A

Variações e aplicações do algoritmo de Dijkstra

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Patrícia Takaki Neves e aprovada pela Banca Examinadora.

Campinas, 10 de setembro de 2007.

Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Patrícia Takaki Neves
e aprovada pela Banca Examinadora.
Campinas, 08 de Janeiro de 2008
Paulo de Azevedo
COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Orlando Lee

Orlando Lee (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Neves, Patrícia Takaki
N414v Variações e aplicações do algoritmo de Dijkstra / Patrícia Takaki Neves -- Campinas, [S.P. :s.n.], 2007.

Orientador : Orlando Lee
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Estruturas de dados (Computação). I. Lee, Orlando. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Variants and applications of Dijkstra´s algorithms.

Palavras-chave em inglês (Keywords): Data structures (Computer science).

Área de concentração: Ciência da Computação.

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Orlando Lee (IC-UNICAMP)
Prof. Dr. Flavio Keidi Miyazawa (IC-UNICAMP)
Prof. Dr. José Coelho de Pina Junior (IME-USP)
Prof. Dr. Cid Carvalho de Souza (IC-UNICAMP)

Data da defesa: 10/08/2007

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 10 de agosto de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. José Coelho de Pina Junior
IME - USP.

Prof. Dr. Flávio Keidi Miyazawa
IC - UNICAMP.

Prof. Dr. Orlando Lee
IC - UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Variações e aplicações do algoritmo de Dijkstra

Patrícia Takaki Neves

Setembro de 2007

Banca Examinadora:

- Orlando Lee (Orientador)
- Flávio Keidi Miyazawa (IC - UNICAMP)
- José Coelho de Pina Júnior (IME - USP)
- Cid Carvalho de Souza (IC - UNICAMP - Suplente)

Frases

"Se você acredita que pode, poderá, porque só a vontade gera energia, e a disposição de fazer é o começo da obra já concluída."
Autor Desconhecido

"Uma jornada de milhões de milhas começa com um simples passo e se este passo é o passo certo, ele se torna o último passo."
Lao Tsu

Agradecimentos

Quero agradecer a Deus por ter me confiado mais este desafio e por ter iluminado meu caminho para conseguir vencê-lo. Agradeço a meus pais, Olga e Lauro Takaki, pelo apoio incondicional e irrestrito. Ao meu marido Fabrício e minha filha Júlia, pelo incentivo, amor, inspiração e compreensão. Aos irmãos Law, Igor e Deb pelo apoio constante e verdadeiro sentimento de irmandade, à cunhada Anne, sobrinhos e demais familiares pelo carinho de sempre.

Agradeço imensamente ao meu orientador, Prof. Orlando Lee, que com muita disposição trouxe-me de volta para este tortuoso caminho que foi a escrita de uma segunda dissertação, permitindo-me estudar este assunto. Sou grata por ter contribuído de maneira tão especial para o meu crescimento e aprendizagem. Sua simplicidade e compreensão, características de um grande coração, nunca serão esquecidas.

Agradeço ao Prof. Flávio Keidi Miyazawa pelos incentivos dados para a conclusão deste trabalho e pela disposição e compreensão em ajudar. Ao Prof. Renato Werneck que, mesmo dos EUA, foi presença importante para a escrita final mostrando-me caminhos e materiais de grande relevância para trazer esta dissertação para a fronteira do conhecimento.

Um agradecimento especial à Lin e ao Zanoni pela sincera amizade, pelas contribuições e por me receberem várias vezes em Campinas com tanta hospitalidade e paciência.

Aos colegas de trabalho, alunos e professores, que de várias formas me incentivaram a concluir este mestrado.

Que todos saibam a dimensão do meu *"domo arigato"*.

Resumo

O problema de encontrar caminhos mínimos em um grafo com pesos nas arestas é considerado fundamental em otimização combinatória. Diversos problemas do mundo real podem ser modelados dessa forma: percurso mais curto/rápido entre duas cidades, transmissão de dados em uma rede de computadores, reconhecimento de voz, segmentação de imagens entre outros. O algoritmo proposto por Dijkstra em 1959 resolve o problema de caminhos mínimos em grafos sem arestas de peso negativo, o que não chega a ser restritivo na maior parte das aplicações. Desde então, o algoritmo tem sido refinado com o uso de estruturas de dados cada vez mais sofisticadas, reduzindo seu tempo de execução de pior caso (ao menos, do ponto de vista teórico).

Recentemente, problemas de caminhos mínimos têm aparecido no contexto de Sistemas de Informação Geográfica (SIG). Neste modelo, o usuário faz consultas ao sistema para encontrar o trajeto mais curto (ou rápido) entre dois pontos especificados (problema ponto-a-ponto ou problema P2P). Além disso, pode haver várias consultas. Instâncias neste tipo de modelo são relativamente grandes: o mapa rodoviário dos Estados Unidos tem mais de 20 milhões de vértices (cada vértice representa intersecções de vias). Mesmo as implementações mais sofisticadas do algoritmo de Dijkstra não apresentam um desempenho prático capaz de atender às demandas que esse tipo de modelo requer.

A pesquisa recente tem tentado reduzir este *gap* entre a teoria e a prática. Várias técnicas de aceleração de algoritmos têm sido propostas e implementadas: busca bidirecional, algoritmo A*, alcance (*reach*), *landmarks* e muitos outros. Algumas dessas técnicas têm restrições de domínio e outras podem ser usadas em qualquer contexto.

Neste trabalho, estudamos algumas variações da versão original do algoritmo de Dijkstra, caracterizadas pelas diferentes estruturas de dados. Implementamos quatro dessas variações e realizamos testes experimentais utilizando os mapas do mundo real. Nosso objetivo foi analisar o desempenho prático dessas. Dedicamos também uma atenção especial ao problema P2P, apresentando algumas das principais técnicas de aceleração.

Abstract

The problem of finding shortest paths in a weighted graph is a fundamental one in combinatorial optimization. Several real world problems can be modeled in this way: shortest or fastest tour between two cities, data transmission on a computer network, voice recognition, image segmentation among others. The algorithm proposed by Dijkstra in 1959 solves this problem when the graph has no edge with negative weight, which is not a serious restriction in most applications. Since then, the algorithm has been improved with the use of sophisticated data structures, reducing the worst case running time (at least, from a theoretical viewpoint).

Recently shortest path problems has appeared in the context of Geographic Information System (GIS). In this model, the user asks the system to find out the shortest path between two given points (point-to-point problem or P2P problem). Moreover, there can be several queries. Instances in this model are relatively large: the road network map of the United States has more than 20 million vertices (each vertex represents an intersection of two roads). Even the fastest implementations of Dijkstra's algorithm do not have a performance in practice which is satisfactory to meet the requirements of this model.

Recent research has tried to reduce this gap between theory and practice. Several speed-up techniques for these algorithms have been proposed and implemented: bidirectional search, algorithm A*, reach, landmarks and many others. Some of them are domain-restricted and others are applicable in any context.

In this work, we studied some variants of Dijkstra's algorithm characterized by its different data structures. We have implemented four of those variants and performed experimental tests using real-world maps. Our goal was to analyze their practical performance. We also paid special attention to the P2P problem, and presented some of the main speed-up techniques.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O *problema dos caminhos mínimos*[1] é um problema clássico em otimização combinatória: dados um grafo G com pesos não negativos nas arestas e um vértice fonte s, encontrar caminhos (de pesos) mínimos de s a todos os demais vértices. Na literatura, este problema também é conhecido como *problema de caminhos mínimos com fonte única*[2]. Outra versão do problema, conhecida como *problema do caminho mínimo ponto-a-ponto*[3] ou simplesmente *problema* P2P, requer apenas que seja encontrado um caminho mínimo entre dois vértices especificados.

Diversos problemas do mundo real podem ser modelados como um problema de caminhos mínimos com fonte única ou problema P2P.

- O roteamento de veículos em um sistema de transporte é atualmente uma das principais aplicações. Neste contexto, vértices representam os cruzamentos de vias, as arestas representam as vias e os pesos representam alguma métrica. Por exemplo, distâncias entre cruzamentos, tempo de percurso da via, condições das estradas, entre outras medidas. Este tipo de aplicação tem se tornado cada vez mais freqüente com o desenvolvimento dos computadores pessoais ou portáteis, como por exemplo, sistemas de navegação de carros, sistemas embarcados GPS, MapQuest da Yahoo!, MapPoint da Microsoft etc. Estes tipos de aplicação caem dentro do modelo dos chamados Sistemas de Informações Geográficas (SIG). A pesquisa científica mais recente, com vista nos resultados práticos, tem se concentrado nesses tipos de aplicação.

- Outro exemplo clássico de aplicação é a transmissão de pacotes de dados em uma rede de computadores. Uma rede pode ser modelada por um grafo em que os

[1] *Shortest paths problem.*
[2] *Single source shortest paths problem.*
[3] *Point-to-point shortest path problem.*

vértices representem equipamentos envolvidos (computadores, roteadores, *switch* etc), as arestas representam a infra-estrutura de comunicação de dados (cabos, fios, *wireless*). Os pesos podem representar taxa máxima de transmissão, disponibilidade de transmissão etc. Alguns protocolos de transmissão de pacotes que utilizam algoritmos de caminhos mínimos são o RIP (*Routing Information Protocol*) que usa um protocolo de roteamento baseado no algoritmo de Bellman-Ford e o OSPF (*Open Shortest Paths First*) que é um protocolo baseado no algoritmo de Dijkstra.

- Há inúmeros outros contextos de aplicação. Por exemplo, projeto de circuitos integrados em que os componentes precisam ser ligados por um menor caminho possível. Outras aplicações menos comuns são encontradas na literatura em diversas áreas: projeto de circuitos integrados, reconhecimento de voz [RS96], problema de segmentação de objetos em processamento de imagens, desenho de um grafo, especulação financeira, colocação ótima de torres de transmissão[FG03] dentre outras. Para as aplicações mencionadas sem referência, consulte o livro *Network Flows* [AMO93] de Ahuja, Magnanti e Orlin. Este reúne várias aplicações importantes e interessantes para o problema de caminhos mínimos, o que demonstra sua relevância na prática.

O artigo de E.W. Dijkstra [Dij59] *A note on two problems in connexion with graphs*, é considerado um artigo clássico da área de otimização combinatória. Nele, são considerados dois problemas de conexidade em grafos. Um deles é o conhecido problema da *árvore geradora mínima* e o outro, o problema de caminhos mínimos. Dijkstra comenta que o segundo problema foi motivado pela necessidade de elaborar uma demonstração pública para o "computador automático" *ARMAC* que estava prestes a ser terminado na época. Ele usou um mapa simplificado do sistema ferroviário da Holanda e na demonstração o computador calculava a distância de duas cidades escolhidas pela audiência. O "programa" usado pelo computador era o que hoje é conhecido como algoritmo de Dijkstra (para caminhos mínimos).

Atualmente, nas aplicações mais importantes, tem-se que resolver problemas de caminhos mínimos para instâncias de dimensões muito grandes. Por exemplo, os mapas rodoviários dos Estados Unidos e Europa possuem cerca de 20 milhões de vértices. Neste contexto de larga escala, o tempo gasto para determinar um caminho mínimo usando uma implementação ingênua do algoritmo de Dijkstra torna-se proibitiva.

Implementações mais eficientes do algoritmo de Dijkstra, usando estruturas de dados mais sofisticadas, apareceram na depois [Dia69, Joh77, FT87, AMOT90]. Entretanto, mesmo essas implementações ainda são insuficientes para atender às demandas dos modelos atuais de problemas P2P. Neste, distingue-se duas fases: a fase de pré-processamento e a fase de busca/consulta. Cada busca ou consulta deve ser extremamente rápida e pode

haver várias dessas. Na fase de pré-processamento, permite-se gastar um tempo maior de computação. Idealmente, esta fase não deve ser executada muitas vezes, e o intervalo de tempo entre duas execuções deve ser relativamente longo. Um exemplo mencionado por Schulz, Wagner e Weihe [SWW00] é o seguinte. As estações de trem em cidades da Alemanha possuem seus próprios quadros de horário indicando partidas e chegadas de trens; o que deseja-se então, a qualquer momento, é percurso mais rápido de uma estação a uma outra (não necessariamente vizinha) a partir de um dado horário. Os horários mudam apenas 2 vezes por ano. Assim, basta executar um pré-processamento (possivelmente caro) apenas no início do verão e do inverno.

Muitos trabalhos recentes [Gut04, GW05, SS05, HSWW05, GKW06a, GKW06b, SS06, GKW07] têm se concentrado neste tipo de modelo. Várias técnicas de *aceleração* têm sido propostas e incorporadas ao algoritmo original de Dijkstra e os resultados práticos têm sido muito bons. Muito do estudo feito neste trabalho foi baseado nesses e em outros artigos indicados nas referências.

Este trabalho reúne conceitos e técnicas relevantes ao problema de caminhos mínimos, tanto com fonte única quanto para a versão P2P.

Além disso, foram feitas quatro implementações para a versão com fonte única, com o intuito de avaliar os desempenhos práticos do algoritmo de Dijkstra com diferentes estruturas de dados. As instâncias utilizadas foram grafos que representam mapas do mundo real. Este estudo prático, em combinação com estudos teóricos envolvendo problemas ponto-a-ponto no mundo real, permitiu alcançar uma maior compreensão das idéias e técnicas que apareceram e que ainda devem surgir nesta área de pesquisa.

Não foram estudadas outras variantes de problemas de caminhos mínimos nem algoritmos para essas. Em particular, não foram consideradas versões onde o grafo de entrada é *conservativo*, isto é, pode ter pesos negativos, mas não existe nenhum circuito de peso negativo. Exemplos clássicos que não foram considerados são: (1) o algoritmo de Bellman-Ford, proposto independentemente por Bellman [Bel58], Ford e Fulkerson [FR62] e implementado por Moore [Moo59]. Este algoritmo resolve o problema de caminhos mínimos com fonte única em grafos conservativos; (2) o algoritmo Floyd-Warshall [Flo62] determina caminhos mínimos entre todos os pares de vértices, em um grafo conservativo.

Organização

O Capítulo 2 apresenta as definições e notação adotadas neste trabalho. Este capítulo descreve o algoritmo de Dijkstra, suas subrotinas e fornece um exemplo de simulação. Ao final é demonstrada sua corretude e analisada sua complexidade.

O Capítulo 3 dedica-se a apresentar e comparar as principiais estrutura de dados utilizada na representação da fila de prioridades do algoritmo de Dijkstra. São elas: heap

binário, heap de Fibonacci, radix heap e estrutura de dados de bucket (ou Algoritmo de Dial). São descritas suas operações e analisadas suas complexidades. São ainda apresentados alguns resultados experimentais obtidos pela implementação desenvolvida do algoritmo de Dijkstra com quatro variações de estruturas de dados. São apresentados e discutidos alguns dados de teste obtidos com entradas de dados que representam mapas do mundo real disponibilizados pelo DIMACS [DGJ06].

O Capítulo 4 aborda o Algoritmo de Dijkstra no problema de menor caminho de ponto-a-ponto aplicado ao contexto de roteamento de veículos em mapas. Este capítulo discute algumas das principais técnicas de aceleração do algoritmo como a busca bidirecional, a busca heurística e o pré-processamento. São apresentados alguns resultados experimentais de referências recentes envolvendo estas técnicas. Os principais trabalhos publicados nesta área foram consultados.

O Capítulo 5 resume os resultados apresentados nos capítulos anteriores e expõe algumas conclusões e trabalhos futuros. Este trabalho é finalizado com uma visão geral das atuais e futuras linhas de investigação que podem ser desenvolvidas sobre este assunto.

Capítulo 2

O algoritmo de Dijkstra

Um algoritmo para resolver o problema de caminhos mínimos com fonte única foi proposto por E. W. Dijkstra [Dij59]. Dijkstra comenta a motivação de dois problemas de conexidade em grafos: árvore geradora mínima e caminhos mínimos. O primeiro surgiu da tarefa de elaborar uma demonstração pública para o segundo computador automático *ARMAC* que estava sendo fabricado. Assim, ele propôs e apresentou uma aplicação para encontrar um caminho mínimo entre duas cidades no mapa ferroviário holandês. O segundo problema surgiu quando lhe foi solicitado minimizar a quantidade de cabos no painel de um computador que estava sendo projetado.

Diversos textos apresentam o algoritmo de Dijkstra de formas diferentes. Alguns livros pesquisados que trazem explicações teóricas e práticas com implementações em linguagens de programação são: [Sed02] com códigos em linguagem C; [Dro02] com a linguagem C++, [Pre00] com a linguagem JAVA e [Ziv04] com Pascal e C. Outros livros que analisam o algoritmo teoricamente incluem [BM76, AMO93, SM94, AHU74, AHU87, BB96, CLR99] e [Man89], dentre outros tantos. Ademais, em uma grande quantidade de publicações científicas, este algoritmo reaparece em diferentes contextos e com pseudo-códigos ligeiramente diferentes.

No resto do capítulo é apresentada uma descrição formal do algoritmo de Dijkstra e as análises de corretude e complexidade desse. Não houve aqui a preocupação em fornecer detalhes muito específicos de implementação, mas sim introduzir os passos e as idéias mais importantes do algoritmo.

A Seção 2.1 apresenta as definições e notação usada ao longo deste trabalho. A Seção 2.2 apresenta a descrição do algoritmo. As Seções 2.3 e 2.4 apresentam as análises de corretude e complexidade, respectivamente.

2.1 Definições e notação

Um grafo orientado é um par $G = (V, E)$ onde V é um conjunto finito de *vértices* e E é um conjunto de pares ordenados de vértices, chamados de *arestas*. Uma aresta de G é denotada por (u, v) e diz-se que u é o *início* da aresta e v o *final* da aresta. Neste trabalho, convenciona-se que n é o número de vértices de G e m é o número de arestas de G.

Assume-se também que o grafo é *ponderado*, ou seja, a ele está associada implicitamente uma função peso $w : E \to \mathbb{R}_{\geq 0}$ que mapeia arestas em valores reais não negativos[1].

Um *caminho* em G é uma seqüência de vértices $P = \langle v_0, v_1, \ldots, v_k \rangle$ tal que (v_i, v_{i+1}) é uma aresta de G para $i = 0, 1, \ldots, k-1$. Diz-se ainda que P é um caminho de v_0 a v_k.

O *peso* de um caminho $P =< v_0, v_1, \ldots, v_k >$ é a soma dos pesos das suas arestas e é denotado por $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Para dois caminhos P e Q de s a v, dizemos que P é *menor* que Q se $w(P) < w(Q)$. A *distância* de u a v em G, denotada por $\text{dist}(u, v)$, é o peso mínimo de um caminho de u a v em G. Se tal caminho não existir, denota-se $\text{dist}(u, v) = \infty$.

2.2 Algoritmo de Dijkstra

Neste e nos outros capítulos, convenciona-se que o grafo de entrada é representado por listas de adjacências [CLR99]; a lista de vizinhos de v em G é denotada por Adjacência$[v]$.

O algoritmo de Dijkstra resolve o seguinte problema. Dado um grafo ponderado G e um vértice fonte s como entradas, ele devolve:

- para cada vértice $v \in V$, o peso de um caminho mínimo de s a v e
- uma *árvore de caminhos mínimos* com raiz em s. Um caminho de s a v nesta árvore é um caminho mínimo de s a v em G.

Resumidamente, o algoritmo faz o seguinte. Este começa com uma árvore contendo apenas s e, a cada iteração, um novo vértice é acrescentado à árvore segundo uma estratégia gulosa. Ao final, obtém-se uma árvore de caminhos mínimos.

Durante sua execução, o algoritmo mantém dois atributos $d[v]$ e $p[v]$ para cada vértice v.

- $d[v]$ é uma estimativa (ou limite) *superior* da distância do vértice fonte s até v. Além disso, se $d[v]$ for finito, significa que o algoritmo encontrou até aquele momento um caminho de s a v com peso $d[v]$.

[1] Algumas estruturas de dados que serão analisadas no próximo capítulo exigem que tal função mapeie valores inteiros não negativos ($\mathbb{Z}_{\geq 0}$). Isto não é muito restritivo, pois basta multiplicar todos os pesos por um inteiro suficientemente grande. O novo problema com esta função peso modificada é equivalente ao problema original.

- $p[v]$ armazena o *predecessor* de v em um caminho de s a v com peso $d[v]$. É utilizado para representar a árvore de caminhos mínimos que o algoritmo devolve.

O primeiro passo do algoritmo está descrito na subrotina `Inicializa` apresentada no Algoritmo 2.1.

Algoritmo 2.1 Inicializa (G, s)

```
1: for all v ∈ V(G) do
2:     d[v] ← ∞
3:     p[v] ← NULL
4: d[s] ← 0
```

Se $v \neq s$ e $p[v] = NULL$ isto significa que v não pertence à árvore determinada até o momento.

Após a inicialização, o algoritmo executa uma série de "relaxações" das arestas. Uma *relaxação* consiste em atualizar a estimativa de distância de um vértice ao longo de uma aresta. No Algoritmo 2.2 é descrita a subrotina `Relaxa` que realiza essa operação.

Algoritmo 2.2 Relaxa (u, v)

```
1: if d[v] > d[u] + w(u, v) then
2:     d[v] ← d[u] + w(u, v)
3:     p[v] ← u
```

Intuitivamente, `Relaxa` verifica se é mais "barato" chegar a v passando por u. Se $d[v] > d[u] + w(u, v)$, isto significa que o caminho de s a u de peso $d[u]$ (determinado pelo algoritmo) acrescido da aresta (u, v) é menor que o caminho de peso $d[v]$ encontrado até o momento. Se isso ocorre, então o caminho de s a v é substituído por esse novo caminho que passa por u. Isto é feito atribuindo a $d[v]$ o valor $d[u] + w(u, v)$ e atribuindo ao predecessor $p[v]$ o vértice u, como descrito no Algoritmo 2.2. Veja a Figura 2.1. Diz-se

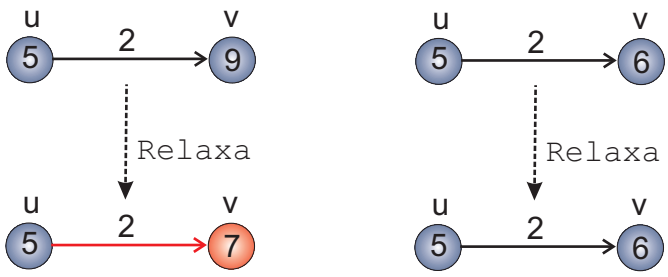

Figura 2.1: Execução de duas chamadas a `Relaxa`. Os valores de $d[v]$ estão representados no interior dos vértices. À esquerda $d[v]$ é atualizado e à direita este permanece o mesmo.

que uma aresta (u, v) está *relaxada* se foi feita uma chamada `Relaxa(u,v)`.

Em cada iteração, o algoritmo seleciona um vértice u e, para cada $v \in$ Adjacência$[u]$, aplica-se `Relaxa(u,v)`. Um vértice para o qual esse passo foi executado é chamado vértice *escaneado*. Veja a Figura 2.2.

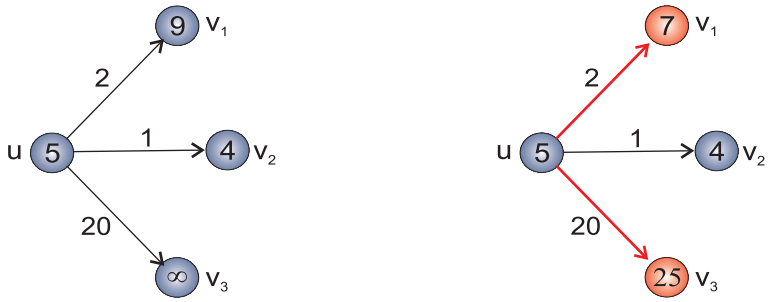

Figura 2.2: Execução da operação `Relaxa` para cada vizinho v de u. Ao final, o vértice u é tido como *escaneado*.

Por fim, o pseudocódigo do algoritmo principal de Dijkstra é apresentado no Algoritmo 2.3.

Algoritmo 2.3 Dijkstra (G, s)

```
1: Inicializa(G, s)
2: S ← ∅
3: Q ← V
4: while Q ≠ ∅ do
5:     u ← ExtraiMin(Q)
6:     for all v ∈ Adjacência[u] do
7:         Relaxa(u, v)
8:     S ← S∪{u}
```

O conjunto $\mathcal{Q}$ é implementado como uma fila de prioridades. Uma *fila de prioridades* é um tipo abstrato de dados formado por um conjunto de elementos, cada um desses com um valor chave associado. Além disso, tal tipo deve suportar as seguintes operações:

- `Insere`: insere um novo elemento na fila,
- `DiminuiChave`: diminui o valor da chave de um elemento específico da fila e
- `ExtraiMin`: remove um elemento com menor chave da fila.

No pseudo-código do Algoritmo 2.3, há n chamadas implícitas de `Insere` na linha 3 e uma de `DiminuiChave` na linha 7 (se a estimativa de $d[v]$ diminuir).

O conjunto $\mathcal{S}$ contém os vértices escaneados[2]. Intuitivamente, os vértices em $\mathcal{S}$ cor-

[2]Este conjunto não é realmente necessário, mas simplifica a análise do algoritmo.

respondem àqueles que estão mais "próximos" de s. O elemento u de menor chave em $\mathcal{Q}$ é o vértice mais próximo de $\mathcal{S}$ ainda não escaneado. Este é extraído de $\mathcal{Q}$, escaneado e inserido em $\mathcal{S}$. Então as distâncias estimadas dos vértices em Adjacência$[u]$ e a árvore dos caminhos mínimos são atualizadas. Note que um vértice é escaneado no máximo uma vez durante a execução do algoritmo.

A Figura 2.3 mostra uma execução do algoritmo de Dijkstra.

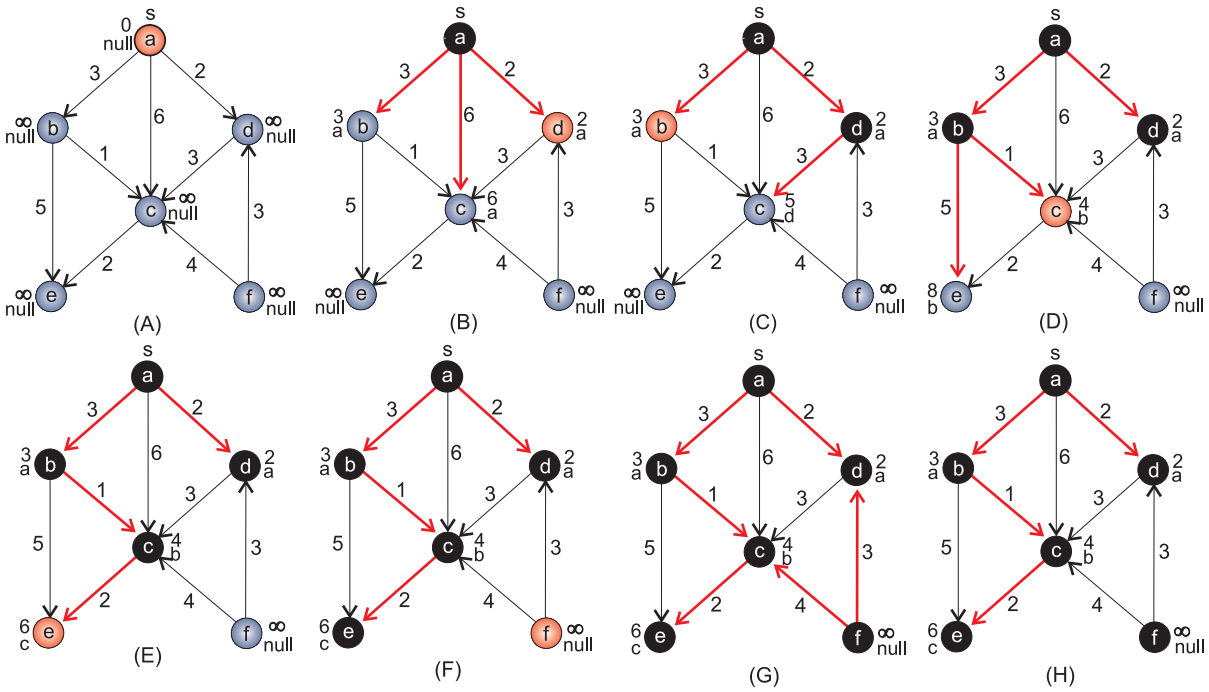

Figura 2.3: Simulação do algoritmo de Dijkstra. Os rótulos $d[v]$ e $p[v]$ estão ao lado dos vértices. (A) O algoritmo extrai a de $\mathcal{Q}$. (B) As arestas (a, b), (a, c) e (a, d) são relaxadas, as estimativas e os predecessores de b, c e d são atualizados e a é inserido em $\mathcal{S}$. (C) O algoritmo extrai d e relaxa (d, c). O valor de $d[c]$ diminui de 6 para 5, $p[c]$ passa de a para d, e d é inserido em $\mathcal{S}$. As próximas iterações (D)-(H) seguem esse padrão.

Para o grafo de exemplo dado, a Figura 2.1 apresenta um quadro passo-a-passo da execução com os valores das variáveis principais.

Note que, uma vez determinada a árvore de caminhos mínimos, para reconstruir um caminho mínimo de s a v basta utilizar o vetor de predecessores $p[.]$. Mais precisamente, este caminho é a seqüência $\langle s, \ldots, p[p[p[v]]], p[p[v]], p[v], v\rangle$.

Iteração:	S:	Q:	u	d[a]	d[b]	d[c]	d[d]	d[e]	d[f]
Inicializa	Ø	a,b,c,d,e,f	Ø	0	∞	∞	∞	∞	∞
1	a	b,c,d,e,f	a		3	6	2	∞	∞
2	a,d	b,c,e,f	d		3	5		∞	∞
3	a,d,b	c,e,f	b			4		8	∞
4	a,d,b,c	e,f	c					6	∞
5	a,d,b,c,e	f	e						∞
6	a,d,b,c,e,f	Ø	f						

Tabela 2.1: Tabela da simulação do algoritmo de Dijkstra para o grafo da Figura 2.3. A primeira linha corresponde à inicialização, onde $\mathcal{S}$ é vazio, $\mathcal{Q}$ recebe todos os vértices de G e as distâncias dos vértices são infinitas, com exceção de a cuja distância é 0. As demais linhas correspondem às iterações do algoritmo. Quando um vértice é escaneado, a sua coluna é cancelada por um traço vertical que cancela suas linhas inferiores.

2.3 Corretude do algoritmo de Dijkstra

Para provar que o algoritmo de Dijkstra está correto, será demonstrado que ao final do algoritmo:

- $d[v] = \text{dist}(s, v)$ para todo $v \in V$ e
- $p[.]$ define uma árvore de caminhos mínimos. Mais precisamente, o conjunto $\{(p[x], x) : x \in V - \{s\}\}$ forma uma árvore de caminhos mínimos.

É fácil perceber que os seguintes invariantes valem em cada iteração (linha 4) do Algoritmo 2.3:

- Se $d[x] < \infty$ então $p[x] \in \mathcal{S}$.
- Para cada vértice x em $\mathcal{S}$, a seqüência $\langle s, \ldots, p[p[x]], p[x], x\rangle$ é um caminho de s a x com peso $d[x]$.

Outros invariantes fáceis de verificar são:

- $\text{dist}(s, x) \leq d[x]$ para cada vértice x em $\mathcal{S}$.
- o conjunto $\{(p[x], x) : x \in V - \{s\}\}$ forma uma árvore.
- Se $x \in \mathcal{S}$, $y \in \mathcal{Q}$ e (x, y) é uma aresta de G, então $d[y] \leq d[x] + w(x, y)$. Isto vale pois quando x foi inserido em $\mathcal{S}$, foi executada a operação `Relaxa`.

Uma propriedade importante no contexto dos algoritmos de caminhos mínimos é a seguinte. Se $P = \langle s, v_1, v_2, \ldots, v_n\rangle$ é um caminho mínimo de s a v_n, então $P_i = \langle s, v_1, \ldots, v_i\rangle$

é um caminho mínimo de s a v_i para todo $1 \leq i \leq n$. Intuitivamente, isto explica porque o algoritmo de Dijkstra explora os vértices mais próximos de $\mathcal{S}$ para expandi-lo.

Agora será demonstrado que o seguinte invariante é sempre válido no início de cada iteração da linha 4 do algoritmo de Dijkstra:

Invariante: $d[x] = \text{dist}(s, x)$ para cada $x \in S$.

Note que, ao final do algoritmo, $\mathcal{S}$ consiste dos vértices atingíveis a partir de s. Portanto, se o invariante vale, o valor de $d[v]$ é exatamente a distância de s a v, para cada $v \in V$. Em vista dos invariantes acima, isto é suficiente para demonstrar a corretude do algoritmo.

Prova do invariante

Claramente o invariante vale na primeira iteração, pois $\mathcal{S} = \emptyset$. Também vale no início da segunda iteração pois $\mathcal{S} = \{s\}$ e $d[s] = 0$.

Note que uma vez que um vértice u é colocado em $\mathcal{S}$, o valor $d[u]$ não é alterado no restante do algoritmo. O algoritmo de Dijkstra escolhe um vértice u com menor $d[u]$ em $\mathcal{Q}$ e atualiza $\mathcal{S} \leftarrow \mathcal{S} \cup \{u\}$. Basta verificar então que neste momento, $d[u] = \text{dist}(s, u)$.

Suponha, por contradição, que $d[u] > \text{dist}(s, u)$. Seja P um caminho mínimo de s a u (ou seja, um caminho com peso $\text{dist}(s, u)$). Seja y o primeiro vértice de P que não pertence a $\mathcal{S}$. Seja x o vértice em P que precede y. Este esquema está representado na Figura 2.4.

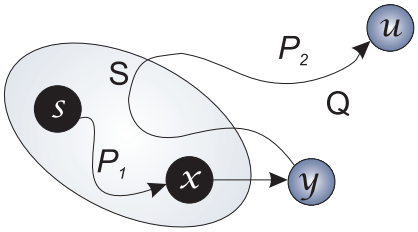

Figura 2.4: Corretude do algoritmo de Dijkstra.

Por hipótese, $\text{dist}(s, u) < d[u]$. Como x está em $\mathcal{S}$, segue do invariante que $d[x] = \text{dist}(s, x)$. Então

$$\begin{aligned} d[y] &\leq d[x] + w(x, y) \\ &\leq \text{dist}(s, x) + w(x, y) + w(P_2) \\ &= w(P_1) + w(x, y) + w(P_2) \\ &= \text{dist}(s, u) \\ &< d[u], \end{aligned}$$

onde a primeira desigualdade segue do fato de que x está em $\mathcal{S}$ e, portanto, a aresta (x, y) já foi relaxada.

Mas então $d[y] < d[u]$ o que contraria a escolha de u. Portanto, $d[u] = \text{dist}(s, u)$ e o invariante vale.

2.4 Análise de complexidade

A complexidade do algoritmo de Dijkstra como um todo depende da forma como é implementado o conjunto $\mathcal{Q}$ e as operações que manipulam a fila de prioridades. No que segue, é demonstrada a complexidade do algoritmo supondo que $\mathcal{Q}$ foi implementado como um vetor ou uma lista encadeada sem nenhuma ordem entre os elementos.

A subrotina `Inicializa` consome tempo $O(n)$. O loop principal do algoritmo nas linhas $2-6$ escaneia cada vértice no máximo uma vez. Cada operação `ExtraiMin` requer que sejam comparados os nós em $\mathcal{Q}$ em busca da menor estimativa e consome tempo $O(|\mathcal{Q}|)$. Então o tempo total de `ExtraiMin` em todo o algoritmo é $n + (n-1) + (n-2) + \ldots + 1 = O(n^2)$. Quando um vértice é escaneado, todas as arestas que saem dele são consideradas pela subrotina `Relaxa`. Cada aresta do grafo é relaxada no máximo uma vez. Logo, o número total de chamadas a `Relaxa` é $O(m)$. O tempo total do algoritmo de Dijkstra é então $O(n^2 + m) = O(n^2)$.

Este limite de tempo de $O(n^2)$ é o melhor possível para grafos densos (isto é, $m = \Omega(n^2)$), mas pode ser melhorado para grafos esparsos. Como é óbvio, o tempo requerido pelo `ExtraiMin` é maior que o de `Relaxa`. Assim, a busca por algoritmos mais eficientes tem se concentrado em tentar reduzir o tempo de extração dos nós, sem aumentar substancialmente o tempo de atualização das estimativas [AMO93]. O próximo capítulo apresenta diferentes implementações do algoritmo de Dijkstra, mais eficientes que a apresentada aqui.

Capítulo 3

Variações do algoritmo de Dijkstra

Há vários fatores que devem ser levados em consideração ao implementar-se um algoritmo: consumo de tempo, consumo de espaço e o grau de dificuldade da codificação.

Considere o pseudocódigo do algoritmo de Dijkstra visto no último capítulo (Algoritmo 2.3). Desde a sua publicação, têm sido propostas diferentes implementações para o algoritmo de Dijkstra. Estas diferenças consistem principalmente nas estruturas de dados utilizadas que representam a fila de prioridades $\mathcal{Q}$. Como exposto no Capítulo 2, o desempenho do algoritmo de Dijkstra depende principalmente da eficiência da operação `ExtraiMin`. Também foi visto que a complexidade do algoritmo de Dijkstra implementando a fila como uma lista não ordenada é $O(n^2)$, onde n é o número de vértices do grafo de entrada. Esta variação é geralmente referenciada como "implementação ingênua" (*naive implementation*). Neste capítulo serão vistas implementações mais sofisticadas de filas de prioridades e conseqüentemente do algoritmo de Dijkstra.

Talvez a primeira e mais significativa contribuição para reduzir a complexidade da implementação ingênua tenha sido feita por Donald Johnson [Joh77] que introduziu a estrutura de dados denominada *heap k-ário*. Esta variação melhora o comportamento assintótico do algoritmo de Dijkstra para grafos esparsos. Implementando-se este algoritmo com um heap binário ($k = 2$), sua complexidade é melhorada de $O(n^2)$ para $O((m + n) \log n)$, onde m denota o número de arestas.

Outra estrutura de dados, denominada *heap de Fibonacci*, foi introduzida por Fredman e Tarjan [FT87] e oferece o melhor resultado teórico conhecido, alcançando a complexidade de tempo de $O(m + n \log n)$. Entretanto, em aplicações práticas, ao se comparar o tempo de execução de implementações com diferentes estruturas, observa-se que o heap de Fibonacci apresenta resultados práticos inferiores aos de heap k-ário. Um estudo desse tipo foi feito por Cherkassky, Goldberg e Radzik [CGR94] que apresenta resultados experimentais e teóricos de várias implementações de algoritmos para caminhos mínimos (algoritmos de Dijkstra, Bellman-Ford-Moore, Pallotino e outros). É considerado um dos

trabalhos comparativos mais importantes da área e foi uma das principais referências deste capítulo.

Outra variação do algoritmo de Dijkstra, proposto por Dial [Dia69], usa uma estrutura de dados denominada *bucket*. O tempo total desse algoritmo é $O(m + nC)$, onde C é o máximo dos pesos da arestas. Este algoritmo é apropriado para aplicações em que C é pequeno, inteiro e não negativo. Duas variações da estrutura de bucket são os buckets multi-níveis proposta por Denardo e Fox [DF79] e os *hot-queue* (*heap-on-top priority queue*) propostos por Cherkassky, Goldberg e Silverstein [CGS97] que combinam buckets multi-níveis com heaps binários. Os resultados experimentais apresentados nestes artigos são bastante promissores. Estas variações não foram estudadas neste trabalho.

Quando os pesos das arestas assumem apenas alguns dos valores dentro do conjunto $\{0, 1, 2, \ldots, C\}$, Ahuja, Mehlhorn, Orlin e Tarjan [AMOT90] descrevem um algoritmo $O(m + n \log C)$. Este usa uma estrutura baseada na estrutura de bucket, denominada *radix heap*. Além disso, os autores mostram que esta estrutura, quando combinada com o heap de Fibonacci, reduz a complexidade para $O(m + n\sqrt{\log C})$.

O trabalho de Cherkassky *et al.* [CGR94] mencionado acima, reúne 17 implementações de algoritmos para caminhos mínimos. Além desse artigo, há outros que fazem estudos similares. Outra referência completa e bastante citada é o artigo de Gallo e Pallotino [GP88]. Em [Iso02] encontra-se uma dissertação que analisa e implementa diversas variações dos algoritmos de Dijkstra, de Thorup e outros algoritmos utilizando grafos gerados aleatoriamente.

A Seção 3.1 apresenta uma breve discussão sobre filas de prioridades. A Seção 3.2 apresenta o pseudocódigo mais detalhado do algoritmo de Dijkstra explicitando as operações `Insere`, `DiminuiChave` e `ExtraiMin`. Além disso, é apresentado um quadro comparativo das complexidades da implementação ingênua e dos algoritmos que implementam a fila de prioridades como heap binário, heap de Fibonacci, estrutura de bucket e radix heap. As descrições das quatro estruturas mais sofisticadas encontram-se nas Seções 3.3-3.6. A apresentação é informal e o objetivo principal é descrever o funcionamento das operações. Não houve preocupação quanto à formalização nem quanto à análise de complexidade. Uma referência completa com análises de corretude e complexidade pode ser encontrada em [AMO93] ou em [CLR99]. A Seção 3.7 apresenta o programa desenvolvido para este trabalho. Ele implementa a versão ingênua, o heap binário, o heap de Fibonacci e a estrutura de bucket. O radix heap não foi implementado devido às restrições de tempo. Os testes utilizaram nove mapas do mundo real disponibilizados pelo DIMACS [DGJ06].

3.1 Filas de prioridades

Filas de prioridades têm diversas aplicações. Podem ser utilizadas por sistemas operacionais para escalonar os processos a serem executados ou para gerenciar as impressões a serem atendidas, por algoritmos para ordenação de conjuntos de dados, por protocolos de redes para garantir a qualidade de serviço na transferência de pacotes, por sistemas de gerência de memória para substituir as páginas de memória menos utilizadas, dentre outras aplicações. Há inúmeros possíveis critérios de prioridade tais como freqüência de uso, data de nascimento, tempo gasto, posição, status e outros [Dro02]. Em geral, sua utilização em diferentes contextos está associada à necessidade de manipular um conjunto de itens de forma diferenciada da fila convencional, cujo princípio é o de "primeiro a entrar é o primeiro a sair" (FIFO - *first in first out*). A idéia básica é a de uma fila de espera com elementos aos quais são atribuídas preferências que determinam a ordem de atendimento [AHU87].

3.1.1 Conceitos

Conforme [Ziv04, p.107], a estrutura de dados denominada fila de prioridades tem essa denominação "porque a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente". Outro conceito é apresentado em [CLR99, p.149], onde "uma fila de prioridades é uma estrutura de dados que mantém um conjunto de elementos, cada um associado a um *valor chave*". Em [AMO93, p.773] o conceito de fila de prioridades é considerado equivalente ao de heap, sendo inicialmente definido como uma "estrutura de dados para armazenamento e manipulação eficientes de uma coleção H de elementos (ou objetos) onde cada elemento $i \in H$ tem um número real associado, denotado por $key(i)$". Este número pode, em um sentido mais amplo, pertencer a qualquer conjunto ordenado linearmente.

O desafio na utilização de uma fila de prioridades é encontrar uma implementação eficiente que permita uma rápida colocação e uma rápida retirada da fila. Como os elementos podem ser inseridos em qualquer ordem na fila, não há garantia de que os elementos frontais serão os mais prováveis de serem retirados ou de que os elementos colocados no final serão os últimos candidatos para a retirada da fila.

3.1.2 Operações

As operações mais comuns sobre o tipo abstrato de dados fila de prioridades são:

1. *Insere* um item novo na fila de prioridades.
2. *ExtraiMin* que retira o item da fila de prioridades que contém o menor valor chave.

3. *DiminuiChave* que diminui o valor chave de um item da fila de prioridades.

Entretanto, este tipo abstrato de dados pode ser empregado de tal modo a suportar algumas das seguintes operações adicionais [AMO93, Ziv04]:

1. *Constrói* uma fila de prioridades a partir de um conjunto com n itens.
2. *Consulta* os dados de um item da fila de prioridades.
3. *Mínimo* que devolve o item da fila de prioridades com menor valor chave.
4. *Remove* um item arbitrário da fila de prioridades.
5. *Altera* o valor da chave de um item da fila de prioridades.
6. *Une* duas filas de prioridades em uma nova fila de prioridades.

Vale ressaltar que apenas as operações `Insere`, `ExtraiMin` e `DiminuiChave` são utilizadas nas implementações do algoritmo de Dijkstra apresentadas nas seções subseqüentes.

É possível implementar essas operações através da representação da fila de prioridades por meio de variadas estruturas de dados, estáticas ou dinâmicas. Na implementação ingênua, o custo da operação `Insere` é $O(1)$ e o custo tanto de `ExtraiMin` quanto de `DiminuiChave` é $O(n)$. Por outro lado, se a fila de prioridades for implementada como uma lista linear *ordenada*, o custo de `Insere` é $O(n)$, o custo de `ExtraiMin` é $O(1)$ enquanto o custo de `DiminuiChave` é $O(n)$.

Outra possibilidade é usar uma árvore de busca balanceada, como por exemplo, uma árvore AVL, que realiza cada operação básica em tempo $O(\log n)$. Entretanto esta estrutura possui funcionalidades que vão além das necessidades de uma fila de prioridades, além de utilizar espaço adicional para armazenar o endereçamento das subárvores [Pre00].

Uma representação particularmente eficiente de fila de prioridades utiliza uma estrutura de dados denominada *heap*. Heaps possuem uma variedade de aplicações nos algoritmos de fluxos em redes. Duas tais aplicações são o algoritmo de Dijkstra para caminhos mínimos e o algoritmo de Prim para o problema da árvore geradora mínima [AMO93].

No restante do capítulo, são apresentadas quatro variações do algoritmo de Dijkstra que usam o heap binário, heap de Fibonacci, estrutura de bucket e radix heap. O que segue é uma descrição informal das operações de fila de prioridades para cada uma dessas estruturas. Para uma referência completa com demonstrações das corretudes e das complexidades, veja por exemplo [AMO93, CLR99].

3.2 Algoritmo de Dijkstra modificado

Nesta seção será apresentado o algoritmo de Dijkstra modificado que é similar ao Algoritmo 2.3 apresentado na Seção 2.2. No Algoritmo 3.1, foram incluídas explicitamente as operações `Insere` e `DiminuiChave`.

Algoritmo 3.1 Algoritmo de Dijkstra Modificado

```
1:  Inicializa(G, s)
2:  S ← ∅
3:  Q ← ∅
4:  Insere(s, Q)
5:  while Q ≠ ∅ do
6:    u ← ExtraiMin(Q)
7:    for all v ∈ Adjacencia[u] do
8:      valor ← d[u] + w(u, v)
9:      if d[v] > valor then
10:       if v ∉ Q then
11:         d[v] ← valor
12:         p[v] ← u
13:         Insere(v, Q)
14:       else
15:         p[v] ← u
16:         DiminuiChave(valor, v, Q)
17:   S ← S∪{u}
```

Esta versão pode ser utilizada com todas as estruturas de dados vistas a seguir. A operação `Inicializa` é a mesma do Algoritmo 2.1. Percebe-se claramente que as operações `Insere` (linhas 4 e 13) e `DiminuiChave` (linha 16) não estão presentes no Algoritmo 2.3.

Outra diferença nesta versão é que o conjunto $\mathcal{Q}$, após ser inicializado na linha 3, recebe o vértice fonte na linha 4 em vez de receber o conjunto de vértices de G. Todavia, ambas as versões são equivalentes, tanto em termos de análise quanto de complexidade, uma vez que serão feitas (no máximo) n inserções no heap.

Observando o Algoritmo 3.1, pode-se concluir que o tempo total $T(D)$ consumido (no pior caso) pode ser descrito pela fórmula:

$$T(D) = T(Inicializa) + n \times T(Insere) + n \times T(ExtraiMin) + m \times T(DiminuiChave)$$

A Tabela 3.1 apresenta um quadro comparando o custo de cada operação para cinco diferentes tipos de representação de filas de prioridade. Na tabela também é indicada a complexidade do algoritmo de Dijkstra usando cada uma dessas representações.

Variação	Inicializa	Insere	ExtraiMin	DiminuiChave	T(D)
Ingênua	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
Heap Binário	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((m+n)\log n)$
Bucket	$O(n)$	$O(1)$	$O(C)$	$O(1)$	$O(m+nC)$
Heap de Fibonacci	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(m+n\log n)$
Radix Heap	$O(n)$	$O(\log C)$	$O(\log C)$	$O(1)$	$O(m+n\log C)$

Tabela 3.1: Custos das operações para as diferentes estruturas e a complexidade do algoritmo de Dijkstra resultante.

Destas estruturas, foram implementadas as quatro primeiras (veja a Seção 3.7). A última, o radix heap, é um refinamento da idéia de bucket. A versão do algoritmo de Dijkstra com esta estrutura de dados não foi implementada devido às restrições de tempo para a realização deste trabalho.

3.3 Algoritmo de Dijkstra com heap binário

Um heap (binário) consiste de um vetor A de n posições organizado da seguinte forma:

1. o *pai* de um elemento $A[i]$ é $A[\lfloor i/2 \rfloor]$, se $i > 1$,
2. o *filho esquerdo* de $A[i]$ é $A[2i]$, se $2i \leq n$,
3. o *filho direito* de $A[i]$ é $A[2i+1]$ se $2i+1 \leq n$.

Note que nesta representação, não é necessário armazenar o índice do pai de um elemento e nem os índices dos filhos; suas posições podem ser calculadas durante a execução do algoritmo. Assim, um heap binário pode ser visto como uma árvore binária completa. Vale a pena lembrar que uma árvore binária completa com n nós, tem altura $\lfloor \log n \rfloor$.

Além da organização acima, um heap deve satisfazer a seguinte *propriedade de heap*:

$$A[\lfloor i/2 \rfloor] \leq A[i],$$

para todo $i = 2, \ldots, n$. Ou seja, o pai é menor ou igual ao filho.

Um exemplo de heap binário é dado na Figura 3.1 juntamente com sua representação por meio de árvore binária completa.

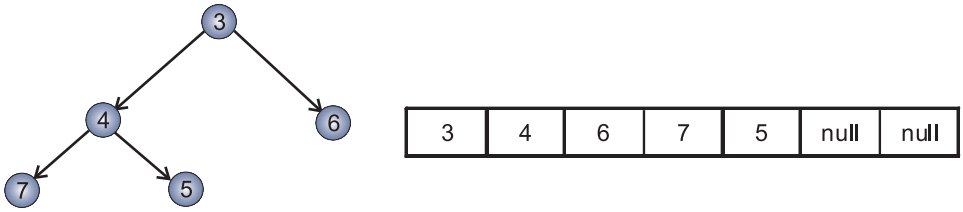

Figura 3.1: Heap binário: representação por árvore binária completa e armazenamento por vetor. A árvore possui 5 nós, então as 5 primeiras posições do vetor são ocupadas para armazenar o heap em questão.

3.3.1 Operações sobre o heap binário

As operações que manipulam o heap devem manter a propriedade de heap. As inserções e remoções, nesta ordem, se baseiam nas idéias de "puxar para cima" e "empurrar para baixo" na árvore o elemento que provocar a não observância dessa propriedade, para restaurá-la.

A operação `Insere` coloca o novo elemento na última posição livre do vetor. Ela "sobe" recursivamente na árvore até que a propriedade de heap seja verificada ou que a raiz seja alcançada. A Figura 3.2 demonstra como é realizada a inserção da chave 2 no heap.

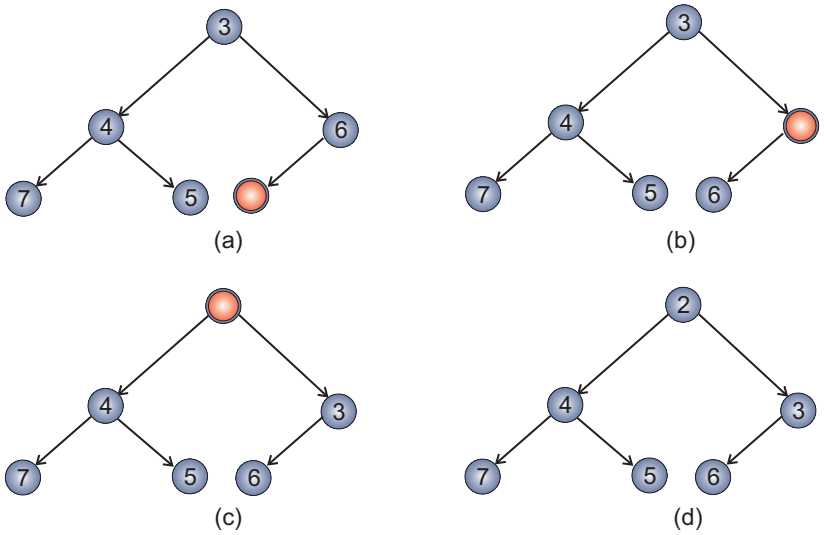

Figura 3.2: Inserindo a chave 2 num heap binário. (a) A lacuna para receber a chave é a primeira posição livre da esquerda para a direita no último nível da árvore. Esta lacuna é "puxada para cima" até que se obedeça a propriedade de heap. (b) A chave 2 foi trocada com a chave 6 e depois (c) foi trocada com a chave 3. (d) Heap final.

A operação `DiminuiChave` diminui o valor de um elemento $A[i]$. Esta operação pode

temporariamente violar a propriedade de heap. Por exemplo: supondo que a chave $A[i]$ seja diminuída e que $A[i] \geq A[\lfloor i/2 \rfloor]$. Neste caso, o heap ainda satisfaz a propriedade de heap, caso contrário, a operação é similar ao `Insere`. Ela "sobe" recursivamente na árvore, iniciando em i, até que a propriedade de heap seja verificada ou que a raiz seja alcançada.

A operação `ExtraiMin` extrai a raiz da árvore e a substitui pelo último elemento do vetor. A operação "desce" recursivamente na árvore até que a propriedade de heap seja verificada ou que uma folha seja alcançada. A Figura 3.3 demonstra a extração do mínimo na árvore.

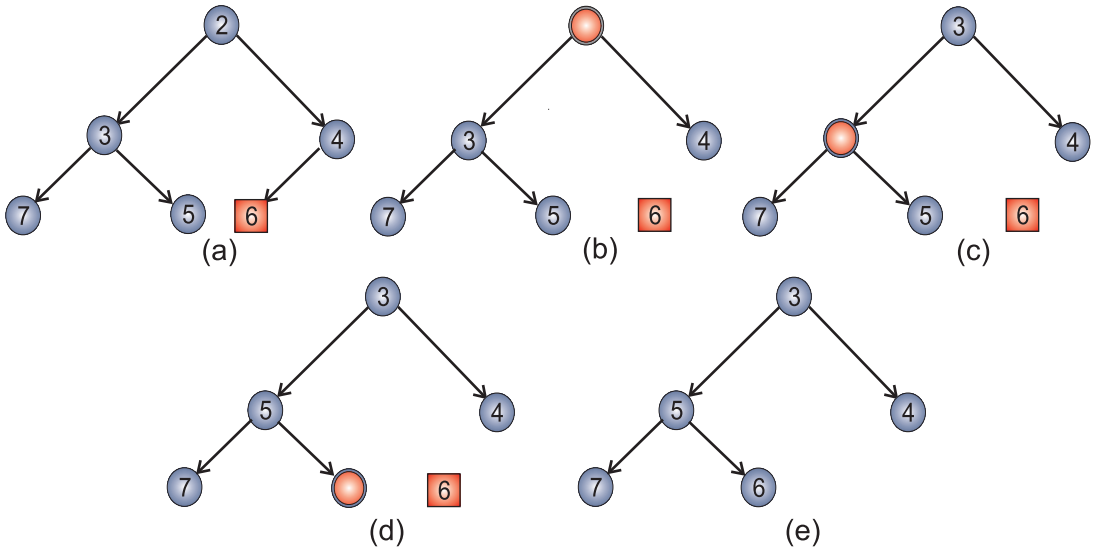

Figura 3.3: Extraindo o valor mínimo do heap binário. (a) A chave candidata a substituir a raiz da árvore é o nó mais à direita do último nível da árvore, a chave 6. Em seguida, a lacuna onde o 6 será alocado é então "empurrada para baixo" até que a propriedade de heap seja verificada ou até que uma folha seja alcançada. (b)-(d) Sempre que a lacuna for movida para baixo, ou seja, sempre que a chave 6 for maior que as subárvores da lacuna, a mesma é trocada de posição com o nó da subárvore de menor valor. (e) é o heap final.

Cada uma das três operações acima pode ser executada em tempo $O(\log n)$. Maiores detalhes sobre heaps podem ser encontrados em [CLR99] e [AMO93].

3.3.2 Complexidade do algoritmo de Dijkstra usando o heap binário

Como mostrado no início desta seção, um heap requer tempo $O(\log n)$ para cada operação de inserção, remoção e diminuição de chave. Considere o Algoritmo 3.1 (al-

goritmo de Dijkstra modificado). A operação `Inicializa` na linha 1 do heap gasta tempo $O(n)$. O loop **while** das linhas 5-17 do Algoritmo 3.1 é executado $O(n)$ vezes e, como a operação `ExtraiMin` tem custo $O(\log n)$, o tempo total gasto com esta operação é $O(n \log n)$. Além disso, o loop **for** das linhas 7-16 analisa cada aresta no máximo uma vez, totalizando $O(m)$ iterações. Cada teste da linha 9 implica numa chamada a `Insere` ou a `DiminuiChave` (ambas têm custo $O(\log n)$). O tempo total gasto nestas duas operações é $O(m \log n)$. Assim, o tempo total do algoritmo é $O(n \log n + m \log n) = O((m+n) \log n)$.

3.4 Algoritmo de Dijkstra usando heap de Fibonacci

O heap de Fibonacci é uma estrutura de dados introduzida por Fredman e Tarjan [FT87] que permite que as operações sobre o heap sejam executadas mais eficientemente do que os heaps binários [AMO93]. Este heap suporta as mesmas operações que os anteriores, com a vantagem que cada operação que não envolve a remoção de um elemento (inclusive as operações adicionais de filas de prioridades) pode ser executada em tempo amortizado $O(1)$ [CLR99].

Intuitivamente, a complexidade amortizada de uma operação é a complexidade *média* de pior caso para executar tal operação. A análise amortizada difere da análise do caso médio já que probabilidade não é envolvida; uma análise amortizada garante o desempenho médio de cada operação no pior caso [AMO93, CLR99].

3.4.1 O heap de Fibonacci

Historicamente, o propósito original do desenvolvimento dos heaps de Fibonacci[1] foi a aceleração do algoritmo de Dijkstra. Como foi observado por Fredman e Tarjan [FT87], existem potencialmente mais operações `DiminuiChave` do que `ExtraiMin`. Então, qualquer método que reduzisse o tempo amortizado da operação `DiminuiChave`, sem aumentar o tempo amortizado de `ExtraiMin`, poderia produzir uma implementação assintoticamente mais rápida [CLR99]. Algoritmos para outros três problemas também foram melhorados com a utilização dos heaps de Fibonacci: o problema dos caminhos mínimos entre todos os pares de vértices, o problema da árvore geradora mínima e o problema de emparelhamento de peso máximo em grafos bipartidos [FT87].

O heap de Fibonacci permite que as operações `Insere` e `DiminuiChave` possam ser executadas em tempo amortizado $O(1)$ e a operação `ExtraiMin` em tempo amortizado $O(\log n)$.

[1] O nome heap de Fibonacci é devido à prova de seu tempo usar as propriedades dos conhecidos números de Fibonacci. Estes são definidos recursivamente como $F(1) = 1, F(2) = 1$ e $F(k) = F(k-1) + F(k-2)$, para todo $k \geq 3$. Esses números aparecem na análise de complexidade das operações.

Sob o ponto de vista prático, no entanto, os fatores constantes e a complexidade da programação dos heaps de Fibonacci os tornam menos desejáveis que os heaps binários (ou k-ários) na maioria das aplicações. Assim, os heaps de Fibonacci são de interesse predominantemente teórico.

Um heap de Fibonacci consiste numa coleção de árvores enraizadas com a *propriedade de heap*: a chave de um elemento é menor ou igual à chave do seu filho. Há duas operações fundamentais para manipular um heap de Fibonacci: *ligação* e *corte*. Todas as operações sobre filas de prioridades podem ser implementadas como uma combinação de uma seqüência dessas, com exceção da operação `Insere`.

- `Ligação(x,y)`: combina duas árvores com raízes x e y da seguinte forma. Se a chave de x for menor que a de y, y torna-se filho de x, caso contrário x torna-se filho de y.
- `Corte(x)`: apaga a aresta $(x, pai(x))$ formando duas novas árvores. O nó x torna-se raiz da árvore que o contém. A raiz da outra árvore continua sendo a raiz da árvore original.

As Figuras 3.4 e 3.5 mostram como funcionam as operações `Ligação` e `Corte`.

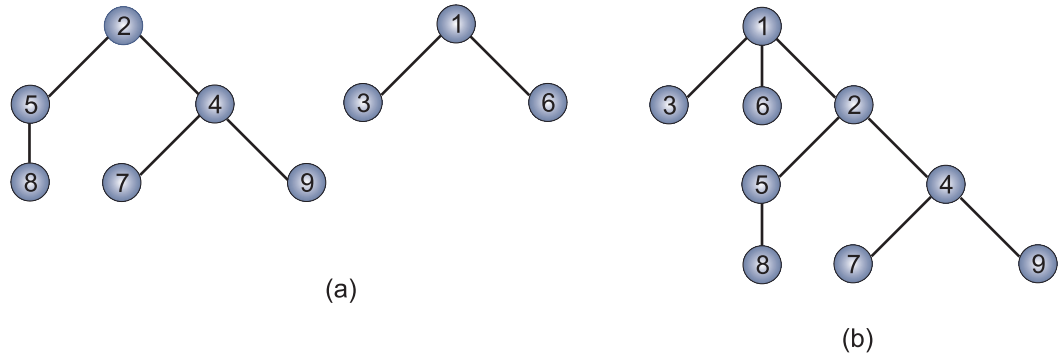

Figura 3.4: Ligação de duas árvores em um heap de Fibonacci: (a) heap original e (b) heap após `Ligação(1,2)`.

Cada nó de um heap de Fibonacci deve satisfazer a seguinte restrição: um nó pode perder no máximo um filho a partir do momento em que se tornou um nó não raiz. Dizemos que um nó é *marcado* se ele não é raiz e perdeu exatamente um filho.

Pela propriedade de heap, a raiz de qualquer árvore contém a menor chave dessa. Assim, o mínimo do heap é uma das raízes. As árvores não têm nenhuma forma especial e em princípio, cada árvore poderia consistir de um único nó ou o heap poderia ser formado apenas por uma única árvore de profundidade n.

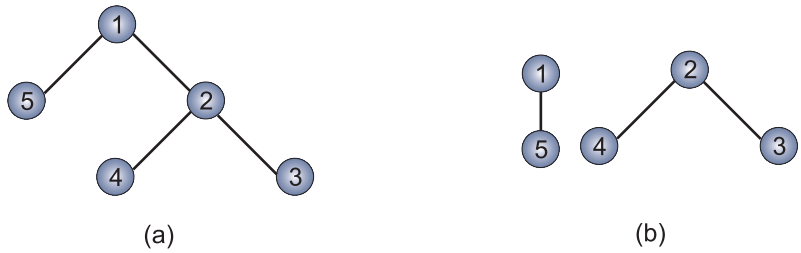

Figura 3.5: Corte do nó 2 num heap de Fibonacci: (a) heap original e (b) heap após `Corte(2)`.

Esta flexibilidade permite que algumas operações possam ser executadas de maneira "despreocupada", adiando o trabalho de "arrumar" o heap para as operações posteriores, quando for mais conveniente.

Representação de um heap de Fibonacci

Para representar um heap de Fibonacci pode-se definir os seguintes atributos e seus conteúdos como segue. Para cada vértice são necessários os apontadores:

- *pai*, representa o pai do nó ou 0 se é uma raiz;
- *filho*, qualquer um dos filhos do nó;
- *direita*, irmão da direita;
- *esquerda*, irmão da esquerda;

e dois atributos:

- *rank*, o número de filhos e
- *marca* que indica se o nó foi marcado.

As raízes de todas as árvores do heap e os filhos de cada nó estão em duas listas duplamente encadeadas circulares. Um heap de Fibonacci é acessado por meio de um ponteiro, *min[H]* para a raiz de uma das árvores que contém a menor chave do heap.

A Figura 3.6 mostra um exemplo de um heap de Fibonacci. O mesmo heap de Fibonacci é representado na Figura 3.7 por meio de uma tabela com os atributos recém explicados. Por fim, a Figura 3.8 ressalta os encadeamentos dos ponteiros utilizados para o mesmo heap de Fibonacci. Nestas e nas demais figuras, não há distinção entre os items e suas chaves.

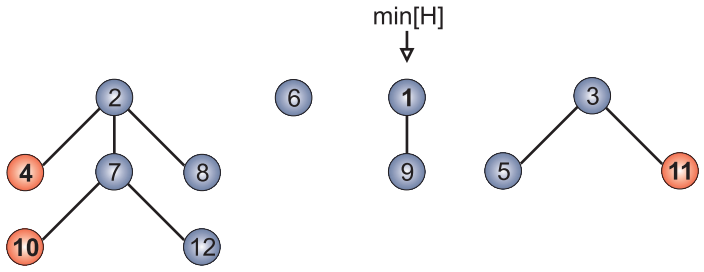

Figura 3.6: Heap de Fibonacci: coleção de árvores heap-ordenadas. Os nós marcados estão sombreados.

i	1	2	3	4	5	6	7	8	9	10	11	12
pai(i)	0	0	0	2	3	0	2	2	1	7	3	7
esquerda(i)	1	2	3	8	11	6	4	7	9	12	5	10
direita(i)	1	2	3	7	11	6	8	4	9	12	5	10
filho(i)	9	4	11	0	0	0	10	0	0	0	0	0
rank(i)	1	3	2	0	0	0	2	0	0	0	0	0
marca(i)	V	F	F	V	F	F	F	F	F	V	V	F

Figura 3.7: Heap de Fibonacci: tabela com os atributos do heap.

3.4.2 Operações sobre um heap de Fibonacci

Aqui serão descritas as operações de filas necessárias para o algoritmo de Dijkstra.

A operação `Insere` cria uma nova árvore contendo apenas o novo elemento, atualizando min[H] se necessário. O custo de `Insere` é $O(1)$. As operações `DiminuiChave` e `ExtraiMin` são implementadas como uma seqüência de ligações e cortes.

A operação `DiminuiChave(v,x,H)` atualiza o valor da chave de x para v e então verifica se esta atribuição violou a propriedade de heap. Se não for violada, então nenhuma mudança é necessária. Caso contrário, x precisa ser separado de seu pai y por meio de uma operação `Corte`. Neste momento precisa-se verificar se y está marcado. Se assim o for, é necessário cortar y de seu pai z e assim sucessivamente, realizando o que se denomina "cortes em cascata" até que uma raiz ou um nó não marcado seja alcançado. Em princípio, esta operação pode gastar tempo $O(n)$. Entretanto, usando análise amortizada, pode-se mostrar que o tempo amortizado de `DiminuiChave` é $O(1)$[2].

A Figura 3.9 mostra uma execução de `DiminuiChave` que provoca os cortes em cascata. Esta seqüência de cortes é também denominada de "multicascateamento" [AMO93].

A operação `ExtraiMin` começa removendo o nó de chave mínima x (apontado por min[H]) e concatenando a lista dos filhos de x com a lista das raízes de H (são promovidos a raízes). Em seguida, repete-se o seguinte procedimento: enquanto houver

[2]Uma demonstração particularmente interessante deste limite é dada em [CLR99]

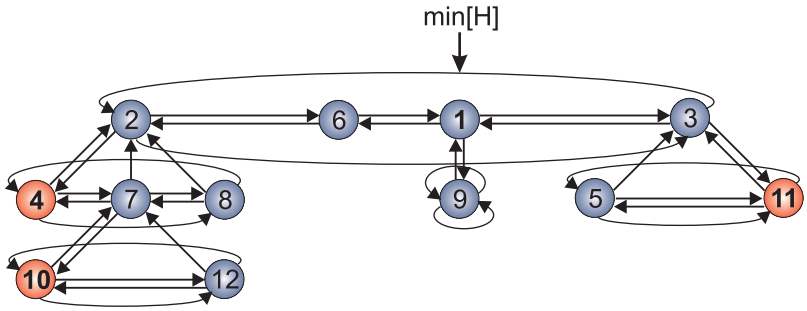

Figura 3.8: Heap de Fibonacci: listas duplamente encadeadas circulares (LDEC) de raízes. Cada nó x contém um ponteiro, $filho[x]$, para qualquer um dos seus filhos. Se um nó x é um filho único, então $esquerda[x] = direita[x] = x$. Se é uma raiz, $pai[x] = 0$. Se é uma folha, $filho[x] = 0$.

duas raízes x e y com o mesmo *rank*, executa-se a operação `Ligação(x,y)`. Ao final, a variável min$[H]$ é atualizada. A seqüência de passos de `ExtraiMin` é demonstrada na Figura 3.10. Novamente, é possível que uma operação `ExtraiMin` gaste tempo $O(n)$, mas usando análise amortizada pode-se mostrar que o tempo amortizado é $O(\log n)$.

3.4.3 Complexidade do algoritmo de Dijkstra usando o heap de Fibonacci

Considere novamente o Algoritmo 3.1. A operação `Inicializa` gasta tempo $O(n)$. As operações `Insere` e `DiminuiChave` gastam tempo (amortizado) $O(1)$ e `ExtraiMin` gasta tempo (amortizado) $O(\log n)$. Assim, o tempo total do algoritmo de Dijkstra com heap de Fibonacci é $O(m + n \log n)$.

3.5 Algoritmo de Dijkstra com bucket

A estrutura de *bucket*, proposta em um artigo de Dial [Dia69] (e independentemente por Wagner [Wag76]), foi concebida com o objetivo de reduzir o tempo gasto pela operação `ExtraiMin` no algoritmo de Dijkstra. A complexidade desse algoritmo depende de C, o peso máximo das arestas de G.

Para entender a idéia chave do algoritmo, convém lembrar como o algoritmo de Dijkstra funciona. Enquanto a fila $\mathcal{Q}$ não estiver vazia, ele seleciona um vértice u com $d[u]$ mínimo, escaneia u e o insere em $\mathcal{S}$ (neste momento, $d[u] = \text{dist}(s, u)$). Além disso, tem-se que $d[x] \leq d[u]$ para todo $x \in \mathcal{S}$, ou seja, os vértices são inseridos em $\mathcal{S}$ em ordem não decrescente de distâncias.

O algoritmo de Dijkstra armazena os vértices de $\mathcal{Q}$ em um vetor de $(n-1)C + 1$

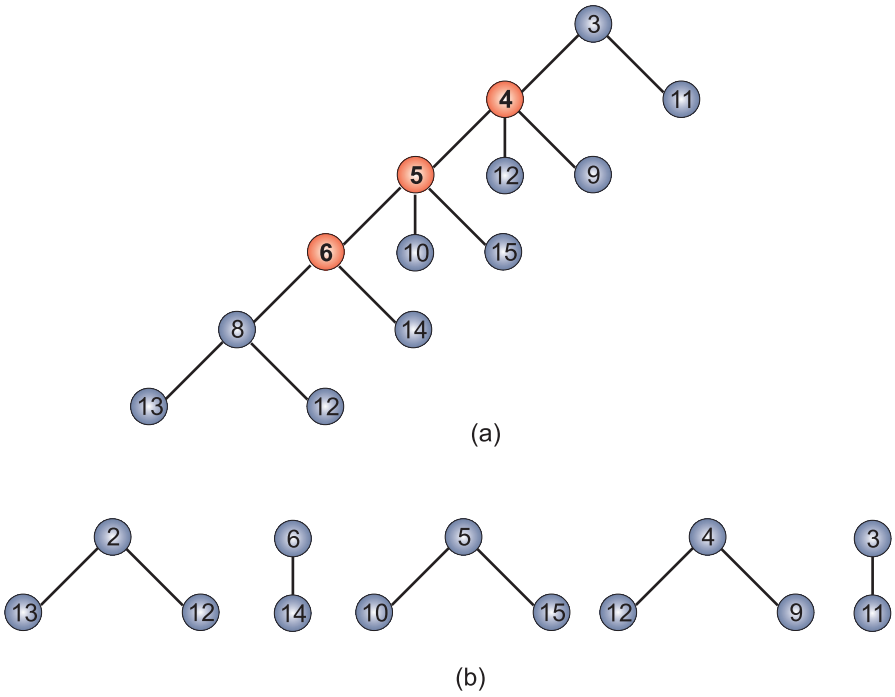

Figura 3.9: Diminuindo a chave em um heap de Fibonacci: (a) heap original; (b) heap após `DiminuiChave(2,8,H)`. O corte atual separa 8 de 6 e os cortes em cascata separam 6 de 5, 5 de 4 e 4 de 3.

buckets de forma que o k-ésimo bucket contenha todos os vértices v com $d[v] = k$. Os buckets são numerados de 0 a $(n-1)C$. Note que $(n-1)C$ é o limite superior para o peso de um caminho em G.

Para que o algoritmo possa ser utilizado, os pesos das arestas devem ser inteiros. Esta restrição não é muito forte pois basta multiplicar o peso de cada aresta por um inteiro suficientemente grande. Cada bucket pode ser implementado como uma lista duplamente encadeada [AMO93] ou como uma fila (FIFO) [CGR94]. Com isso, é possível em tempo constante, verificar se um bucket está vazio, inserir um vértice em um bucket ou remover um vértice de um bucket.

3.5.1 Operações sobre o bucket

A estrutura de bucket mantém um índice k, inicialmente com valor 0, que indica o último bucket "percorrido". Este índice satisfaz a propriedade de que todo bucket de posição $i < k$ está vazio. Ao longo das execuções, o valor de k não diminui. Note que no algoritmo de Dijkstra, o vértice a ser removido da fila deve pertencer ao bucket$[i]$ para algum $i \geq k$.

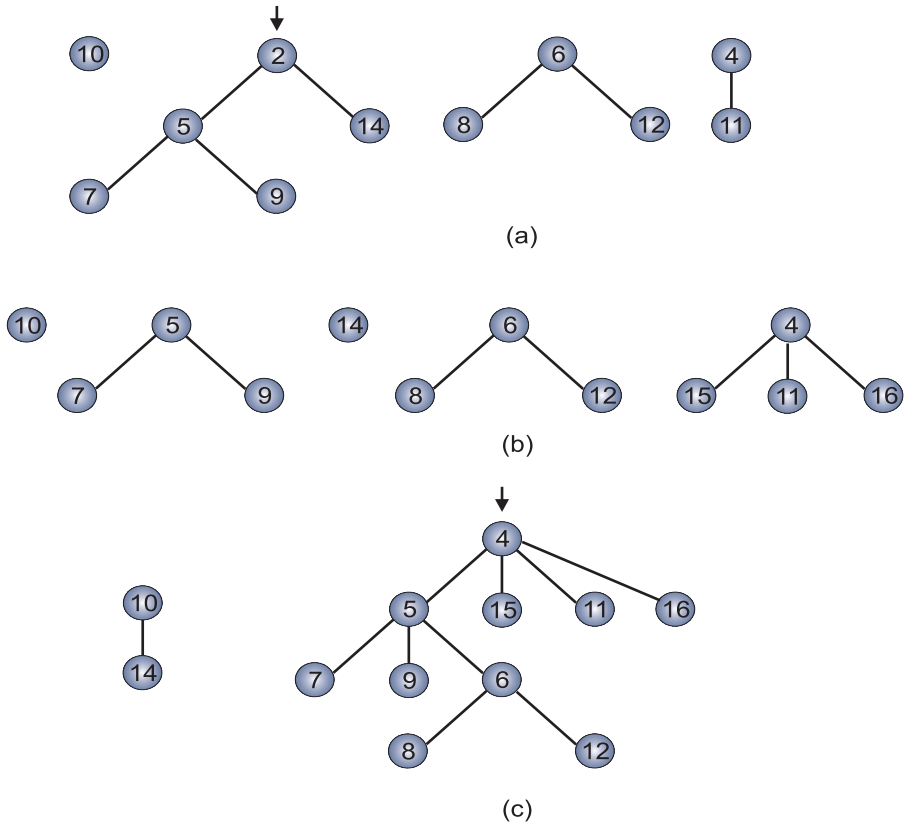

Figura 3.10: Extraindo o mínimo em um heap de Fibonacci: (a) Heap original; (b) Heap logo após a inserção dos filhos de 2 na lista de raízes. (c) Heap final. A operação `Ligação` é chamada 3 vezes: para as ligações de 10 e 14, depois de 5 e 6 e então de 5 e 4.

A operação `Insere` coloca o vértice v no bucket$[d[v]]$. Esta operação gasta tempo $O(1)$.

A operação `DiminuiChave` diminui o valor $d[v]$ de um elemento v. Se antes $d[v] = c_1$ e após a alteração, $d[v] = c_2$, então v é removido do bucket$[c_1]$ e inserido no bucket$[c_2]$. Esta operação também gasta tempo $O(1)$.

A operação `ExtraiMin` visita os buckets seqüencialmente a partir da posição k para encontrar o primeiro bucket não vazio, suponha o bucket i. Um elemento qualquer do bucket é removido e atualiza-se $k \leftarrow i$. Note que os buckets de 0 a $k-1$ estarão sempre vazios nas iterações subseqüentes. Esta operação gasta tempo $O(C)$ no pior caso.

Na implementação descrita acima, o vetor possui $(n-1)C+1$ buckets. Entretanto, este consumo de espaço pode ser proibitivo em várias aplicações. É possível reduzir o número de buckets para $C+1$ [AMO93] como é descrito a seguir.

Suponha que em alguma iteração do algoritmo, u é o vértice removido de $\mathcal{Q}$, cada aresta (u, v) de G é relaxada e u é colocado em $\mathcal{S}$. Logo, $d[x] \leq d[u]$ para todo x

em $\mathcal{S}$. Além disso, para cada vértice y em $\mathcal{Q}$, existe algum vértice x em $\mathcal{S}$ tal que $d[y] = d[x] + w(x, y)$. Como $d[x] \leq d[u]$ e $w(x, y) \leq C$, então $d[u] \leq d[y] \leq d[u] + C$. Em outras palavras, todos as estimativas dos vértices em $\mathcal{Q}$ estão entre $d[u]$ e $d[u] + C$. Assim, $C + 1$ buckets são suficientes para armazenar os vértices em $\mathcal{Q}$.

A Figura 3.11 mostra um possível esquema circular para representar a estrutura de bucket utilizada pelo algoritmo de Dial segundo essa idéia. Nela, um vértice v com estimativa $d[v]$ é armazenado no bucket $d[v] \mod (C + 1)$. Com este tipo de tabela *hash* circular, durante a execução do algoritmo, o bucket k armazena vértices com estimativas iguais a k, $k + (C + 1)$, $k + 2(C + 1)$ e assim sucessivamente. Neste esquema, deve-se usar um índice k que indica o bucket com a menor estimativa. As operações `Insere` e `DiminuiChave` podem ser facilmente modificadas de modo a gastar tempo $O(1)$. Já o `ExtraiMin` é implementado de modo similar, com a diferença de que no pior caso são percorridas $O(C)$ posições do vetor.

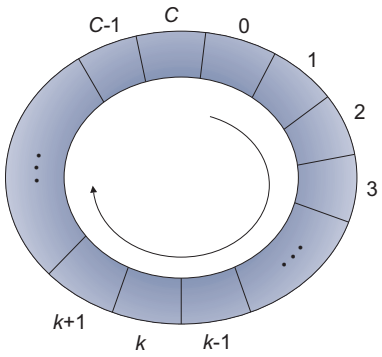

Figura 3.11: Estrutura de bucket circular. Neste esquema, caso o bucket k contenha a menor estimativa, então os buckets $k + 1$, $k + 2$, ..., C, 0, 1, 2, ..., $k - 1$ armazenam valores não decrescentes de estimativas.

3.5.2 Complexidade do algoritmo de Dijkstra usando o bucket

Considere o Algoritmo 3.1. A operação `Inicializa` gasta tempo $O(n)$. As operações `Insere` e `DiminuiChave` gastam tempo $O(1)$ e `ExtraiMin` gasta tempo $O(C)$. Assim, o tempo total do algoritmo de Dijkstra com estrutura de bucket é $O(m + nC)$.

Seu tempo de execução $O(m + nC)$ não é nem mesmo polinomial, antes sim pseudo-polinomial. Por exemplo, se $C = n^4$, o algoritmo consome tempo $O(n^5)$, mas para $C = 2^n$, o algoritmo consome tempo exponencial no pior caso. Entretanto, para muitas aplicações, C tem um tamanho modesto e o número de deslocamentos ao longo do vetor é menor que $n - 1$. Assim, na prática, o tempo de execução deste algoritmo é bem melhor do

que indica sua complexidade de pior caso. Filas de prioridades baseadas em estruturas de buckets são particularmente eficientes quando o valor máximo C é pequeno [CGS97].

Outra potencial desvantagem do Algoritmo de Dial comparado à implementação ingênua é que ela requer uma grande quantidade de armazenamento quando C é grande.

3.5.3 Outras estruturas baseadas em buckets

No estudo experimental [CGR94] a implementação do algoritmo de Dijkstra que utiliza o bucket de 2 níveis obteve o melhor resultado dentre as variações do Dijkstra testados, provando ser mais robusto que a implementação clássica que utiliza um bucket apenas. Em [GS95] é analisada a dependência do desempenho das implementações baseadas em buckets com o número de níveis de buckets utilizados e os resultados são comparados sob os pontos de vista teórico e pratico. Tais variações da estrutura de bucket têm sido elaboradas e combinadas com outras estruturas, como propõe [LO00], onde é formulado o *bucket-heap* que utiliza um heap binário para armazenar os buckets não vazios. Em [CGR94] são apresentadas duas implementações que visam diminuir a memória necessária pelo algoritmo de Dial: a *overflow bag* e a implementação do *approximate bucket*.

O *bucket duplo* (ou bucket de 2 níveis) separa os buckets em 2 diferentes níveis: os de *alto nível* e os buckets de *baixo nível*. O número dos buckets são determinados por um parâmetro constante B.

Genericamente, as variações da estrutura de dados de buckets podem ser reunidas nos chamados *buckets multi-níveis* propostos detalhadamente por Denardo e Fox [DF79] e analisados empiricamente em inúmeros trabalhos [GS95, CGS97, LO00, SWW00, KP06]. Nesta abordagem, os elementos são referenciados por meio do seu bucket e do nível do seu bucket. Por exemplo $B(i, j)$ representa o bucket j no nível i. A combinação destes buckets multi-níveis (também denominados k-buckets) com um heap binário deram origem ao *hot-queue* (*heap-on-top priority queue*) ou simplesmente "HQ", o qual tem se mostrado superior tanto teórica quanto praticamente em estudos experimentais como [CGS97].

3.6 Algoritmo de Dijkstra usando o radix heap

A variação do algoritmo de Dijkstra que utiliza o *radix heap* foi proposta por Ahuja, Mehlhorn, Orlin e Tarjan em [AMOT90]. Esta estrutura também explora a propriedade que sucessivas operações `ExtraiMin` do algoritmo de Dijkstra devolvem vértices em ordem não decrescente de distâncias. O algoritmo que utiliza o radix heap é um híbrido da implementação ingênua ($O(n^2)$) com a implementação de Dial ($O(m + nC)$). Ambas estão em extremos opostos no que se refere à quantidade dos buckets. Por um lado, a implementação ingênua manipula todos os vértices de $\mathcal{Q}$ como em um grande bucket. Por

outro lado, a implementação de Dial utiliza um grande número de buckets e separa os vértices de acordo com suas estimativas de distâncias [AMO93].

A variação do algoritmo que utiliza o radix heap melhora ambos os métodos acima. Ela adota a seguinte solução intermediária: armazena-se vários, mas não todos os elementos em um mesmo bucket. Por exemplo, em vez de armazenar em um bucket k apenas os vértices com $d[v] = k$ pode-se armazenar todos os vértices com $d[v]$ pertencendo ao intervalo $[100k, 100k + 99]$ no bucket k [AMO93].

Para o bucket$[k]$ é definido um *intervalo* de valores denotado por intervalo(k). O número de inteiros no intervalo é denominado de *largura* do intervalo e denotado largura(k). No exemplo acima, o intervalo do bucket k é $[100k, 100k + 99]$ e sua largura é 100. Além disso, cada bucket k é implementado como uma lista duplamente encadeada apontada por conteúdo(k).

Com essa representação, usar larguras de tamanho t permite a redução do número de buckets necessários por um fator de t. Entretanto, agora a operação `ExtraiMin` deve analisar todos os elementos no bucket não vazio de menor índice. Para superar esta desvantagem, foi proposta uma forma de estipular os intervalos para cada bucket de modo a reservar bucket$[0]$ para armazenar apenas os vértices de menor estimativa de distância [AMOT90, AMO93]. Atribuindo largura 1 a esse bucket torna-se possível manter as vantagens das soluções com "poucos" buckets e "muitos" buckets.

Neste novo esquema, o algoritmo mantém $\lceil \log(nC) + 1 \rceil$ buckets numerados de 0 até $K = \lceil \log(nC) \rceil$ (superando a desvantagem do Algoritmo de Dial de utilizar muitos buckets). O *limite superior*, $u(k)$, de um bucket k, é inicialmente definido em $2^k - 1$ para $k \geq 1$ e $u(0) = 0$. Assim, para $k \geq 1$, tem-se:

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

Os intervalos podem mudar durante a execução do algoritmo.

3.6.1 Operações sobre o radix heap

Note que determinar qual intervalo contém um dado valor pode ser feito em tempo $O(K) = O(\log nC)$ percorrendo o vetor de buckets. Assim, a operação `Insere` que coloca um vértice v na estrutura, gasta tempo $O(K)$.

A operação `DiminuiChave` que diminui a chave de um elemento v pode ser executada de modo similar. Em tempo $O(K)$ determina-se o bucket k tal que $d[v] \in$ intervalo(k). Após diminuir $d[v]$ basta determinar o novo bucket que conterá v o que pode ser feito em tempo $O(K)$.

Para descrever o `ExtraiMin` considere a seguinte exemplo: suponha que a menor estimativa é 9 e está no bucket 4 com intervalo(4)= $[8, 15]$. O algoritmo então percorre o bucket para encontrar um vértice v com $d[v] = 9$. Se os intervalos nunca fossem alterados, então os buckets $0, 1, 2$ e 3 nunca seriam usados novamente pois os vértices são removidos em ordem não decrescente de distância. Em vez de deixar esses buckets sem uso, o algoritmo *redistribui* o intervalo $[9, 15]$ nos buckets anteriores resultando nos intervalos intervalo(0)= $[9]$, intervalo(1)= $[10]$, intervalo(2)= $[11, 12]$, intervalo(3)= $[13, 15]$ e intervalo(4)= $\emptyset$. Como intervalo(4) ficou vazio, o algoritmo redistribui os vértices que estavam no bucket[4] nos buckets apropriados (de índices menores).

Para demonstrar o funcionamento do Algoritmo de Dijkstra com radix heap, seguem algumas figuras retiradas do livro *Network Flows* [AMO93]. A Figura 3.12 mostra o grafo sobre o qual será explicado o algoritmo.

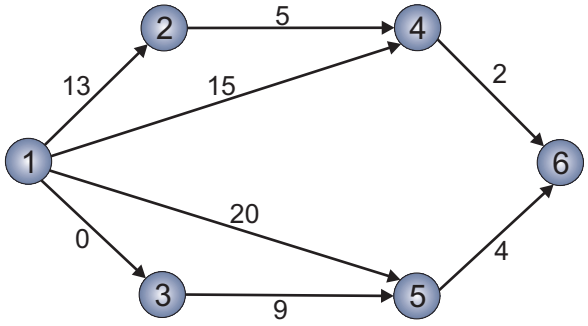

Figura 3.12: Grafo para exemplificar o radix heap. O vértice fonte neste exemplo é $s = 1$. O peso da maior aresta é $C = 20$ e então $K = \lceil \log(nC) \rceil = \lceil \log(120) \rceil = 7$.

As Tabelas 3.2, 3.3 e 3.4 mostram a execução das operações `Insere`, `ExtraiMin` e `DiminuiChave` na primeira iteração.

bucket k	0	1	2	3	4	5	6	7
$intervalo(k)$	$[0]$	$[1]$	$[2, 3]$	$[4, 7]$	$[8, 15]$	$[16, 31]$	$[32, 63]$	$[64, 127]$
$conteudo(k)$	1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tabela 3.2: Configuração do radix heap após o vértice 1 ser inserido.

A Tabela 3.5 mostra a configuração da estrutura ao final da segunda iteração.

i	1	2	3	4	5	6
$d[i]$	0	13	0	15	20	∞

bucket k	0	1	2	3	4	5	6	7
$intervalo(k)$	[0]	[1]	$[2,3]$	$[4,7]$	$[8,15]$	$[16,31]$	$[32,63]$	$[64,127]$
$conteudo(k)$	3	$\emptyset$	$\emptyset$	$\emptyset$	$2,4$	5	$\emptyset$	$\emptyset$

Tabela 3.3: Configuração do radix heap após a execução de `ExtraiMin`.

vértice i	2	4	5	6
rótulo $d[i]$	13	15	9	∞

bucket k	0	1	2	3	4	5	6	7
intervalo(k)	[0]	[1]	$[2,3]$	$[4,7]$	$[8,15]$	$[16,31]$	$[32,63]$	$[64,127]$
conteúdo(k)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$2,4,5$	$\emptyset$	$\emptyset$	$\emptyset$

Tabela 3.4: Configuração no início da segunda iteração. O bucket que continha a menor chave (3) era $k = 0$. O vértice 3 foi removido da estrutura, suas arestas foram relaxadas e ele foi inserido no conjunto $\mathcal{S}$. Ao relaxar a aresta $(3, 5)$ a chave do vértice 5 diminui de 20 para 9. A operação `DiminuiChave` verifica que $9 \notin$ intervalo(5). O vértice 5 então é movido para o bucket 4.

3.6.2 Complexidade do algoritmo de Dijkstra usando o radix heap

Cada operação `Insere` consome tempo $O(K)$.

As operações `ExtraiMin` e `DiminuiChave` movimentam vértices. Cada vértice pode ser movido no máximo K vezes. Assim, $O(nK)$ é um limite para o número total de movimentos de vértices. Como são feitas $O(m)$ chamadas a `DiminuiChave`, tem-se que $O(m+nK)$ é o tempo total gasto em atualizações de estimativas. A operação `ExtraiMin` envolve encontrar o primeiro bucket não vazio k no radix heap que gasta tempo $O(K)$ por iteração e portanto, $O(nK)$ no total. Isto é seguido de uma redistribuição do intervalo quando $k \geq 2$ e gasta tempo $O(K)$ por iteração e portanto, $O(nK)$ no total. Logo, `ExtraiMin` gasta tempo $O(nK)$.

Portanto, o tempo de execução do algoritmo de Dijkstra com radix heap é $O(m + nK) = O(m + n\log(nC))$. Utilizando a mesma idéia do algoritmo de Dial, é possível reduzir o número de buckets para $O(C)$ reduzindo a complexidade para $O(m + n\log C)$.

Para concluir a seção com variações do algoritmo de Dijkstra, vale a pena mencionar duas variações de radix heap recentes e mais eficientes. O radix heap apresentado aqui é

bucket k	0	1	2	3	4
intervalo(k)	$[9]$	$[10]$	$[11, 12]$	$[13, 15]$	$\emptyset$
conteúdo(k)	$\{5\}$	$\emptyset$	$\emptyset$	$\{2, 4\}$	$\emptyset$

Tabela 3.5: Configuração ao final da segunda iteração. O bucket $k = 4$ era o primeiro bucket não vazio e continha mais de um elemento. Tem-se que intervalo$(4) = [8, 15]$ mas como sua menor estimativa é 9, o novo intervalo a ser redistribuído será $[9, 15]$. Os vértices do bucket 4 foram redistribuídos nos buckets 0 a 3. Os demais intervalos não mudam.

também chamado radix heap de 1 nível. O algoritmo de Dijkstra com *radix heap de* 2 *níveis* consome tempo $O(m + n \log C / \log \log C)$. Além disso, uma combinação desse último com o heap de Fibonacci produz um algoritmo de tempo $O(m + n\sqrt{\log C})$ [AMOT90].

3.7 Resultados experimentais

Como exposto no Capítulo 1, o algoritmo de Dijkstra pode ser utilizado para resolver o problema de caminhos mínimos em grafos com arestas de pesos não negativos. Esta seção reúne algumas informações relevantes sobre o desenvolvimento de um programa que implementa o algoritmo de Dijkstra para o problema de fonte única, reportando os resultados obtidos.

A principal motivação desta implementação foi a avaliação experimental dos desempenhos do algoritmo de Dijkstra usando diferentes estruturas de dados. As variações implementadas foram: implementação ingênua, heap binário, estrutura de bucket e heap de Fibonacci. Para comparar os desempenhos, o programa foi executado com as quatro estruturas tendo como instâncias de entrada nove diferentes mapas do mundo real disponibilizados pelo DIMACS [DGJ06].

Estudos detalhados da manipulação destas estruturas de dados foram realizados para codificar da melhor forma as operações especificadas apenas em pseudocódigo na literatura estudada.

Recomendações referentes à condução de análises experimentais de algoritmos, como as de D. Johnson [Joh02] e McGeoch e Moret [MM99] foram consultadas e, na medida do possível, incorporadas.

3.7.1 Ambiente de teste e instâncias

O programa foi desenvolvido na linguagem C e compilado com a IDE Dev C++ versão 4.9.9.2 da *BloodShed* cujo compilador é o gcc versão 3.4.2. Não foram utilizadas opções de

otimização. Todos os testes foram executados em um *notebook DELL Latitude D520* com um *Intel® Celeron* M 420 de 1.6GHz com 512Mb de memória RAM rodando o *Microsoft Windows®XP Professional Service Pack 2*.

As instâncias recebidas como entrada, estão em arquivos texto no formato do *9th Implementation Challenge - Shortest Paths* do DIMACS [DGJ06]. Todos os arquivos utilizados pelo desafio, tanto de entrada quanto de saída, seguem as seguintes convenções:

- Arquivos consistem de caracteres ASCII (arquivos de texto)
- O conteúdo de um arquivo consiste de vários tipos de linhas
- Cada linha começa com um especificador de tipo de linha de um caracter
- Campos da linha são separados por pelo menos um espaço em branco
- Cada linha termina com um separador de quebra de linha
- Valores numéricos armazenados no arquivo (p.ex. pesos das arestas, identificador dos vértices etc) podem ser strings de dígitos de qualquer tamanho. Os programas devem carregá-los em palavras de memória grandes o suficiente (p.ex. 32 ou 64 bits, dependendo do contexto).

Por convenção, os nomes dos arquivos dos grafos para caminhos mínimos devem ter extensão `*.gr`. Para os arquivos de entradas dos grafos, são ainda utilizadas as seguintes regras:

Linhas de Comentário	Linhas de comentário podem aparecer em qualquer local e são ignoradas pelos programas	**c** `Isto é um comentário`
Linha do Problema	A linha do problema é única e deve aparecer como a primeira linha não comentada. O total de vértices é **n** e o total de arestas é **m**	**p sp** `n m`
Linhas das Arestas	`U` e `V` correspondem aos vértices de origem e destino de uma aresta, sendo `W` o seu peso	**a** `U V W`

A seguir, um exemplo de conteúdo de um arquivo que representa um grafo segundo estas regras:

```
c 9th DIMACS Implementation Challenge: Shortest Paths
c http://www.dis.uniroma1.it/~challenge9
c Arquivo exemplo de grafo
c
p sp 6 8
c o grafo contém 6 vértices e 8 arestas
c os identificadores dos vértices são numerados de 1 a 6
c
a 1 2 17
c aresta do vértice 1 para o vértice 2 de peso 17
a 1 3 10
a 2 4 2
a 3 5 0
a 4 3 0
a 4 6 3
a 5 2 0
a 5 6 20
```

Os arquivos dos grafos de entrada são lidos inicialmente pelo programa por meio de uma rotina de "parser" que gera uma representação de G utilizando listas de adjacências. O *9th Implementation Challenge* disponibiliza[3], para cada mapa do mundo real, seus respectivos grafos com a métrica de *distâncias* entre os cruzamentos (ruas, avenidas, estradas), métrica de *tempos* de percurso destas vias e outra terceira métrica com as *coordenadas geográficas* dos pontos nestes mapas. Neste trabalho foram utilizadas as métricas relativas às distâncias entre os cruzamentos.A Figura 3.13 reúne dados dos mapas do DIMACS utilizados pelo programa:

3.7.2 Implementação

O programa em C foi desenvolvido de maneira padronizada de modo que as comparações sobre os tempos de execução fossem significativas. Isto implica que o algoritmo principal, as subrotinas e seus parâmetros são praticamente os mesmos nas quatro variações. Além disso, os tipos de dados e algoritmos foram implementados na tentativa de serem os mais eficientes possível.

Ao arquivo main.c foram incluídos os arquivos .h que codificam (1) o parser, (2) os tipos de dados utilizados por cada variação, (3-6) os quatro tipos de estruturas, (7) a pilha usada para mostrar os caminhos mínimos.

[3]URL do *9th Implementation Challenge* - http://www.dis.uniroma1.it/ challenge9/

INFORMAÇÕES SOBRE OS GRAFOS UTILIZADOS					
Mapa	**n**	**m**	**C**	**longitude**	**latitude**
Roma	3.353	8.870	12.711	não consta	não consta
Nova Iorque	264.346	733.846	36.946	[40.3; 41.3]	[73.5; 74.5
São Francisco	321.270	800.172	94.305	[37.0; 39.0]	[121.1; 123]
Colorado	435.666	1.057.066	137.384	[37.0; 41.0]	[102.0;109.0]
Flórida	1.070.376	2.712.798	214.013	[24.0; 31.0]	[79; 87.5]
NO-EUA	1.207.945	2.840.208	128.569	[42.0; 50.0]	[116.0; 126.0]
NE-EUA	1.524.453	3.897.636	63.247	[39.5; 43.0]	[-infty; 76.0]
Califórnia	1.890.815	4.657.742	215.354	[32.5; 42.0]	[114.0; 125.0]
Grandes Lagos	2.758.119	6.885.658	138.911	[41.0; 50.0]	[74.0; 93.0]

Figura 3.13: Dados relativos aos grafos dos mapas utilizados como instâncias pela implementação.

A contagem dos clocks é executada logo após a chamada à subrotina da estrutura escolhida e ao final dos cálculos, ambos dentro da subrotina. Assim, não são computados os tempos gastos com entrada e saída dos dados. Cada chamada a uma estrutura retorna o total de *clock's* gastos. A medição utilizada é considerada de alta precisão já que não depende da plataforma de software e dos detalhes do *hardware* do computador. Ela retorna a quantidade de *clock's* do processador. Para apresentar os tempos obtidos em segundos, os *clocks* gastos foram divididos pela freqüência do processador usado[4].

Cada execução do programa realiza uma leitura do arquivo escolhido e vinte iterações da variação do algoritmo de Dijkstra escolhida. O vértice fonte em todas as execuções foi fixado no vértice de rótulo 1. Esta escolha não prejudicou os resultados obtidos já que outros valores foram testados obtendo-se o mesmo padrão de desempenho.

3.7.3 Resultados obtidos

Ao final de cada execução do programa, são mostrados os tempos gastos por cada uma das vinte execuções pré-programadas. Foram desprezados o maior e o menor tempo e calculada a média dos dezoito tempos resultantes. A Figura 3.14 reúne as médias dos tempos (em segundos) de todas as execuções realizadas.

As Figuras de 3.15 a 3.17 mostram em detalhes os tempos gastos por cada variação do algoritmo em cada mapa.

Os resultados obtidos demonstram claramente que os desempenhos dos algoritmos que utilizaram a estrutura de bucket e o heap binário foram superiores às implementações ingênua e usando o heap de Fibonacci na maioria dos mapas utilizados.

[4]O computador utilizado nos testes não possui *clock* variável.

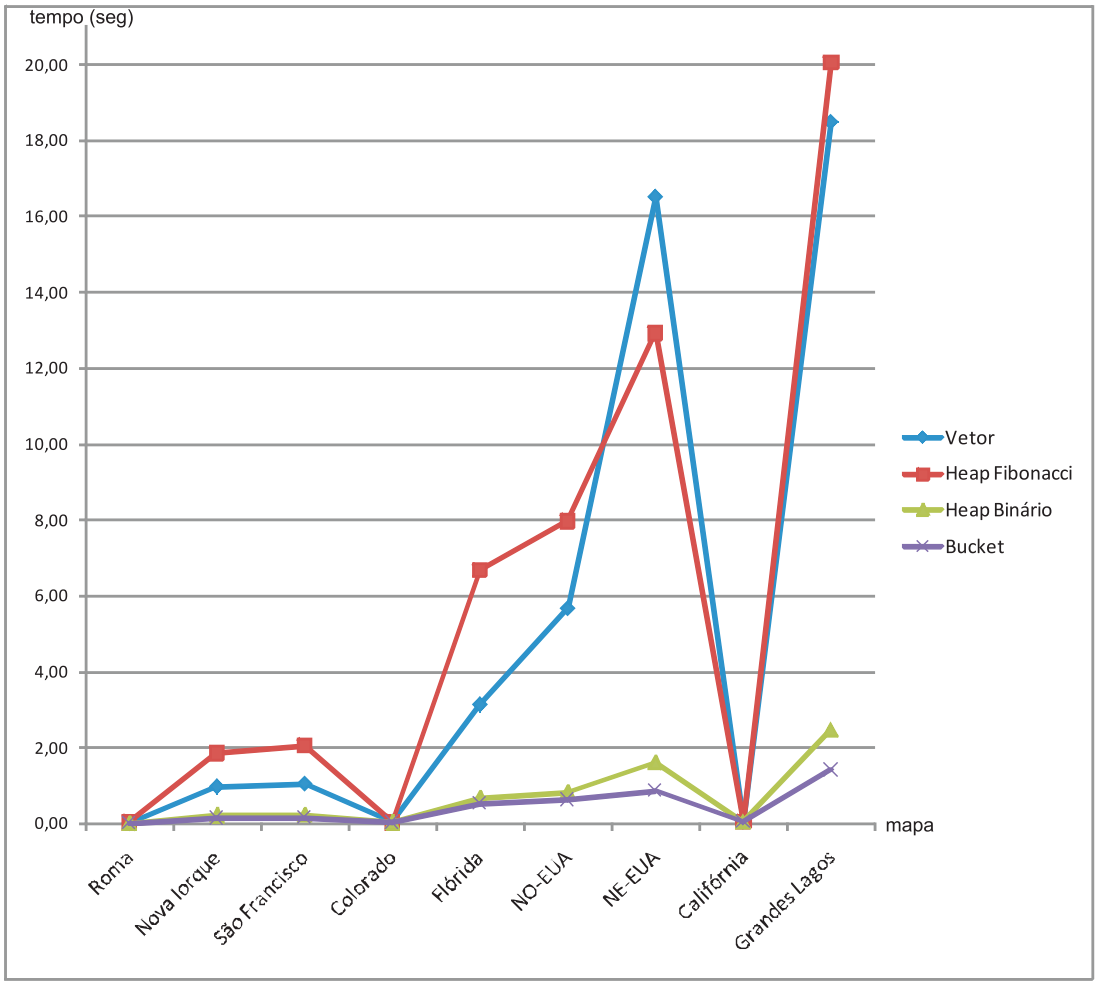

Figura 3.14: Gráfico com os resultados obtidos sobre todas as execuções das quatro variações do algoritmo de Dijkstra tendo como entradas nove mapas do mundo real.

Dos nove mapas seis obtiveram os melhores tempos usando a estrutura de bucket. O heap binário obteve o melhor tempo apenas no mapa de Roma, que é relativamente pequeno. Entretanto, os mapas do Colorado e da Califórnia apresentaram resultados diferentes e interessantes: o heap de Fibonacci apresentou os melhores tempos. Neste mapas todas as estruturas apresentaram baixos tempos, comparáveis com os do menor mapa. Isto nos leva a supor que alguma estrutura particular está favorecendo os heaps de Fibonacci. A densidade do grafo não parece ser um motivo para este comportamento já que as razões m/n dos mapas estão no intervalo de $2,35$ a $2,77$.

Estes dados estão de acordo com as observações da literatura vigente. O heap de Fibonacci de fato contrariou, na prática, seu desempenho teórico de melhores resultados, chegando a ser quinze vezes mais lento que o bucket no mapa NO-EUA, por exemplo.

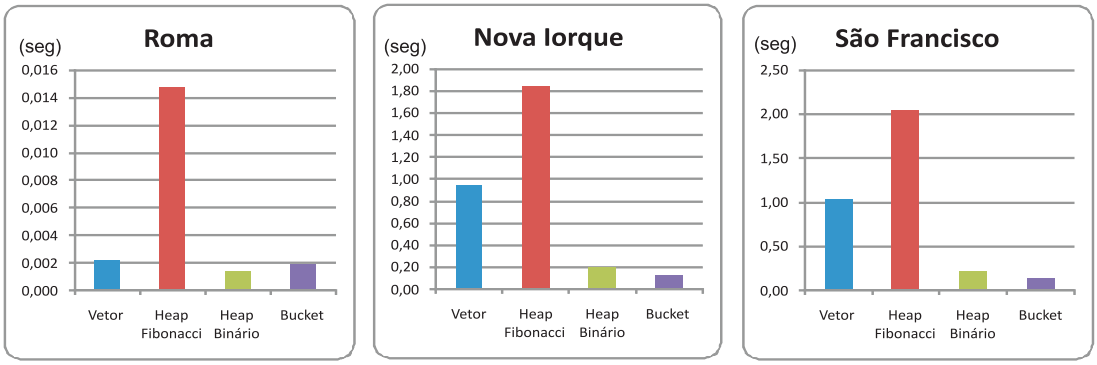

Figura 3.15: Gráficos com os resultados obtidos com os mapas de Roma (à esquerda), Nova Iorque (ao meio) e da Bahia de São Francisco (à direita).

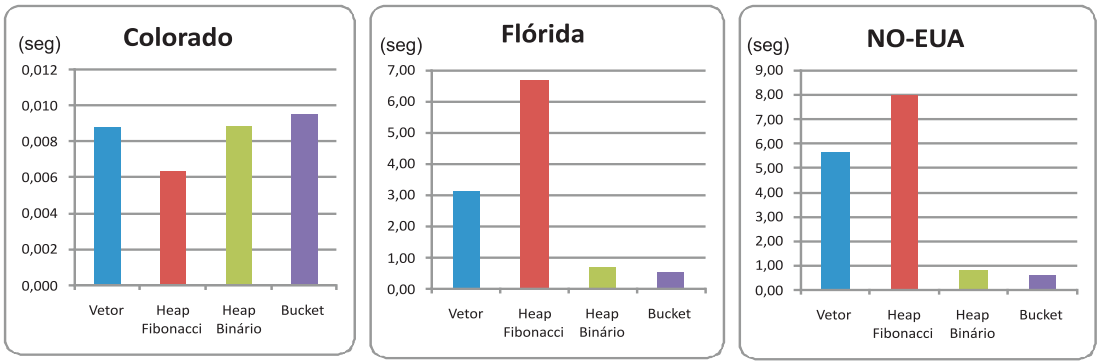

Figura 3.16: Gráficos com os resultados obtidos com os mapas de Colorado (à esquerda), Flórida (ao meio) e Noroeste dos Estados Unidos (à direita).

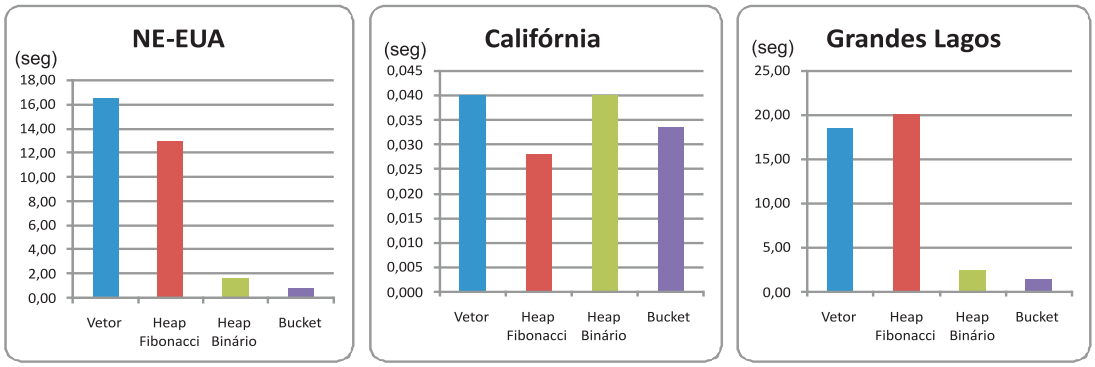

Figura 3.17: Gráficos com os resultados obtidos com os mapas do Nordeste dos Estados Unidos (à esquerda), Califórnia (ao meio) e Região dos Grandes Lagos (à direita).

Capítulo 4

Algoritmo de Dijkstra em problemas P2P

Algoritmos para encontrar um caminho mínimo entre dois vértices específicos num grafo são muito utilizados atualmente. Isto se explica pela vasta disseminação dos chamados Sistemas de Informações Geográficas (SIG). Estes sistemas estão presentes em vários contextos, atendendo demandas de indivíduos e organizações. Alguns exemplos de aplicação são: sistemas de navegação de carros, mapas cartográficos, serviços de infra-estrutura, rotas de empresas de transportes, imagens de bancos de dados espaciais, *engines* de buscas na Internet etc.

Os grafos que aparecem nessas aplicações são freqüentemente muito grandes. Mapas como os dos Estados Unidos e da Europa possuem cerca de 20 milhões de vértices. Naturalmente, isto proíbe qualquer estratégia de processamento (ou pré-processamento) que não seja linear ou ligeiramente superlinear. A busca por um caminho mínimo entre duas localizações específicas em um mapa[1] constitui um problema chave em inúmeras dessas aplicações. Este tipo de problema é conhecido como problema ponto-a-ponto (*point-to-point*) ou problema P2P.

Este capítulo é o resultado de uma pesquisa bibliográfica de trabalhos recentes e importantes da área de problemas P2P. São apresentados conceitos, técnicas e resultados presentes nesses trabalhos com o objetivo de fornecer uma visão geral do tipo de pesquisa feita nesta área.

A Seção 4.1 introduz o problema P2P e discute brevemente as abordagens usadas atualmente para resolver o problema na prática. A Seção 4.2 apresenta as principais técnicas de aceleração do algoritmo de Dijkstra usadas na resolução do problema P2P: algoritmo bidirecional, algoritmo A* e métodos de pré-processamento. Esta última é uma das mais

[1]Os mapas considerados podem ser rodoviários, ou ferroviários, ou ainda de uma rede elétrica, telefônica ou de saneamento, ou outros variando em diferentes escalas e contextos.

estudadas e na literatura pode-se encontrar vários desses métodos. Os métodos estudados foram o de *landmarks* e o de *reach*. Por fim, a Seção 4.3 descreve brevemente alguns dos trabalhos mais recentes sobre resolução de problemas P2P. Estes não foram estudados mais profundamente devido às restrições de tempo.

4.1 O problema P2P

O problema P2P é definido como: dado um grafo orientado G com arestas ponderadas não negativas, um vértice fonte s (*source*) e um vértice destino t (*target*), encontre um caminho mínimo de s para t.

Naturalmente, o problema P2P pode ser resolvido pelo algoritmo de Dijkstra com uma simples modificação: assim que o vértice destino for alcançado, o algoritmo pára. Isto não muda a análise de complexidade (de pior caso) dos algoritmos vistos no capítulo anterior.

Este problema aparece em inúmeros contextos tais como: roteamento, projeto de circuitos integrados e protocolos de redes. Muitas dessas instâncias são muito grandes e na prática é necessário projetar técnicas para *acelerar* esses algoritmos. Tais técnicas, apresentadas a seguir, podem ser utilizadas de modo isolado ou combinadas, obtendo diferentes desempenhos na prática [Dai05, Dav97, FG03, GKW06b, HSWW05, KPE, KP06, SS05].

Há abordagens do problema P2P que não exigem que o caminho encontrado seja mínimo, mas apenas que seja uma solução *aproximada*, isto usando pré-processamento ou não. Vários trabalhos, como [GKW07], enumeram as principais referências para cada tipo de abordagem. Estas não são consideradas neste trabalho.

Os algoritmos aqui apresentados permitem o pré-processamento do grafo, incluindo o uso de informações geométricas [WW03], decomposição hierárquica [SS05] e distâncias de *landmarks* [GW05], por exemplo. Neste contexto, um algoritmo para o problema P2P consiste de duas fases: *pre-processamento*, que calcula dados auxiliares, e *busca* ou *consulta*, que determina uma solução [GKW07].

A fase de pré-processamento recebe um grafo orientado $G = (V, E)$ com pesos não negativos nas arestas e devolve alguma *estrutura auxiliar* que é usada na fase de busca. O que esta estrutura auxiliar armazena, depende do método utilizado. Por exemplo, ele poderia modificar o grafo ou obter informações auxiliares. As restrições impostas ao tempo de execução de pré-processamento dependem da aplicação em si. Se o grafo é estático (ou muda raramente), pode-se gastar mais tempo nessa fase. Um exemplo mencionado por Schulz, Wagner e Weihe [SWW00] é o seguinte. As estações de trem em cidades da Alemanha possuem seus próprios quadros de horário indicando partidas e chegadas de trens; o que deseja-se então, a qualquer momento, é percurso mais rápido de uma estação a uma outra (não necessariamente vizinha) a partir de um dado horário. Os

horários mudam apenas 2 vezes no ano, assim, basta executar um pré-processamento de várias horas a cada inverno/verão. Por outro lado, se o grafo não for estático, pode ser necessário fazer pré-processamentos mais freqüentemente. Por exemplo, ao longo do dia, o tempo de percurso entre dois pontos da cidade pode variar devido ao trânsito, chuvas etc. Neste caso, o tempo de execução de pré-processamento tem que ser rápido (em torno de duas horas).

A metodologia usada para avaliar os desempenhos de algoritmos para problemas P2P é essencialmente empírico. Isto porque de modo geral, não é fácil estimar teoricamente o desempenho de algoritmos que usam técnicas heurísticas [HSWW05]. A prática e estudos experimentais mostram que estas técnicas de fato melhoram drasticamente o desempenho para a maioria das instâncias do mundo real.

4.2 Técnicas de aceleração para algoritmos P2P

Os mapas rodoviários dos Estados Unidos e da Europa Ocidental são muito grandes. O tamanho do mapa rodoviário dos Estados Unidos, por exemplo, é da ordem de 20 milhões de vértices. Quando executada sobre esses mapas, a implementação ingênua gasta quase cinco segundos para calcular um caminho mínimo entre dois vértices em uma *workstation* de última geração [SS06]. Isto é considerado muito lento neste contexto, e portanto, a necessidade de técnicas de *aceleração* para algoritmos de caminhos mínimos.

Nesta seção são apresentados informalmente alguns conceitos relacionados a problemas P2P presentes na maioria das publicações estudadas. Estes foram selecionados devido à relevância das publicações que os introduziram. Particularmente, os contatos estabelecidos com Renato Werneck durante a realização deste trabalho, definiram a escolha deste conjunto de conceitos. Existem outros conceitos na literatura, mas informalmente, pode-se dizer que ou são difíceis de implementar ou não produzem resultados superiores aos aqui apresentados.

A grande maioria das técnicas descritas aqui são específicas ao contexto de roteamento ou sistemas de informação geográfica. Em princípio, elas não são aplicáveis em outros contextos. A lista apresentada aqui não é completa, mas bastante representativa.

As técnicas de aceleração são também denominadas variações do algoritmo de Dijkstra. A maioria delas requer o pré-processamento do grafo [HSWW05]. O uso dessas técnicas ou uma combinação dessas podem tornar a fase de consulta(s) mais rápida e/ou diminuir o consumo de memória. De modo geral, busca-se um compromisso (*trade-off*) entre ambos.

Um exemplo recente de tal compromisso é a utilização do algoritmo REAL (RE + ALT) que combina o algoritmo baseado em **RE**ach com o algoritmo **A*** que utiliza **L**andmarks e Desigualdade **T**riangular [GKW07]. Esta combinação demonstrou que, mantendo os *landmarks* apenas para vértices de longo alcance (*high-reach*) é possível

reduzir os requisitos de memória executando a operação de *pruning* que poda os vértices com curto alcance (*low-reach*). Assim, é possível economizar espaço para armazenar mais *landmarks* que melhoram o desempenho dos algoritmos. Com isso os consumos de espaço e de tempo são ambos melhorados. Outro exemplo de combinação de diferentes técnicas em busca do melhor compromisso entre os consumos de tempo e espaço é apresentado em [HSWW05]. Neste trabalho, foram combinadas técnicas que modificam o espaço de busca do algoritmo e verificou-se que a combinação de busca bidirecional com *multilevel approach* (que utiliza a decomposição hierárquica) e os *containers* (ambos requerendo pré-processamentos) apresentou os melhores resultados para grafos do mundo real. Outras combinações obtiveram melhores resultados em diferentes contextos. Maiores detalhes das técnicas mencionadas podem ser encontradas na seções subseqüentes.

Os algoritmos acima citados usam diferentes métodos de pré-processamento que serão analisados na Subseção 4.2.4. A seguir, são apresentadas as três principais técnicas de aceleração: algoritmo de Dijkstra bidirecional, algoritmo A* e métodos de pré-processamento.

4.2.1 Algoritmo de Dijkstra bidirecional

De acordo com Goldberg, Kaplan e Werneck [GKW07], o algoritmo de Dijkstra bidirecional, ou simplesmente algoritmo bidirecional, foi proposto independentemente na década de 60 por Dantzig (1962), Nicholson (1966) e Dreyfus(1967). Esta versão é específica para o problema P2P.

Segundo Holzer, Schulz, Wagner e Willhalm [HSWW05], em 1969 Pohl [Poh71] mostrou que o espaço de busca pode ser reduzido por um fator de 2 em relação ao algoritmo de Dijkstra. Mais precisamente, em algumas instâncias, o algoritmo visita no máximo metade dos vértices para encontrar uma solução.

O algoritmo alterna entre as execuções das buscas *direta* e *reversa* do algoritmo de Dijkstra, cada uma mantendo seu próprio conjunto de estimativas. A busca direta corresponde à execução do algoritmo de Dijkstra em G começando em s, enquanto a busca reversa corresponde à execução no grafo obtido de G invertendo as orientações de todas as arestas (agora começando em t). Intuitivamente, a versão bidirecional expande dois círculos com centros s e t. A idéia é que o algoritmo pára quando estes círculos se encontram.

Não foi encontrada nenhuma referência com uma descrição formal dessa versão bidirecional. A descrição a seguir é baseada em [GKW06a]. Seja $d_f(v)$ a estimativa de distância de um vértice v mantido pela busca direta (*forward*) e seja $d_r(v)$ a estimativa de distância de v mantida pela busca inversa (*reverse*). Ou seja, $d_f(v)$ representa a distância de s a v em G e $d_r(v)$ representa a distância de v a t em G. Na inicialização, a busca direta escaneia s e a busca reversa escaneia t. O algoritmo mantém ainda o peso μ do menor

caminho de s a t encontrado até então e o menor caminho correspondente. Inicialmente, $\mu = \infty$. Quando uma aresta (u, v) é relaxada pela busca direta e v já foi escaneado pela busca reversa, são conhecidos caminhos mínimos de s a u e de v a t com pesos $d_f(u)$ e $d_r(v)$, respectivamente. Se $\mu > d_f(u) + w(u, v) + d_r(v)$ então foi encontrado um caminho menor que o até então encontrado, e atualiza-se μ e o menor caminho. Atualizações similares são feitas durante a busca reversa. Há uma certa flexibilidade na escolha da ordem em que as duas buscas são executadas. Em [GKW07], a implementação foi feita de modo a alternar-se entre as buscas, escaneando um vértice na busca direta e depois escaneando um vértice na busca reversa.

Quando a busca em uma direção selecionar um vértice já escaneado pela busca na outra direção, o algoritmo pára. Um critério de parada mais sofisticado é o seguinte. Denote por $\min_f$ e $\min_r$ o valor da menor chave das filas de prioridade nas buscas direta e reversa, respectivamente. O algoritmo pára quando $\min_f + \min_r \geq \mu$. Para ver porque este critério está correto, suponha por contradição que no instante em que isso ocorre, o peso de um caminho mínimo P de s a t é tal que $w(P) < \mu$. Então existe alguma aresta (u, v) de P tal que $\text{dist}(s, u) < \min_f$ e $\text{dist}(v, t) < \min_r$. Isto implica que u está na fila de prioridades da busca direta e v está na fila de prioridades da busca reversa. Sem perda de generalidade, suponha que u foi inserido primeiro na sua fila. Então quando o algoritmo escaneou v na busca reversa, a aresta (u, v) foi relaxada e neste momento $d_f(u) = \text{dist}(s, u)$ e $d_r(v) = \text{dist}(v, t)$. Logo, o algoritmo necessariamente encontra um caminho de peso $w(P)$, ou seja, de peso mínimo. A Figura 4.1 mostra a execução do algoritmo bidirecional para uma instância.

Todos os grupos de pesquisadores nos trabalhos estudados usam esta estratégia bidirecional em seus conjuntos de testes, combinando-a com outras técnicas. Todos apresentam vantagem, tanto no tempo de execução quanto no consumo de memória, sobre a versão ingênua do algoritmo de Dijkstra. Atualmente, a busca bidirecional é considerada uma técnica clássica de aceleração do algoritmo. Sua grande vantagem é a de não pressupor nenhuma informação adicional (como informações geométricas) e não utilizar pré-processamento [SS05]. Exemplos de trabalhos que trazem implementações desta abordagem bidirectional são [SWW00, HSWW05, GW05, GKW06a, GKW06b, KP06, GKW07, SS06].

4.2.2 Algoritmo A^*

O algoritmo A* (A-estrela, do inglês *A-star*) é uma técnica herdada da inteligência artificial que integra uma heurística (baseada em uma "função potencial") ao procedimento de busca. Este algoritmo, também chamado de *busca* A* (ou busca heurística) foi proposto por Hart, Nilsson e Raphael [HNR68] para o problema P2P.

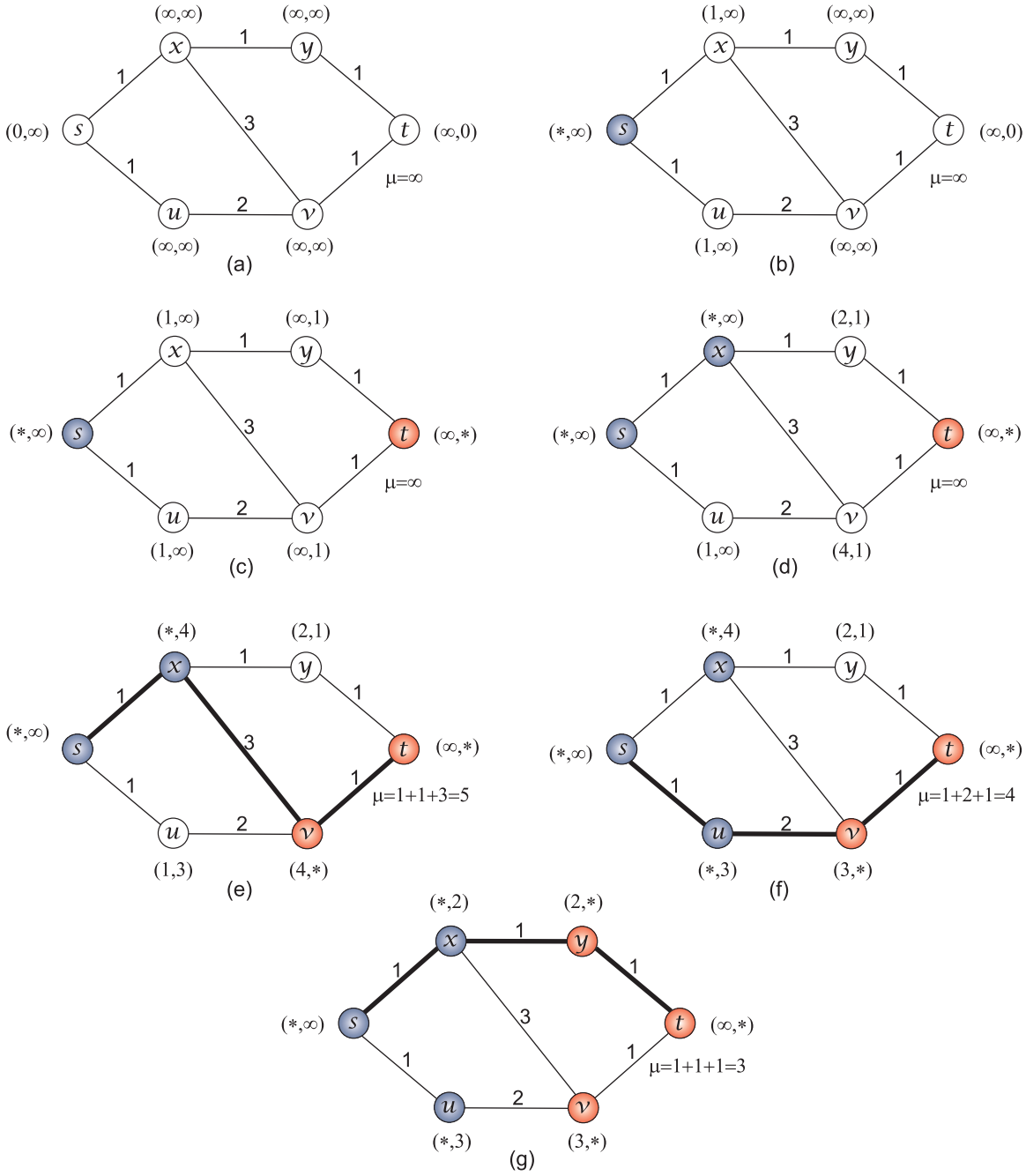

Figura 4.1: Execução do algoritmo de Dijkstra bidirecional. Seja Q_f e Q_r as filas das buscas direta e reversa. As chaves de um vértice i estão indicados como $(d_f(i), d_r(i))$. (a) s é colocado em Q_f e t é colocado em Q_r. (b) s é extraído de Q_f e x, u são colocados em Q_f. (c) t é extraído de Q_r e y, v são colocados em Q_r. (d) x é extraído de Q_f e y, v são colocados em Q_f. (e) v é extraído de Q_r e x, u são colocados em Q_r. O caminho (s, x, v, t) é encontrado e μ é atualizado. O algoritmo não pára pois $1 + 1 = \min Q_f + \min Q_r < \mu = 5$. (f) u é extraído de Q_f e v é colocado em Q_f. O caminho (s, u, v, t) é encontrado e μ é atualizado. O algoritmo não pára pois $2 + 1 = \min Q_f + \min Q_r < \mu = 4$. (g) y é extraído de Q_r e x é colocado em Q_r. O caminho (s, x, y, t) é encontrado e μ é atualizado. O algoritmo pára agora pois $2 + 2 = \min Q_f + \min Q_r > \mu = 3$.

Uma *função potencial* é uma função que associa a cada vértice v de G um valor $\pi(v)$. Dada uma função potencial π, o *custo reduzido* de uma aresta (u, v) é definido como $w_\pi(u, v) = w(u, v) - \pi(u) + \pi(v)$. É fácil ver que para um caminho P de x a y, vale que $w_\pi(P) = w(P) - \pi(x) + \pi(y)$. Assim, o problema de encontrar um caminho mínimo de s a t em G com função peso w é equivalente ao problema de encontrar um caminho mínimo de s a t em G com função peso w_π.

Dizemos que π é *viável* se w_π é não-negativa. É um fato bem conhecido que se π é uma função potencial viável e $\pi(t) \leq 0$ então para todo vértice v, $\pi(v)$ é um limite inferior para a distância de v a t. De fato, se P é um caminho mínimo de v a t então

$$0 \leq w_\pi(P) = w(P) - \pi(v) + \pi(t) \leq w(P) - \pi(v),$$

e portanto $\pi(v) \leq w(P)$. Note que neste caso, não há nenhuma perda de generalidade em supor $\pi(t) = 0$ pois é suficiente definir $\pi'(v) := \pi(v) - \pi(t)$ para cada vértice v. Então para todo vértice v, tem-se que $\pi'(v) \geq \pi(v)$ e $\pi'(v)$ também é um limite inferior para a distância de v a t.

No contexto de problemas P2P, um algoritmo A* usa uma função potencial viável π tal que $\pi(t) \leq 0$. Ele funciona de modo similar ao algoritmo de Dijkstra, exceto que a fila de prioridades usa como chave de um vértice v o valor $d[v]+\pi(v)$. A relaxação das arestas considera apenas as estimativas $d[v]$, como no algoritmo de Dijkstra, mas as operações que manipulam a fila usam a nova chave. Note que se $\pi = 0$, então ele corresponde exatamente ao algoritmo de Dijkstra. Não é difícil ver que executar o algoritmo A* corresponde a executar o algoritmo de Dijkstra no grafo G com função peso w_π (em vez de w).

A idéia por trás do algoritmo A* é que vértices que estão longe de t (ou seja, $\pi(v)$ é grande) são colocados no final da fila e assim, o algoritmo tende a alcançar t mais rapidamente. É um fato conhecido que se π é viável, então o algoritmo A* determina os caminhos mínimos corretamente [Poh71]. Goldberg e Werneck [GW05] mostraram que usando bons limites inferiores, menos vértices são escaneados no algoritmo A*. O exemplo da Figura 4.2 mostra como o uso do algoritmo A* reduz drasticamente o espaço de busca.

Isto naturalmente leva à questão de como obter bons limites inferiores. Em aplicações de mapas rodoviários ou naquelas em que há informação geométrica, uma possibilidade é usar distâncias euclideanas como limites inferiores. Na próxima seção será visto um método geral para obter limites inferiores em qualquer tipo de contexto: o método de *landmarks*.

O algoritmo A* também aparece na literatura com outros nomes. Holzer, Schulz, Wagner e Willhalm o denominam de *goal-directed search* [HSWW05]. Clodoveu [Dav97] o referencia como *heurística Hart-Nillson-Raphael*. Neste último artigo são comparados os tempos de processamento de 3 mecanismos de busca de um caminho mínimo sobre a

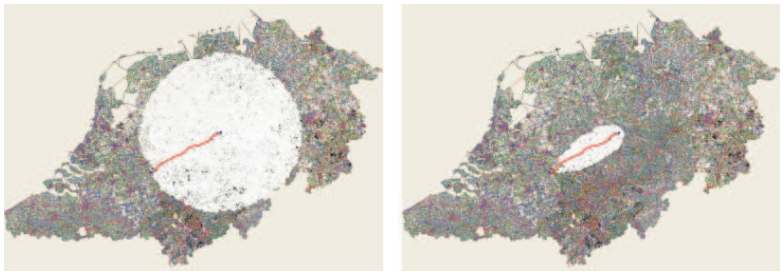

Figura 4.2: Exemplo retirado do artigo de Klunder e Post [KP06]. Enquanto a área visitada pelo algoritmo de Dijkstra (à esquerda) é um círculo centrado em s, a área do algoritmo A* (à direita) é um elipsóide bem menor. A instância corresponde ao mapa de estradas dos Países Baixos com cerca de 3 milhões de vértices que representem cruzamentos de vias rodoviárias.

rede de trânsito de Belo Horizonte/MG. Ele faz uso de (1) mecanismo interno de busca embutido no SIG (Sistema de Informação Geográfica) APIC[2], (2) mecanismo externo utilizando o algoritmo de Dijkstra e (3) mecanismo externo utilizando o algoritmo A*. Constatou-se que o algoritmo A* reduziu em 5 vezes o número de objetos visitados e foi de 5 a 6 vezes mais rápido, com relação ao algoritmo clássico. Comparando com o tempo do software APIC, a aceleração obtida variou entre 200 e 400 vezes.

4.2.3 Combinando busca bidirecional com algoritmo A*

A idéia de combinar a busca bidirecional com o algoritmo A* foi introduzida por Pohl [Poh69, Poh71]. Nesta versão, são usadas duas funções potenciais (viáveis): uma função π_f para a busca direta e outra π_r para a busca reversa. Nesta última, cada aresta original (u, v) corresponde a uma aresta (v, u) e o seu custo reduzido na direção reversa é $w_{\pi_r}(v, u) = w(u, v) - \pi_r(v) + \pi_r(u)$. As funções potenciais π_f e π_r são ditas *consistentes* se, para toda aresta (u, v), vale que $w_{\pi_f}(u, v) = w_{\pi_r}(v, u)$ [GKW07]. No caso de funções potenciais consistentes, pode-se usar o mesmo critério de parada usado pelo algoritmo bidirecional [Poh69, Poh71]. Em caso contrário, não há garantia de que um caminho mínimo foi encontrado quando os conjuntos escaneados na busca direta e reversa se encontram [KP06]. Nesse caso, uma maneira de garantir otimalidade é continuar a execução até exaurir a busca em uma das direções.

Um método que tem sido bastante utilizado recentemente é usar uma *função potencial média* sugerida por Ikeda, Hsu, Imai, Nishimura, Shimoura, Hashimoto, Tenmoku e Mitoh [IHI+94]. A idéia é substituir as funções potenciais π_f e π_r por duas funções médias

[2]APIC - *Atlas Permanent Information Communal* é um software de origem francesa adotado como plataforma da PRODABEL (Empresa de Informática e Informação do Município de Belo Horizonte S.A.), desde 1991

definidas como $p_f(v) = (\pi_f(v) - \pi_r(v))/2$ e $p_r(v) = -p_f(v)$. É fácil ver que p_f e p_r são funções potenciais viáveis, consistentes e que satisfazem $p_f(t) \leq 0$ e $p_r(s) \leq 0$ (logo, são limites inferiores para a busca direta e reversa, respectivamente). A Figura 4.3 mostra um exemplo onde a vantagem da combinação da busca bidirecional com algoritmo A* sobre o algoritmo de Dijkstra bidirecional é bem clara.

Figura 4.3: Áreas visitadas pelo algoritmo de Dijkstra bidirecional, à esquerda, e pelo algoritmo A* bidirecional, à direita [KP06]. Enquanto a área visitada pelo primeiro consiste de dois círculos justapostos centrados em s e t, a área do segundo consiste de duas pequenas elipses justapostas. O mapa é o mesmo da Figura 4.2.

O algoritmo A* bidirecional é amplamente empregado nos problemas P2P. Em problemas envolvendo mapas rodoviários ou com informação geométrica, sua utilização aumenta consideravelmente o desempenho dos algoritmos [SS05]. Isto o torna uma técnica de aceleração quase obrigatória nas aplicações dessa natureza [SS05, GKW06b]. O algoritmo parece ter um bom desempenho em outros contextos. Shulz, Wagner e Weihe [SWW00] relatam que sua utilização resulta em uma aceleração de um fator $1,5$ para grafos com métricas de tempos (em vez de distâncias).

4.2.4 Métodos de pré-processamento

Lembre que em aplicações envolvendo problemas P2P, há duas fases distintas: pré-processamento e busca/consulta. A segunda é executada mais freqüentemente que a primeira. Em muitas aplicações, é desejável que cada consulta seja extremamente rápida. Basta lembrar as instâncias dos mapas rodoviários dos Estados Unidos com mais de 20 milhões de vértices.

A idéia de usar pré-processamento é obter informações adicionais que permitam acelerar a fase de consulta. Os comportamentos dos algoritmos de pré-processamento conhecidos são bastante variados. Alguns procuram reduzir a entrada de dados para a fase de consulta. Outros aumentam consideravelmente o consumo de espaço para armazenar informações adicionais, enquanto outros demandam horas ou mesmo dias para economizar tempo e espaço para a fase de consulta. Em muitas aplicações, essas estratégias são

combinadas entre si, sempre que possível.

De acordo com Goldberg, Kaplan e Werneck [GKW07], os dois métodos descritos a seguir constituem um conjunto bastante representativo do que está sendo estudado, implementado e continuamente melhorado em pesquisas voltadas para o problema P2P.

Landmarks

Lembre que o algoritmo A* usa uma função potencial viável para obter limites inferiores para as distâncias. Por exemplo, em aplicações envolvendo mapas rodoviários, pode-se usar como função potencial as distâncias euclideanas. Entretanto, em muitas aplicações este tipo de informação não está disponível.

O método de pré-processamento por *landmarks* foi proposto em 2005 por Goldberg e Harrelson [GH05]. Seu uso permite obter rapidamente uma função potencial (π) para o algoritmo A*. Este método é de propósito geral, sendo aplicável em qualquer contexto. Esses autores demonstraram que o algoritmo A* com este tipo de pré-processamento possui desempenho superior à versão que usa distâncias euclideanas.

Este método se baseia na idéia de que os cálculos de caminhos mínimos freqüentemente são realizados sobre um conjunto estático de valores. Inicialmente, seleciona-se um pequeno conjunto de vértices denominados *landmarks*. Para cada vértice v de G calcula-se as distâncias de v até todos os *landmarks* e de cada *landmark* até v. Na prática, escolhe-se um conjunto de aproximadamente 16 *landmarks* (mesmo para instâncias com 20 milhões de vértices) [GW05].

Na fase de busca/consulta, estes valores são utilizados para determinar os limites inferiores (potenciais). Dado um *landmark* L e dois vértices u e v, pela desigualdade triangular das distâncias, tem-se que $\text{dist}(u, L) - \text{dist}(v, L) \leq \text{dist}(u, v)$ e $\text{dist}(L, v) - \text{dist}(L, u) \leq \text{dist}(u, v)$. Assim, para um *landmark* fixo L vale que

$$\max\{\text{dist}(u, L) - \text{dist}(v, L), \text{dist}(L, v) - \text{dist}(L, u)\} \leq \text{dist}(u, v).$$

Um modo de obter um limite inferior mais justo é simplesmente tomar o máximo considerando todos os *landmarks*. A Figura 4.4 exemplifica a utilização dos landmarks segundo esta abordagem.

Usando esta idéia pode-se obter duas funções potenciais viáveis $\pi_f(v)$ e $\pi_r(v)$ limitando inferiormente, respectivamente, a distância de v a t e a distância de v a s. Em geral, essas funções não são consistentes, mas pode-se aplicar a função potencial média de Ikeda e outros [IHI+94] vista na seção anterior. Na prática, para obter um limite inferior usa-se um subconjunto dos *landmarks*, chamados de *landmarks ativos*. O resultado dessa combinação de idéias é uma classe de algoritmos denominada ALT, assim chamada por ser baseada em algoritmo A*, *landmarks* e desigualdade triangular. Goldberg e Harrelson [GH05] relatam resultados de um estudo experimental onde o uso dos *landmarks* mostrou-se mais

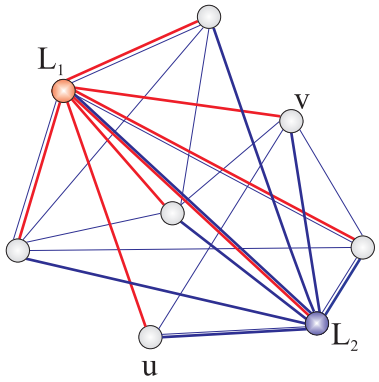

Figura 4.4: Exemplo de utilização de *landmarks*. Pela desigualdade triangular tem-se que $dist(u, L_1) - \text{dist}(v, L_1) < \text{dist}(u, v)$.

rápido do que outros métodos existentes até então, como o *reach* [Gut04] apresentado na próxima parte.

Encontrar bons *landmarks* é crucial para um bom desempenho do algoritmo A*. Há várias estratégias para selecionar os *landmarks*. Os autores [GH05] propõem alguns métodos para encontrar um conjunto de k *landmarks*. O primeiro consiste em escolher aleatoriamente k *landmarks*. Há outros métodos melhores entretanto. O método *farthest* escolhe de modo guloso os *landmarks*. Ele escolhe inicialmente um vértice qualquer e encontra um vértice v_1 mais distante desse. Ele acrescenta v_1 ao conjunto de *landmarks* e repete esse procedimento, escolhendo o vértice mais distante do conjunto de vértices atual, até obter um conjunto de tamanho k. O método *planar* usa informações de layout do grafo e supõe que existe um desenho do grafo no plano tal que distâncias geométricas e distâncias no grafo são fortemente relacionadas (o grafo não precisa ser planar). Inicialmente, determina-se o vértice c mais próximo ao "centro" do grafo. Em seguida, divide-se o desenho em k "fatias-de-pizza" com centro em c, todas com aproximadamente o mesmo tamanho. Em cada fatia, escolhe-se um vértice mais distante de c.

Goldberg e Werneck [GW05] propõe modificações desses métodos e outros métodos novos. Um deles consiste em combinar o método *farthest* com a busca em largura em vez do algoritmo de Dijkstra. O método *planar* é implementado usando a *mediana* e não o centro do grafo além de ser utilizada busca em largura. Eles propõem outros métodos como: *avoid*, *optimization*, *maxcover* etc.

A quantidade de *landmarks* é limitada, na maior parte, pela quantidade de armazenamento secundário disponível [GW05]. Na prática, escolhe-se valores como 2, 4, 8, 16, 23 até 32. Em implementações envolvendo mapas rodoviários [GH05, GW05] usa-se 16 *landmarks*. No trabalho de Klunder e Post [KP06], usa-se 32 *landmarks*. Ao final da próxima seção, a Figura 4.6 apresenta alguns resultados práticos publicados em [GKW06a] que

comparam os consumos de tempo e de memória desta e da próxima técnica em diferentes combinações.

Alcance ou *Reach*

O conceito de *alcance* ou *reach*, foi introduzido por Gutman [Gut04] para diminuir o tempo de processamento gasto pelo algoritmo de Dijkstra. Com este método, é possível podar (*prune*) ou desconsiderar vértices baseando-se nos alcances desses.

Dados vértices s, t e v, o *alcance de v com relação ao par s, t* é $\min\{\text{dist}(s, v), \text{dist}(v, t\}$, se existe algum caminho mínimo de s a t que passa por v; caso contrário, o alcance é 0. O *alcance* de um vértice v, denotado por $r(v)$, é o máximo dos alcances de v com relação a todos os pares de vértices distintos de G que passam por v.

Informalmente, o alcance de um vértice é *longo* se ele está próximo do centro de algum caminho mínimo de peso grande (considerando todos os pares de vértices) e o alcance é *curto* em caso contrário. Em outras palavras, um vértice tem alcance longo se vários caminhos mínimos passam por ele. Em redes rodoviárias, vértices com longo alcance (*high-reach*) correspondem aos cruzamentos de "auto-estradas", enquanto os vértices de curto alcance (*low-reach*) correspondem a cruzamentos locais. A Figura 4.5 exemplifica a classificação dos alcances (*reaches*) dos vértices.

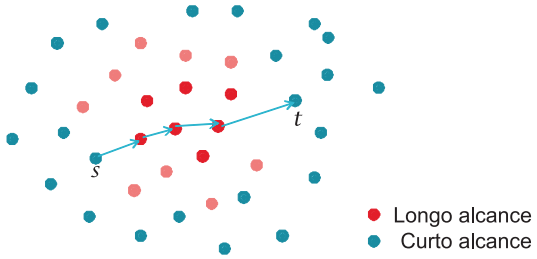

Figura 4.5: Exemplo de classificação de vértices segundo os valores de *reach*.

Uma maneira de calcular os alcances é apresentada por Goldberg, Kaplan e Werneck [GKW06b]. Inicialize $r(v) = 0$ para todo vértice v, e determine para cada vértice x uma árvore de caminhos mínimos T_x enraizada em x. Para cada vértice v em T_x, seu alcance local $r_x(v)$ é dado pelo mínimo entre a distância da raiz até v e a distância de v até a folha mais distante. Se $r_x(v) > r(v)$, atualize $r(v)$. Este método gasta tempo $O(nm)$ tempo, o que é impraticável para grafos grandes. Por exemplo, o cálculo dos alcances em um grafo de 30 milhões de vértices (como da América do Norte), levaria anos com as workstations existentes [GKW06b]. Soluções mais eficientes calculam apenas limites superiores dos alcances de cada vértice, o que é o suficiente para os propósitos dos problemas P2P [GKW06b, GKW07] como visto acima. Os métodos propostos [Gut04, GKW06b] são bastante complicados e não serão apresentados aqui.

No contexto do problema P2P, o alcance pode ser usado da seguinte forma. Suponha que para um vértice v tem-se que $r(v) < \text{dist}(s, v)$ e $r(v) < \text{dist}(v, t)$. Então v não pertence a nenhum caminho mínimo de s a t. Logo, o algoritmo de Dijkstra não precisa escanear v para encontrar uma solução. O mesmo raciocínio pode ser usado se em vez de $r(v)$, usar-se um limite superior $\overline{r}(v)$ para $r(v)$ e, em vez de $\text{dist}(v, t)$, usar-se um limite inferior $\pi(v)$ da distância de v a t. A seguinte regra de poda pode ser incorporada ao algoritmo de Dijkstra:

> Se $\overline{r}(v) < d[v]$ e $\overline{r}(v) < \pi(v)$, então v não é inserido na fila de prioridades.

Este teste pode ser incorporado à subrotina `Relaxa` para evitar que v seja inserido na fila de prioridades. Gutman [Gut04] mostrou que usando esta técnica, o algoritmo de Dijkstra com alcance tem um desempenho 10 vezes melhor que o mesmo sem *reach* para instâncias de 400.000 vértices geradas aleatoriamente. O limite inferior usado foi a distância euclideana.

Goldberg, Kaplan e Werneck [GKW06b] propõe uma variante do algoritmo de Dijkstra bidirecional com poda baseada em alcance. Uma vantagem dessa versão é que não é necessário conhecer de antemão limites inferiores para as distâncias. Na busca bidirecional, há a busca direta e a busca reversa. Para poder utilizar o teste de poda, é necessário conhecer limites inferiores de $\text{dist}(v, t)$ e $\text{dist}(s, v)$. O novo algoritmo usa os próprios limites inferiores implícitos na própria busca para fazer a poda. Considere a busca direta e seja γ a menor estimativa da fila de prioridades da busca reversa. Se um vértice v ainda não foi escaneado na busca reversa, então γ é um limite inferior para a distância de v a t. Uma observação similar se aplica à busca reversa. Quando o vértice v está para ser escaneado na busca direta, sabe-se que $d_f[v] = \text{dist}(s, v)$. Então v pode ser podado se as seguintes condições valem: (1) v não foi escaneado na busca reversa, (2) $\overline{r}(v) < d_f[v]$ e (3) $\overline{r}(v) < \gamma$. Usando este critério de poda, pode-se usar o mesmo critério de parada do algoritmo bidirecional tradicional. Os autores também discutem duas variantes dessa idéia no artigo.

Goldberg, Kaplan e Werneck [GKW06a] propõe uma combinação de alcance, algoritmo A* e *landmarks*, chamada REAL. Em vez de calcular a distância de um *landmark* a todos os outros vértices, calcula-se apenas para os vértices de longo alcance. Usa-se então esta informação para calcular limites inferiores usadas no algoritmo A*. Além disso, a regra de poda é incorporada na fase de consulta. Os autores mencionam uma implementação usada sobre mapas rodoviários dos Estados Unidos e da Europa, cada um com aproximadamente 20 milhões de vértices, que encontra um caminho mínimo em um milisegundo usando 2 horas de pré-processamento. Este é um limite considerado bastante razoável no contexto de problema P2P.

Em [GKW06a] foram realizados testes com consultas aleatórias sobre os mapas rodoviários dos Estados Unidos e da Europa, ambos no formato DIMACS. O grafo dos Estados Unidos é simétrico e possui 23.947.347 vértices e 58.333.444 arestas; o grafo da Europa é orientado, com 18.010.173 vértices e 42.560.279 arestas. Ambas as métricas de distância e de tempo de percurso foram utilizadas.

A Figura 4.6 apresenta o tempo médio das consultas (em milisegundos), o número médio de vértices escaneados e, quando possível, o número máximo de vértices escaneados. Também são mostrados o tempo total de pré-processamento e o espaço total em disco usado pelos dados de pré-processamento. Foram executados os algoritmos ALT, RE e REAL-(i, j). Este último utiliza i *landmarks* mas mantém as distâncias apenas para os n/j vértices de maior alcance (maiores valores de *reach*). A implementação ingênua também foi incluída para a calibração dos testes.

Grafo	Método	Tempo Pré (min)	Espaço Pré (MB)	Consulta		
				Avg. Sc	Max. Sc.	Tempo (ms)
Europa (tempo de percurso)	Dij	-	393	8.984.289	-	4.365,81
	ALT	13,4	1.631	82.348	993.015	158,33
	RE	82,8	626	4.563	8.866	3,48
	REAL (16,1)	96,2	1.864	741	3.518	1,17
	REAL (64,16)	143,9	935	683	2.817	1,14
EUA (tempo de percurso)	Dij	-	536	11.808.864	-	5.440,49
	ALT	18,5	2.608	187.968	2.183.718	399,21
	RE	48,6	890	2.264	4.667	1,86
	REAL (16,1)	67,2	2.962	581	2.732	1,03
	REAL (64,16)	142,5	1408	543	2.509	1,08
Europa (distância)	Dij	-	393	8.991.995	-	2.934,24
	ALT	10,5	1.656	240.750	3.306.755	417,83
	RE	50,3	664	7.007	12.940	5,81
	REAL (16,1)	60,8	1.926	847	3.996	1,47
	REAL (64,16)	98,0	979	575	2.876	1,17
EUA (distância)	Dij	-	536	11.782.104	-	4.576,02
	ALT	16,2	2.463	276.195	2.910.133	533,53
	RE	73,8	928	6.866	13.589	6,06
	REAL (16,1)	90,0	2.854	872	5.563	1,83
	REAL (64,16)	153,2	1.410	633	3.312	1,52

Figura 4.6: Dados de consultas aleatórias realizadas sobre os grafos da Europa e dos Estados Unidos usando diferentes algoritmos [GKW06a].

Ainda neste artigo ([GKW06a]) são apresentados vários resultados com diferentes combinações de valores para *landmarks* e *reaches* pelo algoritmo REAL, bem como suas execuções sobre outros mapas dos Estados Unidos disponibilizados pelo DIMACS. Em [GKW06b] são mostrados resultados nesta mesma linha incluindo *reaches* exatos e adição de atalhos, dentre outras técnicas relevantes.

4.2.5 Outras técnicas de aceleração

Afora as técnicas recém-apresentadas, várias outras são definidas na literatura. Em geral, a utilização destas técnicas, isoladas ou combinadas, é determinada pelas necessidades e restrições do contexto. Algumas obtêm vantagens valendo-se de informações incorporadas ao grafo de entrada. Outras tornam as consultas muito rápidas consumindo uma grande quantidade de espaço, ou vice-versa. O padrão das consultas realizadas também exerce forte influência sobre o desempenho do algoritmo.

Algumas técnicas não explicadas neste trabalho incluem:

- Highway hierarchies - Esta é uma técnica de aceleração proposta por Sanders e Schultes [SS05] e usada principalmente no contexto de mapas rodoviários. Sua idéia se baseia na seguinte observação: em geral, um caminho mínimo passa por ruas ou vias locais apenas no início e no final, enquanto no meio ele passa por rodovias ou estradas importantes. Assim, o algoritmo executa apenas algum tipo de busca local de s para t e então passa para uma busca em uma rede de rodovias maiores, cujo grafo é bem menor do que o completo. Esta técnica define seus próprios algoritmos de pré-processamento e busca. A busca local de s-t é aquela que visita os H mais próximos vértices de s (ou t) onde H é um parâmetro de ajuste[3]. A determinação dos H mais próximos vértices de uma dada extremidade utiliza o próprio algoritmo de Dijkstra em seu cálculo.

- Transit Nodes - Conforme [GKW07] as mais rápidas consultas de caminhos mínimos foram alcançadas pelos algoritmos baseados em *transit node routing*. Este conceito foi introduzido por Bast, Funke e Matijevic [BFM06a] e combinado com o conceito de *highway hierarchy* de Sanders e Schultes [SS05, BFM$^+$06b]. Ele apresenta os melhores resultados conhecidos até o presente momento. A idéia na qual se baseia este método é que, quando se percorre um caminho mínimo a partir de um vértice fonte fixado s para qualquer ponto distante, o caminho deixa a *área local* do vértice fonte via um número muito limitado de *access nodes*. O conjunto de todos os access nodes, considerando todas as fontes possíveis são os *transit nodes* do grafo [SS06].

[3]Os valores de H utilizados nos testes dos autores foram 75, 125, 175 e 300.

- Outras técnicas também referenciadas na literatura são: redução do grafo, *partial trees* [GKW06b], *multilevel approach* e *shortest paths containers* [HSWW05], *separators* [SS05].

4.3 Trabalhos recentes

As publicações selecionadas para este trabalho são eminentemente práticas, no sentido que manipulam explicitamente programas e conjuntos de dados utilizados em problemas do mundo real. Vale ressaltar que juntamente com as publicações que relatam estes trabalhos experimentais, outras são produzidas abordando, principalmente, os aspectos teóricos que fundamentam suas práticas. Os autores destes trabalhos citados, em conjunto com outros pesquisadores, avançam sobremaneira o estado da arte deste tema.

A seguir, são resumidamente apresentados trabalhos que utilizam os conceitos apresentados. Suas pesquisas foram aqui classificadas como Implementações Orientadas a Eficiência de Memória ou a Eficiência de Tempo devido às restrições que cada um busca priorizar.

4.3.1 Implementação orientada a eficiência de memória

Esta implementação refere-se ao artigo "*Computing Point-to-Point Shortest Paths from External Memory*" com autoria de Andrew Goldberg e Renato Werneck [GW05]. Neste trabalho, os autores estudaram o Algoritmo ALT (***A*** ** Search,* ***L****andmarks e* ***T****riangle Inequality*) no contexto de P2P de mapas rodoviários, sugerindo melhorias tanto no algoritmo quanto no pré-processamento.

Foi desenvolvida uma implementação memória-eficiente que roda em um Pocket PC. Ela armazena o grafo em um cartão de memória flash e usa a memória RAM para armazenar apenas parte do grafo visitado pelo atual cálculo de caminho mínimo. Esta implementação funciona bem mesmo sobre grafos muito grandes, incluindo o mapa rodoviário da América do Norte com quase 30 milhões de vértices.

O estudo comparativo realizado consistiu basicamente de modificar as quantidade de landmarks utilizados e os métodos utilizados para a sua seleção.

Outros trabalhos nesta mesma direção conduzidos juntamente com Haim Kaplan incluem [GKW06a] e [GKW06b], que também estão presentes em [GKW07]. Outros resultados recentes foram publicados no *9th DIMACS Implementation Challenge - Shortest Paths [DGJ06]* organizado pelo *Center for Discrete Mathematics and Theoretical Computer Science*.

Desdobramentos deste trabalho levaram à definição da técnica de aceleração denominada "*pruning*" (ou "poda"). Este conceito, associado aos alcances dos vértices, reduz o

espaço de busca do algoritmo de Dijkstra. A idéia desta técnica é a de "podar" ou "cortar" certos vértices em uma busca, como os de curtos alcances, utilizando-se os valores das distâncias euclideanas ou limites inferiores baseados em *landmarks*.

4.3.2 Implementações orientadas a eficiência de tempo

Sistema ferroviário alemão

Esta implementação faz referência ao artigo *"Dijkstra's algorithm on-line: an empirical case study from public railroad transport"* com autoria de Frank Schulz, Dorothea Wagner e Karsten Weihe [SWW00]. O cenário de aplicação deste trabalho é a rede ferroviária da Alemanha. Neste contexto, a aplicação em questão deve oferecer um mecanismo para buscas on-line por caminhos mínimos entre duas estações a partir de um dado horário. Os vértices deste grafo manipulado corresponde aos horários de partida ou de chegada de um trem em uma estação.

Ao contrário do trabalho anterior, o consumo de espaço não é considerado uma restrição forte, mas sim o tempo de resposta de uma consulta. A aplicação disponibiliza consultas (nos terminais das estações e na Internet) de caminhos mínimos considerando todos os horários dos trens da Alemanha. O tempo de resposta médio é mais importante do que o tempo de resposta máximo, caracterizando restrições de aplicações de tempo real.

Foram aplicadas algumas técnicas já definidas, como a busca A* e o algoritmo bidirecional, e outras concebidas para este contexto específico. Isto se deve ao fato de que há uma terceira dimensão neste problema específico, já que uma consulta leva em consideração ainda o tempo, além da origem e do destino. São definidas neste contexto as técnicas de "seleção de ângulo", "seleção de estações" e "restrição de ângulos".

Outras pesquisas realizadas que envolveram estes pesquisadores e colegas incluem o artigo *"Combining Speed-up Techniques for Shortest-Path Computations"* [HSWW05], onde são analisadas quatro técnicas de aceleração para o algoritmo de Dijkstra, isoladamente e/ou combinadas. Tendo em vista que as técnicas citadas diminuem o espaço de busca do algoritmo, são verificadas as melhores combinações de tais técnicas que melhoram o espaço e/ou o custo dos cálculos em diferentes contextos.

Sistema de táxis dos Países Baixos

Um extenso trabalho prático que implementa variações dos conceitos e técnicas citados é *The Shortest Path Problem on Large-Scale Real-Road Networks*, com autoria de G.A. Klunder e H.N. Post [KP06]. O artigo relata um trabalho comparativo realizado para encontrar o algoritmo mais rápido para o problema de P2P em uma grande instância de

mapa rodoviário. Os autores consideraram 6 algoritmos pré-existentes e um de correção de rótulos então desenvolvido (como no algoritmo de Bellman-Ford [CLR99]) , em diferentes variações e executados sobre o mapa rodoviário dos Países Baixos. O cenário em questão é um sistema de escalonamento de táxis (RBS) englobando toda a região dos Países Baixos. O objetivo é encontrar o veículo mais próximo do local do cliente e a menor rota até o destino desejado. No total foram implementadas 168 versões diferentes de algoritmos para caminhos mínimos das quais 18 são variações de um novo algoritmo então proposto pelos autores, que se mostrou o mais rápido de todos. Este algoritmo combina conceitos de buckets e técnicas de correção de rótulos.

Capítulo 5

Considerações finais

Estudar o problema de encontrar caminhos mínimos em grafos se revelou uma tarefa passível de diferentes abordagens. Foi possível perceber que os pontos de vista teórico e prático se relacionam e se complementam neste contexto. O algoritmo de Dijkstra, principal foco deste trabalho, é sem dúvida o mais aplicado em problemas de caminhos mínimos do mundo real. Além disso, as vantagens dessa relação entre teoria e prática podem ser claramente percebidas nas publicações de pesquisas experimentais que o implementam.

O ponto de partida deste trabalho foi a apresentação formal do algoritmo de Dijkstra (com a implementação ingênua), incluindo suas análises de complexidade e corretude e seu funcionamento. Isto levou à percepção de que a organização da fila de prioridades pode determinar sua análise de pior caso.

A seguir, foram descritas diferentes estruturas de dados usadas para representar a fila de prioridades do algoritmo de Dijkstra. Foram apresentadas as operações fundamentais das seguintes estruturas: heap binário, heap de Fibonacci, estrutura de bucket e radix heap. Foi possível formalmente comparar as complexidades isoladas de cada operação em cada estrutura, e combinadas no algoritmo de Dijkstra. Com isso, a intuição que o primeiro estudo desenvolveu foi concretizada nesta fase.

Como resultado deste estudo teórico, foram implementadas quatro variações do algoritmo de Dijkstra usando quatro estruturas de dados. As instâncias utilizadas foram nove mapas do mundo real disponibilizados pelo DIMACS. Percebeu-se que de fato uma estrutura como o heap de Fibonacci, que tem os melhores limites teóricos, apresentou desempenho inferior aos outros, como o bucket que tem limite "pseudo-polinomial". Entretanto, em dois de tais mapas o heap de Fibonacci obteve os (pouco) melhores tempos. Isto nos leva a concluir que não é possível afirmar que uma determinada estrutura de dados seja superior a outra em qualquer situação. Esta classificação é relativa e fortemente influenciada por vários fatores.

Os estudos teóricos no contexto dos problemas P2P foram ainda de suma relevância.

Eles demonstraram que as principais aplicações do mundo real possuem em geral duas fases: pré-processamento e busca. O algoritmo de Dijkstra é utilizado por ambas. Neste contexto algumas técnicas de aceleração têm sido propostas e incorporadas ao algoritmo original de Dijkstra oferecendo resultados práticos muito bons.

Ao final, esta combinação de estudos teóricos e práticos envolvendo problemas ponto-a-ponto no mundo real, permitiu alcançar uma maior compreensão das idéias e técnicas que apareceram e uma intuição das que ainda devem surgir nesta área de pesquisa. Com estes estudos foi possível "trilhar" o caminho de contribuições que o algoritmo de Dijkstra tem percorrido desde a sua publicação. O final deste caminho, é ainda, o foco de alguns trabalhos atualmente desenvolvidos e um forte indício de que outros tantos precisam ser pesquisados.

Assim, espera-se dar prosseguimento aos estudos do algoritmo de Dijkstra, notadamente sob o ponto de vista prático. Pretende-se testar os algoritmos desenvolvidos com outras instâncias de entrada. Pretende-se ainda implementar a estrutura de radix heap e os atuais buckets multiníveis para fins de comparação de seus desempenhos. Para o problema P2P planeja-se utilizar as estruturas já implementadas com as versões bidirecional e com o algoritmo A*. Estas implementações não foram realizadas neste trabalho devido às restrições de tempo.

Referências Bibliográficas

[AHU74] Alfred V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, 1990.

[BB96] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BFM06a] H. Bast, S. Funke, and D. Matijevic. TRANSIT: Ultrafast shortest-path queries with linear time preprocessing. In *9th DIMACS Implementation Challenge: Shortest Paths*, 2006.

[BFM+06b] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the 9th International Workshop on Algorithm Engineering and Experiments SIAM*. SIAM, 2006.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Aplications*. The Macmillian Press LTD, Great Britain, 1976.

[CGR94] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1994.

[CGS97] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms*, pages 83–92, 1997.

[CLR99] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press Publ, Cambridge Mass, 22 edition, 1999.

[Dai05] Liang Dai. Fast shortest path algorithm for road network and implementation. Honours Project, 2005.

[Dav97] Clodoveu Augusto Davis. Aumentando a eficiência da solução de problemas de caminho mínimo em SIG. In *GIS Brasil 97*, Curitiba - Paraná, 1997.

[DF79] E. V. Denardo and B. L. Fox. Shortest-route methods: 1. reaching, pruning, and bucket. *Operations Research*, 27:161–186, 1979.

[DGJ06] C. Demetrescu, A. V. Goldberg, and D. S Johnson. 9th DIMACS implementation challenge: Shortest paths, 2006. http://www.dis.uniroma1.it/ challenge9/.

[Dia69] Robert B. Dial. Algorithm 360: Shortest-path forest with topological ordering. *Commun. ACM*, 12(11):632–633, 1969.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dro02] Adam Drozdek. *Estrutura de Dados e Algoritmos em C++*. Pioneira Thomsom Learning, São Paulo, 2002.

[FG03] João Neiva de Figueiredo and Clovis C. Gonzaga. Aplicação de métodos de busca em grafos com nós parcialmente ordenados à locação de torres de transmissão. *Pesquisa Operacional*, 23(1):209–220, Janeiro a Abril 2003.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[FR62] L. R. Jr. Ford and Fulkerson D. R. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[GH05] Andrew Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*. SIAM, 2005.

[GKW06a] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better landmarks within reach. In *9th DIMACS Implementation Challenge: Shortest Paths*, 2006.

[GKW06b] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, Miami, Florida, 2006.

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest path algorithms with preprocessing. 2007. Submetido.

[GP88] G. Gallo and S. Pallotino. Shortest paths algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.

[GS95] Andrew V. Goldberg and Craig Silverstein. Implementations of Dijkstra's algorithm based on multi-level buckets. Technical Report 95-187, NEC Research Institute, Inc., 1995.

[Gut04] Ron Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, Vancouver, Canada, 2005.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[HSWW05] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. *Journal of Experimental Algorithmics*, 10:1–18, 2005.

[IHI+94] T. Ikeda, M.-Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. In *In Proc. Vehicle Navigation and Information Systems Conference*. IEEE, 1994.

[Iso02] Shigueo Isotani. Algoritmos para caminhos mínimos. Master's thesis, IME - USP, 2002.

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[Joh02] David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In D. S. Johnson M. Goldwasser and C. C. McGeoch, editors, *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, Providence, 2002.

[KP06] G.A. Klunder and H. N. Post. The shortest path problem on large-scale real-road networks. *Networks*, 48(4):182–194, 2006.

[KPE] S.D. Kirkby, S.E.P. Pollitt, and P.W Eklund. Implementing a shortest path algorithm in a 3D GIS environment.

[LO00] A. Lim and W. C. Oon. Dijkstra's algorithm for the shortest path problem using the bucket heap data structure. *International Journal of Computers and Their Applications (IJCA)*, 7(4), december 2000.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[MM99] Catherine C. McGeoch and Bernard M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[Poh69] Ira Sheldon Pohl. *Bi-directional and heuristic search in path problems*. PhD thesis, 1969.

[Poh71] Ira Sheldon Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[Pre00] Bruno R. Preiss. *Estruturas de Dados e Algoritmos*. Editora Campus, Rio de Janeiro, 2000.

[RS96] H. Rau and S. Skiena. Dialing for documents: an experiment in information theory. *Journal of Visual Languages and Computing*, pages 79–95, 1996.

[Sed02] Robert Sedgewick. *Algorithms in C (part 5: Graph Algorithms)*. Addison-Wesley/Longman, Princeton, NJ, 3 edition, 2002.

[SM94] Jayme L. Swarcfiter and Lílian Marckenzon. *Estruturas de Dados e seus Algoritmos*. Livros Técnicos e Científicos, Rio de Janeiro, 2 edition, 1994.

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings 17th European Symposium on Algorithms (ESA)*, volume 3669 of *Springer LNCS*, pages 568–579. Springer, 2005.

[SS06] Peter Sanders and Dominik Schultes. Robust, almost constant tome shortest-path queries in road networks. In *9th DIMACS Implementation Challenge: Shortest Paths*, 2006.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm online: An empirical case study from public railroad transport. *J. Experimental Algorithmics*, 5(12), 2000.

[Wag76] Robert A. Wagner. A shortest path algorithm for edge-sparse graphs. *J. ACM*, 23(1):50–57, 1976.

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proc. 11th European Symposium on Algorithms (ESA)*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.

[Ziv04] Nivio Ziviani. *Projeto de Algoritmos : Com Implementações em Pascal e C*. Pioneira Thomson Learning, São Paulo, 2 edition, 2004.