Universidade Federal de Minas Gerais
Programa de Pós-Graduação em Engenharia Elétrica
Centro de Pesquisa e Desenvolvimento em Engenharia Elétrica

Uma Biblioteca para Desenho de Grafos Construída sob o Paradigma de Programação Genérica

Leandro Terra Cunha Melo

Dissertação de mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Renato Cardoso Mesquita

Belo Horizonte, Maio de 2007.

Resumo

Grafos são estruturas combinatoriais presentes em inúmeras aplicações. Seus vértices e arestas correspondem, respectivamente, às entidades e às relações de adjacência de determinado domínio. Apesar de um grafo ser um conceito puramente matemático, é fundamental que seu desenho seja inteligível e que transmita o máximo de informações possível. Em alguns casos, a clareza do desenho de um grafo não é apenas uma questão estética, mas um requisito essencial da aplicação. Este trabalho apresenta uma biblioteca para desenho de grafos chamada GTAD (*Graph Toolkit for Algorithms and Drawings*). Uma visão introdutória sobre a área de desenho de grafos, em conjunto com uma análise detalhada de três algoritmos de desenho ortogonal de grafos, também é apresentada. A GTAD é desenvolvida na linguagem de programação C++, sob o paradigma de programação genérica, que visa abstrair implementações das estruturas de dados sobre as quais elas operam. O mecanismo de *templates* da linguagem C++, ferramenta fundamental para a programação genérica, garante a resolução de tipos em tempo de compilação. Conseqüentemente, implementações baseadas nesta abordagem são comumente mais eficientes que aquelas que utilizam o paradigma de orientação a objetos em sua forma convencional. Resultados de desempenho de operações básicas de manipulação de grafos mostram que a GTAD é superior a outras bibliotecas, quando comparados os tempos de execução. Implementações de alguns algoritmos também foram submetidas a testes de desempenho e obtiveram resultados positivos. Finalmente, desenhos de grafos de categorias distintas, gerados sob diferentes estratégias, são discutidos e comparados entre si.

Abstract

Graphs are combinatorial structures used in a great variety of applications. Vertices and edges of a graph correspond, respectively, to entities and relationships of a specific domain. Although the concept of a graph is purely mathematical, it is very important that its drawing effectively convey all necessary information. There are cases in which the clarity of a graph drawing is not a simple aesthetical matter, but part of the application's requirement. This work presents the GTAD (Graph Toolkit for Algorithms and Drawings) graph drawing library. An introductory perspective of the graph drawing area, along with a detailed analysis of three orthogonal drawing algorithms, is also presented. The GTAD is developed with the C++ programming language, under the generic programming paradigm, which focus on abstracting implementations from the data structures in which they operate. C++ *template* is a fundamental tool to achieve generic programming. Since type resolution of *templates* is performed at compile time, generic libraries are usually more efficient than libraries developed under the object-oriented paradigm in its traditional form. Performance results of basic graph manipulation operations show that GTAD is faster than other libraries when execution times are compared. Implementations of a few algorithms were also submitted to performance tests and obtained positive results. Finally, drawings of graphs from different categories, generated by different strategies, are discussed and compared against each other.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Símbolos

G	Um grafo.
V	Número de vértices de um grafo.
E	Número de arestas de um grafo.
v,w,u	Vértices de um grafo.
e	Uma aresta de um grafo.
(u, v)	Uma aresta formada por vértices u e v de um grafo.
Γ	Desenho de um grafo.
s	Fonte de uma rede (*source*).
t	Sumidouro de uma rede (*target*).
G^*	Dual de um grafo G.
f	Uma face interna de um embutimento planar.
h	Face externa de um embutimento planar.
d	Grau de um vértice.
C_d	Grafo cíclico formado por d vértices.
p	Total de ângulos-de-vértices e ângulos-de-dobras em uma face.
$a(f)$	Número total de ângulos-de-vértices em uma face f.
$D(v)$	Conjunto de arestas com origem em um vértice v.
$D(f)$	Conjunto de arestas anti-horário da face f.
H	Representação ortogonal de um embutimento planar.
$\alpha(u, v)$	Parâmetro utilizado na designação do ângulo entre duas arestas de H.
$\beta(u, v)$	Parâmetro que indica o número de dobras de uma aresta em H.
$\lambda(u, v)$	Limite inferior de um arco em uma rede.
$\mu(u, v)$	Capacidade de um arco em uma rede.
$\chi(u, v)$	Custo de um arco em uma rede.
η_{hor}	Rede horizontal associada a H.

η_{ver}	Rede vertical associada a H.
σ	Fluxo produzido ou consumido em uma rede.
ϕ	Custo de um fluxo em uma rede.

Capítulo 1

Introdução

Estruturas combinatoriais compostas por um conjunto de entidades e relações de adjacência são utilizadas em uma grande variedade de aplicações. Elas são chamadas de grafos e podem representar modelos físicos como um circuito elétrico ou uma rede de computadores. As entidades e relações de um grafo são designadas, respectivamente, por *vértices* e *arestas*, os quais podem possuir atributos adicionais como peso, cor ou tamanho.

Apesar de serem conhecidos há mais de dois séculos [38], os grafos ainda são tema de muitas obras [116, 102, 73, 118, 88], sendo que do ponto de vista matemático, muitos problemas continuam em aberto [12].

Grafos estão presentes em inúmeras áreas [102], como escalonamento de tarefas, sistemas geográficos, recuperação de informação, projeto de circuitos integrados, geometria computacional ou desenvolvimento de compiladores. Na engenharia de software, por exemplo, é muito comum a adoção de diagramas de classes para visualização da estrutura estática da aplicação. Conforme ilustra a figura 1.1, um grafo, no qual os vértices correspondem às classes e as arestas correspondem às relações entre elas, pode ser utilizado para modelar esse tipo de diagrama.

Além do conteúdo combinatorial do grafo, a forma como ele é desenhado pode ser extremamente importante [30]. Enquanto que um desenho elaborado sob critérios adequados ao contexto em questão pode transmitir informações eficientemente, um desenho confuso pode trazer inúmeras dificuldades de entendimento. Considere, por exemplo, o mesmo diagrama UML da figura 1.1, desenhado sob critérios totalmente aleatórios, como ilustra a figura 1.2.

O desenho de um grafo deve ser altamente inteligível. Mas sua clareza não é simplesmente uma questão estética. Há situações em que determinadas características se tornam extremamente importantes e indispensáveis. Em uma aplicação onde um grafo modela um circuito elétrico, e este precisa ser impresso em uma placa, é inadmissível que o desenho do grafo ocupe uma área maior do que a da placa - nesse caso, há ainda um outro agravante que são os cruzamentos de arestas, que devem ser minimizados. O desenvolvimento de *layouts* de circuitos elétricos é uma aplicação antiga e bem conhecida de desenho de grafos [107].

De maneira sucinta, o desenho de um grafo consiste, basicamente, de duas coisas: um conjunto de coordenadas que indica a posição de cada vértice e um conjunto de curvas que conecta pares de vértices. Apesar de parecer óbvio e trivial, este é um problema complexo, que muitas vezes pode passar desapercebido, dado que o desenho de um grafo é comumente utilizado para

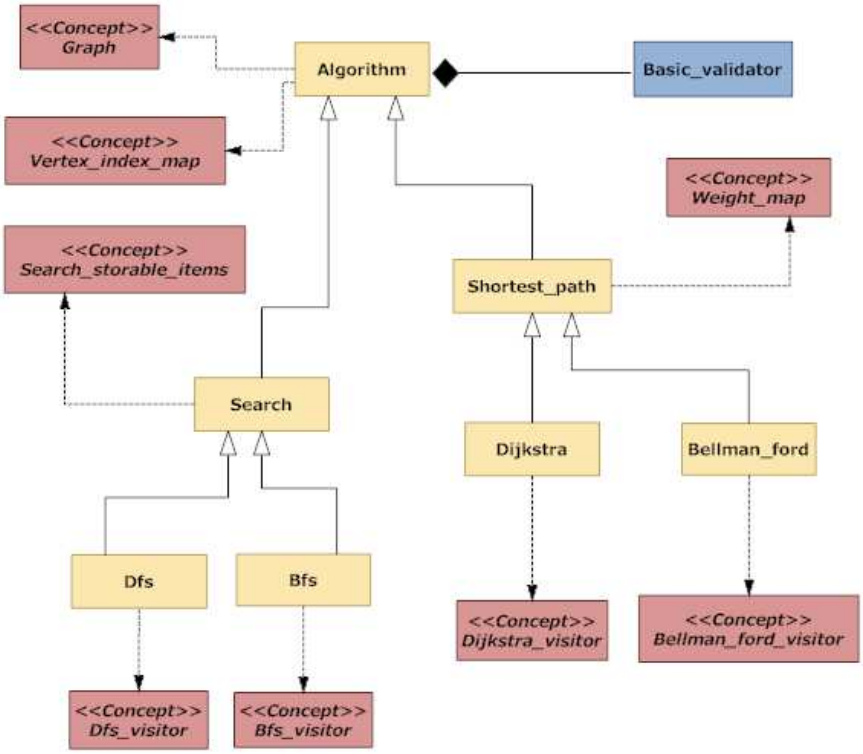

Figura 1.1: Diagrama UML desenhado sob critérios adequados.

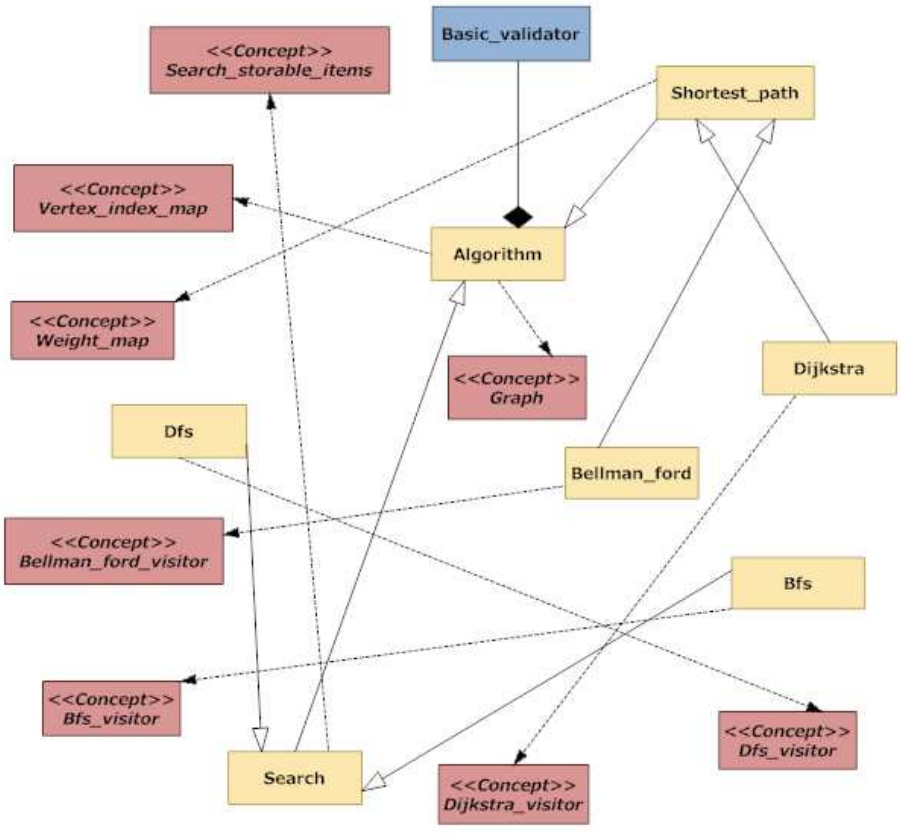

Figura 1.2: Diagrama UML desenhado sob critérios aleatórios.

defini-lo. Euler, por exemplo, se baseou no desenho de um grafo para resolver o problema das pontes de Könisgsberger [75]. No entanto, quando o desenho precisa ser gerado a partir da descrição matemática do grafo, inúmeras peculiaridades e dificuldades entram em cena.

O problema se torna ainda mais desafiador diante da necessidade de qualificação de um "bom"desenho de grafo. Para uma aplicação de construção de compiladores, pode ser melhor que o grafo de chamada de funções seja exibido como uma árvore, onde as as arestas são desenhadas em linhas retas. Por outro lado, arestas verticais e horizontais podem ser preferíveis no desenho de um diagrama esquemático de um banco de dados. Portanto, a definição da qualidade do desenho de um grafo é subjetiva e dependente dos critérios estéticos adotados.

Muitas pesquisas têm sido conduzidas na área de desenho de grafos. Várias metodologias e técnicas são conhecidas. Di Battista et al. [29] apresentam um bibliografia extensa sobre o assunto.

Algoritmos de desenho de grafo são normalmente difíceis de serem implementados. Na maioria das vezes, dependem de outros algoritmos para cumprir etapas intermediárias. Em alguns casos, esses algoritmos também podem ser bem trabalhosos e, até mesmo, mais complicados do que o algoritmo de desenho, propriamente dito. Testes de planaridade [76] são exemplos típicos dessa situação.

Um estudo introdutório sobre a área de desenho de grafos, e uma análise minuciosa dos principais algoritmos de desenho ortogonal, onde as arestas do grafo são representadas por linhas verticais e horizontais, e os vértices são posicionados em um grid de coordenadas inteiras, são apresentados neste trabalho. Vários detalhes de implementação, e seus principais obstáculos, são abordados e discutidos. Implementações desses algoritmos também são fornecidas através de uma biblioteca desenvolvida sob o paradigma de programação genérica [21].

Em poucas palavras, o objetivo da programação genérica é tornar um *software* independente das estruturas de dados sobre as quais ele opera. Para isso, são definidos *conceitos* que modelam as abstrações de determinado domínio. Dessa forma, é possível criar mapeamentos entre implementações de algoritmos e conceitos, sem a necessidade de estabelecer estruturas de dados específicas. Uma consequência direta dessa estratégia é a reutilização de componentes, visto que uma mesma estrutura de dados pode ser mapeada a diferentes conceitos.

Um aspecto importante na programação genérica é o desempenho. É comum a existência de restrições de tempo de execução em centenas de aplicações reais. Portanto, é imprescindível que implementações genéricas sejam tão eficientes quanto implementações especializadas. Felizmente, o mecanismo de *templates* da linguagem de programação C++ [108], fundamental para o desenvolvimento de componentes genéricos, tem resolução de tipos em tempo de compilação. Conseqüentemente, bibliotecas desenvolvidas sob o paradigma de programação genérica são freqüentemente mais rápidas que bibliotecas desenvolvidas sob o paradigma convencional de orientação a objetos [65].

A biblioteca desenvolvida neste trabalho é altamente flexível e extensível. Sua lista de adjacência, estrutura de dados que representa o grafo, permite a parametrização dos vértices, arestas e outras propriedades do grafo. Adicionalmente, a arquitetura da biblioteca também permite que estruturas de dados alternativas possam substituir a lista de adjacência padrão, uma característica decorrente da utilização de programação genérica. Foram realizados testes de desempenho, baseados em comparações com outras bibliotecas, para várias operações da lista de adjacência, como inserção e remoção de vértices e arestas, nos quais foram obtidos resultados excelentes.

As implementações de algoritmos da biblioteca são parametrizadas pela representação do grafo sobre o qual elas operam, estruturas de dados que correspondem a mapas de atributos dos vértices e arestas, políticas que modelam partes do comportamento interno dos algoritmos e dados que devem ou não ser armazenados durante a execução. Alguns algoritmos *tradicionais* de grafo, como de busca e caminho mínimo, foram submetidos a testes de desempenho e também apresentaram bons resultados, quando comparados a implementações equivalentes de outras bibliotecas. Além disso, a arquitetura modular da biblioteca facilita a adição de novos algoritmos e extensões sobre os já existentes.

O trabalho é organizado da seguinte forma: inicialmente, o capítulo 2 faz uma revisão geral da terminologia de grafos, algoritmos tradicionais e os principais fundamentos da área de desenho de grafos; em seguida, no capítulo 3, são analisados e discutidos detalhadamente três algoritmos de desenho ortogonal de grafos; uma descrição geral da biblioteca desenvolvida, incluindo aspectos e decisões de implementação, assim como das principais funcionalidades, é feita no capítulo 4; finalmente, os resultados obtidos, relacionados ao desempenho e aos desenhos gerados com implementações da biblioteca, e as conclusões finais são apresentadas, respectivamente, nos capítulos 5 e 6.

Capítulo 2

Desenho de Grafos

Existem inúmeras categorias de grafos. Cada uma delas está associada a um conjunto de características combinatoriais específicas. Em um desenho de um grafo, é normalmente desejável que essas informações combinatoriais sejam transmitidas eficientemente [30]. Da mesma forma que um bom diagrama pode auxiliar o entendimento de um sistema, um diagrama ruim pode torná-lo confuso.

Os fundamentos da área de desenho de grafos são introduzidos neste capítulo. Metodologias de desenho, assim como os principais parâmetros envolvidos, são discutidos. Uma taxonomia de classes de grafos e funções que as mapeiam entre si é apresentada. Porém, antes de aprofundar no assunto, é necessário uma revisão básica sobre grafos e algoritmos de grafos.

2.1 Revisão sobre Grafos e Algoritmos

Os principais termos utilizados ao longo deste trabalho são listados abaixo. Em seguida, alguns algoritmos clássicos de grafos são revisados. A nomenclatura adotada é baseada em [30, 102].

Definição 2.1 Um grafo $G = (V, E)$ consiste de um conjunto finito de V vértices e E arestas, pares não-ordenados de vértices (u, w).

Definição 2.2 Uma aresta (u, w) com $u = w$ é chamada de *auto-ciclo* (*self-loop*). Se a aresta ocorre mais de uma vez em E, ela é uma *aresta paralela* (figura 2.1).

Definição 2.3 Um grafo *simples* é um grafo que não possui auto-ciclos nem arestas paralelas.

Definição 2.4 Os *vértices-finais* ou *extremidades* de uma aresta $e = (u, w)$ são os vértices u e w. O vértice u é chamado de *adjacente* ao vértice w e a aresta e é chamada de *incidente* aos vértices u e w (figura 2.1).

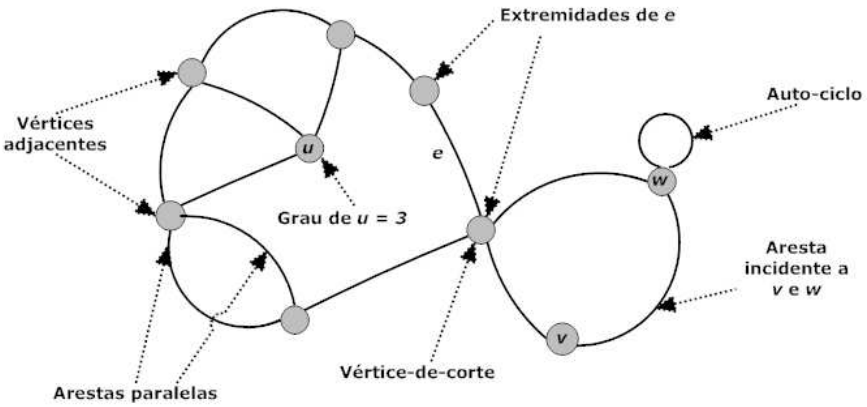

Figura 2.1: Grafo não-direcionado conectado, mas não-biconectado.

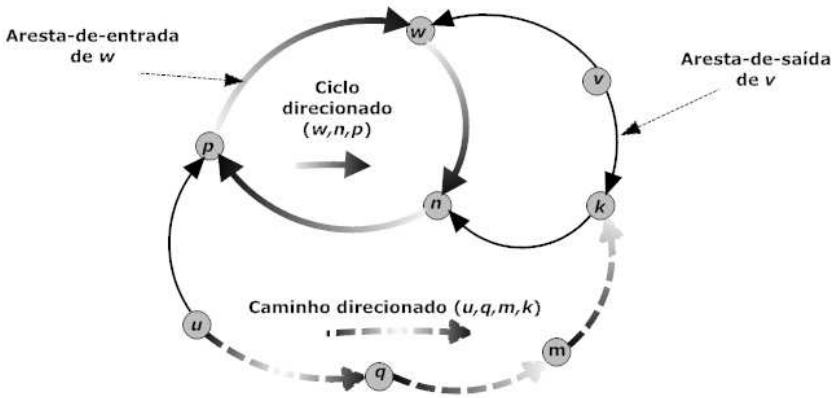

Figura 2.2: Grafo direcionado com ciclo.

Definição 2.5 O(s) *vizinho(s)* de um vértice w são os vértices adjacentes a w.

Definição 2.6 O *grau* de um vértice w é o número de vizinhos que ele possui (figura 2.1).

Definição 2.7 Um grafo *direcionado* (dígrafo) é aquele onde os elementos de E, chamados de arestas *direcionadas*, são pares ordenados de vértices. Um grafo *bidirecionado* é um grafo direcionado no qual, para cada aresta $e = (u, w)$, está presente também sua *inversa* $e = (w, u)$ (figura 2.2).

Definição 2.8 Uma *aresta-de-saída* de um vértice v é uma aresta direcionada $e = (u, w)$, tal que $v = u$. Uma *aresta-de-entrada* de um vértice v é uma aresta direcionada $e = (u, w)$, tal que $v = w$ (figura 2.2).

Definição 2.9 Um *caminho* em um grafo $G = (V, E)$ é uma sequência $(v_1, v_2, ..., v_h)$ de vértices distintos de G, tal que $(v_i, v_{i+1}) \in E$ para $1 \leq i \leq h-1$. Um caminho é um *ciclo* se $(v_h, v_1) \in E$. Um grafo é *acíclico* se não possui ciclos (figura 2.2). Em um grafo direcionado, um caminho é chamado de *caminho direcionado* e um ciclo é chamado de *ciclo direcionado*.

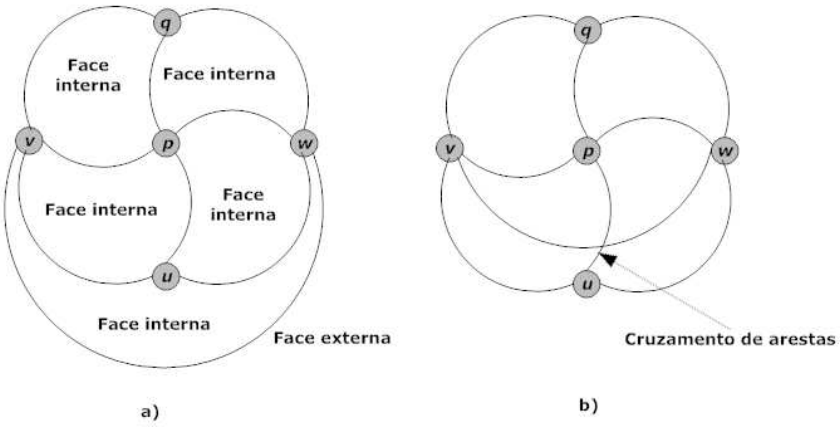

Figura 2.3: Desenhos de um grafo planar: a) desenho planar; b) desenho não-planar.

Definição 2.10 Um grafo é *conectado* se existe um caminho entre u e w para qualquer par (u, w) de vértices (figura 2.1).

Definição 2.11 Um *vértice-de-corte* (*cutvertex*) de um grafo G é um vértice tal que sua retirada desconecta G. Um grafo conectado que não possui vértices de corte é chamado de *biconectado* (figura 2.1).

Definição 2.12 Uma *rede* é um grafo direcionado ou bidirecionado no qual os vértices são chamados *nós* e as arestas são chamadas *arcos*. As *fontes* (*source*) de uma rede são os nós produtores de fluxo. Os *sumidouros* (*sink*) são os nós consumidores de fluxo.

Definição 2.13 Um *desenho* Γ de um grafo (dígrafo) G é uma função que mapeia cada vértice v a um ponto distinto $\Gamma(v)$ e cada aresta (u, v) a um curva simples de Jordan $\Gamma(u, v)$, com pontos finais em $\Gamma(u)$ e $\Gamma(v)$ - arestas direcionadas são normalmente desenhadas como setas.

Definição 2.14 Um desenho Γ de G é *planar* se não há cruzamento de arestas distintas. O grafo G é dito *planar* se ele admite um desenho planar (figura 2.3).

Definição 2.15 Um *desenho planar* particiona o plano em regiões conectadas chamadas de *faces*. A face ilimitada é chamada de *face externa*, enquanto que as outras são chamadas de *faces internas* (figura 2.3).

Definição 2.16 Um *embutimento* (planar) de um grafo é uma classe de equivalência de desenhos planares descrita pela ordem circular, em sentido horário, dos vizinhos de cada vértice. A figura 2.4 ilustra três desenhos planares de um mesmo grafo, sendo que os dois da esquerda possuem o mesmo embutimento planar.

Um algoritmo de grafos clássico e versátil é o de busca em profundidade (DFS - *Depth First Search*). Além de ser empregado para tarefas simples, como de visitar todos os vértices de um

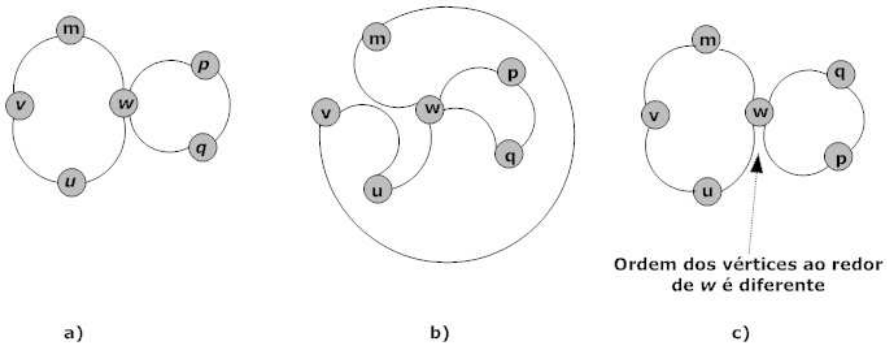

Figura 2.4: Embutimentos planares de um grafo - a) e b) são iguais; c) é diferente.

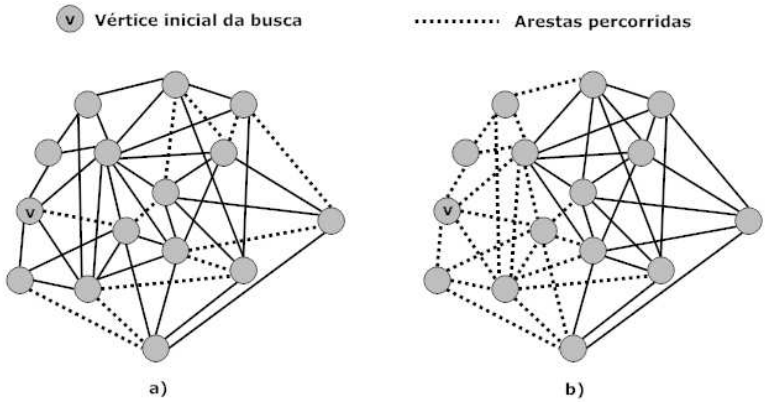

Figura 2.5: Buscas em um grafo: a) DFS; b) BFS.

grafo, ele também constitui a base de algoritmos mais complexos, como o de biconectividade [102]. Grande parte desses algoritmos clássicos, aqui chamados de *tradicionais*, são conhecidos e estudados há bastante tempo [103]. Muitos deles são peças importantes no desenvolvimento de algoritmos de desenho de grafos. A listagem a seguir revisa, brevemente, alguns algoritmos tradicionais e os problemas aos quais eles estão associados. Explicações detalhadas são encontradas em [102, 46, 19, 57].

- **Busca**
  O problema de busca consiste em explorar e examinar os vértices e arestas do grafo. Algoritmos comuns para essa tarefa são:

  → DFS (*Depth First Search*) - Percorre o grafo em profundidade. Equivalente a um soldado visitando áreas de território inimigo, tentando alcançar pontos de distância aleatória a cada movimento. Na figura 2.5-a, as arestas já percorridas até determinado momento são ilustradas em pontilhado;

  → BFS (*Breadth First Search*) - Percorre o grafo em largura. Equivale a um exército visitando áreas de território inimigo homogeneamente, tentando alcançar pontos de mesma distância a cada movimento. Novamente, as arestas pontilhadas da figura 2.5-b são aquelas que já foram percorridas até determinado momento.

  Os algoritmos de busca percorrem os vértices do grafo em uma *ordem* específica. A partir dessa ordem é possível construir uma *árvore*, da qual podem ser obtidas muitas informações combinatoriais sobre o grafo. Arestas do grafo que estão presentes na árvore são chamadas

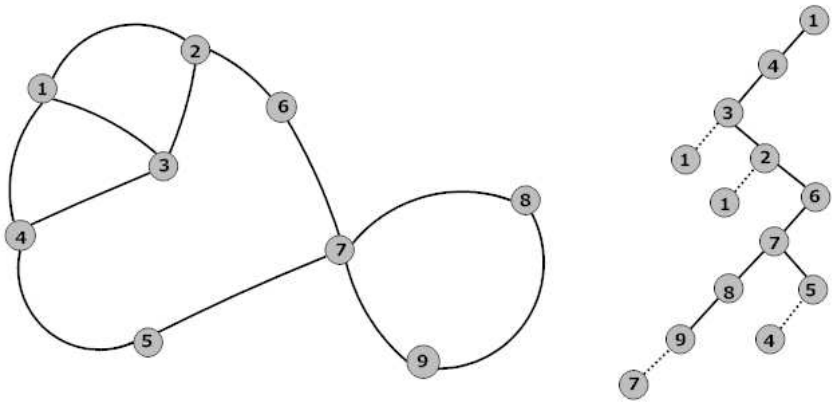

Figura 2.6: Grafo e uma de suas árvores DFS (arestas-de-retorno em pontilhado).

arestas-de-árvore, enquanto que aquelas que não estão presentes são chamadas de *arestas-de-retorno*. O vértice w que leva a um vértice v em uma árvore de busca é chamado de *pai* de v. O vértice u, de menor número de ordem, seguido de uma ou mais arestas-de-árvore e uma aresta-de-retorno a partir de v é chamado de *alcance* de v. A figura 2.6 ilustra um grafo e uma de suas possíveis árvores DFS. Nela, as arestas-de-retorno são mostradas em linhas pontilhadas. A tabela 2.1 mostra os valores de ordem, os pais e os alcances dos vértices.

Tabela 2.1: Propriedades da árvore DFS da figura 2.6.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Ordem	1	4	3	2	9	5	6	7	8
Pai	-	3	4	1	7	2	6	7	8
Alcance	1	1	1	1	4	4	4	7	7

- **Caminho Mínimo**
  Também conhecido como caminho de menor custo, este é o problema de se encontrar o "menor"caminho entre o vértice v e um vértice w no grafo. O significado preciso de "menor"depende do contexto da aplicação. Ele pode ser simples e designar o caminho com o menor número de arestas, ou pode levar em conta atributos adicionais. No primeiro caso, uma busca BFS também pode resolver o problema. Para o segundo caso, os seguintes algoritmos podem ser utilizados:

  → Caminho Mínimo de Dijkstra;
  → Caminho Mínimo de Bellman-Ford.

  Ambos algoritmos adotam idéias similares para fazer as comparações entre menores caminhos. O algoritmo de Dijkstra opera localmente em cada vértice do grafo, enquanto que o algoritmo de Bellman-Ford se baseia em características gerais que afetam um conjunto de vértices. Em situações onde os atributos das arestas assumem valores negativos e o grafo contém ciclos, o algoritmo de Bellman-Ford deve ser utilizado.

- **Fluxo Máximo**
  Dada uma rede com uma fonte s e um sumidouro t, o problema consiste em encontrar um fluxo de s para t com o máximo valor. Algoritmos comuns para essa tarefa são:

→ Fluxo Máximo por Incremento de Caminhos - A idéia desse algoritmo é encontrar caminhos pela rede, e, ao longo dele, aumentar o fluxo sem extrapolar a capacidade dos arcos. Quando não existirem mais tais caminhos, o fluxo atingiu seu valor máximo;

→ Fluxo Máximo por Formação de Pré-Fluxo - Esse algoritmo é significativamente diferente do anterior. A idéia é transportar o maior fluxo possível da fonte para os nós intermediários, e deles para o sumidouro da rede. O nó que recebe um pré-fluxo é responsável por entregá-lo ao próximo nó ou devolvê-lo ao nó anterior. Esse processo se repete até que o fluxo chegue no sumidouro.

- **Fluxo de Custo Mínimo**
  Quando os custos associados aos arcos da rede são diferentes entre si, é possível que se deseje encontrar o fluxo de menor custo. Naturalmente, faz sentido encontrar o menor fluxo de menor custo apenas se existirem limites inferiores nos arcos da rede. Caso contrário, o menor fluxo seria igual a zero. Um algoritmo comum para essa tarefa é:

  → Fluxo de Custo Mínimo por Cancelamento de Ciclos - O fluxo em uma rede é o de menor custo se e somente se não houver ciclos negativos na rede *residual*. Este algoritmo se baseia diretamente nessa propriedade: ele calcula o fluxo máximo da rede e, em seguida, remove todos seus ciclos negativos. A definição de rede residual é apresentada em detalhes em [102, 19].

2.2 Fundamentos de Desenho de Grafos

Ao desenhar um grafo, é importante que sejam enfatizadas características combinatoriais como planaridade, conectividade, biconectividade, entre outras. Na grande maioria das aplicações, essa transparência pode ser bastante útil para o usuário. Além disso, algumas dessas características são consideradas na elaboração de uma taxonomia de classes e algoritmos de desenho de grafos.

A taxonomia é composta por uma hierarquia de classes de grafos comumente encontrados na literatura [118, 73, 40, 70] e um conjunto de funções que mapeiam grafos entre classes distintas. Entre os principais ganhos advindos da utilização da taxonomia estão a adoção do conceito de *herança*, bastante conhecido em linguagens de programação [108, 23], e a possibilidade de síntese de novas metodologias de desenho, visto que grande parte dos algoritmos são baseados em etapas relativamente independentes nas quais a entrada e a saída são grafos. A figura 2.7 ilustra a taxonomia.

Não há uma forma clara e objetiva de classificar o "melhor"desenho de um grafo. Um desenho pode ser considerado ótimo em determinado contexto, mas pode não atender aos requisitos de alguma outra aplicação. Portanto, é necessário a especificação de parâmetros bem definidos que descrevam a qualidade do desenho de um grafo. Os principais parâmetros encontrados na literatura são as *convenções*, *restrições* e *critérios estéticos* [30]. As próximas seções abordam, separadamente, cada um deles.

2.2.1 Convenções de Desenho

A *convenção de desenho* corresponde a uma regra básica que precisa ser satisfeita. Em determinado tipo de diagrama, por exemplo, pode ser necessário que todas as setas (arestas

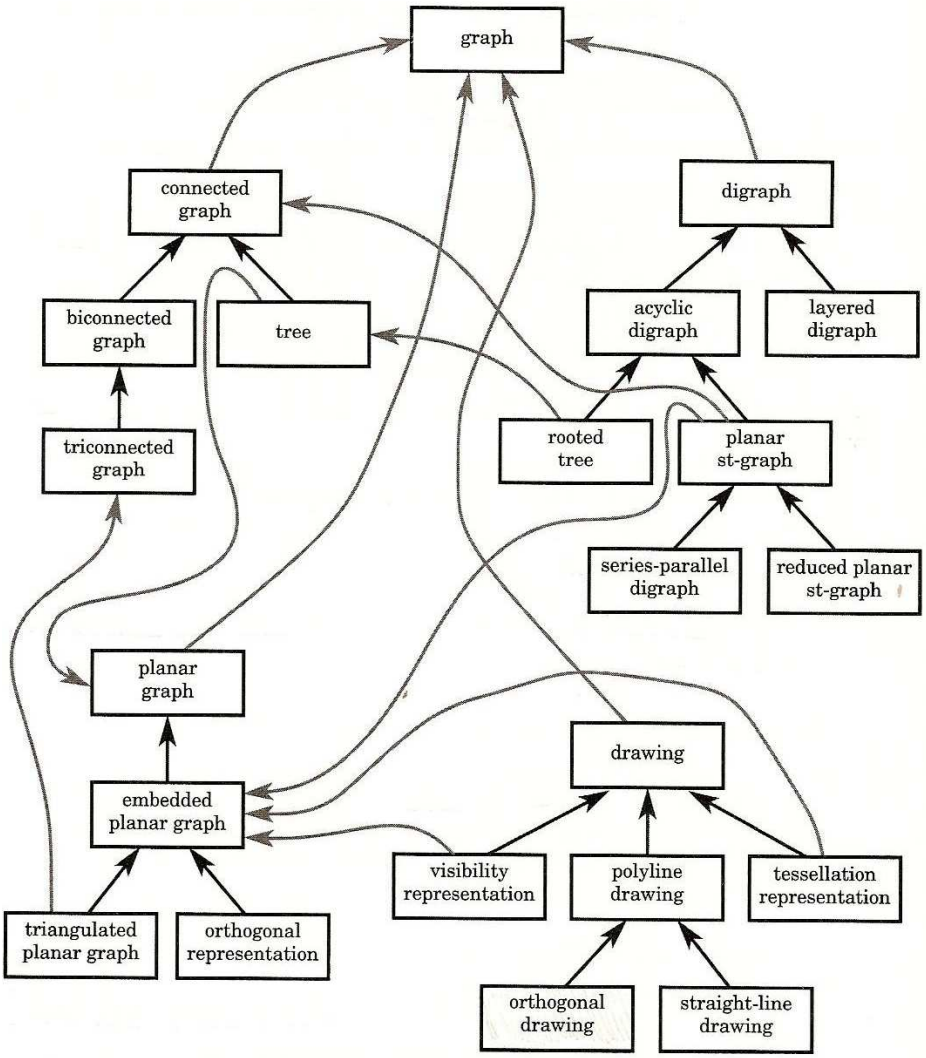

Figura 2.7: Taxonomia de classes e funções [30].

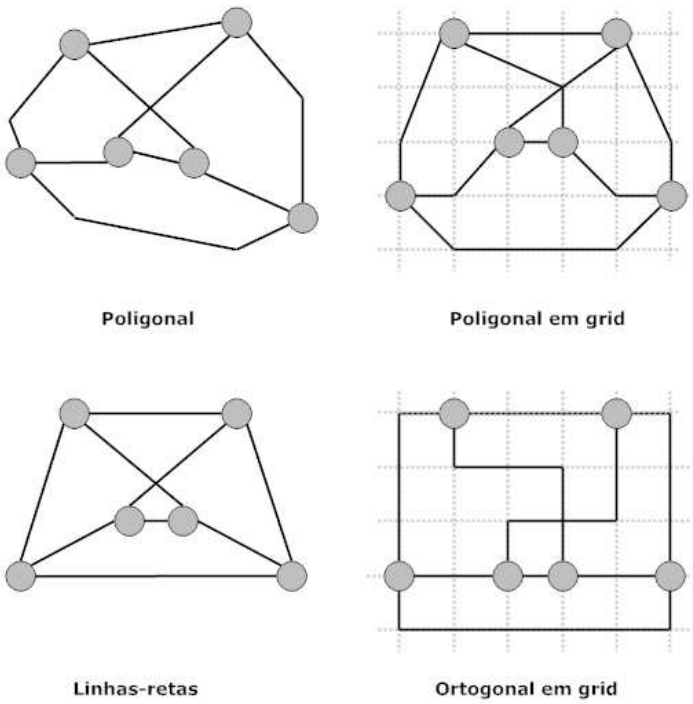

Figura 2.8: Convenções de desenho.

direcionadas) sejam desenhadas da esquerda para direita - note que isso nem sempre é possível. Algumas convenções bastante conhecidas [30] são ilustradas na figura 2.8 e descritas em seguida:

Poligonal - Cada aresta do grafo é desenhada como uma cadeia poligonal;

Linhas-retas - Cada aresta do grafo é desenhada como segmentos retos;

Ortogonal - Cata aresta do grafo é desenhada como uma cadeia poligonal de segmentos horizontais e verticais;

Grid - Vértices, cruzamentos de arestas e dobras de arestas possuem coordenadas inteiras;

Planar - Não há cruzamento de arestas.

As convenções linhas-retas e ortogonal são especializações da convenção poligonal, já que nos três casos as arestas são cadeias poligonais. Também é possível que um desenho adote mais de uma convenção. Desenhos de circuitos elétricos, por exemplo, são normalmente desenhados dentro das convenções ortogonal, grid e planar.

2.2.2 Critérios Estéticos

Os *critérios estéticos* especificam propriedades gráficas do desenho que devem ser aplicadas ao máximo possível. Critérios freqüentemente utilizados são os seguintes [27, 97, 109]:

Cruzamentos de Arestas - Os cruzamentos de arestas contribuem negativamente para a inteligibilidade do desenho. Desenhos planares são sempre desejados. Quando isso não é possível, deve-se tentar *minimizar o número de cruzamentos de arestas*;

Área - Quando a convenção grid é adotada, esse critério é de extrema importância, pois os desenhos não podem ser reduzidos ilimitadamente (qualquer aresta precisa ter, pelo menos, uma unidade de medida para seu comprimento). Além disso, aplicações práticas podem conter grafos relativamente grandes, os quais precisam ser exibidos inteiramente na tela do computador. Portanto, a *minimização da área* do desenho deve ser levada em conta sempre que possível;

Há duas formas de definir a área de um desenho: pelo fecho convexo (menor polígono que engloba o desenho) ou pelo menor retângulo com lados horizontais e verticais cobrindo o desenho;

Razão de Aspecto - A razão de aspecto é definida pela razão entre o comprimento do maior lado e o comprimento do menor lado do menor retângulo capaz de englobar o desenho. Assim como a área, a *razão de aspecto também deve ser mantida pequena.* Mesmo um desenho de área pequena pode não ser exibido inteiramente na tela do computador se sua razão de aspecto for grande. O objetivo de manter a razão de aspecto pequena é promover maior flexibilidade de enquadramento de desenhos em telas de formas arbitrárias;

Comprimento Total de Arestas - Assim como no caso da área, deve-se tentar *minimizar a soma total dos comprimentos das arestas* quando a convenção grid é adotada. Nesse caso, o desenho não pode ser reduzido ilimitadamente;

Máximo Comprimento de Aresta - Mais uma vez, caso um desenho que adote a convenção grid possua uma aresta muito grande, ele pode não ser exibido inteiramente na tela do computador. Portanto, se necessário, o *máximo comprimento de uma aresta também deve ser minimizado*;

Comprimento Uniforme de Arestas - Normalmente, a *minimização da variância do comprimento das arestas* é um critério puramente estético. No entanto, ele não deve ser utilizado em situações onde os comprimentos das arestas são proporcionais a algum de seus atributos como custo ou capacidade;

Dobras em Arestas - Para desenhos ortogonais, é importante que o *número de dobras por aresta, assim como o número total de dobras, seja minimizado.* Quando as arestas possuem muitas dobras, o desenho pode ficar confuso e pouco inteligível. Idealmente, a *variância do número de dobras nas arestas também deve ser minimizada*;

Resolução Angular - Em desenhos que adotam a convenção linhas-retas, a *maximização do menor ângulo entre duas arestas incidente a um vértice é especialmente relevante.* Se o grafo possui vértices com grau elevado e esse critério não for levado em conta, o desenho pode ficar pouco inteligível;

Simetria - A *ênfase das simetrias de um desenho do grafo* também pode ser importante em alguns contextos. Existem modelos matemáticos que definem precisamente o que são simetrias em grafos e seus respectivos desenhos [53, 86].

É importante notar a existência de conflitos entre critérios. Isso indica que contemplar simultaneamente alguns deles pode não ser factível. Adicionalmente, a tarefa de lidar com vários critérios ao mesmo tempo pode ser bastante difícil. Conseqüentemente, as metodologias de desenho de grafos estabelecem relações de precedência e adotam uma abordagem de divisão

do processo em subetapas, nas quais cada algoritmo resolve um problema específico de forma independente.

2.2.3 Restrições

Enquanto que as convenções e os critérios estéticos estão associados ao desenho do grafo como um todo, as *restrições* se aplicam apenas aos subgrafos/subdesenhos de G. As restrições usualmente utilizadas são listadas abaixo [85, 111]:

Centralização - Posicionar um dado vértice no centro do desenho;

Externalização - Posicionar um dado vértice no limite exterior do desenho;

Agrupamento - Posicionar um conjunto de vértices próximos uns dos outros;

Esquerda-Direita (de cima-para baixo) - Desenhar determinado caminho horizontalmente alinhado da esquerda para direita (verticalmente alinhado de cima para baixo);

Forma - Desenhar determinado subgrafo em uma forma predefinida.

Em conjunto com as convenções, critérios estéticos, o contexto da aplicação e eficiência (naturalmente), as restrições formam os parâmetros fundamentais associados a algoritmos de desenho de grafos [30].

2.3 Metodologias de Desenho

Os parâmetros de um desenho, assim como os requisitos de uma aplicação, variam constantemente. Muitos deles são contraditórios entre si e não podem ser contemplados simultaneamente. No entanto, mesmo algoritmos de desenho com características diferentes podem possuir etapas em comum. A partir da combinação de etapas intermediárias dos algoritmos é possível elaborar inúmeras metodologias de desenho. Consequentemente, o processo de criação de novos algoritmos se torna mais simples e bem estruturado. Metodologias bem conhecidas são apresentadas nas próximas seções.

2.3.1 Metodologia da Topologia-Forma-Métrica

Esta metodologia, originalmente proposta em [110] reapresentada em [26, 111], possui como principal fundamento a caracterização de um desenho ortogonal a partir de três propriedades:

Topologia - Está associada à natureza do grafo, particularmente ao seu embutimento planar. Dois desenhos ortogonais possuem a mesma topologia se um pode ser obtido a partir do outro apenas através de deformações contínuas, que não alteram as sequências das arestas contornando as faces do desenho. Os dois desenhos da esquerda na figura 2.4 possuem a mesma topologia;

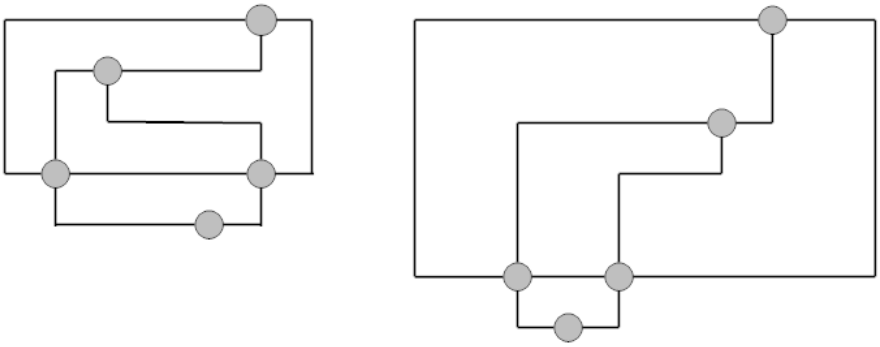

Figura 2.9: Dois desenhos de mesma forma.

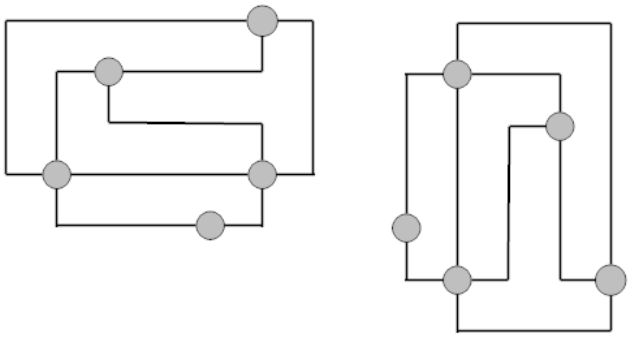

Figura 2.10: Dois desenhos de mesma métrica.

Forma - Representa a forma do desenho. Dois desenhos ortogonais possuem a mesma forma se eles têm a mesma topologia e um pode ser obtido a partir do outro modificando apenas o comprimento dos segmentos que representam as arestas do grafo, sem alterar os ângulos formados por elas, independentemente da sequência em que eles ocorrem. A figura 2.9 ilustra dois desenhos de mesma forma;

Métrica - Caracteriza o enquadramento de determinada forma em uma certa área. Desenhos ortogonais possuem as mesmas métricas se um puder ser obtido a partir do outro através de operações de rotação ou translação. A figura 2.10 ilustra dois desenhos de mesma métrica.

A relação hierárquica existente entre essas três propriedades (se dois grafos têm a mesma métrica, então eles têm a mesma forma e, consequentemente, a mesma topologia) sugere um algoritmo dividido em etapas. Precisamente, cada propriedade é associada à representação intermediária resultante de cada uma das seguintes etapas [26]:

Planarização → Determina a topologia do desenho. O objetivo desta etapa é eliminar todos os cruzamentos de arestas, os quais são substituídos por vértices falsos;

Ortogonalização → Determina a forma do desenho. Os vértices ainda não possuem coordenadas, mas arestas são associadas a valores que indicam sua orientação ao redor do vértice e a sequência de dobras que elas terão no desenho final;

Compactação → Determina a métrica do desenho. Nesta etapa, são atribuídas coordenadas

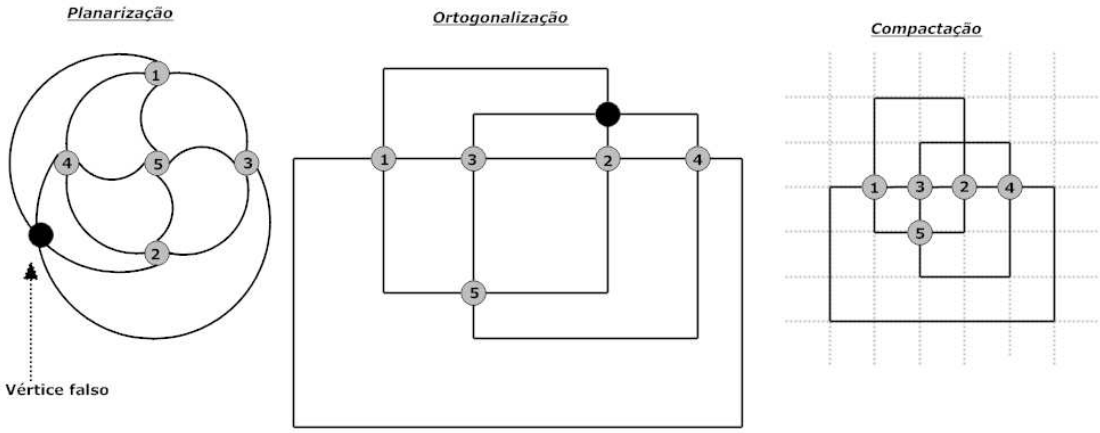

Figura 2.11: Metodologia topologia-forma-métrica.

aos vértices do grafo e dobras das arestas. Os vértices falsos adicionados na planarização também são removidos do grafo.

A figura 2.11 ilustra as transformações resultantes de um algoritmo baseado na metodologia topologia-forma-métrica. Inicialmente, um embutimento planar é construído. Em seguida, o embutimento é ortogonalizado e, finalmente, desenhado em um grid.

2.3.2 Metodologia da Hierarquia

Como o próprio nome diz, esta metodologia é indicada para desenhos que correspondem a uma estrutura hierárquica como uma relação *é-um* na engenharia de software ou uma pilha de chamadas de funções de um compilador. Proposta por diferentes autores em [109, 44, 115], esta metodologia é composta pelas seguintes etapas:

Atribuição de Nível (Layer) → Um dígrafo sem ciclos é transformado em um *dígrafo propriamente nivelado*, onde os vértices de G são atribuídos a níveis $L_1, L_2, ..., L_h$ (um vértice no nível L_i tem coordenada-y igual a i), tais que, se (u, v) é uma aresta com $u \in L_i$ e $v \in L_j$, então $i = j + 1$. Durante essa etapa, vértices falsos são adicionados ao grafo;

Redução de Cruzamentos de Arestas → É atribuída ordem aos vértices de cada nível de forma que o número de cruzamentos de arestas seja o menor possível;

Atribuição da Coordenada-x → Os vértices falsos são removidos do grafo e os outros recebem as coordenadas-x, em função da ordenação da etapa anterior. Em seguida, as arestas são desenhadas com segmentos de linhas retas. A figura 2.12 ilustra um desenho gerado por esta metodologia.

2.3.3 Metodologia da Visibilidade

Esta é uma metodologia geral para desenhos que adotam a convenção poligonal. Ela foi originalmente proposta em [32] e refinada em [33]. A listagem abaixo descreve suas etapas:

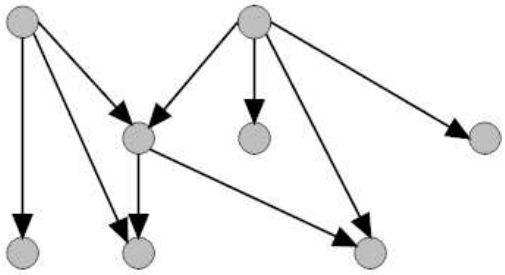

Figura 2.12: Desenho gerado por metodologia hierarquica.

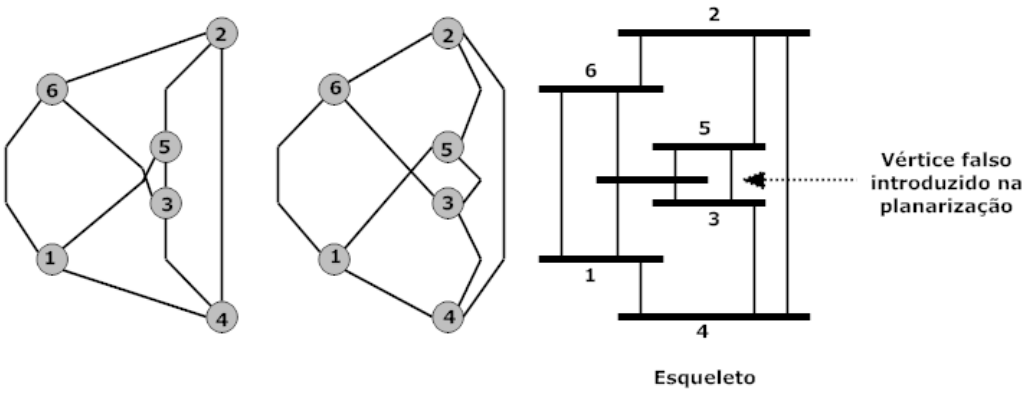

Figura 2.13: Desenhos gerados por algoritmos distintos a partir de um esqueleto comum.

Planarização → Exatamente a mesma daquela descrita na metodologia da topologia-formamétrica;

Visibilidade → Cada vértice é mapeado em um segmento horizontal e cada aresta é mapeada em um segmento vertical, constituindo o *esqueleto* do desenho do grafo;

Substituição → Os segmentos horizontais e verticais são substituídos, respectivamente, por vértices e cadeias poligonais que representam as arestas. Algoritmos diferentes se baseiam em estratégias diferentes para fazer a substituição. A figura 2.13 ilustra desenhos gerados por dois algoritmos distintos a partir do mesmo esqueleto.

2.3.4 Metodologia de Adições Sucessivas

Esta também é uma metodologia geral para desenhos que adotam a convenção poligonal. A idéia central é adicionar vértices e arestas ao grafo para que ele contenha propriedades estruturais fortes, bem definidas e que possibilitem um algoritmo de desenho mais simples. As principais etapas desta metodologia são [30]:

Planarização → Exatamente a mesma daquela descrita na metodologia da topologia-formamétrica;

Adições → Vértices e arestas são adicionados ao grafo para se obter um embutimento planar onde todas as faces são triangulares. Ou seja, um *maximal* grafo planar é construído;

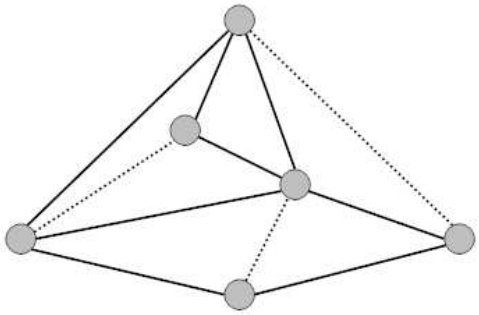

Figura 2.14: Desenho gerado pela metodologia de adições sucessivas (arestas em pontilhado são removidas).

Triangulação → O grafo triangulado é desenhado e, em seguida, os vértices e arestas falsos são removidos. A figura 2.14 ilustra um desenhado gerado por esta metodologia.

2.3.5 Metodologia da Numeração

Apesar de não ser uma metodologia formalmente estabelecida, ela é claramente visível em alguns algoritmos de desenho ortogonal de grafos [93, 37]. Primeiramente, os vértices do grafo são numerados a partir de um critério específico. Um algoritmo comumente utilizado nesta etapa é o de numeração-st, apresentado no capítulo 3. Em seguida, cada um dos vértices é posicionado em linhas do grid de acordo com sua numeração, e colunas são alocadas para cada uma de suas arestas. As principais etapas são resumidas abaixo:

Numeração → Os vértices do grafo são numerados a partir de um critério, como uma numeração-st, por exemplo.

Embutimento no grid → A partir da numeração da etapa anterior, os vértices e arestas são embutidos em um grid, de acordo com regras específicas de cada algoritmo. A figura 2.15 ilustra desenhos de um mesmo grafo gerados por algoritmos de numeração distintos.

2.3.6 Metodologia Direcionada por Força

A grande vantagem desta metodologia é que ela possui um modelo conceitual intuitivo e simples de implementar. Normalmente, técnicas direcionadas por força são utilizadas para desenhos que adotam a convenção poligonal. Seus constituintes fundamentais são [30]:

- Um modelo matemático para definir um sistema de forças associado ao grafo. Um exemplo seria o de modelar as arestas do grafo como sendo molas, nas quais seus comprimentos equivalem às distâncias de menor número de arestas entre dois vértices u e v. Neste caso, a força induzida corresponde a distância euclideana entre os vértices u e v menos o comprimento da mola;
- Uma técnica de otimização que garante o mínimo de energia do sistema e, conseqüentemente, uma configuração para as arestas do grafo.

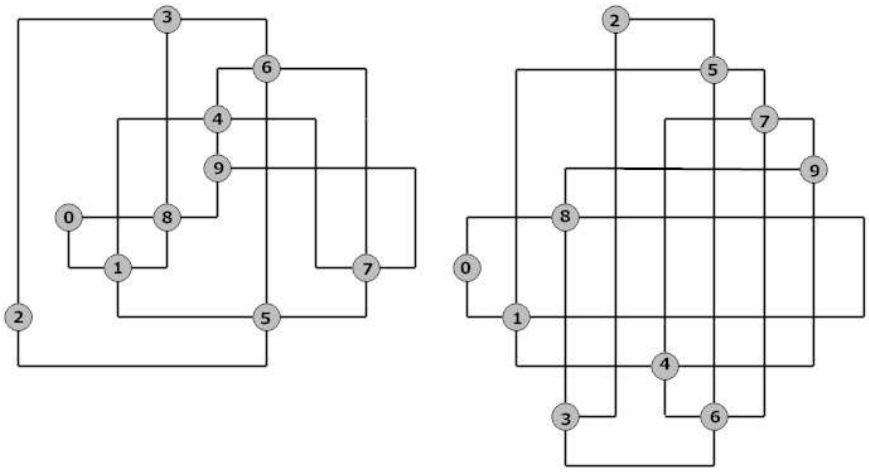

Figura 2.15: Desenhos de um mesmo grafo gerados por algoritmos de numeração distintos.

Capítulo 3

Algoritmos de Desenho

Um desenho de um grafo pode ser gerado por diferentes abordagens, as quais estabelecem relações de precedência variadas entre critérios estéticos. Portanto, o contexto da aplicação deve sugerir características desejadas para o desenho e o melhor algoritmo a ser utilizado.

Diagramas ortogonais são utilizados em uma ampla gama de aplicações [26]. Usualmente, desenhos de grafos que correspondem a este tipo de diagrama também adotam a convenção grid, na qual as coordenadas de vértices e dobras de arestas são números inteiros. Muitos algoritmos de desenho que seguem esse modelo são conhecidos [29].

Um exemplo clássico de aplicação prática de diagramas ortogonais é a confecção de circuitos eletrônicos, no qual os componentes devem ser posicionados em uma placa sob restrições de uma área específica, com o mínimo número de cruzamento de arestas [107].

Este capítulo apresenta, detalhadamente, diferentes abordagens para geração de desenhos ortogonais em grid. O primeiro algoritmo segue a metodologia topologia-forma-métrica e garante resultados muito bons sob o ponto de vista de alguns critérios estéticos. Os algoritmos seguintes não produzem desenhos com a mesma qualidade do primeiro, mas são significativamente mais rápidos e mais fáceis de serem implementados.

3.1 Algoritmo *Giotto*

Um dos algoritmos de desenho ortogonal em grid mais eficiente em termos de vários critérios estéticos [31] segue a metodologia topologia-forma-métrica. As etapas de planarização, ortogonalização e compactação deste algoritmo, apresentado pela primeira vez em [110], são detalhadamente explicadas nas próximas seções.

3.1.1 Planarização

A planarização de um grafo é construída a partir de sucessivas aplicações de uma *operação de planarização*, a qual possui como princípio básico a substituição de cruzamentos de arestas

por vértices falsos. Conseqüentemente, esta etapa estabelece a precedência da minimização dos cruzamentos de arestas sobre quaisquer outros critérios estéticos.

A topologia do grafo, diretamente relacionada ao número de dobras do desenho, também é determinada nesta etapa, através da criação de um embutimento planar. Esse é o motivo pelo qual grafos planares também devem ser submetidos à etapa de planarização.

O objetivo principal da planarização é reduzir ao máximo o número de cruzamento de arestas. Este problema é equivalente ao problema do *maximum* subgrafo planar, bastante estudado em [79, 77, 82, 92]. Ambos são problemas *NP-hard*, o que leva grande parte dos algoritmos de planarização a utilizarem heurísticas.

Uma heurística bastante comum para a construção de um *maximal* subgrafo planar é adicionar as arestas do grafo uma a uma, e a cada iteração realizar um teste de planaridade. Caso o grafo deixe de ser planar, a última aresta inserida é removida e classificada como não-planar. Essa heurística é descrita pelo procedimento 1.

O procedimento 2 descreve um algoritmo de planarização baseado na heurística do parágrafo anterior. Inicialmente, as arestas não-planares são identificadas e um embutimento planar do grafo é construído. Em seguida, cada aresta não-planar é inserida no embutimento de forma a minimizar o cruzamento de arestas.

Procedimento 1 *Maximal* Subgrafo Planar

Entrada: Grafo G.
Saída: *Maximal* subgrafo planar G' de G.

1: Crie um grafo G' composto apenas pelos vértices de G.
2: ***Para*** toda aresta $e \in G$ ***faça***
3: ***Se*** o grafo obtido com sua inserção em G' for planar ***então***
4: Adicione e a G' e classifique-a como planar.
5: ***Senão***
6: Rejeite e e classifique-a como não-planar.
7: ***Fim Se***
8: ***Fim Para***

Procedimento 2 Planariza Grafo

Entrada: Grafo G.
Saída: Embutimento planar G' de G.

1: Construa um *maximal* subgrafo planar S de G utilizando o procedimento 1.
2: Construa um embutimento planar do subgrafo G' e seu dual.
3: Adicione as arestas não-planares a G' minimizando o número de cruzamentos da seguinte forma. Para cada aresta (u, v):

- Encontre um caminho de custo mínimo no grafo dual do embutimento atual G' entre as faces incidentes a u e v.
- Adicione esta aresta não-planar (com o vértice falso), atualize G' e seu dual.

Apesar da aparência simples, há alguns pontos, abordados nas próximas seções, que tornam o algoritmo 2 bastante difícil.

Teste de Planaridade

A classificação de arestas em planares e não-planares exige um teste de planaridade no grafo. Alguns dos primeiros testes de planaridade encontrados na literatura utilizam o paradigma de *dividir para conquistar* [24, 69, 103]. O principal problema destes algoritmos está no fato da complexidade de tempo ser muito alta (não-linear).

O primeiro teste de planaridade com complexidade de tempo linear foi apresentado por Hopcroft e Tarjan em [76]. No entanto, o algoritmo, conhecido como método da adição de arestas, é bastante complexo. Alguns livros sobre teoria de grafos [57, 64, 70] mencionam o algoritmo, mas devido ao grau de dificuldade, não tentam prová-lo. Outros livros [87, 68, 40] fazem opção por um algoritmo de planaridade mais simples [50], mas menos eficiente. Di Battista et. al [29] comentam sobre a séria limitação dos, até então, atuais algoritmos de teste de planaridade para aplicação em sistemas reais, devido ao alto grau de dificuldade de entendimento e implementação.

Outros algoritmos lineares de teste de planaridade foram desenvolvidos. O algoritmo de Booth e Lueker [41], conhecido como método da adição de vértices utiliza uma árvore para manter informações necessárias sobre determinadas partes do grafo a cada adição de vértice.

Árvores DFS (*Depth-first search*) foram utilizadas pela primeira vez na caracterização de planaridade de um grafo por De Fraysseix e Rosentiehl [49, 48]. Boyer e Myrvold [42] também apresentaram um algoritmo interessante e eficiente baseado em propriedades de árvores DFS. A operação fundamental é a adição de arestas (especificamente, arestas-de-retorno) a um embutimento planar parcial do grafo. Componentes biconectados são mantidos e criados ao longo do processamento. Sempre que necessário, eles são invertidos (sofrem uma operação de *flip*). A não-planaridade é detectada quando não há possibilidade de adicionar uma nova aresta-de-retorno ao embutimento parcial sem quebrar sua planaridade.

Apesar da evolução de idéias e conseqüente diminuição de complexidade nos algoritmos, os testes de planaridade ainda são difíceis e trabalhosos para serem implementados, quando comparados a algoritmos tradicionais de grafos como de busca ou caminho mínimo.

Construção do Dual

Dado um embutimento planar de um grafo planar, G, seu dual (geométrico), G^*, pode ser construído pelo seguinte conjunto de passos [118]:

- Para cada face f de G, adicione um novo vértice v^* em G^*.
- Para cada aresta e de G, adicione uma nova aresta e^* em G^* conectando os dois vértices de G^* correspondentes as duas faces de G incidentes a e.

A estratégia mencionada é precisa e simples. No entanto, leva em consideração aspectos geométricos do grafo, não disponíveis em representações comuns como listas ou matrizes de adjacência.

Para identificar as faces de um grafo e construir informações de adjacência, uma abordagem conhecida e bastante estudada na área de geometria computacional é a de subdivisões planares

[47]. Existem várias estruturas de dados capazes de representar subdivisões planares de um grafo eficientemente. A DCEL (*Doubly Connected Edge Lists*)[91] e a *Quad-Edge* [74], uma espécie de variante da *Winged-Edge* [34], são algumas das mais conhecidas. Entre essas, a *Quad-Edge* é a única capaz de armazenar simultaneamente informações tanto do grafo primal quanto do dual, o que a torna bastante atraente. Porém, o modelo original mantém essas informações de forma unificada, o que impossibilita a distinção explícita das faces e vértices dos grafos primal e dual.

As estruturas de dados do parágrafo anterior são muito interessantes e ricas, pois agregam um conjunto extenso de informações sobre características topológicas do grafo. Apesar de atenderem aos requisitos da etapa de adição de arestas não-planares no procedimento 2, elas são difíceis de se implementar e são mais poderosas do que é realmente necessário. Portanto, o próximo parágrafo apresenta uma idéia mais simples, porém suficiente para resolver esse problema.

O principal ponto a ser observado é que o embutimento planar já contém as informações fundamentais para a definição das faces do grafo. Através de uma navegação um pouco mais elaborada sobre as arestas do grafo, é possível construir seu dual e computar as informações de adjacência necessárias ao procedimento 2. As arestas devem ser percorridas no sentido que define as faces do grafo, o que caracteriza a noção de *ciclos faciais*. O procedimento 3 [84] implementa um algoritmo baseado nessas idéias. Garantindo que ele seja chamado para todas as arestas do embutimento planar sem repetições, o grafo dual pode ser facilmente construído. Cada ciclo facial corresponde a um dos vértices do grafo dual. Além disso, qualquer tipo de informação de correspondência entre vértices e arestas dos grafos dual e primal pode ser computada ao longo deste procedimento.

Procedimento 3 Ciclo Facial

Entrada: Aresta e e um embutimento planar G.
Saída: Ciclo facial que contém a aresta e.

1: $inicio \leftarrow e$.
2: $w \leftarrow$ vértice destino de e.
3: ***Repita***
4: $\quad inv \leftarrow$ aresta inversa de e.
5: $\quad e \leftarrow$ aresta seguinte de inv ao redor de w em sentido horário.
6: $\quad w \leftarrow$ vértice destino de e.
7: ***Enquanto*** e != $inicio$

Caminho Mínimo

Existem vários algoritmos para cálculo do caminho de menor custo. Quando todas arestas do grafo possuírem o mesmo peso, uma busca em largura (BFS) pode ser utilizada. Para os outros casos, os algoritmos de Dijkstra [52] e Bellman-Ford [35, 62] são boas opções. O importante é que exista alguma forma de navegabilidade entre o grafo dual e o primal, já que o caminho mínimo é encontrado no primeiro, e a adição de arestas é feita no segundo.

Caso haja alguma restrição de cruzamento em algum conjunto de arestas de G, é possível modificar o procedimento 2 para inserir essas arestas primeiro. É também necessário atribuir a elas um custo relativamente alto em relação as outras arestas do grafo, evitando que os eventuais caminhos de custo mínimo contenham arestas desse conjunto.

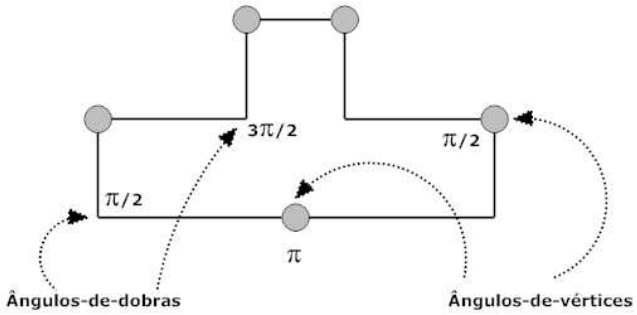

Figura 3.1: Ângulos de um desenho ortogonal.

3.1.2 Ortogonalização

A ortogonalização é responsável por criar uma *representação ortogonal* H de um embutimento planar. Ao fim desta etapa, os vértices ainda não possuem coordenadas, mas cada aresta contém uma lista de ângulos que descreve para que lado serão suas dobras.

Naturalmente, é desejado que o número de dobras em arestas seja o mínimo possível. Para isso, o problema de ortogonalização é transformado em um problema de fluxo de custo mínimo, onde os vértices são *produtores* e as faces são *consumidores*. Todo fluxo gerado pelos vértices deve ser consumido pelas faces e cada unidade de fluxo corresponde a um ângulo de $\pi/2$.

As próximas seções descrevem a modelagem e solução do problema. Maiores informações sobre os resultados apresentados e uma análise mais detalhada podem ser encontradas em [30]. São considerados apenas grafos que possuem vértices de grau máximo quatro. Grafos com vértices de grau maior do que quatro devem sofrer a seguinte simplificação.

- Seja v um vértice de G com grau $d > 4$. Transforma-se v em C_d, o grafo cíclico formado por d vértices. Cada vértice de C_d possui duas arestas conectadas a dois outros vértices do próprio C_d e uma aresta conectada a um vértice u anteriormente conectado a v. Todos os vértices de C_d possuem grau 3 e a modelagem apresentada neste capítulo pode ser aplicada a G.

Representação Ortogonal

Existem dois tipos de ângulos em um desenho ortogonal Γ, como ilustra a figura 3.1.

Ângulos-de-vértices - Formados por duas arestas incidentes a um mesmo vértice.

Ângulos-de-dobras - Formados por segmentos consecutivos de uma mesma aresta.

Duas consequências diretas das definições acima podem ser percebidas. A primeira, bastante óbvia, é que a soma dos ângulos-de-vértices de cada vértice deve ser igual a 2π. A outra, menos trivial, é que para cada face interna f de Γ, a soma dos ângulos-de-vértice e de dobras é igual a $\pi(p-2)$, onde p é o total de ângulos. Para a face externa, a soma é igual a $\pi(p+2)$.

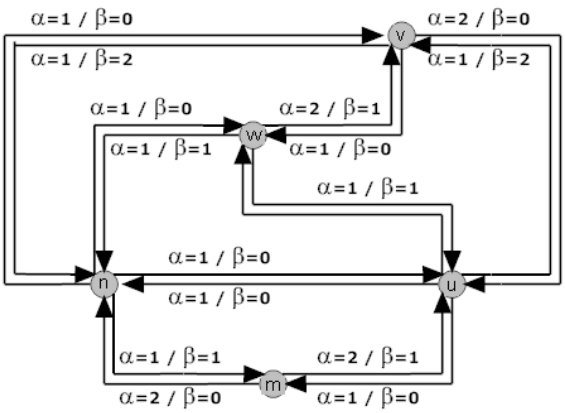

Figura 3.2: Valores de α e β.

Considere um embutimento planar de um grafo bidirecionado G, com vértices de grau máximo quatro. O número total de ângulos-de-vértices dentro de uma face f é designado como $a(f)$. Como G é bidirecionado, para cada aresta $e = (u, v)$, também existe a aresta $e = (v, u)$. A aresta que percorre determinada face f no sentido anti-horário é chamada de *aresta anti-horário*. O conjunto de arestas com origem em um vértice v é denotado por $D(v)$, enquanto que o conjunto de arestas anti-horário da face f é denotado por $D(f)$. Dado um desenho planar ortogonal Γ de G, são definidos os seguintes valores para cada uma de suas arestas. A figura 3.2 ilustra estas definições.

- $\alpha(u, v) \cdot \pi/2$ é o ângulo em um vértice u formado pelo primeiro segmento da aresta (u, v) e o primeiro segmento da próxima aresta anti-horário ao redor de u. Por exemplo, na figura 3.2, o ângulo entre o primeiro segmento da aresta (v, u) e o primeiro segmento da aresta (v, n) é π. Por isso, o valor de α da aresta (v, u) é dois.
- $\beta(u, v)$ é o número de dobras ao longo da aresta (u, v) com o ângulo de $\pi/2$ em seu lado esquerdo. Por exemplo, na figura 3.2, a aresta (v, u) não possui nenhum ângulo de $\pi/2$ em seu lado esquerdo. Conseqüentemente, seu valor de β é zero. Já a aresta (u, v) faz duas curvas à esquerda e possui valor de β igual a dois.

Os valores de α e β são os parâmetros que caracterizam uma representação ortogonal. Desenhos com o mesmos valores desses parâmetros são considerados equivalentes. Portanto, uma representação ortogonal H é definida através da atribuição de valores inteiros α e β para todas arestas de G, de forma que as seguintes propriedades sejam satisfeitas:

- $1 \leq \alpha(u, v) \leq 4$;
- $\beta(u, v) \geq 0$;
- Para cada vértice u, a soma de $\alpha(u, v)$ para todas arestas orientadas a partir de u é 4;
- Para cada face interna f, a soma de $\alpha(u, v) + \beta(v, u) + \beta(u, v)$ para todos as arestas anti-horário de f é igual a $2a(f) - 4$;
- Para a face externa h, esta soma é igual a $2a(h) + 4$.

Transformação do Problema

O problema de se encontrar um desenho ortogonal com o menor número de dobras é modelado através de uma rede com produtores e consumidores [30]. Nessa rede, os ângulos representam o fluxo produzido pelos vértices, transportados pelas faces ao longo de suas dobras incidentes e, eventualmente, consumidos pelas próprias faces. Como o desenho deve ser ortogonal, todos os ângulos possuem medida do tipo $k \cdot \pi/2$, onde $1 \leq k \leq 4$. Uma unidade de fluxo corresponde a um ângulo de $\pi/2$.

A transformação do embutimento planar bidirecionado de G em uma rede é feita sob premissas que garantem que o fluxo de menor custo corresponde ao desenho de menor número de dobras. A descrição abaixo, ilustrada pela figura 3.3, explica como a rede deve ser construída. Os valores de $\lambda(u,v)$, $\mu(u,v)$ e $\chi(u,v)$ indicam, respectivamente, o limite inferior, a capacidade e o custo de um arco (u,v).

- Os nós da rede η são os vértices e faces de G. Um nó derivado de um vértice é chamado *nó-vértice*. Um nó derivado de uma face é chamado *nó-face*.
- Um nó-vértice v de η produz fluxo $\sigma(v) = 4$.
- Um nó-face f de η consume fluxo $\sigma(f) = 2a(f) - 4$ se f é uma face interna, e fluxo $\sigma(h) = 2a(h) + 4$ se h é a face externa.
- Para cada aresta (u,v) de G, com faces f e g a sua esquerda e direita, respectivamente, η possui dois arcos:

  Arco (u,f) Este arco possui $\lambda(u,f) = 1$, $\mu(u,f) = 4$ e $\chi(u,f) = 0$.
  O arco (u,f) representa a quantidade $\alpha(u,v)$. O limite inferior e a capacidade são definidos dessa forma em decorrência da definição de produção de fluxo por um vértice v. O custo associado a esse arco é zero, pois o ângulo se trata de um ângulo de vértice (note que o arco é do vértice u para face f) e a solução buscada é a de mínimo número de dobras (ângulos nas arestas).

  Arco (f,g) Este arco possui $\lambda(f,g) = 0$, $\mu(f,g) = \infty$ e $\chi(f,g) = 1$.
  O arco (f,g) representa a quantidade $\beta(u,v)$. O limite inferior e a capacidade indicam que a aresta pode ou não conter dobras. A unidade do custo equivale à unidade de fluxo definida ($\pi/2$). Como ele ocorre em uma aresta (note que o arco é da face f para a face g), deve ser contabilizado.

O total de fluxo produzido pelos nós-vértices deve ser igual ao total de fluxo consumido pelos nós-faces. Portanto, a conservação do fluxo na rede η associada ao embutimento planar de G é descrita pela equação 3.1 (as faces internas e a face externa são consideradas dentro de um único somatório).

$$\sum_{v} \sigma(v) - \sum_{f} \sigma(f) = 4 * v - (\sum_{f} (2a(f) - 4)) - 8 \tag{3.1}$$

O custo de um fluxo ϕ em uma rede construída conforme a descrição acima equivale ao número total de dobras de uma representação ortogonal H de G. Além disso, o fluxo de um arco entre nós-vértices e nós-faces, $\phi(u,f)$, corresponde ao valor $\alpha(u,v)$ da aresta associada em H, enquanto que o fluxo de um arco entre nós-faces, $\phi(f,g)$, corresponde ao valor $\beta(u,v)$ dessa mesma aresta. Com base nessas idéias, o procedimento 4 descreve a etapa de ortogonalização.

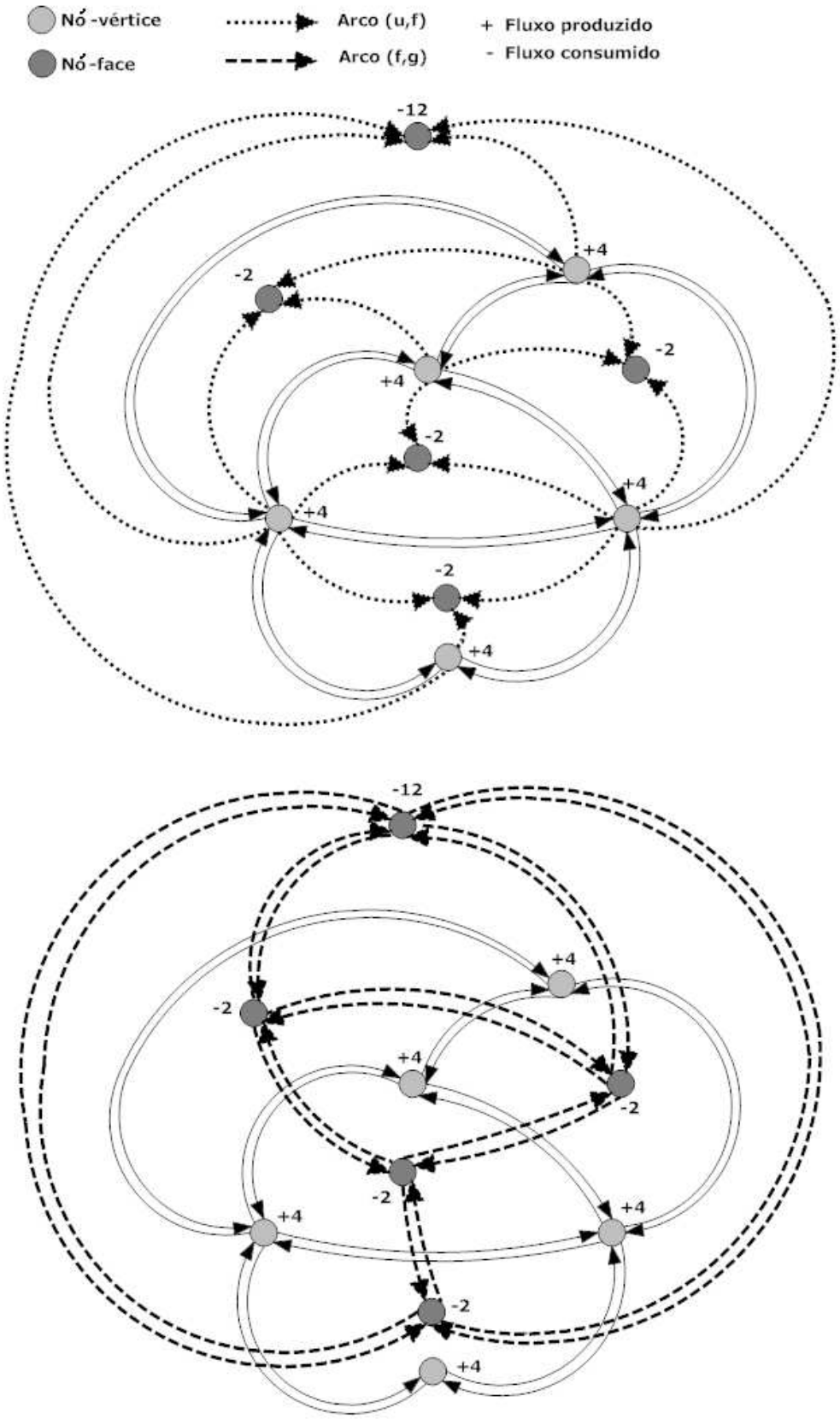

Figura 3.3: Rede associada ao embutimento planar da figura 3.2 (dividida em dois desenhos apenas para facilitar o entendimento).

Procedimento 4 Ortogonaliza

Entrada: Embutimento planar bidirecionado de G com grau máximo 4.
Saída: Representação ortogonal H com o menor número de dobras.

1: Construa a rede de fluxo η associada a G.
2: Calcule o fluxo ϕ de mínimo custo em η.
3: Construa uma representação ortogonal H, através da equivalência de fluxos nos arcos da rede e valores de α e β.

3.1.3 Compactação

A compactação consiste da atribuição de comprimento aos segmentos de arestas de uma representação ortogonal. É nesta etapa que são determinadas as coordenadas finais dos vértices e dobras de arestas. O objetivo é construir um desenho de pequena área, utilizando apenas coordenadas inteiras para os pontos, já que a convenção grid é adotada.

A estratégia apresentada inicialmente exige que todas as faces da representação ortogonal sejam retangulares. Situações gerais são tratadas posteriormente.

Faces Retangulares

Representações ortogonais com faces retangulares possuem no máximo quatro dobras em arestas. Com exceção de possíveis oito segmentos que delimitam os cantos da face externa, todos os outros são arestas de G. A formalização da retangularidade de uma face interna f de H é obtida do fato de que todas as arestas $(u, v) \in D(f)$ devem possuir $\alpha(u, v) \leq 2$ e $\beta(v, u) = 0$. Similarmente, para a face externa h de H, todas as arestas $(u, v) \in D(h)$ devem possuir $\alpha(u, v) \geq 2$ e $\beta(u, v) = 0$.

Novamente, a solução adotada para este problema é baseada em uma abordagem de fluxo em redes. São criadas duas redes, η_{hor} e η_{ver}, associadas, respectivamente, aos segmentos horizontais e verticais da representação ortogonal H. A rede η_{ver} possui um nó para cada face interna de H e dois nós especiais, s e t, que representam, respectivamente, a região inferior e superior da face externa. Além disso, a rede η_{ver} conta com um arco (f, g) para cada par de faces f e g que compartilham um segmento horizontal e, com f abaixo de g. Esses arcos representam os comprimentos dos segmentos no desenho ortogonal e possuem $\lambda(f, g) = 1$, $\mu(f, g) = \infty$ e $\chi(f, g) = 1$. A rede η_{hor} possui características análogas.

Neste modelo, as redes η_{hor} e η_{ver} são dígrafos planares acíclicos com apenas uma fonte e um sumidouro. A lei de conservação do fluxo é observada pelo fato de que as bordas superiores e inferiores (da esquerda e da direita) são do mesmo comprimento. A soma dos custos dos fluxos em η_{hor} e η_{ver} é igual ao comprimento total das arestas. O valor do fluxo ϕ em η_{hor} (η_{ver}) é igual à largura (altura) do desenho. Finalmente, o resultado mais importante é que o valor do fluxo ϕ em cada arco (f, g) representa o comprimento de seu segmento associado. Conseqüentemente, ao se encontrar o menor fluxo de custo mínimo, o desenho de menor área é encontrado. A figura 3.4 ilustra as redes η_{hor} e η_{ver}. O algoritmo de compactação é descrito pelo procedimento 5.

Tanto a compactação quanto a ortogonalização adotam um mecanismo de transformação de problemas. Em ambos os casos, são criados modelos matemáticos de redes que incorporam as peculiaridades dos respectivos problemas. Apesar de algumas similaridades, há uma diferença

Procedimento 5 Compacta

Entrada: Representação ortogonal H, onde todas as faces são retangulares.

Saída: Comprimento dos segmentos de H em uma representação de Grid.

1: Construa as redes de fluxo η_{hor} e η_{ver} associadas a H.
2: Calcule os fluxos de custo mínimo em ambas as redes.
3: Atribua ao comprimento de cada segmento de H o valor do fluxo nos arcos correspondentes em η_{hor} e η_{ver}.

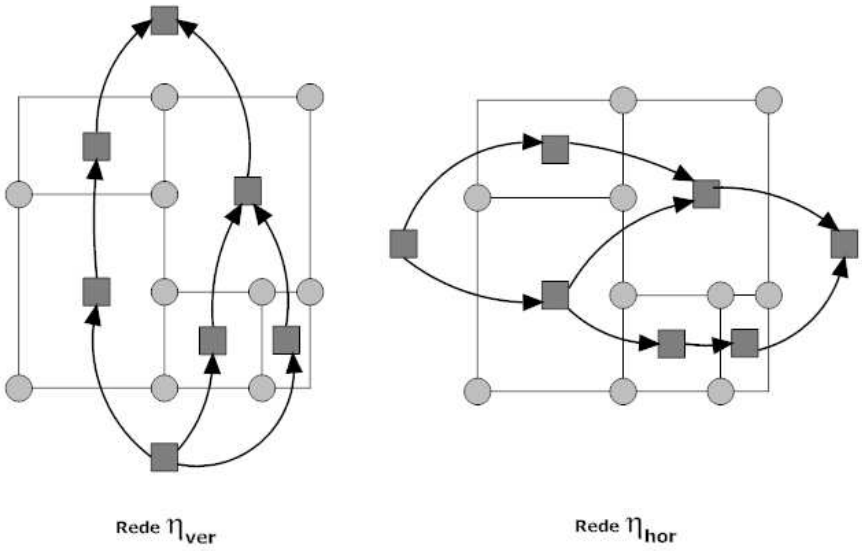

Figura 3.4: Redes η_{hor} e η_{ver}.

notável entre a rede criada na ortogonalização e aquela criada na compactação. O primeiro caso consiste de uma rede de produtores e consumidores, na qual um fluxo factível de menor custo deve ser encontrado. Este problema é equivalente ao problema de fluxo máximo de menor custo em uma rede [102]. No caso da compactação, o objetivo é construir um desenho de menor área possível. Portanto, o problema é encontrar o menor fluxo de menor custo, em uma rede com limites inferiores nos arcos.

Faces Não-Retangulares

A restrição anterior imposta sobre as faces da representação ortogonal é relativamente grande. Normalmente, aplicações não produzem grafos com essas características. Mudanças no algoritmo de compactação poderiam torná-lo muito mais complexo. Portanto, a estratégia mais eficiente é *refinar* representações ortogonais gerais em representações de faces retangulares, através da inclusão de arestas falsas. Esse processo consiste de três etapas: adição de um vértice isolado, adição de um vértice ao longo de uma aresta e adição da aresta que conecta os dois vértices.

O objetivo é garantir que uma representação ortogonal H seja um subconjunto de uma outra representação ortogonal refinada H'. Como consequência, um desenho de G', derivado de H', contém o desenho de G, associado à representação ortogonal H original. Os detalhes de construção da nova representação ortogonal H' são descritos abaixo.

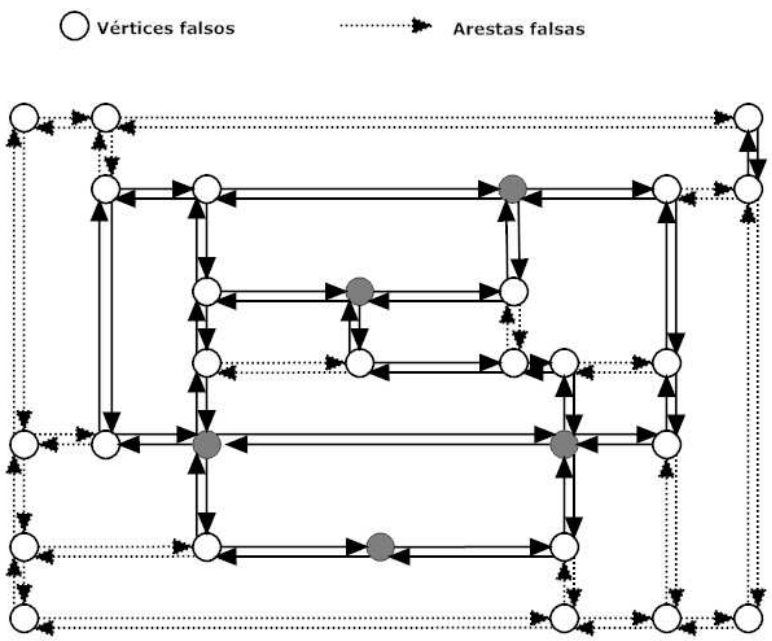

Figura 3.5: Representação ortogonal refinada.

1. Insira um vértice para cada dobra de H.
2. Seja f uma face interna de H. Se f for retangular, não faça nada. Senão:
   (a) Para cada aresta e de f, seja *next(e)* a próxima aresta de f, adjacente a e, quando percorremos f em sentido anti-horário. O vértice comum de e e *next*(e) é chamado de *corner(e)*.
   (b) Para cada aresta e de f, *turn(e)* = +1 se e e *next*(e) fazem uma curva para esquerda; *turn(e)* = 0 se e e *next*(e) estão alinhadas; *turn(e)* = -1 se e e *next*(e) fazem uma curva para direita.
   (c) Para cada aresta e, encontre a primeira aresta e' seguindo e no sentido anti-horário, tal que a soma dos *turns* das arestas entre e (incluindo) e e' (excluindo) seja igual a 1. Configure *front(e)* = e'.
   (d) Para cada aresta e, tal que *turn*(e) = -1, insira um vértice *project(e)* ao longo da aresta *front*(e), e, em seguida, insira a aresta *extend(e)* = (*corner*(e), *project*(e)). Atualize a representação H'. Note que a nova aresta não possui dobras e está alinhada com e.

A construção da face externa do refinamento é realizada de forma similar. A diferença básica é que para este caso deve ser adicionado um retângulo externo à face externa. A figura 3.5 ilustra o resultado final da transformação da representação ortogonal da figura 3.2 em uma representação ortogonal de faces retangulares.

O principal aspecto que deve ser observado é que os novos vértices e arestas devem ser inseridos sem quebrar o embutimento planar do grafo. A lista de adjacência necessita de suporte a operações para separar uma aresta por um vértice e inserir uma nova aresta em uma posição específica na lista de arestas de um determinado vértice.

3.1.4 Análise de Complexidade

A etapa de planarização é determinante na eficiência do *Giotto*. Basicamente, quanto maior o número de vértices falsos adicionados, mais lenta será a execução do algoritmo.

Existem dois fatores que estão diretamente ligados ao número de vértices adicionados na planarização. O primeiro é uma característica inerente ao grafo, relacionado a sua densidade. Grafos mais densos provavelmente sofrerão mais operações de substituição de cruzamentos de arestas. O segundo está relacionado à heurística de construção de um subgrafo planar *maximal*.

Conforme mencionado anteriormente, a construção do subgrafo planar *maximum* é um problema *NP-hard*. O algoritmo conhecido mais eficiente para essa tarefa é apresentado em [79]. Porém, ele é difícil de se implementar e possui um pior caso que leva a uma complexidade de tempo exponencial. Já o procedimento 1, descreve uma heurística simples de construção de um subgrafo planar *maximal*, que, para grafos esparsos, leva a um número pequeno de vértices falsos.

A construção do embutimento planar pode ser realizada em tempo linear [42, 76]. Similarmente, o cálculo de caminho mínimo para grafos planares também pode ser realizado com a mesma complexidade de tempo [83]. Portanto, levando em conta o pior caso, a etapa de planarização impõe um limite inferior na complexidade de tempo do algoritmo *Giotto* de $O(V')$, onde V' pode ser $\Omega(V^4)$.

A complexidade de tempo das etapas de ortogonalização e compactação do *Giotto* é determinada pelos algoritmos de fluxo em redes. Os algoritmos de fluxo máximo por incremento de caminhos e fluxo máximo por formação de pré-fluxo possuem complexidade de tempo diferentes e fortemente dependentes da densidade do grafo [102]. De qualquer forma, nenhum deles garante complexidade de tempo inferior a $O(V^2)$. Há ainda uma opção de um algoritmo mais elaborado para essa tarefa, o qual garante complexidade de tempo de $O(V^{7/4}logV)$ [67].

Agregando os comentários dos parágrafos acima, o que deve ficar claro é que a complexidade de tempo do *Giotto* pode ser acima de $O(V^4)$, para o pior caso, onde são adicionados vários vértices durante a etapa de planarização. Mas vale lembrar que em aplicações onde os grafos são esparsos e o número de vértices falsos é insignificante, este fato pode não ser um problema relevante, já que neste caso os fatores limitantes seriam relativos às etapas de ortogonalização e compactação.

3.2 Algoritmo *Column*

A etapa de planarização existente no algoritmo da seção 3.1 pode introduzir vários vértices fictícios ao grafo, os quais contribuem para o aumento complexidade e dificuldade de implementação do desenho. Diferentemente, o algoritmo desta seção, originalmente apresentado em [37], é capaz de construir representações ortogonais independentemente da planaridade do grafo.

A primeira fase do algoritmo é responsável por ordenar os vértices de um grafo biconectado a partir de uma *fonte* e um *sumidouro* (*numeração-st*). Em seguida, os vértices do grafo são adicionados em linhas consecutivas do grid e colunas são alocadas para suas arestas.

Com essa estratégia, é possível desenhar um grafo biconectado de grau máximo quatro em um grid de $(E - V + 1)$ x V. Adicionalmente, com uma única possível exceção [1], todas arestas do grafo possuem zero, uma ou duas dobras. O algoritmo também pode ser aplicado a grafos simplesmente conectados, desde que sejam feitas as adaptações necessárias, as quais se resumem a uma das seguintes opções:

- Decompor o grafo simplesmente conectado em blocos de componentes biconectados. Neste caso, o algoritmo também precisará sofrer algumas pequenas alterações;
- Adicionar arestas ao grafo de forma a torná-lo biconectado - essas arestas podem ser removidas posteriormente. Idealmente, o número de arestas adicionadas deve ser o menor possível, já que algumas garantias do algoritmo podem ser afetadas.

3.2.1 Orientação de grafos biconectados não-direcionados

Os vértices de um grafo podem ser numerados de variadas formas. Em função dessa numeração, é possível estabelecer determinada orientação para o grafo. A numeração de vértices a partir de uma fonte s (*source*) a um sumidouro t (*target*), conhecida como *numeração-st*, é muito utilizada em algoritmos de teste de planaridade e de desenho de grafos [43]. Sua definição é apresentada a seguir.

Definição 3.1 *Seja $G(V, E)$ um grafo biconectado. Dada uma aresta $s - t$ de G, uma função g tal que $g : V \rightarrow \{1, 2, 3...|V|\}$ é chamada uma numeração-st se todas as condições abaixo forem verdadeiras:*

1. $g(s) = 1$
2. $g(t) = |V|$
3. $\forall v \in V - \{s, t\}$ *existem vértices adjacentes* u *e* w *tal que* $g(u) < g(v) < g(w)$.

O primeiro algoritmo linear para computar uma numeração-st de um grafo foi criado por Even e Tarjan [58, 59]. Esse algoritmo também é apresentado em [57]. No entanto, a apresentação é feita de forma bastante descritiva e vários detalhes de implementação são deixados de lado. Os procedimentos 6 e 7 descrevem esse algoritmo com uma perspectiva mais voltada para a implementação. Neles, são utilizadas uma pilha P e uma lista L.

O procedimento 6 é bastante simples. Inicialmente, é executada uma DFS no grafo para que os valores de ordem, pai e alcance de cada vértice sejam computados. Posteriormente, o sumidouro e a fonte do grafo são adicionados em P. Enquanto a pilha não estiver vazia, o vértice v no topo de P é removido. Se existir algum caminho *desconhecido* no grafo que contém v em uma de suas extremidades, os vértices que compõe esse caminho são adicionados em P. Caso contrário, v recebe seu número identificador.

Os caminhos desconhecidos no grafo são encontrados pelo procedimento 7. Um vértice v é passado como argumento e suas arestas incidentes são percorridas. O caminho, representado

[1] A única aresta que possivelmente terá três dobras é a que conecta determinado vértice ao sumidouro, se este for de grau quatro.

pela lista L, é construído quando uma aresta *nova* (ainda não visitada) de árvore ou de retorno é encontrada. No primeiro caso, o ciclo interno adiciona as arestas que definem o alcance de v em L. No segundo caso, as arestas-de-árvore ainda não visitadas que chegam em v são adicionadas em L. Na inicialização do procedimento 7, é necessário que a fonte s e o sumidouro t, assim como a aresta que os conecta, sejam marcados como *velhos*.

Procedimento 6 Numeração-st

Entrada: Grafo biconectado G, uma fonte s e um sumidouro t.
Saída: Numeração-st de G.

1: Realize uma DFS em G e $\forall v \in V$ compute a *ordem(v)*, o *pai(v)* e *alcance(v)*.
2: $Num \leftarrow 1$
3: Adicione t em P
4: Adicione s em P
5: ***Enquanto*** o topo de $P \neq t$ ***faça***
6: $v \leftarrow$ topo de P
7: Execute o procedimento 7 para encontrar um caminho desconhecido em G que inicia ou termina em v. Utilize L para armazená-lo.
8: ***Se*** $L = \emptyset$ ***então***
9: $g(v) \leftarrow Num$
10: $Num \leftarrow Num + 1$.
11: ***Senão***
12: ***Para*** todos os vértices w de L em ordem ***faça***
13: Adicione w em P
14: ***Fim Para***
15: Adicione v em P
16: Limpe L.
17: ***Fim Se***
18: ***Fim Enquanto***

3.2.2 Embutimento no Grid

A numeração dos vértices estabelece uma conseqüente orientação do grafo, designada pela característica de que qualquer aresta pode ser considerada direcionada de um vértice de número menor a um vértice de número maior. Essa propriedade é fundamental para a construção de um algoritmo, descrito pelo procedimento 8, capaz de desenhar um grafo biconectado de grau máximo quatro em um grid de $(E-V+1)$ x V em tempo linear ($O(V)$). Uma análise detalhada deste algoritmo é encontrada em [37].

Um ponto importante do algoritmo deve ser mencionado. Idealmente, as arestas-de-saída dos vértices devem ser alocadas em colunas imediatamente vizinhas ao vértice em questão. Isso garante que se um embutimento planar for fornecido como argumento de entrada para o algoritmo, o desenho não conterá cruzamentos de arestas. Essa também é a razão pela qual deve-se garantir que a aresta que conecta (v_1, v_n) deve ser posicionada na face externa do desenho.

O grande desafio de implementação da alocação de colunas é a eficiência. A medida em que o grid é construído, as arestas ainda não possuem uma coordenada específica. Consequentemente, não há uma forma direta de saber qual é a coluna imediatamente vizinha de algum ponto no grid.

A solução deste problema é utilizar uma estrutura de dados que garanta a ordenação das

Procedimento 7 Encontra caminho

Entrada: Grafo biconectado G, um vértice v e uma lista L.

Saída: Lista L de vértices que formam um caminho direcionado em G, no qual a origem ou o destino é representado por v.

1: ***Para*** as arestas e adjacentes a v ***faça***
2: ***Se*** e é uma aresta *velha* ***então***
3: Passe para outra aresta (próxima execução do ciclo)
4: ***Fim Se***
5: $w \leftarrow$ outra extremidade de e.
6: ***Se*** *ordem(v)* $>$ *ordem(w)* ***E*** e for uma *aresta de retorno* ***então***
7: Marque e como *velha*
8: Adicione um vértice *nulo* em L.
9: ***Retorne***
10: ***Senão se*** *ordem(v)* $<$ *ordem(w)* ***E*** e for uma *aresta de árvore* ***então***
11: Marque e como *velha*
12: $d \leftarrow$ *alcance(w)*
13: ***Enquanto*** $d \neq$ *ordem(w)* ***faça***
14: Adicione w em L
15: Marque w como *velho*
16: $e \leftarrow$ aresta com extremidade w que define *alcance(w)*
17: Marque e como *velha*
18: $w \leftarrow$ extremidade de e oposta a w
19: ***Fim Enquanto***
20: ***Retorne***
21: ***Senão se*** *ordem(v)* $<$ *ordem(w)* ***E*** e for uma *aresta de retorno* ***então***
22: Marque e como *velha*
23: ***Enquanto*** w for um vértice *novo* ***faça***
24: Adicione w em L
25: Marque w como *velho*
26: $e \leftarrow$ *aresta de árvore* que leva a w
27: Marque e como *velha*
28: $w \leftarrow$ extremidade de e oposta a w
29: ***Fim Enquanto***
30: ***Retorne***
31: ***Fim Se***
32: ***Fim Para***

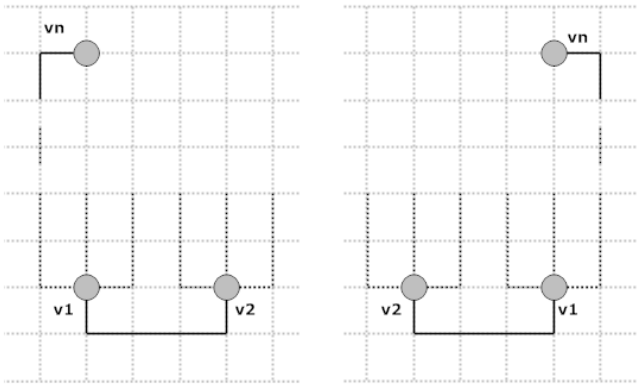

Figura 3.6: Posição e alocação de colunas de v_1, v_2 e v_n.

Procedimento 8 *Column*

Entrada: Grafo biconectado G de grau máximo quatro.

Saída: Desenho ortogonal de G em um grid.

1: Compute uma numeração-st para G.
2: Posicione os vértices v_1 e v_2 na primeira linha do grid.
3: Aloque colunas no grid para cada uma de suas arestas de saída. Veja figura 3.6 para um exemplo onde v_1 e v_2 possuem três arestas de saída cada um.
4: Force que a aresta que conecta (v_1, v_n) esteja na face externa do desenho (se o vértice v_1 foi posicionado à esquerda do vértice v_2, então essa aresta deve ser posicionada na extrema esquerda do grid; caso contrário, a aresta deve ser posicionada na extrema direita do grid). Detalhes na figura 3.6.
5: ***Para*** cada vértice v restante de G de acordo com a numeração-st ***faça***
6: Posicione v em uma nova linha do grid. Se v possui três arestas de entrada, posicione-o na coluna central. Caso v seja o último vértice e possua quatro arestas de entrada, é necessário alocar mais uma linha para uma de suas arestas.
7: Aloque colunas no grid para as arestas de saída de v.
8: ***Fim Para***

arestas já processadas. Neste caso específico, é necessário que tal estrutura de dados permita a inserção, pesquisa e navegação de items em tempo constante, o que não é uma tarefa trivial. Dietz e Sleator [51] propõem uma solução eficiente para o problema de manutenção de ordem. No entanto, as estruturas de dados e algoritmos envolvidos são altamente complexos.

Uma solução alternativa, mas que aumenta o tempo de execução do algoritmo em um fator de $O(logV)$, é a utilização de uma árvore binária balanceada. Apesar da pequena perda em eficiência, a implementação do algoritmo se torna significativamente mais simples.

Finalmente, também é válido mencionar que a alocação de colunas imediatamente vizinhas aos vértices é importante mesmo para grafos não planares, pois tende a diminuir o número de cruzamentos de arestas. É claramente visível que a alocação de colunas nas extremidades do grid pode ocasionar um grande número desnecessário de cruzamentos de arestas.

3.3 Algoritmo *Pair*

O algoritmo da seção 3.2 é simples e com complexidade de tempo linear. No entanto, é ineficiente em termos da área total ocupada. Esta seção apresenta um algoritmo baseado na mesma metodologia, mas com uma otimização que garante uma área máxima de $0.77V^2$.

A primeira fase do algoritmo também consiste da computação de uma numeração-st e não será discutida. Porém, a inserção de vértices ao grid em linhas consecutivas é substituída por uma estratégia mais refinada que visa a reutilização de linhas e colunas. A idéia básica é agrupar alguns vértices em pares e desenhá-los a partir de um conjunto específico de regras, determinado pelo *tipo* de par formado.

Este algoritmo também pode ser implementado em tempo linear, mas não garante que um grafo planar seja realmente desenhado sem cruzamento de arestas. Originalmente apresentado em [93], o algoritmo passou por revisões e correções em [94] e, em seguida, em [30].

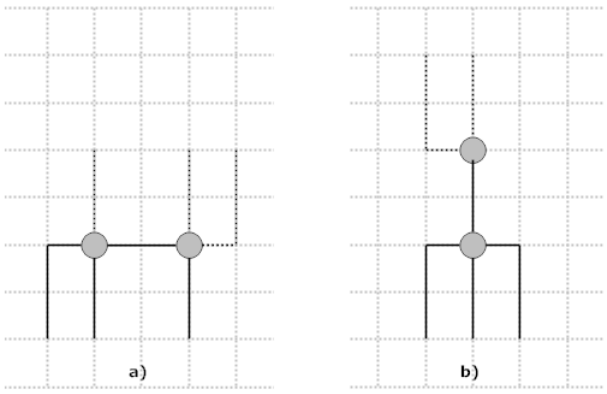

Figura 3.7: Tipos de pares: a) linha; b) coluna.

3.3.1 Formação de pares

O principal objetivo do algoritmo é promover a reutilização de linhas e colunas do grid. Para isso são criados pares de vértices que se enquadram em um dos seguintes tipos:

- *Par de linha*
  Os dois vértices compartilham a mesma linha do grid ou são organizados de tal forma que permita a colocação de algum outro vértice em uma linha já utilizada. Ilustração na figura 3.7-a.
- *Par de coluna*
  Os dois vértices são organizados de tal forma que duas arestas diferentes utilizem a mesma coluna. Ilustração na figura 3.7-b.

A construção dos pares de vértices é realizada a partir de várias regras. O aspecto de maior relevância para esta etapa é a categorização dos vértices, que é definida a partir da numeração-st do grafo. Um vértice com x arestas de entrada e y arestas de saída é chamado de vértice x-y. Além disso, em cada par de vértices, aquele com o menor número-st é considerado o *primeiro*, enquanto que o de maior número-st é considerado o *segundo*.

Para realizar o pareamento dos vértices, o grafo passa, inicialmente, por uma *condensação*. Neste processo, todos os vértices 1-1 que possuem arestas de saída conectadas a vértices 1-2 e 1-3 são temporariamente retirados, ou *absorvidos*, do grafo. A partir de então, a principal tarefa do algoritmo de pareamento é criar exatamente um par para cada vértice 1-2, 1-3 e 2-2 do grafo. Vértices 1-2 e 1-3 são sempre pareados com o vértice anterior a eles na numeração-st. Vértices 2-2 podem formar pares com vértices 1-2, 1-3 e 2-2 ou com vértices 1-1, 2-1 e 3-1 que sejam seus predecessores imediatos.

O procedimento 9 ilustra, em detalhes, o mecanismo de formação de pares descrito no parágrafo anterior. É também importante notar que todos os vértices 2-2, 1-2 e 1-3 são agrupados em pares, com uma possível exceção do vértice v_2. Dependendo do grafo de entrada e da numeração-st, o vértice v_3 pode pode formar um par com um vértice de numeração superior ou não formar par algum. Nesses casos, o ciclo principal não será executado para $i = 2$.

Procedimento 9 Forma pares

Entrada: Grafo biconectado G de grau máximo quatro.
Saída: Pareamento de vértices do grafo condensado de G.

1: Compute uma numeração-st para G.
2: Condense G através da absorção de vértices 1-1.
3: $i \leftarrow n-1$ (n é número de vértices de G).
4: ***Enquanto*** $i > 2$ ***faça***
5: ***Se*** v_i é um vértice 1-1, 2-1 ou 3-1 ***então***
6: $i \leftarrow i-1$.
7: ***Senão se*** v_i é um vértice 1-2 ou 1-3 ***então***
8: $w \leftarrow$ próximo vértice em sentido decrescente da numeração-st.
9: Forme um par contendo os vértices v_i e w.
10: $i \leftarrow i-2$.
11: ***Senão se*** v_i é um vértice 2-2 ***então***
12: $w \leftarrow$ próximo vértice em sentido decrescente da numeração-st.
13: ***Enquanto*** w é um vértice 1-1, 2-1 ou 3-1 não predecessor imediato de v_i ***faça***
14: $i \leftarrow i-1$.
15: $w \leftarrow$ próximo vértice em sentido decrescente da numeração-st.
16: ***Fim Enquanto***
17: Forme um par contendo os vértices v_i e w.
18: $i \leftarrow i-2$.
19: ***Fim Se***
20: ***Fim Enquanto***

3.3.2 Embutimento no Grid

A principal vantagem do pareamento de vértices é a possibilidade de otimizar a área total ocupada pelo desenho. A reutilização de linhas e colunas é elaborada de forma que cada tipo de par é representado no grid de forma eficiente. Devido ao mecanismo de categorização dos vértices, a existência ou não de arestas de entrada e saída a serem desenhadas é facilmente detectada.

Vários tipos de pares de vértices são encontrados em um grafo. Cada um deles deve ser desenhado de forma diferente. Se em um par $<v_j, v_i>$ $(j < i)$, v_j for um predecessor imediato de v_i, ambos compartilham uma relação de *predecessão-sucessão*. Caso contrário, eles são ditos *independentes*. Vértices que participam de algum par são ditos *assinalados*, enquanto que vértices que não participam de nenhum par são ditos *não-assinalados*. A listagem abaixo apresenta os possíveis tipos de pares resultantes do procedimento 9 e suas respectivas regras de desenho.

Tipo A:
- v_i e v_j possuem relação de predecessão-sucessão;
- v_i é um vértice 2-2 e v_j é um vértice 2-2.

Nesse caso, uma coluna pode ser reutilizada (figura 3.8-a).

Tipo B:
- v_i e v_j são independentes; - v_i é um vértice 2-2 e v_j é um vértice 2-2.

Nesse caso, uma coluna pode ser reutilizada (figuras 3.8-b e 3.9-a). Dependendo das colunas alocadas para as arestas de entrada dos vértices, pode ser necessário posicionar v_j acima de v_i, para que uma coluna seja reutilizada.

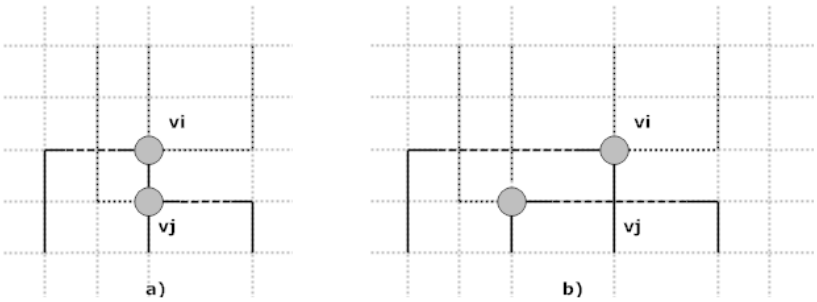

Figura 3.8: Reutilização do grid com pares dos tipos A (a) e B (b).

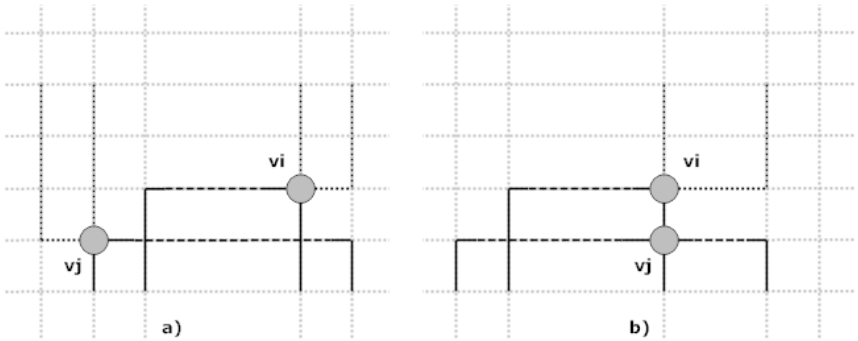

Figura 3.9: Reutilização do grid com pares dos tipos B (a) e C (b).

Tipo C:
- v_i e v_j possuem relação de predecessão-sucessão;
- v_i é um vértice 2-2 e v_j é um vértice 2-1 ou 3-1.
Nesse caso, uma coluna pode ser reutilizada (figura 3.9-b).

Tipo D:
- v_i e v_j possuem relação de predecessão-sucessão;
- v_i é um vértice 2-2 e v_j é um vértice 1-1.
Nesse caso, uma linha pode ser reutilizada (figura 3.10-a).

Tipo E:
- v_i e v_j possuem relação de predecessão-sucessão;
- v_i é um vértice 2-2 e v_j é um vértice 1-2 ou 1-3.
Nesse caso, uma linha pode ser reutilizada (figura 3.10-b).

Tipo F:
- v_i e v_j são independentes;
- v_i é um vértice 2-2 e v_j é um vértice 1-2 ou 2-3.
Nesse caso, uma coluna pode ser reutilizada e v_j deve ser posicionado acima de v_i (figura 3.11-a).

Tipo G:
- v_i é um vértice 1-2 ou 1-3 e v_j é um vértice 2-2, 2-1 ou 3-1.
Nesse caso, uma coluna pode ser reutilizada. Em alguns casos, até duas colunas podem ser reutilizadas. As regras de desenho são similares àquelas descritas para os tipos A, B e C.

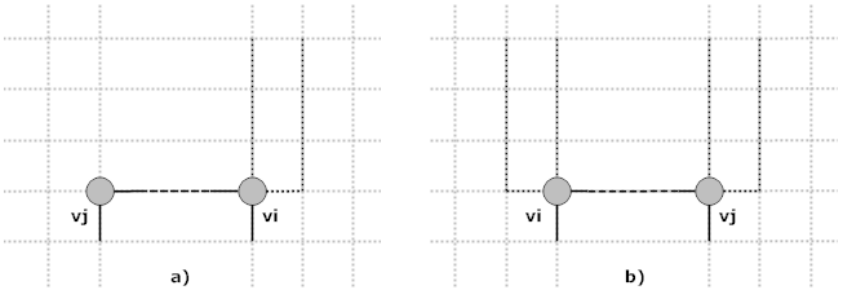

Figura 3.10: Reutilização do grid com pares dos tipos D (a) e E (b).

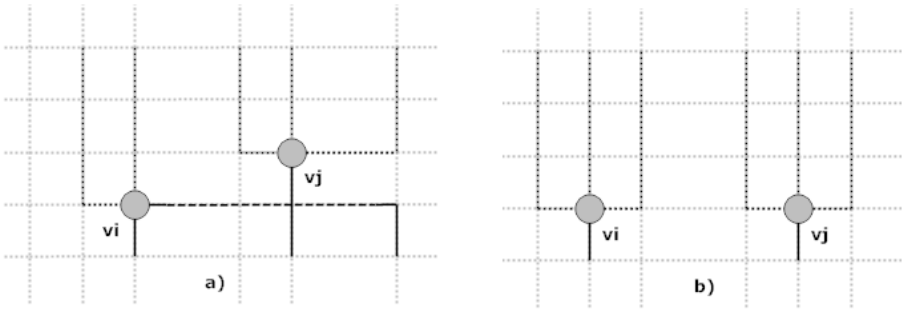

Figura 3.11: Reutilização do grid com pares dos tipos F (a) e H (b).

Tipo H:
- v_i e v_j são independentes;
- v_i é um vértice 1-2 ou 1-3 e v_j é um vértice 1-2 ou 1-3.

Nesse caso, uma linha pode ser reutilizada (figura 3.11-b).

Tipo I:
- v_i e v_j possuem relação de predecessão-sucessão;
- v_i é um vértice 1-2 ou 1-3 e v_j é um vértice 1-2 ou 1-3.
- Adicionalmente, as seguintes condições devem ser verdadeiras:

  - v_i é conectado posteriormente a algum vértice 1-1, 1-2 ou 1-3; Ou, v_i é conectado posteriormente a algum vértice 2-2, o qual é o segundo vértice de um par Tipo D, E ou F.
  - A aresta (v_{i-1}, v_i) não absorveu nenhum vértice 1-1 do grafo original.

Nesse caso, uma coluna do grid pode ser reutilizada (figura 3.12-a). Note que deve-se garantir que a aresta que conectará, posteriormente, o vértice v_x, deve possuir apenas duas dobras.

Tipo J:
- v_i e v_j possuem relação de predecessão-sucessão;
- v_i é um vértice 1-2 ou 1-3 e v_j é um vértice 1-2 ou 2-3.
- Adicionalmente, pelo menos uma das condições necessárias para o tipo I não é verificada.
- Nesse caso, uma linha do grid pode ser reutilizada. A figura 3.12-b ilustra uma situação em que a primeira condição para o tipo I não é atendida. Note que o próximo vértice, v_x, a ser posicionado pode utilizar a mesma linha do vértice v_i. Já a figura 3.13-a, ilustra uma

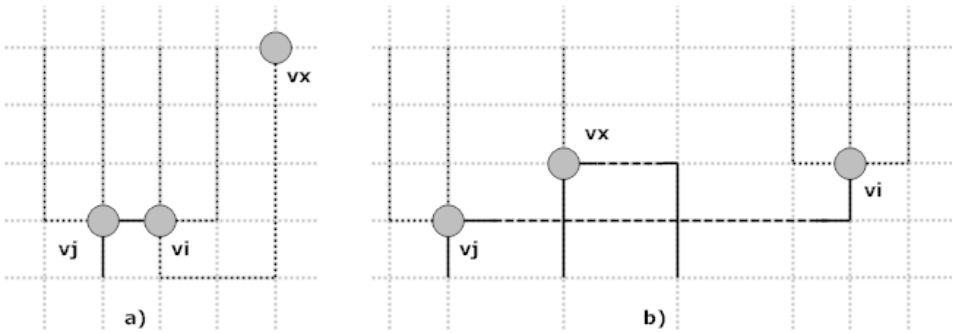

Figura 3.12: Reutilização do grid com pares dos tipos I (a) e J (b).

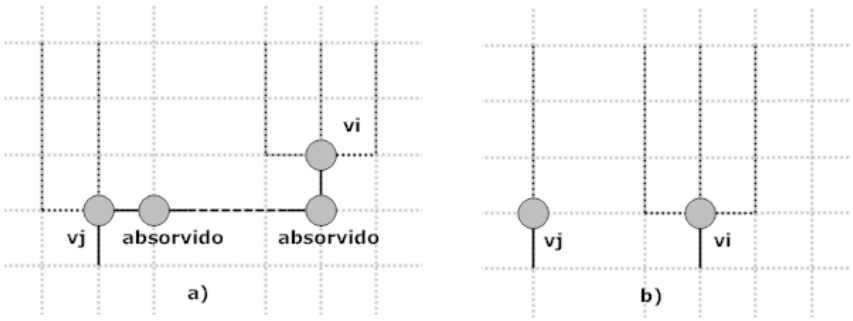

Figura 3.13: Reutilização do grid com pares dos tipos J (a) e L (b).

situação em que a segunda condição para o tipo I não é atendida. Note que o(s) vértice(s) absorvido(s) pode(m) ser posicionado(s) na mesma linha do vértice v_j.

Tipo L:
- v_i é um vértice 1-2 ou 1-3 e v_j é um vértice 1-1 que não é predecessor imediato de v_i [2]
Nesse caso, uma linha do grid pode ser reutilizada (figura 3.13-b).

Tendo sido formado os pares de vértices, o procedimento 10 descreve um algoritmo de desenho ortogonal no qual a área máxima é de $0.77V^2$ e todas as arestas possuem até duas dobras. O algoritmo pode ser implementado em tempo linear, desde que seja utilizada a estrutura de dados para manutenção de ordem das colunas com características similares a daquela sugerida na seção 3.2. Da mesma forma, a alternativa de utilizar uma árvore binária balanceada aumenta a complexidade de tempo do algoritmo em um fator de $O(lg(V))$.

A inserção dos dois primeiros vértices no grid deve ser abordada com atenção. Caso o vértice v_2 esteja agrupado em algum par, o procedimento é simples: basta posicionar o vértice v_1 na primeira linha do grid e deixar que o vértice v_2 siga as regras aplicadas ao seu tipo de par. De acordo com os artigos que apresentam o algoritmo [93, 94, 30], se o vértice v_2 não estiver assinalado, o posicionamento de ambos os vértices deve ser feito em uma ou duas linhas do grid, como ilustra a figura 3.14. No entanto, a estratégia de posicionar v_1 e v_2 sempre na mesma linha do grid resulta no mesmo número de linhas e colunas ocupadas. Portanto, é também uma opção viável e adotada em [31].

[2] Este tipo de par não é mencionado em [93, 94, 30]. Porém, sua existência é verificada, já que os vértices 1-1 absorvidos são apenas aqueles predecessores *imediatos* de vértices 1-2 ou 1-3.

Procedimento 10 *Pair*

Entrada: Grafo biconectado G de grau máximo quatro.

Saída: Pareamento de vértices do grafo condensado de G.

1: Compute uma numeração-st para G.
2: ***Se*** o vértice v_2 estiver assinalado ***então***
3: Posicione o vértice v_1 na primeira linha do grid.
4: Aloque colunas no grid para cada uma de suas arestas de saída. O vértice v_2 será posicionado em conjunto com seu par.
5: ***Senão***
6: Posicione os vértices v_1 e v_2 na primeira linha do grid.
7: Aloque colunas no grid para cada uma de suas arestas de saída.
8: ***Fim Se***
9: ***Repita***
10: Considere os vértices de G de acordo com a numeração-st.
11: ***Se*** o vértice v não tiver sido posicionado no grid ***então***
12: ***Se*** o vértice v estiver assinalado ***então***
13: Posicione v em uma nova linha do grid. Se v possui três arestas de entrada, posicione-o na coluna central.
14: Aloque colunas no grid para as arestas de saída de v.
15: ***Senão***
16: Posicione v e seu respectivo par no grid de acordo com as regras apresentadas anteriormente.
17: ***Fim Se***
18: ***Fim Se***
19: ***Enquanto*** o único vértice restante seja v_n
20: Posicione o vértice v_n em uma nova linha no grid. Se v_n possui quatro arestas de entrada, então há aresta que entra em v_n pelo topo é escolhida como sendo aquela que o conecta a $v_(n-1)$.
21: Reinsira os vértices anteriormente absorvidos do grafo e os posicione em dobras ou pontos sem cruzamentos do grid. Caso não seja possível, insira novas linhas no grid.

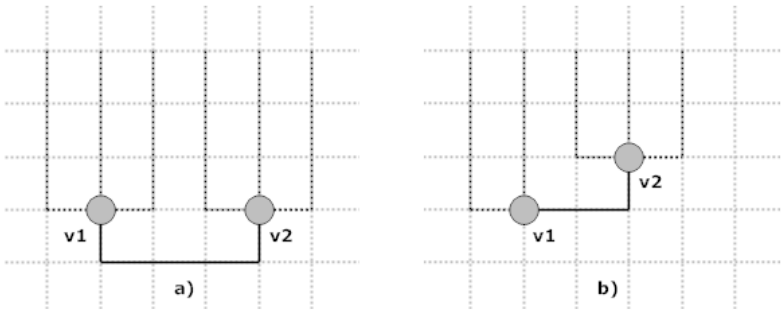

Figura 3.14: Posicionamento dos vértice v_1 e v_2.

No algoritmo da seção 3.2, caso o vértice v_n possua quatro arestas de entrada, é possível que uma delas seja a única com três dobras. Essa terceira dobra é evitada no procedimento 10, justamente pela escolha da aresta que entra pelo topo de v_n ser aquela que conecta v_{n-1}. Como v_{n-1} sempre possui apenas uma aresta de saída, esta conterá, no máximo, duas dobras na última linha do grid. A mesma abordagem não pode ser utilizada no procedimento 8, pois o embutimento planar, caso exista, precisa ser mantido (não há uma forma de garantir que essa aresta seja desenhada em uma das extremidades do grid).

Os vértices absorvidos são os últimos a serem posicionados. Idealmente, eles devem ser colocados em dobras ou em pontos do grid que não contenham cruzamento de arestas. Portanto, é importante manter uma estrutura de dados capaz de prover essa informação de forma eficiente. Em situações em que não forem encontradas posições para posicionamento dos vértices absorvidos, novas linhas podem ser inseridas no grid.

3.4 Análise Comparativa

Os algoritmos apresentados neste capítulo geram desenhos com características bem particulares. Na maioria dos casos, a qualidade dos desenhos gerados pelo *Giotto* é melhor em termos de critérios estéticos [31]. Porém, sua grande desvantagem é sua alta complexidade de tempo de execução. Um sumário geral e comparativo dos pontos positivos e negativos de cada um dos algoritmos é apresentado na tabela 3.1.

Tabela 3.1: Pontos positivos e negativos dos algoritmos *Giotto*, *Pair* e *Column*.

Algoritmo	Pontos positivos	Pontos negativos
Giotto	- Grafos planares geram desenhos planares. - Minimização do número de cruzamentos de arestas (devido à planarização). - Minimização do número de dobras. - Minimização da área. - Suporta diretamente grafos biconectados.	- Implementação difícil. - Complexidade de tempo de execução alta. - Etapa de planarização pode adicionar muitos vértices falsos ao grafo. - Construção dos embutimentos planares e redes é um processo caro.
Column	- Implementação mais simples que do *Giotto* e do *Pair*. - Não necessita de uma etapa de planarização. - Grafos planares geram desenhos planares. - Pode ser implementado em tempo linear.	- Área do desenho é grande. - Não há minimização do número de cruzamentos de arestas. - Não há minimização do número de dobras. - Grafos não-biconectados precisam sofrer um transformação ou o algoritmo precisa ser adaptado.
Pair	- Não necessita de uma etapa de planarização. - Pode ser implementado em tempo linear. - Área do desenho é menor que a do *Column*.	- Área do desenho é maior que a do *Giotto*. - Implementação difícil, devido às regras de desenho dos pares. - Grafos planares não geram desenhos planares, necessariamente. - Não há minimização do número de cruzamentos de arestas. - Não há minimização do número de dobras. - Grafos não-biconectados precisam sofrer um transformação ou o algoritmo precisa ser adaptado.

Capítulo 4

GTAD (*Graph Toolkit for Algorithms and Drawings)*

O foco central deste trabalho é a criação de uma biblioteca C++ de algoritmos em grafos. Além de prover implementações para algoritmos tradicionais como de DFS, BFS, caminho mínimo e variações de problemas de fluxo em redes (fluxo máximo, custo mínimo, etc), também são suportados algoritmos mais complexos e relacionados à área de desenho de grafos. A biblioteca, denominada GTAD (*Graph Toolkit for Algorithms and Drawings*), é desenvolvida sob o paradigma de *programação genérica*, o que permite, entre outras vantagens, a flexibilidade da estrutura de dados utilizada na representação do grafo.

Programação genérica é um termo utilizado para descrever um modelo de desenvolvimento de software de alto desempenho, no qual o objetivo é possibilitar o maior grau de generalização de estruturas de dados e comportamentos, através da redução de compromissos relativos à implementação [90, 108, 21, 114]. A STL (*Standard Template Library*) [78] é a principal referência na adoção desse modelo. Sua arquitetura serviu como base para a BGL (*Boost Graph Library*) [104], uma biblioteca de grafos que emprega inúmeras técnicas de programação genérica. Assim como a GTAD, a BGL é desenvolvida em C++, atualmente a linguagem de programação com maior suporte à programação genérica [66].

Existem várias bibliotecas para manipulação e execução de algoritmos em grafos. Algumas apresentam características interessantes, mas sofrem deficiência de recursos importantes. Aspectos como linguagem de programação, modelo de desenvolvimento, algoritmos suportados, desempenho, tipo de distribuição, usabilidade e extensibilidade também são variantes. Por exemplo, há bibliotecas que suportam uma grande variedade de algoritmos, mas possuem estruturas de dados rígidas e insubstituíveis. Outras, são bastante flexíveis, mas não oferecem implementações para algoritmos de desenho de grafos. Enfim, é possível mencionar várias situações de balanceamento de requisítos que podem ser insatisfatórias em determinado contexto.

Além de bibliotecas, existem também várias ferramentas para aplicações de grafos. Neste trabalho, são consideradas ferramentas, produtos nos quais o usuário final não é um desenvolvedor de software. Ferramentas são freqüentemente baseadas em interfaces gráficas e não fornecem, necessariamente, uma API (*Application Programming Interface*) de desenvolvimento. Algumas delas [8, 4, 2] são de caráter científico e se preocupam com a questão matemática do grafo. Outras [17, 9, 1, 18], contêm recursos de visualização de grafos em uma perspectiva voltada para o domínio de negócio da aplicação.

Uma análise completa de todas bibliotecas disponíveis não é objetivo deste trabalho. Portanto, são introduzidas abaixo apenas algumas bibliotecas que possuem características semelhantes àquela desenvolvida neste trabalho. Todas elas permitem a manipulação de estruturas de dados e funções programaticamente, através de codificação em linguagens de programação imperativas e estaticamente tipadas como C, C++ ou Java.

LEDA (*Library of Efficient Data types and Algorithms*) [22]
Biblioteca comercial em C++ com um rico conjunto de algoritmos e estruturas de dados relacionados a grafos, redes, geometria, otimização combinatorial, entre outros temas. A implementação dos algoritmos é dependente da representação de dados fornecida pela biblioteca. Possibilita parametrização de vértices e arestas do grafo através da inclusão de propriedades adicionais. Oferece alguns algoritmos relacionados à área de desenho de grafos a partir de sua própria interface gráfica. Código fonte fechado, o que dificulta extensão e reutilização.

BGL (*Boost Graph Library*)] [104]
Principal referência de aplicação de programação genérica em aplicações de grafos. Segue o modelo da STL (*Standard Template Library*) [78], no qual algoritmos e estruturas de dados são desacoplados e possibilitam a programação baseada em interfaces. Portanto, o usuário não é obrigado a utilizar as representações de grafos oferecidas pela biblioteca, podendo construir suas próprias. Permite adição de propriedades adicionais em vértices e arestas com alto grau de genericidade. Não oferece algoritmos relacionados à área de desenho de grafos. Desenvolvida em C++ com código fonte aberto.

GTL (*Graph Template Library*) [63]
Apesar do nome, não é uma biblioteca puramente baseada em *templates*. Possui implementações apenas de algoritmos tradicionais em grafos. A forma de representação do grafo é rígida e só permite extensões através de edição do código fonte, que é aberto. Não realiza liberação de versões há vários anos. Desenvolvida em C++.

P.I.G.A.L.E. (*Public Implementation of a Graph Algorithm Library and Editor*) [15]
Projeto de caráter científico, desenvolvido em C++, com várias implementações de algoritmos de desenho, principalmente para grafos planares estáticos. As estruturas de dados não são flexíveis e a visualização é realizada através de uma interface gráfica própria. Oferece alguns algoritmos complexos e avançados. Porém, algoritmos tradicionais como de minimização de fluxo em redes ou caminho mínimo não são suportados. Apesar do código fonte ser aberto e permitir utilização direta, o foco da biblioteca é a manipulação de grafos a partir da interface gráfica.

AGD (*Algorithms for Graph Drawing*) [20]
Biblioteca livre com suporte a vários algoritmos de desenho de grafos e relacionados ao tema. Os componentes são desenvolvidos em módulos e podem ser extendidos pelo usuário. No entanto, é dependente das estruturas de dados e algoritmos tradicionais fornecidos pela biblioteca comercial LEDA. A visualização é desacoplada dos algoritmos de desenho. Portanto, permite a utilização de interfaces gráficas variadas. Desenvolvida em C++ com código fonte fechado. Não são liberadas versões há vários anos, pois o projeto tornou-se parte da ferramenta comercial *GoVisual* [6].

GDToolkit (*Graph Drawing Toolkit*) [28]
Desenvolvida em C++, é uma biblioteca com recursos similares aqueles da AGD. Suporta vários algoritmos de desenho de grafos e relacionados ao tema. Fornece componentização modular e o mecanismo de visualização é flexível. Implementação de algoritmos tradicionais, assim como de estruturas de dados fundamentais, é suportada pela biblioteca

comercial LEDA. Não são liberadas versões há vários anos. A biblioteca é livre, mas o código fonte é fechado.

OpenJGraph [14]
Biblioteca livre desenvolvida em Java com suporte a poucos algoritmos de desenho de grafos. Basicamente, desenhos ortogonais e de linhas retas. Implementação de alguns algoritmos tradicionais. Possui interface gráfica própria, mas a API é disponibilizada. Não são liberadas versões há vários anos e o código fonte é aberto.

Desenvolver uma biblioteca que atenda a todos os requisítos existentes é praticamente impossível. Primeiro, porque alguns deles podem ser contraditórios entre si. Segundo, porque normalmente os requisítos estão associados a determinados tipos de aplicação, os quais também podem variar enormemente e serem significativamente diferentes. Detalhes de implementação e as principais funcionalidades da GTAD são apresentadas nas próximas seções.

4.1 Representação do Grafo

Em sistemas computacionais, grafos são usualmente representados por matrizes ou listas de adjacência [57]. A primeira consiste de uma matriz V x V, com uma entrada na linha u e coluna w definida como 1 se existe uma aresta conectando os vértices u e v, e como 0 caso contrário. Em sua forma mais simples, a outra estrutura de dados consiste de um vetor de V listas encadeadas, onde cada uma delas contém as arestas incidentes ao vértice v. Em ambos os casos, V denota o número de vértices no grafo.

Características de desempenho e flexibilidade de diferentes representações de grafo variam bastante [102]. Uma matriz de adjacência sempre utiliza espaço proporcional a $O(V^2)$, oposto a $O(V + E)$ em uma lista de adjacência. Portanto, uma matriz de adjacência não é recomendada para grafos esparsos. Por outro lado, ela fornece um mecanismo de tempo constante para detectar se uma aresta composta pelos vértices u e v existe no grafo. Conseqüentemente, operações que dependem dessa habilidade, como desabilitação de arestas paralelas, também são eficientemente implementadas.

Listas de adjacência são normalmente a melhor escolha, pois grande parte dos grafos que aparecem em aplicações práticas são esparsos [88, 106]. Adicionalmente, alguns algoritmos de desenho requerem que a representação seja feita com esse tipo de estrutura, pois a informação combinatorial das listas encadeadas de cada um dos vértices pode ser importante.

A GTAD oferece uma implementação padrão para representação de um grafo utilizando uma lista de adjacência. Através da combinação de um modelo bem desenhado e utilização de técnicas eficientes de C++, como polimorfismo estático, *traits*, especialização completa e parcial de templates, despacho de tags e banco de memória, operações como inserção ou remoção de vértices e arestas são implementadas com bastante eficiência. Além disso, com auxílio de uma tabela de hash, o teste de verificação de existência de aresta é suportado em tempo constante amortizado.

Uma característica interessante da lista de adjacência da biblioteca é que os vértices e arestas do grafo podem ser definidos pelo usuário de forma totalmente transparente. Basta parametrizar a lista de adjacência com os respectivos tipos - desde que eles sigam as regras de interface

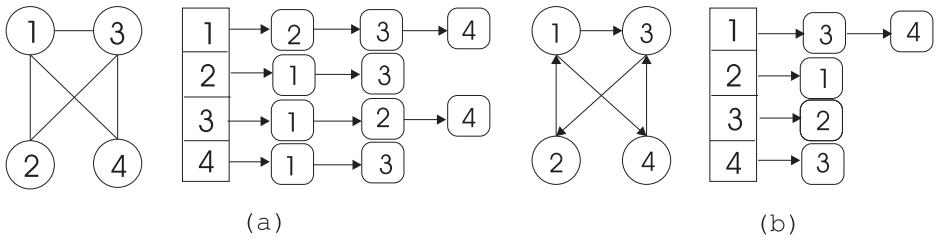

Figura 4.1: Grafos e suas listas de adjacência; a) não-direcionado; b) direcionado

estabelecidas. Essa estratégia é significativamente diferente de uma simples anexação de propriedades aos vértices e arestas do grafo, como é feito em bibliotecas como a LEDA e BGL, pois não demanda o código extra necessário para navegação entre estruturas de dados. Outras bibliotecas nem sequer oferecem este tipo de recurso.

A política de inserção de arestas no grafo pode ser rigorosamente estabelecida. A responsabilidade de controlar a presença ou não de arestas paralelas e auto-ciclos não é deixada por conta do usuário. Este grau de flexibilidade, não observado em nenhuma das bibliotecas mencionadas anteriormente, também é conseguido através da parametrização da lista de adjacência. A próxima seção apresenta aspectos conceituais de modelagem da lista de adjacência. A seção seguinte detalha alguns pontos de implementação que são extremamente importantes para atingir um alto nível de flexibilidade e desempenho.

4.1.1 Modelagem Conceitual

Listas de adjacência comumente apresentadas em livros [102, 88, 46, 113] consistem de um vetor de elementos que apontam para listas encadeadas que contêm as *arestas-de-saída* do vértice representado. Uma variação freqüente dessa estrutura é modificar as listas para armazenarem apenas os vértices de destino das arestas. A Figura 4.1 ilustra o uso de listas de adjacência para um grafo não-direcionado e um grafo direcionado.

Representações como essas possuem várias desvantagens:

- O teste de existência de uma aresta (u, w) pode levar tempo proporcional a V (no melhor caso, quando duplicatas não são permitidas). Conseqüentemente, operações que dependem desse teste, como desabilitação de arestas paralelas ou remoção de uma aresta (u, w), são relativamente caras.
- Quando o grafo é não-direcionado, a operação de remoção de arestas precisa destruir ambas representações da mesma aresta, as quais estão em listas diferentes. Não há uma forma eficiente de realizar essa tarefa.
- Remover um vértice v requer a destruição de sua lista e uma busca para remover nodos nas listas de todos os vértices presentes nas arestas anteriormente adjacentes a v. O custo desse procedimento é proibitivo.
- Especialmente para grafos não-direcionados, iteração sobre todas as arestas não é feita de forma natural, já que existem duas representações da mesma aresta.
- Iteração sobre os vértice do grafo é facilmente implementada. No entanto, quando vértices são removidos o vetor conterá espaços vazios. Portanto, a operação deixará de ser natural

e sua eficiência será deteriorada. Alternativamente, os elementos do vetor podem ser realocados, o que também causará impactos negativos no desempenho.

- Para grafos direcionados, informação sobre as *arestas-de-entrada* de um vértice não é obtida diretamente.
- O número total de arestas deve ser computado separadamente através de um contador. Caso contrário, seria necessário a soma do tamanho de cada lista encadeada, o que levaria tempo proporcional a V.

A lista de adjacência da GTAD trata todos esses problemas. Algumas das soluções adotadas são propostas em [102].

Contêiner de Vértices e Arestas

Como grafos são compostos por um conjunto de vértices e arestas, é natural pensar em uma representação na qual existe um contêiner separado para cada um deles. A idéia é abordada através da construção de duas listas encadeadas: uma para os vértices e a outra para as arestas. Isso permite a implementação de operações de inserção e remoção eficientemente. Adicionalmente, sob nenhuma circunstância tais operações causariam invalidação de ponteiros para nodos das listas. Também é importante observar que a obtenção do tamanho dos contêineres, assim como a travessia em ambos sentidos, é facilmente implementada.

Infelizmente, acessar um elemento arbitrário utilizando uma lista é um processo lento. À primeira vista, isso pode parecer um problema, já que a operação de remoção em tempo constante não é útil se a busca pelo elemento desejado não é rápida. Para resolver esse problema, é utilizada uma estrutura auxiliar de tabela de símbolos.

Hashing é um exemplo clássico de tabela de símbolos que encaixa perfeitamente na solução desse problema, pois operações de inserção, busca e remoção são eficientemente suportadas. Particularmente, a lista de adjacência da GTAD utiliza uma combinação de tabelas de *hash* dinâmicas e seqüenciamento linear (*Linear Probing*) para tratamento de colisões. Maiores detalhes sobre essa estratégia são encontrados em [101, 60]. Devido à utilização de tabelas dinâmicas de *hash*, para se obter melhor desempenho, o grafo deve ser inicializado com valores próximos do número máximo de vértices e arestas.

A lista de adjacência contém duas tabelas de *hash*: uma associada ao contêiner de vértices e outra associada ao contêiner de arestas. Os items dessas estruturas de dados mantêm ponteiros para os vértices e arestas que eles representam, e, conseqüentemente, precisam ser atualizados corretamente. Essa estratégia permite a remoção de um vértice ou aresta em tempo constante amortizado, basta fazer uma busca na tabela de símbolos e seguir o ponteiro do item desejado - na verdade, a operação de remoção de arestas para grafos não-direcionados ainda possui um problema que será tratado na próxima seção.

Um aspecto muito importante que deve ser levado em conta é o gerenciamento de memória. O processo de requisitar e devolver memória ao sistema operacional é caro, e se não for tratado corretamente pode prejudicar o desempenho da aplicação. A solução adotada pela lista de adjacência da GTAD é utilizar a estratégia de um banco de memória segregado em porções simples (*simple segregated storage pool*)[117]. A idéia básica é alocar porções de memória de um tamanho arbitrário e particioná-las em blocos de tamanho fixo que podem ser usados para

um tipo de dado específico. Assim, é possível prover operações eficientes para alocação de desalocação da memória associada aos contêineres de vértices e arestas.

Informações de Adjacência

Relações de adjacência são mantidas de uma forma muito similar àquela utilizada em listas de adjacência tradicionais. A diferença mais significativa é que ao invés das próprias arestas estarem presentes nas listas de cada vértice, são utilizados ponteiros para elas. Conseqüentemente, os mecanismos usuais de recuperação de qualquer tipo de informação de adjacência são preservados.

As listas encadeadas de cada vértice são duplamente encadeadas. Também existem ponteiros para início e para o fim da lista. Isso é particularmente importante para situações nas quais a circulação sobre os vértices da lista é necessária. Um exemplo é o algoritmo de construção do dual de um grafo planar, descrito no capítulo 3.

Com o intuito de promover acesso fácil às arestas-de-entrada em grafos direcionados, são mantidas duas listas para cada vértice. A estrutura das listas de entrada e saída é exatamente a mesma. Naturalmente, representações de grafos não-direcionados sempre terão as listas de entrada vazias. Mas isso não causará nenhum problema, já que memória não será alocada para elas.

O problema ainda pendente é o de destruição de ambas representações de uma aresta em um grafo não-direcionado. Apesar de não ter sido mencionado (por razões didáticas), os contêineres de vértices e arestas são um pouco mais complexos do que foi descrito até o momento. Seus elementos não são os vértices e arestas propriamente ditos. Eles são estruturas de dados mais complexas que agregam uma série de items. A estrutura de dados associada a um vértice contém os seguintes membros:

- O vértice propriamente dito;
- Um ponteiro para o primeiro nodo da lista de arestas-de-saída;
- Um ponteiro para o último nodo da lista de arestas-de-saída;
- Um ponteiro para o primeiro nodo da lista de arestas-de-entrada;
- Um ponteiro para o último nodo da lista de arestas-de-entrada;
- O grau-de-saída do vértice;
- O grau-de-entrada do vértice.

Similarmente, a estrutura de dados associada a uma aresta contém os seguintes membros:

- A aresta propriamente dita;
- Um ponteiro para o nodo correspondente na lista de arestas-de-saída do vértice de origem;
- Um ponteiro para o nodo correspondente na lista de arestas-de-saída ou de arestas-de-entrada do vértice de destino. Caso o grafo seja não-direcionado, apenas as listas de arestas-de-saída são utilizadas.

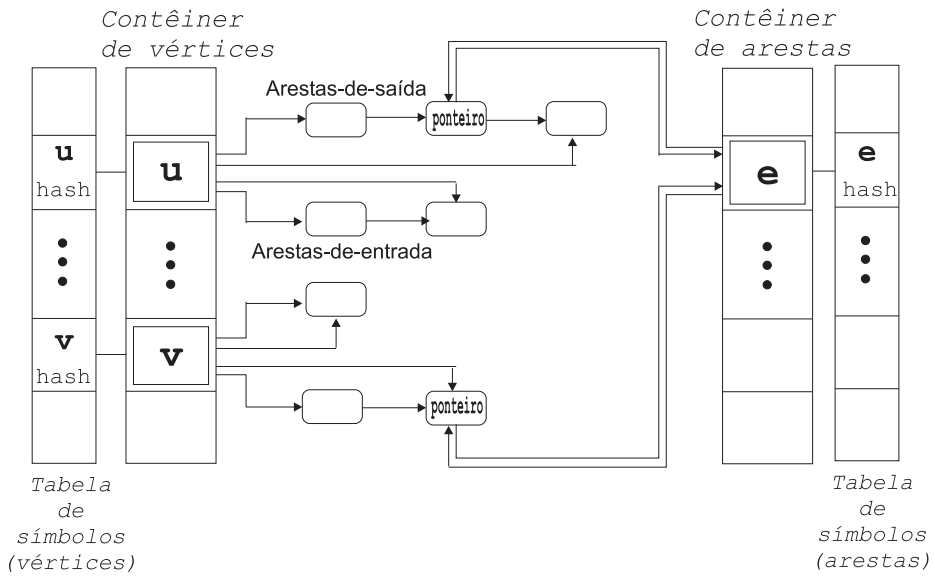

Figura 4.2: Visão estrutural simplificada da lista de adjacência.

Com a disponibilidade das informações fornecidas por essas estruturas de dados auxiliares, uma implementação eficiente da operação de remoção de arestas para grafos não-direcionadas é facilmente concretizada. Basta seguir os ponteiros cruzados das listas dos vértices de origem e destino.

Uma lista de adjacência derivada a partir deste modelo conceitual deve ser conduzida com muito cuidado, pois existe uma grande quantidade de manipulações de ponteiros. Adicionalmente, a forma como as operações são implementadas pode afetar o desempenho. A Figura 4.2 mostra uma visão estrutural da lista de adjacência da biblioteca.

4.1.2 Implementação Baseada em Templates

Na linguagem C++, programação genérica é contemplada através do uso de *templates*, que constituem um mecanismo poderoso de parametrização de tipos em tempo de compilação. Apesar de não existir uma definição formal e bem estabelecida para programação genérica, muitas de suas características são apontadas em [90, 108, 21, 114, 71, 72]. Portanto, neste trabalho, considera-se que o modelo de desenvolvimento baseado em programação genérica está intimamente relacionado aos seguinte tópicos:

√ *Conceitos*

- As operações são construídas em termos de especificação de interfaces, as quais podem ser implementadas por uma grande coleção de tipos de dados;

√ *Eficiência*

- Resolução de *templates* em C++ é realizada em tempo de compilação, o que possibilita tomada de decisões durante a construção do código. Polimorfismo estático é um exemplo clássico;

√ *Generalização*

– Componentes genéricos são programados para sustentar o menor grau possível de especificidade. Conseqüentemente, são bastante reutilizáveis e flexíveis.

Diversas técnicas de programação genérica são empregadas na biblioteca. Inúmeras construções C++, comprovadamente eficientes [89, 90], são amplamente utilizadas. A próxima seção apresenta detalhes da arquitetura geral da lista de adjacência da biblioteca.

Arquitetura da Lista de Adjacência

A partir do modelo conceitual apresentado, um importante passo em sua construção é decidir quais aspectos relacionados à estrutura do grafo podem ser parametrizados. São identificados os seguintes itens:

- *Critérios de inserção de arestas* - O grafo deve permitir auto-ciclos? O grafo deve permitir arestas paralelas? Essas perguntas são respondidas através da seleção deste parâmetro;
- *Direcionalidade* - Indica se o grafo é direcionado ou não-direcionado;
- *Tipos dos vértices e arestas* - Permite ao usuário definir seus próprios tipos de dados para representação dos vértices e arestas do grafo.

O código seguinte ilustra a parametrização da classe `Adjacency_list` com os items mencionados acima. Essa é a classe que modela a lista de adjacência da GTAD. Note que a implementação faz uso de técnicas avançadas de C++. Portanto, é necessário um compilador que seja condizente com o padrão da linguagem.

```
template <
  class Criteria_t,
  class Directionality_t,
  class Vertex_t,
  template <class, class> class Edge_t>
class Adjacency_list
{
  //...
};
```

Sempre que há uma tentativa de inserção de uma aresta no grafo, uma chamada a uma versão sobrecarregada do método `allow_insertion` é realizada com base no tipo selecionado para o parâmetro `Criteria_t`. Esse tipo de mecanismo configurado através de sobrecarga de tipos em tempo de compilação é normalmente chamado de despacho de tags (*tag dispatching*) [25]. O código abaixo ilustra seu princípio de funcionamento.

```
typedef typename Graph_t::Criteria_t Criteria;
if (graph.allow_insertion(..., ..., Criteria()))
{
  //Insere aresta.
}
```

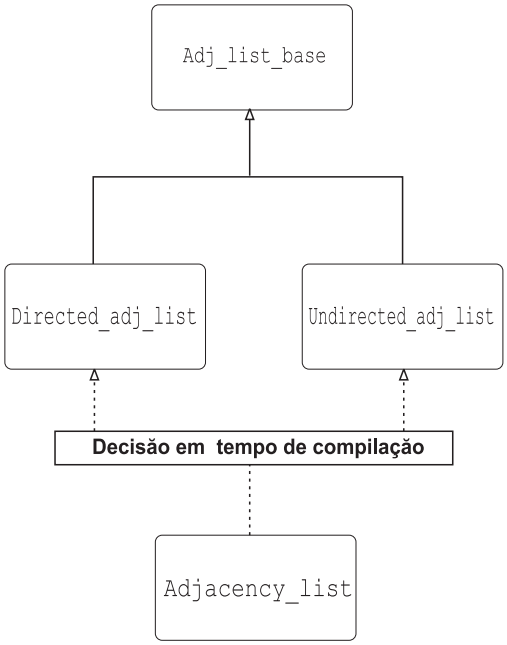

Figura 4.3: Estrutura hierárquica da lista de adjacência

Atualmente, existem quatro opções de argumento para a política de inserção de arestas: `Simple_graph`, `Multigraph`, `No_self_loops`, and `No_parallel_edges`. A diferença entre elas é clara. Um `Simple_graph` não permite arestas paralelas nem auto-ciclos, enquanto que um `Multigraph` permite ambos. Os outros dois tipos impões regras independentes a essas características.

Uma técnica muito interessante utilizada na implementação da lista de adjacência é a CRTP (*Curiosly Recurring Template Pattern*) [114], a qual consiste em passar uma classe derivada como argumento *template* para a própria classe base. A idéia por traz disso é construir, em tempo de compilação, uma estrutura hierárquica para a lista de adjacência como a descrita pela Figura 4.3.

Grafos direcionados e não-direcionados, representados, respectivamente, pelas classes `Directed_adj_list` e `Undirected_adj_list`, precisam se comportar diferentemente em operações relacionadas com inserção e remoção de arestas. É natural pensar em uma forma de fazer uma ligação estrutural entre os métodos envolvidos nessas operações, em tempo de compilação. Para isso, o segundo argumento da definição da classe `Adjacency_list` é utilizado. Há dois valores possíveis para ele: `Directed` e `Undirected`. Baseado no mecanismo de especialização parcial de *templates* ilustrado abaixo, é possível selecionar o tipo de grafo desejado e construir a hierarquia da Figura 4.3. Outros detalhes são abordados nos próximos parágrafos.

```
template
<class Directionality_t, class Directed_t, class Undirected_t>
class Directionality_selector;

template <class Directed_t, class Undirected_t>
class Directionality_selector<Directed, Directed_t, Undirected_t>
{
public:
  typedef Directed_t Selected_t;
```

```
};

template <class Directed_t, class Undirected_t>
class Directionality_selector<Undirected, Directed_t, Undirected_t>
{
public:
  typedef Undirected_t Selected_t;
};
```

Há um aspecto fundamental nessa estrutura que precisa ser enfatizado. Operações que são comuns a grafos direcionados e não-direcionados são implementados em uma classe base: `Adj_list_base`. Essa classe, assim como as derivadas `Directed_adj_list` e `Undirected_adj_list`, necessitam de conter as definições de tipo que são realizadas na classe parametrizada pelo usuário. É impossível adquiri-las simplesmente importando-as da classe `Adjacency_list`, pois o compilador entraria em um loop recursivo infinito. Por conseguinte, o uso de uma classe de *traits* é bastante útil para essa tarefa.

A classe `Adj_list_traits` funciona como uma espécie de armazém de tipos de dados. Como ela é uma estrutura de dados independente, sua utilização pode ser feita pelas classes da hierarquia sem grandes problemas. A única restrição é que a definição de tipos deve ser realizada através da classe de *traits*. O código abaixo demonstra detalhes da implementação da classe `Adjacency_list` relacionados aos parâmetros discutidos, a seleção da estrutura hierárquica e a classe de *traits*.

```
template <
  class Criteria_t,
  class Directionality_t,
  class Vertex_t,
  template <class, class> class Edge_t>
class Adjacency_list:
  public Directionality_selector<
    Directionality_t,
    Directed_adj_list<Adjacency_list<Criteria_t, Directionality_t, Vertex_t, Edge_t>,
      Adj_list_traits<Criteria_t, Directionality_t, Vertex_t, Edge_t> >,
    Undirected_adj_list<Adjacency_list<Criteria_t, Directionality_t, Vertex_t, Edge_t>,
      Adj_list_traits<Criteria_t, Directionality_t, Vertex_t, Edge_t> > >::
    Selected_t
{
public:
  typedef Adj_list_traits<Criteria_t, Directionality_t, Vertex_t, Edge_t> Traits;
  typedef typename Traits::Vertex_handle Vertex_handle;
  typedef typename Traits::Edge_handle Edge_handle;
  typedef typename Traits::Vertex_iterator Vertex_iterator;
  typedef typename Traits::Edge_iterator Edge_iterator;
  typedef typename Traits::Adjacent_edge_iterator Adjacent_edge_iterator;
  typedef typename Traits::Adjacent_vertex_iterator Adjacent_vertex_iterator;

  //...
};
```

Os dois últimos parâmetros *template* da lista de adjacência são relativamente simples. Naturalmente, eles correspondem às estruturas de dados que representam os vértices e arestas do grafo. O primeiro deles deve implementar o conceito de um vértice (*Vertex Concept*), enquanto o outro o conceito de uma aresta (*Edge Concept*). Os conceitos são um tipo de especificação que a interface de determinada classe deve implementar. Mais detalhes sobre assunto são apresentados na seção 4.2.

A observação final é relativa ao fato de que parâmetros *template template* têm uma participação especial na especificação da classe `Adjacency_list`. No caso, o tipo de aresta do grafo é parametrizado pelos tipos do vértice e direcionalidade. Como arestas são compostas por vértice, a necessidade da primeira parametrização é clara. No entanto, a razão da direcionalidade também ser um parâmetro *template* para a aresta pode ser um pouco obscura. O seguinte código demonstra o motivo - arestas direcionadas e não-direcionadas devem ser comparadas por igualdade de formas diferentes.

```
template <class Directionality_t, class Vertex_t> inline bool
Edge<Directionality_t, Vertex_t>::
operator ==(const Edge<Directionality_t, Vertex_t>& e)const
{
  return this->verify_equality(e._source, e._target, Directionality_t());
}


template <class Directionality_t, class Vertex_t>
inline bool
Edge<Directionality_t, Vertex_t>::
verify_equality(const Vertex_t& s, const Vertex_t& t, Directed)const
{
  return this->_source == s && this->_target == t;
}


template <class Directionality_t, class Vertex_t>
inline bool
Edge<Directionality_t, Vertex_t>::
verify_equality(const Vertex_t& s, const Vertex_t& t, Undirected)const
{
  return ((this->_source == s && this->_target == t)||
    (this->_source == t && this->_target == s));
}
```

4.2 Funções de Acesso e Conceitos

A interface de usuário para manipulação do grafo na GTAD é implementada com funções globais. Esse é um requisito necessário para atingir o grau de genericidade desejado. Outro ponto importante dentro desse contexto é que os algoritmos ou quaisquer funções que manipulem dados do grafo, devem utilizar as funções e tipos definidos em uma classe de *traits*. São essas características que garantem a flexibilidade de integração com variadas representações de grafos.

Nem todos os algoritmos dependem de todas as funções especificadas na interface do grafo. A idéia de um *conceito* serve para atender a esse tipo de necessidade. Cada conceito de grafo compreende uma gama de funções e tipos de dados que devem ser implementados por determinada estrutura. Essa abordagem foi inicialmente introduzida pela biblioteca padrão C++ (*STL*), e hoje é amplamente difundida em algumas áreas de aplicação. Alguns exemplos são as bibliotecas BGL e CGAL (*Computational Geometry Algorithms Library*)[3]. A GTAD contém diversos conceitos de grafos, os quais são detalhados no apêndice A e ilustrados na figura 4.4.

Funções que se comportam diferentemente para grafos direcionados e não-direcionados são sobrecarregadas com as classes ilustradas na Figura 4.3. Esse mecanismo é mostrado abaixo para a operação de inserção de arestas.

```
template <class Graph_t>
inline typename Graph_t::Edge_handle&
insert_edge(const typename Graph_t::Vertex_handle&,
            const typename Graph_t::Vertex_handle&,
            Directed_adj_list<Graph_t, typename Graph_t::Traits>&)
{
  //...
}

template <class Graph_t>
inline typename Graph_t::Edge_handle&
insert_edge(const typename Graph_t::Vertex_handle&,
            const typename Graph_t::Vertex_handle&,
            Undirected_adj_list<Graph_t, typename Graph_t::Traits>&)
{
  //...
}
```

Usuários que desejarem utilizar suas próprias estruturas de dados, podem criar outras sobrecargas para a funções especificas do conceito em questão. Além disso, é também necessário especializar a classe `Graph_traits`, que define, principalmente, os tipos de dados do grafo. Sua implementação padrão e uma possível especialização para um suposto grafo representado por uma classe `Some_other_graph_representation` são exibidos a seguir.

```
template <typename Graph_t>
class Graph_traits
{
public:
  typedef typename Graph_t::Vertex_handle Vertex_handle;
  typedef typename Graph_t::Edge_handle Edge_handle;
  typedef typename Graph_t::Vertex_iterator Vertex_iterator;
  typedef typename Graph_t::Edge_iterator Edge_iterator;
  typedef typename Graph_t::Adjacent_vertex_iterator Adjacent_vertex_iterator;
  typedef typename Graph_t::Adjacent_edge_iterator Adjacent_edge_iterator;

  static inline Vertex_handle null_vertex();
  static inline Edge_handle null_edge();
```

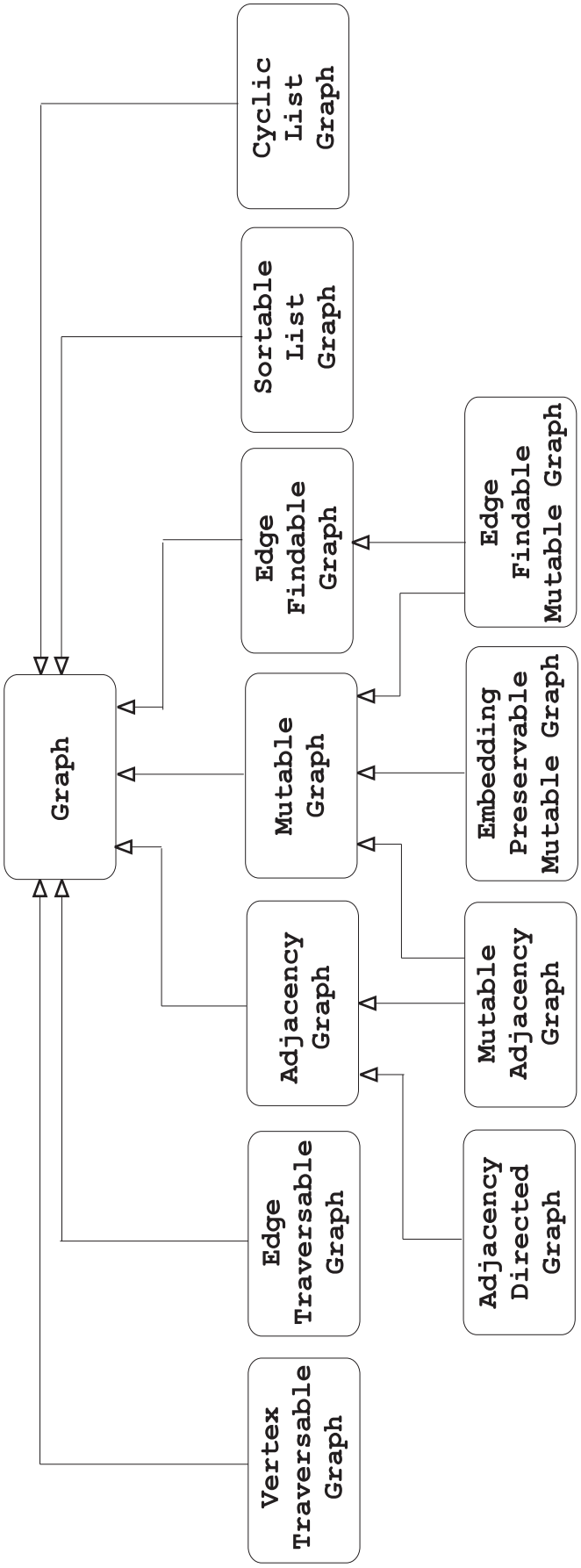

Figura 4.4: Conceitos de grafos da GTAD.

```
};

template <>
class Graph_traits<Some_other_graph_representation>
{
public:
  typedef Some_other_vertex_type Vertex_handle;
  typedef Some_other_edge_type Vertex_iterator;

  //...
};
```

4.3 Algoritmos Tradicionais

Na biblioteca GTAD, os algoritmos são implementados como classes, ao invés de procedimentos. A grande vantagem na adoção de classes é a flexibilidade [61]. Além de ser possível manter um acesso fácil às informações de estado do algoritmo (através de funções membros), a adição de eventuais operações na interface da classe pode ser feita de forma simples, sem demandar uma atualização imediata de seus clientes.

O aspecto fundamental da implementação dos algoritmos é que eles operem apenas sobre os tipos de dados e funções definidos na classe `Graph_traits` ou através da interface especificada por determinado conceito. Isso garante que qualquer estrutura de dados que especialize a classe de *traits* e implemente o conceito exigido pelo algoritmo, funcione corretamente. Uma consequência direta dessa abordagem é que todos os algoritmos são parametrizados pelo tipo do grafo.

Com o intuito de promover o desacoplamento entre grafos e seus atributos, os algoritmos exigem que tais informações adicionais sejam fornecidas separadamente. Portanto, os algoritmos também são parametrizados por estruturas de dados que representam eventuais mapas de atributos. Por exemplo, o algoritmo de BFS espera um argumento *template* capaz de mapear os vértices do grafo em inteiros no intervalo $[0, V-1]$. Esses números são utilizados durante o algoritmo para construir a sequência de busca dos vértices. Além da parametrização, é necessário configurar, através de um método modificador, a instância do mapa que contém os dados em questão. O código seguinte demonstra, parcialmente, a estrutura da classe BFS. O mapa mencionado acima é denominado *vertex_ index_ map* e corresponde ao parâmetro *template* `Vertex_index_map_t`.

```
template<
  class Graph_t,
  class Vertex_index_map_t =
    Vector_attr_map<typename Graph_traits<Graph_t>::Vertex_handle, int>,
  class Dfs_visitor_t =
    Dfs_default_visitor,
  class Search_storable_items_t =
    Search_storable_items<Search_vertices, Tree_edges, Back_edges> >
class Dfs :
  public Search<Graph_t, Vertex_index_map_t>
{
```

```
  //...

  void vertex_index_map(const Vertex_index_map_t& m);

  //...
};
```

A classe `Vector_attr_map` é a implementação padrão da biblioteca para mapas que contêm vértices ou arestas da classe `Adjacency_list`. Assim como quaisquer outros tipos de mapas a serem utilizados nos algoritmos, ela implementa o *Conceito Mapa de Atributos* (*Attribute Map Concept*), que estabelece como o acesso aos dados do mapa deve ser feito. A especificação deste conceito é apresentada no apêndice A.

O terceiro parâmetro *template* da classe `Dfs` é o *visitante* que será usado no algoritmo. O padrão de projeto *Visitor* é apresentado com detalhes em Gamma et al. [65]. Resumidamente, seu objetivo principal é fornecer ao usuário de determinado componente, um mecanismo de *callback* para executar funções próprias, em momentos pré-determinados. No caso da classe `Dfs`, o visitante é chamado, entre outros momentos, na inicialização do algoritmo e ao primeiro encontro de um vértice. A implementação padrão, `Dfs_default_visitor`, é composta por métodos vazios.

O último parâmetro *template* da classe `Dfs` está inteiramente ligado a questões de desempenho. Por padrão, a implementação de DFS da GTAD armazena os vértices, arestas-de-árvore e arestas-de-retorno encontrados durante a execução do algoritmo. Muitas vezes, esses dados são úteis para o usuário. Mas quando isso não acontece, é indesejável que o usuário pague por uma funcionalidade que não precisa. Sendo assim, basta que ele passe como argumento para a classe `Search_storable_items` apenas os items relativos aos dados desejados.

Há um detalhe na classe `Search_storable_items` que a torna versátil e elegante. Sua implementação foi feita de maneira a permitir a intercalação da ordem relativa entre seus parâmetros *template*. Na linguagem C++, quando um argumento *template* é especificado, todos os argumentos precedentes da definição da classe em questão também devem ser. Porém, na implementação da classe `Search_storable_items`, não existe uma ordem definida para especificação dos argumentos. Por exemplo, qualquer uma das opções abaixo constitui uma definição correta da classe:

```
Search_storable_items<Search_vertices>;
Search_storable_items<Tree_edges>;
Search_storable_items<Back_edges>;
Search_storable_items<Search_vertices, Tree_edges>;
Search_storable_items<Search_vertices, Back_edges>;
Search_storable_items<Tree_edges, Back_edges>;
Search_storable_items<Tree_edges, Search_vertices>;
Search_storable_items<Back_edges, Tree_edges>;
Search_storable_items<Back_edges, Search_vertices>;
Search_storable_items<Search_vertices, Tree_edges, Back_edges>;
Search_storable_items<Tree_edges, Back_edges, Search_vertices>;
Search_storable_items<Back_edges, Search_vertices, Tree_edges>;
Search_storable_items<Search_vertices, Back_edges, Tree_edges>;
Search_storable_items<Tree_edges, Search_vertices, Back_edges>;
```

```
Search_storable_items<Back_edges, Tree_edges, Search_vertices>;
```

Uma funcionalidade interessante que está presente na GTAD é a possibilidade de modificar partes do comportamento interno de alguns algoritmos. Esse recurso é conseguido através do uso de *políticas* [21], uma reminiscência do padrão *Strategy* [65]. Para exemplificá-la, considere o problema de fluxo máximo em uma rede [102, 19] e um algoritmo para sua solução [62]. Sem entrar em detalhes, o sumário do algoritmo pode ser descrito pelos seguintes passos.

1. Inicialize a rede com fluxo zero em todos os *arcos* (arestas do grafo).
2. Aumento o fluxo ao longo de determinado caminho que sai da *fonte* e leva ao *sumidouro*, sem extrapolar a *capacidade* dos *arcos*. Repita esse procedimento até que não existam mais tais caminhos na rede.

Em nenhum momento, o algoritmo especifica como devem ser encontrados os caminhos pela rede. A forma mais comum de implementação é utilizar os caminhos mais curtos, aqueles com o menor número de arestas. Bibliotecas como LEDA, BGL e GTL adotam essa estratégia. No entanto, nada impede que em determinado contexto, a melhor opção seja utilizar os caminhos mais longos da rede ou os caminhos que aumentam o fluxo em maior quantidade. Esse grau de genericidade é possível na GTAD, conforme mostrado na definição da classe `Augmenting_path_maxflow`.

```
template <
  class Graph_t,
  class Flow_unit_map_t =
    Vector_attr_map<typename Graph_traits<Graph_t>::Edge_handle, double>,
  class Reverse_edge_map_t =
    Vector_attr_map<typename Graph_traits<Graph_t>::Edge_handle,
      typename Graph_traits<Graph_t>::Edge_handle>,
  class Vertex_index_map_t =
    Vector_attr_map<typename Graph_traits<Graph_t>::Vertex_handle, int>,
  template <class> class Policy_agumenting_path_t = Policy_shortest_augmenting_path>
class Augmenting_path_maxflow :
  public Flow<
    Graph_t,
    Flow_unit_map_t,
    Reverse_edge_map_t,
    Vertex_index_map_t>
{
public:
  typedef Graph_t Graph;
  typedef Flow_unit_map_t Flow_unit_map;
  typedef Reverse_edge_map_t Reverse_edge_map;
  typedef Vertex_index_map_t Vertex_index_map;

  //...
};
```

O parâmetro *template* de interesse é o último, `Policy_agumenting_path_t`, que por sinal é um parâmetro *template template*. Este parâmetro indica qual política deve ser utilizada para en-

contrar os caminhos pela rede. A implementação padrão, `Policy_shortest_augmenting_path`, adota a estratégia de caminhos mínimos mencionada anteriormente. Mas o usuário é livre para parametrizar esse comportamento.

As classes que representam políticas devem ser capazes de acessar informações das classes que as utilizam. Por exemplo, a classe `Policy_shortest_augmenting_path` precisa acessar os valores que indicam o comprimento de cada aresta do grafo. Mas para mantê-las em uma forma realmente genérica, elas não devem estar acopladas às classes de algoritmos específicos. No caso de `Policy_shortest_augmenting_path`, pode ser que algum outro algoritmo, além do de fluxo máximo, necessite exatamente do mesmo comportamento por ela provido. Consequentemente, as classes de política são parametrizadas por tipos de classes de algoritmos, e por isso são parâmetros *template template*, conforme o código abaixo.

```
template <class Algorithm_t>
class Policy_shortest_augmenting_path
{
private:
  Policy_shortest_augmenting_path();

  typedef typename Algorithm_t::Graph Graph_t;
  typedef typename Graph_traits<Graph_t>::Vertex_handle Vertex_handle;
  typedef typename Graph_traits<Graph_t>::Edge_handle Edge_handle;

  //...

public:
  static void
  execute(Algorithm_t& algo,
          Graph_t& g,
          std::vector<Edge_handle>& path)
  {
    //...
  }
};
```

4.3.1 Implementações Oferecidas

São conhecidos inúmeros algoritmos de grafos. Vários deles podem suportar diferentes variações, como o caso do algoritmo de fluxo máximo mencionado acima. Além disso, há situações em que alguns algoritmos são componentes fundamentais para o desenvolvimento de outros. Exemplos interessantes são o de detecção de ciclos negativos em uma rede e de biconectividade em um grafo. Exatamente por motivos como esses, características de flexibilidade, genericidade e extensibilidade são importantes em uma biblioteca de grafos.

Atualmente, a GTAD oferece implementações para algoritmos tradicionais, como de BFS ou DFS, e para outros mais elaborados. A listagem seguinte introduz todos algoritmos atualmente suportados, com exceção daqueles diretamente relacionados a área de desenho de grafos, os quais serão apresentados em maiores detalhes nas próximas seções. Maiores informações sobre os algoritmos abaixo podem ser encontradas em [102, 46, 19, 42, 84, 57].

- ***Busca em Largura***
  Representado pela classe `Bfs`.
- ***Busca em Profundidade***
  Representado pela classe `Dfs`.
- ***Caminho Mínimo de Bellman-Ford***
  Representado pela classe `Bellman_ford_shortest_path`.
- ***Caminho Mínimo de Dijkstra***
  Representado pela classe `Dijkstra_shortest_path`.
- ***Fluxo Máximo por Incremento de Caminhos***
  Representado pela classe `Augmenting_path_maxflow`.
- ***Fluxo Máximo por Formação de Pré-Fluxo***
  Representado pela classe `Preflow_push_maxflow`.
- ***Fluxo de Custo Mínimo por Cancelamento de Ciclos***
  Representado pela classe `Cycle_cancelling_mincost_flow`.
- ***Teste de Planaridade de Boyer e Myrvold***
  Representado pela classe `Boyer_myrv_planarity_test`, que é um adaptador para uma implementação existente do algoritmo.
- ***Planarização por Adição de Vértices***
  Representado pela classe `Vertex_addition_planarization`.
- ***Biconectividade***
  Representado pela classe `Biconnectivity`.
- ***Dualidade por Ciclos Faciais***
  Representado pela classe `Facial_cycle_duality`.
- ***Numeração-st***
  Representado pela classe `St_numbering`.

É importante mencionar que as implementações seguem um modelo bastante modular, o que simplifica e facilita possíveis extensões. Algoritmos semelhantes herdam de classes base que fornecem a estrutura primária de dados utilizados no contexto em questão. Portanto, o trabalho de implementação de novos algoritmos é bastante reduzido, dado o alto grau de reutilização.

4.4 Algoritmos de Desenho e Relacionados

Os algoritmos de desenho são construídos seguindo o mesmo modelo dos algoritmos tradicionais. Porém, eles não implementam a visualização do grafo na tela. O objetivo dessa separação é justamente promover o desacoplamento com bibliotecas ou ferramentas gráficas.

Na biblioteca GTAD, o mecanismo padrão de visualização utiliza a OpenGL [13], uma especificação aberta e multiplataforma de bibliotecas de computação gráfica. No entanto, o usuário é livre para integrar qualquer outra biblioteca ou ferramenta, já que as interfaces das classes dos

algoritmos de desenho fornecem toda a informação necessária, como coordenadas de vértices ou dobras de arestas.

Uma consequência direta desse desacoplamento é que a visualização de um grafo deve ser realizada em dois passos: O primeiro consiste do algoritmo de desenho propriamente dito; O segundo é responsável pela visualização do grafo na tela. O código seguinte mostra a utilização do algoritmo *Pair*, apresentado no capítulo 3, e da função de visualização padrão para algoritmos de desenhos ortogonais.

```
typedef Adjacency_list<> Graph;
typedef Graph_traits<Graph>::Vertex_handle Vertex_handle;
typedef Graph_traits<Graph>::Edge_handle Edge_handle;

Graph g;
Vertex_handle v0 = insert_vertex(g);
Vertex_handle v1 = insert_vertex(g);
//...
Edge_handle e12 = insert_edge(v1, v2, g);
Edge_handle e13 = insert_edge(v1, v3, g);
//...

typedef Pair<Graph> Algorithm;
//...

Algorithm pair;
pair.vertex_index_map(Attr_map_util::make_index_map(g, Vertex_index_map()));
pair.edge_index_map(Attr_map_util::make_index_map(g, Edge_index_map()));
pair.execute(g);

basic_ortho_drawing(g, pair);
```

A função `basic_ortho_drawing` simplesmente obtém as coordenadas dos vértices e dobras de arestas computadas pelo algoritmo de desenho e renderiza os vértices como quadrados cheios e as arestas como linhas simples. Caso o usuário deseje continuar utilizando a função de visualização padrão do algoritmo, mas queira renderizar os vértices e arestas de outra forma, há uma solução simples. Ele pode implementar sua própria função de renderização e passar um ponteiro para ela na chamada à função `basic_ortho_drawing`, como mostra o código abaixo.

```
basic_ortho_drawing(g, pair, other_rendering_function);
```

4.4.1 Implementações Oferecidas

Atualmente, a GTAD oferece implementação para todos os algoritmos de desenho apresentados no capítulo 3. Todas elas assumem que os vértices do grafo de entrada têm grau máximo quatro.

As implementações dos algoritmos *Column* e *Pair* possuem um parâmetro *template*, correspondente à política de adição de arestas para tornar o grafo biconectado. Ela só é executada

quando necessário. A implementação padrão é ingênua: simplesmente insere uma aresta a cada dois vizinhos de um vértice-de-corte que não pertencem a um mesmo componente biconectado [100]. Já a implementação do *Giotto* não requer um tratamento especial para grafos não-biconectados.

Com o intuito de promover maior genericidade, a implementação do *Giotto* possui três parâmetros *template* que correspondem, respectivamente, às políticas de planarização, ortogonalização e compactação. Dessa forma, a variação de algoritmos de teste de planaridade e heurísticas de construção do *maximal* subgrafo planar é facilmente implementada. A escolha dos algoritmos de fluxo em redes, assim como de compactação de representações ortogonais, também é totalmente flexível.

A grande desvantagem do *Giotto* é o fato de sua complexidade de tempo ser $O(V^4)$, no pior caso. Um algoritmo também baseado na metodologia topologia-forma-métrica, conhecido como *Bend-Stretch*, possui complexidade de tempo linear e é apresentado em [112]. A única diferença deste algoritmo para o *Giotto* está na etapa de ortogonalização, que não garante o menor número de dobras. Conseqüentemente, ele pode ser implementado na GTAD apenas substituído-se a política que implementa a ortogonalização do *Giotto* por uma política que implemente sua própria ortogonalização.

A flexibilidade do uso de políticas também está presente nos algoritmos *Column* e *Pair* para computação da numeração-st. Como a qualidade do desenho resultante desses algoritmos é fortemente dependente de propriedades do grafo e da numeração-st [30], o usuário pode experimentar diferentes algoritmos de numeração-st e identificar qual a opção se enquadra melhor dentro do contexto desejado.

A estrutura de dados para manutenção de ordem das colunas nas implementações do Column e Pair é uma árvore binária balanceada. Especificamente, é o contêiner `std::map` da STL. Este é um ponto que ainda pode ser otimizado e até mesmo transformado em parâmetro *template*. Dessa forma, o usuário poderia utilizar sua própria estrutura de dados para tal tarefa.

As classes que representam os algoritmos de desenho mencionados acima são listadas abaixo. A estrutura hierárquica dessas classes é ilustrada pela Figura 4.5.

- `Giotto` - Implementação do algoritmo *Giotto*, descrito na Seção 3.1.
- `Column` - Implementação do algoritmo *Column*, descrito na Seção 3.2.
- `Pair` - Implementação do algoritmo *Pair*, descrito na Seção 3.3.

4.5 Utilitários Gerais

A GTAD oferece alguns recursos de usabilidade. A grosso modo, são funções simples que realizam tarefas corriqueiras e chatas de serem implementadas manualmente. Tais funções são normalmente estáticas e estão localizadas em classes utilitárias. A classe `Graph_util` é um exemplo. Ela provê funções para determinar se um vértice é uma das extremidades de uma aresta, para encontrar o segundo vértice de uma aresta quando o primeiro é fornecido, e para tornar um grafo bidirecionado.

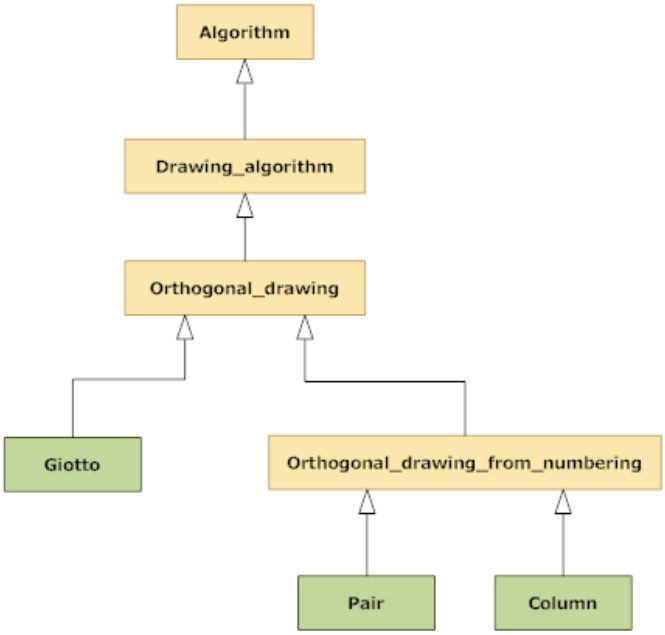

Figura 4.5: Hierarquia das classes de algoritmos de desenho.

O exemplo mais interessante desse tipo de funcionalidade está presente na classe `Attr_map_util`. Todos os algoritmos da biblioteca requerem algum tipo de parametrização por mapas, os quais são responsáveis por armazenar informações relacionadas aos atributos dos vértices e arestas. Considere a definição parcial da classe `Dijkstra_shortest_path`.

```
template <
  class Graph_t,
  class Weight_map_t =
    Vector_attr_map<typename Graph_traits<Graph_t>::Edge_handle, double>,
  class Vertex_index_map_t =
    Vector_attr_map<typename Graph_traits<Graph_t>::Vertex_handle, int> >
class Dijkstra_shortest_path :
  public Shortest_path<
    Graph_t,
    Weight_map_t,
    Vertex_index_map_t>
{
  //...

  void vertex_index_map(const Vertex_index_map_t& m);

  void weights(const Weight_map_t& weights);
};
```

O primeiro parâmetro *template* está associado à estrutura de dados de representação do grafo. O segundo e o terceiro são mapas de atributos que correspondem, respectivamente, aos pesos das arestas do grafo e a inteiros que serão utilizados para indexar os vértices do grafo

internamente. Para evitar que o usuário tenha que criar o mapa de vértices e a eles associar inteiros de 0 a $V-1$, a biblioteca fornece uma função que realiza essa tarefa. O código abaixo ilustra o uso da função `make_index_map()`.

```
Graph g; Vertex_handle v0 = insert_vertex(g);
Vertex_handle v1 = insert_vertex(g);
//...
Edge_handle e12 = insert_edge(v1, v2, g);
Edge_handle e13 = insert_edge(v1, v3, g);
//...

typedef Vector_attr_map<Vertex_handle, int> Index_map;
typedef Vector_attr_map<Edge_handle, double> Weight_map;
typedef Dijkstra_shortest_path<Graph, Index_map, Weight_map> Algorithm;
//...

Algorithm dijkstra;
dijkstra.vertex_index_map(Attr_map_util::make_index_map(g, Index_map());
```

Apesar de não parecer algo sofisticado à primeira vista, a função `make_index_map()` é bastante versátil. Não é necessário especificar se o mapa de índices a ser construído é de vértices ou arestas. Se for passado como argumento para a função uma instância de `Vector_attr_map <Vertex_handle, int>`, a indexação será feita sobre os vértices. Mas se o mapa for uma instância de `Vector_attr_map<Edge_handle, int>`, a indexação será feita sobre as arestas. Essa transparência é possível pelo uso da classe `Attr_map_traits`, que é capaz de detectar quais são os tipos dos dados definidos no mapa.

Há uma outra classe de *traits* que participa ativamente na elaboração de uma função que copia os atributos de um vértice ou aresta para determinado mapa. A classe `Member_funct_traits` define especializações parciais de *template* capazes de detectar a estrutura de funções membro que retornam tipos de dados primitivos (*int, double, short, long*). Este é o caso de uma hipotética função `double weight() const` da classe que representa o tipo de aresta do grafo. Através de uma construção que combina as classes de *traits* mencionadas e o recurso de C++ que permite a criação de ponteiros para funções membros, uma função utilitária que preenche um mapa com atributos de vértices e arestas pode ser implementada.

```
template <class Graph_t, class Attr_map_t>
static Attr_map_t&
make_edge_map(const Graph_t& g, Attr_map_t& m,
  typename Member_funct_traits<typename Attr_map_traits<Attr_map_t>::Key_t,
    typename Attr_map_traits<Attr_map_t>::Value_t>::Pointer p)
{
  typedef typename Graph_traits<Graph_t>::Edge_iterator Edge_iterator;
  std::pair<Edge_iterator, Edge_iterator> ei;
  for (ei = edges(g); ei.first != ei.second; ++ei.first)
  {
    typename Graph_traits<Graph_t>::Edge_handle e = *ei.first;
    typename Attr_map_traits<Attr_map_t>::Value_t attr = (e.*p)();
    m.put(e, attr);
```

```
  }
  return m;
}
```

A especialização da classe `Member_funct_traits` para funções que retornam valores do tipo `double` e a classe `Attr_map_traits` são exibidas a seguir.

```
template <class Class_t, typename Return_t>
class Member_funct_traits;

template <class Class_t>
class Member_funct_traits<Class_t, double>
{
public:
  typedef double (Class_t::*Pointer)();
  typedef double (Class_t::*Const_pointer)()const;
};

template <class Attr_map_t>
class Attr_map_traits
{
public:
  typedef typename Attr_map_t::Key_t Key_t;
  typedef typename Attr_map_t::Value_t Value_t;
};
```

Concluindo, caso o usuário da classe `Dijkstra_shortest_path` deseje transportar os valores dos pesos das arestas diretamente para o mapa que deve ser fornecido ao algoritmo, basta utilizar a seguinte sintaxe.

```
dijkstra.weights(Attr_map_util::make_edge_map(
  g, Weight_map(), &Edge_handle::weight));
```

4.6 Comentários Adicionais

A GTAD segue o modelo conceitual da BGL, que por sua vez adota as princípios da STL. Muitas das técnicas apresentadas neste capítulo também são observadas na BGL. Ambas bibliotecas de grafos são construídas sob o paradigma de programação genérica e fazem uso de técnicas semelhantes. No entanto, a arquitetura geral e, consequentemente, a forma de implementação da GTAD são completamente diferentes daquelas da BGL. Alguns pontos de mais alto nível que podem ser observados sobre esse assunto são discutidos nos próximos parágrafos. Porém, deve ficar claro que estas são observações bastante gerais, e não uma comparação minuciosa entre as duas bibliotecas.

Inicialmente, a lista de adjacência da GTAD implementa um modelo conceitual muito diferente daquele da BGL. Os próprios parâmetros *template* da estrutura são incompatíveis. A

GTAD oferece um mecanismo explícito de controlar a política de inserção de arestas no grafo. Na BGL, a presença ou não de auto-ciclos não pode ser parametrizada e a existência de arestas paralelas é dependente do contêiner STL utilizado para armazená-las. Além disso, a GTAD possibilita que o usuário defina seus próprios tipos de vértices e arestas, ao contrário de uma simples anexação de atributos a eles, como acontece na BGL. A vantagem dessa funcionalidade é que o código extra para navegação entre vértices (arestas) e seus atributos não é mais necessário. Adicionalmente, para que o mecanismo de atributos da BGL permita a navegação no sentido oposto (do atributo para o vértice ou aresta) é necessário incluir uma cópia de cada vértice ou aresta nos atributos a eles anexados.

Os algoritmos da GTAD são implementados como classes e alguns deles possibilitam parametrização de partes do comportamento interno e dos dados que serão armazenados. Essa funcionalidade não é encontrada na BGL, além de seus algoritmos serem implementados como funções. Mas, por outro lado, a BGL é mais poderosa em relação aos tipos de dados que podem ser utilizados em seus algoritmos. Por exemplo, a função de que calcula o menor caminho pelo algoritmo de Dijkstra recebe vários parâmetros *template*. O parâmetro correspondente aos pesos das arestas pode ser um mapa com tipos de dados quaisquer, opostamente a GTAD, que exige uma estrutura de dados que mapeie arestas para números. Devido ao fato do mapa de pesos ser totalmente flexível, a função de Dijkstra também deve possuir parâmetros *template* que indiquem o que é a "distância zero", o que é a "distância infinito"e como deve ser realizada a comparação dos valores do mapa.

Ambas bibliotecas se baseiam na idéia de atributos (na BGL são chamados de propriedades) para passar informações aos algoritmos. A especificação dos conceitos associados a atributos na BGL é mais complexa do que na GTAD. O exemplo do parágrafo anterior demonstra esse fato. Devido às características estruturais e da forma de utilização, os mapas da GTAD são consideravelmente mais simples e, provavelmente, atendem a grande parte dos requisitos. Afinal de contas, as chances do peso de uma aresta ter seu valor convertido em um número (inteiro ou de ponto flutuante) são grandes.

Por fim, no momento desta escrita a BGL não suporta nenhum algoritmo de desenho de grafos. Apesar de existirem vários algoritmos básicos na GTAD, seu foco principal é ser uma biblioteca voltada para o desenho de grafos. As implementações de caminho mínimo, máximo fluxo, entre outras, estão presentes porque são componentes estruturais de outros algoritmos. Como visto no capítulo 3, umas das etapas do algoritmo de desenho Giotto é construir uma rede associada ao embutimento planar do grafo. A execução de um algoritmo de fluxo de custo mínimo nessa rede é responsável por construir várias características do desenho.

Capítulo 5

Resultados

Este trabalho tem dois focos principais. O primeiro deles é relacionado ao estudo de algoritmos em grafos, com o foco em algoritmos de desenho de grafos. O segundo é um aspecto prático da engenharia de software que é o desenvolvimento de bibliotecas. Particularmente, bibliotecas que adotam o paradigma de programação genérica.

Salvo em alguns contextos específicos, é razoável considerar que o principal resultado de um algoritmo de desenho de grafo seja, obviamente, o desenho do grafo. Especificamente, em se tratando de desenhos ortogonais em grid, esse resultado é caracterizado pelas coordenadas de vértices e dobras de arestas. Este capítulo apresenta várias ilustrações de desenhos de grafos gerados com as implementações dos algoritmos do capítulo 3, disponíveis na biblioteca GTAD.

Por outro lado, uma avaliação completa da biblioteca desenvolvida é uma tarefa complexa, pois questões ligadas a flexibilidade, reutilização, manutenção, entre outras, não são claramente perceptíveis [54, 95, 96]. Vários pontos positivos relacionados à programação genérica, assim como as técnicas empregadas no desenvolvimento da biblioteca, foram abordados no capítulo 4. Mas diante das dificuldades, mencionadas acima, para avaliação completa de um software, e do fato de que muitas dessas informações não são de relevância no contexto deste trabalho, este capítulo se restringe a avaliar apenas resultados relativos ao desempenho da GTAD. Mais precisamente, são comparados os tempos de execução de algumas operações da representação do grafo da biblioteca e de alguns de seus algoritmos.

5.1 Avaliação de Desempenho

Inicialmente, esta seção apresenta comparações de desempenho entre a lista de adjacência da GTAD e representações de grafo de outras bibliotecas. Posteriormente, alguns algoritmos tradicionais também são comparados para o mesmo conjunto de bibliotecas.

5.1.1 Operações da Lista de Adjacência

Para a realização dos primeiros testes, foi definido um conjunto de operações *básicas*, comumente utilizadas para manipulação de grafos. Esse conjunto é composto pelos seguintes itens:

1. Operação *insere vértice*;
2. Operação *insere aresta*;
3. Operação *remove vértice*;
4. Operação *remove aresta*;
5. Operação *remove aresta por vértices*;
6. Operação *encontra aresta*

O significado semântico dos items acima é claramente obtido através dos próprios nomes das operações. No entanto, três deles devem ser enfatizados:

- A operação 3 espera que um vértice e todas suas arestas adjacentes sejam removidas do grafo;
- A operação 4 espera que uma aresta seja removida do grafo. Essa aresta deve ser passada como argumento para a função correspondente;
- A operação 5 espera que uma ou mais arestas sejam removidas do grafo. Dois vértices devem ser passados como argumento para a função correspondente. Todas as arestas compostas pelos vértices em questão devem ser removidas do grafo.

Para exemplificação e condução dos testes, foi simulada uma situação na qual um mapa é modelado por um grafo. Nesse mapa, os vértices são cidades e as arestas são estradas que as conectam. Conseqüentemente, os tipos de dados dos vértices e das aresta sãos similares aos descritos abaixo[1]:

```
class city
{
  int _id; //ID da cidade.
  std::string _name; //Nome da cidade.
  std::string _state; //Estado.
  int _population; //Populacao.
  double _area; //Area.

  //...
};

class highway
{
  int _id; //ID da estrada.
  std::string _name; //Nome da estrada.
  double _length; //Comprimento da estrada.
  int _speed_limit; //Velocidade maxima.
  int _lanes; //Numero de faixas.
```

[1]Para a GTAD, a definição do tipo de dado que representa a aresta do grafo é ligeiramente diferente, já que ela depende de dois parâmetros *template*.

```
  //...
};
```

As bibliotecas envolvidas nas comparações foram apresentadas no capítulo 4. São elas:

- LEDA (Library of Efficient Data types and Algorithms) 4.1
- BGL (Boost Graph Library) 1.33.1
- GTL (Graph Template Library) 1.2.2

Os testes foram executados em um Pentium 4 (2.8 GHz), sistema operacional Windows XP Professional (SP2) com 1 GB de RAM. Com exceção da LEDA, o código foi compilado com o Microsoft Visual C++ .NET 8.0. Infelizmente, a *Algorithmic Solutions Software GmbH*, empresa distribuidora da LEDA, não concedeu um versão atual da biblioteca para a realização dos testes. A solução encontrada foi utilizar uma das últimas versões com licença de pesquisa da LEDA (4.1), compilada com o Microsoft Visual C++ 6.0. Opções de compilação e ligação dos testes da LEDA foram mantidas o mais próximo possível das outras.

Naturalmente, se há necessidade de parametrização da representação do grafo com tipos de vértices e arestas definidos pelo usuário, é fundamental que a biblioteca permita isso, o que não acontece em nenhuma das bibliotecas acima. Porém, a LEDA e BGL oferecem um recurso similar para anexação de propriedades aos vértices e arestas do grafo. Para atingir o mesmo grau de abstração da parametrização direta, essas propriedades devem conter cópias dos vértices e arestas aos quais elas estão anexadas. Dessa forma, é possível a obtenção de informações em ambos sentidos: dos vértices (arestas) para as propriedades e das propriedades para os vértices (arestas).

No caso da GTL, o recurso de anexação de propriedades aos vértices e arestas não está presente. Portanto, para realização dos testes, foi necessário modificar o código-fonte das classes que representam os vértices e arestas do grafo.

A maioria das operações foram testadas com grafos direcionados e não-direcionados. Como a BGL permite parametrização dos contêineres de vértices e arestas, ambos foram testados como listas encadeadas (`listS`) e vetores (`vecS`). Adicionalmente, para grafos direcionados, foi utilizada a *tag* `didirectionalS`, que garante que informações sobre as arestas de saída e entrada de um vértice sejam computadas (as outras bibliotecas e a GTAD fazem isso como comportamento padrão). Outras observações a respeito dos testes são listadas abaixo:

- A operação *insere vértice* foi a única testada apenas com grafos não-direcionados. Todas as outras foram testadas com grafos direcionados e não-direcionados;
- A operação *insere aresta* foi testada com grafos previamente construídos com 50000 vértices;
- Todas as variações das operações de *remoção* e a operação de *encontra aresta* foram testadas com grafos esparsos de 25000 vértices e 150000 arestas, gerados aleatoriamente.

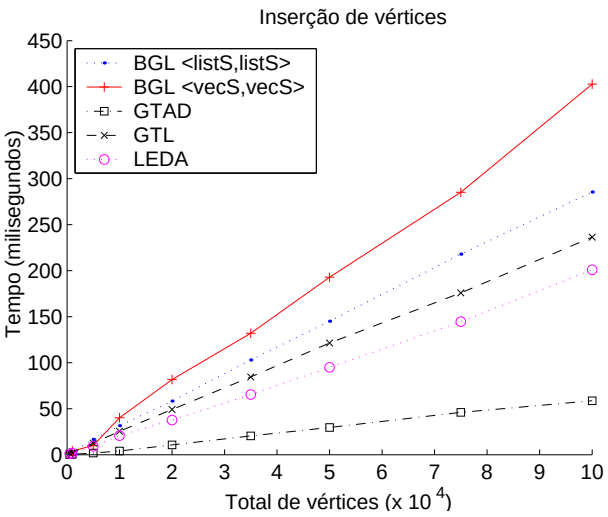

Figura 5.1: Insere vértice (grafo não-direcionado)

Todas as operações testadas apresentaram comportamento assintótico linear no número de vértices e arestas do grafo. As figuras 5.1 e 5.2 ilustram esse comportamento para as operações *insere vértice* e *insere aresta*. A figura 5.3 ilustra o mesmo comportamento para a operação *remove aresta*. A tabela 5.1 contém a média dos valores obtidos para todas as operações testadas.

Tabela 5.1: Tempos de execução das operações (ms).

Operação	Número de operações	Tipo do grafo	**GTAD**	**BGL (list)**	**BGL (vec)**	**GTL**	**LEDA**
insere vértice	100000	não-direcionado	**58**	285	402	236	200
insere aresta	100000	não-direcionado	**235**	526	574	1323	430
insere aresta	100000	direcionado	**229**	523	566	1255	434
remove vértice	10000	não-direcionado	**109**	308	-	520	137
remove vértice	10000	direcionado	**101**	255	-	522	132
remove aresta	50000	não-direcionado	**72**	188	97	239	92
remove aresta	50000	direcionado	**66**	151	92	291	92
remove aresta por vértices	50000	não-direcionado	**86**	269	102	ND	ND
remove aresta por vértices	50000	direcionado	**67**	212	95	ND	ND
encontra aresta	50000	não-direcionado	**45**	132	46	ND	ND
encontra aresta	50000	direcionado	**35**	85	43	ND	ND

Como mostra a tabela 5.1, para o cenário em questão, o desempenho da GTAD foi o melhor em todas as operações, sendo que para algumas delas, a diferença foi significativa. Particularmente, para operação *insere vértice*, o tempo de execução foi quase quatro vezes menor que o da LEDA. Para as duas variações da operação *insere aresta*, o tempo de execução da GTAD foi quase duas vezes menor que o da LEDA e mais de duas vezes menor que o da BGL.

Uma observação interessante é que a operação *insere aresta* da BGL é mais eficiente quando o contêiner de arestas é parametrizado por uma lista encadeada (`listS`). No entanto, para a operação *remove aresta*, essa parametrização se mostrou, surpreendentemente, pior do que quando o contêiner de arestas é parametrizado por um vetor (`vecS`). Também foi constatado que a operação *remove vértice* da BGL é extremamente lenta quando o contêiner de vértices é parametrizado por um vetor - razão da utilização do hífen na tabela 5.1.

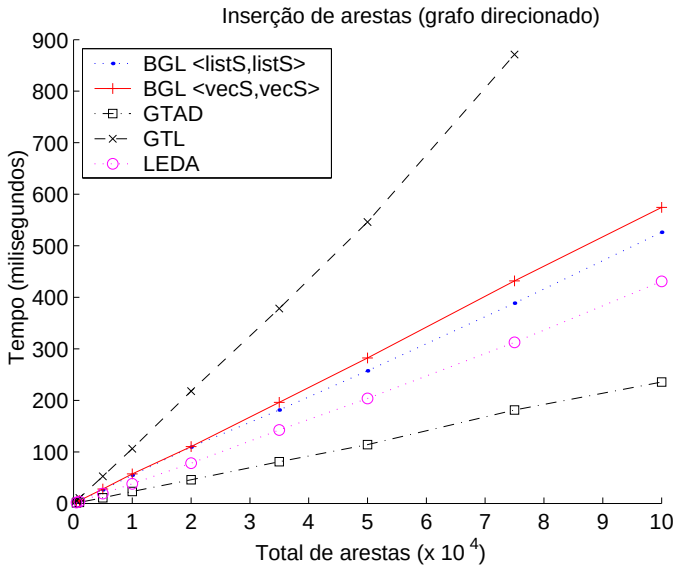

Figura 5.2: Insere aresta (grafo direcionado)

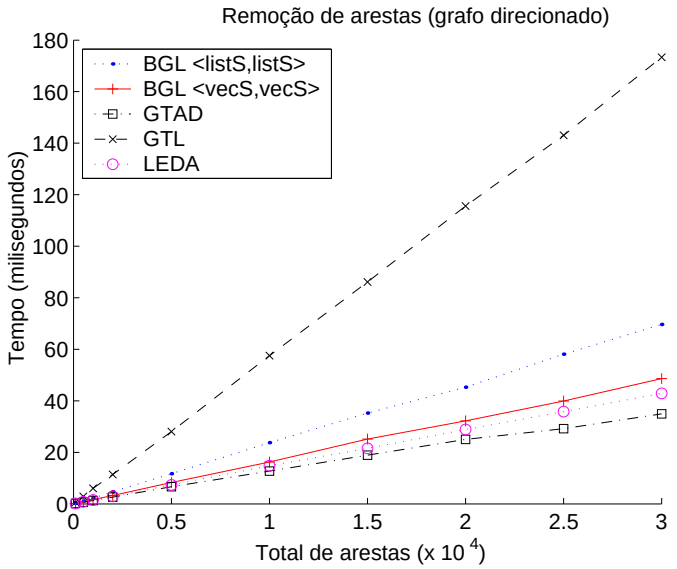

Figura 5.3: Remove aresta (grafo direcionado)

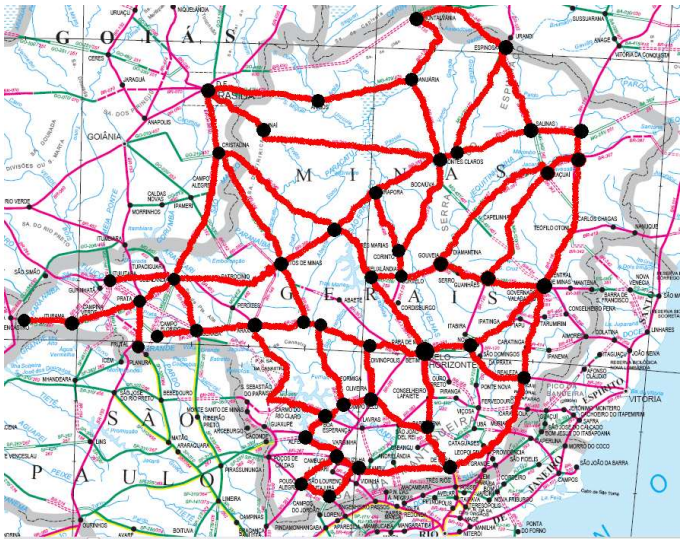

Figura 5.4: Grafo utilizado nos testes (em destaque no mapa).

5.1.2 Algoritmos

As comparações de desempenho das implementações dos algoritmos foram realizadas para um conjunto pequeno. A razão disso é que nem todas as bibliotecas oferecem implementações para os mesmos algoritmos, os quais ainda podem existir em inúmeras variações. Outro ponto que torna o processo comparativo mais complexo são as diferenças dos parâmetros de entrada e saída de implementações distintas. Além disso, em alguns casos, informações *extras* são construídas durante a execução do algoritmo. Apesar de contribuir negativamente para o desempenho, elas podem aumentar o grau de flexibilidade e serem úteis para o usuário. Finalmente, a geração de grafos de testes nem sempre é uma tarefa trivial [45].

Foi comparado o tempo de execução dos algoritmos de busca e caminho mínimo. Precisamente, das implementações de DFS, BFS, Bellman-Ford e de Dijkstra das mesmas bibliotecas das comparações anteriores. O grafo utilizado nos testes, ilustrado na figura 5.4, foi elaborado a partir do mapa rodoviário do estado de Minas Gerais, ilustrado na figura 5.5. As distâncias entre pontos do mapa, correspondentes aos custos das arestas, são apenas aproximações. Devido a algumas restrições e peculiaridades das bibliotecas, apenas grafos não-direcionados foram envolvidos nos testes. Os contêineres de vértices e arestas da BGL foram parametrizados, respectivamente, por uma lista encadeada (`listS`) e um vetor (`vecS`). A tabela 5.2 mostra os resultados obtidos.

Tabela 5.2: Tempos de execução dos algoritmos (μs).

Algoritmo	Número de execuções	**GTAD**	**BGL**	**GTL**	**LEDA**
BFS	1000	**10.3**	10.5	189.1	10.6
DFS	1000	**8.2**	16.4	196.9	11.7
Dijkstra	1000	148.5	**22.3**	962.9	40.5
Bellman-Ford	1000	19.8	**12.02**	5654	38.5

A tabela 5.2 mostra resultados interessantes. Com relação aos algoritmos de busca, os tempos de execução foram bastante similares, com exceção daqueles da GTL, notavelmente mais lentos. Particularmente para as implementações de DFS na GTAD, BGL e LEDA, a variação da medida

Figura 5.5: Mapa rodoviário de Minas Gerais [11].

de desempenho é praticamente desprezível.

A BGL obteve, claramente, os melhores resultados para os algoritmos de caminho mínimo. Por outro lado, a GTAD, apesar de ter o segundo melhor tempo de execução para o algoritmo de Bellman-Ford, se mostrou relativamente lenta para o algoritmo de Dijkstra. A razão disso deve ser analisada com mais cautela.

5.2 Desenhos de Grafos Gerados

Esta seção apresenta desenhos de grafo gerados pela GTAD e aponta diferenças interessantes entre os algoritmos do cap. 3. Algumas delas já foram mencionadas. Porém, a constatação e comparação dos resultados de cada uma das abordagens, sob um ponto de vista prático, é de grande valia.

Os critérios estéticos do cap. 2 constituem a principal ferramenta de análise dos desenhos. Com exceção daqueles relacionados à simetria e resolução angular, todos os outros são aqui discutidos. Critérios de simetria estão fora deste conjunto, pois exigem uma análise formal, bem detalhada e com forte embasamento matemático, que foge do escopo deste trabalho. Já a resolução angular, não é um critério importante dentro da convenção de desenhos ortogonais em grid.

Critérios estéticos como minimização do número de dobras ou cruzamento de arestas são claramente perceptíveis a partir de um desenho, mesmo que este não esteja necessariamente posicionado sobre um grid. Outros critérios, como área e comprimento das arestas, são mais fáceis de serem analisados se os desenhos estiverem sob a mesma escala, o que acontece nos desenhos de grafos ilustrados nas próximas seções.

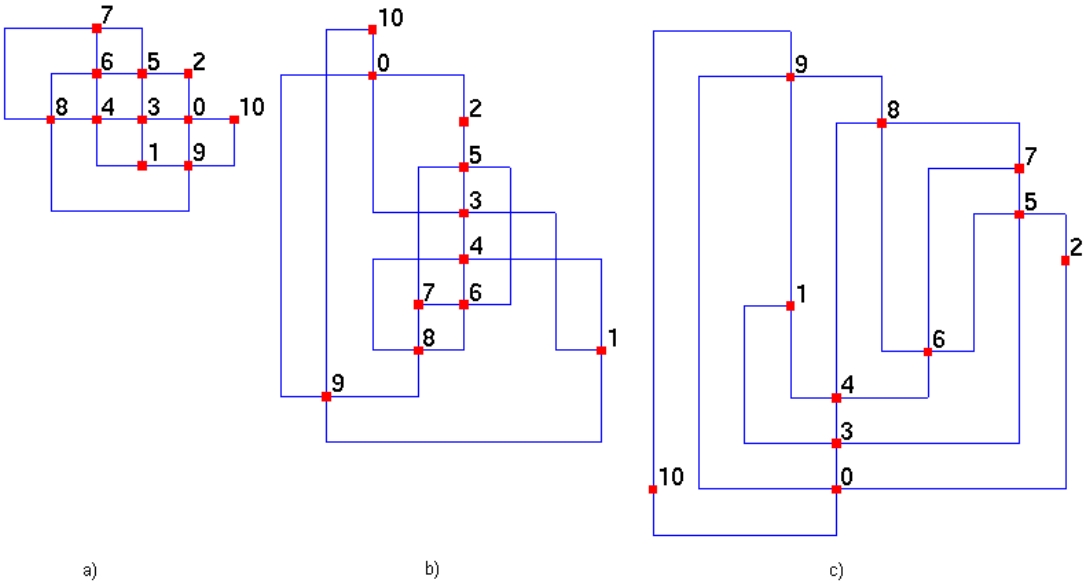

Figura 5.6: Desenhos de um grafo planar: a) Giotto; b) Pair; c) Column.

5.2.1 Grafo Planar Biconectado

Existe uma grande diferença entre os algoritmos *Giotto*, *Column* e *Pair*, quando considerados aspectos relativos à planaridade. O primeiro e o segundo garantem que um grafo planar seja, de fato, desenhado sem cruzamento de arestas, enquanto que o último não impõe nenhuma garantia a esse respeito. A figura 5.6 mostra três desenhos de um mesmo grafo planar biconectado, gerados pelos algoritmos Giotto, Pair e Column.

Claramente, o algoritmo que faz o melhor aproveitamento da área é o *Giotto*. E apesar de não garantir um desenho planar, o algoritmo *Pair* produz um desenho de área menor que a do algoritmo *Column*. Esse fato é uma consequência direta do pareamento de vértices e reutilização de linhas e colunas do grid. Particularmente, para o grafo da figura 5.6-b, são formados três pares: um deles pelos vértices 1 e 8, os quais reutilizam uma linha; outro pelos vértices 7 e 6, os quais também reutilizam uma linha; por último, há um par formado pelos vértices 3 e 4, os quais reutilizam uma coluna.

O *Giotto* também é o melhor algoritmo em termos de minimização do número de dobras total do desenho. Conforme mencionado no cap. 3, essa garantia advém da modelagem de equivalência entre a ortogonalização e o problema de fluxo de menor custo em uma rede. Já os algoritmos *Pair* e *Column* se focam em garantir o número máximo de dobras por arestas, e consequente uniformidade, o que não acontece no *Giotto*. Salvo uma situação específica, mencionada no cap. 3, na qual o desenho gerado pelo *Column* pode ter uma única aresta com três dobras, todas as outras arestas dos desenhos gerados pelo *Pair* e pelo *Column* têm no máximo duas dobras.

A razão de aspecto e critérios relacionados ao comprimento das arestas não são tratados explicitamente no algoritmo *Giotto*. Mas devido às etapas de ortogonalização e compactação, é provável que a distribuição dos vértices no desenho contribua para que a razão de aspecto e o comprimento das arestas sejam pequenos. O *Pair* e o *Column* possuem características similares para esses critérios. Resultados práticos [31] mostram que o *Pair* é ligeiramente mais eficiente que o *Column*, mas menos eficiente que o *Giotto*.

Um ponto importante, não relacionado a critérios estéticos, é o tempo de execução dos algoritmos. As implementações dos algoritmos *Pair* e *Column* da GTAD têm complexidade de tempo $O(lg(V))$. Já a implementação do Giotto, é $O(V^4)$, no pior caso. Conseqüentemente, o tempo gasto para construção do desenho da figura 5.6-a é significativamente maior que aquele gasto para as figuras 5.6-b e 5.6-c. Portanto, o balanceamento de fatores associados ao tempo de execução e aos variados critérios estéticos deve ser determinado em função dos requisítos da aplicação.

5.2.2 Grafo Não-Planar Biconectado

As características de desenhos de grafos não-planares biconectados gerados pelos algoritmos *Giotto*, *Pair* e *Column* são praticamente as mesmas mencionadas na seção anterior. O único critério estético que merece uma análise separada e mais detalhada é o de minimização de cruzamento de arestas.

Ambos os algoritmos *Giotto* e *Column* garantem desenhos planares de grafos planares. Mas quando o grafo é não-planar, a relação entre o número de cruzamentos de arestas produzido por esses dois algoritmos é bastante diferente. Conforme apresentado no cap. 3, a primeira etapa do algoritmo *Giotto* é a de planarização, a qual substitui os cruzamentos de arestas por vértices falsos, através da construção de um *maximal* grafo planar e adição de arestas não-planares sucessivamente. Como as arestas são adicionadas a partir de um caminho de custo mínimo no grafo dual, conforme descreve o procedimento 2, o desenho resultante terá um número pequeno de cruzamento de arestas.

O algoritmo *Column* não introduz nenhum tipo de tratamento para minimização do número de cruzamento de arestas. O mesmo acontece com o algoritmo *Pair*. Particularmente para os desenhos da figura 5.7, o número de cruzamento de arestas gerado pelo *Giotto* é duas vezes menor que aquele gerado pelo *Column*, e mais de três vezes menor que aquele gerado pelo *Pair*.

5.2.3 Grafo Não-Biconectado

Grafos não-biconectados não podem ser submetidos diretamente aos algoritmos *Pair* e *Column*. O motivo dessa limitação é a necessidade de computação de uma numeração-st, que é definida apenas para grafos biconectados.

A melhor solução para esse problema é realizar uma decomposição em blocos biconectados, aplicar o algoritmo desejado com as modificações necessárias e compor o desenho final a partir de desenhos dos subgrafos criados. O principal problema dessa abordagem é a dificuldade de implementação. Adicionalmente, as modificações necessárias para tratamento de blocos biconectados são geralmente específicas para cada algoritmo [94][37].

Uma abordagem mais simples e generalizável, adotada na GTAD, é a de construir, sempre que necessário, um grafo biconectado, através da adição de arestas falsas. A desvantagem dessa estratégia é que garantias, especialmente de área, podem ser perdidas. Precisamente, quanto maior o número de arestas falsas adicionadas, maiores são as chances de isso acontecer.

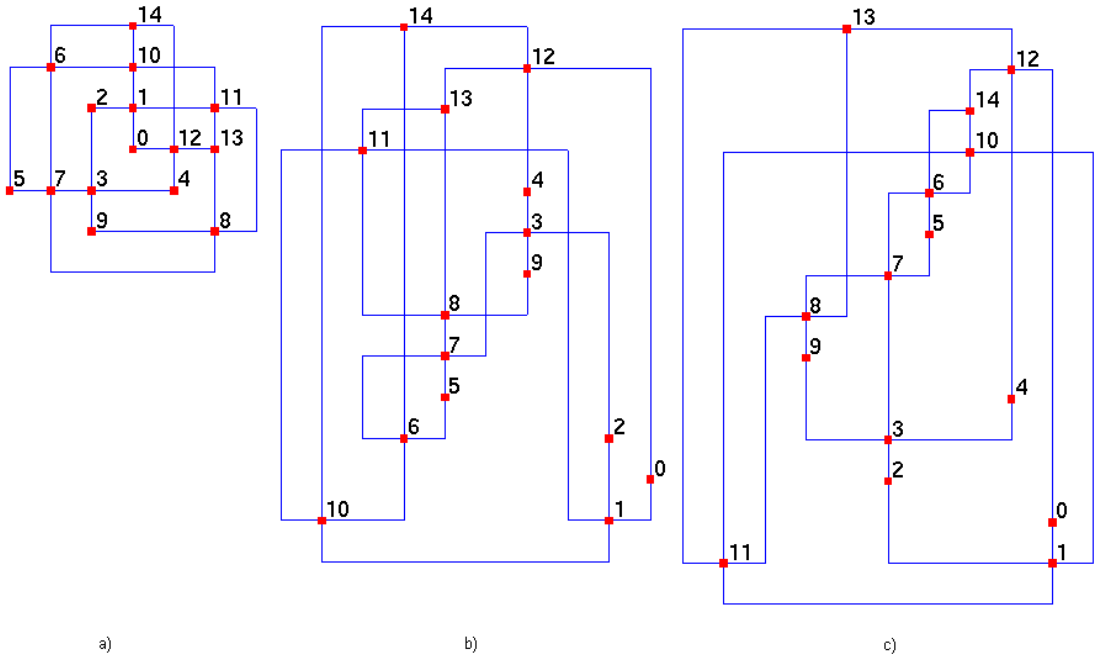

Figura 5.7: Desenhos de um grafo não-planar: a) Giotto; b) Pair; c) Column.

O reflexo da adição de arestas falsas em grafos não-biconectados, para os algoritmos *Pair* e *Column*, pode ser observada nos desenhos ilustrados nas figuras 5.9 e 5.10. Em ambos os casos, a área do desenho é notadamente maior que a necessária, pois todos os vértices de grau um ocupam linhas extras do grid.

O grafo da figura 5.8 foi retirado de uma aplicação real. Ele representa parte de um diagrama elétrico de uma companhia de energia do Brasil [10]. O desenho original desse grafo, gerado por um desenhista, para compor o painel sinóptico no qual ele aparece, é ilustrado na figura 5.8-a. A figura 5.8-b ilustra o desenho desse grafo gerado pela implementação do *Giotto* da GTAD.

Sob todos os critérios estéticos analisados nas seções anteriores, o desenho gerado pelo *Giotto* supera os desenhos gerados pelo *Pair* e pelo *Column*. Particularmente para este caso, as diferenças de área, número de cruzamento de arestas, entre outros critérios, é imensa.

Novamente, é válido mencionar que o *Giotto* é o algoritmo mais lento dos três. Sua implementação também é a mais difícil e trabalhosa. Portanto, cabe ao usuário identificar os requisitos da aplicação e escolher a melhor opção de algoritmo. Caso o tempo de geração do desenho não seja um fator predominante, o *Giotto* é normalmente uma boa escolha. A tabela 5.3 mostra os tempos de execução dos algoritmos *Giotto*, *Pair* e *Column*, para os grafos desta seção.

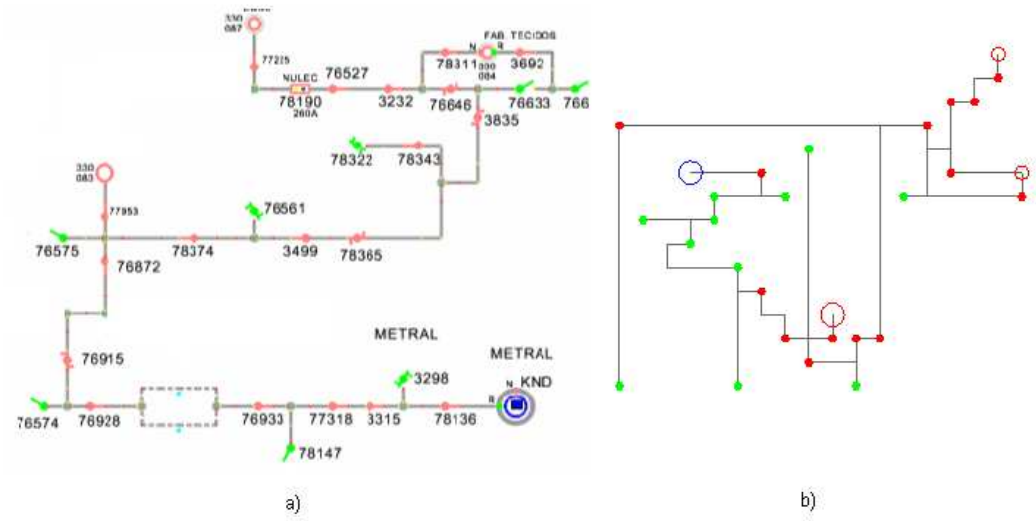

Figura 5.8: Diagrama elétrico: a) Original; b) Desenhado com Giotto.

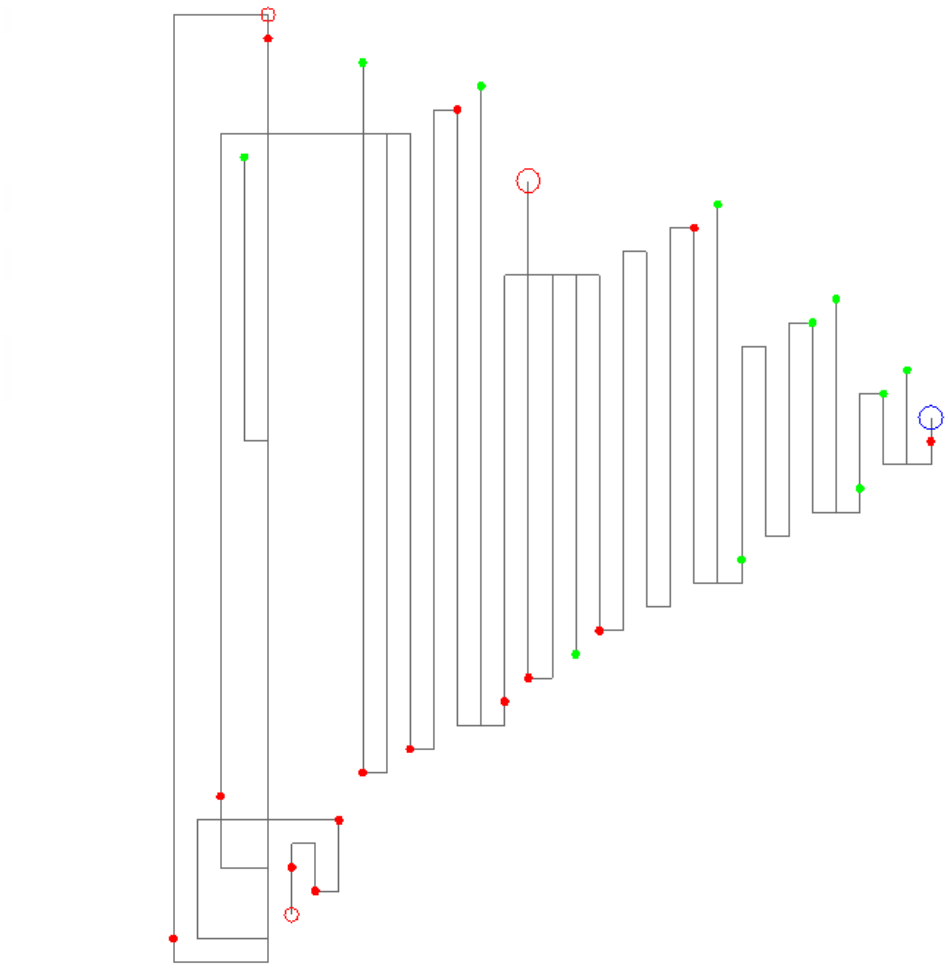

Figura 5.9: Diagrama elétrico desenhado com Pair.

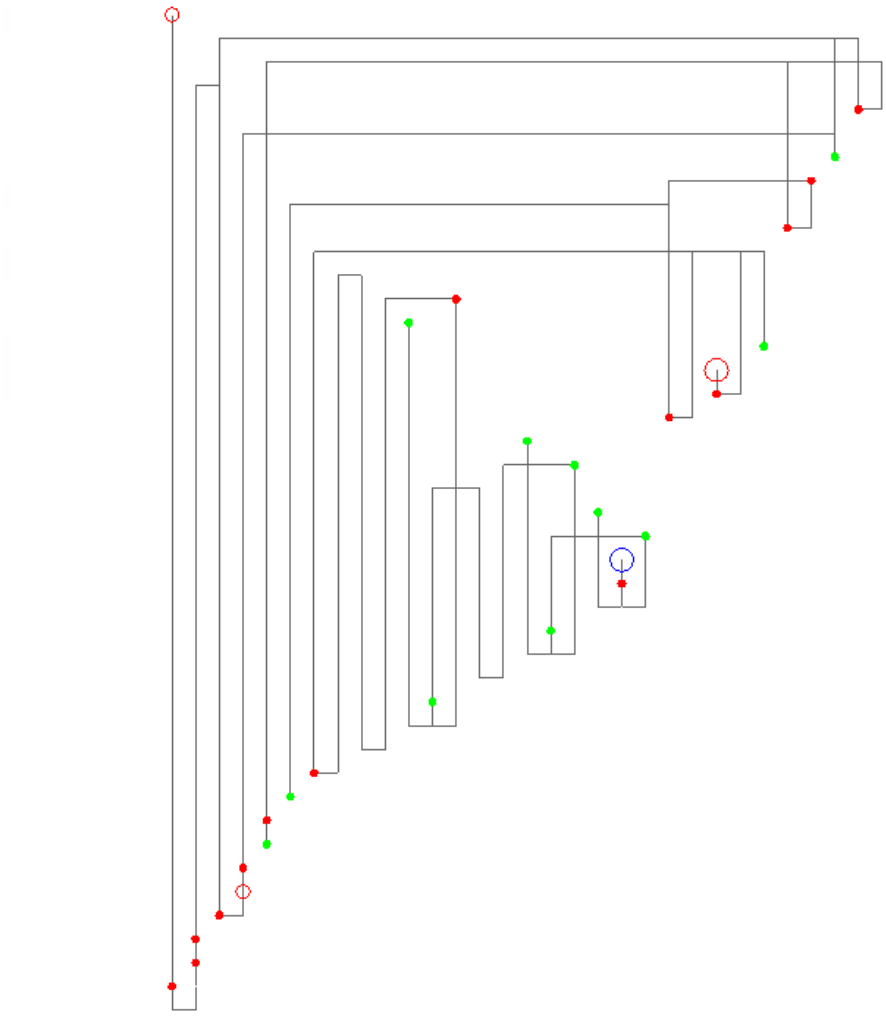

Figura 5.10: Diagrama elétrico desenhado com Column.

Tabela 5.3: Tempos de execução dos algoritmos de desenho (ms).

	Giotto				*Pair*	*Column*
	Planarização	Ortogonalização	Compactação	Total	Total	Total
Grafo Planar Biconectado	39.5	566.9	451.5	1057.9	24.3	**7.7**
Grafo Não-Planar Biconectado	121.5	1269.7	575.0	1966.2	11.3	**9.7**
Grafo Não-Biconectado	177.5	1746.5	1033.6	2957.6	**165.6**	176.9

Capítulo 6

Conclusão

Este trabalho mostrou um panorama geral e introdutório sobre a área de desenho de grafos. Um estudo aprofundado de alguns algoritmos de desenho ortogonal, incluindo detalhes importantes de implementação, e o desenvolvimento de uma biblioteca genérica, flexível e extensível de algoritmos de grafos foi apresentado.

A biblioteca desenvolvida, chamada de GTAD (*Graph Toolkit for Algorithms and Drawings*), adota o paradigma de programação genérica. Conseqüentemente, a estrutura de dados que representa o grafo pode ser substituída por outras implementações, desde que sejam seguidas as regras envolvidas. É provável que na maioria das aplicações, a própria lista de adjacência da GTAD atenda aos requisitos necessários. Pois, além de possibilitar a parametrização dos vértices, arestas e outras propriedades do grafo, seu desempenho é excelente, quando comparado com o de representações de grafos de outras bibliotecas. Os testes realizados mostram que, sob as condições de avaliação, a lista de adjacência da GTAD é mais rápida que representações de grafos das bibliotecas comparadas em várias operações fundamentais, como inserção de vértices e arestas.

Com o intuito de promover um alto grau de flexibilidade, as implementações de algoritmos da GTAD são altamente parametrizáveis. Há recursos que permitem a personalização de partes do comportamento interno do algoritmo e a definição de quais dados devem ser armazenados durante sua execução. Como essas configurações são feitas em tempo de compilação, não há impactos ou sobrecarga no desempenho. De fato, algoritmos de busca e caminho mínimo tiveram seus tempos de execução comparados com os de outras bibliotecas, e os resultados foram positivos. Particularmente, as implementações de DFS e BFS da GTAD são as mais rápidas do conjunto avaliado.

Os algoritmos de desenho de grafos apresentados, *Giotto*, *Pair* e *Column*, são referências importantes e bastante estudadas na literatura [30]. Cada um deles oferece vantagens e desvantagens, sob o ponto de vista de inúmeros critérios estéticos, desempenho e dificuldade de implementação.

Duas qualidades interessantes do *Giotto* são a minimização do número de cruzamento de

arestas e a minimização do número de dobras. A primeira delas pode ser atingida por qualquer algoritmo que defina uma etapa prévia de planarização do grafo. A outra, mais elaborada, é decorrente da modelagem do problema através da maximização do fluxo em uma rede. Há algoritmos mais recentes que também garantem o número mínimo de dobras, para categorias específicas de grafos, através de abordagens diferentes [36, 98, 99].

Apesar de ser mais lento e de implementação mais trabalhosa do que os algoritmos *Pair* e *Column*, a qualidade dos desenhos gerados pelo *Giotto* é relativamente melhor. Experimentos práticos mostram que esse resultado é observado para uma grande categoria de grafos [31]. Adicionalmente, a implementação do *Giotto* da GTAD trata grafos não-biconectados diretamente, o que não acontece para os outros algoritmos, que são submetidos a um processo de adição de arestas.

Todas as implementações de algoritmos de desenho da GTAD suportam apenas grafos nos quais os vértices têm grau máximo quatro. Idealmente, essa limitação não deveria existir. No entanto, retirá-la é uma tarefa difícil, e que deve ser realizada individualmente para cada algoritmo, pois, apesar da estratégia geral ser a mesma, os detalhes de implementação variam. Mesmo assim, vale mencionar que existem aplicações reais, como a descrita no capítulo 5, onde essa limitação não acarreta nenhum empecilho, pois os grafos sempre possuem vértices de grau menor ou igual a quatro. Em última instância, também há a possibilidade de subdividir os vértices de grau maior que quatro em uma etapa de modelagem anterior à execução do algoritmo.

A visualização de desenhos gerados pelos algoritmos da GTAD deve ser feita programaticamente, ou seja, escrevendo código em C++. Não há, por exemplo, uma funcionalidade para geração de arquivos descritivos. Na verdade, não existe um formato de arquivos para representação de grafos e desenhos que seja globalmente ou oficialmente estabelecido como padrão. Provavelmente, a solução adotada pela GTAD no futuro será de fornecer arquivos em formatos variados [5, 7].

Finalmente, deve-se ressaltar que não foram encontradas bibliotecas de grafos alternativas que agrupem os seguintes benefícios presentes na GTAD:

- Lista de adjacência parametrizada pelos vértices, arestas e outra propriedades do grafo;
- Flexibilidade para utilização de representações de grafos proprietárias;
- Parametrização de partes do comportamento interno de algoritmos, assim como dos dados que devem ser armazenados durante sua execução;
- Flexibilidade para utilização de estruturas de dados proprietárias, que correspondem aos mapas de dados processados nos algoritmos;
- Implementações de algoritmos de desenho de grafos.
- Alto desempenho da lista de adjacência e de implementações de alguns algoritmos [1];
- Código-fonte aberto;
- Utilização livre (licença não-comercial).

[1] Não é possível garantir que todos algoritmos possuem implementações eficientes, pois não foram feitas medidas formais para todos eles.

Um assunto interessante para trabalhos futuros e que não foi abordado neste texto é adição de restrições ao desenho. A etapa de ortogonalização do *Giotto* é responsável pela garantia do menor número de dobras no desenho. Porém, a abordagem original de modelagem através de um problema de fluxo em redes impõe várias limitações ao algoritmo. Por exemplo, é muito difícil tratar restrições relativas ao posicionamento e tamanho de determinados vértices ou arestas. Para atender necessidades como essa, existe a alternativa de remodelar a etapa de ortogonalização do *Giotto* através de programação linear inteira [55, 56]. Nesse caso, restrições podem ser adicionados ao desenho com maior naturalidade, já que elas correspondem a um conjunto de novas equações ou inequações do sistema.

Além de restrições relacionadas ao posicionamento e tamanho de vértices e arestas, entre muitas outras, é relativamente comum a necessidade de adicionar labels (ou nomes) aos componentes do grafo. Essa não é uma tarefa trivial, pois o desenho deve ser elaborado de forma que haja espaço suficiente para adição de um ou até vários labels a determinados vértices e arestas. Naturalmente, é desejável que tais labels sejam localizados próximos aos vértices e arestas correspondentes, tornando o problema ainda mais difícil. Diferentes propostas para solução desse problema são apresentadas em [39, 55, 56, 80, 81].

Etapas de pós-processamento também podem ser utilizadas para refinamento de desenhos. Particularmente, a área, número de cruzamentos de arestas, número de dobras e comprimento total de arestas de um desenho ortogonal podem ser significativamente diminuídos. No caso do *Giotto*, esses números podem chegar, respectivamente, a 29%, 11%, 13% e 24%, da medida original, para cada um desses items. Técnicas com esse objetivo são apresentadas em [105]. Porém, há ainda um grande número de situações e variações de refinamento que podem ser abordadas em trabalhos futuros.

O algoritmo *Column* é o de implementação mais simples dos três apresentados neste trabalho. No entanto, quando avaliado sob determinados critérios estéticos, é o que produz piores resultados. Já o *Giotto*, é um algoritmo eficiente em inúmeros critérios, mas com implementação difícil e trabalhosa, além de ser totalmente dependente da etapa de planarização. Uma questão ainda aberta e de grande importância, que pode ser estudada em trabalhos futuros, é a de criação de novos algoritmos de desenho ortogonal que atinjam resultados tão bons quanto os do *Giotto*, mas que sejam de fácil implementação e não dependam de uma etapa de planarização.

Referências Bibliográficas

[1] aisee - graph visualization. http://www.aisee.com/ [10 de Junho de 2006].

[2] Astral. http://www.ic.unicamp.br/ rezende/Astral.htm [10 de Junho de 2006].

[3] Cgal - computational geometry algorithms library. http://www.cgal.org/ [13 de Fevereiro de 2007].

[4] Giden - a graphical environment for network optimization. http://giden.northwestern.edu/ [10 de Junho de 2006].

[5] The gml file format. http://www.infosun.fim.uni-passau.de/Graphlet/GML/ [22 de Abril de 2007].

[6] Govisual software libraries. http://www.oreas.com/ [1 de Fevereiro de 2007].

[7] The graphml file format. http://graphml.graphdrawing.org/ [22 de Abril de 2007].

[8] Grin (graph interface). http://www.geocities.com/pechv_ru/ [10 de Junho de 2006].

[9] Jgraph - free graph library. http://jgrapht.sourceforge.net/ [10 de Fevereiro de 2007].

[10] Light energia. http://www.lightenergia.com.br [18 de Abril de 2007].

[11] Mapa rodoviário de minas gerais. http://www.transportes.gov.br/bit/mapas/mapdoc/ufs/mapas-dnit/Brasil_PNV.pdf [25 de Março de 2007].

[12] Open problems - graph theory and combinatorics. http://www.math.uiuc.edu/∼ west/openp/ [20 de Novembro de 2006].

[13] Opengl - the industry's foundation for high performance graphics. http://www.opengl.com/ [14 de Fevereiro de 2007].

[14] Openjgraph. http://openjgraph.sourceforge.net/ [10 de Junho de 2006].

[15] P.i.g.a.l.e. - public implementation of a graph algorithm library and editor. http://pigale.sourceforge.net/ [1 de Fevereiro de 2007].

[16] Standard template library programmer's guide. http://www.sgi.com/tech/stl/table_of_contents.html [17 de Maio de 2007].

[17] Tom sawyer software. http://www.tomsawyer.com/home/index.php/ [10 de Junho de 2006].

[18] yworks - the diagramming company. http://www.yworks.com/ [10 de Junho de 2006].

[19] R. K. AHUJA, T. L. MAGNANTI, and J. B. ORLIN. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, New Jersey, USA, first edition, 1993.

[20] D. ALBERTS, C. GUTWENGER, P. MUTZEL, S. NAHER, G. F. ITALIANO, and S. ORLANDO. Agd library: A library of algorithms for graph drawing. *Proceedings of the Workshop on Algorithm Engineering (WAE '97) - Venice, Italy*, pages 112–123, September 1997.

[21] A. ALEXANDRESCU. *Generic Programming and Design Patterns Applied*. Addison-Wesley, USA, 2001.

[22] Algorithmic-Solutions. Leda (library of efficient data types and algorithms. http://www.algorithmic-solutions.com/enleda.htm.

[23] K. ARNOLD, J. GOSLING, and D. HOLMES. *The Java Programming Language*. Addison-Wesley, fourth edition, 2005.

[24] L. AUSLANDER and S. V. PARTER. On imbedding graphs in the plane. *J. Math. and Mech.*, 10:517–523, 1961.

[25] M. H. AUSTERN. *Generic Programming and the STL*. Addison-Wesley Longman, USA, first edition, 1998.

[26] C. BATINI, E. NARDELLI, and R. TAMASSIA. A layout algorithm for data-flow diagrams. *IEEE Trans. Softw. Eng.*, 4:538–546, 1986.

[27] C. BATTINE, L. FURLANI, and E. NARDELLI. What's a good diagram? a pragmatic approach. *In Proc. 4th Internat. Conf. on the Entity Relationship Approach*, 1985.

[28] G. D. BATTISTA, W. DIDIMO, A. LEONFORTE, M. PATRIGNANI, and M. PIZZONIA. Gdtoolkit (graph drawing toolkit). http://www.dia.uniroma3.it/ gdt/index.html.

[29] G. DI BATTISTA, P. EADES, R. TAMASSIA, and I. TOLLIS. Algorithms for drawing graphs: An annotated bibliography. Technical report cs-89-09, Brown University, 1994.

[30] G. DI BATTISTA, P. EADES, R. TAMASSIA, and I. TOLLIS. *Graph Drawing - Algorithms for the visualization of graphs*. Prentice-Hall, New Jersey, USA, first edition, 1999.

[31] G. DI BATTISTA, A. GARG, G. LIOTTA, R. TAMASSIA, E. TASSINARI, and F. VARGIU. An experimental comparison of four graph drawing algorithms. *11th ACM Symposium on Computational Geometry*, 7(5-6):303–325, 1997.

[32] G. DI BATTISTA and R. TAMASSIA. Algorithms for plane representations of cyclic digraphs. *Theoret. Comput. Sci*, 61:175–198, 1988.

[33] G. DI BATTISTA, T. TAMASSIA, and I. TOLLIS. Area requirement and symmetry display of planar upward drawings. *Discrete Comput. Geom.*, 7:381–401, 1992.

[34] B. G. BAUMGART. A polyhedron representation for computer vision. *AFIPS Natl. Comput. Conf.*, 44:589–596, 1975.

[35] R. BELLMAN. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[36] P. BERTOLAZZI, G. DI BATTISTA, and W. DIDIMO. Computing orthogonal drawings with the minimum number of bends. *IEEE Trans. Computers*, 49(8):826–840, 2000.

[37] T. BIEDL and G. KANT. A better heuristic for orthogonal graph drawings. *In Proc. 2nd Annu. European Sympo. Algorithms (ESA '94) - Lecture Notes in Computer Science*, 855:24–35, 1998.

[38] N. L. BIGGS, E. K. LLOYD, and R. WILSON. *Graph Theory 1736 - 1936*. Oxford University Press, England, 1976.

[39] C. BINUCCI, W. DIDIMO, G. LIOTTA, and M. NONATO. Orthogonal drawings of graphs with vertex and edge labels. *Computational Geometry-Theory and Applications*, 32(2):71–114, 2005.

[40] J. A. BONDY and U. S. R. MURTY. *Graph Theory with Applications*. Macmillan, London, first edition, 1976.

[41] K. S. BOOTH and G. S. LUEKER. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and Systems Sciences*, 13:335–379, 1976.

[42] J. M. BOYER and WENDY J. MYRVOLD. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Alg. and App.*, 3:241–273, 2004.

[43] U. BRANDES. Eager st-ordering. *Lecture Notes in Comp. Science - Proc. of 10th Annual European Symposium, Italy*, 2461(10):247–256, 2002.

[44] M. J. CARPANO. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Trans. Syst. Man Cybern., SMC-10*, 11:705–715, 1980.

[45] B. V. CHERKASSKY and A. V. GOLDBERG. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.

[46] T. CORMEN, C. LEISERSON, and R. RIVEST. *Introduction to Algorithms*. MIT Press - McGraw-Hill, USA, 2001.

[47] M. de BERG, M. van KREVELD, M. OVERMARS, and O. SCHWARZKOPF. *Computation Geometry: Algorithms and Applications*. Springer-Verlag, New York, first edition, 1998.

[48] H. de FRAYSSEIX and P. ROSENSTIEHL. A depth-first search characterization of planarity. *Annals of Discrete Mathematics*, pages 75–80, 1982.

[49] H. de FRAYSSEIX and P. ROSENSTIEHL. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5:127–135, 1985.

[50] G. DEMOUCRON, Y. MALGRANGE, and R. PERTUISET. Graphes planaires: reconnaissance et construction des représentations planaires topologiques. *Rev. Francaise Recherche Opérationelle*, 8:34–47, 1965.

[51] P. F. DIETZ and D. D. SLEATOR. Two algorithms for maintaining order in a list. *In Proc. 19th Annu. ACM Sympos. Theory Comput.*, pages 365–372, 1987.

[52] E. DIJKSTRA. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[53] P. EADES. Symmetry finding algorithms. *In G. T. Toussanint editor, Computational Morphology*, pages 41–51, 1988.

[54] A. EDEN and T. MENS. Measuring software flexibility. *IEE Software*, 153(3):113–126, 2006.

[55] M. EIGLSPERGER, U. FOBMEIER, and M. KAUFMANN. Orthogonal graph drawing with constraints. *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 3–11, 2000.

[56] M. EIGLSPERGER and M. KAUFMANN. Fast compaction for orthogonal drawings with vertices of prescribed size. *Lecture Notes in Computer Science*, 2265:124–138, 2002.

[57] S. EVEN. *Graph Algorithms.* Computer Science Press, Maryland, first edition, 1979.

[58] S. EVEN and R. E. TARJAN. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339–344, 1976.

[59] S. EVEN and R. E. TARJAN. Corrigendum: Computing an st-numbering. *Theoretical Computer Science*, 4(1):123, 1977.

[60] P. FLAJOLET, P. POBLETE, and A. VIOLA. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–415, December 2004.

[61] B. FLAMING. *Practical Algorithms in C++.* John Wiley and Sons, USA, first edition, 1995.

[62] L. R. FORD and D. R. FULKERSON. *Flows in Networks.* Princeton University Press, 1962.

[63] M. FORSTER, A. PICK, M. RAITNER, and C. BACHMAIER. The graph template library. http://infosun.fmi.uni-passau.de/GTL/index.html.

[64] L. R. FOULDS. *Graph Theory Applications.* Springer-Verlag, New York, first edition, 1992.

[65] E. GAMMA, R. HELM, R. JOHNSON, and JOHN VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, USA, first edition, 1995.

[66] R. GARCIA, J. JARVI, A. LUMSDAINE, J. SIEK, and J. WILLCOCK. A comparative study of language support for generic programming. *Proc. of 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 115–134, 2003.

[67] A. GARG and R. TAMASSIA. A new minimum cost flow algorithm with applications to graph drawing. *In S. C. North, Graph Drawing (Proc. GT '96), Lecture Notes Comput. Sci.*, 1997.

[68] A. GIBBONS. *Algorithmic Graph Theory.* Cambridge University Press, Cambridge U.K., first edition, 1985.

[69] A. J. GOLDSTEIN. An efficient and constructive algorithm for testing whether a graph can be embedded in the plane. *In Graph and Combinatorics Conf.*, 1963.

[70] R. GOULD. *Graph Theory.* Benjamin-Cummings Publishing, Inc., Menlo Park, California, first edition, 1988.

[71] D. GREGOR, J. JARVI, J. SIEK, B. STROUSTRUP, G. REIS, and L. LUMSDAINE. Concepts: First-class language support for generic programming in c++. *OOPSLA*, To Appear, October 2006, October 22-26.

[72] D. GREGOR, M. KULKARNI, A. LUMSDAINE, D. MUSSER, and S. SCHUPP. Generic programming and high-performance libraries. *International Journal of Parallel Programming*, 33(2), June 2005.

[73] J. GROSS and J. YELLEN. *Graph Theory and its Applications*. CRC Press, New York, first edition, 1999.

[74] L. J. GUIBAS and J. STOLFI. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[75] I. HERMAN, G. MELANCON, and M. S. MARSHALL. Graph visualization and navigation in information visualization: A survey. *IEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[76] J. HOPCROFT and R. E. TARJAN. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–569, October 1974.

[77] R. JAYAKUMAR, K. THULASIRAMAN, and M. N. S. SWAMY. An optimal algorithm for maximal planarization of nonplanar graphs. *IEEE Internat. Sympos. on Circuits and Systems*, pages 1237–1240, 1986.

[78] N. JOSUTTIS. *The C++ Standard Template Library*. Addison-Wesley, USA, first edition, 1999.

[79] M. JÜNGER and P. MUTZEL. Maximum planar subgraphs an nice emeddings: Practical layout tools. *Proc. GD '97*, 1353 of Lecture Notes in Computer Science:193–204, 1997.

[80] K. G. KAKOULIS and I. G. TOLLIS. A unified approach to automatic label placement. *International Journal of Computational Geometry and Applications*, 13(1):23–59, 2003.

[81] K. G. KAKOULIS and I. G. TOLLIS. Algorithms for the multiple label placement problem. *Computational Geometry-Theory and Applications*, 35(3):134–161, 2006.

[82] G. KANT. An $o(n^2)$ maximal planarization algorithm based on pq-trees. Technical report ruu-cs-92-03, Dept. Comput. Sci. Utrecht Univ., Netherlands, 1992.

[83] P. N. KLEIN, S. RAO, M. RAUCH, and S. SUBRAMANIAN. Faster shortest-path algorithms for planar graphs. *In Proc. ACM Symp. on Theory of Computing*, 1994.

[84] W. KOCAY, D. NEILSON, and R. SZYPOWSKI. Drawing graphs on the torus. *Ars Combinatoria*, 59(2):259–277, April 2001.

[85] C. KOSAK, J. MARKS, and S. SHIEBER. Automating the layout of network diagramas with specified visual organization. *IEEE Trans. Syst. Man Cybern., SMC-11*, 3:440–454, 1994.

[86] R. J. LIPTON, S. C. NORTH, and J. S. SANDBERG. A method for drawing graphs. *In Proc. 1st Annu. ACM Symposium Comput. Geom.*, pages 153–160, 1985.

[87] J. A. MCHUGH. *Algorithmic Graph Theory*. Prentice Hall, Inc., New Jersey, first edition, 1990.

[88] K. MEHLHORN. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, Germany, 1984.

[89] S. MEYER. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman, Inc, USA, first edition, 1996.

[90] S. MEYER. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Longman, Inc, USA, third edition, 2006.

[91] D. E. MULLER and F. P. PREPARATA. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[92] E. NARDELLI and M. TALAMO. A fast algorithm for planarization of sparse diagrams. Technical report r.105, IASI-CNR, Rome, 1984.

[93] A. PAPAKOSTAS and I. G. TOLLIS. Improved algorithms and bounds for orthogonal drawings. *In R. Tamassia and I. G. Tollis, editors, Graph Drawing (Proc. GD '94) - Lecture Notes in Computer Science*, 894:40–51, 1994.

[94] A. PAPAKOSTAS and I. G. TOLLIS. Algorithms for area-efficient orthogonal drawings. *Computation Geometry - Special Issue on Geometric Representations of Graphs*, 9(1-2):83–110, 1998.

[95] J. S. POULIN. Measuring software reusability. *Proc. of Third International Conference in Software Reuse: Advances in Software Reusability*, pages 126–138, 1994.

[96] J. S. POULIN. Measuring software quality: A case study. *IEEE Computer Society Press*, 29(11):78–87, 1996.

[97] H. C. PURCHASE, R. F. COHEN, and M. JAMES. Validating graph drawing aesthetics. *Graph Drawing (Proc. GD '95)*, Lecture Notes Comput. Sci. 1027:435–446, Springer-Verlag, 1996.

[98] M. RAHMAN, S. NAKANO, and T. NISHIZEKI. A linear algorithm for bend-optimal orthogonal drawings of triconnected cubic plane graphs. *Lecture Notes in Comp. Sci. - Graph Drawing (Proc. GD '97)*, 1353:99–110.

[99] M. RAHMAN and T. NISHIZEKI. Bend-minimum orthogonal drawings of plane 3-graphs. *Graph-Theoretic Concepts in Computer Science: 28th International Workshop, WG 2002*, 2573:367–378, 2002.

[100] R. C. READ. A new method for drawing a planar graph given the cyclic order of the edges at each vertex. *Congressus Numerantium*, 56:31–44, 1987.

[101] R. SEDGEWICK. *Algorithms in C++ - Fundamentals, Data Structures, Sorting and Searching.* Addison-Wesley, USA, third edition, 1998.

[102] R. SEDGEWICK. *Algorithms in C++ - Graph Algorithms.* Addison-Wesley, USA, third edition, 2002.

[103] R. W. SHIREY. *Implementation and Analysis of Efficient Graph Planarity Testing Algorithms.* PhD thesis, Univ. of Wisconsin, Madison, 1969.

[104] J. SIEK, L. Q. LEE, and A. LUMSDAINE. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley, USA, first edition, 2002.

[105] JANE M. SIX, K. G. KAKOULIS, and I. G. TOLLIS. Techniques for the refinement of orthogonal graph drawings. *Journal of Graph Algorithms and Applications*, 4(3):75–103, 2000.

[106] S. SKIENA. *The Algorithm Design Manual.* Springer-Verlag, New York, USA, 1998.

[107] J. SOUKUP. Circuit layout. *Proc. IEEE*, 69(10):197–213, 1972.

[108] B. STROUSTRUP. *The C++ Programming Language.* Addison-Wesley, New York, third edition, 2000.

[109] K. SUGIYAMA, S. TAGAWA, and M. TODA. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern., SMC-11*, 2:109–125, 1981.

[110] R. TAMASSIA. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal Comput.*, 3:421–444, 1987.

[111] R. TAMASSIA, G. DI BATISTA, and C. BATINI. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 1:61–79, 1988.

[112] R. TAMASSIA and I. TOLLIS. Planar grid embedding in linear time. *IEEE Trans. Circuits and Systems., CAS-36*, 9:1230–1234, 1989.

[113] G. VALIENTE. *Algorithms on Trees and Graphs.* Springer, USA, 2002.

[114] D. VANDEVOORDE and N. M. JOSUTTIS. *C++ Templates.* Addison-Wesley, USA, 2003.

[115] J. WARFIELD. Crossing theory and hierarchy mapping. *IEEE Trans. Syst. Man Cybern., SMC-7*, 7:502–523, 1977.

[116] D. WEST. *Introduction to Graph Theory.* Prentice Hall, third edition, 2006.

[117] P. WILSON, M. JOHNSTONE, M. NEELY, and D. BOLES. Dynamic storage allocation: A survey and critical review. *Proceedings of the Workshop on Memory Management (IWMM)*, pages 1–116, September 1995.

[118] R. WILSON. *Introduction to Graph Theory.* Longman, Edinburgh, Harlow, England, fourth edition, 1996.

Apêndice A

Este apêndice apresenta os conceitos de grafos existentes na GTAD, ilustrados na figura 4.4. O formato da documentação é baseado na STL [16], que define alguns conceitos como *Assignable* e *Default Constructible*, que apesar de serem utilizados neste apêndice, não são apresentados. O texto é escrito em inglês por ser parte da documentação oficial da GTAD, e para manter coerência com a terminologia introduzida pela STL.

Graph

Description

A Graph represents a combinatorial structure composed by vertices and edges.

Refinement of

Assignable and Default Constructible.

Notation

G A type that is a model of Graph.
g An object of type G.
e An object of type Graph_traits<G>::Edge_handle.

Associated types

Vertex handle	Graph_traits<G>::Vertex_handle	The type of a vertex in a graph. It must be Assignable, Default Constructible and Equality Comparable.
Edge handle	Graph_traits<G>::Edge_handle	The type of an edge in a graph. It must be Assignable, Default Constructible and Equality Comparable.

Valid expressions

Name	Expression	Type requirements	Return type
Null vertex	Graph_traits<G>::null_vertex()		Vertex_handle[1]
Null edge	Graph_traits<G>::null_edge()		Edge_handle[1]
Total vertices	total_vertices(g)		std::size_t
Total edges	total_edges(g)		std::size_t
Edge source	edge_source(e, g)		Vertex_handle[1]
Edge target	edge_target(e, g)		Vertex_handle[1]

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Null vertex	Graph_traits<G>::null_vertex()		Returns a Vertex handle that does not correspond to any vertex in the graph.	
Null edge	Graph_traits<G>::null_edge()		Returns an Edge handle that does not correspond to any edge in the graph.	
Total vertices	total_vertices(g)		Returns the number of vertices in g.	
Total edges	total_edges(g)		Returns the number of edges in g.	
Edge source	edge_source(e, g)		Returns the Vertex handle, which is the source of edge e.	
Edge target	edge_target(e, g)		Returns the Vertex handle, which is the target of edge e.	

Complexity guarantees

All operations are constant time.

Notes

[1] Vertex_handle and Edge_handle are just shortcut names to Graph_traits<G>::Vertex_handle and Graph_traits<G>::Edge_handle, respectively.

Vertex Traversable Graph

Description

A Vertex Traversable Graph is a Graph that provides iteration over its vertices.

Refinement of

Graph.

Notation

G A type that is a model of Vertex Traversable Graph.
g An object of type G.

Associated types

Vertex iterator	Graph_traits<G>::Vertex_iterator	The type of an iterator used to traverse the vertices of a graph. It must be a Forward Iterator.

Valid expressions

Name	Expression	Type requirements	Return type
Vertices	vertices(g)		std::pair<Vertex_iterator, Vertex_iterator>[1]

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Vertices	vertices(g)		Returns a pair of iterators to the vertices of the graph. The first iterator corresponds to any valid vertex. The second iterator is a past-the-end iterator.	

Complexity guarantees

All operations are constant time.

Notes

[1] Vertex_iterator is just a shortcut name to Graph_traits<G>::Vertex_iterator.

Edge Traversable Graph

Description

An Edge Traversable Graph is a Graph that provides iteration over its edges.

Refinement of

Graph.

Notation

G A type that is a model of Edge Traversable Graph.
g An object of type G.

Associated types

Edge iterator	Graph_traits<G>::Edge_iterator	The type of an iterator used to traverse the edges of a graph. It must be a Forward Iterator.

Valid expressions

Name	Expression	Type requirements	Return type
Edges	edges(g)		std::pair<Edge_iterator, Edge_iterator>[1]

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Edges	edges(g)		Returns a pair of iterators to the edges of the graph. The first iterator corresponds to any valid edge. The second iterator is a past-the-end iterator.	

Complexity guarantees

All operations are constant time.

Notes

[1] Edge_iterator is just a shortcut name to Graph_traits<G>::Edge_iterator.

Adjacency Graph

Description

An Adjacency Graph is a Graph that provides information about the out-edges and out-vertices of a vertex.

Refinement of

Graph.

Notation

G A type that is a model of Adjacency Graph.
g An object of type G.
v An object of type Graph_traits<G>::Vertex_handle.

Associated types

Adjacent out-vertex iterator	Graph_traits<G>::Adjacent_out_vertex_iterator	The type of an iterator used to traverse the out-vertices of a vertex. It must be a Forward Iterator.
Adjacent out-edge iterator	Graph_traits<G>::Adjacent_out_edge_iterator	The type of an iterator used to traverse the out-edges of a vertex. It must be a Forward Iterator.

Valid expressions

Name	**Expression**	**Type Requirements**	**Return type**
Adjacent out-vertices	adj_out_vertices(v, g)		std::pair<Adjacent_out_vertex_iterator, Adjacent_out_vertex_iterator>[1]
Adjacent out-edges	adj_out_edges(v, g)		std::pair<Adjacent_out_edge_iterator, Adjacent_out_edge_iterator>[1]
Out-degree	out_degree(v, g)		std::size_t

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Adjacent out-vertices	adj_out_vertices(v, g)	Vertex v must be valid.	Returns a pair of iterators to the adjacent out-vertices of vertex v. The first iterator corresponds to any valid vertex. The second iterator is a past-the-end iterator.	
Adjacent out-edges	adj_out_edges(v, g)	Vertex v must be valid.	Returns a pair of iterators to the adjacent out-edges of vertex v. The first iterator corresponds to any valid edge. The second iterator is a past-the-end iterator.	
Out-degree	out_degree(v, g)	Vertex v must be valid.	Returns the out-degree of vertex v.	

Complexity guarantees

All operations are constant time.

Notes

[1] Adjacent_out_vertex_iterator and Adjacent_out_edge_iterator are just shortcut names to Graph_traits<G>::Adjacent_out_vertex_iterator and Graph_traits<G>::Adjacent_out_edge_iterator, respectively.

Adjacency Directed Graph

Description

An Adjacency Directed Graph is a Graph that provides information about the in-edges and in-vertices of a vertex.

Refinement of

Adjacency Graph.

Notation

G A type that is a model of Adjacency Directed Graph.
g An object of type G.
v An object of type Graph_traits<G>::Vertex_handle.

Associated types

Adjacent in-vertex iterator	Graph_traits<G>::Adjacent_in_vertex_iterator	The type of an iterator used to traverse the in-vertices of a vertex. It must be a Forward Iterator.
Adjacent in-edge iterator	Graph_traits<G>::Adjacent_in_edge_iterator	The type of an iterator used to traverse the in-edges of a vertex. It must be a Forward Iterator.

Valid expressions

Name	**Expression**	**Type Requirements**	**Return type**
Adjacent in-vertices	adj_in_vertices(v, g)		std::pair<Adjacent_in_vertex_iterator, Adjacent_in_vertex_iterator>[1]
Adjacent in-edges	adj_in_edges(v, g)		std::pair<Adjacent_in_edge_iterator, Adjacent_in_edge_iterator>[1]
In-degree	in_degree(v, g)		std::size_t

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Adjacent in-vertices	adj_in_vertices(v, g)	Vertex v must be valid.	Returns a pair of iterators to the adjacent in-vertices of vertex v. The first iterator corresponds to any valid vertex. The second iterator is a past-the-end iterator.	
Adjacent in-edges	adj_in_edges(v, g)	Vertex v must be valid.	Returns a pair of iterators to the adjacent in-edges of vertex v. The first iterator corresponds to any valid edge. The second iterator is a past-the-end iterator.	
In-degree	in_degree(v, g)	Vertex v must be valid.	Returns the in-degree of the vertex v.	

Complexity guarantees

All operations are constant time.

Notes

[1] Adjacent_in_vertex_iterator and Adjacent_in_edge_iterator are just shortcut names to Graph_traits<G>::Adjacent_in_vertex_iterator and Graph_traits<G>::Adjacent_in_ edge_iterator, respectively.

Mutable Graph

Description

A Mutable Graph is a Graph that allows its internal combinatorial structure to be modified.

Refinement of

Graph.

Notation

G A type that is a model of Mutable Graph.
g An object of type G.
v, s, t Objects of type Graph_traits<G>::Vertex_handle.
e An object of type Graph_traits<G>::Edge_handle.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	**Expression**	**Type Requirements**	**Return type**
Insert vertex	insert_vertex(g)		Vertex_handle&[1]
Remove vertex	remove_vertex(v, g)		void
Insert edge	insert_edge(s, t, g)		Edge_handle&[1]
Remove edge	remove_edge(e, g)		void
Clear graph	clear_graph(g)		void

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Insert vertex	insert_vertex(g)		Inserts a new vertex into the graph and returns a reference to it.	Number of vertices in g is incremented by one.
Remove vertex	remove_vertex(v, g)		Removes vertex v and its adjacent edges from the graph. If the given vertex is invalid, nothing is done.	Number of vertices in g is decremented by one. Number of edges in g is decremented by the number of edges adjacent to v.
Insert edge	insert_edge(s, t, g)	Vertices s and t must be valid. If the graph does not accept self-loops, s must be different than t. If the graph does not accept parallel edges, the edge must not exist in the graph.	Inserts a new edge into the graph and returns a reference to it.	Number of edges in g is incremented by one.
Remove edge	remove_edge(e, g)		Removes edge e from the graph. If the given edge is invalid, nothing is done.	Number of edges in g is decremented by one.
Clear graph	clear_graph(g)		Removes all vertices and edges from the graph.	Number of edges and vertices in g is zero.

Complexity guarantees

Note: Consider V as the number of vertices in the graph and E as the number of edges in the graph.

Insert vertex and insert edge are constant time.

Remove vertex is dependent on the number of edges adjacent to the given vertex. For an average-case where vertices have E/V edges in their lists, this operation is $O(E/V)$ amortized.

Remove edge is $O(p)$ amortized, where p is the number of edges parallel to the given edge.

Clear graph is $O(V + E)$ amortized.

Notes

[1] Vertex_handle and Edge_handle are just shortcut names to Graph_traits<G>::Vertex_handle and Graph_traits<G>::Edge_handle, respectively.

Mutable Adjacency Graph

Description

A Mutable Adjacency Graph is a Mutable Graph that provides access to adjacency information.

Refinement of

Mutable Graph, Adjacency Graph.

Notation

G A type that is a model of Mutable Adjacency Graph.
g An object of type G.
v An object of type Graph_traits<G>::Vertex_handle.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	Expression	Type Requirements	Return type
Remove adjacent edges	remove_adj_edges(v, g)		void

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Remove adjacent edges	remove_adj_edges(v, g)		Removes all edges adjacent to vertex v.	Number of edges in g is decreased by the number of edges adjacent to v.

Complexity guarantees

Note: Consider V as the number of vertices in the graph and E as the number of edges in the graph.

Remove adjacent edges is dependent on the number of edges adjacent to the given vertex. For an average-case where vertices have E/V edges in their lists, this operation is $O(E/V)$ amortized.

Notes

Edge Findable Graph

Description

An Edge Findable Graph is a Graph that provides a way to determine whether or not a specific edge exists in the graph.

Refinement of

Graph.

Notation

G A type that is a model of Edge Findable Graph.
g An object of type G.
s, t Objects of type Graph_traits<G>::Vertex_handle.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	Expression	Type Requirements	Return type
Find edge	find_edge(s, t, g)		Edge_handle[1]

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Find edge	find_edge(s, t, g)		If the edge (s, t) exists in the graph, returns a copy of it. Otherwise, returns an invalid edge. Note that if the graph is undirected, the correct way to determine whether an edge exists in the graph is looking for edge (s, t) and edge (t, s).	

Complexity guarantees

Find edge is amortized constant time.

Notes

[1] Edge_handle is just a shortcut name to Graph_traits<G>::Edge_handle.

Edge Findable Mutable Graph

Description

An Edge Findable Mutable Graph is a Graph that provides mutable operations that are dependent on the ability of finding a specific edge.

Refinement of

Edge Findable Graph, Mutable Graph.

Notation

G A type that is a model of Edge Findable Mutable Graph.
g An object of type G.
s, t Objects of type Graph_traits<G>::Vertex_handle.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	Expression	Type Requirements	Return type
Remove edge by vertices	remove_edge(s, t, g)		void

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Remove edge by vertices	remove_edge(s, t, g)		Removes all (s, t) edges from the graph. If no (s, t) edge exists in the graph, nothing is done.	Number of edges in g is decreased by the number of existent (s, t) edges.

Complexity guarantees

Remove edge by vertices is amortized constant time for each (s, t) edge removed.

Notes

Cyclic List Graph

Description

A Cyclic List Graph is Graph that provides access to its adjacency list in a cyclic manner. Therefore, this concept is not applicable for graphs represented with adjacency matrixes.

Refinement of

Graph.

Notation

G A type that is a model of Cyclic List Graph.
g An object of type G.
v An objects of type Graph_traits<G>::Vertex_handle.
e An object of type Graph_traits<G>::Edge_handle.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	Expression	Type Requirements	Return type
Next edge	next_edge(e, v, g)		Edge_handle[1]
Previous edge	previous_edge(e, v, g)		Edge_handle[1]

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Next edge	next_edge(e, v, g)	Vertex v and edge e must be valid.	Returns the edge next to e in the adjacency list of v.	
Previous edge	previous_edge(e, v, g)	Vertex v and edge e must be valid.	Returns the edge previous to e in the adjacency list of v.	

Complexity guarantees

All operations are amortized constant time.

Notes

[1] Edge_handle is just a shortcut name to Graph_traits<G>::Edge_handle.

Sortable List Graph

Description

A Sortable List Graph is Graph that provides operations that modify the order of the edges in its adjacency list. Therefore, this concept is not applicable for graphs represented with adjacency matrixes.

Refinement of

Graph.

Notation

G A type that is a model of Sortable List Graph.
g An object of type G.
v An object of type Graph_traits<G>::Vertex_handle.
e An object of type Graph_traits<G>::Edge_handle.
i An int.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	Expression	Type Requirements	Return type
Switch edge	switch_edge(e, v, i, g)		void
Rotate list	rotate_list(v, i, g)		void

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Switch edge	switch_edge(e, v, i, g)	Vertex v and edge e must be valid.	Switches the positions of edges e and the one at the position specified by i, in the adjacency list of v.	Order of the edges in the adjacency list of v might change.
Rotate list	rotate_list(v, i, g)	Vertex v must be valid.	Rotates the adjacency list of v. If i is 1, it is rotate towards the end (the last edge will become the first). If i is -1, it is rotated towards the begin (the first edge will become the last).	Position of each edge in the adjacency list of v is incremented or decremented by one.

Complexity guarantees

All operations are amortized constant time.

Notes

[1] Edge_handle is just a shortcut name to Graph_traits<G>::Edge_handle.

Embedding Preservable Mutable Graph

Description

An Embedding Preservable Mutable Graph is a Mutable Graph the provides operations commonly used when working with planar embeddings. Therefore, this concept is not applicable for graphs represented with adjacency matrixes.

Refinement of

Mutable Graph.

Notation

G A type that is a model of Embedding Preservable Mutable Graph.
g An object of type G.
e An object of type Graph_traits<G>::Edge_handle.
source(e) The source Graph_traits<G>::Vertex_handle of e.
target(e) The target Graph_traits<G>::Vertex_handle of e.

Associated types

This concept does not introduce any new associated type.

Valid expressions

Name	**Expression**	**Type Requirements**	**Return type**
Split edge	split_edge(e, v, g)		std::pair<Edge_handle, Edge_handle>[1]
Insert edge after	insert_edge_after(s, t, e, g)		Edge_handle&[1]
Insert edge before	insert_edge_before(s, t, e, g)		Edge_handle&[1]

Expressions semantics

Name	Expression	Precondition	Semantics	Postcondition
Split edge	split_edge(e, v, g)	Vertex v and edge e must be valid.	Splits edge e in two edges (source(e), v) and (v, target(e)). The position of the new edges in the adjacency lists of source(e) and target(e) are required to be the same as the position of e.	Number of edges in g is incremented by one.
Insert edge before	insert_edge_before(s, t, e, g)	Vertices s and t and edge e must be valid.	For directed graphs, inserts edge (s, t) before edge e in the adjacency list of vertex s and returns a reference to it. This operation is not defined for undirected graphs.	Number of edges in g is incremented by one.
Insert edge after	insert_edge_after(s, t, e, g)	Vertices s and t and edge e must be valid.	For directed graphs, inserts edge (s, t) after edge e in the list of vertex s and returns a reference to it. This operation is not defined for undirected graphs.	Number of edges in g is incremented by one.

Complexity guarantees

All operations are amortized constant time.

Notes

[1] Edge_handle is just a shortcut name to Graph_traits<G>::Edge_handle.