    graph.addVertex(MUL_11);
    Node ADD_32 = new Node("ADD_32", "ADD", "ADD");
    graph.addVertex(ADD_32);
    Node LOD_175 = new Node("LOD_175", "LOD", "LOD");
    graph.addVertex(LOD_175);
    Node ADD_112 = new Node("ADD_112", "ADD", "ADD");
    graph.addVertex(ADD_112);
    .
    .
    .
    graph.addEdge(ADD_183, LOD_184);
    graph.addEdge(MUL_138, ADD_141);
    graph.addEdge(MUL_180, ADD_183);
    graph.addEdge(MUL_171, ADD_174);
    graph.addEdge(MUL_73, ADD_76);
    graph.addEdge(ADD_108, LOD_109);
    graph.addEdge(MUL_129, ADD_132);
    graph.addEdge(MUL_93, ADD_94);
    graph.addEdge(ADD_112, STR_119);
    graph.addEdge(MUL_153, ADD_154);
    graph.addEdge(ADD_90, LOD_91);
    graph.addEdge(MUL_45, ADD_48);
    graph.addEdge(MUL_69, ADD_70);
    graph.addEdge(ADD_99, LOD_100);
    graph.addEdge(ADD_1, MUL_2);
    graph.addEdge(ADD_124, LOD_125);
    graph.addEdge(ADD_202, STR_203);
    graph.addEdge(LOD_125, MUL_127);
    graph.addEdge(MUL_96, ADD_99);
    graph.addEdge(ADD_160, STR_161);
    graph.addEdge(MUL_29, ADD_32);
    graph.addEdge(ADD_187, ADD_196);
    graph.addEdge(ADD_10, MUL_11);
    graph.addEdge(MUL_147, ADD_150);
    graph.addEdge(LOD_83, MUL_85);
    graph.addEdge(MUL_2, ADD_5);
    graph.addEdge(MUL_121, ADD_124);
    graph.addEdge(MUL_186, ADD_187);
    graph.addEdge(ADD_192, LOD_193);
    graph.addEdge(LOD_100, MUL_102);
    graph.addEdge(MUL_102, ADD_103);
    graph.addEdge(ADD_52, ADD_61);
    graph.addEdge(MUL_189, ADD_192);
    graph.addEdge(MUL_169, ADD_178);
    graph.addEdge(MUL_115, ADD_118);
    graph.addEdge(ADD_82, LOD_83);
    graph.addEdge(LOD_184, MUL_186);
    graph.addEdge(MUL_63, ADD_66);
    graph.addEdge(MUL_177, ADD_178);
    graph.addEdge(ADD_145, ADD_154);
    graph.addEdge(MUL_37, ADD_40);
    .
    .
    .
  } catch (Exception e) { }
 }
}
```

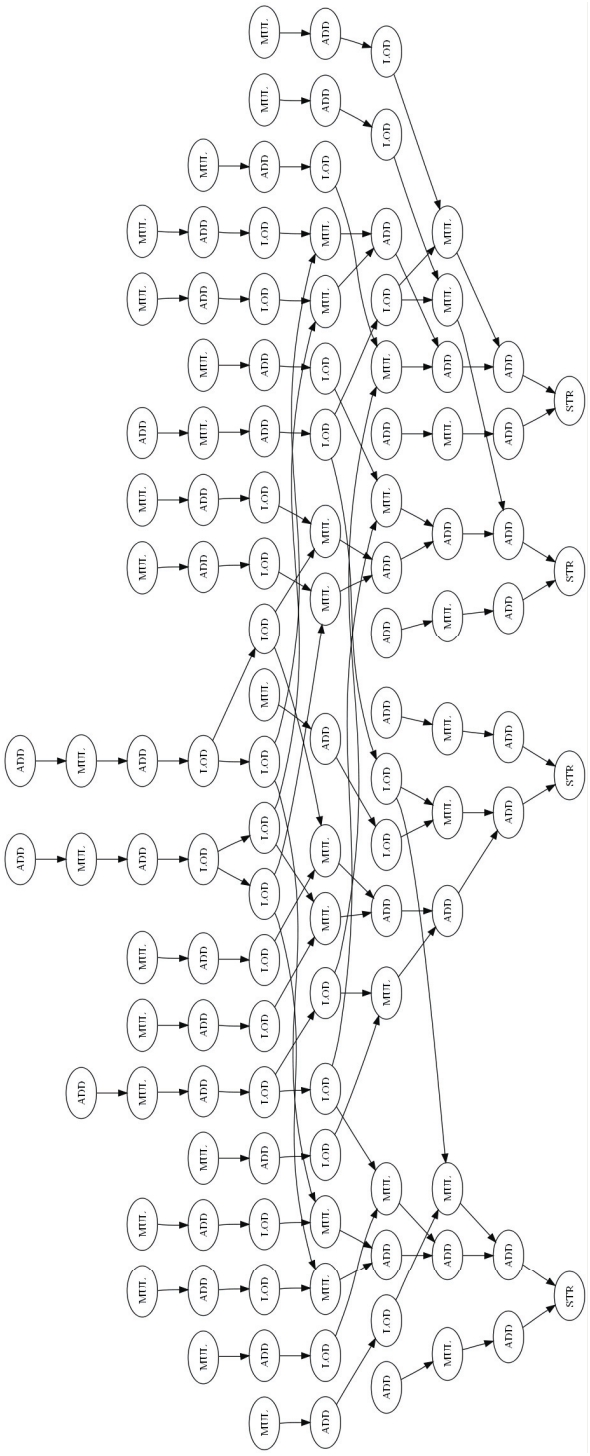


Figura C.7. Gráfico do dataflow do matmul

Anexo D

XML-Schemas

Nesse anexo apresentamos os XML-Schemas desenvolvidos durante esse trabalho. Com base neles foi possível a criação dinâmica de dataflows, arquiteturas e grupos de benchmarks.

O Algoritmo D.1 permite a construção de um dataflow em XML, o processo de conversão para a linguagem java é feito pelo Algoritmo D.2, com esses dois Algoritmos é possível a construção dinâmica de dataflows gerando automaticamente o código java correspondente e carregando-o para a memória, esse é o processo utilizado para a geração de dataflows a partir do programa TGFF, esse programa gera um dataflow em um formato texto especifico, esse formato é traduzido para XML utilizando o Algoritmo D.1 como esqueleto, sendo aplicado à saída o Algoritmo D.2 gerando assim um dataflow que pode ser carregado para a memória e mapeado em uma arquitetura.

A partir do Algoritmo D.3 é possível a criação de arquiteturas de forma dinâmica, a lógica de criação desse Algoritmo foi descrita na seção 3.2, onde é apresentada parte desse Algoritmo, no entanto, a visualização completa do Algoritmo permite verificar a possibilidade de utilização de padrões de conexão diferentes de 0_n_hop, podendo ser utilizado padrões de inversão de bits, porém para esse trabalho os resultados obtidos por esses padrões apresentaram pouca variação em relação ao padrão 0_n_hop, formando no final arquiteturas irregulares, por isso, esses padrões foram desconsiderados na apresentação de resultados, mas ainda fazem parte da especificação do Algoritmo para possíveis usos futuros.

O último XML-Schema apresentado no Algoritmo D.4 tem por finalidade facilitar a criação de grupos de dataflows, para isso ele possui informações básicas como localização e nome do dataflow, além do número de arcos e vértices.

Algoritmo D.1: XML-Schema utilizado na criação de dataflows

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="COMPONENT">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="PORT" maxOccurs="unbounded" />
   </xsd:sequence>
   <xsd:attribute name="unit" type="xsd:string" use="required" />
   <xsd:attribute name="operation" type="xsd:string" use="required" />
   <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="SIGNAL">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="SOURCE" maxOccurs="1" />
    <xsd:element ref="SINK" maxOccurs="1" />
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="PORT">
  <xsd:complexType>
   <xsd:attribute name="name" type="xsd:string" use="required" />
   <xsd:attribute name="value" type="xsd:int" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="SOURCE">
  <xsd:complexType>
   <xsd:attribute name="name" type="xsd:string" use="required" />
   <xsd:attribute name="port" type="xsd:string" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="SINK">
  <xsd:complexType>
   <xsd:attribute name="name" type="xsd:string" use="required" />
   <xsd:attribute name="port" type="xsd:string" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="DESIGN">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="COMPONENT" maxOccurs="unbounded" />
    <xsd:element ref="SIGNAL" maxOccurs="unbounded" />
   </xsd:sequence>
   <xsd:attribute name="trace" use="required">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="true" />
      <xsd:enumeration value="false" />
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
   <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Algoritmo D.2: XML-Stylesheet utilizado na conversão de XML para Java

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 3  <!- XML to Java XSLT Transformer
 4  August, 2007
 5  University of Vicosa, Brazil
 6  ->
 7  <!- TAB char unicode = 9->
 8  <xsl:output method="text" omit-xml-declaration="no" indent="yes"/>
 9  <!- global variable indicating the value of the trace option ->
10  <!-- <xsl:variable name="trace" select="/DESIGN/@trace"/> ->
11  <xsl:variable name="topology" select="/DESIGN/@name"/>
12  <xsl:variable name="nome" select="/DESIGN/COMPONENT/@name"></xsl:variable>
13  <!- end globals ->
14  <xsl:template match="/">
15   <xsl:text>package files.dataflow;&#xA;&#xA;</xsl:text>
16   <xsl:text>import architectures.dataflows.graph.Node;&#xA;</xsl:text>
17   <xsl:text>import architectures.dataflows.graph.RGraph;&#xA;</xsl:text>
18   <xsl:text>import architectures.dataflows.javagraphs.DataflowGraph;&#xA;&#xA;</xsl:text>
19   <xsl:text>public class </xsl:text>
20   <xsl:value-of select="$topology"/>
21   <xsl:text> extends DataflowGraph
     {&#xA;&#xA;</xsl:text>
22   <xsl:text>&#9;RGraph graph = new
     RGraph();&#xA;&#xA;</xsl:text>
23   <xsl:text>&#9;@Override&#xA;</xsl:text>
24   <xsl:text>&#9;public RGraph getGraph()
     {&#xA;</xsl:text>
25   <xsl:text>&#9;&#9;return graph;&#xA;</xsl:text>
26   <xsl:text>&#9;}&#xA;&#xA;</xsl:text>
27   <!- ### BEGIN generate method for creating
     design->
28   <xsl:text>&#9;public </xsl:text>
29   <xsl:value-of select="$topology"/>
30   <xsl:text>() {&#xA;</xsl:text>
31   <xsl:text>&#9;&#9;try {&#xA;</xsl:text>
32   <xsl:apply-templates select="//COMPONENT"/>
33   <xsl:apply-templates select="//SIGNAL"/>
34   <xsl:text>&#9;&#9;} catch (Exception e) { }
     &#xA;</xsl:text>
35   <xsl:text>&#9;}&#xA;</xsl:text>
36   <xsl:text>}&#xA;</xsl:text>
37   <!- ### END generate method for creating
     design->
38  </xsl:template>
39  <!- #### BEGIN generation of components ->
40  <xsl:template match="COMPONENT">
41   <xsl:variable name="nome"><xsl:value-of
     select="@name"/></xsl:variable>
42   <xsl:text>&#9;&#9;&#9;Node </xsl:text>
43   <xsl:value-of select="$nome"/>
44   <xsl:text> = new Node("</xsl:text>
45   <xsl:value-of select="$nome"/>
46   <xsl:text>", "</xsl:text>
47   <xsl:value-of select="@unit"/>
48   <xsl:text>", "</xsl:text>
49   <xsl:value-of select="@operation"/>
50   <xsl:text>");&#xA;</xsl:text>
51   <xsl:text>&#9;&#9;&#9;graph.addVertex(</xsl:text>
52   <xsl:value-of select="$nome"/>
53   <xsl:text>);</xsl:text>
54   <xsl:apply-templates select="./PORT"/>
55   <xsl:text>&#xA;&#xA;</xsl:text>
56  </xsl:template>
57  <xsl:template match="PORT">
58   <xsl:text>&#xA;&#9;&#9;&#9;</xsl:text>
59   <xsl:value-of select="$nome"/>
60   <xsl:text>.setPortValue("</xsl:text>
61   <xsl:value-of select="@name"/>
62   <xsl:text>", "</xsl:text>
63   <xsl:value-of select="@value"/>
64   <xsl:text>");</xsl:text>
65  </xsl:template>
66  <!- #### END generation of components ->
67  <!- #### BEGIN generation of connections ->
68  <xsl:template match="SIGNAL">
69   <xsl:text>&#9;&#9;&#9;graph.addEdge(</xsl:text>
70   <xsl:apply-templates select="./SOURCE"/>
71   <xsl:text>, </xsl:text>
72   <xsl:apply-templates select="./SINK"/>
73   <xsl:text>);&#xA;</xsl:text>
74  </xsl:template>
75  <xsl:template match="SOURCE">
76   <xsl:value-of select="@name"/>
77  </xsl:template>
78  <xsl:template match="SINK">
79   <xsl:value-of select="@name"/>
80  </xsl:template>
81  <!- #### END generation of connections ->
82 </xsl:stylesheet>
```

Algoritmo D.3: XML-Schema para a geração de arquiteturas

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="bit" type="xsd:int"/>
 <xsd:element name="lenghtString" type="xsd:int"/>
 <xsd:element name="hop" type="xsd:int"/>
 <xsd:element name="element">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="link" maxOccurs="unbounded" />
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:int" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="line">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="hop"/>
    <xsd:element ref="lenghtString" minOccurs="0"/>
    <xsd:element ref="bit" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="family" use="required">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Base" />
      <xsd:enumeration value="MultiStage" />
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
   <xsd:attribute name="type" use="required">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="HOP" />
      <xsd:enumeration value="InversePerfectShuffler" />
      <xsd:enumeration value="PerfectShuffler" />
      <xsd:enumeration value="BitReverse" />
      <xsd:enumeration value="Butterfly" />
      <xsd:enumeration value="BaseLine" />
      <xsd:enumeration value="Cube" />
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="colun">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="hop"/>
    <xsd:element ref="lenghtString" minOccurs="0"/>
    <xsd:element ref="bit" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="family" use="required">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Base" />
      <xsd:enumeration value="MultiStage" />
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
   <xsd:attribute name="type" use="required">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="HOP" />
      <xsd:enumeration value="InversePerfectShuffler" />
      <xsd:enumeration value="PerfectShuffler" />
      <xsd:enumeration value="BitReverse" />
      <xsd:enumeration value="Butterfly" />
      <xsd:enumeration value="BaseLine" />
      <xsd:enumeration value="Cube" />
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="link">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="line"/>
    <xsd:element ref="colun"/>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:int" use="required" />
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="Topologies">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="element" maxOccurs="unbounded" />
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Algoritmo D.4: XML-Schema para a criação de grupos de dataflows

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 3  <xsd:element name="nodes" type="xsd:int"/>
 4  <xsd:element name="wires" type="xsd:int"/>
 5  <xsd:element name="name" type="xsd:string"/>
 6  <xsd:element name="path" type="xsd:string"/>
 7  <xsd:element name="bench">
 8   <xsd:complexType>
 9    <xsd:sequence>
10     <xsd:element ref="name"/>
11     <xsd:element ref="path"/>
12     <xsd:element ref="wires"/>
13     <xsd:element ref="nodes"/>
14    </xsd:sequence>
15    <xsd:attribute name="id" type="xsd:int" use="required" />
16   </xsd:complexType>
17  </xsd:element>
18  <xsd:element name="Benchmarks">
19   <xsd:complexType>
20    <xsd:sequence>
21     <xsd:element ref="bench" maxOccurs="unbounded" />
22    </xsd:sequence>
23   </xsd:complexType>
24  </xsd:element>
25 </xsd:schema>
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo