VISUALIZAÇÃO DE GRAFOS DE PROGRAMA : Uma Abordagem sem Reposicionamento

PLÍNIO ROBERTO SOUZA VILELA[1] JOSÉ CARLOS MALDONADO[2] MARIO JINO[3] MARCOS L. CHAIM[4]

[1]DCA-FEE-UNICAMP
Av. Albert Einstein 400, Cep 13081-970, CxP. 6101, Campinas, SP, Brasil
vilela@dca.fee.unicamp.br

[2]Inst. de Ciências Matemáticas de Sao Carlos - USP
jcmaldon@icmsc.sc.usp.br

[3]DCA-FEE-UNICAMP
jino@dca.fee.unicamp.br

[4]CNPTIA/Embrapa
chaim@cnptia.embrapa.br

Abstract. Program graph visualization has important applications from design to maintenance in software production. Specially in testing several software characteristics are observed through visualization; this is a motivation for the development of algorithms for automatic generation of program graphs.

Several authors have presented works in associated areas such as trees, directed planar graphs, general graphs, and specially program graphs display. Most of these works make use of trial-and-error methods; in other words, if the necessary space to position a node is not reserved, a relocation of its ancestors is required in order to avoid overlapping.

An algorithm which deals with the program graph drawing problem is shown. The algorithm is composed of two steps : node position computation and branch routing. For node position computation a modified tree level calculation function is used to find Y coordinates. In determining X coordinates a new concept, named scope, was defined. The main characteristic of this approach is to allow X positions to be found without relocation.

The branch routing uses positions left free by node position computation to find routing options; the best ones are chosen for each branch.

1 Introdução

A visualização dos grafos de programa tem importante aplicação na engenharia de software; sua utilização vai desde a fase de projeto, onde pode-se observar a estrutura do programa mesmo antes da codificação, até a fase de manutenção.

Em especial na fase de teste a visualização dos grafos de programa pode ser utilizada para identificar a estrutura interna do programa em teste, seus principais comandos, caminhos executados ou não pelos casos de teste, associações requeridas por diversos critérios de teste e medidas de complexidade. Além disso, pode-se identificar imediatamente anomalias como código morto e falta de estruturação.

No teste estrutural de software devemos gerar casos de teste que exercitem caminhos, ou associações específicas, dentro da estrutura do programa; assim, um conhecimento detalhado das informações associadas a esses elementos é essencial para que seja gerado um conjunto efetivo de casos de teste.

O desenho, mesmo manual, de grafos de programa não é trivial; erros freqüentemente ocorrem, o que leva essa atividade a ser dispendiosa, tanto em relação ao tempo quanto em relação ao custo.

A importância da visualização de informações de teste e dos grafos de programa em ferramentas de teste estrutural de software e o alto custo de sua elaboração manual motivam a construção de ferramentas que automatizem essa atividade.

A abordagem proposta neste trabalho gera, a partir de uma representação textual em forma de lista de adjacência (Figura 1-A), uma representação gráfica do grafo de programa tal que, uma vez definida uma posição, tanto de um nó quanto de um arco, essa posição não mais é alterada. A representação em forma textual utilizada é gerada pela

POKE-TOOL [1], onde o primeiro número indica o número total de nós do grafo e o número 0 indica o fim da lista de sucessores.

Essa abordagem tem alguns pontos em comum e algumas diferenças em relação a outros trabalhos publicados na literatura; por exemplo : Rowe et. al. [ROW87] apresentam um programa de propósito geral para posicionar grafos dirigidos; Gansner et. al. [GAN88] descrevem o programa DAG para desenhar grafos dirigidos; Kamada et. al. [KAM89] apresentam um algoritmo para desenhar grafos gerais não dirigidos; Wetherell e Shannon [WET79] e Reingold e Tilford [REI81] desenvolvem algoritmos para posicionamento de árvores; por fim, Price et. al. [PRI87] mostram um algoritmo para desenhar grafos de programa.

Na próxima seção são apresentados os principais problemas da visualização dos grafos de programa; na Seção 3, estão os algoritmos propostos para resolver esses problemas; e, na Seção 4, apresentam-se as conclusões desse trabalho.

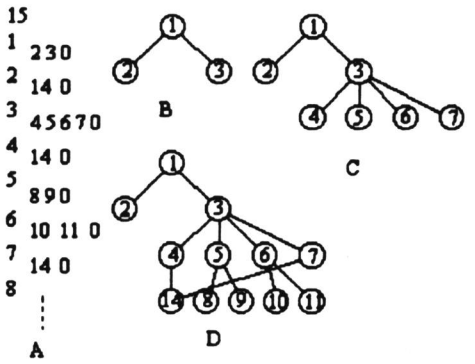

Figura 1: Problemas no Posicionamento dos Grafos de Programa

2 Problemas na Visualização de Grafos de Programa

Basicamente os problemas que devem ser tratados por algoritmos de visualização de grafos de programa são separados em dois tipos: como definir, a priori, qual será o espaço ocupado por um dado subgrafo (tanto na dimensão X quanto na Y) dentro do grafo completo, e como definir as rotas por onde devem ser traçados os arcos de forma a evitar os cruzamentos.

Tomando-se como exemplo o grafo de programa mostrado na Figura 1-A, observa-se que uma das abordagens para se definir posições na coordenada X é atribuir deslocamentos padrões entre os nós (veja Figura 1-B). Apesar de essa abordagem poder ser aplicada sem nenhum tipo de conhecimento prévio da estrutura global do grafo e, conseqüentemente, não exigir nenhum processamento extra para adquirir essa informação, ela ainda assim mostra-se ineficiente com relação ao tempo de processamento; principalmente porque, para cada nó que não puder ser posicionado devido à falta de espaço, deverá ser feito um rearranjo de todos os seus antecessores, além de uma avaliação (e possível rearranjo) de todos os outros nós do grafo.

Essa situação motiva a utilização de informações que, de algum modo, reflitam a disposição global do grafo de programa; ou que pelo menos forneçam uma aproximação dessa disposição. Esse objetivo foi atingido neste trabalho através da utilização de um conceito denominado *escopo* que será apresentado na Seção 3.1; informalmente, ele reflete o espaço que deve ser reservado abaixo de cada nó de forma que seus filhos possam ser posicionados sem sobreposição.

O problema de alocação de espaçamento padrão pode ser observado na Figura 1-C que mostra o posicionamento dos nós filhos do nó 3; como um espaço insuficiente foi reservado, não houve como evitar o desalinhamento dos descendentes dos nós 5 e 6 (Figura 1-D). Mesmo não havendo sobreposição esse tipo de disposição é desaconselhável segundo regras estéticas já consolidadas [WET79], [REI81].

Outro problema, apresentado na Figura 1-D, é o cruzamento do arco (7,14), provocado pelo posicionamento incorreto do nó 14; esse caso ilustra o problema da determinação da coordenada Y dos nós.

Os problemas apresentados até agora dizem respeito, principalmente, à determinação das coordenadas X e Y de cada nó; outro problema, que deve ser tratado pelo algoritmo de visualização, diz respeito à determinação do melhor traçado que deve ser seguido pelos arcos do grafo de programa.

Na próxima seção encontra-se a descrição dos algoritmos definidos para solucionar tanto o problema de posicionamento dos nós de um grafo de programa, quanto o roteamento dos seus arcos na geração automática dos grafos de programa.

3 Visualização de Grafos de Programa : Os Algoritmos Propostos

O desenho automático de grafos de programa envolve, basicamente, duas atividades : 1) determinar as coordenadas X e Y dos nós do grafo e 2) determinar o roteamento de cada um dos arcos da estrutura. Tendo em vista essa divisão os algoritmos foram desenvolvidos para trabalhar em duas etapas : posicionamento dos nós e roteamento dos arcos.

A abordagem que será apresentada faz uso de

[1] A POKE-TOOL [CHA91] é uma ferramenta que apóia a aplicação dos Critérios Potenciais Usos [MAL91] ao Teste Estrutural de Programas

um conceito, já conhecido na bibliografia, na determinação das coordenadas Y dos nós; esse conceito é denominado nível e é utilizado com algumas adaptações ao problema de posicionamento de grafos de programa. Para a determinação das coordenadas X é proposta a utilização do conceito de escopo que, como será visto, permite que os descendentes de um dado nó possam ser posicionados sem sobreposição; além disso, esse método evita o reposicionamento de nós, prática comum em outras abordagens.

3.1 Posicionamento dos Nós

O nível de cada nó, definido inicialmente para árvores [WET79], [REI81], fornece uma relação direta na determinação da coordenada Y, isto é, o nível n de um nó x é a sua coordenada Y a menos de ajustes de escala, espaçamento e deslocamento.

No caso de grafos de programa a definição do algoritmo para se determinar os níveis dos nós não pode ser aplicada diretamente, sofrendo algumas modificações, principalmente devido a dois problemas :

1. Num grafo de programa um nó pode ter mais de um antecessor; assim, não se pode definir que o nível de um nó é uma unidade maior que o nível de seu pai, sem se especificar qual é o pai em questão.

2. Uma estrutura que se destaca nos grafos de programa é a estrutura *case*; ela é iniciada por um nó x sendo que OUT(x) > 2 [2]; ou seja, o nó x tem mais de dois filhos e é chamado de nó de entrada do case. Nesta estrutura também são identificados mais dois componentes: os nós internos ao case, aqueles que se ligam diretamente ao nó de entrada e o nó de saída do case, aquele que fecha toda a estrutura. O segundo problema da determinação do nível advém do fato de que, em alguns casos, o nó de saída do case pode não estar em nível consecutivo ao de nenhum de seus antecessores.

No caso de grafos de programa podemos considerar que qualquer nó com mais de dois filhos é o nó inicial de uma estrutura *case* porque nenhuma outra estrutura tem essa característica. Essa restrição deve ser retirada no caso de se desejar aplicar os algoritmos em outras estruturas com relacionamento hierárquico.

A Figura 2-A mostra uma estrutura *case* comum com o nó de entrada, nós internos e o nó de saída identificados. Observa-se que existem arcos ligando os nós internos ao nó de saída; isso ocorre quando cada um dos blocos de comandos (representados aqui por nós), internos ao case, terminam com um comando "break". É importante lembrar que essas estruturas originam-se de programas com código fonte escrito em alguma linguagem de programação que possua um comando análogo ao comando "break" da linguagem C.

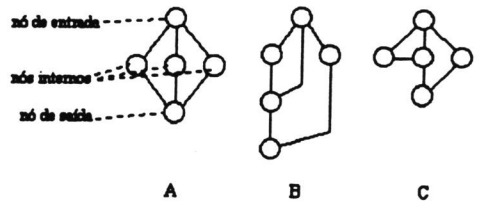

Figura 2: Problemas da Estrutura Case

Se um desses blocos internos não terminar com o comando "break", surgirá um arco entre blocos vizinhos na estrutura *case*. Assim, tem-se que dois nós internos ao *case* são pai e filho, respectivamente; esses nós, segundo a regra para calcular os níveis citada acima, deveriam ter níveis, no mínimo, consecutivos (Figura 2-B).

A determinação do nó de saída é feita da seguinte forma: é considerado nó de saída da estrutura *case* o filho mais à direita do nó inicial, se ele tiver apenas um filho; senão será considerado nó de saída o primeiro sucessor desse nó com no máximo 1 filho e apenas 1 pai.

A Figura 3-A mostra o desenho de uma estrutura case com o posicionamento do nó 8 incorreto. O nó 8 é o nó de saída da estrutura *case* iniciada no nó 1 e é filho direto dos nós 2 e 4 que têm nível 2; portanto, sem um tratamento especial para essa estrutura, o nível do nó 8 é igual a 3. O problema é que um dos ramos internos ao case, o iniciado no nó 3 e encerrado no nó 7, tem nível máximo igual a 4 (nível do nó 7); como esse detalhe não foi considerado, o desenho ficou desconfigurado.

Observa-se na Figura 3-B que o nível correto para o nó 8 é 5, um a mais que o nível do nó 7, apesar de esses dois nós não se ligarem diretamente. O nível do nó de saída deve ser uma unidade maior que o nível máximo dos nós internos ao case e seus descendentes.

Antes da apresentação do algoritmo completo de determinação dos níveis, deve-se definir alguns conjuntos e funções :

pai(n) : Retorna os pais do nó n.

conjunto_cases : Conjunto de todos os nós que pertençam a alguma estrutura case do grafo.

entrada_case[n] : Retorna o nó de entrada da estrutura case à qual pertença o nó n.

nós_internos_case : Conjunto de nós que se ligam diretamente aos nós de entrada das estruturas case, mas que não são nós de saída.

interior_case(n) : Conjunto de todos os nós que

[2] OUT(n) : Indica o número de filhos de n

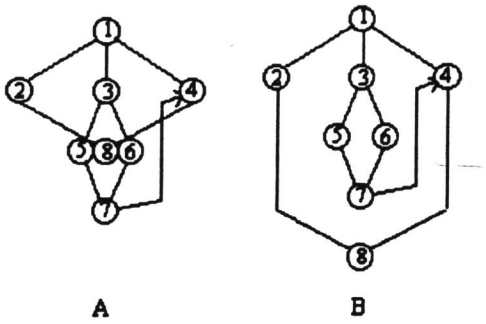

Figura 3: Posicionamento do Nó de Saída do Case

pertençam à estrutura case que têm como nó de entrada o nó n e que não são nem o próprio nó n nem o nó de saída dessa estrutura.

nó_de_saída[n] : Retorna o nó de saída da estrutura case da qual n é nó de entrada.

Assim, determina-se o nível de cada nó como segue.

Determinação do Nível de cada Nó: Regras

RN1 . nível[1] = 1.

RN2 . nível[n] = máx($\forall$ nível[pai(n)]) + 1 :- $n \notin$ conjunto_cases.

RN3 . nível[n] = nível[entrada_case[n]] + 1 :- $n \in$ conjunto_cases e n $\in$ nós_internos_case.

RN4 . nível[n] = máx(nível[interior_case (entrada_case[n])]) + 1 :- $n \in$ conjunto_cases e n = nó_de_saída (entrada_case[n]).

O algoritmo acima é basicamente dividido em três partes : 1) a semente da recursão, isto é, o nível do primeiro nó é definido como 1, Regra RN1; 2) o tratamento dos nós que não pertencem a nenhuma estrutura case, Regra RN2; 3) o tratamento dos nós que pertencem a alguma estrutura case: a Regra RN3 que trata os nós internos ao case e a Regra RN4, que trata especificamente do nó de saída.

Com isso encerra-se a determinação da coordenada Y dos nós de grafos de programa; os níveis dos nós são aplicados diretamente nessa determinação, a menos de parâmetros de escala, deslocamento e espaçamento. Passa-se agora ao estudo da determinação das coordenadas X dos nós.

Na determinação da coordenada X de cada nó surge um problema : como saber a priori qual espaço ocupará cada subgrafo descendente de um dado nó? Assim define-se o conceito de escopo e com a sua ajuda aplica-se o algoritmo de posicionamento mostrado a seguir.

Algumas funções necessárias na determinação do escopo são:

é_nó_folha(n) : Função que verifica se o nó n é nó folha. É considerado nó folha aquele nó que não possui descendentes no níveis consecutivo ao seu próprio nível.

filhos_no_nível_abaixo(n) : São os filhos do nó n que estão no nível consecutivo ao dele.

tem_mais_de_um_filho_abaixo(n) : Retorna verdadeiro se o nó n tiver mais de um filho no nível consecutivo ao dele.

no_de_pais(n) : Número de pais do nó n, no nível diretamente anterior ao dele.

filho_único(f,n) : Retorna verdadeiro se f é filho único de n.

tem_outros_pais(f,n) : Retorna verdadeiro se f tem outros pais além de n.

Cálculo do Escopo de cada Nó: Regras

RE1 . escopo(nó) = 1 :- é_nó_folha(nó)

RE2 . escopo(nó) = $\sum$escopo (filhos_no_nível_abaixo (nó)) :- tem_mais_de_um_filho_abaixo(nó)

RE3 . escopo(nó) = escopo(filho)/no_de_pais (filho) :- filho_único(filho,nó), tem_outros_pais (filho,nó)

A Regra RE1 é o fim da recursão para o cálculo do escopo, indica que o escopo de um nó folha vale 1. A Regra RE2 estabelece que o escopo de um nó seja a soma do escopo de seus filhos, a menos que esse filho tenha outros pais; nesse caso, o nó pai recebe apenas uma parcela do escopo do filho, o que é definido na Regra RE3.

A Figura 4 mostra o valor do escopo de cada nó para um programa real, denominado compress.c (apresentado em [VIL94], Apêndice A), calculado segundo o algoritmo acima. Observa-se que os arcos de ciclo não são considerados no cálculo do escopo; portanto, o nó 15 é tratado como nó folha e seu escopo é igual a 1.

Segundo a regra RE3, os nós 11 e 12 deveriam ter escopo = 0.5, mas esta variável é definida como sendo inteira; portanto, o escopo desses nós é igual a 1. Já o escopo do nó 10 é igual a 2, pois segundo a regra RE2 o seu escopo é a soma dos escopos de seus filhos. Verifica-se que este valor reflete exatamente o espaço necessário para o posicionamento de seus filhos, um espaço para o nó 11 e um espaço para o nó 12, exatamente o valor de seus escopos. O mesmo ocorre, por exemplo, para o nó 3; o valor do seu escopo é 5, sendo que desses 5 espaços, 2 são para o nó 4 e 3 são para o nó 8.

A aplicação pura e simples do escopo não resolve todos os casos; por exemplo, o escopo do nó 1 é igual a 6, mas este nó só tem um filho, o nó 2, que não

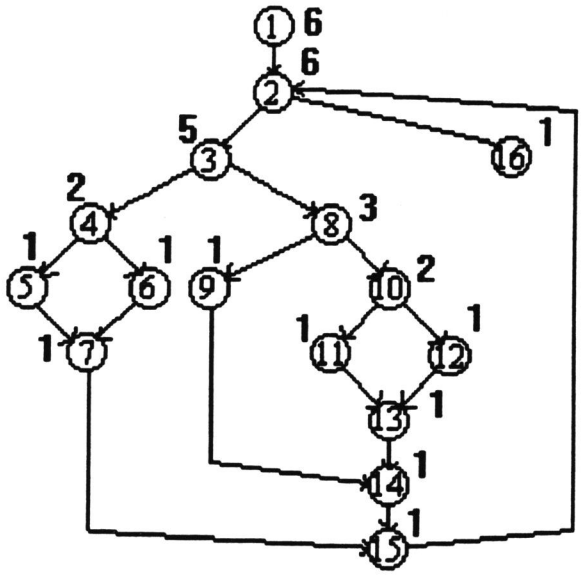

Figura 4: Escopo dos Nós de compress.c [VIL94]

precisa ser deslocado dentro do seu escopo, pois não tem nenhum vizinho.

O posicionamento dos nós é feito de forma a criar uma malha de posições relativas, onde a posição do nó 1 é igual a zero, as posições dos nós à esquerda recebem valores negativos e as dos nós à direita recebem valores positivos.

O algoritmo de posicionamento dos nós é definido a seguir; ele leva em conta o cálculo dos níveis e dos escopos apresentados anteriormente.

Considere as seguintes definições :

não_tem_outros_pais_no_nível_de_cima(n,p) : Retorna verdadeiro se o nó n tiver apenas um pai, o nó p, no nível diretamente acima ao dele.

média_posição_pais_no_nível_de_cima(n) : Equivale à média aritmética das coordenadas X dos pais do nó n que se encontram no nível acima ao dele.

tem_outros_pais_no_nível_de_cima(n,p) : Retorna verdadeiro se o nó n tiver outros pais, além de p, no nível diretamente acima ao dele.

irmãos_esq_no_mesmo_nível(n) : Retorna os irmãos à esquerda de n, que estão no mesmo nível que ele.

é_saída_case(n) : Retorna verdadeiro se n for nó de saída de alguma estrutura case do grafo.

Cálculo do Posicionamento dos Nós: Regras

RP1 . posição[1] = 0

RP2 . posição[nó] = posição[pai] :- filho_único(nó), não_tem_outros_pais_no_nível_de_cima(nó,pai)

RP3 . posiçao [nó] = média_posição_pais _no_ nível_de_cima(nó) :- filho_único (nó), tem_outros_pais_no_nível_de_cima (nó,pai)

RP4 . posição[nó] = posição[pai] - escopo[pai] / 2 + $\sum$(escopo(irmãos_esq_no_mesmo_nível(nó))) + escopo[nó]/2 :- não_filho_único(nó)

RP5 . posição[nó] = posição [entrada_case (nó)] :- é_saída_case(nó)

A Regra RP2 define a posição de um nó como sendo a mesma posição de seu pai, quando o nó é filho único do pai e ele não tem outros pais no nível anterior ao dele. Este é o caso do nó 14 da Figura 4; ele é filho único do nó 13 e, apesar de ter outro pai, o nó 9, este não está no mesmo nível do nó 13.

Quando um nó tem mais de um pai no nível de cima, a sua posição deve ser a média das posições desses pais, conforme a Regra RP3; esse é o caso dos nós 7 e 13 da Figura 4.

Se um nó tem vários filhos no nível consecutivo ao dele, esses nós são espalhados no escopo do pai, proporcionalmente ao escopo de cada um dos filhos.

A aplicação da Regra RP4 é tal que: a posição de um nó é dada pela soma da posição inicial do escopo do seu pai com a sua própria posição dentro desse escopo. O escopo do nó pai deve ficar centralizado em relação à posição desse nó; assim, a posição inicial do escopo é a posição do nó pai menos a metade do seu escopo. O posicionamento do nó dentro do escopo do pai é calculado somando-se os escopos dos seus irmãos à esquerda, com metade do seu próprio escopo; assim, o nó ficará centralizado no espaço que lhe cabe.

A Regra RP5 diz respeito às estruturas case; o nó de saída dessas estruturas deve receber a mesma posição X do nó de entrada.

Com isso tem-se a determinação do posicionamento dos nós concluída; na próxima seção encontra-se o algoritmo que trata o roteamento dos arcos de um grafo de programa.

3.2 Roteamento dos Arcos

Alguns algoritmos de roteamento de arcos geram diagramas com arcos complexos, utilizando-se de curvas desenhadas com "splines" e outros recursos na tentativa de minimizar os cruzamentos, como em [GAN88] e [ROW87].

No caso de grafos de programa, o próprio posicionamento dos nós definido anteriormente gera um diagrama com colunas vagas entre os nós; essas colunas podem ser aproveitadas como opções de caminhos no roteamento dos arcos.

Definições.

Alguns conceitos e variáveis devem ser definidos antes de apresentar-se o algoritmo de roteamento de arcos :

grade de posicionamento : é a máscara de posições livres e ocupadas gerada pelo posicionamento dos nós e utilizada como guia no roteamento dos arcos.

opções do arco : são as possíveis posições (ou os possíveis caminhos) que podem ser ocupados pelos arcos na grade de posicionamento.

delta_nível : é a distância em níveis que separa as duas extremidades de cada arco; é usada como parâmetro de classificação e ordenação dos arcos.

arco interno e arco externo : os arcos internos são aqueles que estão mais próximos do centro da estrutura do grafo (dado pela posição X do nó 1) e os arcos externos são os que estão mais distantes.

Classificação dos Arcos.

Os arcos são classificados de acordo com três características principais : tamanho, sentido e número de opções.

Tipos de Arcos Segundo o Tamanho :

1. Arcos Curtos : São aqueles que ligam nós em níveis consecutivos ou iguais, ou seja, $0 \leq$ delta_nível ≤ 1.

2. Arcos Longos : São aqueles que ligam nós em níveis não consecutivos nem iguais, ou seja, delta_nível > 1.

Tipos de Arcos Segundo o Sentido :

1. Arcos Diretos : Para um arco (n,m) temos $n < m$.

2. Arcos de Ciclo : Para um arco (n,m) temos $n > m$.

No caso de grafos de programa não existe a possibilidade de n=m.

Tipos de Arcos Segundo o Número de Opções :

1. Arcos com Opção Única : Possuem apenas um caminho possível.

2. Arcos com Opções Múltiplas : Possuem vários caminhos possíveis.

3. Arcos sem Opção : Não possuem nenhum caminho possível.

Estratégia da Abordagem.

A estratégia adotada para o roteamento de um arco leva em conta algumas de suas características ligadas principalmente ao seu tipo.

1- Arcos diretos e curtos têm posicionamento trivial; para esses arcos não é necessário calcular o roteamento nem, tão pouco, guardar suas posições; eles são tratados diretamente pelo módulo de desenho, constituindo-se basicamente de uma linha reta entre as duas circunferências que representam os nós.

O posicionamento dos nós gera uma grade de posicionamento ocupada por alguns nós, mas com muitas colunas vazias; essas colunas serão usadas pelo algoritmo de roteamento dos arcos, sendo que a posição de um arco corresponderá à coluna que ele ocupará na grade de posicionamento.

2- Arcos de ciclo e curtos têm também posicionamento trivial, mas, mesmo assim, têm a sua posição guardada numa estrutura de dados (como os outros arcos, à exceção dos arcos diretos e curtos); eles recebem a coluna correspondente ao nó inicial do arco.

A Figura 5 mostra exemplos dos dois conjuntos de arcos anteriores posicionados numa grade de posicionamento ocupado por alguns nós.

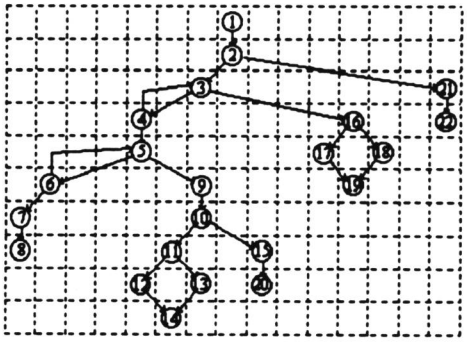

Figura 5: Grade de Posicionamento Ocupada por Nós e Alguns Arcos

Para os outros arcos devem ser encontradas opções de posicionamento e a cada arco deve ser atribuída a melhor dessas opções.

Ordenação dos Arcos.

Uma ordenação dos arcos deve ser feita, segundo a ordem crescente de delta_nível; arcos com menor delta_nível tendem a ser arcos internos e, portanto, devem ser posicionados antes dos arcos externos que, normalmente, têm delta_nível maior. Assim, roteando os arcos em ordem crescente de delta_nível melhora-se a possibilidade de posicioná-los sem cruzamentos.

Como podem existir vários arcos com o mesmo delta_nível eles ficam divididos em blocos de arcos de mesmo delta_nível.

A estratégia do algoritmo é tratar esses blocos separadamente e só passar para o próximo bloco quando o anterior estiver completamente posicionado.

Determinação das Opções.

As opções de cada arco são calculadas investigando-se a grade de posicionamento levando em conta a posição dos nós e as posições dos arcos já posicionados.

Determinação das Opções de cada Nó: Regras

RO1 . Os filhos do nó de menor nível definem os caminhos possíveis iniciais.

RO2 . Os subgrafos vizinhos ao nó de menor nível restringem as possibilidades.

RO3 . A partir do nó de menor nível, percorre-se o grafo redefinindo-se as opções.

Segundo a Regra RO1, as opções iniciais de roteamento de um arco são definidas pelo nó que estiver acima na estrutura do grafo, não importando se este nó é o nó inicial ou o nó final do arco. Observa-se na Figura 6 que três opções iniciais foram definidas: a opção pela esquerda, a opção central e a opção pela direita.

Na Figura 7 observa-se a aplicação da Regra RO2; o subgrafo à direita anulou a opção de posicionamento pela direita.

Com as opções que sobraram aplica-se a Regra RO3.

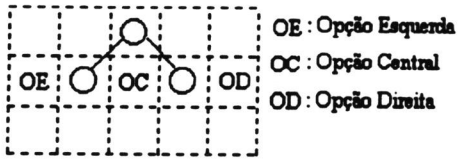

Figura 6: Determinação de Opções Possíveis

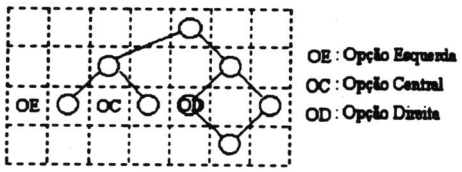

Figura 7: Os Vizinhos Restringem as Opções

Assim, as opções de cada nó são calculadas e armazenadas; essas opções devem ser recalculadas a cada vez que um arco tenha o seu roteamento definido.

Estratégia de Roteamento para Cada Uma das Opções.

Arcos com opção única : Deve ser alocado ao arco a opção encontrada, mesmo que essa atribuição faça com que outro arco fique sem nenhuma opção.

Arcos com opções múltiplas : Deve-se escolher uma opção que não faça outro arco ficar sem opção nenhuma; dentre as opções que se encaixem nessa ressalva pode-se ainda aplicar heurísticas como: escolher sempre a opção mais à esquerda (direita), ou escolher aquela mais próxima do nó inicial (final), etc.

Arcos sem opção: Nesse caso o cruzamento não poderá ser evitado, atribui-se a esse arco uma posição padrão, que pode ser a posição do nó inicial ou a posição do nó final do arco.

Apresentação do Algoritmo Completo.

O algoritmo de roteamento de arcos pode ser dividido em quatro partes principais : o pré-processamento, o roteamento dos arcos triviais (curtos), a verificação inicial das opções dos arcos e o laço de definição do roteamento de todos os arcos longos.

No pré-processamento é montada a lista de arcos que serão tratados no algoritmo; essa lista é montada a partir da estrutura de adjacência gerada a partir do arquivo que armazena o grafo de programa em forma de texto; em seguida, o delta_nível de cada arco é calculado e os arcos são ordenados de acordo com o delta_nível.

Os arcos com roteamento trivial são aqueles com delta_nível igual a 1 (arcos curtos); desses os que são arcos de ciclos recebem a posiçao do nó inicial (coordenada da coluna que será ocupada por esse arco) e os que são arcos diretos serão tratados diretamente pelo módulo de desenho.

Segundo a regra para determinação das opções dos arcos são achadas as opções iniciais de todos os arcos do grafo que ainda não tenham sido posicionados.

Por fim, o laço de definição de roteamento para os arcos longos funciona da seguinte forma : a partir desse ponto o algoritmo trabalha em blocos de arcos que têm o mesmo delta_nível, passando para o próximo bloco - em ordem crescente - apenas quando todos os arcos daquele bloco estiverem posicionados; levando isso em conta, tem-se que os arcos com opção única são posicionados e a cada arco posicionado verificam-se novamente as opções de todos os arcos, até que todos os arcos com opção única tenham sido posicionados; em seguida, posicionam-se os arcos com opções múltiplas e, por fim, os sem opção.

Características dos Algoritmos.

O algoritmo apresentado foi desenvolvido para desenhar grafos de programa; assim, foram levadas em conta algumas restrições :

1. O algoritmo gera um posicionamento padronizado de estruturas como o IF-THEN-ELSE. Esse posicionamento é mantido mesmo quando acarretar o cruzamento de arcos.

2. A estrutura *case* também tem um posicionamento padronizado; ou seja, os filhos diretos do nó inicial sempre são posicionados no nível consecutivo ao dele, mesmo que haja um relacionamento hierárquico entre eles (veja Figura 2-C). Outra restrição é a própria determinação dessa estrutura que considera qualquer nó com mais de dois filhos um nó de entrada de case.

3. Devido à necessidade de se rotear os arcos dos grafos de programa, o posicionamento gerado para os

nós pode resultar numa disposição não mínima, justamente para que as folgas no posicionamento sejam usadas como opções de roteamento para os arcos.

4. É considerada a existência de apenas um nó de entrada, consideração razoável para grafos de programa.

Outras Abordagens.

Rowe et. al. [ROW87] definem um algoritmo para posicionar grafos dirigidos que também utiliza o cálculo dos níveis dos nós para determinar as coordenadas Y. Para determinar as coordenadas X é utilizado um método de tentativa e falha tentando minimizar o número de cruzamentos. A estratégia de posicionamento de arcos usa arcos quebrados, com pontos de inflexão a cada nível do grafo; essa abordagem pode dificultar o entendimento do diagrama gerado.

Wetherell e Shannon [WET79] e Reingold e Tilford [REI81] propõem algoritmos para posicionar árvores em que utilizam o nível do nó para definir as coordenadas Y; já as coordenadas X são definidas de acordo com a ordem em que os nós aparecem durante uma busca na árvore. Os algoritmos variam de acordo com o método de busca utilizado.

Price et. al. [PRI87] apresentam um algoritmo para desenhar grafos de programa em que é utilizada a definição de nível para determinar as coordenadas Y, mas não é feita nenhuma consideração a respeito de adaptações na determinação dos níveis para o caso específico de grafos de programa. As coordenadas X são definidas através de alocação de deslocamento padrão, que vai sendo reajustado de acordo com a necessidade.

4 Conclusões

Com as restrições citadas anteriormente os algoritmos apresentados funcionam bem para grafos de programas tendo como principais características :

Muitos algoritmos de posicionamento de nós de grafos de programa utilizam uma estratégia de redefinir o posicionamento de nós já posicionados, à medida que é requisitado mais espaço para os subgrafos desses nós; essa abordagem desperdiça tempo pois é necessário reposicionar todos os antecessores de um nó que teve a sua posição alterada; na abordagem apresentada neste trabalho, esse problema é eliminado através do uso do escopo; assim, pode-se determinar diretamente as posições dos nós sem haver repetição de cálculos.

Os grafos gerados tendem a ser equilibrados pois, devido à estratégia para o cálculo e uso do escopo, os subgrafos mais densos tendem a ficar mais internos à estrutura do grafo.

O algoritmo pode ser aplicado para outras estruturas como : árvores, grafos planares dirigidos, diagramas de modelo entidade-relacionamento, diagramas de relacionamento de classes de objetos; necessita, para tanto, de modificações simples no interpretador do arquivo de entrada e na eliminação do tratamento especial de estruturas como o case.

A abordagem adotada na construção dos algoritmos permite gerar um posicionamento relativo dos elementos que constituem um grafo de programa; esse posicionamento relativo pode, então, ser mapeado no dispositivo de saída, sob diversas formas estéticas diferentes, possibilitando que os algoritmos sejam utilizados numa ampla gama de aplicações.

Uma implementação dos algoritmos foi utilizada numa ferramenta denominada ViewGraph [VIL94]. O objetivo dessa ferramenta é auxiliar o teste e a depuração de programas através da visualização gráfica de informações de teste. A ViewGraph deverá ser integrada na ferramenta de teste POKE-TOOL.

5 Referências Bibliográficas

[CHA91] Chaim, M.L., POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados. *Dissertação de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, Abril 1991.

[GAN88] Gansner, E. R., North, S. C., Vo, K. P., DAG - A Program that Draws Directed Graphs. *Software Pratice and Experience*, Vol.18, No11, pp.1047-1062, Nov. 1988.

[KAM89] Kamada, T., Kawai, S., An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31, (1), 7-15 (1989).

[MAL91] Maldonado, J.C., Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. *Tese de Doutorado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, Julho 1991.

[PRI87] Price, A. M., Garcia, F., Purper, C., Visualizando o Fluxo de Controle de Programas. *Anais I Simpósio Brasileiro de Engenharia de Software*, pp.1-11, Petrópolis, RJ, 22-23 Outubro 1987.

[REI81] Reingold, E., Tilford, J., Tidier Drawings of Trees. *IEEE Trans. Soft. Eng.*, Vol. SE-7, No.2, pp.223-228, 1981.

[ROW87] Row, L.A., Davis, M., Messinger, E., et al., A Browser for Directed Graphs. *Software-Practice and Experience*, Vol.17, No.1, pp.61-76, Jan. 1987.

[VIL94] Vilela, P.R.S., Uma Ferramenta para Auxílio Visual ao Teste e Depuração de Programas. *Dissertação de Mestrado*, DCA /FEE /UNICAMP - Campinas, SP, Brasil, Março 1994.

[WET79] Wetherell, C., Shannon, A., Tidy Drawing of Trees. *IEEE Trans. Soft. Eng.*, Vol.SE-5, pp.514-520, 1979.