UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

UMA FERRAMENTA PARA AUXÍLIO VISUAL AO TESTE E DEPURAÇÃO DE PROGRAMAS

Plínio Roberto Souza Vilela

Orientador :

Prof. Dr. Mario Jino - DCA/FEE/UNICAMP

Co-orientador :

Prof. Dr. José Carlos Maldonado - ICMSC/USP

Dissertação apresentada à Faculdade de Engenharia Elétrica da UNICAMP, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica

CAMPINAS, SÃO PAULO

MARÇO, 1994

À minha mãe Teresa e à minha irmã Flávia

Goto, n. :

A programming tool that exists to allow
structured programmers to complain about
unstructured programmers.

-- Ray Simard.

AGRADECIMENTOS

Primeiro de tudo e antes de mais nada gostaria de agradecer ao meu pai Maury e à minha mãe Teresa, por motivos que eu considero óbvios. Segundo de tudo e antes de algumas coisas gostaria de agradecer à minha irmã Flávia por ter me aturado esses 23 anos (quero dizer 20 anos, pois nos três primeiros ela ainda não era nascida!).

Ao meu amigo José Carlos Maldonado (e por um acaso do destino meu orientador) agradeço : a confiança, a amizade, a bebedeira e principalmente a extrema competência em conduzir não só o meu trabalho, mas o de todo um grupo de pesquisa (quero sê qui nem ocê quandu eu crecê).

Ao Mario Jino, meu orientador (pô cara! você tem dois orientadores?... -pois é, quem pode, pode!) pela presença nos momentos difíceis (só nós sabemos quantos foram) e por ser essa pessoal genial.

Ao Marcos Chaim (Chainsão) por ser o meu __-__-orientador (você achou que eu iria escrever isso aqui cara!) e por ter tido paciência de me explicar XXX vezes os conceitos envolvidos na definição dos Critérios Potenciais Usos [MAL91].

À Mitie pela preocupação e pelas homeopatias.

Ao Eiji por ter feito todo o trabalho por mim (tô brincando), por falar nisso eu já assinei a sua carta de libertação.

Não poderia me esquecer do pessoal do grupo de teste DCA/FEE/UNICAMP : Silvia, Rubão, Plínio (que nome buuunito) valeu pessoal.

Ao Alexandre Magalhães (Xandão 94 - nova metodologia) pela horas de depuração, que eu já paguei com dicas de redação (logo eu!). E a todo pessoal do laboratório (se eu me esquecer de alguém por favor não fique chateado, não foi de propósito) : Nelson (o dinossáuro do volei), Elton (o rei da mulherada), Perliiiinha (nada a declarar), Assfalk (vulgo Marco Antonio d'Oliveira de traz dos Montes), Okamura (também conhecido por Fabio, só não se sabe onde), Agnus (ele não deve estar lendo esses agradecimentos agora, pois, provavelmente, ainda está almoçando).

Ao pessoal da USP-São Carlos (C_ _ _A Federal!!! : $CA_ASO..$, $CA_ASO..$) Pastel, Taisa, Titi e Cybelle; vocês que me acompanham desde o tempo de graduação agora podem ter certeza de que eu não mudo nunca. Muita coisa eu devo a esse pessoal, pois além de serem ótimos companheiros são também excelentes conselheiros sentimentais.

Por último de tudo agradeço a mim mesmo; pois, sem a minha participação intensiva, nada disso teria sido possível.

RESUMO

Os principais aspectos da especificação da ViewGraph, uma ferramenta cujo propósito é auxiliar a atividade de teste e depuração de programas, através da visualização de informações de teste fornecidas pela POKE-TOOL [CHA91], são apresentados. Os principais pontos relacionados do Teste Estrutural Baseado em Análise de Fluxo de Dados suportado pela POKE-TOOL, são também apresentados.

Os principais algoritmos utilizados na ferramenta ViewGraph são aqueles que tratam da geração da disposição gráfica dos grafos de programa; sua descrição é mostrada em detalhes nessa dissertação. Uma avaliação empírica dos algoritmos foi realizada e os resultados são apresentados.

As características bem como os principais problemas encontrados na implementação de um subconjunto da especificação da ViewGraph são também discutidos.

ABSTRACT

The specification and main features are presented of ViewGraph, a tool designed to aid in testing and debugging tasks by providing the visualization of test information produced by POKE-TOOL [CHA91]. The main points on Structural Testing based on Data Flow Analysis, supported by POKE-TOOL, are also presented.

The most important algorithms in ViewGraph are the ones which deal with the visualization of program graphs; their detailed description is shown. An empirical evaluation of the algorithms was conducted and the results are presented.

The characteristics as well as the major problems of the implementation of a subset of the specification of ViewGraph are also discussed.

CONTEÚDO

LISTA DE FIGURAS

Página

LISTA DE TABELAS

Página

CAPÍTULO 1

INTRODUÇÃO

Neste capítulo são discutidos o contexto, as motivações e a importância do tema da Dissertação - Visualização de Informações de Teste e Grafos de Programa - dentro da área de Engenharia de Software, em especial no Teste Estrutural e na Depuração de Software. Os objetivos principais a serem atingidos e a organização da Dissertação também são apresentados.

1.1 - Contexto

O software tem se tornado o centro de muitas atividades complexas desenvolvidas atualmente em nossa sociedade; portanto, é natural que se utilizem técnicas, métodos e ferramentas especializadas e eficientes para a sua produção. Em resposta a esta necessidade desenvolve-se a *Engenharia de Software* que aplica muitos dos métodos da engenharia tradicional ao desenvolvimento de produtos de software.

No processo de desenvolvimento de software todos os erros são erros humanos e, apesar da introdução de melhores métodos de desenvolvimento, melhores ferramentas de suporte e treinamento de pessoal, erros permanecem presentes nos diversos produtos de software produzidos e liberados [HOW87]; assim, a atividade de teste continuará a ter um papel importante no desenvolvimento de software.

Duas abordagens para a definição dos objetivos do teste podem ser caracterizadas [COW88] : teste como um processo destrutivo, onde o objetivo é encontrar falhas (indicativo da presença de erros); e teste como um processo construtivo, onde o objetivo é demonstrar que o software não tem erros.

Entende-se, no contexto desta dissertação, que o principal objetivo do teste de software é revelar a presença de erros no programa em teste : em outras palavras, o objetivo do teste de software é refutar a afirmação de que o programa está correto; vários outros autores adotam essa mesma abordagem [MYE79], [PRE87], [HAL91], [COW88]. Frankl acrescenta : "o objetivo do teste de software é detectar a presença de erros ou, não atingindo esse objetivo, aumentar a confiança de que o programa está correto" [FRA87]. Nesse contexto um teste bem sucedido é aquele que revela a presença de um erro; um bom caso de teste é aquele que tem uma alta probabilidade de encontrar um erro ainda não descoberto.

De uma maneira geral, o teste de software, envolve as seguintes atividades: *planejamento, projeto de casos de teste, execução de casos de teste* e *análise (avaliação) dos resultados dos testes*. O planejamento da atividade de teste deve fazer parte do planejamento global do sistema, culminando num *Plano de Teste* que constitui um documento crucial no ciclo de desenvolvimento de software.

O projeto de casos de teste concentra-se em um conjunto de técnicas, critérios e métodos para elaborar os casos de teste. Esses métodos, critérios e técnicas fornecem ao projetista de software uma abordagem sistemática aos testes; além disto, constituem um mecanismo que pode auxiliar a garantir a completude dos testes e uma maior probabilidade de revelar defeitos no software. Em geral, deve-se projetar casos de teste que tenham a maior probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço.

O projeto de casos de teste aborda duas questões chaves de pesquisa na área de teste de software: "Como os dados de teste devem ser selecionados?" e "Como se pode dizer se um programa *P* foi testado suficientemente". Frankl [FRA87] define *método de seleção de dados de teste* como um procedimento para escolher *casos de teste*; e *critério de adequação dos dados de teste* (conjunto de dados de teste) como um predicado usado para avaliar o(s) dado(s) de teste.

Os métodos utilizados para testar o software são geralmente baseados em duas técnicas : funcional e estrutural. A Técnica Funcional procura selecionar casos de teste apoiada na especificação funcional do software.

A Técnica Estrutural de Teste é baseada no conhecimento da estrutura interna da implementação, e o objetivo é caracterizar um conjunto de componentes elementares de um programa que devem ser exercitados pelo conjunto de casos de teste. Independentemente das desvantagens dos critérios e métodos do teste estrutural [NTA88], [RAP85], estes devem ser vistos como complementares aos funcionais, uma vez que cobrem essencialmente classes distintas de erros [MYE79], [PRE87].

Informalmente, a estrutura interna é representada por um *grafo de fluxo de controle* [CHU87], [MYE79]. A estrutura de controle ou grafo de controle é simplesmente uma notação para representar o fluxo de controle lógico de uma unidade de programa e consiste basicamente em um grafo dirigido [MCC76], doravante chamado *grafo de programa*.

Nos grafos de programa cada nó representa uma seqüência de comandos que são sempre executados como um bloco de comandos, e cada arco representa uma possível transferência de controle entre esses blocos; um caminho de um programa é representado como uma seqüência de nós.

Com o grafo de programa podemos identificar a estrutura interna do programa em teste, seus principais comandos, caminhos (executados ou não pelos casos de teste), associações, medidas de complexidade, etc. Além de poder identificar imediatamente anomalias como código morto e falta de estruturação. A Figura 1.1 é um exemplo de grafo de programa com seus principais comandos ressaltados; esse grafo foi extraído do programa *entab.c* apresentado no Apêndice A.

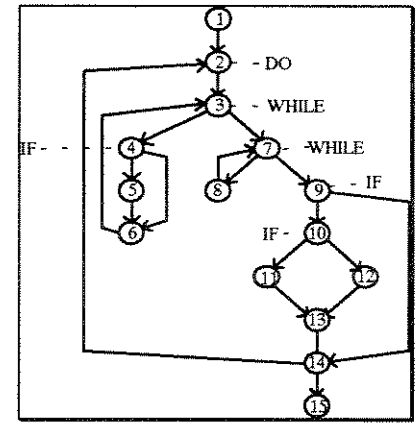


Figura 1.1. Grafo de Programa e Correspondentes Comandos

Várias classes de critérios de adequação foram definidas [RAP82], [WEY86], [FRA88], entre elas os *Critérios Potenciais Usos* definidos por Maldonado, Chaim e Jino [MAL88a], [MAL88b], [MAL91]; esses critérios são baseados no conceito *potencial uso*; os Critérios Potenciais Usos são critérios de teste estrutural, baseados na análise de fluxo de dados e constituem-se, fundamentalmente, em variações da família de critérios apresentada por Rapps e Weyuker [RAP82], [RAP85]; são denominados : critérios *todos-potenciais-du-caminhos*, *todos-potenciais-usos*, *todos-potenciais-usos/du*. Associações são requeridas independentemente da ocorrência explícita de uma referência a uma determinada definição; se um uso pode existir - um *potencial uso* - a potencial associação é requerida.

À medida que os sistemas de software cresceram em tamanho e complexidade, o esforço requerido para testar esses sistemas tem crescido muito além das expectativas, implicando altos custos e produtos com baixa confiabilidade. Tem-se verificado que embora gaste-se, em geral, até 50% do orçamento para desenvolvimento do software em atividades de teste, um número significativo de defeitos permanece sem ser identificado nos softwares liberados; esses defeitos normalmente têm um impacto grande na operação normal do sistema.

Essa situação estimulou o desenvolvimento de ferramentas automáticas para auxiliar na produção de testes efetivos e na análise dos resultados dos testes. O uso de uma ferramenta de software para auxílio ao teste pode ser vinculado a um critério de teste de duas maneiras. A ferramenta de software pode utilizar o critério de teste como um guia para a geração de casos de teste que satisfaçam o critério; outra possibilidade é a utilização do critério para a análise de cobertura de um conjunto de casos de teste; isto é, verificar se os casos de teste aplicados preencheram os requisitos de teste do critério.

Foi desenvolvido pelo grupo de teste do DCA/FEE/UNICAMP uma ferramenta de suporte à aplicação dos Critérios Potenciais Usos ao Teste Estrutural de Programas, a POKE-TOOL [CHA91]. A ferramenta POKE-TOOL está enquadrada na segunda abordagem das ferramentas de teste estrutural de programas. Ela fornece ao usuário os elementos requeridos para satisfazer os Critérios Potenciais Usos e é capaz de verificar qual foi a cobertura do conjunto de casos de teste fornecido pelo usuário em relação aos elementos requeridos.

O protótipo atual da POKE-TOOL fornece informações como: o grafo de programa (mas não de forma gráfica), o conjunto de arcos primitivos, o conjunto de variáveis definidas em cada nó, o programa fonte instrumentado (ou seja, com chamadas de rotinas incluídas que constituem o "trace" do programa) e medidas de adequação em relação a alguns critérios de teste.

A atual interface implementada na POKE-TOOL é simples e não dispõe de recursos gráficos que permitam visualizar, de forma amigável ao usuário, as informações por ela geradas. A ferramenta também não possui recursos para ajudar na geração de casos de teste ou na depuração de programas.

Outra atividade onde a visualização é relevante é a depuração, um processo distinto do teste, mas que sempre ocorre em conseqüência dele (veja Figura 1.2). A psicologia humana, melhor do que fatores tecnológicos, explica as dificuldades da depuração. Pressman [PRE87] coloca algumas características dos defeitos, que nos dão pistas para entender melhor essa dificuldade; por exemplo : o sintoma e a causa podem estar geograficamente distantes; o sintoma pode desaparecer temporariamente quando um outro defeito é corrigido; o sintoma pode ser causado por erros humanos não facilmente rastreados.

Algumas das perguntas que devem ser feitas durante o processo de depuração são : O que ocorreu?; Quando?; Onde?; Com qual extensão?; etc. Nesse processo, a visualização de grafos de programa viria de encontro à pergunta - Onde? - com ela podemos analisar o caminho percorrido dentro da estrutura do programa e identificar a posição na qual esse caminho difere do que, a princípio, deveria ser percorrido. Com o ponto onde o sintoma aparece identificado, podemos rastrear recursivamente o caminho percorrido buscando encontrar o defeito que originou o sintoma detectado.

A visualização de grafos de programa e de informações relativas ao código do programa pode auxiliar na manutenção de software, colaborando na avaliação do projeto inicial e no

desenvolvimento do plano de manutenção. A abordagem sistemática à atividade de teste e a facilidade de produzir documentação apropriada dessa atividade tem aplicação também no teste de regressão, facilitando e incentivando essa atividade.

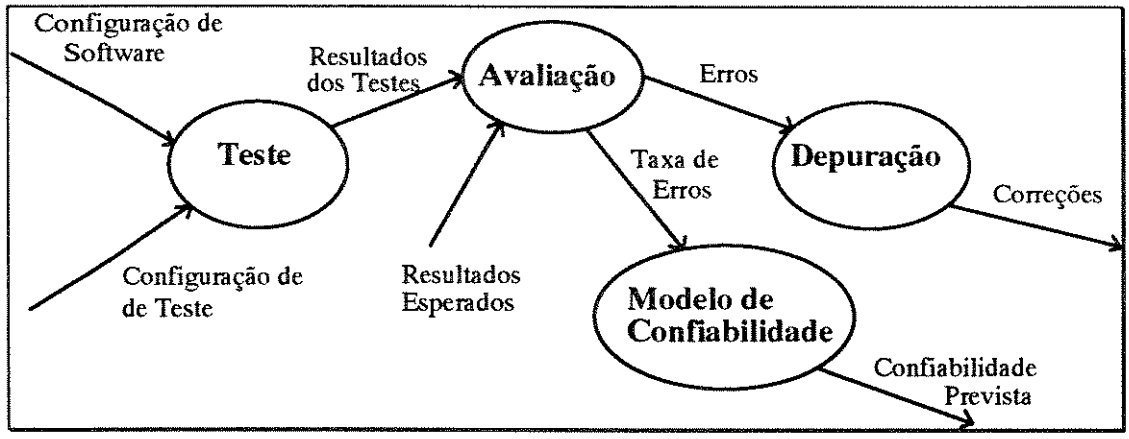


Figura 1.2. Fluxo de Informações de Teste [PRE87].

O desenvolvimento e a integração da visualização gráfica de informações relacionadas à atividade de teste têm relevante importância no sucesso dessa atividade. Motiva-se assim a utilização das informações geradas pela POKE-TOOL numa ferramenta com interface gráfica bem definida e com recursos para interação com essas informações.

1.2 - Motivação

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[1].

O desenho manual de grafos de programa, como o da Figura 1.1, 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 da sua elaboração manual motivam a construção de ferramentas que automatizem essa atividade.

Com uma ferramenta de visualização totalmente integrada com ferramentas de teste como a POKE-TOOL, pode-se criar um ambiente poderoso de teste, depuração e manutenção de programas. Tomando-se como base o grafo de programa, pode-se, através da interferência do usuário (testador), possibilitar o acesso a outras informações de teste geradas pela POKE-TOOL.

A visualização de grafos de programa pode também ser utilizada a partir da Fase de Projeto de um sistema. Mostrar o fluxo de controle do programa, antes mesmo de implementá-lo, possibilita que defeitos sejam descobertos prematuramente. Sabe-se que quanto antes um

[1]Aqui, um conjunto efetivo de casos de teste é aquele que permite uma boa taxa de cobertura para um dado requisito de teste.

defeito for descoberto no ciclo de vida de um sistema, menor é o custo de correção e maiores são as chances de concluí-lo com sucesso.

Além disso, o processo de definição dos algoritmos, que automatizam a visualização de grafos de programa, pode ser abstraído, no intuito de definir algoritmos mais gerais para visualizar outras estruturas com relacionamento hierárquico. A visualização dessas novas estruturas tem aplicação em diversas áreas de trabalho. Por exemplo : no projeto, implementação, teste e manutenção de sistemas computacionais, CAD/CAM, planejamento e controle de projetos e em banco de dados, entre outras.

Projetos são normalmente compostos por diversos diagramas que podem ser visualizados graficamente. Com esses diagramas impressos e catalogados pode-se gerar um meio eficiente de se produzir documentação atualizada do projeto. A documentação gerada pode servir tanto para nortear futuras modificações quanto para auxiliar na comunicação entre o usuário e o projetista, através de uma linguagem comum a ambos - a linguagem gráfica. Essa abordagem, segundo Tamassia [TAM88], facilita o trabalho de definição do sistema.

Essas aplicações motivam a definição de uma ferramenta de visualização de informações de teste e algoritmos para visualização de grafos de programa que possam, com algumas modificações ou restrições, adaptar-se a outros tipos de estruturas com relacionamento hierárquico.

1.3 - Objetivos

O objetivo da dissertação é especificar uma ferramenta cujo propósito é auxiliar a atividade de teste e depuração de programas através da visualização de informações de teste, geradas pela POKE-TOOL. A parte principal dessa ferramenta é a visualização dos grafos de programa; assim, algoritmos que resolvam esse problema devem ser desenvolvidos e implementados.

A implementação dos algoritmos deve poder ser expandida com o objetivo de suportar a visualização de informações relativas a um programa em teste, tomando como entrada arquivos gerados pela POKE-TOOL durante uma (ou várias) sessões de teste sobre um programa. Possibilidades de modificações com o objetivo de utilizar-se os algoritmos definidos para grafos de programa em outras aplicações também devem ser exploradas.

1.4 - Organização do Trabalho

Neste capítulo situou-se o contexto desta dissertação e caracterizou-se sua relevância dentro do Teste Estrutural de Software e da Engenharia de Software; foram também definidos os objetivos principais da Dissertação.

No Capítulo 2 - Revisão Bibliográfica é colocada uma introdução à área de visualização gráfica de estruturas hierárquicas através da apresentação de conceitos básicos, terminologias, aspectos estéticos e algoritmos, definidos em publicações sobre posicionamento de árvores, grafos, grafos de programa e outros.

No Capítulo 3 é fornecida uma descrição sucinta de Teste Estrutural Baseado em Análise de Fluxo de Dados; além da apresentação das características principais da POKE-TOOL e a especificação de uma ferramenta de visualização de informações relativas ao teste de programas.

Problemas e características principais relacionados à visualização de grafos de programa, além das soluções algorítmicas definidas para essa visualização, encontram-se no Capítulo 4 - Visualização de Grafos de Programas.

No Capítulo 5 - Características de Implementação - encontram-se a apresentação dos principais aspectos da implementação dos algoritmos definidos para solucionar o problema de posicionamento de grafos de programa, bem como a descrição da ferramenta ViewGraph.

No Capítulo 6 são apresentadas as conclusões e os desdobramentos do presente trabalho.

No Apêndice A são incluídos alguns exemplos de representações gráficas de grafos de programa geradas pela ViewGraph, nesses exemplos são ressaltados e ilustrados os pontos e soluções mais relevantes apresentados nesse trabalho.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

A representação gráfica de grafos, não só de fluxo de controle, mas grafos de uma maneira geral, vem merecendo grande destaque em publicações internacionais desde meados da década de 40 até os dias de hoje; Fary [FAR48] e Tutte [TUT63] já mostravam estudos para desenhar grafos planares em 1948 e em 1963, respectivamente. Desde então, diversas publicações vêm mostrando os avanços na área. Neste capítulo encontra-se um apanhado geral dessas publicações enfatizando os principais conceitos, a terminologia utilizada, tendências e perspectivas da área.

2.1 - Conceitos Básicos e Terminologia

2.1.1 - Grafos

Definição 2.1 : Um *grafo G* consiste de dois conjuntos V e E. V é um conjunto não vazio de vértices. E é um conjunto de pares de vértices; esses pares são denominados *arcos*. $V(G)$ e $E(G)$ representam os conjuntos de vértices e arcos do grafo G. Também se escreve $G=(V,E)$ para representar um grafo. Num grafo *não - dirigido*, o par de vértices que representa qualquer arco não tem orientação espacial; assim sendo, (v_1,v_2) é o mesmo que (v_2,v_1). Num grafo *dirigido,* cada arco é representado por um par dirigido $[v_1,v_2]$, onde v_1 é a *fonte* e v_2 é o *destino* do arco.

Grafos podem ser representados graficamente e é essa representação gráfica que nos ajuda a entender muitas de suas propriedades. O diagrama de um grafo apenas nos mostra o relacionamento entre os seus vértices e os seus arcos. Deve-se observar que o diagrama de um grafo não é único; pode-se, a partir de um mesmo grafo, gerar vários diagramas distintos.

Note-se que dois arcos de um diagrama de um grafo podem interceptar-se num ponto que não é um vértice. Os grafos que têm um diagrama cujos arcos apenas se interceptam em seus vértices são chamados *planares*. Observamos na Figura 2.1 o exemplo de um grafo planar representado por um diagrama planar e, na Figura 2.2, o mesmo grafo representado por um diagrama não planar.

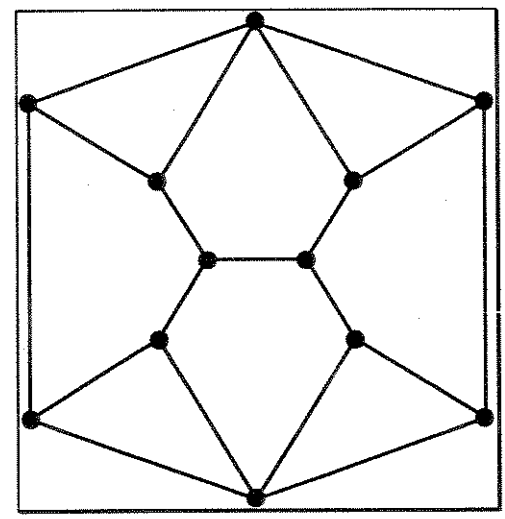


Figura 2.1. Grafo Planar

Muitas das definições e conceitos na teoria de grafos são sugeridas pela representação gráfica. Um arco (v_1,v_2) com $v_1=v_2$ (a fonte é igual ao destino) é chamado um "*loop*", e um arco com nós diferentes é chamado um "*link*".

Um grafo é *finito* se o seu conjunto de vértices e o seu conjunto de arcos são finitos.

Um *subgrafo* de G é um grafo G' de tal forma que $V(G') \subseteq V(G)$ e $E(G') \subseteq E(G)$.

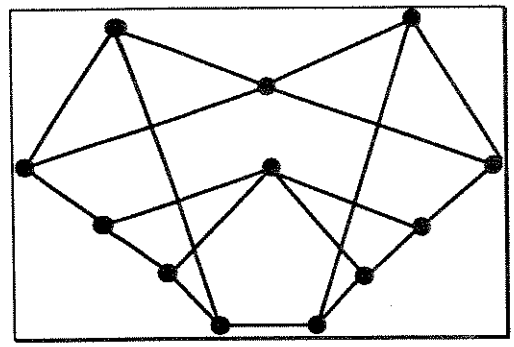


Figura 2.2. Grafo Planar Desenhado com cruzamento de arcos

Um *caminho* do vértice v_p para o vértice v_q no grafo G é uma seqüência de vértices v_p, v_{i_1}, v_{i_2},..., v_{i_n}, v_q de tal maneira que (v_p,v_{i_1}), (v_{i_1},v_{i_2}),..., (v_{i_n},v_q) pertencem a $E(G)$. Se G' for dirigido, o caminho consistirá então de $[v_p,v_{i_1}]$, $[v_{i_1},v_{i_2}]$,..., $[v_{i_n},v_q]$, bordas em $E(G')$.

Um *caminho simples* é um caminho em que todos os vértices são diferentes com a possível exceção do primeiro e do último.

O *comprimento* de um caminho é dado pelo número de arcos que ele contém.

Um *ciclo* é um caminho simples onde o primeiro e o último vértice são os mesmos.

2.1.2 - Grafos de Programas

Os **Grafos de Programa** são grafos dirigidos que mantêm uma relação direta com o programa fonte que os originou. Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos; a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos, na ordem dada, desse bloco. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor; e exatamente um único sucessor, exceto possivelmente o último comando.

A representação de um programa P como um grafo de programa $G=(N,E,s)$ consiste em estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos através de arcos. Considera-se todo grafo de programa como um grafo dirigido com um único nó de entrada $s \in N$ e um único nó de saída $e \in N$.

Algumas de suas características são :

- Normalmente possuem um *nó de entrada* e um *nó de saída*. Seja $IN(x)$ e $OUT(x)$ o número de arcos que entram e que saem do nó x, respectivamente. Se $IN(x) = 0$, x é um nó de entrada, e se $OUT(x) = 0$, x é um nó de saída.
- Programas estruturados podem ser representados por grafos planares, mas a sua representação pode não ser planar para facilitar a identificação de estruturas conhecidas.
- Os seus nós são normalmente numerados para facilitar a identificação com o bloco de comandos correspondente.
- Os grafos de programa podem ter diversas representações gráficas.

A Figura 2.3 mostra um grafo de programa típico, com seus nós numerados e alguns ciclos; observe que, para um programa (ou módulo) bem estruturado, tem-se apenas um nó de entrada (nó 1) e apenas um nó de saída[1] (nó 22), mas podem existir casos de múltiplos nós de entrada e de saída.

Os arcos de um grafo de programa são classificados em dois tipos, com relação à sua direção :

[1] Nó de saída é aquele que não possue sucessor.

Arcos diretos, aqueles que não correspondem a retornos no fluxo de execução, por exemplo na Figura 2.3, os arcos (5,9), (1,2), (16,17).

Arcos de ciclo, são os relacionados a retornos no fluxo de execução do programa, por exemplo, os arcos (20,2), (8,5) e (14,10) na Figura 2.3.

Muitas das idéias utilizadas na visualização de grafos de programa provêm de idéias utilizadas na visualização de estruturas hierárquicas como árvores; assim sendo, na próxima seção são apresentadas algumas definições e conceitos pertinentes a árvores, uma classe especial de grafos.

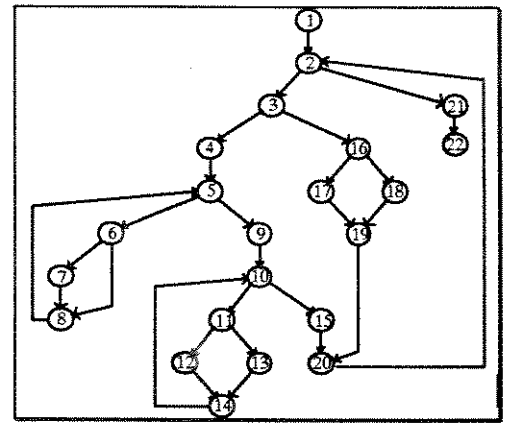


Figura 2.3. Grafo de Programa

2.1.3 - Árvores

As definições aqui colocadas para árvores seguem as de Horowitz e Sahni [HOR86]; outros autores adotam definições diferentes.

Definição 2.2 : Uma *Árvore* é um conjunto finito de um ou mais nós tal que : (i) existe um nó especialmente designado, denominado *raiz*, que não tem antecessor; (ii) os nós restantes estão particionados em zero ou mais conjuntos distintos $T_1,...,T_n$, em que cada um dos referidos conjuntos constitui-se numa árvore. $T_1,...,T_n$ são denominados as sub-árvores da raiz.

A Figura 2.4 é um exemplo típico de árvore com as suas principais características destacadas. Essa árvore possui 18 nós e 17 arcos; observe também que a árvore é normalmente desenhada com a *raiz* (definida a seguir) para cima.

A exigência de que $T_1,...,T_n$ sejam conjuntos disjuntos proíbe que as sub-árvores sejam interligadas; assim cada nó da árvore é raiz de alguma sub-árvore.

O *grau de um nó* é o número de suas sub-árvores; na Figura 2.4 o grau da raiz é 2. Os nós que tiverem grau zero são chamados nós *folha* ou nós *terminal*, enquanto que os outros nós são chamados *não-terminais*.

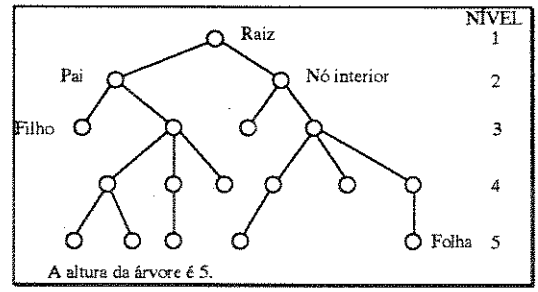


Figura 2.4. Exemplo de Árvore [WET79]

As raízes das sub-árvores de um nó X são os *filhos* de X. X é dito ser o nó *pai*. Os filhos do mesmo pai são denominados *irmãos*.

O grau de uma árvore é o grau máximo dos nós da árvore. A árvore da Figura 2.4 tem grau três.

Definição 2.3 : O *nível* de um nó é definido por admitir-se inicialmente que a raiz está no nível um. Estando um nó no nível *l*, os filhos estarão então no nível *l* + 1. A *altura* ou *profundidade* da árvore é definida como sendo o nível do nó de maior nível.

2.2 - Representação e Aspectos Estéticos

Os **Aspectos Estéticos** são usados para denotar critérios relativos a aspectos gráficos de legibilidade. Alguns aspectos estéticos são requisitos básicos, como a minimização dos cru-

zamentos de arcos. Além disso, para evitar o desperdício de espaço, é usual manter a área ocupada pelo desenho razoavelmente pequena.

Normalmente, mais de um aspecto estético é considerado ao mesmo tempo em aplicações reais. Podendo um desenho adequar-se perfeitamente a um aspecto estético, mas ser reprovado com relação a outro.

A Tabela 2.1 mostra uma taxionomia apresentada por Tamassia, et. al. [TAM88]; os aspectos estéticos podem ter as seguintes categorias:

1. Local / Global : local, quando se refere somente a uma parte do desenho, e, global, caso contrário.

2. Hierárquico / Não Hierárquico (Flat) : hierárquico, quando se refere à posição relativa de um conjunto de símbolos e, não hierárquico, caso contrário.

Tabela 2.1. Uma Taxionomia para Aspectos Estéticos [TAM88]

Abreviação	**Aspecto Estético**	Categ.
AREA	Minimização da área ocupada pelo desenho	G/F
BALAN	Balanceamento do desenho com relação ao eixo vertical ou ao eixo horizontal	G/H
BENDS	Minimização do número de curvas (bends) ao longo dos arcos	G/F
CONVEX	Minimização do número de faces desenhadas como polígonos convexos	G/F
CROSS	Minimização do número de cruzamentos entre arcos	G/F
DEGREE	Vértices de grau (degree) alto, no centro do diagrama	L/F
DIM	Minimização da diferença entre as dimensões dos vértices	G/F
ISO[2]	Sub-árvores isomórficas devem ter o mesmo desenho, e sub-árvores simétricas devem ter desenho espelhado	G/H
LENGTH	Minimização do comprimento global dos arcos	G/F
MAXCON	Minimização do comprimento do arco mais longo	G/F
SYMM	Simetria dos filhos na hierarquia	L/H
UNIDEN	Densidade uniforme de vértices no desenho	G/F
VERT	Estruturas hierárquicas colocadas verticalmente	L/H

[2] Observe que a definição de **ISO**, nessa Tabela, não foi dada por Tamassia [TAM88], mas sim por Reingold e Tilford [REI81].

Cada uma das três divisões de aplicações (árvores, grafos e grafos de programas) tem os seus próprios aspectos estéticos. Segue um resumo dos principais aspectos estéticos definidos em trabalhos apresentados em cada uma das áreas.

Os aspectos estéticos são apresentados seguindo a taxionomia apresentada na Tabela 2.1.

2.2.1 - Árvores e Aspectos Estéticos

Alguns aspectos estéticos relacionados a árvores foram definidos por Wetherell e Shannon [WET79]. Nesse artigo os autores apresentam algumas propriedades relacionadas ao desenho de árvores :

- Árvores são planares; portanto, não deve haver cruzamento de ramos (CROSS).
- Árvores impõem uma distância entre nós; nenhum nó pode estar mais próximo da raiz que qualquer um de seus antecessores (BALAN).

O primeiro aspecto estético definido foi :

Estética 1: Nós de uma árvore que estão no mesmo nível devem ficar sobre uma mesma linha reta e as linhas que definem os níveis devem ser paralelas (VERT).

Limite Físico: O desenho de uma árvore deve ocupar a menor largura possível (a altura do desenho da árvore é fixada pela própria árvore) (AREA).

Para árvores binárias temos o segundo aspecto estético:

Estética 2: Numa árvore binária, o filho da esquerda deve ser posicionado à esquerda do seu pai, e o filho da direita à direita do seu pai (SYMM).

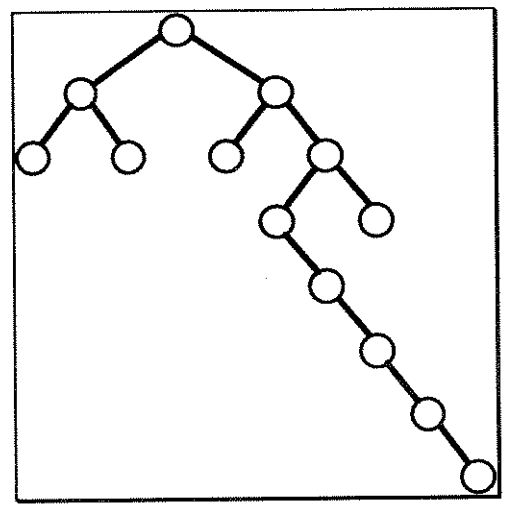


Figura 2.5. Árvore Desenhada pelo Alg. de W-S [WET79]

Estética 3: O nó pai deve estar centralizado em relação a seus filhos (SYMM).

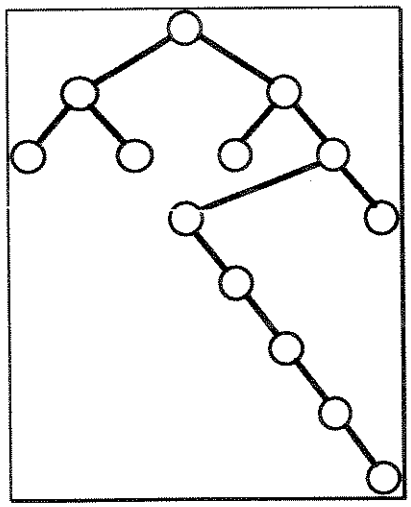


Figura 2.6. Árvore Desenhada com Largura Mínima [WET79]

As Figuras 2.5 e 2.7 mostram desenhos de árvores, dados pela aplicação de um dos algoritmos definidos por Wetherell e Shannon (W-S) [WET79]. A Figura 2.6 leva à definição do seguinte teorema:

Teorema : Desenhos de largura mínima podem existir, mas com a violação da Estética 3 por uma quantidade arbitrária.

Reingold e Tilford [REI81] mostram um problema comum a todos os algoritmos apresentados por W-S : o desenho de uma sub-árvore é influenciado pelo posicionamento de nós fora daquela sub-árvore; assim, árvores simétricas podem ser desenhadas assimetricamente. Verifica-se então a importância de se definir aspectos estéticos que levem em conta o isomorfismo e, por conseguinte, definir algoritmos que os implementem. Nesse

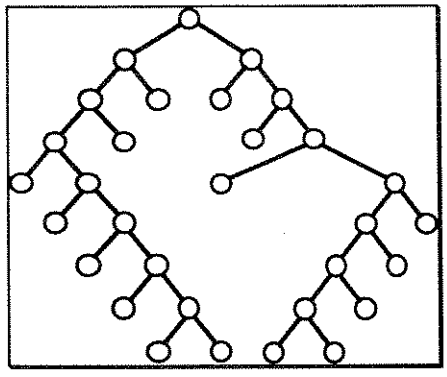


Figura 2.7. Posicionamento Final de uma Árvore pelo Algoritmo de W-S [REI81]

artigo os autores apresentam alguns exemplos onde os algoritmos de W-S falham; definem um novo aspecto estético (Estética 4) e propõem outro algoritmo para desenho de árvores, levando em conta o novo aspecto estético.

Estética 4 : Uma árvore e a sua imagem espelhada devem produzir desenhos que são reflexos um do outro; além disso, uma sub-árvore deve ser desenhada da mesma forma, não importando a posição onde ela ocorra na árvore (ISO).

2.2.2 - Aspectos Estéticos e Grafos Planares

Obviamente não existe um critério absoluto que capture de forma precisa a noção intuitiva de bons desenhos de grafos planares. Entretanto algumas propriedades podem ser colocadas como : todos os arcos são desenhados com segmentos de linha sem cruzamentos, etc; [CHI85A].

Diversos autores definem algoritmos para desenho de diagramas de grafos. Pode-se separar essas abordagens em dois tipos : com posicionamento de vértices restrito e com posicionamento de vértices livre.

O posicionamento dos vértices é restrito quando, por exemplo, são posicionados sobre uma matriz de pontos [BAT86] [TAM88], sobre círculos concêntricos [CAR80], sobre linhas paralelas [ROW87] [SUG81], etc. O posicionamento livre, sem nenhuma restrição, leva em conta a variação da entropia do sistema na busca pelo posicionamento ótimo [KAM89].

Outras restrições são também consideradas, como: estruturas simétricas devem ser desenhadas como figuras simétricas, isso porque, simetria é uma característica importante das estruturas. Alguns exemplos mostram que a simetria é até mais importante que o não cruzamento dos arcos. A Figura 2.8 mostra dois desenhos de um grafo de 16 vértices. Se não estivermos interessados em planaridade, a Figura 2.8 (a) é melhor para a compreensão da estrutura, apesar de ter cinco cruzamentos de arcos. A Figura 2.8 (b) sacrifica a simetria para eliminar o número de cruzamentos de arcos.

2.2.3 - Aspectos Estéticos e Grafos de Programa

Os desenhos dos grafos de programa devem manter um relacionamento com o programa fonte que os originou; assim, as estruturas principais dos programas, como "loops", "if-then-else" e "case", devem estar refletidas nos grafos correspondentes.

Para evidenciar o relacionamento entre o desenho do grafo e o programa fonte algumas convenções podem ser seguidas; por exemplo : para o posicionamento do comando *if-then-else* pode-se adotar a convenção de colocar

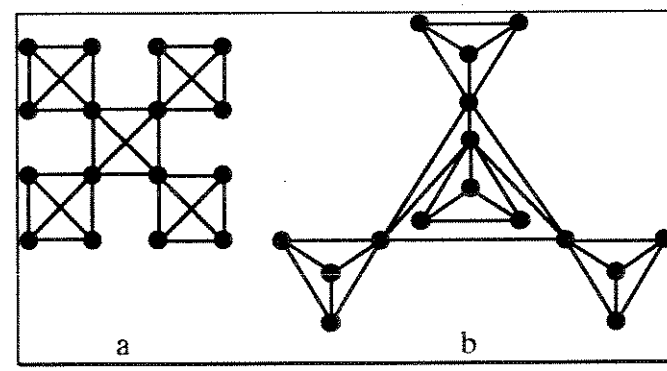


Figura 2.8. Desenho Simétrico e Planar de um Grafo [KAM89]

sempre o nó correspondente à parte THEN de um lado e o nó correspondente à parte ELSE do outro lado.

No caso do comando *case*, deve-se manter todos os nós que iniciam uma opção do *case* no nível diretamente consecutivo ao nível do nó de entrada do case; além disto, deve-se posicionar o nó de saída na mesma coluna do nó de entrada.

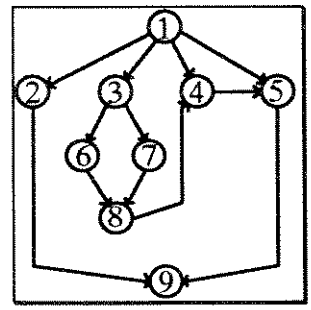


Figura 2.9. Exemplo de Comando Case

Observe na Figura 2.9 que o nó 5 é filho do nó 4; o nó 4 corresponde a uma seqüência de comandos sem um comando *break* para encerrá-los. Mesmo assim, tanto o nó 4 quanto o nó 5 estão posicionados no nível diretamente consecutivo ao nó de entrada do *case* (nó 1). Além disso, o nó 9 (saída do *case*) está posicionado na mesma coluna do nó 1 (entrada do *case*).

Essas restrições possibilitam ao usuário de uma ferramenta de visualização de grafos de programa identificar com maior facilidade as estruturas envolvidas nos diagramas.

2.3 - Trabalhos Relacionados

Vêm sendo publicados diversos algoritmos para resolver o problema de posicionamento de estruturas como : grafos, grafos planares, grafos hierárquicos, grafos de programas e árvores; uma gama enorme de novas idéias vêm surgindo. Algumas dessas idéias são resumidas nessa seção, como forma de nortear o leitor em sua própria pesquisa e situar o trabalho desenvolvido para essa dissertação.

Rowe et. al. [ROW87] apresentam um programa de propósito geral para posicionar grafos dirigidos. O programa fornece operações para examinar, editar e gerar automaticamente o desenho de um grafo, minimizando o número de cruzamentos de arcos. O programa tem ainda características que facilitam a sua integração a outras ferramentas.

O algoritmo de desenho do grafo é dividido em três fases. A primeira fase associa um nível a cada nó do grafo. A segunda fase posiciona os nós de cada nível a fim de minimizar o número de cruzamentos de arcos. A terceira e última fase ajusta o posicionamento dos nós e o roteamento dos arcos na tentativa de tornar o grafo mais fácil de ser entendido.

O cálculo dos níveis é feito de baixo para cima: primeiro acham-se todos os nós que não possuem descendentes, ficando esses nós no último nível; em seguida são calculados os níveis dos antecessores.

Os arcos longos são quebrados em pequenos segmentos entre níveis consecutivos; em cada nível é criado um nó imaginário (Figura 2.10) por onde passará o arco. Essa abordagem permite uma maior flexibilidade no posicionamento dos arcos gerando desenhos mais compactos.

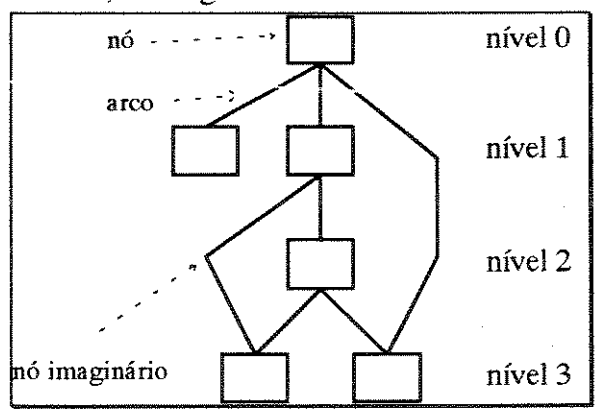


Figura 2.10. Nomenclatura para Grafos [ROW87].

Um número que estima a melhor posição horizontal de cada nó, chamado baricentro, é calculado e usado para posicionar os nós de cada um dos níveis. O baricentro corresponde

à posição média dos sucessores do nó no nível imediatamente consecutivo, incluindo nesse cálculo os nós imaginários, alocados para o posicionamento dos arcos.

Os nós imaginários são movidos de forma a ficar alinhados com os seus sucessores e predecessores, na tentativa de tornar os arcos longos o mais retilíneos possível.

Essa abordagem tem como vantagem a possibilidade de se desenhar os arcos de forma curvilínea, apesar de ser cada pedaço do arco ainda um segmento de reta. O algoritmo apresenta problemas quando da determinação dos níveis dos nós, pois ao se calcular o nível a partir dos nós folha, não podemos garantir que cada nó estará no nível mínimo ao qual ele pode ser atribuído. Esse problema no algoritmo gera desenhos de grafos com posicionamento vertical maior do que o necessário.

Gansner et. al. [GAN88] apresentam o programa DAG, que desenha grafos dirigidos a partir de uma lista de nós e arcos, computando o posicionamento e escrevendo a saída em formato PostScript. Instruções opcionais de desenho especificam a maneira pela qual os nós serão desenhados, atribuindo "labels" e espaços de controle. O programa trabalha melhor com grafos dirigidos acíclicos, que são freqüentemente utilizados para representar relacionamentos hierárquicos.

São apresentadas três propriedades desejáveis em desenhos de grafos direcionados; são elas:

P1 : O desenho deve enfatizar a ordem parcial implicada pelos arcos do grafo, posicionando nós em níveis, de tal forma que um nó fonte esteja sempre num nível mais abaixo do correspondente nó destino.

P2 : O desenho deve ter um número mínimo de cruzamentos de arcos e evitar quinas pontiagudas nos arcos.

P3 : O desenho deve manter os arcos curtos.

O algoritmo de posicionamento apresentado possui quatro passos :

Passo 1 : Os ciclos são eliminados, revertendo-se os arcos durante uma busca em profundidade primeiro. Os níveis dos nós são calculados. O problema de se associar níveis ótimos aos nós é computar níveis inteiros para os nós, de forma que a soma dos custos dos arcos seja minimizada. O custo de um arco é o produto do seu peso e da sua altura, onde a altura é a diferença de nível entre o nó fonte e o nó destino e o peso é uma aproximação da densidade de cada subgrafo.

Devido à natureza especial do problema, o método simplex de programação linear pode ser usado. No caso do DAG, foi desenvolvida uma variação combinatória do método simplex, no intuito de minimizar os cálculos.

Também nesse caso são criados nós imaginários nos pontos onde os arcos longos cruzam os níveis. Na implementação do DAG os nós imaginários são, mais tarde, utilizados como pontos de controle na geração de "*splines*" para os arcos longos, na tentativa de adaptar-se à propriedade **P2**.

Passo 2 : Ordenar os arcos da esquerda para a direita em cada nível, de forma a tentar minimizar o cruzamento de arcos. A abordagem é a mesma dada por Rowe [ROW87] a não ser pelo fato de que a posição de cada nó não é mais apenas a média das posições dos seus sucessores, mas sim uma média ponderada, dependendo de pesos dados aos sucessores do nó.

Passo 3 : Atribuir coordenadas absolutas aos nós, respeitando a ordenação determinada pelo passo anterior.

Passo 4 : Determina "B-Splines" para cada arco longo.

Kamada et. al. [KAM89] apresentam um algoritmo para desenhar grafos gerais não dirigidos. A principal característica da sua abordagem é o fato de que o posicionamento dos vértices não é restrito à linhas paralelas, circunferências concêntricas, etc. Essa característica aumenta as chances do algoritmo encontrar um posicionamento final do grafo, de forma a ficar em conformidade com aspectos estéticos definidos.

A idéia principal do algoritmo envolve um modelo chamado **Modelo de Molas** ("spring model"). É introduzido um sistema dinâmico de n partículas (n=|V|), conectadas mutuamente por **molas** imaginárias; cada partícula corresponde a um nó no grafo a ser desenhado. Assim, o grau de desordem do sistema pode ser formulado como a energia total E das molas.

Bons posicionamentos podem ser encontrados minimizando-se a energia total E do sistema de molas. O melhor posicionamento é o estado correspondente ao mínimo E (veja Figura 2.11).

Essa estratégia, definida por Kamada, utilizando um sistema de energia, gera desenhos de grafos que têm forte apoio na simetria, em detrimento do não cruzamento de arcos, veja Figura 2.8. Uma das principais aplicações desses algoritmos é na comparação de grafos isomorfos.

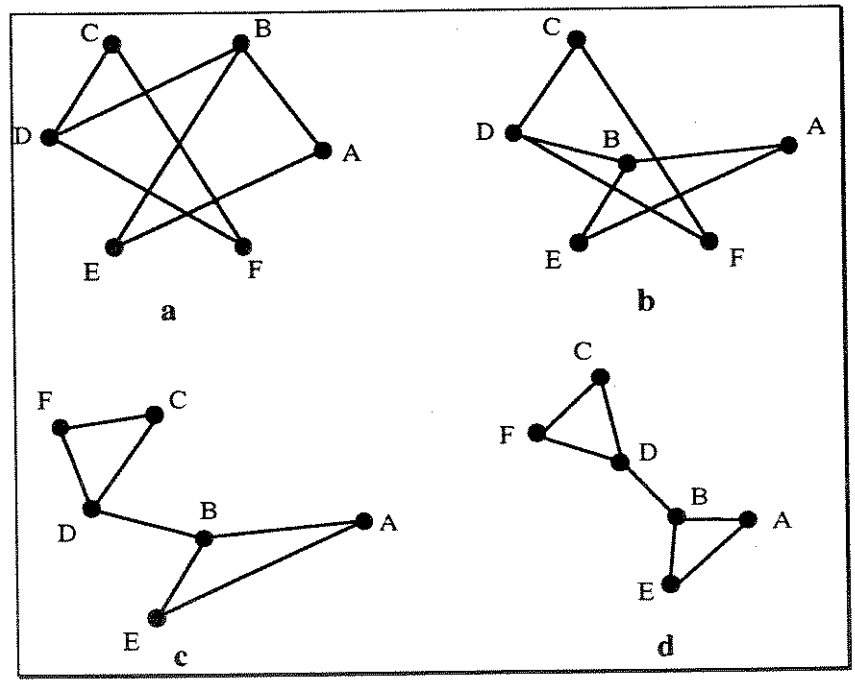


Figura 2.11. Processo de Minimização de Energia [KAM89].

Wetherell e **Shannon** [WET79] apresentam três algoritmos para posicionamento de árvores. Segundo eles, árvores modelam muitos dos problemas do mundo real e existem muitos algoritmos, baseados em árvores, úteis e eficientes. Igualmente importante, um bom desenho de uma árvore é um guia altamente intuitivo no modelamento de problemas.

Algoritmo 1: Para cada altura posicionar os nós mais à esquerda quanto possível, Figura 2.12.

O Algoritmo 1 é o mais simples apresentado no artigo, e satisfaz tanto o Limite Físico quanto a Estética 1 (Seção 2.2.1). Isto é, ele ocupa o menor espaço possível no dispositivo de saída e nós de um mesmo nível estão posicionados sobre uma mesma linha reta.

O **Algoritmo 2**, de Knuth [KNU71], satisfaz a Estética 2 (Seção 2.2.1), associando a cada nó uma coordenada x proporcional ao índice do nó na numeração "in-order" da árvore. Como

o índice "in-order" de qualquer nó é sempre maior que o do seu filho a esquerda e menor do que o do seu filho a direita, por indução, todos os nós são posicionados corretamente segundo a Estética 2. Um exemplo de árvore posicionada através desse algoritmo é mostrado na Figura 2.13.

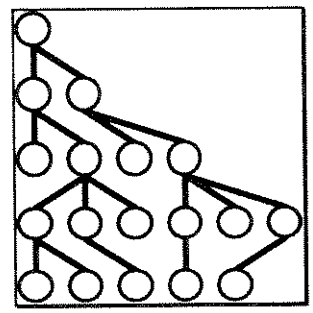


Figura 2.12. Exemplo Algoritmo 1 de W-S [WET79]

O problema do Algoritmo 2 é que ele ocupa uma largura muito grande, tendo em vista que cada nó ocupa uma coluna específica e que nenhum outro nó pode ocupar aquela mesma coluna.

Tanto o Algoritmo 1 quanto o 2 satisfazem completamente uma condição e ignoram a outra; em ambos os casos resultados grotescos podem aparecer.

O **Algoritmo 3** é definido, misturando as idéias dos dois anteriores. Como no Algoritmo 1 um "array" mantém a próxima posição livre para um nó da árvore a cada nível. Se uma folha no nível h está sendo considerada, o posicionamento da folha em $array[h]$ é permitido e satisfaz o requisito de largura mínima ocupada. Mas um nó interno posicionado de qualquer forma em $array[h]$ pode violar a Estética 2. Assim sendo, uma posição temporária para um nó interno é a média da posição dos seus filhos. A posição definitiva é o valor máximo entre a posição temporária e o valor de $array[h]$, evitando que os filhos arrastem os pais muito para a esquerda de forma a colidir com seus parentes à esquerda.

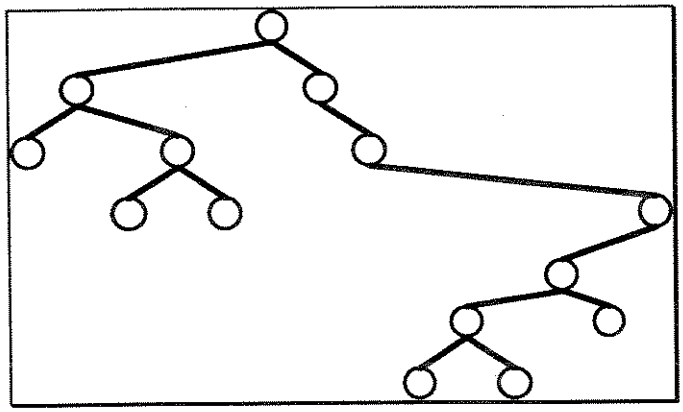


Figura 2.13. Exemplo Algoritmo 2 de W-S [WET79]

Se a posição definitiva de um nó interior é à direita da sua posição provisória, a sub-árvore iniciada no nó em questão deve ser movida inteiramente para a direita, centralizando os filhos em torno do pai.

O Algoritmo 3 pode, em alguns casos, não respeitar o Limite Físico pois dá um peso maior à Estética 2.

Apesar de melhor que os dois primeiros o Algoritmo 3 ainda apresenta problemas; pode não respeitar o Limite Físico em alguns casos e, basicamente, não possui uma característica importante que é a de posicionar árvores simétricas de forma simétrica. As Figuras 2.4 e 2.6, da Seção 2.2.1 - Grafos, mostram árvores posicionadas pelo Algoritmo 3 de W-S.

Como os exemplos ilustram, a dificuldade dos algoritmos de W-S reside no fato de que a forma de uma sub-árvore é influenciada pelo posicionamento de nós fora desta sub-árvore. Como conseqüência disso, árvores simétricas podem ser desenhadas de forma assimétrica, ou generalizando, uma árvore e o seu reflexo nem sempre produzirão desenhos espelhados.

Reingold e **Tilford** [REI81] mostram alguns problemas nos algoritmos de W-S e propõem um novo algoritmo que produz desenhos melhores.

Para satisfazer a Estética 4 (seção 2.2.1) é necessário um algoritmo em que nós fora de uma sub-árvore não interfiram no posicionamento relativo de nós da sub-árvore. Para este fim é proposto o algoritmo TR, baseado na seguinte heurística: duas sub-árvores de um nó devem ser formadas independentemente, e, então, serem colocadas tão próximas quanto possível.

Esta heurística é aplicada enquanto a árvore é percorrida numa busca pós-ordem, como segue. A cada nó **T**, imagine que as suas duas sub-árvores tenham sido desenhadas e cortadas do papel ao longo do seu contorno. Então as duas são sobrepostas no seu nó raiz e afastadas até que nenhum ponto esteja mais em contato. Inicialmente, as raízes são separadas por um espaço mínimo pré definido; então no nível de baixo, as sub-árvores são afastadas até que esse espaço mínimo seja novamente alcançado. Esse processo continua em níveis sucessivamente menores até que a parte de baixo da menor sub-árvore seja alcançada.

Segundo os autores, as Estéticas 1,2 e 3 são satisfeitas; pais são visitados depois de seus filhos, assim podem ser centralizados em relação a eles; e sub-árvores vazias são tratadas como no Algoritmo W-S (Algoritmo 3 de W-S). A diferença essencial é que imagens de árvores espelhadas produzem desenhos com imagens espelhadas e uma sub-árvore é desenhada do mesmo jeito, não importando onde ela ocorra.

Price, et. al. [PRI87] apresentam um algoritmo de projeção de grafos que parte da matriz de adjacência que representa o grafo de fluxo de controle do programa em análise. A cada nó é atribuído um intervalo na coordenada X de modo que o intervalo de um nó pai contenha os intervalos de seus descendentes, não podendo conter quaisquer sobreposições de intervalos.

O cálculo do tamanho de cada intervalo é feito a partir do nó inicial para o nó final, segundo a numeração crescente dos nós. Para cada nó pai verifica-se o número de nós filhos, atribuindo-se a cada um deles um intervalo padrão. Se a soma dos intervalos dos nós filhos for maior que a do nó pai, atualiza-se o intervalo do nó pai. Esse arranjo deve seguir para os antecessores do nó pai. Depois de calculados todos os intervalos, os nós são posicionados no centro do seu próprio intervalo. A coordenada Y relativa a cada nó é calculada da seguinte forma : cada nó filho recebe a coordenada Y de seu pai acrescentada de um desvio padrão.

Essa abordagem contém alguns problemas; por exemplo :

- A alocação de espaço para os nós só é recalculada no caso de falta de espaço; nada é feito no caso de sobra de espaço, o que pode levar a uma disposição não mínima do grafo.

- O cálculo da coordenada Y omite a cláusula de que o nó deve receber a coordenada de seu pai de maior coordenada Y, mais um desvio padrão. Isso gera problemas porque um nó pode ter vários pais com diferentes posicionamentos na coordenada Y. Se esse detalhe for esquecido, pode-se gerar diagramas incorretos, dependendo da ordem de busca no grafo durante o cálculo dessa variável.

- Ainda no cálculo da coordenada Y, nada é especificado para o cálculo dessa variável na presença da estrutura case, que possui alguns detalhes especiais (veja Seção 4.2).

- Nada é especificado sobre o roteamento dos arcos, detalhe importante principalmente em grafos mais complexos.

Na Tabela 2.2 encontra-se uma síntese dos principais trabalhos apresentados na literatura, com respeito à visualização automática de diagramas.

2.4 - Tendências e Perspectivas

Projetistas de todas as disciplinas sempre esquematizam graficamente as suas idéias antes de apresentá-las por meio de expressões de uma linguagem simbólica. Os diagramas ajudam a formar uma imagem mental clara e correta das estruturas e funções de um projeto.

No contexto de programação, os diagramas aparecem em várias vertentes, como diagramas de estruturas de dados, diagramas de blocos, histogramas, e grafos de fluxo de programa. Mas o que faz um diagrama parecer bom? Ding e Mateti [DIN90] apresentam um estudo sobre os aspectos estéticos envolvidos no desenho de diagramas, tentando responder a essa pergunta carregada de subjetividade.

Além do estudo de aspectos estéticos, visando gerar diagramas mais inteligíveis e, conseqüentemente, mais úteis, as tendências da área de geração de diagramas caminham em duas direções : desenvolvimento de aplicativos e aprimoramento dos algoritmos de posicionamento.

O desenvolvimento de aplicativos envolve a utilização da visualização de diagramas em áreas como : depuradores [ISO87], [MAT86]; entendimento de programas [BRO84], [LON85]; e programação visual [BOR81], [FIN84], [GLI84], [PON83].

O aprimoramento dos algoritmos de posicionamento tem sido estudado, ultimamente, em áreas como : desenho de árvores dinâmicas [MOE90]; desenho de grafos dirigidos de tamanho excessivo [MES91]; grafos dirigidos normais [FRU91], [GAN93]; além de visualizações especiais, como grafos vistos através de simuladores de lentes tipo "olho de peixe" [SAR92].

Tabela 2.2. Algoritmos para Geração Automática de Diagramas

Autor (es)	Domínio	Atributos	Referência
Chiba, *et al.*	Grafo Planar não Direcionado	Layouts convexos	[CHI84]
Tamassia	Grafo Planar não Direcionado	Arcos com o Mínimo Número de Curvas	[TAM87]
Eades	Grafo Geral não Direcionado	Desenhos Simétricos	[EAD86]
Fruchterman e Reingold	Grafo Geral não Direcionado	Aparência Tridimensional	[FRU91]
Sarkar e Brown	Grafo Geral não Direcionado	Visão "Olho de Peixe"	[SAR92]
Lipton, *et al.*	Grafo Geral não Direcionado	Desenhos Simétricos	[LIP85]
Kamada e Kawai	Grafo Geral não Direcionado	Simetria, Minimização de Energia	[KAM89]
Carpano	Grafo Geral Direcionado	Hierárquico, Redução dos Cruzam.	[CAR80]
Sugiyama, *et al.*	Grafo Geral Direcionado	Hierárquico, Redução de Cruzamentos	[SUG81]
Rowe, *et al.*	Grafo Geral Direcionado	Ciclos, Manipulação Interativa	[ROW87]
Gansner, *et al.*	Grafo Geral Direcionado	Grafos Acíclicos	[GAN88]
Gansner, *et al.*	Grafo Geral Direcionado	"Splines" para Desenhar Curvas	[GAN93]
Wetherell e Shannon	Árvores	Desenho Hierárquico de Árvores	[WET79]
Vaucher	Árvores	Impressão de Árvores em Páginas	[VAU80]
Reingold e Tilford	Árvores	Desenho Hierárquico de Árvores	[REI81]
Moen	Árvores	Desenho de Árvores Dinâmicas	[MOE90]
Messinger, *et al.*	Grafo Grande Direcionado	Hierárquico, Divisão em Páginas	[MES91]
Knuth	Fluxograma	Documentação de Código Fonte	[KNU63]
Stockenberg	Fluxograma	Representação Pictorial de Programas	[STO75]
Batini	Fluxograma	Aspectos Estéticos Considerados	[BAT86]
Protsko, *et al.*	Fluxograma	Geração Automática de Diagramas de Software.	[PRO91]
Battista e Tamassia	Grafos Acíclicos	Hierárquico, representação plana	[BAT88]
Price, *et al.*	Grafo de Programa	Preserva Estruturas Principais do Programa	[PRI87]

CAPÍTULO 3

ESPECIFICAÇÃO DE UMA FERRAMENTA DE VISUALIZAÇÃO DE INFORMAÇÕES PERTINENTES AO TESTE BASEADO EM ANÁLISE DE FLUXO DE DADOS

Neste capítulo encontra-se um resumo do tema Teste Baseado em Análise de Fluxo de Dados para introduzir a apresentação de alguns aspectos da POKE-TOOL, ferramenta de apoio ao Teste Estrutural de Programas [CHA91]. Por fim são apresentadas algumas problemáticas da visualização de informações de teste e a especificação de uma ferramenta que visa atuar paralelamente à POKE-TOOL, realizando a visualização de informações de teste por ela geradas.

3.1 - Teste Estrutural Baseado em Análise de Fluxo de Dados

Os conceitos apresentados aqui sobre Teste Estrutural Baseado em Análise de Fluxo de Dados são considerados na apresentação da POKE-TOOL, Seção 3.2; e alguns deles são passíveis de visualização e portanto considerados na especificação de uma ferramenta de visualização de informações de teste denominada ViewGraph, Seção 3.2.

Uma Técnica de Teste que se utiliza da especificação do software para derivar os requisitos de teste é denominada *técnica baseada em especificação*; é, também, conhecida como *técnica funcional* ou *técnica de caixa preta*. Aquela que se utiliza principalmente de informações sobre os erros mais comuns cometidos durante o processo de desenvolvimento de software, é chamada de *técnica baseada em erros*. A que se utiliza, essencialmente, de informações oriundas do texto do programa é denominada *técnica baseada em programa*; é também denominada *técnica estrutural de teste* ou de *caixa branca* [RAP85], [URA88], [NTA84], [MAL91].

Os primeiros Critérios de Teste Estrutural de programas utilizados eram baseados unicamente no fluxo de controle de programas, sendo os mais conhecidos : o *Critério Todos os Nós (todos-nós)* requer que todos os comandos sejam executados pelo menos uma vez, o *Critério Todos os Ramos (todos-ramos)* requer que todas as transferências de controle entre blocos de comandos sejam exercitadas pelo menos uma vez e o *Critério Todos os Caminhos (todos-caminhos)* requer que todos os caminhos possíveis de um programa sejam exercitados.

Uma vez que um critério é escolhido, dados de teste devem ser selecionados de modo a satisfazer o critério. Uma forma de realizar isso é selecionar caminhos completos no programa, com elementos que satisfaçam o critério escolhido, e então encontrar os dados de entrada que exercitem cada um desses caminhos. A busca por dados de entrada adequados pode ser facilitada através da visualização do grafo do programa e dos caminhos a serem exercitados.

Uma das principais desvantagens do teste estrutural é que, dependendo do dado de entrada, um determinado caminho pode ou não revelar a presença de um defeito; sendo assim, executar todos os caminhos do programa não garante que todos os defeitos serão revelados, necessariamente.

Os critérios citados anteriormente baseiam-se na análise de fluxo de controle do programa para derivar requisitos de teste. Posteriormente, surgiram os Critérios Baseados em Análise de Fluxo de Dados do Programa. A análise de fluxo de dados é largamente utilizada por

compiladores na otimização de código e pode também ser considerada para derivar requisitos de teste.

A análise de fluxo de dados focaliza definições de variáveis e como elas são utilizadas. As ocorrências de uma variável em um programa podem ser uma definição de variável, um uso de variável ou uma indefinição. A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo. Dois tipos de usos são distinguidos - c-uso e p-uso. O primeiro tipo afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado. O segundo tipo afeta diretamente o fluxo de controle do programa.

Uma variável está indefinida quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar definida em memória.

Na Figura 3.1 observam-se três tipos de ocorrências de variáveis, a definição das variáveis X e Z no nó 1 e o c-uso de X no mesmo nó; além dos p-usos de Y nos arcos (1,2) e (1,3). A visualização de informações como as definições, c-uso e p-uso das variáveis de um programa, relacionadas diretamente com o nó ou arco onde aparecem, tem importante aplicação tanto na determinação dos dados de teste quanto, posteriormente, na fase de depuração.

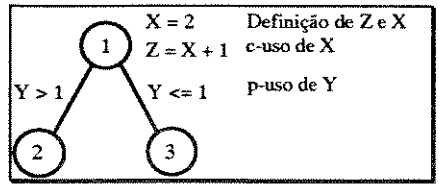


Figura 3.1. Tipos de Ocorrências de Variáveis

Um caminho (i, n_1, ..., n_m, j), m≥0 que não contenha definição de uma variável nos nós n_1, ..., n_m é chamado de *caminho livre de definição* com respeito a (c.r.a) *x* do nó i ao nó j e do nó i ao arco (n_m,j).

Um nó i possui uma definição global de uma variável *x* se ocorre uma definição de *x* no nó i e existe um caminho livre de definição de i para algum nó ou para algum arco que contém um c-uso ou um p-uso, respectivamente, da variável *x*. Um c-uso da variável *x* em um nó j é um *c-uso global* se não existir uma definição de *x* no nó j precedendo este c-uso; caso contrário é um *c-uso local.*

Alguns critérios baseados em análise de fluxo de dados foram definidos por Rapps e Weyuker [RAP85], [WEY84], e são denominados Família de Critérios Baseados em Fluxo de Dados. Uma característica comum nessa classe de critérios é a de que elas requerem que sejam testadas as interações que envolvam definições de variáveis de programas e subseqüentes referências a essas definições.

Foram introduzidos alguns conceitos e definições para estabelecer esta Família de Critérios; um deles foi a proposição do *grafo def-uso* ('def-use graph') que consiste em uma extensão do grafo de fluxo de controle, incorporando-se informações semânticas do programa a este grafo. O grafo def-uso é obtido a partir do grafo de programa associando-se a cada nó i os conjuntos c-use(i)={variáveis com c-uso global no bloco i} e def(i)={variáveis com definições globais no bloco i}, e a cada arco (i,j) o conjunto p-use(i,j)={variáveis com p-usos no arco (i,j)}.

O grafo def-uso pode ser visualizado a partir do grafo de programa, bastando para tanto mostrar os conjuntos c-uso(i) e def(i) para cada nó i e o conjunto p-uso(i,j) para cada arco (i,j), veja a Figura 3.2.

Maldonado, Chaim e Jino [MAL88], [MAL91], definem a Família de Critérios Potenciais Usos (PU) e a correspondente Família de Critérios Potenciais Usos Executáveis (FPU). Esses critérios diferem dos definidos por Rapps e Weyuker pois exigem associações independente-

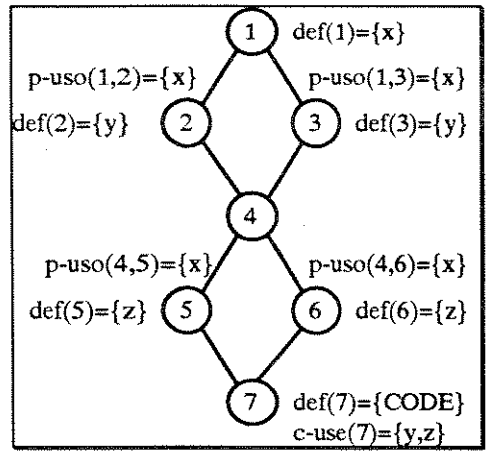


Figura 3.2. Grafo Def-Uso

mente da ocorrência explícita de uma referência a uma determinada definição : se um uso de uma definição **pode** existir - um *potencial uso* - a potencial associação é requerida.

Esses critérios portanto são mais rigorosos que os anteriores e possibilitam, por exemplo, verificar que o valor de uma variável *x* definida em *i* não foi alterado indevidamente (devido a efeitos colaterais) nos caminhos onde não há redefinição de *x*, ganhando-se, desta forma, maior confiança de que a computação correta é realizada; isto está de acordo com a filosofia discutida por Myers [MYE79] : um erro está claramente presente se um programa não faz o que supõe-se que ele faça, mas erros estão também presentes se um programa faz o que supõe-se que não faça.

3.2 - Estrutura da POKE-TOOL

Como já mencionado no Capítulo, a POKE-TOOL [CHA91] é uma ferramenta de apoio ao Teste Estrutural de programas, que suporta a aplicação dos Critérios Potenciais Usos para o teste de unidades - onde unidades são entendidas como procedimentos de uma linguagem procedural; sua estrutura é apresentada na Figura 3.3.

A ferramenta pode ser configurada para trabalhar com código fonte escrito em linguagens procedurais com gramática livre de contexto, que satisfaçam algumas limitações. Basicamente, o configurador deve especificar o analisador léxico e o analisador sintático para a linguagem alvo. Chaim [CHA91a] descreve como realizar essas tarefas.

A POKE-TOOL é uma ferramenta interativa, com operação direcionada a *sessão de trabalho*. Numa sessão de trabalho o usuário pode executar as seguintes tarefas de teste : análise estática da unidade; preparação para o teste; submissão de casos de teste; avaliação de casos de teste; e gerenciamento dos resultados de teste.

A sessão de trabalho na POKE-TOOL é dividida em duas fases : a fase estática, e a fase dinâmica.

- Na fase estática a ferramenta analisa o código fonte, obtendo informações necessárias para a aplicação dos Critérios de Teste; e instrumentando o código fonte, através da inserção de instruções de escrita - chamadas *pontas de prova* - que produzem um "trace" do caminho percorrido quando da execução dessa unidade. Essa instrumentação gera uma nova versão da unidade em teste - *versão instrumentada* -, que viabiliza a posterior avaliação da adequação de um dado conjunto de casos de teste aos Critérios de Teste.

Após essa primeira etapa, a POKE-TOOL pode apresentar informações como : o conjunto de caminhos requeridos pelo *Critério Todos Potenciais-du-Caminhos* e o conjunto de associações requeridas pelo *Critério Todos Potenciais-Usos* e *Todos Potenciais-Usos/du*. Com essas informações o usuário pode projetar os casos de teste, a fim de executar os caminhos ou associações exigidas, ou alternativamente, avaliar a cobertura fornecida por um dado conjunto de casos de teste T.

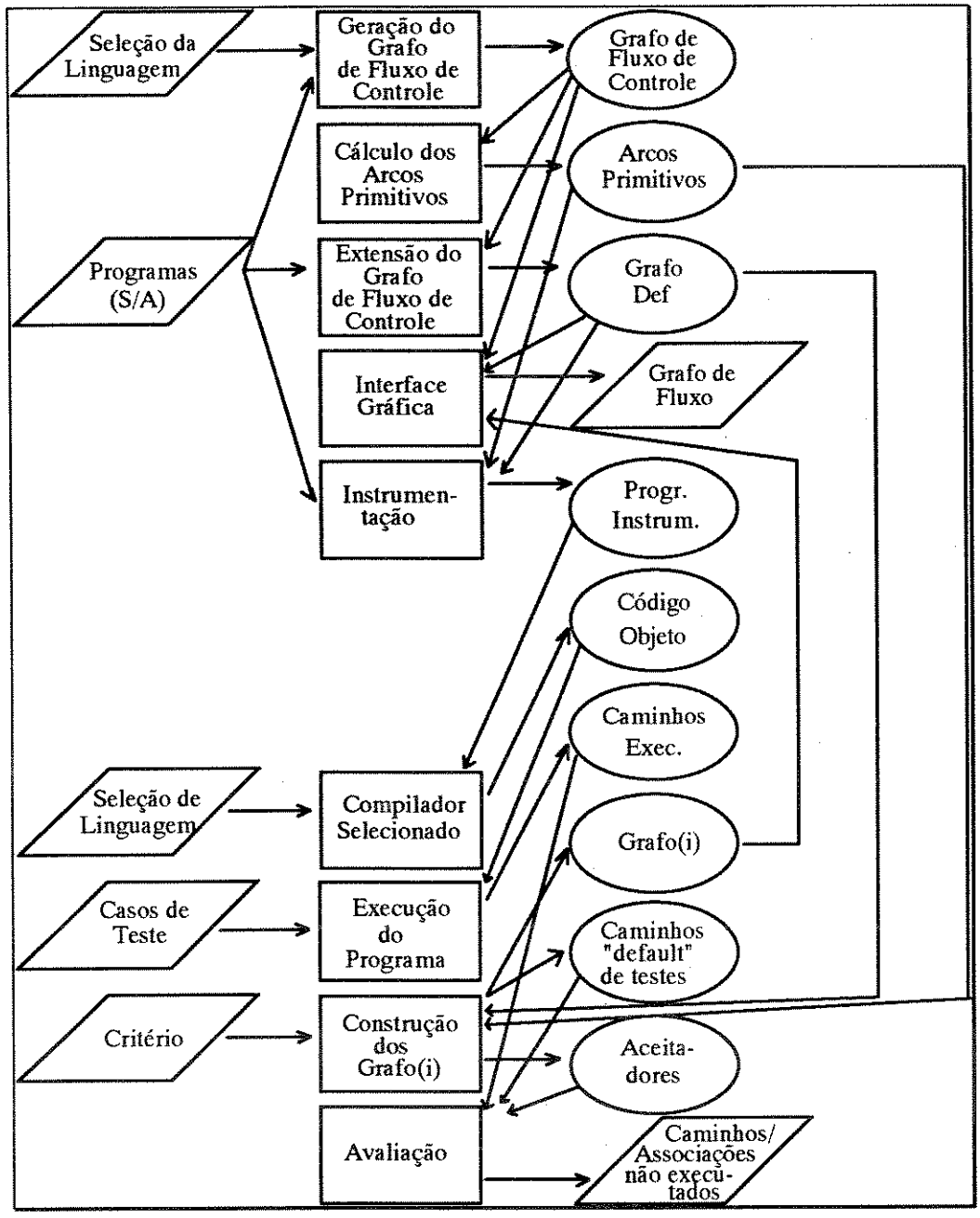


Figura 3.3. Arquitetura da Ferramenta de Teste Estrutural POKE-TOOL [CHA91].

- Na fase dinâmica a POKE-TOOL executa e avalia os casos de teste, para cada um dos Critérios Potenciais Usos. O resultado da avaliação é um conjunto de caminhos ou associações ainda não cobertos e o percentual de cobertura dos casos de teste, para cada um dos critérios.

Casos de teste devem ser executados até que a cobertura atinja 100% - todas associações executadas ou se não for executada, que seja detectada a sua não executabilidade - claro que o usuário pode interromper esse processo e abandonar a sessão de trabalho; para isso a POKE-TOOL fornece meios de armazenar os dados gerados e o estado atual da ferramenta, para posterior recuperação.

Na Figura 3.3 é apresentada a estrutura da POKE-TOOL; observa-se na coluna central que estava prevista uma função denominada : Interface Gráfica, essa função deveria apresentar

graficamente o grafo de programa, mas ainda não está disponível no protótipo atual da POKE-TOOL.

A POKE-TOOL gera, além do grafo de programa, outras informações relevantes à visualização, como : a versão instrumentada do programa, o grafo-def, associações requeridas, caminhos executados, associações executadas, etc...

3.3 - Especificação da ViewGraph

Com os conceitos sobre Teste Estrutural Baseado em Análise de Fluxo de Dados e as idéias básicas a respeito da POKE-TOOL apresentadas, pode-se passar agora para a especificação de um ferramenta que se propõe a visualizar informações de teste oriundas da POKE-TOOL com o objetivo de auxiliar em tarefas de teste como : escolha de dados de teste e teste de regressão, e em tarefas de depuração. O diagrama principal dessa especificação é mostrado na Figura 3.4.

Os principais problemas de uma ferramenta que se propõe a visualizar informações de teste estão relacionadas à representação gráfica do grafo de programa; os algoritmos propostos para solucionar esse problema são apresentados no Capítulo 4.

A ferramenta ViewGraph, além da visualização das informações de teste, deve proporcionar também um ambiente interativo e amigável, com o objetivo de tornar a atividade de teste a mais atraente e produtiva quanto possível. Assim, definem-se alguns de seus requisitos básicos :

1. Possibilitar a visualização de informações provenientes da análise estática do módulo em teste. Estas informações são basicamente : o programa fonte, os requisitos de teste exigidos pelos diversos critérios, o código fonte de cada nó do programa e as variáveis definidas em cada nó.

Na definição dos casos de teste é fundamental ter-se a maior quantidade possível de informações sobre a estrutura interna do programa em teste. Assim pode-se definir com precisão quais dados de entrada selecionar, no intuito de exercitar associações específicas do programa, associações essas exigidas pelos critérios de teste.

A visualização das informações de origem estática auxilia na definição do conjunto de casos de teste, pois facilita a observação da estrutura do programa em confronto com os requisitos exigidos pelos critérios de teste.

2. Mostrar informações derivadas da análise dinâmica feita pela POKE-TOOL, como: entrada dos casos de teste, entrada do teclado, saída dos casos de teste, caminhos percorridos pelos casos de teste, etc.

Com a submissão e avaliação dos casos de teste começa a análise da parte dinâmica das informações de teste geradas pela POKE-TOOL, nesta fase a ViewGraph começa a atuar sobre a depuração do programa. Se for detectada uma saída incorreta, pode-se recuperar o caminho percorrido pelo caso de teste que a gerou, na tentativa de se localizar o defeito no programa.

3. Para que as duas partes anteriores possam ser visualizadas de forma amigável e interativa, é imprescindível a visualização gráfica do grafo de programa. Isso viria a suprir a falta da interface gráfica, prevista para a POKE-TOOL mas ausente no seu protótipo atual.

Por fim, com toda a informação gerada pela POKE-TOOL e com o grafo do programa visualizado pela ViewGraph, pode-se facilmente imprimir e catalogar os procedimentos e resultados dos testes, criando um ambiente propício para o gerenciamento dos resultados de teste.

A ferramenta é dividida em duas partes principais, uma que gera a disposição gráfica do grafo de programa e outra que controla um menu de opções que manipulam o grafo gerado e as informações de teste a serem visualizadas; segue a descrição das funções :

Calcula Disposição Gráfica do Grafo de Programa : Essa função tem como entrada o nome do arquivo que contém o grafo do programa, em forma textual, a ser visualizado; esse arquivo é lido e é gerada uma representação dele em memória, em forma de lista de adjacências; uma disposição gráfica do grafo é calculada levando em consideração posições pré-definidas, se existirem.

Controla Menu : Essa função é responsável por gerar a interface da ferramenta e controlar a execução de diversas funções de acordo com as opções selecionadas pelo usuário.

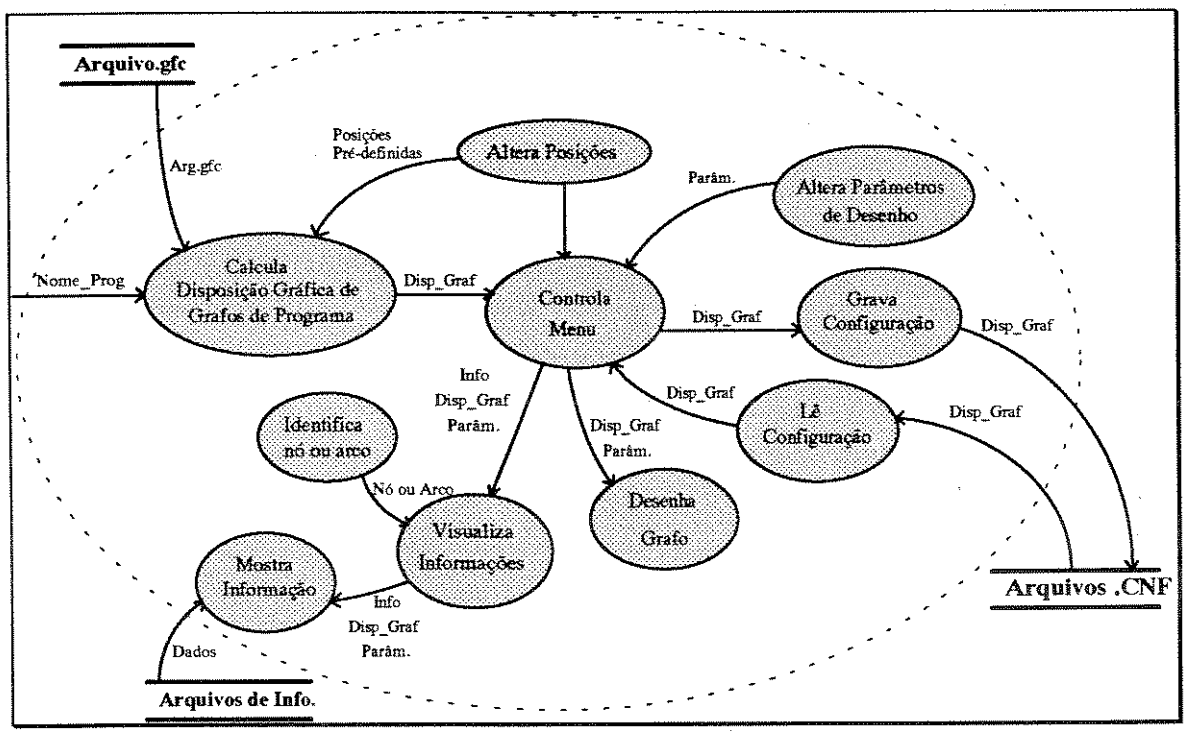


Figura 3.4. Especificação da Ferramenta ViewGraph

Altera Posições : Essa função é acionada no caso do usuário querer modificar a disposição inicial fornecida para o grafo de programa.

Altera Parâmetros de Desenho : Modifica os parâmetros de espaçamento, deslocamento e escala do grafo de programa.

Grava Configuração : No caso do usuário ter modificado o grafo inicial essa função armazena essas modificações para futura recuperação.

Lê Configuração : Recupera modificações de um grafo de programa previamente gravadas.

Desenha Grafo : Desenha o grafo de programa no dispositivo de saída, recebe como entradas a disposição gráfica gerada na função Calcula Disposição Gráfica do Grafo de Programa e os parâmetros do desenho, alterados ou não pela função Altera Parâmetros de Desenho. Essa função é chamada logo que o Controla Menu é acionado e é novamente executada se a disposição gráfica ou os parâmetros forem modificados.

Visualiza Informações : Essa função está encarregada de mostrar as informações de teste geradas pela POKE-TOOL, ela aciona duas outras funções : a que identifica o nó ou o arco, no caso da informação a ser visualizada estar relacionada a alguma dessas estruturas, e a que lê as informações do arquivo e mostra no dispositivo de saída.

Na Figura 3.5 é apresentado um detalhamento da função que calcula a disposição gráfica do grafo de programa.

A função que calcula a disposição gráfica do grafo de programa está dividida em duas partes : o posicionamento dos nós e o roteamento dos arcos. Essas etapas estão mais detalhadas no Capítulo 4.

Essa ferramenta deve ter uma interface interativa e com manipulação direta sobre os componentes do grafo de programa, para que informações como o código fonte de cada nó e as variáveis neles definidas, por exemplo, possam ser visualizadas bastando que o usuário, com o "mouse", indique qual o nó a ser selecionado. Além disso, o grafo de programa deve ser visualizado o tempo todo na tela, sendo que as outras informações devem aparecer em janelas temporárias ativadas e desativadas pelo usuário; uma sugestão de interface é apresentada a seguir.

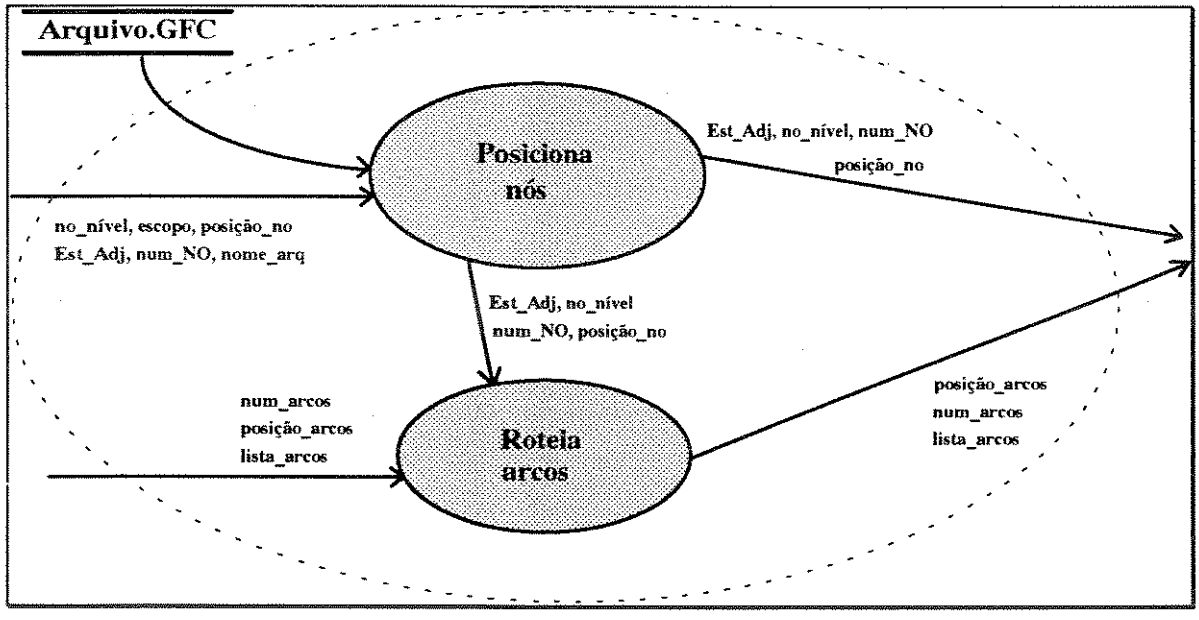


Figura 3.5. Cálculo da Disposição Gráfica do Grafo de Programa

A Figura 3.6 mostra a tela principal da ferramenta ViewGraph (versão para estação de trabalho). Essa tela é dividida em duas partes : o Painel de Controle e a Área de Desenho.

No Painel de Controle estão os seis botões do menu principal; os cinco primeiros têm um triângulo à direita do nome do botão, esses triângulos indicam a existência de sub-menus. O

painel de controle é utilizado para ativar esses sub-menus e a partir deles indicar quais as informações de teste que devem ser visualizadas.

A Área de Desenho é utilizada pelo Módulo Desenha Grafo para visualizar o grafo de programa previamente posicionado pelo Módulo Calcula Disposição Gráfica de Grafos de Programas; essa área pode ser ampliada ou diminuída deslocando-se os cantos do "frame" utilizado para enquadrar essa área; se o usuário não quiser aumentar ou diminuir a área de desenho pode-se ainda optar pela utilização dos "scrollbars" que são as setas posicionadas à direita e abaixo da área de desenho. Os "scrollbars" deslocam o grafo de programa dentro da área de desenho.

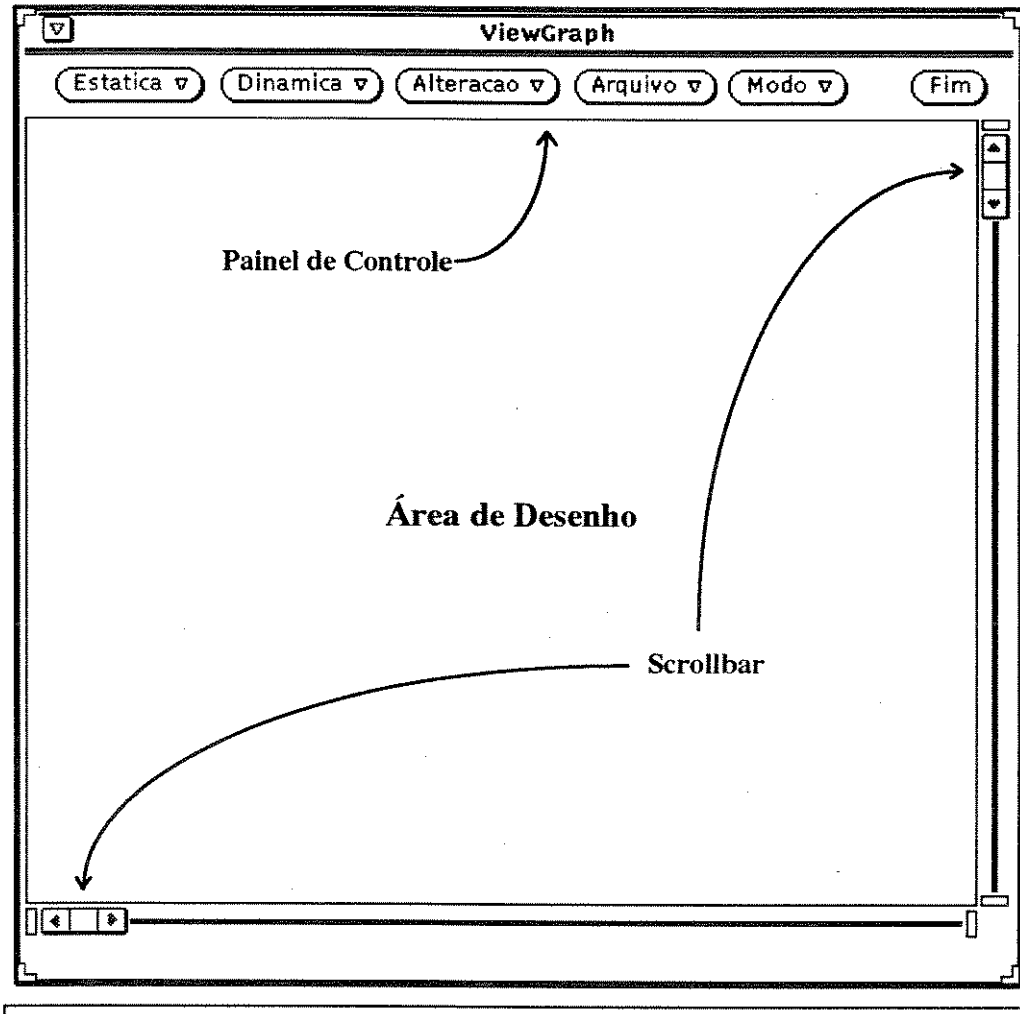


Figura 3.6. Tela Principal - ViewGraph

As opções do menu principal são : Estática, Dinâmica, Alteração, Arquivo, Modo e Fim; segue uma descrição de cada uma delas.

A opção Estática é acionada no caso do usuário desejar visualizar informação geradas na fase estática da POKE-TOOL; essas informações dizem respeito principalmente ao código fonte do programa. O sub-menu correspondente a essa opção é mostrado na Figura 3.7.

A opção dinâmica diz respeito às informações geradas na fase dinâmica da POKE-TOOL; acionando essa opção o usuário pode observar informações sobre a aplicação dos casos de teste. As opções contidas no sub-menu correspondente são mostradas na Figura 3.8.

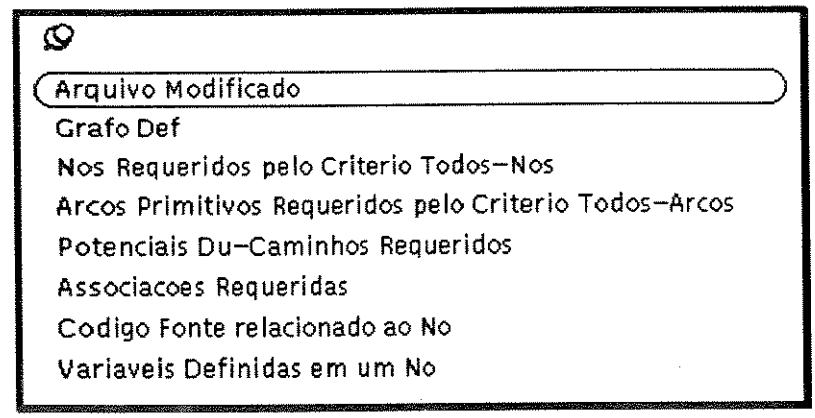


Figura 3.7. Opções do Sub-menu Estática

As duas opções anteriores encerram a parte de visualização de informações de teste geradas pela POKE-TOOL; os menus subseqüentes tratam da manipulação das características estéticas do grafo de programa. As opções possibilitam alterar posições de arcos ou nós do grafo, Figura 3.9-A; gravar configuração definida ou recuperar configurações anteriores, Figura 3.9-B; além de permitir visualizar o grafo em sua disposição completa ou resumida, Figura 3.9-C.

A opção Alteração deve ser acionada no caso de o usuário desejar alterar características estéticas do grafo de programa desenhado; acionando-a é mostrado um sub-menu com duas opções : Posição Nó e Posição Arcos; no caso de o usuário querer alterar a disposição de um nó ele deve ativar Posição Nó e indicar com o "mouse" qual o nó que deve ser reposicionado, diretamente na área de desenho, fornecendo em seguida a nova posição; a operação análoga deve ser aplicada no caso de mudança de posição de arcos.

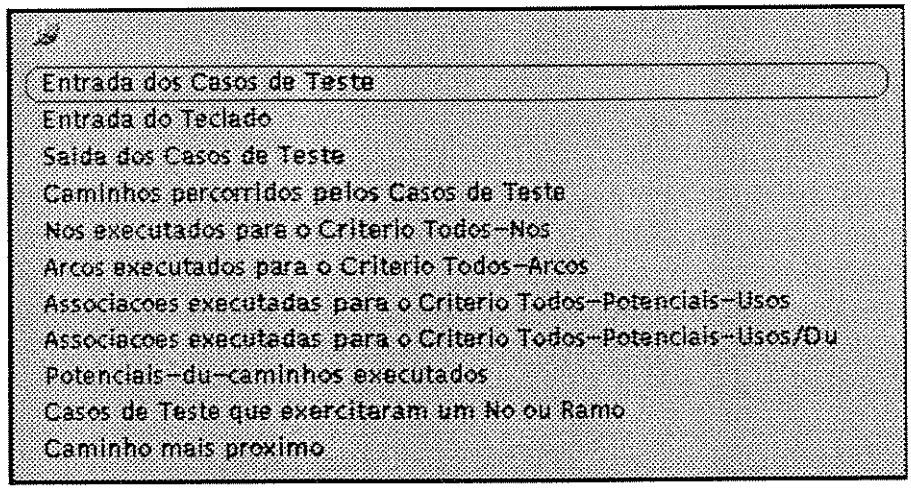


Figura 3.8. Opções do Sub-menu Dinâmica

Se alguma modificação for feita no desenho do grafo de programa ela pode ser gravada e posteriormente recuperada através da utilização das opções do sub-menu Arquivos, são elas :

Lê Arquivo : mudar o grafo de programa correntemente visualizado para um outro indicado pelo usuário.

Salva Grafo Corrente : gravar em disco as modificações feitas sobre o grafo de programa.

Lê Arquivo de Configuração : ler do disco uma configuração previamente gravada.

A opção Modo possibilita que o usuário escolha entre visualizar o grafo na forma resumida (nós com raio pequeno e sem numeração) ou na forma completa (nós com raio suficientemente grande para circundar a numeração).

No menu principal ainda existe a opção Fim que encerra a execução da ViewGraph.

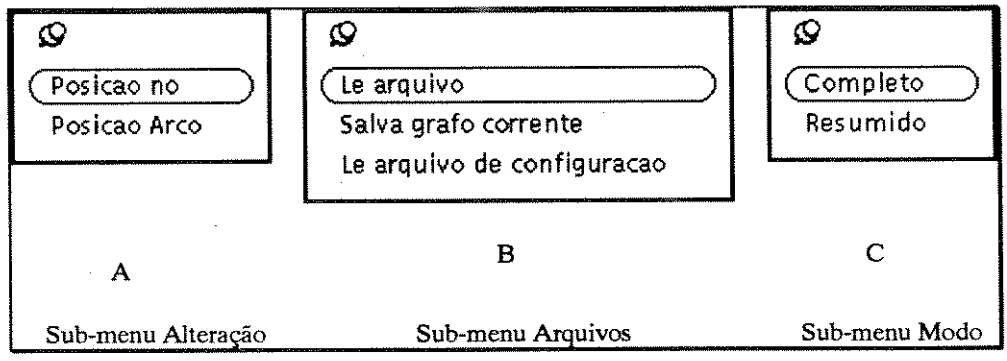


Figura 3.9. Três Sub-Menus do Menu Principal.

A interface apresentada aqui e a funcionalidade apresentada anteriormente, constituem a especificação para a ferramenta ViewGraph. No próximo capítulo são apresentados os algoritmos para resolver o problema da disposição gráfica dos grafos de programa e no Capítulo 5 são apresentadas os principais módulos implementados no protótipo atual da ViewGraph.

CAPÍTULO 4

VISUALIZAÇÃO DE GRAFOS DE PROGRAMA

Neste capítulo são apresentadas as características teóricas da visualização de grafos de programa, seus principais problemas e os algoritmos definidos no intuito de solucionar esses problemas. Também são apresentadas as características principais dos algoritmos propostos.

4.1 - O Problema da Visualização de Grafos de Programa

O processo de geração manual do grafo de programa parte de uma representação desse grafo numa forma textual, onde estejam indicados o número total de nós e as ligações entre eles.

A POKE-TOOL, apresentada no Capítulo 3, gera uma representação textual do grafo de programa gravada num arquivo com extensão .gfc, um exemplo é dado na Figura 4.1. No arquivo .gfc, o primeiro número representa o total de nós do grafo, em seguida cada número da primeira coluna da esquerda tem abaixo dele uma lista de números terminada por um 0, isso significa que o nó dado pelo número à esquerda liga-se a cada um dos nós dados pelos números da lista terminada pelo 0. Por exemplo na Figura 4.1 está indicado que existe um arco ligando o nó 1 ao nó 2 (1,2) e um arco ligando o nó 2 ao nó 3 (2,3) e assim por diante até o nó 15 que não tem nenhum sucessor.

A ordem dos números dada nos arquivos .gfc indica uma hierarquia entre os nós; ou seja, se o número do nó inicial for maior que o número do nó final, então o arco ligando esses dois nós será um arco de ciclo (Seção 2.1.2), por exemplo o arco (6,3) do grafo (em forma textual) da Figura 4.1.

```
15
1
 2 0
2
 3 0
3
 4 7 0
4
 5 6 0
5
 6 0
6
 3 0
7
 8 9 0
8
 7 0
9
 10 14 0
10
 11 12 0
11
 13 0
12
 13 0
13
 14 0
14
 2 15 0
15
 0
```

Figura 4.1. Exemplo de Arquivo .gfc

A característica da numeração do arquivo .gfc será aproveitada na geração automática dos grafos de programa, em duas situações principais : na classificação dos arcos do grafo, separação dos arcos de ciclo e dos arcos diretos e no posicionamento padronizado de algumas estruturas.

Os arcos dos grafos de programa podem ser separados em dois tipos, arcos de ciclo e arcos diretos; tomando-se como base a ordenação dos nós nos arquivos .gfc essa classificação é direta.

Algumas estruturas, como o IF-THEN-ELSE, devem sempre ser posicionadas da mesma forma para facilitar a visualização da estrutura por parte do usuário. Esse posicionamento deve ser tal que a parte relativa ao THEN fique sempre para a esquerda e a parte relativa ao ELSE fique para a direita. A ordenação dessas estruturas já é dada automaticamente pela POKE-TOOL, por meio da numeração dos nós do grafo (apresentada no arquivo .gfc). Observa-se na Figura 4.1 que o nó 10 liga-se ao nó 11 e ao nó 12, no caso dessa estrutura ser um IF-THEN-ELSE, o nó 11 corresponderia ao THEN e o 12 ao ELSE.

Tomando-se como exemplo o grafo de programa mostrado na Figura 4.1, ilustra-se o desenho manual do grafo de programa, como forma de apontar alguns de seus problemas.

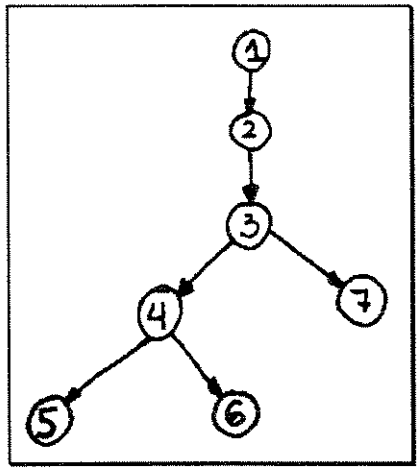


Figura 4.2. Problema no Posicionamento de IF-THEN

Observa-se na Figura 4.1 que o nó 1 liga-se somente ao nó 2, por isso tende-se a desenhá-los um abaixo do outro, como pode ser visto na Figura 4.2; o mesmo ocorre para o nó 2 que liga-se unicamente ao nó 3.

Já o nó 3 liga-se a outros dois nós, o 4 e o 7; para posicionar esses dois novos nós deve-se reservar um espaço entre eles, suficiente para que os subgrafos correspondentes não se intercalem. Como a priori não se sabe quão grande deve ser esse espaço, aparece aí o primeiro problema : reserva-se um espaço qualquer, baseando-se na intuição; se este espaço não for suficiente, o grafo deverá ser re-desenhado.

A mesma explicação dada para o nó 3 se aplica ao nó 4, esse nó se liga aos nós 5 e 6. Neste ponto o desenho manual do grafo de programa seria algo como o que é mostrado na Figura 4.2

No tratamento do nó 5 aparece o segundo problema : o nó 5 liga-se ao nó 6 mas como os dois nós foram posicionados logo abaixo do nó 4, essa estrutura é dificil de ser identificada no contexto de grafos de programa. Na verdade a estrutura que se inicia no nó 4 foi posicionada como se o comando do nó 4 fosse um IF-THEN-ELSE, mas na verdade ele é, apenas, um IF-THEN e o nó 6 é a saída dessa estrutura; portanto, uma opção mais indicada para o posicionamento desses nós seria o com os três nós em linha reta (veja Figura 4.3).

Observa-se também na Figura 4.3 que o posicionamento das estruturas iniciadas nos nós 7, 9 e 10, segue a mesma regra usada no nó 3. Agora, para o posicionamento do nó 11 tem-se um outro problema : o nó 11 liga-se ao nó 13; então, posiciona-se o nó 13 diretamente abaixo dele; mas o nó 12 também se liga ao nó 13, porque esse nó é saída de um IF-THEN-ELSE iniciado no nó 10; sendo assim a posição ideal para o nó 13 é entre a posição do nó 11 e a posição do nó 12 (como pode ser visto na Figura 4.4).

O quarto problema desse posicionamento aparece porque o nó 13 liga-se ao nó 14, mas esse último já havia sido posicionado abaixo do nó 9, ao lado do nó 10. O nó 14 é saída da estrutura DO iniciada no nó 2 e deve ser posicionado, para melhor refletir a estrutura de fluxo de controle do programa, abaixo do nó 13, como pode ser visto na Figura 4.5.

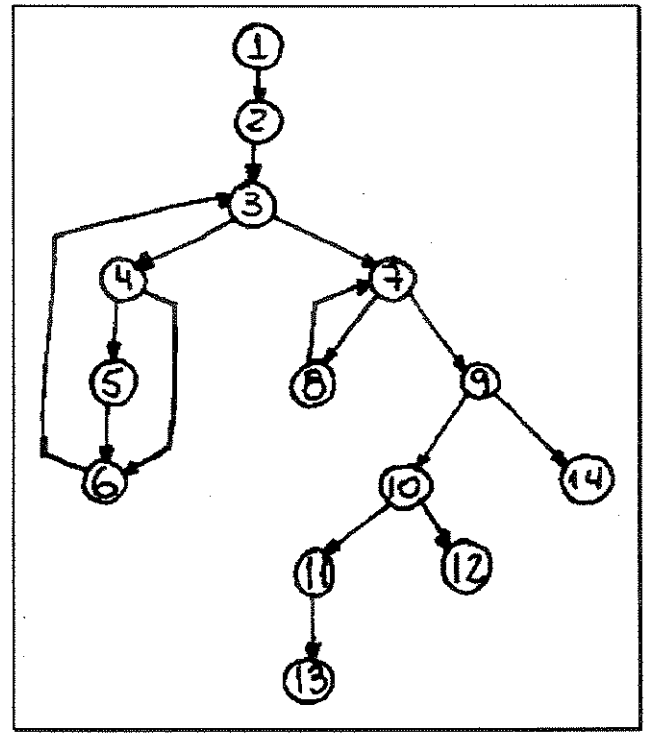


Figura 4.3. Problema no Posicionamento de nó de Saída

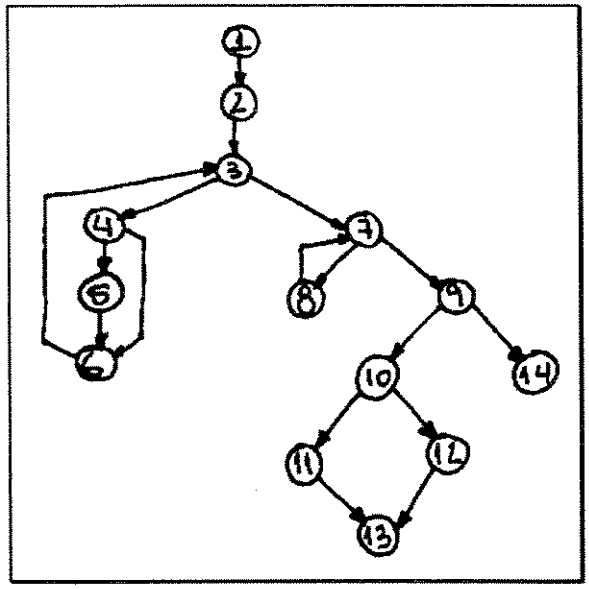


Figura 4.4. Problema nó de Saída de IF-THEN

O grafo de programa apresentado na Figura 4.5 corresponde ao posicionamento final gerado para o grafo de programa da Figura 4.1, esse posicionamento foi gerado manualmente, seguindo a idéia de não utilizar informações a respeito dos comandos internos a cada nó para gerar a disposição gráfica (apesar de terem sido citados alguns comandos nas explicações, eles não são necessariamente utilizados na geração manual do grafo de programa); a mesma idéia é utilizada na implementação dos algoritmos apresentados mais adiante nesse capítulo. Na Figura 4.7 mostra-se o mesmo grafo desenhado automaticamente.

Utilizando-se de informações a respeito dos nós pode-se gerar disposições gráficas como a que é mostrada na Figura 4.6; essa abordagem não foi utilizada nos algoritmos apresentados nesse capítulo para possibilitar que os mesmos algoritmos aplicados ao desenho de grafos de programa pudessem ser utilizados, a menos de poucas modificações, em outras estruturas com relacionamento hierárquico.

Os problemas apresentados até agora dizem respeito principalmente à determinação da coordenada Y de cada nó; a coordenada X não apresentou problemas, pois o grafo de programa considerado não possui subgrafos largos. Claro que a determinação automática da coordenada X, mesmo em grafos simples, deve ser tratada de forma sistemática, e é na presença de estruturas *case* que essa determinação se torna mais crítica. O Tratamento dessas estruturas é mostrado mais adiante nesse capítulo.

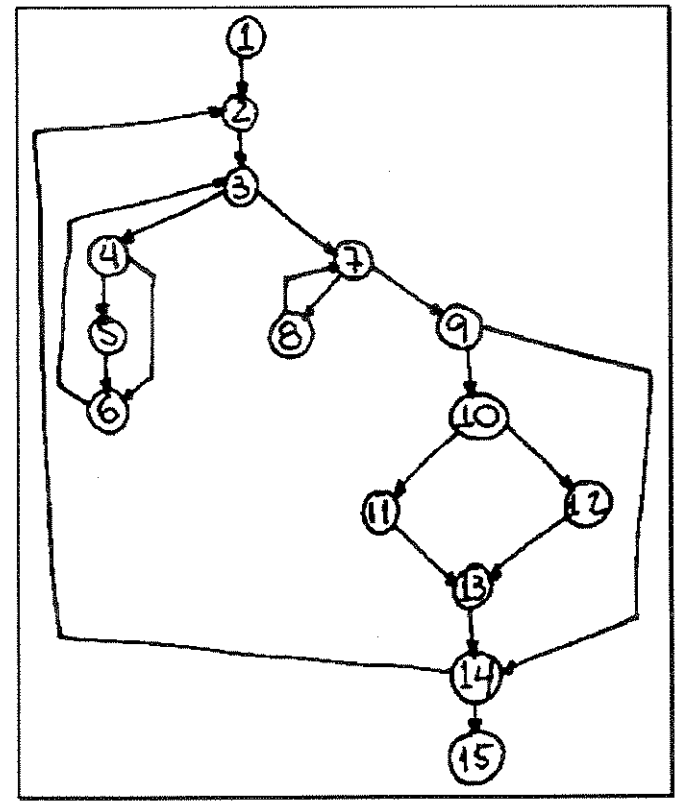


Figura 4.5. Posicionamento Manual Final

Na geração manual dos grafos de programas o roteamento dos arcos não apresenta muitos problemas, pois como tem-se um visão geral da disposição dos nós, achar o melhor caminho para um arco não se torna uma tarefa difícil de ser realizada. Mas na geração automática dos grafos de programa o roteamento dos arcos deve ser tratado de forma sistemática, objetivando minimizar os cruzamentos.

Na próxima seção encontra-se a

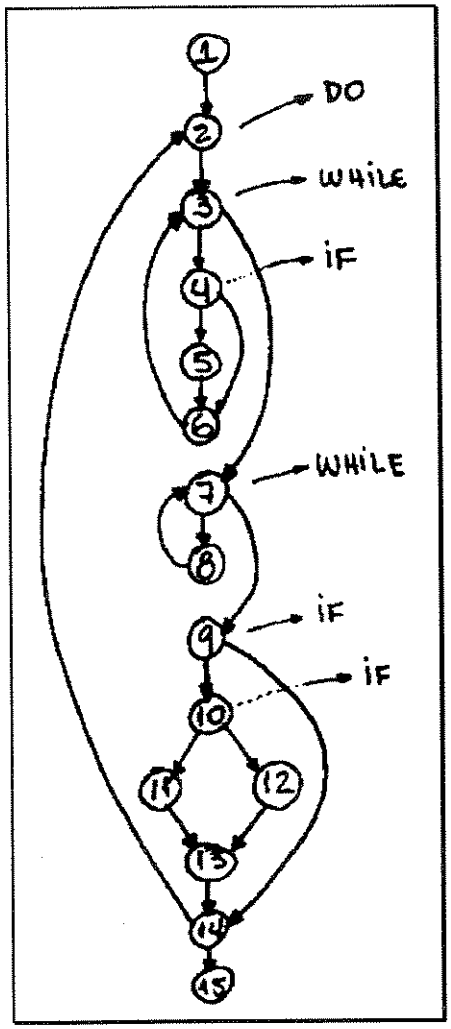


Figura 4.6. Outro Exemplo de Representação

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 de grafos de programa.

4.2 - Soluções Algorítmicas

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, apresentadas a seguir.

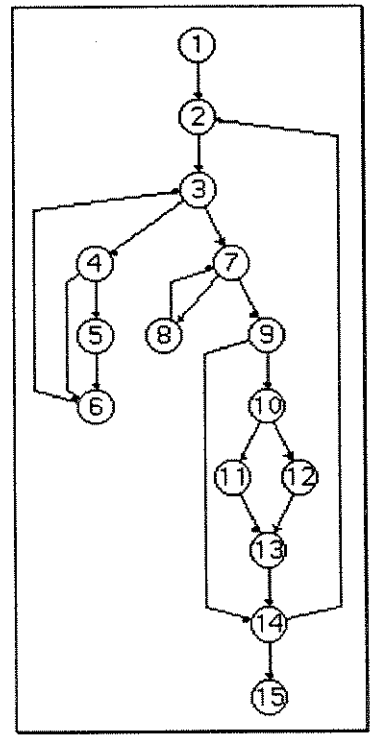


Figura 4.7. Posicionamento Automático do entab.gfc

4.2.1 - Posicionamento dos Nós

O nível de cada nó, definido para árvores no Capítulo 2, 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 (veja a Figura 4.8).

A determinação da coordenada Y a partir do nível dos nós é dada pela seguinte fórmula :

$$Y(n) = D + nível(n) \times E + ((nível(n)-1) \times 2R) + R$$

Y(n) = coordenada Y para o nó n, equivale à coordenada Y do centro da circunferência que representa o nó n
D = deslocamento vertical inicial
nível(n) = nível do nó n
R = raio da circunferência que representa um nó
E = espaçamento entre os níveis do desenho

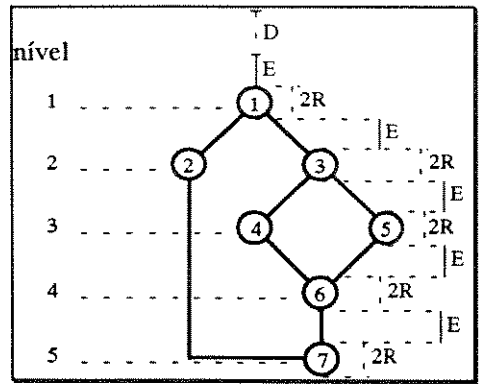


Figura 4.8. Uso do Nível na Determinação da Coordenada Y

Duas variáveis importantes são apresentadas na Figura 4.8, a variável D (deslocamento vertical inicial) e a variável E (espaçamento entre níveis). Aumentando-se ou diminuindo-se o valor da variável D pode-se mover o grafo na direção Y no dispositivo de saída; essa variável pode assumir valores tanto positivos quanto negativos. A variável E possibilita afastar ou aproximar os nós do grafo, na direção Y, e só assume valores positivos

A fórmula para o cálculo da coordenada Y, dada acima, pode ser melhor entendida considerando-se por exemplo o grafo de programa da Figura 4.8; a coordenada Y de um nó é na prática a coordenada Y do centro da circunferência que representa esse nó que é dada pela soma da distância D com o raio R, valores que independem do nível do nó em questão, acrescidos do valor que depende do nível do nó. Esse valor é achado da seguinte forma : para um nó no nível n tem-se n vezes o valor de E e (n-1) vezes o valor de 2R (diâmetro).

Os valores das variáveis D, E e R são definidos dependendo do aspecto estético que se deseja obter do grafo que estiver sendo desenhado; portanto, o problema de determinar a coordenada Y dos nós do grafo resume-se a determinar os níveis desses nós.

No caso de grafos de programas 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; ou seja, o nó x tem mais de dois filhos, esse nó 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.

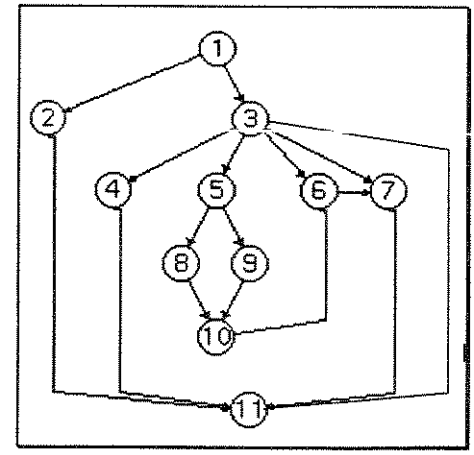


Figura 4.9. Estrutura Case sem "Otherwise"

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, definidos nesse capítulo, em outras estruturas com relacionamento hierárquico.

A determinação do nó de saída é feita da seguinte forma : é considerado nó de saída da estrutura case o

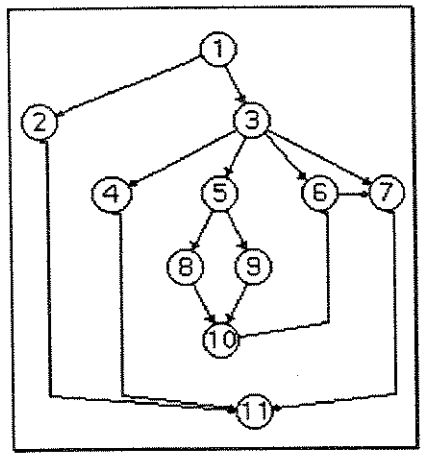


Figura 4.10. Estrutura Case com "Otherwise"

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.

Essa abordagem simples resolve a maioria dos casos apresentados em programas reais, por exemplo no caso de estrutura case sem "otherwise" (Figura 4.9), pois nesse caso o nó de entrada liga-se diretamente com o nó de saída; e no caso de estrutura case com "othewise" constituido de apenas um nó (Figura 4.10). Um problema aparece quando o "otherwise" é constituido de um conjunto de nós, normalmente um IF-THEN ou um IF-THEN-ELSE, o algoritmo determinará erroneamente o nó equivalente ao THEN como sendo o nó de saída, Figura 4.11.

A Figura 4.8 mostra um exemplo de um nó com mais de um pai, o nó 7, filho dos nós 2 e 6, sendo que esses pais estão em níveis diferentes entre si, o nó 2 está no nível 2 e o nó 6 está no nível 4. Esse é um exemplo que ilustra o problema número 1, para resolvê-lo basta definir que o nível de um nó é uma unidade maior que o nível do seu pai de maior nível.

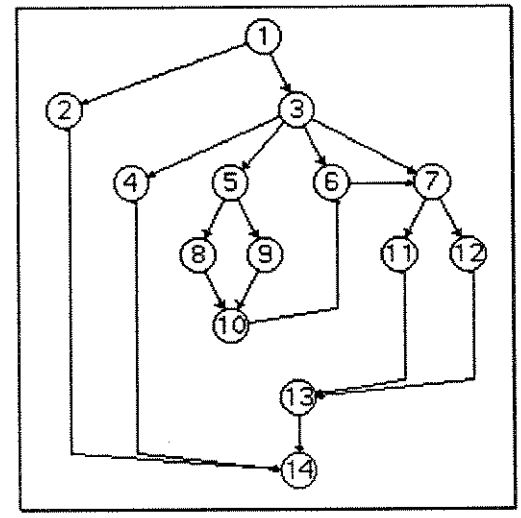


Figura 4.11. Estrutura Case com "Otherwise" Complexo

A Figura 4.12-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, no caso, possua um comando análogo ao comando "break" da linguagem C.

Se um desses blocos internos não terminar com o comando "break", surgirá um arco entre blocos vizinhos na estrutura case. Assim tem-se que um nó interno ao case é pai de outro nó interno ao case; esses nós, segundo a regra para calcular os níveis citada acima, deveriam ter níveis no mínimo consecutivos, como é visto na Figura 4.12-B. Esse desenho desrespeita a definição estética para a estrutura case dada no Capítulo 2; o desenho mais

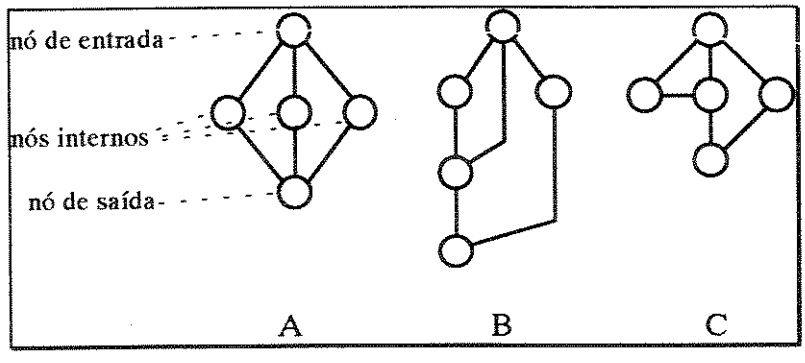


Figura 4.12. Problemas da Estrutura Case

indicado dessa estrutura seria o que é apresentado na Figura 4.12-C, com os nós internos ao case sobre um mesmo nível.

A Figura 4.13-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 caso 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.

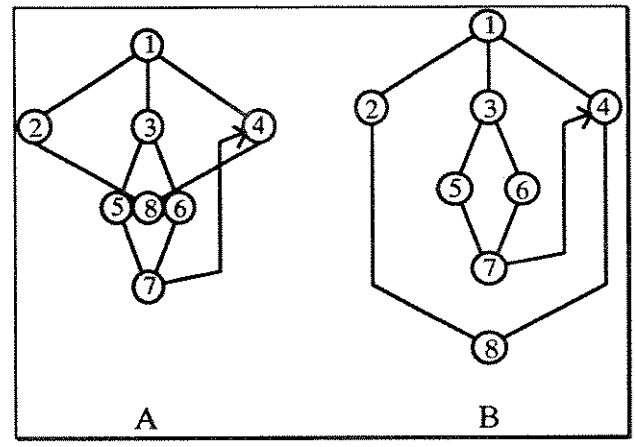


Figura 4.13. Posicionamento do Nó de Saída do Case

Observa-se na Figura 4.13-B que o nível correto para o nó 8 é 5 um a mais que o nível do nó 7, apesar desses 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 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(∀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, ou seja, aqueles que são filhos do nó de entrada e que não são o nó de saída 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; viu-se como determinar os níveis dos nós e como aplicá-los no cálculo da coordenada Y através de uma fórmula que leva em conta alguns parâmetros do desenho final. Passa-se agora ao estudo da determinação das coordenadas X dos nós.

O tratamento realizado na determinação das coordenadas X dos nós é similar ao tratamento dado à determinação das coordenadas Y no que diz respeito ao armazenamento dos valores calculados; serão armazenados as posições dos nós livres de parâmetros de escala e deslocamento; sobre essas posições serão aplicadas uma função X(n) que as transforma na coordenada final do nó n. A função X(n) é similar à função Y(n), dada anteriormente, considerando-se nesse caso as posições na coordenada X.

Na determinação da coordenada X de cada nó surge um problema : como saber a priori qual será o espaço ocupado por cada subgrafo descendente de um dado nó? Para solucionar esse problema define-se uma variável denominada *Escopo*; informalmente ela guarda o espaço que deve ser reservado abaixo de cada nó para que seus descendentes possam ser posicionados sem haver sobreposição. Com essa variável definida é aplicado o algoritmo de posicionamento mostrado a seguir.

Algumas funções são necessárias na determinação do escopo, são elas :

- é_nó_folha(n) : Função que verifica se o nó n é nó folha. É considerado nó folha aquele nó que não possui descendentes em níveis maiores que o 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)=∑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 a semente da recursão para o cálculo do escopo, indica que se o nó é nó folha o seu escopo vale 1. A Regra RE2 faz com 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 4.14 mostra o valor do escopo de cada nó para o programa compress.c (apresentado no Apêndice A) calculado segundo o algoritmo acima. Observe 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 3 para o cálculo do escopo, os nós 11 e 12 deveriam ter escopo = ½, 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 2 o seu escopo é o somatório 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 variável escopo é apenas um valor auxiliar na definição do posicionamento na coordenada X dos nós; um tratamento mais completo é definido a seguir, pois 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 precisa ser deslocado dentro do seu escopo, pois não tem nenhum vizinho.

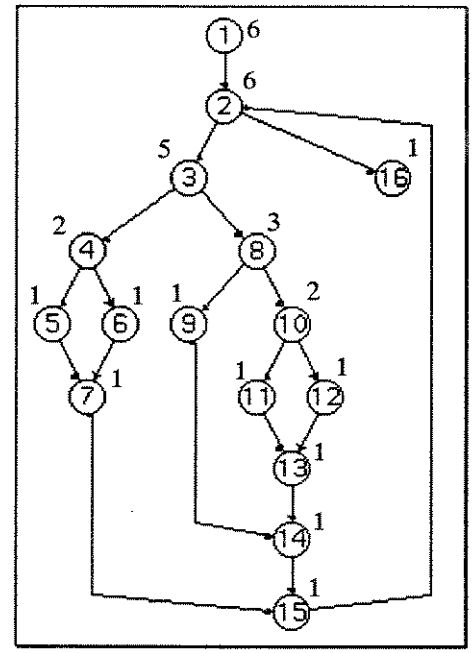


Figura 4.14. Escopo dos Nós de compress.c

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 sua esquerda recebem valores negativos e as dos nós à sua direita recebem valores positivos.

O algoritmo de posicionamento dos nós é definido a seguir; ele leva em conta o cálculo dos níveis e do escopo 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. posição [nó]=média_posição_pais_no_nível_de_cima(nó) :- filho_único (no),
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 de pai e ele não tem outros pais no nível anterior ao dele. Este é o caso, por exemplo, do nó 14 da Figura 4.14; 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, é o que diz a Regra RP3; esse é o caso, por exemplo, dos nós 7 e 13 da Figura 4.14.

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.

O exemplo da Figura 4.15 demonstra a aplicação da Regra RP4, que atua da seguinte forma : 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 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.

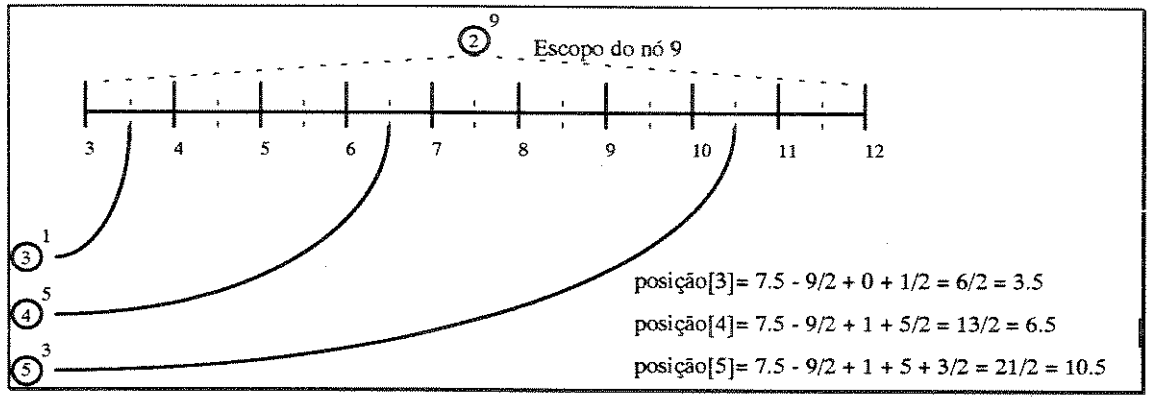


Figura 4.15. Nós Espalhados no Escopo do Pai

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.

4.2.2 - 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, $0 \leq delta\_nível \leq 1$.
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 na grade de posicionamento.

2. *Arcos com Opções Múltiplas* : Possuem vários caminhos possíveis na grade de posicionamento.

3. *Arcos sem Opção* : Não possuem nenhum caminho possível na grade de posicionamento.

Estratégia da Abordagem.

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

- Arcos diretos e curtos têm posicionamento trivial; para esses arcos não é necessário calcular o roteamento nem tão pouco guardar as 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, dado anteriormente, gera uma grade de posicionamento ocupado 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.

- Arcos de ciclo e curtos têm também posicionamento trivial, apesar de ter 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.

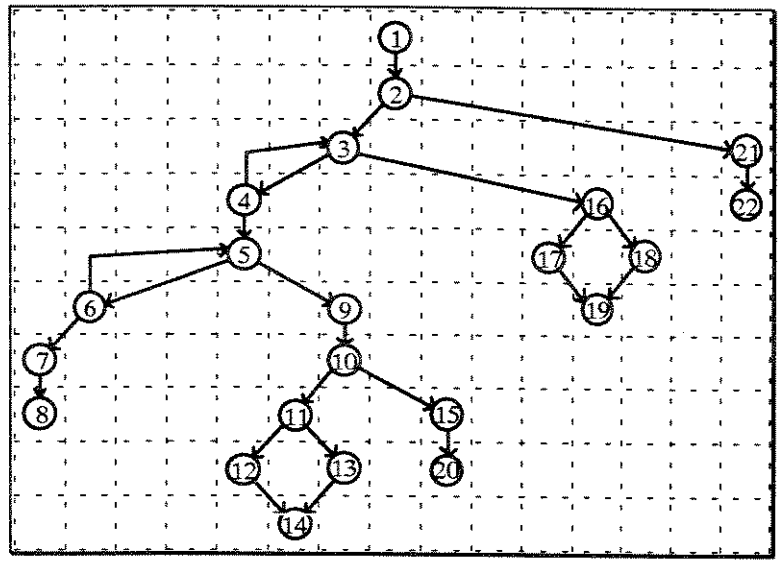


Figura 4.16. Grade de Posicionamento Ocupado por Nós e Alguns Arcos

A Figura 4.16 mostra exemplos dos dois conjuntos de arcos anteriores posicionados numa grade de posicionamento ocupado por alguns nós.

- Para os outros arcos devem ser encontradas opções de posicionamentos 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 a possibilidade de posicioná-los sem cruzamentos.

Observa-se na Figura 4.17 o que pode ocorrer se o roteamento dos arcos for feito sem a ordenação pelo delta_nível; o arco (5,11) tem delta_nível menor que o arco (3,12) e deveria portanto ter sido posicionado primeiro; como isso não ocorreu, o arco (3,12) ocupou a posição mais interna que deveria ser do arco (5,11), provocando o cruzamento; o posicionamento mais indicado desse grafo pode ser visto na Figura 4.18.

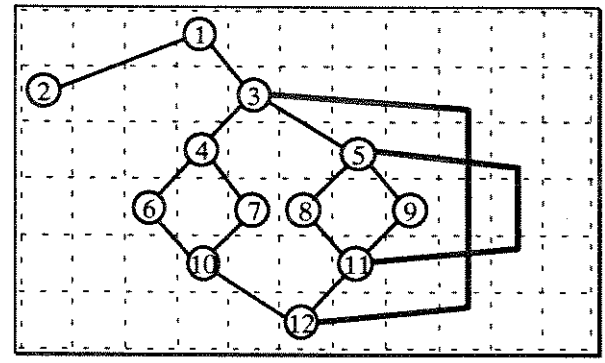


Figura 4.17. Exemplo sem Ordenação

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

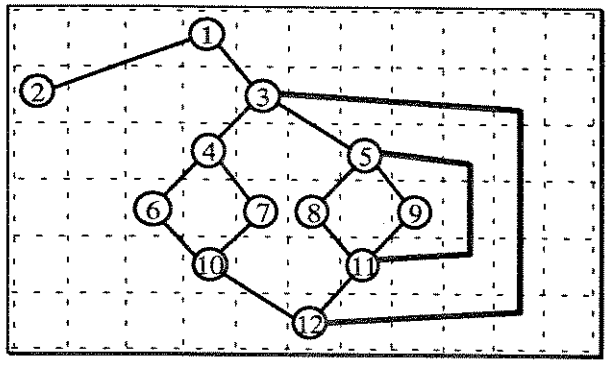


Figura 4.18. Exemplo com Ordenação dos Arcos

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 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 mais acima definem os caminhos possíveis iniciais.

RO2. Os subgrafos vizinhos ao nó mais acima restringem as possibilidades.

RO3. A partir do nó mais acima, percorre-se o grafo redefinindo-se as opções.

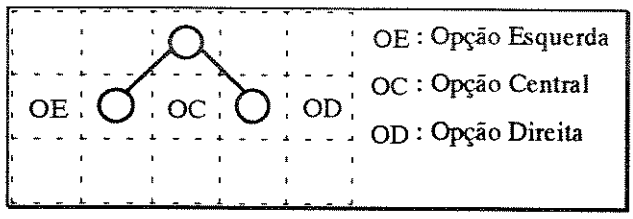


Figura 4.19. Determinação de Opções Possíveis

Segundo a Regra RO1, as opções iniciais de roteamento de um arco são definidas pelo nó que estiver mais acima na estrutura do grafo, não importando se este nó é o nó inicial ou o nó final do arco. Observa-se na Figura 4.19 que três opções iniciais foram definidas : a opção pela esquerda, a opção central e a opção pela direita.

Na Figura 4.20 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.

A Figura 4.21 mostra a dinâmica da definição das opções para o arco (20,2). Inicialmente são definidas a opção da

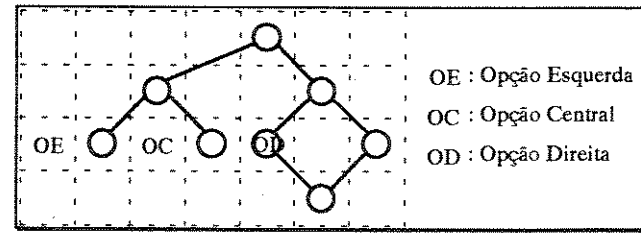


Figura 4.20. Os Vizinhos Restringem as Opções

esquerda, as centrais e a da direita. Observe que a opção da esquerda vai se deslocando cada vez mais para a esquerda à medida que o subgrafo da esquerda vai sendo percorrido. No nível 9 essa opção desaparece pois é atingido o mesmo nível do nó 20 sem haver possibilidade de ligação pelo lado esquerdo. A opção da direita e uma das opções centrais permanecem até o fim, e formam o conjunto de opções de posicionamento do arco (20,2).

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

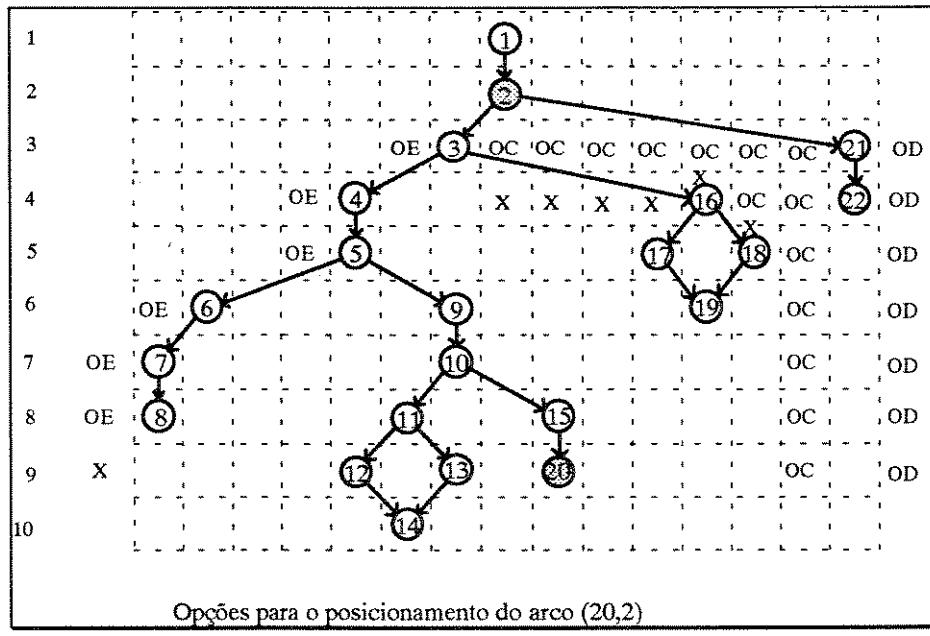


Figura 4.21. Exemplo Definição de Opções para os Arcos

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 com que outro arco fique 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 ou 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. Essa estrutura pode ser vista na Figura 4.22.

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 .gfc do grafo de

programa; 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 posição 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 dada anteriormente 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.

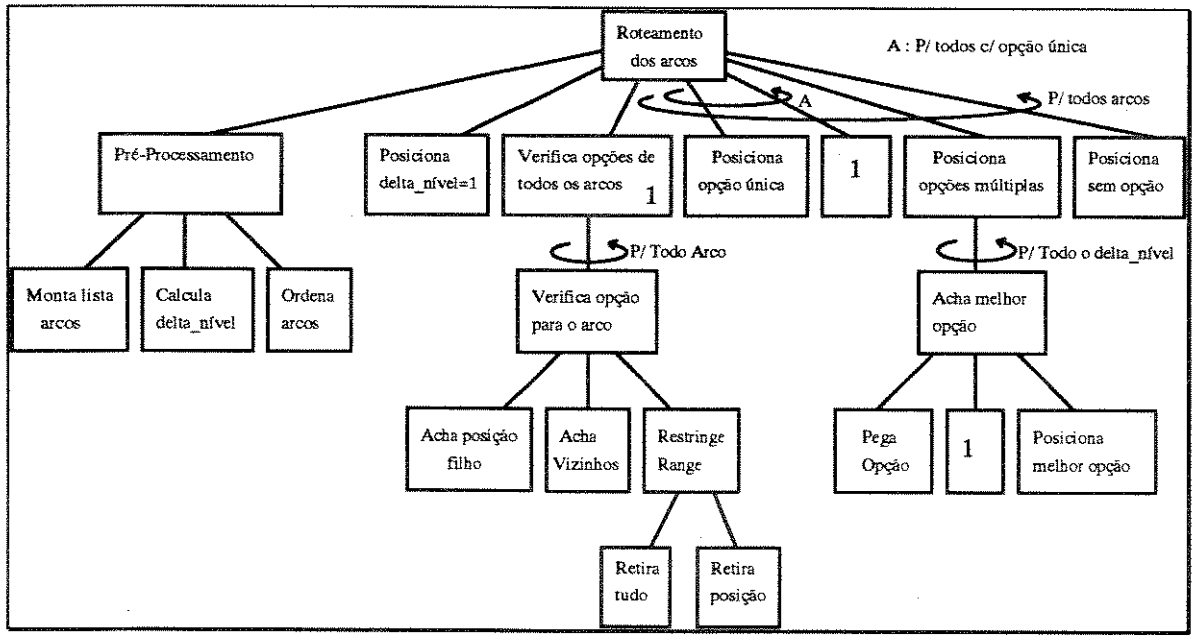


Figura 4.22. Estrutura do Algoritmo de Roteamento de Arcos

4.3 - Características da Solução

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

1. Os nós são sempre posicionados de acordo com a sua numeração no arquivo .gfc; isso, como foi visto, leva ao posicionamento padronizado de estruturas como o IF-THEN-ELSE. Esse posicionamento é mantido mesmo quando acarretar o cruzamento de arcos; observa-se na Figura 4.23 que se o posicionamento dos nós 3 e 4 fosse invertido não haveria o cruzamento do arco (3,11).

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 4.12-C). Outra restrição é a própria determinação dessa estrutura que considera qualquer nó com mais de dois filhos um nó de entrada de case.

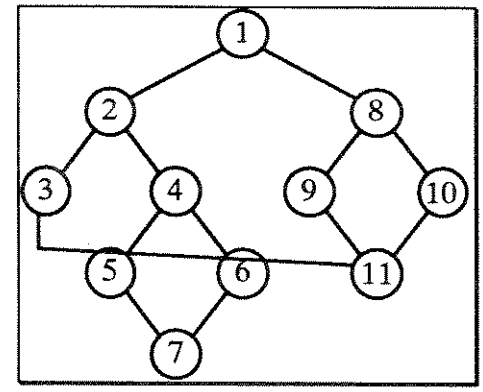


Figura 4.23. Disposição Fixa dos Nós

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. A entrada fornecida para o programa que implementa os algoritmos dados nesse capítulo pode ser editada manualmente mas deve seguir o padrão definido para os arquivos .gfc gerados automaticamente pela POKE-TOOL.

5. A numeração dos nós no arquivo .gfc é muito importante e considerada em diversas situações como na definição dos arcos de ciclo, nós à esquerda, nós vizinhos, e em muitas rotinas auxiliares usadas na implementação dos algoritmos mostrados nesse capítulo.

6. É considerada a existência de apenas um nó de entrada, consideração razoável para grafos de programas.

Com as restrições citadas acima o algoritmo funciona 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, a 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 mostrada nesse capítulo esse problema é eliminado através do uso da variável denominada **escopo**; com essa variável, pode-se determinar diretamente as posições dos nós sem haver repetição de cálculos.

- O posicionamento relativo gerado por essa abordagem possibilita que um mesmo grafo de programa possa ser visualizado com diversas aparências diferentes no dispositivo de saída, bastando para isso alterar-se parâmetros de escala, espaçamento e deslocamento, diretamente no módulo de desenho.

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

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

Para o caso de árvores deve-se retirar a parte que trata as estruturas case na determinação dos níveis dos nós, retirar a parte de roteamento dos arcos e modificar o cálculo do escopo de forma a levar em conta a profundidade de cada sub-árvore, o que levará à definição de um posicionamento com ocupação mínima de espaço.

As outras estruturas com relacionamento hierárquico necessitam de adaptações que levem em conta a presença de "labels" nos nós, além de tratar a possível presença de múltiplos nós de entrada.

As principais contribuições dessa abordagem aparecem na geração de um posicionamento independente do dispositivo de saída e no uso da variável escopo que otimiza a definição do posicionamento dos nós; o roteamento dos arcos segue um conceito definido na literatura como *envoltória*, essa envoltória é ampliada ou anulada à medida que o grafo for sendo percorrido; essa estratégia tem gerado grafos de programas com poucos cruzamentos, como pode ser visto nos exemplos do apêndice A.

CAPÍTULO 5

IMPLEMENTAÇÃO DOS ALGORITMOS DE VISUALIZAÇÃO DE GRAFOS DE PROGRAMA

Neste capítulo são apresentadas algumas características da implementação do protótipo atual da ViewGraph incluindo os algoritmos de visualização apresentados no Capítulo 4 e o módulo de desenho que inclui a visualização de código fonte e variáveis relacionados a um nó.

5.1 - Principais Estruturas de Dados

A primeira estrutura de dados utilizada pelo programa é a lista de adjacências que guarda as ligações entre os nós do programa; essa lista é montada a partir do arquivo .gfc e armazenada em duas variáveis :

não_loop[*n*] : é um vetor de ponteiros que para cada *n* armazena uma lista de descendentes de *n* sendo que esses descendentes são ligados por arcos diretos.

loop[*n*] : é um vetor de ponteiros que para cada *n* armazena uma lista de descendentes de *n* sendo que esses descendentes são ligados por arcos de ciclo.

Essas duas variáveis são montadas dessa forma para que, desde o início, os arcos diretos e de ciclo fiquem separados; essa abordagem é seguida pois os arcos diretos e de ciclo serão utilizados separadamente em vários pontos do programa.

As seguintes variáveis guardam informações calculadas durante as várias fases do programa :

num_NÓ : guarda o número de nós do grafo de programa.

num_arcos : número de arcos do grafo de programa.

saída[*n*] : sendo *n* o nó de entrada de uma estrutura case, saída[*n*] armazena o nó de saída desse case.

nó_nível[*n*] : nível de cada nó *n* do grafo de programa.

escopo[*n*] : escopo de cada nó *n*.

posição_nó[*n*] : posição na horizontal do nó *n*. A posição vertical é calculada diretamente pelo nível de *n*, ver Capítulo 4.

lista_arcos[*p*] : vetor onde cada posição *p* armazena dois campos, um indicando o nó inicial do arco e o outro indicando o nó final.

ordem[*p*] : vetor que indica a ordem em que o vetor lista_arcos deve ser acessado para se obter os arcos em ordem crescente de delta_nível.

ja_pos[*p*] : armazena 1 se o arco lista_arcos[*p*] já tiver sido posicionado.

opções[*a*] : armazenas as opções da esquerda, centrais e da direita para o arco *a*.

delta_nível : armazena qual delta_nível está sendo tratado em um dado momento; isso para tratar todos os arcos de um delta_nível para depois passar para o próximo delta_nível, ver Capítulo 4.

5.2 - Estruturação do Programa

Na Figura 5.1 é apresentada a estrutura implementada da ViewGraph; observa-se que o módulo Posiciona Arcos não está detalhado; a sua implementação corresponde exatamente ao que foi proposto na Figura 4.19.

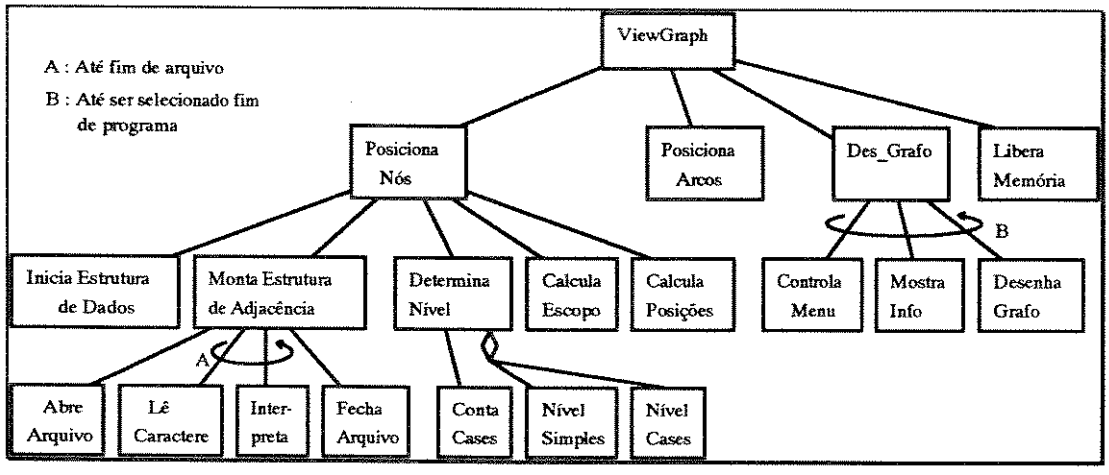


Figura 5.1. Estrutura da ViewGraph

Desses módulos os mais importantes são : o Módulo Posiciona Nós, o Módulo Posiciona Arcos e o Módulo Des_Grafos; esses módulos estão detalhados a seguir e o código fonte de algumas funções é também apresentado.

5.2.1 - Módulo Posiciona Nós

Esse módulo é dividido em cinco etapas : a primeira inicia as estruturas de dados usadas no posicionamento dos nós; a segunda lê o arquivo .gfc e monta a lista ligada contendo as ligações entre os arcos do programa; as outras três são : Determina Nível, Calcula Escopo e Calcula Posições descritas a seguir.

Módulo Determina Nível : A determinação do nível pode ser executada de duas maneiras diferentes dependendo do grafo conter estruturas do tipo case ou não. Inicialmente é contado o número de estruturas cases do programa e a partir daí decide-se qual módulo será executado : ou o Módulo Nível Simples, que calcula o nível dos nós no caso de não existirem estruturas do tipo case, ou o Módulo Nível Cases, que trata grafos de programa com estruturas case.

A seguir é mostrado o código fonte da Função *Determina_Nível*; observa-se que esta função recebe como parâmetro de entrada a variável *não_loop*, mas não recebe a variável *loop*; isso ocorre porque para o cálculo dos níveis não são considerados os arcos de ciclo (que são armazenados na variável loop).

```
determina_nivel(Apont nao_loop[MAX_NO],int num_NO,int no_nivel[MAX_NO],
                int saida[MAX_NO])
{
 int n_case;
 n_case=conta_numero_cases(nao_loop,saida,num_NO);
 if (n_case==0){
  det_nivel_simples(nao_loop,num_NO,no_nivel);
 }else{
```

```
  det_nivel_case(nao_loop,num_NO,no_nivel,saida);
 }
 return;
} /* determina_nivel */
```

A Função *Conta_Número_Cases*, chamada pela Função *Determina_Nível*, verifica o "fan out" (número de filhos) de cada nó; são considerados na contagem de estruturas case os nós com mais de dois filhos. O código fonte dessa função é mostrado a seguir. Observa-se na chamada da Função *Calcula_Fan_Out*, que lhe é passada, somente a variável não_loop; isso porque são considerados na contagem das estruturas case apenas os filhos ligados por arcos diretos.

```
conta_numero_cases(Apont nao_loop[MAX_NO], int saida[MAX_NO], int num_NO)
{
 int i,n=0,fan;
 for(i=0; i<num_NO; i++){
  fan=calcula_fan_out(i,nao_loop,num_NO);
  if(fan>2)
    n++;
 }
 return n;
} /*conta_numeros_case*/
```

A seguinte abordagem é usada no Função *Det_Nível_Simples*: para cada nó do grafo são percorridos todos os seus filhos, se o nível de cada um desses filhos for menor que o nível do pai mais 1, o nível do filho recebe esse valor. Inicialmente o valor dos níveis é 0, com exceção do nó 1 cujo nível vale 1.

```
det_nivel_simples(Apont nao_loop[MAX_NO],int num_NO,int no_nivel[MAX_NO])
{
 struct pont *filho;
 register int i;
 int no_atual=1;
 no_nivel[1]=1;
 do{
  filho=nao_loop[no_atual].prox; /* Primeiro filho de no_atual */
  while(filho!=NULL){ /* Para todo filho de no_atual */
   if (no_nivel[filho->info]<no_nivel[no_atual]+1)
        no_nivel[filho->info]=no_nivel[no_atual]+1;
   filho=filho->prox;
  }
  no_atual++;
 } while (no_atual <= num_NO); /* Para todos os nos do grafo */
 return;
} /*det_nivel_simples*/
```

A idéia utilizada na implementação do Função *Det_Nível_Case* é percorrer o grafo de forma a identificar os nós de entrada de estruturas case; a cada nó de entrada é calculado o nó de saída da estrutura correspondente. Quando um nó de saída do case é encontrado, calcula-se o nível máximo dos nós internos a essa estrutura; o nó de saída recebe o nível máximo encontrado mais 1.

Uma pilha de nós é mantida à cada nó tratado armazenam-se os seus filhos nessa pilha e o próximo nó a ser tratado também é retirado da pilha; isso faz com que o grafo seja percorrido numa ordem que evita problemas na definição dos níveis.

```
det_nivel_case(Apont nao_loop[MAX_NO],int num_NO,int no_nivel[MAX_NO],
                    int saida[MAX_NO])
{
 struct stk {
  struct pont *node;
  int pai;
 } pilha[30];
 struct pont *filho, *tmp;
 int topo=0, fan, no, inicio=0, pai, max;
 pai=1;
 tmp=malloc(sizeof(struct pont));
 filho=nao_loop[1].prox;
 tmp=filho;
 tmp=tmp->prox;
 while(tmp!=NULL){ /* Empilho os filhos do no' inicial */
  pilha[topo++].node=tmp;
  pilha[topo-1].pai=pai;
  tmp=tmp->prox;
 }
 while((filho!=NULL)&&(topo>=0)){
  while(filho!=NULL){
   if(inicio=e_no_de_saida(filho->info,saida)){
    max=max_nivel(inicio,filho->info,nao_loop,no_nivel);
    if(no_nivel[filho->info]<max+1)
     no_nivel[filho->info]=max+1;
    inicio=0;
   }else{
    fan=pai_fan_out(filho->info,nao_loop,num_NO,&no);
    if (no_nivel[filho->info]<no_nivel[pai]+1)
     if (fan<=2)
      no_nivel[filho->info]=no_nivel[pai]+1;
     if(fan>2){ /* Identifiquei pai como inicio de case */
      no_nivel[filho->info]=no_nivel[no]+1;
     if (saida[no]==0){
      saida[no]=qual_no_de_saida(no,nao_loop,num_NO);
      calcula_nivel_interno(no,saida[no],nao_loop,no_nivel,num_NO);
     }
    }
   }
   pai=filho->info;
   filho=nao_loop[filho->info].prox;
   if(filho!=NULL){
    if(no_nivel[pai]==no_nivel[filho->info]){
     pai=filho->info;
     filho=nao_loop[filho->info].prox;
    }else{
     tmp=filho->prox;
     while(tmp!=NULL){
      pilha[topo++].node=tmp;
      pilha[topo-1].pai=pai;
```

```
        tmp=tmp->prox;
       }
      }
     }
    }
    if(topo>0){ /* Retiro um nó da pilha */
     filho=pilha[--topo].node;
     pai=pilha[topo].pai;
    }
  }while((topo>=0)&&(filho!=NULL));
  return;
} /*det_nivel_case*/
```

A Função *Calcula_Escopo* que calcula o escopo dos nós é apresentada a seguir, observa-se que existe uma variável soma_aux que não está sendo utilizada mas que pode ser opcionalmente, usada para alterar-se a aparência final do grafo. Essa modificação (indicada no programa fonte) faz com que nós com filhos em níveis não consecutivos ao dele, ou seja, ligados por arcos longos, tenham o valor do escopo uma unidade maior; isso faz com que se aumente o espaço reservado para o posicionamento dos arcos.

```
calcula_escopo(int no,Apont nao_loop[MAX_NO],int no_nivel[MAX_NO],int num_NO,
                    int escopo[MAX_NO])
{
 int soma=0, soma_aux=0, no_filho, n;
 struct pont *filho;
 filho=nao_loop[no].prox;
 while (filho!=NULL){
  if (no_nivel[filho->info]==(no_nivel[no]+1)){
   soma += calcula_escopo(filho->info,nao_loop,no_nivel,num_NO,escopo);
  } else {
   if(no_nivel[filho->info]>no_nivel[no]+1)
        soma_aux++;
  }
  no_filho=filho->info;
  filho=filho->prox;
 }
 if ((so_um_filho(no,nao_loop))&&
    (n=outros_pais_no_nivel_de_cima(no,no_filho,nao_loop,no_nivel)))
  soma=(int)((soma/(n+1))+.5); /*n+1 pois a funcao volta um pai a menos*/
 if (soma==0){
  soma=1;
 } else {
  if (soma_aux) /* soma++ */; /* Variacao que altera a aparencia do Grafo */
 }
 escopo[no]=soma; /* atualiza o escopo do no em questao */
 return soma; /* joga o resultado para somar-se ao pai */
} /*calcula_escopo*/
```

A função que calcula as posições horizontais dos nós do grafo é mostrada a seguir; essa função implementa o algoritmo de posicionamento de nós dado no Capítulo 4.

```
calcula_posicao(int no,Apont nao_loop[MAX_NO],int no_nivel[MAX_NO],
          int escopo[MAX_NO],float posicao_no[MAX_NO], int saida[MAX_NO],
          int num_NO)
{
 Apont *p;
 int filho, inicio, sai;
 float x;
 filho = tem_filho_diretamente_abaixo(no,nao_loop,no_nivel);
 if (filho){
  if (so_um_filho(no,nao_loop)==1){
   if (outros_pais_no_nivel_de_cima(no,filho,nao_loop,no_nivel))
        x=media_posicao_pais(filho,nao_loop,no_nivel,posicao_no);
   else
        x=posicao_no[no];
   sai=e_no_de_saida(filho,saida);
   if(sai)
    posicao_no[filho]=posicao_no[sai];
   else
    posicao_no[filho]=x;
   calcula_posicao(filho,nao_loop,no_nivel,escopo,posicao_no,saida,num_NO);
  }else{
   x=posicao_no[no]-escopo[no]*0.5;
   p=nao_loop[no].prox;
   while(p){
        if(no_nivel[p->info]==no_nivel[no]+1){
         posicao_no[p->info]=x+escopo[p->info]*0.5;
         x+=escopo[p->info];
        }
        p=p->prox;
   }
   p=nao_loop[no].prox;
   while(p){
        if(no_nivel[p->info]==no_nivel[no]+1)
         calcula_posicao(p->info,nao_loop,no_nivel,escopo,posicao_no,saida,num_NO);
        p=p->prox;
   }
  }
  if(pai_fan_out(filho,nao_loop,num_NO,&inicio)>2)
   posicao_no[saida[inicio]]=posicao_no[inicio];
 }else{
  p=nao_loop[no].prox;
  while(p){
   sai=e_no_de_saida(p->info,saida);
   if(sai){
        posicao_no[p->info]=posicao_no[sai];
        calcula_posicao(p->info,nao_loop,no_nivel,escopo,posicao_no,saida,num_NO);
   }
   p=p->prox;
  }
 }
 return;
} /*calcula_posicao*/
```

5.2.2 - Módulo Posiciona Arcos

A seguir é mostrado o código fonte da Função *Posiciona_Arcos*; observa-se que ela implementa exatamente o que foi proposto na Figura 4.19.

```
posiciona_arcos(Apont completa[MAX_NO],Apont nao_loop[MAX_NO],
        int niveis[MAX_NO],int num_NO,int *num_arcos, float posicao_no[MAX_NO],
        float posicao_arcos[MAX_ARCOS],St_arco lista_arcos[MAX_ARCOS])
{
 int ordem[MAX_ARCOS],
    arcos_delta_nivel[MAX_ARCOS],
    *n_unicas_delta_nivel,
    ja_pos[MAX_ARCOS],
    delta_nivel,
    *n_sem_opcao;
 St_op opcoes[MAX_ARCOS];
 register int i,j;

 monta_lista_de_arcos(completa,niveis,num_NO,lista_arcos,num_arcos);
 if(*num_arcos>0){
  monta_lista_delta_nivel(lista_arcos,niveis,*num_arcos,arcos_delta_nivel);
  for(i=0;i<MAX_ARCOS;i++){
    ordem[i]=i;
    ja_pos[i]=FALSE;
   }

  for(i=0;i<*num_arcos;i++){
    /* Inicia a Estrutura que armazena as opcoes */
    opcoes[i].esquerda=0.0;
    opcoes[i].direita=0.0;
    for(j=0;j<MAX_CENTRO;j++)
        opcoes[i].centro[j]=LIMITE;
   }

  ordena_arcos(*num_arcos,arcos_delta_nivel,ordem);
  posiciono_delta_nivel_1(lista_arcos,ordem,arcos_delta_nivel,*num_arcos,
                          posicao_no,posicao_arcos,ja_pos);
  n_sem_opcao=malloc(sizeof(int));
  *n_sem_opcao=0;
  n_unicas_delta_nivel=malloc(sizeof(int));
  *n_unicas_delta_nivel=0;
  delta_nivel=1;
  /* Verifica as opcoes iniciais para o posicionamento dos arcos */
  verifico_opcoes_de_todos_arcos(lista_arcos,ordem,num_NO,*num_arcos,niveis,
          arcos_delta_nivel,delta_nivel,n_unicas_delta_nivel,n_sem_opcao,
          posicao_arcos,posicao_no,nao_loop,ja_pos,opcoes);
  delta_nivel=arcos_delta_nivel[ordem[0]];
  i=0;
  do{
   do{
        do{
          posiciono_opcao_unica(lista_arcos,ordem,*num_arcos,delta_nivel,
                          arcos_delta_nivel,opcoes,ja_pos,posicao_arcos);
          verifico_opcoes_de_todos_arcos(lista_arcos,ordem,num_NO,*num_arcos,
```

```
                niveis,arcos_delta_nivel,delta_nivel,n_unicas_delta_nivel,
                n_sem_opcao,posicao_arcos,posicao_no,completa,ja_pos,opcoes);
        }while((*n_unicas_delta_nivel)!=0);
        posiciono_arco_opcao_multipla(lista_arcos,ordem,num_NO,*num_arcos,
                delta_nivel,arcos_delta_nivel,opcoes,ja_pos,posicao_arcos,
                niveis,n_unicas_delta_nivel,posicao_no,completa,n_sem_opcao);
        posiciono_arco_sem_opcao(lista_arcos,ordem,delta_nivel,posicao_no,
                arcos_delta_nivel,*num_arcos,opcoes,ja_pos,posicao_arcos,
                niveis);
    }while(!fim_arcos_delta_nivel(lista_arcos,ordem,delta_nivel,
                                  arcos_delta_nivel,*num_arcos,ja_pos));
    while((arcos_delta_nivel[ordem[i]]==delta_nivel)&&(i<(*num_arcos)))
        i++;
    if(i<(*num_arcos))
        delta_nivel=arcos_delta_nivel[ordem[i]];
   } while(i<(*num_arcos));

 }
 return;
} /*posiciona_arcos*/
```

5.2.3 - Módulo Des_Grafo

Das informações previstas para serem visualizadas na especificação da ferramenta ViewGraph, as que já têm a sua visualização implementada são : o código fonte relacionado a um nó e as variáveis que se encontram em um nó, além do próprio grafo do programa.

O Módulo Des_Grafo ativa duas funções do programa, a visualização do código fonte e variáveis relacionadas a um nó e o desenho do grafo de programa.

A Função *Desenha_Grafo* recebe a disposição gráfica do grafo e os parâmetros de desenho e gera o desenho do grafo no dispositivo de saída. As principais características da implementação dessa função são dadas a seguir.

Os nós do grafo de programa podem ser desenhados de duas maneiras diferentes: uma para o modo completo, onde o número do nó é escrito e uma circunferência circundando esse número é desenhada, e outra para o modo resumido, onde uma circunferência de raio mínimo é desenhada, sem o número do nó.

Os arcos do programa são desenhados de forma a não ser necessário restringir de nenhuma forma a quantidade de arcos saindo de um nó ou entrando em um nó e nem mesmo restringir as possíveis posições de entrada ou saída desses arcos. Para isso foi preciso calcular-se para cada arco (n,m) os pontos de intersecção entre a reta C_nC_m e as circunferências N e M, onde C_n e C_m são os centros das circunferências N e M que representam graficamente os nós *n* e *m*, respectivamente. Com isso chega-se a quatro pontos de intersecção P_{n1}, P_{n2}, P_{m1}, P_{m2}, com um estudo das coordenadas desses pontos em comparação com a coordenada dos pontos C_n e C_m pode-se determinar quais pontos estão entre C_n e Cm sobre a reta C_nC_m.

Supondo que os pontos que estão entre C_n e C_m sejam P_{n2} e P_{m1}, o segmento de reta que liga os pontos P_{n2} e P_{m1} representará o arco (n,m) (Figura 5.2).

Dois dos parâmetros aceitos para o desenho do grafo são : o deslocamento horizontal e o deslocamento vertical; alterando-se o valor desses parâmetros pode-se deslocar o grafo no

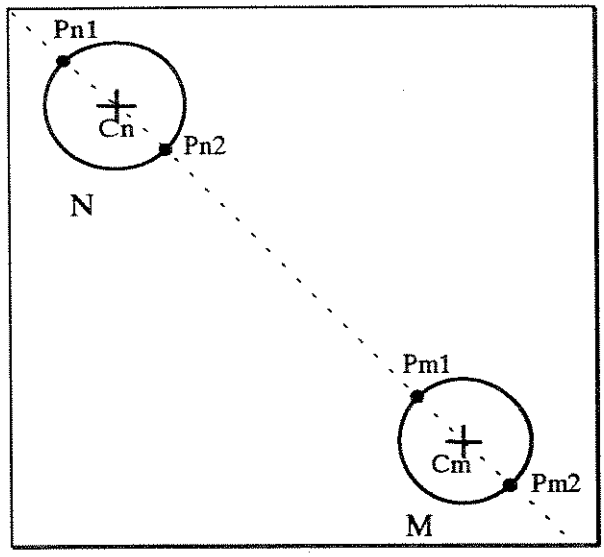


Figura 5.2. Desenho dos Arcos

dispositivo de saída (no caso desse dispositivo ser um monitor de vídeo). Essa abordagem também permite que o grafo seja deslocado para fora dos limites de enquadramento da tela, podendo depois retornar sem que nenhuma informação seja perdida, o que possibilita a visualização de grafos grandes que ocupem um espaço maior que a área de desenho.

Se a cada mudança dos parâmetros de desenho tivessem que ser recalculados todos os pontos de intersecção de retas e circunferências do grafo de programa, o deslocamento do grafo ficaria muito lento. A solução encontrada para otimizar o cálculo das intersecções foi criar uma tabela com as coordenadas dos pontos calculadas para a posição inicial (deslocamento zero); quando for alterado algum parâmetro de deslocamento, a diferença do novo valor para o valor inicial é aplicada sobre a tabela e as novas coordenadas são obtidas.

5.3 - Avaliação Empírica

Uma avaliação empírica do módulo de posicionamento dos nós e do módulo de roteamento dos arcos foi realizada e é apresentada na Tabela 5.1.

A maioria dos arquivos utilizados nessa avaliação foram gerados a partir de programas reais extraídos da literatura; alguns arquivos como xcase*.gfc e tidier.gfc foram editados manualmente para testar características específicas do programa. Alguns dos arquivos listados na Tabela 5.1 são mostrados no Apêndice A, outros foram omitidos por não apresentarem nenhuma característica especial de posicionamento.

A Tabela 5.1 mostra, para cada arquivo, o número de nós que ele contém e o tempo gasto para posicioná-los. Para o caso dos arcos a Tabela mostra o número de arcos longos do programa, ou seja, aqueles relevantes para tempo de processamento do roteamento, a soma dos tamanhos dos arcos e o tempo gasto para rotear esses arcos. Por fim, a Tabela mostra a soma dos dois tempos para cada arquivo. Na última linha da Tabela são apresentados os totais gerais.

A tomada de tempos foi feita num microcomputador modelo PC-XT de 8 MHZ; essa configuração foi escolhida para aumentar a resolução dos tempos; mesmo assim, os tempos registrados ficaram na ordem de meio segundo sendo que em alguns casos o tempo foi zero, pois, devido à resolução conseguida não foi possível identificar nenhuma alteração no "tic" do relógio.

Algumas características podem ser observadas nessa Tabela; por exemplo, o tempo de posicionamento dos nós tem uma ligação mais direta com o número de nós do que com o número e o tamanho dos arcos; observa-se que quanto maior o número de nós o tempo tende a ser maior o que não ocorre no caso dos arcos.

Apesar disso, algumas distorções ocorrem, como no caso dos arquivos translit.gfc e makepat.gfc com 30 nós que são posicionados em cerca de 0.35 segundos e o arquivo tidier.gfc com 31 nós posicionados em apenas 0.16 segundos, o que indica uma dependência do tempo de processamento com a própria estrutura do grafo (veja o desenho do grafo translit.gfc e tidier.gfc no Apêndice A).

Tabela 5.1 - Avaliação Empírica do Posicionamento dos Nós e Roteamento dos Arcos

Arquivos	Nós	Tempo Nós (s)	Arcos Longos	Tamanho Arcos	Tempo Arcos (s)	Tempo Total (s)
AMATCH.GFC	22	0.2747	5	18	0.4396	0.7143
APPEND.GFC	16	0.1648	5	20	0.3846	0.5494
ARCHIVE.GFC	18	0.2198	5	34	0.05495	0.27475
CASE1.GFC	12	0.1648	4	12	0.05495	0.21975
CHANGE.GFC	11	0.1099	4	12	0.1099	0.2198
CKGLOB.GFC	19	0.1648	6	31	0.1648	0.3296
COMPARE.GFC	14	0.1099	6	25	0.2747	0.3846
COMPRESS.GFC	16	0.1648	3	13	0.2747	0.4395
DODASH.GFC	19	0.1648	6	36	0.5495	0.7143
EDIT.GFC	22	0.1648	9	37	1.648	1.8128
ENTAB.GFC	15	0.1648	5	17	0.3846	0.5494
EXEMPLO.GFC	9	0.1099	2	3	0.05495	0.16485
EXPAND.GFC	18	0.1648	6	25	0.5495	0.7143
GETCMD.GFC	43	0.2747	14	108	0.7143	0.989
GETDEF.GFC	26	0.2198	8	35	0.7143	0.9341
GETFN.GFC	13	0.1648	5	13	0.2198	0.3846
GETFNS.GFC	17	0.1648	6	22	0.2198	0.3846
GETLIST.GFC	16	0.1648	5	13	0.3846	0.5494
GETNUM.GFC	19	0.2198	5	26	0.1099	0.3297
GTEXT.GFC	9	0.1099	3	9	0.1099	0.2198
INIGRF.GFC	19	0.1648	7	15	0.4396	0.6044
MAKEPAT.GFC	30	0.3297	9	60	1.538	1.8677
OMATCH.GFC	29	0.6593	9	24	0.4396	1.0989
OPTPAT.GFC	15	0.1648	3	10	0.1099	0.2747
PRICE1.GFC	9	0.1648	1	3	0	0.1648
PRICE3.GFC	11	0.1648	4	12	0.05495	0.21975
PRICE4.GFC	14	0.1648	5	15	0.1099	0.2747
PUSH.GFC	4	0.1099	0	0	0	0.1099
RQUICK.GFC	14	0.1099	5	18	0.3297	0.4396
SPREAD.GFC	14	0.1099	4	18	0.3846	0.4945
SUBST.GFC	27	0.3846	5	27	1.154	1.5386
TIDIER.GFC	31	0.1648	0	0	0	0.1648
TOPICOS.GFC	25	0.1648	8	35	0.3846	0.5494
TRANSLIT.GFC	30	0.3846	11	102	0.9341	1.3187
UNROTATE.GFC	20	0.1648	7	21	0.7692	0.934
VELHO.GFC	26	0.2198	6	34	0.8242	1.044
XCASE1.GFC	14	0.1648	6	20	0.8791	1.0439
XCASE2.GFC	12	0.1648	4	12	0.05495	0.21975
XCASE3.GFC	33	0.7143	9	31	0.2198	0.9341
XCASE4.GFC	16	0.2198	3	7	0.05495	0.27475
XCASE5.GFC	17	0.2747	4	12	0.2198	0.4945
Totais	764	8.6257	222	985	16.3183	24.944

No caso dos arcos a relação do tempo com características estruturais do grafo é mais marcante, o arquivo translit.gfc tem 11 arcos longos com tamanho total de 102 (níveis) e leva 0.93 segundos para ter esses arcos roteados, já o arquivo getcmd.gfc tem 14 arcos com tamanho 108 e leva apenas 0.71 segundos.

De qualquer forma essa avaliação mostrou que os algoritmos funcionam para a maior parte das situações encontradas em programas reais, com tempo aceitável mesmo na plataforma utilizada na avaliação; se for considerado que os módulos de geração do desenho são executados apenas uma vez para cada arquivo, o tempo de execução não é impedimento para a utilização desses algoritmos.

CAPÍTULO 6

CONCLUSÃO

Na primeira seção deste capítulo são apresentadas as conclusões resultantes das atividades desenvolvidas nesta dissertação e na Seção 6.2 são propostos os trabalhos futuros para evolução da ferramenta implementada.

6.1 - Visualização de Informações de Teste e Grafos de Programa

Foi especificada uma ferramenta, denominada ViewGraph, cujo propósito é auxiliar a atividade de teste e depuração de programas através da visualização de informações de teste fornecidas pela POKE-TOOL [CHA91].

Foram definidos alguns de seus requisitos básicos; tais como : possibilitar a visualização de informações provenientes da análise estática do módulo em teste, mostrar informações derivadas da análise dinâmica feita pela POKE-TOOL e ter uma interface amigável e interativa de forma a tornar a atividade de teste o mais atraente e produtiva quanto possível.

Além do estudo de aspectos estéticos, visando gerar diagramas mais inteligíveis e, conseqüentemente, mais úteis, as tendências da área de geração de diagramas caminham em duas direções : desenvolvimento de aplicativos e aprimoramento dos algoritmos de posicionamento.

Parte essencial da ferramenta ViewGraph é a visualização dos grafos de programa; assim, foram desenvolvidos e implementados algoritmos para suprir essa necessidade. No processo de desenvolvimento dos algoritmos foram investigadas as principais publicações relacionadas ao assunto, resultando num apanhado geral dessa área, que inclui : diferentes aplicações, principais soluções, conceitos básicos e terminologias.

Com isso as duas vertentes de estudos na área de visualização foram abordadas, ou seja, tanto o aprimoramento dos algoritmos de posicionamento, com a apresentação dos algoritmos de visualização de grafos de programas, quanto o desenvolvimento de aplicativos, com a especificação da ferramenta ViewGraph.

Os algoritmos de visualização de grafos de programa realizam duas tarefas independentes : o posicionamento dos nós e o roteamento dos arcos. O cálculo das posições dos nós leva em conta o valor do nível de cada nó (modificado a partir do que é definido para árvores) para definir as coordenadas Y. Na determinação da coordenada X de cada nó surge um problema : como saber a priori qual será o espaço ocupado por cada subgrafo descendente de um dado nó? Para solucionar esse problema definiu-se uma variável denominada *Escopo*; informalmente, ela quarda o espaço que deve ser reservado abaixo de cada nó, para que seus descendentes possam ser posicionados sem haver sobreposição.

As diferenças do cálculo do nível definido para árvores e o definido aqui para grafos de programa resumem-se, basicamente, a duas características :

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 do seu pai, sem especificar qual é o pai em questão. Para solucionar esse problema definiu-se que o nível de um nó, pertencente a um grafo de programa, é uma unidade maior que o nível de seu pai de maior nível.

2. O segundo problema ocorre no caso da estrutura case; ocasionalmente nessa estrutura pode ocorrer do nó de saída não ter nenhum antecessor no nível diretamente acima. Para resolver esse problema foi necessário definir uma maneira de determinar os nós pertencentes a

uma estrutura case (a partir somente de informações estruturais) e atribuir ao nó de saída um nível uma unidade maior que o nó de maior nível interno à estrutura case.

A estratégia adotada para o roteamento dos arcos levou em conta algumas de suas características ligadas principalmente ao seu tipo. Arcos curtos têm roteamento trivial e para os outros tipos de arcos foi definido um algoritmo que determina as opções possíveis para o posicionamento e escolhe a melhor.

As principais características dos algoritmos apresentados são :

- 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 mostrada nessa dissertação esse problema é eliminado através do uso da variável denominada **escopo**; com essa variável, pode-se determinar diretamente as posições dos nós sem haver repetição de cálculos.

- O posicionamento relativo gerado por essa abordagem possibilita que um mesmo grafo de programa possa ser visualizado com diversas aparências diferentes no dispositivo de saída, bastando para isso alterar-se parâmetros de escala, espaçamento e deslocamento, diretamente no módulo de desenho.

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

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

As principais contribuições dessa abordagem aparecem na geração de um posicionamento independente do dispositivo de saída e no uso da variável escopo que otimiza a definição do posicionamento dos nós; o roteamento dos arcos segue um conceito definido na literatura como *envoltória*, essa envoltória é ampliada ou anulada à medida que o grafo for sendo percorrido.

Foi desenvolvida uma implementação dos algoritmos definidos e a essa implementação foram adicionadas a visualização do código fonte e das variáveis de um nó; essa implementação constitui o protótipo atual da ViewGraph.

Uma avaliação empírica da implementação dos algoritmos de visualização de grafos de programa foi conduzida, mostrando que o algoritmo funciona satisfatoriamente para diversas situações encontradas em programas reais e que a sua implementação executa num tempo aceitável mesmo em um PC-XT de 8MHZ.

6.2 - Trabalhos Futuros

O que separa o protótipo atual da ViewGraph da sua versão especificada é a visualização das outras informações de teste, que deverá ser incorporada em futuro próximo.

Uma evolução natural da POKE-TOOL envolve o tratamento do teste de integração dos módulos de um programa; com essa evolução novas informações de teste precisarão ser visualizadas. A representação do grafo do programa deverá considerar a possibilidade de um nó não mais representar somente um bloco de comandos, mas também poder representar um

outro módulo do programa. A ferramenta deverá portanto fornecer recursos de navegação sobre esses vários módulos.

Outra variação com relação aos grafos de programa está na possibilidade de representação de algoritmos paralelos; para que esse tipo de grafo possa ser representado graficamente, novas informações no arquivo .gfc devem ser adicionadas e novos padrões de representação devem ser definidos.

Os algoritmos para visualização de grafos de programa podem ser adaptados a vários tipos de diagramas; entre eles : máquinas de estado, árvores, hierarquia de funções, hierarquia de objetos, grafos planares, fluxo de dados, modelos entidade-relacionamento, etc.

Por fim, as duas ferramentas, POKE-TOOL e ViewGraph, devem ser completamente integradas; adicionadas do tratamento de caminhos não executáveis e do teste de integração, ter-se-á um ambiente totalmente automatizado de suporte ao teste e à depuração de programas.

REFERÊNCIAS BIBLIOGRÁFICAS

[BAT86] Batini, C., Nardelli, E., and Tamassia, R., A Layout Algorithm for Data Flow Diagrams. *IEEE Trans. Soft. Eng.*, Vol. SE-12, No.4, pp. 538-546, April 1986.

[BAT88] Battista, D. G., Tamassia, R., Algorithms for Plane Representations of Ayclic Digraphs. *Theoritical Computer Science*, Vol.61, No.2-3, pp.175-198, Nov. 1988.

[BOR81] Borning, A., The Programming Language Aspects of ThingLab. A Constraint-Oriented Simulated Laboratory, *ACM Trans. Program. Lang. Syst.*, Vol.3, No.4, Oct. 1981.

[BRO84] Brown, M. H. and Sedgewick, R., A System for Algorithm Animation, *ACM Comput. Graphics*, Vol.18, No.3, pp. 177-186, July 1984.

[CAR80] Carpano, M.J., Automatic Display of Hierarchized Graphs for Computer-Aided Decision Analysis. *IEEE Trans. on Systems, Man, and Cybern.*, Vol. SMC-10, No.11, Nov. 1980.

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

[CHA91a] Chaim, M.L., Maldonado, J.C. e Jino,M., Manual de Configuração da POKE-TOOL. *Relatório Técnico*, DCA / RT / 088 / 91 - DCA / FEE / UNICAMP - Fev. 1991.

[CHI84] Chiba, N., Yamagushi, T. and Nishizeki, T., Linear Algorithms for Convex Drawings of Planar Graphs, in *Progress in Graph Theory*, Bondy, J.A. and U. S. R. Murty, ed. Orlando, FL : Academic Press, 1984, pp. 153-173.

[CHI85A] Chiba, N., Onoguchi, K., Nishizeki, T., Drawing Plane Graphs Nicely. *Acta Inform.*, Vol.22, pp.187-201, 1985.

[CHU87] Chusho, T., Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing, *IEEE Trans. Soft. Eng.*, Vol. SE-13, No. 5, pp. 509-517, May 1987.

[COW88] Coward, P.D., A Rewiew of Software Testing, *Information and Software Technology*, Vol. 30, No.3, Apr. 1988, pp. 189-198.

[DIN90] Ding, C. and Mateti, P., A Framework for the Automated Drawing of Data Structure Diagrams. *IEEE Trans. Soft. Eng.*, Vol. 16, No. 5, pp. 543-557, May 1990.

[EAD86] Eades, P. A Heuristic for Graph Drawing, in *Congressus Numeratium*, May 1984, pp.149-160.

[FAR48] Fary, I., On Straight Line Representation of Planar Graphs. *Acta Univ. Szeged Sect. Sci. Math.*, Vol. 11, pp. 229-233, 1948.

[FIN84] Finzer, W. and Gould, L., Programming by Rehearsal, *Byte*, Vol.9, No.6, pp. 187-210, June 1984.

[FRA87] Frankl, F.G., The Use of Data Flow Information for the Selection and Evaluation of Software Test Data. *Ph.D dissertation*, New York Univ., New York, Oct. 1987.

[FRA88] Frankl, F.G. and Weyuker, E.J., An Aplicable Family of Data Flow Testing Criteria. *IEEE Trans. on Software Eng.*, Vol. 14, No. 10, Oct. 1988, pp. 1483-1498.

[FRU91] Fruchterman, T. M. J., Reingold, E. M., Graph Drawing by Force-directed Placement. *Software Pratice and Experience*, Vol.21, No.11, pp.1129-1164, Nov. 1991.

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

[GAN93] Gansner, E. R., Koutsofios, E., North, S. C., Vo, K. P., A Technique for Drawing Directed Graphs. *IEEE Trans. Soft. Eng.*, Vol.19, No.3, March 1993.

[GLI84] Glinert, E. P. and Tanimoto, S. L., Pict : An Interactive Graphical Programming Environment, *Computer*, Vol.17, No.11, pp. 7-25, Nov. 1984.

[HAL91] Hall, P.A.V., Relationship between Specifications and Testing, *Information and Software Technology*, Vol. 33, No.1, Jan/Feb, 1991, pp. 47-52.

[HOR86] Horowitz, e., Sahni, S. Fundamentos de Estruturas de Dados. Editora Campus LTDA. 2a edição, Rio de Janeiro, 1986.

[HOW87] Howden W.E., *Functional Program Testing and Analysis*. McGraw-Hill, USA, 1987.

[ISO87] Isoda, S., Shimomura, T., and Ono, Y., VIPS : A Visual Debugger, *IEEE Software*, pp. 8-19, May 1987.

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

[KNU63] Knuth, D. E., Computer-Draw Flowcharts. *Comm. ACM*, Vol.6, pp.555-563, Sept. 1963.

[LIP85] Lipton, R. J., North, S. C. and Sandberg, J. S., A Method for Drawing Graphs, in *Proc. 1st ACM Symp. Computational Geometry*, Baltimore, MD, 1985, pp.153-160.

[LON85] London, R. L. and Duisberg, R. A., Animating Programs Using Smalltalk, *Computer*, Vol.18, No.8, pp. 61-71, Aug. 1985.

[MAL88a] Maldonado, J. C., Chaim, M. L., Jino, M., Seleção de Casos de Testes Baseados em Fluxo de Dados através dos Critérios Potenciais Usos, *in Proc. II Simp. Bras. Eng. de Software*, Canela, R.S., pp. 24-35, Out. 1988.

[MAL88b] Maldonado, J. C., Chaim, M. L., Jino, M., Resultados do estudo de uma família de critérios de teste de programas baseado em fluxo de dados, *Relatório Técnico* DCA/RT/001/88 - DCA/FEE/UNICAMP - Dez. 1988.

[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.

[MAT86] Mateti, P. and Radack, G. M., Automated Drawing of Data Structure Diagrams, in *Proc. 4th Annu. Nat. Conf. Ada Technology*, pp. 165-172, Mar. 1986.

[MCC76] McCabe, T., A Software Complexity Measure, *IEEE Trans. Software Eng.*, Vol. 2, pp.308-320, Dez. 1976.

[MES91] Messinger, Eli B. et al., A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs. *IEEE Trans. on Systems, Man, and Cybern.*, Vol.21, No.1, pp.1-12, Jan-Feb 1991.

[MEY79] Myers, G.J., *The Art of Software Testing.*, Wiley, New York, 1979.

[MOE90] Moen, S., Drawing Dynamic Trees. *IEEE Software*, July 1990.

[NTA84] Ntafos, S.C., On Required Element Testing. *IEEE Trans. Software Eng.*, Vol. SE-10, Nov. 1984, pp. 795-803.

[NTA89] Ntafos, S.C., A Comparison of Some Structural Testing Strategies. *IEEE Trans. Software Eng.*, Vol. 14, No.6, Jun. 1988, pp. 868-873.

[PON83] Pong, M. C. and Ng, N., PIGS - A System for Programming with Interactive Graphical Support. *Software-Practice and Experience*, Vol.13, No.9, pp. 847-855, Sept. 1983.

[PRE87] Pressman, R.B., Software Engineering: A Practioner's Approach. segunda edição, MacGraw-Hill, New York, 1987.

[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.

[PRO91] Protsko, L.B., et al, Towards the Automatic Generation of Software Diagrams. *IEEE Trans. Soft. Eng.*, Vol.17, No.1, pp. 10-21, Jan. 1991.

[RAP82] Rapps, S. and Weyuker, E. J., Data Flow Analysis Techniques for Test Data Selection. *in Proc. Int. Conf. Software Eng.*, Tokio, Japão, pp.272-278, Sept. 1982.

[RAP85] Rapps, S. and Weyuker, E.J., Selecting Software Test Data Using Data Flow Information. *IEEE Trans. Software Eng.*, Vol. SE-11, Apr. 1985, pp. 367-375.

[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.

[SAR92] Sarkar, M. and Brown, M.H., Graphical Fisheye Views of Graphs. *ACM*, pp. 83-91, May 3 - 7, 1992.

[STO75] Stockenberg, J. E. and Dam, A. V., STRUCT Programming Analysis System, *IEEE Trans. Soft. Eng.*, VOL. SE-1, No.4, Dec. 1975.

[SUG81] Sugiyama, K., Tagawa, S., Toda, M., Methods for Visual Understanding of Hierarchical System Structures. *IEEE Trans. on Systems, Man, and Cybern.*, Vol. SMC-11, No.2, Feb. 1981.

[TAM87] Tamassia, R., On Embedding a Graph in the Grid with the Minimum Number of Bends. *SIAM J. Comput.*, Vol.16, No.3, pp.421-444, 1987.

[TAM88] Tamassia, R., Battista, G. D., Batiti, C., Automatic Graph Drawing and Readability of Diagrams. *IEEE Trans. on Systems, Man, and Cybern.*, Vol.18, No.1, Jan/Feb 1988.

[TUT63] Tutte, W.J., How to Draw a Graph. *Proc. London Math. Soc.*, Series 3, 13, 743-768, 1963.

[URA88] Ural, U. and Yang, B., A Structural Test Selection Criterion. *Information Processing Letters*, Vol.28, Jul. 1988 pp. 157-163.

[VAU80] Vaucher, J., Pretty Printing of Trees. *Software Pratice and Experience*, Vol.10, pp.553-561, 1980.

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

[WEY84] Weyuker E.J., The Complexity of Data Flow Criteria for Test Data Selection. *Information Processing Letters*, Vol.19, No.2, Aug. 1984, pp. 103-109.

[WEY86] Weyuker, E.J., Axiomatizing Software Test Data Adequacy, *IEEE Trans. on Soft. Eng.*, Vol. SE-12, No.12, Dec. 1988, pp. 1128-1138.

APÊNDICE A

Nome do Programa : AMATCH.C

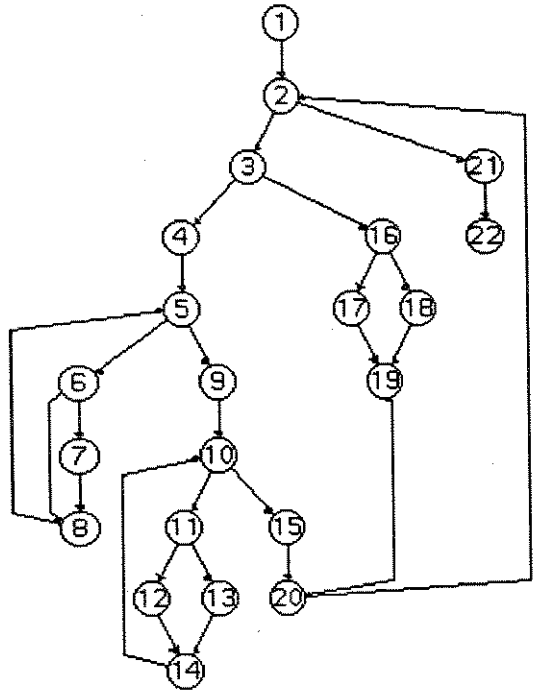
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "editconst.h"
extern FILE * path;
int amatch(lin,offset,pat,j)
char *lin, *pat;
int offset, j;
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int i,k,done;
            ponta_de_prova(1);
/* 1 */       done = 0;
/* 2 */       while((!done) && (pat[j] != ENDSTR))
/* 3 */       {
              ponta_de_prova(2);
              ponta_de_prova(3);
/* 3 */         if(pat[j] == CLOSURE)
/* 4 */         {
              ponta_de_prova(4);
/* 4 */           j = j + patsize(pat,j);
/* 4 */           i = offset;
/* 5 */           while((!done) && (lin[i] != ENDSTR))
/* 6 */           {
              ponta_de_prova(5);
              ponta_de_prova(6);
/* 6 */             if(!omatch(lin,&i,pat,j))
/* 7 */             {
              ponta_de_prova(7);
/* 7 */               done = 1;
/* 7 */             }
              ponta_de_prova(8);
/* 8 */           }
              ponta_de_prova(5);
              ponta_de_prova(9);
/* 9 */           done = 0;
/* 10 */          while((!done) && (i >= offset))
/* 11 */          {
              ponta_de_prova(10);
              ponta_de_prova(11);
/* 11 */            k = amatch(lin,i,pat,j+patsize(pat,j));
/* 11 */            if(k >= 0)
/* 12 */            {
              ponta_de_prova(12);
/* 12 */              done = 1;
/* 12 */            }
/* 13 */            else
/* 13 */            {
              ponta_de_prova(13);
/* 13 */              i--;
/* 13 */            }
              ponta_de_prova(14);
/* 14 */          }
              ponta_de_prova(10);
              ponta_de_prova(15);
/* 15 */          offset = k;
/* 15 */          done = 1;
/* 15 */        }
/* 16 */        else
/* 16 */        {
              ponta_de_prova(16);
/* 16 */          if(!omatch(lin,&offset,pat,j))
/* 17 */          {
              ponta_de_prova(17);
/* 17 */            offset = -1;
/* 17 */            done = 1;
/* 17 */          }
/* 18 */          else
/* 18 */          {
              ponta_de_prova(18);
/* 18 */            j = j + patsize(pat,j);
/* 18 */          }
              ponta_de_prova(19);
/* 19 */        }
              ponta_de_prova(20);
/* 20 */      }
            ponta_de_prova(2);
            ponta_de_prova(21);
            ponta_de_prova(22);
            fclose(path);
/* 21 */      return(offset);
/* 22 */    }
ÿÿÿ
```

Grafo de Programa :

```
22

1
  2 0
2
  3 21 0
3
  4 16 0
4
  5 0
5
  6 9 0
6
  7 8 0
7
  8 0
8
  5 0
9
  10 0
10
  11 15 0
11
  12 13 0
12
  14 0
13
  14 0
14
  10 0
15
  20 0
16
  17 18 0
17
  19 0
18
  19 0
19
  20 0
20
  2 0
21
  22 0
22
  0
```

Desenho do Grafo de Programa :



Amatch.c

Observações Sobre o Desenho :

Nesse desenho podem ser observadas algumas características interessantes, por exemplo com relação à numeração dos nós, observa-se que o nó 15 é o próximo bloco de comando a ser executado após o fechamento, no nó 14, da estrutura iniciada no nó 10 que corresponde a um WHILE.

Nome do Programa : Append.c

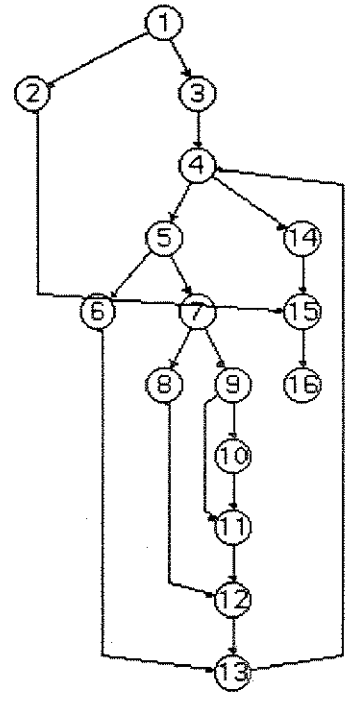
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "editconst.h"
extern int curln;
stcode append (line,glob)
int line;
int glob;
/* 1 */     {
              FILE * path = fopen("path.tes","w");
              static int printed_nodes = 0;
/* 1 */       char inline[MAXSTR];
/* 1 */       stcode stat;
/* 1 */       int done;
              ponta_de_prova(1);
/* 1 */       if(glob)
/* 2 */       {
              ponta_de_prova(2);
/* 2 */         stat = ERR;
/* 2 */       }
/* 3 */       else
/* 3 */       {
              ponta_de_prova(3);
/* 3 */         curln = line;
/* 3 */         stat = OK;
/* 3 */         done = 0;
/* 4 */         while((!done) && (stat == 2))
/* 5 */         {
              ponta_de_prova(4);
              ponta_de_prova(5);
/* 5 */           if(!getline(inline,stdin,MAXSTR))
/* 6 */           {
              ponta_de_prova(6);
/* 6 */             stat = ENDDATA;
/* 6 */           }
/* 7 */           else
/* 7 */           {
              ponta_de_prova(7);
/* 7 */             if((inline[0] == PERIOD) && (inline[1]
== NEWLINE))
/* 8 */             {
              ponta_de_prova(8);
/* 8 */               done = 1;
/* 8 */             }
/* 9 */             else
/* 9 */             {
              ponta_de_prova(9);
/* 9 */               if(puttxt(inline) == ERR)
/* 10 */              {
              ponta_de_prova(10);
/* 10 */                stat = ERR;
/* 10 */              }
              ponta_de_prova(11);
/* 11 */            }
              ponta_de_prova(12);
/* 12 */          }
              ponta_de_prova(13);
/* 13 */        }
              ponta_de_prova(4);
              ponta_de_prova(14);
/* 14 */      }
              ponta_de_prova(15);
              ponta_de_prova(16);
              fclose(path);
/* 15 */      return (stat);
/* 16 */    }
ÿ
```

Grafo de Programa : Append.gfc

```
16
 1
  2 3 0
 2
  15 0
 3
  4 0
 4
  5 14 0
 5
  6 7 0
 6
  13 0
 7
  8 9 0
 8
  12 0
 9
  10 11 0
10
  11 0
11
  12 0
12
  13 0
13
  4 0
14
  15 0
15
  16 0
16
  0
ÿ
```

Desenho de Grafo de Programa :



Nesse desenho observa-se um problema gerado pela restrição feita sobre o posicionamento dos nós. O nó 1 corresponde a uma estrutura IF-THEN-ELSE, sendo o nó 2 o THEN e o nó 3 o ELSE, portanto, segundo o algoritmo de posicionamento dos nós, o nó 2 deve ser posicionado à esquerda e o nó 3 à direita.

Se a abordagem fosse outra o cruzamento do arco (2,15) poderia ser evitado.

Nome do Programa : Compare.c

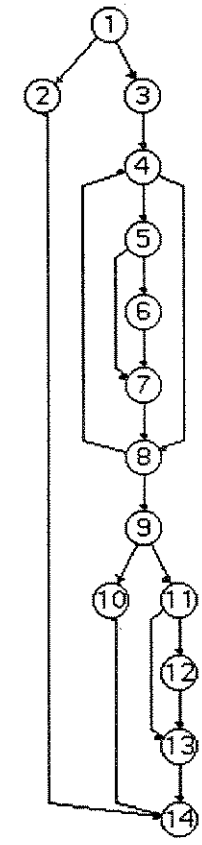
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "cte.h"
void compare()
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int lineno, f1 ,f2,
    infile1, infile2;
/* 1 */       char arg1[MAXSTR], arg2[MAXSTR],
    line1[MAXSTR], line2[MAXSTR];
            ponta_de_prova(1);
/* 1 */       if((!getarg(1,arg1)) || (!getarg(2,arg2)))
/* 2 */       {
            ponta_de_prova(2);
/* 2 */         message ("usage: compare file1 file2");
            ponta_de_prova(14);
            fclose(path);
/* 2 */         return;
/* 2 */       }
            ponta_de_prova(3);
/* 3 */       infile1 = mustopen(arg1,IOREAD);
/* 3 */       infile2 = mustopen(arg2,IOREAD);
/* 3 */       lineno = 0;
/* 4 */       do
/* 4 */       {
            ponta_de_prova(4);
/* 4 */         lineno++;
/* 4 */         f1 = getline(line1,infile1,MAXSTR);
/* 4 */         f2 = getline(line2,infile2,MAXSTR);
/* 4 */         if(f1 && f2)
/* 5 */         {
            ponta_de_prova(5);
/* 5 */           if(!equal(line1,line2))
/* 6 */           {
            ponta_de_prova(6);
/* 6 */             diffmsg(lineno,line1,line2);
/* 6 */           }
            ponta_de_prova(7);
/* 7 */         }
            ponta_de_prova(8);
/* 8 */       }
/* 8 */       while(f1 && f2);;
            ponta_de_prova(9);
/* 9 */       if(f2 && !f1)
/* 10 */      {
            ponta_de_prova(10);
/* 10 */        message("compare: end of file on file1");
/* 10 */      }
/* 11 */      else
/* 11 */      {
            ponta_de_prova(11);
/* 11 */        if(f1 && !f2)
/* 12 */        {
            ponta_de_prova(12);
/* 12 */          message("compare: end of file on file2");
/* 12 */        }
            ponta_de_prova(13);
/* 13 */      }
            ponta_de_prova(14);
            fclose(path);
/* 14 */    }
ÿ
```

Grafo de Programa : Compare.gfc

```
1
 2 3 0
2
 14 0
3
 4 0
4
 5 8 0
5
 6 7 0
6
 7 0
7
 8 0
8
 4 9 0
9
 10 11 0
10
 14 0
11
 12 13 0
12
 13 0
13
 14 0
14
 0
```

Desenho do Grafo de Programa :



Observações Sobre o Desenho :

Nesse exemplo pode-se notar o efeito da ordenação no roteamento dos arcos. Observa-se que o arco (5,7) foi posicionado antes dos arcos (4,8) e (8,4) e isso evitou qualquer cruzamento.

Nome do Programa : Compress.c

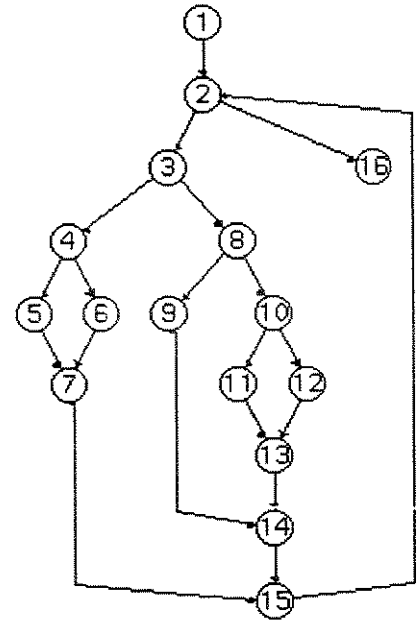
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "cte.h"
void compress()
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int n,lastc,c;
            ponta_de_prova(1);
/* 1 */       n = 1;
/* 1 */       lastc = getchar();
/* 2 */       while(lastc != ENDFILE)
/* 3 */       {
              ponta_de_prova(2);
              ponta_de_prova(3);
/* 3 */         if((c=getchar()) == ENDFILE)
/* 4 */         {
              ponta_de_prova(4);
/* 4 */           if((n>1)||(lastc==WARNING))
/* 5 */           {
              ponta_de_prova(5);
/* 5 */             putrep(n,lastc);
/* 5 */           }
/* 6 */           else
/* 6 */           {
              ponta_de_prova(6);
/* 6 */             putchar(lastc);
/* 6 */           }
              ponta_de_prova(7);
/* 7 */         }
/* 8 */         else
/* 8 */         {
              ponta_de_prova(8);
/* 8 */           if(c==lastc)
/* 9 */           {
              ponta_de_prova(9);
/* 9 */             n++;
/* 9 */           }
/* 10 */          else
/* 10 */          {
              ponta_de_prova(10);
/* 10 */            if((n>1)||(lastc==WARNING))
/* 11 */            {
              ponta_de_prova(11);
/* 11 */              putrep(n,lastc);
/* 11 */              n = 1;
/* 11 */            }
/* 12 */            else
/* 12 */            {
              ponta_de_prova(12);
/* 12 */              putchar(lastc);
/* 12 */            }
              ponta_de_prova(13);
/* 13 */          }
              ponta_de_prova(14);
/* 14 */        }
              ponta_de_prova(15);
/* 15 */        lastc = c;
            }
            ponta_de_prova(2);
            ponta_de_prova(16);
            fclose(path);
/* 16 */    }
ÿ
```

Grafo de Programa : Compress.gfc

```
16

1
 2 0
2
 3 16 0
3
 4 8 0
4
 5 6 0
5
 7 0
6
 7 0
7
 15 0
8
 9 10 0
9
 14 0
10
 11 12 0
11
 13 0
12
 13 0
13
 14 0
14
 15 0
15
 2 0
16
 0
ÿÿÿÿ
```

Desenho do Grafo de Programa :



Observações Sobre o Desenho :

Observa-se que os nós 6 e 9 não se tocam devido ao espaço reservado para o posicionamento dos filhos do nó 3 ter sido suficiente.

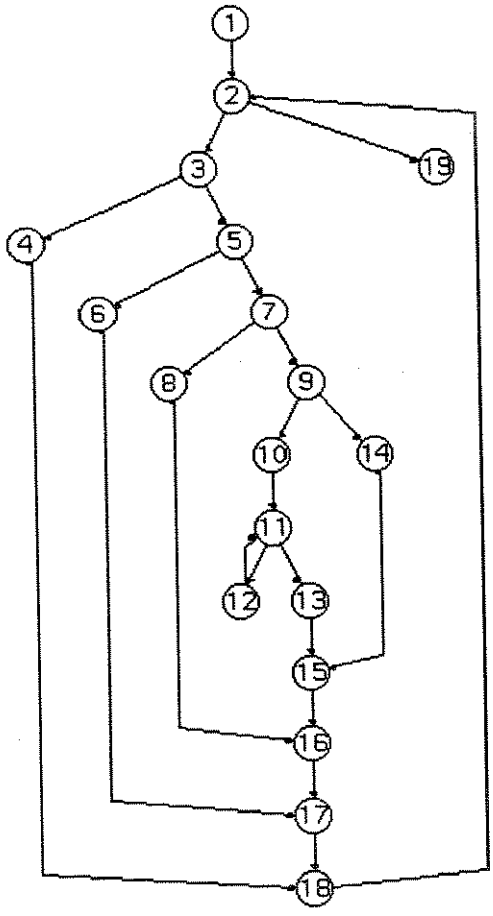
Nome do Programa : Dodash.c
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "cte.h"
void dodash(delim, src, i, dest, j, maxset)
int delim, *i, *j, maxset;
char *src, *dest;
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int k;
            ponta_de_prova(1);
/* 2 */           while((src[*i] != delim) && (src[*i] !=
ENDSTR))
/* 3 */       {
              ponta_de_prova(2);
              ponta_de_prova(3);
/* 3 */         if(src[*i] == ESCAPE)
/* 4 */         {
              ponta_de_prova(4);
/* 4 */           addstr(esc(src,i),dest,j,maxset);
/* 4 */         }
/* 5 */         else
/* 5 */         {
              ponta_de_prova(5);
/* 5 */           if(src[*i] != DASH)
/* 6 */           {
              ponta_de_prova(6);
/* 6 */             addstr(src[*i],dest,j,maxset);
/* 6 */           }
/* 7 */           else
/* 7 */           {
              ponta_de_prova(7);
/* 7 */             if((*j<=0) || (src[*i+1] == ENDSTR))
/* 8 */             {
              ponta_de_prova(8);
/* 8 */               addstr(src[*i],dest,j,maxset);
/* 8 */             }
/* 9 */             else
/* 9 */             {
              ponta_de_prova(9);
/* 9 */                 if((isalnum(src[*i-1])) && (isalnum
(src[*i+1]))
                && (src[*i-1] <= src[*i+1]))
/* 10 */              {
              ponta_de_prova(10);
/* 10 11 12 */                                for(k = src[*i-
1]+1;k<=src[*i+1];k++)
/* 12 */              {
              ponta_de_prova(11);
              ponta_de_prova(12);
/* 12 */                addstr(k,dest,j,maxset);
/* 12 */              }
              ponta_de_prova(11);
              ponta_de_prova(13);
/* 13 */            *i = *i + 1;
/* 13 */            }
/* 14 */            else
/* 14 */            {
              ponta_de_prova(14);
/* 14 */              addstr(DASH,dest,j,maxset);
/* 14 */            }
              ponta_de_prova(15);
/* 15 */          }
              ponta_de_prova(16);
/* 16 */        }
              ponta_de_prova(17);
/* 17 */      }
              ponta_de_prova(18);
/* 18 */        *i= *i + 1;
/* 18 */      }
              ponta_de_prova(2);
              ponta_de_prova(19);
              fclose(path);
/* 19 */    }
ÿ
```

Grafo de Programa : Dodash.gfc

```
19

1
 2 0
2
 3 19 0
3
 4 5 0
4
 18 0
5
 6 7 0
6
 17 0
7
 8 9 0
8
 16 0
9
 10 14 0
10
 11 0
11
 12 13 0
12
 11 0
13
 15 0
14
 15 0
15
 16 0
16
 17 0
17
 18 0
18
 2 0
19
 0
```

Desenho do Grafo de Programa :



Dodash.c

Observações Sobre o Desenho :

O desenho não apresenta nenhum problema, nota-se apenas a disposição dos nós 4, 6 e 8 que reflete a característica do uso da variável escopo na determinação do posicionamento.

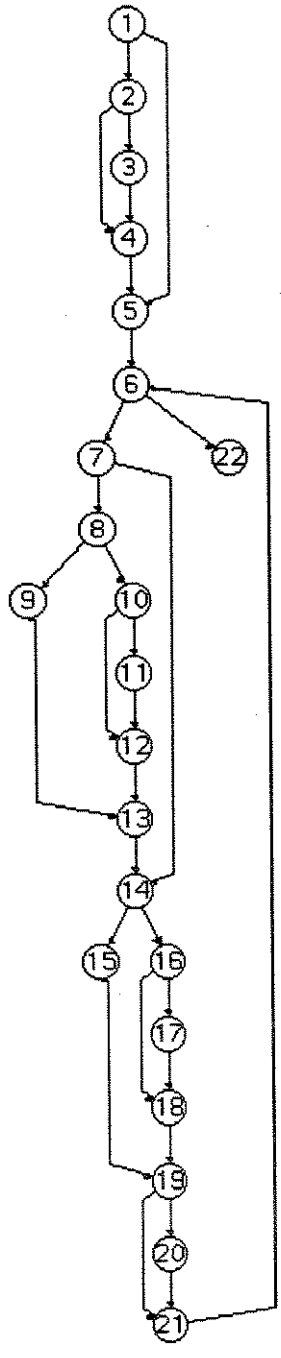
Nome do Programa : Edit.c
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "editconst.h"
extern char lin[MAXSTR];
extern char pat[MAXSTR];
extern char savefile[MAXSTR];
extern int curln, lastln;
void edit()
/* 1 */    {
           FILE * path = fopen("path.tes","w");
           static int printed_nodes = 0;
/* 1 */      int cursave, i, more;
/* 1 */      stcode status;
           ponta_de_prova(1);
/* 1 */      setbuf_();
/* 1 */      pat[0] = ENDSTR;
/* 1 */      savefile[0] = ENDSTR;
/* 1 */      if(getarg(1,savefile))
/* 2 */      {
           ponta_de_prova(2);
/* 2 */        if(doread(0,savefile) == ERR)
/* 3 */        {
           ponta_de_prova(3);
/* 3 */          message("?");
/* 3 */        }
           ponta_de_prova(4);
/* 4 */      }
           ponta_de_prova(5);
/* 5 */      more = getline(lin,stdin,MAXSTR);
/* 6 */      while(more)
/* 7 */      {
           ponta_de_prova(6);
           ponta_de_prova(7);
/* 7 */        i = 0;
/* 7 */        cursave = curln;
/* 7 */        if(getlist(lin,&i,&status) == OK)
/* 8 */        {
           ponta_de_prova(8);
/* 8 */          if(ckglob(lin,&i,&status) == OK)
/* 9 */          {
           ponta_de_prova(9);
/* 9 */            status = doglob(lin,&i,&cursave,&status);
/* 9 */          }
/* 10 */         else
/* 10 */         {
           ponta_de_prova(10);
/* 10 */           if(status != ERR)
/* 11 */           {
           ponta_de_prova(11);
/* 11 */             status = docmd(lin,&i,0,&status);
/* 11 */           }
           ponta_de_prova(12);
/* 12 */         }
           ponta_de_prova(13);
/* 13 */       }
           ponta_de_prova(14);
/* 14 */       if(status == ERR)
/* 15 */       {
           ponta_de_prova(15);
/* 15 */         message("?");
/* 15 */         curln = min(cursave,lastln);
/* 15 */       }
/* 16 */       else
/* 16 */       {
           ponta_de_prova(16);
/* 16 */         if(status == ENDDATA)
/* 17 */         {
           ponta_de_prova(17);
/* 17 */           more = 0;
/* 17 */         }
           ponta_de_prova(18);
/* 18 */       }
           ponta_de_prova(19);
/* 19 */       if(more)
/* 20 */       {
           ponta_de_prova(20);
/* 20 */         more = getline(lin,stdin,MAXSTR);
/* 20 */       }
           ponta_de_prova(21);
/* 21 */       }
           ponta_de_prova(6);
           ponta_de_prova(22);
/* 22 */     clrbuf();
           fclose(path);
/* 22 */     }
ÿ
```

Grafo de Programa : Edit.gfc

```
22

1
  2 5 0
2
  3 4 0
3
  4 0
4
  5 0
5
  6 0
6
  7 22 0
7
  8 14 0
8
  9 10 0
9
  13 0
10
  11 12 0
11
  12 0
12
  13 0
13
  14 0
14
  15 16 0
15
  19 0
16
  17 18 0
17
  18 0
18
  19 0
19
  20 21 0
20
  21 0
21
  6 0
22
  0
```

Desenho do Grafo de Programa :



Edit.c

Observações Sobre o Desenho :

Esse grafo mostra a característica local da ocupação dos espaços dos arcos, observa-se que o arco (2,4) ocupa a mesma coluna dos arcos (10,12) e (15,19), mas todos estão em níveis diferentes.

Nome do Programa : Entab.c

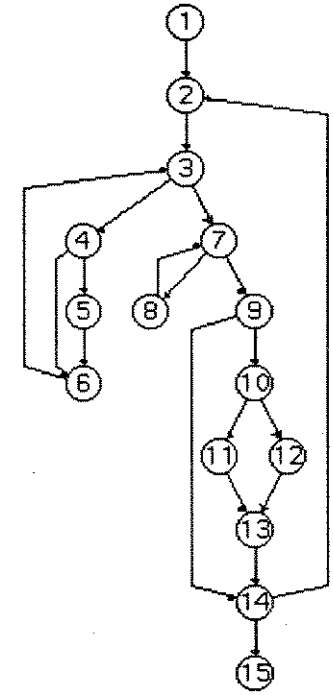
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


void entab()
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int c,col,newcol;
/* 1 */       int tabstops[MAXLINE];
            ponta_de_prova(1);
/* 1 */       settabs(tabstops);
/* 1 */       col = 0;
/* 2 */       do
/* 2 */       {
            ponta_de_prova(2);
/* 2 */         newcol = col;
/* 3 */         while((c = getchar()) == BLANK)
/* 4 */         {
            ponta_de_prova(3);
            ponta_de_prova(4);
/* 4 */           newcol++;
/* 4 */           if(tabpos(newcol,tabstops))
/* 5 */           {
            ponta_de_prova(5);
/* 5 */             putchar(TAB);
/* 5 */             col = newcol;
/* 5 */           }
            ponta_de_prova(6);
/* 6 */         }
            ponta_de_prova(3);
/* 7 */         while(col<newcol)
/* 8 */         {
            ponta_de_prova(7);
            ponta_de_prova(8);
/* 8 */           putchar(BLANK);
/* 8 */           col++;
/* 8 */         }
            ponta_de_prova(7);
            ponta_de_prova(9);
/* 9 */         if(c!=ENDFILE)
/* 10 */        {
            ponta_de_prova(10);
/* 10 */          putchar(c);
/* 10 */          if(c==NEWLINE)
/* 11 */          {
            ponta_de_prova(11);
/* 11 */            col = 0;
/* 11 */          }
/* 12 */          else
/* 12 */          {
            ponta_de_prova(12);
/* 12 */            col++;
/* 12 */          }
            ponta_de_prova(13);
/* 13 */        }
            ponta_de_prova(14);
/* 14 */      }
/* 14 */      while(c!=ENDFILE);;
            ponta_de_prova(15);
            fclose(path);
/* 15 */    }
ÿÿÿ
```

Grafo de Programa : Entab.gfc

```
15

1
  2 0
2
  3 0
3
  4 7 0
4
  5 6 0
5
  6 0
6
  3 0
7
  8 9 0
8
  7 0
9
  10 14 0
10
  11 12 0
11
  13 0
12
  13 0
13
  14 0
14
  2 15 0
15
  0
•
```

Desenho do Grafo de Programa :



Observações Sobre o Desenho :

O desenho não apresenta problemas.

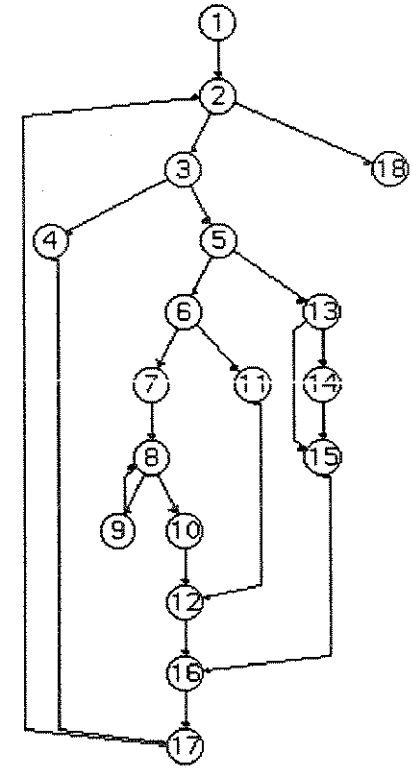
Nome do Programa :Expand.c
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "cte.h"
void expand()
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int c,n;
            ponta_de_prova(1);
/* 2 */       while((c = getchar()) != ENDFILE)
/* 3 */       {
              ponta_de_prova(2);
              ponta_de_prova(3);
/* 3 */         if(c != WARNING)
/* 4 */         {
              ponta_de_prova(4);
/* 4 */           putchar(c);
/* 4 */         }
/* 5 */         else
/* 5 */         {
              ponta_de_prova(5);
/* 5 */           if(isupper(c = getchar()))
/* 6 */           {
              ponta_de_prova(6);
/* 6 */             n = c - 'A' + 1;
/* 6 */             if((c = getchar()) != ENDFILE)
/* 7 */             {
              ponta_de_prova(7);
/* 7 8 9 */           for(n=n;n>=1;n--)
/* 9 */             {
              ponta_de_prova(8);
              ponta_de_prova(9);
/* 9 */               putchar(c);
/* 9 */             }
              ponta_de_prova(8);
              ponta_de_prova(10);
/* 10 */            }
/* 11 */            else
/* 11 */            {
              ponta_de_prova(11);
/* 11 */              putchar(WARNING);
/* 11 */              putchar(n-1+'A');
/* 11 */            }
              ponta_de_prova(12);
/* 12 */          }
/* 13 */          else
/* 13 */          {
              ponta_de_prova(13);
/* 13 */            putchar(WARNING);
/* 13 */            if(c != ENDFILE)
/* 14 */            {
              ponta_de_prova(14);
/* 14 */              putchar(c);
/* 14 */            }
              ponta_de_prova(15);
/* 15 */          }
              ponta_de_prova(16);
/* 16 */        }
              ponta_de_prova(17);
/* 17 */      }
            ponta_de_prova(2);
            ponta_de_prova(18);
            fclose(path);
/* 18 */    }
ÿ
```

Grafo de Programa : Expand.gfc

```
18

1
 2 0
2
 3 18 0
3
 4 5 0
4
 17 0
5
 6 13 0
6
 7 11 0
7
 8 0
8
 9 10 0
9
 8 0
10
 12 0
11
 12 0
12
 16 0
13
 14 15 0
14
 15 0
15
 16 0
16
 17 0
17
 2 0
18
 0
•
```

Desenho do Grafo de Programa :



Expand.c

Observações Sobre o Desenho :

Nesse desenho nota-se que os nós com menos descendentes tendem a ficar mais afastados do centro da estrutura (nó 18). Isso colabora para gerar desenhos centralizados.

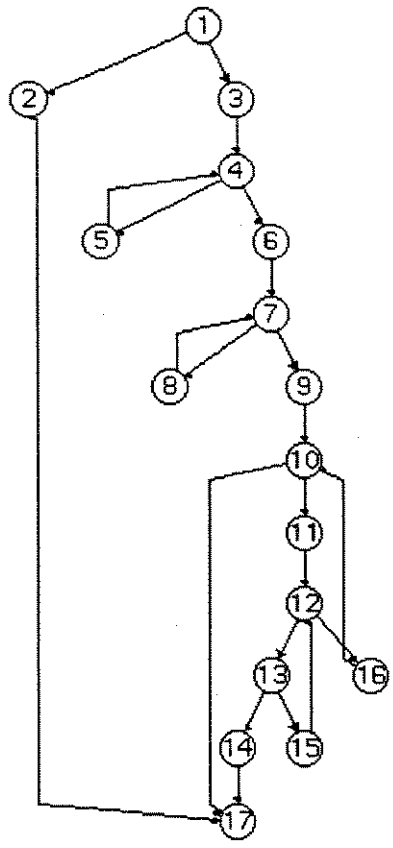
Nome do Programa : Getfns.c
Código Fonte Instrumentado :

```
#define ponta_de_prova(num) if(++printed_nodes % 10)
fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "cte.h"
int errcount;
int nfiles;
char fname[MAXFILES][MAXSTR];
int fstat[MAXFILES];
void getfns()
/* 1 */      {
             FILE * path = fopen("path.tes","w");
             static int printed_nodes = 0;
/* 1 */        int i,j;
             ponta_de_prova(1);
/* 1 */        errcount = 0;
/* 1 */        nfiles = nargs() - 2;
/* 1 */        if(nfiles > 10)
/* 2 */        {
             ponta_de_prova(2);
/* 2 */          printf("archive: too many file names");
             ponta_de_prova(17);
             fclose(path);
/* 2 */          return;
/* 2 */        }
             ponta_de_prova(3);
/* 3 4 5 */    for(i=0;i<nfiles;i++)
/* 5 */        {
             ponta_de_prova(4);
             ponta_de_prova(5);
/* 5 */          getarg(i+3,fname[i]);
/* 5 */        }
             ponta_de_prova(4);
             ponta_de_prova(6);
/* 6 7 8 */    for(i=0;i<nfiles;i++)
/* 8 */        {
             ponta_de_prova(7);
             ponta_de_prova(8);
/* 8 */          fstat[i] = 0;
/* 8 */        }
             ponta_de_prova(7);
             ponta_de_prova(9);
/* 9 10 16 */  for(i=0;i<nfiles-1;i++)
/* 11 */       {
             ponta_de_prova(10);
             ponta_de_prova(11);
/* 11 12 15 */     for(j=(i+1);j<nfiles;j++)
/* 13 */         {
             ponta_de_prova(12);
             ponta_de_prova(13);
/* 13 */           if(equal(fname[i],fname[j]))
/* 14 */           {
             ponta_de_prova(14);
/* 14 */             puts(fname[i]);
/* 14 */             printf("duplicate file name");
             ponta_de_prova(17);
             fclose(path);
/* 14 */             return;
/* 14 */           }
             ponta_de_prova(15);
/* 15 */         }
             ponta_de_prova(12);
             ponta_de_prova(16);
/* 16 */       }
             ponta_de_prova(10);
             ponta_de_prova(17);
             fclose(path);
/* 17 */     }
ÿ
```

Grafo de Programa : Getfns.gfc

```
17

1
 2 3 0
2
 17 0
3
 4 0
4
 5 6 0
5
 4 0
6
 7 0
7
 8 9 0
8
 7 0
9
 10 0
10
 11 17 0
11
 12 0
12
 13 16 0
13
 14 15 0
14
 17 0
15
 12 0
16
 10 0
17
 0
```

Desenho do Grafo de Programa :



Getfns.c

Observações Sobre o Desenho :

Esse desenho apresenta um problema com a escolha da opção central para o posicionamento do arco (16,10). Esse problema pode ser resolvido com pequenas alterações no módulo de desenho.

Nome do Programa : Omatch.c
Código Fonte Instrumentado :

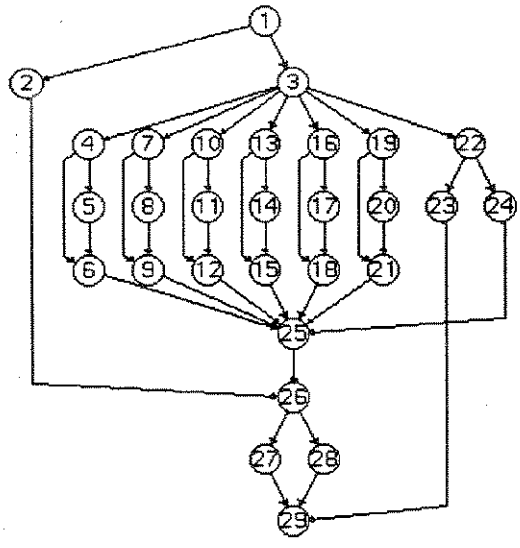
```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "editconst.h"
int omatch(lin,i,pat,j)
char *lin, *pat;
int *i, j;
/* 1 */     {
            FILE * path = fopen("path.tes","w");
            static int printed_nodes = 0;
/* 1 */       int advance = -1;
            ponta_de_prova(1);
/* 1 */       if(lin[*i] == ENDSTR)
/* 2 */       {
            ponta_de_prova(2);
/* 2 */         advance = 0;
/* 2 */       }
/* 3 */       else
/* 3 */       {
            ponta_de_prova(3);
/* 3 */         switch(pat[j])
/* 3 */         {
/* 4 */           case LITCHAR :
/* 4 */           {
            ponta_de_prova(4);
/* 4 */             if(lin[*i] == pat[j+1])
/* 5 */             {
            ponta_de_prova(5);
/* 5 */               advance = 1;
/* 5 */             }
            ponta_de_prova(6);
/* 6 */             break;
/* 6 */           }
/* 7 */           case BOL :
/* 7 */           {
            ponta_de_prova(7);
/* 7 */             if(*i==0)
/* 8 */             {
            ponta_de_prova(8);
/* 8 */               advance = 0;
/* 8 */             }
            ponta_de_prova(9);
/* 9 */             break;
/* 9 */           }
/* 10 */          case ANY :
/* 10 */          {
            ponta_de_prova(10);
/* 10 */            if(lin[*i] != NEWLINE)
/* 11 */            {
            ponta_de_prova(11);
/* 11 */              advance = 1;
/* 11 */            }
            ponta_de_prova(12);
/* 12 */            break;
/* 12 */          }
/* 13 */          case EOL :
/* 13 */          {
            ponta_de_prova(13);
/* 13 */            if(lin[*i] == NEWLINE)
/* 14 */            {
            ponta_de_prova(14);
/* 14 */              advance = 0;
/* 14 */            }
            ponta_de_prova(15);
/* 15 */            break;
/* 15 */          }
/* 16 */          case CCL :
/* 16 */          {
            ponta_de_prova(16);
/* 16 */            if(locate(lin[*i],pat,j+1))
/* 17 */            {
            ponta_de_prova(17);
/* 17 */              advance = 1;
/* 17 */            }
            ponta_de_prova(18);
/* 18 */            break;
/* 18 */          }
/* 19 */          case NCCL :
/* 19 */          {
            ponta_de_prova(19);
/* 19 */            if((lin[*i] != NEWLINE) &&
          (!locate(lin[*i],pat,j+1)))
/* 20 */            {
            ponta_de_prova(20);
/* 20 */              advance =1;
/* 20 */            }
            ponta_de_prova(21);
/* 21 */            break;
/* 21 */          }
/* 22 */          default :
/* 22 */          {
            ponta_de_prova(22);
/* 22 */            if(pat[j] != CLOSURE)
/* 23 */            {
            ponta_de_prova(23);
/* 23 */              printf("in omatch: can't happen");
            ponta_de_prova(29);
            fclose(path);
/* 23 */              return 0;
/* 23 */            }
            ponta_de_prova(24);
/* 24 */          }
/* 25 */        }
          ponta_de_prova(25);
/* 25 */      }
          ponta_de_prova(26);
/* 26 */      if(advance >= 0)
/* 27 */      {
          ponta_de_prova(27);
/* 27 */        *i = *i + advance;
          ponta_de_prova(29);
          fclose(path);
/* 27 */        return 1;
/* 27 */      }
/* 28 */      else
/* 28 */      {
          ponta_de_prova(28);
          ponta_de_prova(29);
          fclose(path);
/* 28 */        return 0;
/* 28 */      }
          ponta_de_prova(29);
          fclose(path);
/* 29 */    }•ÿ
```

Grafo de Programa : Omatch.gfc

```
29

1
  2 3 0
2
  26 0
3
  4 7 10 13 16 19 22 0
4
  5 6 0
5
  6 0
6
  25 0
7
  8 9 0
8
```

```
 9 0
9
 25 0
10
 11 12 0
11
 12 0
12
 25 0
13
 14 15 0
14
 15 0
15
 25 0
16
 17 18 0
17
 18 0
18
 25 0
19
 20 21 0
20
 21 0
21
 25 0
22
 23 24 0
23
 29 0
24
 25 0
25
 26 0
26
 27 28 0
27
 29 0
28
 29 0
29
 0
```

Desenho do Grafo de Programa :



Observações sobre o Desenho :

Esse grafo mostra o posicionamento de uma estrutura case.

Nome do Programa : Subst.c
Código Fonte Instrumentado :

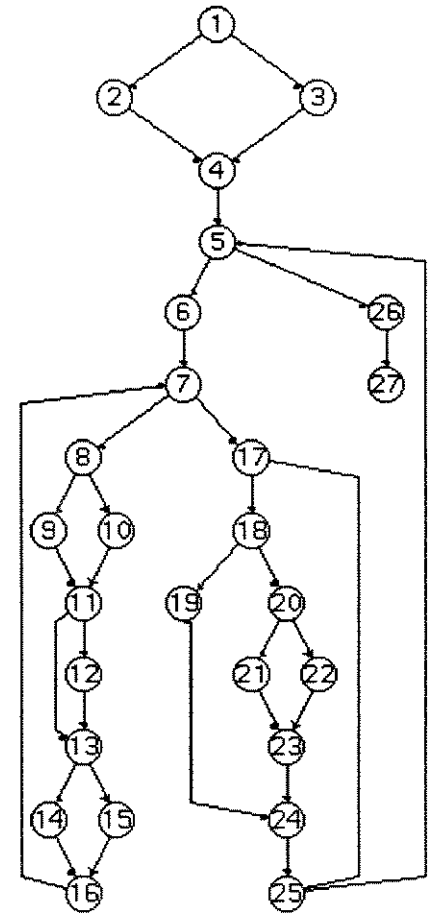
```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "editconst.h"
extern int line1, line2, curln;
extern char pat[MAXSTR];
subst(sub,gflag,glob)
char *sub;
int gflag;
int glob;
/* 1 */     {
              FILE * path = fopen("path.tes","w");
              static int printed_nodes = 0;
/* 1 */       char new[MAXSTR];
/* 1 */       char old[MAXSTR];
/* 1 */       int j,k,lastm,line,m,done,subbed;
/* 1 */       int stat;
              ponta_de_prova(1);
/* 1 */       new[0] = '\0';
/* 1 */       if(glob)
/* 2 */       {
              ponta_de_prova(2);
/* 2 */         stat = OK;
/* 2 */       }
/* 3 */       else
/* 3 */       {
              ponta_de_prova(3);
/* 3 */         stat = ERR;
/* 3 */       }
              ponta_de_prova(4);
/* 4 */       done = (line1 <= 0);
/* 4 */       line = line1;
/* 5 */       while((!done) && (line <= line2))
/* 6 */       {
              ponta_de_prova(5);
              ponta_de_prova(6);
/* 6 */         j = 0;
/* 6 */         subbed = 0;
/* 6 */         gettxt(line,old);
/* 6 */         lastm = -1;
/* 6 */         k = 0;
/* 7 */         while(old[k] != ENDSTR)
/* 8 */         {
              ponta_de_prova(7);
              ponta_de_prova(8);
/* 8 */           if((gflag) || (!subbed))
/* 9 */           {
              ponta_de_prova(9);
/* 9 */             m = amatch(old,k,pat,0);
/* 9 */           }
/* 10 */          else
/* 10 */          {
              ponta_de_prova(10);
/* 10 */            m = -1;
/* 10 */          }
              ponta_de_prova(11);
/* 11 */          if((m >= 0) && (lastm != m))
/* 12 */          {
              ponta_de_prova(12);
/* 12 */            subbed = 1;
/* 12 */            catsub(old,k,m,sub,new,&j,MAXSTR);
/* 12 */            lastm = m;
/* 12 */          }
              ponta_de_prova(13);
/* 13 */          if((m == -1) || (m == k))
/* 14 */          {
              ponta_de_prova(14);
/* 14 */            addstr(old[k],new,&j,MAXSTR);
/* 14 */            k++;
/* 14 */          }
/* 15 */          else
/* 15 */          {
              ponta_de_prova(15);
/* 15 */            k = m;
/* 15 */          }
              ponta_de_prova(16);
/* 16 */        }
              ponta_de_prova(7);
              ponta_de_prova(17);
/* 17 */        if(subbed)
/* 18 */        {
              ponta_de_prova(18);
/* 18 */          if(!addstr(ENDSTR,new,&j,MAXSTR))
/* 19 */          {
              ponta_de_prova(19);
/* 19 */            stat = ERR;
/* 19 */            done = 1;
/* 19 */          }
/* 20 */          else
/* 20 */          {
              ponta_de_prova(20);
/* 20 */            stat = lndelete(line,line,&stat);
/* 20 */            stat = puttxt(new);
/* 20 */            line2 = line2 + curln - line;
/* 20 */            line = curln;
/* 20 */            if(stat == ERR)
/* 21 */            {
              ponta_de_prova(21);
/* 21 */              done = 1;
/* 21 */            }
/* 22 */            else
/* 22 */            {
              ponta_de_prova(22);
/* 22 */              stat = OK;
/* 22 */            }
              ponta_de_prova(23);
/* 23 */          }
              ponta_de_prova(24);
/* 24 */        }
              ponta_de_prova(25);
/* 25 */        line = line + 1;
/* 25 */      }
              ponta_de_prova(5);
              ponta_de_prova(26);
              ponta_de_prova(27);
              fclose(path);
/* 26 */      return(stat);
/* 27 */    }
ÿ
```

Grafo de Programa : Subst.gfc

```
27

1
  2 3 0
2
  4 0
3
  4 0
4
  5 0
5
  6 26 0
6
  7 0
7
  8 17 0
8
  9 10 0
9
  11 0
```

```
10
  11 0
11
  12 13 0
12
  13 0
13
  14 15 0
14
  16 0
15
  16 0
16
  7 0
17
  18 25 0
18
  19 20 0
19
  24 0
20
  21 22 0
21
  23 0
22
  23 0
23
  24 0
24
  25 0
25
  5 0
26
  27 0
27
  0
```

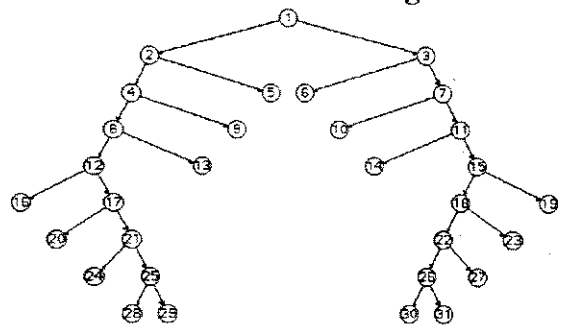
Desenho do Grafo de Programa :



Observações Sobre o Desenho :

Pela característica da numeração observa-se que existe uma estrutura iniciada no nó 7 e que se encerra no 16 e outra iniciada no 17 e encerrada no 25.

Nome do Arquivo : Tidier.gfc
Desenho do Grafo de Programa :



Observações Sobre o Desenho :

Esse é o desenho de uma árvore, colocada como entrada da ViewGraph através da edição manual de uma arquivo .gfc, observa-se que o posicionamento não é mínimo, mas a estrutura se mantem simétrica.

Nome do Programa : Translit.c
Código Fonte Instrumentado :

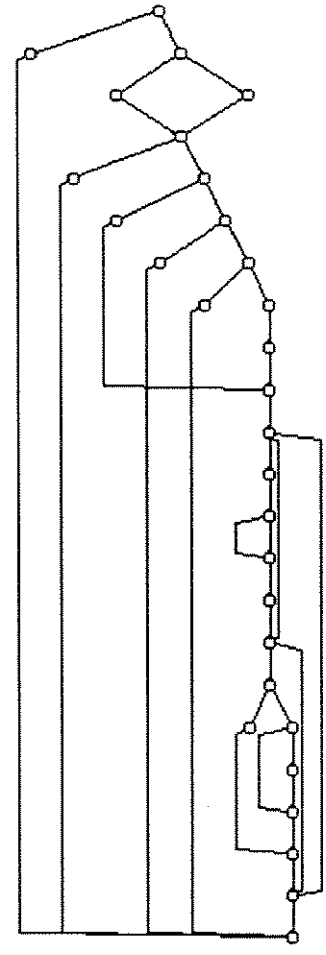
```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include "cte.h"
void translit()
/* 1 */     {
              FILE * path = fopen("path.tes","w");
              static int printed_nodes = 0;
/* 1 */         char fromset[MAXSTR-1], /* conjunto de caracteres a serem substituidos */
        toset[MAXSTR-1], /* corresponde `as substituicoes */
        arg[MAXSTR-1];
/* 1 */     int i,c,lastto, /* comprimento de toset */
        allbut, /* todos menos um conjunto */
        squash;
              ponta_de_prova(1);
/* 1 */       if(!getarg(1,arg))
/* 2 */       {
              ponta_de_prova(2);
/* 2 */         message("usage: translit from to");
              ponta_de_prova(30);
              fclose(path);
/* 2 */         return;
/* 2 */       }
              ponta_de_prova(3);
/* 3 */       allbut = (arg[0] == NEGATE);
/* 3 */       if(allbut)
/* 4 */       {
              ponta_de_prova(4);
/* 4 */         i = 1;
/* 4 */       }
/* 5 */       else
/* 5 */       {
              ponta_de_prova(5);
/* 5 */         i = 0;
/* 5 */       }
              ponta_de_prova(6);
/* 6 */       if(!makeset(arg,i,fromset,MAXSTR))
/* 7 */       {
              ponta_de_prova(7);
/* 7 */         message ("translit: from set too large");
              ponta_de_prova(30);
              fclose(path);
/* 7 */         return;
/* 7 */       }
              ponta_de_prova(8);
/* 8 */       if(!getarg(2,arg))
/* 9 */       {
              ponta_de_prova(9);
/* 9 */         toset[0] = ENDSTR;
/* 9 */       }
/* 10 */      else
/* 10 */      {
              ponta_de_prova(10);
/* 10 */        if(!makeset(arg,0,toset,MAXSTR))
/* 11 */        {
              ponta_de_prova(11);
/* 11 */          message ("translit: to set too large");
              ponta_de_prova(30);
              fclose(path);
/* 11 */          return;
/* 11 */        }
/* 12 */        else
/* 12 */        {
              ponta_de_prova(12);
/* 12 */          if((strlen(fromset)) < (strlen(toset)))
/* 13 */          {
              ponta_de_prova(13);
/* 13 */            message ("translit: from shorter than to");
              ponta_de_prova(30);
              fclose(path);
/* 13 */            return;
/* 13 */          }
              ponta_de_prova(14);
/* 14 */        }
              ponta_de_prova(15);
/* 15 */      }
              ponta_de_prova(16);
/* 16 */      lastto = strlen(toset) - 1;
/* 16 */      squash = ((strlen(fromset)>lastto+1)||(allbut));
/* 17 */      do
/* 17 */      {
              ponta_de_prova(17);
/* 17 */        i = xindex(fromset,c=getchar(),allbut,lastto);
/* 17 */        if((squash)&&(i>=lastto)&&(lastto+1>0))
/* 18 */        {
              ponta_de_prova(18);
/* 18 */          putchar(toset[lastto]);
/* 19 */          do
/* 19 */          {
              ponta_de_prova(19);
/* 19 */                                                  i = xindex(fromset,c=getchar(),allbut,lastto);
              ponta_de_prova(20);
/* 19 */          }
/* 20 */          while(i>=lastto);:
              ponta_de_prova(21);
/* 21 */        }
              ponta_de_prova(22);
/* 22 */        if(c != ENDFILE)
/* 23 */        {
              ponta_de_prova(23);
/* 23 */          if((i>=0) && (lastto+1>0))
/* 24 */          {
              ponta_de_prova(24);
/* 24 */            putchar(toset[i]);
/* 24 */          }
/* 25 */          else
/* 25 */          {
              ponta_de_prova(25);
/* 25 */            if(i==-1)
/* 26 */            {
              ponta_de_prova(26);
/* 26 */              putchar(c);
/* 26 */            }
              ponta_de_prova(27);
/* 27 */          }
              ponta_de_prova(28);
/* 28 */        }
              ponta_de_prova(29);
/* 29 */      }
/* 29 */      while(c!=ENDFILE);:
              ponta_de_prova(30);
              fclose(path);
/* 30 */    }
ÿ
```

Grafo de Programa : Translit.gfc

```
30

1
  2 3 0
2
  30 0
3
  4 5 0
4
  6 0
5
  6 0
6
  7 8 0
7
  30 0
```

```
8
  9 10 0
9
  16 0
10
  11 12 0
11
  30 0
12
  13 14 0
13
  30 0
14
  15 0
15
  16 0
16
  17 0
17
  18 22 0
18
  19 0
19
  20 0
20
  19 21 0
21
  22 0
22
  23 29 0
23
  24 25 0
24
  28 0
25
  26 27 0
26
  27 0
27
  28 0
28
  29 0
29
  17 30 0
30
  0
```

Desenho do Grafo de Programa :



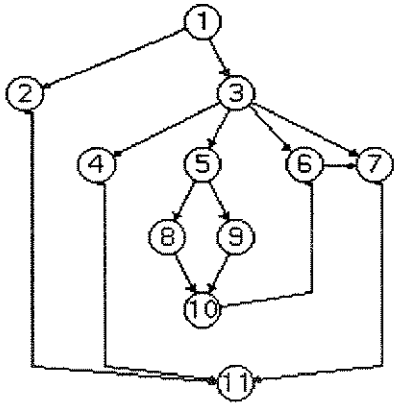
Observações Sobre o Desenho :

Esse grafo foi desenhado em modo resumido, sem numeração e com raio menor para os nós.

Nome do Arquivo : Xcase2.gfc

```
12

1
 2 3 0
2
 11 0
3
 4 5 6 7 0
4
 11 0
5
 8 9 0
6
 7 0
7
 11 0
8
 10 0
9
 10 0
10
 6 0
11
 12 0
12
 0
```

Desenho do Grafo :



Observações :

Observa-se nesse grafo o cálculo do nível do nó 11 (nó de saída do case).