Desenho Automático de Diagramas

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pela srta. Maria Inês Vale da Silva e aprovada pela comissão julgadora.

Campinas, 17 de junho de 1994

Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho

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

Resumo

Diagramas são largamente utilizados como forma de representação gráfica dos mais diversos tipos de informação. A *LegoShell* **[Dru89]** é uma linguagem gráfica, em desenvolvimento no Projeto A_HAND, projetada para representar objetos distribuídos. Nos diagramas *LegoShell*, tais objetos aparecem conectados entre si na forma de um grafo orientado.

A legibilidade de um diagrama *LegoShell* é uma qualidade muito desejável. Dependendo da complexidade, o desenho de um diagrama *LegoShell* pode ser bastante confuso. Desse modo, uma função de desenho automático de seus diagramas pode ser necessária, para garantir a clareza de sua informação.

Para definir legibilidade em um diagrama, estabelecemos alguns critérios de estética, que denotam aspectos gráficos desejados em um desenho legível. Critérios, como distribuição uniforme dos objetos e minimização do número de cruzamentos entre conexões, devem ser considerados no desenho de diagramas da *LegoShell*. Assim, algoritmos para o desenho de diagramas são projetados a partir da definição de tais critérios.

Neste trabalho, foi realizado um levantamento de algoritmos propostos para o desenho de diagramas ou grafos, a partir da definição de alguns critérios de estética, e do tipo do diagrama que está sendo representado. Alguns desses algoritmos foram implementados para experimentação e análise junto aos diagramas da *LegoShell*.

Abstract

Diagrams are widely used as graphical representation for many types of information. *LegoShell* **[Dru89]** is a graphical language, under development at A_HAND Project, designed to represent distributed objects. In *LegoShell* diagrams, these objects appear connected like an oriented graph.

Since *LegoShell*'s practical examples can get very confusing, automatic layout tools can be necessary if we want to guarantee the clarity of its information.

We establish some criteria to evaluate a diagram's readability, expressing some graphical aspects expected from a readable drawing. Criteria as uniform distribution of objects and minimization of the number of crossings between connections should be considered in drawings of *LegoShell*-like diagrams. Algorithms for diagram drawing are designed from these criteria definitions.

This work includes a survey about algorithms for diagram or graph drawing. This survey derives from the definition of some criteria and the type of the represented diagram. Some algorithms were implemented in order to experiment and analyze their performance with *LegoShell* diagrams as subjects.

Agradecimentos

A Deus, que me concedeu tantos momentos divinos durante o período do mestrado, e nunca me deixou perder a fé nos momentos mais difíceis.

A meus pais, Inês e Sebastião, que mesmo distante, sempre estiveram muito perto do meu pensamento e do meu coração. Muito obrigada pelo amor, paciência e força que vocês me dedicaram durante toda minha vida, especialmente durante esses três anos que estive longe de casa. Exclusivamente a vocês, dedico cada minuto deste trabalho.

A meus irmãos, Ricardo e Anice, que também torceram por mim durante todo esse tempo, e tiveram que agüentar a dura saudade da irmãzinha querida.

A todos os parentes paternos e maternos, que sempre me deram apoio para ir até o final dessa longa jornada.

Aos *conspiradores*, com quem tive a chance de aprender o verdadeiro significado da palavra "amizade". Espero que nos encontremos ainda muitas vezes pelas estradas da vida e, claro, do mundo.

A Nilza e Paulo, que mesmo do outro lado do mundo, estão mais perto do que imaginam. Obrigada pelos momentos indescritíveis que passamos juntos.

A Carrilho, que sempre esteve mais presente do que a própria tese. Aliás, você é um presente! Obrigada por seu carinho, sua companhia e sua amizade.

À família do Carrilho (Da. Ana, S. Zeíco, Neide, Ivanildo, Lena & cia., Rose, etc.), por também ter sido minha família durante todo esse tempo.

A Cláudio, com quem dividi tantos momentos felizes nesse último ano. Espero que ainda tenhamos muito tempo e motivos para rir juntos.

A Carlinhos, que resolveu nos deixar de repente, mas sempre vai morar num lugar muito especial do coração.

A Visoli(lein), que mesmo sumindo de vez em quando, é a quem posso chamar "amigo de todas as horas".

A todos os amigos da turma de 91 (Lincoln, Mário, Heitor, Nabor, Marcus, Flávia, Atta, Adauto, Juliano, etc.), com quem compartilhei tantos momentos de trabalho e alegrias (afinal, as reuniões de Teoria eram ou não divertidas?).

A todos os amigos das turmas de 92 e 93 (Evandro, Fileto, Fátima, Carrard, Victor, Tutumi, Anderson, Elaine, Humberto, Clevan, Anderson, Claudão, Flávio, Carlos, Ronaldos, Rober, Daniel, Jaime, George, Nuccio, Luís, etc.). Obrigada por seus sorrisos constantes, que sempre tornaram meus dias mais fáceis e bonitos.

A Kátia e Nair, que além de grandes amigas, são agora parte da minha família.

A Nildinha, Jane e Ana Lúcia, paraibanas "arretadas" que deixaram boas lembranças dos tempos do Cambuí.

A Kamienski, que me ensinou muito sobre amizade e serenidade.

A Da. Lúcia, pelo carinho recebido durante o ano que morei em sua casa.

Aos amigos de Fortaleza (Luiza, Suzi, Eveline, Cléa, Airles, Sandra, Ceres, Franklin, Marum, Manoel, Ríverson, etc.), que provaram que o tempo e a distância não conseguem apagar uma verdadeira amizade.

Aos tios João e Edite, por terem me ajudado a conquistar toda a educação que hoje trago comigo. Muito obrigada pela força que sempre me deram!

Aos amigos Piedade, Mazé, Socorro, tia Pequena, tia Edite, tia Maria, tia Alzira, tio Arimatéia, tia Rita, Rosângela e Da. Socorro, que nunca esqueceram de mim em suas orações.

A todos os integrantes do projeto A_HAND, que no último ano, tornaram-se pessoas muito especiais para mim.

A Wanderley Ceschim, pela paciência de me agüentar todo dia falando: a culpa do meu programa não estar funcionando é sua!

A Elaine, Humberto, Nuccio, Teles, Sheila, e Furuti, pela preciosa ajuda na correção de alguns "bugs" da dissertação. A Islene, por todas as idéias mirabolantes trocadas durante o desenvolvimento da tese.

Aos professores Marcus Vinícius, Xavier, Lucchesi, Hans e Pedro Rezende, pelo esclarecimento de algumas dúvidas, cujas respostas não se encontravam nos artigos.

Aos órgãos de fomento CNPq e FAEP, e ao Projeto A_HAND, pelo financiamento deste trabalho.

Índice

Capítulo 1 Introdução

1.1 A *LegoShell* 2
1.2 O desenho de diagramas da *LegoShell* 4
1.3 Algoritmos para o desenho automático de grafos 6
1.4 Conceitos gerais 7
1.4.1 Aplicações da busca em profundidade 8

Capítulo 2 Árvores

2.1 Introdução 13
2.2 Algoritmo de Wetherell e Shannon 14
2.3 Algoritmo de Vaucher 15
2.4 Algoritmo de Reingold e Tilford 17

Capítulo 3 Grafos Planares

3.1 Introdução 21
3.2 Desenhos de linha-reta 27
3.3 Desenhos de representação de visibilidade 31
3.4 Desenhos ortogonais em grade 34

Capítulo 4 Grafos Orientados Acíclicos

4.1 Introdução 39
4.2 Método do Baricentro 40
4.2.1 Modificações do algoritmo de Sugiyama et al. 49

Capítulo 5 Grafos Gerais não-orientados

5.1 Introdução 51
5.2 *Simulated Annealing* 51
5.2.1 *Simulated Annealing* para o desenho de grafos 53
5.3 *Random Search* 57
5.3.1 *Random Search* para o desenho de grafos 57
5.4 Desenho de estruturas em rede 58

Capítulo 6 Método geral de desenho de grafos

6.1 Introdução 61
6.2 Fases do método 62
6.2.1 Planarização 62
6.2.2 Ortogonalização 64
6.2.3 Compactação 66

Capítulo 7 Implementação e Resultados

7.1 Introdução 69
7.2 O editor da *LegoShell* 70
7.3 Algoritmos Implementados 75
7.3.1 *Simulated Annealing* e *Random Search* 76
7.3.2 Método do Baricentro 79

Capítulo 8 Conclusões e Trabalhos Futuros

8.1 Trabalhos Futuros 85

Apêndice A Sintaxe da Linguagem de Definição de Diagramas

Apêndice B Exemplos

Apêndice C Referências Bibliográficas

Lista de Figuras

FIGURA 1-1	Classificação dos objetos *LegoShell.*	3
FIGURA 1-2	Exemplo de uma computação *LegoShell.*	4
FIGURA 1-3	Exemplo de uma computação abstraída.	4
FIGURA 1-4	Um grafo e uma palmeira correspondente. Segmentos em preto (vermelho) representam arestas de árvore (cíclicas).	9
FIGURA 1-5	(a) Grafo G. (b) Uma palmeira com seus valores *LOWPT*1 entre [ ] e pontos de articulação marcados com *. (c) Componentes biconexas de *G*.	10
FIGURA 2-1	Desenhos de uma mesma árvore, a partir de critérios diferentes.	14
FIGURA 2-2	Estrutura auxiliar para uma árvore de 3 nós.	15
FIGURA 2-3	Um exemplo de execução do algoritmo de Vaucher.	17
FIGURA 2-4	Elos entre subárvores de alturas diferentes.	18
FIGURA 2-5	Árvore desenhada pelo algoritmo de Reingold e Tilford.	19
FIGURA 3-1	Exemplo de execução do algoritmo de cálculo de st-numeração. (a) Grafo de entrada. (b) Palmeira encontrada no primeiro passo. (c) Caminhos gerados no segundo passo.	24
FIGURA 3-2	Exemplo de execução do algoritmo de Chiba et al **[CON85].**	29
FIGURA 3-3	Resultado do algoritmo de Tutte.	30
FIGURA 3-4	Resultado do algoritmo de Chiba et al.	30
FIGURA 3-5	Resultado do algoritmo de Read.	30
FIGURA 3-6	Resultado do algoritmo de Chrobak e Payne.	30
FIGURA 3-7	Um grafo planar e uma representação de visibilidade correspondente.	31

FIGURA 3-8	Um grafo não-orientado G.	33
FIGURA 3-9	Grafos orientados $D^{\wedge}$ (em preto) e D^{*} (em vermelho) derivados do grafo G. O valor $\alpha(f)$ para cada face f aparece entre ().	33
FIGURA 3-10	Representação de visibilidade para G.	33
FIGURA 3-11	Substituições realizadas no passo 2.	35
FIGURA 3-12	Informações adicionais de uma representação ortogonal.	36
FIGURA 3-13	Exemplo de transformação T1.	36
FIGURA 3-14	Exemplo de transformação T2.	37
FIGURA 3-15	Exemplo de transformação T3.	37
FIGURA 3-16	Exemplo de execução d o algoritmo para desenho ortogonal em grade.	38
FIGURA 4-1	Conceitos relacionados a hierarquias. Em (b) os vértices-fantasmas são representados explicitamente.	40
FIGURA 4-2	Hierarquia de dois níveis G_0.	44
FIGURA 4-3	Ordenação final dos vértices de acordo com o algoritmo Baricentro.	45
FIGURA 4-4	Hierarquia de três níveis G_0.	46
FIGURA 4-5	Hierarquia de três níveis G_0, depois da execução do algoritmo POST_ANT.	47
FIGURA 5-1	Exemplo de execução do algoritmo *Simulated Annealing* **[ES90]**.	56
FIGURA 5-2	Desenho do grafo K_5 de acordo com os critérios de simetria e de número de cruzamentos.	59
FIGURA 5-3	Exemplo de execução do algoritmo Spring **[KK88]**.	60
FIGURA 6-1	Representação hierárquica para o desenho de diagramas.	62
FIGURA 6-2	Duas representações planares de um mesmo grafo, onde (b) apresenta uma face F aninhada a mais que (a).	64
FIGURA 6-3	(a) Representação planar corrente, onde a aresta não-planar $e=(v,w)$ deve ser reintroduzida. (b) O grafo G_e, com o caminho c em vermelho. (c) Representação planar depois da introdução da aresta e, com o vértice fictício em vermelho.	64
FIGURA 6-4	Um exemplo de ortogonalização rápida.	65
FIGURA 6-5	Transformação de um grafo através de normalização.	66
FIGURA 7-1	Atual interface da *LegoShell*.	70
FIGURA 7-2	Diálogo de edição dos objetos *LegoShell*.	71
FIGURA 7-3	Diálogo de edição dos parâmetros do processo de *Simulated Annealing*.	72
FIGURA 7-4	Diálogo de edição dos parâmetros do processo de *Random Search*.	73
FIGURA 7-5	Módulos principais do editor da *LegoShell*.	74
FIGURA 7-6	Direções permitidas no movimento de um vértice.	76

FIGURA B-1	Primeiro exemplo: um diagrama com 6 objetos e 9 conexões.	90
FIGURA B-2	Método SA aplicado ao diagrama da figura B-1.	90
FIGURA B-3	Método RS aplicado ao diagrama da figura B-1.	91
FIGURA B-4	Método BC aplicado ao diagrama da figura B-1.	91
FIGURA B-5	Segundo exemplo: um diagrama com 21 objetos e 23 conexões.	92
FIGURA B-6	Método SA aplicado ao diagrama da figura B-5.	92
FIGURA B-7	Método RS aplicado ao diagrama da figura B-5.	93
FIGURA B-8	Método BC aplicado ao diagrama da figura B-5.	93
FIGURA B-9	Terceiro exemplo: um diagrama com 42 objetos e 51 conexões.	94
FIGURA B-10	Método SA aplicado ao diagrama da figura B-9.	95
FIGURA B-11	Método RS aplicado ao diagrama da figura B-9.	96
FIGURA B-12	Método BC aplicado ao diagrama da figura B-9.	97

Capítulo 1

Introdução

Diagramas são amplamente utilizados em várias áreas do conhecimento humano. Em matemática e ciência da computação esse fato é mais que reconhecido. Em ciência da computação, a área de sistemas de informação faz grande uso de diagramas como recurso de visualização de objetos que se relacionam. Diagramas de entidade-relacionamento, diagramas de fluxo de dados, redes de Petri, autômatos de estados finitos, grafos de chamadas de procedimentos, projeto de banco de dados e representação de sistemas distribuídos são apenas algumas aplicações.

Uma vez que diagramas sejam utilizados para representar tais aplicações, a característica mais esperada é a legibilidade da informação, ou seja, a facilidade da captura de seu significado pelos olhos humanos. Fatores como complexidade visual, regularidade da estrutura, simetria, consistência da representação dos objetos, modularidade, tamanho e formato dos objetos podem afetar a legibilidade de um diagrama.

Indiscutivelmente, legibilidade é uma característica bastante subjetiva, que depende do tipo de diagrama a ser desenhado e do gosto particular de cada pessoa. Dependendo da complexidade do diagrama, a tarefa de desenhá-lo de forma legível pode ser uma tarefa consumidora de tempo, e nem sempre de resultados satisfatórios, se deixada a cargo do usuário .

Há algum tempo, editores gráficos de diagramas eram limitados apenas às funções de criação e remoção de objetos, restando ao usuário a missão de desenhar o diagrama de forma legível. Atualmente, essa limitação tem se reduzido através da função de desenho automático dos diagramas fornecida pelo próprio editor. Esse assunto vem ganhando interesse crescente por pesquisadores de várias áreas (vide por exemplo **[TBB88]**, **[DM90]**, **[TN87]**, **[PT90]**), principalmente geometria computacional e teoria dos grafos.

O mapeamento matemático, encontrado para o problema da obtenção de desenhos legíveis de diagramas, consiste em encontrar uma atribuição ótima de coordena-

das, ou próxima da ótima, de um grafo abstrato no plano, onde o critério de otimalidade está relacionado com o significado de "legibilidade" para o usuário. Desse modo, um grafo está para um diagrama como seus vértices e arestas estão para os objetos e relacionamentos do diagrama; e o problema de desenhar um diagrama de forma legível passa a ser um problema de desenho legível de um grafo no plano.

Vários algoritmos foram projetados com o intuito de desenhar um grafo automaticamente, mesmo que o objetivo principal não fosse legibilidade. Alguns desses algoritmos têm aplicação no projeto de circuitos integrados. Contudo, algoritmos projetados para tal propósito não se aplicam aos diagramas de sistemas de informação devido, por exemplo, ao resultado pouco estético da compactação das conexões do circuito, e ao grande consumo de tempo.

A motivação para este trabalho são os diagramas de um editor gráfico utilizado para a representação de objetos distribuídos, que servirá como interface para a *LegoShell*, uma das linguagens desenvolvidas dentro do projeto A_HAND[1].

1.1 A *LegoShell*

A *LegoShell* **[Dru89]** é uma linguagem gráfica de especificações de comandos complexos chamados computações. Uma computação é um conjunto de objetos de um sistema de computação real conectados através de portas de comunicação. A linguagem estende o conceito de *pipe* **[Bac86]** do Unix, que é restrito a conexões lineares de fluxos de dados, para grafos bidimensionais. Além de conectores *pipe*, a *LegoShell* suporta conectores com semântica de caixa de correspondência[2] e conectores com semântica de disseminação[3].

Como a *LegoShell* introduz conectores do tipo muitos-para-muitos, e essas conexões não são diretamente fornecidas pelo núcleo do Unix, sua implementação depende da elaboração de um sistema de execução que simula novas chamadas ao sistema. Essa parte da *LegoShell* está sendo desenvolvida atualmente em **[dC94]**.

Objetos *LegoShell* podem ser agrupados segundo três classificações:

- objeto primitivo/composto: se um objeto é mapeado diretamente em uma entidade física como um arquivo ou um periférico, ele é chamado primitivo. Caso um objeto seja formado a partir do agrupamento de outros objetos, dizemos que ele é composto;
- objeto ativo/passivo: se um objeto gera ou consume dados (através de suas portas de comunicação) sob sua própria requisição, dizemos que ele é ativo, caso contrário, passivo; e
- objeto com/sem conteúdo semântico: se um objeto tem existência mesmo fora do ambiente *LegoShell*, dizemos que ele possui conteúdo semântico. Difere dos

1. Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados.
2. *mailbox*, em inglês.
3. *broadcast*, em inglês.

conectores, que só têm sentido dentro de uma computação interligando outros objetos, sendo considerados objetos sem conteúdo semântico.

A figura 1-1 resume a classificação dos objetos *LegoShell:*

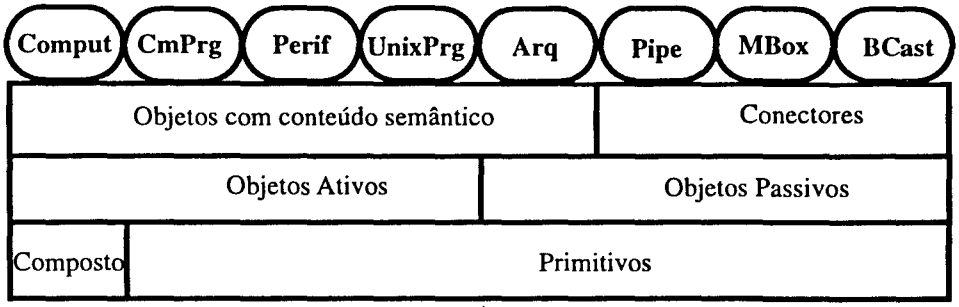


FIGURA 1-1 Classificação dos objetos *LegoShell.*

Abaixo segue uma breve descrição de cada objeto:

- computação: uma computação *LegoShell* é um objeto composto contendo outros objetos conectados entre si através de suas portas. Portas desconectadas formam a interface de comunicação do objeto e são visíveis externamente. Uma computação é representada externamente como uma caixa contendo portas de entrada e saída;
- programas Cm **[Fur91] [Tel93]**: um programa escrito em linguagem Cm contendo portas de entrada e saída definidas no programa;
- dispositivos periféricos: quaisquer periféricos encontrados em um sistema Unix podem ser utilizados em uma computação *LegoShell.* As portas disponíveis indicam se o periférico é um dispositivo de entrada, de saída ou ambos;
- programas Unix: programas executáveis no sistema Unix;
- arquivos: os arquivos do usuário também estão disponíveis e podem ser utilizados em qualquer computação. O arquivo é uma entidade passiva, e só consome ou produz dados sob demanda. As permissões do arquivo (escrita/leitura/execução) são refletidas pela presença ou não de portas de entrada e saída em sua representação visual;
- *pipe*: *pipes* possuem o mesmo significado que em Unix, e implementam um canal de comunicação unidirecional. É o conector mais simples da *LegoShell* e é representado graficamente apenas por uma linha conectando duas portas de objetos de uma computação;
- *mailbox*: assim como o conector de disseminação, o *mailbox* permite várias portas de entradas e de saída, mas cada dado que chega em uma porta de saída é consumido e não está mais disponível através de outras portas; e
- *broadcast*: conector de disseminação. Os dados recebidos pelas portas de entrada deste conector são distribuídos para todas as portas de saída. A seqüência em que os dados serão enviados às portas de saída é a mesma para todas as portas.

A figura 1-2 apresenta um exemplo de computação composta de um arquivo de dados, um conector *mailbox*, três programas Cm, um conector *broadcast*, e dois

periféricos. Essa computação representa graficamente processos responsáveis pela ordenação de um conjunto de dados, contidos em **Partic**.

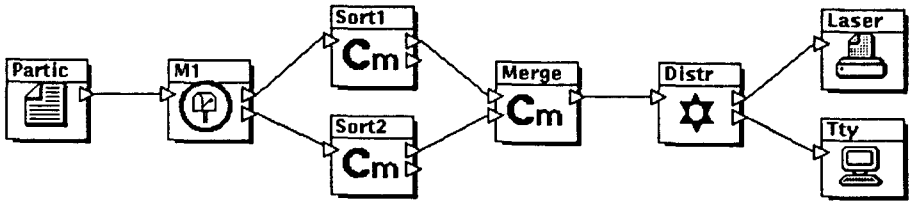


FIGURA 1-2 Exemplo de uma computação *LegoShell*.

Um outro conceito a ser implementado pela *LegoShell* é a abstração de computações. Abstrair uma computação significa criar uma nova computação a partir de um subconjunto das componentes de alguma outra computação. Expansão é o processo inverso da abstração, onde uma computação é substituída por todas as suas componentes. Se, por exemplo, abstrairmos os programas Sort e Merge, junto com o conector mailbox, em uma só computação chamada MSort, poderíamos utilizá-la na formação de uma outra computação, como mostra a figura 1-3.

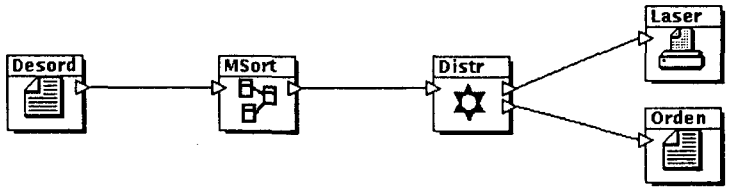


FIGURA 1-3 Exemplo de uma computação abstraída.

Os primeiros ensaios de como deveria ser projetado o editor gráfico da *LegoShell* encontram-se em **[Piñ91]**. Ainda naquele trabalho, foi prevista a necessidade de um redesenhador automático para os diagramas da *LegoShell*, uma vez que computações muito complexas poderiam ser representadas graficamente de maneira confusa.

Nós pretendemos, com este trabalho, fornecer ao editor gráfico da *LegoShell* a capacidade do redesenho legível de suas computações. Para isso, foi realizado um estudo dos diagramas resultantes de computações *LegoShell*, e um levantamento de soluções relacionadas a esse e outros problemas correlatos.

Nas seções a seguir, veremos uma melhor definição do problema e, nos capítulos subseqüentes, um levantamento do estudo realizado com o intuito do desenho legível de diagramas em geral.

1.2 O desenho de diagramas da *LegoShell*

Em um nível conceitual, podemos identificar o problema em questão como desenhar um diagrama *LegoShell* de forma legível. Desenhar um diagrama significa posicionar objetos e rotear as ligações entre eles de modo que a informação de uma computação, representada graficamente, seja facilmente capturada. Nesta seção,

analisaremos aspectos dos diagramas que devem ser considerados na busca de um solução.

Um diagrama *LegoShell* pode representar desde uma computação simples, como MergeSort, até uma computação bastante complexa, como uma que represente a linha de produção de uma indústria. Logo, é difícil definir o comportamento dos objetos de um diagrama *LegoShell*. A quantidade de objetos e conexões em uma computação qualquer, assim como sua disposição inicial, também não seguem qualquer regra.

Devido ao fato de que objetos *LegoShell* representam processos comunicantes, podemos inferir que uma certa ordem é influenciada pela dependência de execução no tempo entre esses processos. Entretanto, o que temos é informação topológica e, geralmente, geométrica do diagrama. Extrair informação semântica do diagrama demandaria, no mínimo, um bom conhecimento do comportamento das computações.

Se quiséssemos abstrair informação semântica do diagrama, um preprocessamento deveria ser realizado antes de qualquer tentativa de melhora no desenho. Impor restrições ao desenho do diagrama seria uma forma de fornecer alguma semântica ao desenho, mas aqui não consideraremos ainda esta possibilidade.

Um outro ponto a ser considerado é o número variável de portas que um objeto *LegoShell* pode vir a ter, o que significa objetos de tamanhos também variáveis dependendo do número de portas que eles contêm. Desse modo, objetos com diferentes tamanhos não devem ser esquecidos na formulação do problema.

Dispondo de uma grade de desenho retangular finita para o desenho de um diagrama *LegoShell*, podemos definir o problema mais especificamente como atribuir coordenadas (x,y), válidas dentro da grade de desenho, aos objetos pertencentes a uma computação *LegoShell*, de tal modo que o resultado não seja confuso aos olhos do espectador.

Não ser confuso está relacionado com a noção de legibilidade que mencionamos anteriormente. De fato, legibilidade costuma ser expressa através de alguns critérios, denominados "critérios de estética para o desenho de grafos" **[TBB88]**, geralmente formulados como objetivos de otimização para alguns algoritmos. Assim, legibilidade pode ser medida através de um ou mais dos critérios relacionados a seguir:

- distribuição uniforme dos objetos dentro da área de desenho;
- minimização do número de cruzamentos entre arestas;
- minimização da área ocupada pelo desenho;
- minimização do número de quebras[4] nas arestas;
- minimização do tamanho global das arestas;

4. Quebra em uma aresta significa uma mudança de direção de um certo ângulo, no desenho da aresta .

- minimização da diferença entre as dimensões dos vértices; e
- apresentação de simetrias.

Nós consideramos os dois primeiros critérios fundamentais para o desenho legível de digramas em geral. No decorrer deste trabalho, consideraremos outros critérios de legibilidade, além daqueles antes enumerados.

A próxima seção introduz o assunto que vem a ser o caminho de solução para o nosso problema e outros correlatos: o desenho automático de grafos.

1.3 Algoritmos para o desenho automático de grafos

Algoritmos projetados para o desenho automático de grafos, visando legibilidade, consideram, em geral, mais de um dos critérios recém-citados. Daí podem surgir conflitos com relação ao que é mais importante para o desenho. Para resolver tais impasses, podemos atribuir uma prioridade para cada critério, ou estabelecer uma ordem no tratamento de um subconjunto deles.

Critérios de estética podem não ser suficientes para definir legibilidade para uma aplicação específica, podendo ser ainda necessária a imposição de algumas restrições ao desenho. Restrições costumam estar relacionadas à semântica da aplicação e, em geral, devem atingir apenas um grupo de objetos do desenho. As restrições a seguir são sugeridas em alguns trabalhos já publicados na área:

- proximidade obrigatória de um grupo de objetos em uma ordem fixa ou não;
- prefixação de posicionamento de alguns objetos;
- localização central ou periférica de objetos;
- representação de um grupo de objetos como uma figura predefinida;
- quantidade máxima de quebras e de cruzamentos em uma aresta individual;
- limitação do número de vértices em uma determinada região; e
- posicionamento relativo entre vértices em termos da distância no grafo entre eles.

A classe na qual o grafo se enquadra é, na verdade, o que mais influencia na criação ou escolha do algoritmo. A grande maioria dos trabalhos relacionados ao desenho de grafos se encontra em uma das classes abaixo relacionadas:

- árvores;
- grafos planares;
- grafos orientados acíclicos (ou hierarquias); e
- grafos gerais não-orientados.

Há uma outra classificação de algoritmos relacionada não com uma classe ou outra de grafo, mas com as características particulares de uma representação diagramática específica. Para tais casos, algoritmos mais direcionados para a aplicação devem ser projetados, como em **[BNT84]** e **[BNT86]**, que tratam, respectivamente,

do desenho de diagramas de entidade-relacionamento e de diagramas de fluxo de dados.

Podemos dizer ainda que alguns critérios de estética estão fortemente ligados à classe do grafo a ser desenhado. Por exemplo, a verticalidade da estrutura é geralmente imprescindível ao desenho de árvores e hierarquias.

Para a representação de um grafo no plano é necessária, ainda, a adoção de um padrão gráfico de desenho. Em geral, vértices são representados como símbolos geométricos, por exemplo, círculos ou retângulos, e ares-tas como curvas abertas entre os símbolos associados com os vértices que conectam. Os padrões mais utilizados pelos algoritmos existentes são:

- linha-reta, onde arestas são mapeadas em segmentos de linha-reta entre dois símbolos que representam seus vértices finais; e
- grade[5], em que arestas são mapeadas em cadeias de segmentos horizontais e verticais, ou seja, as arestas podem ter quebras, que são o encontro entre um segmento horizontal e um vertical.

O primeiro padrão é comumente utilizado em textos de teoria dos grafos, e o segundo usado com maior freqüência em desenhos de diagramas, que devem apresentar uma maior modularidade na estrutura. Há ainda o padrão de desenho onde as arestas são desenhadas curvilíneas, além de outros padrões que são, em geral, variações ou combinações dos padrões acima, e não serão relevantes neste trabalho.

A seguir apresentaremos alguns conceitos e algoritmos básicos que serão utlizados posteriormente.

1.4 Conceitos gerais

Os algoritmos a serem descritos daqui em diante devem seguir algumas notações e definições da teoria dos grafos. Quaisquer dúvidas quanto à nomenclatura utilizada, sugerimos a leitura complementar de **[BM77]**.

Um grafo $G=(V,E)$ consiste de um conjunto de vértices V e um conjunto de arestas E. Dados dois vértices v e $w \in V$, v é adjacente a (ou vizinho de) w, se há uma aresta $e=(v,w) \in E$. Dizemos, ainda, que e é incidente a v e w, e que v e w são os pontos finais de e.

Denotam-se por n e m, respectivamente, o número de vértices e arestas do grafo. O grau de um vértice pertencente a um grafo não-orientado é o número de suas arestas incidentes. No caso de um grafo orientado, um vértice v possui grau de entrada (saída) definido como sendo o número de arestas cujo ponto final (inicial) é v.

5. *grid*, em inglês.

Dizemos que um grafo G é imersível no plano, se ele pode ser desenhado no plano sem que suas arestas se cruzem, exceto em seus pontos finais. Assim, se G é imersível, então ele é chamado grafo planar **[BM77]**. O desenho de um grafo planar é chamado imersão planar, também denominado de grafo plano.

Um grafo plano particiona o plano em um certo número de regiões conexas, denominadas faces. Cada grafo plano tem exatamente uma face ilimitada, a face externa. Uma face é dita incidente aos vértices e arestas que estão na sua fronteira.

A partir de G, nós podemos definir um outro grafo G^* como segue: há um vértice f^* em G^* correspondente a cada face f de G, e há uma aresta $e^*=(f^*, g^*)$ em G^*, se e somente se suas faces correspondentes f e g são separadas por uma aresta e em G. O grafo G^* é chamado grafo dual de G.

A estrutura de dados utilizada para representar um grafo é um conjunto de listas de adjacências, uma para cada vértice pertencente ao grafo. Se $e=(v,w)$ é uma aresta do grafo G, então e pertence às listas de adjacências dos vértices v e w, denotadas respectivamente por $A(v)$ e $A(w)$, caso o grafo seja não-orientado. Caso o grafo seja orientado, e pertece apenas à $A(v)$.

Um grafo G é finito, se V e E são conjuntos finitos. Uma das técnicas mais utilizadas de travessia em um grafo finito é a busca em profundidade proposta por Hopcroft e Tarjan **[Tar72]**, cujo objetivo final é a visita de todos os vértices e arestas do grafo em uma determinada ordem. A busca em profundidade pode ser estendida para a obtenção dos mais diversos resultados relativos a problemas em grafos. Neste texto, veremos aplicações clássicas de tal técnica, além de outros algoritmos úteis na descrição dos algoritmos para o desenho de grafos.

1.4.1 Aplicações da busca em profundidade

Neste texto, a busca em profundidade é referenciada diversas vezes. Sua principal aplicação será a construção de um outro grafo chamado palmeira[6] **[Tar72]** a partir de um grafo original não-orientado. Palmeira é um grafo orientado cujo conjunto de arestas pode ser dividido em dois subconjuntos: um formado por arestas que definem a árvore de espalhamento T do grafo (arestas de árvore); e um outro formado por arestas que ligam um vértice recém-visitado pela busca em profundidade a um outro vértice já visitado anteriormente (arestas cíclicas).

Arestas (v,w) pertencentes à árvore de espalhamento T são denotadas por $v \rightarrow w$; $v \rightarrow^* w$ significa que há um caminho orientado entre v e w em T, e $w \rightarrow u$ denota

6. *palm tree*, em inglês.

uma aresta cíclica. A figura 1-4 ilustra um exemplo de um grafo e uma palmeira construída a partir do vértice 1.

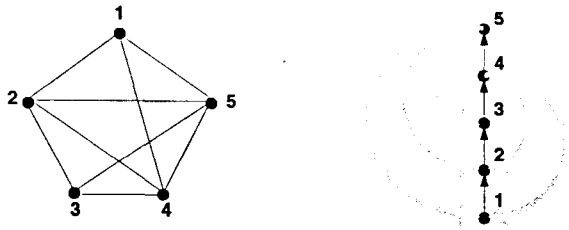


FIGURA 1-4 Um grafo e uma palmeira correspondente. Segmentos em preto (cinza) representam arestas de árvore (cíclicas).

Considerando-se que os vértices são numerados de acordo com a ordem de visita da busca em profundidade. Para cada vértice $v \in V$, pode-se calcular o primeiro e segundo vértices de menor numeração alcançáveis a partir de uma aresta cíclica de algum descendente de v. Esses vértices são denominados *LOWPT1(v)* e *LOWPT2(v)*. Seja $S_v = \{w \mid \exists\ u$ tal que $v \rightarrow^* u$ e $u \dashrightarrow w\}$ o conjunto de vértices alcançáveis por arestas cíclicas originadas em algum vértice descendente de v em T, então definimos para cada vértice v:

$$LOWPT1(v) = min(\{v\} \cup Sv)$$
$$LOWPT2(v) = min(\{v\} \cup (S_v - LOWPT1(v)))$$

Por convenção, esses valores são iguais a v, se ainda não foram calculados. Como os valores *LOWPT* de um certo vértice dependem apenas dos valores *LOWPT* de seus filhos e de suas próprias arestas cíclicas, é fácil calculá-los durante a busca em profundidade. A tabela abaixo mostra os valores LOWPT dos vértices pertencentes ao grafo da figura 1-4.b.

vértice	LOWPT1	LOWPT2
1	1	1
2	1	2
3	1	2
4	1	2
5	1	2

A seguir e na seção 3.1 apresentaremos aplicações da construção da palmeira de um grafo através da busca em profundidade, que serão referenciadas posteriormente.

Separação de componentes biconexas

Seja G um grafo geral não-orientado. Uma condição suficiente e necessária para que G seja biconexo é que, dentre quaisquer três vértices de G, haja um caminho entre dois deles ao qual o terceiro não pertença **[Tar72]**. Seja a um vértice de G que pertence a todo caminho entre dois outros vértices, então a é chamado "ponto de articulação[7]" de G. Assim, os subgrafos G_i que não contém algum ponto de articulação são denominados componentes biconexas de G.

7. *cut point*, em inglês.

Em **[Tar72]** é apresentado um algoritmo para separação de componentes biconexas de um grafo não-orientado, que é uma aplicação direta do método de busca em profundidade em grafos.

A idéia do algoritmo é separar as componentes biconexas do grafo durante a construção de uma palmeira de *G*, durante uma busca em profundidade. Uma vez que os valores *LOWPT*1 de cada vértice foram calculados é fácil descobrir se um vértice do grafo é ou não um ponto de articulação.

Sejam *a*, *v* e *w* vértices distintos de *G*, *(a,v)* uma aresta da árvore de espalhamento *T* de *G*, e suponha que não haja um caminho entre *v* e *w* em *T*. Se $LOWPT1(v) \geq a$, então *a* é um ponto de articulação e sua remoção desconecta *v* e *w*. Dessa forma, se *C* é uma componente biconexa de *G*, os vértices de *C* definem uma subárvore de *T*, e a raiz dessa subárvore é ou um ponto de articulação de *G*, ou a raiz de *T*.

A figura 1-5 ilustra um grafo *G*, uma palmeira correspondente com seus valores *LOWPT1* e suas componentes biconexas.

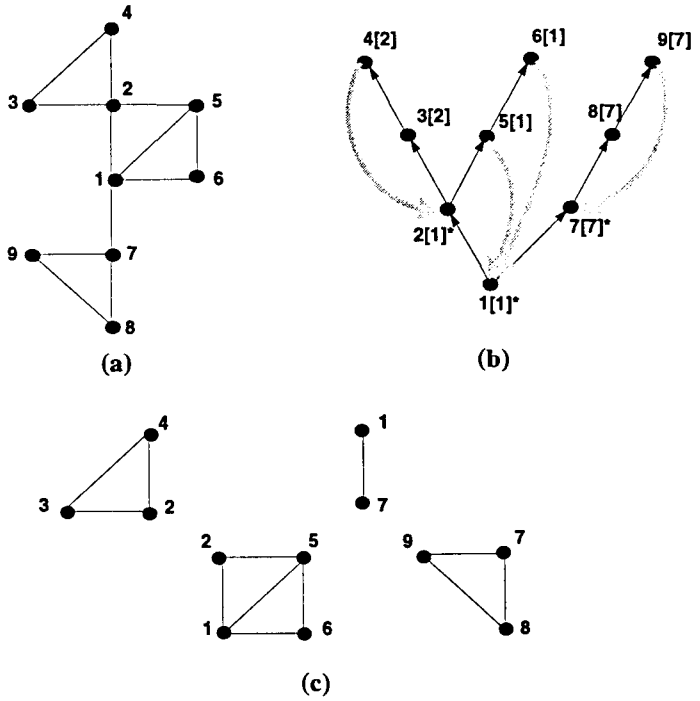


FIGURA 1-5

(a) Grafo G. (b) Uma palmeira com seus valores *LOWPT*1 entre [ ] e pontos de articulação marcados com *. (c) Componentes biconexas de *G*.

Durante o algoritmo, as arestas de *G* são empilhadas à medida que são atravessadas. Quando um ponto de articulação é encontrado, as arestas correspondentes à componente biconexa que se iniciou naquele ponto estão no topo da pilha, e são todas desempilhadas, formando a nova componente.

Esse algoritmo possui complexidade linear, em tempo e espaço, na soma do número de vértices e arestas, por ser uma aplicação direta do método da busca em profundidade, com algumas operações na pilha de arestas.

Váriös algoritmos descritos na literatura corrente, em desenho de grafos, consideram biconexo o grafo de entrada, por isso há a necessidade freqüente da separação das componentes biconexas como preprocessamento.

Nos próximos capítulos descreveremos brevemente alguns algoritmos projetados para o desenho de cada uma das classes de grafos citadas anteriormente. A intenção desses capítulos é fornecer uma idéia geral de como os algoritmos procedem durante a transformação da informação, geralmente abstrata, relativa a um grafo em uma possível representação gráfica para o mesmo. Não serão abordadas questões relativas a melhora de complexidade e desempenho dos algoritmos. Um resumo bibliográfico mais extenso sobre algoritmos para o desenho automático de grafos pode ser encontrado em **[TBB88]** e **[ET89]**.

O capítulo 2 aborda o problema do desenho de árvores, o capítulo 3 é relativo a grafos planares, o capítulo 4 trata de grafos orientados acíclicos, o capítulo 5 de grafos gerais não-orientados, e no capítulo 6 há a descrição de um método geral para o desenho de diagramas. O capítulo 7 traz aspectos da implementação realizada neste trabalho, junto com os resultados obtidos. Finalmente, nossas conclusões e extensões propostas encontram-se no capítulo 8.

Capítulo 2

Árvores

2.1 Introdução

Árvores são comumente representadas por desenhos hierárquicos planares, que adotam o padrão de linha-reta. Os algoritmos clássicos projetados para o desenho de árvores limitam-se à classe de árvores binárias, mas com propostas de extensões para árvores *n*-árias.

A tarefa básica desses algoritmos é atribuir coordenadas a cada um dos nós da árvore, restando apenas o trabalho do desenho final da árvore em um dispositivo de saída, ou seja, mapear os nós naquelas coordenadas e mapear arestas como segmentos de reta entre as coordenadas de seus nós inicial e final.

Critérios de estética freqüentemente adotados por tais algoritmos são os seguintes:

1. vértices devem ser localizados ao longo de linhas horizontais paralelas de acordo com seu nível relativo à raiz;
2. deve haver uma distância mínima de separação entre dois vértices consecutivos em um mesmo nível;
3. a largura do desenho deve ser tão pequena quanto possível;
4. nós-pais devem ser centrados horizontalmente sobre nós-filhos;
5. subárvores isomórficas[1] devem ter desenhos iguais; e
6. subárvores simétricas devem ser desenhadas simetricamente.

Os algoritmos propostos possuem idéias muito parecidas. O cálculo da coordenada *y* de cada nó, por exemplo, sempre é feito com base no nível do nó na árvore, que é

1. Duas subárvores são isomórficas se ambas são vazias, ou se ambas são não-vazias e suas subárvores esquerda e direita são respectivamente isomórficas.

geralmente obtido por uma busca em profundidade na árvore. O cálculo da coordenada x, em geral, depende das coordenadas dos filhos ou vizinhos daquele nó.

A seguir, descreveremos brevemente três algoritmos clássicos para o desenho de árvores. A complexidade de cada um deles é linear no número de nós da árvore.

2.2 Algoritmo de Wetherell e Shannon

Em [WS79], Wetherell e Shannon apresentam dois algoritmos para o desenho de árvores binárias, onde o segundo representa uma leve modificação com relação ao primeiro. Os dois algoritmos consideram os dois primeiros critérios enumerados anteriormente, mas cada um opta entre os critérios 3 e 4 separadamente. A justificativa óbvia é que muito espaço pode ser desperdiçado uma vez que todo nó-pai deva ser centralizado entre seus nós-filhos. Na figura 2-1 podemos observar desenhos de uma mesma árvore, sendo que no primeiro o quarto critério foi respeitado, enquanto que na segunda, esse critério foi desprezado e uma largura menor para o desenho foi obtida.

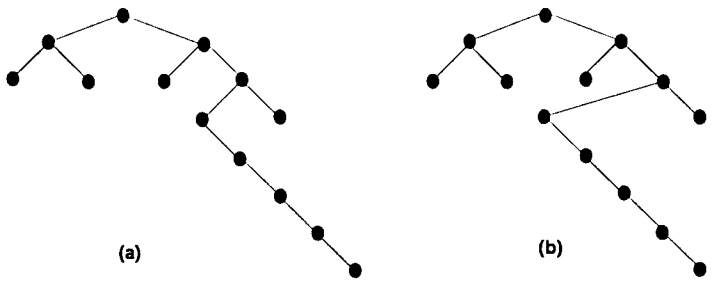


FIGURA 2-1 Desenhos de uma mesma árvore, a partir de critérios diferentes.

Os algoritmos são não-recursivos e o procedimento inicial é igual para ambos. Uma travessia inicial em pós-ordem é realizada na árvore. Durante essa primeira travessia, são guardadas as seguintes informações:

- o nível do nó na árvore;
- um valor temporário para a coordenada x do nó com base na próxima posição disponível naquele nível ou, se o nó tiver algum filho, com base na coordenada x atribuída ao(s) seu(s) filho(s);
- um possível deslocamento para os descendentes de cada nó; e
- a próxima posição disponível em cada nível.

Para cada nível há uma mesma posição disponível inicialmente, que é incrementada de dois a cada novo nó visitado naquele nível. Se a coordenada x encontrada para o nó nesse passo for menor que a próxima posição disponível em seu nível, é atribuída ao nó esta última posição, e é mantida a diferença entre elas para que todos os descendentes desse nó sofram o mesmo deslocamento no passo seguinte.

Durante uma segunda travessia, a posição final de cada nó é calculada com base no seu nível na árvore, no valor temporário da coordenada x e nos deslocamentos acumulados de seus ancestrais e vizinhos esquerdos.

A diferença entre os dois algoritmos aparece exatamente no cálculo final da coordenada x do nó. Na tentativa de centralizar um nó-pai sobre seus filhos, a primeira versão do algoritmo aplica, durante a segunda travessia em pré-ordem, todos os deslocamentos acumulados dos níveis anteriores. A outra versão tenta fugir dessa regra com o intuito de aproveitar posições à esquerda ainda não utilizadas.

No segundo caso, é feita uma travessia em ordem, e a posição final do nó é escolhida como o mínimo entre a mesma posição calculada pelo primeiro algoritmo, a próxima posição disponível no nível ao qual o nó pertence, a posição do filho esquerdo mais uma unidade, e a posição do nó-pai mais uma unidade, se o nó for um filho direito.

O algoritmo foi projetado para o desenho de árvores binárias, mas pode ser modificado para árvores n-árias aumentando-se o número de casos a serem tratados de acordo com o número máximo de subárvores de um mesmo nó.

2.3 Algoritmo de Vaucher

A motivação principal do algoritmo apresentado por Vaucher em **[Vau80]** foi a dificuldade de impressão de estruturas de árvores em formulários de impressão, devido a três fatores:

- posicionamento de nós da árvore em uma página impressa;
- impressão seqüencial (da esquerda para direita e de cima para baixo); e
- estouro na largura da página impressa.

Os quatro primeiros critérios para o desenho de árvores antes enumerados também são considerados por este algoritmo. As fases do algoritmo são o cálculo da posição horizontal de cada nó da árvore e a impressão da árvore em faixas verticais de acordo com a largura da página.

Durante o cálculo da posição dos nós, uma estrutura de lista auxiliar é criada, correspondendo à estrutura gráfica de saída. Para cada nó da árvore existe um elo com um ponteiro para ele mesmo e sua coordenada x já calculada. Os elos dos nós pertencentes a um mesmo nível na árvore são encadeados e encabeçados por outro ponteiro, que indica o primeiro nó daquele nível a ser impresso. Os ponteiros-cabeça, por sua vez, também são encadeados significando para o algoritmo a lista de linhas correspondentes aos níveis a serem impressos. Um exemplo dessa estrutura aparece na figura 2-2 .

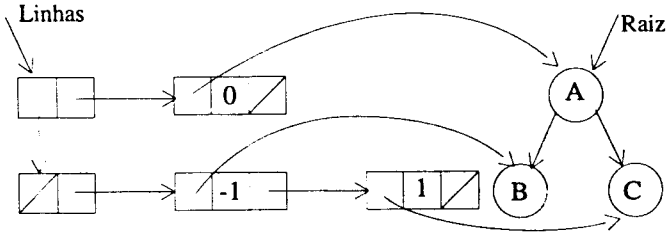


FIGURA 2-2 Estrutura auxiliar para uma árvore de 3 nós.

O procedimento para o cálculo das coordenadas x dos nós da árvore é o seguinte. Durante uma travessia na árvore em pós-ordem, com a visita da subárvore direita antes da subárvore esquerda, a posição do nó é determinada ao mesmo tempo que a estrutura da lista auxiliar é construída. Inicialmente, a posição de um nó é determinada pela posição de seu pai na árvore, podendo ser alterada para menos caso haja um nó vizinho do seu lado direito no mesmo nível.

A posição horizontal final de cada vértice é calculada de acordo com as coordenadas de seus nós-filhos conforme o algoritmo Posição descrito a seguir, ao mesmo tempo que é criada a estrutura auxiliar correspondente à estrutura gráfica de saída.

```
algoritmo Posição (No, Pos) {
  se ( No = NULL ) então
     retornar Pos;
  senão {
     adicionar No na cabeça da lista auxiliar relativa a seu nível, verificando
          se No deve ser deslocado para esquerda;
     se ( filho_direito = NULL ) então
        retornar Posição(filho_esquerdo, Pos-1) + 1;
     senão
        se (filho_esquerdo = NULL) então
          retornar Posição(filho_direito, Pos+1) - 1;
        senão
          retornar [ Posição(filho_direito, Pos+1) +
                     Posição(filho_esquerdo, Pos-1) ] / 2;
  }
}
```

Os parâmetros da função **Posição** são um ponteiro para a raiz da subárvore cuja posição se quer calcular e uma dica de posição do seu nó-pai na árvore.

Depois desses cálculos serem efetuados e da lista auxiliar ser construída, as informações dos nós podem ser impressas linha a linha, bastando para isso, que se percorra a lista de elos encadeados. Uma vez que, em um nível, haja um estouro na largura da página, adia-se a impressão do restante daquele nível da árvore para a próxima página.

Como extensão desse algoritmo para árvores n-árias, a posição de um nó poderia ser calculada como sendo a metade do valor da soma das coordenadas de seu filho

mais à esquerda e seu filho mais à direita **[Vau80]**. A figura 2-3 mostra um exemplo de execução desse algoritmo.

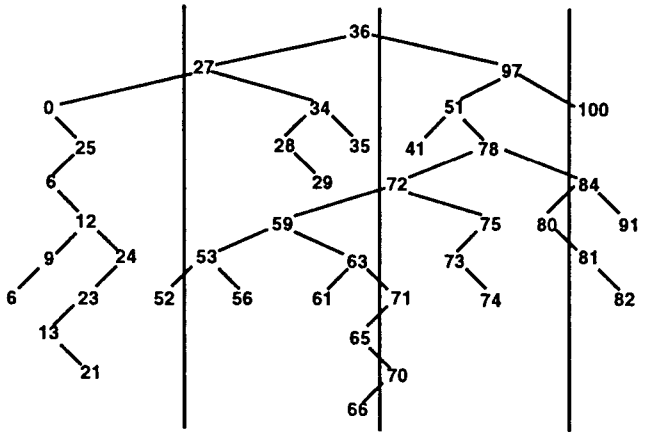


FIGURA 2-3 Um exemplo de execução do algoritmo de Vaucher.

A seguir, descreveremos um outro algoritmo para o desenho de árvores que se propõe a desenhos mais estéticos que os de Wetherell e Shannon, por considerar um critério de estética a mais que todos os outros algoritmos até então existentes.

2.4 Algoritmo de Reingold e Tilford

Reingold e Tilford apresentaram em **[Rei81]** um algoritmo que se preocupa com o desenho idêntico de subárvores isomórficas e desenhos refletidos de árvores simétricas, além de considerar todos os outros critérios anteriores para o desenho de árvores. A heurística utilizada compromete um pouco a largura final do desenho, mas essa não é para o algoritmo a questão mais importante.

O algoritmo funciona como se, para cada nó da árvore, suas duas subárvores fossem desenhadas, cortadas ao longo de seus contornos e sobrepostas a partir de suas raízes. A partir daí, movimentos de afastamento são realizados até que nenhum ponto das subárvores se toquem. Assim, as duas subárvores são construídas independentemente e depois colocadas tão perto quanto possível.

Como os outros algoritmos, é feita inicialmente uma travessia em pós-ordem na árvore, aplicando-se a heurística acima, e em seguida, uma travessia em pré-ordem para que as posições relativas calculadas no primeiro passo sejam transformadas em coordenadas absolutas.

O trabalho extra realizado durante a primeira travessia consiste em descobrir o quão distante duas subárvores irmãs devem ficar no desenho. Para isso é necessário seguir o contorno de ambas subárvores: o contorno direito da subárvore esquerda e o contorno esquerdo da subárvore direita, computando a distância entre elas e afastando-as quando necessário.

Um problema surge quando as duas subárvores de um mesmo nó têm alturas diferentes e são ambas não-vazias, pois os nós que compõem os contornos das subár-

vores são visitados concomitantemente. A solução proposta para tal problema veio da idéia de árvores alinhavadas[1], através da utilização de um elo entre nós mais baixos das subárvores. Por exemplo, se a subárvore esquerda é mais alta que a subárvore direita, um elo deve ser criado entre o nó mais à direita do nível mais baixo da subárvore direita e o nó mais à direita no próximo nível a ser visitado durante a primeira travessia da subárvore esquerda. A figura 2-4 mostra exemplo de tal situação.

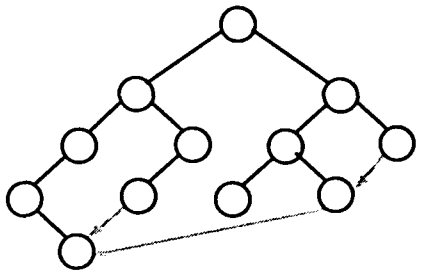


FIGURA 2-4 Elos entre subárvores de alturas diferentes.

Uma vez que esse elo é uma reaproveitação dos ponteiros esquerdo e direito de um dado nó v (que será sempre uma folha), os nós "alinhavados" ficam transparentes se as subárvores devem ser afastadas, por serem tratados como filhos de v e suas posições contribuírem para o cálculo do posicionamento de v e seus ancestrais.

A distância relativa entre um nó e seus filhos é armazenada no próprio nó e passada aos filhos durante a segunda travessia em pré-ordem, uma vez que um nó pai deve ser centralizado entre suas duas subárvores.

1. *threaded*, em inglês.

Finalmente, com a pré-ordem os elos são apagados e as posições absolutas calculadas. O desenho final de uma árvore binária pelo algoritmo descrito nesta seção aparece na figura 2-5.

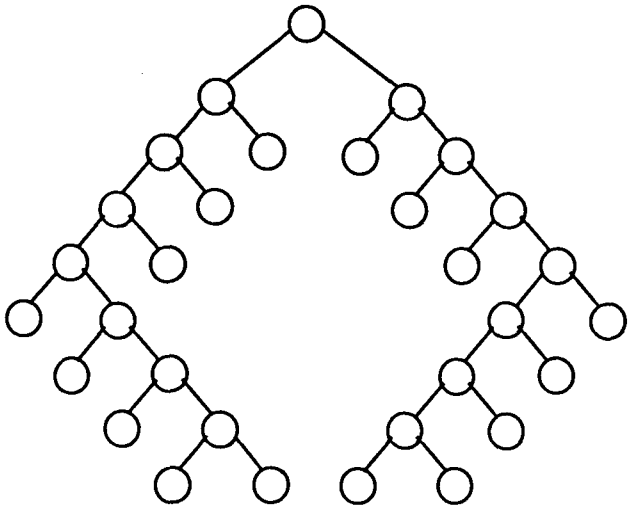


FIGURA 2-5 Árvore desenhada pelo algoritmo de Reingold e Tilford.

No final do trabalho de Reingold e Tilford também é proposta uma generalização para o desenho de árvores *n*-árias e florestas, que não compromete a complexidade linear do algoritmo. Caso exista um maior interesse no estudo da complexidade desse algoritmo e outros, e na generalização proposta, sugerimos a leitura do trabalho de Reingold e Tilford **[Rei81]**, além de **[SR83]** .

Capítulo 3

Grafos Planares

3.1 Introdução

O desenho de um grafo planar geralmente envolve três passos **[JEM+91]**: o teste de planaridade do grafo, a criação de uma representação planar[1], ou uma imersão topológica planar para o mesmo e, finalmente, a atribuição de coordenadas para seus vértices.

Na maioria das vezes, os algoritmos que tratam desse problema assumem que os dois primeiros passos foram realizados, partindo para o desenho do grafo propriamente dito. Testes de planaridade sempre são sugeridos, e a representação planar é considerada resultado de tais testes. O clássico teste de planaridade de Hopcroft e Tarjan **[HT74]**, por exemplo, é constantemente mencionado, e é descrito brevemente ainda nesta seção.

Os critérios de estética comumente considerados para o desenho de grafos planares são os seguintes:

1. minimização do número de cruzamentos;
2. apresentação de simetrias; e
3. distribuição regular dos ângulos que compõem cada face.

A partir dos trabalhos publicados sobre o assunto, podemos dividir os resultados em três áreas particulares de nosso interesse:

- desenhos de linha-reta;
- representações de visibilidade; e
- desenhos ortogonais em grade.

1. Representação planar é uma estrutura de dados que representa as adjacências entre as faces de um desenho planar.

A seguir, apresentaremos os algoritmos de cálculo de st-numeração **[ET76]** e de teste de planaridade **[HT74]**, utilizados na descrição de alguns algoritmos para o desenho de grafos planares. Finalmente, apresentaremos alguns resultados relacionados aos três itens antes citados.

Cálculo de st-numeração

Uma st-numeração para um grafo $G=(V,E)$, onde s e t são vértices distintos de G, chamados respectivamente fonte e sorvedouro, é um mapeamento um-a-um $\xi: V \rightarrow \{1, 2, \ldots, n\}$, tal que $\xi(s)=1$, $\xi(t)=n$, e cada vértice $v \neq \{s, t\}$ tem dois vértices adjacentes u, w para os quais $\xi(u) < \xi(v) < \xi(w)$.

Lempel, Even e Cederbaum provaram que para um grafo G biconexo com n vértices, dada uma aresta (s,t) do grafo, é possível encontrar uma st-numeração para G. O algoritmo apresentado por Even e Tarjan em **[ET76]** constrói uma st-numeração para um grafo G biconexo em tempo linear. O algoritmo consiste de três passos.

No primeiro passo é construída uma palmeira através de busca em profundidade no grafo, de acordo com **[Tar72]**. A raiz da árvore de espalhamento T do grafo deve ser o vértice t, e a primeira aresta da árvore deve ser (t,s). Assim, o vértice t receberá o número 1, o vértice s o número 2, além de serem calculados para cada vértice v seu valor $LOWPT1(v)$.

O segundo passo do algoritmo é encontrar caminhos entre vértices do grafo, que serão utilizados no passo final. Para isso, os vértices t e s, e aresta (t,s) são marcados como "velhos" e todos os outros vértices e arestas são marcados como "novos", e uma nova busca em profundidade é realizada. O resultado é um conjunto de caminhos simples, compostos de arestas "novas", mas com início e fim, respectivamente, em vértices v e w já visitados anteriormente (marcados como "velhos"). Depois disso, os vértices e arestas pertencentes àquele caminho são marcados com "velhos" e o caminho é retornado. Esse caminho pode ser orientado a partir de v ou em direção a v.

Durante a busca em profundidade, um caminho é gerado, a partir de um vértice v, da seguinte maneira:

```
algoritmo Caminho(v) {
  se há uma aresta e=(v,w) cíclica "nova" e  w →* v então {
     marcar e como "velha";
     retornar o caminho {v,w};
  }
  senão
     se há uma aresta e=(v,w) da árvore T "nova" então {
        marcar e como "velha";
        iniciar o caminho igual a {v,w};
        enquanto ( w é "novo" ) fazer {
           encontrar uma aresta "nova" e=(w,x) tal que x=LOWPT1(w)
              ou LOWPT1(x)=LOWPT1(w);
           marcar w e e como "velhos" ;
           adicionar e ao caminho;
           w=x;
```

```
        }
        retornar caminho;
      }
      senão
        se há uma aresta e=(v,w) cíclica "nova" e v →* w então {
          marcar e como "velha";
          iniciar o caminho igual a {v,w};
          enquanto ( w é "novo" ) fazer {
            encontrar uma aresta "nova" e=(w,x) tal que x → w;
            marcar w e e como "velhos";
            adicionar e ao caminho;
            w=x;
          }
          retornar caminho;
        }
        senão          /* todas as arestas incidentes a v são "velhas" */
          retornar o caminho vazio;
}
```

O último passo para o cálculo da st-numeração utiliza a construção de caminhos acima. É mantida uma pilha de vértices "velhos" durante a st-numeração, tal que todos os caminhos a partir do vértice do topo devem ser explorados antes que ele receba sua numeração, o que ocorre no último caso do algoritmo.

Inicialmente, a pilha contém o vértice s no topo da pilha, em cima do vértice t. O algoritmo de geração de caminhos é invocado a partir de s até que o caminho vazio seja retornado, quando s recebe o número 1 e é retirado da pilha. Assim, o algoritmo segue para todos os outros vértices do grafo, que ganham números consecutivos conforme vão sendo retirados da pilha, até que o vértice t receba o n-ésimo número. O algoritmo que utiliza o procedimento **Caminho** é descrito a seguir.

```
algoritmo St_Numer {
  marcar s, t e e=(s,t) como "velhos" e todos os outros vértices e arestas como
"novos";
  empilhar t e s;
  i = 0;
  enquanto ( pilha não vazia ) fazer {
    v = pop(pilha);
    seja (v1, v2, ..., vk-1, vk) o caminho retornado por Caminho(v);
    se ( caminho não vazio ) então
      empilhe vk-1, ..., v2, v1;
    senão
      st-número(v) = i = i+1;
  }
}
```

Para a execução do algoritmo STNUMER é preciso manter algumas informações auxiliares para cada vértice v:

- uma lista de arestas cíclicas (v,w) tal que $v \rightarrow^* w$;
- uma lista de arestas cíclicas (v,w) tal que $w \rightarrow^* v$;

- uma lista de arestas de árvore $v \rightarrow w$;
- a aresta de árvore $u \rightarrow v$; e
- uma aresta (v,w) tal que $w=LOWPT1(v)$ ou $LOWPT1(w)=LOWPT1(v)$;

Todos esses itens podem ser construídos durante o primeiro passo do algoritmo, que executa em tempo linear na soma do número de vértices e arestas, por ser uma busca em profundidade estendida. A complexidade do algoritmo STNUMER é dominada pela geração de caminhos a partir de cada vértice do grafo, o que gasta tempo proporcional a um mais o número de arestas pertencentes ao caminho. Uma vez que cada aresta aparece exatamente em um caminho, o algoritmo para o cálculo da st-numeração possui tempo total linear. Um exemplo de execução do cálculo de st-numeração aparece na figura 3-1.

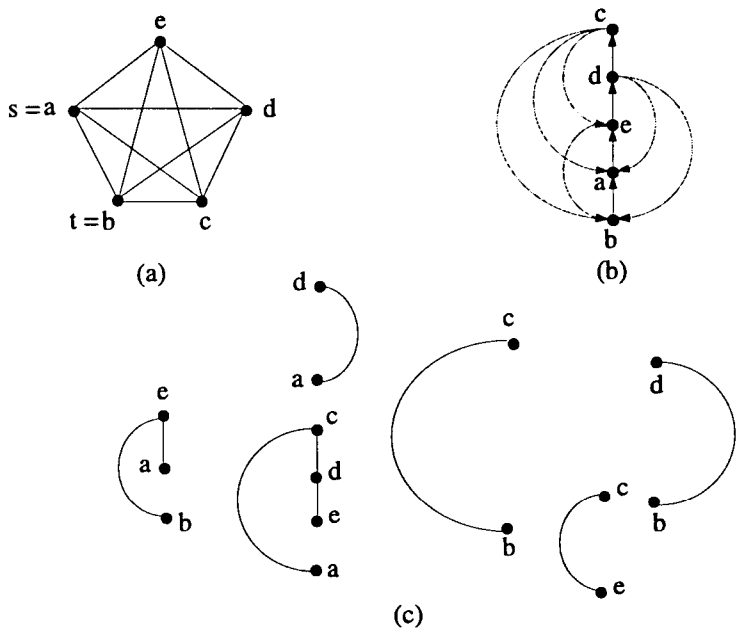


FIGURA 3-1 Exemplo de execução do algoritmo de cálculo de st-numeração. (a) Grafo de entrada. (b) Palmeira encontrada no primeiro passo. (c) Caminhos gerados no segundo passo.

Os st-números para cada vértice do grafo de entrada da figura 3-1 são:.

vértice	st-número
a	1
b	5
c	2
d	3
e	4

O próximo item descreve uma outra aplicação da busca em profundidade em grafos, o famoso teste de planaridade de Hopcroft e Tarjan.

Teste de planaridade

A imersão de um grafo no plano tem grande importância nos procedimentos de desenho automático, seja para o projeto de circuitos integrados ou para o desenho de diagramas de sistemas de informação. No último caso, um ponto crucial é a complexidade em tempo do algoritmo de imersão planar, devido à necessidade de respostas rápidas durante o processo interativo com o usuário.

A idéia do algoritmo apresentado por Hopcroft e Tarjan em **[HT74]** é tentar construir uma imersão planar do grafo incrementalmente, pedaço por pedaço, observando o posicionamento relativo entre eles.

O algoritmo utiliza basicamente busca em profundidade como modo de exploração do grafo. Cada grafo é considerado biconexo, uma vez que dado um grafo não-biconexo, suas componentes biconexas podem ser encontradas **[Tar72]** (vide seção 1.4) e o algoritmo pode ser aplicado a cada uma delas.

O primeiro passo do algoritmo é a transformação do grafo não-orientado em uma palmeira P (vide seção 1.4). Depois é realizada uma ordenação das listas de adjacências da palmeira P de acordo com os valores $LOWPT$ de cada vértice. Seja Θ uma função definida sobre as arestas (v,w) de P como segue:

$$\Theta(v,w) = \begin{cases} 2 \cdot w & \text{se } v \dashrightarrow w; \\ 2 \cdot LOWPT1(w) & \text{se } v \rightarrow w \text{ e } LOWPT2(w) \geq v; \text{ e} \\ 2 \cdot LOWPT1(w) + 1 & \text{se } v \rightarrow w \text{ e } LOWPT2(w) < v. \end{cases}$$

Depois da aplicação da função Θ a todas as arestas de P, as listas de adjacências são ordenadas em ordem crescente do valor de Θ, utilizando-se *radix sort*. Tal ordenação é utilizada na geração de caminhos no grafo posteriormente.

Caminhos entre vértices são gerados por uma segunda busca em profundidade, tal que cada um deles consiste de uma seqüência de zero ou mais arestas de árvore seguidas por uma aresta cíclica. Esses caminhos não compartilham arestas. E, uma vez que o conjunto de caminhos foi gerado, a planaridade do grafo pode ser testada pela tentativa de imersão desses caminhos no plano.

O primeiro caminho considerado é um ciclo que consiste das arestas de árvore $1 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_t$, seguidas por uma aresta cíclica $v_t \dashrightarrow 1$, onde a numeração dos vértices é tal que $1 < v_1 < v_2 < \ldots < v_t$. Encontrado o ciclo, este é retirado do grafo, restando alguns pedaços desconexos do mesmo, aqui denominados segmentos. Um segmento consiste de uma única aresta cíclica $u \dashrightarrow v$, ou de uma aresta de árvore $v \rightarrow w$ mais uma subárvore de raiz w e todas as arestas cíclicas que saem daquela subárvore. Para cada segmento é feita uma tentativa de imersão junto àquele ciclo antes retirado. A operação é aplicada recursivamente a todos os outros segmentos, até que se conclua a possibilidade ou não da imersão planar completa do grafo.

A ordem de geração dos caminhos é tal que todos os caminhos de um segmento são gerados antes dos caminhos de qualquer outro segmento, e cada segmento é explorado em ordem decrescente de v. O ciclo encontrado em cada chamada recursiva consiste de um caminho de arestas que ainda não foram visitadas, mais um caminho de arestas de árvore pertencentes a ciclos anteriores.

A tentativa de inclusão de um segmento na parte do grafo já imersa é realizada da seguinte forma. A partir de um segmento S, é encontrado um caminho p, conforme a definição anterior de caminho em um segmento. Um lado da imersão planar, por exemplo o esquerdo, é escolhido para a tentativa de imersão do caminho p. Assim, p é comparado com outros caminhos já imersos à esquerda, através da observação da existência de uma ou mais arestas cíclicas naquele lado que possam impedir sua imersão. Se isso ocorrer, é feita uma tentativa de movimentação para o lado direiro da imersão daqueles segmentos que atrapalharam a imersão de p, e se depois desse e de mais algum outro deslocamento de segmento necessário, p conseguir ser imerso, o processo é repetido recursivamente para todos os outros caminhos de S, senão o grafo é declarado não-planar.

No caso em que o algoritmo de Hopcroft e Tarjan seja utilizado por algum algoritmo de desenho de grafos, não se espera que o algoritmo termine sua execução declarando a não-planaridade do grafo. Em geral, o esperado é que a representação planar de um subgrafo planar maximal seja fornecida pelo algoritmo, além do conjunto de arestas que causem a não-planaridade daquele grafo. Desse modo, uma modificação no algoritmo deveria ser feita com esse intuito.

Quanto às estruturas de dados utilizadas pelo algoritmo, são mantidas para cada lado da imersão (esquerdo e direito), duas pilhas de vértices da palmeira que tenham arestas cíclicas entrando por aquele lado durante o processo da imersão, denominadas respectivamente por L e R.

Quando um segmento que começa no vértice v_i vai ser imerso, deve-se saber quais vértices com caminho na árvore T de 1 a v_i têm arestas cíclicas entrando pela esquerda ou direita na imersão. A pilha $L(R)$ conterá, em ordem, os vértices v_k tal que há um caminho na árvore de 1 a v_k e de v_k a v_i, $1 < v_k < v_i$, e existe alguma aresta cíclica já imersa que entra em v_k pela esquerda(direita). As pilhas L e R são atualizadas de quatro possíveis maneiras:

- Depois que todos os segmentos que começam em v_{i+1} foram explorados e imersos, todas as ocorrências de v_i em L e R devem ser apagadas, uma vez que segmentos ainda não explorados começarão em vértices menores ou iguais a v_i.
- Se p é um caminho pertencente a um segmento S, cujo vértice final é f, f deve ser inserido na pilha (L ou R) quando p for imerso em um dos lados.
- A aplicação recursiva do algoritmo deve adicionar entradas para outros caminhos do segmento S.
- Entradas devem ser deslocadas de uma pilha para outra, se seus segmentos correspondentes forem movidos de um lado para outro na imersão planar.

Um esboço geral do algoritmo de Hopcroft e Tarjan para imersão planar de um grafo biconexo é apresentado a seguir.

```
algoritmo IMERSÃO PLANAR {
  L = R = Pilha vazia;
  encontrar o primeiro ciclo c e retirá-lo do grafo;
  enquanto ( há segmento inexplorado ) fazer {
    iniciar busca por um caminho no próximo segmento S;
    quando todos os caminhos a partir do vértice v forem explorados, apagar
      entradas em L e R que contenham vértices com numeração maior ou igual a v;
```

```
    Seja p o primeiro caminho de S, com vértices inicial e final iguais a s e f,
      respectivamente;
    enquanto ( p não conseguir ser imerso à esquerda ) fazer {
      se ( há arestas em L que atrapalham a imersão de p ) então
        trocar blocos de arestas de L para R e de R para L;
      se ( ainda existem arestas à esquerda em conflito com p) então {
          o grafo não é planar;
          fim;
      }
    }
    empilhar o vértice f de p em L;
    aplicar o algoritmo recursivamente para a imersão dos outros caminhos de S;
  }
}
```

Para que se mantenha o controle de quais arestas influenciam-se mutuamente na localização (esquerda ou direita) durante a imersão, há uma nova estrutura, denominada bloco. Um bloco é um conjunto de entradas pertencentes a L e R correspondentes a arestas cíclicas, tal que o posicionamento de qualquer uma das arestas pertencentes àquele bloco determina o posicionamento de todas as outras.

O algoritmo de Hopcroft e Tarjan é bastante conhecido por ter complexidade em tempo linear na soma do número de vértices e arestas, e pela simplicidade das estruturas de dados utilizadas. Ele já serviu de base para muitos outros de propósitos mais gerais, tendo inclusive algumas modificações e melhoras **[Rub75]**.

Nas seções a seguir, veremos um pouco dos trabalhos já desenvolvidos em cada uma das subáreas do desenho de grafos planares mencionadas no início deste capítulo.

3.2 Desenhos de linha-reta

Todo grafo planar admite um desenho planar de linha-reta, ou ainda, um desenho convexo, onde cada face do grafo pode ser desenhada como um polígono convexo **[CON85]**.

Peter Eades et al **[JEM+91]** implementaram e fizeram um estudo comparativo de quatro algoritmos publicados entre 1963 e 1990 para o desenho automático de grafos planares utilizando o padrão de linha-reta. A maior parte das observações contidas nesta seção foram baseadas nesse trabalho.

Os principais critérios de estética definidos por tais algoritmos são:

1. o desenho de faces como polígonos;
2. a minimização do tamanho total das arestas;
3. a minimização da área do desenho; e
4. apresentação de simetrias.

Para testar os quatro algoritmos, Eades et al geraram grafos planares compostos de triangulações próprias, ou seja, toda face do grafo, com exceção da face externa, é um triângulo. Os resultados obtidos não podem ser estendidos diretamente para grafos planares gerais, onde faces são compostas por um número variável de arestas.

Dois dos algoritmos foram realmente projetados para o desenho de triangulações próprias. Estes são os algoritmos de Read (1986) e o de Chrobak e Payne (1990). Os dois outros algoritmos foram propostos por Tutte (1963) e Chiba et al (1985).

Os resultados do algoritmo de Tutte são os que apresentam a estrutura do grafo de modo menos legível, talvez por representarem as primeiras investigações do desenho automático de grafos planares. Além disso, sua complexidade é a maior dentre as de todos os outros algoritmos, podendo alcançar $O(n^3)$, dependendo da implementação de algumas operações em matrizes esparsas. A complexidade dos outros três algoritmos é linear no número de vértices.

Os algoritmos projetados por Tutte e Read baseiam-se na mesma idéia de posicionar vértices em função da posição de vértices vizinhos, e são os que apresentam os piores resultados. Aparentemente, isso ocorre por que os algoritmos determinam o tamanho de cada face com relação ao número de vértices que ela contém. Algumas faces grandes são divididas recursivamente em faces cada vez menores, enquanto que em outras faces não ocorre tal divisão. O resultado é o desenho totalmente desigual das faces do grafo.

De acordo com os exemplos e as estatísticas apresentadas em **[JEM+91]**, podemos observar que, mesmo para grafos pequenos (de 20 a 30 vértices), os resultados dos algoritmos de Tutte e Read são inaceitáveis, considerando os critérios de legibilidade anteriormente definidos. Os resultados tornam-se cada vez piores para um número maior de vértices. Os vértices são distribuídos desigualmente no desenho, causando alta concentração de vértices em algumas regiões enquanto que, em outras regiões, ocorre a situação oposta. Densidade maior de vértices ocorre, por exemplo, em regiões mais distantes da fronteira do desenho.

O algoritmo proposto por Chiba et al em **[CON85]** é o que apresenta o formato menos estético das faces. As faces são desenhadas finas e compridas, devido ao posicionamento de vértices adjacentes a determinados vértices do grafo com o intuito de formar polígonos convexos. Assim, as arestas são agrupadas de tal maneira que dificultam sua distinção no desenho final, e esse problema aumenta com o número de vértices, uma vez que arestas adjacentes formam uma espécie de leque em algumas partes do desenho. Esse algoritmo é o que resulta em desenhos com uma maior diferença angular das faces. Os próprios exemplos apresentados em

[CON85], onde as faces não são triangulações próprias, podem ser considerados esteticamente desagradáveis aos olhos (vide figura 3-2).

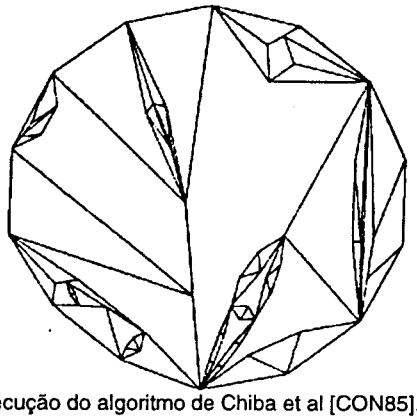


FIGURA 3-2 Exemplo de execução do algoritmo de Chiba et al [CON85].

O algoritmo desenvolvido por Chrobak e Payne é o que possui resultados mais satisfatórios dentre os quatro. O agrupamento de vértices em determinadas áreas do desenho não são tão freqüentes, e as faces são desenhadas mais regularmente e não tão finas quanto nos desenhos de Chiba et al. Além disso, é o mais rápido em execução dentre todos.

As figuras 3-3, 3-4, 3-5 e 3-6 mostram um mesmo grafo planar desenhado por cada um dos algoritmos implementados em **[JEM+91]**.

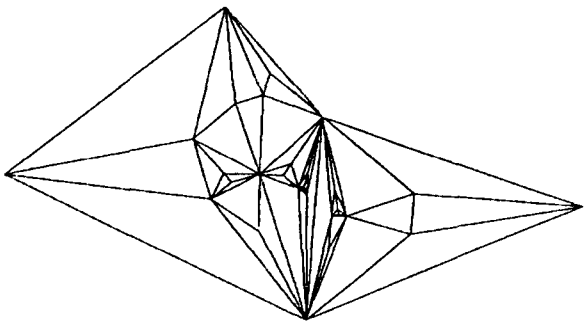


FIGURA 3-3 Resultado do algoritmo de Tutte.

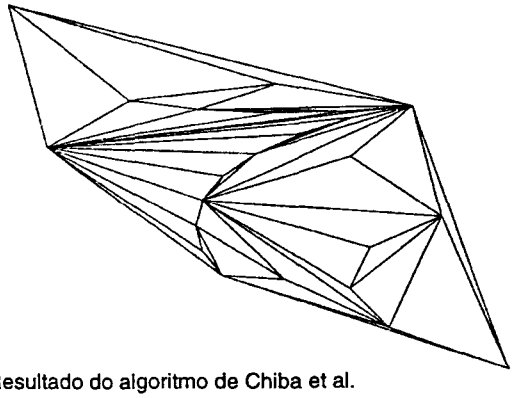


FIGURA 3-4 Resultado do algoritmo de Chiba et al.

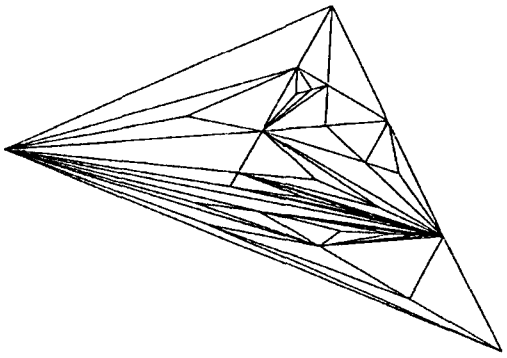


FIGURA 3-5 Resultado do algoritmo de Read.

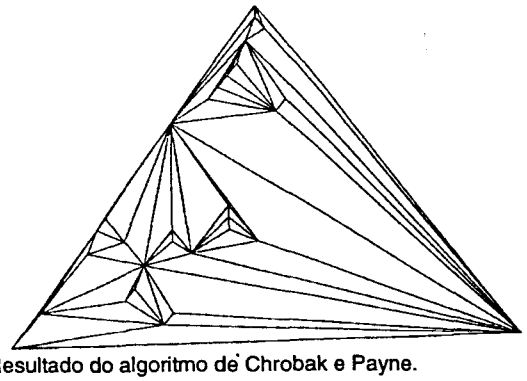


FIGURA 3-6 Resultado do algoritmo de Chrobak e Payne.

Mesmo que intuitivamente o algoritmo proposto por Chrobak e Payne seja o melhor dentre os quatro algoritmos que se propõem ao desenho de grafos planares, Eades et al concluíram em seu trabalho que todos deixam a desejar com relação a um ou outro aspecto de legibilidade, e que ainda não há nenhum algoritmo satisfatório na literatura para tal propósito.

3.3 Desenhos de representação de visibilidade

Representação de visibilidade **[TT86]** é uma representação no plano para grafos, onde vértices e arestas são mapeados em segmentos que não se sobrepõem. Segmentos horizontais correspondem aos vértices do grafo, e são chamados segmentos-vértice. Segmentos verticais correspondem às arestas do grafo, sendo denominados segmentos-aresta. Segmentos-aresta interceptam dois segmentos-vértice somente se estes lhe são adjacentes.

Representações de visibilidade são úteis, por exemplo, durante a compactação de circuitos integrados, onde dois segmentos paralelos em um circuito podem ser ditos visíveis se podem ser ligados por um segmento ortogonal entre eles, que não intercepte qualquer outro segmento. Na seção 3.4 é apresentado um algoritmo para desenhos ortogonais de grafos planares que também faz uso da representação de visibilidade de um grafo.

Todo grafo que admite essa representação deve ser planar **[TT86]**, como o exemplo da figura 3-7.

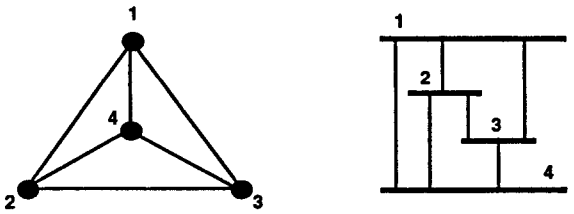


FIGURA 3-7 Um grafo planar e uma representação de visibilidade correspondente.

Em **[TT86]**, Tamassia e Tollis descrevem algoritmos para verificar a existência e construir três tipos de representação de visibilidade para grafos planares. A seguir, descreveremos o algoritmo geral de construção de tais representações.

Primeiramente, apresentaremos o algoritmo que constrói em tempo linear no número de vértices uma representação de visibilidade para um grafo planar biconexo $G=(V,E)$; depois, o algoritmo é estendido para o caso de qualquer grafo planar conexo. O algoritmo produz, ao final, as coordenadas inteiras dos pontos finais de cada segmento-vértice e segmento-aresta.

Um PERT-digrafo $D=(V,A)$ é um digrafo acíclico com exatamente uma fonte s, e um sorvedouro t, onde s é um vértice que possui grau de entrada igual a 0, e t é um vértice que possui grau de saída igual a 0. Um problema clássico sobre PERT-digrafos é encontrar para cada vértice do digrafo o caminho mais longo a partir da fonte. O método do caminho crítico **[Tar72]** resolve esse problema em $O(|A|)$.

Dada uma st-numeração ξ para um grafo $G=(V,E)$, é possível construir um PERT-digrafo $D=(V,A)$ orientando cada aresta do vértice de número menor para o vértice de número maior; ou seja, $(u,v) \in A$, se $\xi(u) < \xi(v)$ **[ET76]**. Utilizando-se dessas informações, enumeramos a seguir os passos do algoritmo, conforme **[TT86]**:

algoritmo Repr_Visibilidade {
  selecionar uma aresta $(s,t) \in E$;
  calcular uma st-numeração para G, a partir de s (vide seção 3.1);
  seja D o grafo orientado induzido pela st-numeração;
  encontrar uma representação planar $D^{\wedge}$ de D tal que o arco $[s,t]$ esteja na face externa e o restante de D esteja do lado direito de $[s,t]$;
  construir um novo digrafo D^* da seguinte forma:
    - vértices de D^* correspondem às faces de $D^{\wedge}$;
    - há um arco $[f,g]$ em D^* se a face f compartilha um arco de $D^{\wedge}$ com a face g e a face f está à esquerda desse arco, quando o mesmo é percorrido da origem em direção ao destino;
  seja D^* um PERT-digrafo, com fonte s^* (a face interna contendo o arco $[s,t]$) e sorvedouro t^* (a face externa);
  aplicar o método do caminho crítico, a partir de s^*, atribuindo o valor 2 a todos os arcos. Ao final, para cada vértice f de D^* haverá um valor $\alpha(f)$ calculado;
  construir a representação de visibilidade como segue:
    - atribuir coordenadas y aos segmentos-vértice a partir da st-numeração calculada no segundo passo;
    - atribuir o valor -1 à coordenada x do arco $[s,t]$;
    - atribuir às coordenadas x de todos os outros arcos de $D^{\wedge}$ um valor inteiro j, tal que $\alpha(f) < j < \alpha(g)$, onde f e g são faces de $D^{\wedge}$ que compartilham aquele arco;
    - atribuir como coordenada y aos pontos finais dos segmentos-aresta um valor igual à coordenada y dos segmentos-vértice aos quais estão conectados;
    - atribuir como coordenada x aos pontos finais esquerdo e direito de cada segmento-vértice as coordenadas x mínima e máxima de seus arcos incidentes, respectivamente;
}

Um exemplo do funcionamento do algoritmo para o grafo G da figura 3-8 pode ser visualizado nas figuras 3-9 e 3-10.

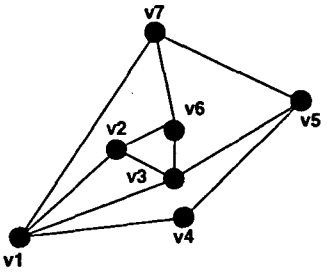


FIGURA 3-8 Um grafo não-orientado G.

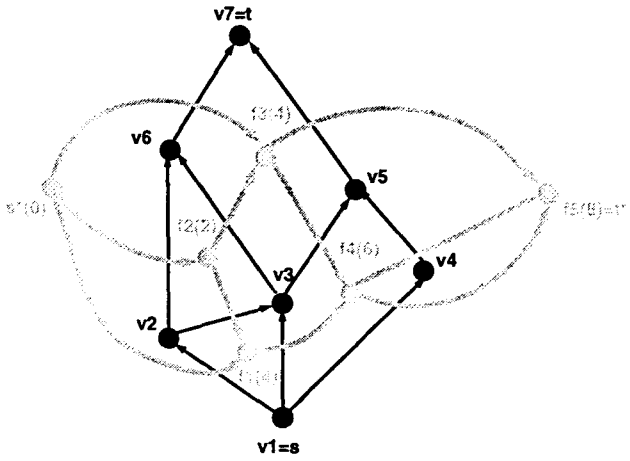


FIGURA 3-9 Grafos orientados $D^{\wedge}$ (em preto) e D^{*} (em cinza) derivados do grafo G. O valor α(f) para cada face f aparece entre ().

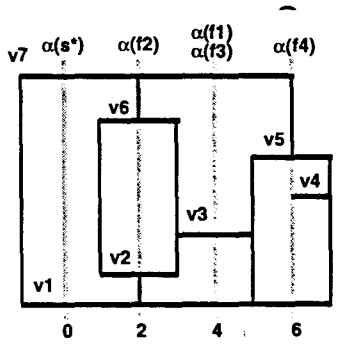


FIGURA 3-10 Representação de visibilidade para G.

A extensão do algoritmo para grafos planares conexos é a seguinte:

algoritmo Repr_Visibilidade_Estendida {
encontrar as componentes biconexas $B_1, ..., B_m$ do grafo G **[Tar72]**;
seja $T = \{B_1, ..., B_m\}$ e $S = \emptyset$;
construir uma representação de visibilidade para B_1;
$T = T - \{B_1\}$;
$S = S \cup \{B_1\}$;

enquanto ($T \neq \emptyset$) fazer{
sejam B_{c1}, ..., B_{ck} todas as componentes biconexas de T que tenham um ponto de articulação c em comum com algum bloco pertencente a S;
encontrar uma representação de visibilidade para cada bloco B_{ci}, usando c como valor de s, no primeiro passo do algoritmo;
a partir da representação de visibilidade construída para S, encaixar as representações de visibilidade construídas para as componentes B_{ci} no topo do segmento-vértice correspondente a c;

$$S = S \cup (\bigcup_{i=1}^{k} Bci)$$

$$T = T - \bigcup_{i=1}^{k} Bci$$

}
}

Ainda em **[TT86]** são caracterizadas outras representações de visibilidade para grafos planares mais restritas onde, por exemplo, vértices são representados por intervalos e não por segmentos ou, ainda, segmentos devem ser visíveis na representação se e somente se são adjacentes no grafo.

Na próxima seção descrevemos um algoritmo para desenhos ortogonais de grafos em grade, cujo núcleo baseia-se no algoritmo descrito nesta seção.

3.4 Desenhos ortogonais em grade

Como o próprio título sugere, desenhos ortogonais em grade seguem o padrão gráfico de grade para o desenho de vértices e arestas. Entre aplicações para a imersão de um grafo em uma grade retilínea podemos citar: desenho de circuitos integrados, projetos arquitetônicos, comunicação por luz ou micro-ondas, problemas de transporte e desenho automático de diagramas.

Nesta seção, descreveremos um algoritmo projetado por Tamassia e Tollis em **[TT89]** onde as principais medidas de qualidade para a imersão planar de um grafo em grade por eles considerados são:

1. minimização da área do menor retângulo que cobre a imersão;
2. minimização do número de quebras ao longo das arestas;
3. minimização do número máximo de quebras ao longo de uma única aresta; e
4. minimização do tamanho total das arestas.

Considerando um grafo planar G com n vértices e grau(G) ≤ 4, o algoritmo descrito constrói uma imersão planar em grade de G em tempo linear com as seguintes propriedades **[TT89]**:

- o número total de quebras nas arestas é, no máximo, $2n + 4$, se G é biconexo, e $2{,}4n + 2$, senão;
- o número de quebras ao longo de cada aresta é no máximo 4; além disso, se G é biconexo, todas as arestas (com exceção de duas) têm no máximo 2 quebras;

- o tamanho de cada aresta é $O(n)$; e
- a área da imersão é $O(n^2)$.

Em um outro artigo **[Tam87]**, Tamassia apresenta um algoritmo baseado em técnicas de fluxo em redes que computa uma imersão em grade com número mínimo de quebras nas arestas em tempo $O(n^2 \log_n)$.

O algoritmo de Tamassia e Tollis compõe-se de quatro fases:

1. construção de uma representação de visibilidade para o grafo G, a partir do algoritmo descrito na seção 3.3 ;
2. transformação da representação de visibilidade em uma imersão preliminar com número total de quebras nas arestas igual a $O(n)$;
3. aplicação de transformações de estiramento de quebras[1], com o intuito de redução do número total de quebras nas arestas; e
4. compactação da imersão planar nas direções horizontal e vertical.

Depois de concluída a primeira fase do algoritmo, a representação de visibilidade correspondente ao grafo é transformada em uma imersão ortogonal[2], através da substituição de cada segmento-vértice por uma das estruturas que aparecem na figura 3-11. Casos simétricos são desenhados simetricamente .

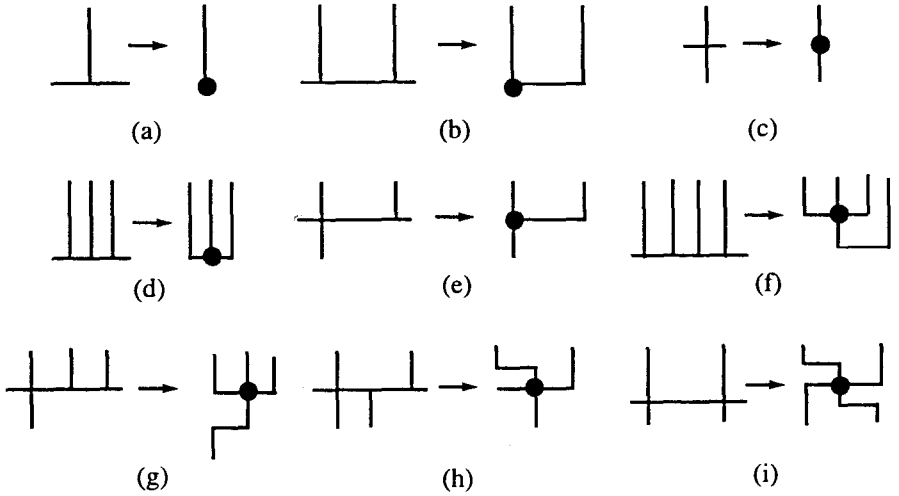


FIGURA 3-11 Substituições realizadas no passo 2.

A representação ortogonal de $G^{\wedge}$ resultante do segundo passo descreve seu formato sem considerar tamanhos de segmentos, consistindo de listas de adjacências com as seguintes informações para cada aresta pertencente à lista de arestas incidentes a um dado vértice v do grafo:

1. *bend-stretching*, em inglês.
2. uma imersão ortogonal $G^{\wedge}$ de um grafo G é uma imersão planar de G cujas arestas são seqüências alternadas de segmentos horizontais e verticais.

1. dest(e): vértice adjacente a v através da aresta e;
2. next(e): próxima aresta incidente a v a partir de e em sua lista de adjacências;
3. sym(e): aresta simétrica de e, pertencente à lista de adjacências de dest(e);
4. angle(e): ângulo que a aresta (v, dest(e)) forma com a aresta (v, dest(next(e))) no vértice v, medido em unidades de $\pi/2$; e
5. bends(e): número de quebras ao longo da aresta (v, dest(e)) tal que o ângulo $\pi/2$ apareça no lado esquerdo da aresta quando se caminha de v a dest(e). As quebras em que o ângulo $\pi/2$ aparece no lado direito da aresta são contadas em bends(sym(e)).

Afigura 3-12 descreve melhor essas informações:

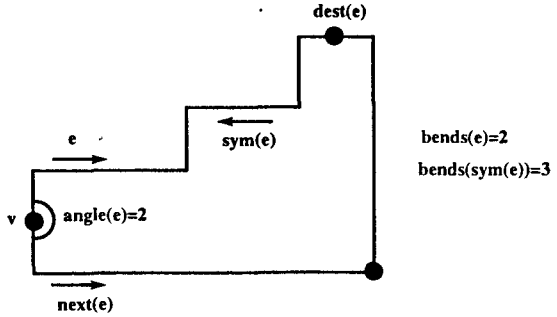


FIGURA 3-12 Informações adicionais de uma representação ortogonal.

Desse modo, o número total de quebras ao longo das arestas em uma imersão ortogonal $\hat{G}$ pode ser definido como:

$$\beta(\hat{G}) = \sum_{v \in V} \sum_{e \in A(v)} bends(e)$$

O objetivo agora é minimizar $\beta(\hat{G})$. Para isso, são aplicados três tipos de transformações de estiramento de quebras descritas e exemplificadas a seguir:

1. Transformação T1: remove pares de quebras direita e esquerda. A condição para que T1 seja aplicada é que uma aresta e sua simétrica apresentem ao mesmo tempo uma ou mais quebras, onde uma anulará o efeito da outra como mostra a figura 3-13.

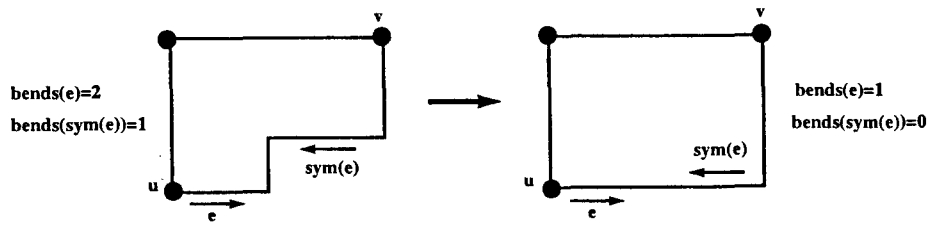


FIGURA 3-13 Exemplo de transformação T1.

2. Transformação T2: rotaciona um vértice v no sentido horário (anti-horário) se todas as suas arestas incidentes tiverem uma quebra direita (esquerda) com relação a v, como mostra a figura 3-14.

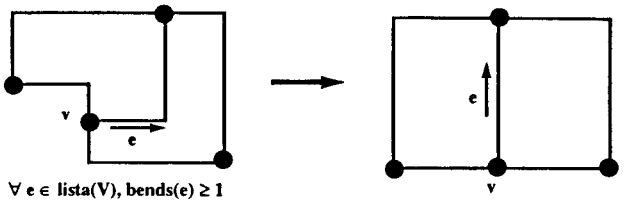


FIGURA 3-14 Exemplo de transformação T2.

3. Transformação T3: uma quebra absorve um vértice quando este possui arestas incidentes consecutivas que fazem entre si um ângulo maior ou igual a π, e a quebra pertence a uma das arestas, como ocorre na figura 3-15.

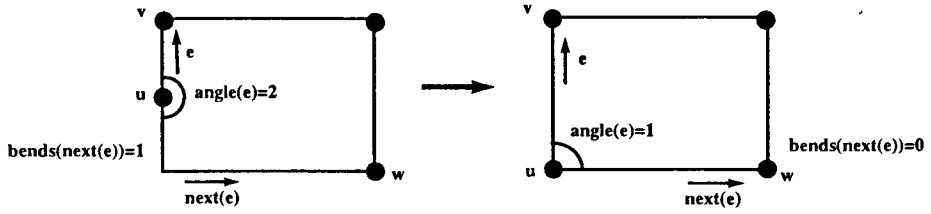


FIGURA 3-15 Exemplo de transformação T3.

Uma vez que todas as transformações sejam oportunamente aplicadas, o número total de quebras da representação ortogonal do grafo G deve diminuir **[TT89]**.

Para que a representação ortogonal seja transformada finalmente em uma representação em grade, devem ser atribuídos aos segmentos tamanhos inteiros. Isso pode ser conseguido, por exemplo, pelo uso de algoritmos de compactação para circuitos integrados, como veremos no Capítulo 6.

A figura 3-16 mostra um exemplo de execução do algoritmo descrito.

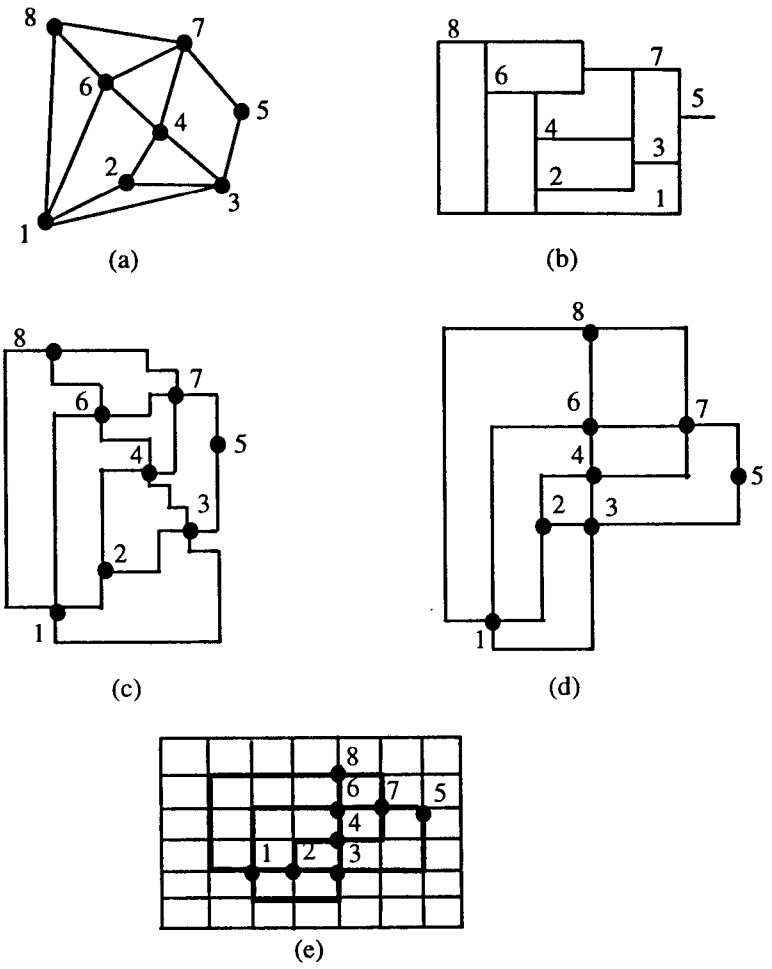


FIGURA 3-16 Exemplo de execução do algoritmo para desenho ortogonal em grade.

Esse algoritmo será referenciado posteriormente (vide Capítulo 6) por compor um outro algoritmo de propósitos mais gerais que, apenas, gerar o desenho de um grafo planar em grade. Uma relação de problemas em aberto relacionados ao desenho ortogonal planar de grafos pode ser encontrada em **[Tam90]**.

Capítulo 4

Grafos Orientados Acíclicos

4.1 Introdução

Grafos orientados acíclicos são geralmente utilizados na representação de relacionamentos hierárquicos como, por exemplo, em representação de estruturas e fluxos de dados, autômatos finitos, grafos de chamadas de procedimento e dependências de configuração de software.

As definições e conceitos relacionados a grafos orientados acíclicos, aqui denominados hierarquias, foram basicamente retirados de **[War77]** e **[STT81]**. Quando falamos de hierarquias, estamos tratando de subconjuntos de vértices de um grafo dispostos em camadas (ou níveis) ordenados de alguma forma, e com um conjunto de arestas sempre orientadas de uma camada para outra.

As primeiras investigações a respeito do desenho de estruturas hierárquicas surgiram a partir da observação de que, desenhos manuais de hierarquias, com um grande número de atividades e relacionamentos, geravam resultados pouco legíveis.

Os principais elementos de legibilidade para o desenho de tais estruturas foram identificados como sendo:

1. disposição dos vértices em níveis ou camadas;
2. minimização do número de cruzamentos entre arestas;
3. minimização do número de quebras nas arestas, ou seja, as arestas devem ser desenhadas o mais reto possível;
4. desenho balanceado de arestas que entram e saem de um dado vértice; e
5. desenho próximo de vértices interconectados, ou seja, o caminho entre eles deve ser o mais curto possível para os olhos do observador.

O desenho de uma hierarquia é chamado mapa. Em um mapa, os vértices pertencentes à i-ésima camada são distribuídos na i-ésima coluna, dentre l colunas corres-

pondentes às l camadas da hierarquia, numeradas em ordem crescente da esquerda para direita. Além disso, dentro de cada camada i deve existir uma ordem σ_i de desenho dos vértices pertencentes àquela camada.

O *span* de uma aresta é a diferença entre as camadas dos vértices por ela conectados. Se cada aresta da hierarquia possui *span* igual a 1, a hierarquia é chamada própria, senão ela é imprópria. Se uma hierarquia é imprópria, ela pode ser sempre transformada em uma hierarquia própria, introdu-zindo-se vértices adicionais, chamados vértices-fantasmas, ao longo das arestas cujo *span* é maior que 1.

O tamanho de uma hierarquia é o seu número de camadas, enquanto que a largura é o número de vértices da camada que possui a maior quantidade de vértices. A figura 4-1 ilustra melhor os conceitos até aqui mencionados. Na figura 4-1.b podemos ver o resultado da transformação de uma herarquia imprópria em própria, onde f1 e f2 são vértices-fantasmas.

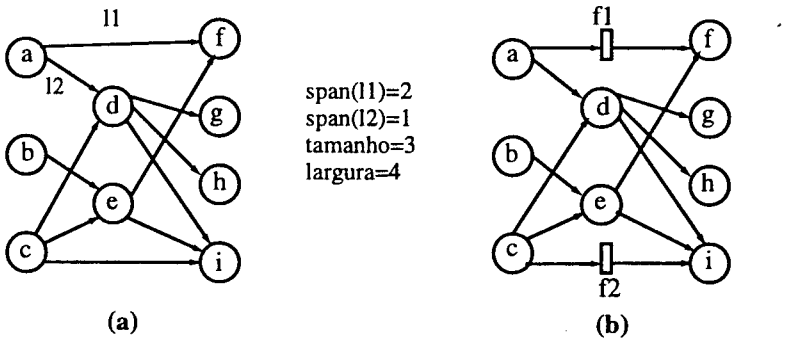


FIGURA 4-1 Conceitos relacionados a hierarquias. Em (b) os vértices-fantasmas são representados explicitamente.

A seguir, descreveremos um algoritmo proposto por Sugiyama et al. **[STT81]** que, utilizando esses conceitos e considerando os critérios de legibilidade citados, serviu de base para alguns dos sistemas de desenho automático de grafos já implementados.

4.2 Método do Baricentro

O problema de gerar mapas legíveis é formulado em **[STT81]** como um problema de quatro estágios ou passos, onde o primeiro estágio é preparatório, o segundo e o terceiro realmente determinam as posições dos vértices e o quarto é o responsável pelo desenho final.

No primeiro passo o grafo orientado é transformado em uma hierarquia própria sem ciclos. A remoção de ciclos pode ser feita de várias maneiras. Na proposta original de Sugiyama et al., os vértices que fazem parte de um ciclo detectado são condensados em um único vértice e tratados a parte no final do algoritmo.

Uma solução melhor para esse problema seria simplesmente reverter temporariamente as arestas que causam ciclos no grafo e tratá-las normalmente durante o processo, lembrando apenas de revertê-las novamente ao final. Isso poder ser con-

seguido com uma simples busca em profundidade no grafo de entrada. Essa idéia foi implementada em nosso trabalho, assim como em outros que utilizam o método do baricentro.

Ainda no primeiro passo, a distribuição dos vértices pertencentes ao grafo orientado, agora acíclico, em camadas, pode ser feita de duas formas. Seja fonte um vértice cujo grau de entrada é zero. Uma das formas de realizar esse passo é o uso do método dos caminhos máximos entre vértices, onde todas as fontes são alocadas à primeira camada, e cada vértice restante v é colocado na camada m, onde o maior caminho de v a uma das fontes tenha comprimento igual a m. A vantagem desse método é a criação de uma hierarquia de tamanho mínimo, mas que concentra um grande número de vértices nas últimas camadas.

Outra maneira de distribuir os vértices em camadas é alocar as fontes na primeira camada, retirando-as do grafo, assim como suas arestas incidentes, e continuar esse processo com os vértices que sobraram no grafo. Essa distribuição não garante o tamanho mínimo, mas apresenta bons resultados.

Ainda no primeiro passo, arestas $u \rightarrow v$ com *span* j maior que 1 são substituídas por caminhos $u = v_0 \rightarrow v_1 \rightarrow ... \rightarrow v_j = v$, onde v_1, ..., v_{j-1} são vértices-fantasmas interligados por arestas-fantasmas.

O segundo passo do algoritmo é o responsável pela redução de cruzamentos entre arestas, através da permutação de vértices dentro de cada camada. Originalmente, foram propostos dois métodos para a resolução desse passo, um teórico e outro heurístico. O teórico é chamado Método da Minimização de Penalidade, enquanto que o método heurístico, modificado posteriormente em outros trabalhos, denomina-se Método do Baricentro - BC. Aqui descreveremos apenas o método BC .

O Método BC é baseado na idéia de que, se cada vértice for posicionado considerando a posição de seus vértices vizinhos nas camadas adjacentes, o número de cruzamentos pode ser bastante reduzido.

No terceiro passo são encontradas as coordenadas reais de cada vértice, respeitando-se a ordem para eles encontrada no passo anterior. Para esse passo também foram propostos dois tipos de solução. Um deles reformula o problema na forma de programação quadrática, e o outro é uma solução heurística, chamada Método de Desenho por Prioridade - PR, que possui uma idéia similar ao método BC, e também será descrita neste capítulo.

O quarto e último passo mostra o resultado do algoritmo na área de desenho, lembrando-se de tratar os nós fantasmas introduzidos no primeiro passo, e de devolver o sentido original às arestas antes revertidas.

As estruturas de dados utilizadas são um conjunto de matrizes de interconexão M^i entre cada duas camadas i e $i+1$. Um valor m^i_{vw} da matriz M^i é igual a 1 se a aresta

v→w pertence ao grafo, e é igual a 0, caso contrário. Para a hierarquia da figura 4-1.b, por exemplo, teríamos as seguintes matrizes de interconexão:

$$M^1 = \begin{array}{c|cccc} & f1 & d & e & f2 \\ a & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 \\ c & 0 & 1 & 1 & 1 \end{array} \qquad M^2 = \begin{array}{c|cccc} & f & g & h & i \\ f1 & 1 & 0 & 0 & 0 \\ d & 0 & 1 & 1 & 1 \\ e & 1 & 0 & 0 & 1 \\ f2 & 0 & 0 & 0 & 1 \end{array}$$

Como veremos a seguir, as matrizes facilitam o processo de cálculo do número de cruzamentos, da conectividade e dos baricentros de cada vértice.

A conectividade de um vértice é simplesmente a quantidade de vértices adjacentes a ele nas camadas vizinhas (anterior e posterior). O baricentro de um vértice é sua posição relativa aos vértices que lhe são adjacentes também nas camadas vizinhas.

Seja M^i uma matriz de interconexão entre duas camadas consecutivas da hierarquia e m_{ab} um elemento de M^i. Sejam q e r, respectivamente, o número de vértices pertencentes às camadas i e $i+1$, incluindo os vértices-fantasmas; então, o número de cruzamentos K de M^i é dado por:

$$K(M^i) = \sum_{j=1}^{q-1} \sum_{k=j+1}^{q} \left(\sum_{\alpha=1}^{r-1} \sum_{\beta=\alpha+1}^{r} m^i_{j\beta} m^i_{k\alpha} \right)$$

E o número total de cruzamentos de um grafo G com l camadas é dado por:

$$K(G) = \sum_{k=1}^{l-1} K(M^i)$$

Sejam p, q e r, respectivamente, o número de vértices das camadas $i-1$, i e $i+1$. Então, definimos as conectividades anterior e posterior de um dado vértice v_k pertencente à camada i, denotado por v_k^i, como sendo:

$$C^A_{ik} = \sum_{j=1}^{p} m^{i-1}_{jk}, \qquad 1 \le k \le q \text{ e } 2 \le i \le l;$$

$$C^P_{ik} = \sum_{j=1}^{r} m^i_{kj}, \qquad 1 \le k \le q \text{ e } 1 \le i \le l-1.$$

São definidos ainda quatro tipos de baricentro para cada vértice. Os baricentros de linha e de coluna, utilizados no segundo passo do algoritmo e os baricentros anterior e posterior, utilizados no terceiro passo.

O baricentro de linha (coluna) significa a posição de um vértice relativa a seus vizinhos da camada posterior (anterior). O baricentro posterior (anterior) indica a coordenada y média de um vértice com relação às coordenadas y de seus vizinhos adjacentes na camada posterior (anterior).

Abaixo definimos os quatro baricentros para v_k^i:

$$B_{ik}^L = \left(\sum_{l=1}^{r} l \cdot m_{kl}^i\right) / \left(\sum_{l=1}^{r} m_{kl}^i\right), \quad 1 \le k \le q;$$

$$B_{ik}^C = \left(\sum_{l=1}^{q} l \cdot m_{lk}^i\right) / \left(\sum_{l=1}^{q} m_{lk}^i\right), \quad 1 \le k \le r;$$

$$B_{ik}^A = \left(\sum_{j=1}^{p} y(v_j^{i-1}) \cdot m_{jk}^{i-1}\right) / C_{ik}^A, \quad 1 \le k \le q \text{ e } 2 \le i \le l;$$

$$B_{ik}^P = \left(\sum_{j=1}^{r} y(v_j^{i+1}) \cdot m_{kj}^i\right) / C_{ik}^P, \quad 1 \le k \le q \text{ e } 1 \le i \le l-1.$$

É importante informarmos agora que as heurísticas utilizadas nos passos descritos a seguir foram desenvolvidas para hierarquias de dois níveis apenas, por isso a necessidade da introdução dos vértices-fantasmas. A gene-ralização dessa heurística para l níveis se dá pela aplicação sucessiva da mesma entre cada dois dos l níveis.

Para a reordenação dos vértices dentro de uma camada, com o intuito de diminuir o número de cruzamentos entre duas camadas consecutivas i e $i+1$, o método do Baricentro se utiliza desses valores de baricentro de acordo com o algoritmo a seguir.

```
algoritmo Baricentro {
  Fase1:
    enquanto ( o número de cruzamentos de M^i for reduzido e
               o número de iterações < k ) {
      calcular baricentros de linha para os vértices pertencentes a M^i;
      se ( o número de cruzamentos não aumentar )
        reordenar linhas de M^i de acordo com o valor crescente
          dos baricentros de linha ;
      calcular baricentros de coluna para os vértices pertencentes a M^i;
      se ( o número de cruzamentos não aumentar )
        reordenar colunas de M^i de acordo com o valor crescente
          dos baricentros de coluna;
    }

  Fase 2:
    enquanto ( o número de iterações < k e
               houver alguma permutação de vértices ) {
      se ( M^i tiver vértices com baricentros de linha iguais ) {
        se ( o número de cruzamentos não aumentar ) {
          permutar as linhas correspondentes àqueles vértices em M^i;
          recalcular os baricentros de coluna de M^i;
        }
        se ( as colunas de M^i estão desordenadas com relação
             aos seus baricentros )
          repetir Fase1 começando pela reordenação das
            colunas de M^i;
```

```
        }
        se ( M^i tiver vértices com baricentros de coluna iguais ) {
          se ( o número de cruzamentos não aumentar ) {
            permutar as colunas correspondentes àqueles vértices em M^i;
            recalcular os baricentros de linha de M^i;
          }
          se ( as linhas de M^i estão desordenadas com relação
               aos seus baricentros )
            repetir Fase1 começando pela reordenação das
              linhas de M^i;
        }
      }
    }
```

Desse modo, a primeira fase ordena linhas ou colunas, que representam vértices e suas posições correntes dentro de cada camada, preservando a ordem se duas linhas ou colunas possuem baricentros iguais. Já a segunda fase permuta tais linhas ou colunas com a intenção de reduzir ainda mais o número de cruzamentos. Devemos lembrar que cada permuta de linhas (colunas), em uma matriz de interconexão, corresponde a uma permuta de colunas (linhas) na matriz de interconexão anterior (posterior).

Considerando a hierarquia de dois níveis da figura 4-2, podemos acompanhar, a seguir, o comportamento do algoritmo descrito acima.

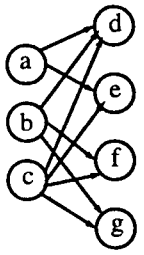


FIGURA 4-2 Hierarquia de dois níveis G_0.

M_0 é a matriz de interconexão inicial de G_0 (os valores que aparecem ao lado da matriz representam os baricentros de linha de cada vértice):

		d	e	f	g	
	a	1	1	0	0	1,5
$M_0 =$	b	1	0	1	1	2,7
	c	1	1	1	1	2,5

$K(M_0) = 7$

A) Reordenando as linhas b e c de M_0, M_1 é obtida (os valores que aparecem abaixo da matriz representam os baricentros de coluna de cada vértice):

		d	e	f	g	
	a	1	1	0	0	1,5
$M_1 =$	c	1	1	1	1	2,5
	b	1	0	1	1	2,7
		2,0	1,5	2,5	2,5	

$K(M_1) = 6$

B) Reordenando as colunas d e e de M_1, M_2 é obtida (fim da Fase1):

		e	d	f	g	
	a	1	1	0	0	1,5
$M_2 =$	c	1	1	1	1	2,5
	b	0	1	1	1	3,0
		1,5	2,0	2,5	2,5	

$K(M_2) = 4$

C) Permutando as colunas f e g de M_2, M_3 é obtida (início e fim da Fase 2):

		e	d	g	f	
	a	1	1	0	0	1,5
$M_3 =$	c	1	1	1	1	2,5
	b	0	1	1	1	3,0
		1,5	2,0	2,5	2,5	

$K(M_3) = 4$

A figura 4-3 representa um mapa da hierarquia G_3 correspondente à matriz M_3:

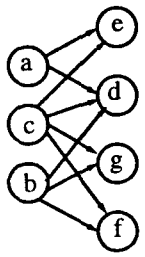


FIGURA 4-3 Ordenação final dos vértices de acordo com o algoritmo Baricentro.

O problema de minimizar cruzamentos em hierarquias de l camadas é reduzido a (l-1) subproblemas P_i relacionados a hierarquias de dois níveis, onde P_i significa minimizar o número de cruzamentos entre as camadas i e i+1. Entretanto, a resolução do subproblema P_i ainda depende das soluções dos subproblemas P_{i-1} e P_{i+1}, uma vez que as posições dos vértices na i-ésima camada devem ser calculadas de acordo como os vértices adjacentes nas duas camadas vizinhas.

Assim, para a resolução de P_i, dois novos subproblemas mais fracos foram introduzidos: P_i^P e P_i^A. P_i^P é a minimização do número de cruzamentos entre as camadas i e i+1, através da permutação dos vértices da camada i+1 de acordo com a

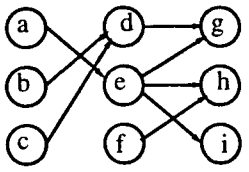
ordenação da camada i. P_i^A tem a mesma definição, contudo a permutação dos vértices é realizada na camada i de acordo com a ordem pertencente à camada i+1.

Resolvendo-se sucessivamente os subproblemas P_i^P (P_i^A) ao longo das camadas, o número de cruzamentos é reduzido **[STT81]**. P_i^P (P_i^A) é resolvido através dos cálculos dos valores de baricentro de coluna (linha) de cada vértice, e o procedimento é denominado POST (ANT). O procedimento inteiro aplicado ao conjunto M de matrizes de adjacências é chamado POST_ANT e é descrito a seguir:

```
Algoritmo POST_ANT {
  i = 1;
  atribuir uma ordem inicial ao nível do topo da hierarquia;
  enquanto ( o número de cruzamentos de M for reduzido e
             o número de iterações < k ) {
    POST: enquanto ( i < l -1 ) {
             Reordenar os vértices do nível i +1 de acordo com os
             valores de baricentro de coluna da matriz M^i;
             i = i + 1;
           }
    ANT: enquanto ( i > 1 ) {
             Reordenar os vértices do nível i de acordo com os
             valores de baricentro de linha da matriz M^i;
             i = i - 1;
         }
  }
}
```

O procedimento POST_ANT é realizado inicialmente como a primeira fase do algoritmo Baricentro, conservando a ordem dos vértices com baricentros iguais dentro da i-ésima camada. Como no algoritmo Baricentro, uma segunda fase pode ser executada sobre M, em que a ordem das linhas (colunas) correspondentes à i-ésima camada pode ser trocada, caso existam vértices com mesmo valor de baricentro. Assim, a primeira fase descrita pelo procedimento POST_ANT é realizada mais uma vez. O método BC de l-níveis consiste da composição dessas duas fases.

Um exemplo de execução do método BC de l-níveis é visto a seguir. Considere, por exemplo, a hierarquia de três níveis da figura 4-4 e, logo após, os passos relativos ao método BC:



K(G) = 3

FIGURA 4-4 Hierarquia de três níveis G_0.

M^1 e M^2 são as matrizes de interconexão iniciais de G_0:

M^1 =

	d	e	f
a	0	1	0
b	1	0	0
c	1	0	0
	2,5	1	0

M^2 =

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

A) Cálculo dos baricentros de coluna de M^1 (acima);

B) Permutação das colunas "d" e "e" de M^1 e, conseqüentemente, das linhas "d" e "e" de M^2;

M^1 =

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

M^2 =

	g	h	i
e	1	1	1
d	1	0	0
f	0	1	0
	1,5	2	1

K(G) = 3

C) Cálculo dos baricentros de coluna da matriz M^2 (acima);

D) Permutação das colunas "g", "h" e "i" (Fim da 1ª etapa, pois as linhas e colunas de M^1 e M^2 estão ordenadas pelos baricentros);

M^1 =

	e	d	f	
a	1	0	0	1
b	0	1	0	2
c	0	1	0	2
	1	2,5	0	

M^2 =

	i	g	h	
e	1	1	1	2
d	0	1	0	2
f	0	0	1	3
	1	1,5	2	

K(G) = 2

E) Permutação das linhas "e" e "d" de M^2 e das colunas "e" e "d" de M^1 (início da 2ª etapa), que é rejeitada por aumentar o número de cruzamentos;

M^1 =

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

M^2 =

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

K(G) = 3

F) Permutação das linhas "b" e "c" de M^1, que não afeta o número de cruzamentos, podendo ser aceita ou rejeitada (aceita , neste caso), terminando a 2ª etapa;

$$M^1 = \begin{array}{c|ccc|c} & e & d & f & \\ a & 1 & 0 & 0 & 1 \\ c & 0 & 1 & 0 & 2 \\ b & 0 & 1 & 0 & 2 \\ \hline & 1 & 2{,}5 & 0 & \end{array} \qquad M^2 = \begin{array}{c|ccc|c} & i & g & h & \\ e & 1 & 1 & 1 & 2 \\ d & 0 & 1 & 0 & 2 \\ f & 0 & 0 & 1 & 3 \\ \hline & 1 & 1{,}5 & 2 & \end{array} \qquad K(G) = 1$$

Um mapa da hierarquia correspondente às matrizes finais M^1 e M^2 seria o da figura 4-5:

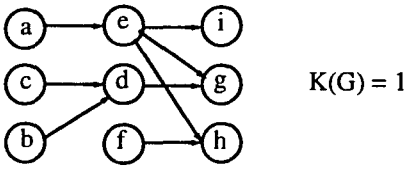


FIGURA 4-5 Hierarquia de três níveis G_0, depois da execução do algoritmo POST_ANT.

Uma vez encontrada uma ordenação para os vértices dentro de cada camada, tal que o número de cruzamentos seja tão pequeno quanto o possível, podemos calcular suas coordenadas reais.

A idéia fundamental do terceiro passo é similar à do método BC, no sentido de que o problema de atribuir posições aos vértices é decomposto em operações entre cada dois níveis sucessivamente.

Inicialmente, a todos os vértices é atribuída a coordenada *x*, de acordo com a camada a qual pertecem. As camadas são distribuídas igualmente em toda a extensão horizontal da área de desenho, e a coordenada *x* é mantida constante até o final do algoritmo.

Depois disso, são atribuídas coordenadas *y* iniciais aos vértices em cada camada. O primeiro vértice, de acordo com a ordenação encontrada no passo anterior, receberá uma posição inicial e, a partir dessa posição, todos os outros vértices do mesmo nível sofrerão um certo deslocamento cons-tante.

Melhoras no posicionamento dos vértices são feitas na ordem dos níveis 2, ..., *n*, *n*-1, ..., 1, *t*, ..., *n*, onde *t* é um inteiro e $2 \leq t \leq n\text{-}1$. Aqui denominamos DOWN (UP) o procedimento responsável pela melhora nos níveis 2, ..., *n* e *t*, ..., *n* (*n*-1, ..., 1).

A cada vértice é atribuída uma prioridade de posicionamento dentro de sua camada, por isso o nome Método de Desenho por Prioridade. Nós-fantasmas, por exemplo, recebem o maior valor de prioridade, significando serem os nós-fantasmas os primeiros posicionados para que consigamos desenhar suas arestas o mais reto possível.

Para todos os outros vértices, a prioridade anterior (posterior) é definida como sendo igual à sua conectividade anterior (posterior), que será utilizada pelo procedimento DOWN (UP).

A intenção é diminuir a diferença entre a posição na qual o vértice se encontra e o seu valor de baricentro anterior (posterior) durante a execução do procedimento DOWN (UP), respeitando as seguintes restrições:

1. a posição do vértice deve ser um inteiro e não pode ser igual à posição de qualquer outro vértice dentro da mesma camada;
2. a ordem dos vértices encontrada no passo anterior deve ser preservada;
3. um vértice só poderá ser reposicionado em função de outro se este último tiver uma prioridade maior ou igual à sua.

Depois de encontradas as coordenadas *(x, y)* de cada vértice, nada mais resta a fazer senão desenhar a hierarquia, lembrando de apagar nós-fantasmas introduzidos anteriormente, e tratar possíveis quebras de arestas causadas pela criação dos mesmos.

4.2.1 Modificações do algoritmo de Sugiyama et al.

O algoritmo apresentado em **[STT81]** foi largamente utilizado em sistemas de edição/desenho de grafos **[RDM+87] [GNV88] [Mül91]**, sofrendo assim diversas melhoras no decorrer do tempo. Algumas dessas modificações são descritas a seguir.

Rowe et al. **[RDM+87]** se propuseram a implementar um navegador de propósitos gerais para grafos orientados que pudesse ser integrado com outras aplicações, como por exemplo a navegação entre procedimentos pertencentes a um mesmo sistema de informação.

O algoritmo original executa suas travessias na hierarquia do nível mais baixo para o mais alto, e vice-versa, calculando ao mesmo tempo posições relativas entre os vértices em níveis consecutivos, de acordo com os baricentro de coluna e linha daqueles vértices. Rowe et al. observaram que esse procedimento faz com que alguns vértices fiquem instáveis dentro de seu nível e, assim, sugeriram que a posição relativa de um vértice fosse calculada considerando a média dos dois baricentros durante uma terceira travessia. Eles observaram, também, que o algoritmo convergia mais rapidamente com relação à diminuição do número de cruzamentos.

Alguns problemas foram apontados em **[RDM+87]** com relação ao algoritmo de Sugiyama et al., mas mesmo assim, não foram solucionados naquele trabalho, como por exemplo, problemas relativos ao fato de que apenas camadas consecutivas são tratadas a um dado instante, ou seja, o algoritmo trabalha apenas com informação local, e não global, da hierarquia.

No trabalho de Gasner et al. **[GNV88]** foi desenvolvido um programa, denominado DAG, para o desenho automático de grafos orientados, representados por listas de arestas. DAG permite que o usuário mude atributos de vértices, como seu formato, tamanho e label. Atributos que afetam o espaçamento na figura entre os vértices também podem ser modificados.

Em **[GNV88]**, a criação da hierarquia a partir do conjunto de arestas fornecido como entrada é realizada através da formulação de um problema de programação inteira, que fornece uma atribuição ótima de vértices aos níveis da hierarquia. O problema dessa abordagem é o aumento da complexidade e de consumo de tempo desse passo. Ao menos, com isso, espera-se reduzir ao máximo o número de vértices-fantasmas na hierarquia.

A geração da ordem inicial dos vértices dentro de cada camada é feita através de uma busca em profundidade começando nos vértices pertencentes à primeira camada, o que pode resultar em uma disposição inicial com poucos cruzamentos.

A modificação principal no núcleo do algoritmo, ou seja, no segundo e terceiro passos, foi o uso de uma função ponderada de cálculo da posição de um dado vértice. A função é chamada mediana ponderada e é calculada do seguinte modo: quando o número de vértices adjacentes em uma camada vizinha é ímpar, a mediana ponderada é a mediana desses vértices adjacentes; caso contrário, a mediana ponderada estará entre as posições medianas esquerda e direita. Neste caso, o conceito de baricentro não é utilizado.

Para o terceiro passo, atribuição de coordenadas aos vértices, são propostos um método heurísitico, similar ao passo anterior, e um teórico, que é a resolução de um sistema de programação linear. O segundo pode produzir melhores resultados, mas pode representar um gargalo para o DAG **[GNV88]**. No DAG, vértices-fantasmas fazem o papel de pontos de *spline* no desenho final das arestas.

A maioria das experiências com o algoritmo de Sugiyama e suas variações apresentou bons resultados em cima de exemplos reais de hierarquias. Em **[Sug87]** e **[ES90]**, há um resumo de algumas versões de cada fase do algoritmo proposto em **[STT81]**. Em **[Mes88]**, é proposta uma extensão do algoritmo de Sugiyama et al. para hierarquias com um grande número de nós (acima de 200, por exemplo).

No capítulo dedicado à implementação contida neste trabalho poderemos analisar o comportamento desse algoritmo aplicado aos diagramas da LegoShell.

Capítulo 5

Grafos Gerais não-orientados

5.1 Introdução

Problemas relacionados ao desenho de grafos gerais não-orientados são, em geral, NP-completos **[ES90] [Bra88]** como a minimização de cruzamentos entre arestas e a determinação de isomorfismos entre grafos. Muitas vezes, métodos heurísiticos são utilizados como solução para tais problemas, gerando resultados bastante satisfatórios.

Em geral, o padrão de linha-reta é adotado para o desenho dessa classe de grafos. Desse modo, os algoritmos trabalham apenas com informações geométricas relativas aos vértices. Um trabalho posterior deve ser realizado, se as arestas devem ser desenhadas como seqüências de segmentos horizontais e verticais.

Os aspectos de legibilidade geralmente considerados por esses algoritmos são:

1. distribuição uniforme dos vértices na área do desenho;
2. uniformização do tamanho das arestas;
3. minimização do número de cruzamentos entre arestas; e
4. apresentação de simetrias.

A seguir, veremos como métodos de otimização podem ser aplicados ao problema do desenho de grafos gerais não-orientados. *Simulated Annealing* e *Random Search* foram implementados neste trabalho, e os resultados obtidos serão posteriormente analisados.

5.2 *Simulated Annealing*

SA - *Simulated Annealing* é um método de otimização originário da área de Mecânica Estatística, que possui aplicações em diversos problemas de otimização

combinatória, incluindo localização e roteamento em circuitos VLSI **[SV85]**, o problema do caixeiro viajante, processamento de imagens **[JRA92]** e desenho de grafos. Recentemente, SA foi utilizado no desenvolvimento de um sistema de *layout* para diagramas de sistemas de informação **[DM90]**.

O método SA faz uma analogia com o processo físico de resfriamento de líquidos a uma forma cristalina, e é iterativo por natureza. Partindo de uma temperatura inicial T_0, um líquido pode ser resfriado rápida ou lentamente. No primeiro caso, os átomos chegariam a uma forma cristalina ainda com uma energia elevada, podendo resultar em uma estrutura amorfa; enquanto que, se o resfriamento for lento e gradativo, os átomos teriam, ao final, níveis mais baixos de energia, e a estrutura teria uma forma totalmente ordenada.

O procedimento utilizado na observação do comportamento de um grupo de átomos sujeito a um resfriamento foi o seguinte: deslocamentos aleatórios de átomos são realizados, gerando uma nova configuração no sistema, passível ou não de ser aceita. A nova configuração é aceita com probabilidade $P = \min(1, e^{-\Delta E/kT})$ onde ΔE é a variação de energia do grupo de átomos entre duas iterações, k é a constante de Boltzmann, e T é a temperatura corrente; ou seja, se a energia no sistema diminuir, o movimento do átomo é aceito, senão a mudança de estado é probabilística e depende da variação de energia no sistema e da temperatura corrente. A temperatura T é reduzida e a operação repetida um determinado número de vezes. É esperado que, quando a temperatura atingir valores muito baixos, a probabilidade de aceitação de um novo estado do sistema cuja energia tenha aumentado, seja muito próxima de zero e, assim, apenas os deslocamentos que levem o sistema a uma energia mínima sejam aceitos.

As entidades a serem identificadas durante um processo de SA são:

1. O conjunto de configurações ou estados do sistema, incluindo a configuração inicial.
2. Uma regra de geração de novas configurações, que define a vizinhança de cada configuração, geralmente escolhida aleatoriamente.
3. A função objetivo a ser minimizada sobre o espaço de configuração; que está relacionado à função energia do processo descrito acima.
4. O parâmetro de controle do processo de resfriamento, incluindo seu valor inicial e regras de quando e como mudá-lo; que está relacionado à variável temperatura e seu decréscimo.
5. A condição de término do processo, geralmente baseada nos valores da função objetivo e/ou no parâmetro de controle.

A constante de Boltzmann pode ser esquecida, uma vez que a temperatura não é real, e para o desenho de grafos funciona apenas como um parâmetro de controle do processo de *Annealing*. O fator de redução de temperatura será aqui denotado por α, onde $0 < \alpha < 1$.

Considerando-se as entidades já definidas, o algoritmo geral do processo de SA é apresentado a seguir.

```
algoritmo SA {
  escolher uma configuração inicial σ;
  escolher uma temperatura inicial T;
  enquanto ( um novo estado for aceito ) fazer {
    repetir ( rep ) vezes {
      escolher uma nova configuração σ' da vizinhança de σ;
      sejam E e E' os valores da função objetivo relativos a σ e σ';
      se ( E' < E ou random() < e^(E-E')/T ) então
        σ = σ' ;
    }
    T = T*α;    /* reduzir a temperatura de um fator α */
  }
}
```

Analisando o algoritmo SA é possível observar a importância da escolha dos valores iniciais das variáveis e do fator de redução da temperatura no desempenho do algoritmo quanto ao tempo de execução e à solução final obtida. Um valor inicial muito pequeno para a temperatura e um fator de redução grande, por exemplo, podem forçar o término do algoritmo bem antes de ser encontrada uma solução próxima da ótima. A situação inversa pode comprometer a eficácia do algoritmo, deixando-o muito lento. Em geral, esses valores são determinados com base em análises de resultados experimentais do algoritmo.

Um outro ponto crítico do algoritmo é a maneira de recálculo da função objetivo a cada mudança na configuração do sistema. O ideal é que a função não precise ser totalmente recalculada para todos os elementos do sistema a cada iteração, e sim, que sejam atualizadas apenas as parcelas referentes aos elementos que sofreram alteração de uma configuração para outra. Esse procedimento pode economizar um tempo considerável de computação para o algoritmo.

5.2.1 *Simulated Annealing* para o desenho de grafos

No mapeamento do método SA para uma solução do problema de desenho de grafos, devem ser identificadas todas as entidades descritas anteriormente. Consideramos que, a partir da descrição abstrata de um grafo, aqui representado por um conjunto de listas de adjacências, desejamos chegar a uma boa disposição dos vértices dentro de uma determinada grade retangular.

Devemos lembrar que, quando aplicado ao problema do desenho de grafos, o método SA fornece um posicionamento final para os vértices restando apenas o trabalho final do desenho das arestas. Na proposta original de sua utilização em tal problema, esse trabalho é trivial, uma vez que o padrão gráfico utilizado é o de linha-reta. No caso da adoção do padrão ortogonal de desenho de arestas, é exigida ainda uma solução independente do SA.

O algoritmo apresentado em **[DH89]** se propõe ao desenho legível de grafos gerais não-orientados, seguindo o padrão de linha-reta, através do método de SA. O objetivo principal do trabalho desenvolvido por Davidson e Harel em **[DH89]** é expressar critérios de legibilidade através das componentes de uma determinada função objetivo a ser minimizada.

Cada parcela da função objetivo pode expressar, por exemplo, qualquer um dos três primeiros critérios de legibilidade enumerados no início do capítulo, além de:

- distância equilibrada dos vértices com relação às bordas do desenho;
- proximidade entre um vértice e arestas que não lhe são incidentes;
- uniformização dos ângulos formados entre arestas incidentes a um mesmo vértice;

ou qualquer outro critério que possa ser traduzido matematicamente. Se esses objetivos forem alcançados, é possível que o desenho apresente ainda aspectos de simetria. Através da função objetivo, deve ser permitida a atribuição de pesos a cada um dos critérios, com o intuito de estabelecer prioridades entre eles.

Com relação ao problema do desenho de grafos, as seguintes entidades podem ser identificadas:

1. Configurações: cada configuração é uma atribuição válida de coordenadas da grade para todos os vértices do grafo. A configuração inicial pode ser escolhida aleatoriamente (por exemplo, todos os vértices em uma mesma coordenada ou espalhados na área de trabalho), ou pode ser fornecida pelo usuário.
2. Definição da vizinhança: a mudança de configuração deve representar apenas uma leve perturbação no estado do sistema. Assim, fica definida a vizinhança de uma configuração como qualquer outra vizinhança que difira daquela pelo reposicionamento de um vértice apenas. Outra idéia seria a troca de posição entre dois vértices quaisquer da configuração. Aqui consideramos, como regra de geração de um novo estado do sistema, um único movimento de um vértice para uma outra localização na grade dentro de um perímetro determinado. O vértice a ser movido e a direção a ser tomada são, em geral, determinados aleatoriamente.
3. Função objetivo: este é o ponto mais importante para a implementação do algoritmo SA, pois é através da função objetivo que os critérios de legibilidade devem ser expressos. Cada critério deve ser traduzido matematicamente como uma parcela contribuinte da função a ser minimizada. Além disso, para cada uma das parcelas é atribuído, ainda, um fator de normalização que define a importância de cada critério.
   - 3.1. Distribuição uniforme dos vértices: a primeira parcela da função objetivo trata da uniformização das distâncias entre vértices, não permitindo muita proximidade entre eles, além de fazer com que uma parte razoável da grade de desenho seja ocupada. Para cada par de vértices i e j, seja d_{ij} a distância euclidiana entre eles, e seja λ_1 o peso desse critério. A primeira parcela é então definida como sendo a soma dos inversos dos quadrados das distâncias entre cada par de vértices do grafo:

$$\lambda_1 \cdot \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{d_{ij}^2}$$

   - 3.2. Distância às bordas: a segunda parcela da função objetivo faz com que vértices não se aproximem muito das bordas do desenho. Ela é definida como sendo a soma dos inversos dos quadrados das distâncias de cada vértice às quatro linhas de borda do desenho. Sejam r_i, l_i, t_i e b_i as distâncias,

respectivamente, do vértice i às bordas direita, esquerda, superior e inferior, e λ_2 o peso para o critério considerado; assim, definimos a segunda componente da função objetivo:

$$\lambda_2 \cdot \sum_{i=1}^{n} \left(\frac{1}{r_i^2} + \frac{1}{l_i^2} + \frac{1}{t_i^2} + \frac{1}{b_i^2} \right)$$

3.3. Tamanho das arestas: para que as arestas do grafo não se tornem muito longas, a distância entre cada dois nós ligados por uma aresta no grafo também deve contribuir para a função objetivo. Seja d_k o comprimento da aresta k que liga dois vértices no grafo e λ_3 o peso dado a esse critério, então a terceira componente da função objetivo é definida:

$$\lambda_3 \cdot \sum_{k=1}^{m} d_k^2$$

3.4. Distância vértice-aresta: esse critério tenta penalizar pequenas distâncias entre um vértice e arestas que não lhe são incidentes. Tal distância é definida como a menor distância entre o vértice e um ponto qualquer sobre o segmento que representa a aresta, e contribui para a função objetivo com o inverso do seu quadrado. Desse modo, seja d_{kl} a distância euclidiana entre o vértice k e algum ponto sobre a aresta l, e λ_4 o peso atribuído a esse critério, acrescentamos à função objetivo a parcela:

$$\lambda_4 \cdot \sum_{k=1}^{n} \sum_{l=1 \mid l \in A(k)}^{m} \frac{1}{d_{kl}^2}$$

3.5. Cruzamentos: uma vez que, dada uma configuração, seja detectado um cruzamento, o valor da função objetivo pode ser aumentado de uma certa constante *cte*. O problema dessa componente é que uma mudança mínima na configuração pode alterar a função de um valor *cte*, diferente das outras parcelas que alteram continuamente o valor da função a cada mudança. Sendo λ_5 o peso critério, a última componente da função objetivo é:

$$\lambda_5 \cdot cte$$

4. Parâmetro de controle do processo: a variável que controla o processo de *Annealing* é chamada temperatura, assim como no processo de resfriamento de líquidos. Duas questões importantes quanto à temperatura são: qual deve ser o seu valor inicial e de quanto ela deve ser reduzida a cada conjunto de repetições.

5. Condição de término do processo: o processo pode ser encerrado de vários modos, seja através da fixação do número de iterações, que pode ou não depender do número de vértices do grafo, ou ainda através de uma condição combinada entre a temperatura corrente e o percentual de configurações aceitas entre uma mudança e outra de temperatura. Experiências podem justificar a escolha de uma ou outra alternativa.

A complexidade do algoritmo SA é dominada pelo cálculo da função objetivo. Consideraremos que esse cálculo é realizado a cada iteração apenas para o vértice v, que sofreu o deslocamento entre duas configurações consecutivas. Mesmo assim,

devemos lembrar de recalcular a função objetivo por inteiro dentro de um certo intervalo de iterações, para que seja evitado o acúmulo de erros nos cálculos da máquina.

O cálculo da distância entre v e todos os outros vértices tem complexidade $O(n)$, enquanto que sua distância às bordas é calculada em $O(1)$. No pior caso, se v estivesse ligado a todos os outros vértices do grafo, o tamanho de suas arestas seria calculado em $O(n)$.

As duas últimas parcelas correspondem aos cálculos mais pesados. Para o cálculo das distâncias vértice-aresta consideramos a distância de todos os outros vértices às arestas de v, somadas com as distâncias de v a todas as outras arestas que não lhe são incidentes. Esse cálculo é realizado, no pior caso, em $O(n^2+m)$. O cálculo do número de cruzamentos seria refeito com base nas arestas incidentes a v e todas as outras arestas do grafo, o que tomaria $O(n.m)$.

Resumindo, teríamos uma complexidade $O(n.m)$ para o cálculo da função objetivo a cada mudança de iteração. Convencionando que, para cada temperatura, há um número de iterações proporcional a n (por exemplo, $20.n$, $30.n$, etc.), a complexidade aumentaria consideravelmente. Resta ainda considerarmos o número de iterações pertencentes ao laço mais externo. Em **[DH89]**, esse valor é considerado uma constante, mas que nem sempre pode ser desprezada. Logo, a complexidade total torna o algoritmo não muito atrativo em alguns casos.

SA é reconhecidamente um método de convergência lenta. Uma execução do algoritmo SA pode ser impraticável em um sistema iterativo, como em um editor de diagramas, que manipulem estruturas com grande número de objetos. Mesmo assim, o método SA pode ter bons resultados, uma vez que não há o cálculo inteiro da função a cada iteração, e se for aplicado, por exemplo, a grafos esparsos.

A figura 5-1.b contém um exemplo da saída produzida pelo algoritmo implementado em **[DH89]**, a partir do grafo da figura 5-1.a.

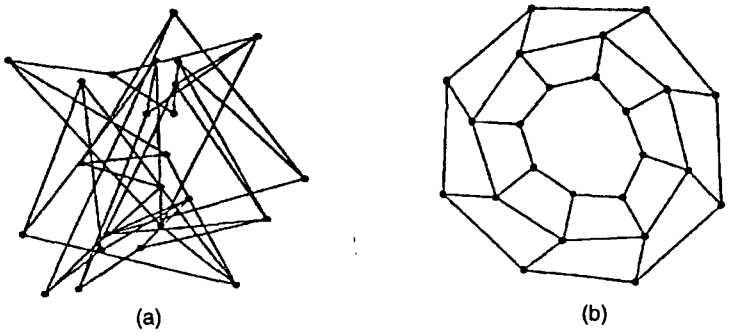


FIGURA 5-1 Exemplo de execução do algoritmo *Simulated Annealing* **[DH89]**.

Mais adiante veremos os resultados obtidos neste trabalho através de uma experiência com SA sobre os diagramas da LegoShell.

5.3 *Random Search*

RS - *Random Search* é um método de otimização cuja característica principal, que o diferencia dos outros métodos, é o armazenamento de s soluções correspondentes a mínimos/máximos locais, encontrados durante o processo de otimização de uma determinada função objetivo.

Uma solução é armazenada em uma estrutura auxiliar, junto com a configuração correspondente, quando equivale a um valor melhor que o pior dentre as soluções antes escolhidas. Esse processo causa a remoção aleatória de uma das s soluções já guardadas na estrutura. A intenção é que, no final do processo, tenhamos um conjunto de s soluções correspondentes a mínimos/máximos locais, dentre as quais esteja presente a solução ótima global.

Assim como em *Simulated Annealing*, algumas entidades devem ser identificadas para um processo de *Random Search*:

1. O conjunto de configurações ou estados do sistema.
2. Uma regra de geração de novas configurações, que define a vizinhança de cada configuração, geralmente escolhida aleatoriamente.
3. A função objetivo a ser minimizada sobre o espaço de configuração.
4. A condição de término do processo, que pode ser baseada nos valores da função objetivo.

Um esboço do algoritmo do processo de RS seria:

```
algoritmo RS {
  escolher uma configuração inicial σ;
  enquanto ( um novo estado for aceito ) fazer
    repetir ( rep ) vezes {
      escolher uma nova configuração σ' da vizinhança de σ;
      seja E' o valor da função objetivo relativo à σ' ;
      se ( E' < maior E pertencente à estrutura ) então {
        σ = σ' ;
        excluir uma das s soluções armazenadas na estrutura auxiliar;
        incluir solução correspondente a E';
      }
    }
}
```

Assim como no método SA, o cálculo da função objetivo é considerado o ponto crítico do algoritmo. Todas as observações relativas a esse cálculo, que aparecem na seção 5.2 são válidas na descrição do método RS.

5.3.1 *Random Search* para o desenho de grafos

O método RS, quando aplicado ao problema do desenho legível de grafos, trabalha apenas com as coordenadas (x,y) dos vértices, restando o trabalho do posicionamento das arestas no desenho final.

Aqui também podemos expressar quaisquer critérios de legibilidade enumerados no início do capítulo, além dos outros critérios sugeridos na seção dedicada ao método SA. As entidades identificadas também são idênticas àquelas relacionadas na seção anterior. As configurações inicial e subseqüentes são igualmente definidas, assim como a definição da vizinhança de uma configuração. A função objetivo será considerada a mesma para efeitos de comparação de resultados dos algoritmos SA e RS.

A principal diferença entre a definição das entidades dos métodos SA e RS está no parâmetro de controle do processo SA, a temperatura. No método SA, a decisão da aceitação ou não de uma nova configuração depende da última solução aceita, da variação do valor da função objetivo e da temperatura corrente. No método RS, a aceitação de uma nova configuração depende apenas dos valores anteriores da função objetivo armazenados na estrutura auxiliar.

A condição de término no método RS pode ser dada por um número fixo de iterações, e pode também depender da razão de aceitação de novas configurações, dentro de um determinado número de repetições.

A complexidade do algoritmo RS também é dominada pelo cálculo da função objetivo. O número de iterações do laço mais interno do algoritmo pode ser um múltiplo do número n de vértices do grafo (por exemplo, $20.n$, $30.n$, etc.).

No Capítulo 7, apresentaremos alguns resultados do método RS aplicado aos diagramas da LegoShell.

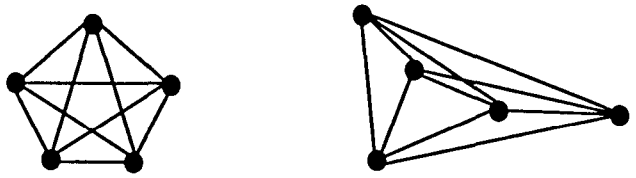
5.4 Desenho de estruturas em rede

Kamada e Kawai **[KK88]** propõem a formulação do problema do desenho de modelos estruturados em rede, representados sob a forma de um grafo, como um problema de minimização da energia de um sistema dinâmico virtual, em que cada dois vértices do grafo são conectados por uma mola de tamanho desejado. A energia total de tal sistema equivaleria ao grau de desequilíbrio do desenho e, quando o mínimo de energia no sistema fosse alcançado, estruturas simétricas seriam desenhados como figuras simétricas, além de grafos isomórficos serem desenhados de maneira idêntica. A idéia geral é relacionar a distância geométrica entre dois vértices no plano com sua distância no grafo. Uma idéia similar foi desenvolvida por Eades **[Ead84]**, também para o desenho de grafos gerais não-orientados.

O algoritmo desenvolvido não considera restrições para o posicionamento dos vértice e arestas são desenhadas seguindo o padrão de linha-reta. Desse modo, o problema a ser resolvido se resume em, dado um grafo G conexo, determinar as posições dos vértices na grade de desenho.

A legibilidade do desenho é relacionada ao balanço total do desenho da estrutura, ou seja, a simetria é mais desejável até que a diminuição do número de cruzamentos por exemplo. Considerando essa definição de legibilidade, o desenho do grafo K_5

que aparece na figura 5-2.a seria melhor que o desenho da figura 5-2.b, que possui um número mínimo de cruzamentos.



(a) (b)

FIGURA 5-2 Desenho do grafo K_5 de acordo com os critérios de simetria e de número de cruzamentos.

Sejam p_1, p_2, ..., p_n as posições no plano dos vértices v_1, v_2, ..., v_n do grafo G. Seja d_{ij} o menor caminho no grafo entre v_i e v_j e K uma constante; k_{ij} é a força da mola entre v_i e v_j, que é definida como sendo:

$$k_{ij} = K/d_{ij}^2$$

Seja $|p_i - p_j|$ a distância euclidiana entre os vértices v_i e v_j no plano e L uma constante, que pode ser definida como o quociente da altura da grade de desenho pelo máximo dos d_{ij}. Definimos l_{ij} como o tamanho desejado da mola entre v_i e v_j no plano da seguinte forma:

$$l_{ij} = L \times d_{ij}$$

Assim, o grau de desequilíbrio dos vértices, ou seja, a energia total das molas que os conectam é definido como sendo:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (|p_i - p_j| - l_{ij})^2$$

A função energia E pode ser interpretada como a soma dos quadrados das diferenças entre as distâncias desejadas e as reais para cada par de vértices.

Uma vez que a função energia esteja definida, desenhos legíveis segundo o critério definido anteriormente podem ser obtidos com a minimização de E, durante o processo de deslocamento dos vértices na grade de desenho.

No algoritmo de Kamada e Kawai, aqui denominado Spring, a minimização da energia global é conseguida com a minimização local da energia em cada estado do sistema, e a condição necessária para que esse mínimo local seja alcançado é que as derivadas parciais da função energia naquele estado sejam iguais a zero, ou seja:

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_i} = 0, \text{ para } (1 \le i \le n)$$

Um estado que satisfaça a expressão acima corresponde ao estado dinâmico em que as forças das molas estão balanceadas. Para que o mínimo local definido acima seja obtido, um vértice v_m é escolhido e movido para uma posição p_m, tal que a energia local seja minimizada a cada um desses movimentos. O vértice a ser mo-

vido é aquele que contribui com a energia mais alta, traduzida como o maior valor de Δ_m, que é definido abaixo:

$$\Delta_m = \sqrt{(\frac{\partial E}{\partial x_m})^2 + (\frac{\partial E}{\partial y_m})^2}$$

A nova posição $p_m=(x_m, y_m)$ do vértice selecionado v_m é conseguida a partir de um deslocamento calculado $\sigma_m=(\sigma_x, \sigma_y)$, tal que no novo estado:

$$\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0$$

Sendo ε a precisão de cálculo do mínimo local e Δ_i a medida de energia do vértice *i*, um esboço do algoritmo é apresentado a seguir.

```
algoritmo SPRING {
  calcular d_ij, para 1 ≤ i < j ≤ n;
  calcular l_ij, para 1 ≤ i < j ≤ n;
  calcular k_ij, para 1 ≤ i < j ≤ n;
  atribuir coordenadas iniciais a p_1, p_2, ..., p_n;
  enquanto ( max Δ_i > ε ) fazer {
    seja v_m o vértice com Δ_m = max Δ_i;
    enquanto ( Δ_m > ε ) fazer
      deslocar v_m de σ_m;
  }
}
```

O algoritmo é executado até que nenhum movimento de vértice seja necessário para a minimização global da função energia. Como a função objetivo é contínua, um método de otimização originário da análise numérica pode ser utilizado durante sua minimização, em vez de *Simulated Annealing*. Em **[KK88]**, o método Newton-Raphson de duas dimensões é aplicado no cálculo das equações das derivadas parciais do vértice a ser movido.

A figura 5-3.b apresenta o estado final do grafo da figura 5-3.a depois de executado o algoritmo SPRING em **[KK88]**.

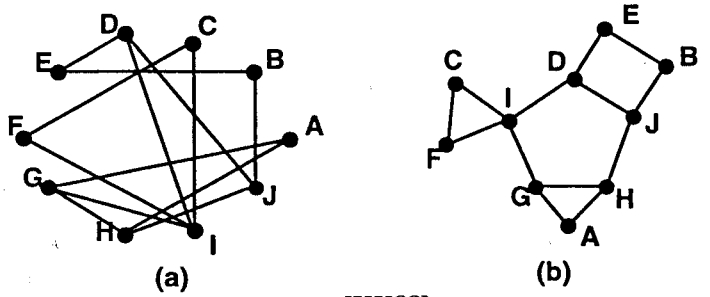


FIGURA 5-3 Exemplo de execução do algoritmo Spring **[KK88]**.

Capítulo 6

Método geral de desenho de grafos

6.1 Introdução

Em [TBB88] e [BNT+84] é descrito um método geral para desenho de diagramas de sistemas de informação baseado em teoria dos grafos. Os diagramas são desenhados de acordo com o padrão gráfico de grade, e suas entidades, vértices do grafo implícito, são representadas por retângulos ou símbolos imersos em um retângulo. O algoritmo propõe-se a tratar um largo espectro de critérios de legibilidade, aceitando, ainda, restrições ao desenho como entrada adicional.

A estratégia básica do algoritmo é construir incrementalmente o desenho do diagrama, de acordo com três propriedades: topologia, formato e métrica. Os critérios de estética considerados pelo algoritmo são enumerados a seguir [BFN85]:

1. minimização de cruzamentos entre arestas;
2. minimização do número global de quebras ao longo das arestas;
3. minimização do tamanho global das arestas; e
4. minimização da área do retângulo que cobre o diagrama.

Como é fácil perceber, existem conflitos entre os critérios, o que sugere o estabelecimento de prioridades entre os mesmos. O modo encontrado para a resolução desse impasse foi a associação de cada um dos critérios a uma das propriedades do diagrama, tal que elas fossem tratadas hierarquicamente. O primeiro critério, por exemplo, foi relacionado à propriedade topológica do desenho; o segundo ao seu formato, que é determinado em função dos ângulos que aparecem ao longo das arestas, e os dois últimos foram relacionados à métrica do desenho, ou seja, às suas dimensões. Cada uma das propriedades fornece uma descrição do desenho que é um refinamento da anterior. Assim, dois grafos com a mesma métrica possuem o mesmo formato, e dois grafos com o mesmo formato têm a mesma topologia.

Uma vez que o algoritmo considera sucessivamente as propriedades acima, um esquema hierárquico de construção do desenho pode ser montado. Conforme mostra a figura 6-1, cada representação é produto da fase do algoritmo que trata de uma das propriedades citadas.

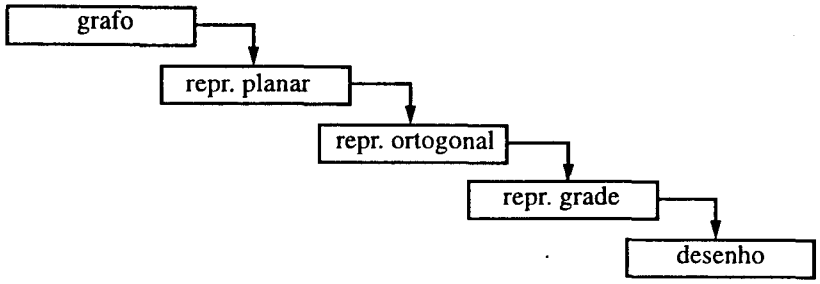


FIGURA 6-1 Representação hierárquica para o desenho de diagramas.

A representação planar descreve as adjacências entre as faces do desenho de um grafo planar. A representação ortogonal descreve apenas o formato do desenho de um grafo ortogonal[1], sem se preocupar com suas dimensões. Por último, a representação em grade contém informações a respeito das dimensões do desenho de um grafo em grade[2].

Essas representações são geradas, respectivamente, pelas fases de planarização, ortogonalização e compactação do algoritmo. Cada fase é baseada em um ou mais algoritmos anteriormente projetados, não exclusivamente para fins do desenho de diagramas, mas que foram adaptados para tais propósitos.

Esse algoritmo sofreu adaptações para diagramas de fluxo de dados **[BNT86]** e diagramas de entidade e relacionamento **[BNT84] [BBd+86] [BNT+85] [Tam85]**, que se comportam de maneira similar.

6.2 Fases do método

Antes da execução da primeira fase do método, o esquema do diagrama, representado sob forma textual, é transformado em um grafo abstrato não-orientado. Assim, as fases de planarização, ortogonalização e compactação são realizadas, conforme descreveremos a seguir.

6.2.1 Planarização

Os passos básicos da fase de planarização são a extração de subgrafos planares do grafo, e a recolocação de arestas que causem sua não-planaridade, se elas existirem.

1. um grafo ortogonal é um grafo planar cujas arestas são seqüências de segmentos horizontais e verticais alternados.

2. um grafo em grade é um grafo ortogonal cujos vértices são pontos com coordenadas inteiras, e cujas arestas correm ao longo de caminhos (sem interseção de vértices) na grade implícita do sistema de coordenadas.

Primeiramente, há um preprocessamento no grafo que trata das restrições impostas ao desenho pelo usuário, como a localização específica ou agrupamento de determinados vértices.

Por exemplo, para que um conjunto de vértices seja posicionado na fronteira externa do desenho, um vértice fantasma x é adicionado ao grafo e conectado a todos os vértices pertencentes àquele conjunto. O objetivo é que, após o término da fase de planarização, esses vértices pertençam à mesma face. Depois disso, o vértice x e todas as suas arestas incidentes são removidos, e aquela face é escolhida como face externa.

Super-vértices são criados a partir da contração de alguns vértices, no caso de agrupamento de um grupo deles, ou se o formato de um subgrafo for predeterminado.

O núcleo da fase de planarização é assim descrito:

```
algoritmo Planarização(G, P) {

 Passo 1: Extrair_Subgrafos_Planares(G,G*) {
   Decompor G em componentes biconexas;
   Para cada Bloco B {
     Planarizar_Bloco(B, B*);
     G* = G* ∪ B*;
   }
 }

 Passo 2: Adicionar_Cruzamentos(G*,P) {
   Encontrar uma representação planar P* de G*;
   Para cada aresta não-planar "e"
     Reintroduzir "e" minimizando o número de cruzamentos;
   P = Resultado da representação planar;
 }
}
```

A decomposição do grafo em componentes biconexas é realizada através do algoritmo apresentado por Tarjan em **[Tar72]**, que utiliza a técnica de busca em profundidade (vide seção 1.4). A planarização dos blocos é baseada no algoritmo para teste de planaridade de Hopcroft e Tarjan **[HT74]** (também descrito anteriormente na seção 3.1). A modificação foi feita no ponto em que o algoritmo pára sua execução em virtude da descoberta de uma aresta causadora da não-planaridade do grafo. Nesse momento, a execução continuaria, guardando-se arestas não-planares para posterior tentativa de reinserção **[NT84]**. Uma vez que a reinserção de tais arestas causa a introdução de um ou mais cruzamentos, tais cruzamentos são substituídos por vértices fictícios para que a representação do grafo continue planar para a próxima fase.

No segundo passo, encontrar uma representação planar para G^* significa principalmente evitar o aninhamento de faces (vide figura 6-2), para que os dois últimos critérios citados na introdução sejam mais facilmente obtidos.

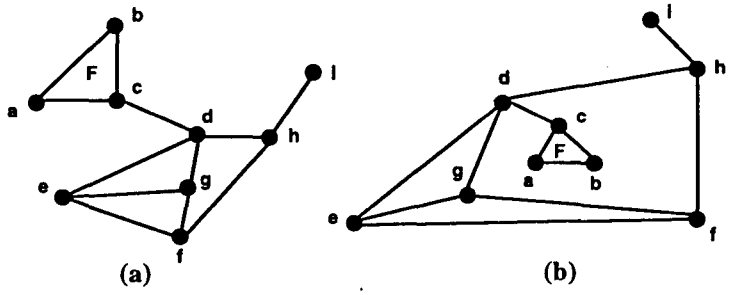


FIGURA 6-2 Duas representações planares de um mesmo grafo, onde (b) apresenta uma face F aninhada a mais que (a).

Seja $e = (v, w)$ uma aresta causadora de não-plaridade a ser reintroduzida. Seja (F,A) o grafo dual da representação planar corrente, onde F é o conjunto de faces e A é o conjunto de arcos que ligam faces adjacentes. Finalmente, seja $G_e = (F+\{v,w\}, A+B)$ um novo grafo, onde B é o conjunto das arestas que ligam v e w às faces que lhe são incidentes. O procedimento de reintrodução de e é o seguinte: são calculados os caminhos mínimos entre v e w no grafo G_e. Dentre esses caminhos, um caminho c é escolhido, e a aresta e introduzida em seu lugar. Um vértice fictício é introduzido onde um cruzamento envolvendo uma das arestas de c não puder ser evitado. A figura 6-3 representa tal situação.

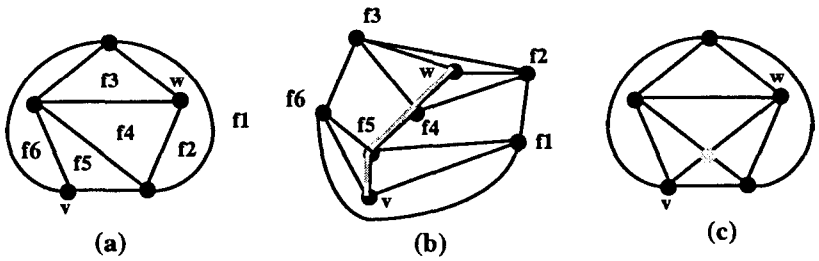


FIGURA 6-3 (a) Representação planar corrente, onde a aresta não-planar $e=(v,w)$ deve ser reintroduzida. (b) O grafo G_e, com o caminho c em cinza. (c) Representação planar depois da introdução da aresta e, com o vértice fictício em cinza.

6.2.2 Ortogonalização

A fase de ortogonalização recebe a representação planar advinda da fase anterior e produz uma representação ortogonal com um pequeno número de quebras. Esta fase pode ser executada de dois modos, um denominado ortogonalização rápida, e o outro ortogonalização lenta.

Uma descrição mais detalhada da ortogonalização rápida pode ser encontrada em **[TT89]**, cujo algoritmo foi exposto na seção 3.4. Um exemplo da execução de suas sub-fases pode ser visto na figura 6-4. Em primeiro lugar, uma representação de visibilidade (vide seção 3.3) é construída para o grafo (vide figura 6-4.b), que é então transformada em uma imersão em grade através de expansões locais (vide

figura 6-4.c) e, finalmente, são aplicadas transformações de estiramento de quebras (vide figuras 6-4.d e 6-4.e).

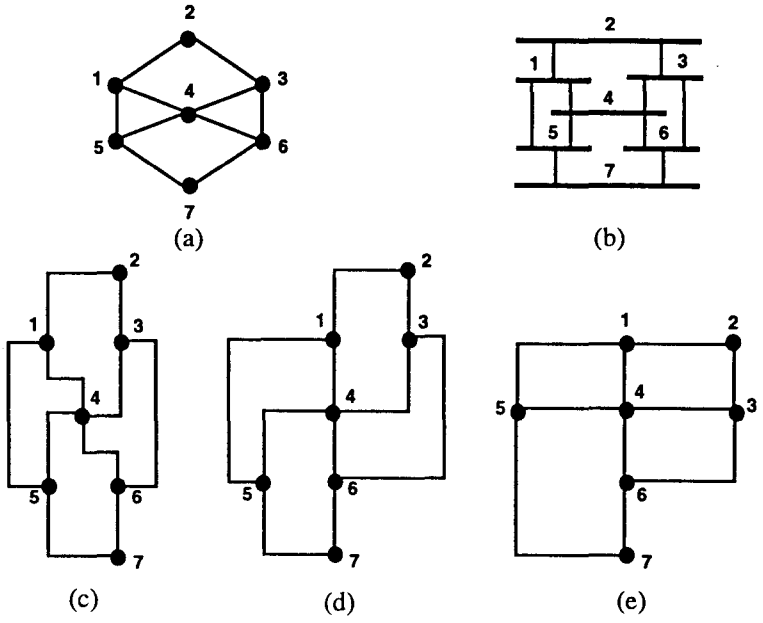


FIGURA 6-4 Um exemplo de ortogonalização rápida.

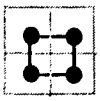
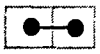
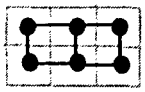
A ortogonalização rápida possui complexidade linear no número de vértices em tempo e, mesmo que o número mínimo de quebras não seja garantido, bons resultados podem ser conseguidos na prática.

A ortogonalização lenta possui complexidade $O(n^2 \log n)$, mas garante um número mínimo de quebras, além de considerar uma maior quantidade de critérios de legibilidade. Em **[Tam87]**, é descrito o algoritmo responsável por essa versão da ortogonalização, utilizando técnicas de fluxo em redes. A idéia básica do algoritmo é a transformação do problema de imersão de um grafo em uma grade retilínea em um problema de fluxo de custo mínimo.

No primeiro passo da fase de ortogonalização lenta são isolados subgrafos hierárquicos com relação a conceitos inerentes a entidades do diagrama, como por exemplo, duas ou mais entidades podem estar relacionadas hierarquicamente a uma terceira, que possivelmente é uma abstração generalizada daquelas. Esses subgrafos hierárquicos são desenhados independentemente considerando critérios relativos a desenhos de hierarquias. O formato de cada subgrafo é, então, mantido fixo até o final da fase de ortogonalização.

No próximo passo dessa mesma fase, denominado normalização, vértices de grau maior que 4 são transformados em novos vértices de grau menor ou igual a 4. A distribuição das arestas incidentes em tais vértices pode ser feita de dois modos. O mais simples deles é transformar cada um desses vértices em um ciclo de l novos

vértices. l é um número de células da grade de desenho, que depende do grau do vértice. Podemos observar a relação entre o grau do vértice e l na tabela abaixo.

Grau	l	Grau	l
1-4		7-8	
5-6		9-10	

Uma vez realizada esta transformação, as arestas podem ser atribuídas seqüencialmente a cada um dos vértices do ciclo de acordo com a ordem circular existente na representação planar do grafo. A figura 6-5 apresenta um exemplo dessa forma de normalização, onde o vértice v de grau 5 é transformado em um ciclo de dois novos vértices.

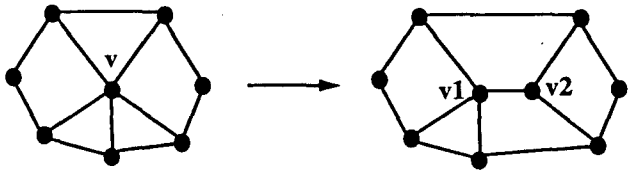


FIGURA 6-5 Transformação de um grafo através de normalização.

Uma estratégia mais sofisticada consiste em encontrar uma atribuição ótima de arestas para cada vértice através de um algoritmo baseado no conceito de representação k-gonal. Uma representação k-gonal descreve um grafo planar cujas arestas são seqüências de segmentos cujos ângulos são múltiplos de $180^o/k$ com relação ao eixo x. Quanto maior for o valor de k, mais arestas poderão ser atribuídas a um determinado lado do retângulo. Um grafo 2-gonal, por exemplo, corresponde a um grafo ortogonal, que suporta apenas uma aresta fixa em cada lado do retângulo.

Uma vez atribuídas as arestas a cada lado do retângulo que representa o vértice, o formato de cada retângulo é mantido fixo no decorrer do algoritmo. Ainda em **[Tam87]** é descrito um algoritmo para o cálculo de uma representação k-gonal ótima com complexidade $O(k^2n^2)$, onde n é o número de vértices da representação planar.

Na tradução do problema de minimização do número de quebras em um problema de fluxo em rede, a função objetivo é definida pelo número de ângulos iguais a 90^o encontrados na extensão de cada aresta. Os nós da rede correspondem aos vértices e faces do grafo original. Os arcos da rede correspondem a faces vizinhas e a cada vértice componente de cada face. A conservação do fluxo em um nó, que representa um vértice, significa que a soma dos ângulos formados pelas arestas incidentes a ele deve ser igual a 360^o, e a conservação do fluxo em um nó que representa uma face significa que esta face deve ter um formato retilíneo.

6.2.3 Compactação

A fase de compactação é a responsável pela atribuição de dimensão à representação ortogonal, resultado da fase anterior. Para que os tamanhos dos segmentos sejam calculados, algumas restrições devem ser satisfeitas, como por exemplo, todos os segmentos devem ser inteiros positivos, e todo circuito do grafo deve ser mapeado em um polígono retilíneo.

Esta fase também possui duas versões: uma rápida, cuja complexidade de tempo é linear na soma do número de vértices, cruzamentos e quebras da representação ortogonal, e uma lenta, também baseada na técnica de fluxo em redes.

O algoritmo para a compactação rápida é baseado em compactação de circuitos integrados. Em primeiro lugar, cada face da representação ortogonal é decomposta em retângulos. Em seguida, dois grafos orientados G_v e G_h são construídos representando as restrições impostas pela representação ortogonal.

Esses grafos são construídos de tal modo que os tamanhos dos caminhos máximos em G_v e G_h são iguais à altura e largura mínimas do desenho, respectivamente. Finalmente, os tamanhos das arestas são computados aplicando-se o método do caminho crítico a G_v e G_h **[Eve79]**.

O algoritmo para compactação lenta tem complexidade quadrática e usa a técnica de fluxo em redes sobre a representação ortogonal já decomposta em retângulos. Cada unidade de fluxo representa uma unidade de tamanho. De fato, são utilizadas duas redes, uma para as dimensões horizontais e a outra para as dimensões verticais. Os nós das redes são os retângulos (ou faces), e a conservação do fluxo em cada nó significa que lados opostos do mesmo retângulo devem ter o mesmo tamanho.

A atribuição de dimensões aos segmentos pode ser expressa por meio de programação linear inteira, onde as variáveis são os tamanhos dos segmentos pertencentes às duas redes separadamente. A minimização do custo de fluxo nas redes corresponde à minimização do tamanho dos segmentos (ou arestas) em ambas as direções.

Depois que a representação em grade é encontrada, o diagrama pode ser desenhado facilmente por uma rotina gráfica, tendo-se o cuidado de remover todos os vértices fictícios introduzidos nas fases de planarização e normalização, por exemplo, e de expandir os super-vértices contraídos anteriormente.

Capítulo 7

Implementação e Resultados

7.1 Introdução

Uma parte dos esforços realizados neste trabalho foi dedicada à implementação de alguns dos algoritmos descritos em capítulos anteriores. Essa decisão foi tomada para que analisássemos a viabilidade de incorporação de algum(ns) deles ao editor da *LegoShell* futuramente.

Neste capítulo, veremos uma descrição da atual interface gráfica da *LegoShell* que servirá de *front-end* para a execução dos algoritmos. Relacionaremos, também, os algoritmos escolhidos para experimentação, junto com alguns resultados obtidos.

Antes da utilização do atual editor da *LegoShell*, foi desenvolvido um protótipo do gerador automático de desenhos de diagramas, onde pôde ser observado o comportamento dos algoritmos sobre diagramas que não tinham a semântica nem a aparência das computações *LegoShell*.

Naquela interface, também desenvolvida sobre o sistema X **[SCO91]**, os objetos não possuíam portas visíveis, e sim um centro geométrico de onde emergiam todas as suas conexões. Agora que as portas dos objetos também são objetos visíveis, a partir de onde as conexões são desenhadas, seu posicionamento também influencia na legibilidade do desenho final. Desse modo, uma reordenação no conjunto de portas de um objeto deve ser realizada, a cada execução dos algoritmos. Esse trabalho foi desenvolvido em **[Ces94]**.

Em **[KMN89]**, é apresentada uma linguagem gráfica também projetada para a representação de objetos distribuídos, mas nenhum algoritmo para o desenho legível de seus diagramas é adotado. Mesmo assim, algumas heurísticas simples são utilizadas para que o desenho não seja tão confuso. Uma delas implementa uma idéia similar à nossa, a do desenho de portas de um objeto de tal maneira que as conexões se tornem as mais curtas possível.

A nova interface da *LegoShell*, atual hospedeira dos algoritmos de geração automática de desenhos é apresentada a seguir.

7.2 O editor da *LegoShell*

O novo editor gráfico da *LegoShell* é um trabalho desenvolvido por Wanderley Ceschim como parte de projeto de iniciação científica **[Ces94]**. A interface está sendo desenvolvida na linguagem C++ **[Str91]**, em ambiente Unix **[Bac86]** e sobre o sistema de janelas X **[SCO91]**.

Seu desenvolvimento teve início no segundo semestre de 1993, tendo como metas a modularidade, a extensiblidade e a facilidade de uso **[Ces94]**. Parte desses objetivos tem sido alcançada através da adoção de um projeto orientado a objetos. O editor é, na verdade, um conjunto de classes organizadas em hierarquias que implementam todas as funções básicas da *LegoShell*.

Toda a interface é construída com o auxílio do *toolkit* Astra **[Fur93]**, desenvolvido dentro do próprio projeto A_HAND. Astra implementa objetos interativos básicos para esse propósito, como janelas, diálogos, botões, menus de opções, campos texto, *canvases*, etc. Maiores detalhes sobre Astra podem ser encontrados em **[Fur93]**.

A interface da *LegoShell*, do ponto de vista do usuário é composta de uma janela principal, que contém um menu de opções, uma barra de ferramentas e uma área de trabalho, onde são editadas as computações. A figura 7-1 mostra a aparência atual do editor.

A barra de menu consiste de um conjunto de botões que permitem o acesso a listas de opções. Os botões são os seguintes:

1. File: fornece ao usuário a capacidade de gravar/editar computações em/advindas de arquivos. A função LOAD instancia um diagrama *LegoShell*, a partir de declarações textuais de seus objetos e conexões. Essa capacidade é fornecida com a ajuda dos geradores de analisadores léxico e sintático *Flex* e *Bison*, da *Gnu Corporation*. A sintaxe da linguagem de definição dos diagramas encontra-se no Apêndice A, junto com um trecho da descrição do diagrama da figura 1-3. A função SAVE grava em um arquivo, de formato ASCII, a configuração corrente do diagrama, durante uma travessia pelas listas de objetos e co-nexões do mesmo.
2. View: permite que o usuário defina o modo de visualização da computação que está sendo editada, como por exemplo, através de uma grade de posicionamento que força objetos a localizações predeterminadas.

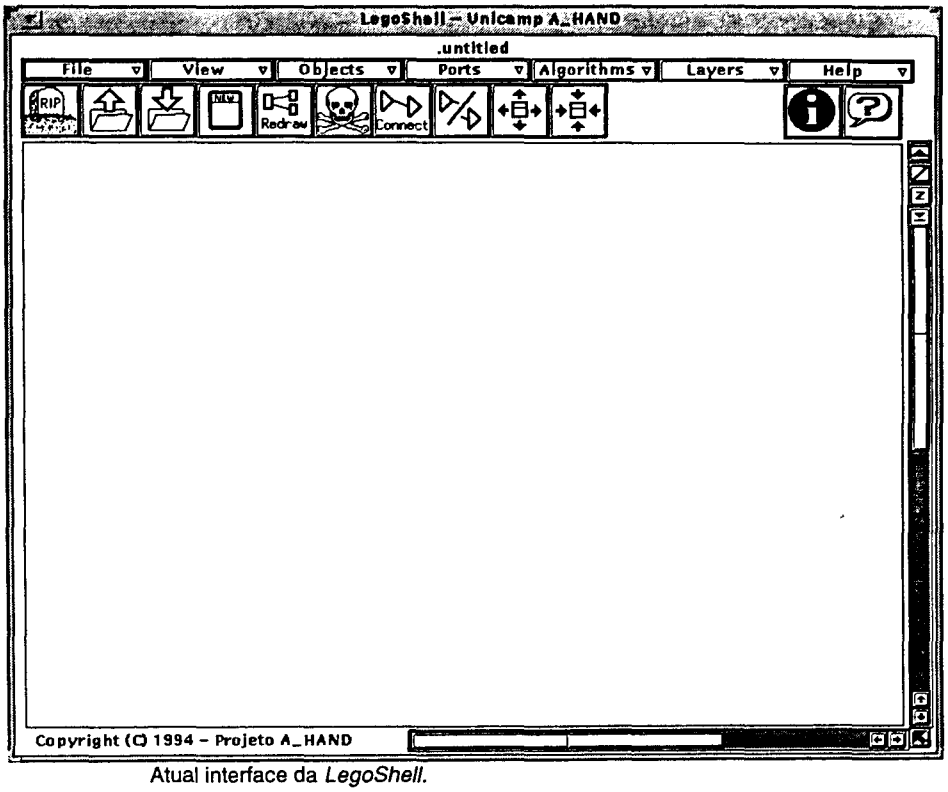


FIGURA 7-1 Atual interface da *LegoShell*.

3. Objects: permite ao usuário controlar e editar características dos objetos das computações. A figura 7-2 mostra o diálogo de criação e edição dos parâmetros dos objetos *LegoShell*.

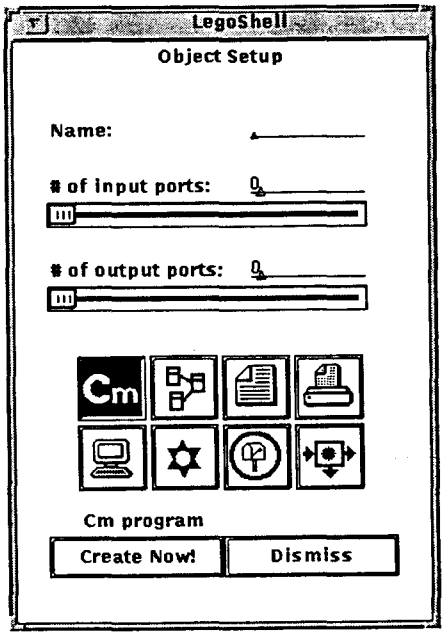


FIGURA 7-2 Diálogo de edição dos objetos *LegoShell*.

4. Ports: permite ao usuário controlar e editar características das portas dos objetos.
5. Algorithms: fornece a lista de programas disponíveis para o redesenho automático dos diagramas *LegoShell*.
   5.1. Animation: liga/desliga o modo de animação durante o redesenho do diagrama. Quando ligado torna bem mais lento os processos de *Simulated Annealing* e *Random Search*, mas é uma boa ferramenta de acompanhamento dos mesmos.
   5.2. Annealing: invoca a janela de configuração (vide figura 7-3) dos parâmetros para o programa que implementa o algoritmo de *Simulated Annealing* para o desenho de grafos.

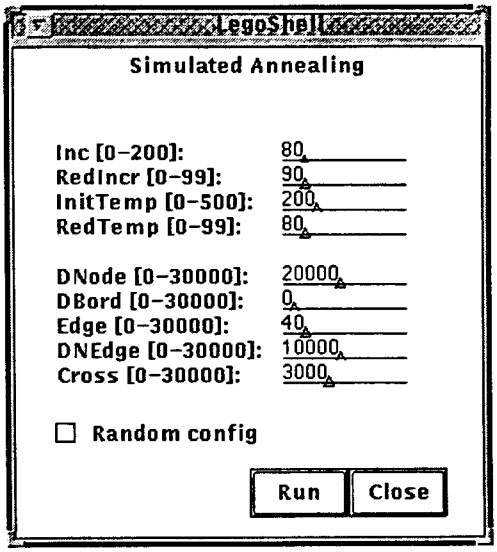


FIGURA 7-3 Diálogo de edição dos parâmetros do processo de *Simulated Annealing*.

Cada um dos parâmetros tem o seguinte significado, intervalo de valores e unidade:

5.2.1. Incr: movimento inicial de um objeto para obtenção de um nova configuração do sistema.
Intervalo: 0-200 (*pixels*).

5.2.2. RedIncr: fator de redução do movimento de um objeto em um determinado intervalo de iterações.
Intervalo: 0-99 (%).

5.2.3. InitTemp: temperatura inicial do processo de *Annealing*.
Intervalo: 0-500 (°).

5.2.4. RedTemp: fator de redução da temperatura.
Intervalo: 0-99 (%).

5.2.5. DNode: fator de normalização do critério "Distância entre nós".
Intervalo: 0-30.000 ( ).

5.2.6. DBord: fator de normalização do critério "Distância de um nó às bordas".
Intervalo: 0-30.000 ( ).

5.2.7. Edge: fator de normalização do critério "Tamanho das arestas".
Intervalo: 0-30.000 ( ).

5.2.8. DNEdge: fator de normalização do critério "Distância nó-aresta".
Intervalo: 0-30.000 ( ).

5.2.9. Cross: fator de normalização do critério "Minimização do número de cruzamentos entre conexões".
Intervalo: 0-30.000 ( ).

5.2.10. RandomConfig: liga/desliga configuração inicial randômica. Se desligado, a configuração inicial considerada pelo programa de *Annealing* é a configuração fornecida pelo usuário.

5.2.11. Run: executa o programa de *Annealing* utilizando os valores correntes dos parâmetros acima.

5.2.12. Close: fecha a janela de diálogo.

5.3. Random Search: invoca a janela de configuração (vide figura 7-4) dos parâmetros para o programa que implementa o algoritmo de *Random Search* para o desenho de grafos.

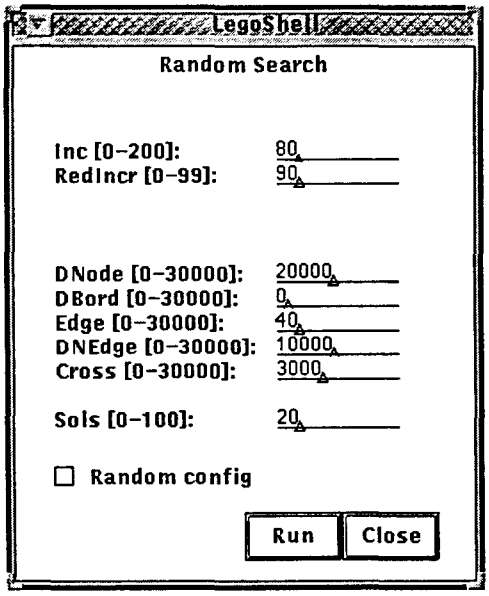


FIGURA 7-4 Diálogo de edição dos parâmetros do processo de *Random Search*.

Cada um dos parâmetros tem o mesmo significado, intervalo de valores e unidade dos parâmetros do processo de *Simulated Annealing* . Os parâmetros relativos a temperatura, InitTemp e RedTemp, não fazem sentido para esse algoritmo, por isso são excluídos. Um parâmentro a mais é consi-derado:

5.3.1. Sols: Número corrente de configurações que devem ser mantidas durante o processo de *Random Search*.
Intervalo: 0-100( ).

6. Layers: provê acesso a um sistema de classificação de computações em camadas, onde objetos são inseridos e removidos de camadas criadas e nomeadas pelo usuário. Essa facilidade pode ser útil em computações muito complexas, onde o usuário deseja classificar um subconjunto de objetos que mereçam uma atenção especial ou tenham características e/ou funções semelhantes.

7. Help: fornece informações sobre a operação da interface, com respeito aos seus elementos interativos.

A barra de ferramentas é composta por um conjunto de botões posicionados horizontalmente sob a barra de menu, que representam atalhos para as opções mais freqüentemente utilizadas dos menus.

A área de trabalho é uma região retangular na janela principal do editor da *LegoShell*, onde o usuário visualiza e edita computações. A área de trabalho possui tamanho máximo fixo, e é acompanhada de duas barras de deslocamento para que o usuário possa selecionar uma subárea visível caso o tamanho real da janela do editor não seja suficiente para mostrar o espaço de trabalho em sua totalidade. A coordenada (0,0) para os algoritmos de redesenho corresponde ao canto superior esquerdo da área de trabalho.

Funcionalmente, o editor da *LegoShell* pode ser dividido em cinco módulos principais:

1. Gerenciador de diálogo: controla aspectos de interação com o usuário através da interface gráfica provida pelo editor. A disposição de vários objetos de interação (*widgets*) segue o modelo WIMP (*Window, Icon, Menu, Pointer*).

2. Gerenciador de visual: controla a edição e o desenho atualizado dos objetos e conexões presentes na área de trabalho.

3. Gerenciador de eventos: cuida da manipulação direta do usuário sobre os objetos disponíveis na computação corrente.

4. Gerenciador de objetos: gerencia os objetos que representam internamente uma computação *LegoShell*. Gerencia os algoritmos para o desenho automático dos diagramas, que também são objetos do editor.

5. Repositório: interface principal entre o editor e o sistema de arquivos. Possui as funções de gravar e recuperar do disco as computações *LegoShell*.

Os gerenciadores de eventos e de objetos são clientes dos serviços prestados pelo *toolkit* Astra. O gerenciador de visual, no entanto, não utiliza *widgets* Astra para reproduzir, na tela, a computação que está sendo edi-tada. Isto é feito utilizando-se diretamente funções de **XLib**, a biblioteca básica do sistema X.

A figura 7-5 estabelece o relacionamento entre os módulos *LegoShell*:

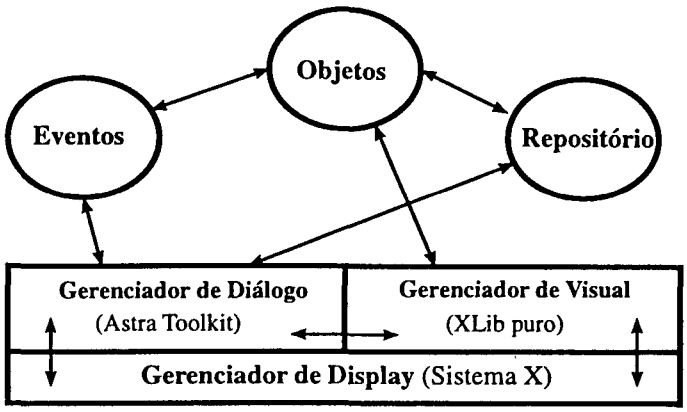


FIGURA 7-5 Módulos principais do editor da *LegoShell*.

Como foi dito antes, o editor da *LegoShell* é um conjunto de classes em C++ que definem, desde um objeto manipulável na área de trabalho, até os tratadores de eventos gerados pelo usuário e interceptados pelo *toolkit* Astra. Os algoritmos de redesenho também são implementados como classes em C++, cujos métodos são invocados a cada chamada de execução na interface.

7.3 Algoritmos Implementados

Os principais fatores que influenciaram na escolha dos algoritmos a serem experimentados neste trabalho foram:

1. As classes de grafos nas quais um diagrama *LegoShell* poderia se enquadrar; e
2. Os critérios de estética mais importantes a serem considerados no desenho de um diagrama *LegoShell*.

O padrão de desenho de linha-reta foi adotado por tornar mais simples o desenho final das conexões, todavia o padrão grade de desenho não foi esquecido completamente, e deve ser reconsiderado em trabalhos futuros.

As computações *LegoShell* são representadas internamente como um grafo orientado. Contudo, para efeitos de representação gráfica, os diagramas que representam essas computações podem ser vistos como grafos gerais, orientados ou não.

Dessa forma, nossa atenção foi mais direcionada aos algoritmos projetados para o desenho de grafos gerais, orientados ou não. Duas técnicas de otimização foram utilizadas para a resolução do problema de desenho legível dos diagramas, considerando sua representação interna como a de um grafo geral não-orientado: *Simulated Annealing* e *Random Search* (vide Capítulo 5).

Observando aspectos do fluxo de informação (dados e serviços) em um diagrama *LegoShell*, resolvemos considerar a orientação das conexões do diagrama. Assim, a heurística do baricentro (vide Capítulo 4), utilizada para a redução de cruzamentos em grafos acíclicos orientados, também foi explorada.

Os critérios de legibilidade julgados mais importantes foram:

- distribuição uniforme dos objetos na área de desenho;
- não sobreposição de conexões a objetos;
- minimização do número de cruzamentos entre conexões; e
- minimização do tamanho das conexões.

Detalhes de implementação e resultados de cada uma dos algoritmos serão vistos nas próximas seções.

7.3.1 *Simulated Annealing* e *Random Search*

Esta seção é dedicada aos algoritmos de *Simulated Annealing* - SA e *Random Search* - RS simultaneamente, uma vez que os dois algoritmos tentam minimizar a mesma função objetivo, e trabalham dentro de um mesmo espaço de configuração. Quaisquer pontos diferentes serão identificados separadamente. Antes de comentar os resultados obtidos, devemos definir as entidades identificadas na seção 5.2.1 para ambos os processos.

Nas configurações válidas dentro dos processos SA e RS, os objetos podem ocupar quaisquer posições dentro dos limites da área de desenho. É fornecida a possibilidade da configuração inicial aleatória para os objetos, o que causa uma distribuição inicial dos objetos que destrói a configuração inicial fornecida pelo usuário. O uso dessa capacidade não é muito indicada quando o objetivo final é o redesenho de diagramas recém-projetados pelo usuário, mas foi fornecida para efeitos de verificação do comportamento da função objetivo durante o processo de otimização.

A vizinhança de uma configuração deve ser uma que se diferencie da anterior pelo movimento de um objeto apenas. Este objeto é escolhido aleatoriamente dentre todos os que compõem a computação, e a direção a ser tomada é escolhida também aleatoriamente, dentre uma das oito que aparecem na figura 7-6.

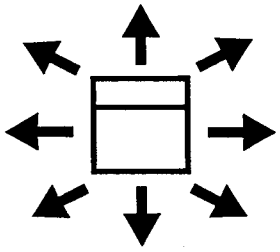


FIGURA 7-6 Direções permitidas no movimento de um vértice.

O raio do movimento *incr* é medido em *pixels*, podendo ser iniciado pelo usuário. No caso de o valor inicial de *incr* ser muito alto, uma boa configuração pode ser perdida facilmente. Por outro lado, para que o sistema saia de uma configuração ruim, um grande deslocamento do objeto pode ser necessário.

A conclusão do quão grande ou pequeno deve ser o movimento de um vértice depende, principalmente, da configuração inicial. É difícil precisar um único valor de *incr* para qualquer que seja o sistema.

Depois de cada execução do laço mais interno em ambos os algoritmos, o raio do movimento pode ser decrescido de um certo fator β. A função do fator de redução de movimento β é fazer com que movimentos de objetos sejam maiores no início do processo de otimização. Assim, objetos podem ter a chance de fugir mais facilmente de uma configuração inicial ruim. No final, movimentos menores seriam realizados com o intuito apenas de me-lhorar as posições antes obtidas, sem grandes alterações no valor da função objetivo. O fator β também poder ser modificado pelo usuário.

A função objetivo é calculada da mesma forma para ambos os processos. Cada uma das parcelas utiliza as coordenadas reais (x,y) de cada objeto do diagrama, onde (x,y) corresponde ao ponto central do retângulo que o re-presenta graficamente.

Devemos lembrar que, a cada parcela da função objetivo, é atribuído um fator de normalização, que traduz a importância daquele critério para o desenho do diagrama. Chamaremos *lambda1*, *lambda2*, *lambda3*, *lambda4* e *lambda5*, respectivamente, os fatores de normalização relativos a cada uma das parcelas analisadas a seguir, por ordem de aparição.

Na primeira parcela é calculada a distância euclidiana *dist* entre cada dois objetos do diagrama. *dist* contribui para a função objetivo com o inverso de seu quadrado. Neste caso, quanto maior for o valor de *lambda1*, mais espalhados ficarão os objetos na área de desenho.

Seja (x,y) a coordenada de um objeto, e sejam *width* e *height*, respectivamente, a largura e a altura da área de trabalho, que podem variar de acordo com a área ocupada pelo diagrama. Definimos as distâncias de um vértice às bordas superior, inferior, esquerda e direita, respectivamente, como sendo: y, *height-y*, x e *width-x*. Esses valores também contribuem para a função objetivo com o inverso de seus quadrados.

De fato, a segunda parcela, minimização da distância de um objeto às bordas, dificulta a aceitação de algumas configurações que melhorariam o desenho. Nós fizemos algumas experiências com *lambda2* igual a zero, e obtivemos melhores resultados. No caso de um objeto tentar ultrapassar as bordas da área de trabalho, nós simplesmente permitimos seu posicionamento nas proximidades da borda, e verificamos se esse movimento gera uma configuração melhor.

A terceira parcela, definida como sendo a soma dos quadrados dos tamanhos das conexões do diagrama, é a que contribui com os valores mais altos para a função objetivo. Por exemplo, se atribuíssemos valores iguais de *lambda* a todas as parcelas, os objetos tenderiam ao centro da área de desenho, pois no cálculo final da função, apenas este critério teria um valor significativo. Os pesos atribuídos a todas as outras parcelas devem sempre ser bem maiores que o valor atribuído a *lambda3*, para que todas as parcelas da função pertençam a um mesmo intervalo de valores.

A quarta parcela, que penaliza objetos muito próximos a conexões, é calculada como sendo o inverso do quadrado da distância entre o objeto e o ponto médio do segmento que representa a conexão.

Na última parcela, cada cruzamento é detectado da seguinte forma: é calculado o ponto de interseção entre as retas correspondentes aos segmentos que representam cada duas conexões do diagrama. Se esse ponto ocorre dentro dos limites dos dois segmentos, um determinado valor é acrescido à função. Esse valor deve estar dentro do intervalo de valores ao qual as outras parcelas pertencem, para que não haja uma grande descontinuidade da função objetivo.

O número de repetições *rep* do laço mais interno foi definido como sendo trinta vezes o número de objetos do diagrama. Esse valor parece ser suficiente para a obtenção de resultados satisfatórios. Um número de repetições maior torna o processo bem mais lento, caso o diagrama tenha um grande número de objetos e conexões.

Para o método SA, esse valor significa que, para cada mudança de temperatura, há *rep* chances do algoritmo tentar mudar de configuração. Caso a temperatura esteja muito alta, o resultado pode ser uma configuração decorrente de até *rep* mudanças, que apenas estragaram o desenho. Essa é uma consideração bastante pessimista, mas deve ser considerada na escolha do valor de *rep* e da própria temperatura inicial T_0.

Caso a configuração inicial seja aleatória, no método SA, o valor de T_0 deve ser alto o suficiente para que, quase todo movimento seja aceito no início do processo. Caso contrário, o ideal é que pudéssemos avaliar o quão boa é a configuração inicial e, a partir daí, encontrar um valor para T_0 que não contribua para a degradação dessa configuração. Em todo caso, T_0 é um valor que pode ser facilmente modificado e experimentado pelo usuário.

O fator de redução da temperatura *redTemp* também é muito importante para o processo de *Annealing*. Dependendo de T_0, um valor muito alto para *redTemp* (por exemplo, um valor entre 95% e 99%) pode significar a aceitação de muitas configurações, que estejam cada vez mais distantes da solução ótima, durante todo o processo. E nós queremos que, ao final do processo, a aceitação de soluções que estraguem o desenho seja totalmente desconsiderada. *redTemp* também pode ser modificado facilmente pelo usuário.

Todas as considerações feitas nos parágrafos anteriores, sobre os parâmetros relacionados à temperatura, não afetam o processo de *Random Search*. Relacionadas à aceitação de uma nova configuração, para o método RS, estão até *s* configurações anteriores cujos valores de solução, ao final de todo o processo, podem ser bem próximos.

Uma diferença importante entre os dois métodos é a seguinte: o método SA, ao aceitar uma configuração que piora o valor da função objetivo, despreza a solução anterior, que poderia equivaler a um mínimo local. Enquanto que, no método RS, há o armazenamento de soluções que, provalvelmente, equivalerão aos mínimos locais encontrados durante o pro-cesso inteiro.

O número de soluções, *Sols*, a serem guardadas é, então, o parâmetro em RS, que pode abreviar ou tornar mais demorado o processo de minimização da função objetivo. Assim, quanto maior o valor de *Sols*, confi-gurações mais diversas serão aceitas, e o método pode demorar mais a convergir. Por outro lado, um valor pequeno

para *Sols* pode resultar em um término prematuro do processo. Na maioria dos nossos exemplos, atribuímos o valor 20 (vinte) a *Sols*.

A condição de término nos dois processos, correspondente ao laço mais externo nos dois algoritmos, depende da razão de aceitação no último conjunto de repetições. A razão é calculada como sendo o número total de configurações aceitas, dividido pelo número de iterações realizadas no laço mais interno. Uma razão mínima de aceitação igual a 25% tem fornecido bons resultados nos dois métodos.

Os maiores problemas relacionados aos métodos SA e RS são o grande consumo de tempo, e a dificuldade de encontrarmos um conjunto de parâmetros *P*, que forneça uma boa configuração final para os diagramas. Além disso, um conjunto *P* pode ser adequado a alguns diagramas, e a ou-tros não. O número de objetos dos diagramas, assim como sua disposição inicial são fatores decisivos na escolha de *P*.

A escolha de *P* não é uma tarefa simples. No caso dos métodos SA ou RS serem incorporados a um editor de diagramas, não se espera que o usuário saiba adequar os parâmetros dos dois processos ao seu diagrama específico.

Em **[dM93]**, foi implementado um sistema que tenta encontrar, para um processo de SA, um bom conjunto *P* de parâmetros, a partir da configuração inicial fornecida pelo usuário. Tal sistema é denominado *Parameter Learner* - PL, e é implementado também como um processo de SA, onde as configurações desse processo são os conjuntos *P*.

Indiscutivelmente, uma solução desse tipo deveria ser incorporada aos programas que implementam SA e RS, em um sistema interativo de desenho de diagramas. O problema dessa alternativa é o aumento considerável de tempo de execução, uma vez que o processo de otimização inteiro deve ser repetido, para cada conjunto *P* encontrado.

Resta ao usuário decidir se é válido ou não gastar um tempo maior, para a obtenção de melhores resultados.

No Apêndice B, veremos alguns resultados obtidos a partir da execução dos métodos discutidos nesta seção, aplicados ao desenhos de diagramas da *LegoShell*.

7.3.2 Método do Baricentro

O método heurístico do Baricentro-BC foi descrito anteriormente na seção 4.2. A diferença principal entre este e os outros algoritmos implementados é a disposição final dos objetos em camadas. Os algoritmos anteriores posicionam os objetos em posições arbitrárias do espaço de trabalho, não considerando a orientação das conexões. Além disso, o método BC trabalha apenas com a informação topológica do diagrama. Isso significa que a configuração geométrica inicial não tem influência alguma no resultado final obtido com sua execução, o que não ocorre com os outros dois métodos.

No nosso trabalho, a remoção dos ciclos é realizada através de uma busca em profundidade no grafo que representa o diagrama. Logo após, a adição de vértices-fan-

tasmas é realizada, ao longo de conexões incidentes a objetos pertencentes a camadas não-vizinhas.

A camada de cada objeto é encontrada, através do procedimento de alocação dos objetos-fonte do diagrama na primeira camada da hierarquia e, a partir daí, todos os outros objetos, de acordo com o que foi descrito na seção 4.2. Depois, as camadas são distribuídas igualmente na extensão da largura da área de trabalho. E, assim, é atribuída a cada objeto a coordenada x final, que é igual para todos os objetos em uma mesma camada.

Agora os objetos devem receber uma ordem inicial dentro de cada camada. Isso é feito através de uma outra busca em profundidade que se inicia nos objetos da primeira camada. O objetivo dessa segunda busca é iniciar o segundo passo com um número já reduzido de cruzamentos, pois os objetos receberão uma posição dentro da camada de acordo com a ordem da visita da busca, que está relacionada com a ordem de visita dos seus ancestrais nas camadas anteriores.

As matrizes de interconexão são preenchidas de acordo com a ordem dos objetos, encontrada no passo anterior. A linha e a coluna de uma matriz correspondem, respectivamente, aos objetos de uma camada e sua vizinha imediata na hierarquia.

O passo de redução de cruzamentos é composto por duas fases. A primeira fase é repetida enquanto houverem mudanças nas matrizes que diminuam o número de cruzamentos, ou até um número determinado de iterações, cujo valor atual é 5[1] (cinco). A segunda fase, que permuta linhas/colunas com mesmo valor de baricentro, também é realizada sob as mesmas condições.

Para a determinação da coordenada y final de cada objeto, devemos respeitar a ordem encontrada no passo anterior, lembrar que os objetos podem ter tamanhos variados e que nenhum deles pode sobrepor a área de desenho de outro objeto. Inicialmente, dentro de cada camada há uma distribuição seqüencial de seus objetos. O primeiro objeto da camada recebe uma posição fixa, e todos os outros, uma posição que é um deslocamento relativo a seu antecessor.

Depois disso, os objetos são classificados de acordo com prioridades de alocação definidas na seção 4.2, para que recebam suas posições finais. Seguindo tal classificação, cada objeto tenta encontrar sua coordenada y dentro da camada. Esse é o terceiro passo do método BC.

Seja l o número de camadas da hierarquia. Consideremos que as camadas da hierarquia sejam numeradas da esquerda para a direita, a partir do número zero. Então, a camada central da hierarquia é definida como sendo a camada de número $l/2$. Assim, no terceiro passo são realizadas seis travessias ao longo da hierarquia, na seguinte ordem:

- uma da primeira camada à camada $l/2$-1;
- uma da última camada à camada $l/2$;

[1] Esse valor foi considerado suficiente (nem sempre necessário) para uma redução aceitável do número de cruzamentos nas hierarquias experimentadas.

- uma da primeira camada à última;
- uma da última camada à primeira;
- uma da camada *l*/2-1 à última; e
- uma da camada *l*/2 à primeira.

As travessias intercaladas, que atingem apenas uma metade da hierarquia de cada vez, foram necessárias, devido ao seguinte fato. Quando as travessias eram realizadas sempre ao longo de toda a hierarquia, havia uma tendência dos objetos serem trazidos apenas para um lado do desenho, dependendo da ordem na qual elas fossem executadas.

Nas travessias realizadas de um nível mais baixo para níveis mais altos, são utilizados os baricentros anteriores; enquanto que, nas travessias de um nível mais alto para níveis mais baixos, os baricentros posteriores é que são utilizados na determinação da nova posição de um objeto.

Na tentativa de posicionamento de um objeto OBJ, três situações podem acontecer:

1. OBJ pode ser alocado sem problemas na posição correspondente ao seu valor de baricentro, que será a coordenada *y* calculada com base em seus objetos adjacentes na camada vizinha;

2. o valor de baricentro de OBJ é maior que a posiçao de um outro objeto subseqüente na mesma camada; e

3. o valor de baricentro de OBJ é menor que a posição de um outro objeto antecedente na mesma camada.

No primeiro caso, não há nada a fazer senão atribuir a posição de baricentro encontrada a OBJ. Nos outros dois casos, deslocamentos devem ser realizados na camada, para que OBJ possa ser posicionado. No segundo caso, por exemplo, todos os outros objetos que estão abaixo de OBJ deveriam ser reposicionados em coordenadas *y* de maior valor. Entretanto, nem todos os objetos podem sofrer um deslocamento incondicional. Aqui entra o conceito de prioridade de posicionamento. Os objetos que possuem uma prioridade maior que OBJ já foram antes deste posicionados, e não devem ser reposicionados em função de um objeto com menor prioridade. O terceiro caso é uma situação simétrica.

O desenho final é simples, uma vez que temos a coordenada (x,y) de cada objeto calculada e usamos o padrão de linha-reta para o desenho das arestas.

O método BC, aplicado aos diagramas da *LegoShell,* teve um desempenho muito melhor que os métodos SA e RS. Os diagramas são desenhados de modo mais comportado, e acompanhar o fluxo dos dados no diagrama torna-se mais fácil.O processo do cálculo das coordenadas dos objetos do diagrama é bem mais rápido. Além disso, a grande vantagem desse método, com relação aos outros, é a não necessidade da escolha de parâmetros, que afetem o resultado de sua execução.

Os mesmos diagramas, desenhados no Apêndice B pelos métodos SA e RS, são desenhados pelo método BC. Lá, poderemos comparar, na prática, o desempenho dos três algoritmos.

Capítulo 8

Conclusões e Trabalhos Futuros

Neste trabalho, foram investigados algoritmos para o desenho automático de grafos e diagramas, com o intuito de obter-se legibilidade. A aplicação de alguns desses algoritmos em nosso problema, o redesenho legível de diagramas *LegoShell*, é quase direta. Aspectos como tamanho e identificação dos objetos podem implicar em algumas pequenas modificações.

Legibilidade é a característica mais esperada de um diagrama, mas ao mesmo tempo, é um conceito bastante subjetivo. Assim, devemos definir legibilidade em função de alguns critérios de estética, que servem como objetivos para aqueles algoritmos.

Critérios de estética, como distribuição uniforme de objetos, não sobreposição de conexões a objetos, minimização do número de cruzamentos e do tamanho das conexões foram julgados, neste trabalho, os mais importantes.

Desse modo, direcionamos nossos estudos àqueles algoritmos, projetados com objetivos semelhantes aos nossos; e três algoritmos para o desenho automático de grafos foram selecionados para experimentação.

Simulated Annealing - SA e *Random Search* - RS são dois métodos de otimização diferentes, para os quais definimos uma mesma função objetivo a ser minimizada. Essa função reflete alguns critérios de legibilidade esperados de um diagrama *LegoShell*. Nos dois casos, consideramos a estrutura interna dos diagramas como um grafo geral não-orientado. Os dois métodos foram descritos no Capítulo 5, e considerações acerca de suas implementações neste trabalho encontram-se no Capítulo 7. No Apêndice B, podemos visualizar exemplos práticos dos dois métodos.

Como comentamos no Capítulo 7, os dois métodos acima podem gerar bons desenhos de diagramas, mas possuem algumas desvantagens. Uma delas é o grande consumo de tempo, ou seja, a convergência dos processos a uma boa configuração

pode ser bastante demorada. Além disso, o grande número de parâmetros, que devem ser iniciados a cada execução, dificulta a utilização desses métodos.

A possibilidade de parametrização dos processos de redesenho pode ser uma característica boa, afinal é uma maneira de expressarmos o quão importante é cada critério de legibilidade considerado. Por outro lado, descobrir qual deve ser o conjunto de parâmetros, que corresponda ao resultado esperado, é uma tarefa bastante complicada. Um usuário da *LegoShell* normalmente não deveria preocupar-se com a atribuição de valores a esses parâmetros.

Desse modo, o uso desses algoritmos no desenho de diagramas da *LegoShell* poderia ser feito se, principalmente, pudéssemos sempre encontrar um conjunto de parâmetros mais adequado a cada diagrama. Se isso fosse possível, provavelmente, o problema relacionado ao grande consumo de tempo também diminuiria.

O método do Baricentro - BC, descrito no Capítulo 4, foi o terceiro método escolhido para o desenho dos diagramas da *LegoShell*. Sua idéia principal é posicionar um objeto de acordo com a posição de seus objetos adjacentes. Além disso, a distribuição dos objetos é feita em camadas, onde objetos nunca se sobrepõem.

A grande vantagem desse método, com relação aos outros dois, é que não precisamos nos preocupar em parametrizar uma execução de acordo com cada diagrama. O algoritmo trabalha apenas com informação topológica do diagrama. Assim, a disposição inicial dos objetos e conexões não afeta o resultado final.

O fato acima pode ser considerado uma desvantagem, pois a configuração inicial do usuário pode ser perdida. Contudo, a disposição dos objetos em camadas está intimamente relacionada com a orientação do fluxo de dados no diagrama. Se o usuário desenha objetos, que são origem de um fluxo de dado, à esquerda daqueles que são destino, podemos inferir que o desenho final do diagrama terá muita semelhança com o desenho original. Portas de entrada (saída), por exemplo, sempre são desenhadas do lado esquerdo (direito) dos objetos.

No Apêndice B, podemos observar que os dois primeiros resultados obtidos com os três métodos são bons, ou seja, os diagramas foram desenhados de modo legível, de acordo com os critérios antes definidos.

O terceiro exemplo pode ser considerado legível, quando desenhado pelos métodos SA e RS, se lembrarmos os critérios considerados pelos dois métodos. O problema desses resultados é a falta de uma certa organização, que facilite o acompanhamento do fluxo da informação. Isso ocorre, em parte, devido ao fato de que os algoritmos tentam desenhar um grafo não-orientado qualquer, e não um diagrama, cujos objetos possuem informação semântica.

Detalhes que estão escondidos atrás desses resultados são, por exemplo, os tempos de execução, menores para o método BC, e a dificuldade de escolha dos parâmetros nos métodos SA e RS, para que aquelas configurações fossem conseguidas.

Se tivéssemos que optar, imediatamente, por um desses métodos, aplicados ao desenho automático de diagramas, sem dúvida, o método BC seria incorporado ao editor da *LegoShell*. Contudo, os métodos SA e RS não devem ser totalmente

desconsiderados, pois também apresentam bons resultados, sob determinadas condições.

Na seção a seguir, enumeraremos alguns pontos escolhidos para estudos posteriores relacionados ao desenho de diagramas *LegoShell.*

8.1 Trabalhos Futuros

Nesta seção, nós sugerimos algumas mudanças e extensões ao trabalho que foi realizado até o presente momento.

Uma mudança mais geral foi citada no Capítulo 1. Trata-se da mudança do padrão de desenho dos diagramas. O padrão de linha-reta é, de fato, o mais utilizado pelos algoritmos de desenho de grafos ou diagramas. Em geral, o resultado é satisfatório, desde que não ocorram sobreposição entre objetos e conexões ou, ainda, um grande número de cruzamentos.

O padrão grade de desenho garante que objetos e conexões não se sobrepõem, pois não é possível que mais de um objeto ocupe uma mesma célula da grade. Além disso, um diagrama desenhado utilizando-se esse padrão parece ter uma maior modularidade de estrutura.

Neste trabalho, o padrão grade de desenho não foi utilizado, pois nossa preocupação inicial era conseguir uma boa distribuição dos objetos, na área de desenho. Agora, um pós-processamento sobre os resultados já obtidos pode ser projetado com esse intuito.

O problema de desenhar as conexões ortogonalmente está intimamente relacionado com o roteamento de conexões no projeto de circuitos integrados. Assim, dada uma disposição qualquer dos objetos, as conexões devem ser roteadas, em uma grade implícita, considerando objetos e conexões antes posicionados como obstáculos, que não devem ser sobrepostos.

Outro ponto, que deve ser ainda considerado, é o tratamento de abstração de computações. Atualmente, objetos que representam computações abstraídas aparecem apenas simbolicamente nos diagramas. O mecanismo de abstração não foi, de fato implementado. Quando computações puderem realmente ser abstraídas, deveremos optar pelo redesenho automático, imediato ou não, do diagrama inteiro, ou apenas parte dele.

No segundo caso, pode ser necessária a implementação de um outro mecanismo, também mencionado no primeiro capítulo, que é a aceitação de restrições ao desenho. Por exemplo, se uma computação fosse abstraída, poderíamos:

1. obrigar alguns objetos a não mudarem de posição durante o desenho;
2. permitir que todos os objetos sejam reposicionados, desde que, dentro de um perímetro determinado; e
3. determinar uma área em torno da região abstraída, que deve ser redesenhada.

Outras restrições que poderiam ser incorporadas ao sistema de redesenho são:

4. pré-posicionamento de objetos em determinadas posições, regiões, ou camadas;
5. agrupamento de um conjunto de objetos em uma determinada região; e
6. concentração de um conjunto de conexões, cujo destino é um mesmo objeto, em um único objeto gráfico.

O último item seria um recurso utilizado para diminuir a complexidade visual de um diagrama que tenha um grande número de conexões.

As extensões aqui propostas são resultados de algumas questões levantadas no decorrer do desenvolvimento deste trabalho. Outras extensões e sugestões devem surgir posteriormente, uma vez que este trabalho será continuado, até que o editor da *LegoShell* esteja funcioando em sua totalidade.

Apêndice A

Sintaxe da Linguagem de Definição de Diagramas

A sintaxe da linguagem de descrição de diagramas LegoShell, projetada para a criação de cada objeto (aqui denominado *box*) e conexão (aqui denomida *link*) componentes de uma computação, é assim definida:

description ::= {*object*} 1+
object ::= *box* | *link*
box ::= **box** { *box_atribute* } 0+
link ::= **link** { *link_atribute* } 0+
box_atribute ::= *common_atribute* | *height* | *width* | *position* | { *port* } 0+
link_atribute ::= *common_atribute* | *source* | *target*
common_atribute ::= *type* | *label*
type ::= **type** *types*
types ::= **cm** | **star** | **mbox** | **tty** | **printer** | **file** | **in** | **out** | **data** | **exe** | **dummy** | **container**
label ::= **label** *identifier*
height ::= **height** *value*
width ::= **width** *value*
position ::= **position** *coordinate*
port ::= **port** { *port_atribute* } 0+
port_atribute ::= *common_atribute* | *position*
source ::= **source** *identifier*
target ::= **target** *identifier*
identifier ::= *id* | *id.id*
coordinate ::= *value* X *value*
id ::= [A-Za-z_][A-Za-z0-9_] 0+
value ::= [0-9] 1+

Um pequeno trecho da descrição do diagrama representado na figura 1-2 seria:

```
box {
  type file
  label Desord
  position 100 X 300
  port {
    type in
    label p1Desord
    position 80 X 300
  }
  port {
    type out
    label p2Desord
    position 120 X 300
  }
}

...

link {
  type data
  label dadosDesord
  source Desord.p2Desord
  target M1.p1M1
}

...
```

Apêndice B

Exemplos

Nas figuras B-1, B-5 e B-9 aparecem três exemplos de diagramas *LegoShell*. Após cada figura, serão apresentados os resultados obtidos, através da execução dos métodos *Simulated Annealing* - SA, *Random Search*-RS e do Baricentro - BC.

Para os métodos SA e RS, definimos o seguinte conjunto de parâmetros:

1. Incr: movimento inicial de um objeto para obtenção de um nova configuração do sistema.
2. RedIncr: fator de redução do movimento de um objeto em um determinado intervalo de iterações.
3. InitTemp: temperatura inicial, válida apenas para o processo de SA.
4. RedTemp: fator de redução da temperatura, válido apenas para o processo de SA.
5. DNode: fator de normalização do critério "Distância entre nós".
6. DBord: fator de normalização do critério "Distância de um nó às bordas".
7. Edge: fator de normalização do critério "Tamanho de arestas".
8. DNEdge: fator de normalização do critério "Distância nó-aresta".
9. Cross: fator de normalização do critério "Minimização do número de cruzamentos entre arestas".
10. Sols: números de soluções armazenadas durante o processo de RS.

O primeiro diagrama (vide figura B-1) representa uma computação simples, composta de três programas Cm, um arquivo de dados e dois dispositivos periféricos. Utilizar algoritmos para o desenho automático de diagramas pode não ser interessante, quando o diagrama possui poucos objetos e conexões. Todavia, há um grafo não-planar do tipo K33, subjacente ao diagrama, e queremos observar como um grafo deste tipo é desenhado pelos algoritmos.

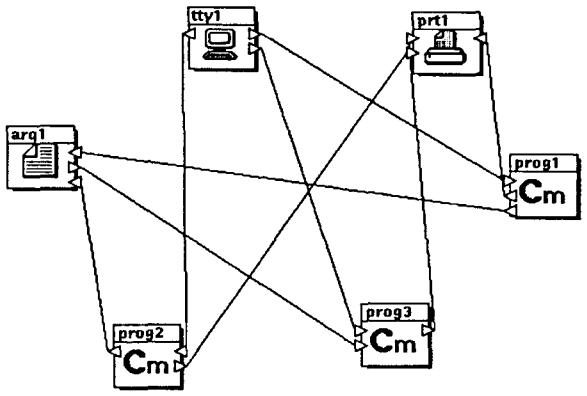


FIGURA B-1 Primeiro exemplo: um diagrama com 6 objetos e 9 conexões.

Para o diagrama obtido na figura B-2, foram utilizados os seguintes parâmetros para o processo de SA:

Parâmetro	Valor	Parâmetro	Valor
Incr	100	Dnode	25.000
RedIncr	80	DBord	0
InitTemp	100	Edge	5
RedTemp	80	DNEdge	10.000
		Cross	1.000

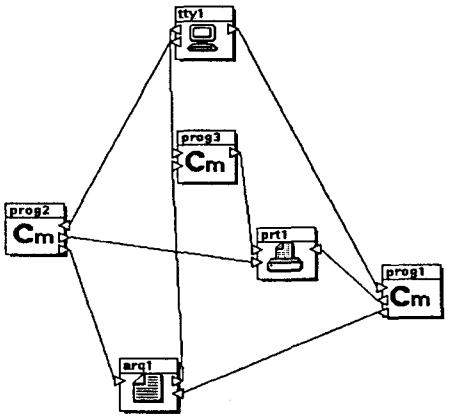


FIGURA B-2 Método SA aplicado ao diagrama da figura B-1.

Para o diagrama obtido na figura B-3, foram utilizados os seguintes parâmetros para o processo de RS:

Parâmetro	Valor	Parâmetro	Valor
Incr	100	Dnode	20.000
RedIncr	80	DBord	0
		Edge	5
		DNEdge	10.000
Sols	20	Cross	500

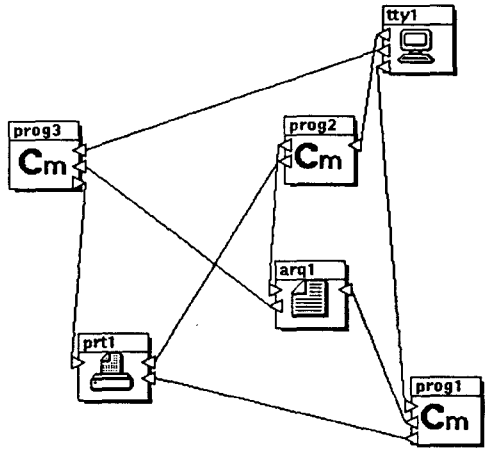


FIGURA B-3 Método RS aplicado ao diagrama da figura B-1.

A figura B-4 apresenta o resultado relativo ao método BC.

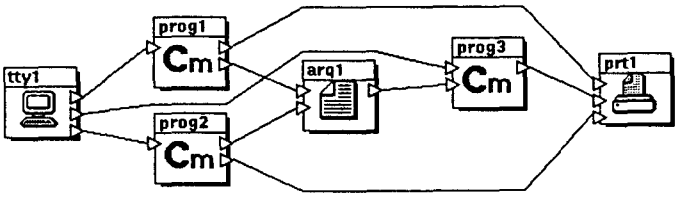


FIGURA B-4 Método BC aplicado ao diagrama da figura B-1.

A figura B-5 apresenta o segundo exemplo de computação *LegoShell*. Nesta computação é representado um sistema simplificado de vendas de um supermercado.

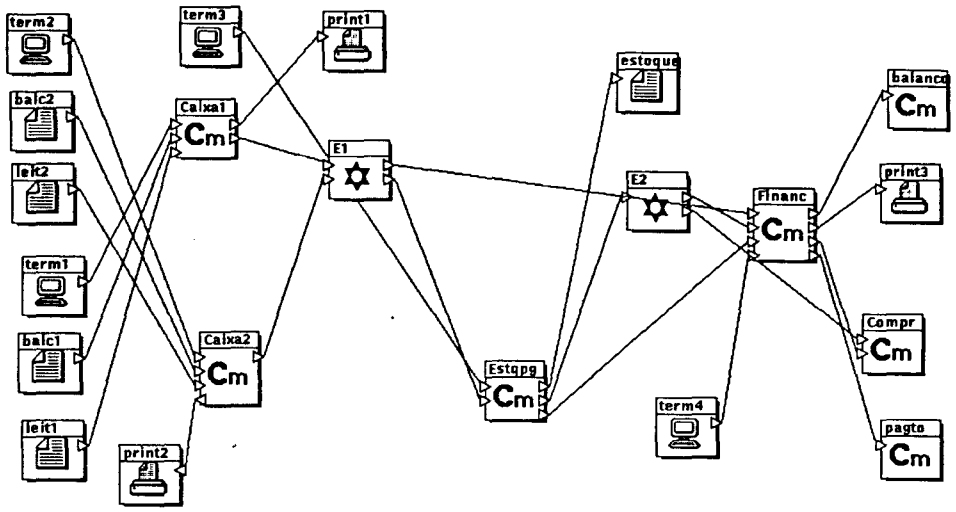


FIGURA B-5 Segundo exemplo: um diagrama com 21 objetos e 23 conexões.

O diagrama obtido na figura B-6, foi conseguido a partir do seguinte conjunto de parâmetros para o processo de SA:

Parâmetro	Valor	Parâmetro	Valor
Incr	80	Dnode	25.000
RedIncr	80	DBord	0
InitTemp	200	Edge	40
RedTemp	80	DNEdge	10.000
		Cross	3.000

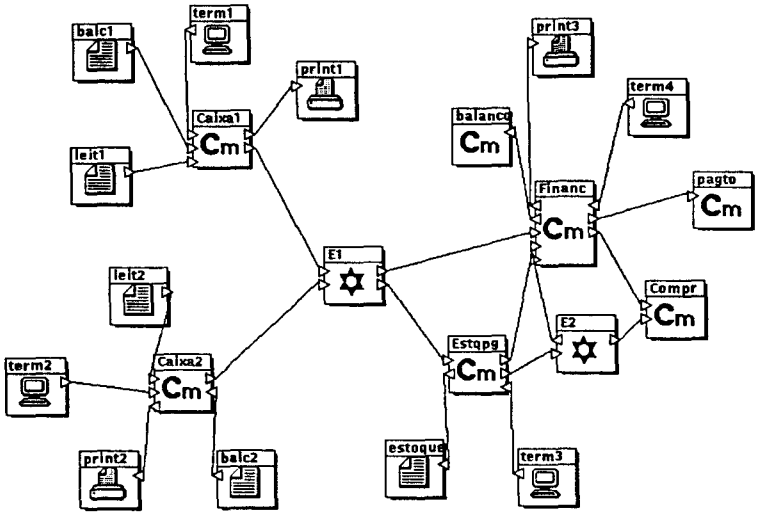


FIGURA B-6 Método SA aplicado ao diagrama da figura B-5.

Para o diagrama obtido na figura B-7, foram utilizados os seguintes parâmetros para o processo de RS:

Parâmetro	Valor	Parâmetro	Valor
Incr	80	Dnode	25.000
RedIncr	80	DBord	0
		Edge	40
		DNEdge	10.000
Sols	20	Cross	3.000

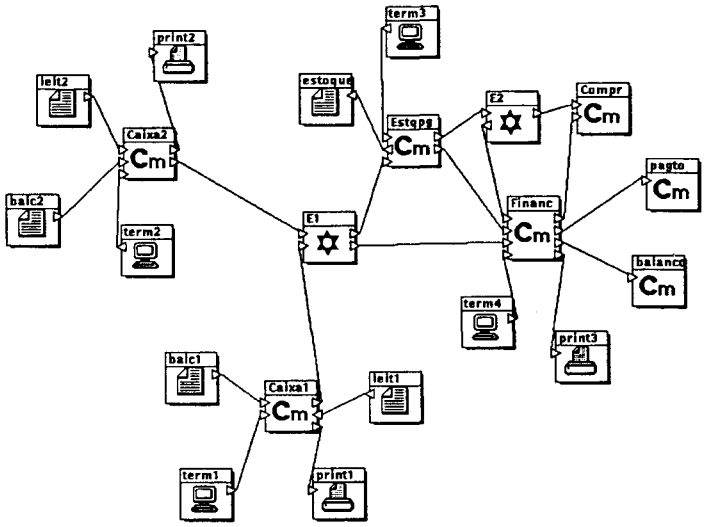


FIGURA B-7 Metodo HS aplicado ao diagrama da figura B-5.

A figura B-8 apresenta o resultado relativo ao método BC.

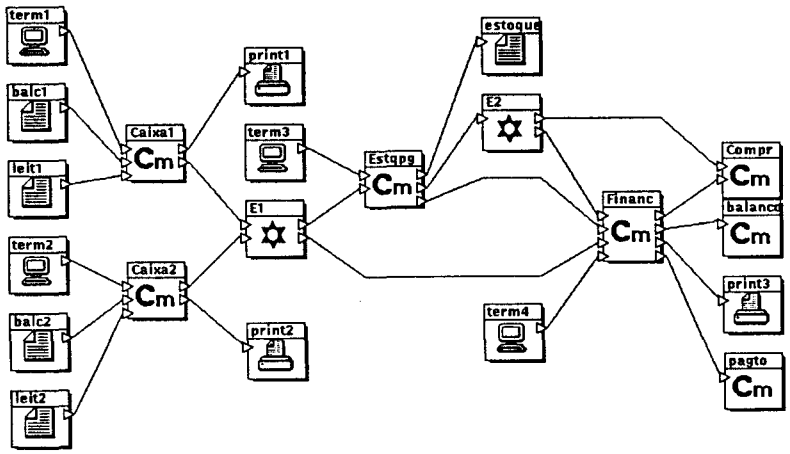


FIGURA B-8 Método BC aplicado ao diagrama da figura B-5.

A figura B-9 mostra o terceiro exemplo de computação *LegoShell*. Esta computação representa um sistema sem qualquer significado aparente. Ela foi criada somente para que testes fossem realizados sobre uma computação composta de um maior número de objetos e conexões.

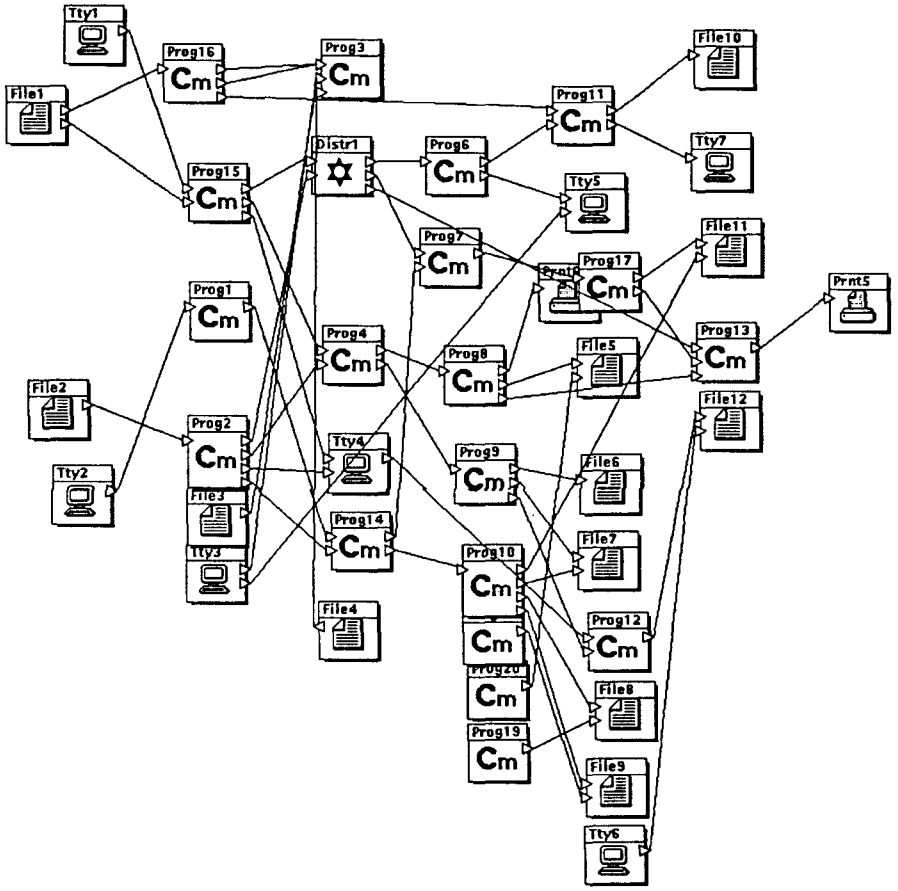


FIGURA B-9 Terceiro exemplo: um diagrama com 42 objetos e 51 conexões.

Para o diagrama obtido na figura B-10, foram utilizados os seguintes parâmetros para o processo de SA:

Parâmetro	Valor	Parâmetro	Valor
Incr	100	Dnode	20.000
RedIncr	80	DBord	0
InitTemp	200	Edge	10
RedTemp	80	DNEdge	10
		Cross	3.000

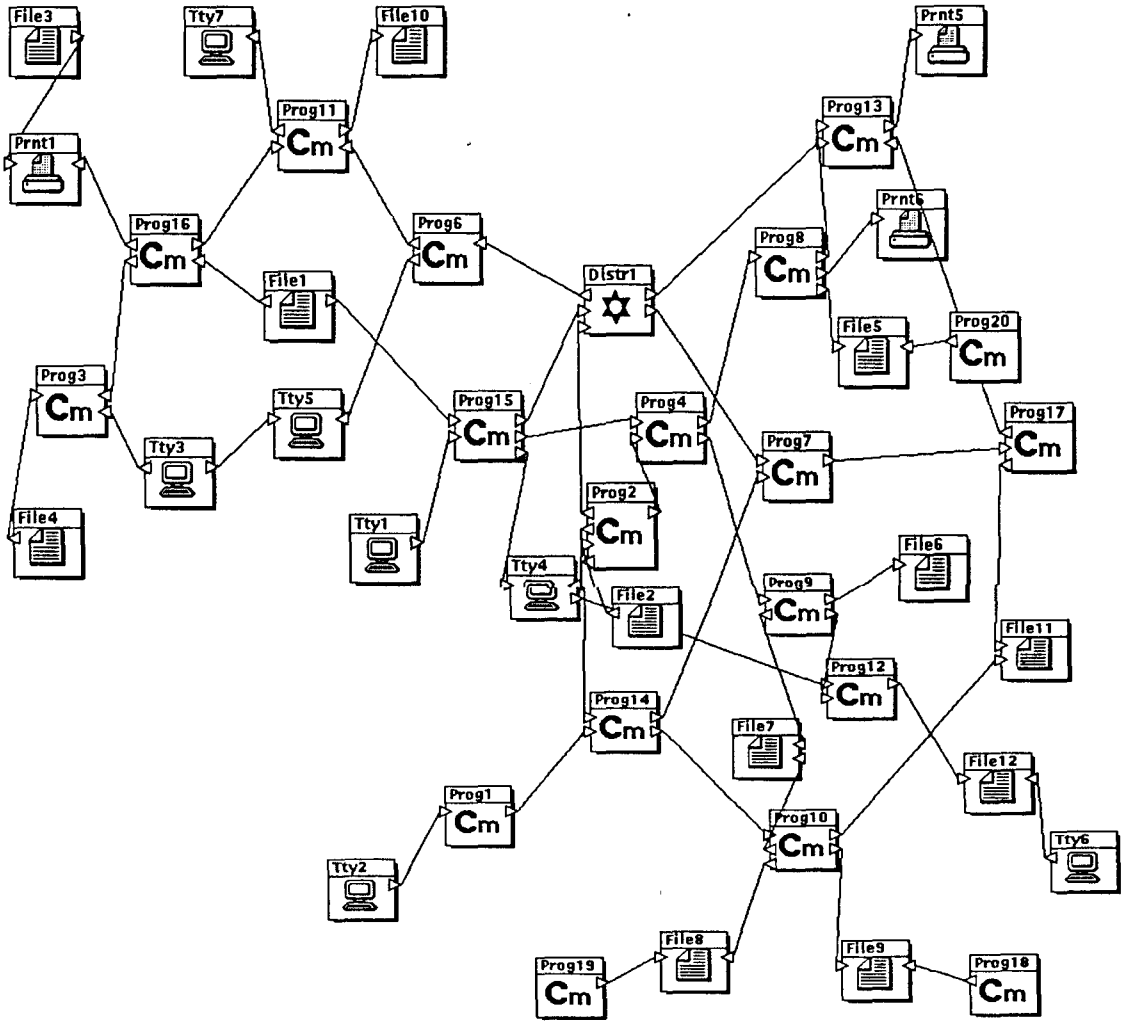


FIGURA B-10 Método SA aplicado ao diagrama da figura B-9.

Para o diagrama obtido na figura B-11, foram utilizados os seguintes parâmetros para o processo de RS:

Parâmetro	Valor	Parâmetro	Valor
Incr	100	Dnode	20.000
RedIncr	80	DBord	0
		Edge	10
		DNEdge	10.000
Sols	20	Cross	1.000

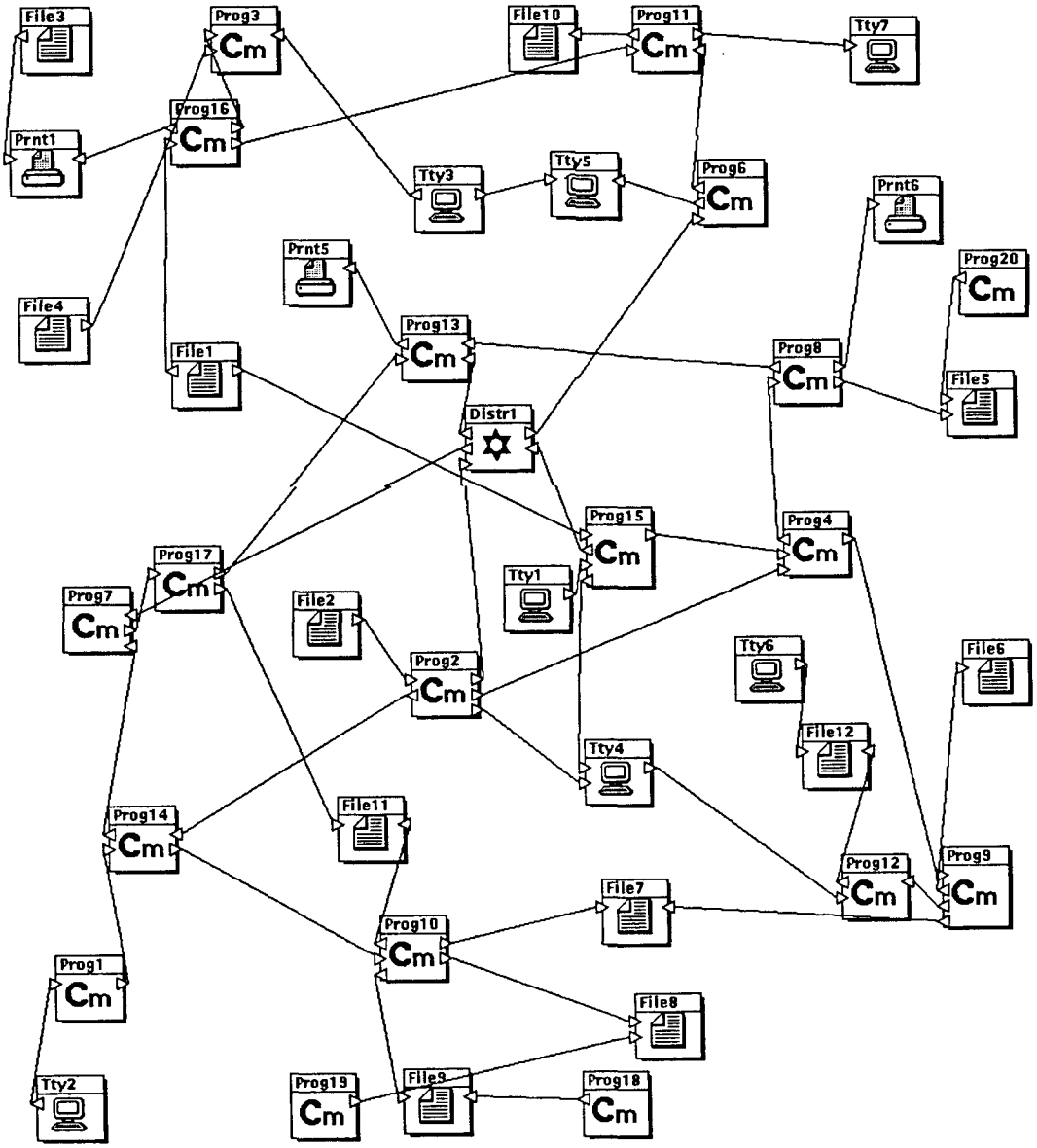


FIGURA B-11 Método RS aplicado ao diagrama da figura B-9.

A figura B-12 apresenta o resultado relativo ao método BC:

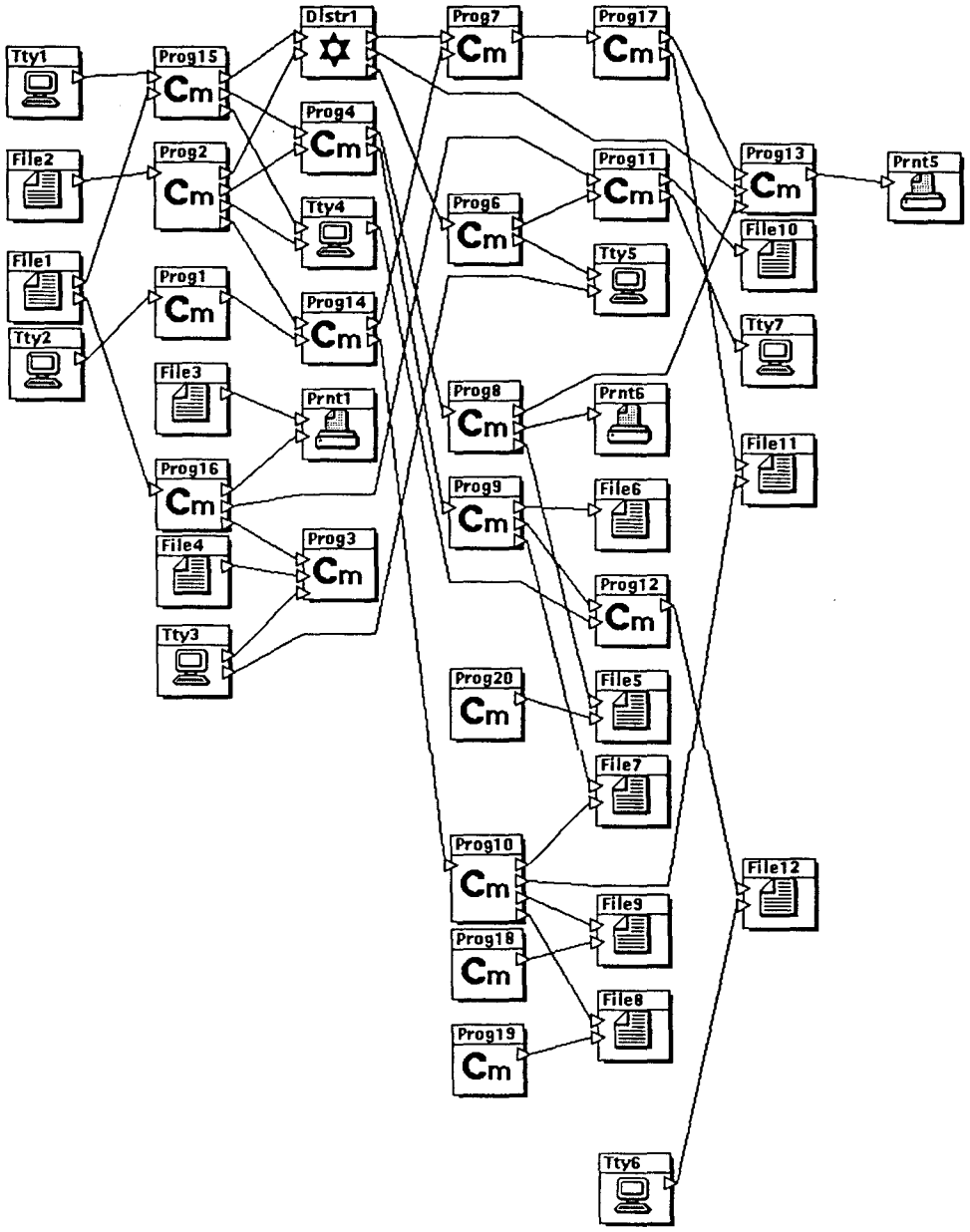


FIGURA B-12 Método BC aplicado ao diagrama da figura B-9.

Apêndice C

Referências Bibliográficas

1. **[Bac86]** **The Design of the Unix Operating System**
M. J. Bach
Prentice-Hall, 1986.

2. **[BBd+86]** **An Automatic Layout Facility and its Applications**
C. Batini, P. Brunetti, G. di Battista, P. Naggar, E. Nardelli, G. Richelli e R. Tamassia
Technical Report 06.86, Dept. di Inform. e Sistemistica, Univ. di Roma "la Sapienza", Jun de 1986.

3. **[BFN85]** **What is a Good Diagram? A Pragmatic Approach**
C. Batini, L. Furlani e E. Nardelli
IEEE Proceedings of the 4th International Conference on Entity-Relationship Approach, Chicago, Out de 1985.

4. **[BM77]** **Graph Theory with Applications**
J. A. Bondy e U. S. R. Murty
American Elsevier Publishing Co., Inc., 1977.

5. **[BNT+84]** **A Graph Theoretic Approach to Aesthetic Layout of Information Systems Diagrams**
C. Batini, E. Nardelli, M. Talamo e R. Tamassia
Proc. of 10th International Workshop on Graph Theoretic Concepts in Computer Science, WG'84, Berlin, 1984.

6. **[BNT84]** **Computer-Aided Layout of Entity-relationship Diagrams**
C. Batini, E. Nardelli e R. Tamassia
Journal of System and Software, 4:163-173, 1984.

7. **[BNT+85]** **GINCOD: A Graphical Tool for Conceptual Design of Data Base Applications**
C. Batini, E. Nardelli, M. Talamo e R. Tamassia
Computer-Aided Database Design: The Dataid Project, Elsevier Science Publ., Cap. II, 1985.

8. [BNT86] **A Layout Algorithm for Data-flow Diagrams**
C. Batini, E. Nardelli e R. Tamassia
IEEE Trans. on Software Engineering, SE-12(4):538-546, Abr de 1986.

9. [Bra88] **Nice Drawing of Graphs are Computationally Hard**
F. J. Brandenburg
TR MIP-8820, Fakultät für Mathematik und Informatik, Univ. Passau, 1988.

10. [Ces94] **Relatório de Atividades de Iniciação Científica**
W. M. Ceschim
Projeto A_HAND, Março de 1994.

11. [CON85] **Drawing Plane Graphs Nicely**
N. Chiba, K. Onoguchi e T. Nishizeki
Acta Informatica, 22:187-201, 1985.

12. [dC94] **OMNI - Sistema de Suporte a Aplicações Distribuídas**
C. di Cianni
Dissertação de Mestrado em andamento, DCC-IMECC-UNICAMP.

13. [DH89] **Drawing Graphs Using Simulated Annealing**
R. Davidson e D. Harel
Technical Report, Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, Jul de 1989.

14. [DM90] **A Framework for the Automated Drawing of Data Structure Diagrams**
C. Ding e P. Mateti
IEEE Trans. on Software Engineering, 16(5):543-557, Mai de 1990.

15. [dM93] **A Layout System for Information System Diagrams**
C. F. X. de Mendonça
Tese de Doutorado, Dept. of Computer Science, University of Queensland, 1993.

16. [Dru89] **LegoShell: Linguagem de Computações**
R. Drummond
Anais do III Simpósio Brasilerio de Engenharia de Software, Out de 1989.

17. [Ead84] **A Heuristic for Graph Drawing**
P. Eades
Congressus Numerantium, 1984.

18. [ES90] **How to Draw a Directed Graph**
P. Eades e K. Sugiyama
Journal of Information Processing, 13(4):424-437, 1990.

19. [ET76] **Computing an st-Numbering**
S. Even e R. Tarjan
Theoretical Computer Science, 2:339-344, 1976.

20. [ET89] **Algorithms for Automatic Graph Drawing: An Annotated Bibliography**
P. Eades e R. Tamassia
Technical Report 82, Dept. of Computer Science, Univ. of Queensland, Out de 1989.

21. [Eve79] **Graph Algorithms**
S. Even
Computer Science Press, Inc., 1979.

22. [Fur91] **Um compilador para uma Linguagem de Programação Orientada a Objetos**
C. A. Furuti
Dissertação de Mestrado, DCC-IMECC-UNICAMP, Jul de 1991.

23. [Fur93] **The Astra Toolkit**
C. A. Furuti
Anais do VII Simpósio Brasileiro de Engenharia de Software, Out de 1993.

24. [GNV88] **DAG - A Program that Draws Directed Graphs**
E. R. Gansner, S. C. North e K. P. Vo
Software - Practice and Experience, 18(11):1047-1062, Nov de 1988.

25. [HT73] **Algorithm 447: Efficient Algorithms for Graph Manipulation**
J. Hopcroft e R. Tarjan
Communications of the ACM, 16:372-378, 1973.

26. [HT74] **Efficient Planarity Testing**
J. Hopcroft e R. Tarjan
Journal of the Assoc. for Computing Machinery, 21(4):549-568, Out de 1974.

27. [JEM+91] **A Note on Planar Graph Drawing Algorithms**
S. Jones, P. Eades, A. Moran, N. Ward, G. Delott e R. Tamassia
Technical Report 216, Dept. of Computer Science, Univ. of Queensland, Dez de 1991.

28. [JRA92] **Métodos Estocásticos em Computação Visual**
J. R. A. Torreão
Anais da VIII Escola de Computação, Ago de 1992.

29. [KK88] **Automatic Display of Network Structures for Human Understanding**
T. Kamada e S. Kawai
Technical Report 88-007, Dept. of Computer Science, Univ. of Tokyo, Fev de 1988.

30. [KMN89] **Graphical Configuration Programming**
J. Kramer, J. Magee e K. Ng
IEEE Computer, 53-65, Out de 1989.

31. [LNS85] **A Method for Drawing Graphs**
R. J. Lipton, S. C. North e J. S. Sandberg
Proc. 1st Symposium on Computational Geometry, 153-160, 1985.

32. [Mes88] **Automatic Layout of Large Directed Graphs**
E. B. Messinger
Tese de Doutorado, Dept. of Computer Science, Univ. of Washington, 1988.

33. [Mül91] **Uma Interface de Comunicação para um Ambiente de Reestruturação de Programas**
B. Müller
Dissertação de Mestrado, DCC-IMECC-UNICAMP, 1991.

34. [NT84] **A Fast Algorithm for Planarization of Sparse Diagrams**
E. Nardelli e M. Talamo
CNR Technical Report R105, IASI, Univ. di Roma "la Sapienza", 1984.

35. [Piñ91] **Editor Topológico para a Linguagem de Especificação de Computações: *LegoShell***
H. Piñón Arias
Dissertação de Mestrado, DCC-IMECC-UNICAMP, Jan de 1991.

36. [PT90] **Edge: An Extendible Graph Editor**
F. N. Paulisch r W. F. Tichy
Software - Practice and Experience, 20(S1):63-88, Jun de 1990.

37. [RDM+87] **A Browser for Directed Graphs**
L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis e A. Tuan
Software - Practice and Experience, 17(1):61-76, Jan de 1987.

38. [Rei81] **Tidier Drawing of Trees**
E. Reingold
IEEE Transactions on Software Enginnering, 7:223-228, 1981.

39. [Rub75] **An Improved Algorithm for Testing the Planarity of a Graph**
F. Rubin
IEEE Trans. on Computers, C-24(2):113-121, Fev de 1975.

40. [SBD87] **Thermodinamic Optimization of Block Placement**
P. Siarry, L. Bergonzi e G. Dreyfus
IEEE Transactions on Computer-Aided Design, CAD-6(2):211-221, Mar de 1987.

41. [SCO91] **X Toolkit Guide**
SCO Open Desktop
The Santa Cruz Operation, Inc., Ago de 1991.

42. [SR83] **The Complexity of Drawing Trees Nicely**
K. Supowit and E. Reingold
Acta Informatica, 18:377-392, 1983.

43. [Str91] **The C++ Programming Language**
B. Stroustrup
Addison-Wesley Publishing Co., 1991.

44. [STT81] **Methods for Visual Understanding of Hierarchical System Structures**
K. Sugiyama, S. Tagawa e M. Toda
IEEE Trans. on Systems, Man, and Cybernetics, SMC-11(2):109-125, Fev de 1981.

45. [Sug87] **A Cognitive Approach for Graph Drawing**
K. Sugiyama
Cybernetics and Systems: An International Journal, 18:447-488, 1987.

46. **[SV85]** **The TimberWolf Placement and Routing Package**
C. Sechen e A. S. Vincentelli
IEEE Journal of Solid-State Circuits, SC-20(2):510-522, Abr de 1985.

47. **[Tam85]** **New Layout Techniques for Entity-Relationship Diagrams**
R. Tamassia
IEEE Proceedings of the 4th International Conference on Entity-Relationship Approach, Chicago, Out de 1985.

48. **[Tam87]** **On Embedding a Graph in the Grid with the Minimum Number of Bends**
R. Tamassia
SIAM J. Computing, 16:421-444, 1987.

49. **[Tam90]** **Planar Orthogonal Drawing of Graphs**
R. Tamassia
Proc. IEEE Int. Symposium on Circuits and Systems, 1990.

50. **[Tar72]** **Depth-first Search and Linear Graph Algorithms**
R. Tarjan
SIAM J. Computing, 1(2):146-160, Jun de 1972.

51. **[TBB88]** **Automatic Graph Drawing and Readability of Diagrams**
R. Tamassia, G. Di Batista e C. Batini
IEEE Trans. on Systems, Man and Cybernetics, 18(1):61-79, Jan-Fev de 1988.[Tel93]

52. **[Tel93]** **A linguagem de Programação Cm (versão 2.0x)**
A. P. Teles
Dissertação de Mestrado, DCC-IMECC-UNICAMP, Out de 1993.

53. **[TN87]** **Knowledge-based Editors for Directed Graphs**
W. F. Tichy e F. J. Newbery
Lectures Notices in Computer Science, 289:101-109, 1987.

54. **[TT86]** **A Unified Approach to Visibility Representations of Planar Graphs**
R. Tamassia e I. G. Tollis
Discrete Computational Geometry, 1(4):321-341, 1986.

55. **[TT89]** **Planar Grid Embedding in Linear Time**
R. Tamassia e I. G. Tollis
IEEE Transactions on Circuits and Systems, 36(9):1230-1234, Set de 1989.

56. **[Vau80]** **Pretty-Printing of Trees**
J. G. Vaucher
Software - Practice and Experience, 10:553-561, 1980.

57. **[War77]** **Crossing Theory and Hierarchy Mapping**
J. N. Warfield
IEEE Trans. on Systems, Man, and Cybernetics, SMC-7(7):505-523, Jul de 1977.

58. [WS79] **Tidy Drawing of Trees**
C. Wetherell e A. Shannon
IEEE Transactions on Software Enginnering, SE-5(5):514-520, Set de 1979.