Universidade Federal Fluminense

RODRIGO FRANCO TOSO

Algoritmos para Atualização de Árvores Geradoras Mínimas em Grafos Dinâmicos

NITERÓI

2006

Universidade Federal Fluminense

RODRIGO FRANCO TOSO

Algoritmos para Atualização de Árvores Geradoras Mínimas em Grafos Dinâmicos

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Otimização Combinatória e Inteligência Artificial.

Orientador:
Prof. Celso da Cruz Carneiro Ribeiro, Dr. Hab.

NITERÓI

2006

Algoritmos para Atualização de Árvores Geradoras Mínimas em Grafos Dinâmicos

Rodrigo Franco Toso

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Otimização Combinatória e Inteligência Artificial.

Aprovada por:

Prof. Celso da Cruz Carneiro Ribeiro, Dr. Hab. / IC-UFF
(Orientador)

Profa. Luciana Salete Buriol, D.Sc. / II-UFRGS

Dr. Maurício Guilherme de Carvalho Resende, Ph.D. /
AT&T Labs. Research

Profa. Maria Cristina Silva Boeres, Ph.D. / IC-UFF

Profa. Simone de Lima Martins, D.Sc. / IC-UFF

Niterói, 7 de Agosto de 2006.

"The purpose of science and technology is to develop useful information for humanity to help people live their lives better. If we promise to withhold that information – if we keep it secret – then we are betraying the mission of our field. And this, I decided I shouldn't do."

"O propósito da ciência e tecnologia é prover conhecimentos úteis para a humanidade, de forma a ajudar as pessoas a viver melhor. Se prometemos reter essa informação – se a mantemos em segredo – então estamos traindo a missão da nossa área. E isso, eu decidi que não devo fazer."

Richard Stallman.

Aos meus pais, Ana Maria e Rubens, que dedicaram suas vidas para me dar oportunidades as quais nunca tiveram. Nada disso seria possível sem o amor e a dedicação de vocês.

Agradecimentos

Antes de mais nada, minha vida só é assim, especial como a considero, por que dela fazem parte pessoas especiais e únicas: meus amigos e familiares. Agradeço então aos amigos da Universidade Federal de Lavras (UFLA), da Universidade Federal Fluminense (UFF) e das famílias Franco e Toso. Em primeiro lugar, além de toda a turma 2000/2 da UFLA, fica um abraço aos membros do quarteto dinâmico, André Fialho, Carlos Eduardo (Sidnelson), Fabrício e Flávio. Seguindo a ordem, nunca poderia imaginar que encontraria na UFF um lugar tão caloroso para trabalhar e, claro, aprontar! Um agradecimento especial aos amigos Aletéia, Aline e Alexandre, Ary, Celso Ribeiro, Cristiane (Criix), Cristina e Vinod, Daniela (Dani), Diego, Haroldo, Idalmis (Ida, minha melhor amiga cubana), Jacques (Ratão), Johnny, Kennedy, Luciana (Brugiolo e Pessôa), Luciana e Robson, Luciano, Luis, Rafael (Guto e Satã), Renatha (Rena) e Marcel, Renato, Tiago (Facada e MacJovem), Stênio Sã, Viviane (Vivs) e Jonivan, Warley (Toca)... E finalmente, aos meus familiares: Ana Maria, Rubens e Ana Silvia (Byll) (pais e irmã); Zenith e José (avós); Zenólia (tia-avó). Tenho tantos tios e primos que daria uma página... Um agradecimento especial às tias e tios Vera (tanto a Franco quanto a Toso), Márcia e Eurico, e Edson, pelo carinho e amor! Com certeza não coloquei aqui o nome de todos os que merecem, mas minha memória (eu diria RAM) no final dessa dissertação já não é mais a mesma...

Gostaria de agradecer aqueles que estiveram diretamente ligados a esse projeto, ou seja, a todos professores do Instituto de Computação da Universidade Federal Fluminense. Tiveram participação fundamental nesse trabalho os professores Celso (projeto e análise de algoritmos), Christiano (teoria da computação), Cristina (estruturas de dados) e Satoru (paralelização de metaheurísticas, técnicas de inteligência computacional e otimização em redes). Merece destaque o professor Vinod pelo exemplo de ética e dedicação à UFF. Tem também os professores da UFLA que fizeram com que minha vida tomasse o rumo acadêmico, em especial: Renata (projeto e análise de algoritmos) e Ricardo (desenvolvimento para sistemas móveis, otimização, redes de computadores, sistemas operacionais e, por

fim (ufa!), orientador de projeto de conclusão de curso).

Os lugares onde morei durante essa jornada também precisam de citação especial. Primeiro aos parceiros da República Bufária (que depois virou A Marvada), em Lavras/MG: Fabrício, Flávio, Marcelo (Harry e Magal) e Renato (Quatí). Depois a Rép Our (Niterói/RJ), com alta rotatividade e, portanto, em ordem cronológica: Fialho, Luis (Paçocão e Suri), Rafael (Bátima), Emmanuel, Carlos Eduardo (Dudu), Rafael (Guto), Diego e Warley (Toca).

Ao CNPq pela bolsa que custeou em torno de 40% do meu mestrado.

Aos membros da banca examinadora pelos comentários e sugestões: Celso Ribeiro, Cristina Boeres, Luciana Buriol, Maurício Resende e Simone Martins.

Agradeço aos funcionários da UFF que fazem toda a base (muitas vezes chata e burocrática). Ângela, Izabela, João e Maria, obrigado por todos os galhos quebrados (e eu sei que não foram poucos – daria para desmatar uma floresta).

Sem dúvida esse trabalho não teria uma bibliografia impecável, texto bem escrito, foco bem definido e consistência (tudo bem, deixei a modéstia de lado) se não fosse por conta de um orientador que, embora geograficamente distante, esteve sempre "interneticamente" perto (sem contar os vários sábados e domingos de reunião). Um agradecimento mais que especial ao amigo, professor e orientador Celso.

O que á vida sem amor? Eu respondo que a vida simplesmente não teria sentido sem amor de pai, de mãe, de irmão e de amigo. Mas, além disso, a vida não teria graça alguma se não existisse o amor de beijos na boca, abraços apertados, confidências, cumplicidade e tudo mais... A vida não teria graça alguma se eu não tivesse encontrado a minha pequenininha Daniela. Sempre ao olhar ao céu existe uma estrela que chama mais atenção que as outras – as vezes por seu brilho, as vezes por sua magia ou as vezes sem qualquer explicação – e acho que você é essa estrela: repleta de magia, alegria, amor, sorrisos, companheirismo, e de um quê a mais que é indescritível! *Love you...*

Resumo

O Problema das Árvores Geradoras Mínimas Dinâmicas (PAGMD) tem como objetivo a manutenção de uma árvore geradora mínima de um grafo sujeito a constantes mudanças estruturais, onde tais mudanças podem ser inserções ou remoções de vértices, inserções ou remoções de arestas e modificações em custos de arestas. Este problema é dito *totalmente dinâmico* quando ambas as operações de inserção e remoção (ou de incremento e decremento em custos de arestas) são permitidas. Por outro lado, este problema é dito *parcialmente dinâmico* ou *semi-dinâmico* quando apenas um tipo de operação é permitido (inserções ou remoções, incrementos ou decrementos). Ainda, o problema é dito *on-line* quando as alterações dinâmicas são processadas em tempo real, ou seja, sem qualquer tipo de pré-processamento.

O estudo de algoritmos para grafos dinâmicos, em particular aqueles para a manutenção da árvore geradora mínima de um grafo em constante atualização, é motivado tanto por razões teóricas quanto por razões práticas. Algoritmos e estruturas de dados dinâmicas podem ser utilizados em uma vasta coleção de problemas cotidianos, a citar problemas de otimização em redes (redes de computadores, telefonia e TV a cabo), metaheurísticas e heurísticas de busca local.

Neste trabalho é realizada uma avaliação experimental dos algoritmos para atualização da árvore geradora mínima de um grafo sujeito a alterações dinâmicas nos custos de suas arestas. Tais algoritmos podem ser úteis na implementação de metaheurísticas e heurísticas de busca local para problemas de projeto e otimização de redes de comunicação, de maneira similar aos algoritmos envolvendo os problemas de caminho mínimo estudados por Buriol et al. [6, 7, 8, 9] no contexto do problema de atribuição de custos para o roteamento de pacotes em redes OSPF/IS-IS. Complementarmente, são propostos um algoritmo e uma estrutura de dados especificamente desenvolvidos para o caso de atualização em custos de arestas. O algoritmo proposto é de simples implementação computacional, podendo ser utilizado com qualquer estrutura de dados para representação de árvores dinâmicas.

Palavras-chave: Árvore geradora mínima, árvore geradora de custo mínimo, grafos dinâmicos, análise experimental de algoritmos, complexidade computacional.

Abstract

The Dynamic Minimum Spanning Tree Problem (DMSTP) is that of maintaining a minimum spanning tree (MST) of a dynamically changing graph, where these changes (or operations) can be insertions and deletions of vertices, insertions or deletions of edges, and modifications of edge weights. The problem is said to be *fully dynamic* if both insertion and deletion operations are allowed (or if the edge weights can increase or decrease). Otherwise, the problem is said to be *partially dynamic* or *semi dynamic* if only one kind of operation is allowed (either edge deletions or insertions, either weight increases or decreases). Also, the problem is said to be *on-line* if the dynamic changes must be processed in real time (i.e. there is no preprocessing and updates are performed one by one).

The study of dynamic graph algorithms, in particular those for maintaining a minimum spanning tree of a dynamically changing graph, is motivated by both practical and theoretical reasons. Dynamic algorithms and data structures can be used in a wide range of real-life problems, e.g. in network-related problems (computer, telephony and cable-TV networks), metaheuristics and local search heuristics.

In this work, we make a step toward the experimental evaluation of algorithms to update a minimum spanning tree after edge weight changes. Such algorithms are particularly helpful in the implementation of metaheuristics and local search heuristics for solving broadcast optimization and design problems in communication networks, similar to the algorithms involving dynamic shortest path problems studied by Buriol et al. [6, 7, 8, 9] in the context of the weight setting problem in OSPF/IS-IS routing. Complementary, we propose and evaluate both a new algorithm and a new data structure specifically designed for the edge weight updating variant of the DMSTP. The new algorithm is quite simple to implement and can be used with any data structure for dynamic trees representation.

Keywords:. Minimum spanning trees, minimum weight spanning trees, dynamic graphs, experimenal analysis of algorithms, computational complexity.

Abreviações

AGM	:	Árvore Geradora Mínima (página 1)
FGM	:	Floresta Geradora Mínima (página 34)
PCM	:	Problema do Caminho Mínimo (página 1)
RD-trees	:	Árvores dinâmicas reversas (página 26)
DRD-trees	:	Árvores dinâmicas duplamente encadeadas (página 52)
ST-trees	:	Árvores de Sleator e Tarjan [41] (página 26)
ET-trees	:	Árvores de ciclos eulerianos [26] (página 26)

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os problemas de otimização em redes são de grande interesse teórico e prático para a ciência da computação. O interesse em sua teoria, onde destacam-se as pesquisas por algoritmos e implementações eficientes, é complementado por suas inúmeras aplicações em situações cotidianas. Exemplos de problemas de otimização em redes incluem os problemas de caminho mínimo (PCM), árvore geradora mínima (AGM) e fluxo máximo.

Ainda que sejam de grande valia do ponto de vista prático, os algoritmos clássicos desenvolvidos para os problemas de otimização em redes não prevêem qualquer forma de alteração na rede propriamente dita. Entretanto, redes reais podem estar sujeitas a alterações freqüentes – inserções ou remoções de nós; inserções ou remoções de arestas; ou mudanças no custo de arestas – trazendo consigo a necessidade de atualização da solução anteriormente gerada. Uma rede é facilmente representada através de um grafo, uma estrutura de dados onde os algoritmos e implementações computacionais são extremamente eficientes. As características dinâmicas das redes fazem com que os algoritmos clássicos sejam, muitas vezes, obrigados a descartar totalmente a solução anterior para, então, reconstruir a nova solução a partir do grafo atualizado. Embora esses algoritmos tenham complexidade polinomial, recalcular uma propriedade de um grafo, além de computacionalmente caro, pode ser ineficiente por não aproveitar a solução gerada no passo anterior à alteração.

Assim, visando evitar o problema de, após cada atualização da rede, ignorar toda a solução existente e partir para a construção de uma nova solução, foram propostos *algoritmos dinâmicos* para os mais importantes problemas de otimização em redes. Dado um grafo representando a rede em questão, a solução para um problema sobre esse grafo e a alteração que ocorreu na estrutura do mesmo, esses algoritmos são capazes de atualizar a solução obsoleta, tornando-a válida para o grafo resultante da modificação estrutural.

A área de algoritmos para grafos dinâmicos avançou muito desde seu início. A partir de então, foram propostos algoritmos e estruturas de dados eficientes para os principais problemas de otimização em redes.

Além de aplicações em problemas reais, os algoritmos dinâmicos podem ser aplicados em sub-problemas de programação inteira e em métodos de busca local de heurísticas e metaheurísticas. Acredita-se que essas aplicações sejam de extrema importância para a área de otimização combinatória, e trazem consigo um requisito adicional para esses algoritmos: além da eficiência, a simplicidade para implementação computacional é fundamental. Nesse caminho, foram propostos trabalhos envolvendo algoritmos dinâmicos para o problema de caminho mínimo em grafos, com aplicações em métodos de busca local de heurísticas e metaheurísticas [6, 7, 8, 9].

Este trabalho têm seu foco em algoritmos e estruturas de dados dinâmicas para o problema da árvore geradora mínima, considerando o fato de não haver na literatura um trabalho como os apresentados em [6, 7, 8, 9] para o problema de caminho mínimo em grafos dinâmicos, bem como sua importância para a área de otimização. Árvores geradoras mínimas têm aplicações em problemas como os de transmissão e de conectividade em redes e, até mesmo, em problemas de biologia computacional. A variante dinâmica estudada foi a de alterações – incrementos e decrementos – nos custos das arestas.

Nesta dissertação, apresenta-se uma nova estrutura de dados simples e eficiente para o problema das árvores dinâmicas. Propõe-se e avalia-se experimentalmente um algoritmo extremamente simples, mas ainda assim eficiente, para a atualização da árvore geradora mínima de um grafo sujeito a alterações nos custos das arestas. O trabalho está organizado em sete capítulos. O conteúdo de cada capítulo é apresentado a seguir.

- Capítulo 1: apresenta este trabalho sob uma ótica geral e abrangente, situando seu contexto e delineando o escopo a ser desenvolvido.
- Capítulo 2: introduz o problema da árvore geradora de custo mínimo, trazendo algoritmos e estruturas de dados para a versão estática desse problema.
- Capítulo 3: apresenta o estado da arte para algoritmos e estruturas de dados dinâmicas para árvores geradoras mínimas.
- Capítulo 4: propõe e analisa uma estrutura de dados para a representação de árvores dinâmicas.

- Capítulo 5: propõe e discute um algoritmo para atualização da árvore geradora mínima de um grafo dinâmico.
- Capítulo 6: conduz uma análise experimental comparando o algoritmo e a estrutura de dados propostos com alguns dos mais importantes algoritmos da literatura.
- Capítulo 7: traz as conclusões e trabalhos futuros a esta dissertação.

Capítulo 2

Árvores Geradoras Mínimas: Algoritmos e Complexidade

Neste capítulo é introduzido o problema clássico da árvore geradora mínima (AGM), incluindo sua formulação matemática, aplicações, algoritmos e complexidades.

2.1 Formulação e Aplicações

Em problemas de conexão em redes, onde n pontos (nós) devem ser conectados através de $n-1$ arestas, geralmente existe o requisito de se obter o conjunto de arestas que minimize um certo custo a ser gasto. Um exemplo que ilustra esta afirmação é o problema de se conectar todos os n computadores de uma rede de forma a minimizar os gastos com fibras óticas. Nestes problemas, deseja-se encontrar uma árvore de custo mínimo que conecte todos os nós da rede.

Formalmente, dada uma rede modelada como um grafo $G = (V, E)$, onde V é um conjunto de n nós e E é um conjunto de m arestas que conectam dois nós de V, e uma função $w : E \to \mathbb{R}$ que especifica o custo da aresta $(u, v) \in E$, o problema da AGM consiste em encontrar a árvore $T = (V, E')$ com $E' \subseteq E$ que minimiza a função objetivo

$$w(T) = \sum_{(u,v)\in E'} w(u, v). \tag{2.1}$$

Árvores geradoras mínimas são comumente utilizadas em problemas como: *broadcast* em redes de computadores, planejamento e concepção de redes de transmissão (telefonia, energia elétrica e TV a cabo, por exemplo), rotas mínimas para a cobertura de cidades, geração de soluções heurísticas para problemas como os do caixeiro viajante [23, 24] e

problemas de biologia computacional, entre outros.

2.2 Algoritmos

Na literatura, existem várias maneiras de se determinar a árvore geradora mínima de um grafo, todas elas baseadas em *algoritmos gulosos*. Estes constituem uma técnica de projeto de algoritmos onde, a cada passo ou iteração, uma decisão é tomada com base na melhor das opções existentes no momento. Este trabalho apresenta os dois algoritmos mais conhecidos: o algoritmo de Kruskal [33], na Seção 2.2.1, e o algoritmo de Prim [37], na Seção 2.2.2.

2.2.1 Algoritmo de Kruskal

Este algoritmo foi proposto em 1956 por Kruskal [33]. Seu princípio de funcionamento é baseado na ordenação de todas as arestas do conjunto E. Constrói-se a árvore geradora mínima $T = (V, E')$, onde E' é inicialmente um conjunto vazio. Analisa-se cada aresta $(u, v) \in E$ em ordem crescente de custo. De acordo com essa ordem, se os nós $u \in V$ e $v \in V$ pertencem a componentes conexas distintas da arborescência corrente $T = (V, E')$, então a aresta (u, v) é adicionada a E' e, conseqüentemente, as componentes conexas onde se encontram os vértices u e v são unidas em uma única componente. Após serem inseridas $n - 1$ arestas, E' contém as arestas necessárias para conectar todos os nós de V com custo mínimo. A Figura 2.1 ilustra a execução do algoritmo de Kruskal em um grafo dado.

Na Figura 2.1, a árvore geradora mínima T é representada pelas arestas mais escuras. Seguindo a ordem crescente de custo, a aresta (b, f) é analisada e incluída no conjunto E', já que os nós b e f não fazem parte da mesma componente conexa (a). Da mesma forma, em (b) é inserida a aresta (c, e), seguida pelas inserções das arestas (f, g), (a, b) e (d, f). Já a aresta (a, d) não deve ser incluída em T, visto que a inclusão desta resultaria em um ciclo em T, descaracterizando a propriedade de que uma árvore não deve possuir ciclos (f). A aresta (b, c) é então inserida em T, já que as mesmas se encontram em componentes distintas (g). Por fim, em (h) e (i), as arestas (e, f) e (e, g), respectivamente, são descartadas por também causarem ciclos em T. É importante salientar que os passos (h) e (i) poderiam ser omitidos pelo algoritmo, visto que $n-1$ arestas já foram selecionadas para compor a árvore geradora mínima T.

O Algoritmo 1 formaliza os passos acima citados, com corretude demonstrada em [12].

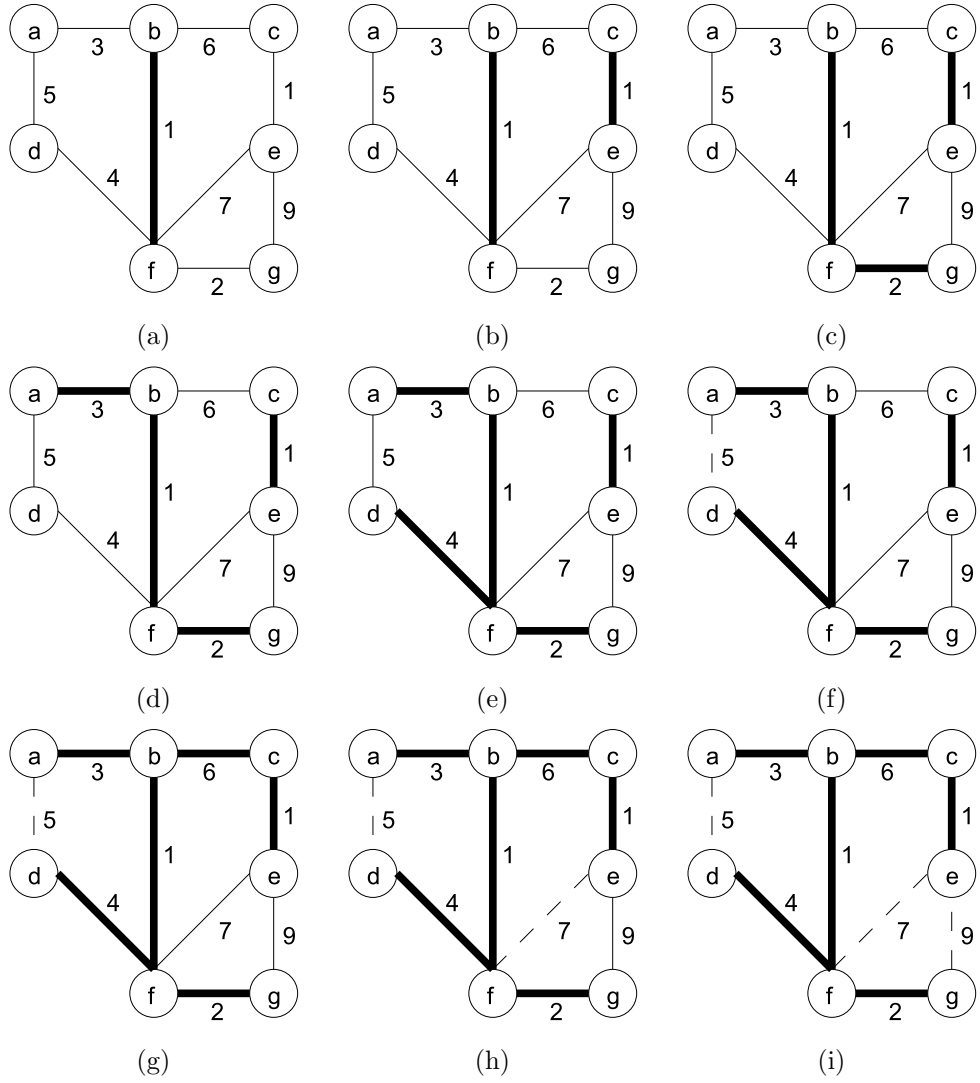


Figura 2.1: Execução do algoritmo de Kruskal.

Sua implementação requer uma estrutura de dados que mantenha atualizados os conjuntos disjuntos que formam as componentes conexas da arborescência corrente T. Suas operações são apresentadas abaixo:

- `MakeSet(`u`)`: cria um vértice u, pertencente ao conjunto disjunto contendo apenas u.
- `Union(`u, v`)`: une os conjuntos disjuntos contendo u e v através da aresta (u, v).
- `Find(`u`)`: retorna o conjunto disjunto ao qual u pertence.

A complexidade do algoritmo de Kruskal depende da maneira como a estrutura de dados para manutenção de componentes conexas é implementada. Se a estrutura *set-union* [46] é utilizada, então a execução de $m \geq n$ operações `Find` e $n-1$ operações `Union`

Algoritmo 1 Método de Kruskal

Entrada: Grafo $G = (V, E)$ e custos w.

1: Ordenar E em ordem crescente de custos;
2: $E' \leftarrow \emptyset$;
3: **para todo** $v \in V$ **faça**
4: `MakeSet(`v`);`
5: **fim para**
6: **para todo** $(u, v) \in E$ em ordem crescente de custos **faça**
7: **se** `Find(`u`)` $\neq$ `Find(`v`)` **então**
8: $E' \leftarrow E' \cup \{(u, v)\}$;
9: `Union(`u, v`);`
10: **fim se**
11: **fim para**

possui complexidade de $O(m\alpha(m, n))$. Se as n operações `MakeSet` realizadas na linha 2 são levadas em consideração, a complexidade relacionada à estrutura de dados *set-union* é $O((m + n) \cdot \alpha(m, n))$. Como $\alpha(m, n)$ é uma função que cresce muito lentamente, calculada a partir do inverso da função de Ackermann [46], a complexidade do algoritmo passa a depender do tempo em que a ordenação das arestas pode ser realizada, ou seja, $O(m \log m)$. Dado ainda que $m < n^2$, então $\log m = O(\log n)$, o que resulta na complexidade de $O(m \log n)$. Um estudo completo sobre estruturas de dados para armazenamento e atualização de componentes conexas é apresentado por Galil e Italiano [20].

2.2.2 Algoritmo de Prim

O algoritmo de Prim [37] também é um método guloso. Porém, usa apenas uma árvore (componente conexa), inicialmente vazia. A AGM é então construída da seguinte maneira: inicia-se uma estrutura de dados denominada *fila de prioridades* (*heap*) contendo todos os nós de V, que inicialmente terão prioridade $p = \infty$. Partindo de um nó raiz r escolhido arbitrariamente, as prioridades dos nós conectados à raiz são atualizadas de acordo com o custo entre r e o nó v analisado. Deste modo, aquele vértice v com a maior prioridade (ou seja, o menor custo para conexão à AGM) é adicionado a T, e as prioridades dos nós conectados a v também são atualizadas na fila de prioridades. Este passo guloso se repete até que todos os nós sejam retirados dessa fila, quando a AGM T conectará todos os nós $v \in V$. A Figura 2.2 ilustra uma execução típica do algoritmo de Prim.

Na Figura 2.2, o primeiro passo é selecionar o nó raiz a e atualizar seu custo para 0, fazendo com que este seja o primeiro nó a ser removido da fila de prioridades e também alterando os custos de acesso aos nós b e d para 3 e 5, respectivamente (a). O próximo

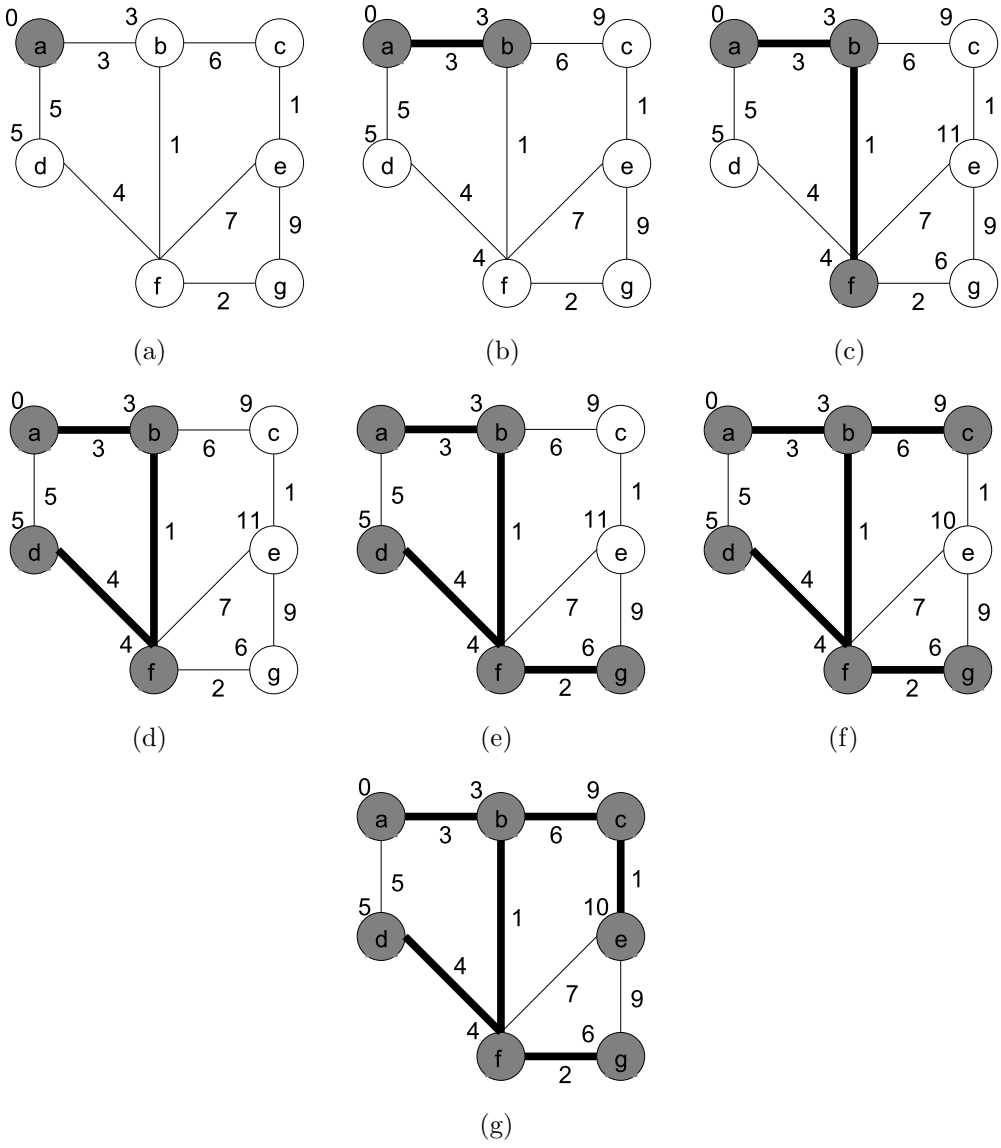


Figura 2.2: Execução do algoritmo de Prim.

passo é remover o nó b da fila de prioridades e atualizar as estimativas dos nós c e f para respectivamente 9 e 4 (b). As demais etapas apresentadas nessa figura completam a ilustração do método de Prim.

O método de Prim é implementado pelo Algoritmo 2 e sua prova de corretude é apresentada em Cormen et al. [12]. As operações típicas de uma fila de prioridades são mostradas abaixo:

- `Insert`(i, p): insere o ítem i na fila, com prioridade p.
- `RetrieveKey`(i): retorna a prioridade do ítem i.
- `DecreaseKey`(i, p): altera a prioridade do ítem i para o valor p. Esta operação

requer que a nova prioridade seja maior (i.e. de menor valor) que a prioridade atual.

- `ExtractMin()`: remove e retorna o ítem da fila com a maior prioridade (i.e. o menor valor para p).

Algoritmo 2 Método de Prim

Entrada: Grafo $G = (V, E)$ organizado em uma lista de adjacências, um nó raiz r e os custos w.
{H: fila de prioridades}
{π: armazena para cada vértice $v \in V$ o vértice $u \in V$ tal que $(u, v) \in E'$}

```
1: para todo v ∈ V faça
2:    π[v] ← NIL;
3:    Insert(v, ∞);
4: fim para
5: DecreaseKey(r, 0);
6: enquanto H ≠ ∅ faça
7:    u ← ExtractMin();
8:    para todo v adjacente a u faça
9:       se v ∈ H e RetrieveKey(u) + w(u, v) < RetrieveKey(v) então
10:         π[v] ← u;
11:         DecreaseKey(v, RetrieveKey(u) + w(u, v));
12:      fim se
13:   fim para
14: fim enquanto
```

A complexidade do Algoritmo 2 depende do quão eficiente pode ser implementada uma estrutura do tipo fila de prioridades. O Capítulo 6 de Cormen et al. [12] desenvolve e analisa a estrutura de dados *binary heap* (*heap* binário), capaz de executar cada operação acima em $O(\log n)$, excetuando-se a operação `RetrieveKey`, que pode ser facilmente implementada em $O(1)$. Usando um *heap* binário, o algoritmo acima possui complexidade $O(m \log n)$ [12].

Uma estrutura ainda mais eficiente é apresentada no Capítulo 20 de Cormen et al. [12]. Esta fila de prioridades é conhecida por *Fibonacci Heap* [19] e é capaz de executar a operação `DecreaseKey` em tempo $O(1)$ amortizado. Assim, a complexidade do Algoritmo 2 torna-se $O(m + n \log n)$ [12]. Embora de grande interesse teórico, os *heaps* de Fibonacci são estruturas muito complexas para serem implementadas de maneira eficiente – são desconhecidos resultados práticos superiores àqueles alcançados por *heaps* binários.

2.3 Resumo do Capítulo

Este capítulo apresentou o problema da árvore geradora mínima em grafos, bem como os algoritmos de Kruskal [33] e de Prim [37]. Também foram apresentadas as estruturas de dados para manutenção e atualização de uma fila de prioridades (ou *heap*) e de conjuntos disjuntos (ou *set-union*).

Capítulo 3

Árvores Geradoras Mínimas em Grafos Dinâmicos

Este capítulo aborda o problema da árvore geradora mínima em grafos dinâmicos. Para isso, na Seção 3.1 é apresentado o problema das árvores dinâmicas e as estruturas de dados propostas para resolvê-lo. Essas estruturas são úteis no armazenamento de árvores geradoras mínimas sujeitas a alterações dinâmicas. Na Seção 3.2 são analisadas todas as alterações que podem ocorrer em uma AGM quando o grafo é modificado. Por fim, é apresentada a revisão bibliográfica do problema de árvores geradoras mínimas em grafos dinâmicos.

3.1 Árvores Dinâmicas

O Problema das Árvores Dinâmicas, também conhecido como *Dynamic Trees Problem* ou *Linking and Cutting Trees Problem*, foi proposto por Sleator e Tarjan [41]. Esse problema introduz a necessidade de uma estrutura de dados para manter uma coleção de árvores disjuntas sujeitas a dois tipos básicos de operações: *link* e *cut*. A primeira operação combina duas árvores em uma só através da inserção de uma aresta, enquanto que a segunda divide uma árvore em duas através da remoção de uma aresta.

As seguintes operações foram propostas em [41] para dar suporte ao problema das árvores dinâmicas:

- `Link`(u, v, c): une, através da aresta (u, v), a árvore enraizada em u com a árvore que contém o vértice v. Esta operação resultará em uma única árvore onde o vértice u, que antes não tinha um nó pai, passa a ter v como pai. Essa aresta terá custo c.
- `Cut`(u, v): separa a árvore que contém os vértices u e v através da remoção da

aresta (u, v).

- `Root(`v`)`: retorna a raiz da árvore à qual o vértice v pertence.
- `Cost(`u, v`)`: retorna o custo da aresta (u, v).
- `Find_min(`v`)` (resp. `Find_max(`v`)`): retorna a aresta de menor (resp. maior) custo no caminho entre o vértice v e a raiz da sua árvore. Se v for a própria raiz, esta função retornará um valor nulo. Se houverem arestas mínimas (resp. máximas) de mesmo custo no caminho até a raiz, a aresta mais próxima da raiz é retornada.
- `Update(`v, c`)`: atualiza os custos de todas as arestas no caminho entre v e a raiz da árvore, adicionando o valor c ao custo de cada aresta.
- `Evert(`v`)`: altera a árvore contendo o vértice v, transformando v na raiz dessa árvore.

Basicamente, conforme as operações de *link* e *cut* são executadas dinamicamente, estruturas de dados para árvores dinâmicas devem oferecer meios para responder perguntas sobre a relação de pertinência entre um vértice e uma árvore. Por exemplo, a qualquer momento pode-se perguntar a qual árvore pertence o vértice v através da operação `Root(`v`)`. Outra pergunta freqüente é a relação entre dois vértices arbitrários u e v perante as árvores dinâmicas, ou seja, esses nós pertencem à mesma árvore? Essa resposta é dada pela operação `Connected(`u, v`)` ou pela comparação `Root(`u`)` = `Root(`v`)`, que consiste em perguntar se a raiz da árvore que contém o vértice v é a mesma raiz da árvore que contém o vértice u. A terceira operação freqüente em árvores dinâmicas está relacionada aos caminhos dessa árvore, onde a operação `Find_min(`v`)` (resp. `Find_max(`v`)`) retorna a aresta de menor (resp. maior) custo no caminho entre os vértices v e $u =$ `Root(`v`)`.

Em suma, uma estrutura de dados para árvores dinâmicas deve armazenar as arborescências ou árvores que são dinamicamente modificadas, bem como oferecer suporte às operações acima explicitadas. As próximas três seções apresentam as estruturas de dados que dão suporte às operações acima. Um resumo comparativo com as principais características de cada uma das árvores dinâmicas existentes na literatura é apresentado na Seção 3.1.1. Na Seção 3.1.2 é apresentada uma implementação direta de árvores dinâmicas e, por isso, ingênua e simplista, o que resulta em complexidades maiores a custos constantes reduzidos. Por fim, na Seção 3.1.3) é introduzida uma estrutura clássica de árvores dinâmicas, proposta por Sleator e Tarjan [41]. Essa estrutura de dados garante

complexidades menores a custos constantes maiores por operação, já que possui organização interna mais complexa que a primeira, o que é um efeito colateral da garantia de complexidade logarítmica por operação.

3.1.1 Análise das Implementações de Árvores Dinâmicas

Algumas das estruturas de dados usadas para a representação de árvores dinâmicas são apresentadas e referenciadas abaixo:

- *RD-trees*: simples e de fácil implementação; oferece, no pior caso, operações relacionadas a caminhos com complexidades lineares. Por outro lado, o restante das operações pode ser executado em tempo constante.
- *ST-trees* [41]: embora seja uma estrutura complexa, garante complexidades de ordem logarítmica por operação. Existem implementações onde essa complexidade é de pior caso e implementações onde essa complexidade é amortizada.
- *Topology trees* [17, 18]: árvores topológicas baseadas em agrupamento (clusterização) de vértices, resultando em grupos de arestas que são mapeados em uma árvore balanceada, garantindo assim complexidade logarítmica por operação. Contudo, sua implementação é complexa e suporta apenas árvores com nós de grau máximo três.
- *ET-trees* [26, 27]: também conhecidas como *Euler tour trees*, não oferecem operações para consultas a caminhos, como por exemplo para a obtenção da aresta de menor custo no caminho entre dois vértices. Mesmo assim, são uma boa opção para consultas sobre a conectividade entre dois nós (verificar se dois nós u e v pertencem à mesma árvore). Sua implementação é relativamente simples e ainda fornece complexidade logarítmica para todas as operações disponíveis.
- *Top trees* [2, 3, 31]: são árvores muito parecidas com as árvores topológicas de Frederickson [17, 18], já que também são baseadas em clusterizações (mais especificamente, partições topológicas do conjunto de nós da árvore). Por outro lado, estendem as *topology trees* de forma a aceitarem árvores de grau ilimitado. Ainda, sua partição topológica permite o uso de algoritmos simples, baseados em divisão-e-conquista. As *top trees* também oferecem operações com complexidade $O(\log n)$, já que são baseadas em árvores binárias balanceadas.

- *RC-trees* [1]: também denominadas *rake-and-compress trees*, são árvores dinâmicas baseadas em contrações [35]. É uma estrutura mais simples que as árvores topológicas, mas assim como elas, suporta apenas árvores com nós de grau máximo três.

- *Self-adjusting top trees* [47]: este trabalho propõe o uso de estruturas de dados auto-ajustáveis [42, 43] (i.e., não possuem mecanismos rígidos de balanceamento, já que quando a árvore é acessada certas rotinas de ajuste são executadas) para estender as *top trees*.

De maneira geral, pode-se concluir que todas as árvores acima citadas excetuando-se as RD-trees possuem um único objetivo: mapear uma árvore arbitrária em uma árvore balanceada de forma a garantir complexidades logarítmicas [47]. As *ET-trees* realizam essa tarefa de maneira elegante, mas não dão suporte a operações relacionadas a caminhos. Já as *ST-trees* representam as árvores balanceadas a partir de caminhos extraídos das árvores arbitrárias, o que resulta em uma estrutura compacta e eficiente; contudo, não fornecem operações em sub-árvores. Árvores topológicas e *RC-trees* representam árvores arbitrárias a partir de contrações, mas são funcionais apenas em árvores ternárias, binárias ou unárias. Por fim, *top trees* e *self-adjusting top trees* procuram eliminar a exigência de árvores com nós de grau limitado, mas ainda não existem implementações eficientes para as mesmas.

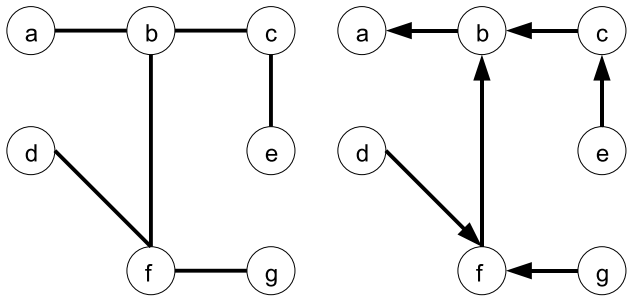
As Seções 3.1.2 e 3.1.3 descrevem duas das principais estruturas de dados para árvores dinâmicas citadas acima, respectivamente *RD-trees* e *ST-trees*.

3.1.2 *RD-trees*: Árvores Dinâmicas Reversas

As *RD-trees* são uma estrutura de dados extremamente simples. Não há na literatura uma referência a essa estrutura; sabe-se que seu surgimento ocorreu através de adaptações à estrutura de dados para árvores reversas (*reversed trees*), utilizadas no algoritmo *set-union* [46].

Estruturas de árvores reversas conectam um nó u ao nó v de forma orientada, através de um arco (u, v). Nesse caso, diz-se que v é pai de u, originando assim o nome de árvores reversas, visto que tradicionalmente um nó possui ponteiros para seus filhos ao invés de um único ponteiro para seu pai. Sob a ótica computacional, um nó $u \in V$ de uma estrutura de árvores reversas $T = (V, E')$ deve possuir os campos $\pi[u]$, onde será armazenado o nó $v \in V$ correspondente ao pai de u, e $\omega[u]$, referente ao custo do arco conectando o nó u ao seu pai.

Cabe ressaltar que neste trabalho será usada a notação *arco*, ao invés de aresta, quando houver referência a estruturas de dados orientadas, como é o caso das árvores reversas. A Figura 3.1 ilustra uma AGM (a) e sua árvore reversa (b).



(a) AGM (b) Árvore reversa

Figura 3.1: Uma AGM mapeada por uma árvore reversa.

Uma implementação simples e direta da estrutura de dados *RD-trees* pode ser obtida a partir de um vetor π com n posições: se cada nó é identificado por uma chave única no intervalo $[1, n]$, então $\pi[u], u \in \{1 \dots n\}$ pode armazenar o inteiro equivalente ao pai de u na árvore $T = (V, E')$. Ainda, o inteiro 0 pode ser usado para indicar a ausência de um pai para um nó arbitrário $r \in \{1 \dots n\}$, nó este que recebe a denominação especial de *nó raiz*. Para armazenar os custos de cada arco (u, v), pode-se utilizar o vetor ω, também com n posições. Nesse vetor, $\omega[u]$ indicaria o custo do arco $(u, \pi[u])$.

Nessa estrutura, onde cada árvore disjunta é representada por uma árvore reversa, as operações propostas por Sleator e Tarjan [41] podem ser assim implementadas:

- `Link`(u, v, c): basta verificar se $\pi[u] = 0$ para determinar se o nó u é realmente uma raiz. Em caso afirmativo, a operação $\pi[u] \leftarrow v$ estabelece a ligação (u, v) enquanto a operação $\omega[u] \leftarrow c$ ajusta o custo desse arco. Esta tarefa tem custo constante.

- `Cut`(u, v): inicialmente, deve-se verificar a existência do arco (v, u) ou do arco (u, v). O arco (v, u) existe quando a condição $\pi[v] = u$ é satisfeita; já a existência do arco (u, v) é verificada quando $\pi[u] = v$. Caso a primeira condição seja constatada, o corte é estabelecido através da operação $\pi[v] \leftarrow 0$; já em caso contrário corta-se o arco através da operação $\pi[u] \leftarrow 0$. Por fim, atualiza-se o vetor ω com um valor especial indicando a ausência desse arco. A complexidade desta operação é $O(1)$.

- `Root`(v): a raiz da árvore que contém o nó v pode ser obtida ao subir pela árvore até que $\pi[u] = 0$; nesse caso a raiz da árvore onde se encontra o nó v seria o nó

u. Podem existir árvores lineares, o que resulta em complexidade linear para que a subida na árvore seja realizada.

- `Cost(`u, v`)`: se $\pi[u] = v$, retorna-se $\omega[u]$. Se $\pi[v] = u$, então retorna-se $\omega[v]$. Quando nenhuma das igualdades acima é satisfeita, o arco (u, v) não existe e um valor especial indicando tal ausência é retornado. Essa operação tem complexidade $O(1)$.

- `Find_min(`v`)`: esta operação é praticamente igual à operação `Root(`v`)`, com a exceção de se salvar o arco de menor custo durante a subida na árvore. Assim, sua complexidade também é $O(n)$.

- `Update(`v, c`)`: ao invés de armazenar o arco de menor custo durante a subida na árvore, como é feito na operação `Find_min(`v`)`, aqui o valor c é adicionado ao valor de ω correspondente aos arcos percorridos durante a subida pela árvore. A complexidade de uma chamada a esse método é $O(n)$.

- `Evert(`v`)`: esta operação, que realiza a troca de raízes em *RD-trees*, é detalhada no Algoritmo 3.

Algoritmo 3 `Evert(`v`)` para a estrutura de dados *RD-trees*

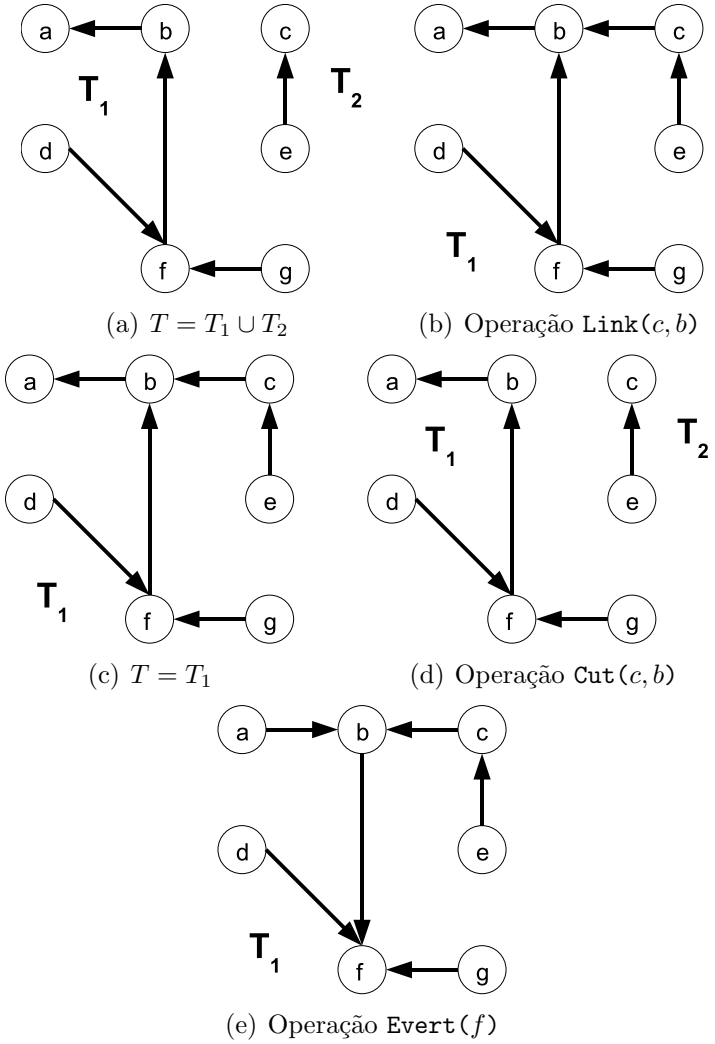
Entrada: O novo nó raiz v e a estrutura de dados *RD-trees* $T = (V, E')$.

```
1: p ← π[v];
2: se p = 0 então
3:     retorne 0;
4: fim se
5: Evert(p);
6: π[p] ← v;
7: π[v] ← 0;
8: ω[p] ← ω[v];
```

A execução do Algoritmo 3 é recursiva. Seu caso base ocorre quando um nó raiz p é atingido, para então as operações das linhas $6 - 8$ transformarem o nó v, que se encontra no topo da pilha de recursividade, na nova raiz da árvore e fazendo do antigo nó raiz p um filho de v. Ao fim da pilha, o nó v da primeira chamada recursiva é transformado em um nó raiz.

O Algoritmo 3 tem seu pior caso dado pela existência de uma árvore linear contendo n nós. Neste caso, são feitas $n - 1$ chamadas recursivas com custo $O(1)$ cada, resultando na complexidade de $O(n)$.

Exemplos das operações acima são mostrados na Figura 3.2. Em (a) e (b), a operação `Link(`c, b`)`; em (c) e (d), a operação `Cut(`c, b`)`; e em (e), a operação `Evert(`f`)`.



(a) $T = T_1 \cup T_2$ (b) Operação `Link`(c, b)

(c) $T = T_1$ (d) Operação `Cut`(c, b)

(e) Operação `Evert`(f)

Figura 3.2: Operações em uma estrutura de dados *RD-trees* $T = (V, E')$.

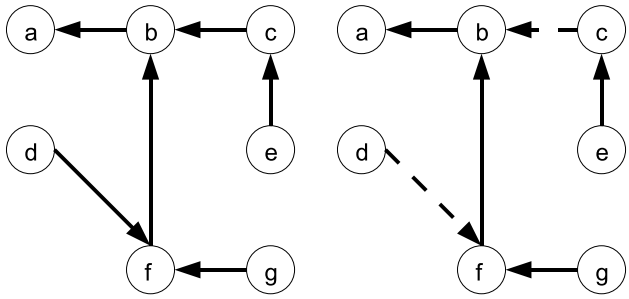
3.1.3 *ST-trees*: Árvores Dinâmicas de Sleator e Tarjan

As *ST-trees* são árvores dinâmicas propostas por Sleator e Tarjan [41], com aplicações em algoritmos eficientes para situações como:

- determinar fluxo máximo em redes;
- calcular árvores geradoras mínimas com restrições; e
- resolver o algoritmo simplex para redes.

Intuitivamente, as *ST-trees* representam os caminhos das árvores dinâmicas, de alturas indefinidas (e possivelmente lineares no pior caso), através de árvores balanceadas, garantindo assim complexidades de ordem logarítmica para as operações sobre árvores

dinâmicas. Para tanto, esta estrutura classifica arbitrariamente os arcos de cada árvore em contínuos ou descontínuos, onde no máximo um arco incidente em cada nó $v \in T$ pode ser contínuo, para então definir uma coleção de caminhos disjuntos formados por arestas contínuas.



(a) Árvore reversa (b) Particionamento

Figura 3.3: Um exemplo de AGM e seu particionamento em caminhos.

A Figura 3.3, que ilustra uma possível classificação dos arcos de uma árvore disjunta representando uma AGM, possui as seguintes definições: os arcos (d, f) e (c, b) são arcos descontínuos, enquanto que os arcos restantes são contínuos. O caminho (g, f), (f, b) e (b, a) é denominado caminho-raiz (*root path*) e deve ser único, dado que o único nó raiz de uma árvore pode ser incidido apenas por uma aresta contínua.

O próximo passo é representar cada caminho contínuo implicitamente, através de uma árvore balanceada. Assim, cada árvore disjunta é mapeada por uma árvore balanceada, que por conseguinte é mapeada, usando seu nó raiz como chave, por uma árvore balanceada global (as árvores balanceadas que representam as árvores dinâmicas são unidas com base em seus caminhos). Para isso, Sleator e Tarjan sugerem o uso de árvores de busca ternárias (*biased search trees*), propostas por Bent et al. [5], que garantem complexidade de ordem logarítmica inclusive para operações de pior caso.

Em [41] são apresentadas duas possíveis implementações de *ST-trees*. De acordo com o esquema de particionamento adotado suas complexidades podem ser amortizadas ou de pior caso, sempre de ordem logarítmica em n. A escolha da árvore balanceada para mapear os caminhos contínuos também pode resultar em complexidades diferentes: árvores binárias de busca (*binary search trees*) resultariam em uma estrutura com complexidade $O(\log^2 n)$ por operação; árvores binárias de busca auto-ajustáveis [43] (*self-adjusting binary search trees* ou *splay trees*) resultariam em *ST-trees* de complexidade amortizada $O(\log n)$ por operação, porém facilitariam sua implementação.

3.2 Árvores Geradoras Mínimas em Grafos Dinâmicos: Algoritmos e Complexidade

Esta seção apresenta as principais estruturas de dados e algoritmos propostos para problemas dinâmicos. A Seção 3.2.1 analisa os principais aspectos de um problema dinâmico, com foco no problema da AGM. Em seguida, é realizado um minucioso estudo dos algoritmos e estruturas de dados propostos para esse problema, considerando cada tipo de alteração que possa ocorrer em tais grafos.

3.2.1 Conceitos Básicos sobre Algoritmos Dinâmicos

O problema em questão consiste em manter a árvore geradora mínima de um grafo sujeito a constantes mudanças. Essas mudanças são divididas em três classes: inserções e remoções de vértices, inserções e remoções de arestas e, por fim, atualizações – incrementos ou decrementos – em custos de arestas. Com relação a essas operações, um algoritmo ou estrutura de dados é dito *totalmente dinâmico* se uma dessas classes de alterações é totalmente suportada. Se apenas uma operação (i.e., apenas incrementos em custos de arestas, ou apenas inserções de vértices) é suportada, esse algoritmo é dito *semi-dinâmico* ou *parcialmente dinâmico*. Ainda, um algoritmo parcialmente dinâmico é dito *incremental* se apenas inserções são suportadas, e *decremental* se apenas remoções são suportadas. Diz-se que um algoritmo é *on-line* quando as alterações são processadas em tempo real (i.e., não há pré-processamento, ou seja, as atualizações são processadas em ordem, uma após a outra).

Abaixo são apresentadas as possíveis mudanças que uma AGM pode sofrer quando seu grafo original (Figura 3.4) é alterado.

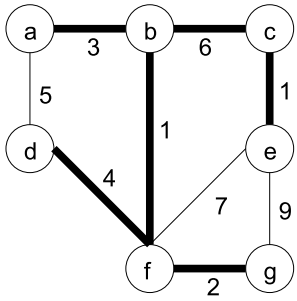
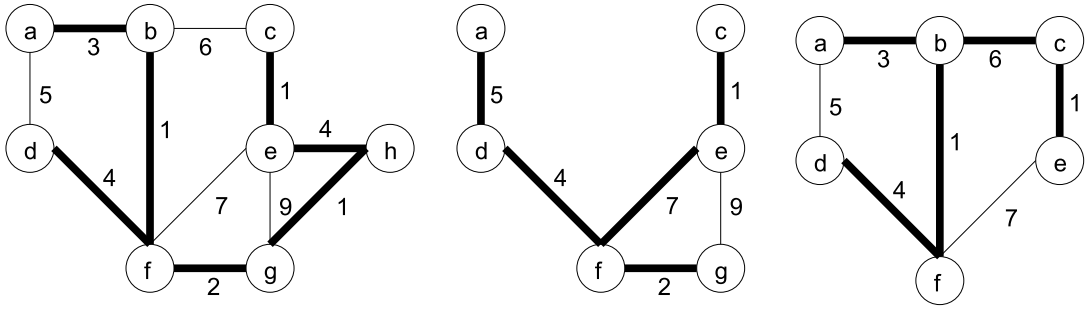


Figura 3.4: Grafo $G = (V, E)$ e sua AGM $T = (V, E')$.

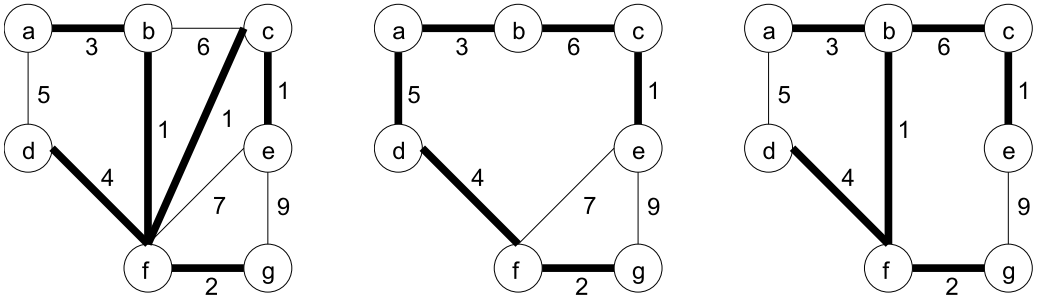
Na Figura 3.5 são apresentados três possíveis casos de inserções e remoções de vértices.



(a) Inserção do vértice h (b) Remoção do vértice b (c) Remoção do vértice g

Figura 3.5: AGMs resultantes de inserções e remoções de vértices em G.

Em (a), o grafo G é modificado através da inserção de um vértice h e um conjunto de arestas $\{(e,h),(g,h)\}$. Remoções de vértices são apresentados em (b) e (c), onde os vértices b em (b), e g em (c), juntamente com suas respectivas arestas incidentes, são removidos do grafo inicial G.



(a) Inserção da aresta (c,f) (b) Remoção da aresta (b,f) (c) Remoção da aresta (e,f)

Figura 3.6: AGMs resultantes de inserções e remoções de arestas em G.

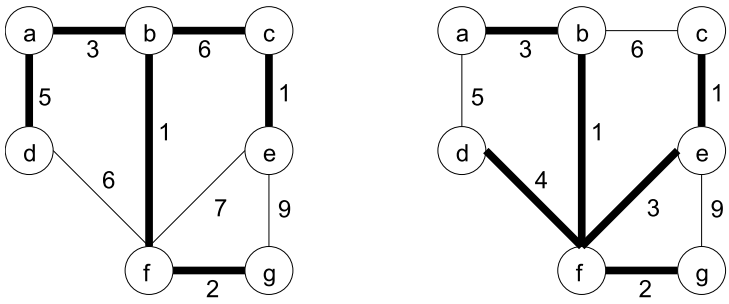
A Figura 3.6 apresenta três das possíveis alterações na AGM causadas por inserções (a) e por remoções de arestas (b) e (c). Em (a) a aresta (c,f) com custo 1 é inserida, refletindo na AGM T a necessidade de remoção da aresta (b,c). Outro caso pode ser formulado a partir da situação (a), onde a mesma aresta (c,f) seria inserida com custo 7 ao invés de 1. Nesse caso a AGM não seria modificada. Por outro lado, a remoção de uma aresta (u,v) só altera a árvore geradora mínima T se $(u,v) \in T$, como pode ser visto em (b), onde a remoção de (b,f) causa alterações em T; em caso contrário, T permanece inalterada, como exemplificado em (c), onde a remoção da aresta (e,f) não reflete em T.

Dois fatos resultam da inserção e remoção de arestas em grafos dinâmicos:

Fato 3.1 *A inserção em G da aresta arbitrária (u,v) resultará em alterações na AGM $T=(V,E')$ se e somente se existir uma aresta (x,y) no ciclo induzido pela inserção de*

(u, v) *em* T *tal que* $w(x, y) > w(u, v)$. *Nesse caso, a nova AGM será* $T = (V, (E' - \{(x, y)\}) \cup \{(u, v)\})$.

Fato 3.2 *A remoção de* G *da aresta arbitrária* (u, v) *causará modificações na AGM* $T = (V, E')$ *se e somente se* $(u, v) \in E'$. *Tomando* $T_u = (V_u, E'_u)$ *e* $T_v = (V_v, E'_v)$ *como as duas árvores resultantes da remoção da aresta* (u, v), *deve-se encontrar a aresta de custo mínimo* (x, y) *tal que* $x \in V_u$ *e* $y \in V_v$. *Caso essa aresta não exista,* T *não mais será uma árvore geradora mínima, mas sim uma floresta geradora mínima (FGM, ou Minimum Spanning Forest – MSF).*



(a) Incremento no custo de (d, f) (b) Decremento no custo de (e, f)

Figura 3.7: AGMs resultantes de mudanças no custo de arestas.

Mudanças nos custos podem ocorrer através de operações de incremento ou decremento no custo de uma aresta, como ilustrado nas Figuras 3.7 (a) e (b), respectivamente. Deve-se lembrar que, se o grafo $G = (V, E)$ possui uma AGM $T = (V, E')$, operações de incremento e decremento não desconectam essa árvore (i.e., a AGM sempre existirá) – apenas seu custo pode ser alterado. Incrementos e decrementos em custos de arestas resultam nos seguintes fatos:

Fato 3.3 *O incremento no custo da aresta arbitrária* (u, v) *resultará em mudanças na AGM* $T = (V, E')$ *se e somente se (a)* $(u, v) \in E'$ *e (b) existir uma aresta* (x, y) *conectando as árvores* T_u *e* T_v *resultantes da remoção de* (u, v) *tal que* $w(x, y) < w(u, v)$.

Fato 3.4 *O decremento no custo da aresta arbitrária* (u, v) *causará alterações na AGM* $T = (V, E')$ *se e somente se (a)* $(u, v) \notin T$ *e (b) existe uma aresta* (x, y) *no ciclo induzido pela aresta* (u, v) *em* T *tal que* $w(x, y) > w(u, v)$. *Nesse caso, a nova AGM é dada por* $T' = (V, (E' - \{(x, y)\}) \cup \{(u, v)\})$.

Os Fatos 3.1, 3.2, 3.3 e 3.4 são baseados nas seguintes propriedades de uma árvore geradora mínima $T = (V, E')$ com relação ao grafo $G = (V, E)$ [32]:

(i) Propriedade do ciclo: para cada ciclo C em G, a aresta de maior custo em C não faz parte de T.

(ii) Propriedade do corte: para qualquer subconjunto $X \subseteq V$ não vazio de vértices, a aresta de menor custo $(u, v) \in E$ tal que $u \in X$ e $v \notin X$ pertence à AGM T.

3.2.2 Algoritmos para Inserção e Remoção de Vértices

Spira e Pan [44] foram os primeiros a estudar os problemas para manutenção de propriedades de grafos sujeitos a alterações dinâmicas. Em [44] foi proposto um procedimento que atualiza uma AGM $T = (V, E')$ após a inserção no grafo $G = (V, E)$ do vértice v e do conjunto de arestas $\{(v, x) \quad | \quad x \in V\}$. O novo conjunto de vértices após a inserção é $V' = V \cup \{v\}$, enquanto que o novo conjunto de arestas é $F = E \cup \{(v, x) \quad | \quad x \in V\}$. Este procedimento é descrito no Algoritmo 4.

Algoritmo 4 Algoritmo de Spira e Pan para inserção de nós.

Entrada: O grafo formado pela AGM $T = (V, E')$, o novo vértice v e as novas arestas adjacentes a v.

1: $V' = V \cup \{v\}$;
2: $E' = \{\emptyset\}$;
3: Armazenar em E' a aresta de menor custo incidente em cada nó de V';
4: Encontrar as componentes conexas do grafo formado por V' e E';
5: Encontrar a aresta de menor custo incidente em cada componente acima;
6: Compactar em um único nó cada uma das componentes conexas acima;
7: Voltar ao passo 3 caso exista mais de um nó;
8: **retorne** $T = (V', E')$;

Teorema 3.5 [44] *O Algoritmo 4 atualiza a árvore geradora mínima com complexidade* $O(|V'|) = O(n)$.

Prova Seja $|V'| = n' = n + 1$. O passo 3 requer no máximo $4n'$ comparações. As componentes conexas (passo 4) podem ser encontradas em $O(n')$ usando o algoritmo de Tarjan [45]. O passo 5 pode ser executado em $O(n')$ se forem processadas apenas as arestas não analisadas no passo 3. O passo 6 é trivial, resultando em um grafo com no máximo a metade do número de vértices existentes no grafo atual. Assim, a recursão

$T(n')$ é resolvida pela equação

$$\begin{aligned} T(n') &\leq T(n'/2) + cn' \\ T(n') &= O(n). \end{aligned} \tag{3.6}$$

Um exemplo da aplicação do Algoritmo 4 é ilustrado pela Figura 3.8

Outros resultados de Spira e Pan [44] dizem respeito a alterações em custos de arestas, onde são provados os limites superiores de $O(n^2)$ arestas a serem analisadas após um incremento de custo e $O(n)$ arestas a serem analisadas após um decremento de custo, e a remoção de nós, onde é provado o limite de $O(n^2)$ para a atualização de T.

Ainda, Chin e Houck [11] apresentam um algoritmo também linear em n para a inserção de nós e um algoritmo que determina em $O(n^2)$ todas as possíveis soluções para a remoção de qualquer vértice de T.

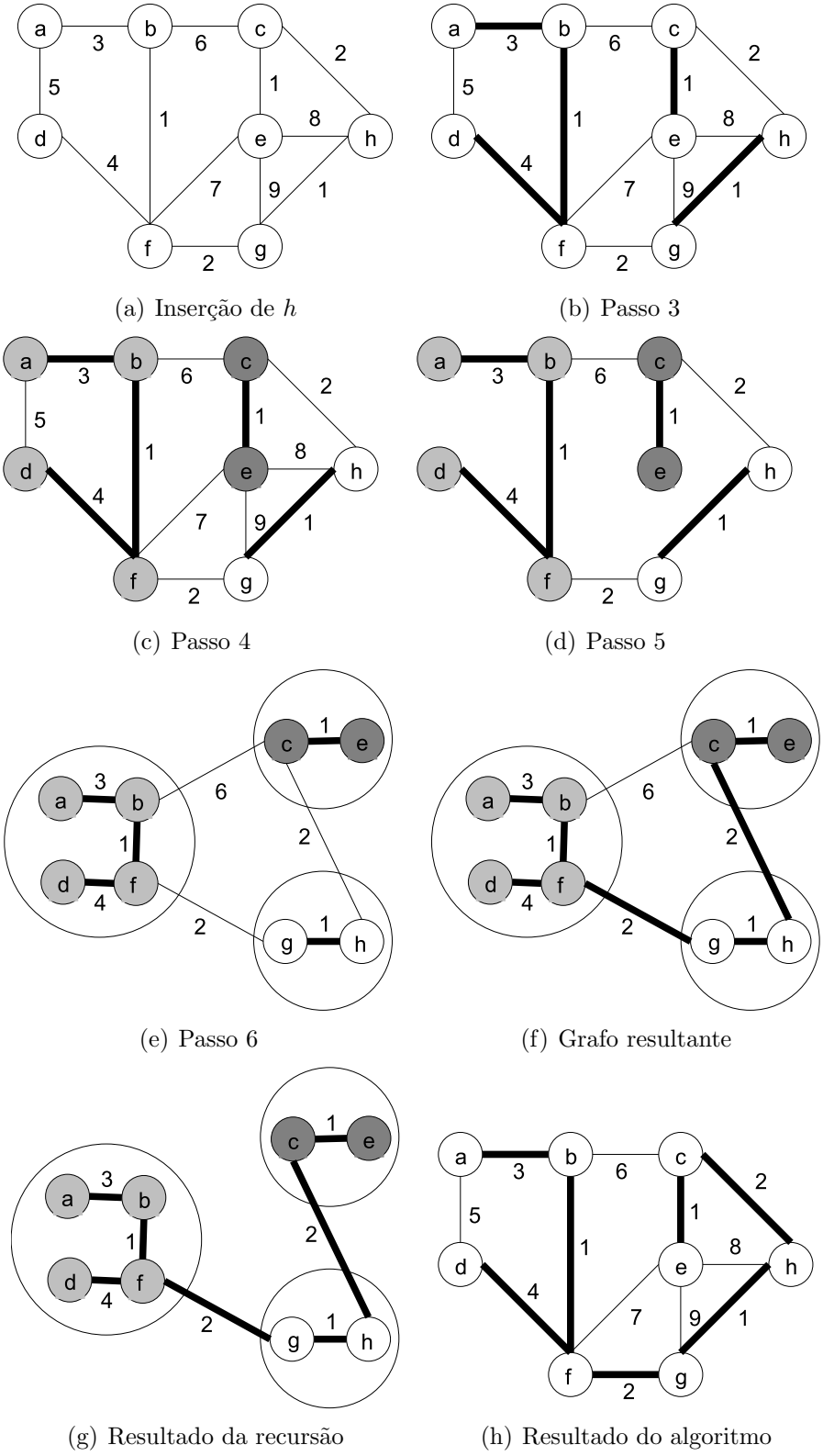
Por fim, Das e Loui [13] propõem um algoritmo semi-dinâmico decremental (somente a remoção de nós é considerada). É fornecido um algoritmo que atualiza a AGM T em $O(m\alpha(m,n))$ se assumida a ordenação prévia das arestas por seus custos; em caso contrário a complexidade é de $O(m \log n)$, o que não melhora a execução de um algoritmo estático (clássico). Em [13] também é apresentado um algoritmo paralelo com complexidade $O(\log^2 n)$ quando utilizados m processadores em uma CREW PRAM (*Concurrent Reading and Exclusive Writing Parallel Random Access Machine*).

3.2.3 Algoritmos e Estruturas de Dados para Inserção e Remoção de Arestas

Algoritmos para atualização de árvores geradoras mínimas após a inserção ou remoção de arestas foram propostos em [16, 17, 18, 26, 31]. Estes trabalhos são apresentados nas seções seguintes.

3.2.3.1 Árvores Topológicas para Inserções e Remoções de Arestas

As árvores topológicas ou *Topology Trees* foram propostas por Frederickson [17] com base em uma técnica denominada *clusterização*. Com esta técnica, os vértices da AGM $T = (V, E')$ são particionados em grupos de sub-árvores conexas, de forma que cada grupo tenha poucas sub-árvores adjacentes. Uma árvore topológica consiste em representar a AGM usando uma árvore 1-2-3-4 balanceada, com altura logarítmica, formada por



(a) Inserção de h

(b) Passo 3

(c) Passo 4

(d) Passo 5

(e) Passo 6

(f) Grafo resultante

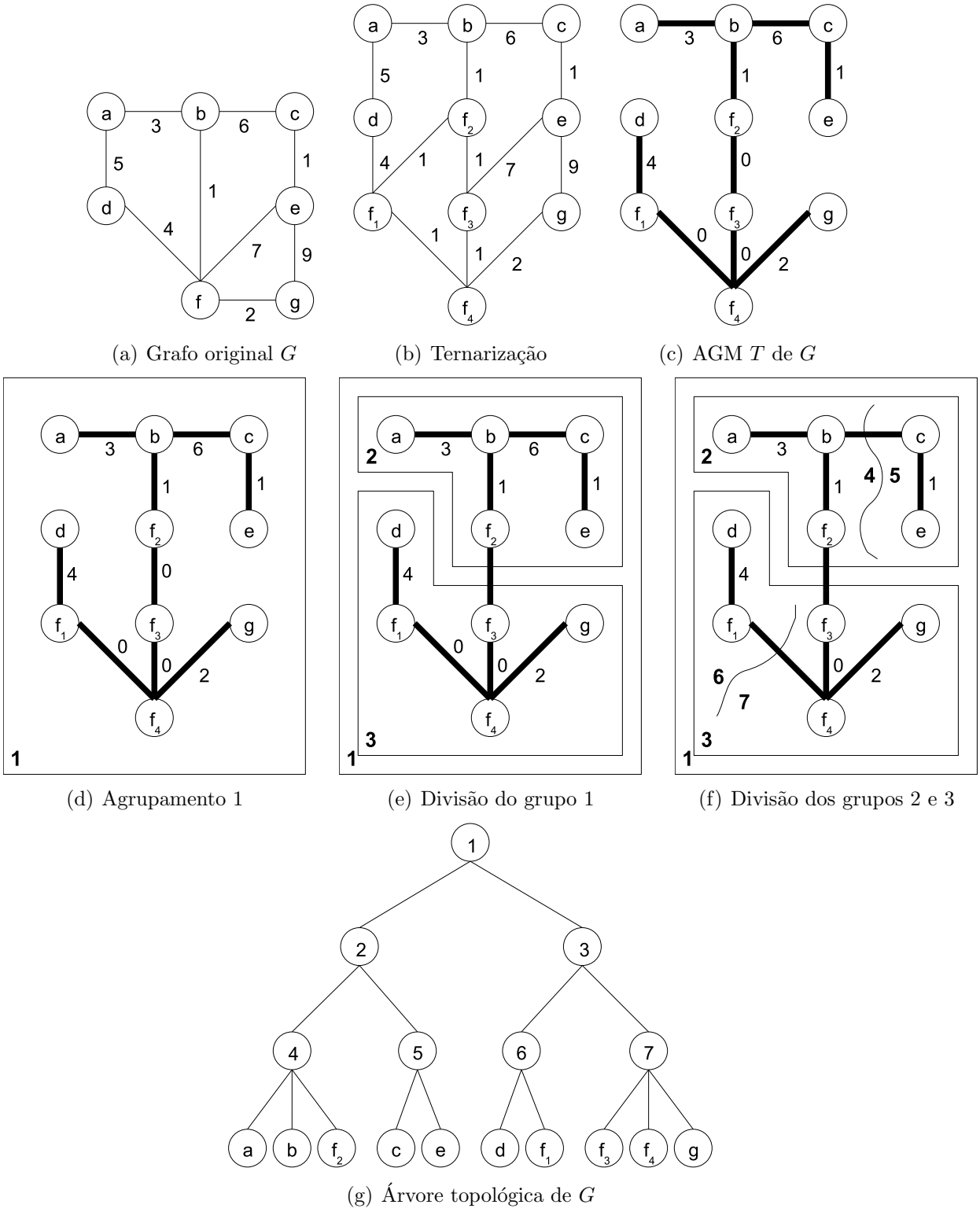
(g) Resultado da recursão

(h) Resultado do algoritmo

Figura 3.8: Execução do algoritmo de Spira e Pan [44] para a inserção do vértice h.

clusterizações recursivas. Adicionalmente, as arestas pertencentes ao subconjunto $E - E'$ podem ser mantidas em uma *árvore topológica bidimensional* (*2-d topology tree*). Uma

árvore topológica é detalhada pela Figura 3.9.



(a) Grafo original G (b) Ternarização (c) AGM T de G

(d) Agrupamento 1 (e) Divisão do grupo 1 (f) Divisão dos grupos 2 e 3

(g) Árvore topológica de G

Figura 3.9: Um grafo e sua respectiva árvore topológica.

Nessa estrutura, árvores geradoras mínimas podem ser mantidas por árvores topológicas bidimensionais com complexidade $O(m^{1/2})$ [17, 18]. Algoritmos mais simples,

que fazem uso apenas de árvores topológicas, atualizam uma AGM com complexidade $O((m \log m)^{1/2})$ [21, 40]. Se apenas a técnica de clusterização é empregada, algoritmos podem ser implementados com complexidade $O(m^{2/3})$.

Os algoritmos propostos por Frederickson [17, 18] são usuais apenas em grafos com vértices de grau máximo três. Com isso, sua utilização é complicada pela necessidade de conversão dos grafos que não satisfazem esta exigência. Ainda assim, um algoritmo para tal conversão é apresentado em [22].

3.2.3.2 Técnica de Esparsificação

Esparsificação [16] é uma técnica utilizada para melhoria de algoritmos que não requer que estes sejam especificamente desenvolvidos para se beneficiar da mesma. Essa técnica faz com que algoritmos com complexidades dependentes do número de arestas do grafo passem a ter complexidades de grafos esparsos, ou seja, reduzidas de um fator $O(m)$ para $O(n)$. Mais especificamente, algoritmos dinâmicos com complexidade $T(n, m)$ para grafos com n vértices e m arestas passam a ter complexidade $T(n, O(n)) \log(m/n)$ ou $T(n, O(n))$, dependendo da técnica de esparsificação utilizada – simples ou otimizada. Por exemplo, os algoritmos de Frederickson [17], de complexidade $O(m^{1/2})$, tem complexidade de $O(n^{1/2})$ ao se usar a versão otimizada de esparsificação, aplicando-se esses algoritmos diretamente sobre os grafos esparsos gerados pela técnica.

A técnica de esparsificação é relativamente simples: as arestas do grafo $G = (V, E)$ são particionadas em $\lceil m/n \rceil$ grupos, cada um contendo exatamente n arestas, exceto o último grupo. Cada grupo deve possuir um *certificado*, especificado na Definição 3.7, que é então encapsulado em um nó folha. Os nós folha são então agrupados dois a dois em nós-certificado, de forma a construir uma árvore binária completa da base até o topo de forma recursiva.

Definição 3.7 [16] *Para qualquer propriedade $\mathcal{P}$ e grafo $G = (V, E)$, um certificado para G é um grafo $G' = (V, E')$ que, para qualquer grafo $H = (V, F)$, $E \cup F$ possui a propriedade $\mathcal{P}$ se e somente se $E' \cup F$ possuir a propriedade $\mathcal{P}$.*

A árvore de esparsificação simples do grafo da Figura 3.4 é apresentada na Figura 3.10. Se uma atualização é realizada em qualquer aresta de G, no máximo $\log n$ nós serão afetados. Como cada nó da árvore de esparsificação contém n arestas, a complexidade dessa atualização do grafo original cai de $O(m)$ para $O(n \log n)$, que corresponde à aplicação

desse algoritmo sobre os $\log n$ níveis da árvore de esparsificação ao custo de $O(n)$ para cada nível. A raiz dessa árvore armazena a solução do problema para o grafo G.

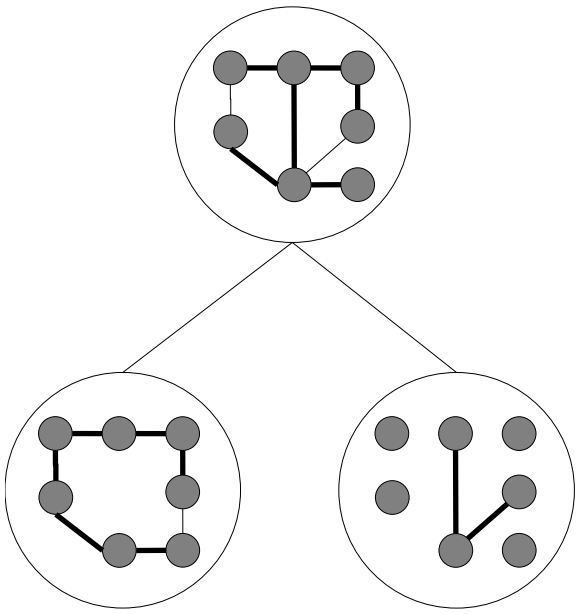


Figura 3.10: Árvore de esparsificação – para construir uma AGM no nível l (nesse caso $l = 0$, o que corresponde ao nó raiz da árvore de esparsificação) só são consideradas as arestas que estão nas AGMs dos filhos do nó atual (nesse caso, no nível $l = l + 1$).

3.2.3.3 Algoritmo HK

Até a seção anterior, todos os métodos para atualização de árvores geradoras mínimas eram baseados em uma única estrutura de dados dinâmica. Essa estrutura deve ser responsável por tratar inserções e remoções de arestas de forma a manter uma AGM do grafo dinâmico. Contudo, essa abordagem chegou a seu limite teórico com a introdução da técnica de esparsificação. Em [16] foi provado que uma estrutura de dados para manutenção de AGMs não poderia mais suportar complexidades assintoticamente rápidas (lineares) sem pagar o preço do armazenamento não-linear, um problema freqüente em estruturas de dados estáticas que também se tornou um limitante para as estruturas de dados dinâmicas.

Desta forma, em [27] é apresentado o primeiro algoritmo randomizado de complexidade polilogarítmica para os problemas dinâmicos de conectividade e árvore geradora mínima aproximada. Para tanto, foram elaborados algoritmos e estruturas de dados que utilizam árvores dinâmicas mais simples e rápidas, as *ET-trees*, ao invés de se utilizar uma única e complexa estrutura de dados.

Seguindo a tendência de se produzir algoritmos bem elaborados ao invés de complexas estruturas de dados monolíticas, o algoritmo para manutenção de árvores geradoras mínimas HK foi proposto por Henzinger e King [26].

Embora utilize uma abordagem completamente diferente daquela apresentada por Frederickson [17, 18], a complexidade do algoritmo HK é de apenas $O(n^{1/3} \log n)$ quando amortizada sobre um mínimo de m operações, o que não reduz significativamente os limites teóricos impostos pela utilização de árvores topológicas. Adicionalmente, os algoritmos propostos são de difícil implementação, e ainda requerem grafos com custos distintos, introduzindo uma camada extra de dificuldade. Contudo, uma importante técnica é proposta nesse trabalho: a partir de um algoritmo semi-dinâmico para remoções de arestas, fez-se possível transformá-lo em um algoritmo totalmente dinâmico através da criação e manutenção de $s = \lceil \log m \rceil$ estruturas semi-dinâmicas.

Os algoritmos apresentados na próxima seção fazem uso da técnica proposta por Henzinger e King [26] para compor um algoritmo com complexidade amortizada de $O(\log^4 n)$ para atualizar uma AGM.

3.2.3.4 Algoritmo HDT

O trabalho de Holm et al. [30, 31] trouxe os primeiros e únicos algoritmos determinísticos de complexidade polilogarítmica para os problemas de conectividade, floresta geradora mínima e biconectividade. Todos os algoritmos determinísticos até então apresentados trazem complexidades maiores que a ordem linear de grandeza.

O algoritmo apresentado possui complexidade $O(\log^4 n)$ e é baseado no trabalho de Henzinger e King [26], onde uma estrutura semi-dinâmica é utilizada como base para a composição de uma estrutura totalmente dinâmica. Tomando por base um algoritmo que mantém uma floresta geradora de um grafo dinâmico (estrutura para manutenção de conectividade), o algoritmo então é especializado de modo a manter uma floresta geradora mínima desse grafo, satisfazendo então o problema aqui estudado.

Neste algoritmo para manutenção de conectividade em grafos dinâmicos, uma floresta geradora F deve ser mantida a partir do grafo $G = (V, E)$. Projetada para fornecer algoritmos com baixa complexidade amortizada, a técnica proposta associa um nível $l(u, v)$ a cada aresta $(u, v) \in E$, com $l(u, v) \leq L = \log n$. Para cada nível i, F_i deve armazenar a floresta geradora para as arestas $(u, v) \in E$ tal que $l(u, v) \geq i$, implicando em $F = F_0 \supseteq F_1 \supseteq F_2 \ldots \supseteq F_L$. Os seguintes invariantes são mantidos:

(i) o número máximo de nós conectados em uma árvore F_i é $\lfloor n/2^i \rfloor$. Isso resulta em L ser o nível onde existe o maior número de nós conectados na estrutura.

(ii) $F = (V, E')$ é a maior floresta geradora de $G = (V, E)$, ou seja, se $(u, v) \in E - E'$, então os vértices u e v já estão conectados na floresta F do nível $l(u, v)$, ou seja, $F_{l(u,v)}$.

Inicialmente, como todas as arestas se encontram no nível mais baixo ($l = 0$), ambos os invariantes acima são satisfeitos. A amortização é baseada no argumento que o nível de uma aresta só pode ser incrementado L vezes e, para tanto, o nível de uma aresta arbitrária nunca pode diminuir. Intuitivamente, uma aresta que não pertence a F deve subir de nível quando seus vértices já estão conectados em um nível mais alto da estrutura. Os algoritmos para inserção e remoção de arestas são apresentados abaixo:

- `Insert`(u, v): a aresta é inserida no nível 0 da estrutura. Caso u não esteja conectado a v em F, então $F_0 \leftarrow F_0 \cup \{(u, v)\}$.

- `Delete`(u, v): se $(u, v) \notin F$, basta remover essa aresta da estrutura. Caso contrário, uma aresta de reposição deve ser encontrada, reconectando F no nível mais alto possível. Pela estratégia adotada para a alocação das arestas em F, sabe-se que esta aresta de reposição se encontra no nível $l \leq l(u, v)$. Desta forma, o algoritmo `Replace`$(u, v, l(u, v))$ faz a busca por uma aresta substituta.

- `Replace`(u, v, i): pela definição da estrutura de dados, não existe uma aresta que conecte os vértices u e v no nível $l > i$. Considere-se $T_u = (V_u, E'_u)$ e $T_v = (V_v, E'_v)$ como as árvores desconectadas pela remoção da aresta (u, v), assumindo-se $|V_u| \leq |V_v|$ sem perda de generalidade. Deve-se notar que $T = (V, E'_u \cup \{(u, v)\} \cup E'_v)$ era uma árvore no nível i com pelo menos o dobro de vértices de T_u e que, por (i), $|V_u| \leq \lfloor n/2^{i+1} \rfloor$. Desta forma, preservando-se os invariantes (i) e (ii), as arestas de T_u podem ter seu nível aumentado em uma unidade, ou seja, o conjunto E_u pode ser movido para o conjunto de árvores de F_{i+1}. Agora, o algoritmo deve realizar uma busca no nível i, dentre cada uma das arestas incidentes a vértices de T_u, até que (a) uma aresta de reposição seja encontrada ou (b) todas as arestas do nível i sejam analisadas. Para isso, seja (x, y) a aresta a ser analisada: se essa aresta conecta as árvores desconectadas pela remoção de (u, v) então a busca é suspensa. Se, por outro lado, essa aresta não conecta T_v a T_u, isso significa que ambos os vértices x e y pertencem à mesma árvore – especificamente T_u ou T_v. O preço da análise dessa aresta é pago através do incremento, em uma unidade, de seu nível. Por fim, caso não existam mais arestas a serem consideradas e o nível analisado é $i > 0$, a busca deve

ser realizada em um nível abaixo, através da chamada recursiva `Replace`$(u, v, i-1)$. Em caso contrário, a remoção da aresta (u, v) deixa desconectadas as árvores T_u e T_v.

Embora de simples descrição, as operações acima necessitam de estruturas de dados mais elaboradas para que atinjam a complexidade anunciada. Essa complexidade será analisada mais adiante. Para cada nível i, é necessária a manutenção da floresta geradora F_i e das arestas em i não pertencentes a F_i. Para um dado vértice v e uma floresta geradora F_i, existe a necessidade das seguintes operações:

- identificar a árvore $T_v \in F_i$;
- obter o tamanho de T_v;
- encontrar uma aresta de T_v no nível i, se houver;
- encontrar uma aresta do nível i incidente a T_v, se houver; e
- remover e inserir uma aresta no nível i, seja em F_i ou não.

Todas as operações acima podem ser executadas pela estrutura de dados *ET-trees*, de Henzinger e King [27]. Além de ser uma representação eficiente para árvores dinâmicas, seu rebalanceamento após operações de inserção e remoção de arestas pode ser realizado em $O(\log n)$. Assim, a cada nível i é associada uma estrutura de dados *ET-trees* para representar as arestas de F_i. Cada nó v dessa estrutura deve armazenar informações sobre a existência de arestas incidentes a v que não fazem parte de F_i mas estão no nível i. O Teorema 3.8 apresenta a complexidade das operações de atualização da estrutura acima.

Teorema 3.8 [30] *Dado um grafo G com m arestas e n vértices, existe um algoritmo determinístico totalmente dinâmico que responde consultas sobre conectividade em tempo $O(\log n / \log\log n)$ no pior caso, e que atualiza a floresta geradora de G em tempo amortizado de $O(\log^2 n)$ para cada operação de inserção ou remoção de aresta.*

Duas alterações fazem com que o algoritmo acima suporte também atualizações de florestas geradoras mínimas. A primeira delas é que a floresta geradora inicial deve ser mínima. A segunda, e mais importante, se encontra na função `Replace`(u, v, i): as arestas incidentes a T_v devem ser analisadas em ordem crescente de custo. Para que isso funcione,

os custos das arestas devem ser distintos. A corretude do algoritmo pode ser provada através da manutenção dos invariantes (i) e (ii), junto com o seguinte invariante:

(iii) Se a aresta (u, v) possui o maior custo dentre as arestas do ciclo C, ela também possui o menor nível l dentre as arestas pertencentes a C.

Este algoritmo também possui complexidade de $O(m \log n^2)$, que amortizado sob uma seqüencia de $\Omega(m)$ operações resulta na complexidade de $O(\log^2 n)$ (c.f. Teorema 7.3 de Holm et al. [30]). Entretanto, operações de inserção de arestas deixam de ser suportadas. Desta forma, de acordo com a técnica apresentada em [26], adicionando-se $\log n$ estruturas semi-dinâmicas para remoção, uma operação de inserção ou remoção de aresta pode ser executada em $O(\log^4 n)$ [30].

Concluindo esta seção, a Tabela 3.2.4 sumariza a evolução dos algoritmos aqui apresentados, enfatizando suas complexidades [30].

3.2.4 Algoritmos para Alteração no Custo de Arestas

Os problemas de atualização de AGMs devido a alterações em custos de arestas não possuem algoritmos especificamente desenvolvidos. Isso porque os algoritmos para inserção e remoção de arestas podem ser usados para esse fim com a mesma complexidade teórica.

Desta forma, considere um algoritmo para inserção e remoção de arestas com complexidade $O(f(n))$ para manter a AGM T. Se, após uma alteração no custo da aresta (x, y), essa aresta for primeiro removida e então inserida com seu novo custo, a complexidade dessa tarefa passa a ser de $O(2f(n)) = O(f(n))$. Assim, todos os algoritmos acima apresentados também podem ser aplicados para o caso de alterações nos custos de arestas.

Contudo, para casos onde o desempenho é mandatório, como por exemplo em buscas locais, usar duas operações (remoção seguida de inserção) para atualizar uma AGM após uma mudança de custo pode resultar em uma tarefa cara. Além disso, todas as estruturas de dados e algoritmos até então estudados introduzem camadas complexas de processamento e armazenamento, resultando em tempos constantes bastante elevados. Essas constantes, embora não sejam exibidas pela análise assintótica, podem ser relevantes no desempenho dos algoritmos.

Tabela 3.1: Evolução dos algoritmos para grafos dinâmicos. Tempos com * significam complexidade esperada (algoritmos randômicos). A notação $\tilde{O}(f(n))$ indica complexidades polilogarítmicas omitidas, em função de $f(n)$, ou seja, esta complexidade corresponde a $O(f(n) \cdot polylog(n))$. Representa-se por $\mathbf{\Delta}$ o maior grau de um vértice pertencente ao grafo. Um grafo é biconexo (resp. conectado por duas arestas) se e somente se a remoção de qualquer vértice (resp. aresta) não resulta na desconexão desse grafo.

Referência	Conectividade	FGM	Conectividade (2 arestas)	Biconectividade
Frederickson [17]	$O(m^{1/2})$	$O(m^{1/2})$	-	-
Frederickson [18]	-	-	$O(m^{1/2})$	-
Eppstein et al. [16]	$O(n^{1/2})$	$O(n^{1/2})$	$O(n^{1/2})$	-
Rauch [38]	-	-	-	$O(m^{2/3})$
Rauch [39]	-	-	-	$O(m^{1/2})$
Henzinger e King [25]	$O(\log^3 n)*$	-	$O(\log^5 n)*$	$\tilde{O}(\Delta)*$
Henzinger e Poutré [28]	-	-	-	$\tilde{O}(n^{1/2})$
Henzinger e Thorup [29]	$O(\log^2 n)*$	-	-	-
Henzinger e King [26]	$\tilde{O}(n^{1/3})$	$\tilde{O}(n^{1/3})$	-	-
Holm et al. [30, 31]	$O(\log^2 n)$	$O(\log^4 n)$	$O(\log^4 n)$	$O(\log^4 n)$

3.3 Análises Experimentais em AGMs Dinâmicas

Dois trabalhos procuram conduzir avaliações experimentais de algoritmos para árvores geradoras mínimas em grafos dinâmicos. O primeiro deles foi proposto por Amato et al. [4] e inclui implementações de clusterização, árvores topológicas, árvores topológicas bidimensionais, esparsificação simples sobre clusterização e um algoritmo ingênuo, parcialmente dinâmico. O segundo trabalho é de autoria de Cattaneo et al. [10] e complementa [4], introduzindo implementações do algoritmo polilogarítmico de Holm et al. [30, 31] e de um algoritmo totalmente dinâmico mais simples e de complexidade elevada.

Os algoritmos implementados por Amato et al. [4] são apresentados abaixo e resumidos na Tabela 3.3.

- **Árvores topológicas**: esses algoritmos foram propostos por Frederickson [17, 18] e são baseados em dois esquemas diferentes de particionamento dos vértices, um proposto em [17] e outro proposto em [18]. Ainda são propostas alterações nos algoritmos de Frederickson a fim de utilizar um esquema de partição mais simples que os demais, além de aplicar atualizações preguiçosas (*lazy updates*), onde um novo nó não é instantaneamente adicionado na árvore topológica quando necessário, evitando assim que essa adição seja seguida pela remoção do mesmo.

- **Esparsificação no topo da clusterização**: os dois algoritmos implementados utilizam a técnica de esparsificação simples, garantindo assim a complexidade de $O(f(n, O(n)) \log(m/n))$ para um algoritmo com tempo $f(n, m)$. Os certificados adotados foram as próprias AGMs dos grafos esparsos gerados pelo particionamento.

- **Algoritmo ingênuo**: o algoritmo ingênuo implementado é parcialmente dinâmico. As arestas do grafo são mantidas em uma fila de prioridades, enquanto as arestas da AGM são armazenadas em *ST-trees*. Assim, quando uma aresta é inserida no grafo, basta consultar a árvore dinâmica sobre a maior aresta no ciclo para determinar se a aresta inserida deve ou não entrar na AGM. Quando uma aresta é removida e pertence à AGM, o algoritmo de Kruskal [33] é aplicado sobre a fila de prioridades.

Dentre os seis algoritmos implementados a partir dos trabalhos de Frederickson, `FredI-85` obteve os melhores tempos computacionais. Desta forma, a análise experimental avaliou, além deste, os seguintes algoritmos: `FredI-Mod`, `Spars(I-85)`, `Spars(I-Mod)` e `Ad-hoc`. A metodologia proposta dividiu os experimentos em duas classes, a citar: classe de experimentos aleatórios e classe de experimentos estruturados.

Tabela 3.2: Algoritmos implementados por Amato et al. [4].

Algoritmo	Estrutura de dados	Complexidade
`FredI-85`	Clusterização	$O(m^{2/3})$
`FredII-85`	Árvores topológicas	$O((m \log m)^{1/2})$
`FredIII-85`	Árvores topológicas bidimensionais	$O(m^{1/2})$
`FredI-91`	Clusterização	$O(m^{2/3})$
`FredII-91`	Árvores topológicas	$O((m \log m)^{1/2})$
`FredIII-91`	Árvores topológicas bidimensionais	$O(m^{1/2})$
`FredI-Mod`	Clusterização	$O(m^{2/3})$
`Spars(I-85)`	Esparsificação sobre clusterização	$O(n^{2/3})$
`Spars(I-Mod)`	Esparsificação sobre clusterização	$O(n^{2/3})$
`Ad-hoc`	*ST-trees* e fila de prioridades	$O(m \log n)$

Para a primeira classe de experimentos, foram gerados grafos aleatórios com n vértices e m arestas, com custos escolhidos aleatoriamente. As seqüências de atualizações do grafo também foram aleatórias, compostas por 2000 atualizações (1000 inserções e 1000 remoções de arestas). Foram feitas execuções utilizando dez sementes diferentes. Resultados para grafos com 200 vértices e de 200 a 10000 arestas e para grafos com 700 vértices e de 700 a 149000 arestas foram apresentados, para os quais o desempenho do algoritmo `Ad-hoc` foi superior aos outros algoritmos. A ordem dos resultados – `Ad-hoc`, `FredI-Mod`, `FredI-85`, `Spars(I-Mod)` e `Spars(I-85)` – pode ser explicada pela teoria dos grafos: o algoritmo `Ad-hoc` possui excelente desempenho em todos os casos com exceção da remoção de arestas da AGM. Como isso ocorre com probabilidade de n/m em grafos aleatórios, o resultado prático foi compatível com a teoria.

Por outro lado, os experimentos estruturados foram projetados para forçar a ocorrência de seqüências caras de atualizações, formadas exclusivamente por remoções de arestas da AGM, que é o pior caso para o problema de manutenção de árvores geradoras mínimas em grafos dinâmicos (todos os outros casos podem ser tratados em $O(\log n)$). Neste caso, o algoritmo `Ad-hoc` foi prejudicado pelas seqüências de atualizações e seu tempo não foi contabilizado nos experimentos. O algoritmo `Spars(I-Mod)` teve a melhor performance, seguido por `FredI-Mod`, `Spars(I-85)` e `FredI-85`.

O trabalho de Cattaneo et al. [10] complementa os estudos realizados em [4] ao introduzir o algoritmo de Holm et al. [31]. Os algoritmos analisados foram:

- `Spars(I-Mod)`: este é o mesmo algoritmo que obteve o melhor desempenho na análise experimental de Amato et al. [4].

- `ST`: para o grafo $G = (V, E)$ e AGM $T = (V, E')$, este algoritmo armazena as arestas do subconjunto $E - E'$ ordenadas sob uma árvore balanceada (AVL), garantindo $O(\log n)$ em operações tanto de inserção quanto de remoção de arestas que não fazem parte da AGM. Esta árvore balanceada é usada em conjunto com uma estrutura *ST-trees*, garantindo também complexidade logarítmica em inserções de arestas. A remoção de arestas da AGM é feita através de um percurso ordenado na árvore, onde para cada aresta visitada é feita uma pergunta sobre conectividade da AGM, através das *ST-trees*. Esta operação tem complexidade $O(m \log n)$, já que, no pior caso, são visitadas $O(m)$ arestas, com complexidade $O(\log n)$ para cada chamada à função `Connected` das *ST-trees*.

- `ET+ST`: foi constatado que a operação `Connected` da estrutura *ST-trees* é muito lenta quando comparada à da estrutura *ET-trees*. Assim sendo, este algoritmo mantém duas AGMs: a primeira, sobre as *ET-trees* propriamente ditas, executa perguntas sobre conectividade no algoritmo de remoção de arestas da AGM. A segunda, sobre as *ST-trees*, obtém a aresta de maior custo em um caminho para o algoritmo de inserção de arestas (já que as *ET-trees* não suportam operações sobre caminhos).

- `HDT`: implementação do algoritmo polilogarítmico de Holm et al. [30, 31].

Os algoritmos acima foram submetidos a quatro classes de experimentos, que procuraram analisar os diversos problemas envolvidos na atualização dinâmica de uma AGM.

Para o primeiro experimento foram consideradas entradas aleatórias de 2000 vértices e 2000 a 95000 arestas, com 20000 atualizações também aleatórias – 10000 inserções e 10000 remoções. Os algoritmos `ST` e `ET+ST` foram consideravelmente mais rápidos que os outros dois. O algoritmo `Spars(I-Mod)` (resp. `HDT`) obteve tempos de CPU em torno de quatorze vezes (resp. três vezes) maiores que os dois algoritmos mais rápidos (`ST` e `ET+ST`).

Por fim, foi realizado um experimento onde aproximadamente 90% das inserções e remoções de arestas resultariam em alterações na AGM. Nesses testes os tempos dos algoritmos aumentaram cerca de dez vezes, resultado das freqüentes alterações na estrutura da árvore geradora mínima. O único algoritmo que apresentou comportamento diferente sob essa situação foi `ST`, pois remoções de arestas da AGM resultam em seguidas consultas de conectividade na busca por uma aresta substituta àquela removida, penalizando as *ST-trees* devido à complexidade de sua estrutura quando comparada com a estrutura das *ET-trees*.

O segundo experimento foi realizado sobre três conjuntos de instâncias semi-aleatórias. Na primeira delas, onde foram utilizadas pequenas componentes conexas em um grafo com 2000 vértices e apenas 1000 arestas, os algoritmos `ST`, `ET+ST` e `HDT` apresentaram ótimo desempenho se comparados com o algoritmo `Spars(I-Mod)`, pois possuem custos constantes muito menores que o último algoritmo. Outros dois experimentos foram realizados para 2000 e 4000 arestas, também resultando em grafos esparsos mas com componentes conexas maiores. Nessa situação, os algoritmos `ET+ST` e `Spars(I-Mod)` apresentaram bom desempenho perante os outros dois algoritmos. O desempenho de `ST` é muito degradado quando utilizado em grafos assim instanciados.

Também foram propostas instâncias denominadas *k-Clique*, onde são geradas k cliques de tamanho c cada, totalizando $n = kc$ vértices. Estas cliques são conectadas entre si através de $2k$ arestas inter-clique aleatoriamente geradas, o que resulta em uma AGM fortemente conectada por arestas intra-cliques, porém fracamente conectada por arestas inter-cliques, resultando no difícil tratamento de remoções dessas arestas. Foi reportado que os algoritmos `ET+ST` e `ST` não foram competitivos para esta classe de grafos e, portanto, não foram testados. O algoritmo `HDT` é o mais rápido desta classe, pois sua estrutura em camadas consegue se adaptar à estrutura também de camadas formada pelas cliques.

O último experimento deste trabalho é realizado sob grafos especificamente criados para simular casos ruins para o algoritmo `HDT`. Tais casos podem ser formulados através de sucessivas mudanças de nível para as arestas do grafo, que então, ao atingir o nível mais alto da estrutura, são removidas. Nesse experimento o algoritmo `Spars(I-Mod)` foi mais rápido, embora não mais que duas vezes, que o algoritmo `HDT`.

O trabalho de Zaroliagis [48] analisa os dois experimentos acima.

3.4 Análise dos Algoritmos para Atualização de Árvores Geradoras Mínimas

Até então, foram apresentadas estruturas de dados para armazenar e tratar árvores sujeitas a alterações dinâmicas. Também foi apresentada a problemática introduzida por alterações estruturais no grafo a partir do qual uma árvore geradora mínima foi calculada. Por fim, algoritmos para o problema de atualização de árvores geradoras mínimas foram ilustrados e analisados experimentalmente.

Contudo, embora existam duas estruturas de dados (*ET-trees* e *ST-trees*) para árvores dinâmicas que são utilizadas em praticamente todos os algoritmos para atualização de

árvores geradoras mínimas em grafos dinâmicos, não existe na literatura um estudo que compare essas estruturas com outras de implementação direta e, por conseqüência, mais simples.

Assim, foi analisado experimentalmente as estruturas de dados mais comuns em algoritmos dinâmicos para árvores geradoras mínimas. Ainda, como resultado de estudos preliminares, foi necessário criar-se uma nova estrutura de dados para árvores dinâmicas, com a função de responder rapidamente a questões sobre conectividade entre dois nós. Esta estrutura, bem como a análise experimental das implementações mais importantes de estruturas de dados para árvores dinâmicas, são apresentadas no Capítulo 4.

Por outro lado, quanto aos algoritmos para atualização de AGMs após inserções e remoções de arestas, análises experimentais mostraram que o algoritmo HDT não possui desempenho favorável na prática, ainda que sua análise teórica afirme o contrário. Seu fraco desempenho é resultado direto de sua difícil implementação, onde são necessárias camadas de estruturas que degradam sensivelmente seu comportamento na prática.

Ainda que suas implementações fossem eficientes, utilizar algoritmos para atualização de AGMs após inserções e remoções de arestas em problemas de modificações em custos de arestas resultaria em desempenho ainda mais prejudicado, já que cada atualização traria a necessidade da execução de duas operações: uma remoção seguida de uma inserção.

Deste modo, observou-se a necessidade de novos algoritmos que sejam, ao mesmo tempo, rápidos e de simples implementação, e ainda específicos para alterações dinâmicas nos custos de arestas. Para isso, um novo algoritmo será proposto e analisado, teorica e experimentalmente, no Capítulo 5 desta dissertação.

3.5 Resumo do Capítulo

Este capítulo apresentou os principais tópicos que cobrem o estudo de algoritmos e estruturas de dados para atualização de árvores geradoras mínimas em grafos dinâmicos. Foi discutida a problemática causada por alterações dinâmicas no grafo, e em seguida foram apresentados os algoritmos e estruturas de dados propostos pela literatura, bem com suas análises experimentais.

Por fim, foram discutidas as principais carências da literatura no que diz respeito a estruturas de dados para árvores dinâmicas e a algoritmos eficientes para atualização de árvores geradoras mínimas em grafos sujeitos a alterações dinâmicas em custos de arestas.

Conseqüentemente, esta dissertação traz, nos Capítulos 4 e 5, alternativas que procuram resolver, respectivamente, ambos os problemas.

Capítulo 4

Uma Estrutura Dinâmica para Consultas Rápidas sobre Conectividade

Este capítulo apresenta a estrutura de dados *DRD-trees* (*doubly-linked reversed dynamic trees*), que é uma contribuição desta dissertação. Essa estrutura de dados foi projetada para responder questões específicas sobre conectividade em tempo constante, quando amortizado sobre o mínimo de n operações.

4.1 *DRD-trees*: Uma Proposta de Árvores Dinâmicas

As árvores dinâmicas são essenciais para a manutenção de árvores geradoras mínimas sujeitas a alterações estruturais freqüentes, o que ocorre em problemas que envolvem grafos dinâmicos. Embora todas as implementações eficientes de árvores dinâmicas possam responder questões sobre conectividade em tempo logarítmico, foi observada experimentalmente uma notável sobrecarga nessa operação. Visando melhorar os tempos computacionais dessa operação, propõe-se uma nova estrutura para árvores dinâmicas, baseada nas *RD-trees*, denominada *DRD-trees* – árvores dinâmicas duplamente encadeadas.

As *DRD-trees* ampliam as *RD-trees* ao estender o conceito de um nó $v \in V$. Na segunda estrutura o nó v armazena apenas as informações $\pi[v]$ (seu pai na árvore dinâmica) e $\omega[v]$ (custo do arco $(v, \pi[v])$). Na primeira estrutura, o nó v passa a armazenar, além desses dois dados, a lista $S(v) = \{x_1, x_2, \ldots\} \quad \forall \quad x_i \in V$ tal que $\pi[x_i] = v$. Assim, são representados também os filhos de v, justificando a nomenclatura de árvore duplamente encadeada.

Armazenar os filhos de um nó permite a realização de buscas em profundidade a partir de um dado nó $v \in V$, o que pode ser utilizado para a realização de consultas

rápidas, em tempo constante amortizado, sobre conectividade entre dois nós (Seção 4.2). Na Figura 4.1 (a), uma AGM é mapeada através de *DRD-trees* (foi omitida a lista de filhos de cada nó); em (b) a aresta (b, f) é removida da AGM; e em (c) é mostrado o resultado da busca em largura realizada a partir do nó f, abrangendo, além de f, os nós d e g (filhos de f na estrutura de dados).

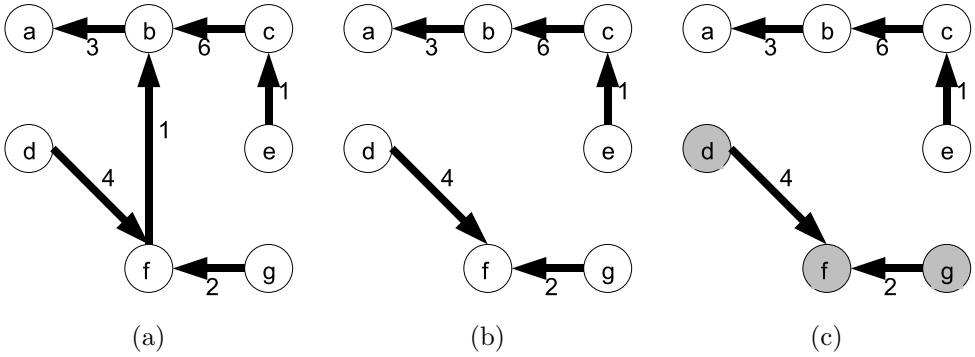


Figura 4.1: AGM representada através de *DRD-trees*.

As operações necessárias para árvores dinâmicas [41] são assim implementadas nas *DRD-trees* aqui propostas:

- `Link`(u, v, c): é implementada de maneira idêntica às *RD-trees*, com exceção da operação $S(u) \leftarrow S(u) \cup \{v\}$, que adiciona o vértice v à lista dos filhos de u. Todas as operações tem tempo constante.

- `Cut`(u, v): além da atualização de $\pi[u]$, deve-se remover o nó u de $S(v)$. Embora a atualização de π seja constante, remover o nó u da lista $S(v)$ tem complexidade linear em n no pior caso.

- `Root`(v): esta operação não é diferente das *RD-trees*; basta subir na árvore até que a raiz da mesma seja encontrada. Sua complexidade é $O(n)$ no pior caso.

- `Cost`(u, v): retorna o custo armazenado em $\omega[u]$ (resp. $\omega[v]$) se $\pi[u] = v$ (resp. $\pi[v] = u$), com complexidade $O(1)$. Caso nenhuma dessas condições seja satisfeita, esta função retorna um valor especial representando a ausência da aresta (u, v).

- `Find_min`(v): sobe na árvore até encontrar sua raiz, armazenando a aresta de menor custo durante a subida. A complexidade deste método é $O(n)$.

- `Update`(v, c): muito parecido com o método `Find_min`(v), onde na subida é adicionado o valor c ao custo de cada aresta percorrida.

- `Evert(`v`)`: implementado pelo Algoritmo 5.

Algoritmo 5 `Evert(`v`)` para estruturas de dados *DRD-trees*

Entrada: O novo nó raiz v e a árvore $T = (V, E')$.

1: $p \leftarrow \pi[v]$;
2: **se** $p = 0$ **então**
3: **retorne** 0;
4: **fim se**
5: `Evert(`p`)`;
6: $\pi[p] \leftarrow v$;
7: $\pi[v] \leftarrow 0$;
8: $S(p) \leftarrow S(p) - \{v\}$;
9: $S(v) \leftarrow S(v) \cup \{p\}$;
10: $\omega[p] \leftarrow \omega[v]$;

A execução do Algoritmo 5 também é recursiva. O caso base da recursão ocorre quando um nó raiz p é atingido, para então as linhas $6-10$ transformarem o nó v na nova raiz da árvore. O nó raiz p é então adicionado na lista dos filhos de v e, ao fim da pilha de recursão, o nó v da primeira chamada recursiva é transformado em nó raiz. Deve-se notar que a operação da linha 8 pode ter custo linear.

A complexidade do Algoritmo 5 é de $O(n)$ no pior caso. Se a árvore transformada é linear, são executadas $O(n)$ chamadas de custo constante – nesse caso a busca na lista de filhos para árvores lineares é executada em tempo constante.

4.2 Consultas de Conectividade em Tempo Constante

O fato das *DRD-trees* serem implementadas sobre árvores duplamente encadeadas pode ser utilizado no desenvolvimento de um algoritmo para avaliação da função `Connected` com complexidade $O(1)$, quando amortizado sobre n operações. Se uma operação de busca em largura é previamente executada, com custo $O(n)$, pode-se construir uma tabela de consultas sob a forma de um vetor com n posições. A necessidade da busca em largura será ilustrada de maneira *top-down*, partindo da necessidade de eficiência para consultas de conectividade, conforme foi salientado no trabalho experimental de Cattaneo et al. [10].

As consultas `Connected(`x, y`)` são executadas todas as vezes que a AGM $T = (V, E')$ é desconectada pela remoção de uma aresta. Essas consultas são feitas sobre as arestas $(x, y) \notin E'$ e demandam um grande esforço computacional, já que cada uma custa $O(\log n)$ em implementações eficientes de árvores dinâmicas. A intuição da busca em largura é a seguinte: considere as árvores T_u e T_v resultantes da remoção da aresta (u, v) de T. Se os

vértices pertencentes a T_u são mapeados em um vetor de tamanho n, é possível verificar se a aresta arbitrária (x, y) conecta as árvores T_u e T_v em $O(1)$ através de duas consultas a esse vetor. Caso o vértice x esteja mapeado no vetor e o vértice y não (ou vice-versa), (x, y) certamente reconecta a árvore geradora mínima T.

A análise amortizada é aqui apresentada através da técnica de contagem [12]. Se, após a operação linear de busca em largura, n operações de tempo constante forem executadas, a complexidade amortizada constante dessas $O(n)$ operações (somando-se aí a operação de busca em largura) pode ser provada pela Equação 4.1.

$$\frac{O(n) + n \cdot O(1)}{n} = \frac{O(n)}{n} = O(1). \tag{4.1}$$

Na Equação 4.1, $O(n)$ é o custo da busca em largura e da alocação das n posições do vetor de consultas e $O(1)$ é o custo de verificar as posições x e y do array para saber se x ou y pertence à árvore T_u.

4.3 Resumo do Capítulo

Neste capítulo foi proposta uma estrutura de dados capaz de executar consultas sobre conectividade em tempo constante, se amortizadas em seqüências de $O(n)$ consultas. A estrutura apresenta desempenho linear para operações relacionadas a caminhos e para a operação `Cut`. Para todas as outras operações são oferecidos algoritmos com tempo constante.

Desta forma, a estrutura de dados *DRD-trees* pode ser utilizada em algoritmos para manutenção de árvores geradoras em grafos dinâmicos. O Capítulo 5 propõe um algoritmo de complexidade $O(m)$ para este problema, usando para isso a estrutura *DRD-trees*.

Capítulo 5

Algoritmo para Atualização de AGMs Dinâmicas

Neste capítulo é apresentado um algoritmo *on-line* totalmente dinâmico para o problema de atualização de árvores geradoras mínimas em grafos dinâmicos. Dois casos são tratados separadamente: o caso de decrementos nos custos de arestas é considerado na Seção 5.2 e o caso de incrementos nos custos de arestas é então considerado na Seção 5.3.

5.1 Estrutura de Dados

Os algoritmos apresentados nas Seções 5.2 e 5.3 são coordenados por um algoritmo centralizado responsável também pelo armazenamento do grafo $G = (V, E)$ sujeito a alterações dinâmicas. Esse algoritmo deve permitir acesso a qualquer aresta do grafo em $O(1)$ e que uma aresta seja atualizada através de incrementos ou decrementos em seu custo. Como requisito para os algoritmos que serão apresentados, tais arestas devem ser armazenadas em uma lista duplamente encadeada e mantidas em ordem crescente de custo. Assim, a cada atualização no custo de uma aresta, esta deve ser movimentada através dessa lista de forma a manter a ordem pré-estabelecida.

Para oferecer acesso em tempo constante a qualquer aresta do grafo, foi adotada uma matriz contendo ponteiros para os nós da lista duplamente encadeada contendo as arestas propriamente ditas. Deste modo, a posição $P(i, j)$ dessa matriz armazena um ponteiro p para posição da lista que contém a aresta (i, j). Em seguida, essa lista de arestas pode ser acessada através da notação $A(p)$. Como a lista em questão é duplamente encadeada, as operações $p \leftarrow p + 1$ e $p \leftarrow p - 1$ retornam ponteiros para o nó seguinte e anterior a p, respectivamente.

O funcionamento deste algoritmo centralizado é apresentado no Algoritmo 6.

Algoritmo 6 Atualização de árvores geradoras mínimas após alterações de custos

Entrada: Grafo $G = (V, E)$ e custos w.

1: Construir uma lista duplamente encadeada A contendo as arestas de G;
2: Organizar a lista A em ordem crescente de custos;
3: Construir uma matriz P de ponteiros para os nós de A;
4: Usar A para construir a AGM $T = (V, E')$ pelo método de Kruskal (Algoritmo 1);
5: **enquanto** existem atualizações a serem processadas **faça**
6: $(i, j) \leftarrow$ aresta a ser atualizada;
7: $c \leftarrow$ novo custo da aresta (i, j);
8: $c' \leftarrow w(i, j)$;
9: $w(i, j) \leftarrow c$;
10: $p \leftarrow P(i, j) + 1$;
11: Reordenar a lista A mantendo-a em ordem crescente de custos;
12: $f \leftarrow P(i, j)$;
13: **se** $w(i, j) < c'$ **então**
14: Executar o Algoritmo 7 com os parâmetros T, (i, j) e w;
15: **senão se** $w(i, j) > c'$ **então**
16: Executar o Algoritmo 9 com os parâmetros T, (i, j), p, f e w;
17: **fim se**
18: **fim enquanto**

O Algoritmo 6 é utilizado para manter a lista de arestas ordenada, bem como decidir sobre qual algoritmo utilizar: incremento ou decremento de custos. Para tanto, este algoritmo recebe inicialmente um grafo $G = (V, E)$, o qual é utilizado para a construção da lista duplamente encadeada A. Esta lista é então ordenada e, para cada nó da mesma, um ponteiro é criado em uma matriz P com o objetivo de oferecer acesso em tempo constante para qualquer aresta de A (i.e., quando deseja-se atualizar uma aresta arbitrária (i, j), não é necessário percorrer A para localizar essa aresta; pode-se acessar (i, j) diretamente através de $P(i, j)$). Constrói-se a AGM inicial $T = (V, E')$ através do Algoritmo 1.

O próximo passo é executar a seguinte seqüência enquanto existem atualizações a serem processadas (i.e., arestas no grafo para serem atualizadas): a linha 6 recebe uma aresta (i, j) que será atualizada com o novo valor c para seu custo, que é recebido como entrada na linha 7. Em seguida o custo atual de (i, j) (antes da atualização propriamente dita) é armazenado em c', para então, na linha 9, o valor $w(i, j)$ ser alterado para c. Neste ponto a lista A de arestas pode ter sua ordem violada, o que é corrigido pela linha 11, onde um procedimento de reordenação é executado. Este procedimento é simples e consiste em deslocar a aresta atualizada para frente (caso seu peso tenha sido incrementado) ou para trás (caso seu peso tenha sido decrementado), até que ela seja disposta em uma posição onde a lista A fique ordenada novamente. Antes dessa operação, o algoritmo ainda guarda em p o ponteiro para o nó $A(P(i, j) + 1)$, ou seja, a posição da aresta imediatamente após

(i, j) na lista A (linha 10). Finalmente, a posição de (i, j) após a reordenação é armazenada em f na linha 12, para então se fazer a chamada ao algoritmo que trata o sub-problema de decremento (linha 14) ou incremento (linha 16) em custos de arestas.

As Seções 5.2 e 5.3 apresentam os algoritmos para incremento e decremento em custos de arestas, respectivamente.

5.2 Algoritmo para Decremento no Custo de Arestas

Resolver o caso de decremento em custos de arestas é a parte mais fácil do problema e pode ser resolvido por qualquer implementação de árvores dinâmicas. Basta para isso armazenar a AGM usando uma dessas árvores. Se a aresta decrementada (i, j) não pertence à AGM, basta verificar na árvore dinâmica qual é a aresta de maior custo no ciclo induzido pela inserção dessa aresta. Se a aresta (x, y) retornada possuir custo maior, é necessário removê-la da AGM e então inserir (i, j) como aresta substituta. O Algoritmo 7 contém a implementação desta tarefa.

Algoritmo 7 Decremento no custo de uma aresta

Entrada: AGM $T = (V, E')$ (possivelmente desatualizada), a aresta (i, j) cujo custo foi decrementado e os custos w.

```
1: se (i, j) ∈ E' então
2:    Atualizar o custo de (i, j) em T;
3: senão
4:    se i ≠ Root(i) então
5:       Evert(i);
6:    fim se
7:    (x, y) ← Find_max(j);
8:    se w(x, y) > w(i, j) então
9:       Cut(x, y);
10:      Link(i, j, w(i, j));
11:   fim se
12: fim se
13: retorne a AGM atualizada T;
```

A eficiência desta operação depende totalmente da implementação de árvore dinâmica utilizada para armazenamento da árvore geradora mínima. Em implementações eficientes de árvores dinâmicas, sua complexidade de tempo é $O(\log n)$.

5.3 Algoritmo para Incremento no Custo de Arestas

Dada uma AGM $T = (V, E')$, incrementar o custo de uma aresta arbitrária $(i, j) \in E'$ da árvore geradora mínima é certamente o caso mais complexo a ser tratado, já que existe a possibilidade da existência de uma aresta (x, y) tal que $w(x, y) < w(i, j)$. Nesse caso, deve-se procurar a aresta (x, y) de menor custo conectando as duas árvores disjuntas resultantes da remoção de (i, j).

A não-trivialidade desta tarefa é assim ilustrada: suponha-se que, na AGM $T = (V, E')$, a aresta $(i, j) \in E'$ foi incrementada em k unidades. A remoção dessa aresta deixaria duas componentes desconectadas, a nomear $T_i = (V_i, E'_i)$ e $T_j = (V_j, E'_j)$, com $V = V_i \cup V_j$ e $E' = E'_i \cup \{(i, j)\} \cup E'_j$. Se $|V_i| = \lceil n/2 \rceil$, então $|V_j| = \lfloor n/2 \rfloor$. Assim, poderiam existir $\lceil n/2 \rceil \cdot \lfloor n/2 \rfloor$ arestas a serem analisadas caso o grafo G seja completo, ou seja, $O(n^2)$ arestas [44].

O Algoritmo 8 apresenta um método simples para atualizar uma árvore geradora mínima $T = (V, E')$ após a aresta (i, j) ter seu custo incrementado. De acordo com o Algoritmo 6, p é um ponteiro que representa a aresta seguinte a (i, j) na lista A antes da chamada ao algoritmo de reordenação ($p = P(i, j) + 1$). Após o algoritmo de reordenação ser executado, é então atribuída a f a posição atual da aresta (i, j) em A.

Algoritmo 8 Incremento no custo de uma aresta

Entrada: AGM $T = (V, E')$ (possivelmente desatualizada), a aresta (i, j) cujo custo foi incrementado, os ponteiros p e f para a lista A e os custos w.

```
1: Cut(i, j);
2: enquanto p ≤ f faça
3:    (x, y) ← A(p);
4:    se Root(x) ≠ Root(y) então
5:       Link(x, y, w(x, y));
6:       retorne a árvore geradora mínima atualizada T;
7:    fim se
8:    p ← p + 1;
9: fim enquanto
```

De acordo com o Algoritmo 8, a aresta (i, j) é removida de E' (conjunto de arestas da árvore $T = (V, E')$ possivelmente desatualizado) através da operação `Cut`(i, j), na linha 1. Em seguida, nas linhas 2-9, inicia-se a operação de busca pela aresta que irá substituir (i, j) em E' (como p pode chegar até o nó $A(f)$ contendo (i, j), em último caso a AGM permanece inalterada pela reinserção dessa aresta). Para tanto, é atribuída a (x, y) a aresta que encontra-se na posição p, ou seja, $(x, y) \leftarrow A(p)$. Como será provado mais adiante que a aresta substituta a (i, j) está obrigatoriamente entre as posições p e f da

lista de arestas A, basta realizar uma busca seqüencial em A, partindo de $A(p)$ e limitada a $A(f)$. Esta busca deverá procurar pela aresta de menor custo (x, y) que conecta as árvores T_i e T_j, o que corresponde a procurar pela menor aresta (x, y) tal que o vértice x se encontre em T_i (resp. T_j) e que o vértice y se encontre em T_j (resp. T_i).

A linha 3 atribui a (x, y) a primeira aresta candidata a substituir (i, j) em E'. Na linha 4, a estrutura de dados para árvores dinâmicas que armazena T é acessada de modo a responder se (x, y) conecta as duas árvores disjuntas T_i e T_j que resultaram da remoção de (i, j) na linha 1. Enquanto esta aresta não for encontrada, as linhas 8 e 3 respectivamente caminham com o ponteiro p para a próxima aresta da lista A e a atribuem a (x, y), aresta que será então analisada na próxima iteração, até que (x, y) satisfaça a condição de conectar T_i a T_j. Nesse ponto, $P(x, y)$ encontra-se ou no intervalo $[p, f)$ (caso em que a AGM será alterada) ou na posição f (caso em que a AGM não mudará). A consolidação dessa busca (atualização de T) é realizada nas linhas 5 e 6, com a execução da operação `Link`$(x, y, w(x, y))$ e, por fim, com o término do algoritmo, retornando a AGM atualizada.

Teorema 5.1 *O Algoritmo 8 atualiza corretamente a AGM T.*

Prova A aresta (i, j) sofreu um incremento de custo. O primeiro passo é provar que a aresta de menor custo (x, y) conectando T_i a T_j não pode estar antes da aresta $A(p)$ na lista ordenada de arestas. Suponha-se que $A(k)$ corresponde à aresta (x, y) para qualquer $k \in \{1, \ldots, p-1\}$. Nesse caso, a AGM antes da atualização de (i, j) não seria mínima, já que $w(x, y) < w(i, j)$ (as arestas onde $w(x, y) = w(i, j)$ não foram incluídas na árvore por conflitarem (i.e., causarem a formação de ciclos em T) com outras arestas que também se encontram entre as posições 1 e $p-1$ da lista A. Também é necessário provar que (x, y) não pode estar depois de $A(f)$. Suponha-se que $A(k)$ seja a aresta (x, y) para algum $k \in \{f+1, \ldots, m\}$. Nesse caso, a nova AGM não poderia ser ótima, já que $w(i, j) < w(x, y)$ (exceto para os casos onde $w(i, j) = w(x, y)$, aos quais o próximo parágrafo contém uma restrição).

Também é necessário provar que o algoritmo acima mantém a política do algoritmo de Kruskal [33], onde a primeira aresta em ordem de custo que conecte duas sub-árvores disjuntas deve ser escolhida para fazer parte da AGM T. Esse caso é de suma importância em grafos onde existem arestas de mesmo custo. Para isso, quando uma aresta tem seu custo incrementado, o método para reordenação de arestas não precisa se preocupar em posicionar a aresta dentre aquelas com mesmo custo, já que o Algoritmo 8 fará a seleção baseada no método de Kruskal. Contudo, se a aresta (i, j) tem seu custo decrementado,

deve-se atentar ao procedimento adotado pelo Algoritmo 7, que somente modifica a AGM quando $w(x, y) > w(i, j)$ (ou seja, quando há empates a AGM não é alterada). Com esse procedimento, o método de reordenação deve deslocar a aresta (i, j) para a esquerda até que encontre a primeira aresta com custo igual ao dela. Desta forma, é garantido que, se (x, y) tiver seu custo incrementado futuramente, (i, j) seria uma aresta substituta no Algoritmo 8.

Teorema 5.2 *O Algoritmo 8 atualiza uma árvore geradora mínima em tempo* $O(m \log n)$ *(resp.* $O(mn)$*) se* T *for representada através de* ST-trees *(resp.* RD-trees*).*

Prova Note que a lista A já se encontra ordenada. Desta forma, o pior caso ocorre quando a aresta (i, j) é deslocada da primeira até a última posição de A e ainda não existe uma aresta substituta. Neste caso, cada chamada ao método `Connected` tem complexidade $O(\log n)$ (resp. $O(n)$), resultando no tempo total de $O(m \log n)$ (resp. $O(mn)$) se a árvore for representada por *ST-trees* (resp. *RD-trees*).

Os algoritmos acima são simples de implementar e possuem baixas constantes de implementação. Experimentos preliminares comprovaram sua viabilidade quando comparados aos métodos estáticos de Kruskal [33] e Prim [37], mesmo todos eles possuindo a mesma complexidade assintótica. Entretanto, foi observado que as chamadas de conectividade dominaram a utilização de tempo dentro do algoritmo, além do fato que cortes e ligações desnecessárias são executados toda vez que uma aresta é incrementada e não existem arestas substitutas no intervalo $[p, f)$ (ou seja, a AGM permaneceu inalterada após a operação de incremento).

Tendo em mente as duas dificuldades citadas acima, é proposto o Algoritmo 9 que utiliza as *DRD-trees* propostas no Capítulo 4. A utilização dessa estrutura por si só já elimina a necessidade de cortes e ligações desnecessárias, já que a busca em profundidade não requer que as árvores sejam desconectadas, ao contrário das outras estruturas dinâmicas onde as raízes de T_i e T_j são pesquisadas a cada aresta verificada.

Nesse algoritmo, as principais alterações com relação ao Algoritmo 8 se encontram nas linhas 1-5, 6 e 9-10. Nas linhas 1-5 é executada a busca em profundidade. Assim, o nó i ou j não é incluído na busca, permitindo que a tabela de consultas mapeie apenas uma das sub-árvores T_i ou T_j. Com isso, a linha 6 limita a busca de forma que a aresta $A(f) = (i, j)$ não precise ser verificada, já que ela não foi removida da AGM. Por fim, as linhas 9-10 são executadas quando uma aresta $(x, y) \neq (i, j)$ é encontrada, removendo a aresta incrementada e adicionando a aresta (x, y).

Algoritmo 9 Incremento no custo de uma aresta utilizando *DRD-trees*

Entrada: AGM $T = (V, E')$ (possivelmente desatualizada), a aresta (i, j) cujo custo foi incrementado, os ponteiros p e f para a lista A e os custos w.

1: **se** $i = parent(j)$ **então**
2: Executar uma busca em profundidade a partir do nó j, armazenando cada nó visitado no vetor de consultas δ com o valor `verdadeiro`. Para os outros nós, δ deve conter o valor `falso`;
3: **senão**
4: Executar uma busca em profundidade a partir do nó i, armazenando cada nó visitado no vetor de consultas δ com o valor `verdadeiro`. Para os outros nós, δ deve conter o valor `falso`;
5: **fim se**
6: **enquanto** $p \neq f$ **faça**
7: $(x, y) \leftarrow A(p)$;
8: **se** $\delta[x] \neq \delta[y]$ **então**
9: `Cut`(i, j);
10: `Link`$(x, y, w(x, y))$;
11: **retorne** a árvore geradora mínima atualizada T;
12: **fim se**
13: $p \leftarrow p + 1$;
14: **fim enquanto**
{Neste ponto, a estrutura da AGM não será alterada, mas seu custo sim.}
15: **se** $i = parent(j)$ **então**
16: $\omega[j] \leftarrow w(i, j)$;
17: **senão**
18: $\omega[i] \leftarrow w(i, j)$;
19: **fim se**
20: **retorne** a árvore geradora mínima atualizada T;

O Algoritmo 9 tem sua corretude mantida pelo Teorema 5.1, mas sua complexidade é ligeiramente diminuída, conforme o Teorema 5.3.

Teorema 5.3 *O Algoritmo 9 utiliza tempo $O(m)$ para atualizar a árvore geradora mínima T.*

Prova A complexidade para iniciar o vetor de consultas é de $\Theta(n)$, já que, embora no máximo $n-1$ nós serão visitados, exatamente n posições de memória terão de ser alocadas. Como a reordenação das arestas é feita em $O(m)$, e as $O(m)$ consultas sobre conectividade podem ser executadas em $O(1)$ cada, chega-se à complexidade de $O(m)$ para atualizar a AGM através do Algoritmo 9 quando considerado o pior caso.

5.4 Resumo do Capítulo

Este capítulo propôs um algoritmo para a atualização de uma árvore geradora mínima de um grafo sujeito a alterações dinâmicas nos custos de suas arestas. O algoritmo proposto é flexível, podendo ser utilizado para atualizar AGMs armazenadas sob qualquer estrutura de dados para representação de árvores dinâmicas. Embora tenha complexidade teórica superior se comparada à complexidade do algoritmo polilogarítmico de Holm et al. [31], espera-se deste algoritmo um bom desempenho prático, já que é baseado em estruturas de dados simples (a citar as *DRD-trees* propostas no Capítulo 4).

Capítulo 6

Análise Experimental

Foram realizados diversos experimentos para verificar a aplicabilidade dos algoritmos e estruturas de dados propostos nos capítulos anteriores. Este capítulo foi dividido em duas seções, que respectivamente analisam o desempenho da estrutura de dados *DRD-trees* (Seção 6.1) e dos algoritmos para atualização de árvores geradoras mínimas em grafos dinâmicos (Seção 6.2).

Para todos os experimentos foi utilizado um computador PC Intel Pentium 4 HT com freqüência de 3,2 GHz, memória física de 512 MBytes e sistema operacional Slackware GNU/Linux versão 10.2, com *kernel* Linux e compilador GNU g++. As versões de kernel e compilador foram, respectivamente, 2.6.16 e 3.4.6. Os algoritmos e estruturas de dados foram codificados em linguagem C++ padrão ANSI/ISO e compilados através do comando `g++ -O3 -mcpu=pentium4`, que otimiza o código binário gerado. Não foram permitidos acessos à memória secundária (*swap*) através do comando `swapoff`, existente no sistema operacional GNU/Linux. Todos os resultados foram obtidos a partir de seqüências de atualizações distintas, geradas a partir de dez sementes diferentes. Os tempos foram contabilizados em segundos e representam a execução de todas as operações de atualização às quais os algoritmos foram submetidos.

6.1 Estruturas de Dados: Árvores Dinâmicas

Esta seção procurou comparar a estrutura de dados *DRD-trees*, proposta no Capítulo 4, com as estruturas *RD-trees*, *ST-trees* e *ET-trees* (detalhadas na Tabela 6.1). Para isso, foram realizados os experimentos apresentados na Tabela 6.2.

A Tabela 6.1 apresenta as estruturas de dados avaliadas nessa seção. Deve-se observar que a estrutura `DRD-C` (*DRD-trees* com consultas rápidas) depende da execução prévia

Tabela 6.1: Estruturas de dados analisadas experimentalmente.

Estruturas de dados	Denominação	Operação de conectividade	Complexidade
RD-trees	`RD`	`Root(`u`)` $\neq$ `Root(`v`)`	$O(n)$
DRD-trees	`DRD-R`	`Root(`u`)` $\neq$ `Root(`v`)`	$O(n)$
DRD-trees	`DRD-C`	`Connected(`u, v`)`	$O(1)$
ST-trees	`ST`	`Root(`u`)` $\neq$ `Root(`v`)`	$O(\log n)$
ET-trees	`ET`	`Root(`u`)` $\neq$ `Root(`v`)`	$O(\log n)$

da operação de busca em profundidade (com complexidade $O(n)$), o que irá garantir complexidade constante quando amortizada sobre n chamadas à função `Connected`. Por outro lado, no caso da estrutura `DRD-R`, a operação de conectividade deve buscar pela raiz da árvore contendo o dado vértice.

Tabela 6.2: Experimentos sobre estruturas de dados para árvores dinâmicas.

Experimento	Descrição	Objetivo
`CAA`	Conectividade em árvores aleatórias	Desempenho no caso médio
`CAL`	Conectividade em árvores lineares	Desempenho no pior caso
`CAB`	Conectividade em árvores balanceadas	*Overhead* das estruturas
`UAA`	Conectividade em árvores aleatórias	*Overhead* das *DRD-trees*
`LAA`	*Link* e *cut* em árvores aleatórias	Desempenho geral

Na Tabela 6.2 encontram-se os experimentos realizados sobre as estruturas de dados da Tabela 6.1. Todos os experimentos foram executados em árvores com número de nós $n \in \{2000, 4000, \ldots, 400000\}$. Os vértices e arestas escolhidos para as operações acima foram aleatoriamente selecionados.

Para os quatro primeiros experimentos, procurou-se observar o comportamento das estruturas de dados quando submetidas a uma seqüencia de $k = 100000$ consultas de conectividade sobre os vértices i e j escolhidos aleatoriamente, para assim precisar seu desempenho em casos onde a árvore é aleatória, linear ou balanceada. Para árvores aleatórias são geradas $n - 1$ arestas de maneira aleatória, sem que hajam ciclos. Já árvores lineares são construídas através das arestas $(1, 2), (2, 3), ..., (n - 1, n)$. Por fim, árvores balanceadas são árvores completas, ou seja, de altura logarítmica.

Os resultados para os experimentos `CAA`, `CAL` e `CAB` são apresentados nas Figuras 6.1, 6.2 e 6.3, respectivamente.

A Figura 6.1 traz o desempenho em caso médio da operação para verificação de conecti-

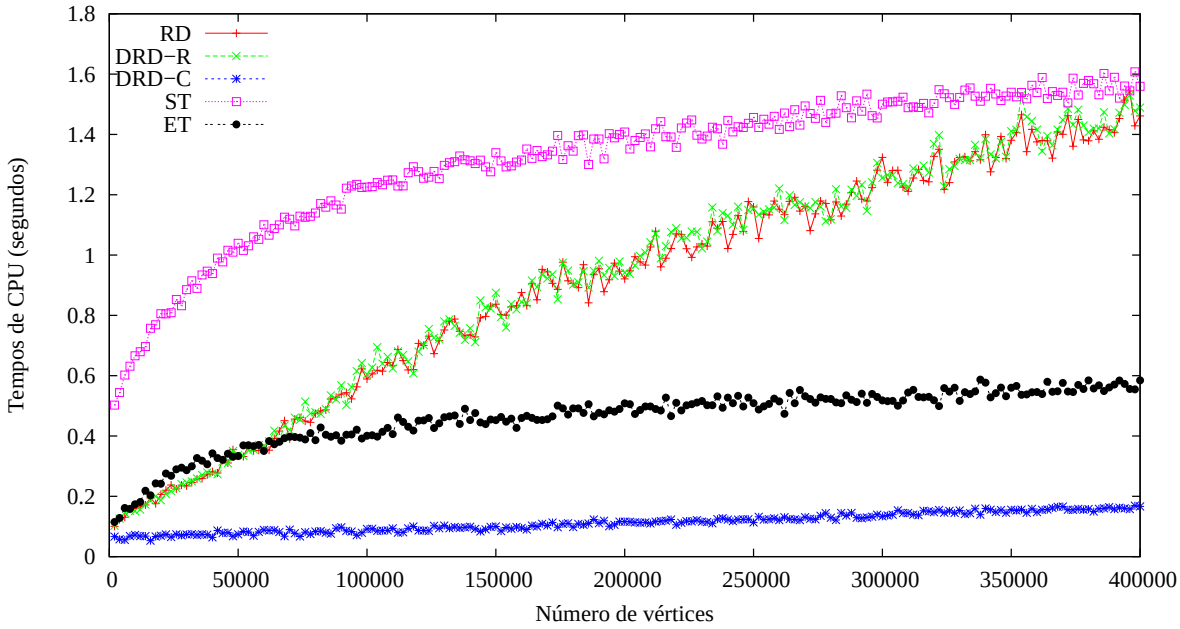


Figura 6.1: Tempos de CPU para a execução de 100000 consultas de conectividade em árvores aleatórias.

vidade das estruturas de dados analisadas. Nessa figura, observa-se o comportamento praticamente constante da estrutura `DRD-C`, em conformidade com sua complexidade teórica. Este fato também pode ser verificado em todas as outras estruturas de dados. Aquelas de complexidade linear, nominalmente `RD` e `DRD-R`, possuem curvas praticamente idênticas, o que é facilmente justificável: ambas estruturas possuem implementações idênticas da operação `Root`. Já as estruturas de complexidade logarítmica apresentam desempenho diferenciado: a estrutura `ET`, uma implementação da estrutura de dados *ET-trees*, realiza consultas de conectividade com tempos visivelmente mais baixos que a implementação da estrutura de dados *ST-trees*. Cabe ainda a comparação entre as estruturas de complexidade linear com aquelas de complexidade logarítmica. Em comparação com a estrutura *ET-trees*, as estruturas lineares são mais rápidas em árvores com até aproximadamente 50000 nós. Já com a estrutura de dados *ST-trees*, o cruzamento das curvas de desempenho ocorre próximo aos 500000 nós (não se encontra no gráfico).

Na Figura 6.2 são apresentados os resultados para árvores de altura linear. Tais árvores representam o pior caso para as estruturas de dados *RD-trees* e *DRD-trees* (nessa última, quando é utilizada a consulta através da operação `Root`). No gráfico apresentado, pode-se verificar que esse pior caso realmente ocorre na prática. Todas as outras estruturas de dados mantiveram o comportamento esperado (em conformidade com a teoria) quando utilizadas para representar árvores com altura linear.

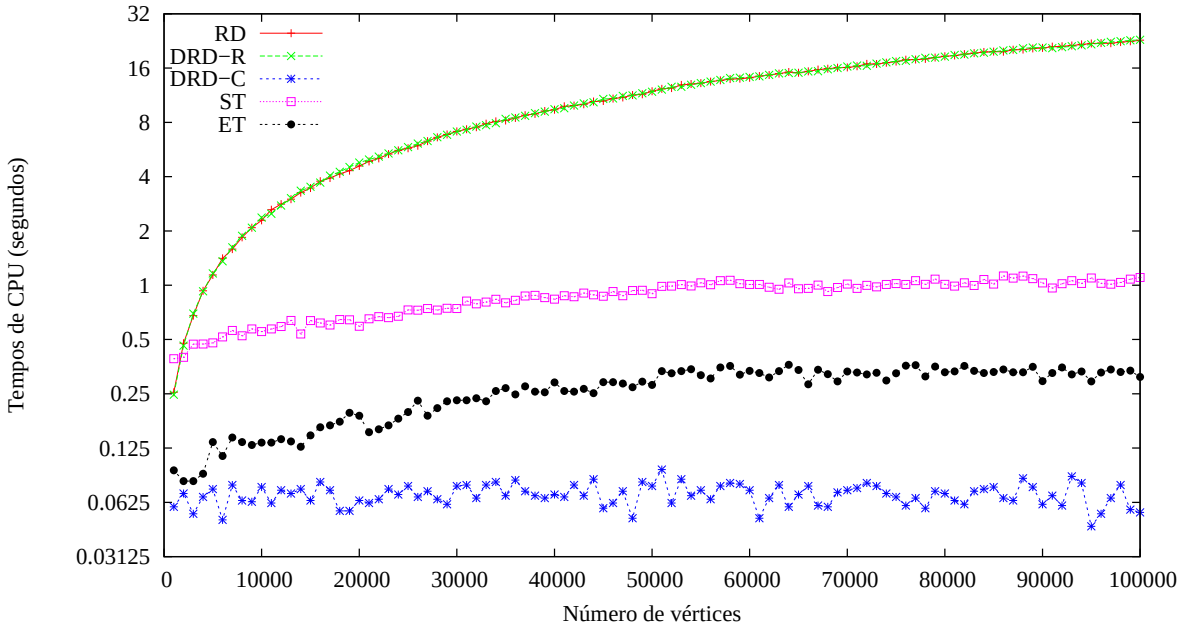


Figura 6.2: Tempos de CPU para a execução de 100000 consultas de conectividade em árvores lineares.

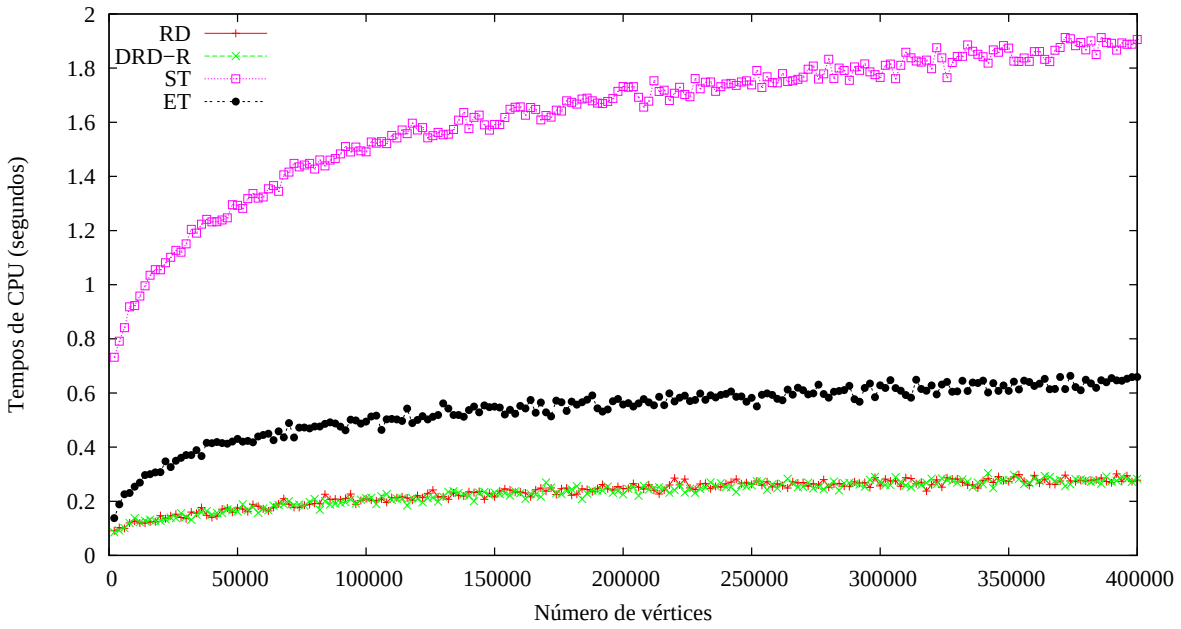


Figura 6.3: Tempos de CPU para a execução de 100000 consultas de conectividade em árvores balanceadas.

O experimento cujos resultados são apresentados na Figura 6.3 faz uma análise da sobrecarga introduzida pelas estruturas de dados projetadas para garantir complexidades logarítmicas, analisando para isso o melhor caso de entrada para as estruturas de dados. Esse experimento consistiu em realizar consultas de conectividade em árvores completas,

ou seja, de altura logarítmica, representando então o melhor caso de entrada para as estruturas. Nesse cenário, todas as estruturas de dados, com exceção da `DRD-C` (essas continuam com a operação de conectividade executada em tempo constante), passam a oferecer operações com complexidades logarítmicas (para as estruturas `DRD-R` e `RD`, subir em uma árvore tem complexidade $O(\log n)$ se a altura desta é logarítmica). Com isso, a estrutura `RD` é utilizada como base dessa análise por ser uma implementação direta de árvores dinâmicas. Pode-se observar através da Figura 6.3 que, dentre as estruturas com operações de complexidade logarítmica, as *ET-trees* possuem a menor sobrecarga sobre sua estrutura, comprovando assim o que já foi analisado teoricamente no Capítulo 3.

Cabe aqui ressaltar que todas as estruturas se comportaram perfeitamente de acordo com suas análises teóricas. As estruturas `RD` e `DRD-R` possuem curvas lineares em todos os casos, com exceção das instâncias de árvores com altura logarítmica. Por outro lado, as estruturas `ET` e `ST` se comportam sempre de maneira logarítmica, o mesmo acontecendo com a estrutura `DRD-C`, desta vez apresentando comportamento constante.

Para completar os experimentos de conectividade, faz-se necessária a verificação do impacto da busca em profundidade na estrutura de dados `DRD-C`. Para isso foi desenvolvido o seguinte ambiente para testes, tendo como base a árvore $T = (V, E'), n = 500000$ e o número de consultas fixado em $k = 100000$: a cada $l = 1000, 2000, \ldots, 100000$ consultas realizadas, uma aresta $(x, y) \in T$ é aleatoriamente escolhida para ser removida da árvore ou uma aresta $(x, y) \notin T$ é aleatoriamente escolhida para ser inserida em T, desde que (x, y) não introduza ciclos nessa árvore. Embora os tempos dessas operações não tenham sido contabilizados, a tabela de consultas da estrutura *DRD-trees* teve de ser recalculada. Assim, este experimento analisa a sobrecarga introduzida pelas freqüentes chamadas à operação de busca em profundidade nas *DRD-trees*. Note que a complexidade anunciada pela Equação 4.1 será violada, já que o argumento de amortização não será mais válido.

O comportamento da estrutura `DRD-C` na Figura 6.4 é compatível com a seguinte análise: se uma busca em profundidade é realizada a cada l operações, então a complexidade do método é de $O((n + l \cdot O(1))/l) = O(n/l)$. Como l inicialmente vale 1000 e n é fixado em 500000, a complexidade nesse caso é $O(n)$, já que $n \gg l$. Conforme $l \to n$, a complexidade das operações tendem a $O(1)$. Isto explica o comportamento constante do algoritmo quando $l > 50000$, ou seja, o valor de l chegou ao limiar do impacto causado pela busca em profundidade nas consultas de conectividade.

Finalizando a seção de experimentos em árvores dinâmicas, a Figura 6.5 apresenta os resultados para seqüências de *links* e *cuts*. Esses resultados se mostram condizentes

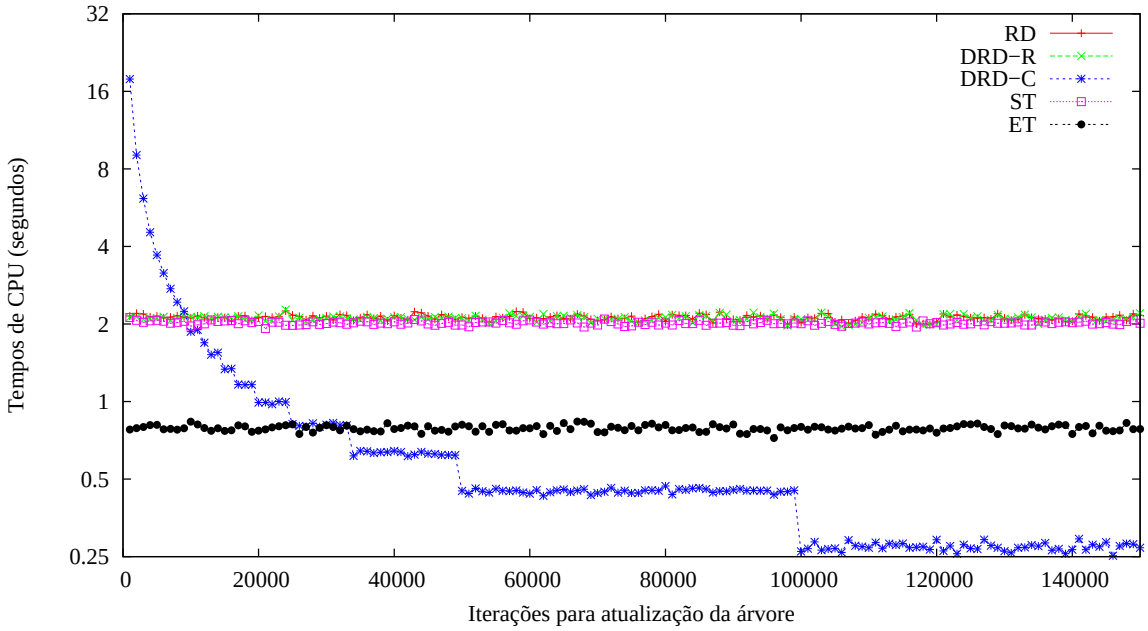


Figura 6.4: Tempos de CPU para a execução de 100000 consultas de conectividade em árvores aleatórias sujeitas alterações estruturais.

com as análises teóricas. Para a estrutura de dados `RD`, operações de *link* e *cut* tiveram complexidades lineares, visto que certas arestas só podem ser inseridas mediante uma operação de `Evert`. Por outro lado, a estrutura `DRD-R` (aqui representando as *DRD-trees*) foi aquela que apresentou a maior complexidade nesse experimento, já que existe o custo de se procurar por nós dentro da lista de filhos de um nó quando se executam operações sobre essa estrutura. Por fim, as duas outras estruturas se comportaram logaritmicamente, de acordo com as perspectivas estabelecidas pela análise teórica.

6.2 Algoritmos: Atualização de Árvores Geradoras Mínimas em Grafos Dinâmicos

Nesta seção é realizada a análise experimental dos algoritmos para atualização da árvore geradora mínima de um grafo dinâmico. Para tanto, foram utilizados dois conjuntos de instâncias de forma a compatibilizar os resultados deste trabalho com aqueles apresentados em [4, 10] (Seção 6.2.1) e ao mesmo tempo oferecer uma análise mais abrangente, a partir de instâncias de problemas reais e também de instâncias com grande número de vértices e arestas (Seção 6.2.2). Os algoritmos analisados são apresentados na Tabela 6.3.

Os algoritmos do grupo `C` foram implementados de acordo com o trabalho [10]: para o grafo $G = (V, E)$ de AGM $T = (V, E')$, uma árvore balanceada do tipo AVL é utilizada

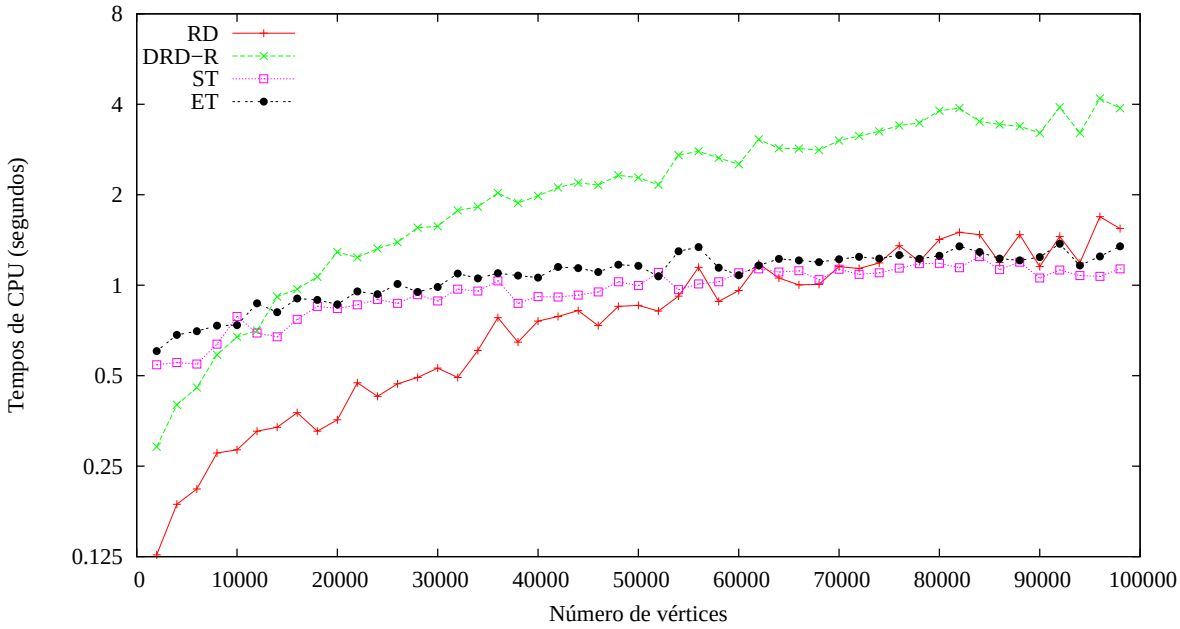


Figura 6.5: Tempos de CPU para a execução de 100000 operações mistas de *link* e *cut*.

Tabela 6.3: Algoritmos analisados experimentalmente.

Grupo	Algoritmo	Referência	Estrutura de dados	Complexidade
`C`	`C(ET+ST)`	Cattaneo et al. [10]	*ET-trees* e *ST-trees*	$O(m \log n)$
	`C-Mod(DRD+ST)`	Modificado de [10]	*DRD-trees* e *ST-trees*	$O(m)$
`RT`	`RT(RD)`	Esta dissertação	*RD-trees*	$O(mn)$
	`RT(DRD)`	Esta dissertação	*DRD-trees*	$O(m)$
	`RT(ST)`	Esta dissertação	*ST-trees*	$O(m \log n)$
	`RT(ET+ST)`	Esta dissertação	*ET-trees* e *ST-trees*	$O(m \log n)$
	`RT(DRD+ST)`	Esta dissertação	*DRD-trees* e *ST-trees*	$O(m)$

para armazenar as arestas de $E - E'$ enquanto duas estruturas de árvores dinâmicas são utilizadas para armazenar T. Quando uma aresta tem seu peso atualizado, duas operações são realizadas: uma remoção, seguida de uma inserção. A primeira operação realiza a remoção da aresta (i, j) e é assim executada: se $(i, j) \in E'$, ela é então removida de E'. Depois disso, uma aresta substituta (x, y) é buscada dentre as arestas da árvore AVL, partindo da aresta de menor custo e percorrendo esta estrutura em ordem. Se toda a AVL foi percorrida e nenhuma aresta conecta as árvores T_i e T_j, então T ficará temporariamente desconectada. A segunda operação corresponde à inserção da aresta (i, j) com o novo peso $w(i, j)$ e é executada da seguinte forma. Verifica-se em T se os vértices i e j encontram-se conectados. Em caso positivo, (i, j) só é adicionada a T se a

aresta (x, y) de maior custo no caminho entre i e j satisfaz a desigualdade $w(x, y) > w(i, j)$. Nesse caso, (x, y) é removida de T através da operação `Cut`(x, y) e (i, j) é adicionada a T através da operação `Link`$(i, j, w(i, j))$. Em caso contrário, (i, j) é adicionada a T através da operação `Link`$(i, j, w(i, j))$.

Os algoritmos do grupo `RT` foram aqueles apresentados no Capítulo 5, onde foram utilizadas diferentes estruturas de dados para representação da árvore geradora mínima.

6.2.1 Experimentos com Instâncias Sintéticas

Esta seção compara os algoritmos e estruturas de dados propostos utilizando o mesmo parâmetro de Cattaneo et al. [10]. Assim, foram executados experimentos apenas sobre grafos sintéticos. Os experimentos foram divididos em *aleatórios*, onde os grafos são aleatoriamente gerados, e *k-Cliques*, que sintetizam o pior caso dos algoritmos aqui implementados através da utilização de grafos contendo k cliques com c nós cada e apenas $2k$ arestas inter-clique (dificultando a busca por arestas substitutas em atualizações aplicadas a arestas inter-clique). A Tabela 6.4 apresenta os experimentos sintéticos realizados neste trabalho.

Tabela 6.4: Experimentos sintéticos realizados. Para o experimento k-Clique, $n = k \cdot c$ e $m = 2 \cdot k + (c \cdot (c-1))/2$, onde k representa o número de cliques e c o número de vértices em cada clique.

Experimento	n	m	Incrementos	Decrementos
Aleatório (1)	1000	$1000 \sim 100000$	10000	10000
Aleatório (2)	2000	$1000 \sim 100000$	10000	10000
Aleatório (3)	4000	$1000 \sim 100000$	10000	10000
k-Clique (1)	2000 $(k = 4, c = 500)$	499008	10000	10000
k-Clique (2)	2000 $(k = 10, c = 200)$	199020	10000	10000
k-Clique (3)	2000 $(k = 20, c = 100)$	99040	10000	10000
k-Clique (4)	2000 $(k = 40, c = 50)$	49080	10000	10000
k-Clique (5)	2000 $(k = 50, c = 40)$	39100	10000	10000
k-Clique (6)	2000 $(k = 100, c = 20)$	19200	10000	10000
k-Clique (7)	2000 $(k = 200, c = 10)$	9400	10000	10000
k-Clique (8)	2000 $(k = 500, c = 4)$	4000	10000	10000

6.2.1.1 Atualizações Aleatórias em Grafos Aleatórios

Com os experimentos Aleatório (1), Aleatório (2) e Aleatório (3), objetiva-se analisar os comportamentos dos algoritmos em um grafo sintético quando variam-se o número de

arestas. Os grafos foram gerados aleatoriamente. Os resultados desses experimentos estão sintetizados nas Figuras 6.6, 6.7 e 6.8.

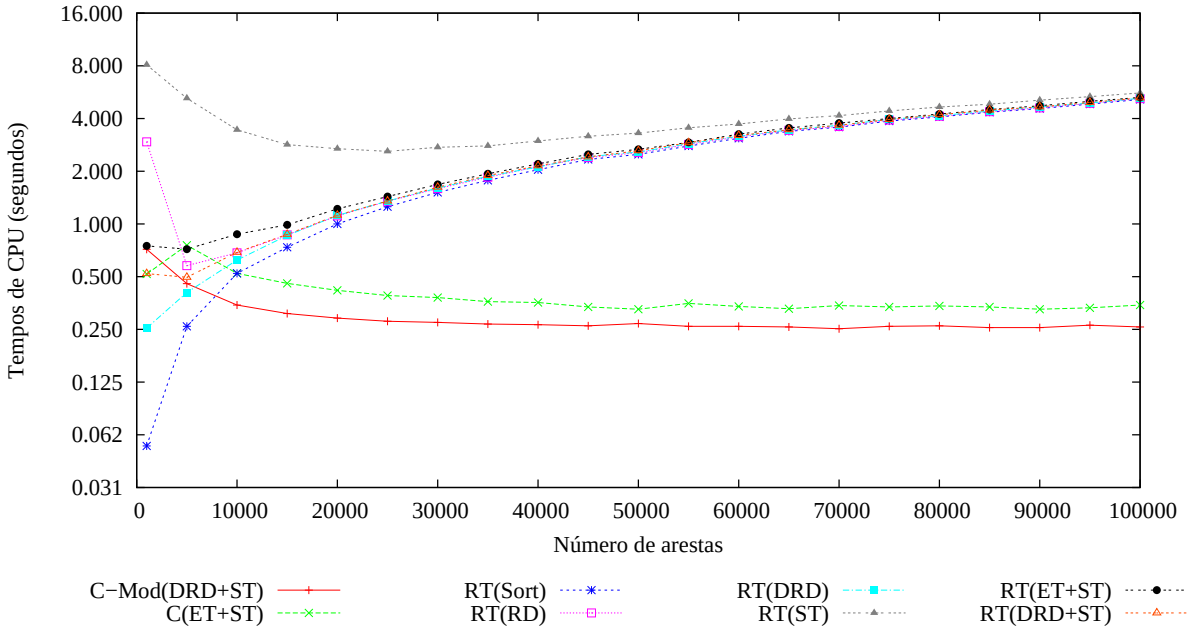


Figura 6.6: Tempos de CPU para a execução de 20000 atualizações aleatórias de custos em grafos aleatórios contendo 1000 vértices.

A Figura 6.6 exibe os resultados dos experimentos em grafos gerados aleatoriamente, sujeitos a seqüências aleatórias de atualizações nos custos de suas arestas. Tais seqüências de atualizações são caracterizadas pela grande quantidade de operações caras em grafos esparsos, correspondendo a incrementos de custos em arestas da AGM. Com isso, todos os algoritmos são obrigados a buscar por uma aresta substituta, de menor peso, a cada atualização, resultando no elevado tempo computacional de todos os métodos quando o grafo é esparso (contendo até 5000 arestas). Por outro lado, conforme o grafo torna-se denso, seqüências caras de atualizações tornam-se raras, pois probabilisticamente incrementam-se custos de arestas que não se encontram na AGM. Isso explica o comportamento dos algoritmos quando o número de arestas varia de 10000 a 100000: para aqueles da classe `RT`, apenas o tempo de reordenação exerce impacto nos resultados; já para aqueles da classe `C`, onde não existe reordenação, seus tempos ficam praticamente estáveis. Dentre as curvas apresentadas será destacado o tempo de CPU que os algoritmos `RT` consomem para reordenar a lista de arestas, sob a denominação `RT(Sort)`. Embora este tempo já se encontre somado nos tempos dos algoritmos propriamente ditos, decidiu-se por contabilizá-lo também separadamente, o que permite uma análise mais abrangente do impacto da reordenação nesses algoritmos.

Decrementos em arestas não pertencentes à AGM não impactam diretamente nos algoritmos, já que seu melhor caso ocorre quando o grafo é esparso. O pior caso para operações de decremento ocorre quando $m \gg n$, resultando em freqüentes decrementos em arestas não pertencentes à AGM (que podem levar a alterações na AGM). Como os tempos dos algoritmos foram maiores quando o grafo é esparso, chega-se a conclusão que as operações que influenciam diretamente na complexidade dos algoritmos são aquelas de incrementos em arestas da AGM.

Na Figura 6.6, a curva `RT(Sort)` indica que a reordenação da lista de arestas tomou praticamente todo o tempo dos algoritmos `RT`. Assim, conclui-se que os algoritmos baseados em [10] são mais eficientes na atualização de AGMs em grafos aleatórios com poucos vértices e muitas arestas. Destaca-se ainda o desempenho do algoritmo híbrido `C-Mod(DRD+ST)`, que é sensivelmente o método mais rápido para as condições aqui analisadas (grafos aleatórios sujeitos a seqüências de atualização geradas aleatoriamente). Em grafos esparsos, onde seqüências caras são processadas, os algoritmos `RT` são mais eficientes.

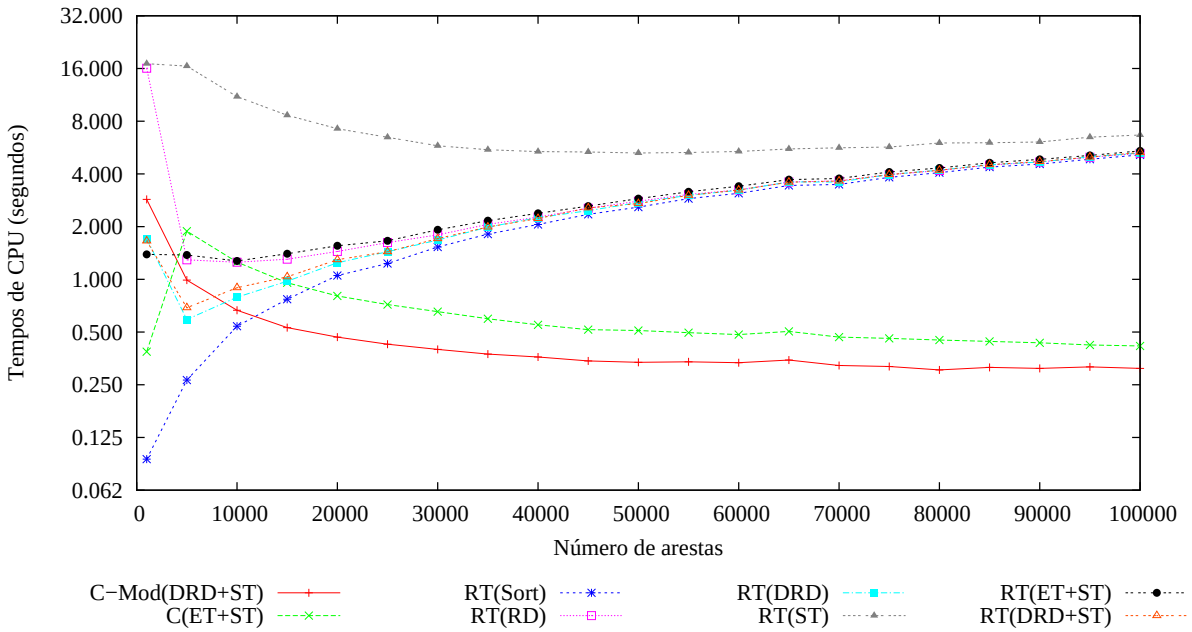


Figura 6.7: Tempos de CPU para a execução de 20000 atualizações aleatórias de custos em grafos aleatórios contendo 2000 vértices.

A Figura 6.7 traz os resultados para grafos com $n = 2000$ e experimentos idênticos aos da Figura 6.6. Embora a ordem e a relevância dos resultados não se alterem, nota-se um deslocamento do ponto de interseção entre as curvas dos algoritmos `RT` com as curvas dos algoritmos `C`.

Com o aumento do número de vértices para $n = 4000$, verifica-se que o desempenho do

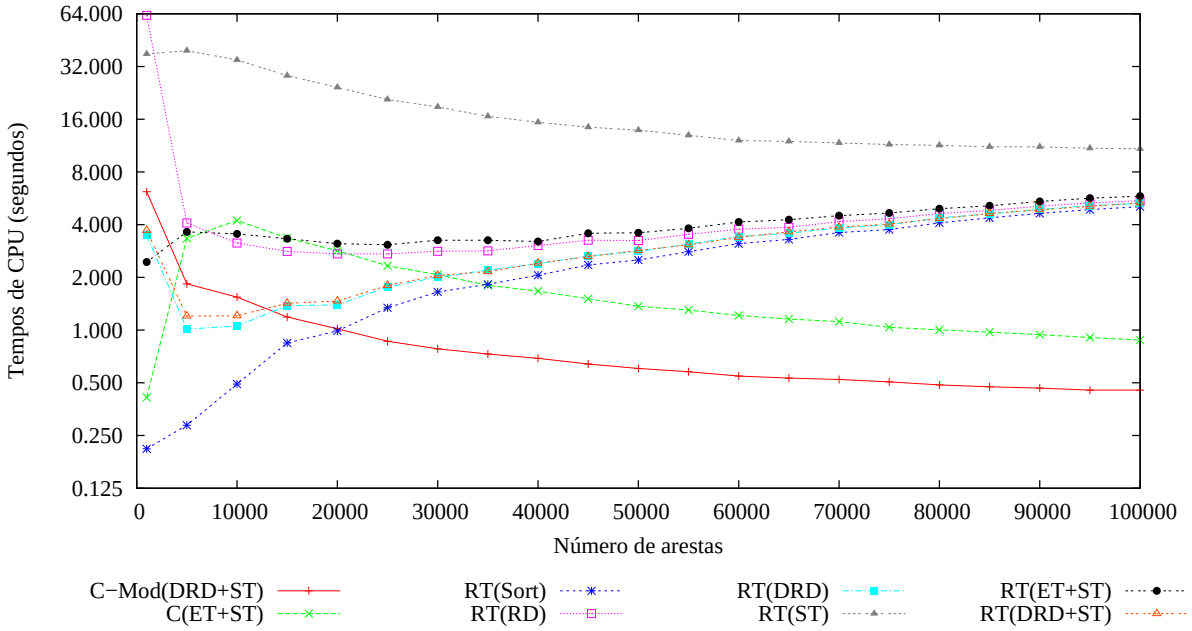


Figura 6.8: Tempos de CPU para a execução de 20000 atualizações aleatórias de custos em grafos aleatórios contendo 4000 vértices.

algoritmo `RT(RD)` é muito ruim quando aplicado a um grafo esparso e conforme aumenta o número de vértices. O deslocamento do ponto de interseção entre as curvas dos algoritmos `RT` e `C` continua, explicado pelo aumento no número de consultas de conectividade realizadas pelos algoritmos `C(ET+ST)` e `C-Mod(DRD+ST)`. Essas consultas, quando cresce o número de vértices, passam a gastar mais tempo de CPU, atingindo especificamente os algoritmos do grupo `C` que, a cada incremento em arestas da AGM, procuram por uma aresta substituta a partir da aresta de menor custo na árvore AVL. Mais eficientes, os algoritmos `RT` iniciam essa busca a partir da aresta incrementada, economizando chamadas ao método `Connected` das árvores dinâmicas.

Concluindo esta seção, seqüências aleatórias de atualizações são caracterizadas por constantes incrementos em arestas do sobconjunto $E - E'$ (tratados em $O(\log n)$ pelos algoritmos do grupo `C`) e decrementos em arestas da AGM, que configuram seqüências fáceis de atualização, um fato que não é interessante para analisar experimentalmente um algoritmo.

6.2.1.2 Atualizações Estruturadas em Grafos Aleatórios

Esta seção pretende melhor desenvolver os experimentos da seção anterior, introduzindo seqüências estruturadas de atualizações. Essas seqüências se dividem em 10% de atuali-

zações aleatórias e 90% de atualizações estruturadas. Em relação às atualizações estruturadas, 90% delas são incrementos em custos de arestas da AGM e 10% são decrementos em custos de arestas que não pertencem à AGM. Com isso, força-se que as atualizações possam alterar a AGM com maior freqüência, resultando em um maior grau de dificuldade para os algoritmos.

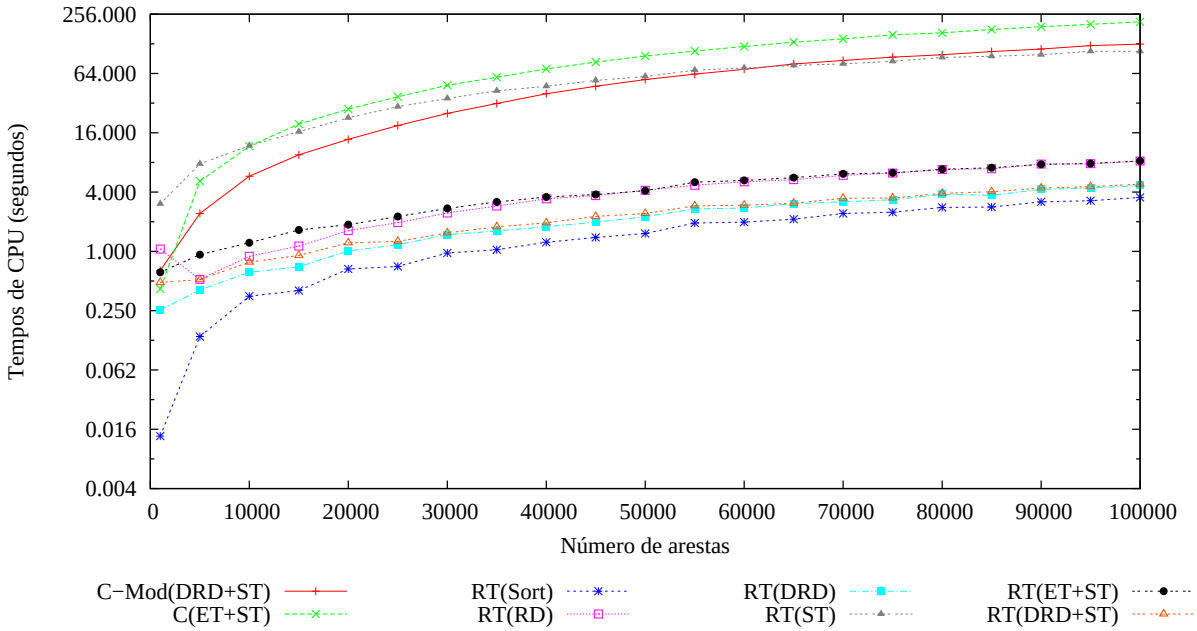


Figura 6.9: Tempos de CPU para a execução de 20000 atualizações estruturadas de custos em grafos aleatórios contendo 1000 vértices.

A Figura 6.9 inverte o panorama apresentado na seção anterior, onde atualizações aleatórias eram processadas. Nesse caso os algoritmos do grupo `RT` possuem desempenho muito melhor que aqueles do grupo `C`. Conforme cresce o número de arestas, os algoritmos que melhor processam atualizações estruturadas são, respectivamente: `RT(DRD)` e `RT(DRD+ST)`. Os algoritmos `RT(ST)`, `C(ET+ST)` e `C-Mod(DRD+ST)` não apresentam um bom desempenho. Nota-se que as curvas dos algoritmos do grupo `RT` encontram-se muito próximas, visto que o número de vértices do experimento é muito pequeno.

Na Figura 6.10 os tempos dos algoritmos começam a se distanciar com a duplicação do número de vértices. Contudo, os resultados não se alteram com relação à Figura 6.9.

Por fim, na Figura 6.11, pode-se verificar que, ao aumentar o número de vértices, as curvas dos algoritmos de `RT` tendem a se distanciar. Como os tempos de CPU dos algoritmos `RT(DRD)` e `RT(DRD+ST)` permaneceram próximos, estes parecem ser os melhores algoritmos no experimento em questão.

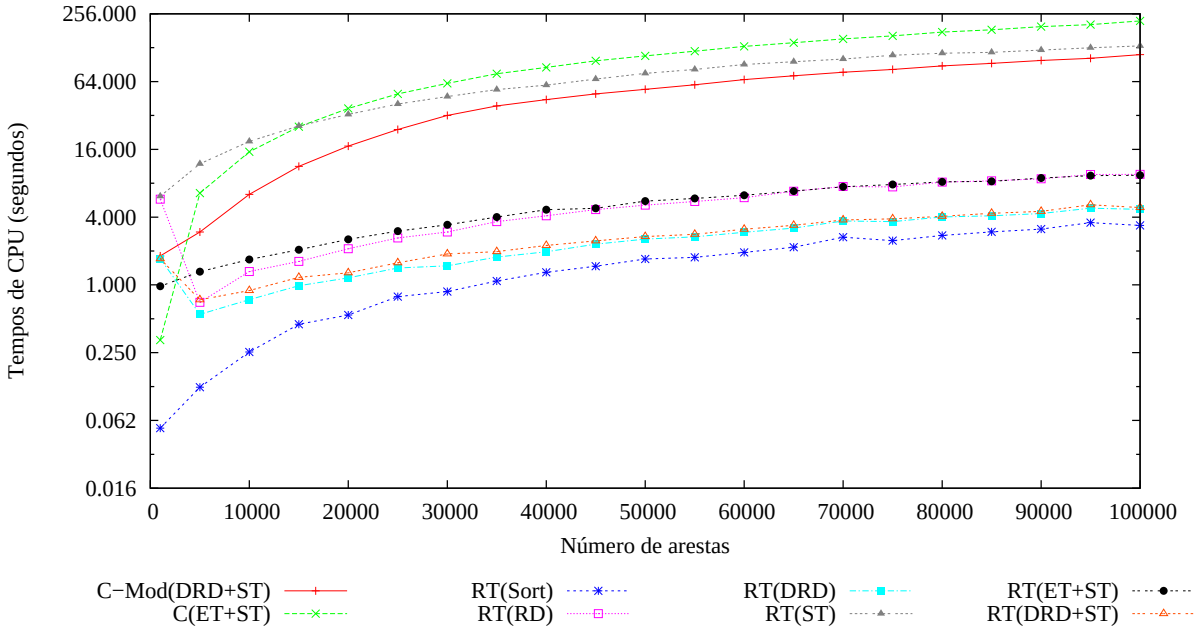


Figura 6.10: Tempos de CPU para a execução de 20000 atualizações estruturadas de custos em grafos aleatórios contendo 2000 vértices.

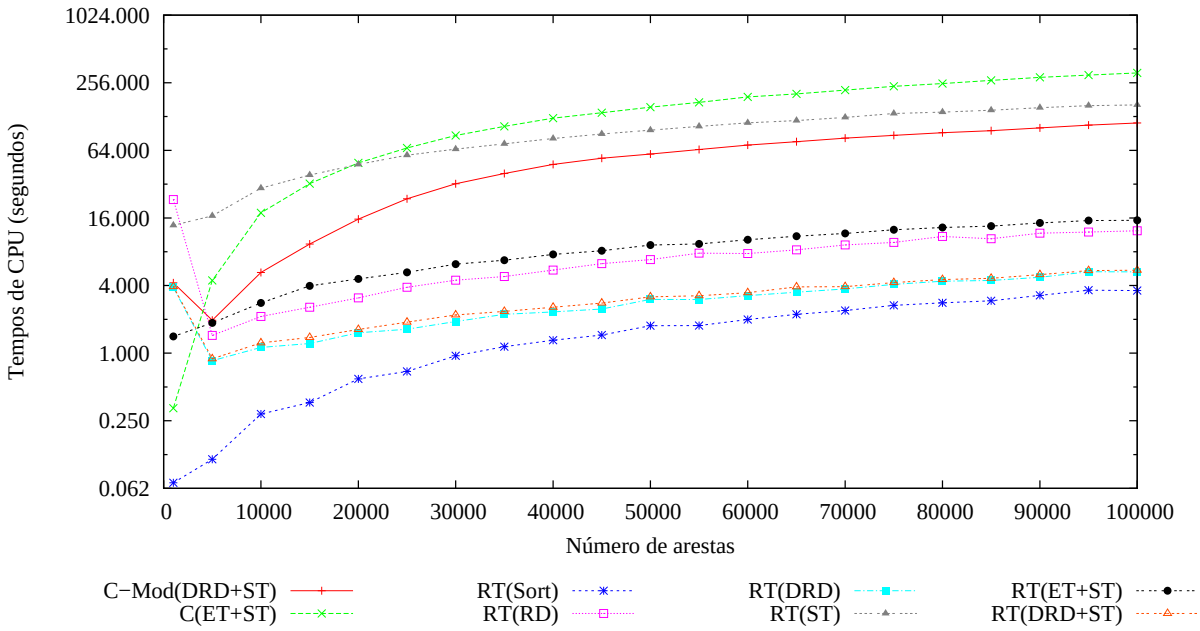


Figura 6.11: Tempos de CPU para a execução de 20000 atualizações estruturadas de custos em grafos aleatórios contendo 4000 vértices.

Na próxima seção, analisa-se o desempenho dos algoritmos com relação ao número de vértices.

6.2.1.3 Experimento Complementar: Variando o Número de Vértices

Este experimento tem por objetivo analisar o impacto do número de vértices nas estruturas para árvores dinâmicas, corrigindo uma deficiência dos trabalhos [4, 10]. Acredita-se que o impacto do número de vértices sobre o desempenho dos algoritmos é maior que o do número de arestas no desempenho dos algoritmos. Desta forma, fixou-se o número de arestas em $m = 99000$, enquanto que o número de vértices variou de $n = 500$ (grafo denso) a 99000 (grafo esparso). Esses resultados são apresentados na Figura 6.12.

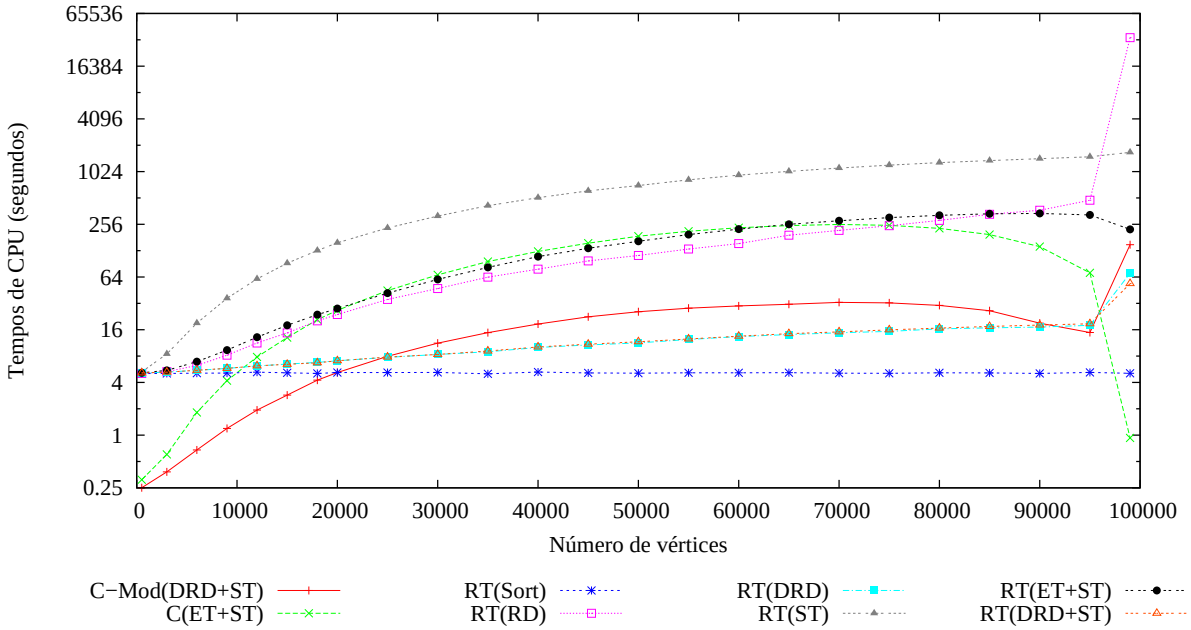


Figura 6.12: Tempos de CPU para a execução de 20000 atualizações aleatórias de custos em grafos aleatórios contendo 99000 arestas.

Os resultados da Figura 6.12 mostram a influência do número de vértices sobre os algoritmos implementados. Em primeiro lugar, pode-se atribuir a influência no comportamento dos algoritmos à complexidade das operações em árvores dinâmicas, já que o número de arestas permaneceu inalterado. Os algoritmos que mais foram afetados pelo aumento no número de vértices foram `RT(ST)` e `RT(RD)`, já que o primeiro possui uma estrutura de dados demasiadamente complexa (embora ofereça operações com complexidade baixa) e o segundo possui uma estrutura de dados demasiadamente simples (oferecendo operações com complexidade muito elevada).

Outro ponto a se considerar é a razão entre o número de arestas e de vértices do grafo, ou seja, quando este é esparso ou denso. Para grafos densos, todos os algoritmos possuem comportamento parecido, uma vez que operações complexas são realizadas poucas vezes

e que é fácil encontrar arestas substitutas quando uma operação complexa é executada. Quando o grafo começa a ficar esparso, a quantidade de operações caras começa a influenciar no comportamento de todos os algoritmos. Entretanto, para os algoritmos do grupo `C`, existe um limiar próximo a $n = 70000$ a partir do qual os tempos computacionais começam a cair. Isto é explicado pela quantidade de arestas armazenadas na árvore AVL, que diminui progressivamente até restar apenas uma (quando $n = 99000$). Para os algoritmos de `RT`, o invervalo entre 95000 e 99000 vértices é crítico pela crescente quantidade de operações que alteram a AGM. De fato, os algoritmos do grupo `C` são eficientes quando utilizados em grafos onde $n \approx m$ justamente pela baixa quantidade de arestas na árvore AVL.

Por fim, faz-se necessária a análise do comportamento do algoritmo `RT(ET+ST)` quando $n \approx m$, já que, enquanto todos os algoritmos de seu grupo apresentaram um notável aumento em seus tempos de CPU, este apresentou uma ligeira queda. Através de um exame de *profiling* [36], foi observado que a diminuição dos tempos se deu pela quase inexistência de alterações na AGM, o que traz a conclusão de que boa parte da sobrecarga deste algoritmo se encontra na atualização de ambas as árvores dinâmicas, e não nas consultas sobre conectividade.

6.2.1.4 Atualizações Aleatórias em Grafos *k*-Clique

Esta seção introduz o experimento de atualizações aleatórias em grafos k-Clique. A Figura 6.13 traz os resultados para os experimentos k-Clique da Tabela 6.4.

O gráfico da Figura 6.13 mostra que não há mudanças no comportamento dos algoritmos mesmo quando ocorrem operações muito custosas, uma vez que a probabilidade de alterações em arestas inter-clique é muito baixa. Assim, a ordem dos resultados se manteve idêntica àquela apresentada nos experimentos em grafos aleatórios.

6.2.1.5 Atualizações Estruturadas em Grafos *k*-Clique

Nesta seção são apresentados os resultados para 20000 operações de incremento nos custos de arestas inter-cliques, o que resulta no pior caso para todos os algoritmos aqui apresentados. Os resultados são sintetizados na Figura 6.14.

Através da Figura 6.14 é observado que o comportamento dos algoritmos do grupo `RT`, quando implementados utilizando boas estruturas de dados para manutenção de árvores dinâmicas (todas as estruturas com exceção das *RD-trees* e *ST-trees*), é superior

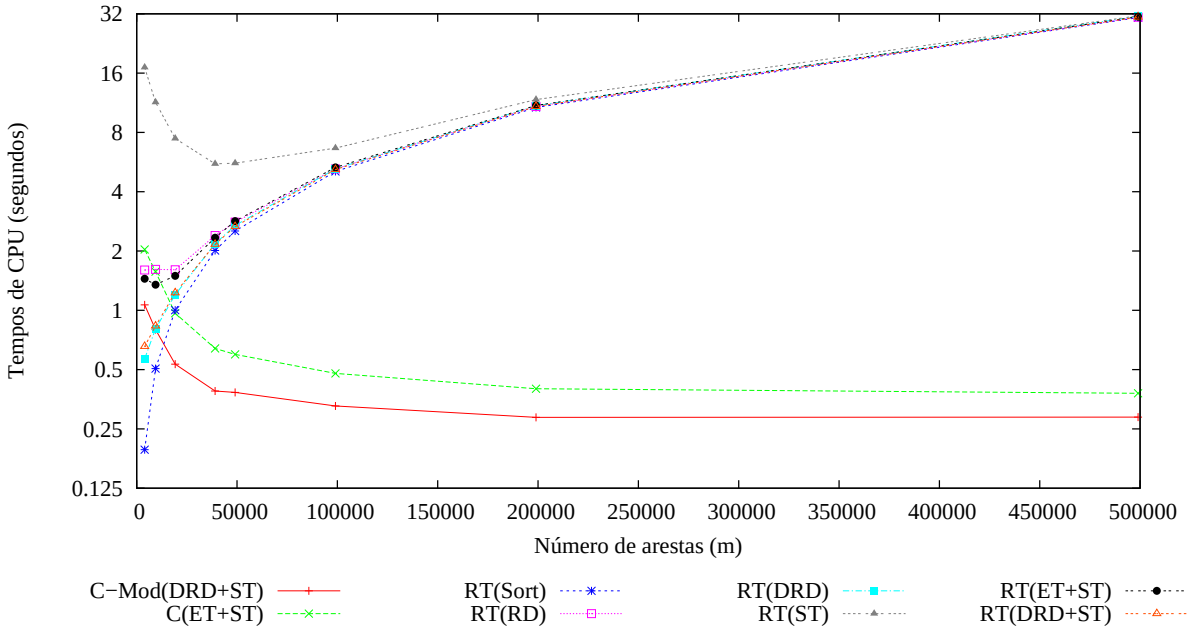


Figura 6.13: Tempos de CPU para a execução de 20000 atualizações aleatórias de custos em grafos k-Clique.

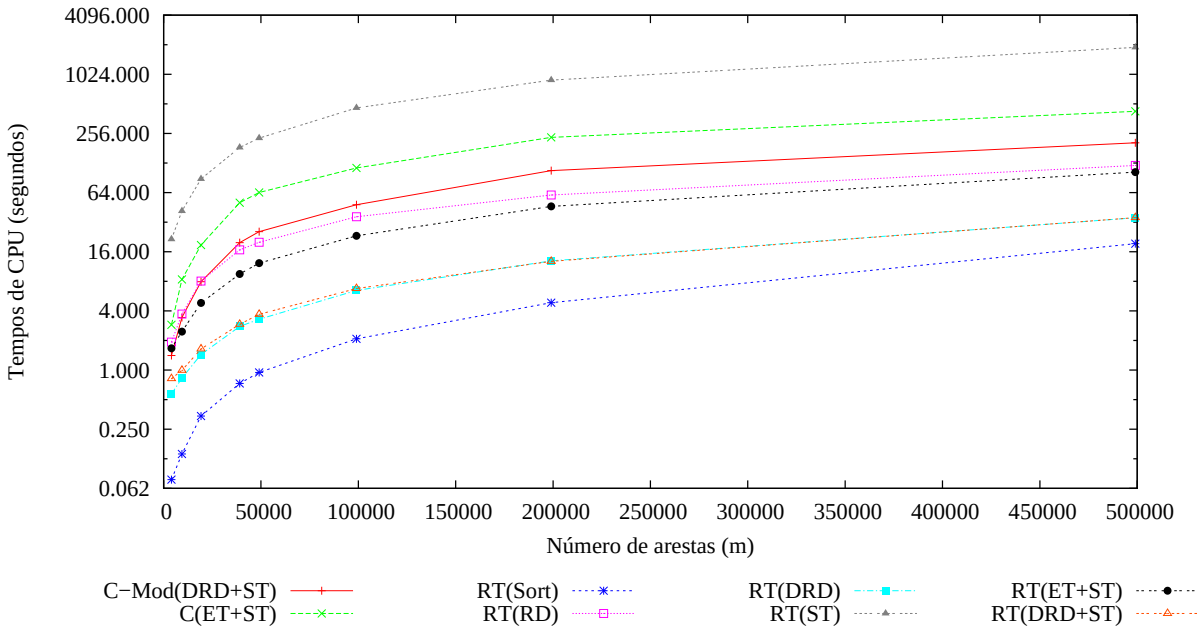


Figura 6.14: Tempos de CPU para a execução de 20000 operações de incremento em arestas inter-clique.

ao comportamento dos algoritmos do grupo `C`. Isto se dá pelo eficiente sistema de busca executado pelo primeiro grupo, onde inúmeras arestas são eliminadas da verificação de conectividade.

Novamente os melhores algoritmos para esta configuração de entrada são `RT(DRD)` e `RT(DRD+ST)`. Tomando por base a curva `RT(Sort)` das Figuras 6.13 e 6.14, que é limitante inferior dos algoritmos do grupo `RT`, fica claro que, mesmo aplicados a um conjunto de instâncias muito difícil, estes algoritmos possuem sobrecarga muito baixa para buscas por arestas substitutas. Por outro lado, ainda fazendo um paralelo entre as Figuras 6.13 e 6.14, é visível o aumento nos tempos de CPU dos algoritmos em `C`, influenciados pelas complexas seqüências de atualizações às quais o grafo foi submetido. Mesmo assim, nesse ambiente o algoritmo híbrido `C-Mod(DRD+ST)` foi superior ao algoritmo `C(ET+ST)`.

Nota-se que experimentos dessa natureza não são os melhores para simular ambientes reais em algoritmos dinâmicos, pois grafos k-Clique são muito específicos e criados apenas com a finalidade de induzir o pior caso nos algoritmos `C` e `RD` e favorecer os algoritmos de Holm et al. [30, 31] e de Frederickson [17, 18], detalhados no Capítulo 3.

6.2.1.6 Considerações sobre os Experimentos com Instâncias Sintéticas

Esta seção avaliou experimentalmente os algoritmos da Tabela 6.3 utilizando para isto um conjunto sintético de instâncias. Foram executados, para grafos aleatoriamente gerados, experimentos aleatórios (Seção 6.2.1.1) e estruturados (Seção 6.2.1.2). Para os experimentos da Seção 6.2.1.1, os algoritmos do grupo `C` obtiveram melhor comportamento, já que nesse experimento os algoritmos são submetidos a seqüências que praticamente não alteram a AGM do grafo. Ainda assim, verificou-se que a estrutura de dados *DRD-trees* conseguiu diminuir ainda mais os tempos obtidos por este grupo de algoritmos. Por outro lado, na Seção 6.2.1.5, os algoritmos são submetidos a seqüências caras de atualização, com 90% delas sendo incrementos em arestas da AGM ou decrementos em arestas de $E - E'$. Nesse ambiente, os algoritmos `RT(DRD)` e `RT(DRD+ST)` se destacam perante os demais, com tempos substancialmente menores que aqueles obtidos pelos algoritmos do grupo `C`.

Complementarmente, a Seção 6.2.1.3 trouxe um experimento onde variou-se o número de vértices do grafo. Novamente, os algoritmos `RT` foram mais eficientes para este conjunto de instâncias, excetuando-se o caso de grafos onde $n \approx m$.

Por fim, as Seções 6.2.1.4 e 6.2.1.5 trouxeram resultados para instâncias k-Clique, onde são gerados grafos compostos por k cliques de c nós cada. Este conjunto de instâncias, utilizado em [10], é formado por oito grafos de 2000 arestas cada, variando-se os valores de k e c. Na Seção 6.2.1.4 foram analisados os tempos de CPU dos algoritmos para seqüências aleatórias de atualizações, onde os algoritmos do grupo `C` obtiveram melhores resultados.

Finalmente, na Seção 6.2.1.5 mostrou-se os tempos de CPU para seqüências estruturadas de atualizações em grafos k-Clique, consistindo em incrementar ou decrementar arestas inter-cliques. Estas seqüências de atualizações são caras e os resultados mostraram que os algoritmos `RT(DRD)` e `RT(DRD+ST)` são mais rápidos.

Desta forma, conclui-se esta bateria de experimentos com a certeza de que os algoritmos propostos nesta dissertação são mais eficientes que os do estado da arte da literatura quando aplicados a seqüências difíceis de atualização.

6.2.2 Experimentos com Instâncias da Literatura

Os experimentos desta seção foram realizados sobre instâncias dos *DIMACS Implementation Challenges* [15], uma competição de algoritmos eficientes para um determinado problema. A nona edição desta competição foi direcionada a algoritmos para o problema de caminho mínimo em grafos (PCM), de onde as instâncias foram retiradas. Tais instâncias podem ser encontradas em [14]. Os grafos para esses problemas foram pré-processados de forma a remover arestas redundantes, já que os mesmos são, em sua versão original, orientados.

As instâncias utilizadas são divididas em quatro grupos. O primeiro grupo, denominado *Random4-n*, contém grafos gerados aleatoriamente. Neste caso, existem tanto componentes contendo grupos de vértices com grande quantidade de arestas para conectá-los quanto componentes contendo grupos de vértices com poucas arestas entre si (formando longos caminhos entre os vértices, onde não existem opções de arestas para conectá-los). O grupo *Long-n* contém grafos com caminhos longos, privilegiando as boas estruturas de dados para árvores dinâmicas, já que as árvores geradoras mínimas serão compostas por esses caminhos. O terceiro grupo, *Square-n*, procura concentrar os vértices em uma região limitada do plano, resultando em grafos com mais opções de arestas para substituição que o grupo *Long-n*. Por fim, o grupo *USA-road-d* é composto por instâncias reais, extraídas a partir do mapa rodoviário dos Estados Unidos da América. Estas instâncias foram divididas em regiões, já que todo o mapa é formado por milhões de vértices.

Para todos os grupos de instâncias acima, com exceção ao grupo *USA-road-d*, o número de vértices varia de 1024 a 524288. O número de arestas para o grupo *Random4-n* é cerca de quatro vezes o número de vértices, enquanto para os demais o número de arestas é em torno de três vezes o número de vértices. Os custos das arestas se encontram no intervalo $[1, n]$, exceto para o grupo *USA-road-d* onde os custos são baseados nas distâncias geográficas entre os vértices do grafo.

Foram omitidos os tempos de execução de alguns algoritmos que não processaram todas as 20000 atualizações às quais foram submetidos após 3600 segundos de execução. Nas tabelas das próximas seções, foram destacados em negrito os tempos de CPU correspondentes aos algoritmos com melhor desempenho para cada instância.

6.2.2.1 Atualizações Aleatórias

O primeiro grupo de experimentos, executado sobre todos os grupos de instâncias, consistiu em realizar 20000 atualizações aleatórias, com custos no intervalo $[1, n]$, em arestas também escolhidas aleatoriamente.

Primeiro, são apresentados e analisados os resultados da Tabela 6.5. Nela, observa-se que o algoritmo `C-Mod(DRD+ST)` tem melhor desempenho para um maior número de vértices e arestas. Salienta-se a grande diferença de desempenho entre esta variante e o algoritmo original, `C(ET+ST)`. Dois outros algoritmos apresentam desempenho competitivo com o algoritmo `C-Mod(DRD+ST)`: `RT(DRD)` e `RT(DRD+ST)`. Prejudicados pelo custo da reordenação da lista de arestas a cada operação, ainda assim estes algoritmos não apresentaram desempenho significativamente inferior àquele. Por outro lado, os algoritmos `RT(RD)` e `RT(ST)` apresentaram resultados muito ruins, já que utilizam estruturas de dados muito lentas para responder questões sobre conectividade.

Finaliza-se a análise da Tabela 6.5 com os resultados dos algoritmos `C(ET+ST)` e `RT(ET+ST)`. À primeira vista, pode parecer estranho que o primeiro algoritmo tenha se comportado de maneira superior ao segundo. Contudo, incrementos nos custos de arestas de $E - E'$ são tratados pelos algoritmos do grupo `C` através de simples inserções e remoções em árvores AVL, que são operações de complexidade logarítmica. Por outro lado, algoritmos da classe `RT` precisam re-ordenar a lista de arestas, para então constatar que a operação se deu sobre uma aresta do subconjunto $E - E'$. Além do custo da reordenação da lista de arestas, os custos das operações constantes nos algoritmos do grupo `RT` são um pouco maiores que os apresentados nos algoritmos do grupo `C`. Desta forma, o algoritmo `C(ET+ST)` mostrou-se superior ao algoritmo `RT(ET+ST)`.

Na Tabela 6.6 são apresentados os resultados dos algoritmos para as instâncias *Long-n*, que são mais difíceis que as instâncias *Random4-n*. O algoritmo de melhor desempenho para este grupo de instâncias foi o `RT(DRD+ST)`, que obteve ligeira vantagem sobre o algoritmo `RT(DRD)`. Esta diferença se deu pela vantagem de se utilizar a estrutura *ST-trees* nas operações de decremento, pois lidou-se com um conjunto de instâncias composto por caminhos longos, o que exige boas implementações da operação `Find_max`.

Tabela 6.5: Tempos de CPU (segundos) para as instâncias *Random4-n* após 20000 atualizações aleatórias.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Random4-n.10.0	1024	4080	0,53	0,81	0,14	0,50	**0,29**	6,08	0,69	0,41
Random4-n.11.0	2048	8178	0,76	1,43	0,27	1,11	**0,55**	13,22	1,18	0,63
Random4-n.12.0	4096	16372	1,23	3,41	0,55	2,64	**1,06**	28,55	3,11	1,10
Random4-n.13.0	8192	32748	2,33	8,80	1,22	7,30	**2,12**	63,13	8,74	2,26
Random4-n.14.0	16384	65524	4,73	23,35	2,49	20,50	**4,33**	142,05	24,15	4,28
Random4-n.15.0	32768	131055	9,95	68,04	4,95	53,95	8,21	322,68	68,39	**8,17**
Random4-n.16.0	65536	262122	20,48	197,47	11,47	180,40	17,59	772,52	202,06	**17,39**
Random4-n.17.0	131072	524273	**40,02**	540,06	30,39	1177,90	42,86	1971,35	583,93	41,83
Random4-n.18.0	262144	1048566	**72,55**	1332,88	64,11	3555,30	86,01	4390,44	1394,13	84,67
Random4-n.19.0	524288	2097139	**129,63**	3091,26	134,22	11622,00	177,51	10214,90	3315,53	175,23

Tabela 6.6: Tempos de CPU (segundos) para as instâncias *Long-n* após 20000 atualizações aleatórias.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Long-n.10.0	1024	2944	0,70	1,08	0,09	1,05	**0,35**	8,53	0,88	0,50
Long-n.11.0	2048	5936	1,05	1,66	0,14	3,07	**0,61**	18,15	1,49	0,76
Long-n.12.0	4096	11808	1,75	3,58	0,27	11,11	**1,20**	39,25	3,76	1,25
Long-n.13.0	8192	23729	3,26	8,72	0,58	92,87	2,93	85,70	11,33	**2,90**
Long-n.14.0	16384	47601	7,28	24,64	1,26	514,02	6,53	191,38	32,32	**6,33**
Long-n.15.0	32768	95150	18,81	73,15	2,64	2317,89	14,12	433,01	90,84	**12,63**
Long-n.16.0	65536	190493	40,93	204,58	5,86	-	30,46	1031,34	273,55	**25,77**
Long-n.17.0	131072	380952	93,79	543,73	13,00	-	68,95	2484,13	753,75	**57,44**
Long-n.18.0	262144	761894	199,10	1305,59	28,37	-	150,23	5720,72	1860,72	**124,70**
Long-n.19.0	524288	1523343	416,26	2980,40	60,27	-	318,97	12915,40	4336,07	**269,84**

Tabela 6.7: Tempos de CPU (segundos) para as instâncias *Square-n* após 20000 atualizações aleatórias.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Square-n.10.0	1024	2964	1,28	1,11	0,07	0,65	**0,62**	8,39	1,24	0,68
Square-n.11.0	2025	5938	1,70	1,75	0,13	1,80	**0,93**	17,97	1,75	0,94
Square-n.12.0	4096	12037	2,89	3,67	0,25	5,59	**1,76**	39,10	4,40	**1,76**
Square-n.13.0	8190	24283	4,32	8,92	0,58	31,30	3,30	87,05	12,13	**3,22**
Square-n.14.0	16384	48742	8,76	24,59	1,27	147,93	6,92	192,24	32,66	**6,17**
Square-n.15.0	32761	97634	19,61	72,95	2,37	662,02	10,65	427,74	90,93	**10,28**
Square-n.16.0	65536	195768	36,33	202,66	5,23	2423,81	17,87	1032,01	268,42	**17,28**
Square-n.17.0	131044	392136	61,94	515,46	12,49	-	36,50	2472,79	736,58	**34,17**
Square-n.18.0	262144	785034	110,97	1233,83	28,78	-	72,68	5747,83	1824,21	**68,67**
Square-n.19.0	524176	1570146	201,30	2803,96	63,47	-	144,76	13103,30	4258,94	**138,41**

Tabela 6.8: Tempos de CPU (segundos) para as instâncias *USA-road-d* após 20000 atualizações aleatórias.

Instância	n	m	RT(Sort)	RT(DRD)	RT(ET+ST)	RT(DRD+ST)
USA-road-d.BAY	321269	443518	33,44	128,87	4068,04	**126,80**
USA-road-d.COL	435665	619686	50,47	171,35	5350,45	**169,82**
USA-road-d.FLA	1070375	1519253	139,27	391,99	15058,30	**390,58**
USA-road-d.NE	1524452	2185573	206,14	**583,79**	23920,80	586,46
USA-road-d.NW	1207944	1576690	144,23	421,13	17927,40	**419,09**
USA-road-d.NY	264345	424580	31,35	102,35	3048,38	**101,30**

Os resultados da Tabela 6.7 não mudam o panorama acima apresentado, embora a distância entre o desempenho dos algoritmos `RT(DRD+ST)` e `RT(DRD)` seja bem menor em função da estrutura dos grafos do grupo *Square-n* (a ocorrência de caminhos longos não é freqüente).

O conjunto de instâncais *USA-road-d* foi utilizado para analisar o comportamento dos algoritmos propostos em um ambiente real ou ainda em um grafo de entrada composto por uma grande quantidade de vértices e arestas. Pela Tabela 6.8, observa-se que os resultados permanecem inalterados. Porém, enquanto os algoritmos `RT(DRD+ST)` e `RT(DRD)` obtiveram tempos extremamente bons, o algoritmo `RT(ET+ST)` se comportou de forma muito ruim. Este comportamento também foi seguido pelos outros algoritmos, que, devido a resultados preliminares, foram descartados. Na Tabela 6.8 não são apresentados os tempos de CPU relativos aos algoritmos do grupo `C`, em razão de serem muito elevados em comparação aos demais.

Por fim, em uma análise geral, observa-se que os algoritmos propostos no Capítulo 5 são mais rápidos do que aqueles já existentes na literatura. O ganho de tempo é considerável, ainda que comparados com o algoritmo `C-Mod(DRD+ST)`. Conclui-se que os melhores algoritmos para este conjunto de instâncias e esta configuração de atualizações são os baseados no algoritmo geral proposto no Capítulo 5, nominalmente os algoritmos `RT(DRD)` e `RT(DRD+ST)`. Este último, em instâncias com grande número de vértices, se apresenta como a melhor opção, uma vez que as *ST-trees* possuem operações com tempo assintoticamente menor que as *DRD-trees*.

A próxima seção segue com uma avaliação mais cuidadosa dos algoritmos propostos, visto que atualizações em arestas aleatórias não fornecem um bom modelo para representar o comportamento dos algoritmos. Nessas seqüências, muitas arestas que não pertencem à AGM são incrementadas, ao mesmo tempo que muitas arestas da AGM são decrementadas, o que é fácil de se resolver e muito distante da realidade para casos onde se quer, na verdade, perturbar a solução inicial em favor de uma nova solução (como é o caso de buscas locais, por exemplo).

6.2.2.2 Atualizações Estruturadas

Apresenta-se nesta seção os resultados para seqüências com apenas 10% de atualizações aleatórias, enquanto que o restante é composto por atualizações focadas em (a) incrementar o custo de arestas da AGM e (b) decrementar o custo de arestas do grafo mas não da AGM. O objetivo desses experimentos é forçar a alteração freqüente da AGM.

Tabela 6.9: Tempos de CPU (segundos) para as instâncias *Random4-n* após 20000 atualizações estruturadas.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Random4-n.10.0	1024	4080	1,64	3,50	0,06	**0,31**	0,37	5,37	0,65	0,55
Random4-n.11.0	2048	8178	3,35	8,84	0,11	0,84	**0,65**	12,59	1,18	0,81
Random4-n.12.0	4096	16372	7,35	29,63	0,27	2,06	**1,22**	31,16	3,13	1,40
Random4-n.13.0	8192	32748	18,90	96,73	0,58	6,84	**2,27**	80,60	10,25	2,59
Random4-n.14.0	16384	65524	50,06	283,26	1,42	22,03	**4,51**	211,93	34,88	4,85
Random4-n.15.0	32768	131055	104,21	849,97	3,32	71,27	**9,36**	545,17	112,12	9,64
Random4-n.16.0	65536	262122	201,07	2558,76	7,34	274,40	**19,22**	1452,86	362,24	19,42
Random4-n.17.0	131072	524273	254,03	-	8,19	1785,24	**30,12**	2424,00	853,67	30,30
Random4-n.18.0	262144	1048566	295,70	-	18,52	-	65,33	-	2453,94	**64,58**
Random4-n.19.0	524288	2097139	554,63	-	43,69	-	142,35	-	6536,71	**141,99**

Tabela 6.10: Tempos de CPU (segundos) para as instâncias *Long-n* após 20000 atualizações estruturadas.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Long-n.10.0	1024	2944	0,85	1,07	0,09	0,65	**0,79**	5,99	0,73	0,88
Long-n.11.0	2048	5936	1,88	2,01	0,09	2,05	**0,63**	12,60	0,97	0,81
Long-n.12.0	4096	11808	3,42	5,03	0,16	7,27	**1,21**	27,41	2,15	1,35
Long-n.13.0	8192	23729	6,88	13,85	0,36	59,09	**2,61**	59,59	6,39	2,68
Long-n.14.0	16384	47601	15,80	40,86	0,82	329,28	6,07	132,68	20,55	**5,94**
Long-n.15.0	32768	95150	39,64	94,11	1,75	1709,10	14,18	330,58	74,85	**12,89**
Long-n.16.0	65536	190493	86,34	285,06	3,99	-	28,97	816,44	264,84	**27,23**
Long-n.17.0	131072	380952	178,30	787,00	8,76	-	67,17	1930,90	822,80	**61,12**
Long-n.18.0	262144	761984	357,91	2029,79	19,53	-	148,92	-	2237,49	**137,93**
Long-n.19.0	524288	1523343	731,00	-	42,94	-	321,90	-	5508,47	**297,70**

Tabela 6.11: Tempos de CPU (segundos) para as instâncias *Square-n* após 20000 atualizações estruturadas.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Square-n.10.0	1024	2964	1,40	1,16	0,05	0,59	**0,62**	6,05	0,82	0,66
Square-n.11.0	2025	5938	2,15	1,98	0,09	1,62	**1,04**	12,57	1,15	1,16
Square-n.12.0	4096	12037	4,04	5,06	0,18	4,76	1,97	27,57	2,60	**1,94**
Square-n.13.0	8190	24283	7,05	14,17	0,39	22,49	**2,50**	60,78	6,40	2,59
Square-n.14.0	16384	48742	15,42	41,11	0,84	109,98	**4,68**	134,33	20,49	4,70
Square-n.15.0	32761	97634	36,05	98,54	1,83	582,57	9,71	325,64	73,53	**9,29**
Square-n.16.0	65536	195768	73,07	294,02	4,07	2015,87	18,67	818,78	262,84	**18,23**
Square-n.17.0	131044	392136	140,81	805,21	8,88	11702,30	37,97	1982,62	816,37	**36,92**
Square-n.18.0	262144	785034	264,06	2053,33	19,70	-	77,83	-	2198,69	**75,88**
Square-n.19.0	524176	1570146	500,72	-	44,21	-	156,67	-	5452,39	**153,00**

Tabela 6.12: Tempos de CPU (segundos) para as instâncias *USA-road-d* após 20000 atualizações estruturadas.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(DRD)	RT(ET+ST)	RT(DRD+ST)
USA-road-d.BAY	321269	443518	129,53	1211,83	24,33	89,49	2328,03	**82,80**
USA-road-d.COL	435665	619686	190,72	1604,13	34,24	117,71	3189,87	**110,44**
USA-road-d.FLA	1070375	1519253	467,91	2234,94	98,37	264,02	9611,59	**258,29**
USA-road-d.NE	1524452	2185573	726,44	1207,91	148,34	406,74	15884,20	**403,82**
USA-road-d.NW	1207944	1576690	507,72	2596,91	99,06	276,76	11251,80	**273,25**
USA-road-d.NY	264345	424580	166,84	1391,26	22,93	73,47	1829,68	**67,83**

Na Tabela 6.9 os tempos dos algoritmos do grupo `C` não se mostraram competitivos, já que essas atualizações forçam mudanças na AGM e fazem com que a busca por arestas substitutas seja realizada. Este fato contribui também para a grande diferença entre o método híbrido `C-Mod(DRD+ST)` e o método `C(ET+ST)`, visto que o uso de *DRD-trees*, em casos onde o número de consultas de conectividade é grande, melhora substancialmente o desempenho do método híbrido.

O comportamento dos algoritmos do grupo `RT` permaneceu inalterado para esta classe de problemas. Isto é explicado pela maneira como esses algoritmos tratam as operações de incrementos em pesos de arestas, que elimina boa parte das arestas que são obrigatoriamente analisadas nos algoritmos do grupo `C`. Para as Tabelas 6.10 e 6.11 os resultados também não se alteraram, indicando que o melhor algoritmo para se trabalhar em incrementos e decrementos estruturados é o algoritmo `RT(DRD+ST)`, por oferecer uma maneira consistente para se lidar com caminhos longos além de, ao mesmo tempo, ser robusto para casos onde esta situação não ocorre. Ainda, o algoritmo `RT(DRD)` também se destaca por ter bom desempenho em grafos com grande quantidade de arestas e, ainda, por manter esse desempenho em grafos esparsos, onde longos caminhos de vértices podem surgir. O algoritmo `C-Mod(DRD+ST)` consumiu pelo menos duas vezes o tempo utilizado pelos algoritmos mais rápidos. O algoritmo `RT(ET+ST)` obteve o pior desempenho dos algoritmos que puderam ter seus tempos medidos.

Na Tabela 6.12 encontram-se os resultados das execuções estruturadas em grafos da classe *USA-road-d*, caracterizados por serem notadamente mais esparsos que os grafos dos outros conjuntos de instâncias de [15]. Assim, como a árvore AVL dos algoritmos `C` não contém grande quantidade de arestas, foi melhor o desempenho do algoritmo `C(ET+ST)` quando comparado com o algoritmo `RT(ET+ST)`, visto que o último analisa muitas arestas que já se encontram na AGM, bem como boa parte das arestas que são também analisadas pelo algoritmo `C(ET+ST)`. Mesmo assim, o algoritmo híbrido `C(ET+ST)` obteve tempos de 9,34 a 1,66 vezes maiores que o algoritmo `C-Mod(DRD+ST)`. Conclui-se esta tabela com o fato de o algoritmo `RT(DRD+ST)` ser a melhor opção para tal conjunto de instâncias. Têm-se ainda que o algoritmo `RT(DRD)` possui tempos muito bons, ficando sempre a poucos segundos das execuções do algoritmo `RT(DRD+ST)`.

6.2.2.3 Comparando Algoritmos Dinâmicos com Algoritmos Estáticos

Com o objetivo de oferecer um ponto de comparação entre os algoritmos dinâmicos e estáticos, a Tabela 6.13 traz os resultados do seguinte experimento: para cada grafo dos

grupos *Random4-n*, *Long-n* e *Square-n* foi dado aos algoritmos de Prim [37] e Kruskal [33] o mesmo tempo de CPU gasto pelo algoritmo `RT(DRD+ST)` para a realização de 20000 atualizações nesses grafos. Assim, computou-se o número de atualizações processadas por cada um desses algoritmos.

Tabela 6.13: Quantidade de atualizações processadas pelos algoritmos clássicos para AGMs utilizando mesmo tempo de CPU gasto pelo Algoritmo `RT(DRD+ST)`.

Instância	n	m	Tempo	`RT(DRD+ST)`	`Prim`	`Kruskal`
Random4-n.10.0	1024	4080	0,37	20000	581	1081
Random4-n.11.0	2048	8178	0,65	20000	440	946
Random4-n.12.0	4096	16372	1,22	20000	350	785
Random4-n.13.0	8192	32748	2,27	20000	290	692
Random4-n.14.0	16384	65524	4,51	20000	206	661
Random4-n.15.0	32768	131055	9,36	20000	161	642
Random4-n.16.0	65536	262122	19,22	20000	126	595
Random4-n.17.0	131072	524273	30,12	20000	75	423
Random4-n.18.0	262144	1048566	65,33	20000	59	388
Random4-n.19.0	524288	2097139	142,35	20000	50	326
Long-n.10.0	1024	2944	0,79	20000	1412	2553
Long-n.11.0	2048	5936	0,63	20000	510	881
Long-n.12.0	4096	11808	1,21	20000	494	798
Long-n.13.0	8192	23729	2,61	20000	442	810
Long-n.14.0	16384	47601	6,07	20000	444	887
Long-n.15.0	32768	95150	14,18	20000	408	969
Long-n.16.0	65536	190493	28,97	20000	352	878
Long-n.17.0	131072	380952	67,17	20000	339	925
Long-n.18.0	262144	761894	148,92	20000	304	863
Long-n.19.0	524288	1523343	321,90	20000	272	705
Square-n.10.0	1024	2964	0,62	20000	1117	1922
Square-n.11.0	2025	5938	1,04	20000	809	1534
Square-n.12.0	4096	12037	1,97	20000	664	1287
Square-n.13.0	8190	24283	2,50	20000	389	769
Square-n.14.0	16384	48742	4,68	20000	294	672
Square-n.15.0	32761	97634	9,71	20000	244	656
Square-n.16.0	65536	195768	18,97	20000	187	581
Square-n.17.0	131044	392136	37,97	20000	150	524
Square-n.18.0	262144	785034	77,83	20000	115	444
Square-n.19.0	524176	1570146	156,67	20000	91	343
			Total	600000	11375	25540
			Média	20000	379	851

Através da Tabela 6.13 pode-se concluir que a diferença entre os tempos de CPU apresentados pelos algoritmos dinâmicos em comparação com os algoritmos estáticos é muito grande. Com esta tabela, conclui-se ainda que a utilização de um bom algoritmo dinâmico

é fundamental para o desempenho de aplicações baseadas em entradas dinâmicas, onde o grafo passa por alterações estruturais ao longo da execução da mesma.

6.2.2.4 Atualizações Incrementais em Arestas da AGM

Estes experimentos foram propostos com o intuito de avaliar o desempenho dos algoritmos quando submetidos a seqüências de incrementos de custo em arestas da AGM. A justificativa para a execução de experimentos nesse ambiente é que buscas locais podem varrer um determinado espaço de busca simplesmente alterando os custos de arestas. Para uma varredura objetiva, essas alterações são normalmente voltadas a incrementos ou decrementos que forcem a alteração da AGM. Nessa seção optou-se por realizar apenas incrementos nos custos de arestas da AGM, o que pode fazer com que essa AGM mude freqüentemente.

As Tabelas 6.14 a 6.16, trazem os tempos de CPU gastos pelos algoritmos para executar uma seqüência de 20000 incrementos de custos no valor de $n/16$ em arestas da AGM. Novamente, comprova-se o que vêm sendo constatado pelos experimentos estruturados realizados ao longo deste trabalho: `RT(DRD)` e `RT(DRD+ST)` são os melhores algoritmos para tais circunstâncias, nas quais o segundo algoritmo obtém uma ligeira vantagem quando aplicado em instâncias contendo longos caminhos de vértices (representadas pelo conjunto de instâncias *Long-n*).

6.2.2.5 Considerações sobre os Experimentos com Instâncias Reais

A seção 6.2.2 avaliou experimentalmente os algoritmos da Tabela 6.3 utilizando um conjunto real de instâncias, obtidas do *The 2005/2006 Ninth DIMACS Implementation Challenge – Shortest Paths*. Este conjunto de instâncias foi escolhido por oferecer um conjunto robusto e completo de grafos, composto de instâncias aleatórias e estruturadas, e ainda por um conjunto de grafos sobre o mapa rodoriário dos Estados Unidos da América.

Para tanto, foram executados experimentos aleatórios (Seção 6.2.2.1) e estruturados (Seção 6.2.2.2). Foi constatado que, tanto para os experimentos da Seção 6.2.2.1 quanto para os experimentos da Seção 6.2.2.2, os algoritmos e estruturas de dados propostos nessa dissertação obtiveram os menores tempos de CPU por execução, destacando-se os algoritmos `RT(DRD)` e `RT(DRD+ST)`.

Tabela 6.14: Tempos de CPU (segundos) para as instâncias *Random4-n* após 20000 incrementos de valor $n/16$ em arestas da AGM.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Random4-n.10.0	1024	4080	1,33	1,86	0,02	0,33	**0,20**	3,79	0,53	0,38
Random4-n.11.0	2048	8178	2,22	2,38	0,01	0,68	**0,32**	6,42	0,73	0,50
Random4-n.12.0	4096	16372	4,19	4,32	0,03	1,22	**0,61**	12,13	1,34	0,77
Random4-n.13.0	8192	32748	8,14	9,25	0,06	3,11	**1,15**	24,67	3,69	1,32
Random4-n.14.0	16384	65524	16,01	22,68	0,15	7,71	**2,43**	53,73	10,41	2,75
Random4-n.15.0	32768	131055	34,65	63,40	0,30	23,34	**4,74**	114,95	36,91	5,27
Random4-n.16.0	65536	262122	77,69	194,91	0,66	74,27	**9,52**	269,24	136,38	10,22
Random4-n.17.0	131072	524273	158,04	529,69	1,29	422,18	**19,36**	645,67	422,77	19,77
Random4-n.18.0	262144	1048566	317,46	1259,43	2,65	1426,46	**38,77**	1498,02	1152,66	39,21
Random4-n.19.0	524288	2097139	641,64	3135,93	5,30	4291,39	**78,05**	3384,15	2916,86	78,59

Tabela 6.15: Tempos de CPU (segundos) para as instâncias *Long-n* após 20000 incrementos de valor $n/16$ em arestas da AGM.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Long-n.10.0	1024	2944	1,24	1,59	0,01	0,55	**0,27**	4,28	0,53	0,43
Long-n.11.0	2048	5936	2,17	2,92	0,01	1,41	**0,50**	7,01	0,71	0,63
Long-n.12.0	4096	11808	4,03	6,77	0,03	3,68	**0,90**	13,05	1,31	1,15
Long-n.13.0	8192	23729	7,87	16,58	0,06	29,92	**1,84**	26,52	3,59	1,91
Long-n.14.0	16384	47601	16,02	46,26	0,18	163,20	**4,45**	57,02	11,43	4,55
Long-n.15.0	32768	95150	38,50	130,22	0,34	702,38	**9,65**	136,02	38,93	10,29
Long-n.16.0	65536	190493	87,45	365,01	0,68	3273,62	**20,60**	319,19	122,40	20,83
Long-n.17.0	131072	380952	185,72	934,10	1,45	21227,40	**44,19**	746,52	353,95	45,64
Long-n.18.0	262144	761894	384,78	2304,84	3,05	-	97,76	1772,30	929,32	**97,41**
Long-n.19.0	524288	1523343	803,80	5198,41	6,36	-	207,67	-	2266,31	**201,94**

Tabela 6.16: Tempos de CPU (segundos) para as instâncias *Square-n* após 20000 incrementos de valor $n/16$ em arestas da AGM.

Instância	n	m	C-Mod(DRD+ST)	C(ET+ST)	RT(Sort)	RT(RD)	RT(DRD)	RT(ST)	RT(ET+ST)	RT(DRD+ST)
Square-n.10.0	1024	2964	1,25	1,39	0,02	0,53	**0,28**	4,05	0,53	0,44
Square-n.11.0	2025	5938	2,17	2,60	0,02	0,95	**0,41**	6,69	0,67	0,65
Square-n.12.0	4096	12037	3,94	5,75	0,03	2,64	**0,76**	12,64	1,35	0,99
Square-n.13.0	8190	24283	7,68	14,21	0,06	14,03	**1,50**	25,09	3,52	1,71
Square-n.14.0	16384	48742	15,31	38,30	0,15	62,54	**3,30**	55,39	10,45	3,46
Square-n.15.0	32761	97634	33,92	112,97	0,32	228,26	**6,55**	128,99	36,53	6,81
Square-n.16.0	65536	195768	74,54	329,62	0,71	794,62	**12,65**	308,13	121,43	12,84
Square-n.17.0	131044	392136	146,44	867,67	1,34	4659,24	**23,02**	738,83	353,27	23,93
Square-n.18.0	262144	785034	285,00	2150,43	2,71	-	**44,78**	1751,10	929,67	45,36
Square-n.19.0	524176	1570146	564,65	-	6,01	-	**88,55**	-	2275,51	88,69

Complementando os resultados estruturados e ainda aproveitando que este é o experimento mais dificil de se executar através de algoritmos dinâmicos (i.e., para os algoritmos dinâmicos estas seqüências de atualização induzem aos casos mais difíceis de serem tratados), foi mostrada na Seção 6.2.2.3 a vantagem numérica de se utilizar um algoritmo dinâmico ao invés de um algoritmo estático, com 52,75 e 23,5 vezes mais atualizações processadas que os algoritmos de Prim [37] e de Kruskal [33].

Na Seção 6.2.2.4 foi realizado um experimento composto apenas por incrementos em arestas da AGM, com o objetivo de se gerar mudanças estruturais nessa árvore. Nesse caso, foi adotado o valor de atualização igual a $n/16$ e novamente os melhores algoritmos foram `RT(DRD)` e `RT(DRD+ST)`.

Embora os algoritmos do grupo `C` tenham se mostrado competitivos quando a seqüência de atualizações foi aleatoriamente gerada, este resultado não permaneceu assim quando esta seqüência forçou mudanças na AGM. Os tempos de CPU desses métodos se mantiveram praticamente iguais para todos os experimentos com exceção do primeiro, o que prova que tais algoritmos não são competitivos. Ainda assim, ficou clara a contribuição da estrutura para árvores dinâmicas *DRD-trees*, proposta no Capítulo 4 desta dissertação. Isto aconteceu porque tais algoritmos analisam uma grande quantidade de arestas com respeito à conectividade, justamente a grande contribuição dessa estrutura.

6.3 Resumo do Capítulo

Este capítulo analisou experimentalmente os algoritmos e estruturas de dados propostos nesta dissertação. Foi demonstrada a eficiência da estrutura de dados para árvores dinâmicas *DRD-trees* com relação à execução da operação de consulta sobre conectividade entre dois vértices. Os algoritmos propostos para atualização da AGM de um grafo dinâmico quando submetido a uma alteração no custo de uma aresta também se mostraram eficientes, apresentando bons resultados nos principais experimentos realizados.

Capítulo 7

Conclusões e Trabalhos Futuros

Esta dissertação abordou o problema de atualização da árvore geradora mínima de um grafo sujeito a incrementos ou decrementos em custos de arestas.

Foram propostos algoritmos e estruturas de dados que, embora não tragam avanços do ponto de vista teórico, conseguem aliar a facilidade de implementação computacional com um excelente desempenho prático.

Este desempenho foi analisado através de três classes de experimentos. A primeira delas foi executada sobre as estruturas de dados para árvores dinâmicas, mostrando que as *DRD-trees* são estruturas rápidas em consultas de conectividade e ao mesmo tempo não apresentam grandes diferenças nos tempos de CPU utilizados para a realização de operações de *link* e *cut*.

A segunda classe de experimentos consistiu em utilizar os mesmos parâmetros de grafos e atualizações apresentados em [10] para comparar os algoritmos aqui propostos com aquele que obteve os melhores resultados naquele trabalho. Os resultados mostraram que a utilização da estrutura de dados *DRD-trees* contribuiu para a redução significativa dos tempos de CPU do algoritmo de Cattaneo et al. [10], bem como os algoritmos aqui propostos obtiveram tempos computacionais menores que esses dois algoritmos em grande parte dos experimentos desta classe.

A terceira e última classe de experimentos utilizou os grafos disponibilizados em [15] para oferecer experimentos condizentes com aqueles encontrados no mundo real. Submetidos a essas instâncias, sob seqüências de atualizações aleatórias, estruturadas e unitárias, os algoritmos propostos neste trabalho continuaram obtendo os melhores tempos de CPU quando comparados aos algoritmos de [10].

Cabe destacar os algoritmos `RT(DRD+ST)` e `RT(DRD)` como os melhores algoritmos para a grande maioria dos experimentos aqui utilizados.

As implementações dos algoritmos e estruturas de dados aqui propostos foram conduzidas com o cuidado de se prover códigos reusáveis e eficientes, baseados na linguagem de programação C++ e no projeto orientado a objetos. Desta forma, um código bastante flexível é então, a partir de agora, disponibilizado ao domínio público sob a licença GPL. Salienta-se que nenhum algoritmo foi implementado beneficiando-se de bibliotecas de código fechado.

Este trabalho pode ser estendido através da implementação eficiente dos algoritmos de Holm et al. [30, 31] e de Frederickson [17, 18], que, embora tenham apresentado resultados práticos ruins, são importantes devido às suas complexidades teóricas. O trabalho de Cattaneo et al. [10] já experimentou ambas as implementações acima e constatou sua ineficiência prática. Contudo, essa implementação foi baseada na biblioteca LEDA [34] (*Library of Efficient Data-types and Algorithms*), uma biblioteca de código fechado e comercial. A implementação aberta desses algoritmos permitiria sua comparação com os algoritmos aqui propostos, principalmente nos experimentos k-Clique em [10], onde o algoritmo `C(ET+ST)` não foi experimentado por ser ineficiente.

Referências

[1] Acar, U., Blelloch, G., Harper, R., Vittes, J., Woo, S. Dynamizing static algorithms, with applications to dynamic trees and history dependence. Em *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, 2004), p. 524–533.

[2] Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M. Minimizing diameters of dynamic trees. Em *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming* (Bologna, 1997), P. Degano, R. Gorrieri, e A. Marchetti-Spaccamela, Eds., vol. 1256 de *Lecture Notes in Computer Science*, Springer-Verlag, p. 270–280.

[3] Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M. Maintaining diameter, center, and median of fully-dynamic trees with top-trees, 2003. Referência on-line em `http://arxiv.org/abs/cs/0310065`, visitada em 15 de janeiro de 2006.

[4] Amato, G., Cattaneo, G., Italiano, G. Experimental analysis of dynamic minimum spanning tree algorithms (extended abstract). Em *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, 1997), p. 314–323.

[5] Bent, S., Sleator, D., Tarjan, R. Biased search trees. *SIAM Journal on Computing 14* (1985), 545–568.

[6] Buriol, L. *Roteamento do Tráfego na Internet: Algoritmos para Projeto e Operação de Redes com Protocolo OSPF*. Tese de Doutorado, Universidade Estadual de Campinas, 2003.

[7] Buriol, L., Resende, M., Ribeiro, C., Thorup, M. A memetic algorithm for OSPF routing. Em *Proceedings of the 6th INFORMS Telecom* (Boca Raton, 2002), p. 187–188.

[8] Buriol, L., Resende, M., Ribeiro, C., Thorup, M. A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. *Networks 46* (2005), 36–56.

[9] Buriol, L., Resende, M., Thorup, M. Speeding up shortest path algorithms. Relatório Técnico TD-5RJ8B, AT&T Labs Research, 2003.

[10] Cattaneo, G., Faruolo, P., Petrillo, U., Italiano, G. Maintaining dynamic minimum spanning trees: An experimental study. Em *Algorithm Engineering and Experiments: 4th International Workshop* (San Francicsco, 2002), D. Mount e C. Stein, Eds., vol. 2409 de *Lecture Notes in Computer Science*, Springer-Verlag, p. 111–125.

[11] Chin, F., Houck, D. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences 16* (1978), 333–344.

[12] Cormen, T., Leiserson, C., Rivest, R., Stein, C. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[13] Das, B., Loui, M. Reconstructing a minimum spanning tree after deletion of any node. *Algorithmica 31* (2001), 530–547.

[14] Demetrescu, C., Goldberg, A., Johnson, D. Ninth DIMACS implementation challenge, 2006. Referência on-line em `http://www.dis.uniroma1.it/~challenge9/`, visitada em 23 de junho de 2006.

[15] DIMACS. DIMACS implementation challenges. Referência on-line em `http://dimacs.rutgers.edu/Challenges/`, visitada em 23 de junho de 2006.

[16] Eppstein, D., Galil, Z., Italiano, G. F., Nissemzweig, A. Sparsification – A technique for speeding up dynamic graph algorithms. *Journal of the ACM 44* (1997), 669–696.

[17] Frederickson, G. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing 14* (1985), 781–798.

[18] Frederickson, G. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. Em *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science* (San Juan, 1991), p. 632–641.

[19] Fredman, M., Tarjan, R. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM 34* (1987), 596–615.

[20] Galil, Z., Italiano, G. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys 23* (1991), 319–244.

[21] Galil, Z., Italiano, G. Fully-dynamic algorithms for 2-edge connectivity. *SIAM Journal on Computing 21* (1992), 1047–1069.

[22] Harary, F. *Graph Theory*. Addison-Wesley, 1969.

[23] Held, M., Karp, R. The traveling-salesman problem and minimum spanning trees. *Operations Research 18* (1970), 1138–1162.

[24] Held, M., Karp, R. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming 1* (1970), 6–25.

[25] Henzinger, M., King, V. Randomized dynamic graph algorithms with polylogarithmic time per operation. Em *Proceedings of the 27th ACM Symposium on Theory of Computing* (Las Vegas, 1995), p. 519–527.

[26] Henzinger, M., King, V. Maintaining minimum spanning trees in dynamic graphs. Em *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming* (Bologna, 1997), P. Degano, R. Gorrieri, e A. Marchetti-Spaccamela, Eds., vol. 1256 de *Lecture Notes in Computer Science*, Springer-Verlag, p. 594–604.

[27] Henzinger, M., King, V. Randomized fully-dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM 46* (1999), 502–516.

[28] Henzinger, M., Poutré, H. Certificates and fast algorithms for biconnectivity in fully dynamic graphs. Em *Proceedings of the 3rd European Symposium on Algorithms* (Corfu, 1995), P. Spirakis, Ed., vol. 959 de *Lecture Notes in Computer Science*, Springer-Verlag, p. 594–604.

[29] Henzinger, M., Thorup, M. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures and Algorithms 11* (1996), 171–184.

[30] Holm, J., de Lichtenberg, K. Top-trees and dynamic graph algorithms. Dissertação de Mestrado, University of Copenhagen, 1998.

[31] Holm, J., de Lichtenberg, K., Thorup, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Em *Proceedings of the 30th ACM Symposium on Theory of Computing* (Dallas, 1998), p. 79–89.

[32] Karger, D., Klein, P., Tarjan, R. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM 42* (1995), 321–328.

[33] Kruskal, J. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society 7* (1956), 48–50.

[34] Mehlhorn, K., Näher, S. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM 38* (1995), 96–102.

[35] Miller, G., Reif, J. Parallel tree contraction and its application. Em *Proceedings of the 26th Annual Symposium on Foundations of Computer Science* (Portland, 1985), p. 478–489.

[36] Petrillo, U. *Algorithm Engineering: Methodologies and Support Tools.* Tese de Doutorado, Universitá di Salerno, 2001.

[37] Prim, R. Shortest connection networks and some generalizations. *Bell Systems Technical Journal 36* (1957), 1389–1401.

[38] Rauch, M. *Fully Dynamic Graph Algorithms and Their Data Structures.* Tese de Doutorado, Princeton University, 1992.

[39] Rauch, M. Improved data structures for fully dynamic biconnectivity. Em *Proceedings of the 26th ACM Symposium on Theory of Computing* (Montreal, 1994), p. 503–538.

[40] Rauch, M. Fully-dynamic biconnectivity in graphs. *Algorithmica 13* (1995), 503–538.

[41] Sleator, D., Tarjan, R. A data structure for dynamic trees. *Journal of Computer and System Sciences 26* (1983), 362–391.

[42] Sleator, D., Tarjan, R. Self-adjusting binary trees. Em *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, 1983), p. 235–245.

[43] Sleator, D., Tarjan, R. Self-adjusting binary search trees. *Journal of the ACM 32* (1985), 652–686.

[44] Spira, P., Pan, A. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing 4* (1975), 375–380.

[45] Tarjan, R. Depth-first and linear graph algorithms. *SIAM Journal on Computing 1* (1972), 146–160.

[46] Tarjan, R. Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22* (1975), 215–225.

[47] Werneck, R., Tarjan, R. Self-adjusting top trees. Em *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms* (Vancouver, 2005), p. 813–822.

[48] Zaroliagis, C. Implementations and experimental studies of dynamic graph algorithms. Em *Experimental Algoritms - From Algorithm Design to Robust and Efficient Software*, R. Fleischer, B. Moret, e E. Schmidt, Eds., vol. 2547 de *Lecture Notes in Computer Science*. Springer-Verlag, 2002, p. 229–278.