Algoritmos para caminhos mínimos

Shigueo Isotani

Dissertação apresentada ao
Instituto de Matemática e
Estatística da
Universidade de São Paulo
como parte dos requisitos
para obtenção do grau de
Mestre em Ciência da Computação

Orientador: Prof. Dr. José Coelho de Pina Jr.

— São Paulo, março de 2002 —

– Durante o desenvolvimento deste trabalho, o aluno recebeu apoio financeiro da CAPES –

Algoritmos para caminhos mínimos

Este exemplar corresponde à redação final
da dissertação devidamente corrigida e
defendida por Shigueo Isotani
e aprovada pela comissão julgadora.

São Paulo, 21 de março de 2002

Banca examinadora:

- Profa. Dra. Cristina Gomes Fernandes - IME-USP
- Prof. Dr. José Coelho de Pina Jr - IME-USP
- Prof. Dr. Marcelo Henriques de Carvalho - DCT-UFMS

Aos meus pais Sadao e Naoko
e meus irmãos Seiji e Mina.

Agradecimentos

Meus sinceros agradecimentos ao professor *José Coelho*, pela forma dedicada e paciente com que me orientou. Esteve sempre presente e pronto a ajudar.

Agradeço ao *Rozante* e ao professor *José Augusto* pelas valiosas dicas em relação ao uso do CWEB, pois sem elas, ainda estaria tentando compilar esta dissertação.

Aos meus pais, *Sadao* e *Naoko*, pela demonstração de confiança e carinho.

Aos meus irmãos, *Seiji* e *Mina*, pelo incentivo e pela ajuda com os deveres de casa, durante o período da redação desta dissertação.

À minha namorada *Regina*.

À Ana Lúcia pela ajuda na preparação dos slides da apresentação.

À CAPES pelo apoio financeiro proporcionado durante o desenvolvimento desta dissertação.

Aos meus amigos *Exec*, *Inoue*, *Kubo*, *Luís Yano*, *Marcelo Vinagreiro*, *Noma* e *Wagner Dias* pela excelente companhia que me proporcionaram todos esses anos.

Resumo

O *problema do caminho mínimo* consiste em: *dados* um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$ e um vértice s *encontrar* um caminho de comprimento mínimo de s até t, para cada vértice t em V.

Desde 1959, quase todos os desenvolvimentos teóricos para esse problema têm se baseado no algoritmo de Dijkstra [11]. Foram desenvolvidas várias estruturas de dados que aumentam a eficiência desse algoritmo. Porém, qualquer implementação do mesmo, examina os vértices em ordem crescente de distância a partir do vértice inicial s. Portanto, ocorre uma ordenação implícita dos vértices de acordo com essas distâncias. Assim, no modelo de comparação-adição, qualquer implementação deste algoritmo consome tempo $\Omega(m + n \log n)$, onde n é o número de vértices e m é o número de arcos do grafo dado.

Para grafos simétricos e comprimentos em $\mathbb{Z}_{>}$, Thorup [39] projetou um algoritmo, no modelo RAM, que consome tempo e espaço $O(m + n)$. O algoritmo utiliza uma decomposição hierárquica do grafo e "bucketing" para identificar eficientemente conjuntos de vértices que podem ser examinados em qualquer ordem, evitando assim, o "gargalo" da ordenação.

Nesta dissertação são descritos e implementados vários algoritmos para o problema do caminho mínimo, inclusive os mencionados acima. Ao final, é feita uma análise experimental das implementações realizadas.

Abstract

The *single source shortest path problem* consists of: *given* a graph (V, A), a weight function c from A to $\mathbb{Z}_{\geq}$ and a source vertex s *find* a shortest path from s to t, for each vertex t in V.

Since 1959, almost all theoretical developments on this problem have been based on Dijkstra's algorithm [11]. Several data structures were developed to speedup its running time. However, any implementation of Dijkstra's algorithm scans the vertices in increasing order of distance from s. Thus, the vertices are implicitly sorted according to their distances from s. Therefore, in the comparation-addition model, any implementation of this algorithm has running time $\Omega(m + n \log n)$, where n is the number of vertices and m is the number of arcs of the given graph.

For undirected graphs and edge lengths in $\mathbb{Z}_{>}$, Thorup [39] proposed an algorithm, on the RAM model, that has running time $O(m+n)$. The algorithm avoids the sorting bottleneck by building a hierarchical bucketing structure identifying vertices that may be scanned in any order.

This dissertation describes several algorithms for the single source shortest path problem, including the ones mentioned above.

Índice

Agradecimentos i

Resumo ii

Lista de Figuras **vii**

Introdução **1**

1 Preliminares **7**

1.1 Notação básica 7

1.2 Teoria dos grafos 7

1.3 Modelo de computação 11

1.4 Filas de prioridade 12

1.5 Linguagem algorítmica e invariantes 13

1.6 Programação literária e CWEB 15

1.7 Stanford Graph Base 17

1.8 Estrutura da implementação 22

2 Problema do caminho mínimo **23**

2.1 Descrição 23

2.2 Funções potencial e critério de otimalidade 24

2.3 Funções predecessor e representação de caminhos 25

2.4 Examinando arcos e vértices 26

2.5 Complexidade 26

3 **Algoritmo de Dijkstra** **29**

3.1 Descrição . . . 29

3.2 Invariantes . . . 30

3.3 Correção . . . 35

3.4 Eficiência . . . 36

3.5 Complexidade . . . 37

3.6 Versão em CWEB . . . 38

4 **Implementações de Filas de Prioridade** **43**

4.1 Heap . . . 44

4.2 D-heap . . . 47

4.3 Fibonacci heap . . . 51

4.4 Bucket heap . . . 59

4.5 Radix heap . . . 62

4.6 Eficiência . . . 67

5 **Algoritmo de Dinitz-Thorup** **69**

5.1 Partições, variante do PCM e elementos maduros . . . 70

5.2 Descrição . . . 71

5.3 Invariantes . . . 73

5.4 Correção . . . 76

5.5 Eficiência . . . 76

5.6 Versão em CWEB . . . 77

6 **Algoritmo de Thorup** **85**

6.1 Família laminar e representação arbórea . . . 85

6.2 Decomposição hierárquica . . . 86

6.3 Descrição . . . 87

6.4 Invariantes . . . 96

6.5 Correção . . . 98

6.6 Eficiência . . . 98

6.7 Construção da decomposição hierárquica . . . 105

6.8 Versão em CWEB 110

7 Resultados Experimentais 123

7.1 Ambiente experimental 124

7.2 SPRAND 125

7.3 SPGRID 130

7.4 SPBAD 136

8 Conclusões 139

A Implementação 143

A.1 Testa condição de otimalidade 153

Referências Bibliográficas 157

Índice Remissivo 161

Índice Remissivo para o Código 165

Lista de Figuras

1 Histórico envolvendo o problema do caminho mínimo . . . 3

1.1 Exemplos de grafos . . . 8
1.2 Matriz de adjacência de um grafo . . . 10
1.3 Matriz de incidência de um grafo . . . 11
1.4 Lista de adjacência de um grafo . . . 11
1.5 Estruturas do SGB . . . 20
1.6 Representação de um grafo no SGB . . . 21

2.1 Representação de caminhos através da função predecessor . . . 25

3.1 Simulação do algoritmo de Dijkstra . . . 31
3.2 Invariantes do algoritmo de Dijsktra . . . 32
3.3 Algoritmo de Dijsktra ordenando uma seqüência . . . 38

4.1 Heap . . . 44
4.2 D-heap . . . 48
4.3 Fibonacci heap . . . 52
4.4 Buckets . . . 60
4.5 Eficiência das implementações de uma fila de prioridade . . . 67

5.1 δ-partição . . . 70
5.2 Simulação do algoritmo de Dinitz-Thorup . . . 72
5.3 Invariantes do algoritmo de Dinitz-Thorup . . . 74

5.4 Representação de uma δ-partição na implementação 77

6.1 Família laminar e representação arbórea 86

6.2 Decomposição hierárquica (condição (h2)) 87

6.3 Exemplos de decomposições hierárquicas 88

6.4 Filhos maduros 88

6.5 Simulação do algoritmo de Thorup (1) 91

6.6 Simulação do algoritmo de Thorup (2) 92

6.7 Simulação do algoritmo de Thorup (3) 93

6.8 Simulação do algoritmo de Thorup (4) 94

6.9 Simulação do algoritmo de Thorup (5) 95

6.10 Simulação da implementação do algoritmo de Thorup 103

6.11 Simulação da implementação do algoritmo de Thorup (continuação) 104

6.12 Representação de um número em ponto flutuante 107

6.13 Construção de uma decomposição hierárquica 109

6.14 União de dois elementos na construção da decomposição hierárquica 114

7.1 Número de vértices em relação ao tempo em grafos esparsos gerados por SPRAND . . 126

7.2 Memória utilizada quando executado em grafos esparsos gerados por SPRAND 127

7.3 Valor de C em relação ao tempo em grafos esparsos gerados por SPRAND 127

7.4 Número de vértices em relação ao tempo em grafos densos gerados por SPRAND . . . 128

7.5 Memória utilizada quando executado em grafos densos gerados por SPRAND 129

7.6 Valor de C em relação ao tempo em grafos densos gerados por SPRAND 129

7.7 Exemplo de um grafo gerado por SPGRID 130

7.8 Número de vértices em relação ao tempo em grafos grade gerados por SPGRID com $X = 16$ 131

7.9 Memória utilizada quando executado em grafos grade gerados por SPGRID com $X = 16$ . . 132

7.10 Valor de C em relação ao tempo em grafos grade gerados por SPGRID com $X = 16$ 132

7.11 Número de vértices em relação ao tempo em grafos grade gerados por SPGRID com $Y = 16$ 134

7.12 Memória utilizada quando executado em grafos grade gerados por SPGRID com $Y = 16$ . . . 135

7.13 Valor de C em relação ao tempo em grafos grade gerados por SPGRID com $Y = 16$ 135

7.14 Exemplo de um grafo gerado por SPBAD 136

7.15 Número de vértices em relação ao tempo em grafos gerados por SPBAD 137
7.16 Memória utilizada quando executado em grafos gerados por SPBAD 137

8.1 Número de chamadas em relação ao tempo 141

Introdução

"Highways, telephone lines, electric power systems, computer chips, water delivery systems, and rail lines: these physical networks, and many others, are familiar to all of us. In each of these problem settings, we often wish to send some good(s) (vehicles, messages, electricity, or water) from one point to another, typically as efficiently as possible — that is, along a shortest route or via some minimum cost flow pattern."

Ahuja, Magnati, Orlin, and Reddy [2]

O *problema do caminho mínimo* (PCM) consiste em: *dados* um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$ e um vértice s *encontrar* um caminho de comprimento mínimo de s até t, para cada vértice t em V. Este problema é um dos mais comumente encontrados no estudo de problemas de redes de transporte e comunicação. Um rápido passar de olhos pelo artigo "Applications of Network Optimization" de Ahuja, Magnati, Orlin e Reddy [2] é suficiente para convencer alguém sobre o enorme espectro de aplicações de métodos para o problema. Por exemplo, esses métodos podem ser usados para reduzir tempo de vôo, baixar custos de serviços de transporte, diminuir o consumo de energia e ainda, podem ser utilizados para acelerar a distribuição de informações (pacotes) através da rede mundial, a Internet [23, 15]. O PCM é também um dos problemas mais elementares e possivelmente um dos mais fundamentais em otimização de redes. Muitos problemas em otimização combinatória e fluxos em redes usam, como subrotina, algoritmos para encontrar caminhos mínimos [1].

O problema do caminho mínimo têm sido amplamente estudado por um número vasto de pesquisadores. Estudos teóricos a respeito, podem ser encontrados em vários trabalhos [11, 39, 16, 3, 7, 18, 36], bem como estudos experimentais [6, 32, 21].

Desde 1959, quase todos os desenvolvimentos teóricos para esse problema têm se baseado no algoritmo de Dijkstra [11]. Foram aplicadas várias estruturas de dados, como heap [9] e fibonacci heap [16], para aumentar a eficiência desse algoritmo. Porém, qualquer implementação do mesmo examina os vértices em ordem crescente de distância a partir do vértice inicial s, ocorrendo assim, uma ordenação implícita dos vértices de acordo com essas distâncias. Desta forma, no

modelo de comparação-adição, qualquer implementação do algoritmo de Dijkstra consome tempo $\Omega(m+n\log n)$, onde n é o número de vértices e m é o número de arcos do grafo dado. Fredman e Tarjan, utilizando fibonacci heaps, obtiveram uma implementação do algoritmo de Dijkstra que consome tempo $O(m+n\log n)$.

O avanço tecnológico dos computadores tornou viável desenvolver algoritmos que utilizam operações "mais complexas", como endereçamento de memória, shifts, comparações lógicas, sem com isto, prejudicar o consumo de tempo do algoritmo, pois estas agora passam a ser consideradas elementares, e são realizadas em tempo constante. Tais operações fazem parte do chamado modelo RAM. Curiosamente, as comparações lógicas, shifts, que parecem não estar relacionadas com o problema de encontrar caminhos mínimos, proporcionam melhorias assintóticas significativas, como observado por Zwick [43].

Recentemente, um grande número de algoritmos para problemas fundamentais como ordenação, filas de prioridade e caminhos mínimos têm sido desenvolvidos adotando o modelo RAM [4, 40, 17, 5, 39, 31]. Estes algoritmos exibem uma melhor eficiência teórica, em relação aos algoritmos já conhecidos para esses problemas.

Apesar de todo o esforço, problemas básicos relacionados ao PCM ainda aguardam por uma resposta definitiva, por exemplo, a existência ou não de um algoritmo linear para o PCM no modelo RAM continua um problema desafiador. Entretanto, um passo importante foi dado no sentido de resolver esta questão. Para grafos simétricos e comprimentos em $\mathbb{Z}_{>}$, Thorup [39] projetou um algoritmo que consome tempo e espaço $O(m+n)$. O algoritmo utiliza uma decomposição hierárquica do grafo e "bucketing" para identificar eficientemente conjuntos de vértices que podem ser examinados em qualquer ordem, evitando assim, o "gargalo" da ordenação.

Nesta dissertação são descritos e implementados vários algoritmos para o problema do caminho mínimo, inclusive os mencionados acima. Também é apresentada uma análise experimental das implementações.

Organização da dissertação

O primeiro capítulo contém a maior parte das notações, conceitos e definições que são usados ao longo desta dissertação. Em seguida, no capítulo 2, é apresentado o problema do caminho mínimo junto com os ingredientes básicos que o envolvem, como, por exemplo, o certificado de otimalidade. No capítulo 3 é descrito o celebrado algoritmo de Dijkstra e seus invariantes. A sua eficiência é analisada e uma possível implementação é apresentada. O capítulo 4 mostra implementações das estruturas de dados *heap*, D-*heap*, *fibonacci heap*, *bucket heap* e *radix heap*. Cada uma destas estruturas dá origem a uma implementação diferente do algoritmo de Dijkstra. No capítulo 5 é descrito o algoritmo de Dinitz-Thorup, bem como seus invariantes e uma possível

implementação. O algoritmo de Dinitz-Thorup é um passo intermediário entre o algoritmo de Dijkstra e o algoritmo de Thorup, que é mostrado no capítulo 6, junto com seus invariantes, sua análise de eficiência e uma possível implementação. Uma análise experimental das implementações é feita no capítulo 7. Finalmente, no capítulo 8, relatamos as nossas conclusões, frustrações e possíveis trabalhos futuros.

Breve cronologia

A figura 1 traz um pouco do panorama histórico sobre os algoritmos que foram desenvolvidos para o problema do caminho mínimo. Algums deles dependem do maior comprimento de um arco, que é representado por C, alguns resolvem versões mais restritas do problema e outros menos. Na tabela, n é o número de vértices e m é o número de arcos do grafo dado e r é a razão entre o maior e o menor (não-nulo) comprimento de arcos.

Ano	Algoritmo	Consumo de tempo
1959	Dijkstra [11]	$O(m+n^2)$
1969	Dijkstra/Dial [10] (buckets)	$O(m+nC)$
1976	Dijkstra/Wagner [42]	$O(m+nC)$
1977	Dijkstra/Johnson [24] (heap)	$O(m\log n)$
1977	Van Emde Boas [41]	$O(m\log\log C)$
1987	Dijkstra/Fredman e Tarjan [16] (fibonacci heap)	$O(m+n\log n)$
1990	Dijkstra/Ahuja *et al.* [3] (radix heap)	$O(m+n\log(nC))$
1993	Dijkstra/Fredman e Willard [17] (fusion trees)	$O(m\sqrt{\log n})$
1994	Dijkstra/Fredman e Willard [18] (atomic heap)	$O(m+n\log n/\log\log n)$
1996	Dijkstra/Thorup [40] (RAM priority queue)	$O(m\log\log n)$
1997	Raman [35]	$O(m+n(\log C)^{1/4+\varepsilon})$
1999	Thorup [39] (bucketing hierárquico)	$O(m+n)$
2000	Hagerup [22]	$O(n+m\log\omega)$
2002	Pettie e Ramachandran [31]	$O(m\alpha(m,n)+n\log\log r)$

Figura 1: Histórico envolvendo o problema do caminho mínimo.

Como executar esta dissertação

Esta dissertação é um documento CWEB. Os arquivos que compõe essa dissertação podem ser obtidos no endereço `http://www.ime.usp.br/dcc/posgrad/teses/shigueo/` na forma compactada, com o nome dissertacao_cweb.tgz . Para descompactar, utilize o comando

```
meu_prompt> tar -xzvf dissertacao_cweb.tgz
```

O comando irá produzir os arquivos:

1) Capítulos: 01-conceitos.w, 02-problema.w, 03-dijkstra.w, 04-heap.w, 05-dinitz.w, 06-thorup.w, 07-resultados.w, 08-conclusoes.w e ap-implementacoes.w.
2) Complementos: Makefile, capa.tex, agradecimentos.tex, resumo.tex, e introd.tex.
3) Figuras e tabelas: nos diretórios fig/ e graph/.
4) Filtro e geradores (DIMACS): nos diretórios dimacs/ e geradores/
5) Estilos: sty/backref.sty e sty/mythesis.sty.
6) Referências: bib/joseplain.bst e bib/refs.bib.

A dissertação também está disponível em formato postscript, no arquivo "mestrado.ps".

O pacote CWEB consiste, basicamente, de dois programas *cweave* e *ctangle*. O download pode ser feito de `http://www-cs-staff.Stanford.EDU/~knuth/cweb.html`.

Como criar um arquivo postscript da dissertação.

Existem duas maneiras de se criar o arquivo: usando o Makefile ou manualmente.

Escolhendo usar o Makefile, apenas digite na linha de comando

```
meu_prompt> make
```

O arquivo postscript criado é "mestrado.ps".

Caso queira proceder manualmente, siga os seguintes passos:

```
meu_prompt> cweave mestrado.w  (gera o arquivo mestrado.tex)
meu_prompt> latex mestrado.tex (gera o arquivo mestrado.dvi)
meu_prompt> dvips mestrado.dvi -o
```

Como criar o executável das implementações.

Supõe-se que o sistema operacional é do tipo UNIX-like. Os testes foram feitos em um Linux RedHat 7.1 e nas estações Unix/Solaris do IME/USP.

Como as implementações utilizam a plataforma SGB, é necessário a inclusão da *library* libgb.a. Essa *library* pode ser obtida, já pré-compilada, no mesmo endereço do arquivo da dissertação, de duas formas: para Linux (sgb_linux.tgz) e para Unix (sgb_unix.tgz). Descompacte o arquivo escolhido, no mesmo diretório dos arquivos da dissertação. Caso esteja utilizando um Unix, será necessário modificar, no arquivo ap-implementacoes.w, a opção LINUX, em *enum*, para LINUX = 0.

O executável pode ser criado de duas maneiras: usando o Makefile ou manualmente.

Escolhendo usar o Makefile, apenas digite na linha de comando

```
meu_prompt> make programa
```

O executável criado será "programa".

Caso queira proceder manualmente, siga os seguintes passos:

```
meu_prompt> ctangle mestrado.w  (gera o arquivo mestrado.c)
meu_prompt> gcc -I./sgb/include -I./dimacs -L./sgb/lib
            -o programa mestrado.c -lgb -lm
```

Como executar o programa.

Digitando na linha de comando:

```
meu_prompt> programa
```

É gerado um grafo aleatório com 1000 vértices e 100000 arcos. Os comprimentos padrão dos arcos são números inteiros em [1..1000]. É possível alterar esses valores passando parâmetros na linha de comando, por exemplo:

```
meu_prompt> programa -n512 -m4096 -cmin5 -cmax100
```

Esse comando gera um grafo aleatório com 512 vértices, 8192 arcos (ou seja, 4096 arestas) e com os comprimentos em [5..100]. Para obter informações sobre os parâmetros da linha de comando, pode-se utilizar:

```
meu_prompt> programa -h
```

ou

```
meu_prompt> programa --hh
```

Este último fornece informações mais detalhadas.

Caso deseje utilizar os geradores do DIMACS, será preciso compilá-los. Para isso digite na linha de comando:

```
meu_prompt> cd geradores
meu_prompt> make
```

Agora será necessário modificar, no arquivo ap-implementacoes.w, a opção DIMACS, em *enum*, para DIMACS = 1 e compilar novamente a dissertação.

Exemplos de uso:

```
meu_prompt> ./geradores/bin/sprand 8192 16384 561 | programa
meu_prompt> ./geradores/bin/spgrid 16 512 281 |  programa
```

Capítulo 1

Preliminares

Neste capítulo apresentaremos a maior parte das notações e definições que serão usadas intensivamente ao longo desta dissertação.

A maior parte das definições e notações encontradas nestas preliminares seguem de perto as de Feofiloff [13].

1.1 Notação básica

O conjunto dos números inteiros será denotado por $\mathbb{Z}$. O conjunto dos números inteiros não-negativos será $\mathbb{Z}_{\geq}$ e positivos $\mathbb{Z}_{>}$.

$\mathbb{Z}$

$\mathbb{Z}_{\geq}$

$\mathbb{Z}_{>}$

É escrito S como uma **parte** de um conjunto V significando que S é um subconjunto de V.

Uma **lista** é uma seqüência $\langle v_1, v_2, \ldots, v_k \rangle$ de itens. O item v_1 é o primeiro da lista e o item v_k é o último. Uma **pilha** é uma lista que só aceita remoções do último item e inserções após o último item. A ação de remover um item de uma pilha será chamada de **desempilhar** e a ação de inserir um novo item será chamada de **empilhar**. Para pilhas, dizemos que v_k é o item no topo da pilha.

Um **intervalo** $[j..k]$ é uma seqüência de inteiros $j, j+1, \ldots, k$. Se i é um número em $[j..k]$, então i é um número inteiro tal que $j \leq i \leq k$.

$[j..k]$

1.2 Teoria dos grafos

Esta seção introduz os conceitos de grafos, grafos simétricos, passeios, ciclos, arborescências e outros elementos básicos da teoria dos grafos. Também são discutidas as diferentes maneiras de representarmos um grafo no computador.

Grafos e grafos simétricos

Um **grafo** é um objeto da forma (V, A), onde V é um conjunto finito e A é um conjunto de pares ordenados de elementos de V.

Os elementos de V são chamados **vértices** e os elementos de A são chamados **arcos**. Para cada arco (u, v), os vértices u e v representam a ponta inicial e a ponta final de (u, v), respectivamente. Um arco (u, v) também poderá ser denotado por uv.

Um grafo é **simétrico** se para cada arco uv existe também o arco vu. Diremos às vezes que o arco vu é **reverso** do arco uv e que o par $\{(u, v), (v, u)\}$ é uma **aresta**.

Um grafo pode ser naturalmente representado através de um diagrama, como o da figura 1.1, onde os vértices são pequenas bolas e os arcos são as flechas ligando estas bolas.

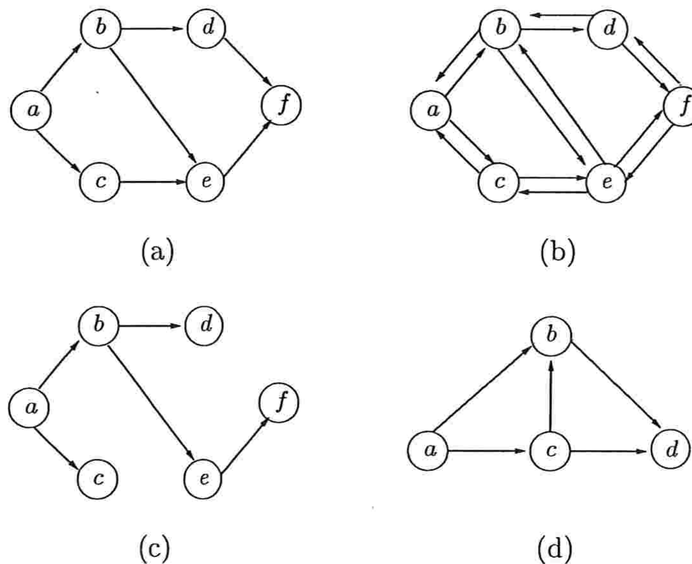


Figura 1.1: Em (a), (b), (c) e (d) são mostrados exemplos de grafos. Na figura (b) é ilustrado um grafo simétrico e em (c) uma arborescência.

n
m

Denotaremos, quando não houver ambigüidade, por n e m os números $|V|$ e $|A|$, respectivamente. O **tamanho do grafo** é o número $m + n$.

Cortes e conjuntos induzidos

$A(S, Q)$
(S, Q)

Seja (V, A) um grafo e S e Q partes[1] de V. Será denotado por $A(S, Q)$, ou simplesmente (S, Q), o conjunto dos arcos com ponta inicial em S e ponta final em Q. Quando Q for o conjunto

[1] Comumente será escrito *parte* significando *subconjunto*.

$V \setminus S$ será usada a abreviação $A(S)$ significando $A(S,Q)$. Por $A[S]$ entenda-se o conjunto dos arcos com ambas as pontas em S.

$A[S]$

Para qualquer parte S de V, o **corte determinado por** S é o conjunto $A(S)$ e o **conjunto de arcos induzidos por** S é o conjunto $A[S]$.

Passeios, caminhos e ciclos

Um **passeio** num grafo (V, A) é qualquer seqüência da forma

$$\langle v_0, \alpha_1, v_1, \ldots, \alpha_k, v_k \rangle \tag{1.1}$$

onde $v_0, \ldots, v_k$ são vértices, $\alpha_1, \ldots, \alpha_k$ são arcos e, para cada i, α_i é o arco $v_{i-1}v_i$. O vértice v_0 é o **início** do passeio e o v_k é seu **término**. Um **passeio não-orientado** é uma seqüência como (1.1) onde, para cada i, α_i é o arco $v_{i-1}v_i$ ou o arco $v_i v_{i-1}$.

Na figura 1.1(a) a seqüência $\langle a, ab, b, be, e, ef, f \rangle$ é um passeio com início em a e término em f e a seqüência $\langle a, ac, c, ce, e, be, b, bd, d, df, f \rangle$ é um passeio não-orientado com início em a e término em f.

Um **ciclo** é um passeio onde o início e término coincidem. Um **ciclo não-orientado** é um passeio não-orientado onde o início e término coincidem. Na figura 1.1(b) a seqüência $\langle a, ab, b, be, e, ec, c, ca, a \rangle$ é um ciclo com início e término em a. Em 1.1(a) a seqüência $\langle a, ab, b, be, e, ce, c, ac, a \rangle$ é um ciclo não-orientado com início e término em a.

Um **caminho** é um passeio sem vértices repetidos. Um **caminho não-orientado** é um passeio não-orientado sem vértices repetidos. Na figura 1.1(a) a seqüência $\langle a, ab, b, be, e, ef, f \rangle$ é um caminho com início em a e término em f e a seqüência $\langle a, ac, c, ce, e, be, b, bd, d, df, f \rangle$ é um caminho não-orientado com início em a e término em f.

Um vértice t é **acessível** a partir de um vértice s se existe um caminho de s a t. O **território** de um vértice s é o conjunto de todos os vértices acessíveis a partir de s. Se S é o território de um vértice, então não existe arco que saia de S, ou seja, $A(S) = \emptyset$.

Um grafo (V, A) é (fortemente) **conexo** se para todo par (u, v) de vértices, u é acessível a partir de v e v é acessível a partir de u.

Grafos acíclicos, arborescências e árvores geradoras

Um grafo que não possui ciclos é dito **acíclico**. Um grafo simétrico é acíclico se não possui ciclos com pelo menos três arcos.

Um grafo acíclico (V, A) com $|V| = |A| + 1$ é uma **arborescência** se todo vértice, exceto um vértice especial chamado de **raiz**, é ponta final de exatamente um arco. Uma arborescência está ilustrada na figura 1.1(c). A raiz dessa arborescência é o vértice a. Se uv é um arco de uma arborescência, então u é o **pai** de v e v é o **filho** de u. Uma **folha** de uma arborescência é um vértice que não é ponta inicial de algum arco.

Um grafo simétrico acíclico (V, A') com $|V| - 1$ arestas, é uma **árvore geradora** de um grafo simétrico (V, A) se $A' \subseteq A$.

Representação de grafos no computador

Existem pelo menos três maneiras populares de representarmos um grafo no computador, são elas: (1) matriz de adjacência; (2) matriz de incidência; e (3) listas de adjacência. Do ponto de vista desta dissertação, listas de adjacência é a representação mais importante.

Matriz de adjacência. Uma matriz de adjacência de um grafo (V, A) é uma matriz com valores em $\{0, 1\}$, e indexada por $V \times V$, onde cada entrada (u, v) da matriz tem valor 1 se existe no grafo um arco de u a v, e 0 caso contrário. Para grafos simétricos a matriz de adjacências é simétrica. O espaço gasto com esta representação é proporcional a n^2, onde n é o número de vértices do grafo. Uma matriz de adjacência é mostrada na figura 1.2.

	a	b	c	d
a	0	1	1	0
b	0	0	0	1
c	0	1	0	1
d	0	0	0	0

Figura 1.2: Matriz de adjacência do grafo da figura 1.1(d).

Matriz de incidência. Uma matriz de incidência de um grafo (V, A) é uma matriz com valores em $\{-1, 0, +1\}$ e indexada por $V \times A$, onde cada entrada (u, a) é -1 se u é ponta inicial de a, $+1$ se u é ponta final de a, e 0 caso contrário. O espaço gasto com esta representação é proporcional a nm, onde n é o número de vértices e m é o número de arcos do grafo. Uma matriz de incidência da figura 1.1(d) pode ser vista em 1.3.

Listas de adjacência. Na representação de um grafo (V, A) através de listas de adjacência tem-se, para cada vértice u, uma lista dos arcos deixando u. Desta forma, para cada vértice u, o conjunto $A(u)$ é representado por uma lista. O espaço gasto com esta representação é proporcional a $n + m$, onde n é o número de vértices e m é o número de arcos do grafo. Uma lista de adjacência está ilustrada na figura 1.4.

	ab	ac	cb	cd	bd
a	-1	-1	0	0	0
b	$+1$	0	$+1$	0	-1
c	0	$+1$	-1	-1	0
d	0	0	0	$+1$	$+1$

Figura 1.3: Matriz de incidência do grafo da figura 1.1(d).

$A(a)$: ab, ac
$A(b)$: bd
$A(c)$: cb, cd
$A(d)$:

Figura 1.4: Listas de adjacência do grafo da figura 1.1(d).

1.3 Modelo de computação

Um **modelo de computação** é uma descrição abstrata e conceitual (não necessariamente realista) de um computador que será usado para executar um algoritmo. Um modelo de computação especifica as operações elementares que um algoritmo pode executar e o critério empregado para medir a quantidade de tempo que cada operação consome. Exemplo de operações elementares típicas são operações aritméticas entre números e comparações. A escolha de um modelo de computação envolve um compromisso entre realidade e tratabilidade matemática. O modelo escolhido deve capturar as características do dispositivo computacional e ainda deve ser suficientemente simples para que permita uma estimativa do número de operações dos algoritmos escritos para o modelo.

Existem muitos modelos de computação que diferem em seu poder computacional (isto é, alguns modelos podem realizar computações impossíveis para outros) e no consumo de tempo de várias operações. Nesta dissertação estamos interessados em dois modelos de computação: modelo de comparação-adição e modelo Random Access Machine.

O modelo de **comparação-adição-subtração** é mais conhecido como modelo de **comparação-adição**, já que a subtração pode ser simulada através de adições [31]. Este modelo consiste, entre outras coisas, de m números inteiros ou reais, inicialmente armazenados em variáveis $v_1, \ldots, v_m$. Cada variável v_i, com i em $[1..m]$, pode guardar um número inteiro ou real e somente pode ser manipulada por comparações, da forma "$v_i < v_j$" e adições, da forma "$v_i := v_j + v_k$". No modelo de **comparação**, a única operação de interesse é a comparação.

Em um grande número de algoritmos para o problema do caminho mínimo, é comum utilizar o modelo de comparação-adição. Enquanto esse modelo é mais elegante, por ser mais generalista, os

computadores reais possuem outras operações que gastam tempo constante além das comparações e adições, motivando o interesse pelo modelo RAM.

RAM

O modelo ω **Random Access Machine (RAM)**, ou simplesmente modelo RAM, supõe que cada palavra de memória do computador tem ω bits, capaz de manter um inteiro em $[-2^{\omega-1}..2^{\omega-1}-1]$ e que toda distância num grafo cabe em uma palavra. Neste modelo, as operações realizadas em tempo constantes são: *comparação*, *adição*, *subtração*, *bitwise lógico*, *shift* arbitrário dos bits e *multiplicação*. Além disso, supõe-se que o número de palavras de memória do computador, que podem ser endereçadas em tempo constante, é 2^{ω}. Isto significa que é exigido que ω seja suficientemente grande para que a mémoria comporte os dados do problema.

Os projetos de algoritmos neste modelo são de grande interesse, pois oferecem melhorias assintóticas significativas e fazem sentido do ponto de vista prático. Porém, ainda não se espera que eles sejam mais rápidos quando executados nessa corrente geração de CPU's, que normalmente admitem operações em dados de 32-bits, e no máximo de 64-bits. Os novos algoritmos necessitam que o tamanho de ω seja grande, como analisado por Rahman e Raman [34]. Por outro lado, segundo eles, acredita-se que os novos hardwares em desenvolvimento devem transformar esses avanços teóricos em práticos.

Dos algoritmos que serão apresentados, o de Dijkstra trabalha sobre o modelo de comparação-adição e o de Dinitz-Thorup e o de Thorup trabalham sobre o modelo RAM.

A palavra **complexidade** é utilizada nesta dissertação como sinônimo de "recurso computacional necessário". Assim, por **complexidade de um problema** entenda-se um número de operações elementares necessárias para resolver o problema em um certo modelo. Esta é uma medida intrínseca da dificuldade para resolver o problema. Já por **complexidade de um algoritmo** entenda-se um número de operações elementares realizadas, no pior caso, por qualquer implementação do algoritmo em um certo modelo.

1.4 Filas de prioridade

Sempre que dados são representados em um computador, os seguintes aspectos são considerados:

(1) a maneira que essas informações (ou objetos do mundo real) são modelados como objetos matemáticos;

(2) o conjunto de operações definidas sobre estes objetos matemáticos;

(3) a maneira na qual estes objetos serão armazenados (representados) na memória de um computador;

(4) os algoritmos que são usados para executar as operações sobre os objetos com a representação escolhida.

Para entender melhor esses aspectos é preciso entender a diferença entre os seguintes termos: tipo de dados, tipo abstrato de dados e estrutura de dados.

O **tipo de dado** de uma variável é o conjunto de valores que esta variável pode assumir. Por exemplo, uma variável do tipo boolean só pode assumir os valores TRUE e FALSE.

Os itens (1) e (2) acima dizem respeito ao **tipo abstrato de dados**, ou seja, ao modelo matemático junto com uma coleção de operações definidas sobre este modelo. Um exemplo de tipo abstrato de dados é o conjunto dos números inteiros com as operações de adição, subtração, multiplicação e divisão sobre inteiros.

Já os itens (3) e (4) estão relacionados a aspectos de implementação.

Para representar um tipo abstrato de dados em um computador usa-se uma **estrutura de dados**, que é uma coleção de variáveis, possivelmente de diferentes tipos, relacionadas de diversas maneiras.

Uma **fila de prioridade** é um tipo abstrato de dados que consiste de uma coleção de itens, cada um com um valor ou prioridade associada. Algumas das operações permitidas em uma fila de prioridade são:

insert(v, x): Adiciona o vértice v com valor x na coleção.

delete(v): Remove o vértice v da coleção.

delete-min(): Devolve o vértice com o menor valor e o remove da coleção.

decrease-key(v, x): Muda para x o valor associado ao vértice v; supõe-se que x não é maior que o valor corrente associado a v. Note que decrease-key sempre pode ser implementado como um delete seguido por um insert.

Uma seqüência de operações é chamada **monótona** se os valores devolvidos por sucessivos delete-min's são não-decrescentes. O algoritmo de Dijkstra, descrito no capítulo 3 realiza uma seqüência monótona de operações.

1.5 Linguagem algorítmica e invariantes

A linguagem algorítmica adotada nesta dissertação é a de Feofiloff [13, 14]. Abaixo encontra-se um exemplo onde esta notação é utilizada.

Algoritmo busca seqüencial. Recebe um inteiro positivo n, um vetor de números inteiros $v[0..n-1]$ e um inteiro x e devolve TRUE se existe um índice i em $[0..n-1]$ tal que $v[i] = x$, e, em caso contrário, devolve FALSE.

O algoritmo é iterativo e no início de cada iteração tem-se um inteiro i em $[0..n]$. No início da primeira iteração $i = 0$.

Cada iteração consiste no seguinte.

Caso 1: $i = n$.

Devolva FALSE e pare.

Caso 2: $i < n$.

Caso 2A: $v[i] = x$.

Devolva TRUE e pare.

Caso 2B: $v[i] \neq x$.

$i' := i + 1$.

Comece uma nova iteração com i' no papel de i. □

A ordem em que os casos são enunciados é irrelevante: em cada iteração, qualquer um dos casos aplicáveis pode ser executado. Os casos podem não ser mutuamente exclusivos, e a definição de um caso *não* supõe implicitamente que os demais não se aplicam. Serão utilizadas ainda expressões como "Escolha um i em $[1..n]$", quando não faz diferença qual o valor escolhido. Portanto, a descrição de um algoritmo não é completamente determinística.

A correção dos algoritmos descritos nesta dissertação baseia-se em demonstrações da validade de invariantes. Estes invariantes são afirmações envolvendo objetos mantidos pelo algoritmo que são válidos no início de cada iteração. Exemplos de invariantes para o algoritmo descrito acima são:

(i1) i é um valor em $[0..n]$.

(i2) para todo j em $[0..i-1]$, vale que $v[j] \neq x$.

Deve-se demonstrar que os invariantes valem no início de cada iteração; não faremos isto para o presente exemplo.

Invariantes nos ajudam a entender e demonstrar a correção de algoritmos. No exemplo em questão vê-se facilmente que quando o algoritmo devolve TRUE, no caso 2A, ele não está mentindo,

ou seja, existe i em $[0..n-1]$ tal que $v[i] = x$. Ademais, quando $i = n$, do invariante (i2), tem-se que $v[j] \neq x$ para todo j em $[0..i-1] = [0..n-1]$, e portanto, no caso 1, o algoritmo corretamente devolve FALSE. Finalmente, o algoritmo pára, já que em cada iteração em que não ocorrem os casos 1 e 2A, o caso 2B ocorre, e portanto, o valor de i é acrescido de 1.

1.6 Programação literária e CWEB

Knuth [26] descreve programação literária da seguinte maneira:

"**Programação literária** é uma metodologia que combina linguagem de programação com documentação, deste modo, é possível fazer programas mais robustos, mais portáveis, mais facilmente alteráveis, e mais divertidos de se escrever do que programas escritos somente em linguagem de alto nível. A principal idéia é tratar um programa como parte da literatura, direcionado mais aos seres humanos do que aos computadores. O programa também é visto como um documento hypertexto, mais propriamente, como o World Wide Web. Este livro é uma antologia de experiências incluindo meus recentes documentos nos tópicos relacionados com programação estruturada, bem como o artigo no *The Computer Journal* que lançou a programação literária."

Donald Knuth e Silvio Levy conceberam o CWEB que é um sistema de programação literária. Knuth criou WEB e Levy adaptou o sistema à linguagem C. Ele combina programação em C com documentação tipografada em LaTeX. O sistema está descrito no livro "The CWEB System of Structured Documentation" [28].

Para entender como um programa em CWEB deve ser lido, é preciso entender a notação de **blocos de código**, que são representados por

$$\langle \text{ Nome do bloco } id \rangle \equiv$$
$$\text{código C}$$

A referência a um bloco de código é feita usando-se ⟨ Nome do bloco *id* ⟩ e significa que uma parte de código C será inserida no lugar do mesmo, ou seja, funciona de forma análoga às macros da linguagem C.

A seguir está um exemplo de programa em CWEB, que implementa o algoritmo de busca seqüencial descrito na seção 1.5.

Estrutura geral do programa. O fonte C desse programa será escrito no arquivo "busca.c".

1 ⟨busca.c 1⟩ ≡
⟨Arquivos header e definições 2⟩
⟨Main 3⟩

Inclusão do arquivo, da biblioteca C, necessário ao programa e definição de TRUE e FALSE.

```
2  ⟨ Arquivos header e definições 2 ⟩ ≡
   #include <stdio.h>
   #define TRUE 1
   #define FALSE 0
```

Este código é usado no bloco 1.

Programa principal.

```
3  ⟨ Main 3 ⟩ ≡
   int main( )
   { ⟨ Variáveis locais 4 ⟩
     ⟨ Leitura dos parâmetros de entrada 5 ⟩
     ⟨ Busca seqüencial por x 6 ⟩
   }
```

Este código é usado no bloco 1.

```
4  ⟨ Variáveis locais 4 ⟩ ≡
   int n;
   int v[100];
   int x;
   int i;
```

Este código é usado no bloco 3.

Leitura do inteiro n, do vetor $v[0..n-1]$ e do inteiro x.

```
5  ⟨ Leitura dos parâmetros de entrada 5 ⟩ ≡
   printf("n␣=␣");
   scanf("%d", &n);
   for (i = 0; i < n; i++) scanf("%d", &v[i]);
   printf("x␣=␣");
   scanf("%d", &x);
```

Este código é usado no bloco 3.

6 ⟨ Busca seqüencial por x 6 ⟩ ≡
```
for (i = 0; i < n; i++) {
   ⟨ Verifica se v[i] = x 7 ⟩
}
⟨ Não encontrou x 8 ⟩
```
Este código é usado no bloco 3.

Se $v[i] = x$ devolve TRUE e pára. Caso contrário, continua o loop.

7 ⟨ Verifica se $v[i] = x$ 7 ⟩ ≡
```
if (v[i] ≡ x) {
   printf("TRUE\n");
   return TRUE;
}
```
Este código é usado no bloco 6.

Caso em que $i = n$. Devolve FALSE e para.

8 ⟨ Não encontrou x 8 ⟩ ≡
```
printf("FALSE\n");
return FALSE;
```
Este código é usado no bloco 6.

1.7 Stanford Graph Base

O Stanford Graph Base (SGB) é uma plataforma para algoritmos combinatórios concebida por Knuth [27]. No SGB um grafo é representado internamente através de listas de adjacência. A implementação dos algoritmos faz uso desta plataforma. A seguir está uma breve apresentação das estruturas e de algumas funções do SGB.

A representação de um grafo utiliza estruturas (**structs**) para vértices, arcos e grafos, além de funções básicas para manipular estas estruturas, como descrito a seguir.

Vértice (Vertex). Cada vértice é representado no SGB através de uma estrutura com dois campos padrão e seis campos de utilidade geral (do tipo **util**): portanto, um vértice ocupa 32

bytes na maioria dos sistemas, não contando a memória necessária para strings suplementares. Os campos padrão são:

arcs: apontador para um **Arc**. Armazena o início de uma lista ligada de arcos; e

name: apontador para uma cadeia de caracteres (string). Identifica simbolicamente cada vértice.

Se v aponta para um **Vertex** e $v \rightarrow arcs$ é NULL, então não existem arcos saindo de v. Entretanto, se $v \rightarrow arcs$ não é NULL, ele aponta para uma estrutura do tipo **Arc** representando um arco saindo de v, e esta estrutura **Arc** tem um campo *next* que aponta para o próximo arco na lista ligada encabeçada por $v \rightarrow arcs$. Tal lista contém todos os arcos saindo de v.

Os campos para uso geral são chamados u, v, w, x, y, z. Macros podem ser usadas para dar a estes campos significado, dependendo da aplicação.

```
typedef struct vertex_struct {
    struct arc_struct *arcs;
    char *name;
    util u, v, w, x, y, z;
} Vertex;
```

Arco (Arc). Cada arco é representado por uma estrutura do tipo **Arc**. Cada **Arc** tem três campos padrão e dois campos para uso geral. Portanto, esta estrutura ocupa 20 bytes na maioria dos computadores. Os campos padrão são:

tip: um apontador para um **Vertex**;

next: um apontador para um **Arc**; e

len: um inteiro (long).

Se a aponta para um **Arc** em uma lista de arcos saindo de v, ele representa um arco de comprimento $a \rightarrow len$, indo de v até $a \rightarrow tip$, e o próximo arco saindo de v na lista é representado por $a \rightarrow next$.

Os campos para uso geral são chamados a e b.

```
typedef struct arc_struct {
    struct vertex_struct *tip;
    struct arc_struct *next;
    long len;
    util a,b;
} Arc;
```

Grafo (Graph). Estamos agora preparados para olhar a estrutura do tipo **Graph**. Esta estrutura pode ser passada para um algoritmo que trabalha sobre grafos.

Uma estrutura do tipo **Graph** tem sete campos padrão e seis campos para uso geral. Os campos padrão são:

vertices: um apontador para um vetor de **Vertex**;

n: o número total de vértices;

m: o número total de arcos;

id: um identificador simbólico para o grafo;

util_types: uma representação simbólica do tipo em cada campo para uso geral;

data: aponta para a área usada para armazenar arcos e strings;

aux_data: aponta para a área usada para informação auxiliar.

Os campos para uso geral são *uu, vv, ww, xx, yy, zz.* Exemplo:

```
typedef struct graph_struct {
    Vertex *vertices;
    long n;
    long m;
    char id[ID_FIELD_SIZE];
    char util_types[15];
    Area data;
    Area aux_data;
    util uu,vv,ww,xx,yy,zz;
} Graph;
```

Como uma conseqüência destas convenções, nós visitamos todos os arcos de um grafo g usando o seguinte trecho de programa:

```
Vertex *v;
Arc *a;

for (v = g->vertices; v < g->vertices + g->n; v++)
    for (a = v->arcs; a; a = a->next)
        visite(v,a);
```

Campos para uso geral (util). As estruturas **Vertex**, **Arc** e **Graph** possuem vários campos para uso geral chamados de **util**, que são ou não usados dependendo da aplicação. Cada campo

de uso geral é do tipo **union**, que pode armazenar vários tipos de apontadores ou um inteiro longo.

Os sufixos *.V, .A, .G e .S* no nome de uma variável de uso geral representa um apontador para um **Vertex**, **Arc**, **Graph** ou uma string, respectivamente. O sufixo *.I* significa que a variável é um inteiro (longo).

```
typedef union {
    struct vertex_struct *V;
    struct arc_struct *A;
    struct graph_struct *G;
    char *S;
    long I;
} util;
```

Campo *util_types*. O campo *util_types* deve sempre armazenar uma string de comprimento 14, seguida do usual '/0' (caracter nulo). Os primeiros seis caracteres do *util_types* especificam o uso dos campos de uso geral *u, v, w, x, y, z*; os próximos dois caracteres dão o formato dos campos de uso geral da estrutura **Arc**; os últimos seis dão o formato dos campos de uso geral da estrutura **Graph**. Cada caracter deve ser um *I* (denotando um inteiro longo), *S* (denotando um apontador para uma string), *V* (denotando um apontador para um **Vertex**), *A* (denotando um apontador para um **Arc**), ou *Z* (denotando um campo que não está sendo usado). O valor default de *util_types* é *"ZZZZZZZZZZZZZZ"*, quando nenhum campo para uso geral está sendo usado.

Por exemplo, suponha que um grafo bipartido *g* usa o campo *g→uu.I* para guardar o tamanho da primeira partição. Suponha ainda que *g* tem uma string em cada campo de uso geral *a* de cada **Arc** e usa o campo para uso geral *w* de cada **Vertex** para apontar para um **Arc**. Se *g* não usa nenhum dos demais campos de uso geral, então o seu *util_types* deve conter *"ZZAZZZSZIZZZZZ"*.

Ilustração da representação de um grafo no SGB. As figuras 1.5 e 1.6 ilustram as estruturas do SGB e a representação de um grafo.

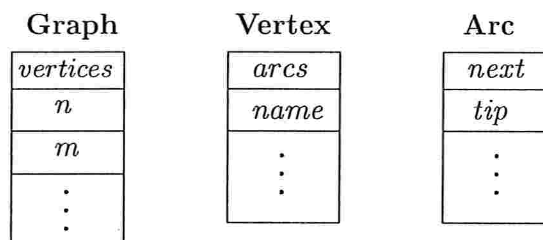


Figura 1.5: Ilustração das estruturas **Graph**, **Vertex** e **Arc**.

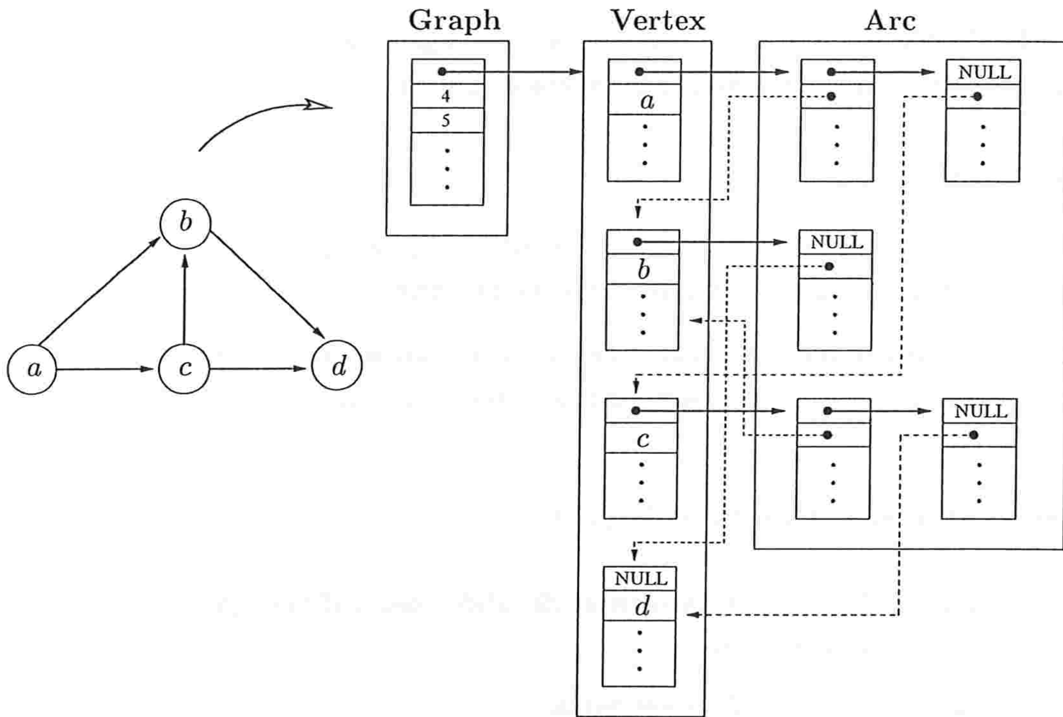


Figura 1.6: Um grafo e sua representação no SGB.

Módulo GB_GRAPH do SGB. Este módulo inclui rotinas para alocar e armazenar novos grafos, novos arcos, novas strings e novas estruturas de dados de todos os tipos.

#include gb_graph.h

Graph **gb_new_graph(long n).* Um novo grafo é criado chamando-se *gb_new_graph(n)*, que devolve um apontador para um registro do tipo **Graph** com n vértices e nenhum arco.

void *gb_new_arc(Vertex *u, Vertex *v, long len).* Cria um novo arco de comprimento *len* de u até v. O arco passa a ser parte do grafo "corrente". O novo arco será apontado por *u→arcs*.

void *gb_new_edge(Vertex *u, Vertex *v, long len).* Similar a *gb_new_arc*. Registros para dois arcos são criados, um de u a v e outro de v a u. Os dois arcos aparecem em posições consecutivas na memória: *v→arcs* é *u→arcs* + 1 quando $u < v$.

char **gb_save_string (char *s).* Faz uma cópia de s para ser usada no grafo "corrente".

void *gb_recycle (Graph *g).* Remove o grafo apontado por g da memória.

Módulo GB_SAVE do SGB. Este módulo contém código para converter grafos da sua representação interna para uma representação simbólica e vice-versa.

#include gb_save.h

long *save_graph (**Graph** *g, **char** *filename).* Converte o grafo apontado por *g* para formato texto e salva o grafo no arquivo de nome *filename*.

Graph **restore_graph (**char** *filename).* Converte um grafo armazenado no arquivo *filename* do formato texto para a representação interna do SGB.

1.8 Estrutura da implementação

As implementações de todos os algoritmos discutidos nesta dissertação estão escritas em CWEB, e fazem uso da plataforma SGB.

A seguir, segue a estrutura geral do programa:

9 ⟨ Inclusão de arquivos header 118 ⟩
⟨ Definições 12 ⟩
⟨ Variáveis globais 13 ⟩
⟨ Funções auxiliares 32 ⟩
⟨ Filas de prioridade 11 ⟩
⟨ Algoritmo de Dijkstra 14 ⟩
⟨ Algoritmo de Dinitz-Thorup 76 ⟩
⟨ Algoritmo de Thorup 88 ⟩
⟨ Teste da condição de otimalidade 138 ⟩
⟨ Programa principal 121 ⟩

Capítulo 2

Problema do caminho mínimo

Estão descritos neste capítulo os ingredientes básicos que envolvem o problema do caminho mínimo, tais como função comprimento, função potencial, função predecessor, critério de otimalidade, etc. A referência básica para este capítulo é Feofiloff [13].

2.1 Descrição

Uma **função comprimento** em (V, A) é uma função de A em $\mathbb{Z}_{\geq}$. Se c é uma função comprimento em (V, A) e uv está em A, então, denotaremos por $c(u, v)$ o valor de c em uv. Se (V, A) é um grafo simétrico e c é uma função comprimento em (V, A), então c é **simétrica** se $c(u, v) = c(v, u)$ para todo arco uv. O **maior comprimento** de um arco será denotado por C, ou seja, $C = \max\{c(u, v) : uv \in A\}$.

C

Se P é um passeio em um grafo (V, A) e c é uma função comprimento, denotaremos por $c(P)$ o **comprimento do caminho** P, ou seja, $c(P)$ é o somatório dos comprimentos de todos os arcos em P. Um passeio P tem **comprimento mínimo** se $c(P) \leq c(P')$ para todo passeio P' que tenha o mesmo início e término que P. A **distância** de um vértice s a um vértice t é o menor comprimento de um caminho de s a t. A distância de s a t em relação a c será denotada por $\mathrm{dist}_c(s, t)$, ou simplesmente, quando a função comprimento estiver subentendida, $\mathrm{dist}(s, t)$ denota a distância de s a t.

$\mathrm{dist}_c(s, t)$
$\mathrm{dist}(s, t)$

Nesta dissertação, estamos interessados no seguinte **problema do caminho mínimo**:

> **Problema** PCM(V, A, c, s): Dado um grafo (V, A), uma função comprimento c e um vértice s, encontrar um caminho de comprimento mínimo de s até t, para cada vértice t em V.

PCM

Na literatura, essa versão do problema é conhecida como *single-source shortest path problem*.

2.2 Funções potencial e critério de otimalidade

Os algoritmos para o problema do caminho mínimo descritos nesta dissertação fornecem certificados de garantia para as suas respostas. Se um vértice t não é acessível a partir de s, um algoritmo pode, para comprovar este fato, devolver uma parte S de V tal que $s \in S$, $t \notin S$ e não existe uv com u em S e v em $V \setminus S$, i.e, $A(S) = \emptyset$. Este seria um certificado combinatório da não acessibilidade de t por s. Entretanto, os certificados fornecidos pelos algoritmos, baseados em funções potencial, serão um atestado compacto para certificar ambos, a minimalidade dos caminhos fornecidos, e a não acessibilidade de alguns vértices por s.

Uma **função potencial** é uma função de V em $\mathbb{Z}_{\geq}$. Se d é uma função potencial e c é uma função comprimento, então dizemos que d **respeita** c em uma parte A' de A se

$$d(v) - d(u) \leq c(u, v) \text{ para cada arco } uv \text{ em } A'.$$

Se d respeita c em A então diz-se que d é **viável**.

Funções potencial viáveis fornecem limitantes inferiores para comprimentos de caminhos. Suponha que c é uma função comprimento em (V, A) e que P é um caminho de um vértice s a um vértice t. Suponha ainda que d é uma função potencial viável. Vale que

$$c(P) \geq d(t) - d(s).$$

De fato, suponha que P é o caminho $\langle s = v_0, \alpha_1, v_1, \ldots, \alpha_k, v_k = t \rangle$. Tem-se que

$$\begin{aligned} c(P) &= c(\alpha_1) + \ldots + c(\alpha_k) \\ &\geq (d(v_1) - d(v_0)) + (d(v_2) - d(v_1)) + \ldots + (d(v_k) - d(v_{k-1})) \\ &= d(v_k) - d(v_0) = d(t) - d(s). \end{aligned}$$

Este fato é resumido através do lema a seguir, que é uma particularização do conhecido lema da dualidade de programação linear [14].

Lema 2.1 (lema da dualidade): *Seja (V, A) um grafo e c uma função comprimento sobre V. Para todo caminho P em (V, A) e toda função potencial viável d sobre V, vale que*

$$d(t) - d(s) \leq c(P),$$

onde s e t são o início e término de P, respectivamente. ∎

Do lema 2.1 tem-se imediatamente os seguintes corolários.

Corolário 2.2 (condição de inacessibilidade): *Se (V, A) é um grafo, c é uma função comprimento, d é uma função potencial viável e s e t são vértices tais que*

$$d(t) - d(s) \geq nC + 1$$

então t não é acessível a partir de s ∎

Corolário 2.3 (condição de otimalidade): *Seja (V, A) um grafo e c é uma função comprimento. Para toda função potencial viável e todo caminho P em (V, A) de s a t, se $d(t) - d(s) = c(P)$, então P é um caminho de comprimento mínimo.* ∎

2.3 Funções predecessor e representação de caminhos

Uma maneira compacta de representar caminhos de um dado vértice até cada um dos demais vértices de um grafo, é através de funções predecessor. Uma **função predecessor** é uma função de V em V. Se (V, A) é um grafo, ψ uma função predecessor sobre V e $v_0, v_1, \ldots, v_k$ são vértices tais que

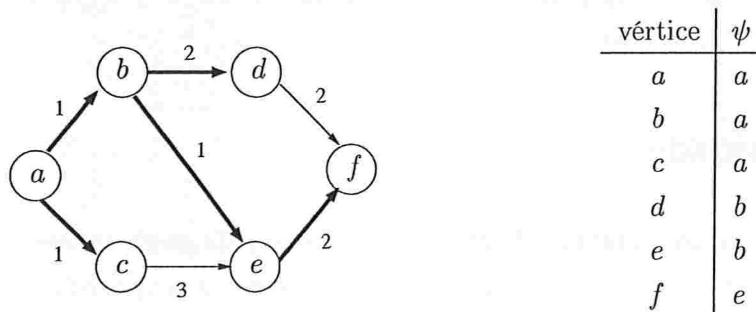
(1) $v_0 = \psi(v_1), v_2 = \psi(v_3), \ldots, v_{k-1} = \psi(v_k)$; e

(2) $\alpha_i = v_{i-1}v_i$ está em A para $i = 1, \ldots, k$,

então dizemos que $\langle v_0, \alpha_1, v_1, \ldots, \alpha_k, v_k \rangle$ é um **caminho determinado por** ψ.

Seja ψ uma função predecessor e $\Psi := \{uv \in A : u = \psi(v)\}$. É dito que ψ **determina uma arborescência** quando o grafo (V, Ψ) é uma arborescência.

Os algoritmos descritos nesta dissertação utilizam funções predecessor para, compactamente, representar todos os caminhos de comprimento mínimo a partir de um dado vértice, conforme ilustrado na figura 2.1.



vértice	ψ
a	a
b	a
c	a
d	b
e	b
f	e

Figura 2.1: Representação de caminhos através da função predecessor ψ com vértice inicial a. Os números próximos aos arcos representam a função comprimento. Os arcos em destaque formam uma arborescência. A tabela ao lado mostra os valores de ψ.

2.4 Examinando arcos e vértices

Algoritmos para encontrar caminhos mínimos mantém, tipicamente, além de uma função predecessor, uma função potencial. O valor desta função potencial para cada vértice é um limitante superior para a distância a partir do vértice s. Esta função é intuitivamente interpretada como uma **distância tentativa** a partir de s.

Seja d uma função potencial e ψ uma função predecessor. Uma operação básica envolvendo as funções ψ e d é **examinar um arco** (*relaxing* [9], *labeling step* [38]). Examinar um arco uv consiste em verificar se d respeita c em uv e, caso não respeite, ou seja, caso

$$d(v) - d(u) > c(u,v) \quad \text{ou, equivalentemente} \quad d(v) > d(u) + c(u,v),$$

fazer

$$d(v) := d(u) + c(u,v) \quad \text{e} \quad \psi(v) := u.$$

Intuitivamente, ao examinar um arco uv tenta-se encontrar um "atalho" para o caminho de s a v determinado por ψ, passando por uv. O passo de examinar uv pode diminuir o valor da distância tentativa dos vértices v e atualizar o predecessor, também tentativo, de v no caminho de comprimento mínimo de s a v. O consumo de tempo para examinar um arco é constante.

Outra operação básica é **examinar um vértice**. Se u é um vértice, examinar u consiste em examinar todos os arcos da forma uv. Em linguagem algorítmica tem-se

Para cada arco uv faça
  se $d(v) > d(u) + c(u,v)$
    então $d(v) := d(u) + c(u,v)$ e $\psi(v) := u$.

O consumo de tempo para examinar um vértice é proporcional ao número de arcos com ponta inicial no vértice.

2.5 Complexidade

O problema do caminho mínimo (PCM), na sua forma mais geral, ou seja, aceitando comprimentos negativos, é NP-difícil: o problema do caminho hamiltoniano pode facilmente ser reduzido a este problema.

Existem limitantes inferiores bem "fracos" para a complexidade do PCM no modelo comparação-adição. Spira e Pan [37] mostraram que, independente do número de adições, pelo menos cn^2 comparações são necessárias para resolver o problema em um grafo simétrico completo[1] com

[1] Um grafo simétrico (V, A) é **completo**, se uv e vu estão em A para todo par de vértices distintos u e v.

n vértices, onde c é uma constante positiva. Limitantes para outros problemas relacionados são listados em Pettie e Ramachandran [31].

Alguns algoritmos para o PCM utilizam como subrotina um algoritmo para o **problema da árvore geradora mínima**, que consiste em

Problema AGM(V, A, c): Dado um grafo simétrico (V, A), uma função comprimento simétrica c, encontrar uma árvore geradora de comprimento mínimo.

AGM

O comprimento de uma árvore é a soma dos comprimentos das suas arestas.

Pettie e Ramachandran [31] observaram que, no pior caso, o tempo necessário para resolver o problema da árvore geradora mínima, no modelo de comparação, não é superior ao tempo necessário para resolver o problema do caminho mínimo no modelo de comparação-adição. Desta forma, um algoritmo para o problema do caminho mínimo, pode utilizar uma subrotina que constrói uma árvore geradora mínima sem que isto comprometa o seu desempenho assintótico no pior caso.

No modelo RAM, o problema AGM pode ser resolvido em tempo linear, como descrito por Fredman e Willard [18]. Assim, um algoritmo para o PCM, no modelo RAM, pode utilizar, digamos, um algoritmo para o problema AGM como pré-processamento, sem comprometer a eficiência teórica do algoritmo. O algoritmo de Thorup, descrito no capítulo 6, resolve o problema AGM em seu pré-processamento.

Capítulo 3

Algoritmo de Dijkstra

Neste capítulo será descrito o celebrado algoritmo de Dijkstra [11] que resolve o problema do caminho mínimo, apresentado na seção 2.1, ou seja:

> **Problema** PCM(V, A, c, s): Dado um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$ e um vértice s, encontrar um caminho de comprimento mínimo de s até t, para cada vértice t em V.

PCM

A idéia geral do algoritmo de Dijkstra para resolver o problema é a seguinte. O algoritmo é iterativo. No início de cada iteração tem-se dois conjuntos disjuntos de vértices S e Q. O algoritmo conhece caminhos de s a cada vértice em $S \cup Q$ e para os vértices em S o algoritmo sabe que o caminho conhecido tem comprimento mínimo. Cada iteração consiste em remover um vértice apropriado de Q, incluí-lo em S e examiná-lo, acrescentando, eventualmente, novos vértices a Q.

A descrição abaixo segue de perto a feita por Feofiloff [13].

3.1 Descrição

Para cada vértice t, acessível a partir de um dado vértice s, o algoritmo de Dijkstra encontra um caminho de s a t que tem comprimento mínimo. Da condição de otimalidade (corolário 2.3) tem-se que, para provar que um caminho P de um vértice s a um vértice t tem comprimento mínimo, basta exibir uma função potencial viável d tal que $c(P) = d(t) - d(s)$. Por outro lado, a condição de inacessibilidade (corolário 2.2) diz que se d é uma função potencial viável com $d(t) - d(s) = nC + 1$, então t não é acessível a partir de s, onde n é o número de vértices do grafo e $C := \max\{c(u, v) : uv \in A\}$. A correção do algoritmo de Dijkstra fornecerá a recíproca dessas condições.

Algoritmo de Dijkstra. Recebe um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$ e um vértice s e devolve uma função predecessor ψ e uma função potencial d que respeita c tais que, para cada vértice t, se t é acessível a partir de s, então ψ determina um caminho de s a t que tem comprimento $d(t) - d(s)$, caso contrário, $d(t) - d(s) = nC + 1$, onde $C := \max\{c(u, v) : uv \in A\}$ e $n := |V|$.

Cada iteração começa com uma função potencial d, uma função predecessor ψ, e partes S, Q e U de V.

No início da primeira iteração $d(s) = 0$ e $d(v) = nC + 1$ para cada vértice v distinto de s, $\psi(v) = v$ para cada vértice v, $S = \emptyset$, $Q = \{s\}$ e $U = V \setminus \{s\}$.

Cada iteração consiste no seguinte.

Caso 1: $Q = \emptyset$.

Devolva d e ψ e pare.

Caso 2: $Q \neq \emptyset$.

Escolha u em Q tal que $d(u)$ seja mínimo.

$S' := S \cup \{u\}$.

$Q' := Q \setminus \{u\}$.

$U' := U$.

Para cada v em V faça $d'(v) := d(v)$ e $\psi'(v) := \psi(v)$.

Para cada arco uv faça

Se $d'(v) > d(u) + c(u, v)$ então

$d'(v) := d(u) + c(u, v)$, $\psi'(v) := u$ e remova[1] v de U' e acrescente a Q'.

Comece nova iteração com d', ψ', S', Q' e U' nos papéis de d, ψ, S, Q e U. □

Nas interpretações intuitivas do algoritmo de Dijkstra é comum dizer que d guarda **distâncias tentativas**, que S é o conjunto dos vértices **examinados**, Q é o conjunto dos vértices **visitados** e que U é o conjunto dos vértices **adormecidos**. A figura 3.1 ilustra o algoritmo de Dijkstra em ação.

3.2 Invariantes

A correção do algoritmo de Dijkstra baseia-se nas demonstrações da validade de uma série de invariantes, enunciados a seguir. Estes invariantes são afirmações envolvendo os dados do

[1] É possível que v já pertença a Q' e não esteja em U'. Nesse caso, estas últimas duas instruções são redundantes.

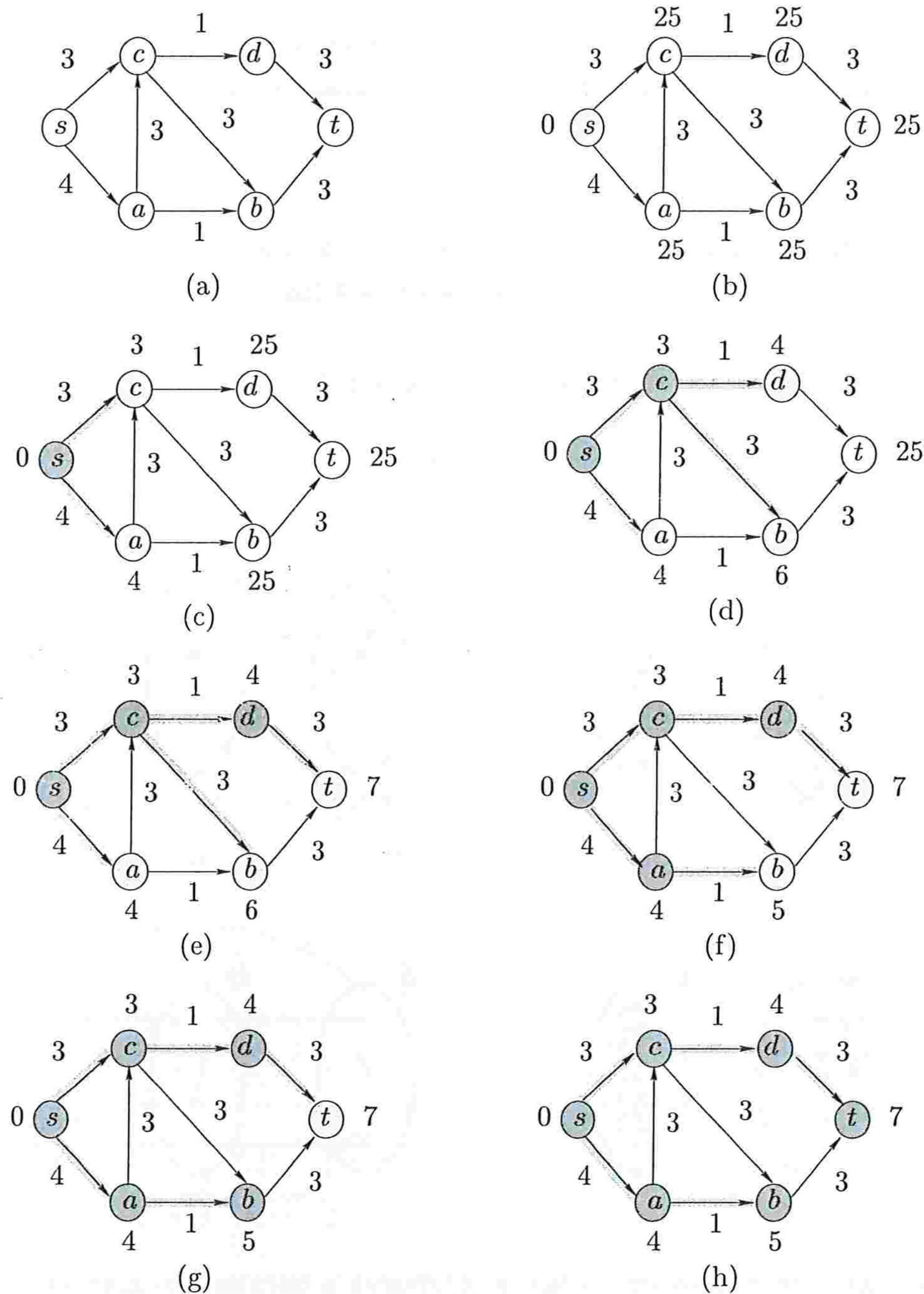


Figura 3.1: Execução do algoritmo de Dijkstra. O vértice inicial é s. (a) exibe um grafo com comprimento nos arcos. (b) mostra a situação no início da primeira iteração. Se um arco (u, v) está sombreado, então $\psi(v) = u$. Os potenciais são os números próximos a cada vértice. Os vértices pretos são os de S, os vértices cinzas são os de Q, e os vértices brancos estão em U. (c)-(g) exibem a situação após cada iteração do caso 2. Os valores finais da função potencial d, e da função predecessor ψ, são mostrados em (h).

problema V, A, c e s e os objetos d, ψ, S, Q e U. As afirmações são válidas no início de cada iteração do algoritmo e dizem como estes objetos se relacionam entre si e com os dados do problema.

Os invariantes estão agrupados conforme os objetos envolvidos.

Estrutura do grafo. Os dois invariantes a seguir envolvem somente as partes S, Q e U. A estrutura induzida por estas partições é ilustrada na figura 3.2(a).

(dk1) (partição) As partes S, Q e U formam uma partição de V.

(dk2) ($A(S,U) = \emptyset$) não existe arco uv com u em S e v em U.

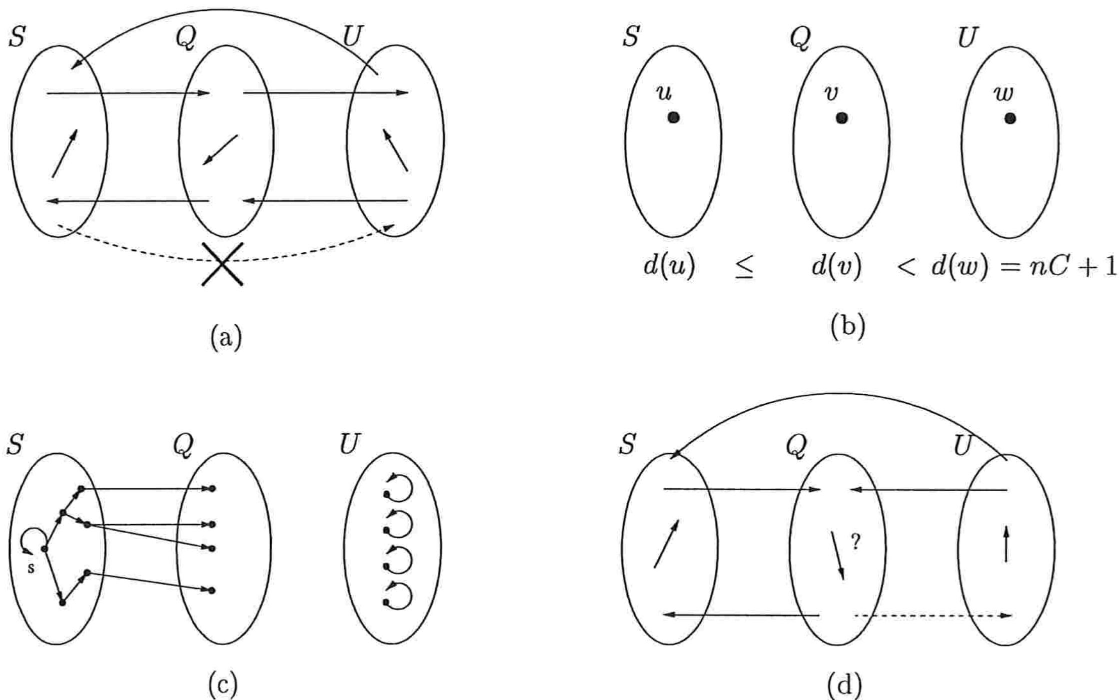


Figura 3.2: Ilustração dos invariantes. A figura (a) mostra a estrutura do grafo em relação à partição S, Q e U de V (invariantes (dk1) e (dk2)). (b) exibe a relação de ordem envolvendo os potenciais dos vértices e as partes S, Q e U (invariante (dk3)). A figura (c) mostra os arcos determinados pela função predecessor ψ (invariantes (dk4), (dk5) e (dk6)). Em (d) os arcos "pontilhados" são os que desrespeitam c, os com o símbolo "?" ao lado são os que podem ou não respeitar c, e o arcos "cheios" são os que respeitam c (invariantes de (dk7) a (dk11)).

Função potencial e $\{S, Q, U\}$. O próximo invariante reflete o fato do algoritmo examinar os vértices em ordem não-decrescente da distância a partir de s, ilustrado na figura 3.2(b).

(dk3) (monotonicidade) para cada u em S, v em Q e w em U vale que[2]

$$d(u) \leq d(v) < d(w) = nC + 1.$$

Função predecessor e $\{S, Q, U\}$. A estrutura da função predecessor em cada iteração, descrita pelos invariantes abaixo, está ilustrada na figura 3.2(c).

(dk4) ($\psi(Q) \subseteq S$) Para cada u em Q vale que $\psi(u)$ está em S.

(dk5) (identidade) Para cada u em U vale que $\psi(u) = u$.

(dk6) (estrutura arbórea de ψ) A função ψ restrita aos vértices em $S \cup Q$ determina uma arborescência em (V, A) com raiz s.

Arcos onde a função potencial respeita ou desrespeita c. Com excessão feita aos arcos com ambas as pontas em Q, o algoritmo sabe se d respeita ou 'desrespeita' c em cada um dos demais arcos do grafo. Isto não é nenhuma surpresa, tendo em vista que os vértices em Q tiveram seus potenciais alterados e os arcos com ambas as pontas em Q ainda não foram examinados. A situação encontra-se ilustrada na figura 3.2(d).

(dk7) (d respeita c em $A[S]$) Para cada arco uv com u e v em S vale que

$$d(v) - d(u) \leq c(u, v).$$

(dk8) (d respeita c em $A[U]$) Para cada arco uv com u e v em U vale que

$$d(v) - d(u) = (nC + 1) - (nC + 1) = 0 \leq c(u, v).$$

(dk9) (d respeita c em $A(S, Q)$ e $A(Q, S)$) Para cada arco uv com u em S e v em Q ou u em Q e v em S vale que

$$d(v) - d(u) \leq c(u, v).$$

(dk10) (d respeita c em $A(U, S)$ e $A(U, Q)$) Para cada arco uv com u em U e v em $S \cup Q$ vale que

$$d(v) - d(u) = d(v) - (nC + 1) < c(u, v).$$

(dk11) (d desrespeita c em $A(Q, U)$) Para cada arco uv com u em Q e v em U vale que

$$d(v) - d(u) = (nC + 1) - d(u) > c(u, v).$$

[2] A expressão "$d(u) \leq d(v) < d(w) = nC+1$" deve ser entendida como uma abreviação. Se algum dos conjuntos envolvidos é vazio, considere as desigualdades que fazem sentido.

Função potencial e função predecessor

(dk12) (folgas complementares) para cada arco uv tal que $\psi(v) = u$ vale que

$$d(v) - d(u) = c(u, v).$$

Para demonstrar que as afirmações acima são legítimos invariantes deve-se demonstrar que:

(a) as afirmações valem no início da primeira iteração; e

(b) se as afirmações valem no início de uma iteração em que ocorre o caso 2, então as afirmações também valem ao final da iteração com d', ψ', S', Q' e U' nos papéis de d, ψ, S, Q e U.

De (a) e (b) conclui-se que as afirmações também valem no início da última iteração; quando ocorre o caso 1. Supondo-se, é claro, que mais cedo ou mais tarde o caso 1 sempre ocorre.

É evidente que cada uma das afirmações valem no início da primeira iteração e não é difícil verificar (b) para cada umas das afirmações. A seguir estão as demonstrações de (dk1), (dk2), (dk3), (dk7) e (dk9) a título de ilustração. Nas demonstrações, são feitas referências ao procedimento de examinar um vértice ou arco como descrito na seção 2.4.

Demonstração de (dk1): Considere uma iteração em que ocorre o caso 2. No início da iteração, $\{S, Q, U\}$ é uma partição de V. Portanto, antes do algoritmo examinar o vértice u tem-se que $\{S', Q', U'\}$ é uma partição de V, já que $S' = S \setminus \{u\}$, $Q' = Q \cup \{u\}$, e $U' = U$. Durante o exame de cada arco, os vértices eventualmente removidos de U' são a seguir inseridos em Q'. Logo, no final da iteração, $\{S', Q', U'\}$ forma uma partição de V. ∎

Demonstração de (dk2): Considere uma iteração em que ocorre o caso 2. No início da iteração tem-se que $A(S, U) = \emptyset$. Imediatamente antes do vértice u ser examinado vale que $S' = S \cup \{u\}$ e $U' = U$. Do invariante (dk11) sabe-se que, para cada arco uv com u em S' e v em U',

$$d(v) - d(u) > c(u, v).$$

Assim, cada arco uv com v em U' será, durante o exame do arco uv, removido de U' e acrescentado a Q'. Portanto, no final da iteração, tem-se que $A(S', U') = \emptyset$. ∎

É importante notar que na demonstração de (dk3) e (dk9) que é utilizada a escolha apropriada do vértice u: $d(u) \leq d(v)$ para cada v em Q.

Demonstração de (dk3): Considere uma iteração em que ocorre o caso 2. Note que ao final da iteração tem-se que $d'(v) \neq d(v)$ se e somente se v está em $Q' \subseteq (Q \setminus \{u\}) \cup U$ e uv é um arco do grafo com $d(v) - d(u) > c(u, v)$. Ademais, se $d'(v) \neq d(v)$ então $d'(v) = d(u) + c(u, v) < d(v)$.

Do invariante (dk3) sabe-se que para cada x em S, y em Q e z em U vale que $d(x) \leq d(y) < d(z) = nC + 1$. Assim, pela escolha de u (e como c é uma função de A em $\mathbb{Z}_{\geq}$), após examinar o vértice u tem-se que

$$d'(x) \leq d(u) \leq d'(y) < d'(z) = nC + 1$$

para cada x em $S' = S \cup \{u\}$, y em Q' e z em U'. ∎

Demonstração de (dk7): No início da iteração tem-se que d respeita c em $A[S]$ (invariante (dk7)), em $A(S,Q)$, e em $A(Q,S)$. Desta forma, no final de uma iteração em que ocorre o caso 2 vale que d' respeita c em $A[S']$, pois $S' = S \cup \{u\}$ e $d'(v) = d(v)$ para todo v em S'. ∎

Demonstração de (dk9): Novamente, considere uma iteração em que ocorre o caso 2. Da demonstração do invariante (dk3), obtem-se que no final da iteração vale que $d'(x) - d'(y) \leq 0$ para cada x em $S' = S \cup \{u\}$, y em Q'. Portanto, como os comprimentos dos arcos são não-negativos, d' respeita c em $A(Q', S')$.

O processo de examinar u faz com que d' respeite c em cada arco com ponta inicial em u. Do invariante (dk9) sabe-se que d respeita c em $A(S,Q)$. Assim, como $d'(v) = d(v)$ para cada v em S' e $d'(v) \leq d(v)$ para cada v em Q', concluí-se que d' respeita c em $A(S', Q')$.

∎

3.3 Correção

A correção do algoritmo de Dijkstra é facilmente demonstrada através dos invariantes apresentados.

Teorema 3.1 (teorema da correção): *Para cada vértice t acessível a partir de s o algoritmo de Dijkstra devolve um caminho de s a t que tem comprimento mínimo.*

Demonstração: Suponha que ψ e d são as funções devolvidas pelo algoritmo. Quando o algoritmo pára tem-se que $Q = \emptyset$ e do invariante (dk1) vale que S e U é uma partição de V.

Ao final do algoritmo, S é o conjunto de vértices acessíveis a partir de s. De fato, pelo invariante (dk2) sabe-se que nenhum vértice de U é acessível a partir de s, já que $A(S,U) = \emptyset$, e pelo invariante (dk6) tem-se que para cada vértice t em S a função ψ determina um caminho de s a t, já que ψ determina uma arborescência em $(S, A[S])$ com raiz em s.

Como $Q = \emptyset$ e $A(S,U) = \emptyset$, então dos invariantes (dk7) a (dk10) conclui-se que a função potencial d é viável.

Resta apenas verificar que cada caminho P com início em s, e determinado por ψ, tem

comprimento mínimo. Suponha que t em S é o término de P. Do invariante (dk12) obtem-se que $c(P) = d(t) - d(s)$. Finalmente, como d é viável, pela condição de otimalidade conclui-se que P tem comprimento mínimo. ∎

Um corolário importante da correção do algoritmo é a seguinte especialização do teorema da dualidade de programação linear ao PCM.

Teorema 3.2 (teorema da dualidade): *Seja (V, A) um grafo, c uma função comprimento de A em $\mathbb{Z}_{\geq}$ e s e t vértices do grafo. Se t é acessível a partir de s, então vale que*

$$\min\{c(P) : P \text{ é um caminho de } s \text{ a } t\} = \max\{d(t) - d(s) : d \text{ é uma função potencial viável}\}.$$

Demonstração: Do lema da dualidade 2.1 tem-se que para todo caminho P' e toda função potencial viável d' vale que $c(P') \geq d'(t) - d'(s)$. O algoritmo de Dijkstra devolve um caminho P e uma função potencial viável d tal que $c(P) = d(t) - d(s)$. Logo, P é um caminho que tem comprimento mínimo e d é uma função potencial viável para a qual a diferença de potencial entre s e t é máxima. ∎

3.4 Eficiência

Primeiro, é conveniente notar que d', ψ', S', Q' e U' foram introduzidos na descrição do algoritmo por meras razões técnicas: ajudam nas demonstrações de invariantes. Desta forma, em termos de eficiência, não há necessidade de levar em consideração as instruções que inicializam estes objetos, já que estas podem ser simplesmente eliminadas da descrição. A descrição pode ser feita inteiramente em termos de d, ψ, S, Q e U.

O número de ocorrências do caso 2 é no máximo n, pois em cada ocorrência é acrescentado um novo vértice a S e $\{S, Q, U\}$ é uma partição de V. Portanto, o número de iterações é no máximo $n + 1$.

As duas seguintes operações são as principais responsáveis pelo consumo de tempo assintótico do algoritmo:

Escolha de um vértice com potencial mínimo. Cada execução desta operação gasta tempo $O(n)$. Como o número de ocorrências do caso 2 é no máximo n, então o tempo total gasto pelo algoritmo para realizar essa operação é $O(n^2)$.

Atualização do potencial. Ao examinar um arco o algoritmo eventualmente diminui o potencial da ponta final. Essa atualização de potencial é realizada não mais que $|A(u)|$ vezes ao examinar o vértice u. Ao todo, o algoritmo pode realizar essa operação não mais que

$\sum_{u \in V} |A(u)| = m$ vezes. Desde que cada atualização seja feita em tempo constante, o algoritmo requer uma quantidade de tempo proporcional a m para atualizar potenciais.

Assim, o consumo de tempo do algoritmo no pior caso é $O(n^2 + m) = O(n^2)$. O teorema abaixo resume a discussão.

Teorema 3.3 (consumo de tempo): *O algoritmo de Dijkstra, quando executado, no modelo de comparação-adição, em um grafo com n vértices e m arcos, gasta tempo $O(n^2)$.* ∎

Para grafos densos, ou seja, grafos onde $m = \Omega(n^2)$, o consumo de tempo de $O(n^2)$ do algoritmo de Dijkstra é ótimo, pois, é necessário que todos os arcos do grafo sejam examinados. Entretanto, se $m = O(n^{2-\epsilon})$ para algum ϵ positivo, existem métodos sofisticados, como o heap de Johnson [24], o fibonacci heap de Fredman e Tarjan [16], que permitem diminuir o tempo gasto para encontrar um vértice com potencial mínimo, gerando assim implementações que consomem menos tempo para resolver o problema.

A maneira mais utilizada para realizar as operações acima é através de implementações de filas de prioridade, que será objeto de estudo do próximo capítulo.

3.5 Complexidade

Fredman e Tarjan [16] observaram que como o algoritmo de Dijkstra examina os vértices em ordem de distância a partir de s (invariante (dk3)) então o algoritmo está, implicitamente, ordenando estes valores (ver figura 3.3). Assim, qualquer implementação do algoritmo de Dijkstra para o modelo de comparação-adição realiza, no pior caso, $\Omega(n \log n)$ comparações. Portanto, qualquer implementação do algoritmo para o modelo de comparação-adição faz $\Omega(m + n \log n)$ operações elementares.

Teorema 3.4 (limitante inferior): *No modelo de comparação-adição, o algoritmo de Dijkstra, quando executado em um grafo com n vértices e m arcos, gasta tempo $\Omega(m + n \log n)$.* ∎

Por outro lado, Thorup [40] mostrou que um algoritmo linear para ordenação pode ser usado em uma implementação do algoritmo de Dijkstra que executa um número de operações proporcional ao tamanho do grafo.

No próximo capítulo será mostrada a implementação de Fredman e Tarjan [16] para modelo de comparação-adição que gasta tempo $O(m + n \log n)$. O algoritmo de Dijkstra com essa

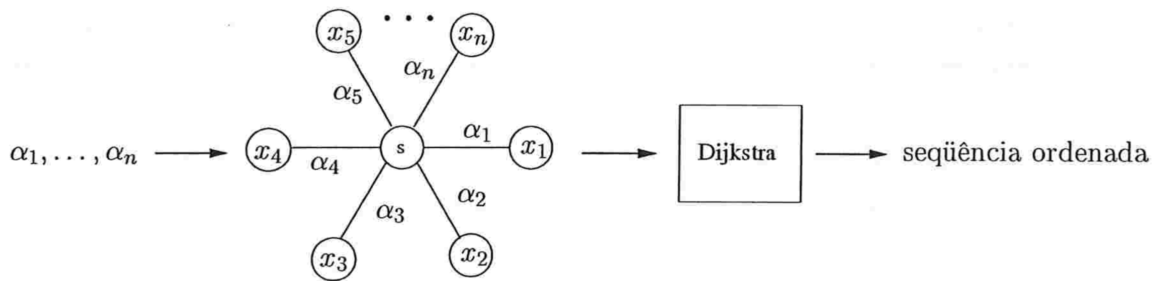


Figura 3.3: Esquema ilustrando como o algoritmo de Dijkstra pode ser utilizado para ordenar uma seqüência de números $\alpha_1, \ldots, \alpha_n$.

implementação é ótimo.

3.6 Versão em CWEB

Das partes S, Q e U descritas na seção 3.1, as partes S e U não entram na implementação. E a parte Q é representada através de uma fila de prioridade (seção 1.4).
Para criar Q é utilizada a operação create_pq, e as operações de Q serão representadas pelas funções insert, delete_min e decrease_key.

11 ⟨ Filas de prioridade 11 ⟩ ≡

```
void (*create_pq)();      /* cria Q */
void (*insert)();
Vertex *(*delete_min)();
void (*decrease_key)();
```

Veja também blocos 25, 26, 27, 28, 29, 33, 34, 35, 36, 38, 42, 43, 44, 45, 48, 53, 54, 55, 56, 57, 61, 62, 63, 64 e 69.

Este código é usado no bloco 9.

Os itens armazenados em Q são vértices, e como valor associado a cada um deles, temos o potencial. O potencial em cada vértice será representado pelo campo *dist*, e o predecessor pelo campo *pred*.

12 ⟨ Definições 12 ⟩ ≡

```
#define dist v.I
#define pred u.V
```

Veja também blocos 23, 40, 52, 59, 72, 89, 94, 104 e 119.

Este código é usado no bloco 9.

Será armazenado na variável *qsize*, o número de vértices em Q.

13 ⟨ Variáveis globais 13 ⟩ ≡

```
unsigned long qsize;
```

Veja também blocos 24, 31, 71, 90, 103, 120 e 125.

Este código é usado no bloco 9.

14 ⟨ Algoritmo de Dijkstra 14 ⟩ ≡

```
void dijkstra(g, s)
      Graph *g;
      Vertex *s;      /* vértice inicial */
{ ⟨ Variáveis de Dijkstra 15 ⟩
  ⟨ Inicializa d e ψ 16 ⟩
  ⟨ Inicializa a fila Q com s 17 ⟩
  while (⟨ Fila Q não está vazia 18 ⟩) {
    ⟨ Escolha u em Q tal que d(u) seja mínimo 19 ⟩
    ⟨ Examine o vértice u 20 ⟩
  }
}
```

Este código é citado no bloco 10.

Este código é usado no bloco 9.

15 ⟨ Variáveis de Dijkstra 15 ⟩ ≡

```
register Vertex *v, *u;
register Arc *a;
```

Este código é usado no bloco 14.

No início da primeira iteração tem-se que $d(s) = 0$ e $d(v) = nC+1$ para cada vértice v distinto de s, $\psi(v) = v$ para cada vértice v. Na implementação, $nC + 1$ será representado pela variável *infinito*.

16 ⟨ Inicializa d e ψ 16 ⟩ ≡

```
for (v = g->vertices; v < g->vertices + g->n; v++) {
  v->dist = infinito;
  v->pred = v;
}
```

$s\rightarrow dist = 0;$
$num_exam = 0;$
$atualiza_fp = 0;$

Este código é usado no bloco 14.

Cria Q e a inicializa com o vértice inicial s.

17 ⟨ Inicializa a fila Q com s 17 ⟩ ≡
$create_pq(g);$
$qsize = 0;$
$insert(s);$
$qsize{+}{+};$

Este código é usado no bloco 14.

18 ⟨ Fila Q não está vazia 18 ⟩ ≡
$(qsize > 0)$

Este código é usado no bloco 14.

19 ⟨ Escolha u em Q tal que $d(u)$ seja mínimo 19 ⟩ ≡
$u = delete_min();$
$qsize{-}{-};$

Este código é usado no bloco 14.

Examina todos os arcos da forma uv (seção 2.4).

20 ⟨ Examine o vértice u 20 ⟩ ≡
for $(a = u\rightarrow arcs;\ a;\ a = a\rightarrow next)$ {
  $v = a\rightarrow tip;$
  ⟨ Examine o arco uv 21 ⟩
}
$num_exam{+}{+};$

Este código é usado no bloco 14.

Faz d respeitar c em uv (seção 2.4).

```
21  ⟨ Examine o arco uv 21 ⟩ ≡
  if (v→dist − u→dist > a→len) {        /* se a função potencial não é viável */
    atualiza_fp++;
    if (v→dist ≡ infinito) {      /* v não está na fila */
      v→dist = u→dist + a→len;
      v→pred = u;
      insert(v);
      qsize++;
    }
    else {      /* v já estava na fila */
      v→dist = u→dist + a→len;
      v→pred = u;
      decrease_key(v);
    }
  }
```

Este código é usado no bloco 20.

O teorema a seguir resume o número de operações feitas pela implementação acima, para manipular a fila de prioridades Q.

Teorema 3.5 (número de operações): *O algoritmo de Dijkstra, quando executado em um grafo com n vértices e m arcos, realiza uma seqüência de n operações insert, n operações delete-min e no máximo m operações decrease-key.* ∎

Diferentes maneiras de se implementar as funções insert, delete_min e decrease_key serão estudadas no próximo capítulo.

Capítulo 4

Implementações de Filas de Prioridade

O algoritmo de Dijkstra, segundo o teorema 3.5, realiza uma seqüência de n insert, n deletemin e no máximo m decrease-key operações, em um grafo com n vértices e m arcos. Logo, o consumo de tempo do algoritmo de Dijkstra pode variar conforme a implementação de cada uma dessas operações da fila de prioridade.

Existem muitos trabalhos envolvendo implementações de filas de prioridade com o intuito de reduzir o tempo gasto pelo algoritmo de Dijkstra. Para citar alguns bons exemplos temos [3, 7, 16].

As estruturas de dados utilizadas na implementação das filas de prioridade podem ser divididas em duas categorias, conforme as operações elementares utilizadas:

(1) (modelo de comparação) estruturas baseadas em comparações; e

(2) (modelo RAM) estruturas baseadas em "bucketing".

Bucketing é um método de organização dos dados que particiona um conjunto em partes chamadas **buckets**. No que diz respeito ao algoritmo de Dijkstra, cada bucket agrupa vértices de um grafo relacionados através de prioridades, que nesse caso, são os potenciais.

Neste capítulo, são descritas as implementações das estruturas de dados *heap*, D-*heap* e *fibonacci heap* que são baseadas em comparações e as estruturas *bucket heap*[1] e *radix heap* que trabalham no modelo RAM. Nas implementações, cada item da fila será um vértice e a prioridade desse vértice será um valor numérico, o seu potencial. As implementações foram baseadas nos livros, Network Flows [1], Introduction to Algorithms [8], The Stanford GraphBase [27] e no livro de Schrijver [36].

[1] Na literatura, a implementação do algoritmo que utiliza buckets é conhecida como implementação de Dial [10].

4.1 Heap

Um **heap** é uma estrutura de dados que mantêm uma seqüência de itens, onde cada item é acessado através de um índice numérico. No nosso caso, os itens são vértices, que serão dispostos em um vetor Q conforme o seu potencial. A organização dos vértices em Q respeita a seguinte propriedade:

Propriedade 4.1 (ordem):

$$d(Q[\lfloor i/2 \rfloor]) \leq d(Q[i]), \text{ para cada posição } i \text{ em } [2..qsize],$$

onde qsize representa o número de vértices em Q.

Intuitivamente, o vetor Q também pode ser visto como uma árvore binária, onde cada posição do vetor representa um vértice da árvore. A figura 4.1 ilustra essa representação.

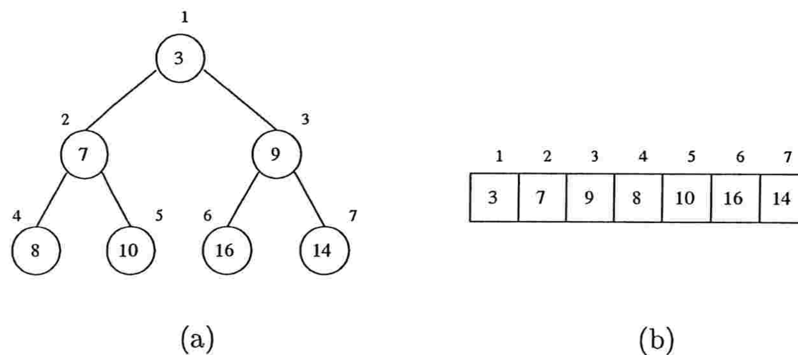


Figura 4.1: (a) um heap visto como uma árvore binária; (b) o mesmo heap visto como um vetor.

Na implementação, o vetor Q é simulado utilizando-se o campo utilitário x da estrutura do SGB (seção 1.7), ou seja, $(Q + i)\to x.V$ aponta para o vértice da posição i. Para simplificar a notação, $(Q+i)\to x.V$ é definido como $Q(i)$. Além do mais, se $Q(i)$ aponta para o vértice v, então $v\to posicao = i$.

23 ⟨ Definições 12 ⟩ +≡
#define $Q(i)$ $(Q + i)\to x.V$
#define *posicao* $w.I$

24 ⟨ Variáveis globais 13 ⟩ +≡
Vertex $*Q$; /* heap */

Para criar o heap Q é utilizada a função *create_heap*.

25 ⟨ Filas de prioridade 11 ⟩ +≡

```
void create_heap(g)      /* cria um heap vazio */
    Graph *g;
{
  register Vertex *v;
  Q = g->vertices;
  for (v = g->vertices; v < g->vertices + g->n; v++) {
    v->posicao = 0;
    v->x.V = Λ;
  }
}
```

As implementações das operações insert, delete-min e decrease-key, são representadas pelas funções *insert_heap, delete_min_heap* e *decrease_key_heap.*

26 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_heap(Vertex *v);
Vertex *delete_min_heap();
void decrease_key_heap(Vertex *v);
```

O tempo gasto para inserir um vértice no heap é $O(1)$, somado ao tempo consumido pela operação *decrease_key_heap*. Essa operação funciona como se o último vértice tivesse seu potencial alterado (diminuído), então é necessário o reposicionamento deste vértice em Q, de modo que a propriedade 4.1 continue sendo respeitada.

27 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_heap(v)
    Vertex *v;
{
  v->posicao = qsize + 1;
  decrease_key_heap(v);
}
```

Em um heap, o vértice com o menor potencial se encontra na primeira posição, ou seja, em $Q(1)$. Porém, após sua remoção, é preciso encontrar o novo vértice com potencial mínimo. No pior caso, essa operação gasta tempo $O(\log qsize)$.

28 ⟨ Filas de prioridade 11 ⟩ +≡

```
Vertex *delete_min_heap()
{
  register unsigned long p, f;
  register Vertex *v, *vmin;
  if (qsize ≡ 0) return Λ;
  vmin = Q(1);
  v = Q(qsize);
  p = 1;
  f = p ≪ 1;      /* f = 2p */
  while (f ≤ (qsize − 1)) {
    if ((f < (qsize − 1)) ∧ (Q(f)→dist > Q(f + 1)→dist)) f++;
    if (Q(f)→dist ≥ v→dist) break;
    Q(p) = Q(f);
    Q(p)→posicao = p;
    p = f;
    f = p ≪ 1;
  }
  Q(p) = v;
  Q(p)→posicao = p;
  return vmin;
}
```

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reposicioná-lo em Q, de modo que a propriedade 4.1 continue sendo respeitada. No pior caso, essa operação gasta tempo $O(\log qsize)$.

29 ⟨ Filas de prioridade 11 ⟩ +≡

```
void decrease_key_heap(v)
    Vertex *v;
{
  register unsigned long p, f;
  f = v→posicao;
  p = f ≫ 1;      /* ⌊f/2⌋ */
  while ((p > 0) ∧ (Q(p)→dist > v→dist)) {
    Q(f) = Q(p);
    Q(f)→posicao = f;
    f = p;
```

```
      p = f >> 1;
    }
    Q(f) = v;
    Q(f)->posicao = f;
}
```

Lema 4.2 (operações heap): *Na implementação da fila de prioridade Q, utilizando um heap, cada operação insert, delete-min e decrease-key, gasta tempo $O(\log n)$.* ▮

Portanto, do teorema 3.5 e do lema 4.2, o tempo gasto pelo algoritmo de Dijkstra utilizando um heap, é

$$\underbrace{O(n \log n)}_{\text{insert}} + \underbrace{O(m \log n)}_{\text{decrease-key}} + \underbrace{O(n \log n)}_{\text{delete-min}} = O(m \log n).$$

Teorema 4.3: *A implementação do algoritmo de Dijkstra que utiliza um heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m \log n)$.* ▮

4.2 D-heap

Um **D-heap**, assim como o heap (seção 4.1), é uma estrutura de dados que mantêm uma seqüência de vértices, num vetor Q, dispostos de maneira organizada em relação a função potencial, de maneira que:

Propriedade 4.4 (ordem):

$$d(Q[\lceil (i-1)/\mathrm{D} \rceil]) \leq d(Q[i]), \text{ para cada posição } i \text{ em } [2..qsize],$$

onde $qsize$ representa o número de vértices em Q e D é o maior número de filhos de um vértice.

Intuitivamente, o vetor Q pode ser visto como uma árvore, em que cada vértice pertencente a ele pode ter no máximo D filhos. A figura 4.2 ilustra a representação de um 3-heap.

Na implementação, D é o maior número de filhos de um vértice. A escolha de D é feita dinamicamente, em relação a $\max\{2, \lceil m/n \rceil\}$. O motivo pelo qual D é escolhido dessa maneira

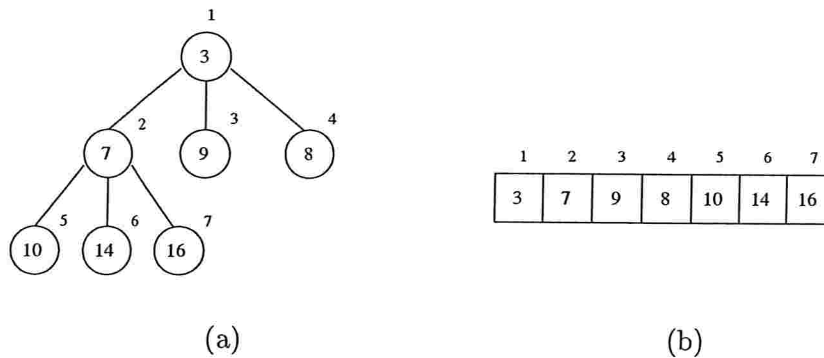


Figura 4.2: Ilustração de um 3-heap. Em (a) o 3-heap é visto como uma árvore e em (b) como um vetor.

é mostrado no final da seção.

31 ⟨ Variáveis globais 13 ⟩ +≡
unsigned long D; /* número máximo de filhos de um vértice */

A função *teto*(i, j) devolve $\lceil i/j \rceil$. A primeira utilidade da função é encontrar um valor para D.

32 ⟨ Funções auxiliares 32 ⟩ ≡

```
unsigned long teto(i, j)
    unsigned long i;
    unsigned long j;
{
  return (i % j ≡ 0 ? i/j : i/j + 1);
}
```

Veja também blocos 41, 50, 60, 73, 74, 75, 86, 96, 105 e 129.

Este código é usado no bloco 9.

Para criar o D-heap Q é utilizada a função *create_dheap*.

33 ⟨ Filas de prioridade 11 ⟩ +≡

```
void create_dheap(g)    /* cria um heap vazio */
    Graph *g;
{
  unsigned long t;
  register Vertex *v;
```

```
    Q = g->vertices;
    for (v = g->vertices; v < g->vertices + g->n; v++) {
      v->posicao = 0;
    }
    t = teto(g->m/2, g->n);        /* número de arestas é o dobro do número de arcos */
    D = t > 2 ? t : 2;
  }
```

As implementações das operações insert, delete-min e decrease-key, são representadas pelas funções *insert_dheap*, *delete_min_dheap* e *decrease_key_dheap*.

```
34 ⟨ Filas de prioridade 11 ⟩ +≡
  void insert_dheap(Vertex *v);
  Vertex *delete_min_dheap();
  void decrease_key_dheap(Vertex *v);
```

Inserir um vértice no D-heap funciona de forma análoga a inserir um vértice no heap.

```
35 ⟨ Filas de prioridade 11 ⟩ +≡
  void insert_dheap(v)
      Vertex *v;
  {
    v->posicao = qsize + 1;
    decrease_key_dheap(v);
  }
```

Assim como no heap 4.1, o vértice com o menor potencial pode ser encontrado na primeira posição, ou seja, em $Q(1)$. E após sua remoção, é preciso encontrar o novo vértice com potencial mínimo. No pior caso, essa operação gasta tempo $O(D \log_D qsize)$

```
36 ⟨ Filas de prioridade 11 ⟩ +≡
  Vertex *delete_min_dheap()
  {
    register unsigned long p, f, i, ultimo_filho;
    register Vertex *v, *vmin;
    if (qsize ≡ 0) return Λ;
    vmin = Q(1);
    v = Q(qsize);
```

```
p = 1;     /* os filhos do vértice da posicão p, são encontrados nas posições
      pD − D + 2, ..., min{n, pD + 1} */
f = p * D − D + 2;
while (f ≤ (qsize − 1)) {
  ⟨ Encontra filho f de p tal que d(f) seja mínimo 37 ⟩
  if (Q(f)→dist ≥ v→dist) break;
  Q(p) = Q(f);
  Q(p)→posicao = p;
  p = f;
  f = p * D − D + 2;
}
Q(p) = v;
Q(p)→posicao = p;
return vmin;
}
```

Encontra o $\min\{Q(f)\text{→}dist\}$ para cada filho f de p.

37 ⟨ Encontra filho f de p tal que $d(f)$ seja mínimo 37 ⟩ ≡

```
ultimo_filho = (qsize − 1) < p * D + 1 ? (qsize − 1) : p * D + 1;
for (i = f + 1; i ≤ ultimo_filho; i++)
  if (Q(f)→dist > Q(i)→dist) f = i;
```

Este código é usado no bloco 36.

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reposicioná-lo em Q, de modo que a propriedade 4.4 continue sendo respeitada. No pior caso, essa operação gasta tempo $O(\log_D qsize)$.

38 ⟨ Filas de prioridade 11 ⟩ +≡

```
void decrease_key_dheap(v)
    Vertex *v;
{
  register unsigned long p, f;
  f = v→posicao;
  p = teto(f − 1, D);     /* pai de um vértice é encontrado na posição ⌈(f − 1)/D⌉ */
  while ((p > 0) ∧ (Q(p)→dist > v→dist)) {
    Q(f) = Q(p);
    Q(f)→posicao = f;
```

```
        f = p;
        p = teto(f − 1, D);
    }
    Q(f) = v;
    Q(f)→posicao = f;
}
```

Lema 4.5 (operações D-heap): *Na implementação da fila de prioridade Q, utilizando um D-heap, cada operação insert e decrease-key gasta tempo $O(\log_D n)$ e a operação delete-min gasta tempo $O(D \log_D n)$, onde $D = \max\{2, \lceil m/n \rceil\}$.* ∎

Portanto, do teorema 3.5 e do lema 4.5, o tempo gasto pelo algoritmo de Dijkstra utilizando um D-heap, é

$$\underbrace{O(n \log_D n)}_{\text{insert}} + \underbrace{O(m \log_D n)}_{\text{decrease-key}} + \underbrace{O(nD \log_D n)}_{\text{delete-min}} = O(m \log_D n).$$

Note que o valor escolhido para D na implementação é ótimo, pois para grafos esparsos, isto é, quando $m = O(n)$, o tempo gasto é $O(n \log n)$, e para grafos não-esparsos, isto é, quando $m = \Omega(n^{1+\epsilon})$ para algum $\epsilon > 0$, o tempo gasto é $O(m \log_D n) = O((m \log n)/(\log D)) = O((m \log n)/(\log n^{\epsilon})) = O((m \log n)/(\epsilon \log n)) = O(m/\epsilon) = O(m)$. A última igualdade é verdadeira desde que ϵ seja uma constante. Portanto, o tempo gasto é $O(m)$ que é ótimo.

Teorema 4.6: *A implementação do algoritmo de Dijkstra que utiliza um D-heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m \log_D n)$.* ∎

4.3 Fibonacci heap

Um **fibonacci heap** é um conjunto de arborescências que respeitam as seguintes propriedades:

Propriedade 4.7 (ordem): *Para cada par (p, f), em uma mesma arborescência, onde p é o pai do vértice f, $d(p) \leq d(f)$ (análoga a propriedade 4.1).*

Propriedade 4.8 (descendentes): *Para cada vértice p, os filhos de p podem ser ordenados de maneira que o i-ésimo filho tenha pelo menos $i - 2$ filhos.*

Dizemos que o **grau** de um vértice é o seu número de filhos. Como conseqüência da propriedade 4.8 pode-se notar que todo vértice de grau k possui pelo menos F_{k+2} descendentes (incluindo ele mesmo), onde F_k é o número de fibonacci,

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

ou

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

dando a entender a origem do nome fibonacci heap.

A implementação de um fibonacci heap é feita através de uma lista ligada Q, como ilustrada na figura 4.3.

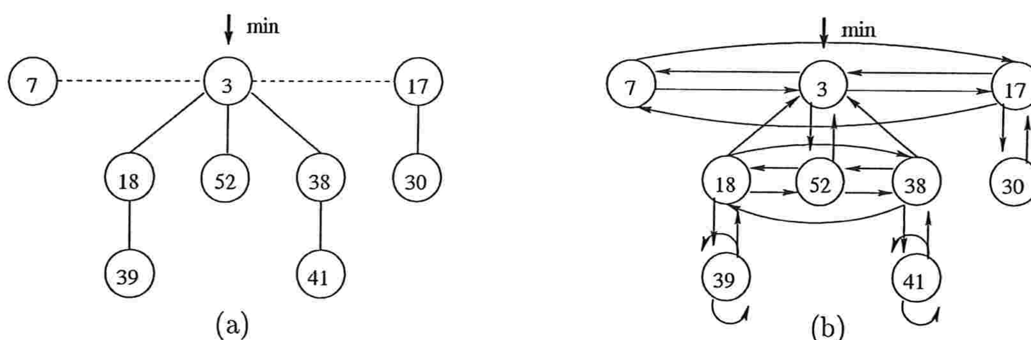


Figura 4.3: Em (a) é exibido um fibonacci heap composto por três arborescências. A linha pontilhada enfatiza o fato do fibonacci heap ser formado pelo conjunto das arborescências. Em (b) é ilustrada a representação em forma de lista, mostrando os ponteiros da estrutura.

A lista Q pode ser dividida em dois tipos de listas: (1) (*root*) uma lista formada pelas raízes das arborescências; e (2) (*child*) lista formada por filhos de um mesmo vértice. Cada vértice num fibonacci heap contém sete campos, que são: ponteiros *pai*, *filho*, *direita*, *esquerda*, que representam listas duplamente ligadas circulares (lista *root* e *child*); *grau_marca*, que guarda o número $2 * grau + marca$, onde o *grau* é o grau do vértice e *marca* tem valor 1 se o vértice (exceto o raiz) já perdeu algum filho, e 0 caso contrário. Os campos *predecessor* (ψ) e *dist* (d)

são os mesmos utilizados anteriormente. Note que a estrutura Vertex do SGB tem seis campos utilitários. Então, para implementar o fibonacci heap foi redefinida a estrutura **Vertex** do SGB para sete campos.

40 ⟨ Definições 12 ⟩ +≡
#define *pai* *t.V*
#define *filho* *w.V*
#define *direita* *y.V*
#define *esquerda* *z.V*
#define *grau_marca* *x.I*

As funções a seguir são responsáveis por inserir e remover vértices de uma lista. A função *insere_na_lista* insere na lista *l* o vértice *v* e a função *remove_da_lista* remove o vértice *v* da lista.

41 ⟨ Funções auxiliares 32 ⟩ +≡

```
void insere_na_lista(l, v)
     Vertex *l;
     Vertex *v;
{
  v->direita = l->direita;
  (l->direita)->esquerda = v;
  v->esquerda = l;
  l->direita = v;
}
void remove_da_lista(v)
     Vertex *v;
{
  (v->esquerda)->direita = v->direita;
  (v->direita)->esquerda = v->esquerda;
}
```

Para criar o fibonacci heap *Q* é utilizada a função *create_fheap*.

42 ⟨ Filas de prioridade 11 ⟩ +≡

```
void create_fheap(g)
     Graph *g;
{
  register Vertex *v;
```

```
    Q = Λ;
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      v→direita = v→esquerda = v;
      v→grau_marca = 0;
      v→pai = v→filho = Λ;
    }
  }
```

As implementações das operações insert, delete-min e decrease-key são representadas pelas funções *insert_fheap*, *delete_min_fheap* e *decrease_key_fheap*.

43 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_fheap(Vertex *v);
Vertex *delete_min_fheap();
void decrease_key_fheap(Vertex *v);
```

Inserir um vértice no fibonacci heap significa colocá-lo na lista *root*. No pior caso, essa operação gasta tempo $O(1)$.

44 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_fheap(v)
    Vertex *v;
{
  if (Q ≡ Λ) {
    Q = v;
  }
  else {
    insere_na_lista(Q, v);      /* insere v na lista root */
    if (v→dist < Q→dist) Q = v;
  }
}
```

O vértice com o menor potencial no fibonacci heap se encontra na primeira posição, ou seja, Q aponta para o vértice com potencial mínimo. Remover esse vértice implica em adicionar seus filhos à lista *root*, e em seguida é necessário utilizar o bloco ⟨ Consolidate 46 ⟩, para atualizar o número de arborescências e encontrar o novo vértice com potencial mínimo. Essa operação gasta tempo amortizado $O(\log qsize)$.

45 ⟨ Filas de prioridade 11 ⟩ +≡

```
Vertex *delete_min_fheap()
{
  register Vertex *min, *f, *v, *x, *y, *aux;
  register Vertex **A;
  register long i, d, Dn;
  min = Q;
  if (min ≠ Λ) {
    f = min→filho;
    while (f ≠ Λ) {     /* se min tem filhos */     /* para cada filho v de min */
      v = f;
      f = f→direita;
      if (v ≡ f) f = Λ;
      remove_da_lista(v);     /* remove v da lista child */
      insere_na_lista(Q, v);     /* insere v na lista root */
      v→pai = Λ;
    }
    remove_da_lista(min);     /* remove min da lista root */
    if (min→direita ≡ min) Q = Λ;
    else {
      Q = Q→direita;
      ⟨ Consolidate 46 ⟩
    }
  }
  return min;
}
```

O bloco ⟨ Consolidate 46 ⟩ é responsável por reduzir o número de raízes de Q, ou seja, reduzir o número de arborescências. Ele consiste em executar repetidamente os seguintes passos, até que todo vértice da lista *root* tenha graus distintos:

(1) encontre duas raízes x e y na lista *root* com o mesmo grau. Supõe-se que x→*dist* $\leq$ y→*dist*; e

(2) junte (link) y a x, removendo y da lista *root*, e adicionando y como um filho de x.

A operação (2) é feita em ⟨ Link arborescência y com a arborescência x 47 ⟩

46 ⟨ Consolidate 46 ⟩ ≡

```
Dn = 1 + 8 * sizeof(long);     /* 1+log(n): número máximo de arborescências */
```

```
if ((A = (Vertex **) malloc(Dn * sizeof(Vertex *))) ≡ Λ) {
  printf("%s\n", err_message[ERROR_2]);
  exit(0);
}
for (i = 0; i < Dn; i++)        /* A[d] guarda vértices de grau d */
  A[i] = Λ;
v = Q;
do {
  x = v;
  v = v→direita;
  if (x ≡ v) v = Λ;
  remove_da_lista(x);       /* remove x da lista root */
  d = (int) x→grau_marca/2;      /* 2 * grau + marca */
  while ((d < Dn) ∧ (A[d] ≠ Λ)) {
    y = A[d];
    if (x→dist > y→dist) {
      aux = x;
      x = y;
      y = aux;
    }
    ⟨Link arborescência y com a arborescência x 47⟩
    A[d] = Λ;
    d++;
  }
  A[d] = x→direita = x→esquerda = x;
} while (v ≠ Λ);
Q = Λ;
for (i = 0; i < Dn; i++) {
  if (A[i] ≠ Λ) {
    if (Q ≡ Λ) Q = A[i]→direita = A[i]→esquerda = A[i];
    else {
      insere_na_lista(Q, A[i]);      /* adiciona A[i] na lista de root */
      if (A[i]→dist < Q→dist) Q = A[i];
    }
  }
}
free(A);
```

Este código é citado nos blocos 45 e 46.

Este código é usado no bloco 45.

47 ⟨ Link arborescência y com a arborescência x 47 ⟩ ≡

```
remove_da_lista(y);      /* remove y da lista root */
f = x→filho;
if (f) {
  insere_na_lista(f, y);      /* insere y como filho de x */
}
else x→filho = y→esquerda = y→direita = y;
y→pai = x;
x→grau_marca += 2;
y→grau_marca = (y→grau_marca >> 1) << 1;      /* parte inteira da divisão por 2 e
    depois multiplicada por 2 */
```

Este código é citado no bloco 46.

Este código é usado no bloco 46.

Diminuir o potencial de um vértice v, ao visitá-lo, pode tornar necessário reposicioná-lo em Q, de modo que as propriedades 4.7 e 4.8 continuem sendo respeitadas. Isso é feito chamando-se ⟨ Remove v da lista de filhos de p 49 ⟩, onde p é o pai de v. Caso p esteja perdendo um segundo filho, ainda é necessário chamar a função *cascading_cut* para garantir a propriedade 4.8. Essa operação gasta tempo amortizado $O(1)$.

48 ⟨ Filas de prioridade 11 ⟩ +≡

```
void decrease_key_fheap(v)
    Vertex *v;
{
  register Vertex *p;
  p = v→pai;
  if ((p ≠ Λ) ∧ (v→dist < p→dist)) {
    ⟨ Remove v da lista de filhos de p 49 ⟩
    cascading_cut(p);
  }
  if (v→dist < Q→dist) Q = v;
}
```

49 ⟨ Remove v da lista de filhos de p 49 ⟩ ≡

```
if (v→direita ≡ v) p→filho = Λ;
```

```
else {
  if (p→filho ≡ v) p→filho = (p→filho)→direita;
  remove_da_lista(v);        /* remove v da lista de filhos de p */
}    /* decrementa o grau de p */
p→grau_marca −= 2;
insere_na_lista(Q, v);      /* adiciona v na lista root */
v→pai = Λ;
v→grau_marca = (v→grau_marca ≫ 1) ≪ 1;      /* parte inteira da divisão por 2 e
      depois multiplicada por 2 */
```

Este código é citado nos blocos 48 e 50.

Este código é usado nos blocos 48 e 50.

A função *cascading_cut* repete o processo de ⟨ Remove v da lista de filhos de p 49 ⟩ até encontrar um vértice que ainda não tenha perdido nenhum filho.

50 ⟨ Funções auxiliares 32 ⟩ +≡

```
void cascading_cut(v)
    Vertex *v;
{
  register Vertex *p;
  p = v→pai;
  if (p ≠ Λ) {
    if (v→grau_marca % 2 ≡ 0) v→grau_marca++;
    else {
      ⟨ Remove v da lista de filhos de p 49 ⟩
      cascading_cut(p);
    }
  }
}
```

Lema 4.9 (operações fibonacci heap): *Na implementação da fila de prioridade Q, utilizando um fibonacci heap, as operações insert e decrease-key gastam tempo amortizado $O(1)$ e a operação delete-min gasta tempo amortizado $O(\log n)$.* ∎

Portanto, do teorema 3.5 e do lema 4.9, o tempo gasto pelo algoritmo de Dijkstra utilizando um fibonacci heap, é

$$\underbrace{O(n)}_{\textsf{insert}} + \underbrace{O(m)}_{\textsf{decrease-key}} + \underbrace{O(n \log n)}_{\textsf{delete-min}} = O(m + n \log n).$$

Teorema 4.10: *A implementação do algoritmo de Dijkstra que utiliza um fibonacci heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m + n \log n)$.* ∎

4.4 Bucket heap

Um **bucket heap** é uma estrutura de dados que mantêm uma seqüência de vértices num vetor Q de listas ligadas. Para cada posição k do vetor, $Q(k)$ é uma lista ligada de vértices. Em cada $Q(k)$, são mantidos os vértices com potencial igual a k[2].

Observe que nC, onde C é o comprimento do maior arco, é um limitante superior para qualquer distância no grafo. Portanto, são necessários no máximo $nC + 1$ buckets, já que as distâncias podem variar de 0 a nC. Porém, é possível diminuir o número de buckets para $C + 1$. Observe que: (1) $d(u) \leq d(v)$ para cada vértice u em S e v em Q (invariante (dk3)); e (2) para cada vértice w em Q, $d(w) = d(u) + c(u, w)$ para algum vértice u em S (invariantes (dk4) e (dk12)). Portanto, de (1) e (2), $d(w) = d(u) + c(u, w) \leq d(u) + C$, onde u é o vértice escolhido no caso 2 do algoritmo de Dijkstra. Em outras palavras, toda distância tentativa é limitada inferiormente por $d(u)$, e superiormente por $d(u) + C$. Conseqüentemente, $C + 1$ buckets são suficientes. Resultando na seguinte propriedade.

Propriedade 4.11 (ordem): *Cada bucket $Q(k \mod (C + 1))$ mantêm os vértices com potencial igual a k, onde C é o comprimento do maior arco.*

Os $C + 1$ buckets, numerados de $[0..C]$, simulam as $nC + 1$ posições funcionando de maneira circular. A figura 4.4 ilustra a organização dos buckets.

O ponteiro *pos_corrente* indica a posição do bucket que está sendo visitada.

52 ⟨ Definições 12 ⟩ +≡
#define *pos_corrente* $w.I$

[2]Note que a ordem entre os vértices em $Q(k)$ não é importante

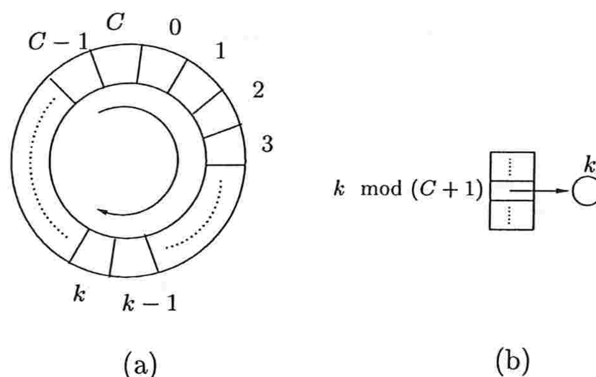


Figura 4.4: A figura (a) enfatiza a organização circular dos buckets. (b) mostra que o bucket k mod $(C+1)$ guarda o vértice com potencial k.

Para criar o bucket heap Q é utilizada a função *create_bkheap*. Redefinimos $Q(k)$ (a definição anterior está seção 4.1), de modo que seja o cabeça de lista do bucket k.

```
53 ⟨Filas de prioridade 11⟩ +≡
#undef Q
#define Q(k) (Q + k)
  void create_bkheap(g)
      Graph *g;
  {
    register unsigned long i;
    register Vertex *v;

    if ((Q = (Vertex *) malloc((C + 1) * sizeof(Vertex))) ≡ Λ) {
      printf("%s\n", err_message[ERROR_2]);
      exit(0);
    }
    for (i = 0; i < C + 1; i++) {
      Q(i)→direita = Q(i)→esquerda = Q(i);      /* endereço Q + i */
    }
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      v→direita = v→esquerda = v;
    }
    Q→pos_corrente = 0;
  }
```

As implementações das operações insert, delete-min e decrease-key são representadas pelas funções *insert_ bkheap, delete_ min_ bkheap* e *decrease_ key_ bkheap*.

```
54 ⟨Filas de prioridade 11⟩ +≡
  void insert_bkheap(Vertex *v);
  Vertex *delete_min_bkheap();
  void decrease_key_bkheap(Vertex *v);
```

Como a implementação é feita usando $C+1$ buckets, cada vértice v é inserido na posição $v\text{→}dist \mod (C+1)$. Essa operação gasta tempo $O(1)$.

```
55 ⟨Filas de prioridade 11⟩ +≡
  void insert_bkheap(v)
      Vertex *v;
  {
    register unsigned long k;
    k = v→dist % (C + 1);
    insere_na_lista(Q(k), v);
  }
```

O vértice com o menor potencial pode ser encontrado na primeira posição não-vazia de Q. No pior caso, essa operação gasta tempo $O(C)$.

```
56 ⟨Filas de prioridade 11⟩ +≡
  Vertex *delete_min_bkheap()
  {
    register unsigned long k;
    Vertex *u;      /* procura o primeiro bucket não-vazio */
    for (k = Q→pos_corrente; Q(k) ≡ Q(k)→direita; k = ++k % (C + 1)) ;
    Q→pos_corrente = k;
    u = Q(k)→direita;
    remove_da_lista(u);
    return u;
  }
```

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reinseri-lo em Q, de modo que a propriedade 4.11 continue sendo respeitada. Essa operação gasta tempo $O(1)$.

57 ⟨ Filas de prioridade 11 ⟩ +≡

```
void decrease_key_bkheap(v)
    Vertex *v;
{
  remove_da_lista(v);
  insert_bkheap(v);
}
```

Lema 4.12 (operações bucket heap): *Na implementação da fila de prioridade Q utilizando um bucket heap, as operações* insert *e* decrease-key *gastam tempo $O(1)$ e* delete-min *gasta tempo $O(C)$.* ∎

Portanto, do teorema 3.5 e do lema 4.12, o tempo gasto pelo algoritmo de Dijkstra utilizando um bucket heap, é

$$\underbrace{O(n)}_{\text{insert}} + \underbrace{O(m)}_{\text{decrease-key}} + \underbrace{O(nC)}_{\text{delete-min}} = O(m + nC).$$

Teorema 4.13: *A implementação do algoritmo de Dijkstra que utiliza um bucket heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m + nC)$, onde C é o maior comprimento de um arco.* ∎

4.5 Radix heap

Um **radix heap**, assim como um bucket heap (seção 4.4), é uma estrutura de dados que mantêm uma seqüência de vértices, num vetor Q, onde para cada posição k, $Q(k)$ é uma lista ligada de vértices. Porém, em vez de manter somente vértices com potencial k na $(k mod(C+1))$-ésima posição do bucket, são mantidos os vértices com potencial em um determinado intervalo. Na implementação do radix heap, os intervalos são consecutivos e têm largura $1, 1, 2, 4, 8, 16, \ldots$. Logo,

$$\begin{aligned}
range(0) &= [0] \\
range(1) &= [1] \\
range(2) &= [2..3] \\
range(3) &= [4..7] \\
range(4) &= [8..15] \\
range(K) &= [2^{K-1}..2^K - 1]
\end{aligned}$$

onde $K = \lceil \log(nC) \rceil$ e $range(k)$ é o intervalo do bucket $Q(k)$. Portanto, o número de buckets necessários é $1 + K$.

Propriedade 4.14 (ordem): *Se $range(k)$ é o intervalo do bucket $Q(k)$, então $Q(k)$ mantêm os vértices com potencial em $range(k)$.*

Os intervalos dos buckets mudam dinamicamente durante a execução do algoritmo, rearranjando-se de forma que os vértices com menor potencial fiquem nos buckets de largura 1. Por exemplo, supondo que o primeiro bucket não-vazio é o $Q(4)$, inicialmente, seu respectivo intervalo é [8..15], que é maior do que 1. Então, é necessário fazer a redistribuição dos intervalos da seguinte maneira.

$$range(0) = [8]$$
$$range(1) = [9]$$
$$range(2) = [10..11]$$
$$range(3) = [12..15]$$
$$range(4) = \emptyset$$

E também, é necessário redistribuir os vértices pertencentes a $Q(4)$ nos buckets $Q(0)$, $Q(1)$, $Q(2)$ e $Q(3)$, assim os vértices que podem ser examinados estão sempre em $Q(0)$ e $Q(1)$.

Na implementação, é suficiente guardar apenas o valor inicial do intervalo. Esse valor é mantido em *range* e cada vértice v será inserido no bucket $Q(bucket)$.

```
59 ⟨Definições 12⟩ +≡
#define range w.I
#define bucket x.I
```

A função *log2*(x) calcula o $\lfloor \log_2 x \rfloor$ para $x \geq 1$.

```
60 ⟨Funções auxiliares 32⟩ +≡
int log2(x)
    unsigned long x;
{
  register int lg = −1;
  do {
    x = x ≫ 1;
    lg++;
  } while (x);
  return lg;
}
```

Para criar o radix heap Q é utilizada a função *create_rxheap*.

```
61 ⟨ Filas de prioridade 11 ⟩ +≡
void create_rxheap(g)
    Graph *g;
{
  register int i, K, range_len;
  register Vertex *v;
  K = log2(infinito) + 1;      /* número máximo de buckets */
  if ((Q = (Vertex *) malloc((1 + K) * sizeof(Vertex))) ≡ Λ) {
    printf("%s\n", err_message[ERROR_2]);
    exit(0);
  }
  Q(0)→direita = Q(0)→esquerda = Q(0);
  Q(0)→range = 0;
  for (i = range_len = 1; i ≤ K; i++) {
    Q(i)→direita = Q(i)→esquerda = Q(i);
    Q(i)→range = Q(i − 1)→range + range_len;
    if (i ≠ 1) range_len <<= 1;      /* range_len = range_len * 2 */
  }
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    v→direita = v→esquerda = v;
    v→bucket = K;      /* indica que v está em Q(K) */
  }
}
```

As implementações das operações insert, delete-min e decrease-key são representadas pelas funções *insert_rxheap, delete_min_rxheap* e *decrease_key_rxheap.*

```
62 ⟨ Filas de prioridade 11 ⟩ +≡
void insert_rxheap(Vertex *v);
Vertex *delete_min_rxheap( );
void decrease_key_rxheap(Vertex *v);
```

A inserção dos vértices é feita respeitando-se os intervalos. No pior caso, essa operação gasta tempo $O(\log(nC))$.

```
63 ⟨ Filas de prioridade 11 ⟩ +≡
void insert_rxheap(v)
```

```
    Vertex *v;
  {
    register unsigned long k;
    for (k = v->bucket; k > 0; k--) {      /* encontra o bucket para inserir v */
      if (v->dist >= Q(k)->range) break;
    }
    v->bucket = k;
    insere_na_lista(Q(k), v);
  }
```

O vértice com o menor potencial é encontrado em $Q(0)$ ou $Q(1)$. Caso não haja vértices nessas posições é necessário fazer a redistribuição dos intervalos, e reinserir os vértices nos buckets corretos. No pior caso, essa operação gasta tempo $O(\log(nC))$.

64 ⟨Filas de prioridade 11⟩ +≡

```
  Vertex *delete_min_rxheap()
  {
    register unsigned long range_len, k, i, min;
    register Vertex *u;
    if ((Q(0) ≡ Q(0)->direita) ∧ (Q(1) ≡ Q(1)->direita)) {
      ⟨Encontra a posição k do primeiro bucket não-vazio 65⟩
      ⟨Encontra o valor do menor potencial em Q(k) 66⟩
      ⟨Redistribui os intervalos 67⟩
    }
    k = (Q(0) ≡ Q(0)->direita) ? 1 : 0;
    u = Q(k)->direita;
    remove_da_lista(u);
    return u;
  }
```

65 ⟨Encontra a posição k do primeiro bucket não-vazio 65⟩ ≡

```
  for (k = 2; Q(k)->direita ≡ Q(k); k++) ;
```

Este código é usado no bloco 64.

66 ⟨Encontra o valor do menor potencial em $Q(k)$ 66⟩ ≡

```
  for (u = Q(k)->direita, min = infinito; u ≠ Q(k); u = u->direita)
    if (u->dist < min) min = u->dist;
```

Este código é usado no bloco 64.

67 ⟨ Redistribui os intervalos 67 ⟩ ≡

```
Q(0)->range = min;
for (range_len = i = 1; i ≤ k - 1; i++) {
  Q(i)->range = Q(i - 1)->range + range_len;
  if (i ≠ 1) range_len <<= 1;
}    /* o k-ésimo intervalo fica vazio */
Q(k)->range = infinito;
⟨ Remove e distribui os vértices de Q(k) nos buckets anteriores 68 ⟩
```

Este código é usado no bloco 64.

68 ⟨ Remove e distribui os vértices de $Q(k)$ nos buckets anteriores 68 ⟩ ≡

```
for (u = Q(k)->direita; Q(k) ≠ u; u = Q(k)->direita) {
  remove_da_lista(u);
  i = u->dist - Q(0)->range;
  u->bucket = (i ≡ 0) ? 0 : log2(i) + 1;
  insert_rxheap(u);
}
```

Este código é usado no bloco 67.

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reinseri-lo em Q, de modo que a propriedade 4.14 continue sendo respeitada. Essa operação gasta tempo $O(1)$, pois o tempo gasto por *insert_rxheap*, já inclui as reinserções dos vértices.

69 ⟨ Filas de prioridade 11 ⟩ +≡

```
void decrease_key_rxheap(v)
    Vertex *v;
{
  remove_da_lista(v);
  insert_rxheap(v);
}
```

Lema 4.15 (operações radix heap): *Na implementação da fila de prioridade Q, utilizando um radix heap, as operações insert e delete-min gastam tempo $O(\log(nC))$ e a operação decrease-key gasta tempo $O(1)$.* ∎

Portanto, do teorema 3.5 e do lema 4.15, o tempo gasto pelo algoritmo de Dijkstra utilizando um radix heap, é

$$\underbrace{O(n\log(nC))}_{\text{insert}} + \underbrace{O(m)}_{\text{decrease-key}} + \underbrace{O(n\log(nC))}_{\text{delete-min}} = O(m + n\log(nC)).$$

Teorema 4.16: *A implementação do algoritmo de Dijkstra que utiliza um radix heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m + n\log(nC))$, onde C é o maior comprimento de um arco.* ∎

4.6 Eficiência

A figura 4.5 resume o que foi visto neste capítulo, com relação aos tempos gasto para cada implementação de Q. No fibonacci heap, o tempo gasto é amortizado.

	heap	D-heap	fibonacci heap	bucket heap	radix heap
insert	$O(\log n)$	$O(\log_D n)$	$O(1)$	$O(1)$	$O(\log(nC))$
delete-min	$O(\log n)$	$O(D\log_D n)$	$O(\log n)$	$O(C)$	$O(\log(nC))$
decrease-key	$O(\log n)$	$O(\log_D n)$	$O(1)$	$O(1)$	$O(1)$
Dijkstra	$O(m\log n)$	$O(m\log_D n)$	$O(m + n\log n)$	$O(m + nC)$	$O(m + n\log(nC))$

Figura 4.5: Eficiência das implementações de uma fila de prioridade.

CAPÍTULO 5

Algoritmo de Dinitz-Thorup

O algoritmo apresentado neste capítulo é um primeiro passo para o projeto de um algoritmo linear, no modelo RAM, para o problema do caminho mínimo (PCM) restrito a grafos simétricos com funções comprimento simétricas.

PCM

Um componente fundamental no algoritmo de Dijkstra é a escolha apropriada do próximo vértice a ser examinado: escolha um vértice u em um certo conjunto Q de vértices visitados tal que o potencial ou distância tentativa $d(u)$ associada a u é mínimo. Devido a esta escolha, tem-se que os vértices do grafo são examinados em ordem crescente de distância a partir de um dado vértice origem[1] e que qualquer implementação do algoritmo de Dijkstra, no modelo comparação-adição, faz $\Omega(n \log n)$ comparações (seção 3.5). Assim, qualquer variante do algoritmo de Dijkstra que tem a pretensão de executar um número de operações proporcional ao tamanho do grafo não pode ser tão seletiva em relação ao próximo vértice a ser examinado.

Dinitz [12] observou que se δ é o menor comprimento de um arco então, no algoritmo de Dijkstra, pode-se escolher o próximo vértice a ser examinado da seguinte maneira: escolha u em Q tal que

$$\min\{d(v) : v \in Q\} \leq d(u) \leq \min\{d(v) : v \in Q\} + \delta.$$

Dinitz combinou esta observação a uma partição (bucketing) dos vértices do grafo a fim de determinar um conjunto de vértices que podem ser examinados em qualquer ordem: os vértices com potenciais em[2] $[\min\{d(v) : v \in Q\} .. \min\{d(v) : v \in Q\} + \delta]$.

O algoritmo que nesta dissertação é chamado de algoritmo de Dinitz-Thorup foi apresentado por Thorup [39]. Este algoritmo combina dois ingredientes, a saber, a idéia de bucketing de Dinitz para determinar vértices que podem ser examinados em qualquer ordem e uma certa decomposição do grafo proposta por Thorup, descrita na próxima seção. Este algoritmo é um estágio intermediário entre o algoritmo de Dijkstra (capítulo 3) e algoritmo de Thorup, estudado no próximo capítulo.

[1] Esta ordem é conseqüência do invariante da monotonicidade (dk3) do algoritmo de Dijkstra.

[2] Para $\delta = 0$ a observação de Dinitz coincide com a implementação do algoritmo de Dijkstra.

5.1 Partições, variante do PCM e elementos maduros

Seja (V, A) um grafo, c uma função comprimento de A em $\mathbb{Z}_{\geq}$ e δ um número inteiro. Uma partição $\mathcal{P}$ de V é uma **δ-partição** (em relação a c) se todo arco com ponta inicial e ponta final em elementos distintos de $\mathcal{P}$ tem comprimento pelo menos δ (figura 5.1).

-partição

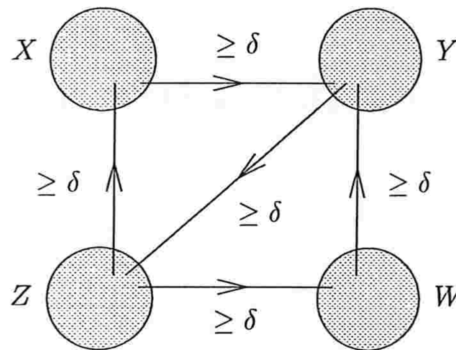


Figura 5.1: Ilustração de uma δ-partição $\mathcal{P} = \{X, Y, Z, W\}$.

O algoritmo descrito na próxima seção resolve a seguinte variante do problema do caminho mínimo:

PCMV

> **Problema** PCMV$(V, A, c, \delta, \mathcal{P}, s)$: Dado um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$, um inteiro positivo δ, uma δ-partição $\mathcal{P}$ e um vértice s, encontrar um caminho de comprimento mínimo de s até t, para cada vértice t em V.

Um algoritmo para o PCMV pode ser facilmente adaptado para resolver o PCM: basta tomar $\mathcal{P} := \{\{v\} : v \in V\}$ e $\delta := \min\{c(u, v) : uv \in A\}$.

Vale a pena chamar a atenção para o seguinte fato. Do ponto de vista da definição do problema faz sentido permitir $\delta = 0$. Entretanto, os algoritmos e implementações neste e no próximo capítulo utilizam δ como denominador de expressões. O tipo de divisão por δ utilizada se reduz a um simples shift, que no modelo RAM, consome tempo constante.

Para resolver o PCMV o algoritmo de Dinitz-Thorup, como o algoritmo de Dijkstra, mantém, entre outros objetos, uma função potencial d e o conjunto Q dos vértices visitados. É dito que um elemento X da δ-partição dada é **maduro** se $Q \cap X \neq \emptyset$ e

$$\min\{d(v) : v \in Q\} \leq \min\{d(v) : v \in Q \cap X\} \leq \min\{d(v) : v \in Q\} + \delta.$$

A primeira desigualdade acima é óbvia. A segunda desigualdade é, de fato, a condição.

5.2 Descrição

O algoritmo de Dinitz-Thorup resolve o PCMV e devolve os mesmos certificados que o algoritmo de Dijkstra, a saber, uma função predecessor codificando os caminhos compactamente e uma função potencial viável atestando a minimalidade dos caminhos e a eventual não-acessibilidade de alguns vértices.

Algoritmo de Dinitz-Thorup. Recebe um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$, um inteiro positivo δ, uma δ-partição $\mathcal{P}$ e um vértice s, e devolve uma função predecessor ψ e uma função potencial d que respeita c tais que, para todo vértice t, se t é acessível a partir de s, então ψ determina um caminho de s a t que tem comprimento $d(t) - d(s)$, caso contrário $d(t) - d(s) = nC + 1$, onde $C := \max\{c(u,v) : uv \in A\}$.

Cada iteração começa com uma função potencial d, uma função predecessor ψ, partes S, Q e U de V.

No início da primeira iteração $d(s) = 0$ e $d(v) = nC + 1$ para cada vértice v distinto de s, $\psi(v) = v$ para cada vértice v, $S = \emptyset$, $Q = \{s\}$, $U = V \setminus \{s\}$.

Cada iteração consiste em:

Caso 1: $Q = \emptyset$.

Devolva d e ψ e pare.

Caso 2: $Q \neq \emptyset$

Escolha X em $\mathcal{P}$ tal que X é maduro.

Escolha u em $Q \cap X$ tal que $d(u)$ é mínimo.

$S' := S \cup \{u\}$.

$Q' := Q \setminus \{u\}$.

$U' := U$.

Para cada v faça $d'(v) := d(v)$ e $\psi'(v) := \psi(v)$.

Para cada arco uv faça

Se $d(v) > d(u) + c(u,v)$ então

$d'(v) := d(u) + c(u,v)$, $\psi'(v) := u$ e remova[3] v de U' e acrescente a Q'.

Comece nova iteração com d', ψ', S', Q' e U' nos papéis de d, ψ, S, Q e U. □

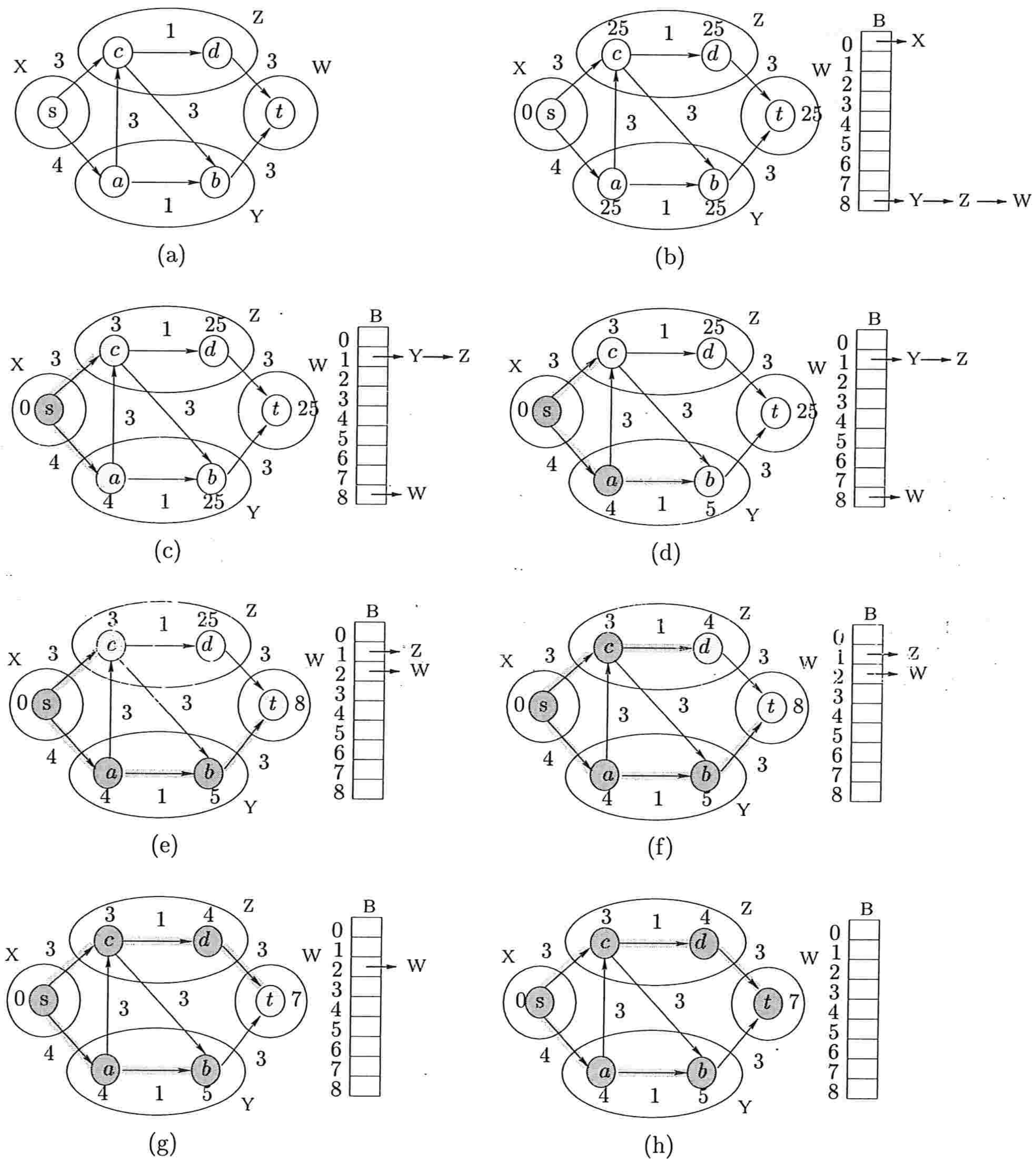


Figura 5.2: Execução do algoritmo de Dinitz-Thorup. (a) exibe um grafo com comprimentos nos arcos junto com uma 3-partição cujos elementos são os conjuntos X, Y, W, Z da figura. O vértice inicial é s. (b) mostra a situação no início da primeira iteração. Se um arco uv está sombreado, então $\psi(v) = u$. Os potenciais são os números próximos a cada vértice. Os vértices pretos são os de S, os vértices cinzas são os de Q, e os vértices brancos são os de U. (c) – (g) exibem a situação após cada iteração do caso 2. Os valores finais da função potencial d, e da função predecessor ψ, são mostrados em (h).

Se $\delta = 0$ o algoritmo de Dinitz-Thorup, em sua forma abstrata, examina os vértices na mesma ordem crescente de distância que o algoritmo de Dijkstra. A situação de interesse é quando esta ordenação implícita, pela distância a partir da origem s, pode ser evitada, e isto ocorre quando δ é um inteiro positivo (quanto maior, melhor). Uma simulação do algoritmo de Dinitz-Thorup está ilustrada na figura 5.2, onde no momento, os vetores ao lado dos grafos devem ser ignorados.

5.3 Invariantes

O algoritmo de Dinitz-Thorup mantém todos os invariantes do algoritmo de Dijkstra (seção 3.2), exceto aquele que é o responsável pela ordem que os vértices são examinados, o invariante da monotonicidade (dk3).

No lugar do invariante da monotonicidade o algoritmo de Dinitz-Thorup mantém os dois invariantes a seguir, onde o primeiro envolve a partição $\mathcal{P}$ e o segundo envolve o número δ.

(dt1) (monotonicidade local) para cada parte X de V em $\mathcal{P}$ e para cada u em $S \cap X$, v em $Q \cap X$ e w em $U \cap X$ vale que[4]

$$d(u) \leq d(v) < d(w) = nC + 1.$$

(dt2) (monotonicidade relaxada) para cada u em S, v em Q e w em U vale que

$$d(u) \leq d(v) + \delta < d(w) = nC + 1.$$

Os invariantes estão ilustrados na figura 5.3

O algoritmo mantém os invariantes (dk1) e (dk2) de estrutura, os invariantes (dk4), (dk5) e (dk6) envolvendo a função predecessor e $\{S, Q, U\}$, os invariantes de (dk7) a (dk11) dos arcos onde a função potencial respeita ou desrespeita c e o invariante (dk12) das folgas complementares.

A seguir estão as demonstrações de (dt1) e (dt2) e uma nova demonstração de (dk9), já que a demonstração de (dk9), apresentada no capítulo 3, faz uso do invariante da monotonicidade, que no, algoritmo de Dinitz-Thorup, foi substituido pelos invariantes (dt1) e (dt2).

Demonstração de (dt1): Considere uma iteração em que ocorre o caso 2. Ao final da iteração tem-se que $d'(v) \neq d(v)$ se e somente se v está em $Q' \subseteq (Q \setminus \{u\}) \cup U$ e uv é um arco do grafo com $d(v) - d(u) > c(u, v)$. Ademais, se $d'(v) \neq d(v)$ então $d'(v) = d(u) + c(u, v) < d(v)$.

Do invariante (dt1) sabe-se que para cada x em $S \cap X$, y em $Q \cap X$ e z em $U \cap X$ vale que $d(x) \leq d(y) < d(z) = nC + 1$. Assim, pela escolha de X e u (e como c é uma função de A em

[3]É possível que v já pertença a Q' e não esteja em U'. Nesse caso estas últimas duas instruções são redundantes.
[4]Se alguma das intersecções é vazia, considere apenas as desigualdades que fazem sentido.

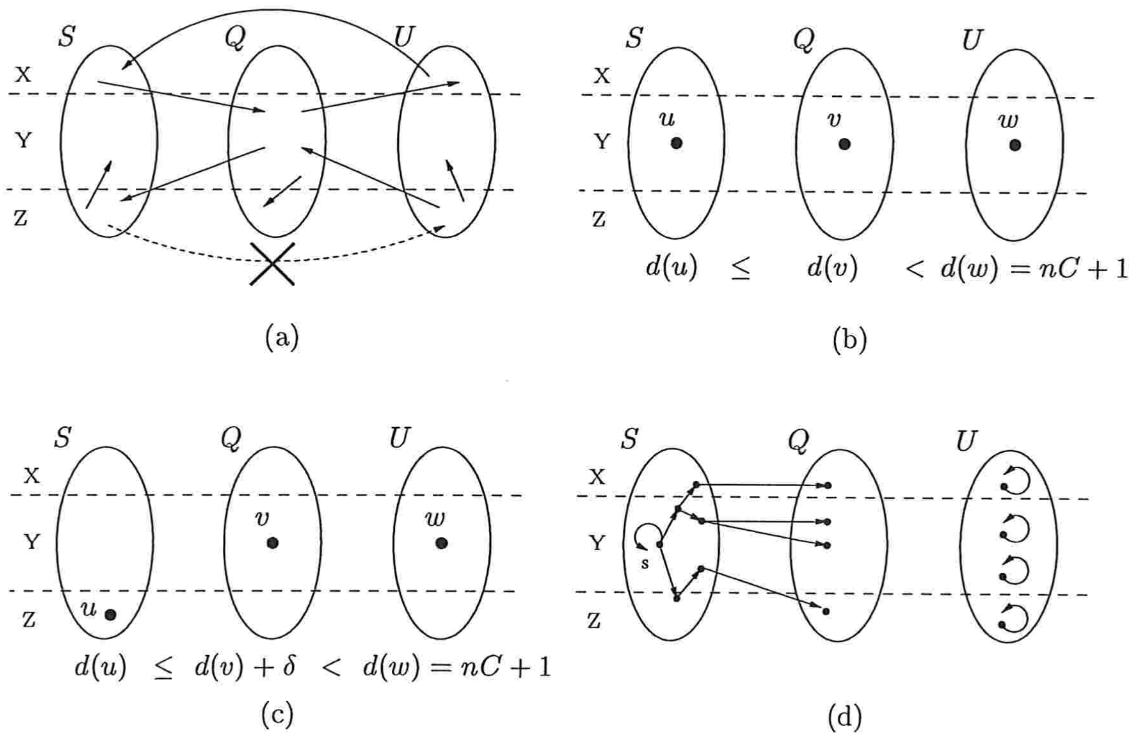


Figura 5.3: Ilustração dos invariantes. X, Y e Z são os elementos da δ-partição. Figura (a) mostra a estrutura do grafo em relação à partição S, Q e U de V (invariantes (dk1) e (dk2)), onde os arcos que cruzam as linhas tracejadas têm comprimento pelo menos δ. (b) e (c) exibem a relação de ordem envolvendo os potenciais dos vértices, as partes S, Q e U bem como os elementos X, Y e Z da δ-partição (invariantes (dt1) e (dt2)). Em (d) os arcos determinados pela função predecessor ψ são exibidos (invariantes (dk4), (dk5) e (dk6)).

$\mathbb{Z}_{\geq}$), após examinar o vértice u tem-se que

$$d'(x) \leq d(u) \leq d'(y) < d'(z) = nC + 1$$

para cada x em $S' \cap X = (S \cup \{u\}) \cap X$, y em $Q' \cap X$ e z em $U' \cap X$.

Seja agora Y um elemento em $\mathcal{P}$ distinto de X. Sejam x um vértice em $S' \cap Y = (S \cup \{u\}) \cap Y$, y um vértice em $Q' \cap Y$ e z um vértice em $U' \cap Y$. Se $d'(y) = d(y)$, então pelo invariante (dt1) vale que

$$d'(x) = d(x) \leq d(y) = d'(y) < d'(z) = nC + 1.$$

Suponha que $d'(y) < d(y)$ e nesse caso $d'(y) = d(u) + c(u, y) \geq d(u) + \delta$. Do invariante (dt2) sabe-se que $d(x) \leq d(u) + \delta$ e portanto,

$$d'(x) = d(x) \leq d(u) + \delta \leq d'(y) < d'(z) = nC + 1.$$

O que conclui a demonstração do invariante (dt1). ∎

Demonstração de (dt2): Considere uma iteração em que ocorre o caso 2. É claro que ao final da iteração vale que $d'(w) = nC + 1$, para cada w em U'. Além disso, combinando os invariantes (dk4), (dk5) e (dk6) que envolvem a função predecessor e $\{S, Q, U\}$ e o invariante (dk12) das folgas complementares, obtem-se que

$$d'(v) + \delta \leq d(v) + \delta \leq (n-1)C + \delta < nC + 1 = d'(w),$$

para cada v em Q' e w em U'. Assim, nos concentraremos em verificar que para cada x em S' e y em Q', ao final da iteração, vale que $d'(x) \leq d'(y) + \delta$.

Seja x um vértice em S' e y um vértice em Q'. Se ao final da iteração, $d'(y) = d(y)$, então pela escolha de X e u e pelo invariante (dt2) tem-se que

$$d'(x) = d(x) \leq d(y) + \delta = d'(y) + \delta.$$

Assim, suponha que, ao final da iteração, vale que $d'(y) < d(y)$. Portanto, $d(u) \leq d(u) + c(u, y) = d'(y)$. Do invariante (dt2), tem-se que $d(x) \leq d(u) + \delta$. Portanto,

$$d'(x) = d(x) \leq d(u) + \delta \leq d'(y) + \delta.$$

∎

Demonstração de (dk9): Novamente, considere uma iteração em que ocorre o caso 2. Do invariante (dt1), obtem-se que no final da iteração vale que $d'(x) - d'(y) \leq 0$ para cada x em $S' \cap Y = (S \cup \{u\}) \cap Y$, y em $Q' \cap Y$, onde Y é um elemento de $\mathcal{P}$. Ademais, do invariante (dt2), sabe-se que $d'(x) - d'(y) \leq \delta$ para cada x em $S' \cap Y$ e y em $Q' \cap Z$, onde Y e Z são

elementos distintos de $\mathcal{P}$. Portanto, como os comprimentos dos arcos são não-negativos e $\mathcal{P}$ é uma δ-partição, d' respeita c em $A(Q', S')$.

O processo de examinar u faz com que d' respeite c em cada arco com ponta inicial em u. Do invariante (dk9) sabe-se que d respeita c em $A(S, Q)$. Assim, como $d'(v) = d(v)$ para cada v em S' e $d'(v) \leq d(v)$ para cada v em Q', conclui-se que d' respeita c em $A(S', Q')$. ∎

5.4 Correção

A correção do algoritmo de Dinitz-Thorup é facilmente demonstrada através dos invariantes apresentados. Mais ainda, a demonstração da correção do algoritmo de Dinitz-Thorup é textualmente a mesma da correção do algoritmo de Dijkstra (seção 3.3), já que este último não utiliza o invariante da monotonicidade (dk3).

Teorema 5.1 (teorema da correção): *Para cada vértice t acessível a partir de s o algoritmo de Dinitz-Thorup devolve um caminho de s a t que tem comprimento mínimo.*

Demonstração: Textualmente a mesma do teorema 3.1.

∎

5.5 Eficiência

Como no algoritmo de Dijkstra, d', ψ', S', Q' e U' foram introduzidos na descrição do algoritmo por meras razões técnicas. Em termos de eficiência, não há necessidade de levar em consideração as instruções que inicializam estes objetos. A descrição pode ser feita inteiramente em termos de d, ψ, S, Q e U.

Uma possível implementação do algoritmo de Dinitz-Thorup é a seguinte. Seja $\Delta := \lfloor (nC + 1)/\delta \rfloor$. Os elementos de $\mathcal{P}$ são mantidos em buckets $B(0), B(1), \ldots, B(\Delta)$. Para cada i em $[0..\Delta - 1]$, $B(i)$ contém os elementos maduros X de $\mathcal{P}$ tais que

$$i\delta \leq \min\{d(v) : v \in Q \cap X\} \leq (i + 1)\delta - 1,$$

ou, equivalentemente,

$$i = \lfloor \min\{d(v) : v \in Q \cap X\}/\delta \rfloor.$$

O bucket $B(\Delta)$ contém os elementos X de $\mathcal{P}$ tais que $Q \cap X = \emptyset$ e $U \cap X \neq \emptyset$. Desta forma, $Q = \emptyset$ se e somente se $B(i) = \emptyset$ para cada i em $[0..\Delta - 1]$.

Em cada iteração, para escolher um elemento maduro X da δ-partição $\mathcal{P}$ basta encontrar o menor k em $[0..\Delta - 1]$ tal que $B(k)$ é não-vazio. Devido ao invariante das folgas relaxadas (dt2),

os elementos maduros podem ser escolhidos pelo algoritmo a medida que os buckets são visitados na ordem $B(0), B(1), \ldots, B(\Delta - 1)$. As operações principais envolvendo os buckets são remoções e inserções. Suponha que cada bucket é representado através de uma lista ligada. Assim, cada operação de remoção e inserção de elementos de $\mathcal{P}$ em buckets pode ser realizada, no modelo RAM, em tempo constante: para determinar o bucket $B(k)$ que contém um certo elemento X de $\mathcal{P}$ basta computar $k = \lfloor \min\{d(v) : v \in X \setminus S\}/\delta \rfloor$. Portanto, o consumo total de tempo das operações envolvendo os buckets é proporcional a $n + m + \Delta$. Na simulação do algoritmo de Dinitz-Thorup, ilustrada na figura 5.2, os buckets são representados pelo vetor ao lado de cada grafo.

Teorema 5.2 (consumo de tempo): *O algoritmo de Dinitz-Thorup, quando executado, no modelo RAM, em um grafo com n vértices e m arcos, gasta tempo $O(n+m+\Delta)$ mais o tempo necessário para manter o vértice com menor potencial em $X \setminus S$, para cada X na δ-partição, onde $\Delta = \lfloor (nC+1)/\delta \rfloor$ e C é o maior comprimento de um arco.* ∎

5.6 Versão em CWEB

Conforme visto na seção anterior, a implementação do algoritmo de Dinitz-Thorup usa os buckets $B(0), B(1), \ldots, B(\Delta)$ para manter os elementos de uma dada δ-partição $\mathcal{P}$, que neste caso, é representada por um grafo p, cujos vértices são os elementos de $\mathcal{P}$. A figura 5.4 mostra a representação de $\mathcal{P}$ interpretada pelo programa.

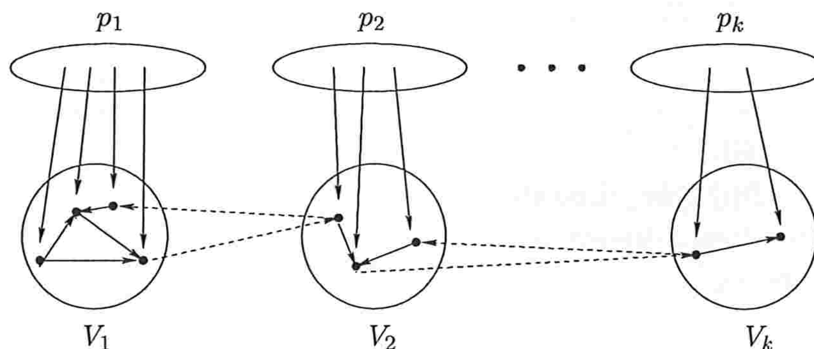


Figura 5.4: Cada elemento V_i da δ-partição $\mathcal{P}$ é representado por um vértice p_i do grafo p. Cada arco de p tem ponta inicial em algum vértice p_i e ponta final em algum vértice do grafo de entrada. Os arcos pontilhados são os arcos do grafo de entrada e têm comprimento pelo menos δ.

71 ⟨Variáveis globais 13⟩ +≡
Vertex $*B$; /* bucket */

A implementação utiliza os seguintes ponteiros para cada vértice: *prox_elemento* e *ant_elemento* que mantêm uma lista de elementos da δ-partição $\mathcal{P}$ em cada $B(i)$; o ponteiro *elemento* indica, para cada vértice de G (grafo de entrada), a qual elemento de $\mathcal{P}$[5] ele pertence; *maduro* aponta para os vértices de G que minimizam d em cada elemento de $\mathcal{P}$; e *status* indica se um vértice de G ou elemento de $\mathcal{P}$ foi examinado ou não.

72 ⟨ Definições 12 ⟩ +≡

```
#define prox_elemento s.V   /* próxima elemento da lista */
#define ant_elemento t.V    /* elemento anterior da lista */
#define elemento y.V        /* aponta para o elemento de P que contém o vértice */
#define maduro x.V          /* próximo vértice que pode ser examinado */
#define status z.I
#define B(i) (B + i)
```

Conforme as implementações de filas de prioridade (capítulo 4), na implementação do algoritmo de Dinitz-Thorup também teremos as operações insert, delete-min e decrease-key que são representadas pelas funções *insert_dinitz*, *delete_min_dinitz* e *decrease_key_dinitz*.

A função *insert_dinitz*, insere um elemento x de p no bucket $B(i)$. O tempo gasto por essa função é $O(1)$.

73 ⟨ Funções auxiliares 32 ⟩ +≡

```
void insert_dinitz(x, i)
    Vertex *x;
    long i;
{
  x→ant_elemento = B(i);
  x→prox_elemento = B(i)→prox_elemento;
  (B(i)→prox_elemento)→ant_elemento = x;
  B(i)→prox_elemento = x;
}
```

A função *delete_min_dinitz* remove o vértice, que tem o menor potencial, do elemento x e escolhe o novo vértice cujo potencial é mínimo. Se todos os vértices de x já foram examinados, x→*status* ≡ `EXAMINADO`. Caso contrário, x→*maduro* aponta para o vértice com o menor potencial. O tempo gasto por essa função é $O(|x|)$, onde $|x|$ é o número de vértices de x.

[5] Na implementação um elemento de $\mathcal{P}$ é um vértice de p.

74 ⟨Funções auxiliares 32⟩ +≡

```
Vertex *delete_min_dinitz(x)
    Vertex *x;
{
  register Vertex *u, *v;
  register Arc *a;
  u = x→maduro;
  u→status = EXAMINADO;        /* Escolhe o novo mínimo de x */
  x→dist = infinito;
  x→maduro = Λ;
  for (a = x→arcs; a; a = a→next) {      /* percorre os vértices do elemento x */
    v = a→tip;
    if (v→status ≡ EXAMINADO) continue;
    if (v→dist ≤ x→dist) {
      x→dist = v→dist;
      x→maduro = v;
    }
  }
  if (x→maduro ≡ Λ) x→status = EXAMINADO;
  return u;
}
```

A função *decrease_key_dinitz* remove, usando a função *remove_elemento*, e move, caso necessário, um elemento y do bucket atual para o bucket $B(\lfloor d(v)/\delta \rfloor)$, onde v é um vértice que pertence ao elemento y. O tempo gasto por essa função é $O(1)$.

75 ⟨Funções auxiliares 32⟩ +≡

```
void remove_elemento(x)
    Vertex *x;
{
  (x→prox_elemento)→ant_elemento = x→ant_elemento;
  (x→ant_elemento)→prox_elemento = x→prox_elemento;
}
void decrease_key_dinitz(v, dt)
    Vertex *v;
    long dt;
{
  register Vertex *y;
```

```
    register long i, j;
    y = v->elemento;      /* y é o elemento que contém v */
    if (v->dist < y->dist) {
      j = (long) y->dist/dt;      /* guarda a posição antiga */
      y->dist = v->dist;
      y->maduro = v;
      i = (long) v->dist/dt;      /* ⌊d(v)/δ⌋ */
      if (i < j) {
        remove_elemento(y);      /* remove y do bucket atual */
        insert_dinitz(y, i);     /* insere y em B(i) */
      }
    }
  }
```

76 ⟨ Algoritmo de Dinitz-Thorup 76 ⟩ ≡

```
  void dinitz(g, dt, p, s)
      Graph *g;
      long dt;      /* arestas com comprimento pelo menos dt */
      Graph *p;     /* p é a dt-partição */
      Vertex *s;    /* vértice inicial */
  { ⟨ Variáveis de dinitz 77 ⟩
    ⟨ Inicializações de dinitz 78 ⟩
    ⟨ Coloque os elementos de p nos buckets B(0), B(1), ..., B(dtg) 79 ⟩
    for (k = 0; k < dtg; k++) {
      while (B(k) ≠ B(k)->prox_elemento) {
        ⟨ Seja x um elemento em B(k) 80 ⟩
        ⟨ Escolha u em x tal que d(u) seja mínimo 81 ⟩
        ⟨ Examine vértice u 82 ⟩
        ⟨ Verifique se x deve mudar de bucket 84 ⟩
      }
    }
    free(B);
  }
```

Este código é usado no bloco 9.

77 ⟨ Variáveis de dinitz 77 ⟩ ≡

```
  register Vertex *v, *u;
```

```
register Vertex *x;        /* elementos de p */
register Arc *a;
register long i, k;
register long dtg = 0;     /* Δ */
```

Este código é usado no bloco 76.

Inicializa d e ψ como no algoritmo de Dijkstra (seção 3.6). Além disso, o mínimo de cada partição é inicializado com *infinito*, com exceção da partição que contém o vértice inicial s.

78 ⟨Inicializações de dinitz 78⟩ ≡

```
for (v = g->vertices; v < g->vertices + g->n; v++) {
  x = v->elemento;
  x->dist = v->dist = infinito;
  v->pred = v;
  x->status = v->status = NAO_EXAMINADO;
}
s->dist = 0;
(s->elemento)->dist = 0;      /* inicializa o elemento que contém s */
(s->elemento)->maduro = s;    /* d(s) é mínimo no elemento s->elemento */
```

Este código é usado no bloco 76.

79 ⟨Coloque os elementos de p nos buckets $B(0), B(1), \ldots, B(dtg)$ 79⟩ ≡

```
if (dt ≤ 0) {
  printf("%s\n", err_message[ERROR_4]);
  exit(0);
}
dtg = (long)(infinito/dt) + 1;
if ((B = (Vertex *) malloc(dtg * sizeof(Vertex))) ≡ Λ) {
  printf("%s\n", err_message[ERROR_2]);
  exit(0);
}
for (i = 0; i < dtg; i++) {      /* Inicializa cabeças de lista */
  B(i)->ant_elemento = B(i)->prox_elemento = B(i);
}
for (x = p->vertices; x < p->vertices + p->n; x++) {
  i = (long)(x->dist/dt);      /* posição do bucket em que x será inserido */
  insert_dinitz(x, i);         /* insere x no bucket B(i) */
```

```
}
```

Este código é usado no bloco 76.

80 ⟨ Seja x um elemento em $B(k)$ 80 ⟩ ≡

```
x = B(k)→prox_elemento;
```

Este código é usado no bloco 76.

81 ⟨ Escolha u em x tal que $d(u)$ seja mínimo 81 ⟩ ≡

```
u = delete_min_dinitz(x);
```

Este código é usado no bloco 76.

82 ⟨ Examine vértice u 82 ⟩ ≡

```
for (a = u→arcs; a; a = a→next) {
  v = a→tip;
  ⟨ Examine a aresta uv 83 ⟩
}
```

Este código é usado no bloco 76.

Faz d respeitar c em uv (seção 2.4).

83 ⟨ Examine a aresta uv 83 ⟩ ≡

```
if (v→dist − u→dist > a→len) {      /* se a função potencial não é viável */
  v→dist = u→dist + a→len;
  v→pred = u;
  decrease_key_dinitz(v, dt);
}
```

Este código é usado no bloco 82.

84 ⟨ Verifique se x deve mudar de bucket 84 ⟩ ≡

```
if (x→status ≡ EXAMINADO) {      /* todos os vértices de x já foram examinados */
  remove_elemento(x);      /* remove x do bucket atual */
}
else {
  i = (long) x→dist/dt;      /* nova posição do elemento x */
  if (i > k) {
```

```
        remove_elemento(x);        /* remove x do bucket atual */
        insert_dinitz(x, i);       /* insere x no bucket B(i) */
    }
}
```

Este código é usado no bloco 76.

Lema 5.3 (operações Dinitz-Thorup): *A implementação de Dinitz-Thorup executa as operações insert_ dinitz e decrease_ key_ dinitz em tempo $O(1)$ e delete_ min_ dinitz em tempo $O(n)$.* ∎

Portanto, do teorema 5.2 e do lema 5.3, essa implementação do algoritmo de Dinitz-Thorup gasta tempo

$$\underbrace{O(n)}_{\text{insert_dinitz}} + \underbrace{O(m)}_{\text{decrease_key_dinitz}} + \underbrace{O(n^2)}_{\text{delete_min_dinitz}} + \underbrace{O(\Delta)}_{\text{percorrer os buckets}} = O(n^2+m+\Delta).$$

Teorema 5.4: *A implementação do algoritmo de Dinitz-Thorup resolve o PCMV em um grafo com n vértices, m arcos e uma δ-partição em tempo $O(n^2 + m + \Delta)$, onde $\Delta = \lfloor (nC + 1)/\delta \rfloor$ e C é o maior comprimento de um arco.* ∎

Capítulo 6

Algoritmo de Thorup

O algoritmo apresentado neste capítulo, devido a Mikkel Thorup [39], resolve o seguinte problema do caminho mínimo restrito a grafos simétricos com funções comprimento simétricas, em tempo e espaço linear, no modelo RAM:

> **Problema** PCMS(V, A, c, s): Dado um grafo simétrico (V, A), uma função comprimento simétrica c de A em $\mathbb{Z}_{>}$ e um vértice s, encontrar um caminho de comprimento mínimo de s até t, para cada vértice t em V.

PCMS

Da mesma forma que o algoritmo de Dinitz-Thorup (capítulo 5), o algoritmo de Thorup utiliza a idéia de δ-partição e bucketing, para determinar vértices que podem ser examinados em qualquer ordem. Em linhas gerais, a fim de evitar o custo computacional de examinar os vértices do grafo conforme a distância do vértice origem s, Thorup juntou a este bucketing uma certa decomposição hierárquica do grafo e aplicou o algoritmo de Dinitz-Thorup recursivamente a cada elemento desta decomposição.

6.1 Família laminar e representação arbórea

Seja V um conjunto finito. Uma família $\mathcal{L}$ de partes de V é **laminar** se para todo par X e Y de elementos de $\mathcal{L}$ vale que $X \subseteq Y$ ou $Y \subseteq X$ ou $X \cap Y = \emptyset$. Se X e Y são elementos de $\mathcal{L}$ tais que X é um subconjunto maximal propriamente contido em Y então é dito que X é **filho** de Y e Y é **pai** de X.

Uma **representação arbórea** de uma família laminar $\mathcal{L}$ é um grafo $(\mathcal{L}, \mathcal{A})$ tal que (Y, X) é um arco em $\mathcal{A}$ se e somente se X é um filho de Y. Logo, $(\mathcal{L}, \mathcal{A})$ é a união de arborescências. A ilustração de uma família laminar e sua representação arbórea pode ser vista na figura 6.1. Um elemento X em $(\mathcal{L}, \mathcal{A})$ é dito uma **folha** se X é uma folha da arborescência de $(\mathcal{L}, \mathcal{A})$ que contém X. Os elementos em $\mathcal{L}$ que não são folhas são chamados de **internos**.

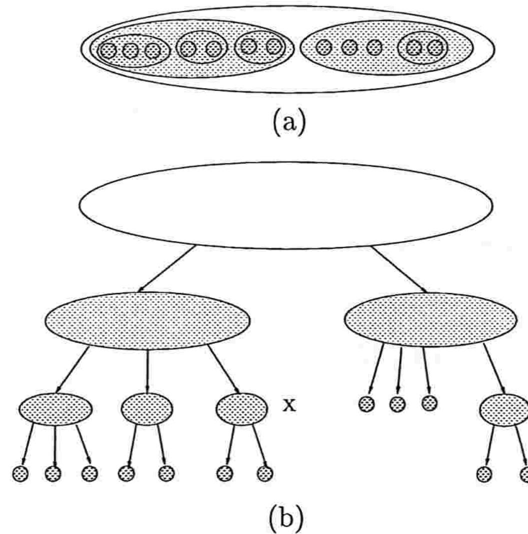


Figura 6.1: A figura (a) mostra o diagrama de Venn de uma família laminar $\mathcal{L}$. (b) é a representação arbórea $(\mathcal{L}, \mathcal{A})$. O nível de X é 2 e a altura de $(\mathcal{L}, \mathcal{A})$ é 3.

Os **ancestrais** de um elemento X são todos os elementos de $\mathcal{L}$ no caminho da raiz da arborescência contendo X até X.

O **nível** de um elemento X de $\mathcal{L}$ é o comprimento do caminho da raiz da arborescência de $(\mathcal{L}, \mathcal{A})$ que contém X até X. A **altura** de $\mathcal{L}$ é o maior nível de um elemento X de $\mathcal{L}$.

Uma família $\mathcal{L}$ de partes de V é **W-completa** para alguma parte W de V se: (c1) $\mathcal{L}$ é laminar; (c2) V está em $\mathcal{L}$; e (c3) $\{v\} \in \mathcal{L}$ se e somente se $v \in W$. Se $\mathcal{L}$ é W-completa, então sua representação arbórea $(\mathcal{L}, \mathcal{A})$ é uma arborescência com raiz V.

6.2 Decomposição hierárquica

Sejam (V, A) um grafo e c uma função comprimento de A em $\mathbb{Z}_{>}$. Seja $\mathcal{L}$ uma família W-completa e seja δ uma função que associa a cada elemento interno X de $\mathcal{L}$ um inteiro positivo $\delta(X)$. É dito que $\mathcal{L}$ forma uma **δ-decomposição W-hierárquica** de (V, A) em relação a c, se para cada elemento X de $\mathcal{L}$ vale que:

(h1) o grafo $(X, A(X))$ é conexo, exceto, possivelmente, se $X = V$; e

(h2) cada aresta com pontas em filhos distintos de X tem comprimento pelo menos $\delta(X)$.

Quando a função δ ou o conjunto W forem evidentes ou irrelevantes serão omitidos. Assim, algumas vezes é escrito simplesmente decomposição hierárquica, ou δ-decomposição hierárquica.

Do ponto de vista da implementação do algoritmo descrito neste capítulo, é conveniente que, para cada X em $\mathcal{L}$, $\delta(X)$ seja um número da forma 2^k, para algum k. A figura 6.2 ilustra a condição (h2) e a figura 6.3 mostra a decomposição hierárquica de um grafo.

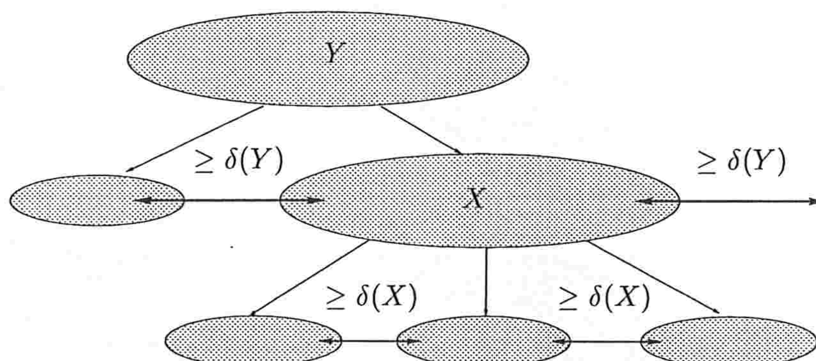


Figura 6.2: Decomposição hierárquica (condição (h2)).

Para resolver o PCMS o algoritmo de Thorup, como os algoritmos de Dijkstra e Dinitz-Thorup, mantém, entre outros objetos, uma função potencial d (distância tentativa) e o conjunto Q dos vértices visitados. Além disso, também mantém uma δ-decomposição hierárquica $\mathcal{L}$. Se X é um elemento de $\mathcal{L}$, então X é **maduro** se $Q \cap X \neq \emptyset$ e

$$X = V \quad \text{ou} \quad \min\{d(v) : v \in Q \cap Y\} \leq \min\{d(v) : v \in Q \cap X\} \leq \min\{d(v) : v \in Q \cap Y\} + \delta(Y)/2,$$

onde Y é o pai de X. A figura 6.4 mostra exemplos de elementos maduros.

6.3 Descrição

O algoritmo de Thorup resolve o PCMS e devolve os mesmos certificados que o algoritmo de Dijkstra e Dinitz-Thorup, a saber, uma função predecessor codificando os caminhos compactamente e uma função potencial viável atestando a minimalidade dos caminhos e a eventual não acessibilidade de alguns vértices. O algoritmo de Thorup aplica, de certa forma, recursivamente o algoritmo de Dinitz-Thorup, a fim de examinar um vértice v se e somente se $\{v\}$ e todos os seus ancestrais são maduros. A função δ da δ-decomposição V-hierárquica $\mathcal{L}$ utilizada pelo algoritmo, deve ser tal que, se X é filho de Y então $\delta(X) \leq \delta(Y)/2$. A seção 6.7 é descrito como δ e $\mathcal{L}$ são convenientemente construídos.

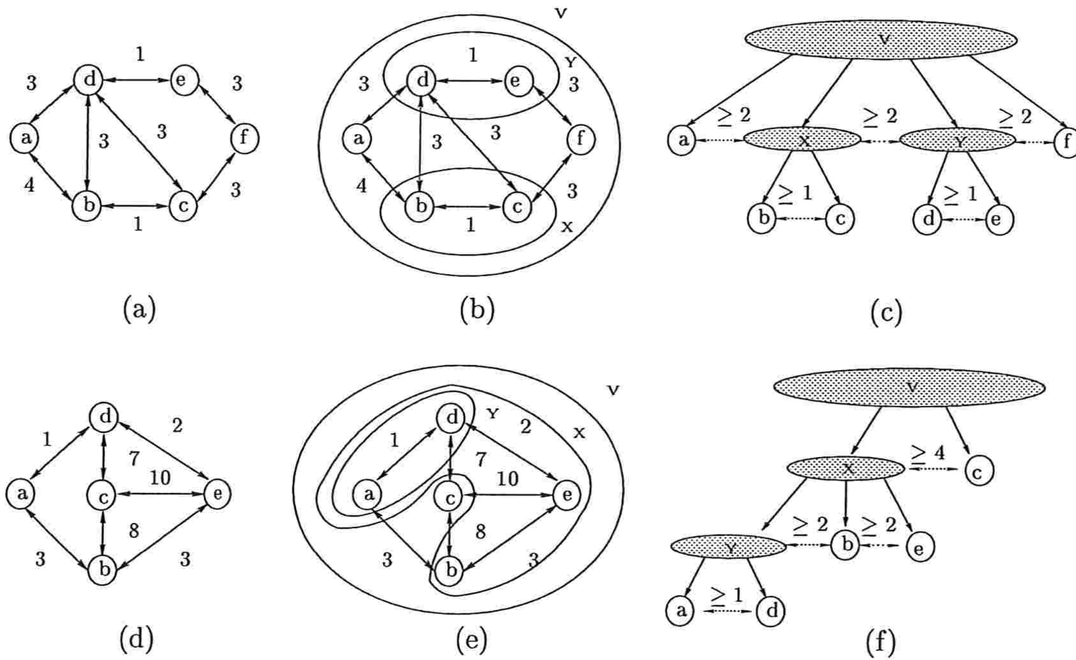


Figura 6.3: (a) e (d) mostram grafos simétricos. A figura (b) mostra a δ-decomposição V-hierárquica do grafo em (a), onde $\delta(V) = 2$, $\delta(X) = 1$ e $\delta(Y) = 1$, e (c) mostra a correspondente representação arbórea. De maneira análoga, (e) exibe a δ-decomposição V-hierárquica do grafo em (d), onde $\delta(V) = 4$, $\delta(X) = 2$ e $\delta(Y) = 1$, e (f) ilustra a correspondente representação arbórea.

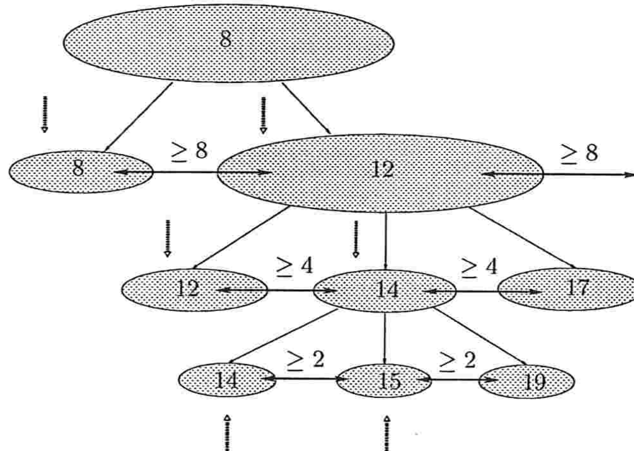


Figura 6.4: As elipses representam elementos de $\mathcal{L}$. O número dentro de cada elemento X de $\mathcal{L}$ é o menor potencial de um vértice de $X \in Q$. A seta pontilhada indica quais são os elementos maduros.

Algoritmo de Thorup. Recebe um grafo (V, A), uma função comprimento simétrica c de A em $\mathbb{Z}_{>}$, e um vértice s e devolve uma função predecessor ψ e uma função potencial d que respeita c tais que, para cada vértice t, se t é acessível a partir de s, então ψ determina um caminho de s a t que tem comprimento $d(t) - d(s)$, caso contrário, $d(t) - d(s) = nC + 1$, onde $C := \max\{c(u, v) : uv \in A\}$.

Cada iteração começa com uma função potencial d, uma função predecessor ψ, partes S, Q e U de V, uma decomposição $(Q \cup U)$-hierárquica $\mathcal{L}$, e uma pilha $L = \langle X_0, X_1, \ldots, X_t \rangle$ de elementos de $\mathcal{L}$, tais que X_{i+1} é filho de X_i, para $i = 0, \ldots, t-1$.

No início da primeira iteração $d(s) = 0$ e $d(v) = nC + 1$ para cada vértice v distinto de s, $\psi(v) = v$ para cada vértice v, $S = \emptyset$, $Q = \{s\}$, $U = V \setminus \{s\}$, $\mathcal{L}$ é uma decomposição V-hierárquica de (V, A) em relação a c e $L = \langle V \rangle$. [O algoritmo supõe que se X é filho de Y, então $\delta(X) \leq \delta(Y)/2$.]

A ação em cada iteração depende do elemento X no topo da pilha L e consiste em:

Caso 1: X não é maduro.

Caso 1A: $X = V$.

Devolva d e ψ e pare.

Caso 1B: $X \neq V$ e X é folha.

Seja $\mathcal{L}'$ a decomposição $(Q \cup U)$-hierárquica obtida após a remoção de X de $\mathcal{L}$.

Seja L' a pilha obtida após desempilhar X de L.

Comece nova iteração com $\mathcal{L}'$ e L' nos papéis de $\mathcal{L}$ e L.

Caso 1C: $X \neq V$ e X não é folha.

Seja L' a pilha obtida após desempilhar X de L.

Comece nova iteração com L' no papel de L.

Caso 2: X é maduro.

Caso 2A: $X = \{u\}$, para algum u em V [isto é, X é folha].

$S' := S \cup \{u\}$.

$Q' := Q \setminus \{u\}$.

$U' := U$.

Para cada v em V faça $d'(v) := d(v)$ e $\psi'(v) := \psi(v)$.

Para cada arco uv faça

Se $d(v) > d(u) + c(u, v)$ então

$d'(v) := d(u) + c(u,v)$, $\psi'(v) := u$ e remova[1] v de U' e acrescente a Q'.

Seja $\mathcal{L}'$ a decomposição $(Q' \cup U')$-hierárquica obtida após a remoção de $\{u\}$ de $\mathcal{L}$.

Seja L' a pilha obtida após desempilhar X de L.

Comece nova iteração com d', ψ', S', Q', U', $\mathcal{L}'$ e L' nos papéis de d, ψ, S, Q, U, $\mathcal{L}$ e L.

Caso 2B: $|X| > 1$ [isto é, X não é folha].

Seja X' um filho maduro de X.

Seja L' a pilha obtida após empilhar X' em L.

Comece nova iteração com L' no papel de L. □

Diremos que o algoritmo **visita** um elemento X de $\mathcal{L}$ sempre que X é empilhado em L. O algoritmo visita apenas elementos maduros, entretanto, um elemento em L pode deixar de ser maduro durante a visita a algum dos seus descendentes. As figuras 6.5, 6.6, 6.7, 6.8 e 6.9 mostram a simulação do algoritmo de Thorup no exemplo da figura 6.3(a).

Considere uma iteração em que ocorre o caso 2A. Suponha que no início da iteração $L = \langle X_0, X_1, \ldots, X_t \rangle$, onde $X_0 = V$ e $X_t = X = \{u\}$. Como $\{u\}$ e todos os seus ancestrais são maduros, então

$$\begin{aligned}
\min\{d(v) : v \in Q \cap X_1\} &\leq \min\{d(v) : v \in Q\} + \delta(X_0)/2 \\
\min\{d(v) : v \in Q \cap X_2\} &\leq \min\{d(v) : v \in Q \cap X_1\} + \delta(X_1)/2 \\
&\vdots \\
\min\{d(v) : v \in Q \cap X_t\} &\leq \min\{d(v) : v \in Q \cap X_{t-1}\} + \delta(X_{t-1})/2
\end{aligned}$$

Portanto, para cada vértice v em Q, vale que $d(u) \leq d(v) + (\delta(V) + \delta(X_1) + \cdots + \delta(X_{t-1}))/2 < d(v) + \delta(V)$. A segunda desigualdade vale, pois $\delta(X_i) \leq \delta(X_{i-1})/2$ para cada i em $[1..t-1]$. Esta observação é a essência do invariante (th1) da próxima seção.

[1] É possível que v já pertença a Q' e não esteja em U'. Nesse caso estas últimas duas instruções são redundantes.

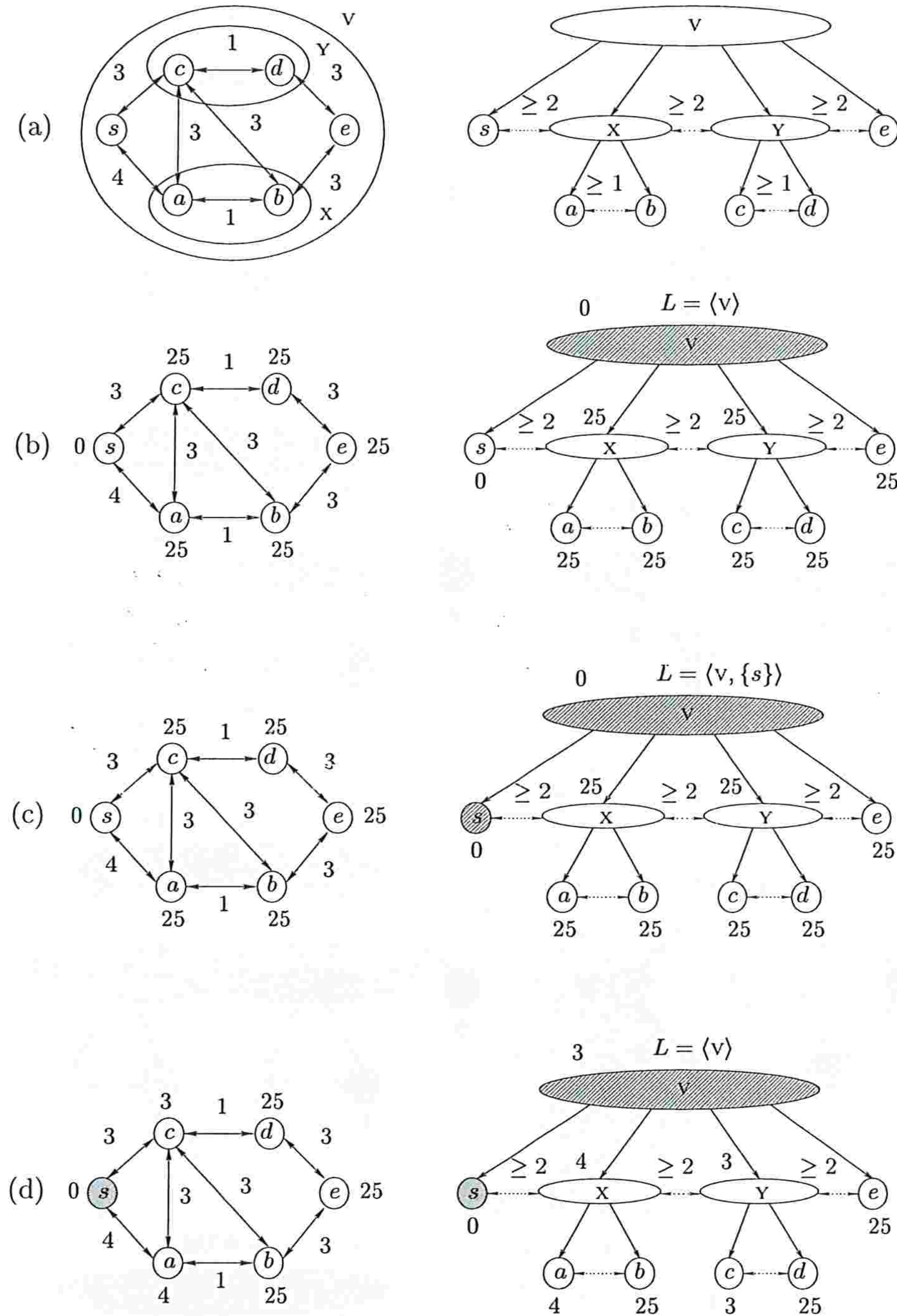


Figura 6.5: Execução do algoritmo de Thorup. (a) exibe um grafo com comprimento nos arcos, junto com uma δ-decomposição V-hierárquica $\mathcal{L}$. O vértice inicial é s. (b) mostra a situação no início da primeira iteração. Um número próximo a um vértice ou elemento é o seu potencial. No grafo, os vértices pretos são os de S, os vértices cinzas são os de Q, e os vértices brancos são os de U. Na decomposição hierárquica, os elementos hachurados são os que estão em L e os elementos em preto são os que foram removidos de $\mathcal{L}$. (c) mostra a situação após empilhar o elemento $\{s\}$ (caso 2B) e (d) é a situação após examinar $\{s\}$ (caso 2A).

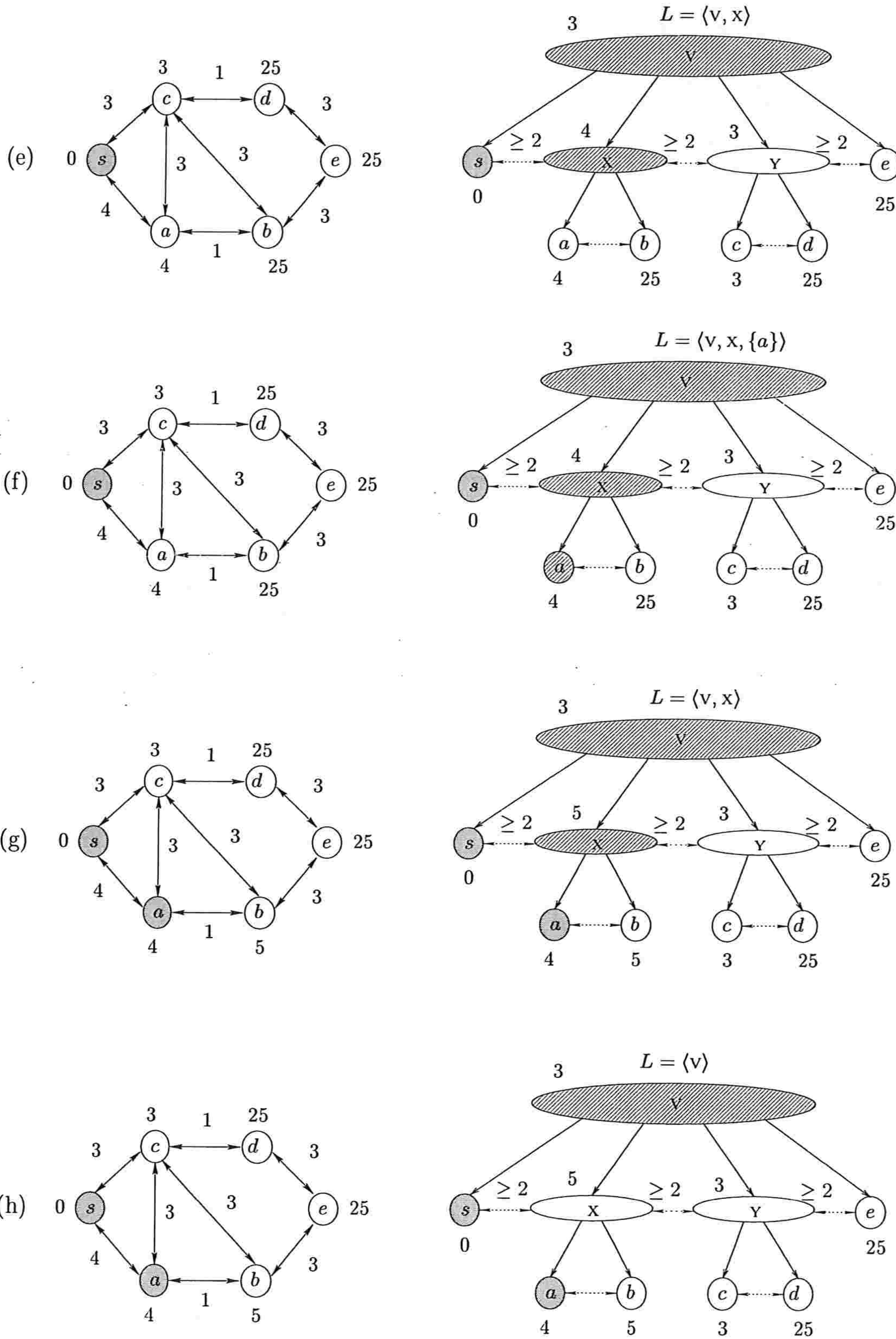


Figura 6.6: Execução do algoritmo de Thorup (continuação). Em (e) e (f) ocorre o caso 2B, e em (g) ocorre o caso 2A. Note que em (h), x não é maduro, pois $3 + (2/2) < 5$, então é desempilhado (caso 1C).

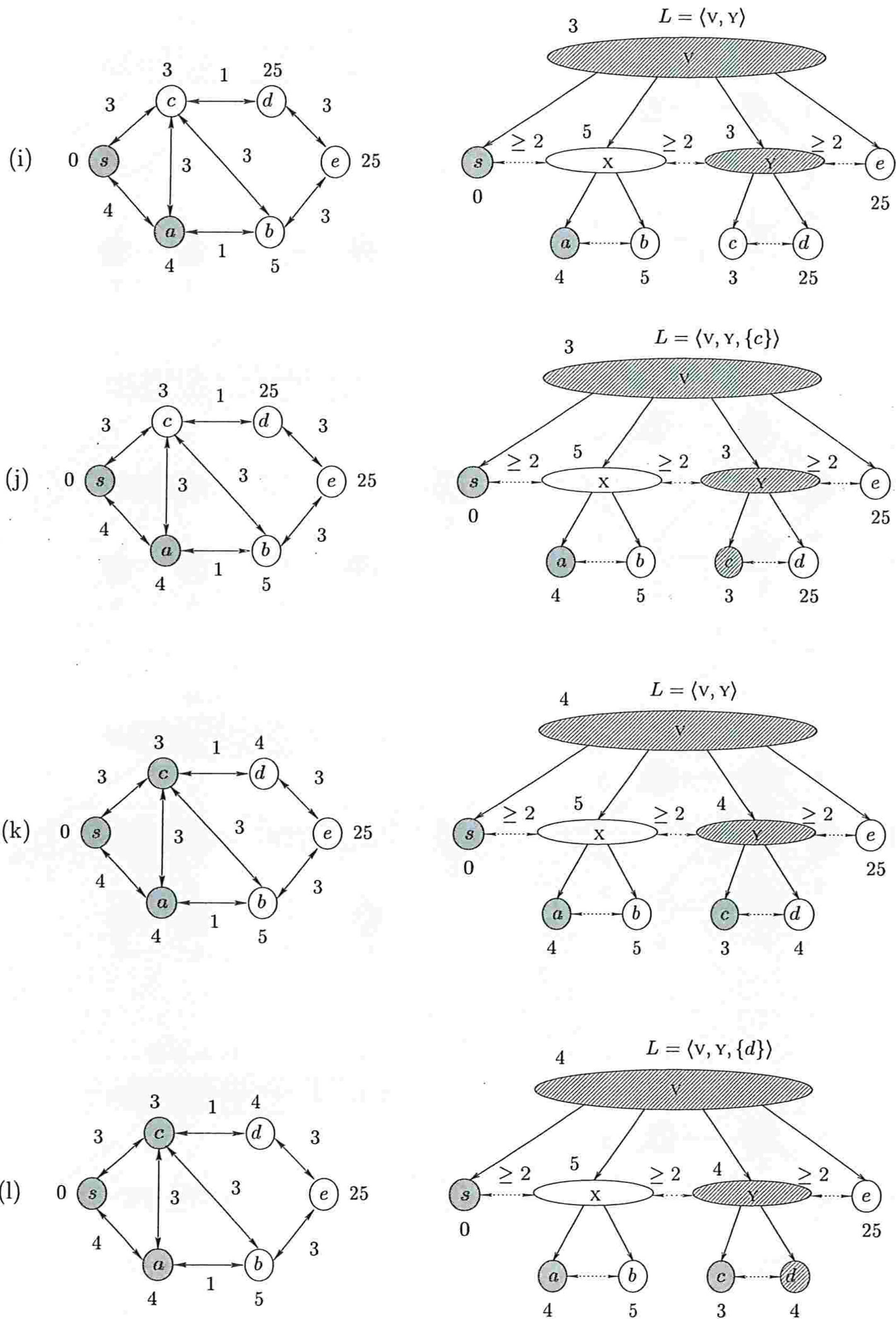


Figura 6.7: Execução do algoritmo de Thorup (continuação). Em (i), (j) e (l) ocorre o caso 2B, e em (k) ocorre o caso 2A.

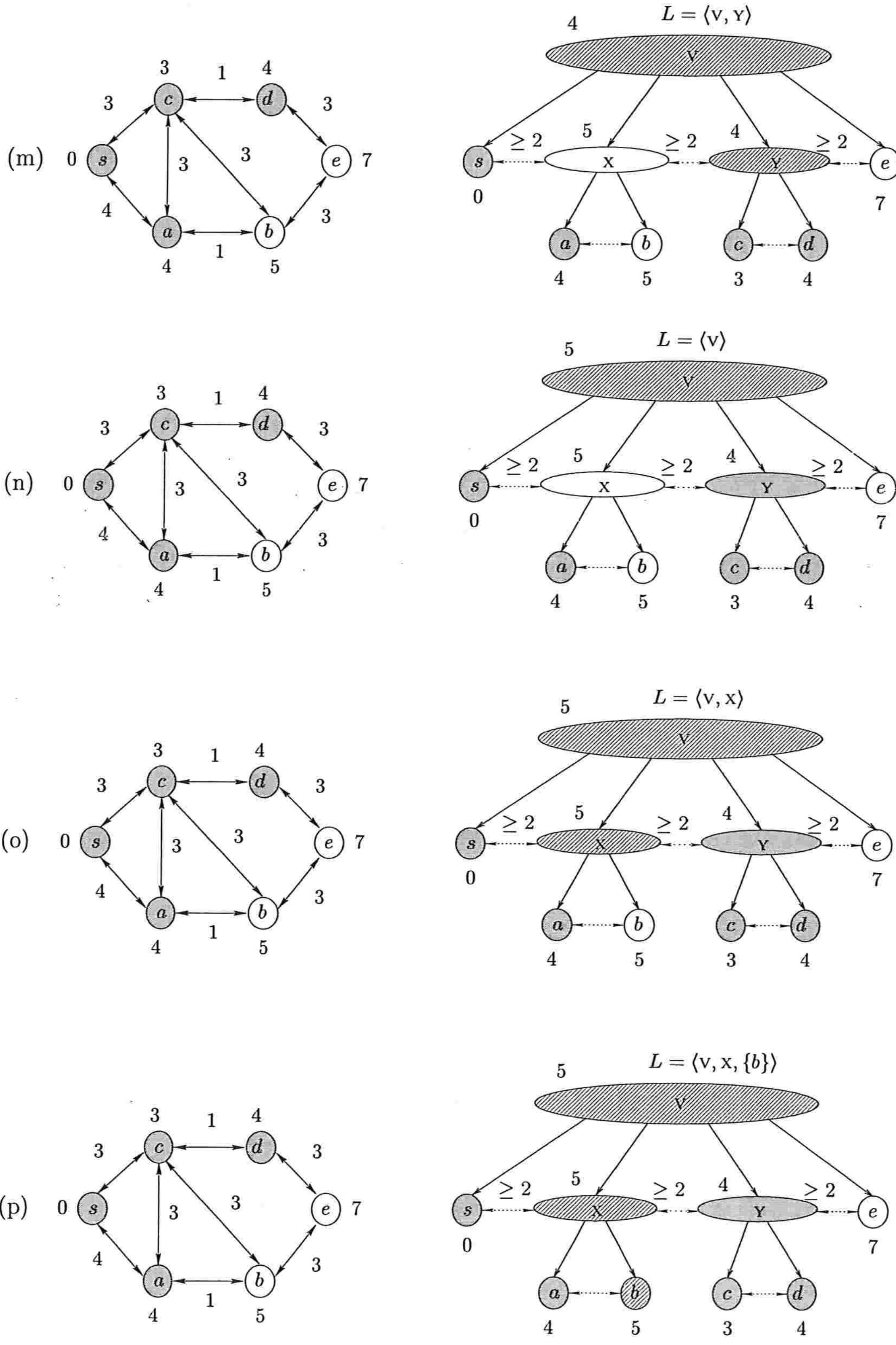


Figura 6.8: Execução do algoritmo de Thorup (continuação). Em (m) ocorre o caso 2A, em (n) o caso 1B e em (o) e (p) o caso 2B.

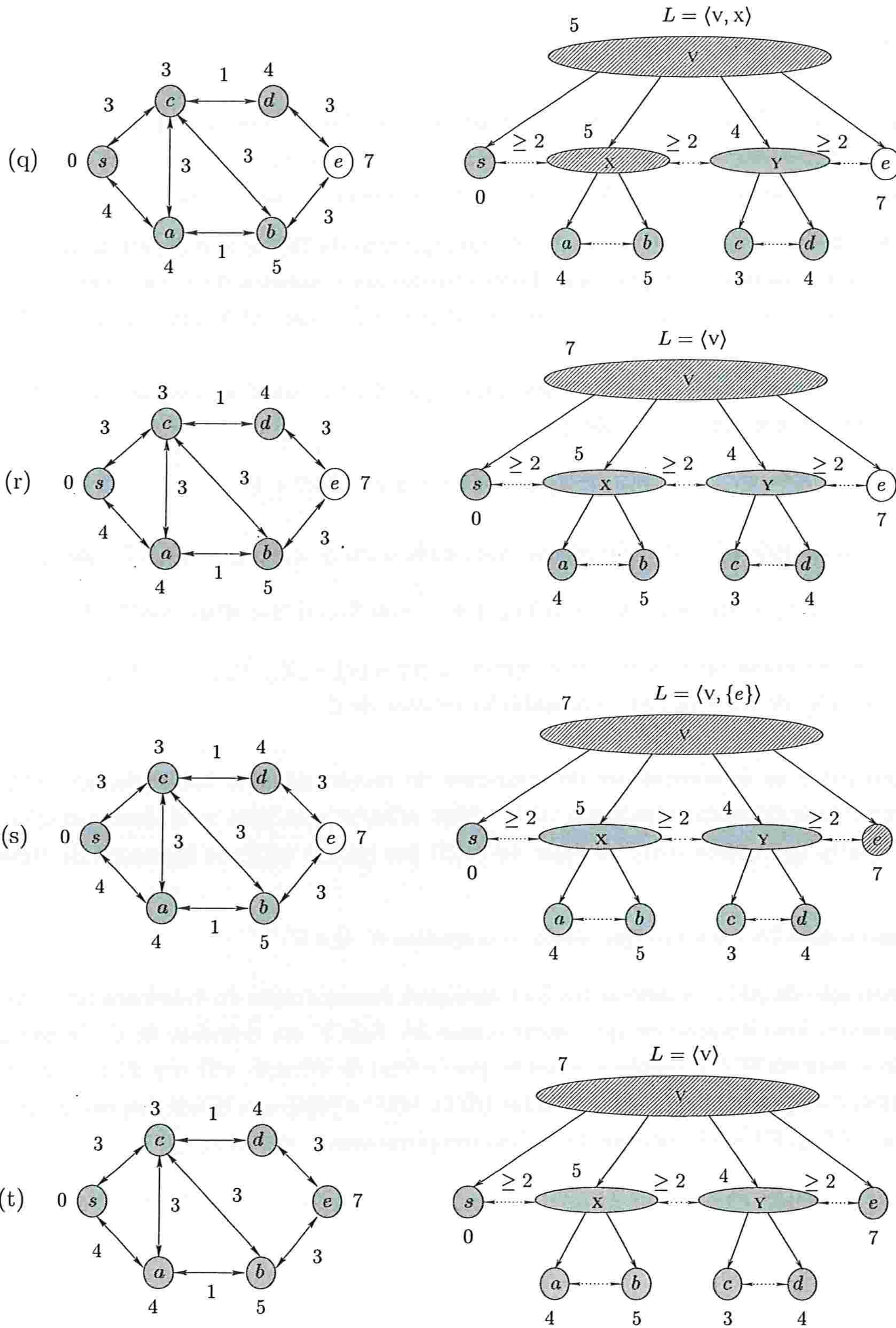


Figura 6.9: Execução do algoritmo de Thorup (continuação). Em (q) ocorre o caso 2A, em (r) o caso 1B, em (s) o caso 2B, e em (t) ocorre o caso 2A.

6.4 Invariantes

O algoritmo de Thorup, a exemplo do algoritmo de Dinitz-Thorup (capítulo 5), mantém todos os invariantes do algoritmo de Dijkstra (seção 3.2), exceto aquele que é o responsável pela ordem que os vértices são examinados, o invariante da monotonicidade (dk3).

No lugar do invariante da monotonicidade do algoritmo de Dijkstra, o algoritmo de Thorup mantém os dois invariantes a seguir, que dizem respeito aos elementos de $\mathcal{L}$ e à δ-decomposição hierárquica. Note que a função δ do algoritmo é tal que, se X é filho de Y, então $\delta(X) \leq \delta(Y)/2$.

(th1) (monotonicidade local relaxada) para cada parte X de V em $\mathcal{L}$ e para cada u em $S \cap X$, v em $Q \cap X$ e w em $U \cap X$ vale que[2]

$$d(u) \leq d(v) + \delta(X) < d(w) = nC + 1.$$

(th2) (monotonicidade relaxada telescópica) para cada u em S, v em Q e w em U vale que

$$d(u) \leq d(v) + (\delta(X_k) + \delta(X_{k+1}) + \cdots + \delta(X_{t-1}))/2 < d(w) = nC + 1$$

onde X_k é o ancestral de maior nível comum a $\{u\}$ e $\{v\}$ e $\langle X_k, X_{k+1}, \ldots, X_{t-1}, X_t = \{u\}\rangle$ é o caminho de X_k a $\{u\}$ na representação arbórea de $\mathcal{L}$.

A seguir estão as demonstrações do invariante da monotonicidade local relaxada (th1) e da monotonicidade relaxada telescópica (th2). Mais adiante, está uma nova demonstração do invariante (dk9), que utiliza (th1) no lugar de (dk3) (ou (dt1) e (dt2) do algoritmo de Dinitz-Thorup).

Note que o caso 2A é o único que altera os conjuntos S, Q e U.

Demonstração de (th1): A demonstração é análoga à demonstração do invariante (dt1). Basta considerarmos uma iteração em que ocorre o caso 2A. Seja Y um elemento de $\mathcal{L}$. Se no final da iteração w está em $U' \cap Y$, então é evidente que no final da iteração vale que $d'(w) = d(w) = nC + 1$. Além disso, combinando os invariantes (dk4), (dk5) e (dk6) que dizem respeito a função predecessor e $\{S, Q, U\}$ e o invariante das folgas complementares, obtém-se que

$$d'(v) + \delta(Y) \leq d(v) + \delta(Y) \leq (n-1)C + \delta(Y) < nC + 1 = d'(w)$$

para cada v em $(Q' \cap Y)$.

Assim, nos concentramos em verificar que para cada x em $S' \cap Y$ e y em $Q' \cap Y$, ao final da iteração, vale que $d'(x) \leq d(y) + \delta(Y)$.

[2] A expressão "$d(u) \leq d(v) + \delta(X) < d(w) = nC + 1$" deve ser entendida como uma abreviação. Se algum dos conjuntos envolvidos é vazio, considere apenas as desigualdades que fazem sentido.

Seja x um vértice em $S' \cap Y$ e y um vértice em $Q' \cap Y$. Se $d'(y) = d(y)$, então como $X = \{u\}$ e todos os seus ancestrais são maduros e pelo invariante (th1) tem-se que

$$d'(x) = d(x) \leq d(y) + \delta(Y) = d'(y) + \delta(Y).$$

Suponha que $d'(y) < d(y)$. Portanto,

$$d(u) \leq d(u) + c(u, y) = d'(y).$$

Como $\mathcal{L}$ é uma decomposição hierárquica e do invariante (th1), vale que $d(x) \leq d(u) + \delta(Y)$. Portanto,

$$d'(x) = d(x) \leq d(u) + \delta(Y) \leq d'(y) + \delta(Y).$$

∎

Demonstração de (th2): Considere uma iteração em que ocorre o caso 2B. Seja w um vértice em U'. Ao final da iteração $d'(w) = d(w) = nC + 1$. Combinando-se os invariantes (dk4), (dk5) e (dk6) com o invariante das folgas complementares (dk12) obtém-se que

$$\begin{aligned} d'(v) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2 &\leq d(v) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2 \\ &\leq (n-1)C + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2 \\ &< nC + 1 = d'(w) \end{aligned}$$

para cada v em Q' e X_{i+1} é filho de X_i para $i = k, \ldots, t-2$.

Assim, resta demonstrar que para cada x em S e y em Q, vale que

$$d(x) \leq d(y) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2.$$

Seja x um vértice em S' e y um vértice em Q'. Se ao final da iteração $d'(y) = d(y)$, então pelo invariante (th2) e como $\{u\}$ e todos os seus ancestrais são maduros, tem-se que

$$d'(x) = d(x) \leq d(y) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2 = d'(y) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2.$$

Logo, podemos supor que ao final da iteração vale que $d'(y) < d(y)$. Portanto,

$$d(u) \leq d(u) + c(u, y) = d'(y).$$

Do invariante (th2) sabe-se que $d(x) \leq d(u) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2$. Portanto,

$$d'(x) = d(x) \leq d(u) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2 \leq d'(y) + (\delta(X_k) + \cdots + \delta(X_{t-1}))/2 < d'(y) + \delta(X_k).$$

∎

Demonstração de (dk9): Considere uma iteração em que ocorre o caso 2B. Seja yx um arco qualquer em $A(Q', S')$ ao final da iteração. Seja X em $\mathcal{L}$ o ancestral comum de $\{x\}$ e $\{y\}$ de maior nível. Suponha que o nível de X é $t-1$. Do invariante (th1) obtem-se que

$$d'(x) \leq d'(y) + \delta(X)/2.$$

Assim, como $\mathcal{L}$ é uma decomposição hierárquica, então $d'(x) - d'(y) \leq \delta(X)/2 \leq c(y, x)$. Portanto, ao final da iteração, d respeita c em $A(Q', S')$.

O processo de examinar u faz com que d' respeite c em cada arco com ponta inicial em u. Do invariante (dk9) sabe-se que d respeita c em $A(S, Q)$. Assim, como $d'(v) = d(v)$ para cada v em S' e $d'(v) \leq d(v)$ para cada v em Q', concluí-se que d' respeita c em $A(S', Q')$. ∎

6.5 Correção

A correção do algoritmo de Thorup é facilmente demonstrada através dos seus invariantes, e é textualmente idêntica a demonstração da correção do algoritmo de Dijkstra (seção 3.1), já que este último não utiliza o invariante da monotonicidade (dk3). Note que na última iteração do algoritmo de Thorup, quando ocorre o caso 1A, tem-se que Q é vazio, pois se $X = V$ não é maduro, então $Q \cap V = \emptyset$.

Teorema 6.1 (teorema da correção): *Para cada vértice t acessível a partir de s o algoritmo de Thorup devolve um caminho de s a t que tem comprimento mínimo.* ∎

6.6 Eficiência

Seja (V, A, c, s) uma instância do PCMS. Denotamos por n e m o número de vértices e arestas de (V, A). Uma implementação eficiente do algoritmo de Thorup deve resolver eficientemente os seguintes problemas:

(p1) (pré-processamento) construção da δ-decomposição V-hierárquica $\mathcal{L}$ de (V, A) em relação a c;

(p2) (caso 2A) atualizar os potenciais e manter $\min\{d(v) : v \in Q \cap X\}$ para cada X em $\mathcal{L}$; e

(p3) (caso 2B) escolher o próximo elemento maduro a ser visitado.

Construção da decomposição hierárquica

O problema (p1) é resolvido por um pré-processamento do algoritmo, que é feito em tempo $O(m+n)$ utilizando-se os trabalhos de Andersson, Hagerup, Nilsson e Raman [4], Fredman e Willard [18] e Gabow e Tarjan [19]. A implementação do algoritmo mantém a decomposição hierárquica $\mathcal{L}$ através de sua representação arbórea $(\mathcal{L}, \mathcal{A})$. A construção de $(\mathcal{L}, \mathcal{A})$ está descrita na próxima seção.

Além de construir uma δ-decomposição V-hierárquica o pré-processamento também devolve uma floresta geradora (V, T) de (V, A) tal que

(h3) cada aresta de T com pontas em filhos distintos de algum X em $\mathcal{L}$ tem comprimento inferior a $2\delta(X)$.

As condições (h2) e (h3) juntas implicam que cada aresta de T com pontas em filhos distintos de algum elemento X de $\mathcal{L}$, tem seu comprimento em $[\delta(X)..2\delta(X) - 1]$.

É evidente que, para cada X em $\mathcal{L}$, o valor

$$\sum_{uv \in T(X)} c(u, v)$$

é um limitante superior para o **diâmetro** do grafo $(X, A(X))$, isto é, para o comprimento do maior caminho em $(X, A(X))$.

Atualização dos potenciais

A operação que precisa ser realizada eficientemente é

> atualizar $\min\{d(v) : v \in Q \cap X\}$ sempre que o potencial $d(v)$ de um vértice v em $Q \cap X$ é decrescido.

onde X é um elemento da decomposição $\mathcal{L}$. Thorup [39] descreve como esta operação pode ser implementada em tempo amortizado constante.

Em linhas gerais, a idéia de Thorup é a seguinte. Seja $v_1, \ldots, v_n$ uma ordenação dos vértices em Q induzida por uma ordem arbitrária dos elementos internos da representação arbórea $(\mathcal{L}, \mathcal{A})$. No início de cada iteração do algoritmo, para cada elemento X em $\mathcal{L}$, os elementos $\{\{v\} : v \in Q \cap X\}$ são folhas de $(\mathcal{L}, \mathcal{A})$ que têm X como ancestral. Estas folhas formam um segmento contíguo de $v_1, \ldots, v_n$.

Thorup descreve como, utilizando-se os atomic heaps de Fredman e Willard [18], é possível manter uma partição dinâmica de $v_1, \ldots, v_n$ em segmentos contíguos de tal forma que, para cada X em $\mathcal{L}$, tenha-se que

$$\min\{d(v) : v \in Q \cap X\} = \min\{d(v_i) : i \in [k..l]\}$$

onde k e l dependem de X.

Escolha de um elemento maduro

A fim de escolher o próximo elemento maduro a ser visitado pelo algoritmo, pode-se proceder de maneira análoga ao algoritmo de Dinitz-Thorup, como apresentado a seguir.

Cada elemento interno Y em $\mathcal{L}$ mantém os seus filhos em buckets $B(Y,0), B(Y,1), \ldots, B(Y,\Delta(Y))$. Para cada i em $[0..\Delta(Y)]$, $B(Y,i)$ contém os filhos X de Y tal que

$$i\delta(Y)/2 \leq \min\{d(v) : v \in Q \cap X\} \leq (i+1)\delta(Y)/2.$$

O bucket $B(Y,\Delta(Y))$ contém os filhos X de Y tal que $Q \cap X = \emptyset$ e $U \cap X \neq \emptyset$.

Em cada iteração em que ocorre o caso 2B, para escolher o filho maduro X' de X a ser visitado, basta encontrar o menor k em $[0..\Delta(X)-1]$ tal que $B(X,k)$ é não-vazio. Devido ao invariante da monotonicidade local relaxada (th1), os elementos maduros podem ser escolhidos pelo algoritmo a medida que os buckets são visitados na ordem $B(X,0), B(X,1), \ldots, B(X,\Delta(X)-1)$. As operações principais envolvendo os buckets são remoções e inserções. Suponha que cada bucket é representado através de uma lista ligada. Assim, cada operação de remoção e inserção de filhos de um elemento X em seus buckets pode ser realizada, no modelo RAM, em tempo constante: para determinar o bucket $B(X,k)$ que contém um certo filho X' de X basta computar $k = \lfloor 2\min\{d(v) : v \in X' \setminus S\}/\delta(X) \rfloor$. Portanto, o consumo total de tempo das operações envolvendo os buckets é proporcional a

$$n + m + \sum_{X \in \mathcal{L}^*} (\Delta(X) + 1),$$

onde $\mathcal{L}^*$ é o conjunto dos elementos internos de $\mathcal{L}$. Na simulação do algoritmo de Thorup, ilustrada nas figuras 6.10 e 6.11, os buckets são representados pelo vetor ao lado da decomposição hierárquica.

O valor $\Delta(X)$ deve ser um limitante superior para o número de buckets associado a X. É suficiente tomarmos

$$\Delta(X) := \lceil 2 \sum_{uv \in T(X)} c(u,v)/\delta(X) \rceil \tag{6.1}$$

onde (V,T) é a floresta que satisfaz (h3) e foi construída durante o pré-processamento. O lema a seguir mostra que com a definição de $\Delta(X)$ conforme a equação 6.1, o número total de buckets mantidos pelo algoritmo é inferior a $12n$. Portanto, o consumo total de tempo e espaço das operações envolvendo os buckets é proporcional a

$$n + m + \sum_{X \in \mathcal{L}^*} (\Delta(X) + 1) < n + m + 12n = 13n + m = O(n + m).$$

Lema 6.2 (número de buckets): *Seja $\mathcal{L}$ a δ-decomposição V-hierárquica de (V, A) em relação a c e (V,T) uma floresta gerada de (V, A) tal que δ e (V,T) satisfazem (h3). O número máximo de buckets mantidos pelo algoritmo de Thorup é*

$$\sum_{X \in \mathcal{L}^*} (\Delta(X) + 1) < 12n,$$

onde $\mathcal{L}^*$ é o conjunto dos elementos internos de $\mathcal{L}$.

Demonstração: Para cada elemento X em $\mathcal{L}^*$ temos que

$$\Delta(X) + 1 \leq 2 + 2 \sum_{uv \in T(X)} c(u,v)/\delta(X).$$

Como cada elemento X de $\mathcal{L}$ tem pelo menos dois filhos então $\mathcal{L}$ tem no máximo $2n - 1$ elementos, então

$$\begin{aligned}
\sum_{X \in \mathcal{L}^*} (\Delta(X) + 1) &\leq \sum_{X \in \mathcal{L}^*} 2 + 2 \sum_{X \in \mathcal{L}^*} \sum_{uv \in T(X)} c(u,v)/\delta(X) \\
&\leq 4n + 2 \sum_{X \in \mathcal{L}^*} \sum_{uv \in T(X)} c(u,v)/\delta(X) \\
&= 4n + 2 \sum_{uv \in T} \sum_{X : uv \in T(X)} c(u,v)/\delta(X).
\end{aligned}$$

Considere uma aresta uv em T. Seja $X_0, \ldots, X_t$ uma ordenação dos elementos de $\mathcal{L}$ que contém u e v tal que X_{i+1} é filho de X_i, para $i = 0, \ldots, t-1$. Temos que

$$\begin{aligned}
\sum_{X : uv \in T(X)} c(u,v)/\delta(X) &< \sum_{X : uv \in T(X)} 2\delta(X_t)/\delta(X) \\
&= 2(\delta(X_t)/\delta(X_t) + \delta(X_t)/\delta(X_{t-1}) + \cdots + \delta(X_t)/\delta(X_0)) \\
&< 2(1 + 1/2 + 1/4 + \cdots) \\
&< 4,
\end{aligned}$$

onde a primeira desigualdade é devida a (h3) e onde a segunda é devida ao fato que

$$\delta(X_{i+1}) \leq \delta(X_i)/2, \text{ para } i = 0, \ldots, t-1.$$

Logo,

$$\begin{aligned} 4n + 2 \sum_{uv \in T} \sum_{X : uv \in T(X)} c(u,v)/\delta(X) &< 4n + 2 \sum_{uv \in T} 4 \\ &< 4n + 8n \\ &= 12n \end{aligned}$$

∎

Número de iterações

O caso 1A ocorre apenas uma vez. Inicialmente, a δ-decomposição V-hierárquica $\mathcal{L}$ tem no máximo $2n-1$ elementos. Logo, os casos 1B e 2A juntos ocorrem no máximo $2n$ vezes, pois em cada ocorrência de um desses casos um elemento é removido de $\mathcal{L}$.

Para estimarmos o número de ocorrências dos casos 1C e 2B, basta considerarmos o número de operações empilha e desempilha, de elementos não folhas, realizadas em L. Consideraremos apenas o número de operações desempilha, já que o número de operações empilha é um a mais do que o número de operações desempilha (o algoritmo pára com $L = \langle V \rangle$). Quando um elemento não-folha X é empilhado em $L = \langle X_0, X_1, \ldots, X_t \rangle$ temos que X é maduro e portanto, $Q \cap X \neq \emptyset$ e

$$\min\{d(v) : v \in Q \cap X\} \leq \min\{d(v) : v \in Q \cap X_t\} + \delta(X_t)/2.$$

O elemento X é desempilhado de $\mathcal{L}$ por uma das seguintes razões:

(r1) $Q \cap X = \emptyset$; ou

(r2) $\min\{d(v) : v \in Q \cap X\} > \min\{d(v) : v \in Q \cap X_t\} + \delta(X_t)/2$.

O número total de vezes que desempilhamos um elemento pela razão (r1) é no máximo n (o número de elementos internos de $\mathcal{L}$ é no máximo n).

Sempre que um elemento é desempilhado pela razão (r2), este elemento é removido de um bucket e inserido em outro bucket de X_t. Como o número total de operações envolvendo buckets é limitado por $13n+m$, então o número total de vezes que um elemento é desempilhado pela razão (r2) é no máximo $13n+m$. Portanto, o número total de ocorrências dos casos 1C e 2B juntos é no máximo $2(n+13n+m) = 28n+2m$.

Resumindo, o número total de iterações realizadas pelo algoritmo é inferior a $30n+2m+1 = O(n+m)$. Como cada iteração consome tempo amortizado constante temos o seguinte teorema.

Teorema 6.3 (consumo de tempo): *O algoritmo de Thorup, quando executado, no modelo RAM, em um grafo com n vértices e m arcos, gasta tempo $O(n+m)$.* ∎

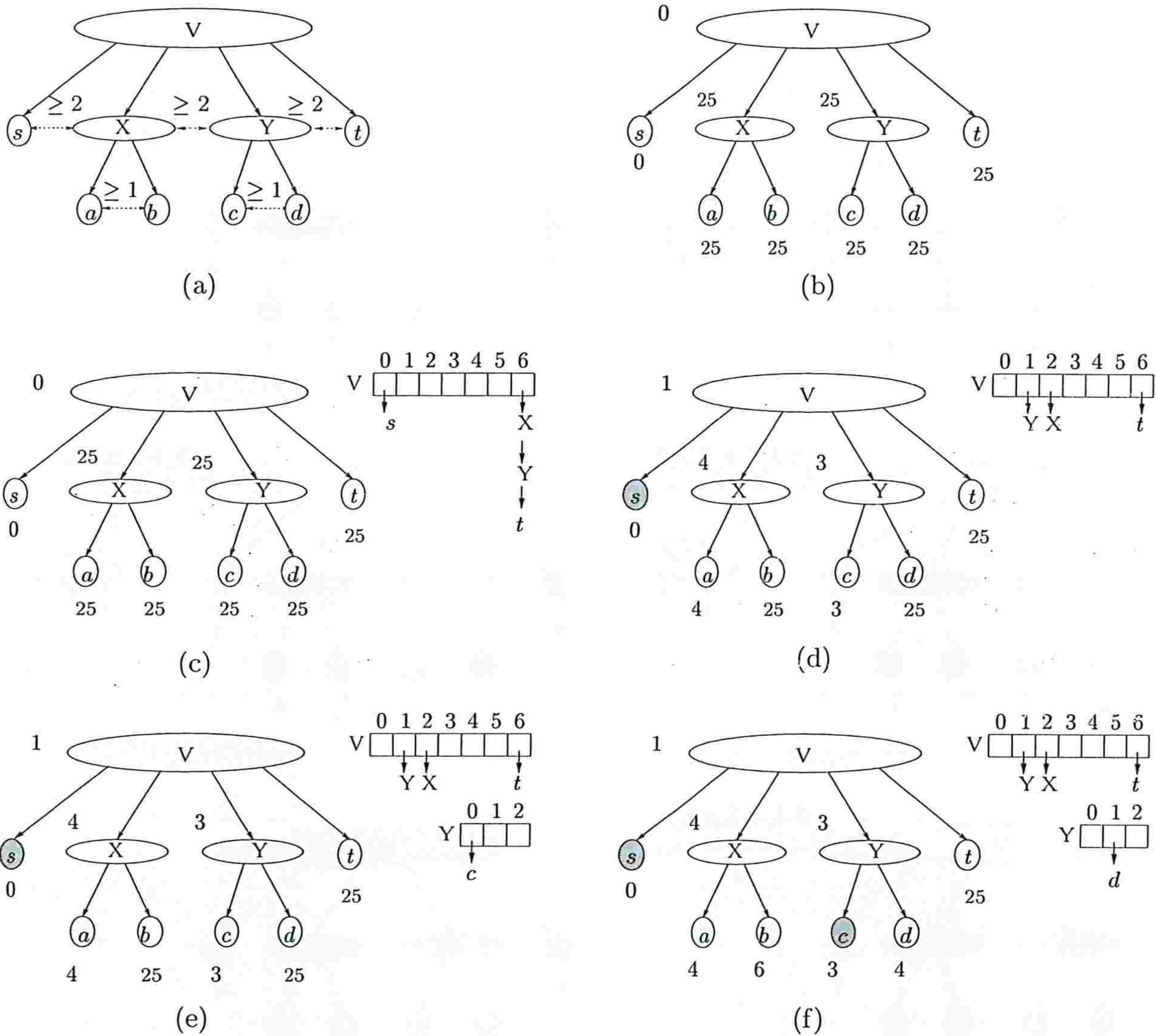


Figura 6.10: Execução do algoritmo de Thorup. (a) exibe a representação arbórea da decomposição V-hierárquica do grafo da figura 6.5(a). O vértice inicial é o s. (b) mostra a situação no início da primeira iteração. O número próximo a um elemento é o seu potencial. Na decomposição hierárquica, os elementos em cinza são os já visitados. Se forem elementos internos significam que já possuem bucket. Os pretos são os removidos da representação arbórea. (c) mostra a situação após criar o bucket de V e (d) a situação após examinar $\{s\}$. Note que nos elementos que já possuem bucket, o potencial mínimo é a primeira posição não-vazia do bucket. Em (e) ocorre a operação de empilhar o filho maduro Y de V (crie bucket de Y). Em (f), $\{s\}$ é empilhado e depois examinado.

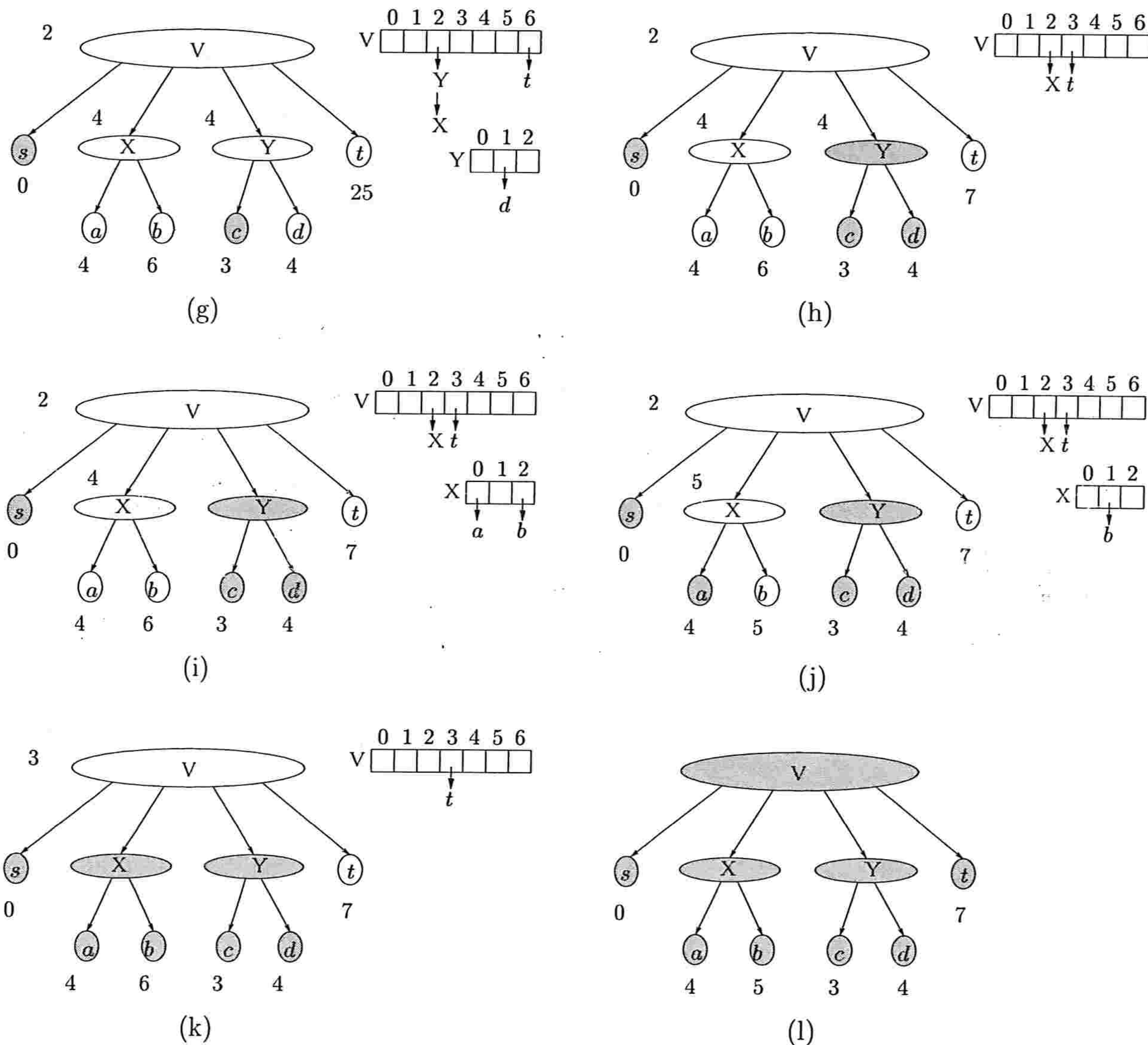


Figura 6.11: Execução do algoritmo de Thorup (continuação). Como não havia elementos em $B(\text{Y}, 0)$ soma-se um ao mínimo de Y, como mostrado em (g). Assim, Y não é mais maduro. Logo, foi necessário mover Y de $B(\text{V}, 1)$ para $B(\text{Y}, 2)$. Por não haver elementos em $B(\text{V}, 1)$ o mínimo de V é adicionado em um. No momento em (g) é possível empilhar novamente V e em seguida examinar $\{d\}$. As figuras (i), (j), (k) e (l) são repetições dos casos anteriores.

6.7 Construção da decomposição hierárquica

Resumidamente, os ingredientes utilizados para construir a δ-decomposição V-hierárquica $\mathcal{L}$ de (V, A) em relação a c, em tempo linear, no modelo RAM, são:

- Encontrar uma árvore geradora mínima em tempo linear. Nesse ponto, é utilizado o algoritmo de Fredman e Willard [18];
- Conseguir executar cada operação de união de conjuntos disjuntos em tempo constante. Para isso, é utilizado o algoritmo de Gabow e Tarjan [19]; e
- Ordenar em tempo linear. Neste caso, é utilizado o algoritmo de Andersson, Hagerup, Nilsson e Raman [4].

Seja (V, A) um grafo e uma função comprimento simétrica c de A em $\mathbb{Z}_{>}$. O primeiro passo na construção da δ-decomposição V-hierárquica $\mathcal{L}$ é construir uma árvore geradora mínima de (V, A). Isso pode ser feito em tempo linear, no modelo RAM, conforme descrito por Fredman e Willard [18]. O método desenvolvido por eles, é uma aplicação direta de uma estrutura de dados, chamada *atomic heap*, que realiza as operações da fila de prioridade, insert, delete-min e decrease-key em tempo amortizado constante.

A idéia do algoritmo é semelhante ao de Dijkstra (capítulo 3). A partir de um vértice inicial s, encontra um caminho de s até t, para cada vértice t em V que é acessível a partir de s. Neste caso, os caminhos encontrados não são necessariamente os mínimos, o importante é que a soma das arestas da arborescência determinada pela função predecessor tenha comprimento mínimo. Da mesma forma que o algoritmo de Dijkstra, cada iteração começa com uma função potencial d, uma função predecessor ψ, e partes S, Q e U de V. Como Q é implementado através de um atomic heap, é necessário controlar o número de elementos que são inseridos em Q. Esse número limite será $\log n$.

O algoritmo que constrói uma árvore geradora mínima em tempo linear é dividido em dois passos. O primeiro passo consiste no seguinte:

No início da primeira iteração $d(v) = nC + 1$ para cada vértice v, $\psi(v) = v$ para cada vértice v, $S = \emptyset$, $Q = \emptyset$ e $U = V$.

Então, o passo seguinte é repetido até que $U = \emptyset$.

No início da primeira iteração escolha u em U tal que $d(u) = nC + 1$, faça $d(u) := 0$, $Q = \{u\}$ e $U := U \setminus \{u\}$.

Cada iteração consiste no seguinte.

Caso 1: $Q = \emptyset$ ou $|Q| \geq \log n$

Pare.

Caso 2: $Q \neq \emptyset$ e $|Q| < \log n$

Escolha u em Q tal que $d(u)$ seja mínimo.

$S := S \cup \{u\}$.

$Q := Q \setminus \{u\}$.

Para cada arco uv faça

Se $c(u, v) < d(v)$ então

$d(v) := c(u, v)$, $\psi(v) := u$ e remova[3] v de U e acrescente a Q.

Comece nova iteração.

Como as operação em Q são feitas em tempo amortizado constante, devido ao uso do atomic heap, o primeiro passo gasta tempo $O(m + n)$, onde n é o número de vértices e m é o número de arcos.

Para o segundo passo, a parte S é condensada como se fosse um único vértice, resultando em um grafo com $n' = O(m/\log n)$ vértices. Utilizado o mesmo algoritmo acima no grafo condensado, mas trocando o atomic heap por um fibonacci heap (seção 4.3)[4], a árvore geradora mínima é obtida em tempo $O(m + n' \log n') = O(m)$.

A principal motivação para trabalhar com uma árvore geradora mínima em vez do grafo é devido ao método desenvolvido por Gabow e Tarjan [19], que mostra como pré-processar uma árvore, em tempo linear, para tornar possível cada operação *union-find*[5] gastar tempo constante. Essas operações são utilizadas na construção das $\delta(X)$-partições (seção 6.2), onde X é um elemento de $\mathcal{L}$. Do ponto de vista do algoritmo descrito neste capítulo, para cada X em $\mathcal{L}$, $\delta(X)$ é um número da forma 2^k, para algum k. A construção de $\mathcal{L}$, começa construindo partes X, conexas maximais, induzidas por arestas de comprimento em $[2^0..2^1 - 1]$. Cada parte X será um elemento interno de $\mathcal{L}$, onde $\delta(X) = 2^0$. Em seguida, são construídas as partes X, conexas maximais, induzidas por arestas de comprimento $[2^1..2^2 - 1]$, logo $\delta(X) = 2^1$. De maneira geral, a cada passo i, as partes X são formadas pelas arestas de comprimento em $[2^i..2^{i+1} - 1]$, e $\delta(X) = 2^i$.

msb

Observe que os comprimentos da arestas em $[2^i..2^{i+1} - 1]$ possuem o mesmo **most significant bit** (msb), já que o msb de um inteiro x é $\lfloor \log_2 x \rfloor$. Por exemplo, considere o intervalo $[4, 5, 6, 7]$. Então, o msb dos números nesse intervalo é 2.

[3] É possível que v já pertença a Q e não esteja em U. Nesse caso estas últimas duas instruções são redundantes.

[4] Note que agora não é necessário controlar o número de elementos inseridos em Q.

[5] Normalmente essas operações são: *union*, que une dois representantes de conjuntos distintos e *find*, que encontra o representante de um conjunto. Pode-se tomar como referência o livro CLRS [9]

De fato, o que é usado para construir os elementos internos de $\mathcal{L}$, é o msb do comprimento das arestas. Embora o msb não é obtido diretamente, ele pode ser calculado, no modelo RAM, usando-se um número constante de operações. O msb de um número x pode ser calculado da seguinte forma: (1) converta x para um número em ponto flutuante. Conforme o padrão IEEE [29] um número em ponto flutuante é representado internamente pelo computador da seguinte maneira: 23 bits para a fração ou mantissa, f; 8 bits para o expoente, e; e um bit para o sinal, s, como ilustrado na figura 6.12. Além disso, todos os expoentes são armazenados depois de serem adicionados a um valor de deslocamento (ou bias), que é $0 \times$ 7FH em hexadecimal, ou 127 em binário; (2) desloque (shift) 23 (0×17) bits à direita; e (3) subtraia 127 ($0 \times$ 7F).

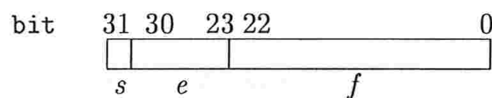


Figura 6.12: Representação de um número em ponto flutuante.

O calculo do msb é feito pela função *msb*. Para poder lidar com o código binário do número ponto flutuante, é utilizado um pequeno trecho em código assembler. Os tutoriais [33, 30] ajudaram a conectar o assembler a linguagem C.

```
86 ⟨Funções auxiliares 32⟩ +≡
int msb(x)
    unsigned long x;
{
register unsigned long lg;
register float f;
if (x ≡ 0) return 0;
f = (float) x;
if (LINUX) {     /* Linux x86 */
  __asm__ ( "mov␣␣␣%1,␣%0\n\t"
  "sar␣␣␣$0x17,␣%0\n\t"
  "sub␣␣␣$0x7F,␣%0\n\t"
  : "=r"(lg)
  : "r"(f)
  );
}
else {     /* SunOS/Solaris */
  __asm__ ( "mov␣␣␣%1,␣%0\n\t"
```

```
"srl␣␣␣%0,␣0x17,␣%0\n\t"
"sub␣␣␣%0,␣0x7F,␣%0\n\t"
: "=r"(lg)
: "r"(f)
);
}
return ((int) lg);
}
```

Agora, seja $a_1, \ldots, a_{n-1}$ a ordenação das arestas da árvore geradora mínima de acordo com o $msb(a_i)$, obtida em tempo linear, utilizando o algoritmo *packed merging*, desenvolvido por Andersson *et al.* [4].

O algoritmo abaixo constrói a δ-decomposição V-hierárquica $\mathcal{L}$ de (V, A) em relação a c em tempo linear, no modelo RAM:

Para i de 1 até $n-2$ faça

Seja $uv = a_i$.

$u := find(u)$ e $v := find(v)$.

$union(u, v)$.

$Y := Y \cup \{find(u)\}$.

Se $msb(a_i) < msb(a_{i+1})$ então

Para todo $v \in Y$ faça

Seja X um elemento de $\mathcal{L}$ que contém as arestas do conjunto representado por v.

$\delta(X) := 2^{msb(a_i)}$.

$Y := \emptyset$.

Em conclusão, temos o seguinte teorema:

Teorema 6.4 (construção de uma decomposição hierárquica): *Seja (V, A) um grafo e uma função comprimento simétrica c de A em $\mathbb{Z}_{>}$. Uma δ-decomposição V-hierárquica de (V, A) em relação a c é construída em tempo $O(m+n)$.* ∎

A figura 6.13 ilustra a construção de uma decomposição hierárquica.

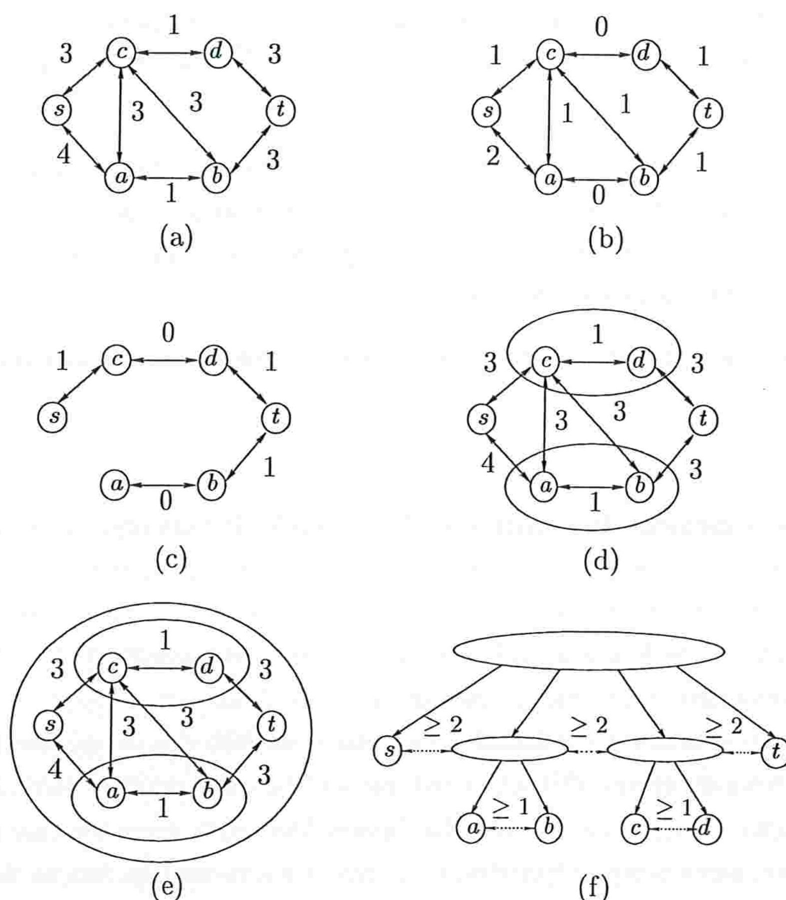


Figura 6.13: (a) mostra um grafo simétrico. Em (b) o mesmo grafo com os comprimentos em relação ao msb. (c) mostra uma possível árvore geradora mínima em relação ao msb dos comprimentos. As curvas da figura (d) representam um nível da δ-decomposição V-hierárquica. (e) mostra a δ-decomposição V-hierárquica do grafo em (a). (f) ilustra a correspondente representação arbórea.

6.8 Versão em CWEB

Esta implementação do algoritmo de Thorup, difere do projeto do algoritmo linear de Thorup [39], mas aplica as mesmas idéias. Por exemplo, não são utilizados os atomic heaps desenvolvido por Fredman e Tarjan [18]. Do ponto de vista prático, acredita-se que mesmo fazendo uso dessa estrutura, não proporcionaria melhorias computacionais, pois conforme Thorup [39] menciona, os atomic heaps são definidos somente para $n > 2^{12^{20}}$ e seu interesse é principalmente teórico.

Conforme visto na seção 6.6, uma implementação do algoritmo de Thorup envolve resolver três problemas: (p1) construção da δ-decomposição V-hierárquica $\mathcal{L}$ de (V, A) em relação a c; (p2) atualizar os potenciais e manter $\min\{d(v) : v \in Q \cap X\}$ para cada X em $\mathcal{L}$; e (p3) escolher o próximo elemento maduro para ser visitado.

Na implementação apresentada, os problemas são solucionados da seguinte maneira:

Construção da decomposição hierárquica. A construção da δ-decomposição V-hierárquica, é feita a partir da árvore geradora mínima do grafo dado. Lembrando que o importante é o msb do comprimento dos arcos (seção 6.7). O algoritmo utilizado para resolver o problema AGM é o de Kruskal utilizando a estrutura *union-find* [9]. No algoritmo de Kruskal é necessário a ordenação dos arcos, que é feita utilizando-se um bucket sort. Assim, como os arcos são ordenados em relação ao msb dos comprimentos, essa etapa pode ser feita em tempo $O(\log C + m)$, onde C é o comprimento do maior arco. Após a ordenação, o algoritmo de Kruskal/*union-find*, para encontrar uma árvore geradora mínima, gasta tempo $O(m\alpha(m, n))$, onde α é a inversa da função de Ackermann. Como a construção da decomposição hierárquica é feita ao mesmo tempo que se determina a árvore geradora mínima, o tempo gasto é $O(\log C + \alpha(m, n)m)$.

Atualização dos potenciais. Na decomposição hierárquica, o $\min\{d(v) : v \in Q \cap X\}$, para algum X em $\mathcal{L}$ é igual ao $\min\{d(v) : v \in Q \cap X'\}$, onde X' é um descendente folha de X. Como $\min\{d(v) : v \in Q \cap X'\}$ pode diminuir, os elementos X, ancestrais de X' devem ser atualizados. Na implementação, utilizamos o método mais natural, isto é, quando $\min\{d(v) : v \in Q \cap X'\}$ diminui, o novo valor é propagado para cima enquanto necessário. No pior caso, o tempo gasto na atualização dos potenciais é, claramente, a altura de $\mathcal{L}$, que é limitada por $\log r$, onde r é a razão entre o maior e o menor comprimento de um arco. Portanto, o tempo gasto, em cada atualização é $O(\log r)$. Como observado por Pettie e Ramachandran [32] na prática, poucos ancestrais precisam ser atualizados.

Escolha de um elemento maduro. A escolha de um elemento maduro é feita através da utilização de buckets, conforme descrito na seção 6.6. Portanto, o tempo total gasto nessa etapa é $O(m+n)$.

A implementação do algoritmo de Thorup está dividida em dois blocos:

```
88  ⟨ Algoritmo de Thorup 88 ⟩ ≡
      ⟨ Construa a decomposição hierárquica do grafo 91 ⟩
      ⟨ Examine os vértices utilizando a decomposição hierárquica 106 ⟩
```

Este código é usado no bloco 9.

O bloco a seguir, corresponde a decomposição hierárquica.

```
89  ⟨ Definições 12 ⟩ +≡
    #define prox_sort a.A    /* próximo arco na ordem será a→prox_sort */
    #define from b.V    /* uma arco vai de a→from para a→tip */
```

Os arcos são ordenados, de acordo com o msb dos comprimentos, utilizando-se um bucket sort. A ordenação dos arcos é mantida no vetor *sort*. Então, cada posição i de *sort* mantém os arcos com comprimento 2^i.

```
90  ⟨ Variáveis globais 13 ⟩ +≡
      Arc *sort[100];    /* cabeças de listas (100 é suficiente) */
```

O decomposição hierárquica é mantida, em *arb*, na forma da sua representação arbórea.

```
91  ⟨ Construa a decomposição hierárquica do grafo 91 ⟩ ≡
      Graph *krusk(g)
          Graph *g;
      { ⟨ Variáveis de krusk 92 ⟩
        ⟨ Ordene os arcos colocando-os nos buckets sort[0], ..., sort[msbC] 93 ⟩
        ⟨ Coloque cada vértice em um conjunto distinto e inicializa a arborescência 95 ⟩
        conj = n;
        for (i = 0; i ≤ msbC; i++) {
          for (a = sort[i]; a ∧ (conj > 1); a = a→prox_sort) {
            u = a→from;
            v = a→tip;
            ⟨ Caso u e v estejam no mesmo conjunto, comece nova iteração 97 ⟩
            ⟨ Faça a união dos conjuntos que contém u e v e construa a arborescência 98 ⟩
```

```
        if (CONEXO) conj --;
      }
    }
    ⟨ Devolva a arborescência 102 ⟩
  }
```

Este código é usado no bloco 88.

92 ⟨ Variáveis de *krusk* 92 ⟩ ≡

```
register Graph *arb;    /* estrutura arbórea */
register Arc *a, *aux;
register unsigned long msbC;    /* msb(C) */
register unsigned long n;
register unsigned long conj;
register long i;
register long delta_atual;    /* δ-partição que está em construção */
register Vertex *u, *v, *w, *arbv;
register Vertex *livre;    /* posição de arb que pode ser usada */
```

Este código é usado no bloco 91.

O algoritmo de Kruskal seleciona os arcos, um a um, em ordem crescente de comprimento e de forma que não formem ciclos. Cada posição i de *sort* guarda os arcos que tem o *msb* do comprimento igual a i.

93 ⟨ Ordene os arcos colocando-os nos buckets $sort[0], \ldots, sort[msbC]$ 93 ⟩ ≡

```
msbC = msb(C);
for (i = 0; i ≤ msbC; i++) sort[i] = Λ;
n = g→n;
for (v = g→vertices; v < g→vertices + n; v++) {
  for (a = v→arcs; a; a = a→next) {    /* só guarda os arcos que apontam pra frente */
    if (a→tip < v) continue;
    a→from = v;
    i = msb(a→len);
    a→prox_sort = sort[i];
    sort[i] = a;
  }
}
```

Este código é usado no bloco 91.

A construção de *arb* envolve o uso dos ponteiros: *rep*, que é usado pelos vértices em g indicando qual elemento, em *arb*, eles pertencem; *bksize* indica o número máximo de posições de um bucket e *delta* as δ-partições.

```
94 ⟨ Definições 12 ⟩ +≡
#define rep x.V     /* aponta para o representante do conjunto */
#define bksize r.I  /* soma dos arcos de um elemento (Δ) */
#define delta w.I   /* δ de uma partição da decomposição hierárquica */
```

Inicializações dos ponteiros dos vértices, do grafo dado g, e daqueles responsáveis pela construção da representação arbórea *arb*.

```
95 ⟨ Coloque cada vértice em um conjunto distinto e inicializa a arborescência 95 ⟩ ≡
  arb = gb_new_graph(n);     /* Nomeia aos vértices da arborescência */
  for (arbv = arb→vertices, v = g→vertices, i = 0; arbv < arb→vertices + n; arbv++, v++) {
    arbv→elemento = arbv→rep = arbv;
    v→bksize = arbv→bksize = 0;
    v→elemento = v→rep = v;
    v→delta = VERTICE_DE_G;
  }
  livre = arb→vertices;     /* próxima posição livre (novo elemento) */
```

Este código é usado no bloco 91.

Os dois blocos seguintes, dizem respeito à estrutura *union-find*.
find: A função *find_set* encontra o representante de um conjunto.

```
96 ⟨ Funções auxiliares 32 ⟩ +≡
  Vertex *find_set(v)
      Vertex *v;
  {
    if (v ≠ v→rep) v→rep = find_set(v→rep);
    return (v→rep);
  }
```

union: Verifica se o representante do conjunto que contém o vértice u é o mesmo que o do vértice v. Em caso afirmativo, significa que u e v pertencem ao mesmo conjunto. Logo, não é possível adicionar o arco uv, aos arcos que estão compondo a árvore geradora mínima, pois com este, um ciclo seria formado.

97 ⟨ Caso u e v estejam no mesmo conjunto, comece nova iteração 97 ⟩ ≡

$u = \mathit{find_set}(u)$;
$v = \mathit{find_set}(v)$;
if $(u \equiv v)$ **continue**;

Este código é usado no bloco 91.

Caso o representante do conjunto que contém u e o que contém v sejam diferentes, o arco uv irá compor a árvore geradora mínima, e os representantes dos conjuntos serão unidos, de três possíveis maneiras, conforme ilustrado na figura 6.14.

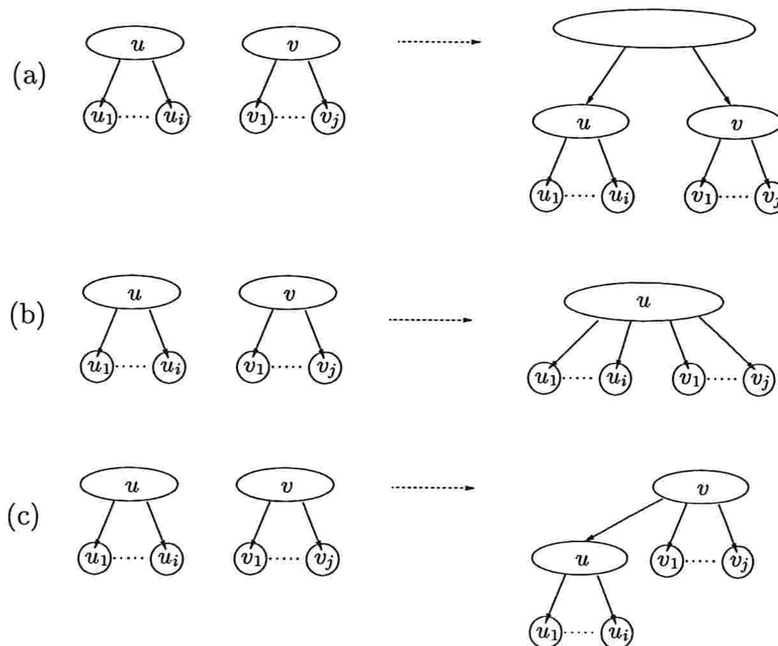


Figura 6.14: As figuras (a), (b) e (c) mostram os três casos possíveis na união de dois elementos. Em (a) um novo elemento é criado e tem como filhos os elementos u e v. (b) é a situação em que os filhos de v se tornam filhos de u. A figura (c) ilustra o caso em que u é colocado como filho de v.

98 ⟨ Faça a união dos conjuntos que contém u e v e construa a arborescência 98 ⟩ ≡

if $(u\text{→}\mathit{delta} \neq$ `VERTICE_DE_G`$)$ {
  if $((v\text{→}\mathit{delta} \equiv$ `VERTICE_DE_G`$) \lor (u > v))$ {
    $w = u$;
    $u = v$;
    $v = w$;

```
      }
    }      /* u vem antes */
    delta_atual = i;
    if ((u→delta ≤ v→delta) ∧ (v→delta < delta_atual)) {
      ⟨ Crie um novo elemento 99 ⟩
    }
    else if ((u→delta ≡ v→delta) ∧ (v→delta ≡ delta_atual)) {
      ⟨ Junte u e v em um único elemento 100 ⟩
    }
    else if ((u→delta < v→delta) ∧ (v→delta ≡ delta_atual)) {
      ⟨ Coloque u como filho de v 101 ⟩
    }
```

Este código é usado no bloco 91.

```
99  ⟨ Crie um novo elemento 99 ⟩ ≡
    livre→delta = delta_atual;
    gb_new_arc(livre, v);
    gb_new_arc(livre, u);
    u→elemento = v→elemento = livre;
    u→rep = v→rep = livre;
    livre→bksize = u→bksize + v→bksize + a→len;
    livre++;
    if (REPORT) nelementos++;
```

Este código é usado no bloco 98.

```
100 ⟨ Junte u e v em um único elemento 100 ⟩ ≡
    for (aux = v→arcs; aux; aux = aux→next) {
      gb_new_arc(u, aux→tip);      /* transfere os arcos de v para u */
      aux→tip→elemento = u;
    }
    v→rep = u;      /* fazendo isso não preciso ajustar os ponteiros rep dos níveis abaixo */
    v→arcs = Λ;
    u→bksize += a→len + v→bksize;
    if (REPORT) nelementos--;
```

Este código é usado no bloco 98.

101 ⟨ Coloque u como filho de v 101 ⟩ ≡

```
gb_new_arc(v, u);
u→elemento = v;
u→rep = v;
v→bksize += a→len + u→bksize;
```

Este código é usado no bloco 98.

Antes de retornar a representação arbórea *arb* da decomposição hierárquica de g, é preciso calcular os valores de *bksize*, que limitam o tamanho dos buckets de cada elemento.

102 ⟨ Devolva a arborescência 102 ⟩ ≡

```
if (livre ≡ arb→vertices) return Λ;   /* não tem nenhum elemento */
for (v = arb→vertices; v ≤ livre; v++) {
  if (v→arcs ≠ Λ) v→bksize = ((v→bksize ≫ v→delta) + 2);
}
return arb;
```

Este código é usado no bloco 91.

O bloco a seguir, corresponde e examinar os vértices, fazendo uso da decomposição hierárquica.

103 ⟨ Variáveis globais 13 ⟩ +≡

```
int desempilha = 0;   /* número de vezes que o programa desempilha vértices */
int atualiza_acima = 0;   /* número de vezes que subiu na arborescência para atualizar */
int nao_maduro = 0;   /* número de vezes que os elementos se tornaram não maduros */
int nelementos = 0;   /* número de nós da arborescência (sem contar os vértices de g) */
Vertex **BK;   /* Bucket */
```

104 ⟨ Definições 12 ⟩ +≡

```
#define dx q.I   /* menor valor de um elemento */
#define prox_topo x.V   /* próximo elemento no topo da pilha */
#define id(i) (i − arb→vertices)
#define BK(i, j) (*(BK + id(i)) + j)
```

A função *insere_elemento*, insere v no bucket BK(u, i).

105 ⟨ Funções auxiliares 32 ⟩ +≡

```
void insere_elemento(arb, v, u, i)
    Graph *arb;
    Vertex *v;
    Vertex *u;
    unsigned long i;
{
  v→ant_elemento = BK(u, i);
  v→prox_elemento = BK(u, i)→prox_elemento;
  (BK(u, i)→prox_elemento)→ant_elemento = v;
  BK(u, i)→prox_elemento = v;
}
```

A implementação tenta ser a mais próxima possível da descrição (seção 6.3).

106 ⟨ Examine os vértices utilizando a decomposição hierárquica 106 ⟩ ≡

```
void thorup(g, arb, s)
    Graph *g;
    Graph *arb;
    Vertex *s;
{ ⟨ Variáveis de thorup 107 ⟩
  ⟨ Inicializações para thorup 108 ⟩
  ⟨ Encontre a raiz root da arborescência 109 ⟩
  topo = root;    /* root fica no topo da pilha */
  while (1) {
    x = topo;
    if ((x→arcs ≡ Λ) ∧ (x→delta ≠ VERTICE_DE_G)) {
        /* x é uma folha e não é madura (não pode ser um vértice de g) */
      if (x ≡ root) {    /* Caso 1A: raiz não é madura */
        free(BK[id(x)]);    /* já posso desalocar o bucket de x */
        break;
      }
      else {    /* Caso 1B: x não é maduro e é folha */
        ⟨ Desempilhe 115 ⟩
        remove_elemento(x);    /* remove x do bucket */
        free(BK[id(x)]);    /* já posso desalocar o bucket de x */
        continue;
      }
```

```
        }
        if (x→delta ≡ VERTICE_DE_G) {      /* Caso 2A: x é vértice de g */
          ⟨ Examine a folha x 111 ⟩
          ⟨ Desempilhe 115 ⟩
          remove_elemento(x);     /* remove x do bucket */
          continue;
        }
        if (x→status ≡ NAO_VISITADO) {
              /* elemento ainda não foi visitado (ainda não tem bucket) */
          ⟨ Crie um bucket para x 110 ⟩
        }
        fmaduro = BK(x, x→dist − x→dx)→prox_elemento;     /* verifica se x tem filho maduro */
        if (fmaduro ≠ BK(x, x→dist − x→dx)) {
              /* Caso 2B: x é maduro e fmaduro é seu filho maduro */
          ⟨ Empilhe o filho maduro de x 114 ⟩
          continue;
        }
        x→dist++;
        if (x→dist ≡ (x→bksize + x→dx))     /* o bucket de x está vazio (x virou folha) */
          x→arcs = Λ;
        else {     /* Caso 1C: x não é maduro e não é folha */
          dif = (x→elemento)→delta − x→delta;
          i = x→dist ≫ dif;
          k = (x→dist − 1) ≫ dif;
          if (i > k) {
            ⟨ Mude x de bucket 113 ⟩
            ⟨ Desempilhe 115 ⟩
          }
        }
      }
    }
```

Este código é usado no bloco 88.

107 ⟨ Variáveis de *thorup* 107 ⟩ ≡

```
register Arc *a;
register Vertex *v, *x, *w, *arbv;
register Vertex *topo, *root, *fmaduro;
register unsigned long n;
```

```
register unsigned long i, k;
register unsigned long dist_antes;
register unsigned long dif;
```

Este código é usado no bloco 106.

108 ⟨ Inicializações para *thorup* 108 ⟩ ≡

```
if (arb ≡ Λ) {
  printf("%s\n", err_message[ERROR_3]);
  exit(0);
}
n = g→n;
if ((BK = (Vertex **) malloc(n * sizeof(Vertex *))) ≡ Λ) {
  printf("%s\n", err_message[ERROR_2]);
  exit(0);
}
for (v = g→vertices, arbv = arb→vertices; v < g→vertices + n; v++, arbv++) {
  v→ant_elemento = v→prox_elemento = v→pred = v;
  arbv→dist = v→dist = infinito;
  arbv→status = v→status = NAO_VISITADO;
  arbv→ant_elemento = arbv→prox_elemento = arbv;
}
num_exam = 0;
atualiza_fp = 0;
```

Este código é usado no bloco 106.

109 ⟨ Encontre a raiz *root* da arborescência 109 ⟩ ≡

```
s→dist = 0;
for (arbv = s; arbv ≠ arbv→elemento; arbv = arbv→elemento) {
  arbv→dist = 0;
}
arbv→dist = 0;
root = arbv;
```

Este código é usado no bloco 106.

110 ⟨ Crie um bucket para *x* 110 ⟩ ≡

```
x→dist = x→dx = x→dist ≫ x→delta;
```

```
x->status = VISITADO;
if ((BK[id(x)] = (Vertex *) malloc((x->bksize) * sizeof(Vertex))) ≡ Λ) {
  printf("%s\n", err_message[ERROR_2]);
  exit(0);
}
for (i = 0; i < x->bksize; i++) {      /* Inicializa cabeça de lista */
  BK(x, i)->ant_elemento = BK(x, i)->prox_elemento = BK(x, i);
}
for (a = x->arcs; a; a = a->next) {
  v = a->tip;
  i = (v->dist >> x->delta);      /* posição do bucket para ser inserido */
  i -= x->dx;      /* preciso fazer o deslocamento, pois só aloquei x->bksize posições */
  if (i < x->bksize) insere_elemento(arb, v, x, i);      /* insere v no BK */
}
```

Este código é usado no bloco 106.

111 ⟨Examine a folha x 111⟩ ≡

```
for (a = x->arcs; a; a = a->next) {
  v = a->tip;
  if (v->dist - x->dist > a->len) {      /* se a função potencial não é viável */
    atualiza_fp++;
    dist_antes = v->dist;      /* guardo o valor do potencial não viável */
    v->dist = x->dist + a->len;
    v->pred = x;
    ⟨Atualize o potencial dos elementos 112⟩
  }
}
num_exam++;
```

Este código é usado no bloco 106.

112 ⟨Atualize o potencial dos elementos 112⟩ ≡

```
while (1) {
  w = v->elemento;
  if (w->status ≡ VISITADO) {
    i = (v->dist >> w->delta);
    i -= w->dx;
    if (i < w->bksize) {
```

```
        dist_antes = (dist_antes ≫ w→delta) − w→dx;
        if (i < dist_antes) {      /* remove v do bucket */
          remove_elemento(v);      /* move o elemento v para o bucket(w,i) */
          insere_elemento(arb, v, w, i);
        }
      }
      break;
    }
    if (v→dist ≥ w→dist) break;      /* não preciso atualizar o mínimo */
    dist_antes = w→dist;
    w→dist = v→dist;      /* atualiza mínimo do elemento w */
    if (REPORT) atualiza_acima++;
    v = w;
  }
```

Este código é usado no bloco 111.

113
```
⟨ Mude x de bucket 113 ⟩ ≡
  if (x ≠ root) {
    w = x→elemento;
    remove_elemento(x);      /* remove x do bucket */
    i −= w→dx;      /* preciso fazer a correção */
    if (i < w→bksize) insere_elemento(arb, x, w, i);      /* move x para o BK(w,i) */
    else x→arcs = Λ;      /* elemento x não pode ter mais filhos */
    if (REPORT) nao_maduro++;
  }
```

Este código é usado no bloco 106.

114
```
⟨ Empilhe o filho maduro de x 114 ⟩ ≡
  w = topo;
  topo = fmaduro;
  fmaduro→prox_topo = w;
```

Este código é usado no bloco 106.

115
```
⟨ Desempilhe 115 ⟩ ≡
  topo = topo→prox_topo;
  if (REPORT) desempilha++;
```

Este código é usado no bloco 106.

Portanto, essa implementação do algoritmo de Thorup gasta tempo

$$\underbrace{O(\log C + m\alpha(m,n))}_{\text{dec. hierárquica}} + \underbrace{O(m \log r)}_{\text{atualizar}} + \underbrace{O(n+m)}_{\text{examinar}} = O(n + \log C + m\alpha(m,n) + m \log r).$$

Teorema 6.5: *A implementação do algoritmo de Thorup resolve o PCMS em um grafo com n vértices e m arcos em tempo $O(n + \log C + m\alpha(m,n) + m \log r)$, onde $\alpha(m,n)$ é a inversa da função de Ackermann, C é o maior comprimento de um arco e r é a razão entre o maior e o menor comprimento de um arco.* ∎

Capítulo 7

Resultados Experimentais

Atualmente, há um grande interesse em trabalhos relacionados à análise experimental de algoritmos. Em particular, no caso do algoritmo de Dijkstra, podemos citar os artigos de Cherkassky, Goldberg, Radzik e Silverstein [6, 21, 7]. Para algoritmos baseados em "decomposição hierárquica", como o de Thorup, podemos citar o artigo de Petti, Ramachandran e Sridhar [32].

O interesse em experimentação é devido ao reconhecimento de que os resultados teóricos, freqüentemente, não trazem informações referentes ao desempenho do algoritmo na prática. Porém, o campo da análise experimental é repleto de armadilhas, como comentado por Johnson [25]. Muitas vezes, a implementação do algoritmo é a parte mais simples do experimento. A parte difícil é usar, com sucesso, a implementação para produzir resultados de pesquisa significativos.

Segundo Johnson [25], pode-se dizer que existem quatro motivos básicos que levam a realizar um trabalho de implementação de um algoritmo:

(1) Para usar o código em uma aplicação particular, cujo propósito é descrever o impacto do algoritmo em um certo contexto;

(2) Para proporcionar evidências da superioridade de um algoritmo;

(3) Para melhor compreensão dos pontos fortes, fracos e do desempenho das operações algorítmicas na prática; e

(4) Para produzir conjecturas sobre o comportamento do algoritmo no caso-médio sob distribuições específicas de instâncias onde a análise probabilística direta é muito difícil.

Nesta dissertação estamos mais interessados no motivo (3).

7.1 Ambiente experimental

A plataforma utilizada nos experimentos é um PC rodando Linux RedHat 7.1 com um processador Pentium III de 733 MHz e 512MB de memória RAM. Os códigos estão compilados com o gcc versão 2.96 e opção de otimização 03.

Na implementação, as instâncias são obtidas utilizando-se os geradores de grafos aleatórios disponibilizados pelo "The Fifth DIMACS Challenge"[1] que estão acessíveis no endereço

`ftp://cs.amherst.edu/pub/dimacs.`

Essas instâncias são convertidas para o formato do SGB (seção 1.7), que é a plataforma utilizada nas implementações desta dissertação. Essa conversão é feita utilizando-se um "driver" também disponível pelo DIMACS, porém com algumas modificações. Em todos os testes, para cada valor de n e m, onde n é o número de vértices e m o número de arcos, foi obtido a média entre cinco valores. Não é utilizado o gerador do SGB, devido ao fato que o tempo gasto para se gerar um grafo com aproximadamente 2000 vértices e 1×10^6 arcos é em torno de 4 horas e o espaço ocupado por esse grafo em disco é de aproximadamente 50MB. Além disso, estamos interessados em grafos de dimensões um pouco maiores que este.

As implementações comparadas neste experimento são heap (Heap), D-heap (Dheap), fibonacci heap (Fheap), bucket heap (Bheap), radix heap (Rheap) e o algoritmo de thorup (Thorup).

A estimativa do tempo é calculada utilizando-se:

```
#include <time.h>
clock_t start, end;
double time;
start = clock();
/* implementação */
end = clock();
time = ((double) (end - start)) / CLOCKS_PER_SEC;
```

O tempo estimado para a construção da decomposição hierárquica (AGM), realizada pela implementação Thorup, é computado separadamente.

A estimativa da memória consumida por uma implementação é calculada utilizando-se o programa *memtime* da seguinte maneira:

memtime implementação

[1] O tópico era "Priority Queues" e "Dictionaries".

Esse programa está disponível em `http://www.update.uu.se/~johanb/memtime` e foi escolhido devido a facilidade na sua manipulação.

Na estimativa de memória, não foi possível computar separadamente a quantidade de memória gasta na construção da decomposição hierárquica, pois o programa *memtime* informa qual foi o máximo de memória utilizada pelo processo que executa a implementação Thorup.

Nas próximas seções, serão apresentados os desempenhos das implementações em grafos produzidos pelos geradores SPRAND, SPGRID e SPBAD.

7.2 SPRAND

SPRAND gera grafos aleatórios conexos. Inicialmente é criado um ciclo hamiltoniano, e depois são adicionados os demais arcos aleatoriamente. Os parâmetros de entrada para o gerador são especificados da seguinte maneira:

`sprand n m seed [parâmetros opcionais]`, onde

`n` é o número de vértices,
`m` é o número de arcos e
`seed` é a semente do gerador de números aleatórios.

Toda vez que o gerador é executado com os mesmos parâmetros, ele gera o mesmo grafo. Por default, os arcos pertencentes ao ciclo tem comprimento 1 e os demais têm comprimento em [0..10000].

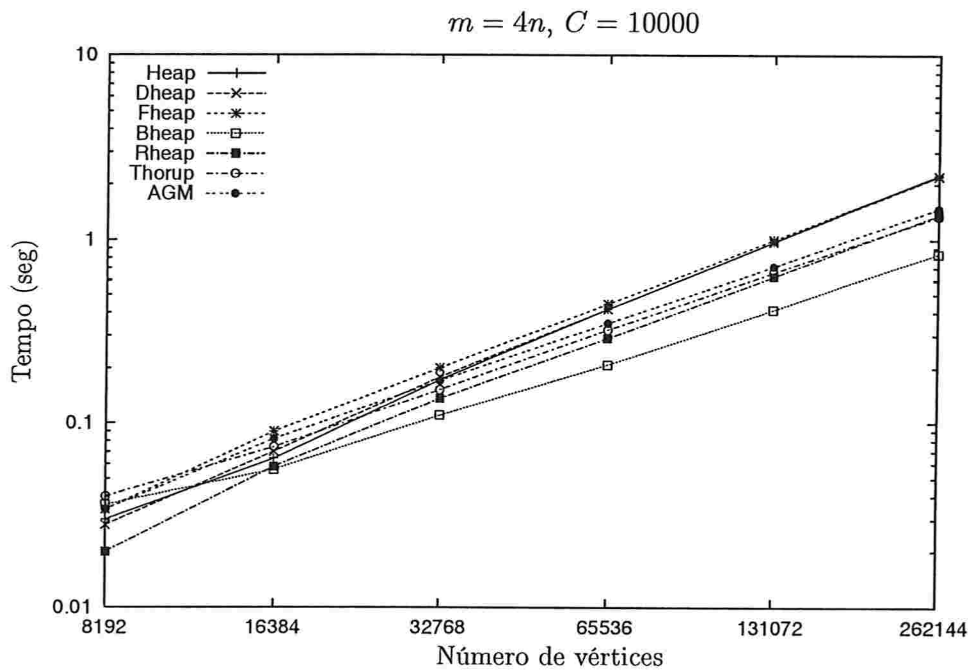
Os experimentos realizados com os grafos gerados por SPRAND se dividem em duas classes: grafos esparsos e grafos densos.

Grafos esparsos gerados por SPRAND

Os grafos desta classe possuem o número de arcos 4 vezes maior que o número de vértices, ou seja, $m = 4n$. A figura 7.1 exibe o tempo gasto pelos algoritmos e a figura 7.2 o quanto de memória foi consumida. Nestes experimentos, n varia de 8192 a 262144, e os comprimentos dos arcos estão em [1..10000].

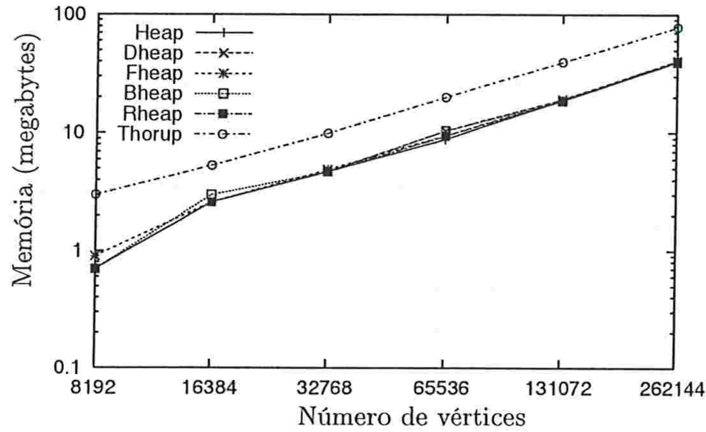
Pode-se observar que o Bheap teve o melhor desempenho. A implementação Thorup foi melhor do que as implementações Heap, Dheap e Fheap, lembrando que o tempo necessário para construir a decomposição hierárquica é computado separadamente. Além disso, nota-se que a correlação entre o tempo e o número de vértices é praticamente linear. A memória consumida por Thorup foi maior que a das demais implementações, mas apenas por um fator constante.

Na figura 7.3 é analisada a sensibilidade dos algoritmos em relação a C, que é o maior comprimento de um arco. Nota-se que Thorup teve um comportamento bem estável. As implementações Heap, Dheap e Fheap foram a mais sensíveis à variação de C.



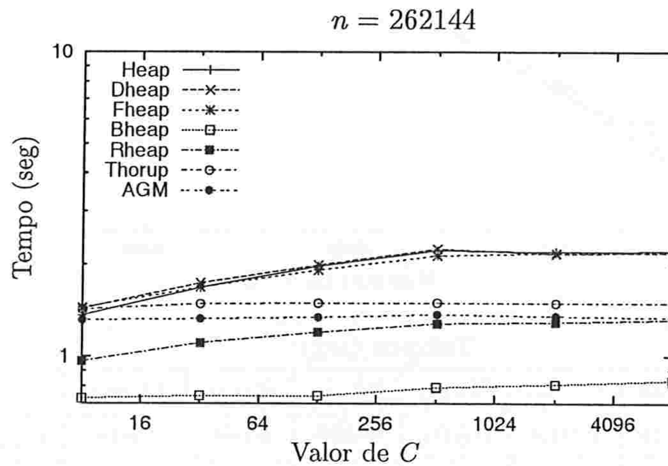
Tempos (seg)							
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8192	0.030	0.028	0.034	0.036	0.020	0.034	0.040
16384	0.064	0.070	0.090	0.056	0.058	0.082	0.074
32768	0.172	0.178	0.200	0.110	0.136	0.170	0.152
65536	0.418	0.418	0.446	0.208	0.290	0.350	0.322
131072	0.976	0.972	1.000	0.414	0.628	0.714	0.662
262144	2.242	2.210	2.218	0.838	1.364	1.474	1.336

Figura 7.1: Número de vértices em relação ao tempo em grafos esparsos gerados por SPRAND.



Memória (megabytes)						
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
8192	0.7	0.7	0.9	0.7	0.7	3.0
16384	2.6	2.6	2.6	3.0	2.6	5.3
32768	4.7	4.7	4.9	4.7	4.7	9.9
65536	8.8	10.4	9.5	10.4	9.4	20.0
131072	18.9	19.3	18.9	18.7	18.7	39.7
262144	40.1	41.0	40.9	40.1	39.9	78.4

Figura 7.2: Memória utilizada quando executado em grafos esparsos gerados por SPRAND.



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	1.362	1.434	1.444	0.730	0.964	1.426	1.314
32	1.678	1.738	1.682	0.742	1.106	1.484	1.326
128	1.974	1.992	1.916	0.742	1.198	1.492	1.342
512	2.224	2.250	2.136	0.790	1.278	1.496	1.370
2048	2.194	2.166	2.178	0.808	1.292	1.490	1.352
8192	2.210	2.186	2.198	0.832	1.316	1.494	1.338

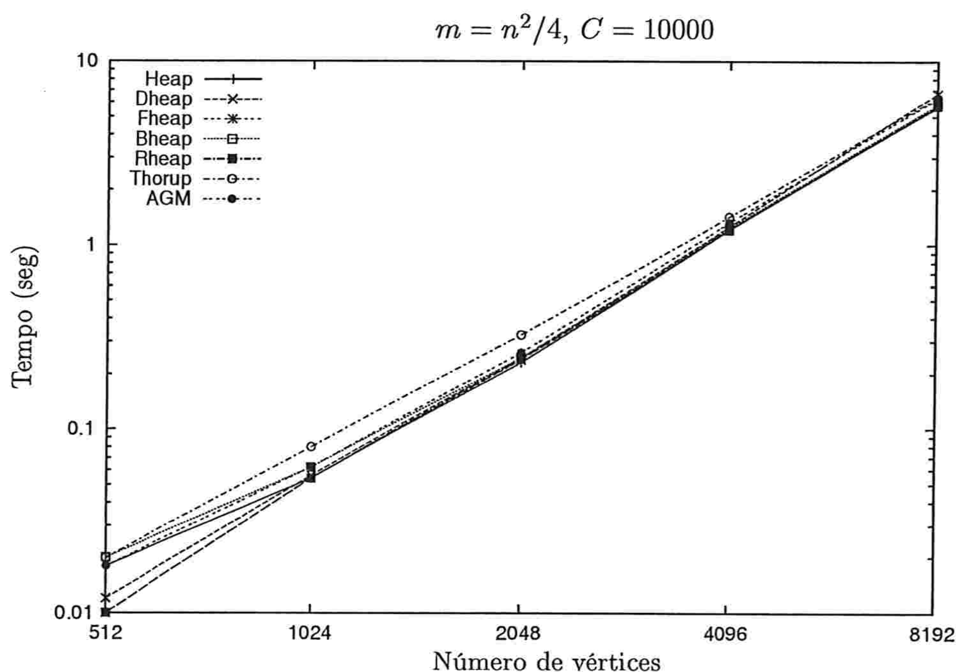
Figura 7.3: Valor de C em relação ao tempo em grafos esparsos gerados por SPRAND.

Grafos densos gerados por SPRAND

Os grafos desta classe possuem $n^2/4$ arcos. A figura 7.4 exibe o tempo gasto pelos algoritmos, e a figura 7.5 o quanto de memória foi consumida. Neste experimentos, n varia de 512 a 8192, e os comprimentos dos arcos são inteiros em [1..10000].

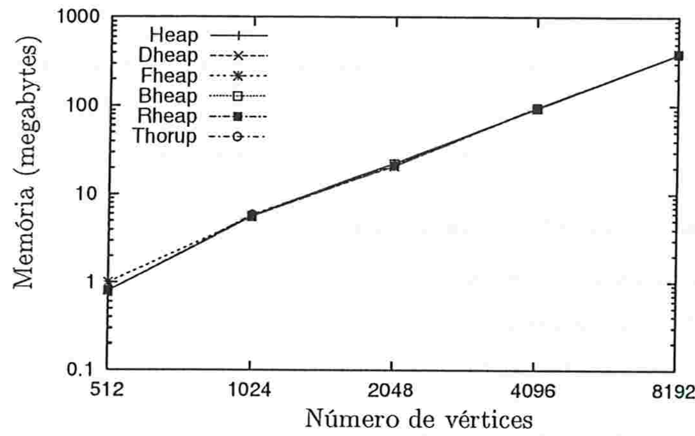
Observa-se que o desempenho das implementações foram muito parecidos, tanto no tempo, quanto no uso da memória.

Na figura 7.6 é analisada a sensibilidade do algoritmo em relação a C, que é o comprimento do maior arco. Nota-se que a construção da decomposição hierárquica (AGM) teve uma pequena variação.



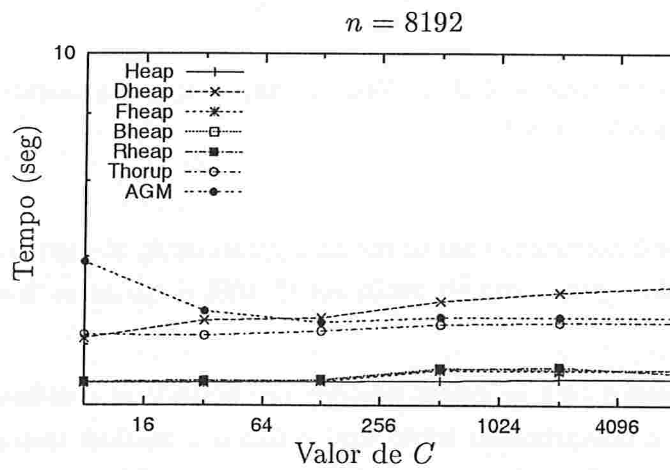
Tempos (seg)							
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.018	0.012	0.010	0.020	0.010	0.018	0.020
1024	0.054	0.056	0.054	0.062	0.054	0.062	0.080
2048	0.230	0.244	0.238	0.244	0.240	0.260	0.326
4096	1.210	1.262	1.220	1.238	1.210	1.326	1.426
8192	5.618	6.658	5.728	5.796	5.722	6.262	6.346

Figura 7.4: Número de vértices em relação ao tempo em grafos densos gerados por SPRAND.



Memória (megabytes)						
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	0.8	0.8	1.0	0.8	0.8	0.8
1024	5.7	5.6	5.6	5.6	5.6	5.9
2048	22.5	20.8	20.8	22.2	21.3	21.2
4096	93.8	94.9	95.4	93.4	95.9	96.0
8192	382.0	385.5	385.3	384.0	382.1	383.9

Figura 7.5: Memória utilizada quando executado em grafos densos gerados por SPRAND.



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	5.618	6.070	5.614	5.620	5.624	6.108	6.938
32	5.620	6.262	5.616	5.630	5.640	6.098	6.364
128	5.626	6.290	5.626	5.644	5.640	6.146	6.240
512	5.632	6.474	5.734	5.758	5.746	6.216	6.294
2048	5.648	6.574	5.732	5.744	5.770	6.236	6.296
8192	5.648	6.644	5.740	5.748	5.712	6.236	6.292

Figura 7.6: Valor de C em relação ao tempo em grafos densos gerados por SPRAND.

7.3 SPGRID

SPGRID gera grafos em forma de grade. Em casos simples, todos os arcos pertencem à grade. Contudo, é possível adicionar mais arcos, tornando a instância mais complicada. Os parâmetros de entrada para o gerador são especificados da seguinte maneira:

`spgrid X Y seed [parâmetros opcionais]`, onde

`X` é o tamanho horizontal da grade;

`Y` é o tamanho vertical da grade; e

`seed` é a semente do gerador de números aleatórios.

Um exemplo de um grafo gerado por SPGRID pode ser visto na figura 7.7.

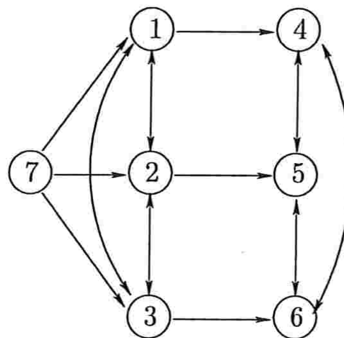


Figura 7.7: Grafo gerado com `spgrid 2 3 1`. Nesse exemplo, os comprimentos dos arcos foram omitidos. O vértice fonte é o $\mathtt{X} \times \mathtt{Y} + 1$.

Toda vez que o gerador é executado com os mesmos parâmetros, ele gera o mesmo grafo. Por default, os comprimento dos arcos verticais estão em [0..100] e dos arcos horizontais estão em [0..10000].

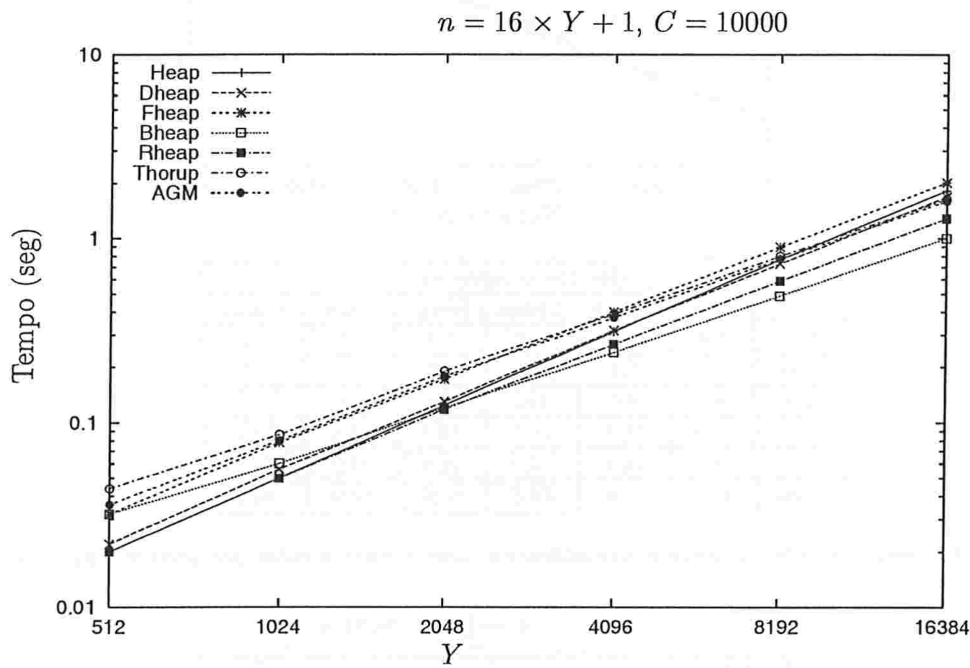
Os experimentos realizados com os grafos gerados por SPGRID se dividem em duas classes: (wide) grafos grade onde o comprimento horizontal é fixo e o vertical cresce com o tamanho da entrada e (long) grafos grade onde o comprimento vertical é fixo e o horizotal cresce com o tamanho da entrada. Observe que o número de arcos com ponta inicial no vértice origem é sempre determinado pelo valor de Y.

Grafos wide gerados com SPGRID

O número fixado de vértices na horizontal é 16, ou seja, $X = 16$. A figura 7.8 exibe o tempo gasto pelos algoritmos e a figura 7.9 o quanto de memória foi consumida. Nestes experimentos, o valor de Y varia de 512 a 16384, e os comprimentos dos arcos são inteiros em [1..10000]. O número de vértices do grafo é $X \times Y + 1$ e o número de arcos é $6(X \times Y)$.

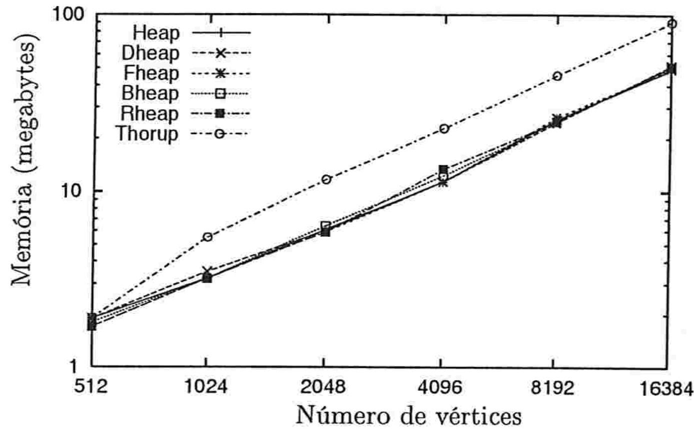
Observa-se que, para valores grandes de Y, o Bheap teve o melhor desempenho, enquanto o Fheap teve o pior. Em relação a memória consumida, novamente Thorup foi o maior consumidor.

Na figura 7.10 é analisada a sensibilidade do algoritmo em relação a C, que é o comprimento do maior arco. As implementações Heap e Dheap foram as mais sensíveis em relação à variação de C.



Tempos (seg)							
Y	heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.02	0.022	0.032	0.032	0.02	0.036	0.044
1024	0.05	0.056	0.078	0.06	0.05	0.08	0.086
2048	0.124	0.13	0.172	0.12	0.118	0.178	0.19
4096	0.312	0.316	0.398	0.24	0.266	0.372	0.388
8192	0.77	0.73	0.9	0.488	0.588	0.776	0.802
16384	1.822	1.69	2.016	1.004	1.284	1.596	1.636

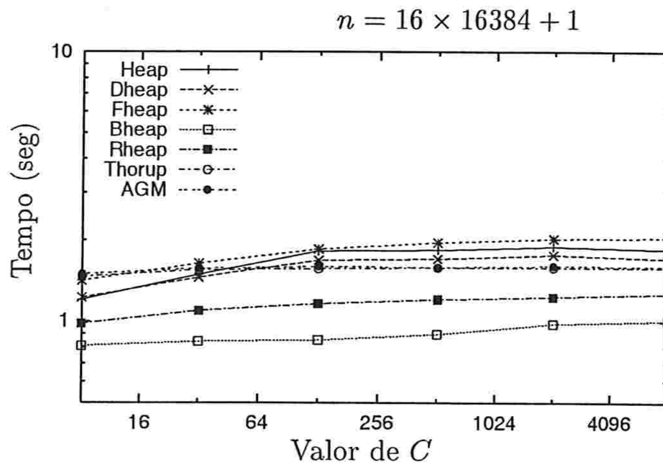
Figura 7.8: Número de vértices em relação ao tempo em grafos grade gerados por SPGRID com $X = 16$.



Memória (megabytes)						
n	heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	1.9	1.9	1.7	1.8	1.7	1.9
1024	3.2	3.5	3.2	3.2	3.2	5.5
2048	6.1	6.0	5.9	6.4	5.9	11.7
4096	11.4	11.4	11.4	12.3	13.4	22.8
8192	25.5	24.5	26.2	24.9	24.9	45.6
16384	49.0	51.9	49.8	51.0	50.9	90.9

Figura 7.9: Memória utilizada quando executado em grafos grade gerados por SPGRID com $X = 16$.

$n = 16 \times 16384 + 1$



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	1.202	1.222	1.412	0.812	0.980	1.460	1.492
32	1.488	1.450	1.638	0.842	1.094	1.550	1.570
128	1.820	1.682	1.850	0.852	1.160	1.564	1.592
512	1.834	1.702	1.956	0.894	1.202	1.576	1.586
2048	1.888	1.758	2.018	0.976	1.228	1.576	1.600
8192	1.844	1.706	2.038	1.002	1.262	1.574	1.594

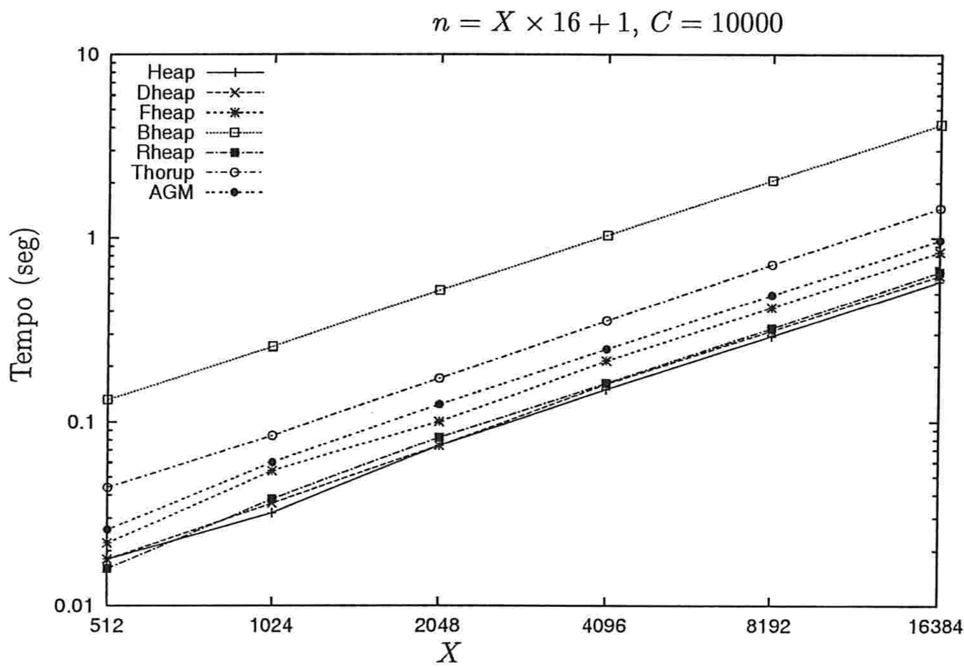
Figura 7.10: Valor de C em relação ao tempo em grafos grade gerados por SPGRID com $X = 16$.

Grafos long gerados com SPGRID

O número fixado de vértices na vertical é 16, ou seja, $Y = 16$. Pode-se perceber que os grafos gerados dessa maneira tendem a ter poucos vértices na fila de prioridade. A figura 7.11 exibe o tempo gasto pelos algoritmos, e a figura 7.12 o quanto de memória foi consumida. Nestes experimentos, X varia de 512 a 16384, e os comprimentos dos arcos são inteiros em [1..10000]. O número de vértices do grafo é $X \times Y + 1$ e o número de arcos é $6(X \times Y)$.

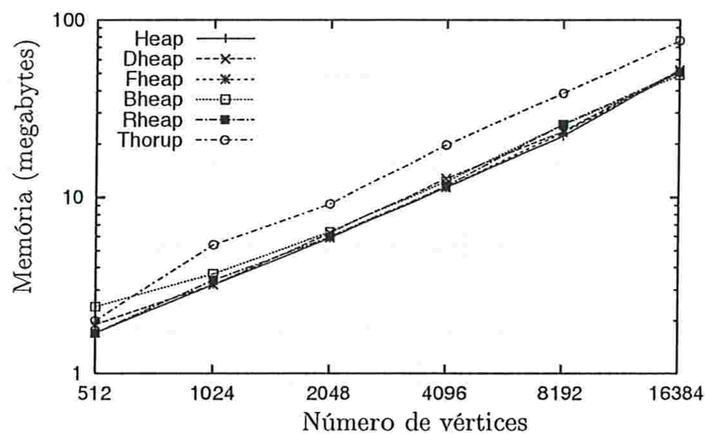
Nestes testes, a implementação Heap foi a vencedora, e a implementação Bheap teve um desempenho bem pior que as demais implementações. Mais uma vez, a implementação Thorup foi a que consumiu mais memória.

Na figura 7.13 é analisada a sensibilidade do algoritmo em relação a C, que é o maior comprimento de um arco. A implementação Bheap foi extremamente sensível. Pôde-se notar que a medida que C cresce, o tempo gasto por Bheap piora exponencialmente.



Tempos (seg)							
X	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.018	0.018	0.022	0.132	0.016	0.026	0.044
1024	0.032	0.036	0.054	0.256	0.038	0.060	0.084
2048	0.074	0.074	0.100	0.520	0.082	0.124	0.172
4096	0.150	0.160	0.214	1.036	0.162	0.248	0.354
8192	0.292	0.312	0.418	2.064	0.322	0.488	0.718
16384	0.578	0.624	0.838	4.152	0.654	0.972	1.454

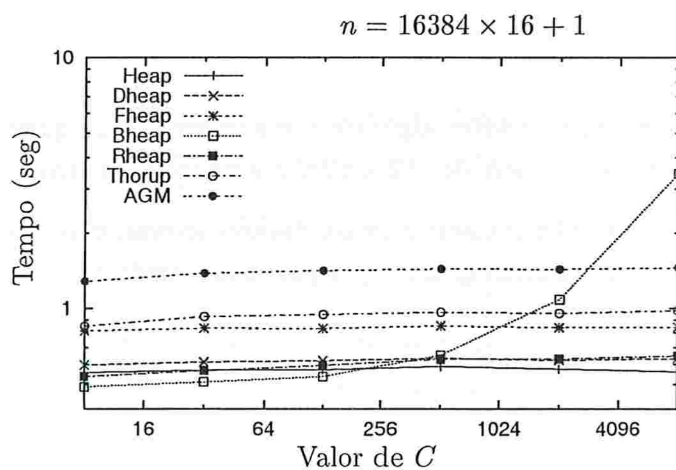
Figura 7.11: Número de vértices em relação ao tempo em grafos grade gerados por SPGRID com $Y = 16$.



Memória (megabytes)						
n	heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	1.7	1.9	1.7	2.4	1.7	2.0
1024	3.2	3.2	3.2	3.7	3.4	5.4
2048	5.9	6.3	5.9	6.4	6.0	9.2
4096	11.4	12.8	11.4	12.3	11.6	19.8
8192	22.2	23.6	23.2	25.7	25.9	38.7
16384	51.9	52.4	51.0	49.0	50.7	76.4

Figura 7.12: Memória utilizada quando executado em grafos grade gerados por SPGRID com $Y = 16$.

$n = 16384 \times 16 + 1$



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	0.558	0.598	0.816	0.490	0.538	0.852	1.284
32	0.570	0.614	0.836	0.512	0.568	0.930	1.382
128	0.572	0.622	0.834	0.538	0.596	0.948	1.416
512	0.590	0.636	0.854	0.654	0.630	0.968	1.438
2048	0.576	0.624	0.840	1.086	0.632	0.958	1.434
8192	0.562	0.634	0.844	3.446	0.650	0.982	1.454

Figura 7.13: Valor de C em relação ao tempo em grafos grade gerados por SPGRID com $Y = 16$.

7.4 SPBAD

Assim como SPRAND, SPBAD gera grafos conexos. Porém, o comprimento dos arcos no ciclo hamiltoniano é sempre 1, e o comprimento dos outros arcos é calculado de forma a produzir um número maior de operações decrease-key (seção 1.4). A forma de se usar o gerador é a seguinte:

```
spbad n d [parâmetros opcionais],
```

onde `n` é o número de vértices (deve ser dois ou mais) e `d` é o grau de saída de cada vértice (deve ser um ou mais). Um exemplo de um grafo desse tipo pode ser visto na figura 7.14.

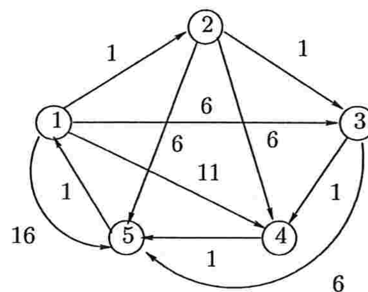
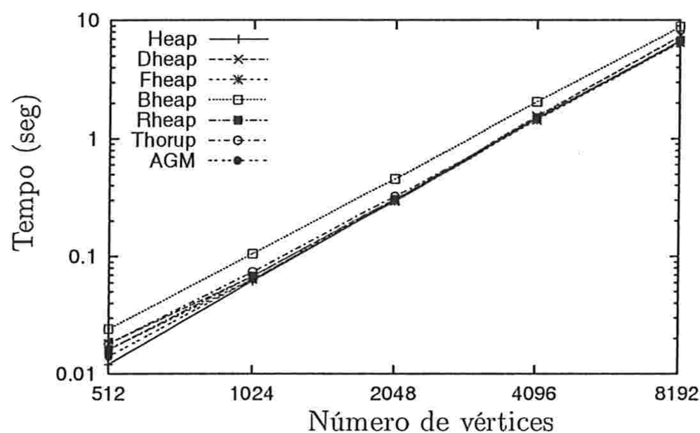


Figura 7.14: Grafo gerado com `spbad 5 4`. Nesse exemplo, alguns arcos foram omitidos, para simplificar a visualização.

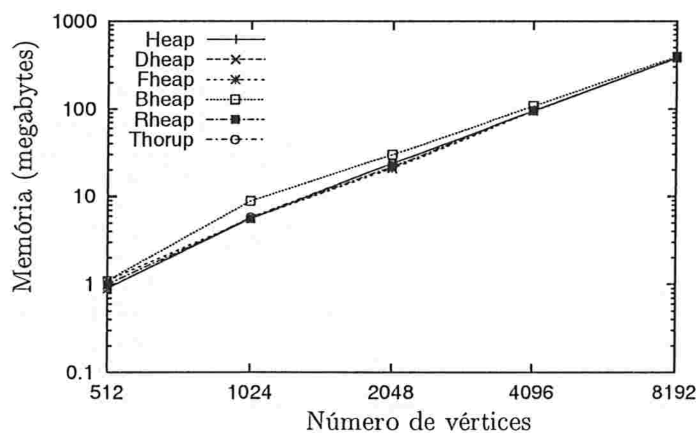
A figura 7.15 exibe o tempo gasto pelos algoritmos e a figura 7.16 o quanto de memória foi consumida. Nestes experimentos, n varia de 512 a 8192 e o número de arcos é $n^2/4$.

Nota-se que a implementação Bheap esteve pior em relação ao tempo e a memória consumida. As demais implementações tiveram desempenhos praticamente idênticos.



Tempos (seg)							
n	heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.012	0.018	0.016	0.024	0.016	0.014	0.018
1024	0.062	0.068	0.064	0.106	0.068	0.064	0.074
2048	0.296	0.304	0.296	0.456	0.308	0.300	0.324
4096	1.476	1.550	1.440	2.056	1.504	1.496	1.466
8192	6.694	7.388	6.544	8.826	6.720	6.768	6.572

Figura 7.15: Número de vértices em relação ao tempo em grafos gerados por SPBAD.



Memória (megabytes)						
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	0.9	0.9	1.1	1.1	1.0	0.9
1024	5.6	5.6	5.6	8.9	5.6	5.8
2048	23.9	23.8	20.9	30.0	21.9	23.8
4096	94.7	94.4	94.0	107.4	95.3	95.1
8192	380.6	384.7	380.6	397.0	383.3	380.5

Figura 7.16: Memória utilizada quando executado em grafos gerados por SPBAD.

CAPÍTULO 8

Conclusões

Nesta dissertação descrevemos, implementamos e testamos, algoritmos para o problema do caminho mínimo. O primeiro algoritmo apresentado é o bem conhecido algoritmo de Dijkstra. Em seguida, é descrito o algoritmo de Dinitz-Thorup, que é um estágio intermediário entre o algoritmo de Dijsktra e o algoritmo de Thorup, que é apresentado ao final. Cada descrição é seguida por uma possível implementação. No caso do algoritmo de Dijkstra, seguem cinco possíveis implementações. Finalmente, é feita uma análise experimental entre as implementações. A implementação de Dinitz-Thorup não entra nas comparações de tempo, pois seu papel é apenas facilitar o entendimento das idéias propostas no algoritmo de Thorup.

A principal diferença entre os algoritmos está na forma de examinar os vértices de um grafo. O algoritmo de Dijkstra examina os vértices em ordem crescente de distância a partir de um dado vértice origem. Conforme observado por Fredman e Tarjan [16], o algoritmo está, implicitamente, ordenando os vértices de acordo com esses valores. Assim, qualquer implementação do algoritmo de Dijkstra para o modelo de comparação-adição realiza, no pior caso, $\Omega(m + n \log n)$ comparações. No algoritmo de Dinitz-Thorup, é utilizada a idéia de bucketing para determinar vértices que podem ser examinados em qualquer ordem. Thorup combinou este bucketing a uma certa decomposição do grafo. Também é importante lembrar que o algoritmo de Dijkstra utiliza o modelo de comparação-adição, enquanto que os algoritmos de Dinitz-Thorup e Thorup foram projetados para o modelo RAM. Os projetos de algoritmos no modelo RAM vêm sendo de grande interesse de pesquisa, pois oferecem melhorias assintóticas significativas e fazem sentido do ponto de vista prático.

O algoritmo linear, no modelo RAM, projetado por Thorup resolve, eficientemente, os seguintes subproblemas:

Construção da decomposição hierárquica. O primeiro passo na construção da decomposição hierárquica é encontrar uma árvore geradora mínima. Isso é feito em tempo

linear, utilizando-se o algoritmo de Fredman e Willard [18]. Segundo, ordenar as arestas do grafo em relação ao *most significant bit* (msb) do comprimento. Utilizando o algoritmo de Andersson, Hagerup, Nilsson e Raman [4], essa tarefa é realizada em tempo linear. E terceiro, utilizar um algoritmo próximo ao de Kruskal junto com a estrutura *union-find* desenvolvida por Gabow e Tarjan [19], que realiza cada operação de união de conjuntos distintos em tempo constante.

Atualização dos potenciais. Utiliza a estrutura atomic heap, desenvolvida por Fredman e Willard [18].

Escolha de um elemento maduro. É feita através do uso de buckets. O tempo total gasto nessa etapa é $O(m+n)$.

Na implementação do algoritmo de Thorup não utilizamos atomic heaps, pois como o próprio Thorup [39] menciona, os atomic heaps são definidos somente para $n > 2^{12^{20}}$, e seu interesse é principalmente teórico. Nós utilizamos na implementação apenas algoritmos mais conhecidos, facilmente encontrados na literatura [1, 9]. Assim, resolvemos os problemas da seguinte maneira:

Construção da decomposição hierárquica. Primeiro, a árvore geradora mínima é encontrada utilizando-se o algoritmo de Kruskal com a estrutura *union-find* [9]. O tempo gasto é $O(m\alpha(m,n))$, onde $\alpha(m,n)$ é a inversa da função de Ackermann. Segundo, ordenar as arestas do grafo em relação ao msb do comprimento é feita utilizando-se um bucket sort. Esse passo gasta $O(\log C + m)$. O tempo total gasto na construção da decomposição hierárquica na nossa implementação é $O(\log C + m + m\alpha(m,n))$.

Atualização dos potenciais. A atualização é feita da maneira mais simples possível, isto é, sempre atualizamos todos os ancestrais de um elemento folha cujo potencial foi decrescido. No pior caso, o tempo gasto na atualização dos potenciais é, claramente, a altura de $\mathcal{L}$, que é limitado por $\log r$, onde r é a razão entre o maior e o menor comprimento de um arco. Portanto, o tempo gasto em cada atualização é $O(\log r)$.

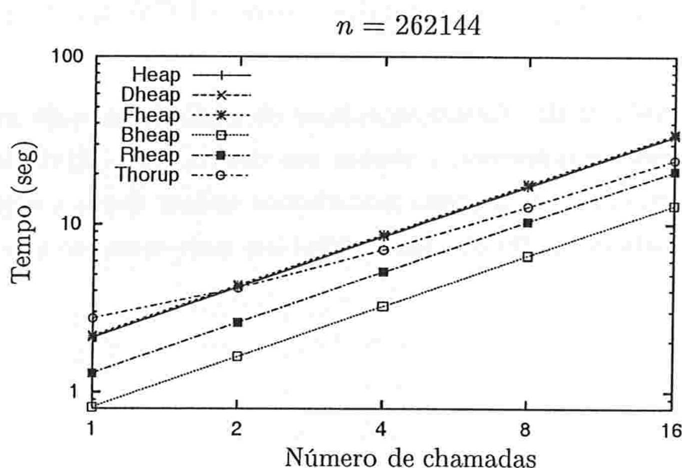
Escolha de um elemento maduro. É feita através do uso de buckets. Portanto, o tempo total gasto nessa etapa é $O(m+n)$.

Logo, a nossa implementação do algoritmo de Thorup gasta tempo $O(n + \log C + m\alpha(m,n) + m\log r)$.

Do ponto de vista teórico, o algoritmo de Thorup apresenta idéias interessantes, como a decomposição hierárquica, e que podem e já estão sendo exploradas por outros pesquisadores, como Pettie e Ramachandran [31, 32].

Ainda do ponto de vista teórico, o pré-processamento do algoritmo de Thorup, isto é, a construção da decomposição hierárquica não afeta a eficiência do algoritmo, como comentado na seção 2.5. Contudo, na prática, o mesmo não ocorre. Basta notar que o algoritmo de Prim-Dijkstra [9] para árvore geradora mínima é praticamente idêntico ao algoritmo de Dijkstra. Logo, apenas o tempo gasto para construir a árvore geradora mínima já será próximo do tempo de resolver o problema do caminho mínimo. Esse fato pode ser comprovado na análise experimental do capítulo 7.

Embora, do ponto de vista prático, o algoritmo de Thorup não tenha se mostrado um sucesso, a sua implementação fez com que entendêssemos melhor o algoritmo, inclusive seus pontos fortes e fracos. Como o pré-processamento é muito "pesado", não faz sentido utilizar esse algoritmo para grafos de dimensões pequenas. Também não vale a pena utilizá-lo em uma única chamada, isto é, encontrar o caminho mínimo uma única vez. Se for desconsiderada a possibilidade de ocorrerem modificações no grafo dado, esse pré-processamento pode ser calculado uma única vez. Então, dependendo do número de chamadas, a um mesmo grafo, para encontrar o caminho mínimo de um vértice a todos os outros, o tempo do pré-processamento acaba não prejudicando o tempo final. A figura 8.1 ilustra o desempenho dos algoritmos conforme o número de chamadas a um mesmo grafo, utilizando um vértice origem distinto a cada chamada. O grafo gerado aleatoriamente é esparso, e foi gerado por SPRAND.



Tempos (seg)						
Número de chamandas	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
1	2.100	2.124	2.162	0.814	1.296	2.744
2	4.198	4.252	4.334	1.626	2.596	4.160
4	8.448	8.532	8.670	3.242	5.190	7.030
8	16.794	17.004	17.344	6.492	10.384	12.740
16	33.618	34.010	34.694	12.976	20.754	24.180
Tempo médio para construir a decomposição hierárquica: 1.3 segundos						

Figura 8.1: Número de chamadas em relação ao tempo em grafos esparsos gerados por SPRAND.

Em relação aos testes, observamos que a implementação Bheap, apesar de ser a melhor em alguns casos, em outros é a pior. A implementação Rheap, apesar de não ganhar em nenhum dos testes, sempre teve um bom desempenho e é bem estável em relação à variação de C. Já com Bheap, isso não aconteceu. As implementações Heap, Bheap e Fheap tiveram os melhores desempenhos quando o número de vértice mantidos por eles é pequeno. Também foi possível observar que a implementação Thorup consome mais memória que as demais e que o seu desempenho é mediano.

Não consideramos que compreendemos inteiramente o algoritmo de Thorup. A descrição apresentada nesta dissertação difere da descrição "mais baixo nível" de Thorup. Tentamos, na medida que nos foi possível, extrair a essência do algoritmo, mas ainda sentimos que mais trabalho precisa ser feito para compreendê-lo melhor e simplificá-lo.

Entre as partes que deixamos por fazer e seriam um próximo passo no sentido de desvendar alguns, por nós, mistérios, destacamos o estudo dos artigos de Andersson *et* al [4], Fredman e Tarjan [18], Gabow e Tarjan [19] e Gabow [20]. Apesar destes representarem avanços teóricos, não acreditamos que do ponto de vista prático colaborem com uma melhora no desempenho da implementação do algoritmo de Thorup.

Em termos teóricos, é interessante estender as técnicas de Thorup para encontrar caminhos mínimos em grafos não necessariamente simétricos. Os primeiros passos nessa direção já foram dados por Hagerup [22] que projetou um algoritmo para o PCM que consome tempo $O(n + m \log \omega)$.

Ainda um outro possível trabalho futuro seria fazer uma implementação híbrida do algoritmo de Thorup, como é feito com o quicksort, e avaliar seu desempenho. Quando o elemento X de $\mathcal{L}$ a ser visitado é "suficientemente" pequeno poderíamos aplicar então o algoritmo de Dijkstra para examinar todos os vértices em $Q \cap X$ cujos potenciais estivessem em um certo "intervalo de segurança".

APÊNDICE A

Implementação

Neste apêndice, se encontra a função *main* do código C, junto com as chamadas para as implementações dos algoritmos. Também são apresentadas as funções de teste da condição de otimalidade.

118 ⟨ Inclusão de arquivos header 118 ⟩ ≡

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include "gb_graph.h"
#include "gb_save.h"
#include "gb_rand.h"
#include "types_dh.h"     /* definições usadas no parser do DIMACS */
#include "parser.c"     /* converte a entrada do formato DIMACS para o SGB */
```

Este código é usado no bloco 9.

119 ⟨ Definições 12 ⟩ +≡

```
#define ERROR_0 0
#define ERROR_1 1
#define ERROR_2 2
#define ERROR_3 3
#define ERROR_4 4
#define TRUE 1
#define FALSE 0
#define EXAMINADO 1
#define NAO_EXAMINADO 0
#define VISITADO 1
```

```
#define NAO_VISITADO 0
#define VERTICE_DE_G (−1)
enum {
  REPORT = 1,   /* se REPORT igual a 1 gera um pequeno relatório */
  DIJK = 1,   /* se DIJK igual a 1 executa dijkstra */
  THORUP = 1,   /* se THORUP igual a 1 executa thorup */
  LINUX = 1,   /* se LINUX igual a 1 então o SO é linux , 0 se for UNIX */
  CONEXO = 1,
  DIMACS = 0
};
```

120 ⟨ Variáveis globais 13 ⟩ +≡

```
char *err_message[] = {   /* 0 */
"N~ao␣conseguiu␣abrir␣arquivo.",   /* 1 */
"N~ao␣conseguiu␣criar␣grafo.",   /* 2 */
"N~ao␣conseguiu␣alocar␣memória.",   /* 3 */
"N~ao␣exite␣arboresc^encia.",   /* 4 */
"Partiç~ao␣inválida."};
```

121 ⟨ Programa principal 121 ⟩ ≡

```
int main(argc, argv)
    int argc;
    char *argv[];
{
  unsigned long n = 1000;   /* número de vértices */
  unsigned long m = 50000;   /* número de arestas */
  unsigned long comp_min = 1;   /* comprimento mínimo de um arco */
  unsigned long comp_max = 1000;   /* comprimento máximo de um arco */
  unsigned long semente = 0;   /* semente do número randômico */
  unsigned long r = 1;   /* número de repetições */
  char *grafo_entrada = Λ;
  char grafo_saida[30];
  Graph *g;
  register Vertex *s;   /* vértice inicial */
  register Vertex *v;
  register Arc *a;
  register Graph *arb;
```

```
clock_t t;
float tempo, tmp;
FILE *tempos, *potencial;
   /* ******** variáveis utilizadas no parser do DIMACS *************** */
arc * arp;
node * ndp, *source;
long nmin;
char name[21];
long mlen;
   /* ************************************************************ */
if (((tempos = fopen("tempos.txt", "a")) ≡ Λ) ∨ ((potencial = fopen("potencial.txt",
      "a")) ≡ Λ)) {
  printf("%s\n", err_message[ERROR_0]);
  exit(0);
}
if (DIMACS) {
  ⟨Entrada DIMACS 123⟩
  if ((g ≡ Λ) ∨ (g→n ≤ 1)) {
    printf("%s\n", err_message[ERROR_1]);
    exit(0);
  }
  ⟨Encontra a árvore de caminhos mínimos 124⟩
  gb_recycle(g);
}
else {
  ⟨Leitura da entrada 122⟩
  while (r−−) {
    if (grafo_entrada) g = restore_graph(grafo_entrada);
    else {
      printf("Criando␣o␣grafo...\n");
      g = random_graph(n, m, 0, 0, 0, Λ, Λ, comp_min, comp_max, semente);
      sprintf(grafo_saida, "SP_%lu_%lu_%lu.gb", n, m, semente);
      sprintf(g→id, "Grafo␣g");
      save_graph(g, grafo_saida);
    }
    if ((g ≡ Λ) ∨ (g→n ≤ 1)) {
      printf("%s\n", err_message[ERROR_1]);
      exit(0);
```

```
        }
        n = g→n;
        ⟨ Encontra a árvore de caminhos mínimos 124 ⟩
        gb_recycle(g);
        semente++;      /* incrementa o valor da semente */
      }
    }
    fclose(tempos);
    fclose(potencial);
    return 0;
  }
```

Este código é usado no bloco 9.

122 ⟨ Leitura da entrada 122 ⟩ ≡

```
  while (--argc) {
    if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) ;
    else if (sscanf(argv[argc], "-m%lu", &m) ≡ 1) ;
    else if (sscanf(argv[argc], "-cmin%lu", &comp_min) ≡ 1) ;
    else if (sscanf(argv[argc], "-cmax%lu", &comp_max) ≡ 1) ;
    else if (sscanf(argv[argc], "-s%lu", &semente) ≡ 1) ;
    else if (sscanf(argv[argc], "-r%lu", &r) ≡ 1) ;
    else if (strncmp(argv[argc], "-f", 2) ≡ 0) grafo_entrada = argv[argc] + 2;
    else if (strncmp(argv[argc], "-h", 2) ≡ 0) {
      printf("Uso: %s [-nN] [-mN] [-cminN] [-cmaxN] [-sN] [-rN] [-farquivo.gb] [-hh]\n",
          argv[0]);
      return 0;
    }
    else if (strncmp(argv[argc], "-hh", 4) ≡ 0) {
      printf("Uso: %s [-nN] [-mN] [-cminN] [-cmaxN] [-sN] [-rN] [-farquivo.gb] [-hh]\n",
          argv[0]);
      printf(" n    - número de vértices\n");
      printf(" m    - número de arestas (o número de arcos é 2m)\n");
      printf(" cmin - menor comprimento de um arco\n");
      printf(" cmax - maior comprimento de um arco\n");
      printf(" s    - semente do número aleatório\n");
      printf(" r    - número de repetiç~oes\n");
      printf(" f    - l^e arquivo.gb\n");
      return 0;
```

```
    }
    else {
      printf("Tente␣\'%s␣-h\'␣para␣mais␣informaç~oes\n", argv[0]);
      return 0;
    }
  }
  if (grafo_entrada) r = 1;
```

Este código é usado no bloco 121.

123 ⟨Entrada DIMACS 123⟩ ≡

```
parse(&n, &m, &ndp, &arp, &source, &nmin, name, &mlen, &g);
printf("%s\nn=␣%ld,␣m=␣%ld,␣nmin=␣%ld,␣source␣=␣%ld,␣maxlen=␣%ld\n", name, n, m,
    nmin, (source − ndp) + nmin, mlen);
comp_max = mlen;
```

Este código é usado no bloco 121.

124 ⟨Encontra a árvore de caminhos mínimos 124⟩ ≡

```
⟨Escolhe o vértice inicial s 126⟩
⟨Encontra o arco de maior comprimento 127⟩
⟨Calcule o valor para infinito 128⟩
if (DIJK) {
  ⟨Execute Dijkstra usando a implementação de Heap 130⟩
  ⟨Execute Dijkstra usando a implementação de D-Heap 131⟩
  ⟨Execute Dijkstra usando a implementação de Fibonacci Heap 132⟩
  ⟨Execute Dijkstra usando a implementação de Bucket heap 133⟩
  ⟨Execute Dijkstra usando a implementação de Radix Heap 134⟩
}
if (THORUP) {
  ⟨Execute o algoritmo de Mikkel Thorup 136⟩
}
```

Este código é usado no bloco 121.

125 ⟨Variáveis globais 13⟩ +≡

```
unsigned long infinito;
unsigned long C;    /* maior comprimento de um arco */
unsigned long num_exam;
```

```
unsigned long atualiza_fp;        /* número de atualizações da função potencial */
double sum;
```

126 ⟨ Escolhe o vértice inicial s 126 ⟩ ≡

```
if (DIMACS) s = g->vertices + (source - ndp) + nmin - 1;
else s = g->vertices;
```

Este código é usado no bloco 124.

127 ⟨ Encontra o arco de maior comprimento 127 ⟩ ≡

```
if (grafo_entrada) {
  for (v = g->vertices; v < g->vertices + g->n; v++) {
    for (a = v->arcs; a; a = a->next) {
      if (C < a->len) C = a->len;
    }
  }
}
else C = comp_max;        /* já estava calculado */
```

Este código é usado no bloco 124.

128 ⟨ Calcule o valor para *infinito* 128 ⟩ ≡

```
infinito = diametro(g, s) * C + 1;
```

Este código é usado no bloco 124.

A função *diametro* faz uma busca em largura para encontrar o maior número de arcos necessário para acessar um vértice, a partir do vértice origem s.

129 ⟨ Funções auxiliares 32 ⟩ +≡

```
unsigned long diametro(g, s)
    Graph *g;
    Vertex *s;
{
  register Vertex *u, *v;
  register Arc *a;
  register long i, j;
  register long diam;
```

```
  diam = 0;
  for (v = g->vertices; v < g->vertices + g->n; v++) {
    v->status = NAO_VISITADO;
  }
  Q = s = g->vertices;
  s->dist = 0;
  Q(0) = s;
  for (i = j = 0, qsize = 1; qsize > 0; qsize--, i++) {
    u = Q(i);
    for (a = u->arcs; a; a = a->next) {
      v = a->tip;
      if (v->status ≡ VISITADO) continue;
      v->dist = u->dist + 1;
      if (diam < v->dist) diam = v->dist;
      v->status = VISITADO;
      Q(++j) = v;
      qsize++;
    }
  }
  return diam;
}
```

130 ⟨ Execute Dijkstra usando a implementação de Heap 130 ⟩ ≡

```
create_pq = create_heap;
insert = insert_heap;
delete_min = delete_min_heap;
decrease_key = decrease_key_heap;
printf("\n++++++++++++++++++++␣DIJKSTRA␣+++++++++++++++++\n");
t = clock();
dijkstra(g, s);
tempo = clock() - t;
printf("\nImplementaç~ao␣de␣Heap\n");
printf("|␣V␣|␣=␣%ld\t|␣A␣|␣=␣%ld\n", g->n, g->m);
⟨ Imprime os dados de saida 135 ⟩
⟨ Testa a corretude da solução 140 ⟩
```

Este código é usado no bloco 124.

131 ⟨ Execute Dijkstra usando a implementação de D-Heap 131 ⟩ ≡

```
create_pq = create_dheap;
insert = insert_dheap;
delete_min = delete_min_dheap;
decrease_key = decrease_key_dheap;
printf("\n+++++++++++++++++++␣DIJKSTRA␣+++++++++++++++++\n");
t = clock();
dijkstra(g, s);
tempo = clock() − t;
printf("\nImplementaç~ao␣de␣D-Heap\n");
printf("|␣V␣|␣=␣%ld\t|␣A␣|␣=␣%ld\tD␣=␣%ld\n", g→n, g→m, D);
⟨ Imprime os dados de saida 135 ⟩
⟨ Testa a corretude da solução 140 ⟩
```

Este código é usado no bloco 124.

132 ⟨ Execute Dijkstra usando a implementação de Fibonacci Heap 132 ⟩ ≡

```
create_pq = create_fheap;
insert = insert_fheap;
delete_min = delete_min_fheap;
decrease_key = decrease_key_fheap;
printf("\n+++++++++++++++++++␣DIJKSTRA␣+++++++++++++++++\n");
t = clock();
dijkstra(g, s);
tempo = clock() − t;
printf("\nImplementaç~ao␣de␣Fibonacci␣Heap\n");
printf("|␣V␣|␣=␣%ld\t|␣A␣|␣=␣%ld\t\n", g→n, g→m);
⟨ Imprime os dados de saida 135 ⟩
⟨ Testa a corretude da solução 140 ⟩
```

Este código é usado no bloco 124.

133 ⟨ Execute Dijkstra usando a implementação de Bucket heap 133 ⟩ ≡

```
create_pq = create_bkheap;
insert = insert_bkheap;
delete_min = delete_min_bkheap;
decrease_key = decrease_key_bkheap;
printf("\n+++++++++++++++++++␣DIJKSTRA␣+++++++++++++++++\n");
t = clock();
```

```
dijkstra(g, s);
free(Q);
tempo = clock() − t;
printf("\nImplementaç~ao␣de␣Bucket␣heap\n");
printf("|␣V␣|␣=␣%ld\t|␣A␣|␣=␣%ld\t%ld␣Buckets\n", g→n, g→m, C + 1);
⟨ Imprime os dados de saida 135 ⟩
⟨ Testa a corretude da solução 140 ⟩
```

Este código é usado no bloco 124.

134 ⟨ Execute Dijkstra usando a implementação de Radix Heap 134 ⟩ ≡

```
create_pq = create_rxheap;
insert = insert_rxheap;
delete_min = delete_min_rxheap;
decrease_key = decrease_key_rxheap;
printf("\n+++++++++++++++++++␣DIJKSTRA␣+++++++++++++++++\n");
t = clock();
dijkstra(g, s);
free(Q);
tempo = clock() − t;
printf("\nImplementaç~ao␣de␣Radix␣Heap\n");
printf("|␣V␣|␣=␣%ld\t|␣A␣|␣=␣%ld\t%d␣Buckets\n", g→n, g→m, log2(g→n ∗ C) + 2);
⟨ Imprime os dados de saida 135 ⟩
⟨ Testa a corretude da solução 140 ⟩
```

Este código é usado no bloco 124.

135 ⟨ Imprime os dados de saida 135 ⟩ ≡

```
printf("\nDurante␣a␣execuç~ao␣do␣programa:\n");
printf("␣Foram␣examinados␣%lu␣vértices\n", num_exam);
for (v = g→vertices, sum = 0; v < g→vertices + g→n; v++, sum += v→dist) ;
printf("␣Soma␣das␣dist^ancias:␣%.0f\n", sum);
printf("␣A␣funç~ao␣pontencial␣precisou␣ser␣atualizada␣%ld␣vezes\n", atualiza_fp);
printf("Tempo:␣%.4f␣segundos\n", tempo/CLOCKS_PER_SEC);
if (g→m ≠ 0) fprintf(potencial, "%.4f\t", (float) atualiza_fp/g→m);
fprintf(tempos, "%.4f\t", tempo/CLOCKS_PER_SEC);
```

Este código é usado nos blocos 130, 131, 132, 133 e 134.

```
136 ⟨ Execute o algoritmo de Mikkel Thorup 136 ⟩ ≡
  printf("\n++++++++++++++++++␣THORUP␣++++++++++++++++++\n");
  t = clock();
  arb = krusk(g);
  tmp = clock() − t;
  printf("Tempo␣para␣construir␣a␣arborescencia:␣%.4f␣segundos\n",
      tmp/CLOCKS_PER_SEC);
  t = clock();
  thorup(g, arb, s);
  gb_recycle(arb);
  free(BK);
  tempo = clock() − t;
  printf("\nImplementaç~ao␣de␣Mikkel␣Thorup\n");
  printf("|␣V␣|␣=␣%ld\t|␣A␣|␣=␣%ld\n", g→n, g→m);
  printf("\nDurante␣a␣execuç~ao␣do␣programa:\n");
  printf("␣Foram␣examinados␣%lu␣vértices\n", num_exam);
  for (v = g→vertices, sum = 0; v < g→vertices + g→n; v++, sum += v→dist) ;
  printf("␣Soma␣das␣dist^ancias:␣%.0f\n", sum);
  printf("␣A␣funç~ao␣potencial␣precisou␣ser␣atualizada␣%ld␣vezes\n", atualiza_fp);
  printf("Tempo:␣%.4f␣segundos\n", tempo/CLOCKS_PER_SEC);
  if (g→m ≠ 0) fprintf(potencial, "%.4f\n", (float) atualiza_fp/g→m);
  fprintf(tempos, "%.4f\t", tempo/CLOCKS_PER_SEC);    /* thorup */
  fprintf(tempos, "%.4f\n", tmp/CLOCKS_PER_SEC);    /* AGM */
  if (REPORT) {
    printf("␣A␣arboresc^encia␣tem␣%d␣nós(fora␣os␣vértices␣de␣g)\n", nelementos);
    printf("␣Precisou␣desempilhar␣%d␣vezes\n", desempilha);
    printf("␣Precisou␣atualizar␣para␣cima␣%d␣vezes\n", atualiza_acima);
    printf("␣Os␣elementos␣mudaram␣de␣maduros␣para␣n~ao␣maduros␣%d␣vezes\n",
        nao_maduro);
  }
  ⟨ Testa a corretude da solução 140 ⟩
```

Este código é usado no bloco 124.

A.1 Testa condição de otimalidade

A função *funcao_potencial_OK* verifica se todos os vértices estão com um potencial válido. Se todos os potenciais forem válidos, a função devolve TRUE. Caso contrário, devolve FALSE e imprime um arco que não respeita a função potencial.

138 ⟨Teste da condição de otimalidade 138⟩ ≡

```
int funcao_potencial_OK(g)
    Graph *g;
{
  register Vertex *v;
  register Vertex *u;
  register Arc *a;
  for (u = g->vertices; u < g->vertices + g->n; u++) {
    for (a = u->arcs; a; a = a->next) {
      v = a->tip;
      if (v->dist - u->dist > a->len) {      /* a função não respeita c */
        printf("\n********  A função potencial não é válida  *********\n");
        printf("****** Arco que não respeita a função potencial ******\n");
        printf("*******  %s -> %s  --  %ld - %ld > %ld  ************\n", u->name,
            v->name, v->dist, u->dist, a->len);
        return FALSE;
      }
    }
  }
  return TRUE;
}
```

Veja também bloco 139.

Este código é usado no bloco 9.

A subrotina *calcula_caminho* calcula o comprimento do caminho, e *verifica_caminhos_OK* verifica se o comprimento do caminho respeita a condição de otimalidade 2.2.

139 ⟨Teste da condição de otimalidade 138⟩ +≡

```
unsigned long comprimento_caminho(v)
    Vertex *v;
{
```

```
  register Vertex *u;
  register Arc *a;
  register unsigned long minarc = infinito;
  if (v->pred ≡ v) return 0;
  for (a = (v->pred)->arcs; a; a = a->next) {
      /* encontra o arco de menor comprimento com ponta final em v */
    u = a->tip;
    if (u ≡ v) {
      if (a->len < minarc) minarc = a->len;
    }
  }
  return (comprimento_caminho(v->pred) + minarc);
}
int verifica_caminhos_OK(g)
    Graph *g;
{
  register Vertex *v;
  register unsigned long comp;
  for (v = g->vertices; v < g->vertices + g->n; v++)
    if ((v->dist ≠ infinito) ∧ ((comp = comprimento_caminho(v)) ≠ v->dist)) {
      printf("\n**␣Comprimento␣do␣caminho␣de␣s␣a␣%s␣está␣errado␣**\n", v->name);
      printf("***␣Caminho:␣%ld\tPotencial:␣%ld␣␣***\n", comp, v->dist);
      return FALSE;
    }
  return TRUE;
}
```

Se não houve nenhum problema, imprime as seguintes mensagens.

140 ⟨ Testa a corretude da solução 140 ⟩ ≡

```
if (REPORT ∧ funcao_potencial_OK(g) ∧ verifica_caminhos_OK(g)) {
  printf(">>>>>␣␣Funç~ao␣potencial␣Ok␣␣<<<<<\n");
  printf(">>>>>␣␣Caminhos␣de␣custo␣mínimo␣Ok␣␣<<<<<\n");
}
```

Este código é usado nos blocos 130, 131, 132, 133, 134 e 136.

Referências Bibliográficas

[1] R.K. Ahuja, T.L. Magnant e J. Orlin, *Network flows: Theory, algorithms, and applications*, Practice Hall, 1993. Citado na(s) página(s) 1, 43, 140

[2] R.K. Ahuja, T.L. Magnanti, J. Orlin e M.R. Reddy, Applications of network optimization, *Handbook in Operations Research and Management Sciences* (M.O. Ball, T.L. Magnanti e C.L. Monma, eds.), vol. 7, North-Holland, Amsterdam,, 1995, pp. 1–83. Citado na(s) página(s) 1

[3] R.K. Ahuja, K. Mehlhorn, J.B. Orlin e R.E. Tarjan, Faster algorithms for the shortest path problem, *J. ACM* **37** (1990), 213–223. Citado na(s) página(s) 1, 3, 43

[4] A. Andersson, T. Hagerup, S. Nilsson e R. Raman, Sorting in linear time?, *J. Comput. System Sci.* **57** (1998), 74–93. Citado na(s) página(s) 2, 99, 105, 108, 140, 142

[5] A. Andersson, P.B. Miltersen e M. Thorup, Fusion trees can be implemented with AC^0 instructions only, *Theoretical Computer Science* **215** (1999), 337–344. Citado na(s) página(s) 2

[6] B.V. Cherkassky, A.V. Goldberg e T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1994. Citado na(s) página(s) 1, 123

[7] B.V. Cherkassky, A.V. Goldberg e C. Silverstein, Buckets, heaps, lists and monotone priority queues, *SIAM J. Comput.* **28** (1999), 1326–1346. Citado na(s) página(s) 1, 43, 123

[8] T.H. Cormen, C.E. Leiserson e R.L. Rivest, *Introduction to algorithms*, McGraw-Hill, 1999. Citado na(s) página(s) 43

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest e C. Stein, *Introduction to algorithms*, 2nd. ed., The MIT Press and McGraw-Hill, 2001. Citado na(s) página(s) 1, 26, 106, 110, 140, 141

[10] R. Dial, Algorithm 360: Shortest path forest with topological ordering, *Communications of the ACM* **12** (1969), 632–633. Citado na(s) página(s) 3, 43

[11] E.W. Dijkstra, A note on two problems in connection with graphs, *Numerische Mathematik* **1** (1959), 269–271. Citado na(s) página(s) ii, 1, 3, 29

[12] E.A. Dinic, Finding shortest paths in a network, *In Y. Popkov and B. Shmulyian Eds.*, Transportation Modeling Systems, Institute for System Studies, Moscow, 1978, pp. 36–44. Citado na(s) página(s) 69

[13] P. Feofiloff, *Notas de aula de MAC 5781 Otimização Combinatória*, "http://www.ime.usp.br/~pf/", 1997. Citado na(s) página(s) 7, 13, 23, 29

[14] ———, *Algoritmos de programação linear*, EDUSP, 2000. Citado na(s) página(s) 13, 24

[15] B. Fortz e M. Thorup, Internet traffic engineering by optimizing ospf weights, *INFOCOM (2)*, 2000, pp. 519–528. Citado na(s) página(s) 1

[16] M.L. Fredman e R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987), 596–615. Citado na(s) página(s) 1, 3, 37, 43, 139

[17] M.L. Fredman e D.E. Willard, Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.* **47** (1993), 424–436. Citado na(s) página(s) 2, 3

[18] ———, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comput. System Sci.* **48** (1994), 533–551. Citado na(s) página(s) 1, 3, 27, 99, 100, 105, 110, 140, 142

[19] H.N. Gabow, A linear-time algorithm for a special case of disjoint set union, *J. Comput. System Sci.* **30** (1985), 209–221. Citado na(s) página(s) 99, 105, 106, 140, 142

[20] ———, A scaling algorithm for weighted matching on general graphs, *26th FOCS* (1985), 90–100. Citado na(s) página(s) 142

[21] A.V. Goldberg e C. Silverstein, *Implementation of Dijkstra's algorithm based on multi-level buckets*, Tech. report, NEC Research Institute, Princeton, NJ, 1995. Citado na(s) página(s) 1, 123

[22] T. Hagerup, Improved shortest path on the word ram, *Proceeding of the 27th International Colloquium on Automata, Languages and Programming* (2000), 61–72. Citado na(s) página(s) 3, 142

[23] S. Halabi, *OSPF Design Guide*, April 1996, http://www.cisco.com/warp/public/104/1.html. Citado na(s) página(s) 1

[24] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24** (1977), 1–13. Citado na(s) página(s) 3, 37

[25] D.S. Johnson, A theoretician's guide to the experimental analysis of algorithms, *To appear in Proceedings of the 5th and 6th DIMACS Implementation Challenges*, 2002. Citado na(s) página(s) 123

[26] D.E. Knuth, *Literate programming*, Center for the study of Language and Information (CS-LI), 1992. Citado na(s) página(s) 15

[27] ______, *The Stanford GraphBase: A plataform for combinatorial computing*, ACM Press, 1993. Citado na(s) página(s) 17, 43

[28] D.E. Knuth e S. Levy, *The CWEB System of Structured Documentation*, Addison-Wesley, 1994. Citado na(s) página(s) 15

[29] Sun Microsystems, *Sun Numerical Computation Guide*, 1999, `"http://www.arlut.utexas.edu/SunCompiler/common"`. Citado na(s) página(s) 107

[30] R. Miyagi, *Introduction to GCC Inline Asm*, `"http://linuxassembly.org/rmiyagi-inline-asm.txt"`. Citado na(s) página(s) 107

[31] S. Pettie e V. Ramachandran, Computing shortest paths with comparisons and additions, *SODA: ACM-SIAM Symposium on Discrete Algorithms*, SIAM, January 6–8 2002, pp. 267–276. Citado na(s) página(s) 2, 3, 11, 27, 140

[32] S. Pettie, V. Ramachandran e S. Sridhar, Experimental evaluation of a new shortest path algorithm, *4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, 2002, pp. ??–?? Citado na(s) página(s) 1, 110, 123, 140

[33] Phillip, *Using Assembly Language in Linux*, `"http://linuxassembly.org/linasm.html"`. Citado na(s) página(s) 107

[34] N. Rahman e R. Raman, An experimental study of word-level parallelism in some sorting algorithms, *Proceedings WAE*, 1998, pp. 193–203. Citado na(s) página(s) 12

[35] R. Raman, Recent results on the single-source shortes paths problem, *SIGACT News* **28** (1997), 81–87. Citado na(s) página(s) 3

[36] A. Schrijver, *Combinatorial optimization - polyhedra and efficiency (forthcoming book) Part I*. Citado na(s) página(s) 1, 43

[37] P.M. Spira e A. Pan, On finding an updating shortest paths and spanning trees, *Proc. Symp. Switching and Automata Theory*, 1973, pp. 82–84. Citado na(s) página(s) 26

[38] R.E. Tarjan, *Data structures and network algorithms*, BMS-NSF Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, PA, 1983. Citado na(s) página(s) 26

[39] M. Thorup, Undirect single source shortest paths with positive integer weights in linear time, *J. ACM* **46** (1999), 362–394. Citado na(s) página(s) ii, 1, 2, 3, 69, 85, 99, 110, 140

[40] ______, On RAM priority queues, *SIAM J. Comput.* **30** (2000), 86–109. Citado na(s) página(s) 2, 3, 37

[41] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inf. Proc. Lett.*, vol. 6, 1977, pp. 80–82. Citado na(s) página(s) 3

[42] R.A. Wagner, A shortest path algorithm for edge-sparse graphs, *J. ACM* **23** (1976), 50–57. Citado na(s) página(s) 3

[43] U. Zwick, Exact and approximate distances in graphs - a survey, *Proceedings of 9th ESA* (2001), 33–48. Citado na(s) página(s) 2

Índice Remissivo

(S, Q), 8
$A(S, Q)$, 8
$A[S]$, 9
$[j..k]$, 7
$\mathbb{Z}$, 7
$\mathbb{Z}_{\geq}$, 7
$\mathbb{Z}_{>}$, 7
δ-decomposição W-hierárquica, 86
δ-partição, 70
$\text{dist}_c(s, t)$, 23
$\text{dist}(s, t)$, 23
m, 8
n, 8
CWEB, 15
SGB, 17

acessível, 9
AGM, 27
algoritmo de
    Dinitz-Thorup, 71
    Dinitz-Thorup, 69
    Thorup, 89
altura da arborescência, 86
ancestrais de um elemento, 86
arborescência, 10
    determinada por ψ, 25
Arc, 18
arco, 8
    reverso, 8
aresta, 8
árvore
    geradora, 10

bloco, 15
bucket, 43
bucketing, 43

caminho, 9
    determinado por ψ, 25
    não-orientado, 9
ciclo, 9
    não-orientado, 9
complexidade, 12
complexidade de
    algoritmo, 12
    problema, 12
comprimento
    do caminho, 23
    função, 23
condição de
    inacessibilidade, 24
    otimalidade, 25
conexo, 9
conjunto de arcos induzidos, 9
corte, 9

desempilhar, 7

diâmetro, 99
distância, 23
  tentativa, 26, 30
dualidade, 24

elemento
  filho, 85
  folha, 85
  interno, 85
  maduro, 87
  pai, 85
elemento maduro, 70
empilhar, 7
estrutura de dados, 13
examinar um/uma
  arco, 26
  vértice, 26

família laminar, 85
  W-completa, 86
fila de prioridade, 13
filho, 10
folha de uma arborescência, 10
função
  comprimento, 23
  comprimento simétrica, 23
  potencial, 24
  potencial respeita, 24
  potencial viável, 24
  predecessor, 25

grafo, 8
  acíclico, 9
  completo, 26
  simétrico, 8
  simétrico acíclico, 9
Graph, 19
grau de um vértice, 52

heap, 44
  D-heap, 47
  bucket, 59
  fibonacci, 51
  radix, 62

inacessibilidade, 24
início de um passeio, 9
intervalo, 7
invariantes do algoritmo de
  Dijkstra, 30
  Dinitz-Thorup, 73
  Thorup, 96

Knuth, 15

lema
  da dualidade, 24
Levy, 15
lista, 7
lista de
  adjacência, 10

maior comprimento, 23
matriz de
  adjacência, 10
  incidência, 10
modelo
  de comparação, 11
  de comparação-adição, 11
  de comparação-adição-subtração, 11
  de computação, 11
  Random Access Machine, 12
monótona, 13
most significant bit, 106

nível de um elemento, 86

otimalidade, 25

pai, 10
parte de um conjunto, 7

passeio, 9
  não orientado, 9
PCM, 23, 29
PCMS, 85
PCMV, 70
pilha, 7
potencial
  função, 24
predecessor
  função, 25
problema
  árvore geradora mínima, 27
  do caminho mínimo, 23
programação literária, 15
propriedade
  número de descendentes de um vértice
    no fibonacci heap, 52
  ordem no D-heap, 47
  ordem no bucket heap, 59
  ordem no fibonacci heap, 51
  ordem no heap, 44
  ordem no radix heap, 63

raiz da arborescência, 10
RAM, 12
representação arbórea, 85

single-source shortest path, 23
Stanford Graph Base, 17

tamanho de
  um grafo, 8
término de um passeio, 9
território, 9
tipo
  abstrato, 13
  de dado, 13

Vertex, 17
vértice, 8
  adormecido, 30
  examinado, 30
  visitado, 30
visitar um elemento, 90

Lista de Refinamentos

⟨ Algoritmo de Dijkstra 14 ⟩ Citado no bloco 10. Usado no bloco 9.
⟨ Algoritmo de Dinitz-Thorup 76 ⟩ Usado no bloco 9.
⟨ Algoritmo de Thorup 88 ⟩ Usado no bloco 9.
⟨ Arquivos header e definições 2 ⟩ Usado no bloco 1.
⟨ Atualize o potencial dos elementos 112 ⟩ Usado no bloco 111.
⟨ Busca seqüencial por x 6 ⟩ Usado no bloco 3.
⟨ Calcule o valor para *infinito* 128 ⟩ Usado no bloco 124.
⟨ Caso u e v estejam no mesmo conjunto, comece nova iteração 97 ⟩ Usado no bloco 91.
⟨ Coloque cada vértice em um conjunto distinto e inicializa a arborescência 95 ⟩ Usado no bloco 91.
⟨ Coloque os elementos de p nos buckets $B(0), B(1), \ldots, B(dtg)$ 79 ⟩ Usado no bloco 76.
⟨ Coloque u como filho de v 101 ⟩ Usado no bloco 98.
⟨ Consolidate 46 ⟩ Citado nos blocos 45 e 46. Usado no bloco 45.
⟨ Construa a decomposição hierárquica do grafo 91 ⟩ Usado no bloco 88.
⟨ Crie um bucket para x 110 ⟩ Usado no bloco 106.
⟨ Crie um novo elemento 99 ⟩ Usado no bloco 98.
⟨ Definições 12, 23, 40, 52, 59, 72, 89, 94, 104, 119 ⟩ Usado no bloco 9.
⟨ Desempilhe 115 ⟩ Usado no bloco 106.
⟨ Devolva a arborescência 102 ⟩ Usado no bloco 91.
⟨ Empilhe o filho maduro de x 114 ⟩ Usado no bloco 106.
⟨ Encontra a árvore de caminhos mínimos 124 ⟩ Usado no bloco 121.
⟨ Encontra a posição k do primeiro bucket não-vazio 65 ⟩ Usado no bloco 64.
⟨ Encontra filho f de p tal que $d(f)$ seja mínimo 37 ⟩ Usado no bloco 36.
⟨ Encontra o arco de maior comprimento 127 ⟩ Usado no bloco 124.
⟨ Encontra o valor do menor potencial em $Q(k)$ 66 ⟩ Usado no bloco 64.
⟨ Encontre a raiz *root* da arborescência 109 ⟩ Usado no bloco 106.
⟨ Entrada DIMACS 123 ⟩ Usado no bloco 121.
⟨ Escolha u em Q tal que $d(u)$ seja mínimo 19 ⟩ Usado no bloco 14.

⟨ Escolha u em x tal que $d(u)$ seja mínimo 81 ⟩ Usado no bloco 76.
⟨ Escolhe o vértice inicial s 126 ⟩ Usado no bloco 124.
⟨ Examine a aresta uv 83 ⟩ Usado no bloco 82.
⟨ Examine a folha x 111 ⟩ Usado no bloco 106.
⟨ Examine o arco uv 21 ⟩ Usado no bloco 20.
⟨ Examine o vértice u 20 ⟩ Usado no bloco 14.
⟨ Examine os vértices utilizando a decomposição hierárquica 106 ⟩ Usado no bloco 88.
⟨ Examine vértice u 82 ⟩ Usado no bloco 76.
⟨ Execute Dijkstra usando a implementação de Bucket heap 133 ⟩ Usado no bloco 124.
⟨ Execute Dijkstra usando a implementação de D-Heap 131 ⟩ Usado no bloco 124.
⟨ Execute Dijkstra usando a implementação de Fibonacci Heap 132 ⟩ Usado no bloco 124.
⟨ Execute Dijkstra usando a implementação de Heap 130 ⟩ Usado no bloco 124.
⟨ Execute Dijkstra usando a implementação de Radix Heap 134 ⟩ Usado no bloco 124.
⟨ Execute o algoritmo de Mikkel Thorup 136 ⟩ Usado no bloco 124.
⟨ Faça a união dos conjuntos que contém u e v e construa a arborescência 98 ⟩ Usado no bloco 91.
⟨ Fila Q não está vazia 18 ⟩ Usado no bloco 14.
⟨ Filas de prioridade 11, 25, 26, 27, 28, 29, 33, 34, 35, 36, 38, 42, 43, 44, 45, 48, 53, 54, 55, 56, 57, 61, 62, 63, 64, 69 ⟩ Usado no bloco 9.
⟨ Funções auxiliares 32, 41, 50, 60, 73, 74, 75, 86, 96, 105, 129 ⟩ Usado no bloco 9.
⟨ Imprime os dados de saida 135 ⟩ Usado nos blocos 130, 131, 132, 133 e 134.
⟨ Inclusão de arquivos header 118 ⟩ Usado no bloco 9.
⟨ Inicializações de dinitz 78 ⟩ Usado no bloco 76.
⟨ Inicializações para *thorup* 108 ⟩ Usado no bloco 106.
⟨ Inicializa d e ψ 16 ⟩ Usado no bloco 14.
⟨ Inicializa a fila Q com s 17 ⟩ Usado no bloco 14.
⟨ Junte u e v em um único elemento 100 ⟩ Usado no bloco 98.
⟨ Leitura da entrada 122 ⟩ Usado no bloco 121.
⟨ Leitura dos parâmetros de entrada 5 ⟩ Usado no bloco 3.
⟨ Link arborescência y com a arborescência x 47 ⟩ Citado no bloco 46. Usado no bloco 46.
⟨ Main 3 ⟩ Usado no bloco 1.
⟨ Mude x de bucket 113 ⟩ Usado no bloco 106.
⟨ Não encontrou x 8 ⟩ Usado no bloco 6.
⟨ Ordene os arcos colocando-os nos buckets $sort[0], \ldots, sort[msbC]$ 93 ⟩ Usado no bloco 91.
⟨ Programa principal 121 ⟩ Usado no bloco 9.
⟨ Redistribui os intervalos 67 ⟩ Usado no bloco 64.
⟨ Remove e distribui os vértices de $Q(k)$ nos buckets anteriores 68 ⟩ Usado no bloco 67.
⟨ Remove v da lista de filhos de p 49 ⟩ Citado nos blocos 48 e 50. Usado nos blocos 48 e 50.
⟨ Seja x um elemento em $B(k)$ 80 ⟩ Usado no bloco 76.

⟨ Testa a corretude da solução 140 ⟩ Usado nos blocos 130, 131, 132, 133, 134 e 136.
⟨ Teste da condição de otimalidade 138, 139 ⟩ Usado no bloco 9.
⟨ Variáveis de Dijkstra 15 ⟩ Usado no bloco 14.
⟨ Variáveis de dinitz 77 ⟩ Usado no bloco 76.
⟨ Variáveis de *krusk* 92 ⟩ Usado no bloco 91.
⟨ Variáveis de *thorup* 107 ⟩ Usado no bloco 106.
⟨ Variáveis globais 13, 24, 31, 71, 90, 103, 120, 125 ⟩ Usado no bloco 9.
⟨ Variáveis locais 4 ⟩ Usado no bloco 3.
⟨ Verifica se $v[i] = x$ 7 ⟩ Usado no bloco 6.
⟨ Verifique se x deve mudar de bucket 84 ⟩ Usado no bloco 76.
⟨ `busca.c` 1 ⟩