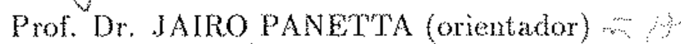
Uma Interface de Comunicação para um Ambiente de Reestruturação de Programas

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. BRUNO MÜLLER JUNIOR e aprovada pela Comissão Julgadora.

Campinas, 21 de Dezembro de 1991



Prof. Dr. JAIRO PANETTA (orientador)

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de MESTRE em Ciência da Computação

M912i

15674/BC

UNICAMP
BIBLIOTECA CENTRAL

AGRADECIMENTOS

Meus sinceros agradecimentos a meu orientador pelo incentivo e pela ajuda para a confecção deste trabalho, e a meu co-orientador pelas sugestões e colaboração no desenvolvimento do trabalho.

Aos amigos do Instituto de Estudos Avançados e do Projeto Computação Científica, pelo apoio que sempre me deram.

Aos meus pais, pelo seu amor e orientação na minha educação e formação; a Lurdinha, pelo seu amor, paciência e incentivo a esta realização.

Agradecimentos especiais ao Prof. Dr. Clésio Luis Tozzi e ao Prof. Dr. Hans Kurt Edmund Liesenberg pelas valiosas sugestões para a confecção da versão final do texto.

Conteúdo

1 Introdução 7

2 Conceitos 9

2.1 Ambientes de Reestruturação de Programas 9

2.1.1 O Projeto Computação Científica (PCC) 12

2.2 Interfaces 15

2.2.1 Janelas 17

2.2.2 Interface com o Programa de Aplicação 21

2.2.3 X Window 22

2.3 A Interface do Usuário do PCC 23

3 Os Widgets da Interface 25

3.1 Preliminares 25

3.1.1 Hierarquia entre Widgets 25

3.1.2 Funções de retorno 26

3.2 Widgets Tradicionais 28

3.2.1 Janela 28

3.2.2 Área de Trabalho 28

3.2.3 Scrollbar Vertical 30

3.2.4 Scrollbar Horizontal 30

3.2.5 Menus 30

3.3 Widgets Específicos 33

3.3.1 Eixos Cartesianos 34

3.3.2 Gráficos de Acompanhamento 35

3.3.3 Hierarquias 36

4 O Algoritmo para Desenhar Hierarquias 42

4.1 Preliminares 43

4.1.1 Características de um Mapa Legível 43
4.1.2 Resumo dos Passos 44
4.2 Definições 45
4.2.1 Hierarquia 45
4.2.2 Matriz de Realização 46
4.2.3 Número de Cruzamentos 47
4.2.4 Conectividade 49
4.2.5 Baricentro 49
4.3 Detalhamento dos Passos 50
4.3.1 Hierarquização de Vértices (Passo 1) 51
4.3.2 O Algoritmo de Redução de Cruzamentos (Passo 2) 53
4.3.3 Posicionamento Horizontal (Passo 3) 63
4.3.4 Desenho (Passo 4) 66

5 Conclusões e Trabalhos Futuros **69**
5.1 Conclusões 69
5.2 Trabalhos Futuros 70

A As Funções da Interface **72**

B As Bibliotecas da Interface **95**
B.1 IUtradic.h 95
B.2 IUhier.h 97
B.3 IUevent.h 99

Lista de Figuras

2.1 Um programa Fortran 14
2.2 O grafo do programa Fortran apresentado, a nível de comandos 15
2.3 O Ambiente de Re-estruturação do PCC 16
2.4 Modelo de Janela com seus Componentes 20

3.1 Os Widgets Tradicionais da Interface 26
3.2 Widgets Hierarquia 26
3.3 Aparência dos Dois Tipos de Eixos Cartesianos 34
3.4 Aparência de um Gráfico de Acompanhamento 35

4.1 Elementos de uma hierarquia 43
4.2 Resultado da Aplicação do Algoritmo em uma Hierarquia 45
4.3 Exemplo de Hierarquia. 47
4.4 Cruzamento Entre Duas Arestas 47
4.5 Mapas Possíveis a Partir do Mesmo Grafo 52
4.6 Hierarquia de Duas Camadas Antes da Redução. 56
4.7 Hierarquia de Duas Camadas Resultante. 58
4.8 Hierarquia Antes de Efetuar a Reduçäo 61
4.9 Hierarquia Resultante 63
4.10 Hierarquia antes de efetuar o Alinhamento 65
4.11 Hierarquia Alinhada 65
4.12 Exemplificação dos Resultados dos Procedimentos Baixo e Cima em uma Hierarquia 67
4.13 Hierarquia com uso misto dos procedimetos baixo e cima 68
4.14 Exemplo de Desenho de Vértices e Local de Incidência de Arestas 68

Resumo

Este trabalho apresenta uma interface de comunicação entre um ambiente de reestruturação de programas e o usuário.
A interface contém um conjunto de componentes gráficos (widgets) que permitem ao usuário interagir amigavelmente com o seu programa. Estes componentes gráficos estão divididos em dois grupos, widgets tradicionais e widgets específicos. Os widgets tradiconais são os comuns a ambientes de programação, enquanto que os widgets específicos são específicos ao ambiente em questão.
Dentre os widgets específicos destaca-se o widget que desenha hierarquias. Grande parte deste trabalho concentra-se no algoritmo que efetua o desenho, apresentando melhoramentos ao algoritmo conhecido na literatura.

Capítulo 1

Introdução

O processamento paralelo é um dos responsáveis pelo aumento na velocidade dos computadores a que estamos assistindo. Porém, as velocidades efetivamente obtidas em aplicações reais estão muito aquém das velocidades máximas destas máquinas. Isto ocorre em grande parte devido à dificuldade de traduzir programas escritos em linguagens seqüenciais para formas adequadas à execução paralela.

Compiladores paralelizadores efetuam esta tradução reestruturando programas. Para tanto selecionam, a partir de um conjunto de regras de reestruturação, aquelas que geram o programa paralelo mais adequado à máquina alvo. Porém, em muitas situações os compiladores adotam regras que impedem um grau maior de paralelismo.

Para obter um desempenho melhor, o usuário é obrigado a reestruturar manualmente o programa, forçando o compilador a utilizar outras regras, o que talvez aumente a velocidade do programa. Esta porém é uma tarefa complexa e demorada.

Felizmente, o processo de reestruturação manual pode ser auxiliado por computador por meio de ambientes de programação que suportem a reestruturação interativa.

Neste trabalho, propõe-se uma interface de comunicação entre um ambiente de reestruturação deste tipo e o usuário. O ambiente em questão, além de auxiliar na reestruturação manual, coordena as atividades de edição, depuração e ajuste de desempenho de programas escritos em linguagens seqüenciais.

A interface contém um conjunto de operações que permite ao usuário interagir amigavelmente com o seu programa. Algumas delas utilizam os conceitos tradicionais de ambientes de programação como janelas, menus, textos, etc., enquanto outras são destinadas especificamente ao ambiente em questão, como gráficos para auxiliar a depuração e a análise de desempenho e um editor/gerenciador de grafos.

Este trabalho está estruturado da seguinte maneira: O capítulo 2 introduz ambientes de reestruturação e interfaces de comunicação. O capítulo 3 apresenta as necessidades

do ambiente em questão e as soluções propostas. O capítulo 4 explica o funcionamento do algoritmo que desenha hierarquias (grafos dirigidos com várias camadas de vértices), uma facilidade gráfica da interface. Finalmente, o capítulo 5 apresenta as conclusões, críticas e trabalhos futuros.

Capítulo 2

Conceitos

O objetivo deste capítulo é introduzir a nomenclatura utilizada ao longo do texto. O capítulo está dividido em três partes. A primeira introduz ambientes de reestruturação de programas, com ênfase ao ambiente alvo deste trabalho. A segunda parte apresenta os conceitos envolvidos com interfaces de comunicação e com o programa gráfico que foi utilizado, o OSF/Motif. A terceira parte detalha o objetivo deste trabalho.

2.1 Ambientes de Reestruturação de Programas

Desde o aparecimento dos computadores, o software destinado à ciência à e engenharia tem sido grande consumidor de recursos computacionais. Este tipo de software, chamado software científico, é utilizado em dezenas de indústrias como a petrolífera, a automobilística, a aeronáutica, etc..

A maioria desses programas foram e continuam sendo desenvolvidos em linguagens seqüenciais como FORTRAN, que ao serem transportados para computadores paralelos, tornaram-se obstáculos para atingir as velocidades potenciais destas máquinas.

Como alternativa, algumas máquinas oferecem extensões paralelas às linguagens seqüenciais. Porém, as extensões variam de máquina a máquina, fazendo com que o software não seja portável.

Este trabalho concentra-se nas linguagens seqüenciais sem extensões. Nelas, o paralelismo é extraído automaticamente pelo compilador. Ele procura trechos do programa original que podem ser executados em paralelo (em sua maioria laços) transformando o programa seqüencial em um programa paralelo semanticamente equivalente.

Infelizmente, muitos dos trechos que poderiam ser paralelizados não o são devido à estrutura do programa seqüencial, onde a execução de alguns comandos depende de

outros comandos. A isso se dá o nome de *dependência*. Existe *dependência* entre dois comandos quando a execução de um causa efeitos na execução do outro. Por exemplo, na computação abaixo,

```
      DO I=1,100
c1       B(I)=C(I)+1
c2       A(I)=A(I-1)+B(I)
      CONTINUE
```

há uma dependência entre o comando c_1 e c_2. O comando c_1 produz um resultado (`B(I)`) consumido em c_2. Para gerar o resultado correto, é necessário que c_1 seja executado antes de c_2.

Existe outra dependência entre duas instâncias de c_2, correspondentes à iterações consecutivas do laço. O resultado consumido em uma iteração (`A(I-1)`) é produzido na iteração anterior.

Além destas dependências, chamadas de *Dependências de Dados*, existem também dependências introduzidas pelo `DO`, chamadas *Dependências de Controle*.

Para aumentar tanto a quantidade de trechos de programa que podem ser executados em paralelo quanto o grau de paralelismo desses trechos, aumentando conseqüentemente a velocidade de execução, é necessário reduzir e se possível eliminar as dependências entre comandos. Isso é possível reestruturando o programa original através de um conjunto de técnicas que eliminam as dependências sem alterar a semântica do programa.

Os compiladores de máquinas paralelas aplicam as técnicas de reestruturação nos programas, porém não conseguem eliminar grande parte das dependências existentes [CDL 88]. Isto ocorre principalmente porque o compilador adota técnicas conservadoras por falta de informações sobre o programa.

Para muitos usuários, o código gerado pelos compiladores possui um desempenho satisfatório, porém aqueles que necessitam de um melhor desempenho são obrigados a reestruturar manualmente os seus programas através do que se chama *depuração de desempenho* [LSVC 89]. A depuração de desempenho compreende a iteração entre as tarefas de medir, modificar e comparar sucessivos protótipos computacionais. Essa tarefa é tediosa e complexa, exigindo conhecimentos de programação paralela, transformação de programas seqüenciais em programas paralelos, hardware da máquina alvo e, é claro, conhecimento da aplicação.

Para auxiliar o usuário nessa tarefa foram desenvolvidas ferramentas semi-automáticas que dão suporte à reestruturação de programas. Essas ferramentas auxiliam o usuário a detectar dependências, escolher as técnicas de reestruturação e a ordem em que elas

devem ser aplicadas e a avaliar o desempenho obtido com as técnicas escolhidas. Essas ferramentas, por sua vez, podem ser agrupadas em ambientes de reestruturação semi-automática de programas.

Os ambientes de reestruturação semi-automática de programas estão em fase de pesquisa acadêmica. Descrevem-se três deles, o Parascope, o Parafrase e o Sigma. Em seguida, descreve-se o Projeto Computação Científica (PCC) e seu ambiente de reestruturação de programas.

Os quatro ambientes de reestruturação apresentam alguns pontos em comum, dos quais destacamos dois. O primeiro é a linguagem a reestruturar. Todos destinam-se principalmente à reestruturação de programas FORTRAN, que é a linguagem mais popular no meio científico.

O segundo é o uso de uma forma intermediária de armazenamento comum a todas as ferramentas de cada ambiente. Na maioria dos casos, o programa original é traduzido para um grafo que contém todas as informações sintáticas e semânticas necessárias para reproduzir o programa.

A seguir, apresentamos os ambientes acima mencionados.

1. **O Parascope:** O Parascope [CCHK 87] é uma evolução do trabalho anterior da Rice University, o R^n, um ambiente para desenvolvimento e manutenção de programas numéricos. Além do R^n, Rice também desenvolveu duas ferramentas automáticas, o PTOOL para detecção de paralelismo e o PFC (Parallel Fortran Converter) para reestruturação.

   O Parascope é basicamente uma extensão do R^n para suportar paralelismo e reestruturação manual de programas. Atualmente, ele consiste somente de um editor e de um reestruturador manual de programas, mas existem planos para a sua expansão, incluindo novas versões do compilador e do depurador existentes no R^n.

2. **O Parafrase:** O Parafrase-2 [PGHL 89], ou simplesmente Parafrase, está em desenvolvimento no CSRD (*Center for Supercomputing Research and Development*), na Universidade de Illinois em Urbana-Champaign. O objetivo do Parafrase é dar suporte às pesquisas do CSRD e futuramente ser utilizado como reestruturador para o multiprocessador Cedar que lá está sendo desenvolvido.

   O Parafrase opera basicamente de dois modos: reestruturação automática e interativa. No modo automático, que funciona basicamente em *batch*, o usuário fornece um arquivo onde estão indicadas as técnicas a serem aplicadas e sua ordem. No modo interativo, o usuário seleciona as técnicas através de um menu de técnicas.

A saída em ambos os casos é o programa original reestruturado de acordo com as especificações.

Para dar melhor suporte ao usuário, o Parafrase tem ferramentas para visualizar o grafo de dependências, o que pode ser utilizado para, por exemplo, auxiliar na seleção das técnicas a serem aplicadas.

3. **O Sigma:** O Sigma [GALS 87] está em desenvolvimento na Universidade de Indiana em Bloomington e é utilizado para estudos sobre reestruturação e atividades relacionadas, como por exemplo, análise de desempenho e paralelização de programas.

   O Sigma dá mais ênfase à reestruturação manual de programas. Assim como o Parafrase, ele permite que o usuário reestruture seu programa com o auxílio de um menu que contém as técnicas a serem aplicadas. Porém o Sigma contém ferramentas mais avançadas para dar suporte ao usuário, como um editor e as ferramentas que utilizam-se da estrutura de armazenamento intermediária, o grafo do programa.

2.1.1 O Projeto Computação Científica (PCC)

O PCC não é apenas um ambiente de reestruturação de programas. Trata-se de um projeto integrado de hardware e software que tem como objetivo construir um computador paralelo de alto desempenho em processamento ponto flutuante cuja operação privilegie o usuário [Maci 90, MVMP 90]. A integração de hardware e software evita que ocorram os erros conceituais comuns aos projetos computacionais desenvolvidos em torno de um conceito de hardware, mas que acabam não tendo um suporte adequado de software e vice-versa.

Na parte de hardware, o objetivo é construir uma máquina paralela composta de quatro processadores MIPS-R3000 e uma memória compartilhada. Ela atua a partir de um computador hospedeiro (por exemplo um PC-386) onde se encontram os compiladores, o sistema operacional e o ambiente que descreveremos a seguir.

A máquina paralela executa somente os programas dos usuários. Esses programas são enviados em fila pelo computador hospedeiro para que sejam executados um de cada vez pela máquina paralela. Portanto, a máquina paralela não visa alta utilização dos processadores, mas sim acelerar a execução de cada programa individualmente.

Na parte de software, o objetivo é construir um ambiente de reestruturação de programas que explore tanto o paralelismo explícito (com extensões paralelas) quanto o paralelismo implícito (sem extensões paralelas) de programas FORTRAN. Nos dois casos,

os programas, ao serem compilados, geram código para a máquina paralela. Essa característica destaca o PCC dos outros ambientes citados. Enquanto Parascope, Parafrase e Sigma tem como saída o programa original reestruturado, o PCC gera código para a máquina alvo.

As ferramentas do ambiente procuram atingir tanto o usuário mais experimentado com paralelismo quanto aquele com menor conhecimento. Assim, para cada ferramenta interativa existe uma ferramenta automática que realiza a mesma função. Desta maneira incentiva-se o usuário menos experiente (que tipicamente utiliza as ferramentas automáticas) a procurar um melhor desempenho de seus programas nas ferramentas interativas.

O grafo do programa contém todas as informações sintáticas e semânticas necessárias para reproduzir o programa, acrescido das dependências e diversas informações adicionais. A cada programa está associado um único grafo que o representa. Entretanto, o grafo não apresenta informações léxicas pormenorizadas, como por exemplo o número de brancos entre duas variáveis. Logo, a cada grafo correspondem múltiplos programas fonte, todos equivalentes ao programa original.

O grafo possui basicamente quatro níveis hierárquicos. O nível superior é de arquivos, onde estão representados os arquivos que compõe o programa. O nível seguinte é de comandos, exemplificado na figura 2.2, onde os vértices representam os comandos do programa Fortran da figura 2.1. No exemplo os nós retangulares representam comandos e os circulares servem apenas para ligação.

No nível intermediário os vértices representam o nível interno dos comandos (por exemplo, expressões). Por fim, o nível inferior do grafo contém as tabelas de símbolos, tipos, rótulos e comentários.

O grafo define um padrão para a comunicação entre as ferramentas. Cada ferramenta usa como entrada o grafo do programa e produz como saída o mesmo grafo acrescido de informações específicas. As exceções são o editor e o parser que recebem o texto do programa e geram o grafo e o gerador de código que recebe o grafo e produz código de máquina.

A seguir, relacionamos as ferramentas que compõem o ambiente de reestruturação de programas.

1. Parser: A entrada para o parser é um programa FORTRAN-77 com ou sem extensões paralelas. Ele faz a análise sintática e semântica do programa e produz, como saída, o grafo do programa.

2. Editor: Trata-se de um editor dirigido à FORTRAN que faz a análise sintática

```
      PROGRAM TESTE
      INTEGER I,J
      REAL Z(100,100),W(100)
      DO 20 I = 1,100
         W(I) = 1.0
         DO 10 J = 1,100
            IF (I.GT.J) THEN
               Z(I,J) = 2.0
            ELSE
               Z(I,J) = 0.0
            ENDIF
10       CONTINUE
20    CONTINUE
      CALL SUBROT(Z,W)
      STOP
      END
```

```
      SUBROUTINE SUBROT(X,Y)
      REAL A,X(100,100),Y(100)
      INTEGER I,J
      DO 20 I = 2,100
         DO 10 J = 2,100
            A = X(I,J) + X(I,J-1)
            X(I,J) = A * Y(J)
10          CONTINUE
20       CONTINUE
      RETURN
      END
```

Figura 2.1: Um programa Fortran

simultânea à edição. Ele produz como saída o grafo do programa e um arquivo contendo o texto do programa.

3. Analisador de Dependências: Detecta, a partir do grafo do programa, as dependências existentes entre os comandos. O usuário pode inserir, remover e examinar as dependências via menu. A saída é o grafo do programa aumentado com as dependências.

4. Detector de Paralelismo: Recebe como entrada o grafo aumentado com as dependências. Transforma os laços seqüenciais em laços paralelos, permitindo tanto a reestruturação automática quanto a reestruturação manual via ferramentas semi-automáticas, onde o usuário seleciona os trechos a serem trabalhados e os métodos a serem empregados nesses trechos via menu. A saída é o grafo transformado.

5. Explorador de Paralelismo: A entrada do Explorador é o grafo paralelizado. O explorador escolhe o método de alocação de processadores em cada laço paralelo. Escolhe, em um aninhamento de laços paralelos, quais serão executados seqüencialmente e quais serão executados em paralelo, em função das características da máquina alvo. Permite tanto exploração automática quanto manual. No caso de exploração manual, o usuário escolhe o laço e o método a empregar via menu. A saída do Explorador é o grafo munido das políticas de alocação.

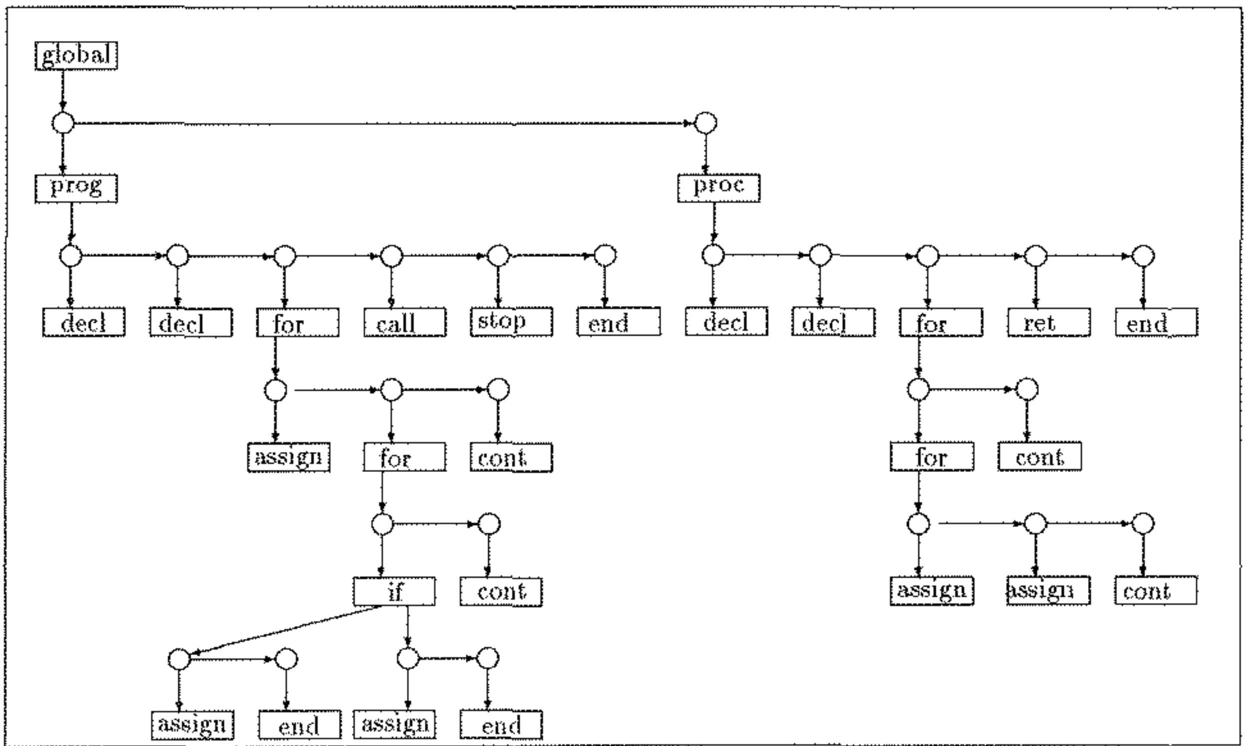


Figura 2.2: O grafo do programa Fortran apresentado, a nível de comandos

6. Gerador de Código: Gera código para a máquina paralela a partir do grafo do programa munido de políticas de alocação.

Todas essas ferramentas comunicam-se com o usuário via uma interface de comunicação padrão. A interface e o grafo devem ser compreendidos como "trilhos" nos quais as ferramentas são encaixadas (figura 2.3).

2.2 Interfaces

Este trabalho concentra-se nas interfaces de software, mais especificamente em interfaces do usuário, que também são chamadas interfaces homem-máquina ou interfaces de comunicação.

A interface do usuário é a ferramenta computacional responsável pela interação entre o usuário final de uma aplicação e o terminal (vídeo, teclado e mouse) usado por essa aplicação [SLL 86]. A interação usuário-interface leva em consideração dois aspectos denominados LOOK e FEEL. O LOOK trata da forma e disposição com que a interface mapeia as informações enviadas pela aplicação ou pela própria interface no terminal de

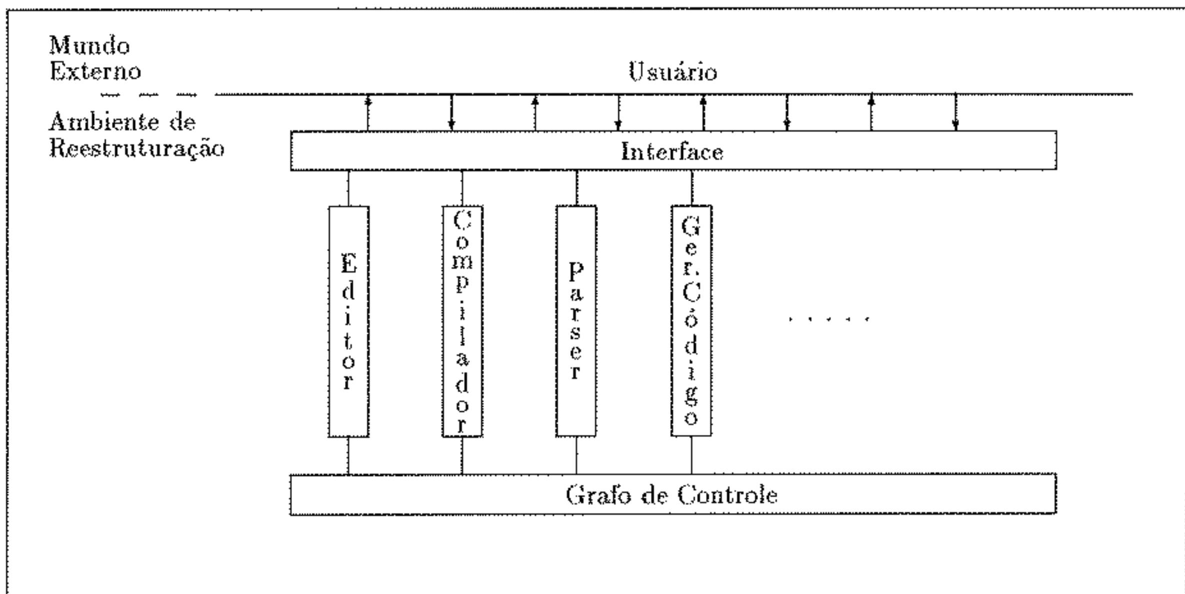


Figura 2.3: O Ambiente de Re-estruturação do PCC

vídeo, enquanto que o FEEL trata das informações que o usuário enviou à interface via teclado ou mouse.

Assim, a interface funciona como um filtro que faz com que somente as informações pertinentes passem de um extremo (usuário ou aplicação) a outro (aplicação ou usuário).

Um dos objetivos da interface de comunicação é permitir independência de diálogos [HaHi 89], ou seja, que as decisões que afetam o diálogo com o usuário sejam isoladas daquelas que afetam a aplicação. Isso significa que a maneira com que se processa a interação com o usuário é desconhecida pela aplicação. Assim, alterações no diálogo normalmente não irão refletir na aplicação e vice-versa.

A interface do usuário é composta por várias camadas e normalmente utiliza programas gráficos portáveis como o PHIGS, o GKS e o X Window como camada inferior. Essa camada é responsável tanto pelo mapeamento de informações para o terminal de vídeo quanto pelo recebimento de ações efetuadas no teclado. Desta maneira ao utilizar as funções providas pelo pacote gráfico, a interface (e em última instância a aplicação) tornam-se tão portáveis quanto o pacote gráfico. Além disso, o modelo de camadas permite uma atenção maior com o estilo de interação, despreocupando-se do modo com o qual ela é efetuada.

As seções seguintes destinam-se a explicar algumas dessas camadas, assim como introduzir alguns dos conceitos que serão utilizados ao longo do texto. Inicia-se com janelas, base para quase todas as interfaces que utilizam terminais de vídeo com *bitmap*

displays[1]. Em seguida explica-se o que é a interface com o programa de aplicação e os conceitos do pacote gráfico escolhido para a implementação.

2.2.1 Janelas

Janelas são áreas retangulares do terminal de vídeo que permitem que sejam simulados diversos terminais lógicos em um mesmo terminal físico. Desta forma, as entradas e saídas de diferentes processos podem ser mantidos fisicamente separados.

Gerenciadores de Janelas

Em ambientes multi-janelas, o gerenciador de janelas é responsável pelo mapeamento das informações para as janelas (ou seja, ele mapeia as entradas e saídas de cada janela para a área retangular do terminal de vídeo) e por gerenciar as alterações na posição e tamanho das janelas. Isso traz vantagens tanto para o usuário quanto para os programadores de aplicações que utilizam janelas. No caso do usuário, ele provê um conjunto de funções padrão para todas as janelas, o que facilita a interação. No caso de programadores, o gerenciador resolve os problemas referentes a mapeamento, permitindo dedicação exclusiva do programador à aplicação.

Já que existe somente um teclado, porém múltiplas janelas, é necessário identificar para qual janela a ação de teclado será direcionada. Os gerenciadores de janelas normalmente referem-se a essa janela com o nome de *janela ativa* [2], destacando-a das demais para que o usuário possa identificá-la visualmente. Existem duas formas para o usuário indicar a janela ativa utilizando o mouse [ScGe 86]. Na primeira, *Real State*, a janela ativa é aquela sobre a qual o mouse está, enquanto que no segundo modo, *listner*, a janela ativa é explicitamente indicada com a ação do mouse.

A seguir, apresentam-se os paradigmas usados por gerenciadores de janelas para comportar várias janelas em um mesmo terminal de vídeo.

Paradigmas Quando várias janelas compartilham um mesmo terminal de vídeo, podem ocorrer indefinições nas áreas que as janelas se interceptam. Por exemplo, suponha que uma janela ocupe toda a área física de um terminal e que sobre esse terminal deseja-se criar uma outra janela. Onde posicionar essa nova janela?

---

[1] Terminais de vídeo com bitmap displays são terminais gráficos de alta resolução onde cada ponto na tela corresponde a um endereço ou bit de memória.

[2] A literatura apresenta outros nomes para janela ativa. Myers [Myer 84] utiliza o nome *Listner*, enquanto que o X Window utiliza-se do nome *focus*. Porém, no decorrer do texto, utiliza-se o nome janela ativa por compreeender que ele é o que faz mais sentido em português.

Os Gerenciadores de Janelas resolvem essa situação utilizando um dos dois paradigmas apresentados a seguir. O primeiro paradigma não permite que uma janela fique sobre a outra. Assim, todas as janelas existentes são colocadas lado a lado no terminal, como se fossem azulejos. Por essa razão, os gerenciadores que utilizam este paradigma são chamados *Tiled*[3] *Window Managers*. O segundo paradigma permite sobreposição de janelas. Assim, as janelas comportam-se como se fossem folhas de papel sobre uma mesa, encobrindo e sendo encobertas quando necessário. Os gerenciadores que utilizam este paradigma são chamados *Overlapping*[4] *Window Managers*. Ambos os paradigmas tem virtudes e defeitos detalhados a seguir.

Tiled Window Managers Nesse paradigma, as janelas estão sempre totalmente visíveis, ou seja, não é permitida a sobreposição de janelas. O gerenciador fica responsável pela maximização do uso da área do terminal de vídeo. Assim, ele gerencia a localização, tamanho e efeitos colaterais das janelas para manter seus conteúdos visíveis o tempo todo. Quando a localização ou tamanho de uma janela muda, as outras janelas são relocadas e redimensionadas se necessário, mas nunca encobertas [BlRo 86].

Um gerenciador desta classe divide a tela em áreas sendo que uma janela pode ocupar uma ou mais áreas. Quando uma janela é aberta, o gerenciador escolhe uma das áreas disponíveis. Caso não existam áreas disponíveis ele desaloca áreas das janelas que ocupam mais de uma área.

O usuário pode controlar a altura e posição de uma janela. Porém, essas alterações podem perder o efeito quando outras janelas forem abertas. Se posteriormente houver oportunidade de satisfazer as especificações do usuário, o gerenciador o fará.

A vantagem do Gerenciadores *Tile* é simplificar a tarefa do programador, uma vez que não é necessário especificar tamanho, posição, etc., o que é feito automaticamente pelo gerenciador.

Sua grande desvantagem está na dificuldade que proporciona ao usuário quando existem muitas janelas abertas ao mesmo tempo, pois todas acabam sendo pequenas demais, dificultando a leitura por parte do usuário. Uma outra desvantagem está no número limitado de janelas que podem ser mostradas ao mesmo tempo.

Overlapped Window Managers Nesse paradigma, as janelas podem sobrepor-se. Ele foi criado nos laboratórios da Xerox em Palo Alto, PARC (*Palo Alto Re-*

[3]Gerenciadores de Janelas Azulejadas. Como esse nome é bastante incomum em português, será adotada a versão em inglês (Tile) no decorrer do texto.

[4]Gerenciadores de Janelas Sobrepostas. A versão em português tem sido freqüentemente utilizada, mas para manter uma unicidade com o paradigma *Tile*, será utilizado o nome em inglês

search Center), e uma das primeiras aplicações comerciais desenvolvidas utilizando desse paradigma foi o SMALLTALK [Tesl 81].

Esse paradigma é mais flexível, permitindo a criação de janelas de tamanho e posição arbitrários na tela.

Existe uma ordenação entre as janelas, onde as janelas de maior ordem cobrem as de menor ordem. Mesmo aquelas que não cobrem e não são encobertas estão ordenadas.

Como as janelas podem ficar encobertas, são fornecidos mecanismos para que o usuário possa tornar as janelas total ou parcialmente visíveis. Normalmente, os gerenciadores colocam as seguintes operações à disposição do usuário [Myer 84]:

- Escolher a janela ativa.
- Trazer uma janela para o topo, fazendo com que ela fique totalmente visível.
- Colocar uma janela abaixo de todas as outras, fazendo com que as janelas que ela encobria apareçam.
- Movimentar uma janela ao longo da tela.
- Redimensionar uma janela, aumentando ou diminuindo-a.
- Reduzir a janela a uma pequena figura, denominada *ícone*, ganhando espaço na tela para outras janelas.
- Remapear o ícone na janela original.

Assim como no paradigma anterior, existem vantagens e desvantagens na utilização de gerenciadores de sobreposição de janelas. A maior vantagem está em permitir várias janelas grandes, o que não é possível no paradigma tile. Além disso, o usuário pode simular o paradigma anterior evitando a sobreposição e fazendo melhor uso do espaço disponível [Myer 86].

A grande desvantagem ocorre quando existem muitas janelas grandes na tela, o que pode deixar o usuário confuso ou pode requerer muitas operações do usuário para buscar janelas escondidas. Mesmo assim, atualmente a maioria dos gerenciadores de janelas utilizam o paradigma de sobreposição [Myer 86].

Componentes

Esta seção apresenta algumas das áreas auxiliares que fazem parte de uma janela, tais como área de trabalho, menus, título, etc.. A figura 2.4 mostra uma janela com grande parte desses componentes.

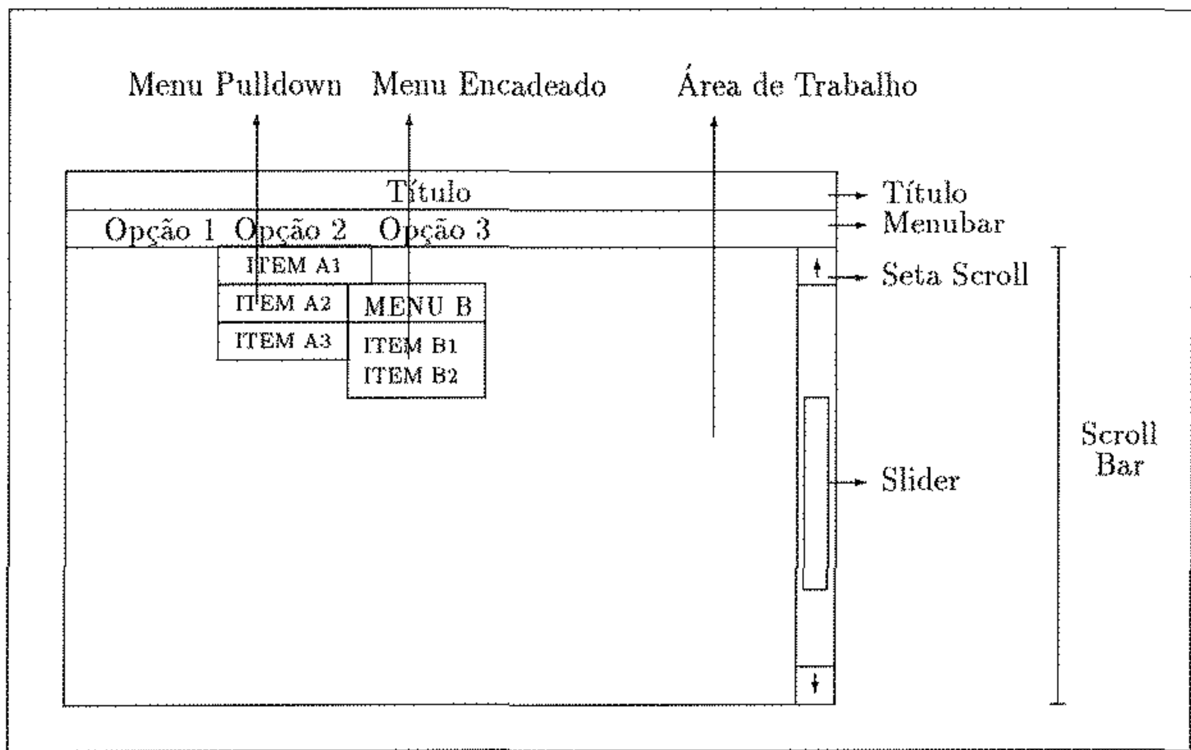


Figura 2.4: Modelo de Janela com seus Componentes

Título A janela pode ter um nome, indicando, por exemplo, qual o processo que está sendo executado. Em alguns gerenciadores de janelas, ela adicionalmente informa qual a janela ativa, alterando a cor de fundo, destacando-a das demais.

Área de Trabalho É o local da janela que contém as informações mais relevantes ao usuário. Por exemplo, se o processo que está sendo executado na janela for o editor, a área de trabalho estará preenchida com o texto.

Menus Os menus são áreas retangulares (que normalmente sobrepõem-se à janela) compostos por um conjunto de opções selecionáveis pelo usuário, denominados itens de menu[5]. Eles são parte integrante do paradigma de sobreposição de janelas desenvolvido no PARC [Tesl 81].

Menus podem ser usados para diversos propósitos e economizam espaço do terminal de vídeo para informações mais relevantes ao usuário. Caso o usuário sinta a necessidade de alguma informação adicional, ou troca de modo de operação, ele deve selecionar o menu correspondente. Além disso, pode haver um encadeamento de menus, onde a

[5] *Buttons*

seleção do item de menu invoca outro menu.

Existem três tipos de menus: Menubar, Menu Pulldown e Menu Popup.

Menubar Como mostra a figura 2.4, o menubar é um menu que está sempre visível na janela. Ele mostra pequenos textos, normalmente de uma palavra, que servem para auxiliar o usuário a encontrar a informação que necessita, ou executar alguma função. O menubar é o único menu que está sempre totalmente visível. Ele é parte integrante da janela, ao contrário do pulldown e popup, que aparecem somente quando são solicitados e desaparecem após alguma seleção.

Pulldown Menu Um pulldown menu normalmente é acionado através de alguma ação do usuário sobre a menubar, desenrolando-se abaixo da opção selecionada.

Pode haver encadeamento de menus (pulldown ou popup) a partir de um pulldown, mas normalmente após selecionada uma opção ele, e todos os menus que foram abertos a partir dele, desaparecem.

Menu Popup O menu popup pode aparecer em qualquer posição da janela e é normalmente acionado através de uma ação de mouse ou conjunto de teclas. Ele segue as mesmas regras de formação e desaparecimento dos pulldown menus.

Scrollbar

Muitas vezes, as informações que devem ser mostradas ficam restritas aos limites físicos da janela ou do próprio terminal de vídeo. Isto acontece, por exemplo, no caso de editores de texto, onde a área de trabalho só comporta um trecho do texto.

Na figura 2.4, há uma scrollbar vertical. Normalmente existe uma scrollbar para cada área de trabalho. Existem duas maneiras de movimentar o conteúdo da área de trabalho: a partir das setas e movimentando o *slider* (vide figura 2.4).

O tamanho do slider varia de acordo com a quantidade de informação mostrada na área de trabalho. Quanto maior o slider com relação à área da scrollbar, maior quantidade, do total de informações, que a janela mostra. Se o slider ocupar toda área da scrollbar, a área de trabalho apresenta toda a informação disponível.

2.2.2 Interface com o Programa de Aplicação

A Interface com o Programa de Aplicação (*Application Programming Interface*-API) é o conjunto de chamadas de funções e as estruturas de dados usados pelo programador

da aplicação para manipular a interface de comunicação [SLL 86].

A API não deve ser confundida com a interface em si. A interface é o programa que mapeia textos e gráficos para a tela. A API é o meio de comunicação entre a interface e a aplicação. A API contém as funções que a aplicação utiliza para agir sobre a interface.

2.2.3 X Window

O X Window (ou simplesmente X) é um software desenvolvido no MIT (*Massachusets Institute of Technology*) que permite o desenvolvimento de interfaces de comunicação gráfica portáveis [Youn 90, JoRe 90]. Uma de suas características mais importantes é a arquitetura independente de máquina. O X permite que as aplicações funcionem em qualquer hardware que suporte o protocolo X (*X Protocol*) sem modificar, recompilar ou religar[6] a aplicação [Jones 89]. A independência de hardware aliada a uma API padronizada permitem comunicação entre aplicações baseadas em X em um ambiente heterogêneo contendo computadores de grande porte, estações de trabalho e computadores pessoais.

Sua API com a linguagem 'C' (chamada Xlib) é composta por um conjunto muito estenso o e complexo de funções e não apresenta mecanismos específicos para o uso dos componentes das janelas. Assim, foi desenvolvida uma camada sobre o X, chamada *Toolkit Intrinsics* (Xt Intrinsics) que facilita a criação de interfaces de comunicação ao combinar um conjunto de componentes conhecidos como *widgets*.

Os toolkits possuem algumas características de softwares orientados a objetos. Widgets são um tipo abstrato de dado (classe) que contém um certo número de recursos (atributos) e operações (métodos ou procedimentos) que estão associados a uma janela X [SLL 86]. Este conceito é estendido às interfaces construídas sobre os toolkits. Assim, o programador deve indicar a função de retorno que deve ser disparada quando ocorrer determinado evento em um widget.

Por exemplo, a figura 2.4 retrata vários widgets, como Menubar, itens de menu, Scrollbar, etc.. Cada um deles tem uma forma gráfica definida e aceita um conjunto de ações do usuário. Cada ação corresponde a um evento. Uma vez ocorrida uma ação, o toolkit indica à aplicação o widget em que houve o evento e o evento ocorrido, disparando a função de retorno associada àquele evento.

O conjunto de widgets provido pelo Xt Intrinsics é reduzido e pobre, mas existem programas que proporcionam um grupo adicional de widgets. As aplicações desenvolvidas sobre X Window usam freqüentemente uma combinação de Xlib, Xt Intrinsics e um

---

[6] relink

conjunto adicional de widgets dentre os quais destacamos os oferecidos pelo Open Look e o OSF/Motif.

A seguir apresenta-se um modelo de referência baseado em [SLL 86] que mostra as várias camadas do X e como elas se relacionam.

1. **Camada Física:** Esta camada destina-se a mapear os dados para tela e recebê-los do teclado ou mouse. É a camada do Protocolo X, que também provê mecanismos para comunicação em redes de computadores.

2. **Interface do Sistema de Janelas:** Esta camada fornece um conjunto de primitivas que manipulam janelas e gráficos. A Xlib está nesta camada.

3. **Toolkit:** Esta camada fornece uma base sobre a qual podem ser criados os widgets. Além disso, fornece um modelo geral de manipulação de widgets. O Xt Intrinsics está nesta camada.

4. **Widgets:** Esta camada fornece um conjunto adicional de widgets que contém look e feel padronizados. O Motif e o Open Look estão nesta camada.

   Devido ao forte relacionamento entre esta camada e a anterior, será utilizado o nome toolkit para referenciar ambas.

5. **Apresentação:** As interfaces dessa camada já são de um nível mais alto. Elas são ferramentas que simplificam a escolha do que mostrar em uma janela e em que posição. Nesta camada estão o UIL (*User Interface Language*) do Motif e o sistema de gerenciamento de formas (Widget FORM) do Open Look.

6. **Diálogo:** Esta camada é muito semelhante à anterior, porém gerencia diálogos.

7. **Aplicação:** A última camada inclui o código da aplicação e os mecanismos de comunicação entre a interface e o usuário.

Os Gerenciadores de Janelas encaixam-se nas camadas 5 e 6, utilizando ferramentas de alto nível para especificar os widgets usados e fornecendo uma forma de comunicação padrão tanto para o programador quanto para o usuário.

2.3 A Interface do Usuário do PCC

Os tópicos anteriores fornecem subsídios para uma definição mais detalhada do propósito deste trabalho.

O objetivo deste trabalho é construir uma interface de comunicação para o ambiente de reestruturação de programas do PCC.

Escolheu-se o X Window como pacote gráfico, utilizando-se o OSF/Motif como toolkit. Além de prover os widgets necessários para a implementação deste trabalho, o Motif contém facilidades para comunicação entre as ferramentas do ambiente.

A interface é composta por um conjunto de widgets divididos em dois grupos. O primeiro contém componentes tradicionais em janelas (menus, scrollbars, etc.) enquanto o segundo é composto por widgets específicos para ambientes de reestruturação de programas.

O primeiro grupo de widgets é um sub-conjunto daqueles providos pelo Motif, com extensões. Este toolkit apresenta uma forma de interação com o usuário (look e feel) definida. Assim, a interface concentra-se na interação destes widgets com as aplicações.

O segundo grupo é composto por widgets que auxiliam na reestruturação de programas. Dentre eles, destaca-se aquele que desenha hierarquias (grafos dirigidos com várias camadas de vértices). Hierarquias são freqüentemente usadas como abstrações onde os vértices representam elementos e as arestas representam as relações entre eles.

Para o ambiente em questão, os vértices podem ser usados para representar um ou mais comandos do programa. As arestas podem ter diversos usos. Por exemplo, representar a ordem dos comandos no texto do programa, a árvore sintática e a seqüência de execução. Adicionalmente, as arestas podem representar as dependências entre comandos.

Este widget facilita a construção de ferramentas que fazem abstrações gráficas de programas, localização de dependências, acompanhamento de execução, etc..

Grande parte desta dissertação relaciona-se com o algoritmo que desenha hierarquias, o que envolve heurísticas para resolução de um problema NP-completo. O capítulo 4 é dedicado a este tópico.

Capítulo 3

Os Widgets da Interface

Este capítulo apresenta os widgets da interface. A seção 3.1 detalha dois tópicos da filosofia de funcionamento do Motif, e como eles foram adaptados à interface do PCC. Em seguida, detalha-se os dois grupos de widgets desenvolvidos para a interface. A seção 3.2 explica os widgets tradicionais da interface, compostos por um sub-conjunto dos widgets do Motif ampliados com extensões que facilitam o seu uso no PCC. A seção 3.3 explica os widgets específicos ao ambiente do PCC, que facilitam a interação com o usuário em tarefas de reestruturação de programas.

3.1 Preliminares

Como já foi mencionado na seção 2.2.3, o Motif é um software dirigido a eventos, e os softwares desenvolvidos sobre ele devem seguir esta filosofia. A seguir serão detalhados dois tópicos desta filosofia e as adaptações efetuadas na interface.

3.1.1 Hierarquia entre Widgets

O X Window organiza as janelas em uma hierarquia, denominada *árvore de janelas*. A janela no topo da árvore é conhecida como *janela raiz*. Cada janela pode ter janelas *filhos* e, com exceção da raiz, todas tem uma janela *pai*.

Esta hierarquia é estendida aos toolkits construídos sobre o X, porém ao invés dos nós serem janelas, eles são widgets.

Os widgets da interface do PCC também estão organizados hierarquicamente. A árvore da figura 3.1 apresenta a hierarquia existente entre os widgets tradicionais da interface, explicados na seção 3.2, enquanto árvore da figura 3.2 corresponde ao widget hierarquia, um dos widgets específicos para ambientes de reestruturação, explicado na

seção 3.3. Os widgets *menubar* e *menu popup* tem o mesmo significado em ambas as figuras.

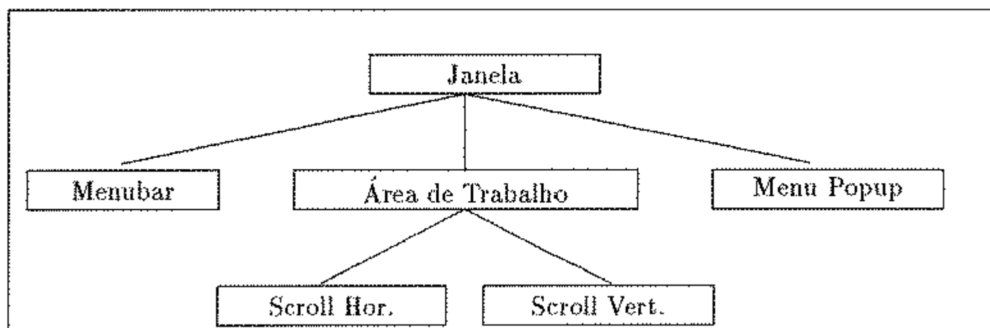


Figura 3.1: Os Widgets Tradicionais da Interface

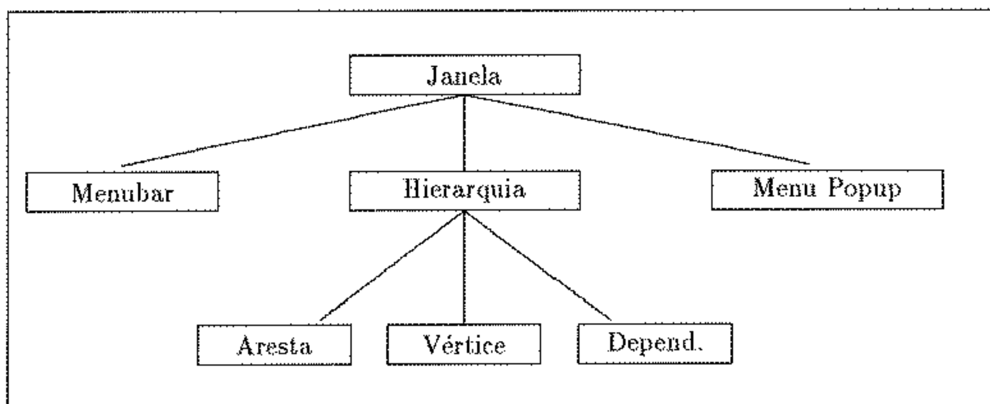


Figura 3.2: Widgets Hierarquia

Observe que na figura 3.2 os widgets Área de Trabalho, Scroll Horizontal e Scroll Vertical desapareceram. Isto aconteceu porque o widget Hierarquia foi construído sobre o widget Área de Trabalho. A vantagem é que o widget Hierarquia gerencia totalmente as scrollbars, sem necessidade de especificação do usuário.

3.1.2 Funções de retorno

No Motif, a interação com o usuário ocorre através de funções de retorno. O programador seleciona os eventos a serem tratados em cada widget, indicando a função de retorno que deve ser disparada ao ocorrer o evento.

Esta filosofia foi estendida à interface do PCC, porém foram introduzidas diferenças na sintaxe das funções que selecionam os eventos para cada widget e em um dos campos da função de retorno. Isto ocorreu para eliminar informações desnecesárias para a aplicação.

A seguir, explica-se a sintaxe das funções do Motif, para, em seguida, apresentar as alterações efetuadas para a interface do PCC.

No Motif

Cada widget do Motif trata um conjunto específico de eventos. Este tratamento inclui funções de retorno para cada evento. Para incluir uma função de retorno `funcao` em um widget `widget` do Motif, utiliza-se a função `XtAddCallback()`, cuja sintaxe é apresentada abaixo.

```
XtAddCallback(widget, callback_name, funcao, client_data)
```

Onde `client_data` será retornado por `funcao` e `callback_name` indica o tipo de evento que dispara `funcao`. Por exemplo, no caso do mouse, a função pode ser disparada quando o mouse for acionado no widget, quando o mouse entrar ou quando sair do widget. Cada uma destas situações é diferenciada neste campo.

A sintaxe das funções de retorno do Motif é a seguinte:

```
void CallbackProcedure(widget, client_data, call_data)
```

onde:

- **widget:** widget onde ocorreu o evento.
- **client_data:** é o parâmetro passado pela aplicação através da função `XtAddCallback()`.
- **call_data:** traz informações específicas do evento ocorrido. Por exemplo, para um evento de mouse, apresenta o tempo decorrido desde o último evento de mouse, as coordenadas em que ocorreu o evento em relação à raiz e ao widget, etc..

As funções de retorno do Motif foram projetadas para gerenciar uma grande variedade de eventos e widgets, podendo ser utilizado por uma grande variedade de aplicações.

Na interface

Cada widget da Interface, assim como no Motif, trata um conjunto específico de eventos através da inclusão de funções de retorno para cada evento tratado.

A interface do PCC, porém, utiliza um número reduzido de widgets e eventos. Assim, procurou-se facilitar a inclusão de funções de retorno e tornar o campo `call_data` mais específico para as necessidades de cada widget da interface do PCC.

A função `XtAddCallback()` foi eliminada na interface. Em seu lugar, incluiu-se uma função para cada evento. Por exemplo, para indicar a função de retorno que tratará de eventos de mouse em uma área de trabalho, utiliza-se a função `IUmouse()`. Para ação de teclado, utiliza-se a função `IUteclado()` e assim por diante.

Foi mantida a sintaxe das funções de retorno do Motif, especificada na função `CallbackProcedure()`. Porém, como o número de eventos tratados é menor e as ferramentas do PCC necessitam de informações mais particulares, alterou-se o valor retornado por `call_data`. No Motif, este campo é um apontador para uma estrutura de dados que contém informações específicas para cada evento ocorrido, mas apresenta informações desnecessárias às ferramentas do PCC.

Na interface do PCC, este campo também é um apontador para uma estrutura. Porém, ela contém informações mais particulares. Por exemplo, no caso de ação de mouse, a estrutura conterá a coordenada (x,y), relativa à área de trabalho, em que ocorreu o evento.

O apêndice A apresenta as funções que associam funções de retorno para eventos, enquanto que o apêndice B apresenta o arquivo `IUEvent.h`, onde estão especificadas tais estruturas, uma para cada evento tratado.

3.2 Widgets Tradicionais

Os widgets tradicionais correspondem aos elementos de uma janela apresentados na seção 2.2.1.

3.2.1 Janela

Este widget é a janela externa da interface. Ao ser utilizada a função `IUabreJanela()`, cria-se esta janela e a estrutura *Janela*.

Ele contém três widgets filhos: menubar, menu popup e área de trabalho, e não trata nenhum tipo de interação com o usuário.

3.2.2 Área de Trabalho

A área de trabalho é o widget que mais freqüentemente interage com o usuário.

Inicialmente, explica-se como e em que casos a aplicação é notificada dos eventos do usuário. Em seguida, explica-se o caminho inverso, com a aplicação enviando informações ao usuário.

Usuário → Aplicação

A área de trabalho pode gerenciar quatro eventos. Dois deles, *expose* e *resize*, correspondem a ações na forma da janela, enquanto que os outros dois, *teclado* e *mouse*, correspondem às ações efetuadas sobre o teclado e o mouse, respectivamente.

- *Expose*: Um evento de expose é uma solicitação para redesenhar a área de trabalho. Ele pode ser gerado por ações do usuário ou da aplicação. No caso do usuário, ações como redimensionar a janela, colocá-la sobre as demais, expandí-la, etc., geram este tipo de evento. No caso da aplicação, um evento de expose pode ser gerado por alguns comandos da Xlib. A função que associa uma função de retorno a este evento é `IUexpose()`, e a estrutura `EventExpose` indica qual área de trabalho deve ser redesenhada.

- *Resize*: Um evento de resize é gerado toda vez que a janela for redimensionada e sempre é gerado um evento de expose em seguida. Quando ocorrer um evento de resize, a aplicação deve indicar a nova altura e a nova largura do desenho que será copiada para a área de trabalho. A função `IUresize()` associa uma função de retorno, e a estrutura `EventResize` contém a nova altura e largura da janela.

- *Mouse*: Ocorre um evento de mouse sempre que o mouse for acionado na área de trabalho especificada. A função `IUmouse()` associa uma função de retorno para cada evento de mouse, e a estrutura `EventMouse` apresenta as coordenadas da área de trabalho onde ocorreu o evento.

- *Teclado*: Ocorre um evento de teclado sempre que alguma tecla for acionada e a janela tratada pela aplicação estiver ativa. A função `IUteclado()` associa uma função de retorno para cada evento de teclado, e a estrutura `EventTeclado` apresenta as coordenadas onde ocorreu o evento e o caractere digitado, representado em uma forma padrão do X Window, habilitando portabilidade.

Aplicação → Usuário

A área de trabalho pode ser utilizada para mostrar textos e/ou gráficos.

Atualmente existe apenas uma função para o envio de textos para a área de trabalho, `IUenviaChar()`, com sintaxe especificada na apêndice A. Esta função e a função de retorno para eventos no teclado, satisfazem as necessidades do editor, que é uma ferramenta do ambiente em fase final de implementação. Para o futuro, deverão ser criadas facilidades adicionais para as outras ferramentas do ambiente.

O processo de desenho envolve duas etapas.

1. Cria-se uma área auxiliar denominada *Pixmap*, onde será efetivamente realizado o desenho. Esta área auxiliar pode ser maior do que a área de trabalho, o que alivia as limitações de tamanho.

2. Copia-se a área auxiliar para a área de trabalho de acordo com a altura e largura da área de trabalho e a parte do desenho efetivamente mostrada. Altura e largura da janela são fornecidas pelas funções de retorno dos eventos de expose e resize, enquanto que a parte do desenho mostrada na área de trabalho é fornecida pelas funções de retorno da scrollbar.

3.2.3 Scrollbar Vertical

Este widget é incluído na área de trabalho através da função `IUscrollV()`, e associa funções de retorno para os eventos gerados por ações do usuário no slider ou setas de scroll.

O componente mais significativo da scrollbar é o slider. Seu tamanho com relação ao tamanho da scrollbar indica a proporção do total do desenho ou texto que está sendo apresentado na área de trabalho. Sua posição na scrollbar indica a parte do desenho que está sendo mostrada.

Estas proporções são alteradas toda vez que a janela for redimensionada ou devido a alguma ação do programador. Por isso, a aplicação necessita de uma função que altere estas proporções.

A sintaxe da função que efetua esta tarefa é a seguinte:

```
IUposicScrollV (at, tot_alt, inicio_desenho, altura_at);
```

onde `at` é a área de trabalho onde está a scrollbar, `tot_alt` é a altura total do desenho na área auxiliar, `inicio_desenho` é a coordenada da área auxiliar onde inicia-se o desenho efetivamente mostrado na área de trabalho e `altura_at` é a altura da área de trabalho.

3.2.4 Scrollbar Horizontal

Este widget é idêntico ao anterior, porém, cria uma scrollbar horizontal.

3.2.5 Menus

Desde o surgimento do SMALLTALK os menus tornaram-se freqüentes em ambientes de programação interativos, e foram desenvolvidas diversas facilidades para seu uso.

No caso do Motif há widgets que criam menus popup, pulldown menus e menubar. Cada item de menu também é um widget, sendo que existem widgets específicos para encadeamento de menus.

Porém, para criar menus são necessárias de dezenas a centenas de linhas de código, principalmente se houver encadeamento. Isto faz com que o processo de confecção de programas torne-se lento.

Em seu livro sobre o Motif, Young [Youn 90] apresenta algumas primitivas para agilizar a criação de menus a partir de estruturas estáticas do programa. Estas estruturas contém informações como título do menu, texto que deve aparecer em cada item, função de retorno associada, menu encadeado, etc..

Adaptou-se esta solução para a interface do PCC, permitindo a criação de menubar e menus popup, através de funções específicas para cada um deles.

Abaixo, apresenta-se a estrutura `IUstruct_menu`, similar àquela utilizada por Young. A diferença está na inclusão de aceleradores para os itens de menu.

```
typedef struct _struct_menu {
        char*   nome;             /* Texto do item de menu              */
        void    (*f_ret)();       /* Funcao de retorno associada        */
        char*   dado_ret;         /* Parametro de f_ret                 */
        struct _struct_menu*
                sub_menu;         /* Apontador p/ menu encadeado        */
        int     n_sub_itens;      /* Numero de itens do menu encadeado */
        char*   tit_sub_menu;     /* Titulo do menu  encadeado          */
        char*   acelerador;       /* Acelerador                         */
} IUstruct_menu;
```

O campo `nome` contém o texto que será mostrado no item de menu. `f_ret()` é a função a ser disparada caso este item seja selecionado e `dado_ret` é o parâmetro retornado nesta função. Caso exista um menu encadeado, seu título será `tit_sub_menu`, e `sub_menu` conterá um apontador para o menu encadeado e `n_sub_itens` indicará o número de ítens em `sub_menu`. Finalmente, `acelerador` pode ser utilizado para indicar um conjunto de teclas que disparam um item, como será visto no exemplo a seguir.

Abaixo exemplifica-se o funcionamento desta facilidade para criação de menus na interface, criando-se uma menubar.

```
/* -------------------------------------------------------------- */
/*          Funcao de retorno a ser disparada caso qualquer        */
```

```
/*          item de menu seja selecionado                              */

f_ret (w, client_data, call_data)
     Widget     w;                /* Widget onde ocorreu a selecao      */
     char*      client_data;      /* parametro associado a f_ret        */
     EventMenu* call_data;        /* Struct retorno p/ eventos em menu  */
{
     printf ("%s\n", client_data);
}

/* ------------------------------------------------------------------- */
/*          MenuA e MenuB com seus respectivos itens                   */

static IU_struct_menu MenuB {
    {"Item B1", f_ret, "Item B-1 selecionado"},
    {"Item B2", f_ret, "Item B-2 selecionado"}
};

static IU_struct_menu MenuA {
    {"Item A1", f_ret, "Item A-1 selecionado",        NULL, 0, NULL},
    {"Item A2", NULL,  NULL, MenuB, XtNumber(MenuB), "Titulo", NULL},
    {"Item A3", f_ret, "Item A-3 selecionado",        NULL, 0,
                                                       "Ctr <key> F"}
};

/* ------------------------------------------------------------------- */
/*          Programa Principal                                         */

void main(argc, argv)
     int      argc;
     char*    argv[];
{
                . . . .

     IUmenubar (j, MenuA, XtNumber(MenuA));
```

```
    . . . .
}
```

onde `XtNumber` é uma macro que retorna o número de itens existentes em uma estrutura de dados estática.

Este trecho de código inclui uma menubar na janela `j`. Os itens da menubar estão especificados em `MenuA`, e são `Item A1`, `Item A2` e `Item A3`, sendo que o segundo tem um menu encadeado, `MenuB`, cujo título é `Titulo`.

Se o usuário selecionar `Item A1`, será disparada a função `f_ret()`, onde `client_data` será igual a "Item A-1 selecionado". Se for selecionado `Item A2`, será aberto o menu `MenuB`, onde valem as mesmas regras de `MenuA`. Finalmente, se for selecionado `Item A3` ou se forem acionadas as teclas <Control><F> no teclado, será disparada a função `f_ret()`, com `client_data` igual a "Item A-3 selecionado".

Se fosse necessário criar um menu popup, com acelerador <Control><A> para o menu `MenuA`, a sua sintaxe seria:

```
IUpopup (j, MenuA, XtNumber(MenuA), "Ctr <key> A");
```

3.3 Widgets Específicos

Esta seção apresenta os widgets específicos ao ambiente do PCC.

Estes widgets destinam-se às tarefas envolvidas com depuração, análise e desempenho de programas, e à tarefa mais significativa para ambientes de reestruturação, em especial o PCC, que é a depuração de desempenho.

O objetivo da depuração de desempenho é obter o desempenho mais adequado de um programa para uma determinada máquina paralela. Para tal, o usuário reestrutura sucessivamente o programa seqüencial, comparando os resultados obtidos em cada novo protótipo.

O desempenho de cada trecho de programa é calculado incluindo-se pontos de parada em locais críticos, de onde são extraídos os desempenhos locais do programa e armazenados para análise posterior [McHe]. As informações extraídas podem ser, por exemplo, as tarefas executadas em cada processador, o tempo gasto em cada tarefa, dentre outras.

O método mais tradicional de análise das informações armazenadas é a análise textual, onde as informações são impressas, tipicamente, em dezenas ou centenas de páginas, para que possam ser estudadas pelo usuário.

Esta obviamente não é uma forma amigável de interação, e mesmo a inclusão de bancos de dados específicos, facilitando o estudo das informações geradas na análise

textual, não muda este panorama.

Um dos grandes problemas deste tipo de representação é a dificuldade de se abstrair as informações, área onde o uso de gráficos gera bons resultados.

A seguir serão apresentados três widgets gráficos propostos para a interface do PCC, que visam suprimir as deficiências da análise textual: eixos cartesianos, gráfico de acompanhamento e hierarquias.

É importante observar que os dois primeiros são propostas para uma futura implementação no ambiente PCC. Por isso, eles são apresentados em linhas gerais, exemplificados com as aplicações a que se destinam. Já o widget de hierarquias é a parte central deste trabalho, e será apresentado em maiores detalhes.

3.3.1 Eixos Cartesianos

Em ambiente de reestruturação, eixos cartesianos podem ser usados para abstrair as informações armazenadas nos pontos de parada, auxiliando a análise do usuário [LSVC 89].

A interface deverá prover dois tipos de eixos, apresentados na figura 3.3.

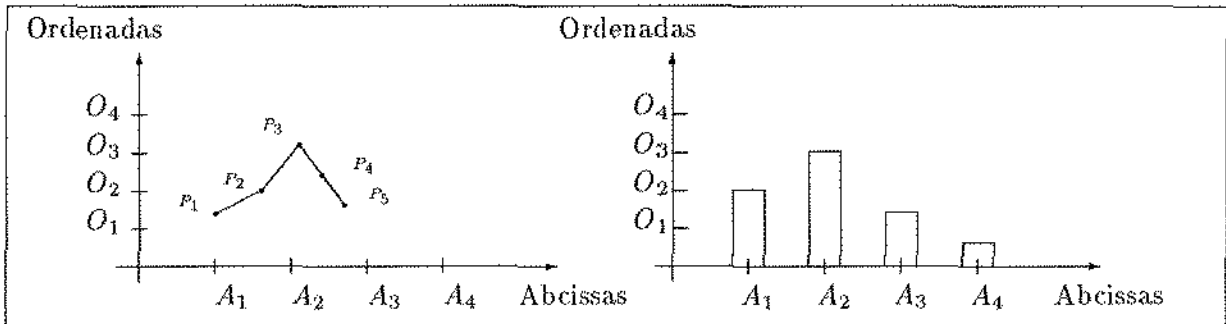


Figura 3.3: Aparência dos Dois Tipos de Eixos Cartesianos

No gráfico da esquerda, o eixo das abcissas valores contínuos, enquanto que no gráfico da direita, também conhecido como histograma, o eixo das abcissas representa valores discretos.

O gráfico da esquerda pode ser utilizado para mostrar as variações do programa de acordo com o tempo, como por exemplo, o desempenho do programa ao longo do tempo, o grau de utilização dos processadores de acordo com o tempo, etc..

Já o gráfico da direita pode ser utilizado para, por exemplo, indicar o grau de utilização de cada processador. Uma forma alternativa de representação é um gráfico do tipo torta, onde cada fatia corresponde ao grau de utilização de cada processador.

Propõe-se dois widgets de eixos cartesianos para a interface do PCC, um para cada gráfico apresentado.

O widget do grafo da esquerda deverá permitir o uso de vários tipos ou cores de linhas, possibilitando comparar os protótipos. Deverá conter também uma legenda onde os tipos ou cores de linhas sejam explicados, sendo que a legenda também será incluida no widget representado pelo eixo da direita.

3.3.2 Gráficos de Acompanhamento

Eixos cartesianos, mesmo sendo uma forma mais amigável de representar informações ao usuário, são limitados. A principal limitação está em não auxiliar na identificação das causas de determinados resultados.

O usuário é obrigado a procurar as razões destas falhas no programa original ou através de uma análise textual.

Em [LSVC 89], é apresentado um tipo especial de gráfico que permite visualizar este tipo de informação. Uma aplicação típica deste gráfico é apresentar o estado de cada processador ao longo do tempo, onde estado significa a tarefa que o processador estava executando em determinado momento da computação. Exemplos de estado incluem espera devido à sincronização, tentativa de acesso à memória, entre outros.

A figura 3.4 é um exemplo deste gráfico, onde cada linha *P1, P2, P3* e *P4*, apresenta o estado do processador em um determinado momento da computação.

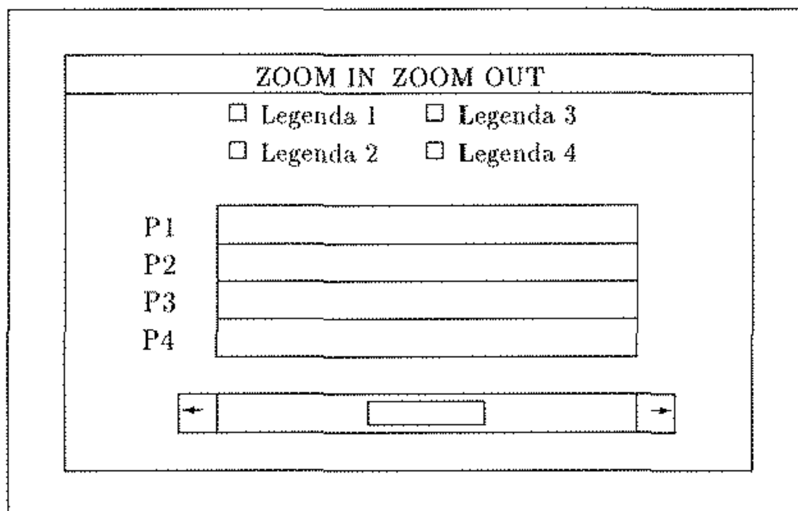


Figura 3.4: Aparência de um Gráfico de Acompanhamento

Além da informação do estado do processador, a figura contém uma legenda com a cor utilizada para representar cada estado, uma scrollbar e uma menubar cujos itens aproximam ou afastam os estados (*zoom in* e *zoom out*), permitindo maior ou menor abstração.

Este widget, interage com o usuário através de uma função de retorno. Ela passa à aplicação, o local em que houve ação do usuário, possibilitando que a aplicação apresente o trecho de programa que gerou aqueles estados para os processadores.

3.3.3 Hierarquias

Este widget é a parte central deste trabalho.

Hierarquias são uma classe especial de grafos dirigidos onde os vértices estão dispostos em camadas [Warf 77].

Representar graficamente uma hierarquia traz benefícios a usuários de ambientes de reestruturação de programas, uma vez que ela pode ser usada para representar graficamente o programa, auxiliando no exame de dependências, acompanhamento de execução, entre outros, de uma forma mais amigável.

Como já foi mencionado, o ambiente do PCC utiliza-se do grafo de controle como forma intermediária de armazenamento e uma ferramenta que possibilite desenhar tal grafo em terminais de vídeo é, sem dúvida, de grande valia ao usuário.

O desenho de hierarquias, porém, é uma tarefa complexa, e será analisada detalhadamente no capítulo 4. Esta seção concentra-se nas tarefas que este widget deverá executar. A seguir, explica-se o widget hierarquia e as funções já implementadas.

Objetivo

O objetivo deste widget é fornecer uma ferramenta de abstração ao ambiente do PCC, para que os programas do usuário possam ser analisados de uma forma conceitual, onde os vértices condensam linhas de código e as arestas indicam o fluxo de execução. Alternativamente, pode ser usado para mostrar a árvore sintática do programa.

Sobre estes desenhos, pode-se simular a execução do programa. Uma forma de fazê-lo é alterando as cores de vértices e arestas, onde cada cor representa um processador. Assim, o usuário pode acompanhar os trechos do programa que são executados em cada processador ao longo do tempo.

Componentes

Para implementar este widget utilizaram-se os widgets tradicionais já citados. Observe que a diferença entre as figuras 3.1 e 3.2 está na substituição do widget área de trabalho pelo widget hierarquia.

Além disso, desapareceram as scrollbars e surgiram três outros widgets: vértices, arestas e dependências. As scrollbars estão presentes no desenho, mas são totalmente

gerenciadas pelo widget hierarquia.

A seguir, detalha-se os atributos de cada novo widget, a partir de sua definição em `IUevent.h` (apêndice B).

Vértice A estrutura `TipoVertice` apresenta os seguintes atributos:

- `rotulo`: é o campo que identifica unicamente o vértice na aplicação
- `texto`: é o texto que será mostrado no desenho, sobre o vértice.
- `forma`: é a forma gráfica do vértice. Pode ser círculo, quadrado ou losango, uma para cada estrutura do programa (seqüência, iteração, condicional).
- `cor_forma`: é a cor usada no desenho do contorno do vértice.
- `cor_texto`: é a cor usada no texto.
- `Visibilidade`: indica se o vértice é invisível ou visível. Quando visível, o vértice pode ser representado de duas formas, mostrando somente os contornos do vértice ou hachurando seu interior.
- `Peso`: este atributo é utilizado no algoritmo para desenhar hierarquais, e será analisado no próximo capítulo.

Aresta A estrutura `TipoAresta` apresenta os seguintes atributos:

- `rotulo_origem`: indica o vértice origem da aresta.
- `rotulo_destino`: indica o vértice destino da aresta.
- `texto`: é o texto que será mostrado no desenho, sobre a aresta.
- `cor_aresta`: é a cor usada no desenho da aresta.
- `cor_texto`: é a cor usada no texto.
- `Visibilidade`: indica se a aresta é invisível ou visível.

Observe que `rotulo_origem` e `rotulo_destino` identificam unicamente a aresta na aplicação.

Dependência A estrutura TipoDepend apresenta os seguintes atributos:

- rotulo_origem: indica o vértice origem da dependência.
- rotulo_destino: indica o vértice destino da dependência.
- texto: é o texto que será mostrado no desenho, sobre a dependência.
- cor_depend: é a cor usada no desenho da dependência.
- cor_texto: é a cor usada no texto.
- Dependencia: indica o tipo de dependência existente entre rotulo_origem e rotulo_destino.

Observe que rotulo_origem, rotulo_destino e dependencia identificam unicamente a dependência na aplicação.

Interação Usuário → Aplicação

A aplicação é notificada das ações do usuário sobre o desenho através de três funções de retorno, uma para cada widget de Hierarquia, vértice, aresta e dependência. Cada uma tem um tipo de evento associado, EventVertice, EventAresta e EventDependencia respectivamente, que retornam os atributos que unicamente definem o elemento na aplicação, permitindo sua identificação.

Interação Aplicação → Usuário

A seguir, relaciona-se algumas funções implementadas para efetuar este sentido da interação. A sintaxe de cada função é apresentada no apêndice A.

- IUhCriaHier(): Cria uma hierarquia na janela especificada. Além da janela, devem também ser especificados os vértices, as arestas e as dependências.
- IUhAtualizaHier(): Atualiza os atributos de vértices, arestas e desenho do mapa especificado.
- IUhEliminaHier(): Elimita o widget criado por IUhCriaHier.
- IUhMostraDep(): Mostra dependências no desenho. As dependências especificadas em IUhCriaMapa() só serão mostradas ao usuário através deste comando. Permite mostrar todas as dependências de um determinado tipo, ou indicar, uma a uma, aquelas a serem mostradas.

- IUhApagaDep(): Apaga as dependências mostradas por IUhMostraDep().
- IUhVertice(): Inclui função de retorno para os vértices de um mapa.
- IUhAresta(): Inclui função de retorno para as arestas de um mapa.
- IUhDepend(): Inclui função de retorno para as dependências entre os vertices de um mapa.

Exemplo

A seguir apresenta-se um programa ilustrando os dois sentidos da interação

```
f_vertice(w, client_data, call_data)
        Widget        w;
        char*         client_data;
        EventVertice* call_data;
{
        printf ("Vertice %d selecionado\n", call_data->rotulo);
}

f_aresta(w, client_data, call_data)
        Widget        w;
        char*         client_data;
        EventAresta*  call_data;
{
        printf ("Aresta (%d, %d) selecionada\n",
                 call_data->rotulo_origem, call_data->rotulo_destino);
}

f_dependencia(w, client_data, call_data)
        Widget        w;
        char*         client_data;
        EventDepend*  call_data;
{
        switch (call_data->dependencia){
                case SAIDA    : printf ("Dependencia de saida ");
                                break;
```

```
            case ANTI     : printf ("Anti-Dependencia ");
                            break;
            case FLUXO    : printf ("Dependencia de fluxo ");
                            break;
            case CONTROLE : printf ("Dependencia de Controle ");
                            break;
        }
        printf ("entre os vertices (%d, %d) selecionada\n",
                call_data->rotulo_origem, call_data->rotulo_destino);
}

      . . .

void    main (argc, argv)
        int          argc;
        char*        argv[];
{
        Janela*       j;              /* Janela da interface              */
        TipoVertice** vs;             /* Vetor de apontadores para vertice*/
        TipoAresta**  as;             /* Vetor de apontadores para arestas*/
        TipoDepend**  ds;             /* Vetor de apontadores para depend.*/
        int           totV,           /* Total de vertices em vs          */
                      totA,           /* Total de arestas  em as          */
                      totD;           /* Total de dependencias em ds      */

        j=IUabreJanela(argc, argv); /* Abre uma janela                    */

        vs=MontaVertices(&totV);    /* Rotinas que transformam vertices */
        as=MontaArestas(&totA);     /* arestas e depend. do grafo de    */
        ds=MontaDepend(&totD);      /* controle para a hierarquia       */

        if (IUhCriaHier(j, vs, totV, as, totA, ds, totD) != 0){
            printf("Erro na criacao de hierarquia\n");
            return;
        }
```

```
/* Inclui Funcoes de retorno para tratar eventos sobre vertices,       */
/*   arestas e dependencias                                            */

        if (IUhVertice (j->hier, f_vertice, NULL) != 0){
                printf("Erro na inclusao da f. ret. de vertices\n");
                return;
        }
        if (IUhAresta (j->hier, f_aresta, NULL) != 0){
                printf("Erro na inclusao da f. ret. de arestas\n");
                return;
        }
        if (IUhDepend (j->hier, f_dependencia, NULL) != 0){
                printf("Erro na inclusao da f. ret. de dependencias\n");
                return;
        }
        XtMainloop();               /* Loop de espera de eventos do X    */
        return;
}
```

Onde `f_vertice`, `f_aresta` e `f_dependencia` são funções de retorno para as ações do usuário sobre estes widgets, incluídos através das funções `IUhVertice()`, `IUhAresta()` e `IUhDepend()`. Observe que parâmetro `client_data` nos três casos é `NULL`.

Capítulo 4

O Algoritmo para Desenhar Hierarquias

Grafos dirigidos são freqüentemente usados para representar relações entre objetos, onde os vértices representam os objetos e as arestas representam as relações. Por exemplo, modelos entidade-relacionamento, redes Petri, árvores sintáticas, diagramas de fluxo de dados, podem ser representados desta forma.

Porém, desenhar manualmente tais grafos de maneira a facilitar a leitura é uma tarefa demorada que não garante resultados satisfatórios.

Warfield [Warf 77] observou que esta tarefa poderia ser efetuada com o uso de computadores para uma classe especial de grafos dirigidos, onde os vértices estão dispostos em camadas, conhecida como *hierarquia*.

Posteriormente, Sugiyama [Sugi 81] propôs critérios estéticos para desenhar hierarquias que conduzem a desenhos legíveis. O trabalho também apresenta um algoritmo de quatro passos que implementa tais critérios.

O algoritmo de Sugiyama tem sido utilizado em diversos projetos de pesquisa. Destacam-se [Rowe 87, TiNe 87, Robi 88] que visam editores de grafos para aplicações genéricas e [JaGu 88, JaGu 89] que propõem editores/gerenciadores de hierarquias para ambientes de reestruturação de programas.

Ambientes de reestruturação utilizam editores de hierarquias para diversas tarefas. Por exemplo, para o exame de abstrações de programas, a visualização de dependências entre comandos e o acompanhamento da execução de programas. Estas são tarefas centrais do PCC, o que torna imperativa a implementação de um editor de hierarquias no PCC.

Este capítulo relata o algoritmo para desenhar hierarquias e as adaptações efetuadas na sua implementação para o PCC. O algoritmo, baseado no trabalho de Sugiyama, é

o componente central do editor de hierarquias. A seção 4.1 apresenta a nomenclatura utilizada, as características gráficas que facilitam a leitura de hierarquias e um resumo dos passos do algoritmo. A seção 4.2 apresenta a base matemática do algoritmo, e a seção 4.3 detalha cada um dos quatro passos do algoritmo.

4.1 Preliminares

O desenho da hierarquia será denominado *mapa*. A figura 4.1 apresenta um mapa com a nomenclatura utilizada. Cada *camada* é composta por um conjunto de vértices. Camadas são enumeradas de cima para baixo, a partir da camada 1.

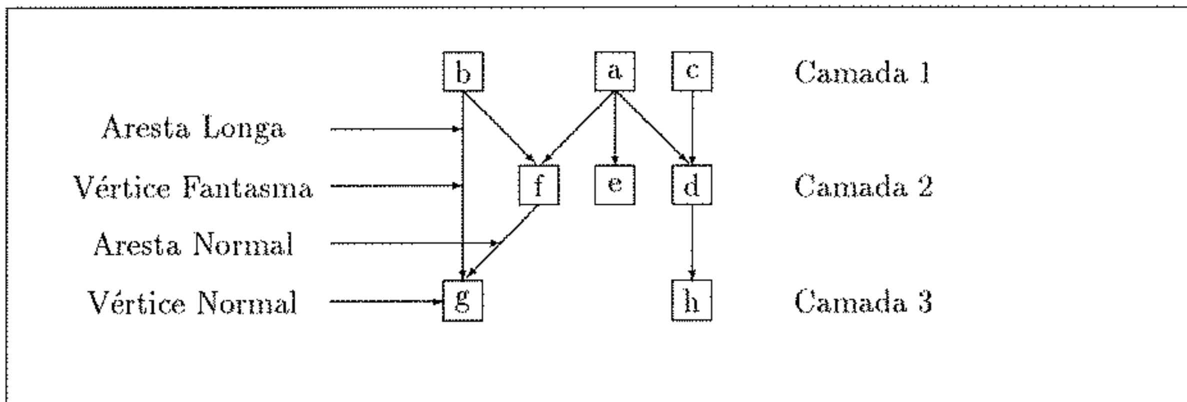


Figura 4.1: Elementos de uma hierarquia

As arestas que cruzam uma ou mais camadas são chamadas de *arestas longas*. Um *vértice fantasma* é acrescentado cada vez que uma aresta longa cruza uma camada.

4.1.1 Características de um Mapa Legível

Quando se apresenta um desenho, pretende-se que o observador compreenda rapidamente o que ele representa. Sugiyama [Sugi 81] observou que em desenhos de grafos, as características a seguir facilitam a visualização:

1. os vértices devem estar dispostos de maneira regular; e

2. as arestas que ligam dois vértices quaisquer devem ser facilmente seguidas pelo observador.

Sugiyama aplicou este critério para hierarquias e definiu cinco características que tornam um mapa legível:

A- Os vértices devem estar dispostos em camadas (hierarquizados).

B- O número de cruzamentos entre as arestas deve ser mínimo.

C- Deve-se evitar ziguezagues nas arestas longas.

D- As arestas devem ser curtas, o que reduz a distância entre os vértices ligados.

E- As arestas que chegam e saem de um vértice devem ser desenhadas de maneira balanceada. Assim, as ramificações e convergências dos caminhos podem ser observadas mais facilmente.

A característica **A** dispõe os vértices de maneira regular, enquanto as demais características facilitam a visualização das arestas.

4.1.2 Resumo dos Passos

As características descritas na seção 4.1.1 norteiam o algoritmo de quatros passos descrito abaixo. As características que são satisfeitas em cada passo estão indicadas entre parênteses para facilitar sua identificação.

Passo 1: *(A)* - A partir de um conjunto de vértices e arestas, monta-se uma hierarquia acrescida dos vértices fantasmas e arestas correspondentes.

Passo 2: *(B)* - Permutando-se a posição dos vértices de cada camada, reduz-se o número de cruzamentos entre as arestas.

Passo 3: *(C, D, E)* - Determina-se a posição horizontal dos vértices da hierarquia.

Passo 4: Desenha-se o grafo no dispositivo de saída.

A figura 4.2 exemplifica as transformações que cada passo do algoritmo efetua em uma hierarquia. Observe as características satisfeitas por cada passo. Passos 1 e 2 são auto-explicativos. Após o passo 3, a única aresta longa está reta (Característica C), as arestas são mais curtas do que após o passo 2 (característica D) e observa-se mais facilmente as ramificações e convergências das arestas (característica E).

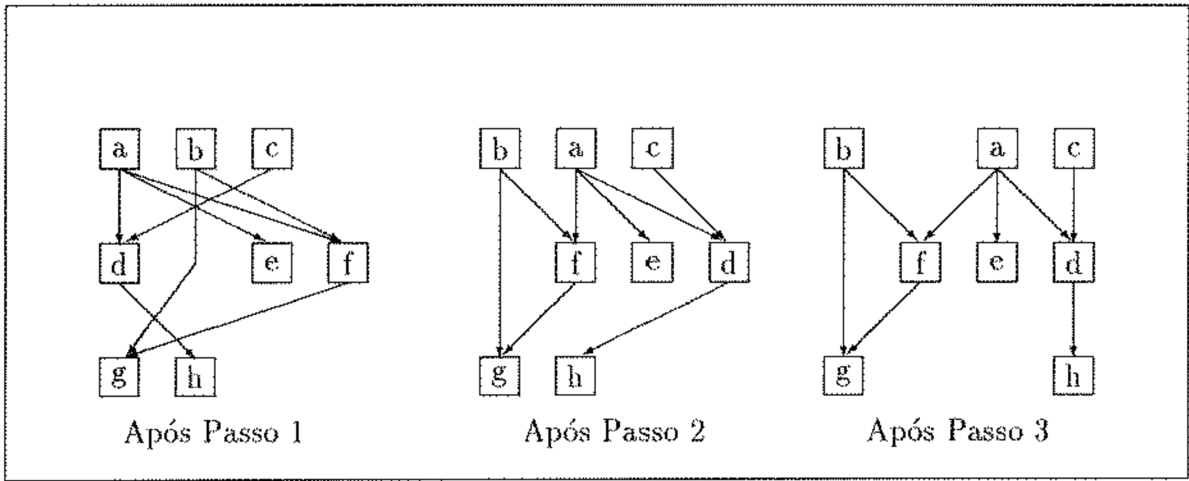


Figura 4.2: Resultado da Aplicação do Algoritmo em uma Hierarquia

4.2 Definições

Esta seção apresenta a base matemática utilizada ao longo do texto, conforme especificado em [Warf 77, Sugi 81].

Apresenta-se, inicialmente, a definição formal de hierarquias. Em seguida, explica-se matrizes de realização, como contar o número de cruzamentos entre as arestas, conectividade e por último os baricentros, chave para os passos 2 e 3 do algoritmo.

4.2.1 Hierarquia

Segundo [Sugi 81], define-se uma hierarquia de n-camadas ($n > 2$) sobre um grafo dirigido (V, A), onde V é o conjunto de vértices e A é o conjunto de arestas da forma abaixo:

1. V é particionado em n sub-conjuntos tais que

   (a) $V = V_1 \cup V_2 \cup \cdots \cup V_n$

   (b) $V_i \cap V_j = \emptyset, \forall i \neq j$

   (c) $\forall a, b \in V$, há no máximo uma aresta $(a, b) \in A$

   (d) $Se\ (a, b) \in A,\ com\ a \in V_i\ e\ b \in V_j, então\ i < j$

2. A é particionado em $n - 1$ sub-conjuntos tais que

   (a) $A = A_1 \cup A_2 \cup \cdots \cup A_{n-1}$

   (b) $A_i \cap A_j = \emptyset, \forall i \neq j$

(c) $A_i \subset V_i \times V_{i+1}, i = 1, 2, \ldots, n-1$.

3. Há uma ordem σ_i para cada V_i, onde "ordem" significa enumerar os vértices de V_i. Logo, $\sigma_i = v_1, v_2, \ldots, v_{|V_i|}$, onde $|V_i|$ é o número de vértices contidos em V_i.

A hierarquia de n-camadas é denominada $G = (V, A, n, \sigma)$, onde $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$.

Há grafos que não podem ser hierarquizados imediatamente, por exemplo, grafos que contém arestas paralelas e/ou ciclos. Arestas paralelas violam a condição ($1d$), enquanto ciclos violam a condição ($1c$), uma vez que pelo menos uma de suas arestas terá $i \geq j$.

Nestes casos, a hierarquização só é possível através de artifícios. No caso de arestas paralelas, deve-se utilizar somente uma, eliminando as demais. No caso de ciclos, deve-se eliminá-los, invertendo o sentido de uma das arestas, como será visto na seção 4.3.1.

4.2.2 Matriz de Realização

Matrizes de realização são similares a matrizes de adjacências, porém ao invés de representarem todo o grafo, representam duas camadas adjacentes da hierarquia. Enquanto as linhas da matriz representam os vértices de uma camada, as colunas representam os vértices da camada seguinte. Indica-se a existência de uma aresta ligando dois vértices destas camadas com um '1' nas coordenadas correspondentes da matriz e com '0' caso contrário.

A definição formal é a seguinte:

1. $M^{(i)} = M(\sigma_i, \sigma_{i+1})$ é uma matriz $|V_i| \times |V_{i+1}|$, onde as posições dos vértices nas linhas e colunas são determinadas por σ_i e σ_{i+1} respectivamente.

2. Seja $\sigma_i = v_1, \ldots, v_k, \ldots, v_{|V_i|}$, e $\sigma_{i+1} = w_1, \ldots, w_l, \ldots, w_{|V_{i+1}|}$. Então o elemento (v_k, w_l) de $M^{(i)}$ é dado por:

$$m^{(i)}(k,l) \begin{cases} = 1 & \text{se } (v_k, w_l) \in A_i \\ = 0 & \text{caso contrário} \end{cases}$$

A Figura 4.3 mostra o mapa de uma hierarquia, com as seguintes matrizes de realização:

$M^{(1)} =$

	c	d	e	f
a	1	1	1	1
b	1	0	0	1

$M^{(2)} =$

	g	h	i	j
c	1	0	0	0
d	0	1	1	1
e	1	0	0	1
f	0	0	0	0

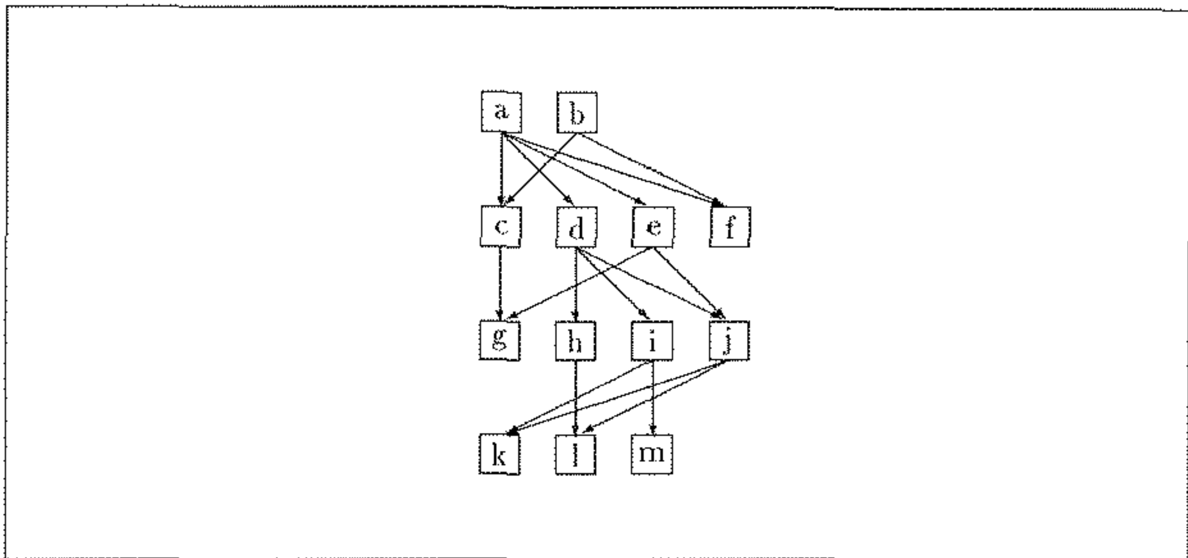


Figura 4.3: Exemplo de Hierarquia.

$M^{(3)} =$

	k	l	m
g	0	0	0
h	0	1	0
i	1	0	1
j	1	1	0

4.2.3 Número de Cruzamentos

Seja $\sigma_i = v_1, \ldots, v_j, \ldots, v_l, \ldots, v_{|V_i|}$ e $\sigma_{i+1} = w_1, \ldots, w_\alpha, \ldots, w_\beta, \ldots, w_{|V_i|}$.

Ocorre um cruzamento entre as arestas (v_j, w_β), com $j < l$, se e somente se $\alpha < \beta$ (vide figura 4.4).

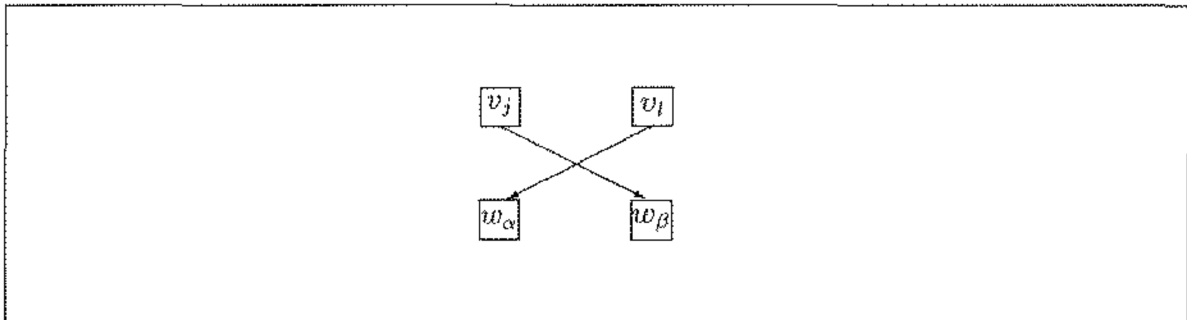


Figura 4.4: Cruzamento Entre Duas Arestas

Para calcular os cruzamentos que ocorrem entre uma aresta (v_j, w_β) e todas as arestas com origem em um vértice v_l, com $j < l$, basta contar as arestas com origem em v_l que verificam a condição acima. Caso $j > l$, inverte-se a condição.

Para calcular os cruzamentos que ocorrem entre todas as arestas com origem em v_j e v_l, basta verificar a condição para todas as arestas com origem em v_j e v_l. Isto induz à forma de calcular o número de cruzamentos entre dois vértices da linha de uma matriz apresentada abaixo:

$$k(v_j, v_l) = \sum_{\alpha=1}^{|V_{i+1}|-1} \sum_{\beta=\alpha+1}^{|V_{i+1}|} m^{(i)}(j,\beta) \cdot m^{(i)}(l,\alpha)$$

sendo $v_j, v_l \in V_i$ e $j < l$ em σ_i.

Para exemplificar este procedimento, aplica-se esta fórmula para calcular o número de cruzamentos entre arestas emergentes dos vértices 'a' e 'b' do grafo retratado em 4.2.2 e representado pela matriz $M^{(1)}$.

$$\begin{aligned} k(a,b) = \;& m(1,2) \cdot m(2,1) + m(1,3) \cdot m(2,1) + m(1,4) \cdot m(2,1) + \\ & m(1,3) \cdot m(2,2) + m(1,4) \cdot m(2,2) + \\ & m(1,4) \cdot m(2,3) \end{aligned}$$

$$\begin{aligned} k(a,b) = \;& 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + \\ & 1 \cdot 0 + 1 \cdot 0 + \\ & 1 \cdot 0 \end{aligned}$$

$$k(a,b) = 3 \text{ cruzamentos}$$

Para calcular os cruzamentos entre todas as arestas de duas camadas consecutivas, aplica-se o cálculo anterior para todos os pares de vértices v_j, v_l, tal que $j < l$, ou seja:

$$K(M^{(i)}) = \sum_{j=1}^{|V_i|-1} \sum_{l=j+1}^{|V_i|} (k(v_j, v_l))$$

O exemplo a seguir apresenta o cálculo do número de cruzamentos que ocorrem entre as camadas 1 e 2 da figura 4.2.2.

$$\begin{aligned} & k(c,d) = 0 + k(c,e) = 0 + k(c,f) = 0 + \\ & k(d,e) = 3 + k(d,f) = 0 + \\ & k(e,f) = 0 \end{aligned}$$

Total: 3 cruzamentos.

Para a hierarquia, soma-se o número de cruzamentos entre todas as camadas, ou seja:

$$K = K(M^{(1)}) + K(M^{(2)}) + \cdots + K(M^{(n-1)})$$

4.2.4 Conectividade

Denomina-se Conectividade de Cima de um vértice v_k na camada i, representado por $C_{i,k}^{Cima}$, ao número de arestas incidentes a ele. De maneira análoga, denomina-se Conectividade de Baixo deste vértice, representado por $C_{i,k}^{Baixo}$, ao número de arestas emergentes dele.

A Conectividade de Cima e de Baixo de um vértice são calculadas por:

$$C_{i,k}^{Cima} = \sum_{j=1}^{|V_{i-1}|} m^{(i-1)}(j,k)$$

$$C_{i,k}^{Baixo} = \sum_{l=1}^{|V_{i+1}|} m^{(i)}(k,l)$$

4.2.5 Baricentro

As formulações desta seção são a parte central da heurística de Sugiyama. Elas serão aplicadas nas seções 4.3.2 e 4.3.3 para redução de cruzamentos e alinhamento horizontal, respectivamente.

O baricentro de linha de um vértice v_k em uma camada i, representado por $B_{i,k}^{Linha}$, indica o "vértice médio" dos vértices destino de suas arestas emergentes. Por exemplo, atribuindo-se as posições $\{1, 2, 3, 4\}$ aos vértices 'g', 'h', 'i', 'j' da camada 3 da figura 4.3, percebe-se que as arestas emergentes de 'd' atingem as posições 2, 3 e 4. O "vértice médio" destino das arestas de 'd' será 3, que também será o valor do baricentro de linha.

Como a ordem das colunas de $M^{(i)}$ obedece à enumeração σ_{i+1}, pode-se calcular o baricentro de linha de um vértice v_k na camada i da seguinte forma:

$$B_{i,k}^{Linha} = \begin{cases} \frac{\sum_{l=1}^{|V_{i+1}|} l \cdot m^{(i)}(k,l)}{\sum_{l=1}^{|V_{i+1}|} m^{(i)}(k,l)}, & \text{Se} \sum_{l=1}^{|V_{i+1}|} m^{(i)}(k,l) \neq 0 \\ 0 & \text{Caso Contrário} \end{cases}$$

O baricentro de coluna de um vértice v_k em uma camada i, representado por $B_{i,k}^{Coluna}$, indica o "vértice médio" dos vértices origem de suas arestas incidentes. Por exemplo, atribuindo-se as posições $\{1, 2, 3, 4\}$ aos vértices 'g', 'h', 'i', 'j' da camada 3 da figura 4.3, percebe-se que as arestas incidentes ao vértice 'l' (ele) da camada 4, tem origem nas posições 2 e 4. O "vértice médio" incidente de 'l' será 3, que também será o valor do baricentro da coluna.

Como a ordem das linhas de $M^{(i-1)}$ obedece à enumeração σ_{i-1}, pode-se calcular o baricentro de coluna de um vértice v_k na camada i da seguinte forma:

$$B_{i,k}^{Coluna} \begin{cases} = \frac{\sum_{l=1}^{|V_{i-1}|} l \cdot m^{(i-1)}(l,k)}{\sum_{l=1}^{|V_{i-1}|} m^{(i-1)}(l,k)}, & \text{Se } \sum_{l=1}^{|V_{i-1}|} m^{(i-1)}(l,k) \neq 0 \\ = 0 & \text{Caso Contrário} \end{cases}$$

Estes dois baricentros são usados no passo 2 do algoritmo para reduzir o número de cruzamentos através de uma heurística. Eles determinarão a ordem de apresentação dos vértices de uma camada da hierarquia, mas não a distância horizontal entre dois vértices consecutivos. Para representar a distância, é necessário que os baricentros indiquem a "posição do vértice médio" e não o "vértice médio".

O passo 3 do algoritmo utiliza este fato para alinhar os vértices horizontalmente. Assim, alteram-se as fórmulas anteriores para que as posições atribuidas sejam as posições horizontais de cada vértice, criando-se dois novos baricentros, denominados baricentro de baixo e baricentro de cima.

O baricentro de baixo de um vértice v_k em uma camada i, representado por $B_{i,k}^{Baixo}$, é similar a $B_{i,k}^{Linha}$. Porém, ao invés de utilizar os índices das colunas de $M^{(i)}$, utiliza a posição horizontal de cada vértice da camada seguinte, representado por $x(v_l^{i+1})$, onde l é o índice do vértice na camada $i+1$.

Assim, o baricentro de baixo de um vértice v_k na camada i é obtido da seguinte forma:

$$B_{i,k}^{Baixo} \begin{cases} = \frac{\sum_{l=1}^{|V_{i+1}|} x(v_l^{i+1}) \cdot m^{(i)}(k,l)}{\sum_{l=1}^{|V_{i+1}|} m^{(i)}(k,l)}, & \text{Se} \sum_{l=1}^{|V_{i+1}|} m^{(i)}(k,l) \neq 0 \\ = 0 & \text{Caso Contrário} \end{cases}$$

Define-se baricentro de cima de forma análoga.

$$B_{i,k}^{Cima} \begin{cases} = \frac{\sum_{l=1}^{|V_{i-1}|} x(v_l^{i-1}) \cdot m^{(i-1)}(l,k)}{\sum_{l=1}^{|V_{i-1}|} m^{(i-1)}(l,k)} & \text{Se} \sum_{l=1}^{|V_{i-1}|} m^{(i-1)}(l,k) \neq 0 \\ = 0 & \text{Caso Contrário} \end{cases}$$

4.3 Detalhamento dos Passos

Nas próximas quatro seções (4.3.1, 4.3.2, 4.3.3 e 4.3.4) detalha-se cada um dos quatro passos do algoritmo.

Cada seção explica o funcionamento do passo correspondente do algoritmo, os melhoramentos encontrados na literatura e as adaptações ou melhoramentos efetuados para o ambiente do PCC.

4.3.1 Hierarquização de Vértices (Passo 1)

Este passo particiona os conjuntos de vértices e arestas nas camadas da hierarquia.

O algoritmo proposto por Sugiyama para o Passo 1 baseia-se no princípio de que um vértice deve ser colocado acima daqueles vértices que são os destinos das suas arestas. Por exemplo, se o vértice v_1 tem três arestas emergentes, que o ligam respectivamente a v_4, v_3 e v_7, então v_1 deve ser posicionado em uma das camadas acima de v_4, v_3 e v_7.

O algoritmo original é:

1. Seleciona-se, do grupo de vértices, aqueles que não tem arestas incidentes, colocando-os em uma camada da hierarquia.

2. Elimina-se, do conjunto de vértices, aqueles selecionados no item anterior e elimina-se, do conjunto de arestas, aquelas com origem nos vértices elimiados.

Executa-se este procedimento recursivamente, compondo as camadas da hierarquia de cima para baixo, até que os conjuntos de vértices e arestas estejam vazios.

Melhoramentos

Este algoritmo não se aplica a grafos com ciclos. Assim, Sugiyama propõe que os vértices que fazem parte de um ciclo sejam condensados em um único vértice. Este vértice condensado é visualmente destacado dos demais no desenho do mapa no dispositivo de saída (pelo passo 4), enquanto que as arestas do ciclo não são mostradas no desenho.

Rowe [Rowe 87] propôs uma adaptação ao algoritmo original baseado em um artifício. Se o conjunto de vértices e arestas não estiver vazio e não for selecionado nenhum vértice (ou seja, existe ciclo), procura-se o ciclo através de uma busca em profundidade a partir do conjunto de vértices. Ao encontrar a aresta que fecha o ciclo, inverte-se sua direção, eliminando o ciclo. No Passo 4, reinverte-se a direção da aresta, retornando ao grafo original.

Adaptações ao PCC

Rowe não impôs nenhuma ordem para inversão de arestas. Assim, no grafo com vértices (A, B, C, D) e arestas (AB, BC, CD, DB), a solução de Rowe pode inverter as arestas BC, CD ou DB, eliminando o ciclo. Porém, dependendo da aresta invertida são gerados mapas diferentes, como pode ser observado na figura 4.5

Esse grau de liberdade é prejudicial no contexto de grafos de programa, onde o mapa (c) é mais adequado por ser coerente com a seqüência de execução do programa. Assim,

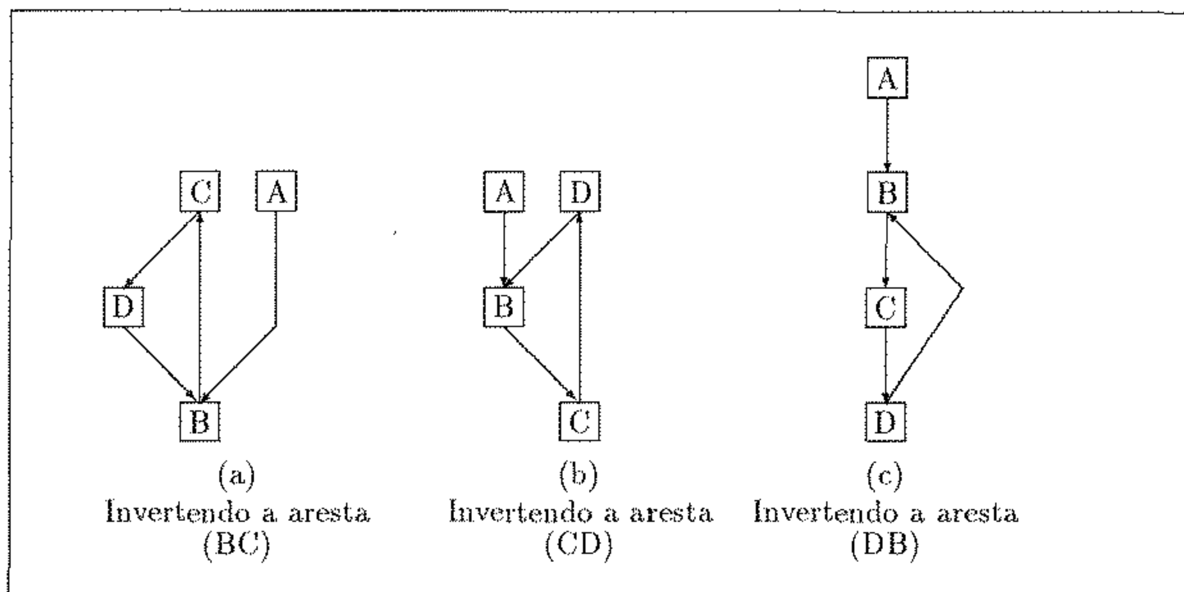


Figura 4.5: Mapas Possíveis a Partir do Mesmo Grafo

adaptou-se o algorimo para que a busca em profundidade seja efetuada somente a partir de um sub-conjunto dos vértices ainda não selecionados. Este sub-conjunto é composto pelos vértices destino das arestas eliminadas. Assim os conjuntos de vértices utilizados são: (1) aqueles que já estão hierarquizados, (2) aqueles que não foram hierarquizados e (3) aqueles do conjunto (2) cujas arestas tem origem em (1). Este último é um sub-conjunto de (2).

No exemplo, após selecionar o vértice A para compor a primeira camada da hierarquia (conjunto 1), os vértices BCD farão parte do segundo conjunto e B fará parte do terceiro. Ao ser detectado o ciclo, inicia-se uma busca em profundidade a partir de B, invertendo as arestas que fecham o ciclo em B. No exemplo isto acontece com DB, obtendo-se o mapa (c) da figura 4.5.

No algoritmo abaixo, H(c) corresponde aos vértices que estão na camada c da hierarquia, *Apont* é o terceiro conjunto de vértices e V e A são os conjuntos de vértices e arestas de entrada.

UNICAMP
BIBLIOTECA CENTRAL

```
Apont ← ∅
c ← 1
Enquanto (V ≠ ∅ e A ≠ ∅)
    H(c) ← Vértices em V que não tem arestas incidentes
    Se H(c) = ∅
        Se Apont = ∅
            Elimina ciclo a partir dos vértices de V
        Senão
            Elimina ciclo a partir dos vértices de Apont
        Fimse
    Senão
        Apont ← Apont − H(c)
        Apont ← Apont ∪ {Vértices destino das arestas de H(c)}
        V ← V − H(c)
        A ← A − {arestas com origem em H(c)}
        c ← c + 1
    Fimse
Fimenquanto
```

Observe que existem casos em que não é possível determinar em qual vértice será iniciada a busca em profundidade. Para tanto basta existir mais de um vértice em *Apont* ou que, quando *Apont* estiver vazio, V contenha mais de um vértice.

Para auxiliar nestes casos, permitiu-se atribuir pesos aos vértices, possibilitando priorizar a escolha do vértice a partir do qual será iniciada a busca em profundidade.

4.3.2 O Algoritmo de Redução de Cruzamentos (Passo 2)

Este passo recebe como entrada a hierarquia gerada no passo 1 e altera a ordem dos vértices de cada camada, de tal maneira a reduzir o número de cruzamentos entre as arestas. Os demais componentes da hierarquia não são alterados.

Este passo é o mais complicado dos quatro e será dispensada maior atenção a ele. Inicialmente, conceitua-se matematicamente o problema. Em seguida, apresenta-se um algoritmo para reduzir o número de cruzamentos entre duas camadas, e por último estende-se este algoritmo para pares de camadas consecutivos da hierarquia.

Seja S_i o conjunto de todas as ordens σ_i de uma camada i em uma hierarquia de n-camadas, e seja $S = S_1 \times S_2 \times \cdots \times S_n$. Assim, o problema de minimizar o número de cruzamentos entre as arestas da hierarquia pode ser visto como:

$$minimize\{K(\sigma) \mid \sigma \in S\}$$

Como este problema é combinatório por natureza, torna-se muito caro obter a solução ótima para um número grande de vértices.

Sugiyama [Sugi 81] apresenta duas soluções para este problema. A primeira, *Penalty Minimization (PM) method* é uma solução com abordagem teórica que utiliza muito tempo computacional. A segunda, *Barycentric Method (BC)*, é uma solução baseada na heurística criada por Warfield [Warf 77], que mostrou-se mais econômica, razão pela qual foi adotada na implementação.

A idéia do método BC consiste em permutar a posição dos vértices da ordem $\sigma_2 = v_1, v_2, \ldots, v_{|V_2|}$ em função da ordem σ_1 em $\mathrm{M}(\sigma_1, \sigma_2)$ obtendo uma nova ordem $\sigma_2' = v_1', v_2', \ldots, v_{|V_2|}'$, que é uma permutação de σ_2. Para obter esta nova ordem, calcula-se os valores de B^{Coluna} para todos os vértices de σ_2, ordenando-os em seguida de acordo os seus baricentros, de tal maneira que $B^{Coluna}_{2,v_1'} \leq B^{Coluna}_{2,v_2'} \leq \ldots \leq B^{Coluna}_{2,v_{|V_2|}'}$.

Como já foi mencionado, o baricentro de coluna de um vértice indica o "vértice médio" em função da enumeração dos vértices origem de suas arestas. Ao ordenar os baricentros do menor para o maior, é intuitivo pensar que haverá redução no número de cruzamentos, o que é comprovado na prática.

Esta transformação de $\mathrm{M}(\sigma_1, \sigma_2)$ para $\mathrm{M}(\sigma_1, \sigma_2')$ é chamada *Ordenação de Colunas*, β_C, ou seja: $\mathrm{M}(\sigma_1, \sigma_2') = \beta_C(\mathrm{M}(\sigma_1, \sigma_2))$.

Analogamente, é possível permutar a posição dos vértices da ordem $\sigma_1 = v_1, v_2, \ldots, v_{|V_1|}$ em função da ordem σ_2 em $\mathrm{M}(\sigma_1, \sigma_2)$ obtendo-se uma nova ordem $\sigma_1' = v_1', v_2', \ldots, v_{|V_1|}'$, que é uma permutação de σ_1. Para obter esta nova ordem, calcula-se os valores de B^{Linha} para todos os vértices de σ_1, ordenando-os em seguida de acordo os seus baricentros, de tal maneira que $B^{Linha}_{1,v_1'} \leq B^{Linha}_{1,v_2'} \leq \ldots \leq B^{Linha}_{1,v_{|V_1|}'}$.

A transformação de $M(\sigma_1, \sigma_2)$ para $\mathrm{M}(\sigma_1', \sigma_2)$ é chamada *Ordenação de Linhas*, β_L, ou seja: $\mathrm{M}(\sigma_1', \sigma_2) = \beta_L(\mathrm{M}(\sigma_1, \sigma_2))$.

Observe que ao efetuar estas duas operações alternadamente sobre uma matriz, é possível reduzir sucessivamente o número de cruzamentos.

O algoritmo da seção 4.3.2 executa este processo em hierarquias de duas camadas, enquanto que seção 4.3.2 apresenta um algoritmo que estende a solução para hierarquias de n-camadas, com $n > 2$.

Hierarquias de Duas Camadas

Para reduzir os cruzamentos em hierarquias de duas camadas, executa-se sobre a matriz M as operações β_C e β_L alternadamente. O processo termina ao atingir um número determinado de iterações.

Na função **Reduz-2**, M_0 é a matriz de entrada, M_* é a matriz com o menor número de cruzamentos encontrada até o momento e K_* é o número de cruzamentos que ocorrem nesta matriz.

```
Reduz − 2(M₀)
    M* ← M₀
    K* ← K(M₀)
    Repetir
        M₁ ← β_C(M₀)
        Se  (K(M₁) < K*)
            M* ← M₁
            K* ← K(M₁)
        Fimse
        M₂ ← β_L(M₁)
        Se  (K(M₂) < K*)
            M* ← M₂
            K* ← K(M₂)
        Fimse
        Se  (M₀ = M₂)
            Troca − Baricentros − Iguais()
        Senão
            M₀ ← M₂
        Fimse
    Até (Número determinado de iterações)
    Retorna(M*)
```

Observe que é possível que dois vértices na linha, ou dois vértices na coluna da matriz, tenham baricentros iguais, porém suas arestas os liguem a vértices diferentes.

Devido à natureza heurística do algoritmo, trocar a ordem destes dois vértices pode provocar mais reduções. Por isso, inclui-se duas funções R_C e R_L, que trocam vértices com baricentros iguais, de tal forma que, caso não ocorram mudanças na ordem dos vértices, a função **Reduz-2**, chama a função **Troca-Baricentros-Iguais**, que aplica as funções R_C e R_L na matriz.

Troca – Baricentros – Iguais()

$M_3 \leftarrow R_C(M_2)$
$Se \quad (M_2 \neq M_3)$
$\quad M_0 \leftarrow M_3$
$Senão$
$\quad M_4 \leftarrow R_L(M_3)$
$\quad Se \quad (M_3 \neq M_4)$
$\quad\quad M_0 \leftarrow M_4$
$\quad Fimse$
$Fimse$

Este algoritmo não garante que se obtenha a ordem de vértices que causa o menor número de cruzamentos. Porém, como foi verificado na prática, ele reduz sensivelmente os cruzamentos, tornando a hierarquia mais legível.

Para exemplificar o funcionamento do algoritmo, será utilizada a hierarquia da figura 4.6.

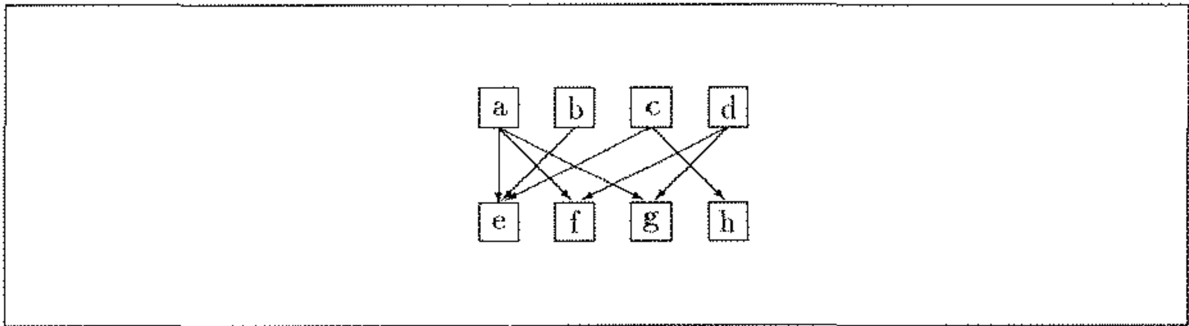


Figura 4.6: Hierarquia de Duas Camadas Antes da Redução.

Neste exemplo, as operações são efetuadas sobre as variáveis M_0, M_1, M_2 apresentadas no algoritmo. Não serão apresentadas as operações sobre a variável M_*.

Suponha que devam ser executadas cinco iterações.

$M_0 =$

	e	f	g	h	B^{Linha}
a	1	1	1	0	2.0
b	1	0	0	0	1.0
c	1	0	0	1	2.5
d	0	1	1	0	2.5
B^{Coluna}	2.0	2.5	2.5	3.0	

$K = 7$

O primeiro passo do algoritmo ordena as colunas de M_0 por B^{Coluna}, ou seja, $M_1 \leftarrow \beta_C(M_0)$, mas elas já estão ordenados. O passo seguinte ordena as linhas de M_1 por B^{Linha}, ou seja, $M_2 \leftarrow \beta_L(M_1)$, quando a ordem (a, b, c, d) é alterada para (b, a, c, d).

$M_2 =$

	e	f	g	h	B^{Linha}
b	1	0	0	0	1.0
a	1	1	1	0	2.0
c	1	0	0	1	2.5
d	0	1	1	0	2.5
B^{Coluna}	2.0	3.0	3.0	3.0	

$K = 5$

Neste ponto $M_0 \neq M_2$, então faz-se $M_0 \leftarrow M_2$, terminando a primeira iteração do algoritmo.

Inicia-se a segunda iteração com $M_1 \leftarrow \beta_C(M_0)$, porém as colunas já estão ordenadas. O passo seguinte é $M_2 = \beta_L(M_1))$, mas as linhas também estão ordenadas.

Neste ponto, $M_0 = M_2$. Assim, executa-se o procedimento Troca-Baricentros-Iguais, onde $M_3 \leftarrow R_C(M_2)$, o que altera a ordem (e, f, g, h) para (e, h, f, g). Como $M_2 \neq M_3$, faz-se $M_0 \leftarrow M_3$, retornando em seguida para a função Reduz-2, quando termina a segunda iteração.

$M_0 =$

	e	h	f	g	B^{Linha}
b	1	0	0	0	1.0
a	1	0	1	1	2.6
c	1	1	0	0	1.5
d	0	0	1	1	3.5
B^{Coluna}	2.0	3.0	3.0	3.0	

$K = 5$

Inicia-se a terceira iteração fazendo $M_1 \leftarrow \beta_C(M_0)$, porém, as colunas já estão ordenadas. Assim, faz-se $M_2 \leftarrow \beta_L(M_1)$, quando obtém-se a nova ordem (b, c, a, d).

$M_2 =$

	e	h	f	g	B^{Linha}
b	1	0	0	0	1.0
c	1	1	0	0	1.5
a	1	0	1	1	2.6
d	0	0	1	1	3.5
B^{Coluna}	2.0	2.0	3.5	3.5	

$K = 2$

Neste momento termina a terceira iteração. Observe que os baricentros das linhas e colunas estão ordenados. Assim, as operações $M_1 \leftarrow \beta_C(M_0)$ e $M_2 \leftarrow \beta_L(M_1)$ não alteram a ordem dos vértices, de tal sorte que ao final de quarta iteração $M_0 = M_2$, quando será executada a função Troca-Baricentros-Iguais(). Novamente a operação $M_3 \leftarrow R_C(M_2)$ altera a ordem das colunas da matriz, atribui a nova matriz para M_0 e devolve o controle à função Reduz-2(), onde inicia-se a quinta e última iteração.

$M_0 =$

	h	e	f	g	B^{Linha}
b	0	1	0	0	2.0
c	1	1	0	0	1.5
a	0	1	1	1	3.0
d	0	0	1	1	3.5
B^{Coluna}	2.0	2.0	3.5	3.5	

$K = 2$

Os baricentros das colunas da matriz estão ordenados, assim $M_1 \leftarrow \beta_C(M_0)$ não altera a ordem dos vértices das colunas. Porém, ao ordenar os baricentros das linhas,

altera-se a ordem dos vértices, ou seja: $M_2 \leftarrow \beta_L(M_1)$ altera a ordem (b, c, a, d) para (c, b, a, d).

$M_2 =$

	h	e	f	g	B^{Linha}
c	1	1	0	0	1.5
b	0	1	0	0	2.0
a	0	1	1	1	3.0
d	0	0	1	1	3.5
B^{Coluna}	1.0	2.0	3.5	3.5	

$K = 1$

Neste ponto, o algoritmo termina, uma vez que foi efetuado o número determinado de iterações (5), e o mapa correspondente à matriz final é apresentada na figura 4.7.

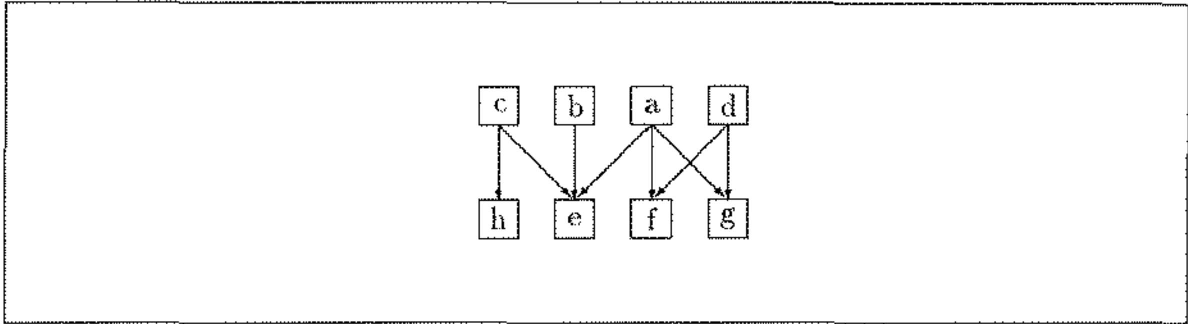


Figura 4.7: Hierarquia de Duas Camadas Resultante.

Observe que mesmo efetuando mais iterações, não se obterá uma nova matriz, uma vez que na matriz M_2 final, os baricentros das linhas e das colunas estão ordenados e que os dois vértices com baricentros iguais (f, g) estão ligados aos mesmos vértices.

Hierarquias de N-Camadas

Uma maneira de reduzir o número de cruzamentos de hierarquias com mais camadas, é utilizar o algoritmo descrito na seção anterior para cada par de camadas da hierarquia.

Porém, os vértices de determinadas camadas terão dois "vértices médios", um obtido em função da enumeração dos vértices destino das arestas emergentes (através de B^{Linha}), e outro em função da enumeração dos vértices origem das arestas incidentes (através de B^{Coluna}). Por exemplo, em uma hierarquia com três camadas, os vértices da segunda camada terão duas ordens $\sigma_2 = (v_1, \ldots, v_{|V_i|})$ e $\sigma_2' = (v_1', \ldots, v_{|V_i|}')$, obtidos por $\beta_L(M^{(2)})$ e $\beta_C(M^{(1)})$, respectivamente.

Observe que estas duas enumerações podem ser diferentes. Ao escolher uma delas é possível alterar o número de cruzamentos com a outra camada, podendo inclusive aumentar o número de cruzamentos da hierarquia.

Assim, utilizar o método da "força bruta", aplicando o algoritmo da seção anterior para pares aleatórios de camadas não parece ser uma boa solução.

Para contornar este problema, Sugiyama propôs uma seqüência de ordenações de linha e coluna para todas as camadas da hierarquia, onde utiliza uma característica das ordenações de linha e coluna.

Observe que a ordenação das colunas de uma matriz $M^{(i)}$ utiliza a enumeração dos vértices da camada i para obter a nova ordem dos vértices da camada $i+1$, podendo reduzir os cruzamentos entre i e $i+1$. Se em seguida forem ordenadas as colunas da matriz $M^{(i+1)}$, utilizar-se-á a enumeração recém obtida dos vértices da camada $i+1$ para ordenar $i+2$. Executando este processo para todas as camadas da hierarquia, é intuitivo que o número de cruzamentos pode ser reduzido, porém, ressalte-se que esta seqüência não garante as reduções e pode, inclusive, aumentá-las.

Assim, o algoritmo para n-camadas é composto por duas seqüências de ordenações, uma pelos baricentros das colunas e outra pelos baricentros das linhas. À seqüência de ordenção de colunas é dado o nome de Procedimento Baixo, pois a ordenação é feita de cima (camada 1) para baixo (camada n). Há uma seqüência similar de ordenação de linhas denominada Procedimento Cima, onde a ordenação é feita de baixo (camada n) para cima (camada 1).

A seguir, apresenta-se a rotina **Reduz-n** que contém o algoritmo para reduzir o número de cruzamentos em hierarquias com n-camadas. Ela utiliza uma nomenclatura diferente de **Reduz-2**, substituindo as matrizes pela ordem dos vértices de cada camada. Ou seja, uma matriz $M^{(i)}$ será representada por (σ_i, σ_{i+1}).

Assim, σ^* é a ordem dos vértices da hierarquia que apresentou o menor número de cruzamentos, enquanto que K^* é o número de cruzamentos de σ^*. σ é a ordem corrente dos vértices da hierarquia, enquanto σ_i é a ordem dos vértices da i-ésima camada de σ.

Nas operações de ordenação (β_C e β_L), seus parâmetros, que eram matrizes, foram substituidos pela ordem das camadas que compõe linha e coluna da matriz. Também o retorno destas operações foi modificado, retornando a nova ordem dos vértices da camada. Assim, $M_1 \leftarrow \beta_C(M_0)$, foi substituido por $\sigma_1 \leftarrow \beta_C(\sigma_1, \sigma_2)$.

As operações de troca de baricentros iguais, R_L e R_C, foram alteradas de maneira semelhante. Além disso, as trocas de baricentros iguais serão efetuadas após cada ordenação em que não houver troca na ordem dos vértices.

Reduz − n (σ)

$\sigma^* \leftarrow \sigma$
$K^* \leftarrow K(\sigma)$
Repetir
  Para $i \leftarrow 1$ *até* $n-1$, *repita*
    $\sigma_{i+1} \leftarrow \beta_C(\sigma_i, \sigma_{i+1})$
    Se (não houveram trocas)
      $\sigma_{i+1} \leftarrow R_C(\sigma_i, \sigma_{i+1})$
    Fimse
  Fimpara
  Se $(K(\sigma) < K^*)$
    $\sigma^* \leftarrow \sigma$
    $K^* \leftarrow K(\sigma)$
  Fimse
  Para $i \leftarrow n-1$ *até* 1, *repita*
    $\sigma_i \leftarrow \beta_L(\sigma_i, \sigma_{i+1})$
    Se (não houveram trocas)
      $\sigma_i \leftarrow R_L(\sigma_i, \sigma_{i+1})$
    Fimse
  Fimpara
  Se $(K(\sigma) < K^*)$
    $\sigma^* \leftarrow \sigma$
    $K^* \leftarrow K(\sigma)$
  Fimse
Até (Número determinado de iterações)
Retorna(σ^*)

Para ilustrar o funcionamento do algoritmo, será utilizado o mapa da figura 4.8.

Neste exemplo, executam-se duas iterações e como no exemplo anterior, não serão analisadas as operações sobre σ^*.

$M^{(1)} =$

	e	f	g
a	0	1	0
b	1	0	0
c	1	0	0
d	0	0	1
	2.5	1.0	4.0

$M^{(2)} =$

	h	i	j
e	1	0	0
f	1	1	1
g	0	1	0

$M^{(3)} =$

	k	l	m
h	0	0	1
i	1	1	0
j	0	0	0

$K(\sigma) = 5$

O primeiro passo do algoritmo é executar o Procedimento Baixo em todas as matrizes da hierarquia.

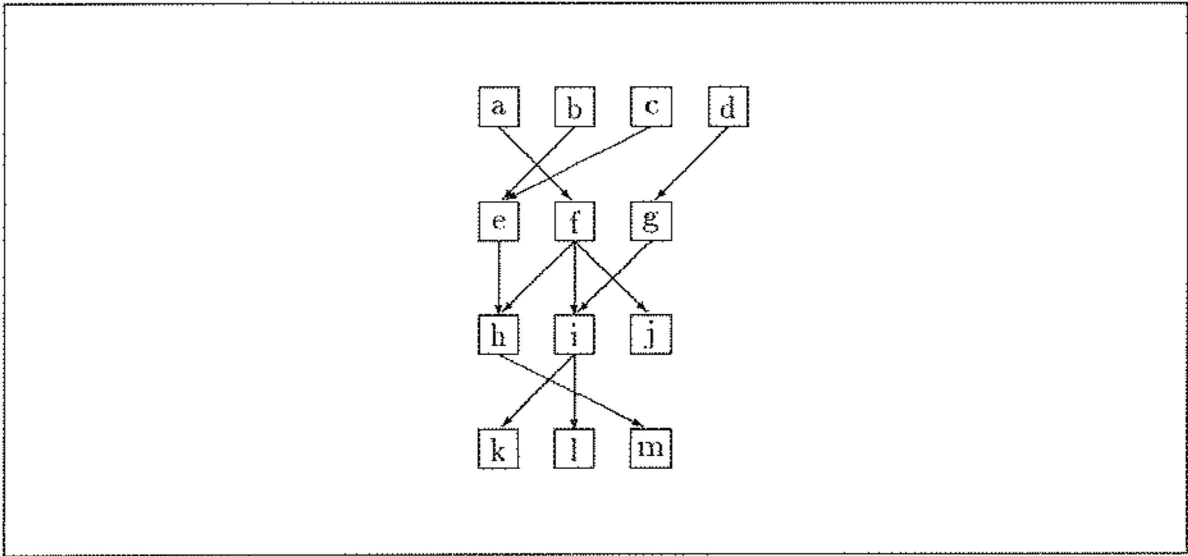


Figura 4.8: Hierarquia Antes de Efetuar a Redução

Inicialmente, ordena-se as colunas de $M^{(1)}$, obtendo a nova ordem (f, e, g). Esta nova ordem das colunas de $M^{(1)}$ implica em um atualização das linhas de $M^{(2)}$, como pode ser observado abaixo.

$$M^{(1)} = \begin{array}{c|ccc|} & f & e & g \\ \hline a & 1 & 0 & 0 \\ b & 0 & 1 & 0 \\ c & 0 & 1 & 0 \\ d & 0 & 0 & 1 \\ \hline \end{array} \quad M^{(2)} = \begin{array}{c|ccc|} & h & i & j \\ \hline f & 1 & 1 & 1 \\ e & 1 & 0 & 0 \\ g & 0 & 1 & 0 \\ \hline & 1.5 & 2.0 & 1.0 \end{array} \quad M^{(3)} = \begin{array}{c|ccc|} & k & l & m \\ \hline h & 0 & 0 & 1 \\ i & 1 & 1 & 0 \\ j & 0 & 0 & 0 \\ \hline \end{array}$$

$$K(\sigma) = 5$$

A segunda matriz tratada pelo Procedimento Baixo é $M^{(2)}$, onde obtém-se a nova ordem (j, h, i).

$$M^{(1)} = \begin{array}{c|ccc|} & f & e & g \\ \hline a & 1 & 0 & 0 \\ b & 0 & 1 & 0 \\ c & 0 & 1 & 0 \\ d & 0 & 0 & 1 \\ \hline \end{array} \quad M^{(2)} = \begin{array}{c|ccc|} & j & h & i \\ \hline f & 1 & 1 & 1 \\ e & 0 & 1 & 0 \\ g & 0 & 0 & 1 \\ \hline \end{array} \quad M^{(3)} = \begin{array}{c|ccc|} & k & l & m \\ \hline j & 0 & 0 & 0 \\ h & 0 & 0 & 1 \\ i & 1 & 1 & 0 \\ \hline & 3.0 & 3.0 & 2.0 \end{array}$$

$$K(\sigma) = 3$$

A última matriz tratada pelo Procedimento Baixo é $M^{(3)}$, onde obtém-se a ordem (m, k, l).

$$M^{(1)} = \begin{array}{c|ccc|} & f & e & g \\ \hline a & 1 & 0 & 0 \\ b & 0 & 1 & 0 \\ c & 0 & 1 & 0 \\ d & 0 & 0 & 1 \\ \hline & 1.0 & 2.5 & 4.0 \end{array} \quad M^{(1)} = \begin{array}{c|ccc|c} & j & h & i & \\ \hline f & 1 & 1 & 1 & 2.0 \\ e & 0 & 1 & 0 & 2.0 \\ g & 0 & 0 & 1 & 3.0 \\ \hline & 1.0 & 1.5 & 2.0 & \end{array} \quad M^{(3)} = \begin{array}{c|ccc|c} & m & k & l & \\ \hline j & 0 & 0 & 0 & 0.0 \\ h & 1 & 0 & 0 & 1.0 \\ i & 0 & 1 & 1 & 2.5 \\ \hline & 2.0 & 3.0 & 3.0 & \end{array}$$

$$K(\sigma) = 1$$

Neste momento, inicia-se o Procedimento Cima, onde ordena-se as linhas das matrizes. A primeira matriz tratada é $M^{(3)}$, porém suas linhas já estão ordenadas. A segunda matriz da seqüência é $M^{(2)}$, cujas linhas também já estão ordenadas, porém existem dois vértices com baricentros iguais que serão invertidos por R_L, obtendo a nova ordem (e, f, g). Observe que esta nova ordem aumentou o número de cruzamentos da matriz $M^{(1)}$, e também da hierarquia.

$M^{(1)} =$

	e	f	g	
a	0	1	0	2.0
b	1	0	0	1.0
c	1	0	0	1.0
d	0	0	1	3.0

$M^{(2)} =$

	j	h	i	
e	0	1	0	2.0
f	1	1	1	2.0
g	0	0	1	3.0

$M^{(3)} =$

	m	k	l
j	0	0	0
h	1	0	0
i	0	1	1

$K(\sigma) = 3$

A última matriz tratada pelo Procedimento Cima é $M^{(1)}$, onde obtém-se a ordem (b, c, a, d).

Aqui termina uma iteração do algoritmo.

$M^{(1)} =$

	e	f	g
b	1	0	0
c	1	0	0
a	0	1	0
d	0	0	1
	1.5	3.0	4.0

$M^{(2)} =$

	j	h	i	
e	0	1	0	2.0
f	1	1	1	2.0
g	0	0	1	3.0
	2.0	1.5	2.5	

$M^{(3)} =$

	m	k	l
j	0	0	0
h	1	0	0
i	0	1	1

$K(\sigma) = 1$

Inicia-se a segunda iteração executando o Procedimento Baixo para todas as matrizes da hierarquia, iniciando pela matriz $M^{(1)}$, cujas colunas já estão ordenadas. A segunda matriz no processo é $M^{(2)}$, onde obtém-se a ordem (h, j, i).

$M^{(1)} =$

	e	f	g	
b	1	0	0	1.0
c	1	0	0	1.0
a	0	1	0	2.0
d	0	0	1	3.0
	1.5	3.0	0.0	

$M^{(2)} =$

	h	j	i	
e	1	0	0	1.0
f	1	1	1	2.0
g	0	0	1	3.0
	1.5	2.0	2.5	

$M^{(3)} =$

	m	k	l	
h	1	0	0	1.0
j	0	0	0	0.0
i	0	1	1	2.5
	1.0	3.0	3.0	

$K(\sigma) = 0$

Após esta ordenação, obtém-se o menor número de cruzamentos para a hierarquia, zero. A figura 4.9 é o mapa da hierarquia obtida ao final do processo.

Melhoramentos

Rowe [Rowe 87] propôs alguns melhoramentos para este algoritmo básico. Observou que era comum os vértices serem jogados de um lado a outro pelas ordenações de linha e coluna. Partindo do princípio que a melhor posição neste caso seria o meio termo, ele introduziu uma nova ordenação que soma os valores dos baricentros de linha e coluna, ordenado-os em seguida. A esse novo passo deu o nome de baricentro médio, e com sua inclusão observou que a partir da quinta iteração as reduções, quando obtidas, eram em pequeno número, o que o levou a fixar o número de iterações em quatro.

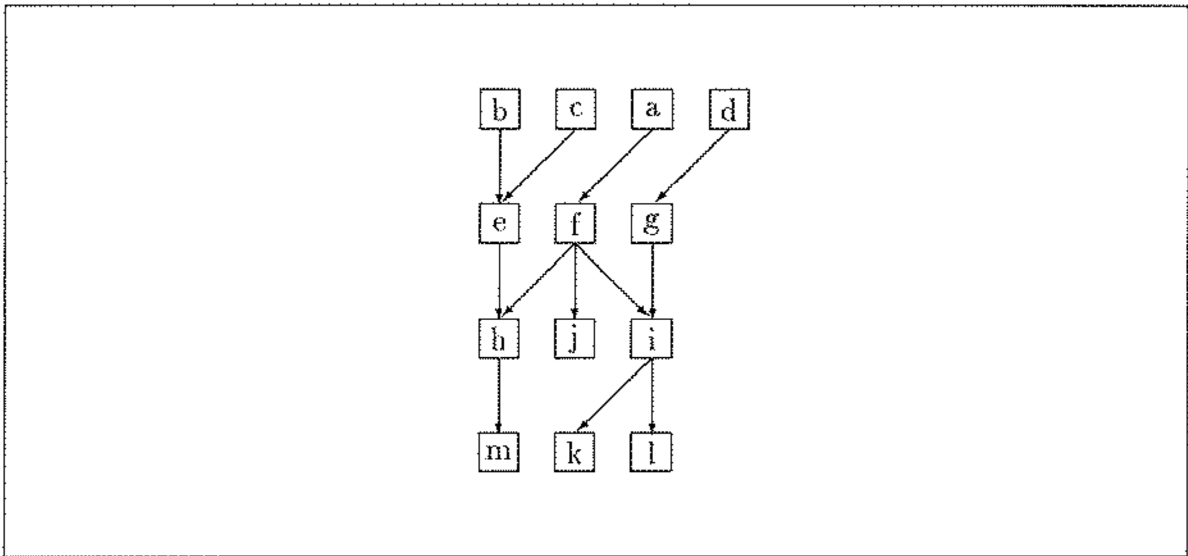


Figura 4.9: Hierarquia Resultante

Adaptações ao PCC

Adotou-se o melhoramento de Rowe, uma vez que gera resultados satisfatórios quando as hierarquias representam programas (principalmente os estruturados, onde freqüentemente há ordens onde não ocorre nenhum cruzamento), que são as hierarquias alvo do ambiente.

4.3.3 Posicionamento Horizontal (Passo 3)

Este passo recebe como entrada a hierarquia com a ordem de vértices obtida pelo passo 2 e determina a posição horizontal dos vértices de forma a facilitar a leitura, obtendo as características **C**, **D** e **E** (seção 4.1.1).

Assim, como no passo anterior, Sugiyama propõe dois métodos, QP (*Quadratic Programming Method*) e PR (*Prioriry Layout Method*). O método QP reduz o problema a equações matemáticas enquanto o método PR baseia-se em uma heurística semelhante ao método BC, sendo que seus resultados são superiores graficamente ao QP, além de ser mais rápido, razões que nos levaram a escolhê-lo para a implementação.

O algoritmo PR assemelha-se ao algoritmo BC, uma vez que alterna entre procedimentos baixo e cima. Porém, ao invés de utilizar os baricentros coluna e linha, utiliza os baricentros cima e baixo, que indicam a posição horizontal de cada vértice de acordo com uma prioridade.

A seguir, descreve-se as regras que devem ser adotadas ao alinhar cada vértice para obter as características **C**, **D** e **E**. Em seguida, apresenta-se um exemplo da aplicação destas regras nos vértices de uma camada da hierarquia.

Observe que a característica **E** é obtida pela posição determinada pelos baricentros Cima e Baixo.

1. Atribui-se valores à posição de cada vértice em uma camada. Por exemplo, sendo x_k^i a posição de v_k na camada i, pode-se atribuir os valores $\{1, 2, \ldots, \mid V_i \mid\}$ às posições $x_k^i, x_{k+1}^i, \ldots, x_{|V_i|}^i$. No caso, os valores atribuidos correspondem à enumeração dos vértices, mas poderiam ter sido utilizados outros valores crescentes.

2. Em cada procedimento Baixo e Cima, efetuam-se as seguintes ações.

   (a) Determinam-se as posições dos vértices uma a uma de acordo com sua prioridade. Os vértices fantasmas recebem maior prioridade para que se obtenha a característica **C**. A prioridade dos outros vértices é obtida de acordo com as equações de Conectividade, privilegiando os vértices com maior número de arestas incidentes (em C^{Cima}) ou emergentes (em C^{Baixo}), dependendo do caso.

   (b) Calcula-se a posição horizontal de cada vértice, iniciando por aquele que tiver a maior prioridade, de acordo com os valores dos baricentros Cima e Baixo nas seguintes condições:

      i. A posição do vértice é um número inteiro, e não pode ser igual às posições dos outros vértices da mesma camada.

      ii. A ordem dos vértices deve ser preservada, para que se mantenha a característica **B**.

      iii. Em caso de colisão (dois ou mais vértices forem mapeados para a mesma posição), o vértice de menor prioridade deve ser deslocado para a posição mais próxima possível da original, obtendo-se a característica **D**.

Assim como no BC, executa-se este algoritmo alternando entre os procedimentos Baixo e Cima quando são utilizados respectivamente B^{Baixo}, B^{Cima}, C^{Baixo} e C^{Cima}.

Exemplo

Este exemplo, consiste em posicionar os vértices da camada i da figura 4.10 calculando os baricentros Cima. Na figura, os vértices u_6 e v_2 são fantasmas.

Calcula-se a prioridade P dos vértices v_1, v_2, v_3 e v_4. Como v_2 é um vértice fantasma, atribui-se a ele um valor maior do que os demais, digamos 10, e aos outros atribui-se os valores obtidos com C^{Cima}, que indica o número de arestas incidentes. Assim, as

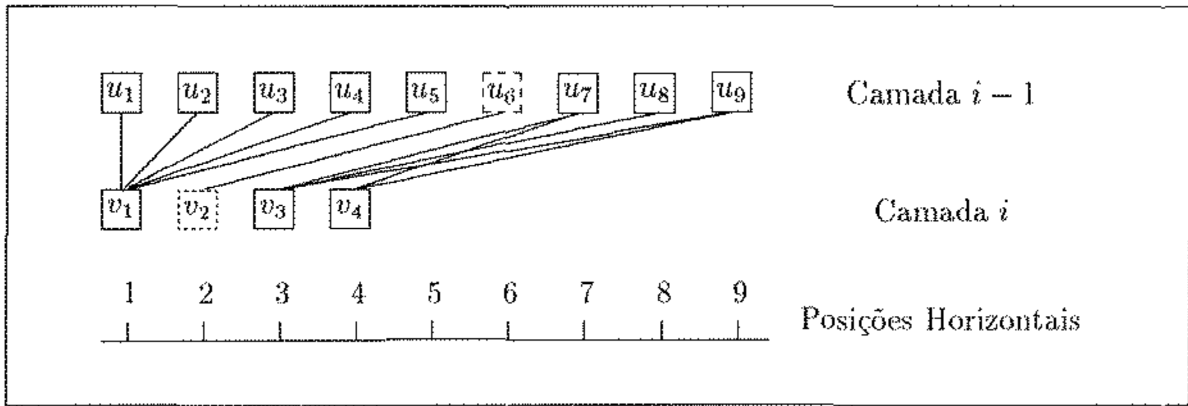


Figura 4.10: Hierarquia antes de efetuar o Alinhamento

prioridades serão as seguintes: $P(v_1) = 5$, $P(v_2) = 10$, $P(v_3) = 3$ e $P(v_4) = 2$. Assim, a ordem de posicionar os vértices será: v_2, v_1, v_3, v_4.

A posição horizontal de cada vértice, é determinada por B^{Cima}. Posiciona-se inicialmente o vértice de maior prioridade (v_2), onde $B^{Cima}(v_2) = 6$.

A nova posição dos vértices será $\{1, 6, 7, 8\}$. Observe que v_3 e v_4 também foram deslocados para manter a ordem obtida após o passo 2. O cálculo do segundo vértice na ordem de prioridade, v_1, altera a posição para $\{3, 6, 7, 8\}$ e o cálculo de v_3 altera para $\{3, 6, 8, 9\}$. O cálculo do baricentro de v_4 tem como resultado 8, porém esta posição já está ocupada por v_3, que tem prioridade maior. Assim, posiciona-se v_4 na posição mais próxima possível a v_3 respeitando a ordem estabelecida no passo 2, ou seja 9, resultando nas posições $\{3, 6, 8, 9\}$ representada graficamente na figura 4.11.

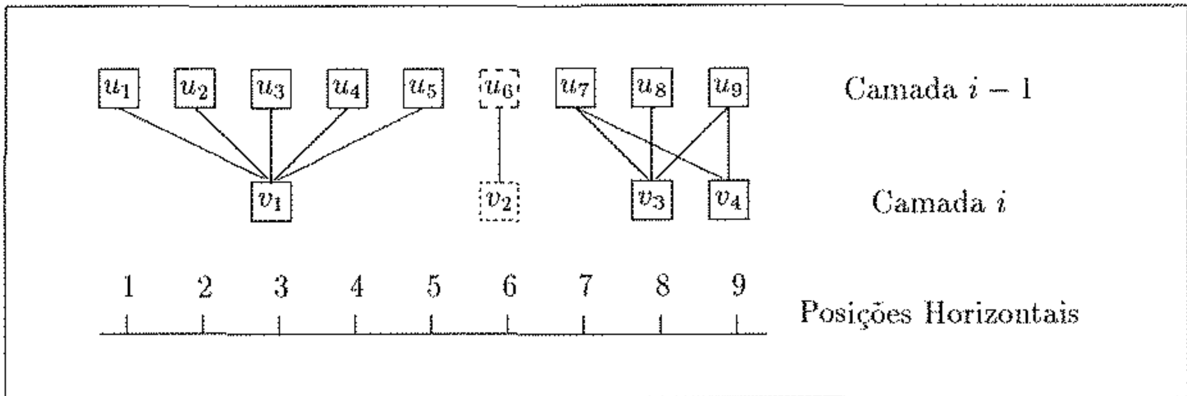


Figura 4.11: Hierarquia Alinhada

Melhoramentos

A literatura não apresenta melhoramentos para este passo.

Adaptações ao PCC

Há uma indefinição na descrição de Sugiyama, uma vez que não é determinada a ordem em que devem ser posicionados os vértices de mesma prioridade.

Nestes casos, adotou-se como prioridade, a enumeração dos vértices na camada. Isto causou uma tendência de alinhamento à esquerda dos vértices das primeiras camadas em Procedimentos Baixo, e um alinhamento à esquerda dos vértices das últimas camadas em Procedimentos Cima, como exemplificado na figura 4.12.

Para resolver este problema, efetua-se o Procedimento Cima para as primeiras camadas, e um Procedimento Baixo para as últimas, da seguinte maneira:

1. Executa-se um Procedimento Baixo para todas as camadas da hierarquia.

2. Procura-se, de baixo para cima, no vértice mais à direita de cada camada, a camada onde inicia-se o alinhamento à esquerda.

3. Executa-se um procedimento Cima a partir desta camada até a primeira.

Executando este procedimento no mapa da figura 4.12, obtém-se a figura 4.13.

4.3.4 Desenho (Passo 4)

Este é o último passo do algoritmo. Ele recebe como entrada a hierarquia com a definição das posições em que seus elementos devem ser desenhados e os desenha no terminal de vídeo.

Os vértices fantasma não são desenhados, e as arestas que incidem e emergem deles são ligadas para que se tenha as arestas longas.

O desenho em si é uma tarefa que envolve conceitos básicos de geometria analítica. Esta seção dará atenção somente à forma na qual será desenhado cada elemento (vértice, aresta, e dependência) e não como isto é feito.

Vértices

Para que o observador possa abstrair um maior número de informações a partir do desenho dos vértices são necessárias diversas representações.

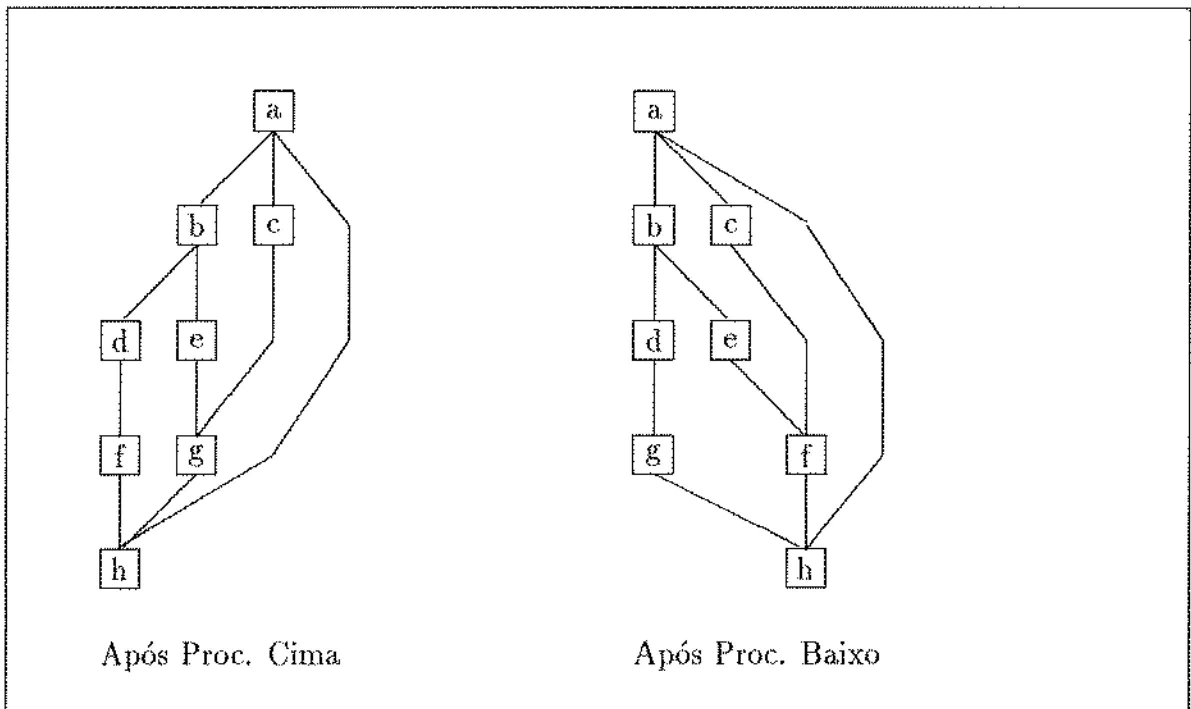


Figura 4.12: Exemplificação dos Resultados dos Procedimentos Baixo e Cima em uma Hierarquia

Como o widget se destina a representar programas, estipulamos três representações de vértices que permitem diferenciar visualmente os três tipos de estruturas que um programa pode conter: seqüencia, condicional e iteração, representadas graficamente por círculo, quadrado e losango.

Arestas

O maior problema em se desenhar as arestas que ligam dois vértices é especificar o local do vértice onde elas devem incidir ou emergir. Como são possíveis várias formas de vértices, decidiu-se por um único local de incidência ou emergência, como exemplificado na figura 4.14. Observe que as setas que indicam o sentido das arestas estão deslocados para que o desenho não fique sobrecarregado naqueles locais.

Dependências

As dependências são representadas por uma aresta. Porém, com elas representam um tipo de informação diferente das demais arestas, elas são desenhadas de forma diferente, além de não serem tratadas pelos passos anteriores do algoritmo, uma vez que nem sempre são apresentadas ao usuário. Assim, elas sobrepõem-se aos outros elementos

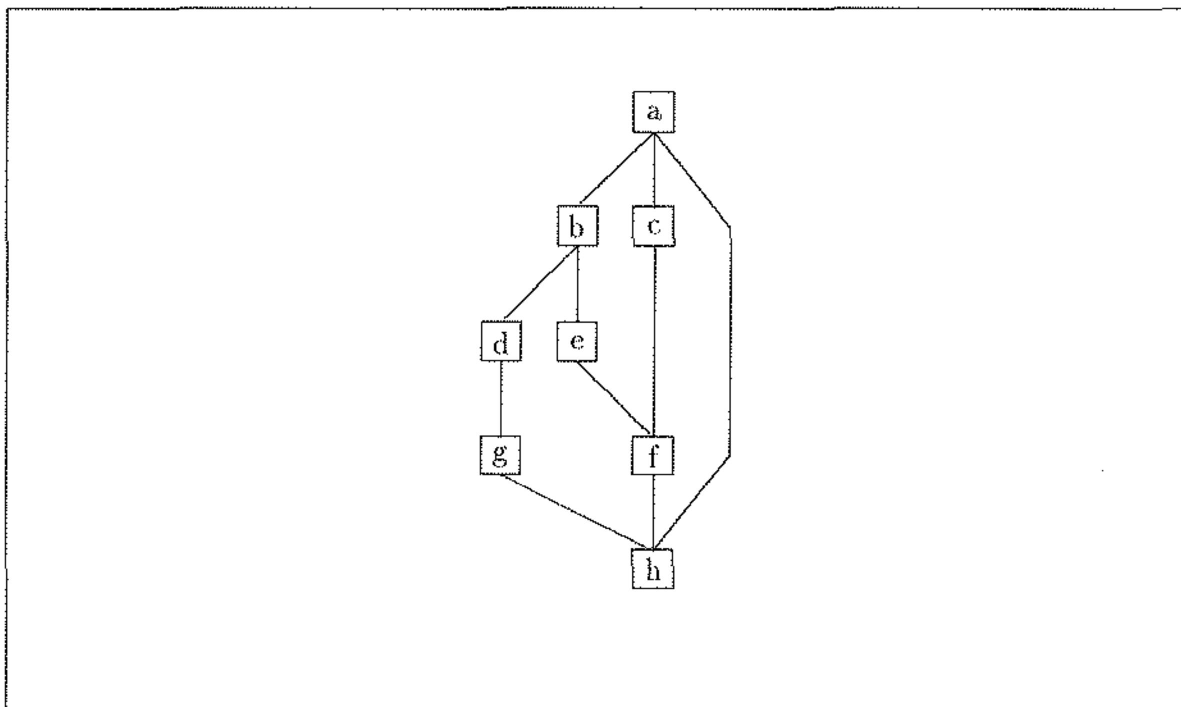


Figura 4.13: Hierarquia com uso misto dos procedimetos baixo e cima

do mapa e diferenciam-se das outras arestas na grossura da linha, mais grossa que as demais.

Para diferenciar os vários tipos de dependência, utiliza-se tipos diferentes de linhas (linha cheia, tracejada, traço e ponto e assim por diante), o que permite observá-las juntas em um mesmo mapa.

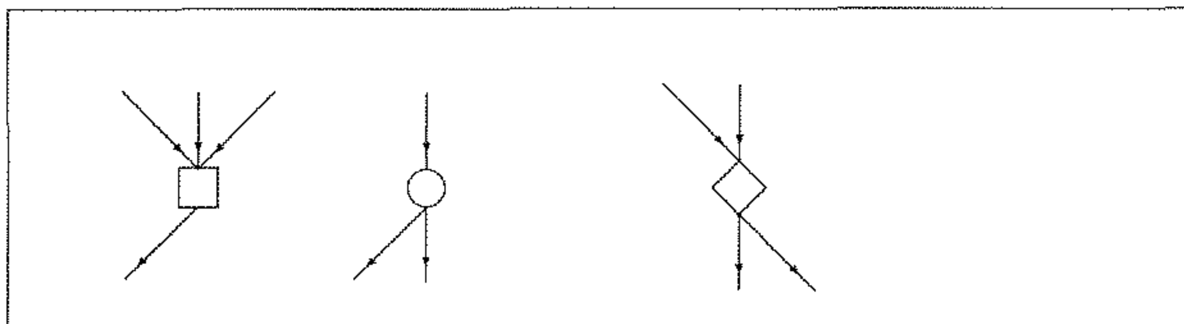


Figura 4.14: Exemplo de Desenho de Vértices e Local de Incidência de Arestas

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Conclusões

Este trabalho propõe uma interface de comunicação para o ambiente de reestruturação de programas do PCC. A proposta consta de dois tipos de widgets[1]: os usuais em ambientes de programação e os específicos à aplicação alvo.

Os widgets usuais são adaptações dos fornecidos pelo Motif. As adaptações eliminam informações desnecessárias ao programador da aplicação, possibilitando concentração nas tarefas mais nobres, vide o exemplo de menus da seção 3.2.5.

Estes widgets já estão sendo utilizados pelos demais membros do projeto em tarefas específicas. A experiência demonstra a espantosa facilidade de produzir aplicativos com estes widgets. O tempo necessário ao aprendizado e uso dos widgets em aplicações são surpreendentemente pequenos.

Dentre os widgets específicos, destaque-se o widget hierarquia (vide seção 3.3.3), que desenha, automaticamente, grafos a partir de conjuntos de vértices e arestas. Este widget é fundamental às ferramentas do ambiente. Dentre seus usos imediatos, destaca-se o desenho de grafos sintáticos de programas e a visualização da execução de um programa. Este widget está sendo utilizado em uma ferramenta que gerencia abstrações do programa que será descrito na próxima seção.

O algoritmo que desenha hierarquias é conhecido na literatura [Warf 77, Sugi 81, Rowe 87], e envolve uma heurística para resolver um problema NP-completo. O capítulo 4 descreve este algoritmo e apresenta alguns melhoramentos propostos e implementados por este trabalho baseado nas necessidades do ambiente, onde a hierarquia será utilizada para representar programas.

[1]Widget são explicados na seção 2.2.3

5.2 Trabalhos Futuros

Nenhum dos ambientes de reestruturação citados [PGHL 89, CCHK 87, GALS 87] apresenta todas as facilidades propostas para o ambiente do PCC, no que se refere a depuração de desempenho.

Porém, estas propostas, assim como os widgets implementados, não são completos. O único meio seguro de produzir interfaces de qualidade é testando protótipos com o usuário e modificando a interface baseado em seus comentários [Myer 89]. Assim, este trabalho deve ser visto como um protótipo passível de melhoramentos.

Um outro aspecto a ser analisado é que o ambiente do PCC ainda está em fase de desenvolvimento. Assim, algumas de suas ferramentas ainda não estão totalmente definidas, o que impossibilita a definição de suas necessidades de comunicação com o usuário.

Mesmo assim, é possível antecipar alguns melhoramentos, frutos de discussões internas ao projeto.

A seguir, apresenta-se algumas destas propostas.

1. **Textos**: este trabalho apresentou uma função para enviar um caractere ao usuário e uma função de retorno para receber um caractere digitados. Isto se deve à influência de uma das ferramentas, o editor, que utiliza o MicroEmacs como plataforma. A interface de comunicação do MicroEmacs utiliza somente duas funções, uma de envio e uma de recebimento de um caractere. As operações apresentadas neste trabalho são coerentes com tal interface.

   No futuro, serão feitos estudos para que as outras ferramentas do ambiente possam utilizar meios mais amigáveis de comunicação com o usuário através de facilidades providas pelo Motif, como o widget Text, por exemplo.

2. **Hierarquias**: propõe-se diversos melhoramentos para este widget.

   (a) inclusão de cursores, que serão uma alternativa para acompanhar a execução de programas. Cursores são pequenas figuras, como setas, bolas e quadrados, que podem representar os processadores. Assim, ao deslocar-se sobre vértices e arestas, os cursores indicam o trecho do programa que está sendo executado em cada processador.

   (b) pretende-se construir um editor de grafos, onde o usuário poderá incluir, alterar e excluir vértices, arestas e dependências. Além disso, também será possível alterar a posição destes widgets, uma vez que a posição gerada automaticamente nem sempre é satisfatória.

(c) criação de uma forma de armazenamento do mapa para futura recuperação. A partir desta forma intermediária será possível copiar o mapa em impressoras.

Atualmente existe um trabalho em andamento que utiliza o widget hierarquia. Ele destina-se a abstrair programas. Para tal, seleciona as informações do grafo do programa que serão desenhados pelo widget hierarquia. Esta ferramenta permite apresentar vários níveis de abstração do programa ao usuário, além de permitir apresentar somente as estruturas especificadas, como iterações que realizam operações sobre determinada matriz, por exemplo.

Apêndice A

As Funções da Interface

Este apêndice detalha as funções da interface do PCC.

NOME

IUabreJanela - Inicia a interface e abre uma janela.

SINTAXE

Janela* IUabreJanela(argc, argv)

int	argc;
char*	argv[];

PARÂMETROS

argc	Número de argumentos
argv	Ponteiro para argumentos

DESCRIÇÃO

Esta função inicia a interface e retorna um apontador para a estrutura Janela. argv e argc são usados pelo X Window.

RETORNO

NULL Erro na abertura da interface

NOME

IUcriaAT - Cria o widget área de trabalho.

SINTAXE

```
int IUcriaAT(janela)
    Janela*        janela;
```

PARÂMETROS

janela A janela em que deve ser aberta a área de trabalho.

DESCRIÇÃO

Esta função abre uma área de trabalho na janela especificada.

RETORNO

-1 Janela inexistente

NOME

IUenviaChar - Envia um caractere para a área de trabalho.

SINTAXE

```
int IUenviaChar(at, x, y, caractere)
    AreaTrabalho*   at;
    int             x;
    int             y;
    char            caractere;
```

PARÂMETROS

at	Área de trabalho.
x	Coluna em que deve ser colocado o caractere
y	Linha em que deve ser colocado o caractere
caractere	Caractere.

DESCRIÇÃO

Esta função coloca *caractere* nas coordenadas especificadas.

RETORNO

-1	Área de trabalho inexistente.

NOME

IUexpose - Associa função de retorno para tratar evento de expose no widget área de trabalho.

SINTAXE

```
int IUexpose(at, f, f_ret)
    AreaTrabalho*   at;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

at	Área de trabalho.
f	Função de retorno.
f_ret	parâmetro da função f.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento de expose.

RETORNO

-1	Área de trabalho inexistente.

NOME

IUhApagaDep - Apaga as dependências desenhadas por `IUMostraDep()`

SINTAXE

int IUhApagaDep(mapa, rotulo_origem, rotulo_destino, tipo)

```
Hierarquia*     mapa;
int             rotulo_origem;
int             rotulo_destino;
Dependencia     tipo;
```

PARÂMETROS

mapa	mapa.
rotulo_origem	vértice origem da dependência.
rotulo_destino	vértice destino da dependência.
tipo	tipo da dependência a ser mostrada.

DESCRIÇÃO

Esta função apaga as dependências mostradas por `IUhMostraDep()`, combinando as seguintes regras:

Se `rotulo_origem` $\neq 0$

Apaga dependências com origem em `rotulo_origem`.

Senão

Apaga todas as dependências do mapa.

Se `rotulo_destino` $\neq 0$

Apaga dependências com destino em `rotulo_destino`.

Senão

Apaga todas as dependências do mapa.

Se `tipo` $\in$ `Dependencia`

Apaga somente o tipo de dependência especificado.

Senão

Apaga todas os tipos de dependência.

RETORNO

-1	mapa inexistente.
-2	vértice origem inexistente.
-3	vértice destino inexistente.

NOME

IUhApagaMapa - Apaga o desenho da hierarquia.

SINTAXE

int IUhApagaMapa(mapa)

Hierarquia* mapa;

PARÂMETROS

mapa mapa.

DESCRIÇÃO

Esta função apaga o desenho da hierarquia.

RETORNO

-1 mapa inexistente.

NOME

IUhAresta - Associa função de retorno para tratar eventos em arestas do mapa especificado.

SINTAXE

```
int IUhAresta(mapa, f, f_ret)
    Hierarquia*    mapa;
    void           (*f)();
    char*          f_ret;
```

PARÂMETROS

mapa	mapa.
f	Função de retorno.
f_ret	parâmetro da função f.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento sobre uma das arestas do mapa.

RETORNO

-1	mapa inexistente.

NOME

IUhAtualizaHier - Altera vértices, arestas e dependências da hierarquia .

SINTAXE

int IUhAtualizaHier(mapa, vert, n_vert, arest, n_arest, dep, n_dep)

```
Hierarquia*     mapa;
TipoVertice*    vert;
int             n_vert;
TipoAresta*     arest;
int             n_arest;
TipoDepend*     dep;
int             n_dep;
```

PARÂMETROS

vert	Conjunto de vértices.
n_vert	Número de vértices.
arest	Conjunto de arestas.
n_arest	Número de arestas.
dep	Conjunto de dependências.
n_dep	Número de dependências.

DESCRIÇÃO

Esta função permite alterar os atributos de vértices, arestas e dependências de um mapa da hierarquia.

RETORNO

-1	vértice inexistente.
-2	aresta inexistente.
-3	dependência inexistente.

NOME

IUhCriaHier - Cria hierarquia na janela especificada.

SINTAXE

```
int IUhCriaHier(janela, vert, n_vert, arest, n_arest, dep, n_dep)
    Janela*        janela;
    TipoVertice*   vert;
    int            n_vert;
    TipoAresta*    arest;
    int            n_arest;
    TipoDepend*    dep;
    int            n_dep;
```

PARÂMETROS

janela	A janela em que deve ser criada a hierarquia.
vert	Conjunto de vértices.
n_vert	Número de vértices.
arest	Conjunto de arestas.
n_arest	Número de arestas.
dep	Conjunto de dependências.
n_dep	Número de dependências.

DESCRIÇÃO

Esta função cria o widget hierarquia na janela especificada.

O widget é definido por vértices, arestas e dependências.

RETORNO

-1	janela inexistente.
-2	conjunto de vértices vazio.

NOME

IUhDepend - Associa função de retorno para tratar eventos em dependências do mapa especificado.

SINTAXE

```
int IUhDepend(mapa, f, f_ret)
    Hierarquia*     mapa;
    void            (*f)();
    char*           f_ret;
```

f_ret parâmetro da função f.

PARÂMETROS

mapa mapa.

f Função de retorno.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento sobre uma das dependências do mapa.

RETORNO

-1 mapa inexistente.

NOME

IUhEliminaHier - Elimina a hierarquia.

SINTAXE

int IUhEliminaHier(mapa)

Hierarquia* mapa;

PARÂMETROS

mapa mapa.

DESCRIÇÃO

Esta função elimina a estrutura de dados utilizada para desenhar o mapa.

RETORNO

-1 mapa inexistente.

NOME

IUhMostraDep - Mostra as dependências de um mapa.

SINTAXE

int IUhMostraDep(mapa, rotulo_origem, rotulo_destino, tipo);

Hierarquia*	mapa;
int	rotulo_origem;
int	rotulo_destino;
Dependencia	tipo;

PARÂMETROS

mapa	mapa.
rotulo_origem	vértice origem da dependência.
rotulo_destino	vértice destino da dependência.
tipo	tipo da dependência a ser mostrada.

DESCRIÇÃO

Esta função mostra as dependências enviadas por `CriaHier()`, combinando as seguintes regras:

Se `rotulo_origem` $\neq 0$

Mostra dependências com origem em `rotulo_origem`.

Senão

Mostra todas as dependências do mapa.

Se `rotulo_destino` $\neq 0$

Mostra dependências com destino em `rotulo_destino`.

Senão

Mostra todas as dependências do mapa.

Se `tipo` $\in$ `Dependencia`

Mostra somente o tipo de dependência especificado.

Senão

Mostra todas os tipos de dependência.

RETORNO

-1	mapa inexistente.
-2	vértice origem inexistente.
-3	vértice destino inexistente.

NOME

IUhMostraMapa - Mostra o desenho da hierarquia.

SINTAXE

```
int IUhMostraMapa(mapa)
    Hierarquia*     mapa;
```

PARÂMETROS

mapa mapa.

DESCRIÇÃO

Esta função desenha a hierarquia no terminal de vídeo.

RETORNO

-1 mapa inexistente.

NOME

IUhVertice - Associa função de retorno para tratar eventos em vértices do mapa especificado.

SINTAXE

```
int IUhVertice(mapa, f, f_ret)
    Hierarquia*     mapa;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

mapa	mapa.
f	Função de retorno.
*f*_ret	parâmetro da função f.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento sobre um dos vértices do mapa.

RETORNO

-1	mapa inexistente.

NOME

IUmenubar - Cria uma menubar na janela especificada.

SINTAXE

```
int IUmenubar(janela, menu, n_itens)
    Janela*          janela;
    IUstruct_menu*   menu;
    int              n_itens;
```

PARÂMETROS

janela	Janela.
menu	Itens que deverão compor a menubar.
n_itens	Número de itens da menubar.

DESCRIÇÃO

Esta função cria uma menubar na janela especificada.

Os itens do menu estão definidos no parâmetro *menu*.

RETORNO

-1	Janela inexistente.
-2	Menu não declarado.

NOME

IUmouse - Associa função de retorno para tratar evento de ação no mouse.

SINTAXE

```
int IUmouse(at, f, f_ret)
    AreaTrabalho*   at;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

at	Área de trabalho.
f	Função de retorno.
f_ret	parâmetro da função f.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento de mouse na área de trabalho.

RETORNO

-1	Área de trabalho inexistente.

NOME

IUpopup - Cria um menu popup na janela especificada.

SINTAXE

int IUpopup(janela, titulo, menu, n_itens, acelerador)

```
Janela*          janela;
char*            titulo;
IUstruct_menu*   menu;
int              n_itens;
char*            acelerador;
```

PARÂMETROS

janela	janela.
titulo	Título do menu popup.
menu	Itens que deverão compor o menu popup.
n_itens	Número de itens do menu popup.
acelerador	acelerador do menu popup.

DESCRIÇÃO

Esta função cria um menu popup na janela especificada.

Os itens do menu estão definidos no parâmetro *menu*.

RETORNO

-1	Janela inexistente.
-2	Menu não declarado.

NOME

IUposicScrollH - Posiciona slider na Scrollbar horizontal.

SINTAXE

```
void IUposicScrollH(at, tot_larg, ini_desenho, largura_at)
    AreaTrabalho*   at;
    int             tot_larg;
    int             ini_desenho;
    int             largura_at;
```

PARÂMETROS

at	Área de trabalho.
tot_larg	Largura total do desenho em *Pixmap*
ini_desenho	Largura de *Pixmap* correspondente ao início do desenho na área de trabalho.
largura_at	Largura da área de trabalho.

DESCRIÇÃO

Esta função dá tamanho e forma corretos ao slider da scrollbar horizontal.

NOME

IUposicScrollV - Posiciona slider na Scrollbar vertical.

SINTAXE

int IUposicScrollH(at, tot_alt, ini_desenho, altura_at)

```
AreaTrabalho*   at;
int             tot_alt;
int             ini_desenho;
int             altura_at;
```

PARÂMETROS

at	Área de trabalho.
tot_alt	Altura total do desenho em *Pixmap*
ini_desenho	Altura de *Pixmap* correspondente ao início do desenho na área de trabalho.
largura_at	Altura da área de trabalho.

DESCRIÇÃO

Esta função dá tamanho e forma corretos ao slider da scrollbar vertical.

NOME

IUresize - Associa função de retorno para tratar evento de redimensionamento da área de trabalho.

SINTAXE

```
int IUresize(at, f, f_ret)
    AreaTrabalho*   at;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

at	Área de trabalho.
f	Função de retorno.
*f*_ret	parâmetro da função f.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento de redimensionamento da área de trabalho.

RETORNO

-1	Área de trabalho inexistente.

NOME

IUscrollH - Cria uma Scrollbar horizontal na área de trabalho.

SINTAXE

```
int IUscrollH(at, f, f_ret)
    AreaTrabalho*   at;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

at	Área de trabalho.
f	Função de retorno.
f_ret	Parâmetro da função f.

DESCRIÇÃO

Esta função cria o widget Scrollbar Horizontal na área de trabalho especificada, associando uma função de retorno a ser disparada ao ocorrer algum ação sobre a scrollbar.

RETORNO

-1	Área de trabalho inexistente

NOME

IUscrollV - Cria uma Scrollbar vertical na Área de Trabalho.

SINTAXE

```
int IUscrollV(at, f, f_ret)
    AreaTrabalho*   at;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

at	Área de trabalho.
f	Função de retorno.
f_ret	parâmetro da função f.

DESCRIÇÃO

Esta função cria o widget Scrollbar Vertical na área de trabalho especificada e associa uma função de retorno a ser disparada ao ocorrer alguma ação sobre a scrollbar.

RETORNO

-1	Área de trabalho inexistente.

NOME

IUteclado - Associa função de retorno para tratar evento de teclado.

SINTAXE

```
int IUteclado(at, f, f_ret)
    AreaTrabalho*   at;
    void            (*f)();
    char*           f_ret;
```

PARÂMETROS

at	Área de trabalho.
f	Função de retorno.
f_ret	parâmetro da função f.

DESCRIÇÃO

Esta função indica a função a ser disparada ao ocorrer um evento de teclado na área de trabalho.

RETORNO

-1	Área de trabalho inexistente.

Apêndice B

As Bibliotecas da Interface

Este apêndice contém as bibiotecas 'C' que compõe a API da interface. A seção B.1, apresenta a biblioteca IUtradic.h, que contém os widgets tradicionais, a seção B.2 apresenta a biblioteca IUhier.h, que contém as bibliotecas para uso de hierarquias, e a seção B.3 apresenta a biblioteca IUevent.h, que contém as estruturas utilizadas nas funções de retorno.

B.1 IUtradic.h

```
#include <X11/Xlib.h>
#include <Xm/Xm.h>
#include <X11/keysym.h>

typedef struct _ChamadaFuncao {
        void    (*funcao)();        /* Funcao de retorno associada       */
        char*   par;                /* Parametro retornado pela funcao() */
} ChamadaFuncao;

/*------------------------------------------------------------------- */
/*                      Estruturas de janela                          */
/*                      --------------------                          */

typedef struct _AreaTrabalho {
        Widget        forma;        /* Constraint de Widget              */
        Widget        sw;           /* Scrolled Window                   */
```

```
        Widget          canvas;         /* Canvas para desenho e textos        */
        GC              gc;             /* Graphic Context para canvas         */
        Widget          sbh;            /* Scrollbar Horizontal                */
        Widget          sbv;            /* Scrollbar Vertical                  */
        ChamadaFuncao   *mouse;         /* Retorno de acao de Mouse            */
        ChamadaFuncao   *teclado;       /* Retorno de acao de Teclado          */
        ChamadaFuncao   *expose;        /* Retorno de acao de Expose           */
        ChamadaFuncao   *resize;        /* Retorno de acao de Resize           */
        ChamadaFuncao   *scrollh;       /* Retorno de acao em scroll horiz.    */
        ChamadaFuncao   *scrollv;       /* Retorno de acao em scroll vert.     */
} AreaTrabalho, Hierarquia;


typedef struct _Janela {
        Widget          topo;           /* Janela base                         */
        Widget          mb;             /* Menubar                             */
        Widget          popup;          /* Menu Popup                          */
        AreaTrabalho* at;               /* Area de trabalho associada a topo */
        Hierararquia* hier;             /* Hierarquia associada ao topo */
} Janela;

/*------------------------------------------------------------------- */
/*                      Estruturas de menu                             */
/*                      ------------------                             */

typedef struct _struct_menu {
        char*   nome;                   /* Texto do item de menu               */
        void    (*f_ret)();             /* Funcao de retorno associada         */
        char*   dado_ret;               /* Parametro de f_ret                  */
        struct _struct_menu*
                sub_menu;               /* Apontador p/ menu encadeado         */
        int     n_sub_itens;            /* Numero de itens do menu encadeado */
        char*   tit_sub_menu;           /* Titulo do menu  encadeado           */
        char*   acelerador;             /* Acelerador                          */
} IUstruct_menu;
```

```
/*-------------------------------------------------------------------------*/
/*                          Prototipos                                     */
/*                          ----------                                     */

Janela  (* IUabreJanela());       /* Inicia interface. Abre uma janela*/
int     IUcriaAT();               /* Cria uma area trab. na janela    */
int     IUexpose();               /* Funcao p/ evento de expose       */
int     IUresize();               /* Funcao p/ evento de resize       */
int     IUmouse();                /* Funcao p/ evento de mouse        */
int     IUteclado();              /* Funcao p/ evento de teclado      */
int     IUscrollV();              /* Cria SBVert. com funcao associada*/
int     IUscrollH();              /* Cria SBHor. com funcao associada */
int     IUposicScrollV();         /* Posiciona Slider Vertical        */
int     IUposicScrollH();         /* Posiciona slider Horizontal      */
int     IUMenubar ();             /* Cria menubar na janela           */
int     IUPopup ();               /* Cria popup na janela             */
int     IUenviaChar();            /* Envia um caractere para o video  */
```

B.2 IUhier.h

```
/* --------------------------------------------------------------------- */
/*                  Estrutras de dados de hierarquias                    */
/*                  ---------------------------------                    */

typedef enum {
        CIRCULO, QUADRADO, LOSANGO
} FormaVertice;

typedef enum {
        INVISIVEL, CONTORNO, HACHURADO
} Visibilidade;

typedef enum {
        SAIDA, ANTI, FLUXO, CONTROLE
} Dependencia;
```

```
typedef enum {
        DEFAULT, BRANCO, PRETO
} Cor;

typedef struct {
        int             rotulo;     /* Rotulo do vertice                  */
        char            texto[15];  /* Texto a ser mostrado no vertice */
        FormaVertice    forma;      /* forma na qual sera desenhado       */
        Cor             cor_forma,  /* cor do contorno                    */
                        cor_texto;  /* cor do texto                       */
        Visibilidade    visibilidade; /* visibilidade                     */
        int             peso;     /* indica a ordem de escolha em ciclos*/
} TipoVertice;

typedef struct {
        int             rotulo_origem, /* vertice origem da aresta        */
                        rotulo_destino;/* vertice destino da aresta       */
        char            texto[15];     /* texto a ser mostrado            */
        Cor             cor_aresta,    /* cor da aresta                   */
                        cor_texto;     /* cor do texto                    */
        Visibilidade    visibilidade;  /* visibilidade                    */
} TipoAresta;

typedef struct {
        int             rotulo_origem, /* vertice origem da depend.       */
                        rotulo_destino;/* vertice destino da depend.      */
        char            texto[15];     /* texto a ser mostrado            */
        Cor             cor_depend.,   /* cor da depend.                  */
                        cor_texto;     /* cor do texto                    */
        Dependencia     dependencia;   /* tipo de depend.                 */
} TipoDepend;

/* ----------------------------------------------------------------- */
/*                Prototipos de hierarquias                          */
/*                -------------------------                          */
```

```
int         IUhCriaHier());          /* Cria Hierarquia                   */
int         IUhVertice();            /* Funcao p/ acao sobre vertice      */
int         IUhAresta();             /* Funcao p/ acao sobre aresta       */
int         IUhDepend();             /* Funcao p/ acao sobre dependencia  */
int         IUhAtualizaHier();       /* Altera vertices, arestas, depend. */
int         IUhMostraMapa();         /* Desenha mapa no video             */
int         IUhApagaMapa();          /* Apaga mapa do video               */
int         IUhEliminaHier();        /* Apaga mapa da memoria             */
int         IUhMostraDep();          /* Mostra as dependencias no mapa    */
int         IUhApagaDep();           /* Apaga as dependencias no mapa     */
```

B.3 IUevent.h

```
typedef struct {
        AreaTrabalho   *at           /* Area Trab. onde houve evento      */
} EventExpose;

typedef struct {
        AreaTrabalho   *at           /* Area Trab. onde houve evento      */
        unsigned int   largura;      /* Nova largura da janela            */
        unsigned int   altura;       /* Nova altura  da janela            */
} EventResize;

typedef struct {
        AreaTrabalho   *at           /* Area Trab. onde houve evento      */
        unsigned int   x;            /* Coordenadas (x,y) da area de      */
        unsigned int   y;            /* trabalho onde houve acao de mouse */
} EventMouse;

typedef struct {
        AreaTrabalho    *at;         /* Area Trab. onde houve evento      */
        unsigned int    x,           /* Coordenadas (x,y) da area de      */
        unsigned int    y;           /* trabalho onde houve acao de teclado*/
        KeySym          caract;      /* char digitado (usa padrao KeySym.h)*/
} EventTeclado;
```

```
typedef struct {
        AreaTrabalho   *at;        /* Area Trab. onde houve evento       */
        unsigned int   inicio_y,  /* coordenada y do desenho onde inicia*/
                                  /* a janela.                          */
} EventScrollV;

typedef struct {
        AreaTrabalho   *at;        /* Area Trab. onde houve evento       */
        unsigned int   inicio_x,  /* coordenada x do desenho onde inicia*/
                                  /* a janela.                          */
} EventScrollH;

typedef struct {
        Hierarquia*     mapa; /* Mapa onde houve acao */
        int             rotulo;    /* Rotulo vert.                       */
} EventVertice;

typedef struct {
        Hierarquia*     mapa;      /* Mapa onde houve acao               */
        int             rotulo_origem;/*  Rotulo vert. orig. da aresta   */
        int             rotulo_destino;/* Rotulo vert. dest. da aresta   */
} EventAresta;

typedef struct {
        Hierarquia*     mapa;      /* Mapa onde houve acao               */
        int             rotulo_origem;/*  Rotulo vert. orig. da depend. */
        int             rotulo_destino;/* Rotulo vert. dest. da depend. */
        Dependencia     dependencia;/* tipo da dependencia selecionada  */
} EventDepend;

typedef struct {
        Janela*         janela;     /* Janela em que ocorreu o evento    */
} EventMenu;
```

Bibliografia

[BlRo 86] S. A. Bly, J. K. Rosenberg: *A Comparison of Tiled and Overlapped Windows*, ACM-CHI Preceedings, Apr. 1986, págs. 101, 106

[CCHK 87] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczan, S. W. Warren: *A Practical Environment for Scientific Programming*, IEEE Computer, Nov. 1987, págs. 75-89.

[CDL 88] D. Callahan, J. Dongarra, D. Levine: *Vectorizing Compilers: A Test Suite and Results*, Proceedings of the Supercomputer Conference, Nov. 1988, págs. 14-18.

[GALS 87] D. Gannon, D. Atapattu, M. A. Lee, B. Shei: *A Software Tool for Building Supercomputer Applications*, Technical Report 224, Computer Science Department, Indianna University, 1987.

[HaHi 89] H. R. Hartson, D. Hix: *Human-Computer Interface Development: Concepts and Systems for its Managment*, ACM Computer Surveys, Vol 21, No. 1, Mar. 1989, págs. 5-91.

[JaGu 88] D. Jablonowski, V. Guarna Jr: *GMB: A Dynamic Graph Tool and Its Use in an Integrated Programming Environment*, Technical Report 746, Computer Science Department, Indianna University, 1988.

[JaGu 89] D. Jablonowski, V. Guarna Jr: *GMB: A Tool for Manipulating and Animating Graph Data Structures*, Software-Practice and Experience, Vol 19, No. 3, Mar 89, págs. 283-301.

[JoRe 90] Johnson & Reichards: *Advanced X Window Applications Programming*, MIS Press, 1990.

[Jones 89] O. Jones: *Introdutcion to the X Window System*, Prentice Hall, 1989.

[LSVC 89] T. Lehr, Z. Segal, D. F. Vraslovic, E. Caplan, A. L. Chung, C. E. Fineman: *Visualizing Performance Debugging*, IEEE Computer, Oct. 1989, págs. 38-51.

[Maci 90] F. B. Maciel: *Um Ambiente de Reestruturação e Compilação de Programas para Máquinas Paralelas*, Tese de Mestrado, ITA, 1990.

[McHe] C. E. McDowell, D. P Helinbold: *Debugging Concurrent Programs*, ACM Computing Surveys, Vol 21, No. 4, Dez 89.

[MVMP 90] B. Müller, E. Voigt, F. C. Mokarzel, J. Panetta, O. C. Imamura, W. L. C. Saliba: *Atividades do Projeto Computação Científica para o Biênio 1991-1992*, Relatório interno do IEAv, Mar. 1991.

[Myer 84] B. A. Myers: *The User Interface for Sapphire*, IEEE Computer Graphics & Applications, Dez. 1984, págs. 13-23.

[Myer 86] B. A. Myers: *A Complete and Efficient Implementation of Covered Windows*, IEEE Computer, Set. 1986, págs. 57-67.

[Myer 89] B. A. Myers: *User-Interface Tools: Introduction and Survey*, IEEE Software, Jan. 1989, págs. 15-23.

[PGHL 89] C. D. Polychronopoulos, M Girkon, M. R. Haghighat, C. L. Lee, B. Leung, D. Shauten: *Parafrase 2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors*, International Journal of High Speed Computing, Vol 1, No. 1, págs. 45-72, 1989.

[Robi 88] G. Robins: *Applications of the ISI Grapher*, Artifitial Intelligence and Advanced Computers Technology Conference, Long-Beach, California, Mai 88, págs. 105-129.

[Rowe 87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, A. Tuan: *A Browser for Directed Graphs*, Software-Practice and Experience, Vol 17, No. 1, Jan 87, págs. 61-76.

[SLL 86] J. Smart, F. Leygues, M. Lichteshein: *User Interface Technical Assessment Report*, ECMA - TC33, Technical Assessment Ad-Hoc Group: User Interfaces, Mar. 1989.

[ScGe 86] R. W. Sheifer, J. Gettis: *The X Window System*, ACM Trans. on Graphics, Vol 5, No. 2, Apr. 1986, págs. 79-109.

[Sugi 81] K. Sugiyama, S. Tagawa, M. Toda: *Methods for Visual Understanding of Hierarchical System Structures*, IEEE Trans. on Systems, Man and Cybernetics, Vol 11, No. 2, Fev 81, págs. 109-125.

[Tesl 81] L. Tesler: *The Smalltalk Environment*, Byte, Ago. 1981, págs. 90-147.

[TiNe 87] W. F. Tichy, F. J. Newbery: *Knowledge-based Editors for Directed Graphs*, 1st European Software Engeneering Conference, Set 87, págs. 101-109.

[Warf 77] J. N. Warfield: *Crossing Theory and Hierarchy Mapping*, IEEE Trans. Syst. Man, Cybern., vol. 7, No. 7, Ago. 77 págs. 505-523.

[Youn 90] D. Young: *The X Window System - Programming and Application with Xt. OSF/Motif Edition*, Prentice Hall, 1990.