Universidade Estadual de Campinas
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ENGENHARIA DE SISTEMAS

Roteamento do Tráfego na Internet: Algoritmos para Projeto e Operação de Redes com Protocolo OSPF

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de Doutora em Engenharia Elétrica.

por

Luciana Salete Buriol

Graduou-se em Informática Bacharelado na Universidade Federal de Santa Maria, Rio Grande do Sul (1994-1998). Cursou mestrado em Engenharia Elétrica, opção Automação, na Universidade Estadual de Campinas (1998-2000). Durante o doutorado foi pesquisadora visitante na AT&T Labs., New Jersey, Estados Unidos.

Orientador: **Prof. Dr. Paulo Morelato França** FEEC/UNICAMP
Co-orientador: **Dr. Mauricio Guilherme de Carvalho Resende** AT&T Labs. Research

Banca Examinadora

Prof. Dr. Celso da Cruz Carneiro Ribeiro	Universidade Federal Fluminense (UFF)
Prof. Dr. Christiano Lyra Filho	Universidade Estadual de Campinas (UNICAMP)
Prof. Dr. Cintia Rigão Scrich	Universidade Paulista (UNIP)
Prof. Dr. Felipe Martins Müller	Universidade Federal de Santa Maria (UFSM)
Prof. Dr. Ivanil Sebastião Bonatti	Universidade Estadual de Campinas (UNICAMP)
Prof. Dr. Paulo Morelato França	Universidade Estadual de Campinas (UNICAMP)

Campinas, 14 de novembro de 2003.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

B917r

Buriol, Luciana Salete
Roteamento do tráfego na internet: algoritmos para projeto e operação de redes com protocolo OSPF / Luciana Salete Buriol.–Campinas, SP:[s.n.], 2003.

Orientadores: Paulo Morelato França e Mauricio Guilherme de Carvalho Resende
Tese (Doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação

1.Algoritmos genéticos. 2. Otimização combinatória. 3. Heurística. Tráfego telefônico. I. França, Paulo Morelato. II. Resende, Mauricio Guilherme de Carvalho. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Resumo

Esta tese trata de problemas relacionados ao gerenciamento e projeto de redes Internet intra-domínio sob roteamento OSPF. No problema de gerenciamento, pesos devem ser atribuídos aos arcos da rede, objetivando minimizar o congestionamento do tráfego. A idéia é aproveitar ao máximo os recursos existentes, visando uma adequada qualidade de serviço. Uma busca local foi proposta e inserida em um algoritmo genético para resolver este problema. Para o projeto de redes, deve-se definir os pesos dos arcos e suas multiplicidades correspondentes, de tal forma que a soma total das multiplicidades seja minimizada. Neste contexto, a multiplicidade de um arco indica o número de replicações deste. A rede projetada deve ser apta para funcionar sem sobrecarga de arcos, até mesmo no caso de acontecer falha de um arco, roteador ou conjunto de arcos específicos. Um algoritmo genético foi prosposto para sua resolução. Um estudo extenso em algoritmos de caminhos mínimos dinâmicos também é apresentado. O uso desses algoritmos é determinante para uma boa eficiência dos métodos propostos para operação e projeto de redes OSPF. São comparados sete algoritmos clássicos, sendo que outros sete foram propostos aplicando-se a técnica de redução de pilha proposta nesta tese.

Abstract

This dissertation addresses control and design problems for intradomain Internet networks under OSPF protocol. In the control problem, arc weights must be defined, aiming to minimize network congestion. The goal is to improve user performance and make more efficient use of network resources, resulting in better quality of service. A local search procedure is proposed and embedded in a genetic algorithm. For network design, a set of arc weights must be defined, as well their corresponding multiplicities, aiming to minimize the sum of multiplicities. In this context, an arc multiplicity is defined to be the number of arc replicas. The resulting network should be able to transport the demand without link overload, even if an arc failure, router failure, or a failure of a specific set of arcs occurs. A genetic algorithm was proposed to solve this problem. An extensive study of dynamic shortest paths algorithms is also presented. The use of these algorithms is essential for an efficient implementation of the proposed methods for operation and design of a network running OSPF. Seven standard algorithms were compared and another seven were generated using a heap reduction technique proposed in this thesis.

DEDICO ESTE TRABALHO À MINHA FAMÍLIA. POR TER ENTENDIDO MINHA AUSÊNCIA, ME APOIADO NAS HORAS EM QUE PRECISEI, ME ABRAÇADO NOS MEUS MOMENTO DIFÍCEIS E COMPARTILHADO OS MOMENTOS FELIZES.

Agradecimentos

Ao meu orientador, Dr. Paulo França, pela orientação guiada por sabedoria, tranquilidade, apoio e entendimento, determinante para o desenvolvimento do meu trabalho. Pelo exemplo de pessoa, cientista e amigo, que além de despertar a minha mais profunda admiração e respeito, é para mim um exemplo a ser seguido.

Ao meu co-orientador Dr. Mauricio Resende, por ser um grande incentivador e amigo, pelo amadurecimento científico que obtive ao trabalharmos juntos, pela atenção que tive durante o período na AT&T e pela paciência (testada e retestada muitas vezes). Também por ter me ensinado a trabalhar com otimismo e tranquilidade.

Aos membros do departamento de Algoritmos e Otimização da AT&T Labs., em especial Dr. David Johnson, pelo carinho, respeito e confiança durante os quinze meses em que eu estive lá.

À minha vozinha, Dina, por estar sempre torcendo e rezando por mim.

À Simone, Márcia, Cris e Paulo, pela ajuda nas horas de maior dificuldade. Com eles aprendi o que é ser amigo de verdade.

À Ivy, minha grande amiga, com quem compatilhei minhas alegrias e tristezas durante meu período nos EUA.

Ao Alexandre, Giovanni, Aman e Anders, pela maravilhosa convivência como amigos durante o período em que morei em New Jersey.

Ao Alessandro Garcia, por ter me recebido no seu apartamento no Rio sempre em que fui trabalhar lá. Também pela amizade, regada com muito carinho.

Aos meus colegas da elétrica, principalmente do laboratório do DENSIS, com quem compartilhei momentos alegres e descontraídos.

À família Resende, por ter me recebido inúmeras vezes em sua casa, sempre com muito carinho e atenção.

Aos membros da banca examinadora, pelas correções e sugestões.

Aos professores e funcionários da FEEC/UNICAMP.

Ao Edison Arruda, Renata Aiex e Ricardo Oliveira pela ajuda em latex.

Ao Vinicius Jacques pela ajuda em Linux.

Ao Nelson Augusto, Ricardo Patara e Jennifer Rexford, pelas explicações detalhadas sobre a organização das redes da Unicamp, Brasil e da AT&T, respectivamente.

Ao ensino público e gratuito, pois sem ele este trabalho não teria sido desenvolvido.

Aos cantores e compositores do mundo todo, que alegraram meus dias mesmo nos períodos de tensão e fadiga.

À CAPES, CNPq e AT&T pelo apoio financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

AS	Autonomous System
BGP	Border Gateway Protocol
DARPA	Defense Advanced Research Projects Agency
DNS	Domain Name System
DSP	Dynamic Shortest Paths
EGP	Exterior Gateway Protocol
EIGRP	Enhanced Interior Gateway Protocol
FTP	File Transfer Protocol
GA	Genetic Algoritm
IDC	International Data Corporation
IGP	Interior Gateway Protocol
IGRP	Inter-Gateway Routing Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
ISP	Internet Service Provider
IS-IS	Intermediate System-to-Intermediate System
ITU	International Telecommunications Union
LACNIC	Latin American and Caribbean Internet Addresses Registry
LAN	Local Area Network
MA	Memetic Algoritm
MAN	Metropolitan Area Network
OSI	Open Systems Interconnections
OSPF	Open Shortest Path First
QoS	Quality of Service
RIP	Routing Information Protocol
RNP	Rede Nacional de Pesquisa
SMTP	Simple Mail Transfer
TCP	Transmission Control Protocol
TS	Tabu Search
VoIP	Voice Over IP
WAN	Wide Area Network
WSP	Weight Setting Problem

Prefácio

Comunicação sempre foi uma prática das civilizações, independente de época e cultura. Com o uso da escrita, a barreira que a distância representava foi superada, mas o tempo passou a ser um inimigo desta prática. A invenção do telégrafo por Samuel Morse, em 1838, deu início a uma nova fase nas comunicações. Desde então a comunicação através de sinais elétricos evoluiu tecnologicamente, tornando o tempo e distância quase imperceptíveis. O telefone, o rádio e a televisão são frutos dessa evolução. A tecnologia também alavancou formas de tratamento, processamento e armazenamento de dados. O computador pode ser considerado o maior avanço nesse sentido. A união dessas tecnologias originou a Internet, a rede mundial de computadores. Através dela, as pessoas comunicam-se usando recursos de texto, imagem, voz e som. Essa combinação de tecnologias, aliada ao seu fácil acesso e às inúmeras facilidades que seu uso proporciona, tem causado uma euforia na área de telecomunicações nos últimos anos. Mesmo em tempos de crise na economia mundial, o número de computadores sendo conectados à Internet não pára de crescer. Até duas décadas atrás a Internet era usada principalmente por pesquisadores e militares. Hoje, ela tornou-se indispensável para uma parcela considerável da população, assim como para a maioria das empresas e centros de pesquisa. As dimensões desse crescimento podem ser inferidas da Tabela 1 [1]
.

Além do crescimento acelerado do número de usuários, a quantidade e o tamanho dos dados que estes trocam têm aumentado consideravelmente. O emprego de recursos multimídia, como som, áudio e vídeo, assim como figuras de alta resolução e documentos extensos, tem sido prática comum entre os usuários da Internet. Além disso, uma quantidade enorme de serviços vêm sendo disponibilizados via Internet, incentivando o surgimento de novos usuários, bem como o grande

[1]Fonte: Internet Software Consortium (http://www.isc.org/). Os dados são do mês de janeiro de cada ano, com exceção de 1990, quando os dados foram coletados em setembro.

Tabela 1: Número de computadores conectados à Internet nas últimas duas décadas.

Ano	*#hosts*	*Ano*	*#hosts*	*Ano*	*#hosts*
1989	80.000	1994	2.217.000	1999	43.230.000
1990	313.000	1995	4.852.000	2000	72.398.092
1991	376.000	1996	9.472.000	2001	109.574.429
1992	727.000	1997	16.146.000	2002	147.344.723
1993	1.313.000	1998	29.670.000	2003	171.638.297

volume de dados por eles movimentados. O instituto de pesquisas *International Data Corporation* (IDC) prevê que o volume de tráfego gerado pelos usuários domésticos de todo o mundo irá dobrar anualmente pelos próximos cinco anos, partindo de 180 petabits por dia em 2002, para 5175 petabits por dia até o fim de 2007.

Inevitavelmente, com o aumento do número de usuários e do volume de dados por eles movimentado, geram-se novos problemas de comunicação. O aumento do tráfego inevitavelmente gera congestionamento, caso medidas preventivas e de controle não sejam adotadas. Dentre estas medidas, o projeto e manutenção de uma rede de proporções adequadas, bem como a utitilização eficiente de seus recursos, podem ser apontadas como as principais.

Dessa forma, o tratamento desses fatores caracteriza dois problemas combinatórios: um de projeto e outro de gerenciamento de recursos, ambos relacionados com o roteamento do tráfego na rede. A área que trata desses problemas é a engenharia de tráfego, que recentemente vem atraindo uma maior atenção da comunidade acadêmica.

Quando dados são transmitidos via Internet, a informação é acondicionada em pequenos pacotes de dados que contêm um cabeçalho com um conjunto de informações necessárias para a sua correta transmissão ao longo da rede. Uma dessas informações é o endereço IP (*Internet Protocol*) da máquina a que se destina. Cada roteador, ao receber um pacote, faz a leitura das informações contidas nele, assim como consulta a tabela de roteamento (disponível em cada roteador), para identificar qual o próximo roteador da rota. O envio destes pacotes respeita uma organização estrutural dos roteadores na Internet.

A Internet é dividida em muitos domínios de roteamento, chamados de sistemas autônomos (*Autonomous Systems - AS*). Estes AS interagem para controlar o envio de tráfego IP. O objetivo da engenharia de tráfego intra-domínio [FRT02] consiste em fazer uso eficiente dos recursos da

rede interna a um AS. Para o roteamento entre AS, é usado o *Border Gateway Protocol* (BGP) [SARK02].

O roteamento dentro de um AS depende do protocolo de roteamento que está sendo usado. *Interior Gateway Protocols* (IGPs), tais como o OSPF (*Open Shortest Path First*) e IS-IS (*Intermediate System-Intermediate System*) são geralmente usados para selecionar caminhos nos quais o tráfego interno a um AS será roteado. Estes protocolos de roteamento direcionam o tráfego baseados nos pesos dos arcos, designados pelo operador da rede. Cada roteador dentro de um AS calcula os caminhos mínimos entre roteadores e cria sua tabela de roteamento, usada para direcionar cada pacote IP para o próximo roteador no caminho para seu destino final.

O protocolo OSPF define que a cada arco seja designado um peso inteiro no intervalo de 1 até 65535 ($= 2^{16} - 1$). O custo de um caminho (rota) é a soma dos pesos do arcos que o compõem. OSPF estabelece que cada roteador r calcule o grafo de caminhos mínimos com r como raiz [Int94]. Este grafo fornece as rotas de custo mínimo para todos os roteadores destino no AS. No caso de roteadores com múltiplos caminhos mínimos, o protocolo OSPF estabelece que a carga passante por cada roteador seja igualmente dividida entre todos os arcos adjacentes pertencentes ao grafo de caminhos mínimos [Moy98]. Ainda, o OSPF requer que cada roteador troque informações de roteamento com todos os outros roteadores no AS. A informação sobre a topologia da rede completa é necessária para calcular os caminhos mínimos.

O cálculo dos caminhos mínimos pelo OSPF é realizado considerando a designação de pesos às ligações (arcos) que conectam os roteadores de um AS. Fixando cada roteador como a raiz, o OSPF calcula, em seguida, os caminhos de custo mínimo. Os pesos devem ser escolhidos adequadamente, de forma a satisfazer um critério técnico ou econômico, como por exemplo, a minimização do congestionamento da rede.

Tendo-se uma rede com topologia e capacidade suficientes para rotear os dados sem sobrecarga nos arcos, a ocorrência de congestionamento dependerá basicamente da atribuição dos pesos aos arcos. No caso em que a topologia deva ser expandida, novamente tem-se uma dependência do conjunto de pesos dos arcos dessa rede. Em ambos os casos, o roteamento deve ser realizado e, para tanto, os caminhos mínimos têm de ser calculados considerando os pesos dos arcos.

Esta tese tem o objetivo de abordar alguns dos problemas oriundos do crescente aumento do

tráfego na Internet. Basicamente, os problemas abordados são referentes ao roteamento de dados usando os protocolos OSPF e IS-IS. A tese encontra-se dividida em 4 capítulos.

O Capítulo 1 apresenta uma explanação de uma rede, objetivando contextualizar o roteamento de dados dentro da organização e funcionamento da Internet. Inicialmente, a arquitetura de redes OSI/ISO e Internet TCP/IP são apresentadas. A seguir, maiores detalhes sobre a camada de rede da Internet são abordados, com ênfase em conceitos relacionados ao roteamento. Os principais protocolos de roteamento baseados em roteamento vetorial e por estado de enlace são introduzidos, e maiores detalhes sobre os protocolos OSPF e IS-IS são fornecidos. O capítulo finaliza com uma discussão sobre abordagens adotadas para evitar e controlar o congestionamento na Internet.

O problema de designação dos pesos aos arcos de uma rede (*Weight Setting Problem* - WSP), relacionado com a operação de redes, é descrito com detalhes no Capítulo 2. O objetivo é encontrar um vetor de pesos, associados aos arcos da rede, de forma que o congestionamento da rede seja minimizado quando o roteamento é realizado. O protocolo adotado é o OSPF ou IS-IS, considerando uma rede interna a um sistema autônomo. São apresentadas algumas das características matemáticas deste problema e dois modelos matemáticos para a sua solução. Ainda, um algoritmo memético para resolver o WSP é proposto. O algoritmo consiste, basicamente, na inserção de uma busca local a um algoritmo genético já existente. Resultados computacionais são apresentados considerando o algoritmo memético implementado, bem como é fornecida uma comparação com outras heurísticas e metaheurísticas aplicadas a este problema. O capítulo finaliza com uma discussão dos resultados computacionais.

Um problema de projeto de dimensionamento da rede envolvendo o roteamento OSPF também é objeto de estudo desta tese. O objetivo é dimensionar as capacidades das ligações, minimizando o número de arcos necessários (multiplicidades) para suprir tais capacidades, de modo a atender as demandas de tráfego com mínimo investimento. O Capítulo 3 inicia com a apresentação do problema de designação das multiplicidades dos arcos de uma rede. Em seguida, um algoritmo genético é proposto para resolução deste problema. Novamente o roteamento é baseado no protocolo OSPF ou IS-IS, para uma rede interna a um AS. Uma série de resultados computacionais são apresentados e analisados. Uma comparação com resultados de outros métodos não foi realizada, visto que não há publicação de outros métodos para este problema.

O Capítulo 4 apresenta um extenso estudo sobre algoritmos de atualização de caminhos mínimos, mais conhecidos como algoritmos de caminhos mínimos dinâmicos (*Dynamic Shortest Paths* - DSP). Estes algoritmos são usados para atualizar os caminhos mínimos após a mudança do peso de um arco. A importância deste capítulo está diretamente relacionada ao fato de que tais algoritmos desempenham um papel fundamental na eficiência global dos métodos de resolução utilizados para resolver os problemas abordados nos Capítulos 2 e 3 desta tese. Alguns desses algoritmos são usados pelo procedimento de busca local do algoritmo memético do Capítulo 2 e no procedimento de simulação de falhas da topologia do problema do Capítulo 3. Sua boa eficiência é fundamental no desempenho dos métodos implementados. Além de apresentar o pseudo-código dos algoritmos mais conhecidos para o problema, uma técnica para redução do tamanho da pilha foi aplicado a cada um deles. O capítulo finaliza com uma explanação e análise dos resultados computacionais.

A tese finaliza com a apresentação das conclusões gerais.

Capítulo 1

Roteamento OSPF no Contexto da Internet

Uma rede de computadores é formada por um conjunto de módulos processadores capazes de trocar informação e compartilhar recursos via um sistema de comunicação. Os módulos processadores são, em sua maioria, computadores. Estes podem ser simples PCs, roteadores, servidores, além de outros computadores com finalidades específicas. Por questões de simplicidade, nesta tese usa-se o termo *host* designando um computador conectado à Internet. O sistema de comunicação é formado por um arranjo topológico que interliga os módulos processadores através de enlaces (meios de transmissão) e de um conjunto de regras com o fim de organizar a comunicação (protocolos).

As redes de computadores podem ser locais (*Local Area Network* - LAN), metropolitanas (*Metropolitan Area Network* - MAN) ou geograficamente distribuídas (*Wide Area Network* - WAN). As rede locais e metropolitanas permitem a interconexão de equipamentos de comunicação de dados numa pequena região, enquanto que as redes geograficamente distribuídas interligam máquinas geograficamente dispersas. As LANs são redes privadas e, em geral, contidas num prédio, enquanto que uma MAN pode abranger uma cidade ([Tan97]). As LANs e MANs utilizam tecnologias semelhantes. Uma MAN, em geral, abrange uma distância de até 25 km, possui altas taxas de transmissão e baixas taxas de erro. Ao contrário, uma WAN tem custo de transmissão elevado (circuitos para satélites e enlaces de microondas). Por isso, em geral, é pública e gerenciada por grandes operadoras (públicas ou privadas).

A Internet é caracterizada como uma WAN e foi criada com o objetivo de interligar redes de tecnologias variadas. Atualmente existem milhões de computadores conectados à Internet no mundo

inteiro. Certamente a organização da Internet é mais uma obra brilhante da inteligência humana. Aos olhos do usuário, a comunicação é geralmente rápida e transparente. Mas seu funcionamento depende de uma organização estrutural, física e lógica muito complexa. A Figura 1.1 ilustra, à esquerda, uma visão simplificada da estrutura da Internet. A figura da direita, embora ainda muito simplificada, possibilita ter uma visão da estrutura geral da Internet. A Internet é composta por sistemas autônomos (AS) representados por pequenas nuvens. Estes são interconectados pelos roteadores. Os usuários acessam a Internet através dos *hosts*.

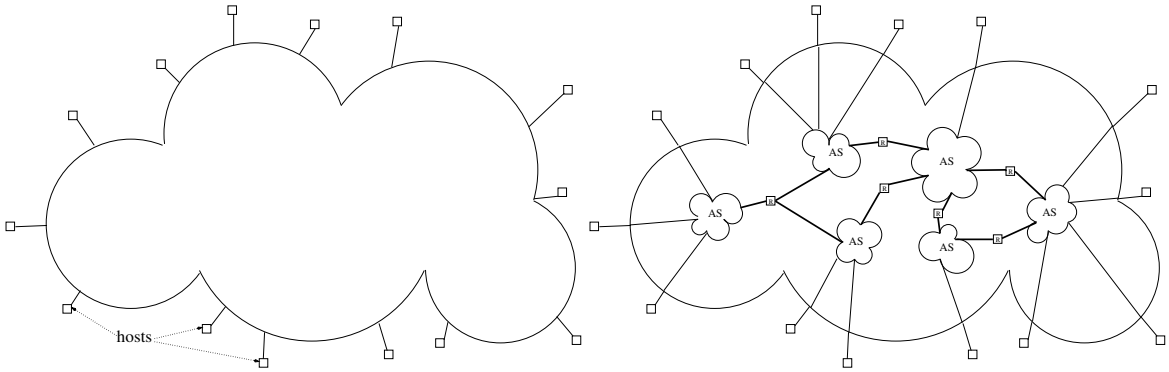

Figura 1.1: Uma visão simplificada (à esquerda) e a estrutura geral (à direita) da Internet.

A Internet é composta pela interligação de várias redes. Cada rede é ligada à Internet através de um roteador (*router*). Dessa forma, a Internet é interconectada por roteadores. Um roteador é um computador com o fim específico de tratar e direcionar dados enviados inter-redes. Segundo Moy [Moy98], originalmente o termo *gateway* era usado para designar o que hoje se chama *router*. Atualmente o termo *gateway* é usado para computadores de níveis mais altos do modelo OSI (*gateways* de nível de aplicação, *gateways* de e-mail, etc.) A conexão entre os roteadores é viabilizada através de meios de transmissão. Os dados consistem em informação de texto, imagem ou som, e são divididos em unidades menores chamados pacotes. O caminho que os pacotes percorrem na rede, desde sua origem até seu destino, é indicado pelos roteadores. A rota dos pacotes é informada aos roteadores através de algoritmos de roteamento, que usam regras especificadas pelo protocolo de roteamento em vigência. Dependendo do protocolo, os pacotes são transmitidos por um único ou múltiplos caminhos. Os pacotes são agrupados no roteador destino para formar os dados originais. A todo esse processo de transmissão de dados dá-se o nome de roteamento. O conjunto de

regras que define a forma que os pacotes são roteados é chamado de protocolo de transmissão.

Os computadores conectados a uma rede são ligados por linhas de comunicação, também referenciados por *arcos* nesta tese. Uma série de tecnologias de conexão são usadas atualmente: linhas de telefone, segmentos de Ethernet, packet radio (utiliza ondas de rádio por difusão), linhas de satélite, frame relay, ATM, FDDI, IEEE 802.5 Token Ring, SMDS, entre outras.

As ligações físicas entre computadores podem ser ponto a ponto, caracterizada pela presença de apenas dois pontos de comunicação, ou multiponto (três ou mais pontos de comunicação). A conexão dos computadores em rede pode usar diversas topologias, tais como em anel, em estrela, em barra, entre outras.

Este capítulo apresenta uma explanação de uma rede, objetivando contextualizar o roteamento de dados dentro da organização e funcionamento da Internet. Inicialmente, a arquitetura de redes OSI/ISO e Internet TCP/IP são apresentadas. A seguir, maiores detalhes sobre a camada de rede da Internet são abordados, com ênfase em conceitos relacionados ao roteamento. Os principais protocolos de roteamento baseados em roteamento vetorial e por estado de enlace são introduzidos, e maiores detalhes sobre o OSPF e IS-IS são fornecidos. O capítulo finaliza com uma discussão sobre abordagens adotadas para evitar e controlar o congestionamento na Internet.

1.1 Arquitetura de redes de computadores

A arquitetura de uma rede é composta por camadas, interfaces e protocolos. Cada camada oferece um conjunto de serviços à camada superior, mas não estabelece a forma como estas operações são implementadas. Devido a esta estrutura hierárquica, as camadas também são denominadas de *níveis*. O número, nome, conjunto de funções e serviços, e protocolo de cada camada variam de uma arquitetura de rede para outra.

A comunicação entre máquinas é possível entre níveis de rede iguais. A comunicação entre níveis é realizada por programas que implementam os protocolos, os quais definem as regras do funcionamento dos serviços. Na realidade, a comunicação entre níveis não é feita horizontalmente. Os dados da máquina origem descem verticalmente do nível *i* ao nível físico, onde então a comunicação é feita horizontalmente com o nível *1* da máquina destino, e depois sobem verticalmente ao nível *i* desta. O limite entre níveis adjacentes é chamado de interface. Uma interface de uma

camada informa como os processos acima dela podem acessá-la.

Inicialmente, cada fabricante desenvolveu sua própria arquitetura, denominadas de proprietárias. Com a necessidade de comunicação entre computadores de fabricantes distintos, a definição de uma arquitetura comum se tornou necessária. Com este fim, a *International Organization for Standardization* (ISO) definiu o modelo de referência denominado *Open Systems Interconnections* (OSI). Segundo Soares et al. [SLC95], a arquitetura *ISO/OSI* foi originalmente projetada para redes de longa distância, embora ela possa ser também utilizada em redes locais.

Com o objetivo de interligar redes heterogêneas (locais, metropolitanas e de longa distância), desenvolveu-se a arquitetura *Internet TCP/IP*, também referenciada somente por *Internet*. Esta se tornou o padrão mundial para interconexão de sistemas abertos. Os acrônimos TCP (*Transmission Control Protocol*) e IP (*Internet Protocol*) são os dois principais protocolos dessa arquitetura.

As próximas seções descrevem com mais detalhes as arquiteturas ISO/OSI e Internet TCP/IP. O roteamento OSPF, objeto de estudo desta tese, é realizado na arquitetura TCP/IP. Por esse motivo, esta última é analisada em profundidade.

1.1.1 O modelo OSI da ISO

O desenvolvimento do modelo OSI (*Open System Interconnections*) da ISO (*International Organization for Standarization*), atualmente conhecida como *International Telecommunications Union* (ITU), foi uma primeira proposta para padronização internacional dos protocolos usados nas camadas de rede, de forma a criar um sistema aberto à comunicação.

O modelo OSI não é um arquitetura propriamente dita, e sim apenas um modelo de referência, pois não especifica os serviços e protocolos que devam ser usados em cada camada. O modelo apenas determina as camadas e o que cada camada pode fazer. O modelo OSI possui sete níveis (camadas), resumidamente descritos a seguir:

- *Nível de aplicação*: nesse nível são definidas funções de gerenciamento e mecanismos genéricos que servem de suporte à construção de aplicações distribuídas.

- *Nível de apresentação*: é função do nível de apresentação realizar transformações adequadas aos dados, tais como compressão, criptografia, conversão de padrões de rede, etc.

- *Nível de Sessão*: os principais serviços prestados pelo nível de sessão são o gerenciamento de *tokens*, controle de diálogo e gerenciamento de atividades.

- *Nível de Transporte*: é função do nível de transporte o controle da transmissão dos pacotes pela rede, fazendo com que estes cheguem a seu destino, definindo as medidas a serem tomadas em relação à pacotes perdidos ou com problemas, e que sejam recebidos na ordem correta. Também o controle de fluxo é feito por este nível, evitando que o receptor receba dados numa taxa maior que sua capacidade.

- *Nível de Rede*: algumas funções deste nível são o endereçamento, roteamento e controle do congestionamento.

- *Nível de Enlace dos Dados*: tem como principal função detectar e, em alguns casos, corrigir os erros ocorridos no nível físico. Além disso, é no nível de enlace que o fluxo entre transmissor e receptor é controlado para evitar que o receptor não seja sobrecarregado.

- *Nível Físico*: fornece as características mecânicas, elétricas, funcionais e de procedimento para ativar, manter e desativar conexões físicas para a transmissão de bits entre entidades de nível de enlace, possivelmente através de sistemas intermediários.

Apesar do modelo OSI da ISO ser usado em redes de computadores, na maioria LANs, o modelo TCP/IP se tornou o padrão mundial para interconexão de sistemas abertos. A próxima seção apresenta esta arquitetura.

1.1.2 A arquitetura Internet TCP/IP

Não existe uma arquitetura única que satisfaça as necessidades de todos os usuários. Dessa forma, comunidades de usuários podem usar uma arquitetura que atenda a seus anseios, formando redes distintas, que são interconectadas pela Internet. Foi com este objetivo, o de interligar redes com tecnologias distintas, que a arquitetura Internet TCP/IP foi desenvolvida. Os protocolos dessa arquitetura oferecem uma solução simples, porém bastante funcional para o problema de interconexão de sistemas abertos. A arquitetura TCP/IP é organizada em quatro camadas:

- *Nível de aplicação*: através do nível de aplicação, um usuário faz uso de programas para acessar serviços na inter-rede, tais como serviço de envio de mensagens (*Simple Mail Transfer* - SMT), serviço de transferência de arquivos (*File Transfer Protocol* - FTP), serviço de terminal virtual (TELNET) e serviço de mapeamento de nomes em endereços na rede (*Domain Name System* - DNS).

- *Transporte*: O nível de transporte tem a finalidade de permitir a comunicação fim-a-fim (comunicação entre origem e destino, e não entre máquinas sucessivas ao longo do caminho) entre aplicações, de forma semelhante de como é feita no modelo OSI.

- *Inter-Redes*: a principal finalidade da camada inter-redes é permitir que os *hosts* enviem pacotes para qualquer rede, além de garantir que eles sejam transmitidos independentemente. Essa camada define o formato de pacote e o protocolo IP. Nela, o roteamento e o controle do congestionamento são tratados e, por isso, sua função se assemelha à camada de rede do modelo OSI.

- *Host/Rede*: é a camada mais baixa dessa arquitetura. Nela é definido o protocolo que faz a comunicação com a rede para possibilitar o envio de pacotes.

O desenvolvimento dessa arquitetura teve como patrocinador a *Defense Advanced Research Projects Agency* (DARPA). A arquitetura basicamente trata de serviço de transporte orientado à conexão (comutação de circuitos), fornecido pela *Transmission Control Protocol* (TCP), e serviço de rede não orientado à conexão (comutação de pacotes), fornecido pela *Internet Protocol* (IP). O serviço de transporte orientado à conexão é dito *confiável*, pois os pacotes são entregues na mesma sequência em que foram gerados e uma série de medidas são tomadas para que todos os pacotes cheguem ao destino. No serviço de transporte sem conexão, cada pacote é tratado de forma individual e entregue ao destino através do caminho mais conveniente, definidos pelos algoritmos de roteamento. Este serviço é dito *não confiável*, pois não há garantia de entrega e nem garantia de manter a ordem em que os pacotes foram gerados. Como cada pacote é independente dos outros, a ordem em que os pacotes chegam pode não ser a mesma em que são enviados. O protocolo IP apenas envia os pacotes, ficando a tarefa de sua reordenação a cargo do protocolo TCP. Comparando as duas formas de transmissão, o serviço orientado à conexão é mais seguro e garante capacidade

suficiente para transmissão (uma vez que a conexão é estabelecida), enquanto que o serviço não orientado à conexão é mais barato e, em geral, bastante eficiente. Devido à combinação de preço e eficiência, entre outros motivos, o serviço não orientado à conexão tem se tornado mais popular.

As redes telefônicas implementam os dois tipos de serviços, mas pouco a pouco está havendo uma migração para comutação de pacotes e possivelmente, num futuro próximo, as redes telefônicas sejam implementadas totalmente via comutação de pacotes. O uso do protocolo IP para transmissão de dados telefônicos é referenciado como *Voice Over IP* (VoIP). Segundo Black [Bla99], o protocolo IP não é particularmente atraente para a telefonia, visto que ele foi projetado para transportar dados da Internet. Sua utilização tem motivação puramente prática, visto que a Internet já se expandiu pelo mundo inteiro, tornando-a uma plataforma conveniente para o suporte de tráfego telefônico.

O serviço de rede não orientado à conexão é um serviço da camada de rede da Internet. O IP (*Internet Protocol*) é o protocolo de camada de rede responsável pela comutação de pacotes na Internet. Na transmissão dos pacotes, os roteadores têm informação para onde e como eles devem ser enviados. Este procedimento é chamado de roteamento e os protocolos que definem as regras para calcular os caminhos, nos quais os dados trafegam, são chamados de protocolos de roteamento. A próxima seção tem o objetivo de descrever com um pouco mais de detalhes o roteamento IP.

1.2 Arquitetura de roteamento da Internet

A operação da Internet não é dependente de nenhuma entidade de controle centralizado. Um *backbone* principal é construído por linhas de grande largura de banda e roteadores rápidos. Conectadas ao *backbone* estão os ASs. Um AS pode pertencer a um provedor de Internet (*Internet Service Provider* - ISP), a empresas, a organizações comerciais, governamentais, de pesquisa ou acadêmicas. Cada AS possui roteadores externos, que possibilitam a conexão com outros ASs, e internos, que agregam tráfego local e o envia para um roteador externo. Um roteador difere de outros computadores por possuir mais de uma interface de rede, possibilitando a transferência de dados entre elas.

O Apêndice A anexo ao final da tese aborda, com mais detalhes, a estrutura atual da Internet e como o Brasil se encaixa neste contexto.

Para que o roteamento seja viabilizado, cada conexão à Internet é identificada por um número. O número dos roteadores de origem e destino devem ser adicionados ao conteúdo de cada pacote. A próxima seção aborda o esquema de endereçamento IP, para que posteriormente seja estudado o roteamento em si.

1.2.1 Endereçamento IP

Endereçamento e roteamento estão intrinsecamente relacionados. Para o envio de um pacote, o protocolo IP precisa ter um esquema de endereçamento que identifica os roteadores de origem e destino. Um endereço IP é um número que identifica unicamente no mundo cada conexão à Internet. Um endereço IP possui 32 bits, normalmente escritos em notação decimal com pontos. Nesta notação, cada um dos quatro bytes é escrito em notação decimal (de 0 a 255), e estes são separados por pontos. Por simplicidade, diz-se que um endereço IP identifica um *host*, mas na verdade ele identifica uma conexão à Internet. Um endereço IP consiste em três partes:

- um endereço de rede, o qual identifica unicamente uma organização;
- um endereço de sub-rede, o qual identifica uma sub-rede dentro da organização;
- uma conexão da sub-rede.

O tamanho de cada campo depende da classe da rede em que o *host* se encontra. Existem três classes de endereços IP: A, B e C. Esta classificação leva em conta o número máximo de *hosts* que elas podem conter. A classe A identifica redes de grande porte, com até 2^{24} = 16.777.216 *hosts*; redes da classe B podem conter até 2^{16} = 65536 *hosts* e redes da Classe C podem conter até 2^{8} = 256 *hosts*. O primeiro número do endereço IP identifica a classe da rede. As faixas de endereços de *hosts* são:

- Classe A: 1.0.0.0 a até 127.255.255.255;
- Classe B: 128.0.0.0 a 191.255.255.255;
- Classe C: 192.0.0.0 a 223. 255.255.255;
- Endereços de *multicast*: 224.0.0.0 a 239.255.255.255;

- Reservado para uso futuro: 240.0.0.0 a 247.255.255.255.

Para facilitar o acesso do usuário à Internet, todo endereço IP possui também uma versão em forma de texto (domínio). O servidor de DNS (*Domain Name System*) tem a função de mapear endereços de domínios em endereços IP. Assim, ao consultar um página na Internet, o usuário fornece um endereço em forma de texto e o servidor de DNS obtém o endereço IP correspondente.

Embora o endereçamento de 32 bits do IPv4 (versão 4 do IP) seja suficiente para a dimensão atual da Internet, o aumento para endereços de 128 bits é uma das grandes mudanças na versão 6 do IP, chamada de IPv6.

Um AS caracteriza um domínio de roteamento (*routing domain*) pois cada sistema autônomo recebe uma faixa de endereços IP. Ao enviar um pacote, este deve conter o endereço do *host* destino. Cada roteador no caminho do pacote para o destino verifica tal endereço para saber qual o próximo roteador da rota. Este procedimento, bem como os protocolos de roteamento, são explorados na próxima seção.

1.3 Roteamento IP

Os dados transmitidos na Internet são organizados em *strings* de bits chamados pacotes. Uma mensagem pode ser subdivida em vários pacotes que são transmitidos independentemente via Internet. Estes pacotes são fragmentados ainda mais, caso sua rota inclua uma rede que estabeleça um tamanho de pacote menor que a rede em que o pacote foi montado. Isso acontece porque o tamanho máximo de um pacote varia de uma rede para outra. O roteamento dos pacotes depende das regras estabelecidas por cada protocolo de roteamento. São estes protocolos que definem a rota que os pacotes devem seguir para chegar ao destino. Durante a transmissão de um pacote, este passa por roteadores intermediários que verificam o endereço de destino (informado no pacote), consultam sua tabela de roteamento e o transmitem para o próximo roteador na rota. Cada roteador possui sua tabela de roteamento. A informação que consta na tabela é, para cada destino, qual o próximo roteador para o qual os pacotes devem ser enviados.

O destinatário pode ser um único *host* (*unicast*), um grupo de *hosts* (*multicast*) ou todos os *hosts* de uma determinada rede (*broadcast*). Os protocolos também devem estabelecer o procedimento

a ser tomado em caso de mudanças repentinas na infra-estrutura da Internet, tais como linhas de transmissão sendo interditadas ou reativadas, falha de roteadores e outras mudanças estruturais.

Os protocolos de roteamento dentro de um AS e inter-AS são diferentes. Roteadores dentro de um AS trocam informações de roteamento através de um protocolo comum chamado de *Protocolo de Gateway Interno* (*Interior Gateway Protocol* - IGP). Roteadores que fazem comunicação entre AS, o fazem através de um protocolo de roteamento chamado *Protocolo de Gateway Externo* (*Exterior Gateway Protocol* - EGP).

Alguns dos IGPs atualmente mais usados na Internet são: *Open Shortest Path First* (OSPF), *Intermediate System-to-Intermediate System* (IS-IS), *Routing Information Protocol* (RIP), *Inter-Gateway Routing Protocol* (IGRP) e *Enhanced Interior Gateway Protocol* (EIGRP). O protocolo EGP usado atualmente na Internet é o *Border Gateway Protocol* (BGP).

A divisão dos protocolos em IGPs e EGPs é porque eles possuem objetivos diferentes. Dentro de um AS o objetivo maior é calcular rotas eficientes (evitam tráfego intenso, transmissão rápida dos dados, etc), além de atualizar rapidamente as rotas quando há uma mudança na rede (falha de uma linha de transmissão, interrupção temporária do funcionamento de um roteador, etc). Os roteadores EGP têm maior preocupação com questões políticas, econômicas e de segurança, sendo estas manualmente configuradas nos roteadores, não fazendo parte do protocolo em si. Essas diferenças fazem com que IGPs e EGPs, na sua maioria, usem tecnologias de roteamento diferentes.

A maioria dos IGPs usa roteamento por *estado de enlace*, enquanto que os EGPs usam roteamento baseado em *vetor de distância*. Estas duas tecnologias são descritas a seguir. Como esta tese está relacionada com algoritmos baseados em roteamento por estado de enlace, apenas uma breve descrição dos algoritmos com vetor de distâncias é fornecido, enquanto o outro é estudado em maior profundidade.

1.3.1 Roteamento por vetor de distância

O termo *roteamento por vetor de distância* (*Distance Vector Routing*) também é encontrado na literatura como *roteamento por distância vetorial* ou *Bellman-Ford algorithm*, este último devido ao algoritmo de Bellman-Ford ser usado para calcular os caminhos mínimos para o roteamento. O algoritmo de roteamento por vetor de distância mantém um vetor que informa a menor distância

conhecida a cada destino, assim como o próximo roteador na rota mínima para o destino. O melhor caminho pode estar relacionado com várias medidas, sendo que o número de roteadores na rota (*hop count*) é a mais utilizada. Os valores das tabelas são calculados inicialmente por cada roteador e, num segundo passo, os roteadores trocam informação entre si a fim de atualizar os caminhos mínimos.

Este roteamento é muito simples, mas na prática pode ser lento. Protocolos que usam esta tecnologia são o RIP, IGRP e o BGP. O RIP usa *hop count* como métrica, sendo que o tamanho máximo que uma rota é 15. O IGRP é o protocolo de roteamento de propriedade da CISCO. Este protocolo é baseado no RIP, com a diferença principal de usar como peso w de um arco um valor no intervalo [1, 65535], correspondendo a uma combinação de até cinco métricas para seu cálculo.

O roteamento por estado de enlace é utilizado pelos protocolos OSPF e IS-IS e é objeto de estudo da próxima subseção.

1.3.2 Roteamento por estado de enlace

Os algoritmos por estado de enlace (*link-state algorithms*) usam uma base de informações (*link-state database*) replicada em cada roteador. Cada roteador cria uma tabela contendo informações de sua conexão na rede (daí o nome *link-state*), tais como arcos adjacentes, peso de cada arco e roteadores vizinhos. Após sua criação, os roteadores distribuem suas tabelas para os outros roteadores do domínio. Desta forma, cada roteador possui informação completa da rede e possibilita o cálculo dos caminhos mínimos dele para cada destino.

O custo de uma rota é igual à soma dos custos dos arcos no seu caminho. A rota é igual ao caminho mínimo entre os roteadores de origem e destino. Os algoritmos de caminhos mínimos mais usados são o algoritmo Dijkstra e o algoritmo Bellman-Ford. Na realidade, o que é feito na prática é calcular, para cada destino, a árvore de caminhos mínimos de cada roteador direcionados para o roteador destino. Para uma rede composta por $|V|$ roteadores e $|E|$ arcos, supondo que cada roteador seja destino de fluxo, são calculadas $|V|$ grafos de caminhos mínimos. Os grafos de caminhos mínimos usados para roteamento são caracterizados por possuir arcos direcionados e por ter como raiz o nó destino, em vez do nó fonte. Uma árvore de caminhos mínimos não é necessariamente exclusiva, visto que um nó qualquer da rede pode possuir múltiplos caminhos

mínimos ao nó de destino. Um grafo de caminhos mínimos é composto por todos os caminhos mínimos. Veja a Figura 1.2. Esta figura apresenta uma rede com suas duas árvores e o grafo de caminhos mínimos.

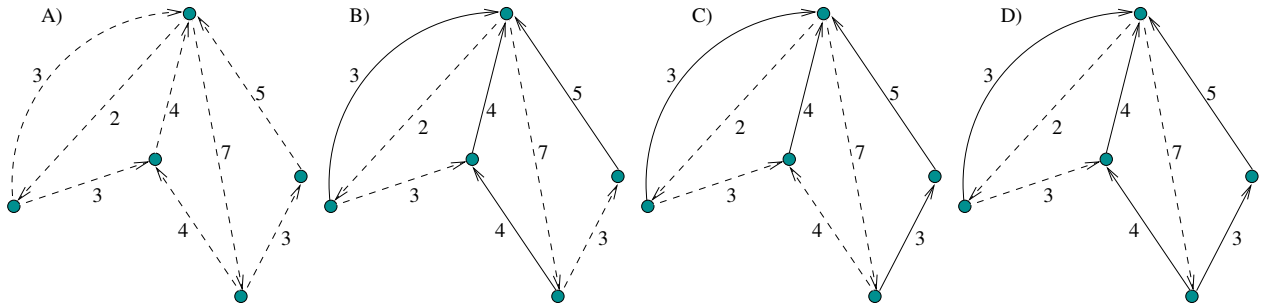

Figura 1.2: Uma rede (A), suas duas árvores de caminhos mínimos (B e C) e seu grafo de caminhos mínimos (D). A raiz das árvores e do grafo de caminhos mínimos é o nó superior.

Os algoritmos de roteamento diferem uns dos outros em termos de como construir e utilizar os grafos de caminhos mínimos. O valor do custo de cada arco depende do protocolo que está sendo usado. Em geral, atribui-se um valor a cada arco, associado a alguma métrica, como distância geográfica em quilômetros, custo de comunicação, tráfego médio, largura da banda, comprimento médio da fila e retardo detectado, entre outras. O valor também pode ser uma função de mais de um valor de medida. Ainda, pode-se ter várias tabelas de caminhos mínimos, cada uma associada a uma medida.

Ainda existem divergências sobre qual dos dois algoritmos de roteamento (com vetor de distância ou por estado de enlace) é melhor, pois seu desempenho varia dada a topologia e condições de tráfego da rede. Segundo Moy ([Moy98]), na maioria dos casos os algoritmos por estado de enlace têm melhor desempenho pois a atualização dos caminhos mínimos é calculada mais rapidamente dada uma modificação repentina da topologia, além de que o conhecimento da topologia geral da rede por cada roteador possibilita uma maior flexibilidade no uso dessa informação. Ainda, o tráfego pode ser melhor controlado. Desta forma, segundo Gross [Gro92], o protocolo recomendado para roteamento interno de um AS na Internet é o *Open Shortest Path First* - OSPF, que usa roteamento por estado de enlace. Além dele, o RIP e o IS-IS também são bastante utilizados. O IS-IS também faz roteamento por estado de enlace.

Nas próximas duas seções, alguns detalhes dos protocolos OSPF e IS-IS serão abordados.

Open Shortest Path First - OSPF

O protocolo IGP recomendado para uso na Internet é o *Open Shortest Path First* (OSPF). O OSPF foi projetado com o objetivo de substituir o então protocolo vigente, o RIP, e sanar os problemas que este último vinha apresentando. Por isso a decisão de criar um protocolo com código aberto e amplamente divulgado na literatura, daí a origem do *Open* do nome do protocolo. O *Shortest Path First* é porque, independente da métrica adotada para atribuir pesos aos arcos, o menor caminho é sempre usado para fazer o roteamento. O OSPF tem uma série de características que o torna flexível para se adaptar melhor à rede em que é implementado. Abaixo, algumas das principais características do OSPF:

- o peso w de cada arco é um valor inteiro de 16 bits, ou seja, w =[1, 65535];
- não há limite no custo máximo de uma rota;
- o protocolo não estabelece como os pesos dos arcos devem ser atribuídos, ficando a cargo do operador da rede tal decisão;
- o OSPF faz roteamento multicaminho, caracterizado pelo balanceamento de carga. Ou seja, a carga passante por cada roteador, e com um mesmo roteador destino t, é dividida igualmente por todos os seus arcos saintes presentes no grafo de caminhos mínimos.

Como o peso de um arco é definido pelo operador, este pode usar qualquer métrica ou combinação de métricas que achar mais conveniente. OSPF atualiza a topologia em caso de falha de arco ou roteador, mas não em caso de congestionamento.

O protocolo IS-IS também faz roteamento por estado de enlace e balanceamento de carga, mas difere do protocolo OSPF em muitos aspectos. A seguir, o protocolo IS-IS é abordado.

Intermediate System-to-Intermediate System (IS-IS)

Assim como o OSPF, este protocolo faz roteamento por estado de enlace. O IS-IS é um dos três protocolos de nível de rede especificados pela ISO para comutação de pacotes. Embora o IS-IS tenha sido projetado para roteamento ISO, ele foi adaptado para o uso em ambientes IP. Este protocolo pode operar em modo dual de forma a suportar simultaneamente a comutação de pacotes em

redes IP e ISO, o que originou o termo *Integrated IS-IS*. O termo *IS* vem da denominação *Intermediate System*, usada para roteadores no modelo OSI. Segundo Martey e Sturgess [MS02], este protocolo tem maior popularidade na Internet que na arquitetura OSI. O IS-IS é usado como protocolo intra-domínio, opera em ISPs com roteadores Cisco e em alguns sistemas celulares digitais.

Apesar do IS-IS também fazer roteamento multicaminho (balanceamento de carga), ele difere do OSPF em alguns aspectos, inclusive quanto ao que se refere à atribuição de pesos aos arcos. Enquanto que no OSPF o peso w de um arco é um valor atribuído no intervalo $w = [1, 65535]$ e o custo máximo de um caminho é ilimitado, o peso de um arco no *IS-IS* é escolhido no intervalo $w = [1, 32]$ e o custo de um caminho, composto pela soma dos pesos dos arcos que o compõe, pode ser no máximo igual a 1024. Assim como no OSPF, o caminho mais curto é computado pelo algoritmo Dijkstra.

Apesar de protocolos como o OSPF e o Integrated IS-IS terem grande flexibilidade para atribuir pesos aos arcos, tendo maior controle sobre as rotas e fluxos de dados, o controle do congestionamento não é uma tarefa fácil. O crescimento considerável do tráfego da Internet fez com que a preocupação com o controle do congestionamento tomasse uma dimensão maior. A seção a seguir aborda esta questão.

1.3.3 Congestionamento na rede: prevenção e medidas de controle

Quando há pacotes demais trafegando numa parte da rede, o desempenho geral da mesma pode diminuir e pacotes podem ser descartados. A esta situação dá-se o nome de congestionamento. O congestionamento pode ser causado por diversos fatores:

- um roteador pode receber uma quantidade grande de fluxo que deve ser transmitido por uma única linha de capacidade de transmissão baixa;
- utilização de roteadores lentos. Estes podem estar ligados por meio de linhas de grande capacidade, mas podem efetuar operações muito lentamente;
- aumento do fluxo na rede, sem uma expansão adequada na sua topologia;
- o fluxo pode estar mal distribuído, com arcos sobrecarregados enquanto parte da rede opera em ociosidade.

O congestionamento é basicamente evitado através de um projeto de manutenção e expansão adequado, associado ao uso de medidas de controle. Algumas medidas são: monitorar o sistema para detectar quando e onde ocorre congestionamento, enviar essas informações para lugares onde alguma providência possa ser tomada e ajustar a operação do sistema para corrigir o problema, entre outras. Também existem várias medidas de prevenção de congestionamento que são abordadas com mais detalhes por Tanenbaum [Tan97]. No nível de rede, estas medidas estão relacionadas com o ordenamento dos pacotes, política de descarte de pacotes, algoritmo de roteamento, gerenciamento de tempo de vida do pacote, etc.

Pode-se citar como algumas dessas medidas:

- a rede normal (sem falhas) opera usando somente parte da banda de cada arco, de forma que uma banda adicional é disponível caso seja necessária;
- parte da capacidade de certos arcos pode ser reservada exclusivamente para alguns tipos de dados;
- usando o mesmo trajeto (evitando ter que fazer reordenamento no *host* destino).

Estas medidas estão relacionadas com qualidade de serviço (*Quality of Service* - QoS). Segundo Venesna [Sri01], funções de QoS IP possibilitam ao operador da rede um maior controle sobre esta, objetivando a garantia de entrega, assim como o fornecimento de serviços Internet diferenciados.

1.4 Contextualização dos tópicos abordados neste capítulo com o trabalho de tese

Neste capítulo foram apresentadas as duas arquiteturas de redes disponíveis: OSI da ISO e a Internet TCP/IP. A arquitetura de roteamento de dados na Internet foi posteriormente abordada, com ênfase em sua estrutura física (composta por sistemas autônomos), endereçamento IP, roteamento IP e protocolos de roteamento IP. Para os protocolos OSPF e IS-IS foram abordadas algumas das características principais, tais como: valores que o peso de um arco pode ter, custo máximo de um caminho e políticas de balanceamento de carga, entre outras. O capítulo finalizou com uma discussão sobre o que é, formas de prevenir e tratar o congestionamento na Internet.

Os problemas abordados nos Capítulos 2 e 3 são relativos à operação e projeto, respectivamente, de uma rede interna a um AS, operando sob o protocolo OSPF. O problema de caminhos mínimos dinâmicos, abordado no Capítulo 4, faz parte da atualização dinâmica dos caminhos mínimos usados pelos problemas dos Capítulos 2 e 3. Além dessa aplicação específica, o estudo foi extendido para casos mais gerais que podem ser usados em outros problemas e aplicações.

Capítulo 2

Minimização do Congestionamento de Redes OSPF

Uma das atribuições da engenharia de tráfego intra-domínio é usar eficientemente os recursos de rede de um sistema autônomo (AS). Protocolos intra-domínios, tais como o OSPF (*Open Shortest Path First*) ou IS-IS (*Intermediate System-Intermediate System*), são comumente usados para selecionar os caminhos pelos quais o tráfego é roteado. Os caminhos mínimos dependem dos pesos dos arcos, designados pelo operador da rede. Cada roteador do AS calcula os caminhos mínimos e cria uma tabela de roteamento. Esta tabela é usada para direcionar cada pacote para o próximo roteador no caminho para o roteador destino.

Considerando uma rede com $|V|$ roteadores e $|E|$ linhas de transmissão, e dado um conjunto de demandas entre pares de roteadores origem-destino [FGL$^+$01], o *problema de designação de pesos* (*Weight Setting Problem* - WSP) para roteamento OSPF consiste em designar pesos aos arcos desta rede de forma a otimizar o custo de uma função, geralmente associada com uma medida de congestionamento. Este problema é NP-difícil [FRT02], o que tem direcionado a pesquisa para métodos aproximados de resolução.

Trabalhos anteriores em otimização dos pesos para roteamento OSPF ou escolheram pesos de tal forma que caminhos mínimos múltiplos da origem ao destino fossem evitados, ou quebraram as regras do protocolo, escolhendo um caminho único para cada par de roteadores origem-destino [RR94, LW93, BGW98]. Fortz e Thorup [FT00] foram os primeiros fazer balanceamento de carga no roteamento OSPF. Eles propuseram uma metaheurística de busca tabu e a aplicaram em dados reais do *backbone* da AT&T e em redes sintéticas (criadas para fim de testes).

Ericsson, Resende e Pardalos [ERP02] propuseram um algoritmo genético e o aplicaram às mesmas instâncias consideradas por Fortz e Thorup [FT00]. Sridharan, Guérin e Diot [SGD02] desenvolveram outra heurística para uma versão um pouco diferente do problema, na qual o fluxo passante por cada roteador é dividido entre um subconjunto dos arcos partindo do roteador origem em direção ao destino.

Neste capítulo, um algoritmo memético é proposto, o qual consiste no algoritmo genético proposto por Ericsson, Resende e Pardalos [ERP02] combinado com uma busca local que faz uso de um eficiente mecanismo para atualizar o grafo de caminhos mínimos após a mudança do peso de um arco. Os resultados gerados pelo algoritmo memético foram comparados com os gerados pelo algoritmo genético e com os resultados da busca tabu proposta por Fortz e Thorup [FT00].

Na próxima seção, uma formulação matemática do problema é apresentada. A seguir, é realizado um estudo das heurísticas simples, buscas locais e metaheurísticas existentes para o WSP e, em seguida, e a busca local proposta é apresentada na Seção 2.2.4. A Seção 2.2.5 descreve os algoritmos de atualização do grafo de caminhos mínimos, usados pela busca local. Os resultados computacionais são apresentados na Seção 2.5 e, a seguir, o capítulo é finalizado com as conclusões.

2.1 Formulação do problema

Considere uma rede de comunicação de dados representada por um grafo direcionado $G = (V,E)$, onde o conjunto de nós V representa os roteadores e o conjunto de arcos E representa as linhas de transmissão. Além disso, considere uma capacidade c_a associada a cada arco $a \in E$ e uma *matriz de demandas D* que, para cada par $(s,t) \in V \times V$, fornece a demanda D_{st} de fluxo de tráfego do nó origem s para o nó destino t. O problema de designação de pesos para roteamento OSPF consiste em atribuir pesos inteiros positivos $w_a \in [1, w_{\max}]$ para cada arco $a \in E$, tal que uma medida do custo de roteamento seja otimizada quando a demanda é roteada de acordo com os regras do protocolo OSPF. Como já salientado, o protocolo OSPF permite valores para $w_{\max} \leq 65535$.

Para cada par (s,t) e para cada arco a, o valor $f_a^{(st)}$ indica quanto do fluxo de tráfego originado em s e com destino em t passa pelo arco a. Sendo l_a a carga total no arco a, isto é, a soma dos fluxos passando pelo arco a, a *taxa de utilização* é calculada como $u_a = l_a/c_a$. A medida de custo

de roteamento corresponde ao congestionamento total da rede. Para isso, considere inicialmente o *custo do congestionamento* em cada arco $a \in E$ sendo dado pela função convexa e linear por partes $\Phi_a(l_a)$, proposta por Fortz e Thorup [FT00] e apresentada na Figura 2.1.

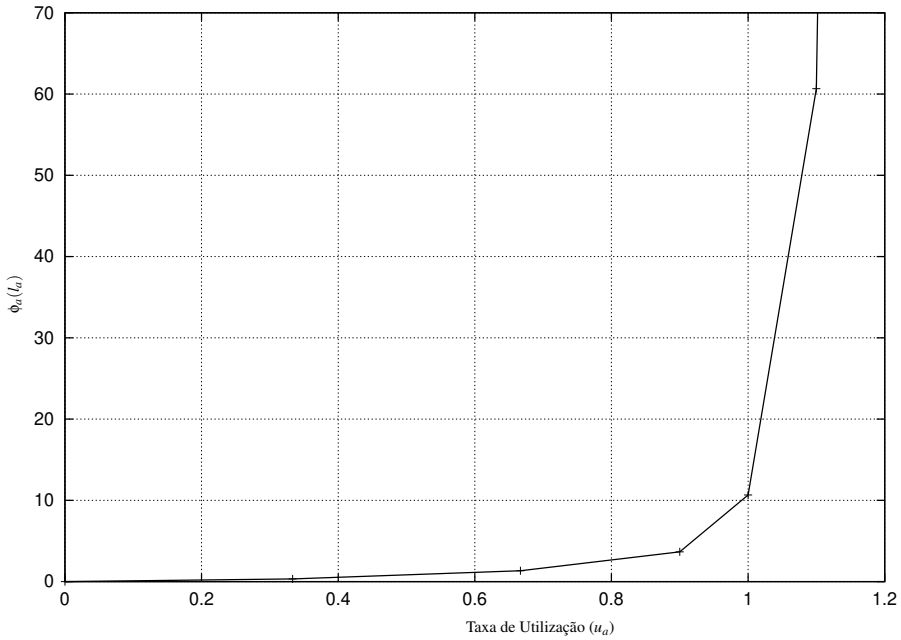

Figura 2.1: Função $\Phi_a(l_a)$ convexa e linear por partes.

A cada arco $a \in E$ calcula-se o valor de $\Phi_a(l_a)$ que relaciona o quão próximo da capacidade c_a está a carga l_a que passa por um arco a: quanto maior a proximidade, maior a penalização. O objetivo é distribuir a carga homogeneamente entre os arcos da rede, a fim de minimizar a soma dos custos Φ_a de cada arco a. Para tanto, a função possui derivadas definidas como:

$$\Phi_a(l_a) = \begin{cases} u_a, & u_a \in [0, 1/3) \\ 3 \cdot u_a - 2/3, & u_a \in [1/3, 2/3), \\ 10 \cdot u_a - 16/3, & u_a \in [2/3, 9/10), \\ 70 \cdot u_a - 178/3, & u_a \in [9/10, 1), \\ 500 \cdot u_a - 1468/3, & u_a \in [1, 11/10), \\ 5000 \cdot u_a - 16318/3, & u_a \in [11/10, \infty). \end{cases} \tag{2.1}$$

Logicamente, se um arco a estiver ocioso, $\Phi_a = 0$. Note que, dado que a função penaliza fortemente fluxos que excedem a capacidade, a solução deve ser tal que, se possível, nenhum arco apresente taxa de utilização $\frac{l_a}{c_a}$ acima do valor 1. Objetiva-se um perfil de fluxos o mais homogêneo possível, sem sobrecarregar nenhum arco.

Dada uma designação de pesos w e sendo as cargas $l_a^{OSPF(w)}$ associadas a cada arco $a \in E$ correspondendo às rotas obtidas com OSPF, denota-se o *custo de roteamento* por $\Phi_{OSPF(w)} = \sum_{a \in E} \Phi_a(l_a^{OSPF(w)})$. O problema de designação de pesos é equivalente a definir, para os arcos, pesos $w^* \in [1, w_{\max}]$ tal que a função $\Phi_{OSPF(w)}$ seja minimizada.

É possível obter uma solução relaxada do WSP por intermédio da resolução de um modelo matemático multiproduto de fluxo de custo mínimo [AMO93]. Este problema é uma relaxação do roteamento OSPF, pois o fluxo passando por cada nó i não é necessariamente dividido igualmente entre os caminhos mínimos partindo de i. A utilidade de tal modelo é ter acesso a um limitante inferior à solução ótima do WSP, que pode ser usado para comparar a qualidade de soluções do WSP obtidas por métodos heurísticos.

O modelo matemático correspondente ao WSP relaxado se define

$$\Phi_{OPT} = \min \Phi = \sum_{a \in E} \Phi_a(l_a) \tag{2.2}$$

sujeito a

$$\sum_{u:(u,v)\in E} f_{(u,v)}^{(st)} - \sum_{u:(v,u)\in E} f_{(v,u)}^{(st)} = \begin{cases} -D_{st} & \text{se } v = s, \\ D_{st} & \text{se } v = t, \\ 0 & \text{caso contrário,} \end{cases} \qquad v,s,t \in V, \tag{2.3}$$

$$l_a = \sum_{(s,t)\in V\times V} f_a^{(st)}, \qquad a \in E, \tag{2.4}$$

$$\Phi_a(l_a) \geq l_a, \qquad a \in E, \tag{2.5}$$

$$\Phi_a(l_a) \geq 3l_a - 2/3c_a, \qquad a \in E, \tag{2.6}$$

$$\Phi_a(l_a) \geq 10l_a - 16/3c_a, \qquad a \in E, \tag{2.7}$$

$$\Phi_a(l_a) \geq 70l_a - 178/3c_a, \qquad a \in E, \tag{2.8}$$

$$\Phi_a(l_a) \geq 500l_a - 1468/3c_a, \qquad a \in E, \tag{2.9}$$

$$\Phi_a(l_a) \geq 5000l_a - 16318/3c_a, \qquad a \in E, \tag{2.10}$$

$$f_a^{(st)} \geq 0, \quad a \in E; s,t \in V. \tag{2.11}$$

A função objetivo minimiza o custo total de congestionamento, calculado como a soma do valor de congestionamento dos arcos da rede. As restrições do tipo (2.3) estabelecem o balanço de fluxo em cada nó da rede, ao mesmo tempo que garantem que a demanda de tráfego entre cada par (s,t)

seja satisfeita. A carga em cada arco a é composta dos fluxos advindos de cada par (s,t) e está assegurada pelas restrições do tipo (2.4). As restrições (2.5) a (2.10) definem os valores de custo da função objetivo $\Phi_a(l_a)$ em função da carga no arco, caracterizando a forma clássica que transforma um modelo linear por partes em um linear. O modelo é uma relaxação do roteamento OSPF, visto que permite roteamento de tráfego arbitrário. Dessa forma, Φ_{OPT} é um limitante inferior do custo.

Fortz e Thorup [FT00] propuseram um fator de escala normalizador para o custo de roteamento o qual torna possível a comparação de redes de tamanhos e topologias diferentes:

$$\Phi_{UNCAP} = \sum_{(s,t)\in V\times V} D_{st}h_{st},$$

onde D_{st} é a demanda com origem em s e destino em t e h_{st} é o mínimo contador de nós (*hop count*) entre os nós s e t. Para qualquer custo de roteamento Φ, o custo de roteamento escalado é definido como

$$\Phi^* = \Phi/\Phi_{UNCAP}.$$

Usando esta notação, as seguintes afirmações são verdadeiras

- O custo ótimo de roteamento satisfaz

$$1 \leq \Phi^*_{OPT} \leq \Phi^*_{OSPF(w^*)} \leq \Phi^*_{OSPF(1)}.$$

- Dada qualquer solução para as restrições (2.2) - (2.11), com custo de roteamento normalizado Φ^*, então $\Phi^* = 1$ se e somente se todas as cargas dos arcos estiverem abaixo de $\frac{1}{3}$ de suas capacidades e todas as demandas forem roteadas pelo mínimo *hop count*.

- Dada qualquer solução para (2.2) - (2.11) onde se todos os arcos estiverem com cargas iguais as suas capacidades máximas, então o custo de roteamento normalizado $\Phi^* = 10\frac{2}{3}$ ($10\frac{2}{3} = 70 - \frac{178}{3}$). Diz-se que o roteamento *congestiona* uma rede se $\Phi^* \geq 10\frac{2}{3}$.

O início deste capítulo objetiva analisar o problema analiticamente. As próximas seções foram escritas sob uma visão mais algoritmica, com o objetivo de apresentar os procedimentos computacionais usados para resolver o problema. Inicia-se com a próxima seção, a qual apresenta o procedimento usado para avaliar uma solução do problema usando como objetivo a função Φ. O intuito é dar uma visão geral da sequência de procedimentos que devem ser executados para que a avaliação seja realizada.

2.1.1 Avaliação de uma solução do problema

A avaliação de uma solução do WSP é uma operação complexa que consome bastante tempo e, por isso, deve ser eficiente. Uma solução corresponde a um vetor de pesos w, na qual cada arco a possui um peso w_a associado. Independentemente de como os pesos sejam atribuídos aos arcos, o procedimento de avaliação é o mesmo. Considera-se a função objetivo Φ mencionada na seção anterior, que calcula Φ_a, o congestionamento de cada arco $a \in E$. A Figura 2.2 descreve o procedimento básico para avaliar uma solução do WSP.

```
procedure EvaluateWSP(w)
1     forall t ∈ T do
2           d^t = Dijkstra(d, w);
3           g^t = calculateCM(d, w);
4           δ^t = calculateDelta(d, w);
5           l^t = routeLoad(t, d^t, g^t, δ^t, D);
6     end forall
7     Φ = 0;
8     forall a ∈ E do
9           l_a = totalLoad(a, l);
10          Φ_a = Phi(a, l);
11          Φ = Φ + Φ_a;
12    end forall
13    return Φ;
end EvaluateWSP.
```

Figura 2.2: Procedimento de avaliação de uma solução do WSP .

Algumas estruturas de dados utilizadas são dados de entrada do programa, como o grafo $G = (V,E)$, o vetor de capacidades c e a matriz de demanda D. Outras são criadas especialmente para a avaliação da solução e pode-se chamá-las de memória da solução: g^{SP}, δ, d, *lat* (parcial) e *la* (total). As estruturas g^{SP}, δ, d e *lat* (parcial) são calculadas para cada destino $t \in T$. O algoritmo de caminhos mínimos `Dijkstra()` é aplicado e a distância de cada nó ao destino t é calculado e armazenado no vetor d (linha 2). Cada posição do vetor g^{SP} armazena o valor *0* ou *1* caso o arco esteja ou não, respectivamente, no grafo de caminhos mínimos direcionado para t, calculado pela função `calculateCM()` (linha 3). Para cada nó destino t, calcula-se, para cada nó $a \in V$, o valor

δ_a, que indica o número de arcos incidentes presentes no grafo de caminhos mínimos (linha 4). Tendo a informação d, g^{SP}, δ, além da matriz de demandas D, a carga lat_a^t passante por cada arco, referente a cada g^{SP}, é calculada pela função `routeLoad()` (linha 5). Finalmente, a carga total la_a passando por cada arco $a \in E$ é somada (linha 9) e a avaliação do tráfego no arco é calculada (linha 10). A função `Phi()`, além de calcular o valor de Φ_a, normaliza este valor, como proposto em Fortz e Thorup [FT00].

Qualquer método heurístico pode usar o cálculo da função de avaliação descrita nesta seção. A seguir, serão discutidas algumas heurísticas simples para o problema.

2.2 Métodos de solução propostos para o WSP

Pelo que se sabe até o momento não há método exato proposto para solucionar o WSP. Na categoria de heurísticas há várias propostas, desde as simples heurísticas construtivas até metaheurísticas sofisticadas. As próximas sub-seções tem o intuito de apresentar as principais heurísticas simples, buscas locais e metaheurísticas propostas para resolver o WSP.

2.2.1 Heurísticas simples

Há algumas heurísticas simples propostas, mas os resultados obtidos são em geral pouco satisfatórios, principalmente se o tráfego na rede é intenso.

A seguir, enumeram-se quatro heurísticas simples, nas quais os pesos são atribuídos aos arcos da seguinte forma:

- pesos com valores proporcionais às distâncias euclidianas entre as extremidades dos arcos;
- pesos designados com valores aleatórios;
- pesos iguais a *1* para todos os arcos;
- pesos inversamente proporcionais a sua capacidade.

Esta última é usada pela Cisco [Tho98] que é uma importante fornecedora de equipamentos para roteadores. O peso de um arco a é dado por $\frac{c_{max}}{c_a}$, onde c_{max} é a maior capacidade da rede.

2.2.2 Buscas locais

Técnicas de busca local dependem da definição precisa de uma vizinhança. Uma vizinhança $N(w)$ de uma solução w é constituída por todas as soluções w' possíveis de serem obtidas por meio de uma perturbação determinística da solução w. As soluções da vizinhança são sucessivamente avaliadas em termos da função objetivo e, no caso de haver uma solução w' com melhor avaliação que w, então ela é aceita como a nova solução e uma nova vizinhança $N(w')$ é construída. O algoritmo termina numa solução chamada de ótimo local, quando nenhuma solução da vizinhança é melhor do que a solução corrente.

No caso do WSP, pode-se definir como uma vizinhança v_1 de uma solução w todas as soluções w' que possuem um único arco com peso diferente. Mais formalmente, uma solução w' vizinha de w é qualquer solução que possui $w'_i \neq w_i$ para um arco qualquer i, e $w'_a = w_a$ para todo $a \neq i \in A$.

Uma vizinhança v_2 maior pode ser formada por toda solução w' que possui um conjunto C de arcos com pesos diferentes dos correspondentes em w. Quanto maior o tamanho do conjunto C, maior é a vizinhança. Uma vizinhança reduzida vr_1 pode restringir a mudança dos pesos a um subconjunto $S_1 \in A$. Uma vizinhança reduzida vr_2 pode restringir a faixa em que o valor do peso de um arco a, ou um conjunto de arcos, pode diferir entre as soluções. Uma terceira redução vr_3 pode restringir os pesos a somente aumentarem ou diminuírem seu valor.

A avaliação de qualquer solução w' pertencente a uma das vizinhanças definidas acima pode ocorrer de forma diferente de como foi descrito no pseudo-código da Figura 2.2. Isso porque a memória da solução w pode ser reaproveitada em parte para o cálculo da solução w'. Ramalingam e Reps ([RR96a]) propuseram um algoritmo, denominado `RR`, que, ao modificar o valor do peso de um arco, somente a parte afetada do grafo de caminhos mínimos é recalculada. Estes algoritmos são conhecidos como algoritmos para caminhos mínimos dinâmicos (*dynamic shortest paths*) e podem facilmente ser generalizados para atualizar um grafo de caminhos mínimos quando um conjunto de arcos muda de peso. Usando esta mesma idéia de aproveitar a memória da solução anterior e apenas atualizar a parte realmente modificada, Fortz e Thorup ([FT00]) propuseram um algoritmo que recalcula as cargas apenas dos arcos que tiveram aumento ou diminuição do tráfego.

A Figura 2.3 apresenta o pseudo-código do cálculo de uma solução w', da vizinhança de w, diferindo apenas no valor do peso do arco e.

```
procedure UpdateWSP(e, w_e, w)
1     forall t ∈ T do
2           d^t = RR(d, w, e, w_e);
3           lat^t = updateLoad(t, d_t, g_t^SP, δ_t, D);
4     end forall
5     Φ = 0;
6     forall a ∈ E do
7           la_a = totalLoad(a, la);
8           Φ_a = Phi(a, l);
9           Φ = Φ + Φ_a;
10    end forall
11    return Φ;
end UpdateWSP.
```

Figura 2.3: Procedimento de avaliação rápida de uma solução do problema.

A função `RR()` atualiza as memórias d, δ e g^{SP}, enquanto que a função `updateLoad()` codifica a atualização da carga proposta por Fortz e Thorup (2000). O laço das linhas 6-10 também poderia ser atualizado, mas como é composto por operações muito simples, a diferença entre atualizar e recalcular estes dados é muito pequena.

Várias heurísticas baseadas em buscas locais simples foram propostas, tais como os trabalhos de Lin e Wang [LW93], Rodrigues e Ramakrishnan [RR94] e Bley, Grötchel e Wessläy [BGW98]. Os dois primeiros usam vizinhanças para orientar uma busca local e o último lança mão de relaxação Lagrangeana. Uma característica comum a todos esses métodos é que eles alocam pesos aos arcos de forma que as demandas sejam roteadas por um único caminho de peso mínimo, evitando o balanceamento de carga. Além disso, apenas redes de pequena dimensão foram avaliadas nos testes computacionais realizados. Recentemente, metaheurísticas mais sofisticadas têm sido aplicadas com sucesso para tratar redes de porte realístico, como será visto na próxima seção.

2.2.3 Metaheurísticas propostas

A literatura especializada de otimização combinatória consagrou o uso da palavra metaheurística para designar o conjunto de técnicas que se utilizam de uma estratégia mestre que se superpõe a uma heurística (em geral uma busca local) de forma a guiá-la e modificá-la a fim de produzir

soluções capazes de transcender a otimalidade local. A seguir, uma metaheurística busca tabu, baseada em busca local, será apresentada. Posteriormente um algoritmo genético e um memético serão apresentados.

Busca tabu

Fortz e Thorup [FT00] propuseram um algoritmo de busca tabu (*Tabu Search* - TS) para resolver o WSP, que permite achar soluções que dividem igualmente o fluxo caso o conjunto de pesos alocado aos arcos defina mais de um caminho mínimo entre um par origem-destino. Este é um preceito básico do protocolo OSPF que é relaxado pelos métodos heurísticos relacionados na seção anterior. A implementação de TS usa funções e tabelas *hash* que podem ser empregadas em substituição aos mecanismos necessários para compor uma estrutura de memória atributiva, o mais comum dos mecanismos usados para evitar ciclagens em TS. O uso de tabelas *hash* para evitar ciclagem numa busca tabu foi proposto por Woodruff e Zemel [WZ93]. Para evitar visitar soluções já avaliadas numa vizinhança dada, usa-se uma função *hash* que mapeia uma atribuição de pesos nos arcos em um número inteiro. É criada uma tabela *hash* onde os inteiros são guardados por um certo número de iterações e usados para evitar que se avaliem soluções já visitadas. Este procedimento é considerado prático, pois para muitos problemas é simples guardar e buscar numa lista de inteiros, comparando-se com a avaliação da vizinhança.

O método usa uma tabela *hash* com 2^{16} entradas, usada para evitar ciclagem, e uma segunda tabela *hash*, com $2^{16} \div 20$ entradas, que é usada para reaproveitar o cálculo de soluções já computadas e também como estratégia de diversificação. A entrada numa tabela obtida por uma solução é calculada através de uma operação XOR dos valores já calculados para cada arco. Para cada arco $a \in E$ tem-se associado um valor inteiro p_a de $(l+m-1)$ bits, tendo este sido gerado aleatoriamente. O cálculo de $p_a * w_a$ gera um inteiro de $(2m+l-1)$ bits, do qual extraem-se os m bits iniciais e os $m-1$ bits finais. O valor de h_a é o inteiro de l bits restante do cálculo de $p_a * w_a$.

Uma vizinhança para busca local dessa TS é caracterizada pela mudança do peso de um arco a selecionado aleatoriamente. Inicialmente apenas 20% dos arcos seriam selecionados, mas este valor é dividido por 3 cada vez que a solução não melhora e multiplicado 2 caso contrário. O algoritmo executa 5000 iterações. Se a solução incumbente não melhorar após 300 iterações, é

somado ao peso de 10% dos arcos um valor $v \in (-2,-1,1,2)$, escolhido aleatoriamente para cada arco.

Até o momento dois métodos populacionais foram propostos para o WSP, um algoritmo genético e um algoritmo memético, apresentados nas próximas duas sub-seções. Em ambos é respeitado o preceito do protocolo OSPF que determina que o fluxo deva ser igualmente direcionado por caminhos de mesmo peso mínimo.

A seguir, é apresentado o algoritmo genético, o qual caracteriza uma metaheurística não baseada em busca local. A partir dele obteve-se um algoritmo memético inserido-se uma busca local e fazendo-se pequenas alterações de parâmetros.

Algoritmo genético

Nesta seção, são descritos detalhes do algoritmo genético proposto por Ericsson, Resende e Pardalos [ERP02]. No contexto do WSP, uma população é um conjunto de soluções. No caso do algoritmo genético descrito neste capítulo, as soluções da população são sempre factíveis. Soluções na população são combinadas (através das operações de *crossover* e mutação) para produzir uma nova população de soluções. Quando soluções são combinadas, genes de soluções de alta qualidade têm maior probabilidade de serem transmitidas às populações seguintes. Este processo é repetido muitas gerações, conduzindo à melhoria da população com o transcorrer das gerações. A seguir é apresentada como esta idéia pode ser explorada para a designação de pesos do OSPF. A Figura 2.4 apresenta o pseudo-código do algoritmo genético.

Cada solução é representada por um vetor de pesos inteiros, com cada componente correspondendo ao peso de um arco da rede. Cada peso pertence ao intervalo $[1, w_{\max}]$. A cada solução w é associado o valor da função objetivo (*fitness*) definido pelo custo de roteamento OSPF ($\Phi_{OSPF(w)}$). A população inicial é gerada aleatoriamente (linha 1), com os pesos dos arcos selecionados de uma distribuição uniforme no intervalo $[1, w_{\max}/3]$. Somente $\frac{1}{3}$ do intervalo inicial é considerado, visto que a busca local (descrita posteriormente) busca melhorar soluções *aumentando* o peso de arcos congestionados. A população é particionada em três conjuntos $\mathcal{A}$, $\mathcal{B}$, e $\mathcal{C}$. As melhores soluções são armazenadas em $\mathcal{A}$, enquanto que as piores em $\mathcal{C}$. Todas as soluções em $\mathcal{A}$ são promovidas para a próxima geração (linha 8). Soluções em $\mathcal{B}$ são substituídas (linha 9) por soluções origina-

```
procedure GeneticAlgoritm()
1     InitializePop(Pop);
2     Cost ← EvaluatePop(1,Pop);
3     best ← ∞;
4     while StoppingCriteria;
5           gen ← gen + 1;
6           SortPop(Pop);
7           if Cost_Pop1 < best then best ← Cost_Pop1;
8           SaveA();
9           CrossoverB();
10          RandomC();
11          EvaluatePop(|A|,Pop);
12    end while;
13    SortPop(Pop);
14    PrintSolution(Pop1);
end GeneticAlgorithm.
```

Figura 2.4: Estrutura geral do algoritmo genético de Ericsson, Resende e Pardalos.

das do *crossover* de um indivíduo de $\mathcal{A}$ com outro de $\mathcal{B} \cup \mathcal{C}$ usando o *crossover random keys* de Bean [Bea94]. Todas as soluções em $\mathcal{C}$ são substituídas por soluções novas geradas aleatoriamente (linha 10), com os pesos dos arcos gerados no intervalo $[1, w_{\max}]$. Para estruturar a população, as soluções são ordenadas em ordem crescente do seu valor de *fitness*. A ordenação é necessária no início de cada geração (linha 6), e antes de retornar a solução final (linha 13).

Uma solução é avaliada em três situações: ao ser inicialmente gerada (linha 2), se for resultado de um *crossover* e ao ser gerada aleatoriamente na classe $\mathcal{C}$ (linha 11).

O *crossover* é executado considerando duas soluções. Como resultado, tem-se uma nova solução que herda genes de ambas soluções que a geraram. Das soluções escolhidas para *crossover*, uma é selecionada de uma população elite, enquanto a outra é oriunda de uma população mais pobre. A solução elite é selecionada com igual probabilidade aleatoriamente no conjunto $\mathcal{A}$, enquanto a outra é selecionada com igual probabilidade aleatoriamente entre as soluções do conjunto $\mathcal{B} \cup \mathcal{C}$. No esquema *random keys*, cada peso da solução resultante do *crossover* é herdado de uma das soluções selecionadas. Ainda, o peso pode receber um valor inteiro aleatoriamente selecionado no intervalo $[1, w_{\max}]$ com probabilidade p_m, caracterizando uma mutação. Se a mutação não ocorrer,

então a solução gerada herda o peso do pai elite com uma probabilidade $p_{\mathcal{A}} = 70\%$. Caso contrário, o gene é proveniente da outra solução.

A próxima seção apresenta um algoritmo memético, gerado basicamente pela inserção de um procedimento local neste algoritmo genético.

2.2.4 Proposta de um algoritmo memético

Os algoritmos meméticos foram inicialmente denominados de *algoritmos genéticos híbridos*. Posteriormente observou-se que o procedimento de busca local inserido num GA atua de forma similar de como os *memes* (da biologia) atuam numa população e, por isso, atualmente os *algoritmos genéticos híbridos* são também denominados de *algoritmos meméticos*. Para uma visão geral do assunto, pode-se citar os trabalhos de Moscato [Mos02].

Esta seção tem o intuito de apresentar o algoritmo memético gerado a partir da inserção de um procedimento de busca local a cada solução gerada pelo operador de *crossover* do algoritmo genético descrito na Seção 2.2.3.

Partindo de uma solução inicial, o procedimento de busca local analisa soluções na vizinhança da solução atual w na busca por uma solução de menor custo de roteamento. Se tal solução existir, então ela substitui a atual. Caso contrário, a solução atual é retornada como um mínimo local.

O procedimento de busca local é incorporado pelo algoritmo genético, descrito na Seção 2.2.3, para aumentar sua habilidade de encontrar soluções de boa qualidade com menor tempo computacional. O procedimento de busca local é aplicado a cada solução gerada pelo operador de *crossover*. Além de consumir muito tempo, o uso de vizinhanças grandes em um algoritmo genético pode causar a falta de diversidade da população, e consequentemente convergência prematura para um mínimo local de baixa qualidade. Testes realizados aplicando-se o operador de busca local às soluções das classes $\mathcal{A}$ e $\mathcal{C}$ não pareceram ajudar no desempenho do algoritmo memético. A seguir, o procedimento de busca local usando vizinhança reduzida é descrito.

Como antes, l_a refere-se à carga total no arco $a \in E$ na solução originada pelo designação de pesos atual w. Lembrando que $\Phi_a(l_a)$ refere-se ao custo de roteamento neste arco. O procedimento de busca local avalia o efeito de aumentar o peso de um subconjunto dos arcos.

Estes arcos candidatos são selecionados entre os que têm o maior custo de roteamento e peso

menor que $w_{\max}$. Para reduzir o custo de roteamento de um arco candidato, o procedimento aumenta o seu peso na esperança de que a carga sobre ele diminua. Se o aumento do peso do arco causar redução no custo da solução, a mudança é aceita e o procedimento é reinicializado. O procedimento pára num ótimo local quando não há melhoria causada pelo aumento do peso dos arcos candidatos. O pseudo-código é descrito na Figura 2.5.

```
procedure LocalImprovement(q, w)
1     Φ_incumbente ← EvaluateWSP(w);
2     forall a ∈ E do dontlook_a ← 0;
3     i ← 1;
4     while i ≤ q do
5           Renumber the arc indices such that
            Φ_a(l_a) ≥ Φ_{a+1}(l_{a+1}) for a = 1,...,|E| − 1;
6           a' ← 0;
7           forall a = 1,...,|E| while a' = 0 do
8                 if dontlook_a = 1 then dontlook_a ← 0;
9                 else if w_a < w_max then a' ← a;
10          end forall;
11          if a' = 0 then return;
12          dontlook_{a'} ← 1;
13          forall ŵ = w_{a'} + 1,...,w_{a'} + ⌈(w_max − w_{a'})/4⌉ do
14                w'_a ← w_a, forall a ∈ E, a ≠ a';
15                w'_{a'} ← ŵ;
16                if Φ_{OSPF(w')} < Φ_incumbente then
17                      w ← w';
18                      dontlook_{a'} ← 0;
19                      i ← 0;
20                      Φ_incumbente ← Φ_{OSPF(w')};
21                end if
22          end forall
23          i ← i + 1;
24    end while
end LocalImprovement.
```

Figura 2.5: Pseudo-código do procedimento `LocalImprovement`.

O procedimento `LocalImprovement` recebe como parâmetro a solução corrente originada pelo conjunto de pesos w e o parâmetro q que especifica o número máximo de arcos candidatos a serem examinados em cada iteração da busca local. Ainda, arcos cujo aumento do peso não resultaram

em melhoria da solução, são ignorados na vez seguinte em que seriam selecionados como arcos candidatos. Para implementar esta estratégia, usa-se um vetor de *don't look*.

As estruturas de dados da solução são preenchidas e o valor da solução inicial é calculado na linha 1. A seguir, os valores do vetor *don't look* bits são marcados como falso na linha 2 e o contador de arcos candidatos é inicializado na linha 3. O laço nas linhas 4-24 considera no máximo q arcos candidatos selecionados para aumento de peso. Os índices dos arcos são renumerados na linha 5, tal que os arcos são considerados em ordem não crescente de custo de roteamento. O laço nas linhas 7-10 procura por arcos marcados como falsos e com pesos menores que $w_{\max}$. Arcos marcados que não podem ser selecionados na iteração atual são marcados na linha 8 para futura investigação. Arco a' é selecionado na linha 9. Se não há arco que satisfaça tais condições, o procedimento pára na linha 11 e retorna o vetor de pesos corrente w como mínimo local. Na linha 12, o arco a' é temporariamente marcado com falso, a menos que uma melhoria da solução é encontrada considerando $w_{a'}$.

O laço nas linhas 13-22 examina todas as possibilidades de aumento do peso do arco a' no intervalo $[w_{a'}+1, w_{a'}+\lceil(w_{\max}-w_{a'})/4\rceil]$. Uma solução vizinha w', armazenando os pesos dos arcos não modificados, exceto pelo arco a', é gerada nas linhas 14 e 15. Se a nova solução w' tiver um custo de roteamento menor do que a solução corrente (teste na linha 16), então a solução corrente é atualizada na linha 17. O arco a' tem seu *don't look* bit desmarcado na linha 18, e o contador de arco i é zerado na linha 19. Na linha 23, o contador do arco candidato i é incrementado.

Tamanhos de vizinhanças variados foram testados. Também foram testados procedimentos que permitem a diminuição do valor dos pesos (de arcos sub-utilizados). Ainda foi testado um procedimento de busca local com vizinhança maior e permitindo aumento e diminuição dos pesos dos arcos. De todos as variações testadas, o procedimento adotado foi o que obteve melhores resultados. Os testes diminuindo o peso de arcos sub-utilizados tiveram um desempenho pobre.

O custo de roteamento $\Phi_{OSPF}(w')$ associado com a solução vizinha w' é calculado na linha 16. Em vez de recalcular tudo novamente, usa-se um procedimento para atualizar o grafo de caminhos mínimos, assim como as cargas. Estes procedimentos são descritos na próxima seção. Uma vez que as novas cargas são conhecidas, o custo de roteamento é calculado como a soma do custo de roteamento de cada arco.

2.2.5 Atualizações rápidas das cargas dos arcos e do grafo de caminhos mínimos

Nesta seção, são descritos os procedimentos usados para atualização rápida do grafo de caminhos mínimos (linha 15 do procedimento `LocalImprovement`) e da carga dos arcos. Supondo que são dadas as estruturas de dados d, g^{SP}, δ, lat e l já preenchidas, para cada nó destino $t \in T$, conforme um vetor w fornecido. Ainda, considere que o arco a' tem seu peso incrementado em exatamente uma unidade. Dado o conjunto T de nós de destino e sabendo que $g^t = (V, E^t)$ é o grafo de caminhos mínimos associado com cada nó de destino $t \in T$. Um ou mais destes grafos de caminhos mínimos podem ser afetados pela mudança do peso do arco a' de $w_{a'}$ para $w_{a'} + 1$. Consequentemente, a carga de alguns arcos em cada grafo muda.

Os procedimentos descritos nesta seção usam o conjunto de estruturas que funcionam como uma memória para a solução. Com uma mudança de peso, o grafo de caminhos mínimos e as cargas podem mudar, e as memórias são atualizadas em vez de recalculadas desde o início.

Cada *grafo de caminhos mínimos* com destino em t possui um vetor g^t, de dimensão $|E|$, indicando os arcos presentes no grafo de caminhos mínimos. Se o arco a está no grafo de caminhos mínimos, então $g^t_a = 1$. Caso contrário, $g^t_a = 0$. Outro vetor de dimensão $|E|$, l^t, associado com os arcos, armazena as *cargas parciais* fluindo para o nó t e passando por cada arco $a \in E$. A carga total de cada arco é representada no la, de dimensão $|E|$ o qual armazena a *carga total* passando por cada arco $a \in E$. Os vetores d^t e δ^t, ambos com dimensão $|V|$, estão associados a cada nó. A *distância* de cada nó ao seu destino t é armazenado em d^t, enquanto que δ^t armazena o número de arcos partindo de cada nó em g^t. Estas estruturas são calculadas e preenchidas no início e liberadas no final do procedimento *LocalImprovement*. Por simplicidade, elas foram omitidas do procedimento, bem como a lista de parâmetros do procedimento descrito nesta seção.

O pseudo-código na Figura 2.6 sintetiza os passos principais do procedimento de atualização. A nova carga l_a em cada arco $a \in E$ é inicializada com zero na linha 1. O laço nas linhas 2-6 considera cada nó de destino $t \in T$. Para cada um deles, o grafo de caminhos mínimos g^t é atualizado na linha 3 e as cargas parciais são atualizadas na linha 4. A carga dos arcos é atualizada na linha 5. Linhas 1 e 5 são removidas deste procedimento no caso da carga dos arcos ser atualizada dentro do procedimento `UpdateLoads`. Ao final, o custo $\Phi_{OSPF(w')}$ da nova solução é computada na

```
procedure UpdateCost(a', d, l)
1     forall a ∈ E do l_a ← 0;
2     forall t ∈ T do
3           UpdateShortestPaths(a', w, t);
4           UpdateLoads(d, t);
5           forall a ∈ E^t do l_a ← l_a + l_a^t;
6     end forall
7     Φ_OSPF(w') ← Σ_{a∈E} Φ_a(l_a);
end UpdateCost.
```

Figura 2.6: Pseudo-código do procedimento `UpdateCost`.

linha 7. A seguir, são descritos os procedimentos `UpdateShortestPaths` e `UpdateLoads` usados nas linhas 3 e 4.

2.3 Algoritmo de caminhos mínimos dinâmicos reversos

Denomina-se por $g^t = (V, E^t)$ o grafo de caminhos mínimos associado a cada nó destino $t \in T$. Se o peso de um único arco a' mudou, o grafo g^t não necessita ser recalculado totalmente. Em vez disso, atualiza-se a parte do grafo afetada pela mudança do peso. Ramalingam e Reps [RR96a] e Frigioni et al. [FMSN96] propuseram algoritmos eficientes para a atualização de g^t, sem necessidade de ser recalculado usando o algoritmo Dijkstra. Estes dois algoritmos são comparados experimentalmente em Frigioni et al. [FINP98]. Embora o algoritmo de Frigioni et al. seja teoricamente melhor, o algoritmo de Ramalingam e Reps [RR96a] geralmente é mais rápido na prática. Devido à natureza do problema de designação de pesos segundo o roteamento OSPF, usa-se a versão reversa do algoritmo para computar o grafo de caminhos mínimos usando Dijkstra. Em Frigioni, Marchetti-Spaccamela e Nanni [FMSN96], e em Ramaligam e Reps [RR96a], é usada a versão não reversa do algoritmo Dijkstra.

Na aplicação no problema em questão, foi proposta uma especialização do algoritmo de Ramalingam e Reps para atualizar o grafo de caminhos mínimos quando o peso de um arco muda em uma unidade (na versão padrão o incremento pode ser em qualquer quantidade). A idéia deste procedimento é fazer a atualização de forma rápida e eficiente. Assim como na versão padrão do Ramalingam e Reps, somente a parte afetada do grafo de caminhos mínimos é atualizada. Mas

diferentemente desta, para a atualização após um incremento unitário o algoritmo não necessita utilizar pilhas. Supondo que a seja o arco que teve seu peso incrementado em uma unidade e Q a parte do grafo contendo os nós que possuem todos os caminhos mínimos para o destino usando o arco a. Se o peso do nó origem do arco a aumentou de um, todos os nós do conjunto Q terão suas distâncias para o nó destino acrescidas de um. Com isso, estes nós podem encontrar caminhos alternativos ligando nós fora de Q. Dessa forma, o algoritmo primeiramente determina o conjunto Q, já incrementando as distâncias desses nós em um, é após verifica todos os arcos saintes para verificar se estes pertencem a caminhos alternativos. A Figura 2.7 ilustra os passos principais deste procedimento. A grafo possui o conjunto Q determinado e o grafo da direita apresenta, em adição, os arcos que foram inseridos representado caminhos alternativos de nós $u \in Q$.

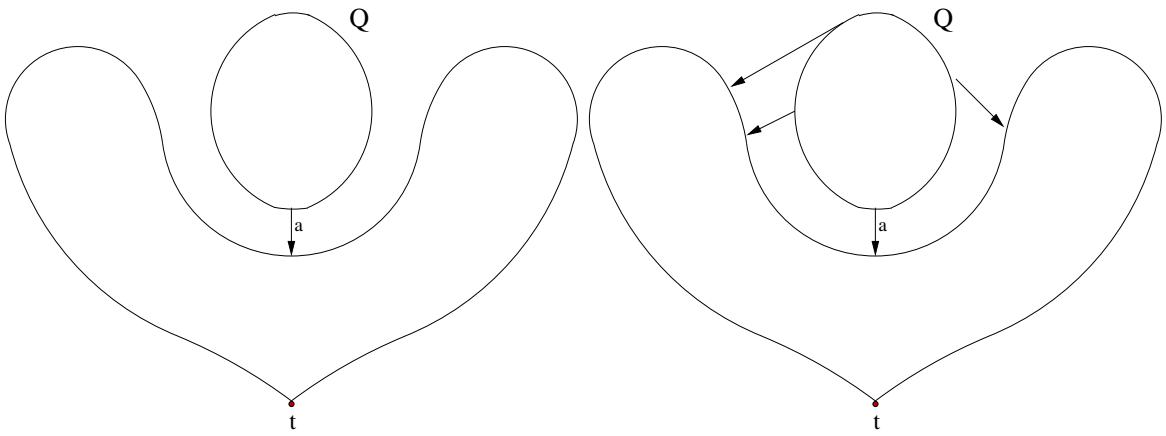

Figura 2.7: Atualização rápida do grafo de caminhos mínimos.

O pseudo-código do algoritmo para a atualização do grafo de caminhos mínimos especializado para considerar o incremento unitário de um arco é fornecido na Figura 2.8. Buriol et al. [BFRR03, BRRT03] mostram empiricamente que este algoritmo é mais rápido que o algoritmo padrão proposto por Ramalingam e Reps [RR96a] para atualização do grafo de caminhos mínimos. Inicialmente, o algoritmo identifica o conjunto Q de nós cujas distâncias mudam devido ao incremento dos pesos. A seguir, o grafo de caminhos mínimos é atualizado pela remoção e adição de arcos os quais no mínimo uma de suas extremidades pertence a Q.

O algoritmo `UpdateShortestPaths` recebe como parâmetros o arco $a' = (\overrightarrow{u,v})$ cujo peso mudou, o vetor de pesos atual w, e um nó destino $t \in T$. O algoritmo verifica, na linha 1, se o arco a'

```
procedure UpdateShortestPaths(a' = (u→, v→), w, t)
1     if a' ∉ g^t return;
2     g^t ← g^t \ {a'};
3     InsertIntoHeapMax(H, u, d^t(u));
4     δ_u ← δ_u − 1;
5     if δ_u > 0 then return
6     Q = {u};
7     forall v ∈ Q do
8           d^t(v) ← d^t(v) + 1;
9           forall a = (u, v) ∈ IN(v) ∩ g^t do
10                g^t ← g^t \ {a};
11                InsertIntoHeapMax(H, u, d^t(u));
12                δ_u ← δ_u − 1;
13                if δ_u = 0 then Q ← Q ∪ {u};
14          end forall
15    end forall
16    forall u ∈ Q do
17          forall a = (u, v) ∈ OUT(u) do
18                if d^t(u) = w_a + d^t(v) then
19                      g^t ← g^t ∪ {a};
20                      InsertIntoHeapMax(H, u, d^t(u));
21                      δ_u ← δ_u + 1;
22                end if
23          end forall
24    end forall
end UpdateShortestPaths.
```

Figura 2.8: Pseudo-código do procedimento `UpdateShortestPaths`.

não pertence ao grafo de caminhos mínimos g^t, caso em que a mudança de peso não afeta g^t e o procedimento termina. O arco a' é eliminado do grafo de caminhos mínimos g^t na linha 2. Na linha 3 o nó origem do arco a' é inserido na pilha contendo todos os nós no qual a carga dos arcos partindo podem mudar. Esta pilha será usada pelo algoritmo de atualização das cargas. As distâncias para os nós destinos t são usadas como chaves prioritárias e a raiz contém o nó com distância máxima. O grau de partida δ_u do nó origem do arco a' é atualizada na linha 4. Na linha 5, verifica-se se há um caminho alternativo para o nó destino a partir do nó u. Neste caso, o procedimento pára, visto que g^t não muda mais. O conjunto Q de nós afetados pela mudança do peso do arco a' é inicializado com o nó u na linha 6. O laço na linhas 7-15 constrói o conjunto Q. Para cada nó identificado neste

conjunto (linha 7), sua distância $d^t(v)$ para o nó destino é incrementada em *1* na linha 8. O laço nas linhas 16-24 atualiza os caminhos mínimos. Cada nó u no conjunto Q é considerado um a um na linha 16, e cada arco a de partida é consultado na linha 17. A seguir, verifica-se (linha 18) se o arco a pertence ao novo caminho mínimo para o nó destino. Neste caso, o arco a é inserido no grafo de caminhos mínimos (linha 19), seu nó origem u é inserido na pilha (linha 20) e seu grau de saída δ_u é atualizado (linha 21).

A Figura 2.9 apresenta um exemplo da aplicação do algoritmo `UpdateShortestPaths`. O grafo `A)`, apresenta a rede com o grafo de caminhos mínimos identificado pelos arcos com linha cheia, enquanto os arcos $a \notin g^t$ são identificados por linhas tracejadas. O nó t, localizado mais à esquerda, é o nó destino. O arco que terá seu peso incrementado em 1 está em destaque. A cada arco está associado um peso e, a cada nó, a sua distância mínima ao nó destino. O grafo `B)` apresenta a rede com os nós do conjunto Q já identificados, com suas distâncias acrescidas em 1, e todos arcos chegantes e saintes removidos do grafo de caminhos mínimos (correspondendo a execução das primeiras 15 linhas do procedimento `UpdateShortestPaths`). A rede `C)` apresenta o grafo de caminhos mínimos já atualizado.

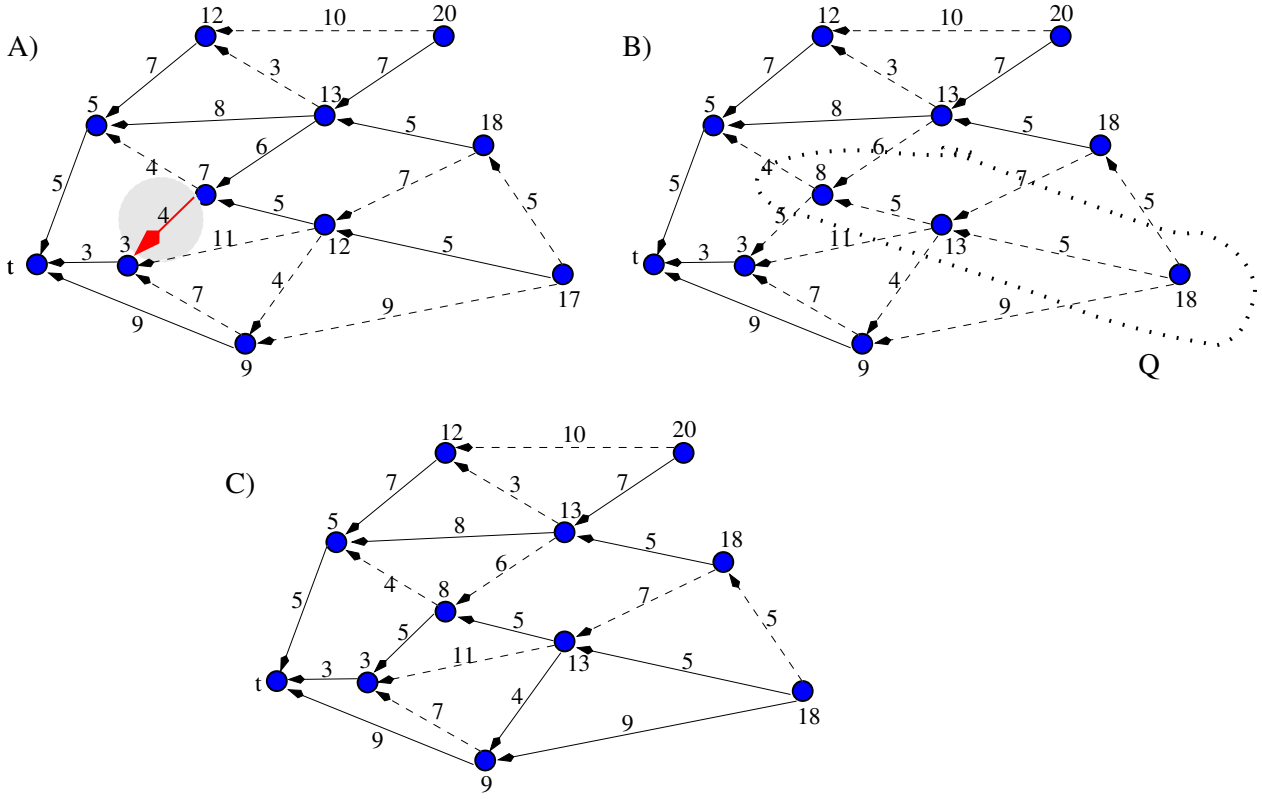

Figura 2.9: Atualização do grafo de caminhos mínimos após o incremento unitário do peso de um arco.

2.4 Atualização dinâmica das cargas

Inicialmente o procedimento `UpdateShortestPaths` constrói uma pilha H contendo todos os nós cujo conjunto de arcos de partida é modificado no grafo de caminhos mínimos.

O pseudo-código na Figura 2.10 sintetiza os passos principais do procedimento de atualização da carga. Denota-se por l_a^t a carga no arco $a \in E$ associado com o nó destino $t \in T$. O procedimento `UpdateLoads` recebe como parâmetro as demandas d e o nó de destino t. O laço nas linhas 1 a 9 remove os nós da pilha até esvaziá-la.

O nó u com máxima distância para o nó destino é removido na linha 2. A carga total fluindo através do nó u é igual a $D_{ut} + \sum_{a=(v,u)\in g^t} l_a$. A carga em cada arco partindo do nó u é calculado na linha 3. O laço nas linhas 4-7 atravessa todos arcos partindo do nó u no grafo de caminhos mínimos atual, os quais possuem a carga parcial l_a^t para ser atualizada. A nova carga parcial é calculada na linha 5 e o nó v, destino do arco a, é inserido na pilha H na linha 6.

```
procedure UpdateLoads(H, D, t)
1       while HeapSize(H) > 0 do
2               u ← FindAndDeleteMax(H);
3               load ← (D_ut + Σ_{a=(v,u)∈g^t} l_a^t)/δ_u;
4               forall a = (u,v) ∈ g^t : l_a^t ≠ load do
5                       l_a^t ← load;
6                       InsertIntoHeapMax(H, v, π(v));
7               end forall
8       end while
end UpdateLoads.
```

Figura 2.10: Pseudo-código do procedimento `UpdateLoads`.

A Figura 2.11 apresenta um exemplo da atualização do fluxo das cargas passante nos arcos do grafo de caminhos mínimos. Os dois grafos apresentados correspondem aos grafos `A` e `C` da Figura 2.9. Considere o caso em que os nós `a`, `b`, `c` e `d` possuem demanda de mesmo valor direcionada para o nó destino `t`. Com o aumento do peso do arco com bastante carga, novos caminhos mínimos alternativos são detectados, e o fluxo passante por este arco diminui.

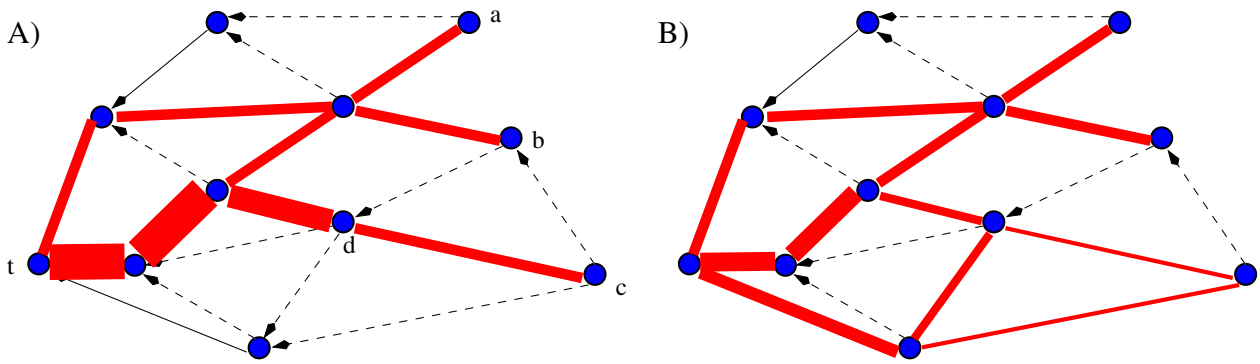

Figura 2.11: Atualização do fluxo de carga nos arcos após a atualização do grafo de caminhos mínimos.

2.5 Resultados computacionais

Nesta seção, são apresentados os experimentos computacionais com o algoritmo memético introduzido nas seções anteriores. O ambiente computacional é primeiramente descrito, a lista de

parâmetros é apresentada e as instâncias testes são a seguir fornecidas.

Nos experimentos, compara-se o algoritmo memético com o limitante inferior associado com o programa linear (2.2) a (2.11), assim como com outras heurísticas e metaheurísticas.

2.5.1 O ambiente computacional

Os experimentos foram conduzidos em um computador SGI Challenge (28 processadores 196-MHz MIPS R10000) com 7.6 Gb de memória. Cada execução usou um único processador. Os algoritmos foram implementados em C e compilados com o compilador `cc`, versão 7.30, usando o flag `-O3`. Os tempos foram medidos com a função `getrusage`. Para geração dos números aleatórios usados pelos algoritmos memético e genético utilizou-se o gerador *Mersenne Twister* de Matsumoto e Nishimura [MN98].

Os parâmetros a seguir foram usados nos algoritmo genético e memético:

- Tamanho da população: 50.
- Intervalo dos pesos: $[1, w_{\max} = 20]$ (sendo que para população inicial é usado o intervalo $[1, w_{\max} = 7]$).
- Particionamento da população em 25% das soluções (arredondado para cima = 13) no conjunto $\mathcal{A}$, 5% das soluções (arredondada para cima = 3) no conjunto $\mathcal{C}$, e as demais soluções no conjunto $\mathcal{B}$.
- Probabilidade de ocorrer mutação: $p_m = 1\%$.
- Probabilidade de um peso ser herdado do pai elite pela solução gerada no *crossover*: $p_{\mathcal{A}} = 70\%$.
- O número de gerações varia de acordo com o tipo do experimento.

Em adição, o número máximo de arcos candidatos é estipulado como $q = 5$ no procedimento de busca local do algoritmo memético.

Os experimentos foram realizados considerando 13 instâncias de quatro classes propostas por Fortz e Thorup [FT00] e também usadas por Ericsson, Resende e Pardalos [ERP02]. Detalhes

sobre as instâncias encontram-se sintetizados na Tabela 2.1. A instância `att`, do *backbone* da rede da *AT&T Worldnet* é uma rede real de 90 roteadores e 274 arcos. A matriz de demanda desta instância é obtida segundo a metodologia proposta por Feldmann et al. [FGL+01]. As redes hierárquicas de 2-níveis (*2-level hierarchical*) são obtidas usando o gerador GT-ITM de Zegura [Zeg96], baseado no modelo de Calvert et al. [CDZ97] e Zegura et al. [ZCB96]. Este modelo usa dois tipos de arcos: arcos de acesso local têm capacidade igual a 200, enquanto que arcos de longa distância têm capacidades iguais a 1000. Para a classe de redes aleatórias, *random*, a probabilidade de haver um arco entre dois nós é dada por um parâmetro que controla a densidade da rede. Todos os arcos dessas redes têm capacidade 1000. Nas redes *Waxman* os nós são pontos uniformemente distribuídos em um quadrado unitário. A probabilidade de haver um arco entre dois nós u e v é $\eta e^{-\Delta(u,v)/(2\theta)}$, onde η é o parâmetro usado para controlar a densidade da rede, $\Delta(u,v)$ é a distância euclidiana entre u e v, e θ é a distância máxima entre quaisquer dois nós na rede [Wax98]. Todos os arcos têm capacidade 1000. Fortz e Thorup geraram as demandas para forçar alguns nós a enviar ou receber mais carga que outros. Nessa geração são designadas cargas altas entre pares de nós localizados proximamente.

Tabela 2.1: Características da rede: nome da classe, nome da instância, número de nós ($|V|$), número de arcos ($|E|$), número de nós destinos ($|T|$), número de pares de demanda (par o-d), demanda total ($\sum D_{uv}$) e fator de escala da demanda (ρ).

Classe	Instância	$\|V\|$	$\|E\|$	$\|T\|$	par o-d	$\sum D_{uv}$	ρ
AT&T *backbone*	`att`	90	274	17	272	18465	0,2036885
2-level hierarchical	`hier100`	100	280	100	9900	921	0,4167095
	`hier100a`	100	360	100	9900	1033	1,0008225
	`hier50a`	50	148	50	2450	276	1,4855075
	`hier50b`	50	212	50	2450	266	1,0534975
Random	`rand100`	100	403	100	9900	994	5,8109135
	`rand100b`	100	503	100	9900	1026	8,1702935
	`rand50`	50	228	50	2450	249	14,139605
	`rand50a`	50	245	50	2450	236	18,941735
Waxman	`wax100`	100	391	100	9900	1143	3,5349025
	`wax100a`	100	476	100	9900	858	6,1694055
	`wax50`	50	169	50	2450	277	7,6458185
	`wax50a`	50	230	50	2450	264	12,454005

Para cada instância, 12 matrizes de demanda distintas $D^1, D^2, \ldots, D^{12}$ são geradas. Iniciando com a matriz de demanda D^1, as outras matrizes de demanda são geradas por repetidamente multiplicando D^1 por um fator de escala: $D^k = \rho^{k-1} D^1, \forall\, k = 1, \ldots, 12$. Para cada instância, a tabela

lista a classe, nome da instância, número de nós, número de arcos, número de nós destinos, número de pares origem-destino, a demanda total de D^1, e o fator de escala.

2.5.2 Comparação com tempo fixo

Nesta subseção, compara-se o algoritmo memético com três outras heurísticas e com o limitante inferior:

- InvCap: os pesos são designados proporcionalmente ao inverso da capacidade do arco, isto é, $w_a = \lceil c_{\max}/c_a \rceil$, sendo $c_{\max}$ a capacidade máxima do arco;
- GA: o algoritmo genético;
- TS: o algoritmo de busca tabu de Fortz e Thorup [FT00];
- LPLB: o limitante inferior Φ_{OPT}.

A heurística InvCap é usado pela Cisco IOS 10.3 [Cis97, Tho98]. O GA é derivado do algoritmo genético de Ericsson, Resende e Pardalos [ERP02], o qual é comparado com o algoritmo memético da Seção 2.2.4.

InvCap e LPLB são executadas em tempo real. Os algoritmos GA, MA e TS são executados por uma hora para cada uma das 12 matrizes das 13 instâncias. Portanto, gastaram-se 10*12*13 = 1560 horas de computador para esta bateria de testes. Como resultado, é apresentada a média das 10 execuções para cada instância.

As Tabelas 2.2 a 2.14 e Figuras 2.12 a 2.24 sintetizam os resultados. Para cada nível de demanda, as tabelas apresentam os custos normalizados para o InvCap e LPLB, assim como a média dos custos apresentados pelos algoritmos GA, MA e TS. A última linha de cada tabela apresenta a soma dos custos de cada algoritmo. Nas tabelas, foi adicionada um linha vertical que separa os valores sem congestionamento dos valores de instâncias congestionadas. A distribuição dos custos pode ser vista nas figuras. Os 10 valores de custo para cada algoritmo e instância estão indicados com um ponto, e uma curva é traçada interligando a média destes pontos.

Tabela 2.2: Custo de roteamento para a instância `att` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
3761,179	1,013	1,000	1,000	1,000	1,00
7522,358	1,013	1,000	1,000	1,000	1,00
11283,536	1,052	1,010	1,008	1,008	1,01
15044,715	1,152	1,057	1,050	1,050	1,05
18805,894	1,356	1,173	1,168	1,168	1,15
22567,073	1,663	1,340	1,332	1,331	1,31
26328,252	2,940	1,520	1,504	1,506	1,48
30089,431	21,051	1,731	1,689	1,691	1,65
33850,609	60,827	2,089	2,007	2,004	1,93
37611,788	116,690	2,663	2,520	2,520	2,40
41372,967	185,671	5,194	4,382	4,377	3,97
45134,146	258,263	20,983	16,433	16,667	15,62
Total	652,691	40,760	35,093	35,322	33,57

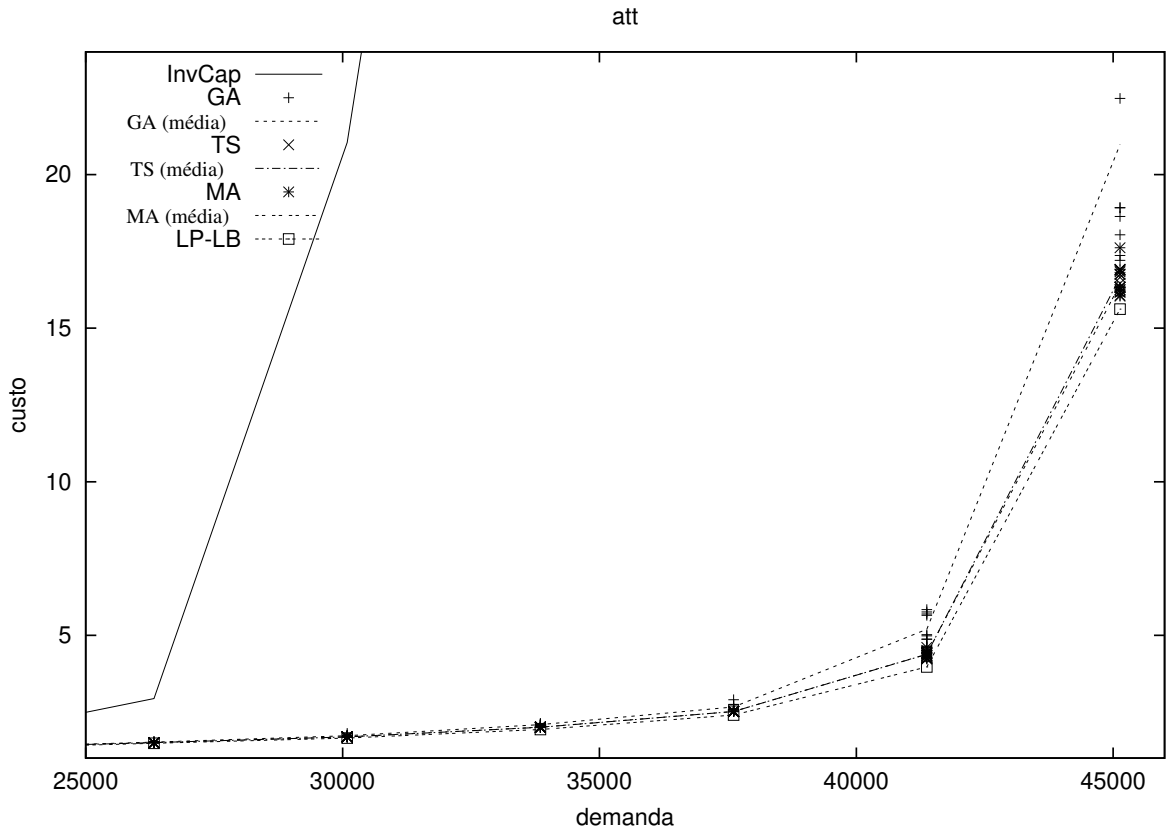

Figura 2.12: InvCap, GA, MA, TS e limitante inferior LP na instância `att`.

Tabela 2.3: Custo de roteamento para a instância `hier50a` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
410,641	1,016	1,000	1,000	1,000	1,00
821,281	1,028	1,000	1,000	1,000	1,00
1231,922	1,056	1,011	1,010	1,010	1,01
1642,563	1,150	1,049	1,044	1,043	1,04
2053,204	2,345	1,116	1,107	1,106	1,10
2463,844	21,890	1,209	1,194	1,193	1,17
2874,485	37,726	1,328	1,302	1,307	1,27
3285,126	56,177	1,490	1,434	1,443	1,39
3695,766	75,968	1,771	1,603	1,644	1,53
4106,407	106,904	2,243	2,013	2,129	1,89
4517,048	140,516	5,273	3,674	3,975	3,44
4927,689	180,299	20,968	15,123	16,837	14,40
Total	626,075	39,458	31,504	33,687	30,24

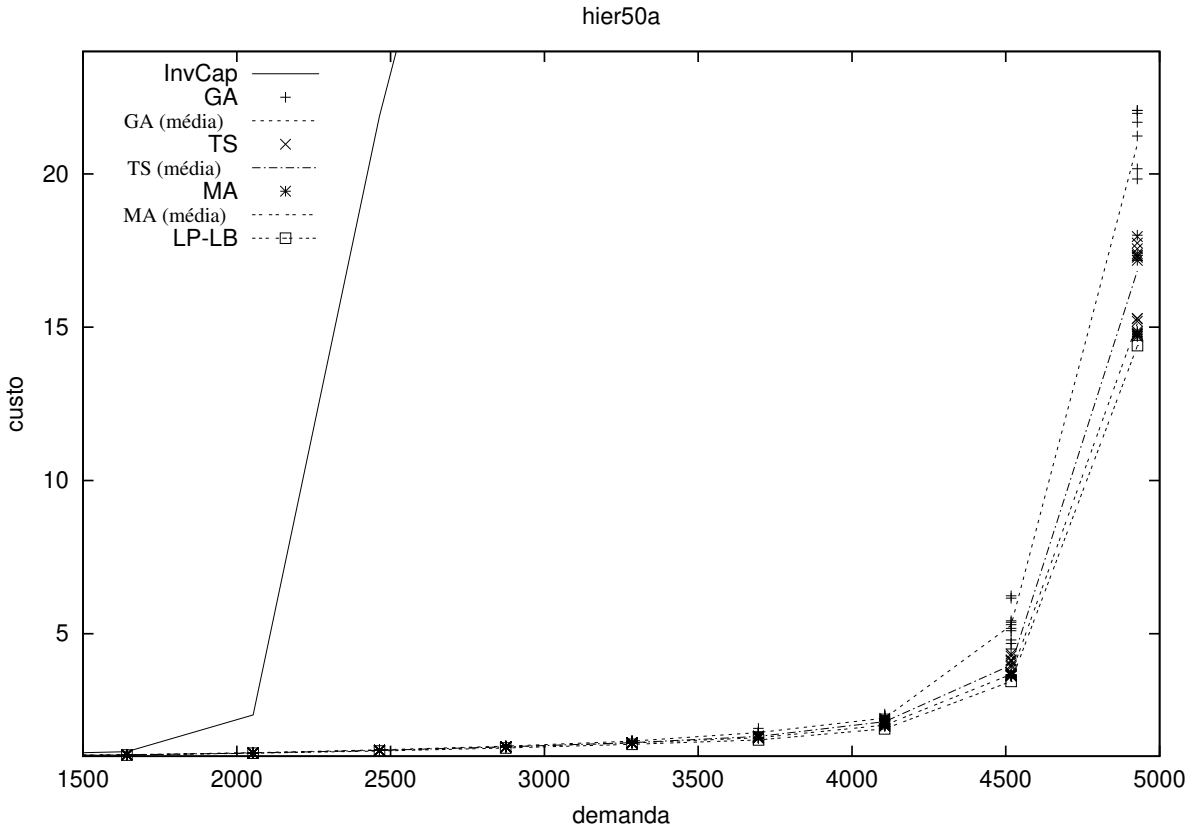

Figura 2.13: InvCap, GA, MA, TS e o limitante inferior LP na instância `hier50a`.

Tabela 2.4: Custo de roteamento para a instância `hier50b` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
280,224	1,005	1,000	1,000	1,000	1,00
560,449	1,012	1,001	1,001	1,001	1,00
840,673	1,039	1,018	1,017	1,017	1,01
1120,898	1,110	1,058	1,054	1,054	1,03
1401,122	1,268	1,098	1,092	1,092	1,06
1681,346	6,281	1,146	1,137	1,137	1,11
1961,571	27,661	1,227	1,208	1,206	1,16
2241,795	44,140	1,352	1,319	1,316	1,24
2522,020	63,905	1,520	1,453	1,452	1,35
2802,244	95,131	1,875	1,718	1,691	1,47
3082,468	128,351	3,153	2,264	2,205	1,61
3362,693	159,848	12,318	4,221	4,166	1,83
Total	530,751	27,766	18,484	18,337	14,87

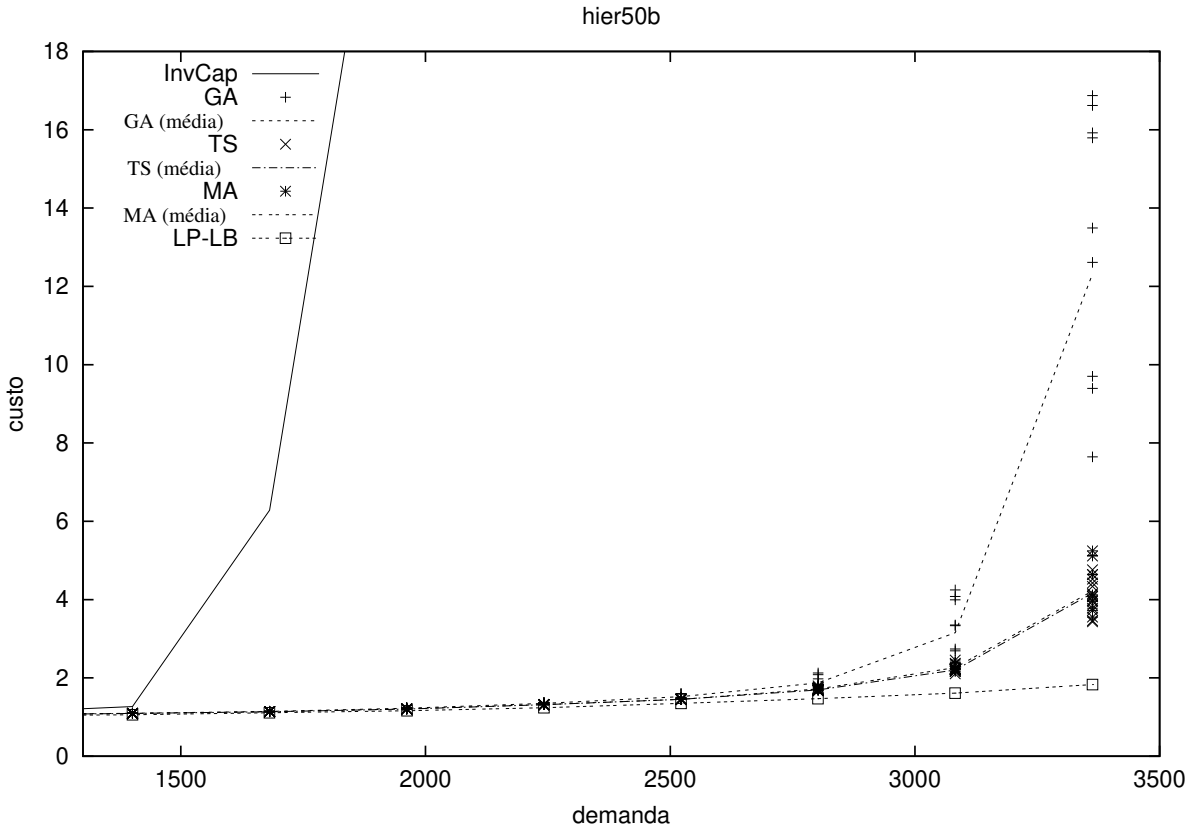

Figura 2.14: InvCap, GA, MA, TS e o limitante inferior LP na instância `hier50b`.

Tabela 2.5: Custo de roteamento para a instância `hier100` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
383,767	1,020	1,000	1,000	1,000	1,00
767,535	1,020	1,000	1,001	1,000	1,00
1151,303	1,030	1,006	1,007	1,005	1,01
1535,070	1,090	1,036	1,037	1,033	1,03
1918,838	1,170	1,083	1,078	1,076	1,06
2302,605	1,590	1,143	1,135	1,132	1,11
2686,373	8,870	1,234	1,220	1,217	1,20
3070,140	17,500	1,337	1,313	1,311	1,28
3453,908	24,930	1,602	1,557	1,558	1,52
3837,675	38,540	3,343	3,244	3,258	3,18
4221,443	70,250	11,976	11,812	11,823	11,71
4605,210	114,550	19,730	19,214	19,245	19,06
Total	281,560	45,490	44,618	44,658	44,16

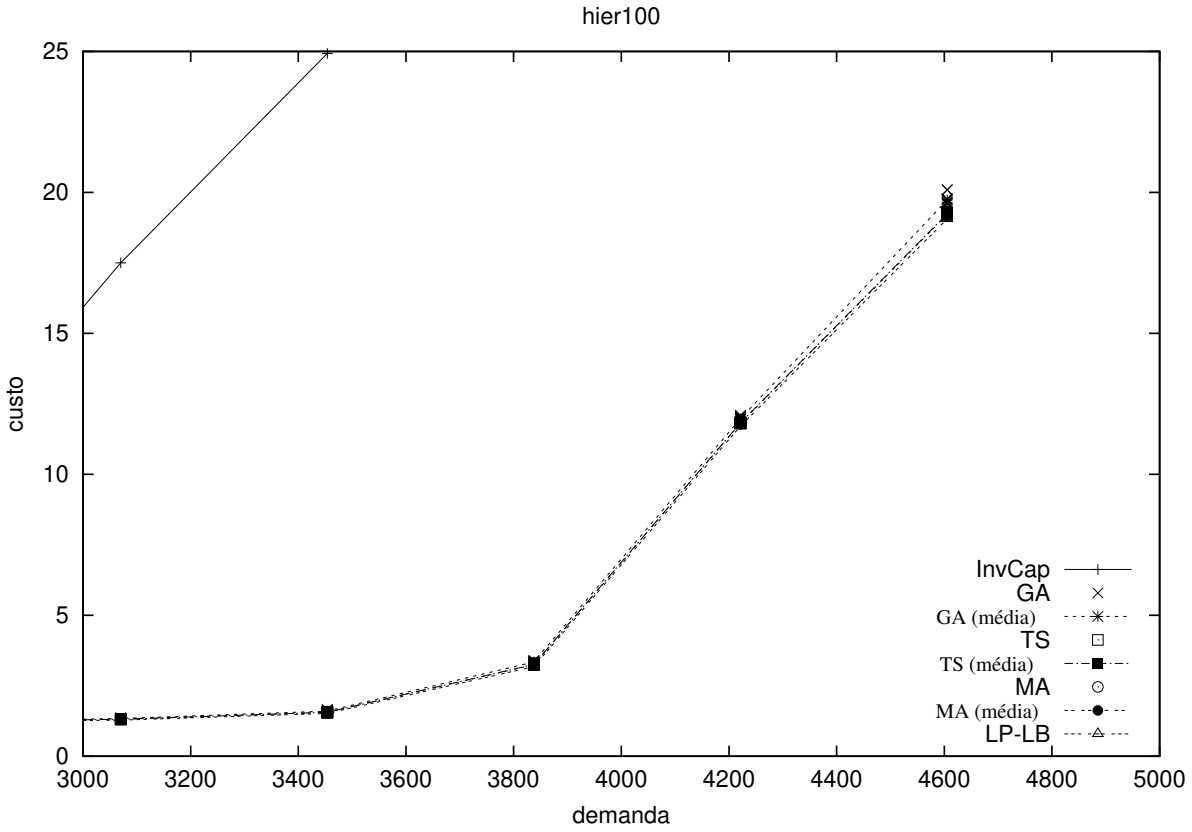

Figura 2.15: InvCap, GA, MA, TS e o limitante inferior LP na instância `hier100`.

Tabela 2.6: Custo de roteamento para a instância `hier100a` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
1033,879	1,170	1,001	1,006	1,000	1,00
2067,757	1,170	1,002	1,008	1,000	1,00
3101,636	1,190	1,012	1,018	1,005	1,00
4135,515	1,270	1,043	1,048	1,028	1,02
5169,394	1,390	1,098	1,091	1,069	1,06
6203,272	1,530	1,160	1,142	1,128	1,10
7237,151	2,390	1,279	1,221	1,208	1,16
8271,030	10,660	1,441	1,331	1,312	1,25
9304,909	24,770	1,696	1,518	1,483	1,38
10338,788	53,240	2,536	2,063	2,077	1,76
11372,667	112,110	8,123	5,846	5,568	4,48
12406,545	181,100	23,401	15,169	18,547	13,32
Total	391,990	44,792	33,461	36,425	29,53

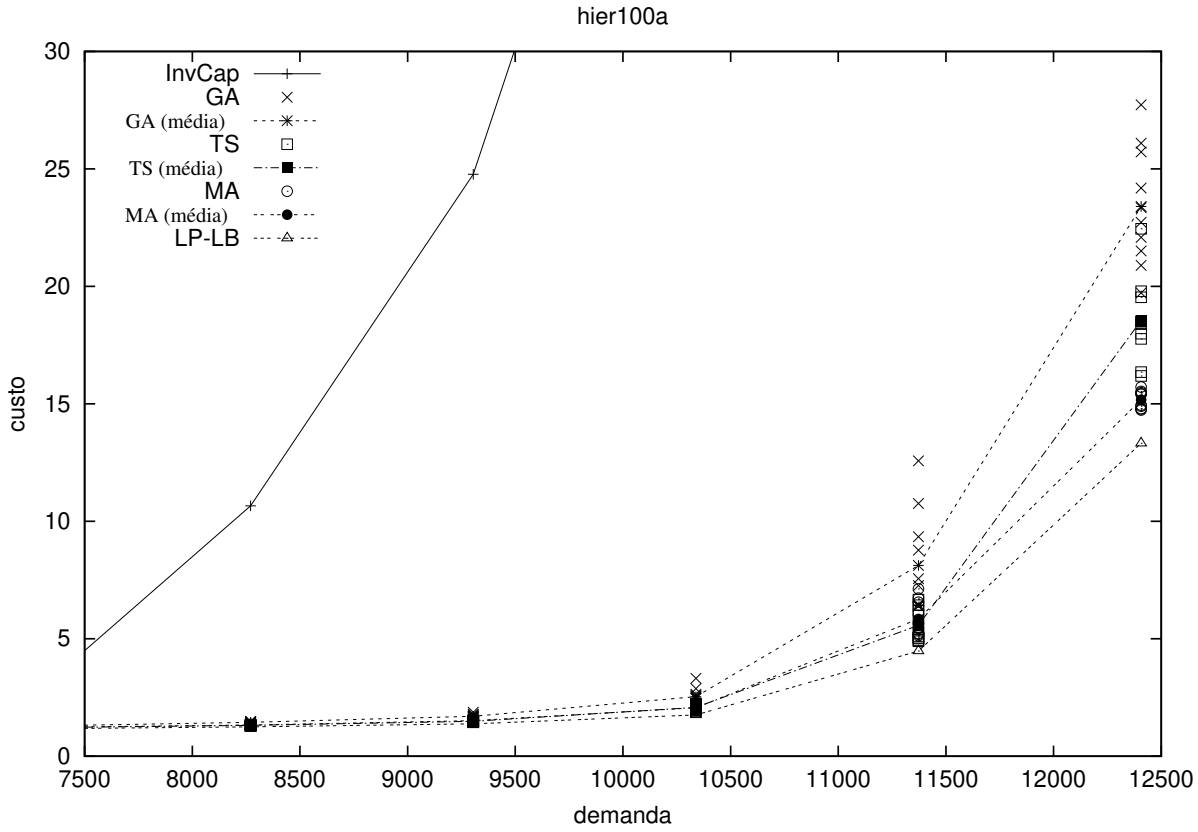

Figura 2.16: InvCap, GA, MA, TS e o limitante inferior LP na instância `hier100a`.

Tabela 2.7: Custo de roteamento para a instância `rand50` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
3523,431	1,000	1,000	1,000	1,000	1,00
7046,861	1,000	1,000	1,000	1,000	1,00
10570,292	1,043	1,002	1,001	1,001	1,00
14093,723	1,136	1,056	1,036	1,036	1,03
17617,154	1,296	1,179	1,151	1,144	1,13
21140,585	1,568	1,327	1,292	1,286	1,27
24664,015	3,647	1,525	1,455	1,447	1,42
28187,446	27,352	1,780	1,672	1,672	1,61
31710,877	66,667	2,173	1,977	1,976	1,90
35234,308	122,869	3,201	2,556	2,569	2,43
38757,739	188,778	7,738	4,607	4,683	4,26
42281,169	264,611	27,375	15,041	15,585	13,75
Total	680,967	50,356	33,788	34,399	31,8

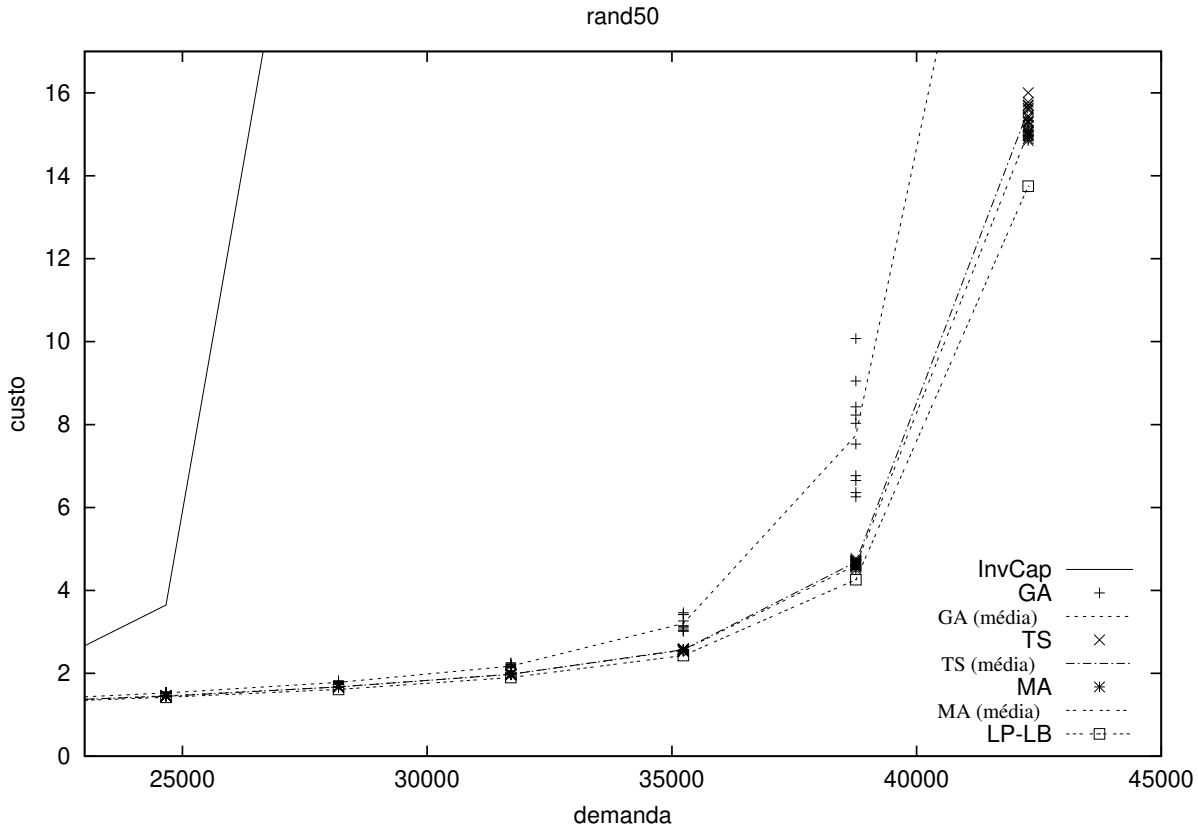

Figura 2.17: InvCap, GA, MA, TS e o limitante inferior LP na instância `rand50`.

Tabela 2.8: Custo de roteamento para a instância `rand50a` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
4463,462	1,000	1,000	1,000	1,000	1,00
8926,923	1,011	1,000	1,000	1,000	1,00
13390,385	1,074	1,014	1,008	1,008	1,01
17853,847	1,217	1,110	1,071	1,069	1,05
22317,308	2,133	1,258	1,221	1,214	1,19
26780,770	16,601	1,479	1,398	1,391	1,35
31244,232	42,653	1,770	1,637	1,628	1,57
35707,693	81,279	2,323	1,970	1,982	1,87
40171,155	131,857	3,590	2,519	2,529	2,29
44634,617	203,556	8,355	3,919	3,891	3,02
49098,079	278,997	55,041	10,239	10,374	5,98
53561,540	357,045	123,604	50,631	53,301	19,65
Total	1118,420	201,544	77,613	80,387	40,98

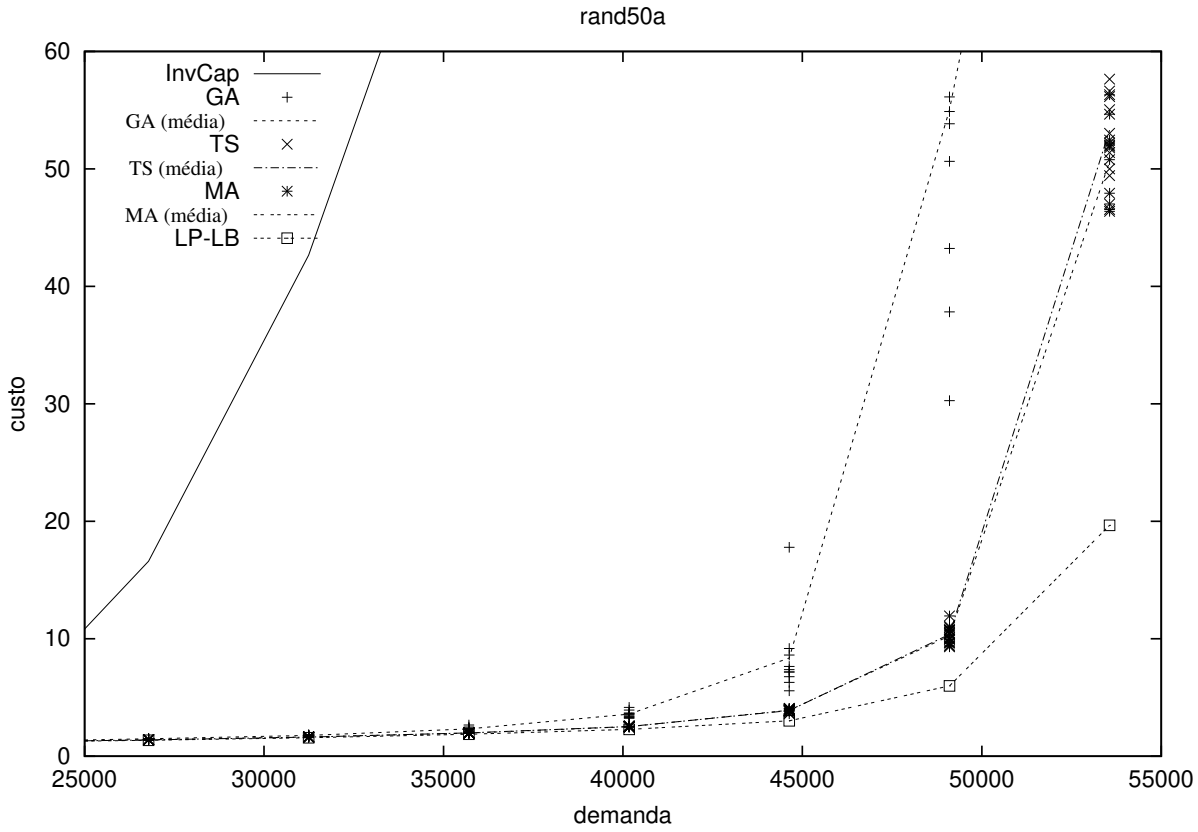

Figura 2.18: InvCap, GA, MA, TS e o limitante inferior LP na instância `rand50a`.

Tabela 2.9: Custo de roteamento para a instância `rand100` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
5774,737	1,000	1,005	1,006	1,000	1,00
11549,474	1,000	1,008	1,006	1,000	1,00
17324,211	1,040	1,034	1,013	1,001	1,00
23098,948	1,130	1,119	1,061	1,036	1,03
28873,685	1,310	1,299	1,217	1,156	1,14
34648,422	1,680	1,546	1,394	1,312	1,29
40423,159	9,250	1,932	1,601	1,507	1,47
46197,896	37,220	2,849	1,882	1,757	1,71
51972,633	71,520	4,375	2,320	2,112	2,02
57747,370	115,260	13,822	3,131	2,703	2,46
63522,107	173,790	41,105	6,729	4,175	3,27
69296,844	238,560	108,485	25,283	10,942	5,79
Total	652,760	179,579	47,643	29,701	23,18

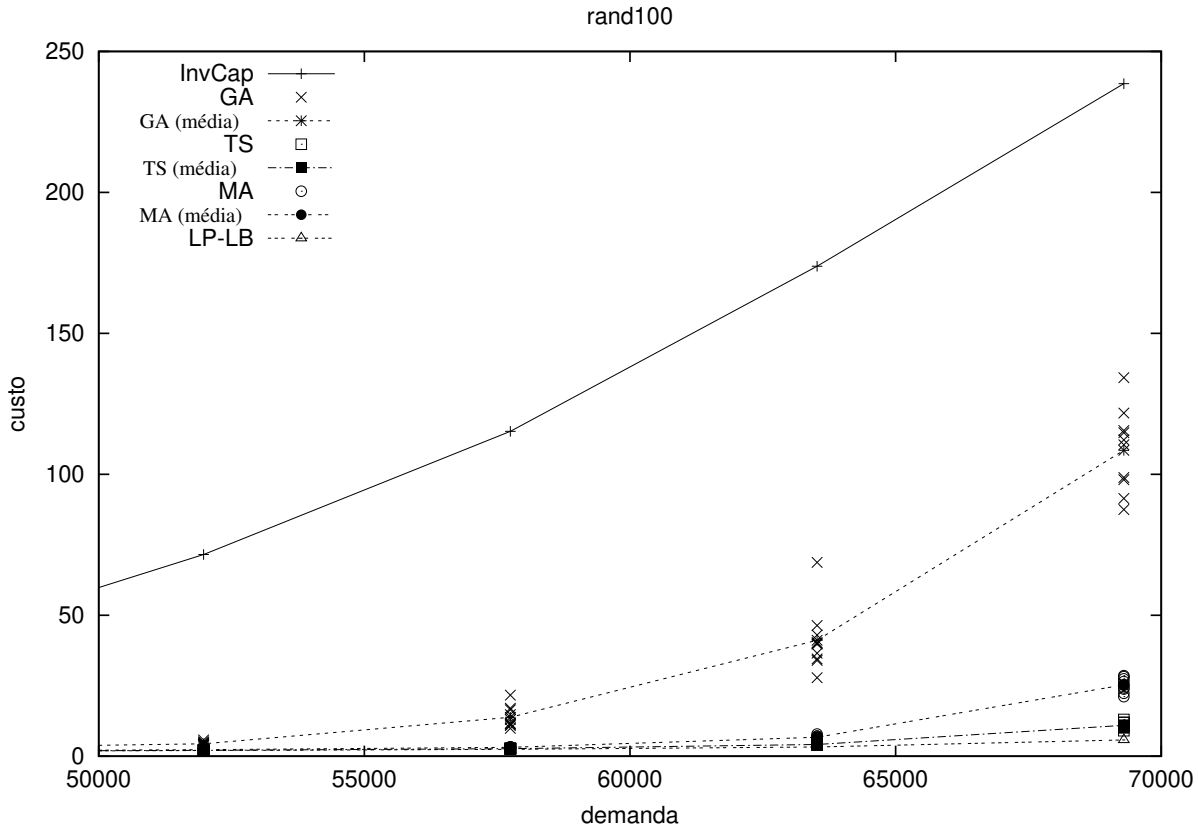

Figura 2.19: InvCap, GA, MA, TS e o limitante inferior LP na instância `rand100`.

Tabela 2.10: Custo de roteamento para a instância `rand100b` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
8382,862	1,000	1,015	1,011	1,000	1,00
16765,725	1,000	1,023	1,011	1,000	1,00
25148,587	1,020	1,063	1,013	1,000	1,00
33531,449	1,090	1,172	1,041	1,011	1,01
41914,312	1,220	1,385	1,186	1,093	1,07
50297,174	1,400	1,698	1,358	1,236	1,20
58680,037	1,960	2,665	1,540	1,407	1,36
67062,899	8,130	4,890	1,773	1,605	1,54
75445,761	20,510	12,206	2,170	1,851	1,76
83828,624	48,850	30,794	2,788	2,286	2,10
92211,486	94,050	77,555	5,179	3,151	2,78
100594,349	155,680	154,880	14,857	7,029	5,87
Total	335,910	290,346	34,927	23,669	21,69

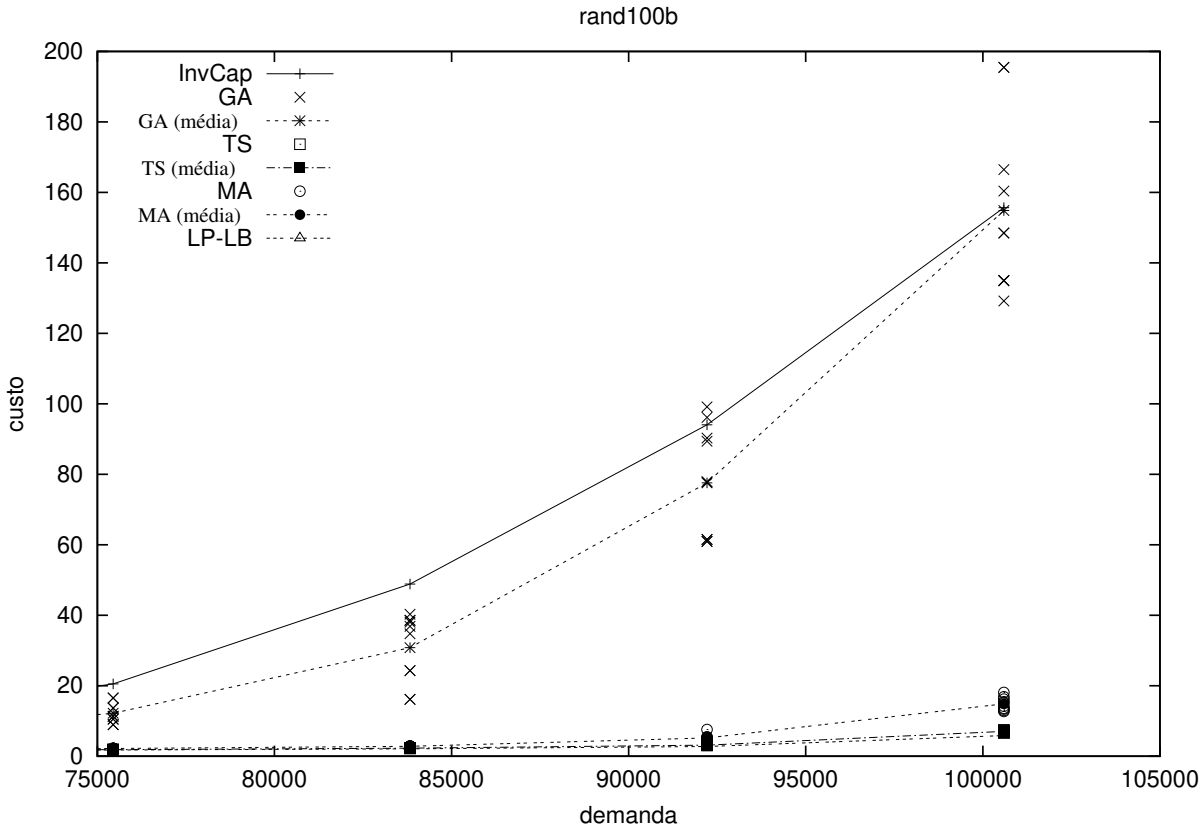

Figura 2.20: InvCap, GA, MA, TS e o limitante inferior LP na instância `rand100b`.

Tabela 2.11: Custo de roteamento para a instância `wax50` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
2117,622	1,000	1,000	1,000	1,000	1,00
4235,244	1,000	1,000	1,000	1,000	1,00
6352,865	1,015	1,000	1,000	1,000	1,00
8470,487	1,080	1,023	1,018	1,017	1,02
10588,109	1,171	1,100	1,088	1,086	1,08
12705,731	1,316	1,205	1,188	1,183	1,18
14823,353	1,605	1,339	1,315	1,309	1,29
16940,975	3,899	1,531	1,483	1,475	1,45
19058,596	20,630	1,798	1,756	1,754	1,72
21176,218	45,769	2,498	2,373	2,361	2,31
23293,840	84,409	6,280	5,988	5,998	3,27
25411,462	139,521	13,633	12,849	12,883	4,36
Total	302,415	33,407	32,058	32,066	20,68

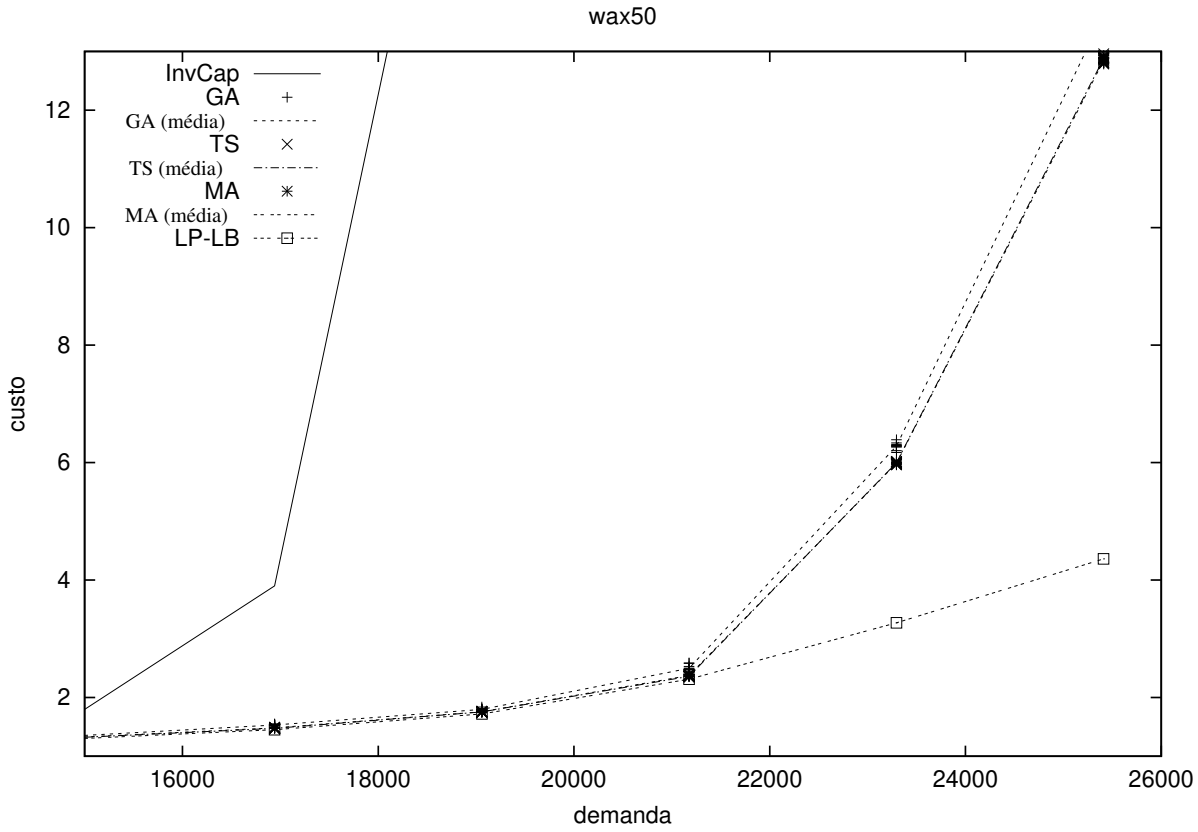

Figura 2.21: InvCap, GA, MA, TS e o limitante inferior LP na instância `wax50`.

Tabela 2.12: Custo de roteamento para a instância `wax50a` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
3287,217	1,000	1,000	1,000	1,000	1,00
6574,434	1,000	1,000	1,000	1,000	1,00
9861,651	1,002	1,000	1,000	1,000	1,00
13148,868	1,049	1,010	1,009	1,009	1,01
16436,085	1,129	1,050	1,031	1,028	1,03
19723,302	1,230	1,137	1,108	1,103	1,09
23010,519	1,393	1,250	1,218	1,210	1,19
26297,735	1,634	1,398	1,357	1,349	1,32
29584,952	2,706	1,593	1,514	1,510	1,48
32872,169	12,816	1,980	1,885	1,875	1,83
36159,386	38,708	4,042	3,874	3,870	2,84
39446,603	78,084	11,588	11,288	11,281	4,30
Total	141,751	28,048	27,284	27,235	19,09

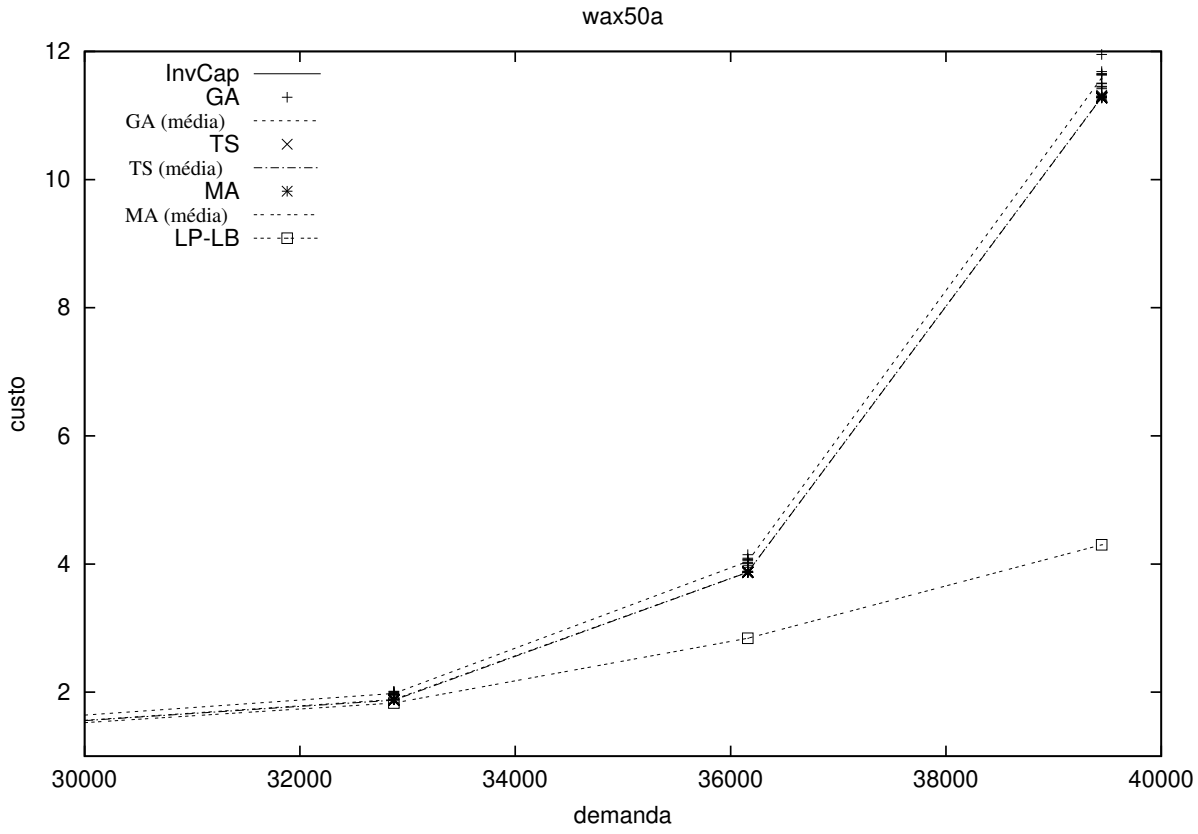

Figura 2.22: InvCap, GA, MA, TS e o limitante inferior LP na instância `wax50a`.

Tabela 2.13: Custo de roteamento para a instância `wax100` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
4039,477	1,000	1,005	1,006	1,000	1,00
8078,954	1,000	1,006	1,005	1,000	1,00
12118,431	1,010	1,010	1,007	1,002	1,00
16157,908	1,030	1,029	1,014	1,006	1,01
20197,385	1,090	1,065	1,029	1,012	1,01
24236,863	1,240	1,136	1,075	1,048	1,04
28276,340	4,660	1,268	1,186	1,130	1,11
32315,817	12,370	1,435	1,324	1,247	1,23
36355,294	23,020	1,835	1,698	1,599	1,57
40394,771	32,230	4,807	4,501	4,391	4,36
44434,248	40,640	8,953	8,317	8,191	8,14
48473,725	54,560	12,647	11,575	11,409	11,35
Total	173,850	37,196	34,737	34,035	33,82

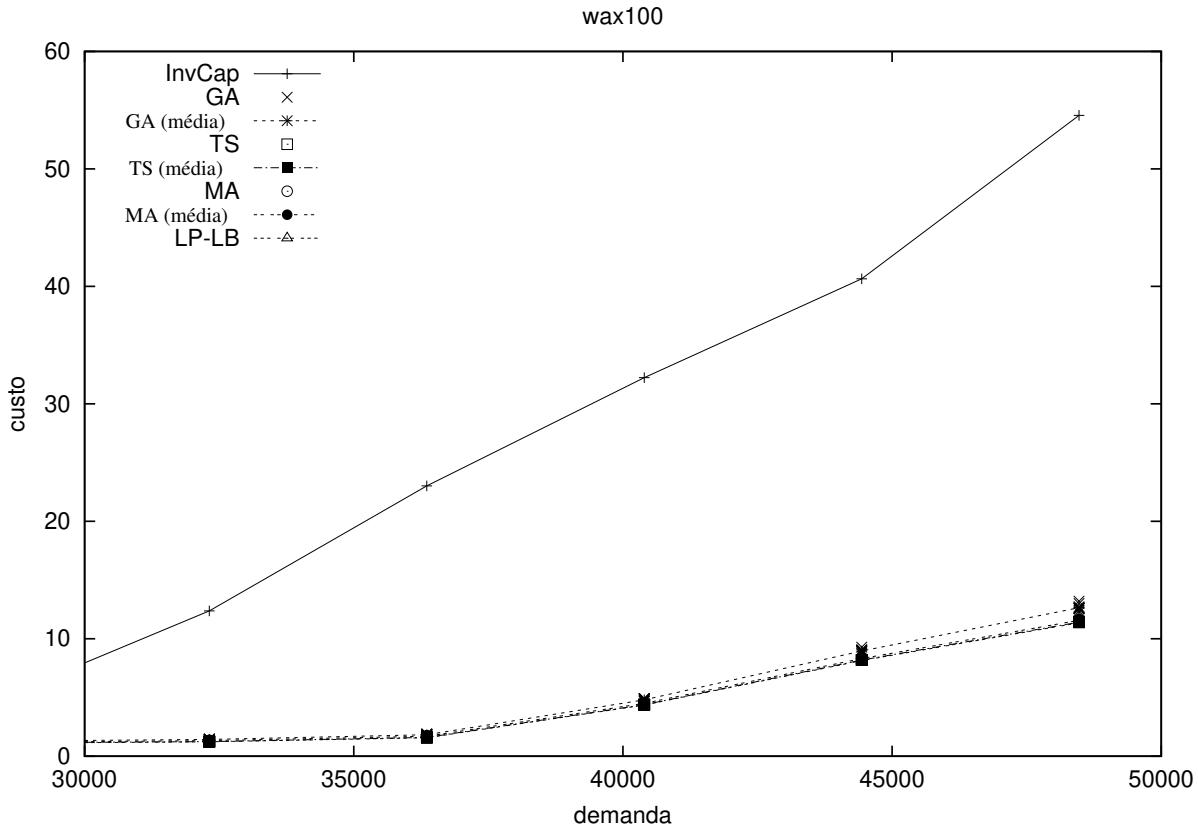

Figura 2.23: InvCap, GA, MA, TS e o limitante inferior LP na instância `wax100`.

Tabela 2.14: Custo de roteamento para a instância `wax100a` com demandas escaladas. Soluções são representadas pela média de 10 execuções de uma hora.

Demanda	InvCap	GA	MA	TS	LPLB
5291,092	1,000	1,011	1,011	1,000	1,00
10582,185	1,000	1,012	1,011	1,000	1,00
15873,277	1,010	1,022	1,011	1,001	1,00
21164,370	1,060	1,064	1,031	1,015	1,02
26455,462	1,130	1,138	1,064	1,036	1,03
31746,555	1,250	1,259	1,169	1,105	1,09
37037,647	1,490	1,458	1,313	1,218	1,19
42328,740	3,380	1,679	1,468	1,354	1,32
47619,832	18,080	2,224	1,732	1,596	1,54
52910,925	38,940	3,832	2,682	2,530	2,41
58202,017	69,400	12,132	9,998	10,436	9,62
63493,110	103,430	26,675	20,213	19,775	19,49
Total	241,170	54,506	43,703	43,066	41,71

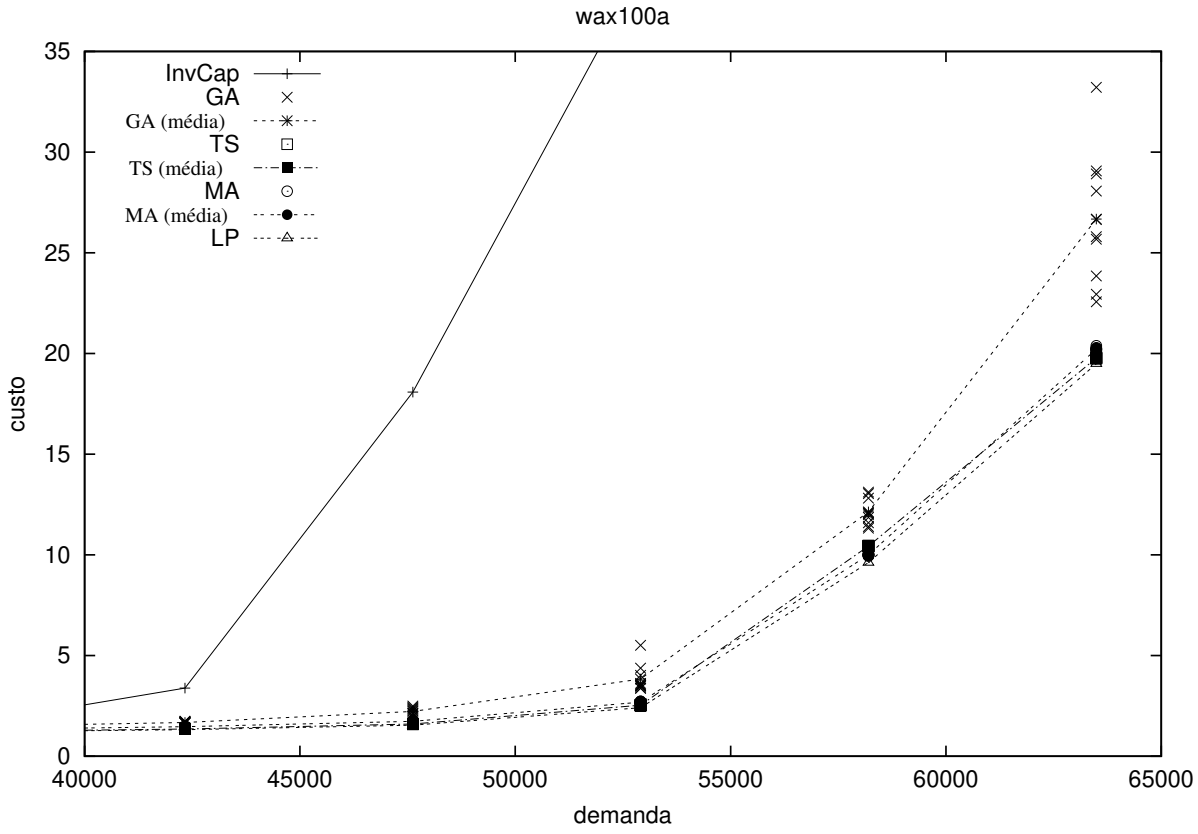

Figura 2.24: InvCap, GA, MA, TS e o limitante inferior LP na instância `wax100a`.

Algumas observações podem ser feitas a respeito dos resultados computacionais. O algoritmo genético (GA) encontrou soluções de melhor qualidade que as soluções obtidas pelo InvCap. A diferença entre a qualidade aumenta ainda mais com o aumento do tráfego. A diferença relativa $|\Phi^*_{InvCap} - \Phi^*_{GA}|/\Phi^*_{InvCap}$ entre os valores das soluções obtidas pelo GA e InvCap variaram de 13,6% na instância `rand100b` até 94,8% na `hier50b`.

MA encontrou soluções no mínimo tão boas quanto as do GA em todas as redes, considerando todos os níveis de demandas. A diferença da solução cresce com o aumento do tráfego. A diferença relativa $|\Phi^*_{GA} - \Phi^*_{MA}|/\Phi^*_{GA}$ variou de 1,9% na instância `hier100` para 88,0% na `rand100b`.

O MA não somente encontrou soluções de melhor qualidade, mas também em menos tempo. Como exemplo, veja as Figuras 2.25 e 2.26, as quais compara uma execução do MA e GA na instância `att` com demanda $D = 45134,146$.

A Figura 2.25 mostra o valor de melhor qualidade da população em função do tempo, enquanto a Figura 2.26 mostra o valor da melhor qualidade da população em função do número de gerações executadas. Estas figuras ilustram o quão próximo do limitante inferior LP o MA se aproxima e o quão rápido o MA converge comparado com o GA.

Das 13 classes de rede, o MA encontrou a melhor solução em sete delas, enquanto o TS encontrou a melhor solução nas outras seis. Nas classes `rand100` e `rand100b`, os valores das soluções encontrados pelo TS são 37,7% e 32,2% menores comparados com os encontrados pelo MA. Por outro lado, nas classes `hier50a` e `hier100a`, os valores de soluções encontrados pelo MA são 6,5% e 8,1% menores comparados com os encontrados pelo TS. Nas outras nove instâncias, a diferença relativa entre as soluções médias encontradas pelo MA e TS são pequenas, variando de 0,09% to 3,5%.

O MA e o TS na média encontraram valores de soluções que variaram de 0,6% a 105,5% da solução do limitante inferior. O MA encontrou soluções com menos de 10% de diferença relativa em seis classes.

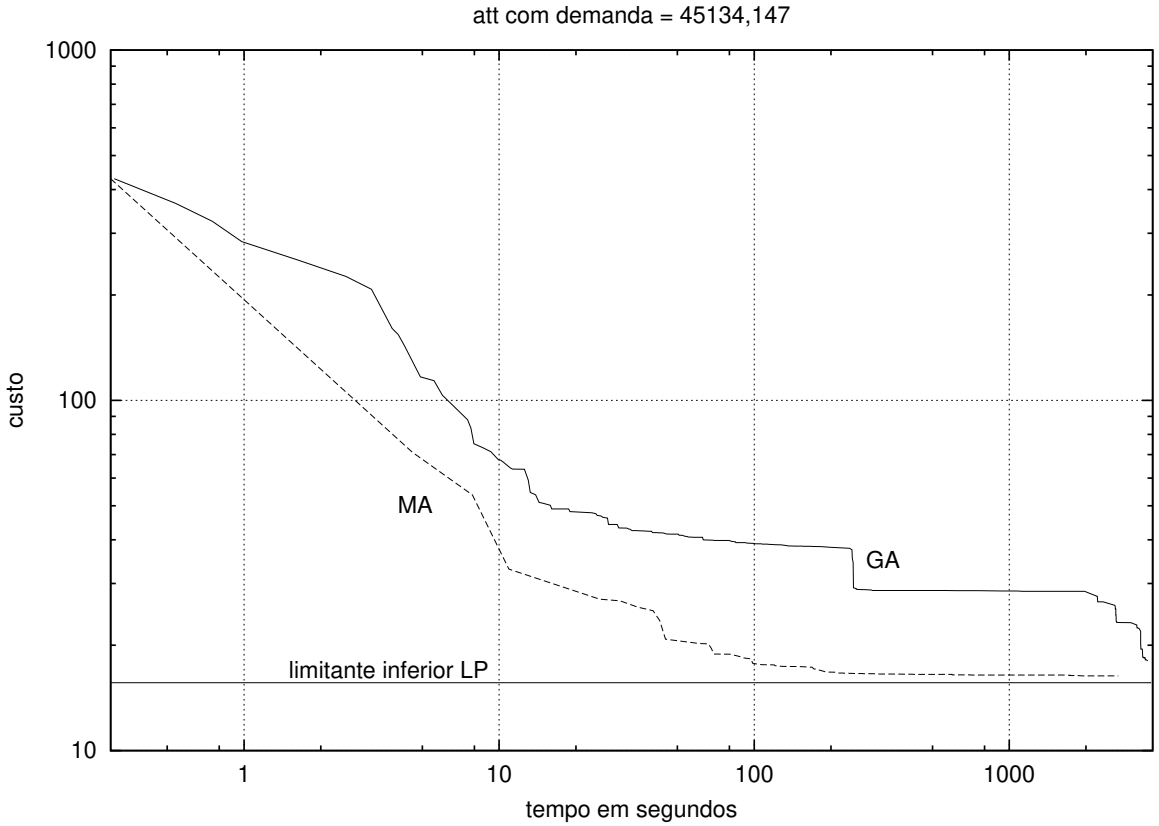

Figura 2.25: Custo em função do tempo em uma execução de uma hora: MA *versus* GA considerando a instância `att` com demanda 45134,146

2.5.3 Desempenho dos algoritmos para encontrar valores de soluções requeridos

Para estudar e comparar os tempos, usou-se a metodologia proposta por Aiex et al.[ARR02] e mais tarde explorada por Resende e Ribeiro [RR03]. Sem perda de generalidade, considerou-se as redes `att` com demanda igual a 37611,788, `hier50a` com demanda igual a 4106,407 e `rand50` com demanda igual a 35234,308 para ilustrar o procedimento geral observado pela maioria das instâncias. Para cada uma delas, 100 execuções independentes (usando sementes diferentes) para cada um dos algoritmos GA, MA e TS, considerando o valor do parâmetro `look4`. Cada execução é finalizada quando um valor de solução menor ou igual ao alvo `look4` é encontrado, ou o tempo limite de uma hora é atingido.

Três valores diferentes (representando os casos fácil, médio e difícil) são almejados para cada rede: 2,89, 2,77 e 2,64 para instância `att`, 2,32, 2,21 e 2,11 para a instância `hier50a`, e 2,93, 2,81

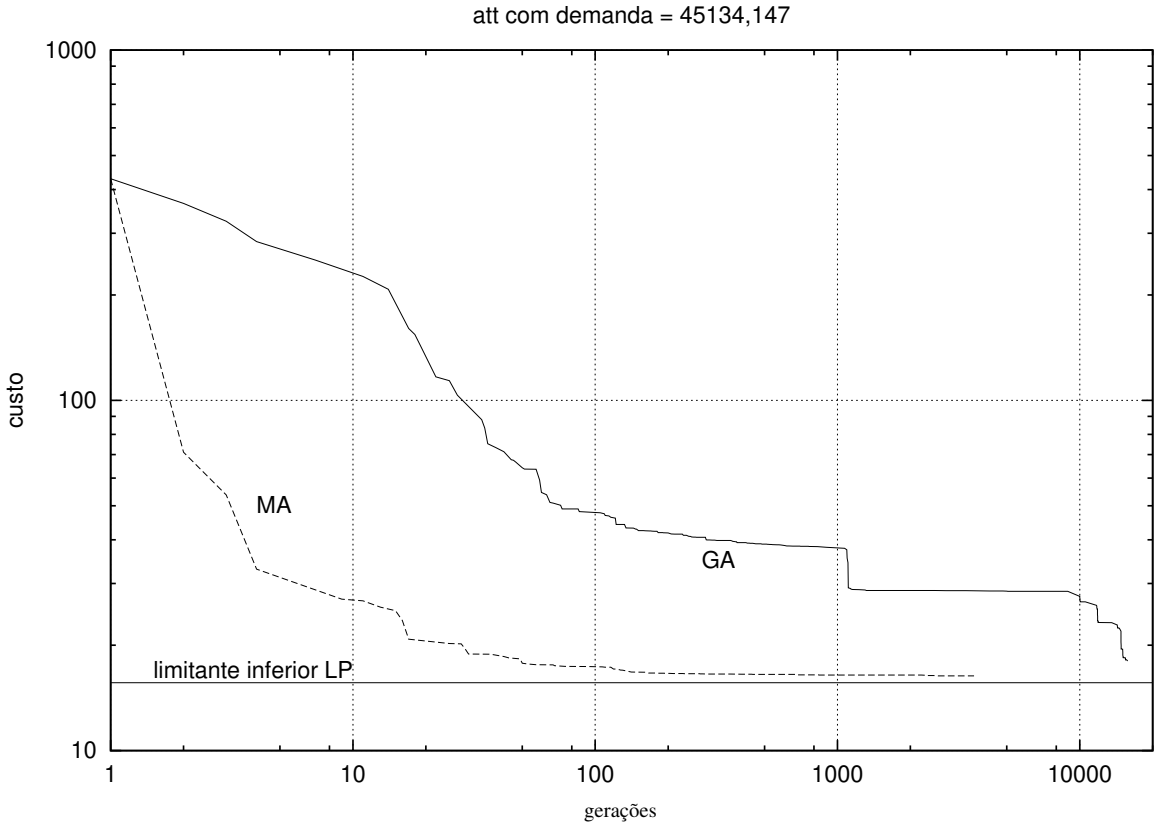

Figura 2.26: Custo em função do número de gerações em uma execução de uma hora: MA *versus* GA considerando a instância `att` com demanda 45134,146

e 2,68 para a instância `rand50`. Valores alvo diferentes são usados para cada instância, visto que as redes e demandas diferem entre elas. Resultados empíricos são apresentados nas Figuras 2.27 a 2.30. Os valores usados para `look4` estão indicados nas figuras. Estes valores foram calculados como sendo, aproximadamente, 15%, 10% e 5% maiores da melhor solução encontrada, para os casos fácil, médio e difícil, respectivamente.

Rodadas que falharam em encontrar a solução alvo com menos de uma hora de CPU foram descartadas. Para apresentar a distribuição empírica de cada algoritmo e instância, associou-se com o *i-menor* tempo computacional t_i a probabilidade $p_i = (i - \frac{1}{2})/100$, e adicionaram-se os pontos $z_i = (t_i, p_i)$ à figura, para $i = 1, \ldots, n_r$, sendo que $n_r \leq 100$ é o número de execuções o qual encontrou uma solução de menor ou igual valor ao alvo `look4` em no máximo uma hora de CPU.

O MA e o TS atingiram o alvo para todas as execuções da instância `att` (Figuras 2.27 e 2.28). O GA não atingiu a solução alvo, em uma hora de CPU, em 8 execuções para o caso fácil, 19 para

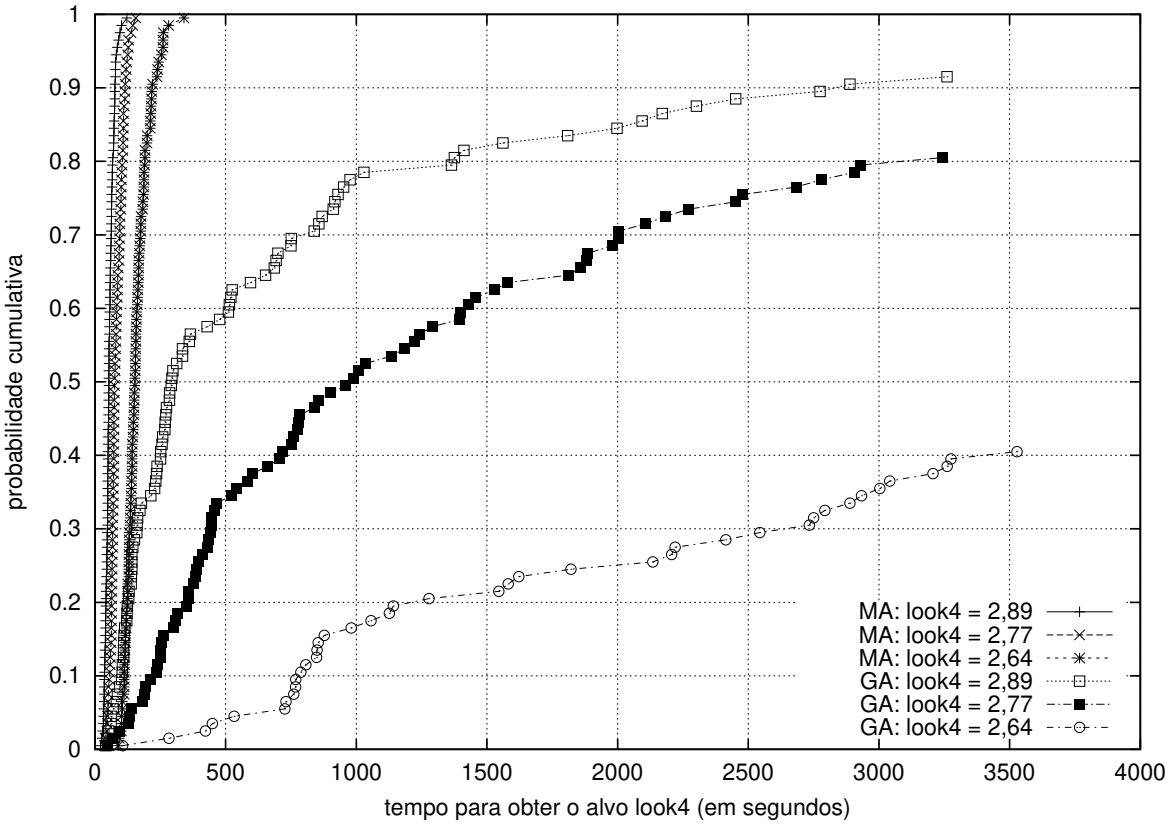

Figura 2.27: Tempo gasto para atingir o alvo: MA *versus* GA na rede `att` com demanda 37611,788

o médio e 59 para o caso mais difícil. O MA não foi somente mais rápido que o GA, mas também encontrou soluções menores. Como pode-se observar na Figura 2.27, em muitas execuções do GA o tempo gasto tem ordens de magnitude maiores que os do MA. Observando a Figura 2.28, percebe-se que o tempo de computação do MA é mais previsível que os do TS. Em mais de 30% das execuções o TS encontra dificuldades para convergir e gasta mais tempo. Um indício dessa diferença pode estar relacionada com o fato que o TS frequentemente pára num mínimo local, e o mecanismo de escape parece ser aplicado muitas vezes até que um mínimo local de melhor qualidade seja encontrado.

No caso da rede `hier50a` (Figuras 2.29 e 2.30), o MA apresentou melhor desempenho que o GA, como pode ser visto na Figura 2.29. O MA encontrou a solução alvo em uma hora de CPU em todas as execuções para os casos fácil e médio, e em 94 das 100 execuções para o caso mais difícil. Contrariamente, o GA encontrou a solução alvo em 44 execuções para o caso fácil, 14 para o caso médio e somente uma para o caso mais difícil (o qual não foi inserido na figura). O MA também

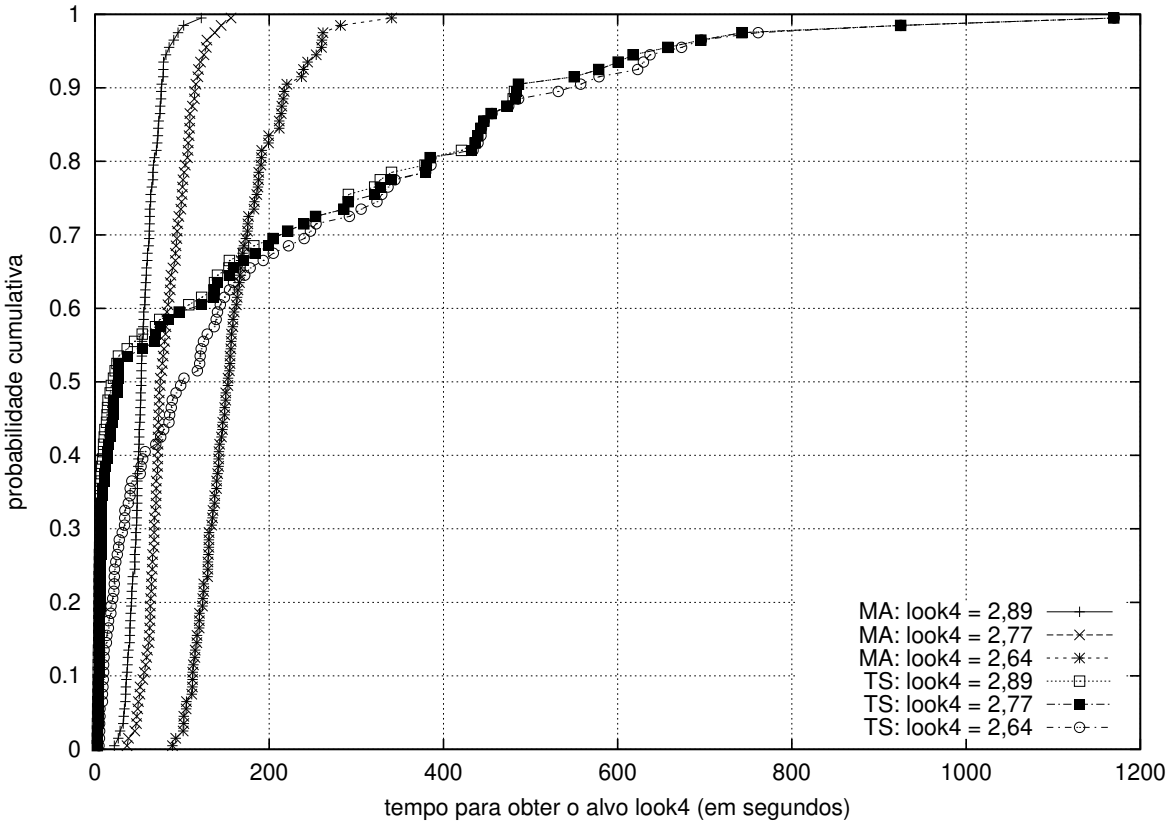

Figura 2.28: Tempo gasto para atingir o alvo: MA *versus* TS na rede `att` com demanda 37611,788

teve melhor desempenho que o TS para esta rede, como mostra a Figura 2.30. O TS encontrou a solução alvo para todas as execuções para o caso fácil. Entretanto, falhou em duas execuções para o caso médio. Para o caso mais difícil, em 48 execuções a solução alvo não foi encontrada.

As instâncias da classe `rand` estão entre as mais difíceis para o GA. A rede `rand50` é particularmente difícil para o GA, o qual falhou em encontrar a solução alvo para todas as execuções de cada um dos três alvos. Contrariamente, o MA encontrou a solução alvo em todas as execuções. Embora o MA apresente um melhor desempenho para as classes `att` e `hier` (como ilustrado nas Figuras 2.28 e 2.30), o TS tem melhor desempenho neste caso. Os dois algoritmos são robustos e encontram a solução alvo em todas as execuções, mas o TS em geral gasta menos tempo.

2.5.4 Comparação com o GA original

Conclui-se a apresentação dos resultados computacionais comparando o GA usado nos experimentos descritos acima com os resultados do GA apresentado por Ericsson, Resende e Pardalos

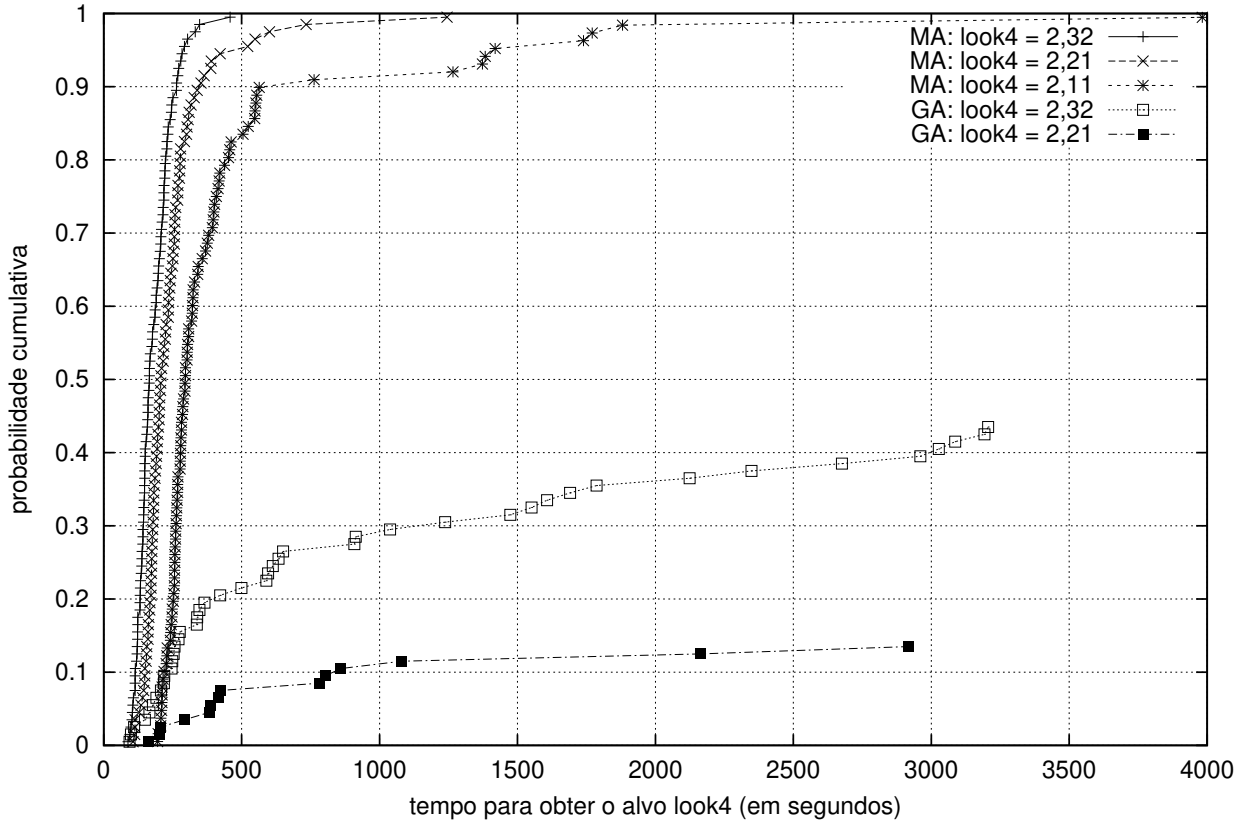

Figura 2.29: Tempo gasto para atingir o alvo: MA *versus* TS na rede `rand50` com demanda 35234,308

[ERP02] para mostrar que as duas implementações apresentam resultados semelhantes. Consequentemente, as conclusões obtidas ao comparar-se o MA e o GA são válidas para o GA de Ericsson, Resende e Pardalos [ERP02], o qual refere-se por GA^0.

Embora o GA e o GA^0 sejam baseados no mesmo código C, eles diferem em alguns aspectos:

- o GA tem população de 50 indivíduos, enquanto que o GA^0 possui 200 indivíduos na população. Nota-se que o tamanho de 200 foi usado em Ericsson, Resende e Pardalos para a instância deste experimento. Para instâncias maiores, eles usam população de 100 indivíduos.

- o GA gera uma solução inicial pelo InvCap e as demais são aleatórias com pesos no intervalo $[1,7]$, enquanto que o GA^0 gera uma solução inicial pelo InvCap, uma com pesos unitários e as demais são aleatórias no intervalo $[1,20]$.

- Os parâmetros $\alpha = |\mathcal{A}|/|\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}|$ e $\beta = |\mathcal{C}|/|\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}|$ diferem em valores nos dois

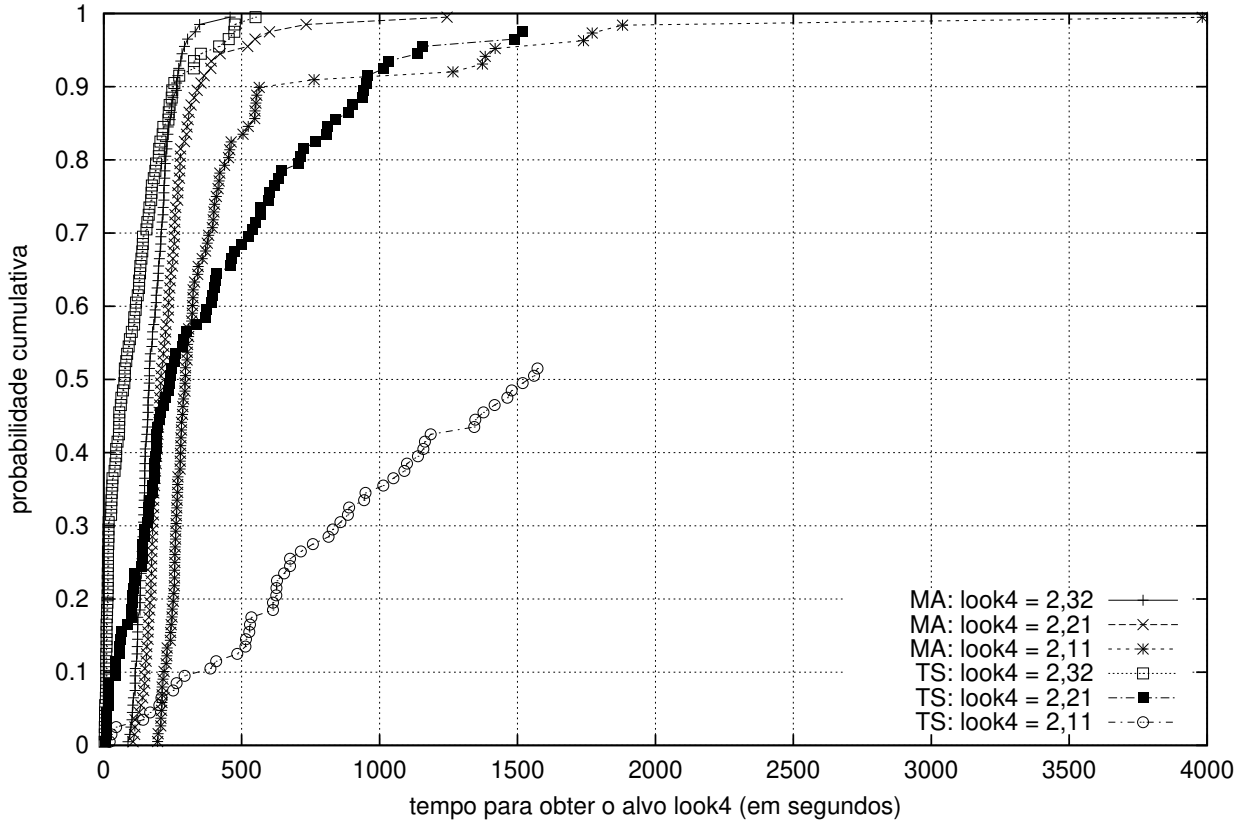

Figura 2.30: Tempo gasto para atingir o alvo: MA *versus* TS na rede `hier50a` com demanda 4106,407

algoritmos. GA usa $(\alpha, \beta) = (25\%, 5\%)$ enquanto GA^0 usa $(\alpha, \beta) = (20\%, 10\%)$,

Dez execuções são realizadas com cada um dos dois algoritmos nas redes `att`, `hier50a`, e `rand50`, com duas matrizes de demanda. As matrizes consideradas são as de maior demanda nos experimentos descritos anteriormente. Para a rede `att`, usou-se as demandas 41372,967 e 45134,146. Para a rede `hier50a`, usou-se as demandas 4517,048 e 4927,689. Para a rede `rand50`, usou-se as demandas 38757,739 e 42281,169.

Para o GA, usou-se os parâmetros descritos anteriormente nesta seção. Para o GA^0, usou-se os parâmetros descritos em Ericsson, Resende e Pardalos [ERP02].

A Tabela 2.15 mostra os valores de mínimo (`min`), médio (`avg`) e máximo (`max`) da solução final para as dez execuções do MA, GA e GA^0. A última linha da tabela apresenta a soma dos valores de cada coluna.

Os resultados experimentais mostram que o MA encontra melhores valores de mínimo, médio

e máximo que o GA e o GA^0, considerando todas as redes e níveis de demanda. Ainda, o GA apresenta um melhor desempenho que o GA^0, encontrando na média soluções que são 48% menores que as encontradas pelo GA^0. Esta grande diferença é devido principalmente ao pobre desempenho do GA^0 na rede `rand50`. Descartando os resultados do GA^0 nesta rede, GA e GA^0 são comparáveis, com o GA encontrado soluções em média 3,5% maiores que as encontradas pelo GA^0. Entretanto, com exceção da rede `att` com demanda 41372,968, o valor de custo mínimo encontrado pelo GA foi sempre menor que o mínimo do GA^0.

2.6 Conclusões

Este capítulo apresentou um algoritmo memético para resolver o problema de designação de pesos (*Weight Setting Problem*) para roteamento OSPF. Este algoritmo foi projetado, basicamente, com a modificação de alguns parâmetros do algoritmo genético proposto em [ERP02], combinado com uma busca local para melhorar as soluções geradas pelo *crossover*. Para a busca local é proposta uma vizinhança reduzida, além de procedimentos rápidos para a atualização do grafo de caminhos mínimos após a mudança do peso de um arco. Um procedimento eficiente de busca local é uma contribuição original proposta.

São também descritos alguns procedimentos especializados que buscam melhorar o desempenho de certas tarefas utilizadas pelos métodos, como o cálculo rápido das soluções vizinhas. Os resultados computacionais sobre quatro conjuntos diferentes de instâncias mostram que as implementações de metaheurísticas são muito superiores às heurísticas simples. As metaheurísticas apresentam resultados de congestionamento muito melhores do que a heurística InvCap, às custas de tempos computacionais bem maiores. Da comparação entre GA e MA, nota-se que MA é superior em todas as instâncias. Quando se comparam os algoritmos MA e TS, as conclusões são menos claras. Enquanto MA apresenta-se melhor nas classes `att` e `hier`, o algoritmo TS é superior nas classes `rand` e `wax`. Tanto o MA como o TS fizeram uso da avaliação rápida da vizinhança mencionada na Seção 2.2.5. A avaliação de uma solução ficou, aproximadamente, 15 vezes mais rápida para TS e 30 vezes mais rápida para MA. Nos casos em que o peso muda unitariamente, o algoritmo de Ramalingam e Reps [RR96a] pode ser especializado de modo a evitar o uso de pilhas (*heaps*) no caso de incrementos unitários, o que reduz em 2 a 3 vezes o tempo computacional gasto por

Tabela 2.15: Experimentos comparando o GA e o MA com o algoritmo genético GA^0 de Ericsson, Resende e Pardalos [ERP02]. Dez execuções de uma hora foram executadas para cada um dos algoritmos. Valores do custo mínimo, médio e máximo das soluções foram listados para cada algoritmo em cada instância.

		MA			GA			GA^0		
Instância	Demanda	min	avg	max	min	avg	max	min	avg	max
att	41372i,967	4,227	4,382	6,604	4,624	5,194	5,833	4,469	5,034	7,517
att	45134,146	16,043	16,433	17,622	17,210	20,983	26,802	17,551	20,017	24,244
hier50a	4517,048	3,616	3,674	3,978	4,501	5,273	6,235	4,672	5,045	6,206
hier50a	4927,689	14,695	15,123	17,184	17,347	20,968	25,252	18,489	20,461	23,309
rand50	38757,739	4,503	4,607	4,669	6,259	7,738	10,073	22,599	25,623	30,404
rand50	42281,169	14,852	15,041	15,331	20,165	27,375	33,994	80,535	93,876	98,750
	Soma	55,936	59,260	65,388	70,106	87,531	109,189	148,315	170,056	190,430

este algoritmo e em 40 vezes o tempo gasto pelo algoritmo Dijkstra. Como a busca local de MA é baseada em incrementos unitários, esta pode se beneficiar da rapidez do algoritmo especializado.

Capítulo 3

Projeto Otimizado de Redes OSPF

O Capítulo 2 abordou um problema de *gerenciamento* de rede, o problema de designação de pesos aos arcos de uma rede Internet intra-domínio, sob roteamento OSPF. O objetivo é utilizar os recursos da rede de forma eficiente. Neste capítulo é estudado um problema de *projeto* de rede, o qual trata da minimização da alocação de capacidade às linhas de transmissão de tal forma a satisfazer a demanda sem que haja sobrecarga de nenhum arco, inclusive quando houver falhas na rede. Neste caso, a preocupação não é somente em projetar uma rede segura e com dimensões adequadas para o roteamento dos dados, mas também com o menor custo possível. Neste caso também o roteamento é realizado sob o protocolo OSPF intra-domínio.

3.1 Introdução

A cada linha de transmissão da Internet está associado um custo. Seja um simples cabo de cobre, uma fibra ótica ou mesmo na transmissão sem fio, sua instalação e manutenção implica em gastos elevados. O custo do cabo em si é incomparavelmente menor que o custo relacionado com a construção adequada da estrutura física por onde este cabo deva passar. Muitas vezes estas estruturas são compartilhadas por mais de uma corporação. Em geral a linha é instalada por uma das corporações e posteriormente outras podem alugá-la. Dessa forma, à empresa não incorre um custo de instalação, mas sim um custo de arrendamento, além do custo de manutenção. Como estabelecer/instalar uma linha de transmissão implica em valores altíssimos, em geral tais redes compõem grafos bastante esparsos e a maior motivação para novas instalações dentro de um sistema autônomo é interligar novos roteadores. Caso a demanda de tráfego numa linha aumente, tornando

necessária uma banda adicional, a solução em geral consiste no aumento das capacidades das linhas já existentes, economizando os investimentos em novas ligações físicas.

Com o crescente aumento do tráfego na Internet, a necessidade da expansão dos meios de transmissão aparece como consequência direta. Supondo que não esteja prevista a interligação de novos roteadores, nem o aumento de tráfego entre roteadores específicos da rede, o que se busca é utilizar a estrutura já existente, apenas aumentando as capacidades dessas linhas. Um problema de otimização combinatória é então caracterizado: qual a capacidade mínima que cada arco deve possuir para suportar o roteamento da demanda que esta rede requer? O objetivo de minimizar o aumento da capacidade total está diretamente associado com a redução de custos de projeto.

Além de um dimensionamento de baixo custo, é indispensável que a rede resultante seja eficiente. Neste caso, por rede eficiente considera-se aquela que satisfaça a demanda existente, sem sobrecarregar nenhum arco, ou seja, com tráfego balanceado. O volume de dados passante depende da capacidade dessa linha. A razão entre carga e capacidade caracteriza o que se chama de *utilização* do arco. Dessa forma, a expansão é realizada de tal forma a manter o valor da utilização máxima de um arco sempre abaixo do valor 1 (carga passante igual à capacidade do arco), garantindo que não haja sobrecarga.

Um arco pode ser composto por vários cabos. Cada cabo possui uma capacidade associada e a capacidade do arco será igual à soma das capacidades dos cabos que o compõem. Supondo cabos idênticos, a capacidade é limitada a valores modulares discretos $0, m, 2m, 3m, \ldots$, isto é, múltiplos de um valor m. Na área de projeto de redes, o número de cabos que um arco contém é chamado de *multiplicidade* deste arco. Portanto, o problema de otimizar a capacidade de uma rede consiste em minimizar o número total de multiplicidades, o que implica diretamente em diminuir os custos associados. Assume-se que a topologia é dada, e apenas as multiplicidades dos arcos devam ser determinadas. Ainda, supõe-se que todos os cabos que compõem um arco possuem a mesma capacidade.

Assim como no Capítulo 2, considera-se aqui o projeto de uma rede interna a um sistema autônomo (AS), sob roteamento OSPF ou IS-IS. Para efeito do roteamento, cada cabo é tratado como se fosse um arco independente. Dessa forma, uma rede com κ cabos usando a estrutura física do arco a entre os roteadores u e v será tratada como tendo κ arcos, isto é, $m_a = \kappa$, entre estes dois

roteadores. Na realidade, a estrutura lógica da rede considera cada multiplicidade como um arco independente, enquanto que o outro caso está associado com a estrutura física. Veja o exemplo ilustrativo da Figura 3.1. Esta figura apresenta a estrutura física e lógica dos arcos $v_1, \ldots, v_4$ saintes do nó u. Os arcos presentes no grafo de caminhos mínimos estão representados por linha cheia.

Figura 3.1: Estruturas lógica (à esquerda) e física (à direita) dos arcos saintes do nó u.

Segundo Moy [Moy98], é também uma atribuição do protocolo de roteamento determinar como a rede deve reagir à contingências ou mudanças na topologia do AS, tais como arcos sendo removidos ou reinseridos, roteadores sendo desativados, mudanças de políticas da rede, etc. Nestas situações, o tráfego deve ser desviado para usar somente os caminhos mínimos da parte operante da rede. Mas os arcos que compõem os caminhos mínimos da parte operante da rede podem estar sendo usados em sua capacidade máxima, não suportando qualquer carga adicional. Por isso, é desejável que se projete uma rede de forma que ela use somente parte de sua capacidade e deixe uma parte ociosa disponível para eventuais "emergências". Ao definir as capacidades dos arcos, é possível adotar esta política. Dessa forma, além de se almejar uma rede eficiente e de baixo custo, deseja-se também uma rede "segura".

Além do algoritmo genético aqui proposto [BFRT03], não se encontrou na literatura menção a outro algoritmo de otimização para o dimensionamento da capacidade em função da definição das multiplicidades e considerando o roteamento OSPF.

Os estudos referentes a roteamento de dados estão concentrados nos EUA. Foi nos EUA que as redes surgiram, e é lá que há o maior número de computadores conectados à Internet, com mais e maiores ASs. Atualmente, 70% das conexões de Internet do mundo estão localizadas nos EUA

(Tabela 3.1[1]).

Tabela 3.1: Quantidade de *hosts* por país.

País	*#hosts*	País	*#hosts*
1^0 Estados Unidos	120.571.516	16^0 México (.mx)	1.107.795
2^0 Japão (.jp)	9.260.117	17^0 Bélgica (.be)	1.052.706
3^0 Itália (.it)	3.864.315	18^0 Polônia (.pl)	843.475
4^0 Canadá (.ca)	2.993.982	19^0 Áustria (.at)	838.026
5^0 Alemanha (.de)	2.891.407	20^0 Suíça (.ch)	723.243
6^0 Reino Unido (.uk)	2.583.753	21^0 Noruega (.no)	589.621
7^0 Austrália (.au)	2.564.339	22^0 Argentina (.ar)	495.920
8^0 Holanda (.nl)	2.415.286	23^0 Rússia (.ru)	477.380
9^0 ***Brasil (.br)***	2.237.527	24^0 Nova Zelândia(.nz)	432.957
10^0 Taiwan (.tw)	2.170.233	25^0 Coréia(.kr)	407.318
11^0 França (.fr)	2.157.628	26^0 Hong Kong (.hk)	398.151
12^0 Espanha (.es)	1.694.601	27^0 Singapura (.sg)	338.349
13^0 Suécia (.se)	1.209.266	28^0 Portugal (.pt)	291.355
14^0 Dinamarca (.dk)	1.154.053	29^0 Hungria (.hu)	254.462
15^0 Finlândia (.fi)	1.140.838	30^0 República Tcheca (.cz)	239.995

Em geral, os administradores dos ASs compram os pacotes de empresas especializadas em problemas de otimização de projeto e expansão de redes. Pouquíssimas empresas têm uma dimensão tal que permite uma equipe dedicada a esse problema. A AT&T, que já foi a maior empresa do mundo nos anos 80 e que atualmente possui um dos maiores ASs do mundo, tem interesse em ter a sua própria solução, em vez de terceirizá-la. No Brasil, o maior ASs é o da *Embratel*.

Nas seções seguintes o problema é descrito e um algoritmo genético é proposto para resolvê-lo. Inicialmente, o problema base é descrito e, em seções posteriores, são analisadas variações deste em relação à função objetivo, características da topologia e tratamento de falhas, entre outras. Os resultados computacionais são apresentados considerando todas as varições do problema abordadas nas sub-seções anteriores. O capítulo finaliza com uma discussão dos resultados e extensões do problema.

[1]Fonte: *Network Wizards - www.nw.com*. Estes dados são de janeiro de 2003.

3.2 Problema de designação das multiplicidades aos arcos de uma rede

Considerando uma rede com $|V|$ roteadores e $|E|$ linhas de transmissão, assim como um valor de capacidade associado a cada arco e um conjunto de demandas entre pares de roteadores origem-destino, o *problema de designação das multiplicidades* para roteamento OSPF consiste em minimizar o número de multiplicidades da rede suficiente para rotear a demanda sem causar sobrecarga de nenhum arco. Ainda, visto que se trata de um problema de projeto, pode-se estipular um valor máximo MU para utilização dos arcos. Dessa forma, uma instância deste problema contém os seguintes dados de entrada:

- uma rede composta por um conjunto V de nós e um conjunto E de arcos;
- um vetor de $|E|$ posições informando a capacidade c_i de cada arco i;
- uma matriz de demanda indicando a carga D^i_j entre cada roteador origem i para o roteador destino j;
- MU: utilização máxima de cada arco da rede.

O valor de MU deve ser informado pois o congestionamento da rede depende dele. O caso em que $MU = 1$ significa que a carga passante nos arcos pode ser igual as suas capacidades, o que não é uma alternativa usada em redes reais.

Para questões de roteamento, *cada cabo é tratado como se fosse um arco independente*. Segundo as regras do protocolo OSPF, *a carga passante por cada nó u, com destino em t, é dividida igualmente entre os arcos saintes de u e presentes no grafo de caminhos mínimos com destino em t*. É muito importante salientar que a regra do OSPF não é referente à carga total (para todos os destinos) passante por cada nó, e sim, à carga com destino em t. Dessa forma, *a carga passante por cada nó u, com destino em t, é dividida entre os arcos presentes no grafo de caminhos mínimos, proporcionalmente as suas multiplicidades*. A Figura 3.2 mostra visualmente como fica a distribuição da carga (indicada na figura) para o sub-grafo da Figura 3.1.

Para determinar a multiplicidade que um arco deve ter, é preciso antes atribuir um peso a cada arco. Então, uma solução do problema das multiplicidades é composta por:

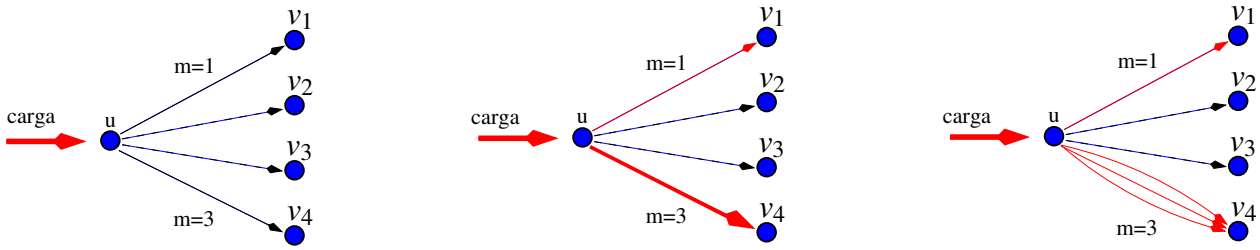

Figura 3.2: Carga passante e arcos saintes do nó u (à esquerda) com suas respectivas estruturas física (figura do meio) e lógica (à direita).

- um vetor w de $|E|$ posições informando o peso w_i associado a cada arco $i \in E$;
- um vetor m de $|E|$ posições informando a multiplicidade m_i para cada arco $i \in E$. O vetor m depende dos valores do vetor w.

A atribuição dos pesos antecede a atribuição das multiplicidades, pois o cálculo da multiplicidade vai depender do fluxo passante pelo arco. A multiplicidade de cada arco é inicializada com valor 1, e posteriormente, para ajustar seu valor, tem-se primeiro de saber o fluxo de carga passante nesse arco. Para tanto, o grafo de caminhos mínimos deve ser calculado, baseado no vetor w. A Figura 3.3 apresenta visualmente a relação de dependência que as multiplicidades têm em relação ao peso dos arcos. Abaixo de cada grafo, encontram-se as multiplicidades dos arcos calculadas considerando *MU=1,00*, considerando capacidade igual a 5. A multiplicidade de um arco é igual ao maior inteiro da razão entre a carga total e sua capacidade$*MU$.

Na Figura 3.3 tem-se apenas uma demanda (entre os nós v_1 e v_3). Caso fossem adicionados mais pares de demanda com destinos em v_3, possivelmente as multiplicidades teriam uma maior variedade de valores. A Figura 3.4 apresenta este caso. Considere apenas o vetor de pesos correspondente ao grafo `B)` da Figura 3.3. A adição de outra demanda com destino em v_3, causou o aumento do valor da multiplicidade do arco a_6. Para o grafo apresentado, os valores das multiplicidades estão especificados para `MU`=1,00 e `MU`=0,80, considerando que cada arco tenha capacidade igual a 5. Quanto menor o valor de *MU*, maiores tendem a ser os valores das multiplicidades.

O ajuste de carga *vs.* multiplicidade é mais complexo quando cargas com diferentes destinos passam por um nó com múltiplos caminhos mínimos, pelo menos para um desses destinos. O ajuste se torna mais complicado porque ao mesmo tempo em que a multiplicidade deve se ajustar

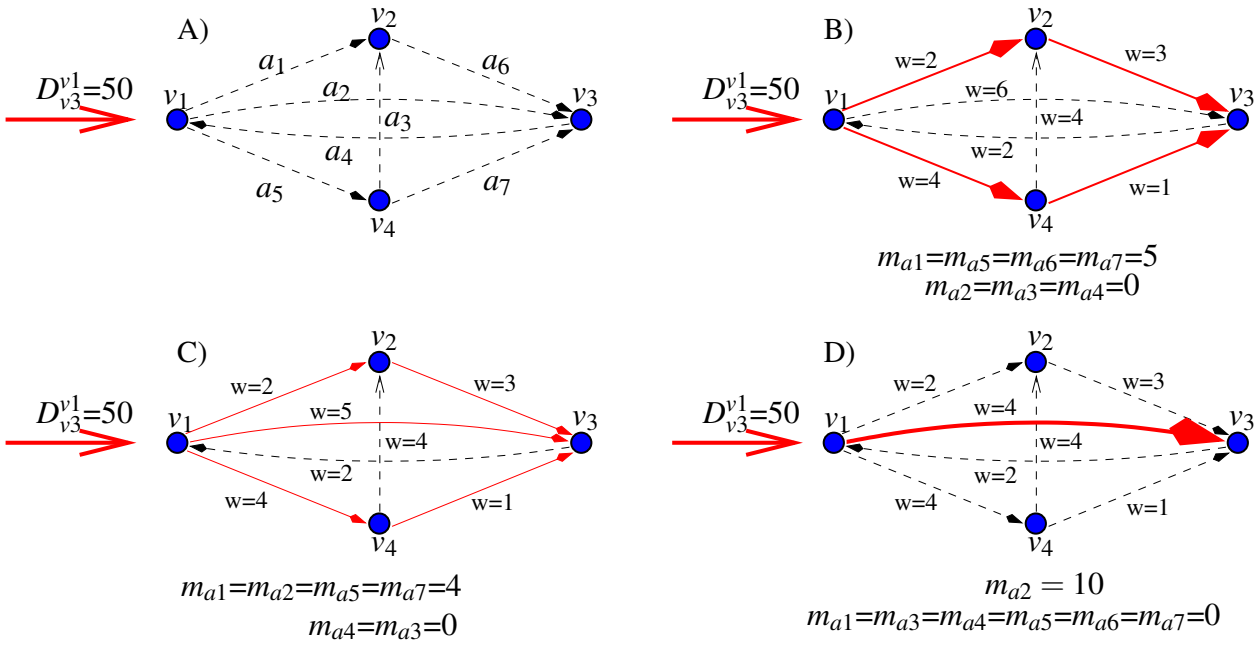

Figura 3.3: O grafo `A)` apresenta uma rede, sendo que cada arco tem capacidade *5*, e os grafos `B)`, `C)` e `D)` apresentam a distribuição das cargas pela rede mudando apenas o peso do arco a_2, o qual apresenta pesos 6, 5 e 4, respectivamente.

à carga, a carga deve se ajustar às multiplicidades para seguir a regra de balanceamento de carga do OSPF. A Figura 3.5 apresenta um exemplo desse caso. O grafo tem dois nós destinos (v_3 e v_4), sendo que o tráfego destas duas demandas passam pelo nó v_1, o qual possui dois caminhos mínimos para v_3. Como as multiplicidades serão definidas nesse processo, mas valores iniciais devem ser considerados para o cálculo das cargas, supõe-se valores iniciais iguais a um para todos os arcos. Os dois primeiros grafos apresentam os pesos e as multiplicidades inicias, respectivamente. Os demais grafos da esquerda correspondem ao ajuste de carga, enquanto que os da direita correspondem ao ajuste das multiplicidades. Novamente supõe-se que a capacidade de cada arco é igual a 5. A rede está estável quando nenhuma multiplicidade muda ao ser realizado o ajuste. Finalmente, no último grafo, o valor *0* é atribuído a todos os arcos que não tiveram carga passante durante a definição do vetor de multiplicidades.

É muito importante observar que este exemplo é muito simples. Em geral, em cada nó passam cargas com muitos destinos. Ainda, dificilmente ocorre o caso em que um arco não tem qualquer fluxo, considerando todos os destinos de carga.

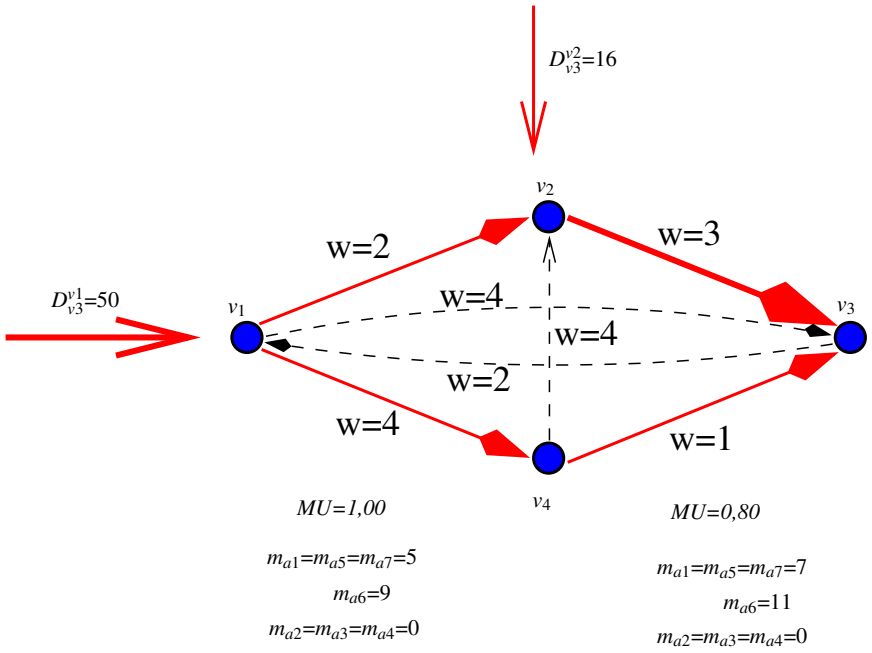

Figura 3.4: Grafo com duas demandas direcionadas ao mesmo destino. Assumindo que a capacidade de cada arco é *5*, abaixo do grafo encontram-se os vetores de multiplicidades considerado *MU=1,00* ou *MU=0,80*.

Este capítulo apresenta uma discussão sobre o problema da designação mínima de multiplicidades e um algoritmo genético (*Genetic Algorithm* - GA) é proposto para resolvê-lo. Ainda, são apresentadas várias especializações do problema, todas representadas por parâmetros opcionais do GA. Ao final, resultados computacionais são apresentados e discutidos.

3.3 Definições, restrições e variações do problema

O problema consiste em definir as multiplicidades dos arcos de uma rede de forma a satisfazer a demanda requerida sem sobrecarregar nenhum arco e objetivando minimizar o número total de multiplicidades. Na prática este problema pode aparecer especializado para as necessidades de sua aplicação. Por isso, algumas restrições podem ser impostas, ou uma variação do problema pode ser necessária. Nas seções seguintes serão discutidas algumas das variações e restrições que o problema pode apresentar. Inicia-se com uma discussão sobre possíveis definições de função objetivo.

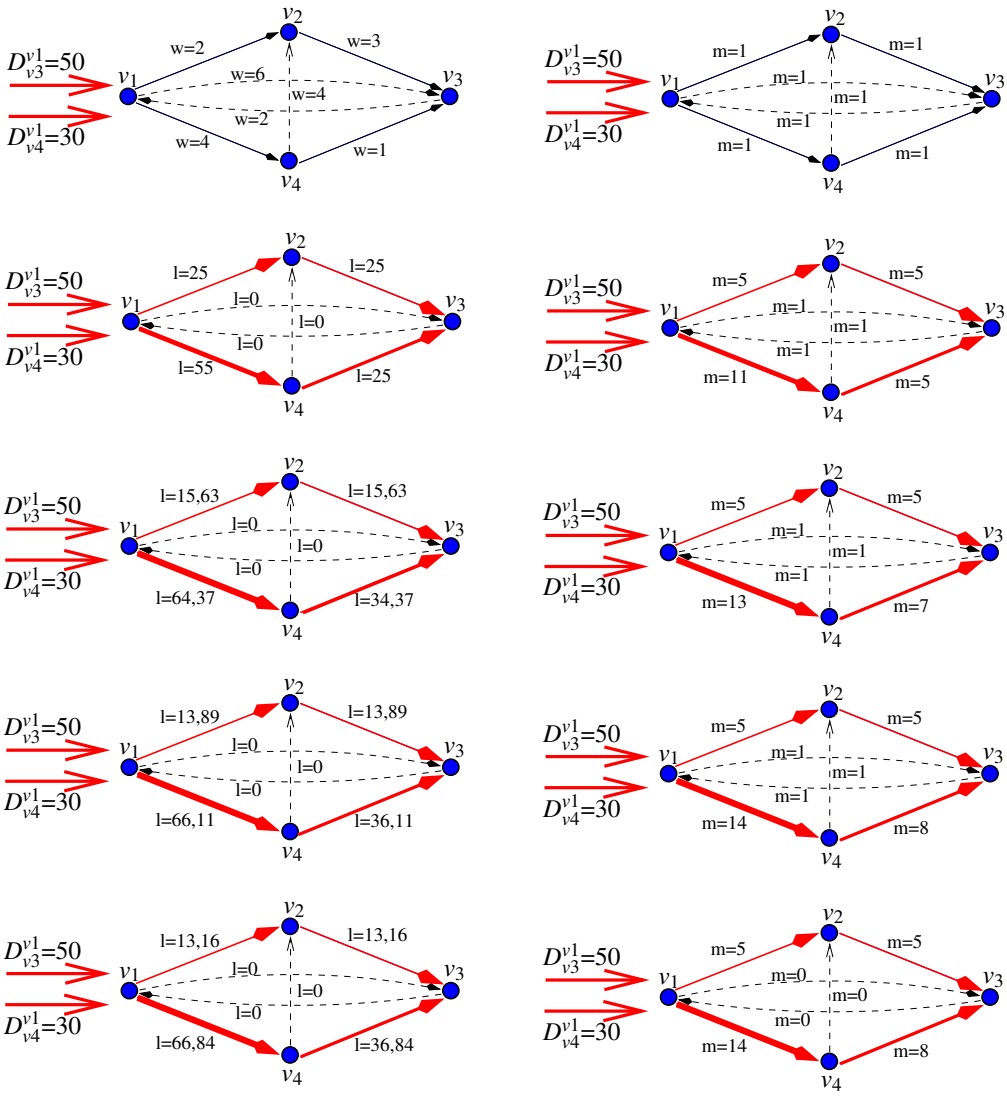

Figura 3.5: Ajuste de carga *vs.* multiplicidade.

3.3.1 Objetivos do problema

Um objetivo possível é minimizar a soma do número de multiplicidades da rede. Ou seja $Obj_1 = Min\sum_{a\in E} m_a$, sendo m_a a multiplicidade do arco a. Dependendo da aplicação, o objetivo pode ser outro. Por exemplo, o objetivo poderia ser minimizar a capacidade total da rede. Neste caso, uma multiplicidade deve estar associada a um valor de capacidade. Neste caso, o objetivo seria $Obj_2 = Min\sum_{a\in E}(m_a * c_a)$. Se a capacidade de todos os arcos forem iguais, os dois objetivos apresentarão as mesmas soluções.

Outro objetivo plausível pode ser minimizar a extensão total dos cabos. Considerando que cada multiplicidade representa um cabo, o objetivo seria $Obj_3 = Min\sum_{a\in E}(m_a * length_a)$, sendo $length_a$ o comprimento do arco a. Para o problema com este objetivo, deve-ser fornecer dados suficientes para obter-se o comprimento de cada arco.

Na subseção a seguir será apresentada uma restrição associada à computação da multiplicidade para arcos simétricos.

3.3.2 Restrição de multiplicidades iguais para arcos simétricos

Pode-se ter a restrição de que as multiplicidades de arcos simétricos (do tipo $(\overrightarrow{u,v})e(\overrightarrow{v,u})$) devam ser iguais. Nenhum dado extra deve ser fornecido para o problema, apenas o parâmetro Υ deve ser definido com valor 1 ou 0, caso tal restrição deva ou não ser considerada. A rede do *backbone* do AS da AT&T considera tal restrição.

A próxima subseção apresenta o caso em que deseja-se manter uma utilização máxima dos arcos, mesmo se uma falha na topologia da rede acontecer.

3.3.3 Designação das multiplicidades considerando falhas na topologia física da rede

Considere a contingência de ocorrer uma falha na rede. Por exemplo, um roteador pode temporariamente ter problemas de funcionamento, ou um arco pode ser desativado por algum motivo. Ainda, há casos em que uma estrutura física suporta a passagem de arcos com destinos variados. Ao conjunto de arcos que, em parte de sua extensão, utilizam a mesma estrutura física dá-se o nome de *span*. No caso em que um *span* falha, todos os arcos que o compõe serão afetados.

No caso de qualquer uma dessas falhas, a carga que passava pela área inoperante deve ser desviada para a parte da rede em funcionamento. Mas se a parte da rede em funcionamento não suportar a adição dessa carga extra, podem ocorrer sérios problemas no roteamento, inclusive perda de dados. Por motivos de segurança, uma das medidas tomadas no sentido de evitar a sobrecarga de alguns arcos é projetar a rede para utilizar somente parte da capacidade dos arcos, deixando o restante da banda livre para o caso de "emergências". A motivação é a possibilidade de roteamento de carga extra em casos de falhas de parte da rede, ou mesmo suportar um fluxo intenso de dados não esperado.

À falha de um arco, *span* e roteador, estão associados os parâmetros *lf* (*link failure*), *sf* (*span failure*) e *rf* (*router failure*), respectivamente. O valor 1 indica a consideração da falha e 0 o caso contrário.

Ao definir os valores das multiplicidades pode-se usar um segundo valor de utilização máxima, *MUf*, para o caso de rede com falha. A próxima seção aborda a questão de ter valores de utilização máxima diferenciados para o caso de rede operando com ou sem falha.

3.3.4 Utilização máxima dos arcos

Ter a rede funcionando com utilização máxima dos arcos igual a 1 não é usado na prática, pois qualquer carga extra pode resultar em sobrecarga de um ou mais arcos. Como visto no Capítulo 1, uma das medidas recomendadas para a prevenção de congestionamento é deixar parte da capacidade do arco livre para o caso de emergência. Esta restrição pode ser imposta de forma que a máxima utilização de um arco seja um valor *MU* informado como parâmetro. Ainda o valor de *MU* pode ser diferenciado no caso da rede estar ou não considerando uma falha. No caso de uma falha da rede, pode-se diminuir a capacidade livre de cada arco. O parâmetro *MUf* ($MUf \geq MU$) deve ser informado para indicar a máxima utilização de um arco em caso de falha.

Dependendo do tipo de falha, poderiam ser estabelecidos valores diferentes de utilidade máxima. Por questões de simplicidade, consideramos o mesmo valor *MUf* para qualquer uma das três falhas propostas.

O peso de um arco também pode agregar restrições. A subseção a seguir discute o caso em que um ou mais arcos da rede recebam um valor de peso fixo.

3.3.5 Pesos com valores fixos

Pode-se requerer que o peso de um ou mais arcos tenham valor fixo. Não há motivos práticos que estabeleçam esta restrição, mas seu uso pode ser útil para comparar soluções já existentes da instância teste em consideração. Nesse caso, deve-se ler como dado de entrada um vetor contendo um conjunto de arcos e, para cada um desses arcos, o valor de peso fixo. Ainda, o parâmetro $fixW$ deve ser informado com valor 1, caso deseja-se considerar o conjunto de pesos fixos, ou 0, caso contrário.

A próxima seção apresenta um algoritmo genético utilizado para otimização das multiplicidades dos arcos de uma rede. Todas as restrições e modificações do problema discutidas anteriormente serão opcionalmente consideradas no problema.

3.4 Solução do problema usando algoritmos genéticos

A estrutura deste algoritmo genético é semelhante a utilizada no algoritmo memético descrito no Capítulo 2. As diferenças encontram-se listadas a seguir:

- uma solução (cromossomo) deste problema é representada por dois vetores, m e w, enquanto que para o problema WSP a solução é representada apenas pelo vetor w. O *crossover* considera apenas o vetor w. As multiplicidades (vetor m) correspondentes ao vetor w são atribuídas durante sua avaliação.

- avaliação da solução;

- o problema de atribuição das multiplicidades envolve a leitura de uma maior variedade de dados de entrada que o problema anterior;

- um procedimento de busca local não foi utilizado, visto que os procedimentos testados não contribuiram para o desempenho do algoritmo.

Aparentemente, as diferenças parecem poucas, mas a avaliação da solução deste problema é complexa e responsável por cerca de 99% do tempo computacional total do GA. Nas seções a seguir, a função de avaliação de uma solução, bem como funções auxiliares desta, serão abordadas uma a uma. Inicia-se esse estudo apresentando o pseudo-código da função de avaliação.

3.4.1 Avaliação de uma solução do problema

Denomina-se por *avaliação de uma solução*, o procedimento de determinar as multiplicidades correspondentes a um vetor de pesos informado, assim como computar sua soma total. Durante a avaliação, as multiplicidades e cargas são ajustadas, considerando as cargas relativas a uma rede sem falhas, e a uma rede com cada uma das três falhas aqui consideradas. Inicialmente as multiplicidades têm valor um, para todos os arcos, e posteriormente seus valores são incrementados (nunca decrementados) até que um vetor de multiplicidades m dimensione a rede adequadamente, sem que haja sobrecarga de arcos.

```
procedure evaluateSol(w, MU, MUf, lf, rf, sp, ϒ, γ)
1     forall a ∈ A do m_a ← 1;
2     sol ← fillSolMemory(w, m);
3     anychange ← 1;
4     while anychange = 1 do
5           setL ← updateMulti(sol, m, γ, MU);
6           if |setL| > 0 then updateSol(w, sol, m, MU, setL);
7           anychange ← 0;
8           if sf = 1 solveFail (sol, m, w, sFS, |sFS|, anychange, MUf);
9           if rf = 1 solveFail (sol, m, w, rFS, |rFS|, anychange, MUf);
10          if lf = 1 solveFail (sol, m, w, E, |E|, anychange, MUf);
11    end while
12    forall a ∈ A do
13          if maxL_a = 0 then
14                if ϒ and maxL_γa = 0 do m_a ← m_γa ← 0;
15                else m_a ← 0;
16          end if
17    end forall
18    fit ← Σ_{a∈A} fit_a;
19    return(fit, m);
end evaluateSol.
```

Figura 3.6: Avaliação de uma solução do problema considerando um vetor de pesos w e uma utilização máxima MU no caso de rede em operação normal e MUf em caso de falha na rede.

A Figura 3.6 apresenta o pseudo-código usado para avaliação de uma solução deste problema. A função `evaluateSol()` recebe como parâmetros: o vetor de pesos w, e os parâmetros MU, MUf, lf, rf, sf e Υ.

Inicialmente a multiplicidade de cada arco é inicializada com valor *1* (linha 2), para posteriormente ser incrementada, caso necessário. A seguir, as estruturas de dados utilizadas durante a avaliação de uma solução são preenchidas considerando os vetores de peso w e de multiplicidade m (linha 1). Estas estruturas são representadas por *sol* e são as mesmas memórias usadas para o problema WSP, ou seja:

- grafo de caminhos mínimos (g^{SP}) com t como raiz;
- vetor de distâncias (d^t) para cada destino t;
- vetor ρ^t associado a cada destino t;
- vetor de cargas (l^t) com destino em t;
- vetor de carga total (l).

No problema descrito no Capítulo 2, o vetor δ é usado para armazenar em cada posição δ_i o número de arcos saintes do nó i e presentes no grafo de caminhos mínimos com destino em t. Em vez de δ, nesse problema utiliza-se o vetor ρ, que informa a soma total de multiplicidades dos arcos saintes do nó i e presentes no grafo de caminhos mínimos com destino em t. As linhas 3-11 calcula os valores das multiplicidades dos arcos, respeitando as restrições informadas como parâmetros. As linhas 8, 9 e 10 somente têm efeito se alguma falha for considerada, isto é, se pelo menos um dos parâmetros *lf*, *rf* ou *sf* possuir o valor 1. Neste caso, a carga l_a de cada arco a não poderá ultrapassar o valor de $MUf * c_a$.

A seguir, a variável *anychange* é inicializada com valor *1* (linha 4). Esta variável tem valor zero sempre depois que a restrição de utilização máxima em caso de rede sem falhas *MU* for satisfeita (linha 7) e seu valor muda para *1* se, em pelo uma das falhas consideradas, causar o incremento da multiplicidade de pelo menos um arco.

A função `updateMulti()` incrementa, se necessário, as multiplicidades dos arcos, de tal forma a satisfazer a restrição de máxima utilização dos arcos (informada como parâmetro). Na linha 5 as multiplicidades são adaptadas (incrementado-as) ao valor de máxima utilização para o caso de rede sem falha (*MU*). O conjunto *setL* conterá todos os nós origem dos arcos que tiveram suas multiplicidades aumentadas em qualquer valor. Se a multiplicidade de pelo menos um arco mudar

($|setL| > 0$), o roteamento consequentemente pode mudar. Nesse caso, a solução *sol* é atualizada pela função `updateSol()` (linha 6). Esta função ajusta os valores de carga dos arcos, segundo suas novas multiplicidades, de forma que a restrição de máxima utilização seja satisfeita.

O laço das linhas 4-11 é executado, até que não haja mais incremento nas multiplicidades, causado pela falha de um *span*, roteador ou arco. Estas falhas são tratadas respectivamente nas linhas 8-10. A ordem em que as falhas são tratadas não altera significantemente o valor da solução e o tempo gasto para avaliá-la. Alguns testes foram realizados para verificar o quanto esta ordem afeta o tempo e a solução, sendo que a ordem adotada ($sf \rightarrow rf \rightarrow lf$) teve, em média, um melhor desempenho. O único parâmetro que muda na chamada da função `solveFail()` é o conjunto de arcos afetados por cada falha, bem como sua dimensão. Estes conjuntos são, respectivamente, *sFS* (*span fail set*), *rFS* (*router fail set*) e o conjunto de arcos da rede (E). Os conjuntos *sFS* e *rFS* são, na realidade, matrizes, as quais possuem um vetor de arcos afetados no caso da falha de cada *span* e roteador, respectivamente.

Nas linhas 12-17 tem-se o cuidado de zerar as multiplicidades dos arcos que não foram utilizados. Por simplicidade, o vetor *maxL* não é refenciado em nenhum pseudo-código. Este vetor possui, para cada arco da rede, o maior valor de carga que este arco teve durante a avaliação da solução. No início de cada avaliação de solução, o valor de $maxL_a$ é inicializado com *0* para cada arco $a \in E$.

Para determinar o valor de cada solução (linha 18) somam-se os valores de função objetivo de cada arco fit_a. Este valor depende de qual função objetivo está-se usando. Os testes computacionais apresentados ao final do capítulo apresentam resultados com as três funções objetivo mencionadas na subseção 3.3.1. Como resultado da avaliação do vetor *w* proposto, são retornados seu valor de função objetivo e o vetor de multiplicidades (linha 19).

A função `updateMulti()`, que foi usada na linha 5 do pseudo-código da Figura 3.6, é descrita com mais detalhes na Figura 3.7. O parâmetro U representa o valor de utilização máxima que deve ser considerado para o cálculo das multiplicidades. Este pode assumir os valores de *MU* ou *MUf*, dependendo de onde esta função for chamada.

Inicialmente, a multiplicidade mínima *mt* para satisfazer a restrição U é calculada para cada arco a da rede (linha 2). Caso $mt_a > m_a$, o valor de m_a deve ser atualizado. Consequentemente, a

```
function updateMulti(sol, m, γ, U)
1     forall a = (u→, v→) ∈ E do
2           mt ← ⌈ la_a / (U * c_a) ⌉;
3           if mt > m_a then
4                 updateRho(sol, mt − m_a, a);
5                 m_a ← mt;
6                 insertInto(setL, u);
7           end if
8           if γ_a > 0 then
4                 updateRho(sol, mt − m_γa, γ_a);
5                 m_γa ← mt;
6                 insertInto(setL, u);
12          end if
13    end forall
14    return(setL);
end updateMulti.
```

Figura 3.7: Pseudo-código da função executada para incrementar os valores do vetor m de forma a satisfazer a restrição U e mantendo multiplicidades iguais para arcos simétricos.

memória ρ da solução *sol* é atualizada (linha 4), o novo valor de m_a passa a ser mt (linha 5) e o nó u é inserido no conjunto $setL$ pela função `insertInto()` (linha 6). Lembrando que o conjunto $setL$ contém todos os nós origem dos arcos que tiveram suas multiplicidades incrementadas. A função `insertInto()` (linha 6) apenas insere um nó u no caso em que $u \notin setL$.

Para incluir a restrição de multiplicidades iguais para arcos simétricos (Υ), uma estrutura de dados auxiliar, dada pelo vetor γ, é usada. Esta estrutura é criada logo após a leitura dos dados da instância e permanece inalterada durante toda execução do GA. O vetor γ indica, para cada arco a, ou o índice do arco simétrico correspondente, caso $\Upsilon = 1$, ou o valor -1 caso o arco a não tenha um arco simétrico.

O laço das linhas 8-12 executa um procedimento similar ao laço das linhas 3-7. O seu objetivo é manter as multiplicidades iguais para arcos simétricos, em caso em que esta restrição tenha sido requerida. A diferença está em considerar o arco simétrico γ_a em vez do arco a. Na linha 14 a função retorna o conjunto $setL$ que é usado pela função `updateSol()`.

A seguir é discutido o caso em que é feito o ajuste entre os valores das multiplicidades e da carga, de forma que a restrição de utilização máxima seja respeitada.

Atualização de uma solução do problema

Esta função ajusta os valores de carga dos arcos, segundo suas multiplicidades, até que a restrição de máxima utilização seja satisfeita. A função `updateSol()` referenciada inicialmente na linha 6 do pseudo-código da Figura 3.6 é descrita com mais detalhes na Figura 3.8. Esta função tem

```
procedure updateSol(w, sol, multi, U, setL)
1     do
2           forall t ∈ T do
3                 H ← ∅;
4                 forall u ∈ setL do InsertIntoHeap(H, u, d_u);
5                 updateLoad(multi, H, sol, t, g_t^SP);
6           end forall
7           setL ← updateMulti(multi, sol, γ, U);
8     while |setL| > 0;
end updateSol.
```

Figura 3.8: Pseudo-código para atualizar solução, caso o valor de alguma multiplicidade tenha mudado.

por objetivo adequar a carga *vs.* multiplicidade dos arcos. A Figura 3.5, apresentada anteriormente, ajuda a compreender este ajuste. Cada vez que uma multiplicidade é incrementada, a carga dos arcos possivelmente também muda. A alteração da carga pode causar o incremento de alguma multiplicidade. Mas como as multiplicidades vão sendo sempre incrementadas, chega-se a um valor em que a modificação da carga dos arcos não causa o incremento de nenhuma multiplicidade. A função é envolta num laço **do-while** principal que é executado enquanto o conjunto $setL$ não estiver vazio (linha 8). Este conjunto estará vazio quando a carga de cada arco encontra-se ajustada ao seu valor de multiplicidade e máxima utilização, ou seja, a carga dos arcos está respeitando a utilização máxima U estipulada.

Os grafos de caminhos mínimos são atualizados no laço **forall** executado nas linhas 2-5. Para cada nó destino $t \in T$ (lembrado que T é o conjunto de nós destinos), todos os nós contidos no conjunto $setL$ são inseridos na pilha H (linha 3) utilizada pela função `updateLoad()`. A carga é então atualizada na linha 4. A seguir, faz-se o ajuste das multiplicidades dos arcos (linha 6).

A Figura 3.9 apresenta o pseudo-código da função que atualiza a carga dos arcos da rede após

```
procedure updateLoad(m, sol, H, d, t, g_t^SP)
1     while HeapSize(H) > 0 do
2         u ← FindAndDeleteMax(H);
3         if d_u < ∞ do
4             ld ← D_ut + Σ_{a=(v,u)∈g_t^SP} lat_a^t;
5             forall a = (u,v) ∈ OUT(u) do
6                 if a ∉ g_t^SP then load ← 0
7                 else load ← m_a * ld/ρ_u;
8                 if lat_a^t ≠ load do
9                     la_a ← la_a + load − lat_a^t;
10                    lat_a^t ← load;
11                    InsertIntoHeapMax(H, v, d_v);
12                end if
13            end forall
14        end if
15    end while
end updateLoad.
```

Figura 3.9: Pseudo-código do procedimento `UpdateLoad` que atualiza o valor das cargas dos arcos.

uma mudança de multiplicidades. Esta função é inicialmente referenciada na linha 4 da Figura 3.8. Os nós u são removidos um a um (linha 2) da pilha H e somente se $d_u < \infty$ a carga de seus arcos adjacentes será atualizada. Esta restrição é motivada pelo fato de que cada nó possui uma distância ao destino, e essa distância é ∞ no caso em que o nó estiver desativado, ou seja, não haverá carga passante por ele (roteador com falha). A seguir, na linha 4, o total de carga sainte do nó u é computada como sendo a soma da carga originada neste nó com destino em t (D_{ut}) mais a soma das cargas incidentes em u. Esta carga é dividida proporcionalmente pelo número de multiplicidades partindo dele, ou seja $l = \frac{load}{\rho_u}$. A porção de carga destinada a cada arco $a \in g^{SP}$, partindo do nó u, recebe o valor de $m_a * l$ (linha 7). Caso a carga l com destino a t seja diferente da atual, ela é devidamente atualizada (linha 9), assim como a carga passante pelo arco a com destino a t é atualizada pelo seu novo valor (linha 10). O nó destino de cada arco que teve sua carga atualizada é inserido na pilha H (linha 11).

A seguir discute-se os valores de utilização máxima dos arcos de uma rede operando normalmente, ou em caso de falha.

Garantindo uma utilização máxima de cada arco no caso de falha na rede

No caso em que *lf*=1 ou *rf*=1 ou *sf*=1, a função `solveFail()`, descrita no pseudo-código da Figura 3.10, é executada. Sua chamada aparece nas linhas 8, 9 e 10 da Figura 3.6. A função recebe como parâmetros a solução *sol*, o vetor de multiplicidades *m*, o vetor de pesos *w*, o conjunto de arcos afetados pela falha de cada elemento do conjunto $failS$, a dimensão *x* do conjunto $failS$, a variável *anychange* e a máxima utilização em caso de falha *MUf*.

Considera-se que uma falha foi devidamente tratada (atende à restrição de máxima utilização *MUf*) quando a simulação de falha de *x* itens subsequentes do conjunto $failS$ não resultar na mudança de nenhuma multiplicidade.

A variável $fail$ indica o elemento cuja falha está sendo considerada, enquanto que *cont* verifica quantas simulações de falhas de elementos do conjunto $failS$ foram feitas sem necessidade de incremento de nenhuma multiplicidade. Inicialmente $fail$ e *cont* recebem o valor 1 (linha 1). Para cada falha testada, a solução atual *sol* é copiada para uma solução auxiliar *solAux* (linha 3). A seguir, a falha $fail$ é incrementada (linha 4). A linha 5 garante a circularidade das falhas testadas, ou seja, quando testou-se a última falha do conjunto *failS*, volta-se à primeira. O laço das linhas 7-10 atribui pesos infinitos para todos os arcos afetados pela falha (linha 11). A função `updateGraph()` atualiza o grafo de caminhos mínimos, desativando todos os arcos no conjunto $failS$. Na linha 12 os pesos originais são restabelecidos para o conjunto de arcos anteriormente desativados. Na linha 13, as multiplicidades são incrementadas caso necessário. Se pelo menos o valor de uma multiplicidade mudar, o laço nas linhas 14-19 deve ser executado. Nesse laço o incremento de alguma multiplicidade é indicado (linha 15), os valores de ρ são recalculados (linha 16) e a solução *sol* é atualizada (linha 17). O procedimento pára quando a simulação de falha de *x* elementos consecutivos do conjunto $failS$ do conjunto *failS* não resultar no incremento de nenhuma multiplicidade (linha 21).

O pseudo-código da Figura 3.10 recebe como parâmetro um conjunto *failS* que, como já mencionado, é diferenciado dependendo da falha considerada. Este conjunto é na realidade uma matriz estática, pois pode ser definido no início do programa. No caso de falha de arcos, o conjunto *failS* conterá $|E|$ vetores, cada vetor $failS_i$ composto por um único arco *a*. No caso de falha de *spans*, o conjunto *failS* conterá *z* vetores, onde *z* indica o número de *spans* a serem considerados. Neste

```
procedure solveFail(sol, m, w, failS, x, anychange, MUf)
1     count ← fail ← 1;
2     while 1 do
3           copyMemory(sol, solAux);
4           fail ← fail + 1;
5           if fail = n + 1 then fail ← 1;
7           forall a ∈ failS[fail] do
8                 waux_a ← w_a;
9                 w_a ← INFINITY;
10          end forall
11          updateGraph(solAux, fail, w, m, failS);
12          forall a ∈ failSet do w_a ← waux_a;
13          setL ← updateMulti(m, solAux, γ, MUf);
14          if |setL| > 0 then
15                anychange ← 1;
16                forall g^SP ∈ G^SP do computeRho();
17                updateSol(w, sol, m, MUf, setL);
18                count ← 0
19          end if
20          count ← count + 1;
21          if count = x then break;
22    end while
end solveFail.
```

Figura 3.10: Procedimento adotado em caso de falha.

caso, cada vetor $failS_i$ conterá k arcos, sendo k o número de arcos do *span* i em questão. No caso de falha de roteadores, o conjunto *failS* deve ser computado no início do programa. Sabe-se que o conjunto conterá $|V|$ vetores, onde cada vetor $failS_i$ terá tamanho h, sendo h o número de arcos desativados devido a falha do roteador i. Estes arcos são identificados pela função descrita no pseudo-código descrito na Figura 3.11.

O procedimento `linksFailed()` possui um laço principal, das linhas 1-17, que considera a falha de um roteador de cada vez. Um conjunto *setR* vai conter o roteador que falhou, assim como todos os roteadores dependentes deste, ou seja, roteadores que possuem todos arcos saintes conectando roteadores falhos (do conjunto *setR*). Inicialmente, todos os arcos partindo de v, o nó falho em questão, são inseridos no conjunto $failS$ (linha 3). A seguir os arcos $a = (\overrightarrow{s,u})$ chegando em cada nó u falho são inseridos no conjunto $failS$ (linha 7) e analisados um a um. Caso todos os

```
procedure linksFailed()
1     forall v ∈ V do
2           rt ← v;
3           forall a ∈ OUT(v) do failSet_v ← failSet_v ∪ {a};
4           setR ← {v};
5           forall u ∈ setR do
6                 forall a = (s,u) ∈ IN(u) do
7                       failSet ← failSet ∪ {a};
8                       forall e = (s,z) ∈ OUT(s) do
9                             flag ← 1
10                            forall u ∉ setR do flag ← 0;
11                            if flag = 1 then
12                                  setR ← setR ∪ {s};
13                                  break;
14                            end if
15                      end forall
16                end forall
17          end forall
18    end forall
end linksFailed.
```

Figura 3.11: Procedimento executado para identificar os arcos desativados devido à falha de cada roteador.

arcos $e = (\overrightarrow{s,z})$ partindo do nó s estiverem conectando nós falhos (do conjunto *setR*), então o nó z é inserido no conjunto *setR* (linha 13).

Em qualquer um dos casos, uma falha pode acarretar na desativação de parte da rede. Isso acontece quando o grafo torna-se desconexo com a remoção de um arco, ou um conjunto de arcos. Neste caso, as cargas chegantes e saíntes dos roteadores desativados são desconsideradas.

Nesta seção, foram apresentados todos os procedimentos usados para cálculo e avaliação de uma solução. Uma solução deve ser avaliada sempre que é gerada. Soluções são geradas em duas situações: ao se iniciar a população e quando uma solução nova é gerada por ocasião da execução do operador de *crossover*. A cada nova solução, além de sua avaliação, pode-se aplicar um procedimento de busca local com o objetivo de melhorá-la. A próxima seção introduz os dois procedimentos de busca local implementados.

3.4.2 Buscas locais implementadas

Definir uma busca local para este problema é uma tarefa bastante complexa. Qualquer mudança de peso ou multiplicidade de um ou mais arcos acarreta uma mudança no fluxo de carga da rede, podendo esta solução melhorar ou piorar sem muita previsão. Ou seja, o problema é descobrir uma modificação na solução que tenha alta probabilidade de contribuir para sua melhoria.

Dois procedimentos de busca local foram implementados, um modificando multiplicidades, e o outro modificando pesos. Nenhum dos dois foi usado na versão final do GA, visto que seu uso não colabora para melhora da solução. Os procedimentos são:

- *LSw*: incremento unitário do peso dos arcos com maior utilização.
- *LSm*: decremento unitário da multiplicidade de um conjunto de arcos. Estes arcos são aqueles que têm o seu aumento, causado pelo arredondamento para o maior inteiro, mais próximo do valor *1*. O objetivo é que a soma do aumento de multiplicidades causado por esta modificação não seja maior que a soma dos valores de multiplicidades subtraídas.

Quando a multiplicidade de um arco é definida, seu valor deve ser um valor inteiro maior ou igual ao obtido pela razão entre a carga e a capacidade livre (depende da máxima utilização estipulada), como foi visto na Figura 3.7. A multiplicidade de um arco pode ser calculada como sendo *3,05*, por exemplo, mas como deve ser considerado o maior inteiro, seu valor passa a ser *4*. A busca local *LSm* reduz em um o valor da multiplicidade dos arcos que tiveram a maior diferença entre os valores *float* e *int* de sua multiplicidade. Vários tamanhos do conjunto de arcos com multiplicidades modificadas foram testados, sem muita diferença entre eles. Apenas verificou-se que para tamanhos muito pequenos (próximos de 1) e muito grandes (maiores que 10) raríssimas vezes resultaram em melhora. Nos testes computacionais apresentados no final deste capítulo, 5 arcos têm simultaneamente sua multiplicidade decrescida. A solução gerada pela modificação de uma multiplicidade fica um pouco imprevisível, pois resulta numa mudança das cargas dos arcos, que em geral faz aumentar a multiplicidade de outros arcos.

A segunda restrição, *LSw*, é muito similar à busca local usada no problema do Capítulo 2. Uma adaptação foi feita para agregar as restrições e as modificações do problema aqui propostas. Além disso, a mudança de um peso neste problema acarreta modificações maiores. A mudança de um

peso geralmente acarreta a mudança de uma ou mais multiplicidades, o que pode modificar em muito o roteamento, ou seja, gerando uma solução totalmente imprevisível.

A próxima seção apresenta resultados computacionais realizados com o problema, incluindo opcionalmente todas as restrições, modificações e especializações do problema propostas anteriormente.

3.5 Resultados computacionais

Nesta seção, são descritos os experimentos computacionais realizados com o algoritmo genético aplicado ao problema de designação de multiplicidades aos arcos. O ambiente computacional é primeiramente descrito, detalhes sobre parâmetros usados e instâncias testadas são a seguir fornecidos, finalizando com a apresentação dos resultados e sua análise. Todos os resultados computacionais apresentados consistem em uma média de três execuções usando sementes diferentes.

Não foram apresentados resultados utilizando os procedimentos de busca local implementados, pois estes não colaboraram para a melhoria do algoritmo. O pouco melhoramento da solução final não é compensado pelo tempo computacional adicional que o uso destes procedimentos acarreta.

Como mencionado anteriormente, não encontrou-se na literatura uma outra proposta de método aplicado a este problema. Por este motivo, os resultados apresentados não puderam ser comparados com resultados de outros métodos.

3.5.1 O ambiente computacional

Os experimentos foram realizados em um Pentium IV 1.7 GHz com 256 Mb de memória RAM. Os algoritmos foram implementados em C e compilados com o `cc`, versão 3.2, usando o flag `-O3`. O sistema operacional é o linux Red Hat, versão 8.0 3.2-7. Os tempos foram medidos com a função `getrusage`. Os números aleatórios gerados pelo algoritmo genético usaram o gerador *Mersenne Twister* [MN98] de Matsumoto e Nishimura.

Os parâmetros a seguir foram usados para o algoritmo genético:

- Particionamento da população: 25% das soluções no conjunto $\mathcal{A}$, 5% das soluções no conjunto $\mathcal{C}$, e 70% no conjunto $\mathcal{B}$;

- Probabilidade de ocorrer mutação: $p_m = 1\%$;
- Probabilidade que um peso seja herdado do pai elite pela solução gerada no *crossover*: $p_{\mathcal{A}} = 70\%$;
- utilização máxima de cada arco no caso da rede operando sem falha (MU) é igual a $0,80$;
- O número de gerações, o tamanho da população e o intervalo dos pesos dos arcos varia de acordo com o tipo do experimento.

Os experimentos foram realizados considerando 7 instâncias de quatro classes propostas por Fortz e Thorup [FT00] usadas no Capítulo 2. Detalhes sobre as instâncias encontram-se sintetizados na Tabela 2.1.

Para cada uma dessas instâncias, 12 matrizes de demanda distintas $D^1, D^2, \ldots, D^{12}$ são geradas. A matriz de demanda inicial é a D^1, as outras são geradas por se multiplicar D^1 repetidamente por um fator de escala: $D^k = \rho^{k-1} D^1, \forall\, k = 1, \ldots, 12$.

Uma nova instância é usada para os testes deste capítulo. Trata-se da instância `attAS` com dados atualizados (julho/2003) do *backbone* do sistema autônomo da AT&T nos Estados Unidos. A matriz de demanda para esta instância, assim como para a instância `att`, é obtida segundo a metodologia proposta por Feldmann et al. [FGL+01]. Esta instância é composta por:

- uma rede de 54 nós e 278 arcos;
- 306 pares de demanda;
- 40 *spans* com um conjunto variando entre 8 e 38 arcos cada um.

Os arcos dessa instância possuem capacidades iguais, enquanto que as outras instâncias possuem capacidades diferenciadas para os arcos. A Figura 3.12 apresenta um mapa simplificado do *Backbone* principal do AS IP *Backbone* da AT&T nos EUA. Um mapa mais completo encontra-se disponível em http://www.geog.ucl.ac.uk/casa/martin/atlas/more_isp_maps.html. Grande parte da estrutura do mapa da Figura 3.12 é compartilhada com outras corporações, pois este representa uma ligação entre os grandes centros dos EUA.

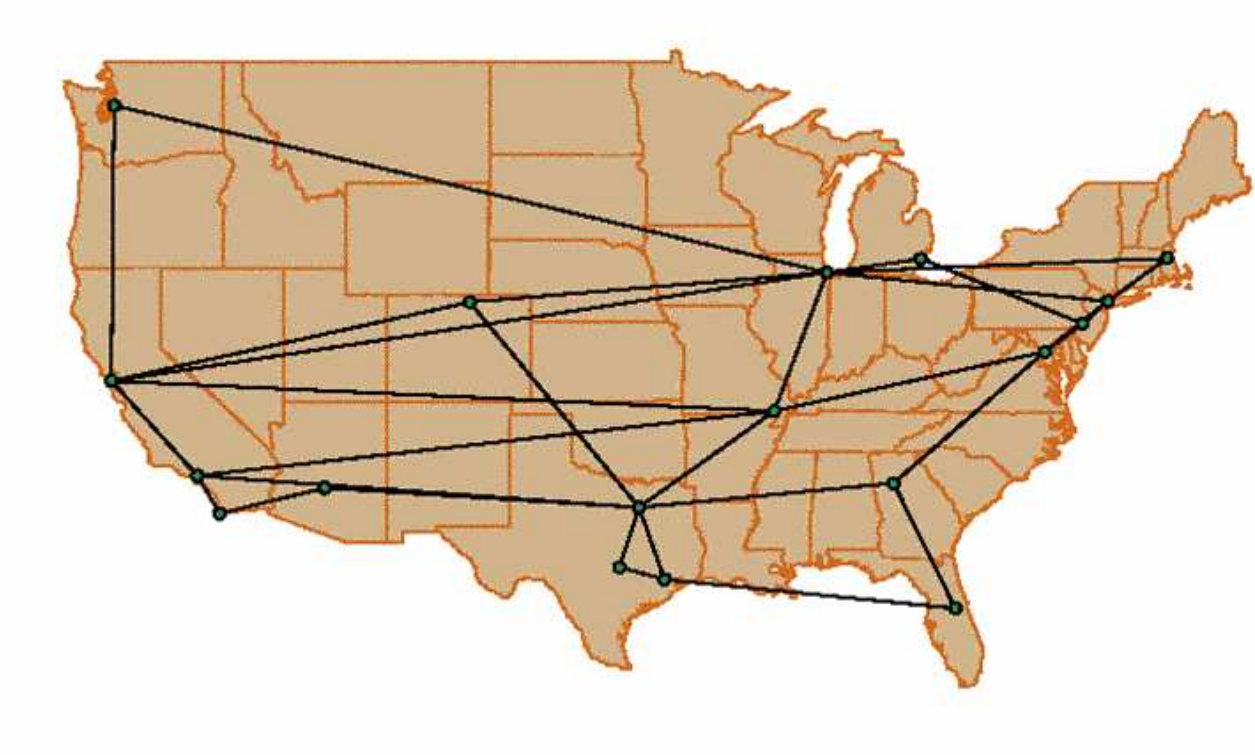

Figura 3.12: *Backbone* ligando os principais roteadores nos EUA.

Os resultados atuais usados pela AT&T para os dados da instância `attAS` não foram comparados com os obtidos aplicando-se o GA a ela, pois a solução atual da AT&T leva em consideração outros dados que não foram aqui considerados, como o atraso (tempo de percurso da mensagem no arco) de cada nó.

As próximas subseções apresentam os resultados computacionais dos testes realizados.

3.5.2 Resultados comparativos do GA com e sem os procedimentos de busca local implementados

Os resultados computacionais apresentados nesta subseção têm o objetivo de avaliar o desempenho do GA com e sem os procedimenos de busca local implementados (*LSm* e *LSw*). Os resultados foram gerados pelo GA considerando a instância `attAS` usando diversas combinações de parâmetros. A Tabela 3.2 apresenta os resultados dessa comparação. Os parâmetros fixos para os testes desta tabela são lf=1, rf=1 e sf=1. Ou seja, todos os três tipos de falhas são considerados simultaneamente. O critério de parada do algoritmo neste experimento computacional foi a execução do

GA por 100 segundos.

Esta tabela é basicamente composta por duas colunas principais, sendo a primeira relativa à comparação dos resultados sem a restrição de multiplicidades simétricas ($\Upsilon = 0$) enquanto a segunda considera esta restrição ($\Upsilon = 1$). Cada uma dessas colunas é subdividida em outras duas, cada uma correspondendo aos valores de MUf igual a $0,85$ e $1,00$, para a utilização de um arco em caso de rede com falha. Para cada um dos quatro casos, a saber $\Upsilon = 1$ e $MUf = 0,85$; $\Upsilon = 1$ e $MUf = 1,00$; $\Upsilon = 0$ e $MUf = 0,85$; $\Upsilon = 0$ e $MUf = 1,00$, são fornecidas três colunas de dados. Os dados correspondem à comparação dos resultados do GA sem busca local (*sLS*), do GA com a busca local *LSm* e do GA com a busca local *LSw*, ambos aplicados apenas às soluções resultantes do *c*rossover. Para cada caso, o menor valor é considerado base, apresentando o valor $0,00$ na tabela, e os outros dois apresentam o desvio percentual em relação ao caso base.

As linhas apresentam os resultados obtidos pela combinação dos valores listados para a função objetivo Obj, tamanho de população Pop e peso máximo W de um arco. Por exemplo, para os parâmetros $Obj = 1$, $Pop = 25$, $W = 10$, $\Upsilon = 0$ e $MUf = 0,85$, os resultados obtidos para *LSm* e *LSw* são $1,28\%$ maior que o obtido por *sLS*. Por isso, estes valores ($0,00$, $1,28$ e $1,28$) aparecem no início da primeira linha em suas posições correspondentes às colunas *sLS*, *LSm* e *LSw*. O objetivo $Obj_2 = \sum_{a \in E} m_a * c_a$ A última linha apresenta a média dos desvios percentuais.

Como pode-se observar, o uso de busca local raramente colaborou para a melhoria das soluções. Para as $72 (= 18 * 4)$ combinações de parâmetros testados, em apenas 3 casos os melhores resultados foram obtidos pelo GA com busca local.

Estes resultados foram obtidos para execuções com tempo fixo em 100 segundos cada uma. Resultados semelhantes foram obtidos com execuções de 1000 segundos para algumas das combinações de parâmetros usados para gerar a Tabela 3.2. Dessa forma, pode-se concluir que os procedimentos de busca local implementados e testados não contribuem para a melhoria das soluções da população.

Uma explicação para tal comportamento é que geralmente a modificação inicial da busca (incremento do peso, no caso de *LSw*, e decremento da multiplicidade, no caso de *LSm*) acarreta modificações maiores na solução, resultando numa solução de pior qualidade. Nesse caso a solução original é retornada como resultado da busca local. Em consequência, gasta-se um tempo

Tabela 3.2: Comparação dos resultados do GA em relação ao uso ou não de um dos dois procedimentos de busca local implementados (*LSw* e *LSm*). O tempo de 100 segundos foi o critério de parada utilizado, usando a instância `attAS` para o caso em que lf=1, rf=1 e sf=1.

			$\Upsilon = 0$						$\Upsilon = 1$					
			$MUf = 0,85$			$MUf = 1,00$			$MUf = 0,85$			$MUf = 1,00$		
Obj	*Pop*	*W*	*sLS*	*LSm*	*LSw*	*sLS*	*LSm*	*LSw*	*sLS*	*LSm*	*LSw*	*sLS*	*LSm*	*LSw*
1	25	10	0,00	1,28	1,28	0,00	1,20	1,20	0,00	1,93	1,78	0,00	2,88	3,65
		20	0,00	0,91	0,91	0,00	1,58	2,12	0,00	5,48	2,81	0,00	0,73	2,28
		40	0,00	1,49	1,49	0,00	1,88	1,88	0,00	3,15	4,18	0,00	0,93	3,22
1	50	10	0,00	0,93	0,93	0,00	0,48	0,48	0,07	0,00	1,46	0,00	1,27	1,66
		20	0,00	2,57	2,57	0,00	2,21	2,21	0,00	4,98	1,18	0,00	3,04	2,34
		40	0,00	0,46	1,46	0,00	0,85	0,85	0,00	4,84	1,51	0,00	3,63	1,34
1	100	10	0,00	0,00	0,00	0,00	0,28	0,28	0,00	2,32	1,30	0,00	1,31	0,69
		20	0,00	0,70	0,70	0,00	0,28	0,28	0,00	2,47	1,94	0,00	1,35	1,87
		40	0,00	0,49	0,49	0,00	0,60	0,60	0,00	2,26	0,79	0,00	2,42	1,32
3	25	10	0,00	7,78	7,78	0,00	1,82	1,84	0,00	7,04	5,15	0,00	8,33	4,20
		20	0,00	3,74	3,74	0,00	2,63	2,69	0,00	6,30	4,43	0,00	6,52	3,93
		40	0,00	2,83	2,83	0,00	3,12	3,12	0,00	8,60	7,58	0,00	3,83	6,54
3	50	10	0,00	1,05	1,05	0,00	1,66	1,66	0,00	4,83	6,09	0,00	4,44	1,61
		20	0,00	3,36	3,36	0,00	1,75	1,75	0,00	3,75	5,55	0,00	3,81	3,38
		40	0,00	2,81	2,81	0,00	3,38	3,38	0,00	4,41	4,18	0,00	4,76	1,35
3	100	10	0,00	0,37	0,37	0,00	0,64	1,56	0,00	5,31	1,38	0,00	0,94	2,25
		20	0,00	0,45	0,78	0,00	1,43	1,43	0,39	0,00	1,48	2,93	0,00	4,24
		40	0,00	1,60	1,60	0,00	0,85	0,85	0,00	5,27	5,65	0,00	4,13	3,06
Média			0,00	1,82	1,90	0,00	1,48	1,57	0,03	4,05	3,25	0,16	2,98	2,72

considerável executando o procedimento de busca local, sem contribuição para a melhoria da solução. Por este motivo, numa mesma quantidade de tempo, o GA sem busca local roda por um número maior de gerações, retornando uma solução final de melhor qualidade.

A próxima subseção apresenta uma comparação de resultados, objetivando escolher adequadamente o tamanho de população e o intervalo dos pesos dentre as possibilidades testadas.

3.5.3 Escolha do tamanho de população e do intervalo dos pesos

Os resultados usados para a comparação nesta subseção são os mesmos usados para preencher os dados da tabela anterior. Entretanto, nesta comparação foram excluídos os dados do GA com busca local, visto que seu uso não contribuiu para o desempenho do GA.

A primeira comparação, apresentada nas Tabelas 3.3 e 3.4, objetiva determinar um tamanho de população adequado para o GA implementado. A Tabela 3.3 apresenta a comparação sem considerar a restrição $\Upsilon = 0$, enquanto que a Tabela 3.4 apresenta a comparação com $\Upsilon = 1$. Estas duas tabelas encontram-se particionadas em quatro colunas principais, sendo que cada coluna representa uma das combinações a seguir: $\Upsilon = 0$ e $MUf = 0,85$, $\Upsilon = 0$ e $MUf = 1,00$, $\Upsilon = 1$ e $MUf = 0,85$ e $\Upsilon = 1$ e $MUf = 1,00$. Cada uma dessas colunas é subdividida em três outras, apresentando uma comparação dos resultados obtidos por populações com 25, 50 ou 100 indivíduos. Usando a mesma forma de comparação usada na tabela anterior, para cada combinação de parâmetros, o menor valor encontrado pelos três tamanhos de populações usados aparece com valor $0,00$, enquanto os outros dois apresentam o desvio percentual do menor valor. As linhas apresentam a combinação dos parâmetros Obj, com valores 1 ou 3, e do parâmetro W, com valores 10, 20 e 40. A última linha apresenta a média do desvio percentual de cada coluna.

Um comportamento similar pode ser observado nas duas tabelas: os resultados pioram com o aumento do tamanho da população. Esta piora é mais acentuada na segunda tabela, a qual considera multiplicidades iguais para arcos simétrico ($\Upsilon = 1$). Desta forma, observando este comportamento, sintetizado na última linha de cada coluna, conclui-se que para estes experimentos a população com apenas 25 indivíduos se mostra mais adequada que populações de tamanho 50 e 100.

A Tabela 3.5 apresenta resultados comparativos com o intuito de selecionar um valor adequado para W. Novamente, os resultados foram extraídos da mesma fonte das duas tabelas anteriores. Para

Tabela 3.3: Comparação dos resultados do GA sem a restrição de multiplicidades iguais para arcos simétricos ($\Upsilon = 0$) e usando tamanhos de população iguais a 25, 50 e 100. A tabela apresenta uma comparação dos resultados do GA com 100 segundos como critério de parada, usando a instância `attAS`, *sLS*, $\Upsilon = 0$, *lf*=1, *rf*=1 e *sf*=1.

		$MUf = 0,85$			$MUf = 1,00$		
Obj	*W*	*Pop*25	*Pop*50	*Pop*100	*Pop*25	*Pop*50	*Pop*100
1	10	0,00	3,89	5,74	0,00	4,09	5,99
	20	0,00	1,52	5,61	0,00	4,78	6,06
	40	0,00	5,35	8,16	0,00	4,89	7,46
3	10	0,00	6,51	11,49	0,00	4,70	9,43
	20	0,00	8,84	14,56	0,00	4,06	10,33
	40	0,00	2,87	11,31	0,00	6,94	15,26
Média		0,00	4,83	9,48	0,00	4,91	9,09

Tabela 3.4: Comparação dos resultados do GA considerando a restrição de multiplicidades iguais para arcos simétricos ($\Upsilon = 1$), usando tamanhos de população iguais a 25, 50 e 100. A tabela apresenta uma comparação dos resultados do GA com 100 segundos como critério de parada, usando a instância `attAS`, *sLS*, $\Upsilon = 1$, *lf*=1, *rf*=1 e *sf*=1.

		$MUf = 0,85$			$MUf = 1,00$		
Obj	*W*	*Pop*25	*Pop*50	*Pop*100	*Pop*25	*Pop*50	*Pop*100
1	10	0,00	6,83	8,61	0,00	7,04	10,09
	20	0,00	6,29	10,88	0,00	4,40	8,71
	40	0,00	6,67	11,79	0,00	1,93	9,81
3	10	0,00	9,47	13,78	0,00	10,65	17,99
	20	0,00	6,97	18,23	0,00	8,53	17,60
	40	0,00	11,55	17,93	0,00	9,39	16,24
Média		0,00	7,96	13,54	0,00	6,99	13,41

este experimento, consideraram-se os resultados dos GA sem busca local (*sLS*) e com população com 25 indivíduos.

Para as oito combinações de parâmetros testados, o peso máximo $W = 10$ apresentou melhores resultados para 7 dos experimentos realizados. Na média, disponível na última linha da tabela, o intervalo de $[1, 10]$ se mostrou o mais adequado para os experimentos realizados.

A geração de resultados com 100 segundos teve o objetivo inicial de selecionar os parâmetros mais adequados para o GA, mas posteriormente os resultados dessa comparação foram usados apenas como uma pré-seleção dos parâmetros pois observou-se que o tempo de 100 segundos não é suficiente para a solução atingir um mínimo local de boa qualidade. Por este motivo, algumas execuções do GA foram realizadas exatamente com os parâmetros anteriores, mas por 1000 segundos

Tabela 3.5: Comparação dos resultados obtidos pelo GA usando peso máximo MW igual a 10, 20 e 40. A tabela apresenta uma comparação dos resultados do GA tendo executado por 100 segundos, usando a instância `attAS`, *sLS*, $Pop25$ e combinações de outros parâmetros.

	$\Upsilon = 0$						$\Upsilon = 1$					
	$MUf = 0,85$			$MUf = 1,00$			$MUf = 0,85$			$MUf = 1,00$		
Obj	$W10$	$W20$	$W40$	$W10$	$W20$	$W40$	$W10$	$W20$	$W40$	$W10$	$W20$	$W40$
1	0,00	1,59	0,62	0,00	1,35	1,05	0,00	0,30	1,34	0,00	4,16	5,43
3	0,00	0,68	3,91	1,97	3,45	0,00	0,00	4,30	2,44	0,00	3,84	2,80
Média	0,00	1,13	2,26	1,97	2,40	1,05	0,00	2,30	1,89	0,00	4,00	4,11

(em vez de 100s). O tempo de 1000 segundos, na máquina usada, é suficiente para a população convergir para uma solução de boa qualidade. Maior tempo acarreta uma solução final ainda melhor, mas não muito diferente daquela encontrada com 1000 segundos.

As Tabelas 3.6 e 3.7 apresentam uma comparação de resultados com critérios de parada igual a 1000 segundos. Esta comparação tem o objetivo de avaliar se os valores mais adequados dos parâmetros Pop e W mudam com o aumento do tempo computacional gasto pelo algoritmo. Populações com 25 e 50 indivíduos foram consideradas, assim como valores de 10 e 20 para W. A última linha de cada tabela apresenta a média do desvio percentual de cada coluna.

Tabela 3.6: Escolha de uma tamanho de população adequado para execuções de 1000 segundos.

	Obj_1				Obj_3			
	$MUf = 0,85$		$MUf = 1,00$		$MUf = 0,85$		$MUf = 1,00$	
W	$Pop25$	$Pop50$	$Pop25$	$Pop50$	$Pop25$	$Pop50$	$Pop25$	$Pop50$
$W10$	0,00	0,29	0,67	0,00	0,00	2,10	1,13	0,00
$W20$	0,94	0,00	0,00	0,22	2,07	0,00	1,32	0,00
Média	0,47	0,14	0,33	0,1	1,03	1,05	1,22	0,00

Pela análise da Tabela 3.6 conclui-se que a população de tamanho 50 se mostra mais adequada quando o GA roda por mais tempo.

Tabela 3.7: Escolha do intervalo dos pesos adequado para execuções de 1000 segundos.

	Obj_1				Obj_3			
	$MUf = 0,85$		$MUf = 1,00$		$MUf = 0,85$		$MUf = 1,00$	
Pop	$W10$	$W20$	$W10$	$W20$	$W10$	$W20$	$W10$	$W20$
$Pop25$	0,00	0,25	0,22	0,00	0,00	0,90	1,98	0,00
$Pop50$	0,99	0,00	0,00	0,67	3,29	0,00	2,17	0,00
Média	0,49	0,12	0,11	0,33	1,64	0,45	2,07	0,00

Observando-se a Tabela 3.7 As soluções obtidas com $W = 10$ e $W = 20$ são muito semelhantes, sendo que a segunda opção, na média, é um pouco melhor.

Como uma solução mais apurada é almejada, isto é, o critério de parada de 1000 segundos tem prioridade sobre o de 100 segundos, os testes apresentados nas próximas seções consideram sempre populações de 50 indivíduos e o intervalo de [1, 20] para o peso dos arcos. Testes preliminares indicam que intervalos maiores ou menores que os utilizados para atribuição de pesos e tamanho da população não contribuem para a melhoria das soluções finais.

A próxima subseção apresenta os resultados obtidos pelo GA rodando a instância `attAS` com todas as combinações de falhas possíveis.

3.5.4 Resultados obtidos pelo GA considerando todas as combinações possíveis de falhas

Esta subseção tem o objetivo de apresentar os resultados computacionais obtidos pelo GA quando aplicado à mesma instância, mas com tipos de falhas diferentes. Foram testadas todas as combinações das falhas *lf*, *rf* e *sf*, assim com o caso de rede sem falha. As Tabelas 3.8 e 3.9 apresentam os resultados desse experimento considerando as funções objetivos Obj_1 e Obj_3, respectivamente. As tabelas encontram-se subdivididas em duas colunas principais, a primeira apresenta resultados desconsiderando a restrição de multiplicidades iguais para arcos simétricos ($\Upsilon = 0$) enquanto a segunda considera este caso ($\Upsilon = 1$). Cada uma dessas duas colunas foi subdividida em outras duas, cada um delas com resultados para dois valores ($0,85$ e $0,90$) diferentes de máxima utilização no caso da rede com falha (MUf). As linhas correspondem às oito combinações possíveis de falhas da rede, sendo o primeiro caso a situação de operação normal desta (sem falha). Para cada combinação dos valores de *lf*, *sf*, *rf*, Υ e *MUf*, três valores foram apresentados: a melhor solução obtida (*Best*), melhoria percentual em relação à melhor solução inicial ($Impr\%$) e o tempo gasto pelo algoritmo até atingir o critério de parada ($tempo$). Neste experimento, a execução do GA por 200 gerações foi o critério de parada utilizado.

Tabela 3.8: Resultados do GA rodando 200 gerações e usando a instância `attAS` e a função objetivo Obj_1.

			$\Upsilon = 0$						$\Upsilon = 1$					
			$MUf = 0,85$			$MUf = 0,90$			$MUf = 0,85$			$MUf = 0,90$		
lf	*sf*	*rf*	*best*	*Impr%*	*tempo*	*best*	*Impr%*	*tempo*	*best*	*Impr%*	*tempo*	*best*	*Impr%*	*tempo*
0	0	0	346,67	17,46	17,63	346,67	17,46	17,73	406,67	33,48	17,40	406,67	33,48	17,44
0	1	0	553,00	21,97	342,34	525,00	22,53	337,08	650,00	32,67	332,99	621,33	32,22	327,87
0	0	1	625,00	17,26	213,99	604,33	15,87	215,13	723,33	25,79	204,71	693,33	24,91	202,52
1	0	0	634,33	15,20	614,33	611,33	15,05	618,59	740,67	23,33	583,61	696,00	24,84	580,23
0	1	1	675,00	19,10	563,09	642,67	19,20	552,10	769,33	28,10	514,39	744,67	26,27	530,39
1	1	0	681,33	18,44	974,36	645,33	18,89	932,07	784,00	26,45	899,73	738,67	27,15	898,17
1	0	1	638,67	17,80	700,06	608,33	17,79	685,81	742,00	25,95	683,21	703,33	26,12	659,50
1	1	1	673,00	20,64	1011,93	646,33	20,07	1047,13	774,00	28,69	990,79	750,67	26,84	970,21

Tabela 3.9: Resultados do GA rodando 200 gerações e usando a instância `attAS` e a função objetivo Obj_3.

			$\Upsilon = 0$						$\Upsilon = 1$					
			$MUf = 0,85$			$MUf = 0,90$			$MUf = 0,85$			$MUf = 0,90$		
lf	*sf*	*rf*	*best*	*Impr%*	*tempo*	*best*	*Impr%*	*tempo*	*best*	*Impr%*	*tempo*	*best*	*Impr%*	*tempo*
0	0	0	114335,13	36,13	17,68	114335,13	36,13	17,65	153047,58	45,83	17,57	153047,58	45,83	17,68
0	1	0	249722,43	35,56	333,68	241515,63	34,88	338,36	309658,90	43,71	327,57	300351,17	41,66	328,75
0	0	1	231657,24	35,14	223,70	219920,13	35,36	225,37	287560,02	43,36	214,50	276051,27	43,01	211,42
1	0	0	215713,41	36,74	648,01	198824,58	39,60	636,56	282273,10	42,00	602,90	266130,00	42,90	589,27
0	1	1	271742,75	36,57	557,19	262494,72	35,76	572,72	323870,80	46,11	530,75	317865,02	44,24	529,11
1	1	0	265444,93	36,65	950,63	255945,64	36,48	953,72	320609,82	45,85	897,92	304919,72	45,27	888,93
1	0	1	229636,43	37,12	747,35	217589,32	38,08	766,89	289706,73	44,35	711,54	283130,48	43,37	711,46
1	1	1	270359,69	37,06	1122,49	259453,22	36,86	1123,36	329286,34	45,75	1007,04	315567,71	45,01	1033,83

Como já esperado, no caso sem falha, não há diferença entre as soluções com valores diferenciados da utilização máxima no caso de falha, $MUf = 0,85$ ou $MUf = 0,90$. Isto pode ser observado na primeira linha de resultados da tabela. Os valores das soluções obtidas neste caso, assim como o tempo computacional gasto, são muito menores que os obtidos em qualquer uma das outras sete combinações de falhas.

Para os casos de rede com um único tipo de falha, a falha de arcos individuais possui os maiores valores de função objetivo, assim como gastou mais tempo sempre, enquanto que a falha de *span* resultou em soluções melhores mas gastou mais tempo que a falha de roteador. O tempo gasto parece estar relacionado com o número de falhas que deve ser testado, que é de 278, 54 e 40 para *lf*, *sf* e *rf*, respectivamente.

Embora o caso em que todas as falhas são consideradas simultaneamente tenha gasto mais tempo, como esperado, não foi o caso em que as soluções apresentam as soluções de maior custo. Como os valores de soluções de "110" e "111" parecem ser muito parecidos, o teste de falha de *span* não parece interferir muito na solução que já teve as outras falhas consideradas.

Novamente, as soluções e tempos obtidos no caso de nenhuma falha são muito menores que em todos os outros casos. Os resultados para do GA usando Obj_3 diferem em alguns aspectos dos obtidos para Obj_1. No caso desses testes, a falha dos *spans* parecem ter tido a maior influência para o aumento do valor das soluções.

A próxima seção tem o objetivo de apresentar os resultados do GA aplicado às instâncias de dados do problema WSP tratado no Capítulo 2.

3.5.5 Resultados do GA aplicado às instâncias do WSP

O objetivo desta subseção é o de apresentar resultados do GA aplicado à outras instâncias de dados. Além disso, tais instâncias apresentam dimensões e volume de dados diversos. Os testes foram feitos com duas instâncias de cada classe de dados usadas para os testes do Capítulo 2. A seleção das duas instâncias de cada classe considerou as duas primeiras listadas, mas com número de nós diferentes. Dessa forma, para cada classe de instâncias `hier`, `rand` e `wax`, uma instância com 50 e outra com 100 nós foram selecionadas. Como a classe `att` possui uma única instância, apenas esta foi considerada. Para todas as 7 instâncias testadas, apenas a matriz 12, com a maior

demanda, foi usada para testes.

Os parâmetros usados para a geração destes resultados são: $lf = 1$, $rf = 1$, $sf = 1$, $\Upsilon = 1$, $MUf = 0,85$, $Pop = 50$ e $W = 20$. As Tabelas 3.10 e 3.11 apresentam os resultados dessa comparação, a primeira considerando a função objetivo Obj_2, enquanto a segunda tabela usa a função objetivo Obj_3. O critério de parada novamente é a execução de 200 gerações. Um segundo critério foi necessário, devido ao alto tempo gasto pelas instâncias de 100 nós. Assim, o algoritmo também pára ao final de 2 horas de execução. Para estas instâncias, a função objetivo Obj_1 foi preterida pela Obj_2 porque os arcos dessas instâncias possuem capacidades iguais. A função objetivo Obj_1 não levaria em consideração este fato. Em cada uma das duas tabelas são informados 5 dados: melhor solução (*best*), melhoria percentual em relação à melhor solução inicial (*Impr%*), tempo total (*tempo*), número de gerações executadas (*gen*) considerando os critérios de parada utilizados e o valor de máxima utilização final da rede para o caso sem falha (U).

Tabela 3.10: Resultados para o GA aplicado à instâncias do problema do Capítulo 2. Os parâmetros foram fixados em $\Upsilon = 1$ e $MUf = 0,85$, usando a função objetivo Obj_2.

Instncia	*best*	*Impr%*	*tempo*	*gen*	U
att_12	221152,46	18,60	832,86	200,00	0,65
hier50a_12	35706,31	1,75	950,68	200,00	0,65
rand50_12	156021,77	23,50	1498,32	200,00	0,72
wax50_12	85648,58	17,70	1315,75	200,00	0,72
hier100a_12	89957,93	9,03	7245,94	71,67	0,69
rand100_12	323598,19	17,38	7251,86	63,33	0,74
wax100a_12	240407,57	16,64	7292,22	51,00	0,74

Tabela 3.11: Resultados para o GA aplicado à instâncias do problema do Capítulo 2. Os parâmetros foram fixados em $\Upsilon = 1$ e $MUf = 0,85$, usando a função objetivo Obj_3.

Instncia	*best*	*Impr%*	*tempo*	*gen*	U
att_12	913106,00	19,48	814,33	200,00	0,65
hier50a_12	77066,67	12,16	961,01	200,00	0,69
rand50_12	320666,67	20,95	1574,22	200,00	0,75
wax50_12	224333,33	16,50	1322,32	200,00	0,70
hier100a_12	205333,33	18,09	7258,03	75,33	0,74
rand100_12	624666,67	15,96	7253,60	63,33	0,75
wax100a_12	630333,33	13,85	7252,43	52,00	0,73

Como pode-se observar, o valor da solução e o tempo gasto dependem da instância que está

sendo usada, assim como dependem de sua dimensão. Já a melhoria obtida ($Impr\%$) depende da instância, mas não parece depender da sua dimensão. O tempo aumenta consideravelmente com o aumento da instância e, consequentemente, o número de gerações executadas diminui. Ainda, o valor de U é sempre menor ou igual ao de $MUf = 0,80$. Isso acontece, porque, no caso de uma falha, normalmente os valores das multiplicidades são incrementados. Com isso a capacidade dos arcos aumenta, mas a capacidade adicional necessária nesse caso não é utilizada se a rede estiver operando sem falha.

A próxima subseção apresenta resultados para o GA rodando a instância `att`, com as 12 matrizes de demanda disponíveis. O objetivo é verificar a alteração da solução e do tempo em função do aumento do tráfego na rede.

3.5.6 Aumento do tráfego na rede

Nesta subseção são usadas a instância `att` com as suas 12 matrizes de demanda. As instâncias estão dispostas em ordem crescente da soma total da demanda. A Tabela 3.12 apresenta os resultados dos testes realizados. Os parâmetros foram os mesmos usados para a geração da tabela anterior, apenas variando-se as instâncias que estão sendo consideradas. A segunda coluna apresenta a demanda total de cada instância. O número de gerações gasto não foi informado, pois o algoritmo sempre parou por atingir primeiramente o critério de parada de 200 gerações. Dessa forma, executaram-se 200 gerações para todas as instâncias.

Como pode-se verificar, o valor das soluções, bem como o tempo gasto para resolvê-las, tende a aumentar com o aumento do tráfego na rede. Outra característica importante que pode ser observada é que a melhoria da solução aumenta com o aumento da demanda.

A máxima utilização (no caso de rede sem falha) apresenta uma leve tendência em aumentar com o aumento do fluxo da rede, mas não seguiu rigorosamente esta tendência para as redes de maior demanda. Estes valores em geral estão bem abaixo de $0,80$, que é o valor da restrição de máxima utilização imposta. Como a restrição de máxima utilização da rede com falha afeta mais a solução, a restrição de máxima utilização de rede sem falha pouco interfere no incremento das multiplicidades.

A seção a seguir discute algumas alterações adicionais que poderiam ser consideradas pelo

Tabela 3.12: Resultados para o GA aplicado à instância `att` com 12 matrizes de demanda diferentes. Os parâmetros foram fixados em $\Upsilon = 1$ e $MUf = 0,85$.

		Obj_2				Obj_3			
Instncia	$\sum_{i,j \in V} D_{ij}$	*best*	*Impr%*	*tempo*	*U*	*best*	*Impr%*	*tempo*	*U*
att_1	3761,18	93666,04	5,91	339,27	0,30	426094,00	13,61	344,88	0,37
att_2	7522,36	96404,43	3,64	397,90	0,51	434384,00	12,52	417,31	0,49
att_3	11283,54	106703,92	3,36	704,58	0,57	460080,00	10,63	693,26	0,55
att_4	15044,71	105278,67	9,37	713,37	0,63	493240,00	15,18	712,87	0,64
att_5	18805,89	115833,40	9,06	743,74	0,71	524326,00	15,27	731,50	0,65
att_6	22567,07	120774,65	21,57	749,45	0,67	564528,00	17,10	734,23	0,65
att_7	26328,25	136028,68	24,46	783,67	0,67	620898,00	21,03	758,07	0,74
att_8	30089,43	174753,27	16,84	774,44	0,61	703378,00	18,41	743,76	0,66
att_9	33850,61	185837,47	14,83	797,98	0,65	752286,00	16,82	772,00	0,62
att_10	37611,79	202868,67	10,76	815,50	0,72	802026,00	17,09	783,22	0,67
att_11	41372,97	211869,86	15,94	806,89	0,61	872904,00	18,02	771,50	0,61
att_12	45134,15	221152,46	18,60	832,86	0,65	913106,00	19,48	814,33	0,65

problema abordado neste capítulo.

3.6 Extensões do problema

Como certamente pode ser observado, este problema possui inúmeras variações e modificações que podem ser usados para adaptá-lo às necessidades de sua aplicação. Além das que já foram discutidas, na prática existem outras que podem ser adicionadas. Algumas delas são discutidas nesta seção.

Na resolução deste problema considerou-se sempre o projeto de uma rede, desconsiderando as multiplicidades (capacidades) da rede atual. Pode-se adicionar ao problema a situação em que apenas se objetive expandir a rede, e não projetá-la. Para tanto, basta que os valores de multiplicidades atuais sejam informados, e as multiplicidades dos arcos sejam inicializados com estes valores (em vez de 1 como é feito na linha 1 da Figura 3.6).

Na abordagem apresentada, considerou-se a situação em que o valor de uma multiplicidade possa ser um inteiro positivo qualquer. Poder-se-ia, por exemplo, impor que as multiplicidades não fossem livremente aumentadas, além de estipular um valor mínimo de multiplicidade para cada arco. A única modificação necessária para o GA tratar esta restrição é a leitura de um vetor de multiplicidades iniciais, lido como parâmetro de entrada, e a atribuição desses valores ao arcos

(linha 1 da Figura 3.6).

No caso de máxima utilização dos arcos em caso de rede com falha, considerou-se o mesmo valor *MUf* para falhas diferentes. Este valor poderia ser diferenciado dependendo da falha que está sendo tratada.

Um quarto objetivo poderia ser proposto para o problema, relacionado com a minimização do atraso, como também sugerido por Moy [Moy98]. Neste caso o atraso está relacionado com o tempo de envio de dados gasto por cada arco.

Finalmente, o problema poderia ser aplicado para o caso em que a topologia não é dada, mas também deve ser definida. Neste caso, a utilização de cada arco poderia ter um preço fixo associado.

As conclusões são apresentadas a seguir, finalizando este capítulo.

3.7 Conclusões

Neste capítulo estudou-se um problema de *projeto* da rede, o qual trata da minimização do número de multiplicidades das linhas de transmissão de tal forma a satisfazer a demanda sem que haja sobrecarga de nenhum arco, inclusive quando houver falhas na rede. O caso de teste de falhas é uma iniciativa de se projetar redes atendendo os requisitos sugeridos para qualidade de serviço (*QoS*) em roteamento de dados na Internet. A rede projetada não somente deve ser segura e eficiente, roteando toda demanda sem sobrecarga de arcos, mas também deve ter o menor custo possível.

Para se resolver este problema, usou-se um algoritmo genético. A estrutura da população foi mantida a mesma usada para o problema WSP, mas a forma de avaliar uma solução deste problema é diferente e representa cerca de 99% do tempo total gasto pelo GA. O desempenho do GA foi testada usando-se combinações variadas de parâmetros e instâncias distintas de dados. Inicialmente, testes foram realizados para selecionar parâmetros que conduzem a soluções de melhor qualidade. Foram escolhidos *sLS* em vez de *LSm* ou *LSw*, $Pop = 50$ em vez de $Pop = 25$ e $pop = 100$, e $W = 20$ em vez de $W = 10$ e $W = 40$. Usando-se estes três parâmetros fixos, assim como $MU = 0,80$, os testes consideraram 3 funções objetivos diferentes, a inclusão e a desconsideração da restrição de multiplicidades iguais para arcos simétricos, assim como valores diferentes para a utilização máxima de um arco em caso de rede com falha. Ainda, os três tipos de falhas foram combinadas

em 8 configurações diferentes.

Pela análise dos resultados, concluiu-se que a restrição de se ter multiplicidades iguais para arcos simétricos encarece muito a solução, além de demandar mais tempo para a avaliação desta. Ainda, quanto menor é o valor de máxima utilização de um arco, maior é o valor da solução, visto que a largura de banda ociosa também é maior.

Como já se esperava, considerar a possibilidade de falhas torna o projeto mais caro, mas fornece uma rede mais segura. A falha de arcos individuais teve maior influência no aumento do custo de solução, assim como maior tempo gasto para avaliá-la. O tempo maior certamente é devido ao maior número de itens de falha que devem ser testados (o número de arcos é bem maior que o número de nós e de *spans*, para as instâncias consideradas nos experimentos).

Finalizando a análise, observou-se que a qualidade de solução, assim como o tempo que o GA gasta para resolvê-la, estão intrinsecamente relacionados com sua dimensão, assim como com o volume de tráfego pela rede demandada. Além disso, a percentagem da melhora *Impr* depende muito da instância sob análise e se beneficia com o aumento do fluxo na instância, mas não mostra influência com o tamanho da instância.

Capítulo 4

Algoritmos de Caminhos Mínimos Dinâmicos

A busca local do MA para o WSP é baseada em mudanças dos pesos de arcos da rede. Ainda, na simulação de falhas na rede, consideradas no problema de projeto, arcos devem ser removidos desta. Nas duas situações, uma mudança simples ocorreu e deve ser refletida no grafo de caminhos mínimos. Este, por sua vez, não deve ser inteiramente recalculado, e sim atualizado com um algoritmo para caminhos mínimos dinâmicos (*Dynamic Shortest Paths* - DSP). Neste capítulo é apresentado um estudo minucioso e extenso de algoritmos de caminhos mínimos dinâmicos.

Sete algoritmos clássicos foram estudados, supostamente os que têm execução mais rápida, sendo que para cada um deles foi proposta uma variante de pilha-reduzida, gerando outros sete algoritmos. A técnica de pilha-reduzida é uma contribuição dessa tese que visa a redução do número de nós manipulados por pilhas durante a atualização dos caminhos mínimos. Esta idéia é diferenciada para o caso de incremento e decremento do peso de um arco, e adaptada para cada algoritmo. Dos sete algoritmos clássicos, quatro são para o caso de incremento e três para decremento. Somente um de incremento e um de decremento são para atualizar grafos, sendo os demais para atualizar árvores de caminhos mínimos.

O capítulo encontra-se organizado da seguinte forma. Na Seção 4.2, alguns detalhes das implementações são fornecidos. Algoritmos de incremento de caminhos mínimos dinâmicos, assim como a técnica de redução de pilha para cada caso, são apresentados na Seção 4.3. Uma discussão dos algoritmos de incremento clássico e de pilha-reduzida, com respeito ao tamanho da pilha e uso de memória, é apresentada na Seção 4.4. Esta é seguida pela Seção 4.5, a qual aborda o caso

de decremento de peso. Uma discussão dos algoritmos de decremento `std` e `rh` com relação ao tamanho da pilha e uso da memória é apresentada na Seção 4.6. Os resultados computacionais são fornecidos na Seção 4.7 e as conclusões aparecem na Seção 4.8.

4.1 Introdução

Atualizar os caminhos mínimos entre os nós de uma rede após uma mudança na estrutura ou no valor do peso de um arco é um problema importante de grafos, sendo base de muitos outros problemas. Considera-se um grafo direcionado com pesos associados aos arcos $G = (V, E, w)$, sendo V o conjunto de vértices, E o conjunto de arcos e $w \in \mathbb{R}^{|E|}$ o vetor de pesos associados aos arcos. Dado um vértice fonte $s \in V$, o problema de caminho mínimo com uma única origem (*single source shortest path*) consiste em encontrar uma árvore de caminhos mínimos do nó fonte s (*source*) para cada nó $v \in V$. Invertendo a direção de cada arco do grafo, o problema é transformado no problema de caminho mínimo com único destino (*single destination shortest path*). Nesse caso o nó s passa a ser chamado de t (*target*). O problema é definido em termos de uma *árvore de caminhos mínimos*, mas uma aplicação particular pode precisar de um *grafo de caminhos mínimos*. A diferença entre os dois é que o grafo inclui todos os caminhos mínimos entre um nó $u \in V$ e o nó destino t, enquanto uma árvore armazena um único caminho entre eles. A Figura 1.2, ilustrada no Capítulo 1, apresenta esta diferença.

Existem aplicações nas quais o g^{SP} (grafo de caminhos mínimos) é um dado do problema, mas ele precisa ser atualizado após a mudança do peso de um ou mais arcos. Considerando a mudança do peso de um único arco $a \in E$, geralmente somente uma pequena parte do grafo é afetada. Por esta razão, é recomendável evitar que o grafo g^{SP} seja recalculado e sim apenas atualizado, focando a parte em que ele foi afetado pela mudança do peso do arco a. Este problema é conhecido como *Caminhos Mínimos Dinâmicos* (*Dynamic Shortest Path* - DSP). Um algoritmo é referenciado como *completamente dinâmico* (*fully-dynamic*) se ambas operações (incremento do peso de um arco em $\triangle$ e decremento em $\bigtriangledown$) são realizadas, e *semi-dinâmico* (*semi-dynamic*) se somente incremento ou decremento é viável.

Alguns algoritmos já foram propostos para resolver este problema, mas até agora o algoritmo (`RR`) de Ramalingam e Reps [RR96a] tem sido o mais difundido [BRRT03, BFRT03, FT00,

FMSN98]. Como trabalhos anteriores, pode-se citar Murchland [Mur70], Goto e Sangiovanni-Vincentelli [GSV78] e Dionne [Dio78]. Considerando os algoritmos de decremento, pode-se salientar Gallo [Gal80] e Fujishige [Fuj81], enquanto que para incremento (incluindo desativação de arcos), é relevante o trabalho de Even e Shiloach [ES81].

Recentemente Demetrescu, Frigioni e Nanni [DFMSN00] propuseram uma especialização do algoritmo do Ramalingam e Reps para atualizar uma árvore de caminhos mínimos, sendo este uma revisão do trabalho anterior [FMSN98]. Na versão mais recente eles propõem um algoritmo completamente dinâmico, enquanto que na primeira somente um algoritmo de incremento foi proposto. Quando uma parcela pequena dos nós afetados possuem caminhos mínimos alternativos, o algoritmo de incremento proposto por Demetrescu [Dem01] geralmente tem melhor desempenho. Para atualizar todos os pares de caminhos mínimos em grafos direcionados com pesos inteiros e positivos, cita-se o algoritmo completamente dinâmico de Demetrescu e Italiano [DI01, DI03], com resultados experimentais publicados por Demetrescu, Emiliozzi e Italiano [DEI03].

Neste capítulo, é mostrado experimentalmente que o algoritmo de Ramalingam e Reps não é a melhor opção em todas as situações. Entretanto, uma de suas vantagens principais é ter uma bom desempenho em todas as situações testadas. Este algoritmo atualiza um grafo de caminhos mínimos, em vez de uma árvore, embora ele possa ser facilmente especializado para atualizar uma árvore [DFMSN00]. Even e Shiloach [ES81] propuseram um algoritmo de incremento semi-dinâmico que funciona em cascata. Este pode demandar muito tempo de execução no caso de grandes incrementos de peso, isto é, $\triangle > 5$. O algoritmo `RR` possui bom desempenho independente do valor de $\triangle$. O algoritmo semi-dinâmico de Demetrescu [Dem01], para atualizar uma árvore de caminhos mínimos, pode ter um desempenho muito bom se a grande maioria dos nós afetados não tiver caminhos alternativos, mas seu desempenho pode ser altamente influenciado no caso em que uma parcela considerável dos nós afetados possuir caminhos alternativos. Novamente, o algoritmo `RR` possui bom desempenho nas duas situações. Ainda que o algoritmo de Frigioni, Marchetti-Spaccamela e Nanni [FMSN96] seja teoricamente melhor que o `RR`, este apresentou um desempenho tão boa quanto o `RR` nos experimentos computacionais apresentados [FMSN96].

Muitos estudos teóricos para algoritmos para caminhos mínimos dinâmicos têm sido apresentados, mas poucos experimentos foram realizados. Frigioni et al. [FINP98] compararam o algoritmo

de Ramalingam e Reps com o algoritmo de Frigioni et al. [FMSN96], para atualizar um caminho mínimo de fonte única. Eles concluíram que o algoritmo de Ramalingam e Reps é geralmente melhor na prática, com respeito a tempos computacionais, embora seu algoritmo tenha uma complexidade de pior caso melhor, segundo a avaliação de número total de arcos consultados proposta por Ramalingam e Reps [RR96b]. Demetrescu et al. [DFMSN00] compararam os algoritmos de incremento de Demetrescu et al. [DFMSN00], Frigioni et al. [FINP98] e Ramalingam e Reps [RR96a] e uma especialização do primeiro [DFMSN00], descrita por Demetrescu [Dem01]. Para o grupo de instâncias testadas, os resultados mostraram que o algoritmo proposto tem melhor desempenho na atualização de árvores.

O trabalho apresentado neste capítulo [BRRT03] propõe uma técnica para a redução do tamanho da pilha usada em muitos algoritmos de caminhos mínimos. Para mudanças unitárias de peso (incrementos e decrementos), a atualização é realizada sem o uso de pilhas. Apesar das pilhas fornecerem um mecanismo eficiente de organização dos elementos de um conjunto, cada operação de atualização, inserção ou remoção de um elemento é realizada com complexidade $O(log_n)$. Então é desejável que esta não seja utilizada caso não haja necessidade. A redução do tamanho da pilha quase sempre acarreta na diminuição dos tempos gastos por estes algoritmos. No caso de incremento, quando esta idéia é aplicada ao RR, este reduz o tempo computacional independente do valor do incremento do peso do arco ser grande ou pequeno. Nos experimentos computacionais, muitos algoritmos de caminhos mínimos dinâmicos são comparados, com e sem a técnica de pilha-reduzida. O objetivo da comparação dos algoritmos clássicos com os de pilha-reduzida é saber se a facilidade do uso das pilhas compensa o tempo por elas demandado, mesmo se outra estrutura mais simples pudesse ser usada. Também comparam-se os resultados com o algoritmo de Dijkstra, o qual recalcula totalmente a árvore de caminhos mínimos. Os pseudo-códigos destes algoritmos são descritos para o problema de caminhos mínimos com destino único.

O nome de cada um dos algoritmos apresentados nas próximas seções é inicializado pela letra G ou T, se a atualização é realizada num grafo ou árvore de caminhos mínimos, respectivamente. O nome também indica os autores da idéia original do algoritmo (em sobrescrito) e o sinal + ou − (em subscrito) se ele se refere a um algoritmo de incremento ou decremento, respectivamente. Quando o nome do algoritmo é usado sem sinal, refere-se a ambos casos (incremento e decremento), ou este

é seguido da indicação `Incr` ou `Decr`. Para as variantes de pilha-reduzida, o nome é antecedido por `rh`. Os termos `std` e `rh` são usados para referenciar os algoritmos clássicos (*standard*) e de pilha-reduzida (*reduced heap*), respectivamente.

Dos algoritmos clássicos, o T_+^{RR} e T_-^{RR} são particularizações para atualização de árvores dos algoritmos G_+^{RR} e G_-^{RR} para atualização de grafos. Esta particularização foi proposta nesta tese, mas uma especialização similar, supostamente mais lenta, já havia sido proposta por Frigioni et al. [FMSN98]. Os algoritmos T_+^{KT} e T_-^{KT} são descritos e experimentados computacionalmente pela primeira vez, embora a idéia da árvore especial tenha sido proposta por King e Thorup [KT01].

A seguir, alguns detalhes de implementação são descritos.

4.2 Detalhes das implementações

Nesta seção são apresentados os dois conjuntos de estruturas de dados usadas na implementação dos algoritmos. Ainda, alguns detalhes de implementação são fornecidos. A maior parte dessas estruturas foram usadas para os métodos dos Capítulos 2 e 3. Porém, uma nova apresentação se torna necessária, agora mais detalhada, pois cada algoritmo é minuciosamente descrito e, uma modificação da implementação aqui apresentada, pode gerar um algoritmo com um desempenho muito diferente.

4.2.1 Estruturas de dados

Inicialmente, os dois conjuntos de estruturas de dados usados nas implementações são fornecidos: um para representar os dados do problema e outro para representar uma solução. A Figura 4.1 mostra um exemplo com um grafo e suas estruturas correspondentes. No lado esquerdo da figura está o grafo com arcos e nós indexados, enquanto que no lado direito as estruturas de dados correspondentes (para representação de grafos e árvores) são fornecidas.

As estruturas da parte superior são usadas para representar os dados de entrada, enquanto que as estruturas de dados mais abaixo são usadas para representar uma solução armazenada em grafo (`w`, `d`, $\mathtt{g}^{\mathtt{SP}}$ e δ) ou em árvore (`w`, `d`, e $\mathtt{t}^{\mathtt{SP}}$).

Os dados de entrada são armazenados em representação *progressiva* e *reversa*. Quatro vetores são usados com este fim. O vetor `forward`, com $|E|$ posições, armazena os arcos, sendo que cada

arco consiste de seus nós origem (*tail*) e destino (*head*). Os arcos encontram-se ordenados por seus nós origens, com os empates resolvidos ordenado-os pelos nós destinos. O vetor `point`, com $|E|+1$ posições, contém a lista de arcos saintes de cada nó. Cada elemento desse vetor indica a posição inicial no vetor `forward` da lista de arcos saintes do nó i. Assume-se que o último arco da lista esteja na posição $\texttt{point}[i+1]-1$ do vetor `forward`.

O vetor `reverse`, o qual possui $|E|$ posições, armazena os arcos no sentido reverso. Para tanto, estes são ordenados em ordem crescente do valor dos seus nós destinos, com os empates resolvidos ordenando-os pelos nós origens. Para poupar espaço, cada arco é representado no vetor `reverse` pelo seu índice no vetor `forward`. A lista de arcos entrantes em cada nó é obtida através do vetor `rpoint`, o qual possui $|E|+1$ posições. Cada elemento desse vetor indica a posição inicial no vetor `reverse` da lista de arcos entrantes no nó i. Assume-se que o último arco da lista esteja na posição $\texttt{point}[i+1]-1$ do vetor `reverse`.

As soluções são representadas como árvores ou grafos, dependendo do algoritmo. Em ambos os casos, o vetor `w`, com $|E|$ posições, armazena os pesos, enquanto o vetor `d`, com $|V|$ posições, armazena as distâncias de cada nó ao nó destino.

No caso das árvores, um vetor $\texttt{t}^{\texttt{SP}}$, com $|V|$ posições, indica, para cada posição i, o índice do arco de saída presente num dos caminhos mínimos do nó i ao nó destino. Como o nó de destino (no caso o nó 3) não possui arco sainte em t^{SP} ou g^{SP}, sua posição em t^{SP} é preenchida com o valor zero.

No caso de grafos, dois vetores são usados. O vetor g^{SP}, com $|E|$ posições, é um indicador $0-1$.

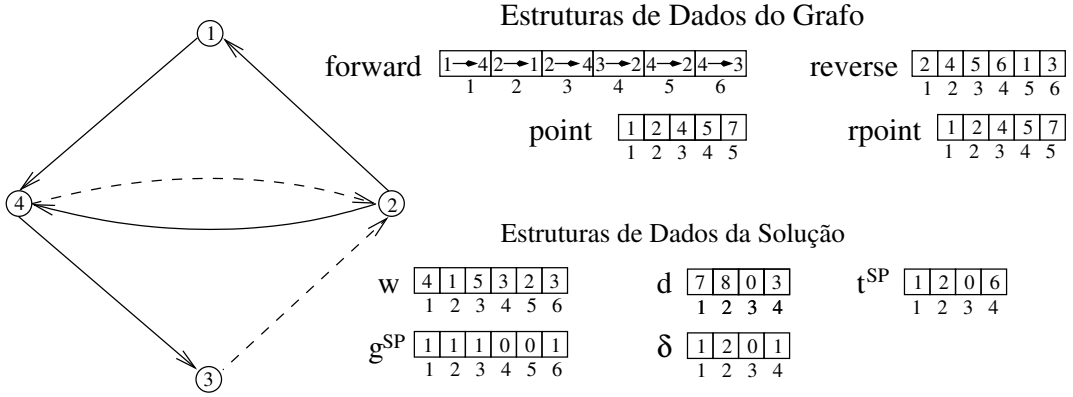

Figura 4.1: Estruturas de dados relacionadas aos dados e soluções do problema.

A posição *i* possui o valor *1* somente no caso em que o arco correspondente faz parte de pelo menos um caminho mínimo entre um nó qualquer e o nó destino. Finalmente, o vetor δ, com $|V|$ posições, armazena em cada posição *i* o número de arcos saintes do nó *i* no grafo de caminhos mínimos.

4.2.2 Implementação

Os pseudo-códigos são descritos em alto nível, suprimindo alguns detalhes de implementação, objetivando uma apresentação mais compacta dos algoritmos. Nesta subseção, algumas operações básicas referenciadas nos pseudo-códigos são implementadas.

Muito frequentemente deve-se consultar todos os arcos entrantes ou saintes de um determinado nó. Para consultar todos os arcos saintes de um nó u, isto é, $e = (\overrightarrow{u,v}) \in \texttt{OUT}(u)$, consultam-se as posições $\texttt{point}[u], \ldots, \texttt{point}[u+1]-1$ do vetor `forward`. Similarmente, para consultar os arcos entrantes num nó u, isto é, $e = (\overrightarrow{s,u}) \in \texttt{IN}(u)$, consultam-se as posições $\texttt{reverse}[\texttt{rpoint}[u]], \ldots, \texttt{reverse}[\texttt{rpoint}[u+1]-1]$ do vetor `forward`.

Nos algoritmos de King e Thorup [KT01] apresentados nas próximas seções, uma árvore particular é usada. Quando o algoritmo precisa determinar se o nó u tem um caminhos mínimo alternativo, o algoritmo não consulta todos arcos do conjunto $\texttt{OUT}(u)$. Ao invés disso, ele consulta os arcos $e = (\overrightarrow{u,v}) \in \texttt{OUT}_e^{KT}(u) \subseteq \texttt{OUT}(u)$. Considere ser $e = (\overrightarrow{u,v}) \in \texttt{OUT}(u)$ o arco armazenado em $\texttt{t}_\texttt{u}^{\texttt{SP}}$. O conjunto $\texttt{OUT}_e^{KT}(u)$ é composto por todos os arcos $\texttt{forward}[e+1], \ldots, \texttt{forward}[\texttt{point}[u+1]-1]$.

O conjunto Q é armazenado como um vetor de $|V|$ posições, o qual armazena em cada posição *i* o i-ésimo elemento do conjunto. O conjunto U é representado da mesma forma.

Todos os algoritmos descritos neste capítulo fazem uso da estrutura de dados do tipo *pilha*. Uma pilha é um vetor organizado como uma árvore binária e tem por objetivo possibilitar a manutenção ordenada dos elementos de um conjunto que estejam continuamente sendo inseridos, removidos ou modificados. Cada uma destas operações é executada com complexidade $O(log_n)$. Cada elemento possui uma chave associada, sendo este o valor pela qual se deseja organizá-los. As principais funções de pilha utilizadas na descrição dos pseudo-códigos apresentam a mesma denominação usada por Ramalingam e Reps [RR96a]. São elas:

- `HeapMember(`H,u`)`: retorna o valor *1* se o elemento u está na pilha H, e *0* caso contrário;
- `HeapSize(`H`)`: retorna o número de elementos da pilha H;

- `FindAndDeleteMin(`H`)`: retorna o item da pilha H com a chave mínima e o remove de H;
- `InsertIntoHeap(`H,u,k`)`: insere o item u com chave k na pilha H que tem o seu menor elemento como raiz;
- `AdjustHeap(`H,u,k`)`: se $u \in H$, muda-se a chave do elemento u para k e atualiza-se a pilha H. Caso contrário, o nó u é inserido na pilha H.

As funções `FindAndDeleteMax()` e `InsertIntoHeapMax()` são usadas no caso em que a pilha está organizada de forma a manter o maior elemento como raiz. A função `AdjustHeap()` atualiza a pilha contendo raiz como máximo ou mínimo, depende de um parâmetro que fornece esta informação e que será suprimido nos códigos.

4.3 Algoritmos de incremento

Nesta seção, são descritos quatro algoritmos de caminhos mínimos dinâmicos para o caso de incremento do peso de arcos (*Incremental Dynamic Shortest Path Algorithms*). Inicialmente, o algoritmo do Ramalingam e Reps [RR96a] para atualizar um grafo de caminhos mínimos é apresentado, bem como sua especialização para árvores. Ainda, a árvore especial proposta por King e Thorup [KT01] é usada em outra especialização do algoritmo de Ramalingan e Reps para atualizar uma árvore de caminhos mínimos. Finalmente, o algorithmo de Demetrescu [Dem01], também para atualizar árvores, é apresentado. Estes algoritmos padrão (*standard - std*) são refenciados por G_+^{RR}, T_+^{RR}, T_+^{KT} e T_+^{D}, respectivamente.

Uma técnica para reduzir o uso do tamanho das pilhas usadas por estes algoritmos também é apresentada. Esta técnica é aplicada a todos os algoritmos `std` mencionados anteriormente. Estes algoritmos com pilhas reduzidas (*reduced heap* - `rh`) são referenciados como rhG_+^{RR}, rhT_+^{RR}, rhT_+^{KT}, e rhT_+^{D}, respectivamente. Sua aplicação é requerida quando o peso de um arco for incrementado em qualquer quantidade. Se uma remoção explícita de arco é requerida, o valor infinito pode simplesmente ser designado a seu peso. Os algoritmos recebem como entrada o arco a, o qual tem seu peso originalmente incrementado, o vetor w de pesos (com w_a já atualizado), e o vetor de distância d. Além disso, no caso de um algoritmo de atualização de árvore, o vetor t^{SP} é também

um dado de entrada para o algoritmo. Quando a atualização é realizada num grafo de caminhos mínimos, g^{SP} e δ são dados de entrada.

As variantes com pilhas reduzidas também recebem como entrada o parâmetro $\triangle$. Como resultado, os algoritmos atualizam os valores dos dados de entrada.

Todos estes algoritmos são baseados na mesma idéia: um conjunto Q de nós afetados é determinado e as mudanças apenas são aplicadas à parte do grafo afetada. Mudanças podem ocorrer nas distâncias desses nós, assim como a remoção e adição de arcos entrantes e saintes. Para a atualização, as versões `std` dos algoritmos inserem numa pilha H todos os nós inicialmente no conjunto Q, enquanto as variantes `rh` inserem apenas um subconjunto dos nós de Q na pilha H. Nas variantes com pilhas reduzidas, o conjunto Q é indefinido e, inicialmente, todos nós $u \in Q$ têm suas distâncias incrementadas em $\triangle$. Em seguida, a diferença entre a distância original e a atual do nó origem do arco a é calculada e armazenada em $\bigtriangledown$. Todos os nós $u \in Q$ têm suas distâncias decrementadas de $\bigtriangledown$. Somente os nós que têm caminhos ainda mais curtos, e que por isso têm suas distâncias decrescidas, são inseridos na pilha H. Como será visto na seção de experimentos, o número de nós inseridos na pilha é em geral muito menor que o número de nós do conjunto Q. Nas próximas subseções, os pseudo-códigos das versões `std` e `rh` destes algoritmos são apresentados.

4.3.1 Algoritmo de incremento proposto por Ramalingam e Reps para atualizar um grafo de caminhos mínimos (G_+^{RR})

O algoritmo G_+^{RR} atualiza um grafo de caminhos mínimos $g^{SP} = (V, E^{SP})$ quando o peso do arco a é incrementado de $\triangle$. A Figura 4.2 mostra o pseudo-código para o algoritmo G_+^{RR}. Claramente, se o arco a não está no grafo atual g^{SP}, o algoritmo pára (linha 1). Caso contrário, o arco a é removido de g^{SP} (linha 2) e δ_u é decrementado em um (linha 3). Se u, o nó inicial do arco a, tem um caminho alternativo para o nó destino, isto é se $\delta_u > 0$, então o algoritmo pára (linha 4). Caso contrário, o conjunto Q é inicializado com o nó u (linha 5).

A Figura 4.3 apresenta um exemplo ilustrativo para auxiliar a compreensão do algoritmo. Esta figura é composta por três grafos: `A)` rede original; `B)` grafo após execução das primeiras 21 linhas do pseudo-código; `C)` grafo final atualizado.

O laço na linhas 6-15 identifia os demais nós afetados de g^{SP} e os adiciona no conjunto Q. Para

```
procedure G_+^RR(a = (u,v), w, d, δ, g^SP)
1     if g_a^SP = 0 return;
2     g_a^SP = 0;
3     δ_u = δ_u − 1;
4     if δ_u > 0 then return;
5     Q = {u};
6     for u ∈ Q do
7           d_u = ∞;
8           for e = (s,u) ∈ IN(u) do
9                 if g_e^SP = 1 then
10                      g_e^SP = 0;
11                      δ_s = δ_s − 1;
12                      if δ_s = 0 then Q = Q ∪ {s};
13                end if
14          end for
15    end for
16    for u ∈ Q do
17          for e = (u,v) ∈ OUT(u) do
18                if d_u > d_v + w_e then d_u = d_v + w_e;
19          end for
20          if d_u ≠ ∞ then InsertIntoHeap(H, u, d_u);
21    end for
22    while HeapSize(H) > 0 do
23          u = FindAndDeleteMin(H, d);
24          for e = (s,u) ∈ IN(u) do
25                if d_s > d_u + w_e then
26                      d_s = d_u + w_e;
27                      AdjustHeap(H, s, d_s);
28                end if
29          end for
30          for e = (u,v) ∈ OUT(u) do
31                if d_u = w_e + d_v then
32                      g_e^SP = 1;
33                      δ_u = δ_u + 1;
34                end if
35          end for
36    end while
end G_+^RR.
```

Figura 4.2: Pseudo-código do procedimento G_+^{RR}.

cada nó $u \in Q$, sua distância recebe o valor ∞ (linha 7). Para cada arco $e = (\overrightarrow{s,u})$ incidente em u, se $e \in g^{SP}$, e é removido de g^{SP} (linha 10) e δ_s é atualizado (linha 11). Se s, o nó final do arco e, não tiver caminho alternativo para o nó destino, ele é então inserido no conjunto Q (linha 12).

O laço nas linhas 16-21 atualiza as distâncias dos nós $u \in Q$ (linha 18) e insere estes nós na pilha H (linha 20), caso suas distâncias tiverem sido decrescidas. O objetivo principal deste laço é atualizar as distâncias do nós que tiverem caminhos alternativos ligando nós fora do conjunto Q.

O laço nas linhas 22-36 atualiza as distâncias dos nós de Q usando a pilha H (linhas 24-29) e restaura g^{SP} (linhas 30-35). Os nós u com distância mínima do destino são removidos um a um da pilha H (linha 23) e todos os arcos $e = (\overrightarrow{s,u})$ partindo do nó u são consultados. No caso em que d_s

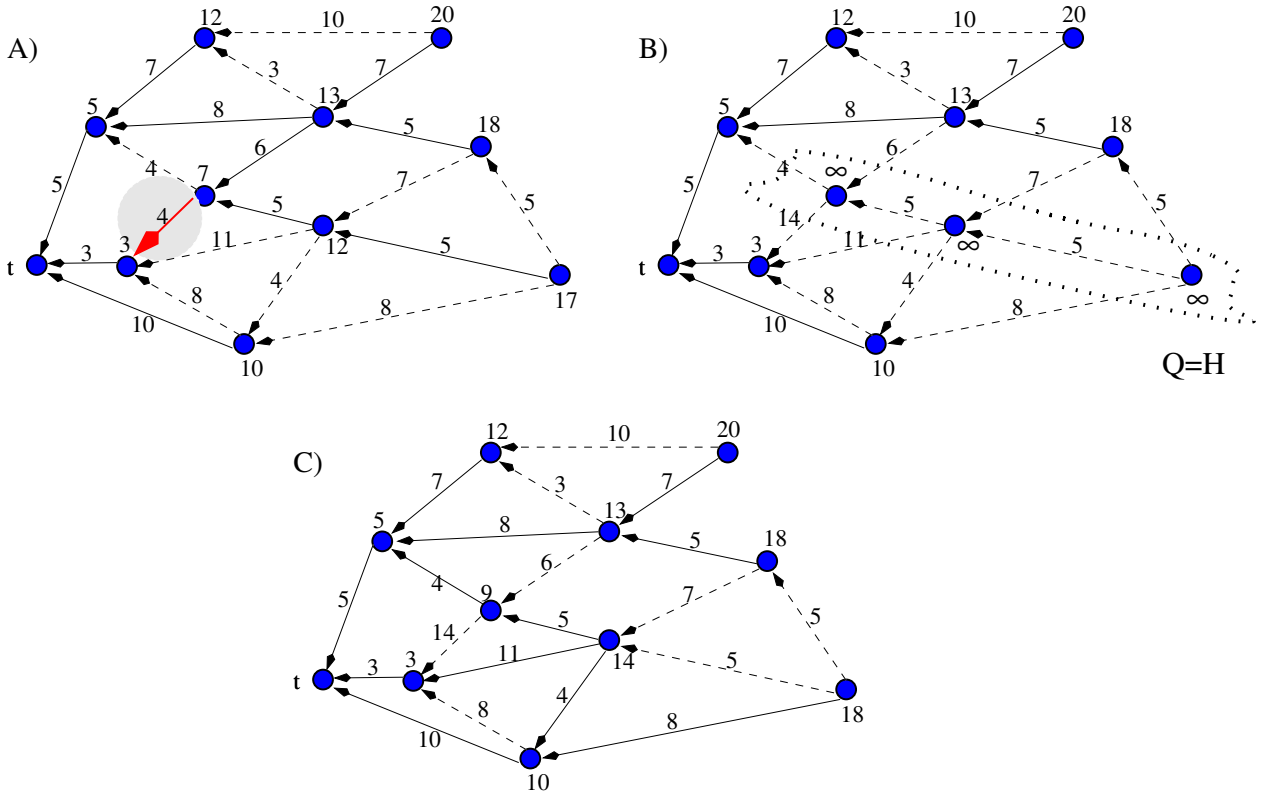

Figura 4.3: Exemplo ilustrativo da operação do algoritmo G_{+}^{RR}.

possa ser diminuir, s recebe a nova distância (line 26) e a pilha H é ajustada (linha 27). Em fato, este laço é o algoritmo de Dijkstra aplicado aos nós de Q. A seguir, todos arcos $e = (\overrightarrow{u,v})$ partindo do nó u são consultados. Se d_u é a menor distância para o nó destino, então e é adicionado ao grafo g^{SP} (linha 32) e δ_u é atualizado (linha 33).

A Figura 4.4 apresenta o pseudo-código para rhG_{+}^{RR}, a variante com pilha reduzida. As primeiras 15 linhas são idênticas às primeiras 15 da Figura 4.2, com excessão da linha 7. Em vez da distância do nó u receber o valor ∞, ela é apenas incrementada em $\triangle$.

No caso de incremento unitário, $\triangle = 1$, os comandos das linhas 16-42 não são executados e a pilha H não é usada.

Linhas 17-21 calcula a quantidade que a distância dos nós $u \in Q$ decrescerão. Denota-se por Q_0 o primeiro elemento inserido no conjunto Q, o qual é o nó inicial do arco a (inserido em Q na linha 5). Inicialmente, a distância atual do nó Q_0 é armazenada (linha 17). A seguir, todos os arcos partindo de Q_0 são consultados e se um caminho alternativo é identificado, d_{Q_0} é atualizado

```
procedure rhG+^RR(a = (u→,v), w, d, δ, g^SP, △)
1    if g_a^SP = 0 return;
2    g_a^SP = 0;
3    δ_u = δ_u − 1;
4    if δ_u > 0 then return;
5    Q = {u};
6    for u ∈ Q do
7        d_u = d_u + △;
8        for e = (s→,u) ∈ IN(u) do
9            if g_e^SP = 1 then
10               g_e^SP = 0;
11               δ_s = δ_s − 1;
12               if δ_s = 0 then Q = Q ∪ {s};
13           end if
14       end for
15   end for
16   if △ > 1 then
17       dist = d_Q0;
18       for e = (Q0→,v) ∈ OUT(Q0) do
19           if d_Q0 > d_v + w_e then d_Q0 = d_v + w_e;
20       end for
21       ▽ = dist − d_Q0;
22       for u ∈ Q \ Q0 do
23           d_u = d_u − ▽;
24           flag = 0;
25           for e = (u→,v) ∈ OUT(u) do
26               if d_u > d_v + w_e then
27                   d_u = d_v + w_e;
28                   flag = 1;
29               end if
30           end for
31           if flag = 1 then InsertIntoHeap(H,u,d_u);
32       end for
33       while HeapSize(H) > 0 do
34           u = FindAndDeleteMin(H,d);
35           for e = (s→,u) ∈ IN(u) do
36               if d_s > d_u + w_e then
37                   d_s = d_u + w_e;
38                   AdjustHeap(H,s,d_s);
39               end if
40           end for
41       end while
42   end if
43   for u ∈ Q do
44       for e = (u→,v) ∈ OUT(u) do
45           if d_u = w_e + d_v then
46               g_e^SP = 1;
47               δ_u = δ_u + 1;
48           end if
49       end for
50   end for
end rhG+^RR.
```

Figura 4.4: Pseudo-código do procedimento rhG_+^{RR}.

(linha 19). A seguir, é calculado o valor de $\bigtriangledown$, correspondendo ao total em que a distância do nó u irá decrescer (linha 21).

No laço das linhas 22-32, todos os nós do conjunto Q, excluindo o nó Q_0, têm suas distâncias decrescidas em $\bigtriangledown$ (linha 23). Além disso, as distâncias dos nós $u \in Q \setminus \{Q_0\}$ com o caminho mínimo ligando nós $v \notin Q$ são atualizadas. No laço das linhas 25-30, todos arcos partindo dos nós $u \in Q \setminus \{Q_0\}$ são consultados. Se um caminho mínimo ligando nós fora do conjunto Q é encontrado, a distância do nó u é atualizada (linha 27) e mais tarde (linha 31) o nó u é inserido na pilha H.

No laço das linhas 33-42, os nós u da pilha H são removidos um por um. Todos arcos $e = (\overrightarrow{s,u})$ inserindo em u são pesquisados. Caso o nó s tiver um caminho mínimo alternativo passando pelo nó u, sua distância é atualizada (linha 37) e a pilha H é ajustada (linha 38). No laço das linhas 43-50, restaura-se g^{SP} adicionando os arcos faltantes nos caminhos mínimos dos nós $u \in Q$.

4.3.2 Especialização do algoritmo G_+^{RR} para atualizar uma árvore de caminhos mínimos (T_+^{RR})

A maior diferença com G_+^{RR} é que T_+^{RR} é especializado para atualizar uma árvore de caminhos mínimos em vez de um grafo de caminhos mínimos. Em vez de atualizar g^{SP}, este algoritmo atualiza a árvore t^{SP}: cada nó u armazena um arco $a = (\overrightarrow{u,v})$ da lista de arcos saintes do nó u e pertencente a um caminho mínimo.

O pseudo-código do algoritmo padrão é descrito na Figura 4.5. Caso o arco $a = (\overrightarrow{u,v})$ não estiver na árvore de caminhos mínimos o algoritmo pára (linha 1). Caso contrário, os arcos saintes do nó u são consultados. Se um caminho mínimo alternativo for encontrado, t_u^{SP} é atualizado (linha 4) e o algoritmo pára (linha 5). Caso contrário, o conjunto Q é inicializado com o nó u (linha 8).

O laço nas linhas 9-22 identifica o conjunto Q de nos afetados. Para cada nó $u \in Q$, d_u recebe o valor ∞ (linha 10), e cada arco incidente $e = (\overrightarrow{s,u})$ é investigado. Caso e for o arco de saída do nó s armazenado na árvore de caminhos mínimos, isto é, $t_s^{SP} = e$, todos arcos de saída do nó s são consultados. Se um caminho mínimo alternativo for encontrado, t_s^{SP} é atualizado (linha 15) e os demais arcos partindo do nó s são consultados (linha 16). Caso contrário, se um caminho alternativo mínimo for encontrado, s é inserido no conjunto Q (linha 19).

```
procedure T_+^RR(a = (u→,v→), w, d, t^SP)
1     if t_u^SP ≠ a return;
2     for e = (u→,v→) ∈ OUT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = ∞;
11          for e = (s→,u→) ∈ IN(u) then
12                if t_s^SP = e do
13                      for a = (s→,v→) ∈ OUT(s) do
14                            if d_s = d_v + w_a then
15                                  t_s^SP = a;
16                                  break;
17                            end if
18                      end for
19                      if t_s^SP = e then Q = Q ∪ {s};
20                end if
21          end for
22    end for
23    for u ∈ Q do
24          for e = (u→,v→) ∈ OUT(u) do
25                if d_u > d_v + w_e then
26                      d_u = d_v + w_e;
27                      t_u^SP = e;
28                end if
29          end for
30          if d_u ≠ ∞ then InsertIntoHeap(H,u,d_u);
31    end for
32    while HeapSize(H) > 0 do
33          u = FindAndDeleteMin(H,d);
34          for e = (s→,u→) ∈ IN(u) do
35                if d_s > d_u + w_e then
36                      d_s = d_u + w_e;
37                      t_s^SP = e;
38                      AdjustHeap(H,s,d_s);
39                end if
40          end for
41    end while
end T_+^RR.
```

Figura 4.5: Pseudo-código do procedimento T_+^{RR}.

O laço nas linhas 23-31 atualiza as distâncias dos nós $u \in Q$ que possuem caminhos mínimos ligando nós fora do conjunto Q. Todos os arcos $e = (\overrightarrow{u,v})$, partindo dos nós $u \in Q$, são consultados. Se d_u decrescer, seu novo valor é atribuído (linha 26) e t_u^{SP} é atualizado (linha 27). Todos os nós $u \in Q$, cujas distâncias decresceram, são inseridos na pilha H (linha 30).

O laço das linhas 32-41 atualiza as distâncias dos nós de Q usando a pilha H. Os nós com distância mínima ao destino são removidos de H e todos arcos de entrada $e = (\overrightarrow{s,u})$ são consultados. Caso d_s possa ser decrescido, a nova distância é atribuída (linha 36), t_s^{SP} é atualizado (linha 37) e a pilha H é atualizada (linha 38).

A Figura 4.6 apresenta um grafo para ilustrar o funcionamento do algoritmo após a execução das primeiras 32 linhas de seu pseudo-código. Os grafos inicial e final são os mesmos apresentados na Figura 4.3. Observe que a distância dos nós $u \in Q$ não recebem o valor ∞, mas apenas são incrementadas do valor $\triangle = 2$. A atualização final destas distâncias é realizada no último laço **while** do algoritmo. Para este exemplo em específico, o algoritmo padrão insere todos os nós $u \in Q$ na pilha H, enquando que a variante `rh` insere apenas um nó em H.

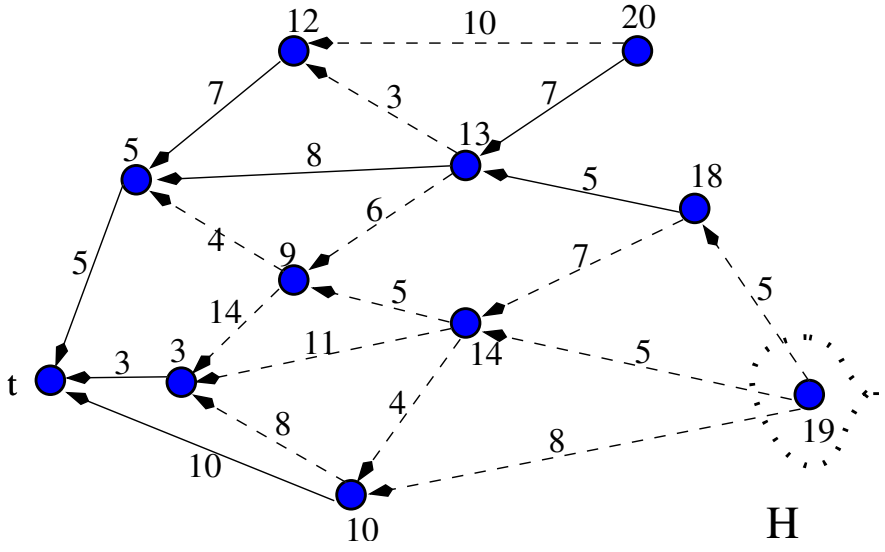

Figura 4.6: Exemplo ilustrativo da primeira fase do algoritmo rhG_+^{RR}.

Na variante com pilha reduzida, descrita na Figura 4.7, o valor de aumento $\triangle$ é um dado de entrada. As primeiras 22 linhas do pseudo-código são idênticas às do algoritmo padrão T_+^{RR}, com exceção das linha 10, que na variante `rh` o valor de d_u é aumentado em $\triangle$ em vez de receber o valor

```
procedure rhT+^RR(a = (u→,v→), w, d, t^SP, △)
1     if t_u^SP ≠ a return;
2     for e = (u→,v→) ∈ OUT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = d_u + △;
11          for e = (s→,u→) ∈ IN(u) do
12                if t_s^SP = e do
13                      for a = (s→,v→) ∈ OUT(s) do
14                            if d_s = d_v + w_a then
15                                  t_s^SP = A;
16                                  break;
17                            end if
18                      end for
19                      if t_s^SP = e then Q = Q ∪ {s};
20                end if
21          end for
22    end for
23    if △ = 1 then return;
24    dist = d_Q0;
25    for e = (Q0→,v→) ∈ OUT(Q0) do
26          if d_Q0 > d_v + w_e then
27                d_Q0 = d_v + w_e;
28                t_Q0^SP = e;
29          end if
30    end for
31    ▽ = dist − d_Q0;
32    for u ∈ Q \ Q0 do
33          d_u = d_u − ▽;
34          flag = 0;
35          for e = (u→,v→) ∈ OUT(u) do
36                if d_u > d_v + w_e then
37                      d_u = d_v + w_e;
38                      t_u^SP = e;
39                      flag = 1;
40                end if
41          end for
42          if flag = 1 then InsertIntoHeap(H, u, d_u);
43    end for
44    while HeapSize(H) > 0 do
45          u = FindAndDeleteMin(H, d);
46          for e = (s→,u→) ∈ IN(u) do
47                if d_s > d_u + w_e then
48                      d_s = d_u + w_e;
49                      t_s^SP = e;
50                      AdjustHeap(H, s, d_s);
51                end if
52          end for
53    end while
end rhT+^RR.
```

Figura 4.7: Pseudo-código do procedimento rhT_{+}^{RR}.

de ∞. Assim como em G_+^{RR}, a idéia é incrementar as distâncias dos nós $u \in Q$ por $\triangle$, e mais tarde decrementá-las em $\bigtriangledown$. No caso de incremento unitário ($\triangle = 1$) o algoritmo pára na linha 23. Nas linhas 24-31, o valor de $\bigtriangledown$ é calculado. Na linha 28 o arco partindo do nó u é armazenado em t^{SP}, visto que a árvore de caminhos mínimos pode ser restaurada enquanto a distância é atualizada, o que não acontece no algoritmo G_+^{RR}, no qual o último laço possui este objetivo.

O laço nas linhas 32-43 atualiza as distâncias dos nós $u \in Q \setminus \{Q_0\}$ com um caminho mínimo não passando pelo arco a. Lembrando que Q_0 é o primeiro nó inserido no conjunto Q (linha 8). Cada nó $u \in Q$, com exceção do nó Q_0, tem sua distância decrementada em $\bigtriangledown$ (linha 33). Cada arco $e = (\overrightarrow{u,v})$ partindo do nó u é consultado e se d_u pode ser decrescido, então d_u e t_u^{SP} são atualizados (linhas 37 e 38, respectivamente) e em seguida o nó u é inserido na pilha H (linha 42) .

O laço das linhas 44-53 é o mesmo laço das linhas 32-41 do pseudo-código do algoritmo T_+^{RR} (Figura 4.5).

4.3.3 O algoritmo de incremento para atualizar a árvore de caminhos mínimos proposta por King e Thorup (T_+^{KT})

A principal diferença com T_+^{RR} é que T_+^{KT} atualiza uma árvore especial de caminhos mínimos. Nesta árvore especial, t^{SP} armazena para cada nó u o primeiro arco $a = (\overrightarrow{u,v})$ da lista de arcos saintes de u, pertencente a um caminho mínimo. A vantagem de armazenar esta árvore particular é possibilitar que um caminho alternativo seja encontrado sem necessidade de explorar todos os arcos saintes do conjunto $\texttt{OUT}(u)$. Por exemplo, na Figura 4.1, t_2^{SP} deve ser igual a *2* para T^{KT}, mas ele pode ser igual a *2* ou *3* para T^{RR}. O pseudo-código deste algoritmo é descrito na Figura 4.8.

No caso em que $a \notin t^{SP}$, o algoritmo pára (linha 1). Caso contrário, a existência de um caminho mínimo alternativo é verificado consultando-se os arcos da lista de arcos saintes do nó u, iniciando-se na primeira posição após o arco a. No pseudo-código da Figura 4.8, denota-se por $\texttt{OUT}^{KT}(u) \subseteq \texttt{OUT}(u)$ o conjunto de arcos localizados após t_u^{SP} na lista de arcos saintes do nó u. Se outro caminho mínimo for encontrado, o arco e é armazenado em t_u^{SP} (linha 4) e o algoritmo pára (linha 5). Caso contrário, o conjunto Q é inicializado com o nó u (linha 8).

O laço das linhas 9-22 identifica os demais nós afetados de t^{SP} e os adiciona no conjunto Q. Para cada nó $u \in Q$, o valor ∞ é atribuído à sua distância (linha 10), e todos os arcos $e = (\overrightarrow{s,u})$ incidentes

```
procedure T_+^KT(a = (u→,v→), w, d, t^SP)
1     if t_u^SP ≠ a then return;
2     for e = (u→,v→) ∈ OUT^KT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = ∞;
11          for e = (s→,u→) ∈ IN(u) do
12                if t_s^SP = e do
13                      for a = (s→,v→) ∈ OUT^KT(s) do
14                            if d_s = d_v + w_a then
15                                  t_s^SP = a;
16                                  break;
17                            end if
18                      end for
19                      if t_s^SP = e then Q = Q ∪ {s};
20                end if
21          end for
22    end for
23    for u ∈ Q do
24          for e = (u→,v→) ∈ OUT(u) do
25                if d_u > d_v + w_e then
26                      d_u = d_v + w_e;
27                      t_u^SP = e;
28                end if
29          end for
30          if d_u ≠ ∞ then InsertIntoHeap(H, u, d_u);
31    end for
32    while HeapSize(H) > 0 do
33          u = FindAndDeleteMin(H, d);
34          for e = (s→,u→) ∈ IN(u) do
35                if d_s > d_u + w_e then
36                      d_s = d_u + w_e;
37                      t_s^SP = e;
38                      AdjustHeap(H, s, d_s);
39                end if
40                else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
41          end for
42    end while
end T_+^KT.
```

Figura 4.8: Pseudo-código do procedimento T_+^{KT}.

no nó u são investigados. Em caso $t_s^{SP} = e$, os arcos do conjunto $\texttt{OUT}^{KT}(s)$ são consultados. Se o nó s tiver um caminho mínimo alternativo, t_s^{SP} é atualizado (linha 15) e os arcos subsequentes no conjunto $\texttt{OUT}^{KT}(u)$ não são pesquisados (linha 16). Caso contrário, se um caminho mínimo alternativo para o nó s não for encontrado, o nó s é inserido no conjunto Q (linha 19).

O laço das linhas 23-31 consulta todos arcos e partindo de cada nó $u \in Q$. As distâncias e t^{SP} dos nós $u \in Q$ são atualizadas (linhas 26 e 27) e t_u^{SP} é atualizado com e (linha 27). Caso d_u for decrementado, o nó u é inserido na pilha H (linha 30). O objetivo principal deste laço é atualizar as distâncias dos nós $u \in Q$ os quais têm um caminho menor ligando nós $v \notin Q$.

O laço das linhas 32-42 atualiza as distâncias dos nós em Q usando a pilha H. O laço é idêntico ao laço das linhas 32-41 do algoritmo T_+^{RR}. Além disso, em caso o arco $e = (\overrightarrow{s,u})$ esteja no caminho mínimo, isto é, $d_s = d_u + w_e$, e sua posição na lista de arcos saintes for anterior a informação atual armazenada, isto é, $t_s^{SP} > e$, então o valor de t_s^{SP} é atualizado (linha 40).

A variante rhT_+^{KT} é executada em duas fases. A primeira fase identifica o conjunto Q, o qual contém os nós afetados. A segunda fase atualiza as distâncias dos nós $u \in Q$ e restaura a árvore de caminhos mínimos t^{SP}.

A primeira fase não está presente no pseudo-código, pois é idêntica às primeiras 22 linhas do `std` T_+^{KT}. A única diferença é que em rhT_+^{KT} a distância do nó u (linha 10) é incrementado em $\triangle$, enquanto que em T_+^{KT} d_u recebe valor ∞. Figura 4.9 apresenta o pseudo-código da segunda fase $rhT_+^{KT}Ph2$ do algoritmo rhT_+^{KT}.

O laço das linhas 1-11 é somente executado no caso de incremento unitário. Neste caso, os nós afetados já têm suas distâncias incrementadas por 1 na primeira fase do algoritmo. A árvore de caminhos mínimos t^{SP} somente precisa ser restaurada para caracterizar a árvore particular. Denota-se por $\texttt{OUT}^*(u) \subseteq \texttt{OUT}(u)$ o conjunto de arcos partindo do nó u localizados anteriormente ao arco e, isto é $\texttt{OUT}^*(u) = OUT(u) \setminus \texttt{OUT}^{KT}(u) \setminus \{t_u^{SP}\}$. Para cada nó $u \in Q$, o conjunto de arcos saintes $\texttt{OUT}^*(u)$ são consultados. Caso um arco $e = (\overrightarrow{u,v})$ estiver na árvore de caminhos mínimos, t_u^{SP} é atualizado (linha 5) e o algoritmo pára (linha 10). Nota-se que no caso de incremento unitário, a variante `rh` rhT_+^{KT} pára (linha 10) sem usar pilhas.

Nas linhas 12-19, cada arco $e = (\overrightarrow{Q_0,v})$ partindo do nó Q_0, o nó destino do arco a o qual teve seu peso originalmente decrementado, é consultado. O laço é executado no sentido reverso para

```
procedure rhT+KT Ph2(a = (u→,v→), w, d, tSP, △)
1    if △ = 1 then
2        for u ∈ Q do
3            for e = (u→,v→) ∈ OUT*(u) do
4                if d_u = d_v + w_e then
5                    t_u^SP = e;
6                    break;
7                end if
8            end for
9        end for
10       return;
11   end if
12   dist = d_Q0;
13   backward for e = (Q0→,v→) ∈ OUT(Q0) do
14       if d_Q0 ≥ d_v + w_e then
15           d_Q0 = d_v + w_e;
16           t_Q0^SP = e;
17       end if
18   end for
19   ▽ = dist − d_Q0;
20   for u ∈ Q \ Q0 do
21       d_u = d_u − ▽;
22       flag = 0;
23       backward for e = (u→,v→) ∈ OUT(u) do
24           if d_u > d_v + w_e then
25               d_u = d_v + w_e;
26               t_u^SP = e;
27               flag = 1;
28           end if
29           else if d_u = d_v + w_e then t_u^SP = e;
30       end for
31       if flag = 1 InsertIntoHeap(H, u, d_u);
32   end for
33   while HeapSize(H) > 0 do
34       u = FindAndDeleteMin(H, d);
35       for e = (s,u→) ∈ IN(u) do
36           if d_s > d_u + w_e then
37               d_s = d_u + w_e;
38               t_s^SP = e;
39               AdjustHeap(H, s, d_s);
40           end if
41           else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
42       end for
43   end while
end rhT+KT.
```

Figura 4.9: Pseudo-código da segunda fase do algoritmo rhT_{+}^{KT}.

permitir uma atualização correta da árvore especial t^{SP} sem precisar testar se $t^{SP}_{Q_0} > e$. No caso em que $d_{Q_0} \geq d_v + w_e$, então d_{Q_0} e $t^{SP}_{Q_0}$ são atualizados. O valor de $\bigtriangledown$ é calculado na linha 19.

O laço das linhas 20-32 têm o objetivo de atualizar as distâncias e t^{SP} dos nós $u \in Q$ os quais têm um caminho mínimo ligando nós fora do conjunto Q. Novamente, o laço é executado em sentido reverso para evitar testar se $t^{SP}_s > e$ na linha 29.

O laço das linhas 33-43 é idêntico ao laço das linhas 32-42 da Figura 4.8.

4.3.4 Algoritmo de incremento proposto por Demetrescu para atualizar uma árvore de caminhos mínimos (T^D_+)

Assim com em T^{RR}_+ e T^{KT}_+, T^D_+ também atualiza uma árvore de caminhos mínimos. A principal diferença é que T^D_+ usa um mecanismo mais simples para identificar o conjunto de nós afetados Q^D. Entretanto, $|Q^{RR}| = |Q^{KT}| \leq |Q^D|$. Supondo que o arco $a = (\overrightarrow{u,v})$ teve seu peso incrementado em $\bigtriangleup$, o conjunto Q^{RR} é composto somente pelos nós que têm o arco a em todos seus caminhos mínimos, enquanto o conjunto Q^D é composto por estes nós e também alguns dos nós que tem um caminho alternativo sem passar pelo arco a. A idéia é que em grafos com poucos nós (ou nenhum) com mais de um caminho mínimo, $|Q^D| \approx |Q^{RR}|$, com a vantagem de Q^D ser identificado usando um mecanismo mais simples.

O pseudo-código deste algoritmo é descrito na Figura 4.10. Este pseudo-código é idêntico ao algoritmo T^{RR}_+, diferindo somente no laço que faz a identificação do conjunto Q. O laço das linhas 9-14 identifica os nós *afetados* de forma mais simples de como é feito nos algoritmos de incremento descritos nas subseções anteriores. Entretanto, alguns dos nós $u \in Q$ podem não terem sido afetados, mas serem tratados como se tivesem. Para cada nó $u \in Q$, d_u recebe o valor ∞ (linha 10), e cada arco incidente e é consultado. No caso em que e é o arco sainte do nó u na árvore de caminhos mínimos, isto é $t^{SP}_s = e$, o nó s é inserido no conjunto Q, mesmo se ele tiver um caminho mínimo alternativo ao nó destino. A idéia deste algoritmo é não gastar tempo procurando por um caminho alternativo. Esta estratégia tem vantagem no caso em que os pesos dos arcos são escolhidos num intervalo grande e a probabilidade de empates é pequena.

Na variante com pilha reduzida, descrita na Figura 4.11,

o valor de aumento $\bigtriangleup$ é um parâmetro fornecido. Este algoritmo é idêntico ao rhT^{RR}_+, somente

```
procedure T_+^D(a = (u→,v→), w, d, t^SP)
1     if t_u^SP ≠ a return;
2     for e = (u→,v→) ∈ OUT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = ∞;
11          for e = (s→,u→) ∈ IN(u) do
12                if t_s^SP = e then Q = Q ∪ {s};
13          end for
14    end for
15    for u ∈ Q do
16          for e = (u→,v→) ∈ OUT(u) do
17                if d_u > d_v + w_e then
18                      d_u = d_v + w_e;
19                      t_u^SP = e;
20                end if
21          end for
22          if d_u ≠ ∞ then InsertIntoHeap(H, u, d_u);
23    end for
24    while HeapSize(H) > 0 do
25          u = FindAndDeleteMin(H, d);
26          for e = (s→,u→) ∈ IN(u) do
27                if d_s > d_u + w_e then
28                      d_s = d_u + w_e;
29                      t_s^SP = e;
30                      AdjustHeap(H, s, d_s);
31                end if
32          end for
33    end while
end T_+^D.
```

Figura 4.10: Pseudo-código do procedimento T_+^D.

diferindo em como eles identificam o conjunto Q. Ainda, a execução do algoritmo rhT_+^D não pára no caso de incremento unitário, assim como rhT_+^{RR} na linha 23 do pseudo-código da Figura 4.7. A variante `rh` identifica o conjunto Q da mesma forma que o algoritmo `std`. Note que no caso de aumento unitário do peso somente os nós não afetados do conjunto Q são inseridos na pilha H pelo rhT_+^D, enquanto rhT_+^{RR} e rhT_+^{KT} não inserem nenhum nó em H.

A Figura 4.12 apresenta um grafo ilustrativo da operação da primeira fase dos algoritmos T_+^D e rhT_+^D. O tamanho do conjunto Q é igual para ambos os casos, mas o conjunto de nós inseridos na pilha é em geral bem menor para o algoritmo rhT_+^D.

```
procedure rhT+^D(a = (u→,v→), △, w, d, t^SP)
1     if t_u^SP ≠ a return;
2     for e = (u→,v→) ∈ OUT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = d_u + △;
11          for e = (s→,u→) ∈ IN(u) do
12                if t_s^SP = e then Q = Q ∪ {s};
13          end for
14    end for
15    dist = d_Q0;
16    for e = (Q0→,v→) ∈ OUT(Q0) do
17          if d_Q0 > d_v + w_e then
18                d_Q0 = d_v − w_e;
19                t_Q0^SP = e;
20          end if
21    end for
22    ▽ = dist − d_Q0;
23    for u ∈ Q \ Q0 do
24          d_u = d_u − ▽;
25          flag = 0;
26          for e = (u→,v→) ∈ OUT(u) do
27                if d_u > d_v + w_e then
28                      d_u = d_v + w_e;
29                      t_u^SP = e;
30                      flag = 1;
31                end if
32          end for
33          if flag = 1 then InsertIntoHeap(H, u, d_u);
34    end for
35    while HeapSize(H) > 0 do
36          u = FindAndDeleteMin(H, d);
37          for e = (s→,u→) ∈ IN(u) do
38                if d_s > d_u + w_e then
39                      d_s = d_u + w_e;
40                      t_s^SP = e;
41                      AdjustHeap(H, s, d_s);
42                end if
43          end for
44    end while
end T+^D.
```

Figura 4.11: Pseudo-code of procedure rhT_+^D.

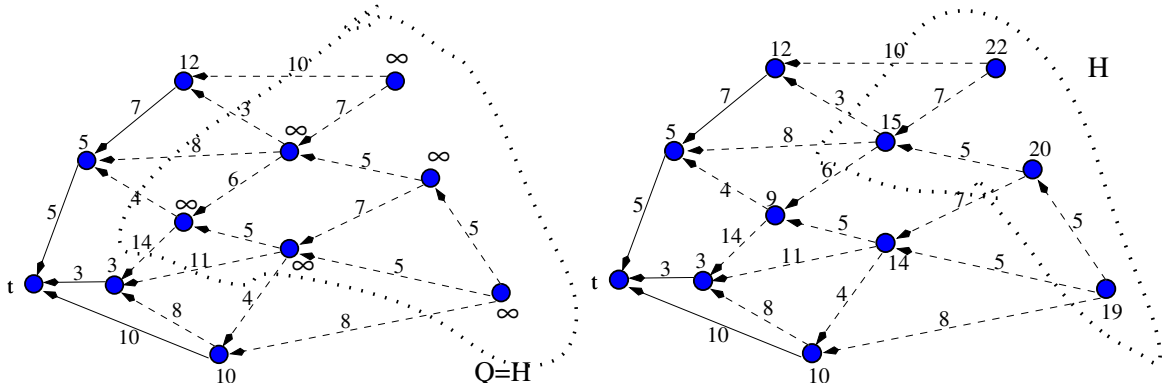

Figura 4.12: Exemplo ilustrativo da primeira fase dos algoritmos T_+^D (à esquerda) e rhT_+^D (à direita).

4.4 Versões padrão e com pilha reduzida dos algoritmos de incremento

Nesta seção são feitas algumas observações da comparação entre as versões padrão e de pilha reduzida dos algoritmos de atualização dos caminhos mínimos após o incremento de um peso. As figuras 4.13 e 4.14 mostram os tamanhos das pilhas usadas pelos algoritmos de incremento discutidos na seção anterior.

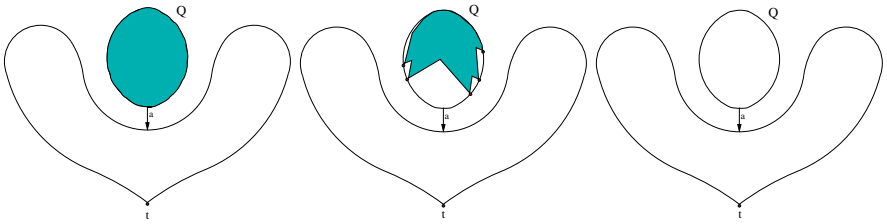

Figura 4.13: Tamanhos das pilhas usadas pelos algoritmos de incremento G_+^{RR}, T_+^{RR} e T_+^{KT} e suas variantes de pilha reduzida. O grafo da esquerda representa o algoritmo `std`. O grafo do meio representa as variantes `rh` para incremento aleatório de pesos. Na direita, para as variantes `rh` com incremento unitário de peso, a pilha é vazia.

Como pode-se ver nas figuras, no caso de incremento unitário para os algoritmos G_+^{RR}, T_+^{RR}, e T_+^{KT}, a pilha H é vazia, enquanto que para T_+^D alguns nós $u \in Q$ são inseridos em H. Estes nós são aqueles não afetados, mas tratados como se fossem. Devido a isso, comparando qualquer caso de `std`, `rh` e `incremento unitário`, $Q^D \supseteq Q^{RR} = Q^{KT}$. Para todos estes algoritmos, $H^{std} \supseteq H^{rh}_{random} \supseteq H^{rh}_{unit}$.

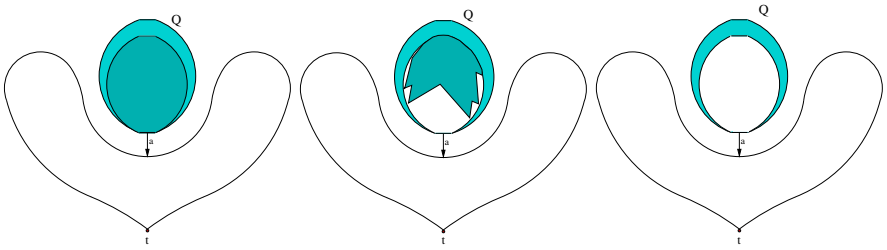

Figura 4.14: Tamanhos das pilhas usadas pelos algoritmos de incremento T_+^D e rhT_+^D. O grafo da esquerda representa o algoritmo `std`. O grafo do meio representa as variantes `rh` para incremento aleatório de pesos. Na direita, para as variantes `rh` com incremento unitário de peso, a pilha contém somente os nós não afetados que foram inseridos no conjunto Q.

Em cada grafo dessas figuras, o conjunto Q de nós afetados é ligado ao restante do grafo pelo arco a, o qual teve seu peso originalmente incrementado. O nó t é o destino. A parte sombreada do conjunto Q é composto pelos nós que foram inseridos na pilha H.

Em termos de memória, os algoritmos de pilha reduzida não requerem espaço adicional em suas implementações.

4.5 Algoritmos de decremento

Nesta seção, são descritos três algoritmos para atualizar um grafo após o decremento de um arco. Ainda, a idéia de redução de pilha é aplicada a todos eles. O peso pode ser decrescido em qualquer valor. No texto, esses algoritmos são referenciados como G_-^{RR}, T_-^{RR} e T_-^{KT}. Os algoritmos que implementam a idéia de pilha-reduzida são denominados rhG_-^{RR}, rhT_-^{RR} e rhT_-^{KT}.

Considere que $a = (\overrightarrow{u,v}) \in E$ seja o arco cujo peso foi decrescido. Para estes algoritmos, Q é o conjunto de nós que possuem o arco a em pelo menos um caminho mínimo. Outro conjunto, U, é composto pelo nós cujos caminhos mínimos originalmente não continham o arco a, mas passaram a ter após o decremento do peso w_a. Os algoritmos `std` inserem todos os nós de $Q \cup U$ em H. A idéia de pilha-reduzida é usada para evitar o uso de pilhas para atualizar nós contidos em Q, e apenas usá-la para atualizar os nós contidos em U.

Os algoritmos têm duas fases. Inicialmente, os caminhos mínimos são atualizados considerando nós de Q. A seguir, eles são atualizados considerando os nós pertencentes a U. O valor de decremento $\bigtriangledown$ é calculado antes do conjunto Q ser identificado. Visto que o conjunto Q contém

```
procedure G_-^RR(a = (u→,v), w, d_V, δ_V, g^SP)
1    if d_u < d_v + w_a then return;
2    if d_u = d_v + w_a then
3        g_a^SP = 1;
4        δ_u = δ_u + 1;
5        return;
6    end if
7    d_u = d_v + w_a;
8    InserIntoHeap(H, u, d_u);
9    while HeapSize(H) > 0 do
10       u = FindAndDeleteMin(H, d);
11       δ_u = 0;
12       forall e = (u→,v) ∈ OUT(u) do
13           if d_u = d_v + w_e then
14               δ_u = δ_u + 1;
15               g_e^SP = 1;
16           end if
17           else g_e^SP = 0;
18       end forall
19       forall e = (s→,u) ∈ IN(u) do
20           if d_s > d_u + w_e then
21               d_s = d_u + w_e;
22               AdjustHeap(H, s, d_s);
23           end if
24           else if g_e^SP = 0 and d_s = d_u + w_e then
25               g_e^SP = 1;
26               δ_s = δ_s + 1;
27           end if
28       end forall
29   end while
end G_-^RR.
```

Figura 4.15: Pseudo-código do procedimento G_{-}^{RR}.

todos os nós que possuem no mínimo um caminho mínimo usando o arco a, a distância destes nós é decrescida em exatamente $\bigtriangledown$, sem necessidade de usar pilhas para isso. Entretanto, pilhas são necessárias para calcular as distâncias dos nós $u \in U$. Nas próximas seções, serão apresentados os códigos das versões `std` e `rh` dos algoritmos de decremento.

4.5.1 Algoritmo de decremento proposto por Ramalingam e Reps para atualizar um grafo de caminhos mínimos (G_{-}^{RR})

O algoritmo de decremento do Ramalingam e Reps aplicado a grafos [RR96a] atualiza o grafo de caminhos mínimos considerando o decremento do peso de um arco em qualquer valor. O pseudo-código deste procedimento é mostrado na Figura 4.15.

Lembra-se que $a = (\overrightarrow{u,v}) \in E$ é o arco que teve seu peso originalmente decrementado. Se d_u não decrescer, o algoritmo pára (linha 1). Caso contrário, se o nó u tiver um caminho mínimo

alternativo passando pelo arco a, a é inserido no grafo g^{SP} (linha 3), a variável δ_u é atualizada (linha 4) e o algoritmo pára (linha 5). Caso contrário, a distância d_u é atualizada (linha 7) e a pilha H é inicializada com o nó u (linha 8).

No laço das linhas 9-29, os nós são removidos um a um de H, do mais próximo ao mais distante do nó destino, e suas respectivas distâncias e arcos entrantes e saintes são atualizados em g^{SP}. O nó u é removido da pilha H (linha 10) e δ_u é zerado (linha 11). Todos os arcos saintes do nó u são consultados e inseridos em g^{SP} (linha 15) caso eles tenham distância mínima do destino. Caso contrário, $g_e^{SP} = 0$ (linha 17). A seguir, cada arco entrante $e = (\overrightarrow{s,u})$ em um nó $u \in Q$ é consultado. Se d_s puder diminuir, a nova distância é calculada (linha 21) e H ajustada (linha 22). Se o arco e não estiver no grafo de caminhos mínimos atual e $d_s = d_u + w_e$, então o laço das linhas linhas 24-27 insere o arco e em g^{SP} (linha 25) e atualiza δ_s (linha 26).

A Figura 4.16 apresenta um exemplo ilustrativo do funcionamento do algoritmo G_-^{RR}. Considerando o decremento do arco em destaque, o algoritmo identifica os conjuntos Q e U de nós. A pilha é composta por todos nós $u \in (U \cup Q)$.

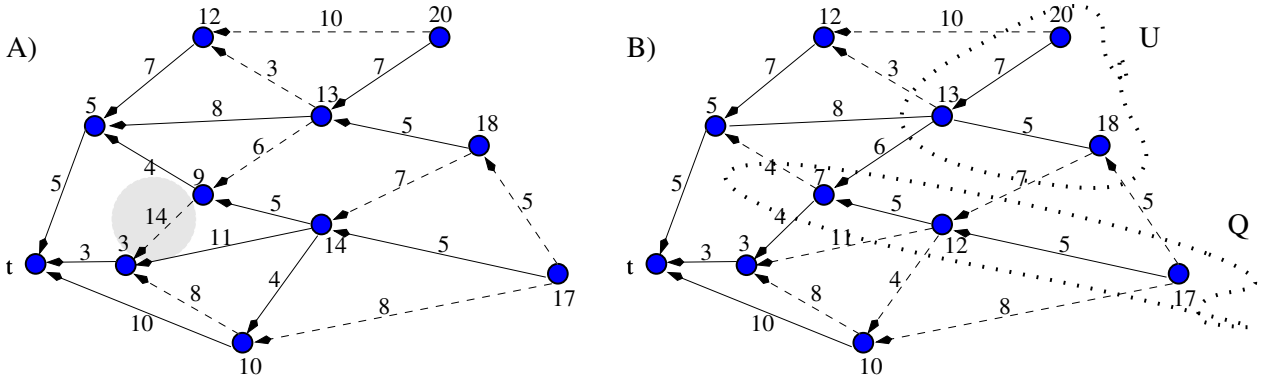

Figura 4.16: Exemplo ilustrativo do funcionamento do algoritmo G_-^{RR}.

A seguir a técnica de pilha-reduzida é aplicada ao algoritmo de Ramalingam e Reps.

A primeira fase do rhG_-^{RR} é descrita na Figura 4.17. As primeiras 6 linhas deste pseudo-código são idênticas às primeiras seis linhas do algoritmo clássico. Na linha 7 o valor $\bigtriangledown$, do qual a distância do nó u será decrementada, é calculado. A distância do nó u é decrementado em $\bigtriangledown$ (linha 8), ele é inserido no conjunto Q (linha 9) e $degree_u$ é inicializado (linha 10).O vetor $degree$ é usado para evitar que um nó u seja inserido em Q mais que uma vez. Além disso, $degree$ é usado

para identificar os nós de Q tendo caminhos alternativos não passando por a.

O laço das linhas 11-26 identifica os nós $u \in Q$, fazendo uso do vetor *degree*, e atualiza o grafo g^{SP} no caso de decremento unitário. Para cada nó $u \in Q$, todos os arcos incidentes $e = (\overrightarrow{s,u})$ são verificados. Se o arco e estiver no grafo de caminhos mínimos e se $degree_s = 0$, então a distância do nó s é decrementada em $\bigtriangledown$ (linha 15), s é inserido no conjunto Q (linha 16) e $degree_s$ recebe o valor $\delta_s - 1$ (linha 17). Caso contrário, se $e \in g^{SP}$ mas $degree_s > 0$, o valor de $degree_s$ é decrementado em uma unidade (linha 19). No caso de decremento unitário g^{SP} e δ são atualizadas nas linhas 22 e 23, respectivamente.

Se um nó $u \in Q$ tiver um ou mais caminhos alternativos passando pelo arco a, estes caminhos não são mínimos após a redução do peso do arco a e devem ser removidos do grafo de caminhos mínimos. A existência de um caminho mínimo alternativo é determinado fazendo uso do vetor *degree*. O laço nas linhas 27-37 remove estes caminhos analisando cada nó $u \in Q$, um de cada vez. Se um ou mais caminhos alternativos partindo de u forem detectados, então $degree_u$ é zerado nas linhas 30-35 e todos os arcos $e = (\overrightarrow{u,v})$ saintes de u são verificados. Se o arco e não mais pertencer a um caminho mínimo, ele é removido (linha 32) e o número de arcos saintes de u (δ_u) é atualizado (linha 33).

No caso de decremento unitário, o algoritmo pára na linha 38.

No laço das linhas 39-52, todos os arcos $e = (\overrightarrow{s,u})$ incidentes em $u \in Q$ são verificados. O teste na linha 41 permite que o algoritmo não consulte arcos $e = (\overrightarrow{s,u})$ com $s \in Q$ e $e \in g^{SP}$. No processo de projeto do algoritmo, escolheu-se manter um vetor indicador dos nós pertencentes ao conjunto Q, o que torna possível evitar a consulta do arcos $e = (\overrightarrow{s,u})$ com $s \in Q$ e $e \notin g^{SP}$, pois qualquer ganho associado com seu uso não parece justificar o esforço computacional gasto para tanto. Nas linhas 42-45, se o nó s tiver um caminho mínimo alternativo passando por e, e é inserido em g^{SP} (linha 43) e δ é atualizado (linha 44). Se a distância do nó s diminui considerando o caminho passando pelo arco a, o valor de d_s é atualizado (linha 47) e H ajustada (linha 48). É importante salientar que somente os nós não pertencentes ao conjunto Q são inseridos na pilha. Considerando o exemplo da Figura 4.16, somente os nós $u \in U$ são inseridos na pilha H.

Lembrando que a fase `1` do algoritmo identifica o conjunto Q, o qual contém todos os nós que possuem pelo menos um caminho mínimo incluindo o arco a e atualiza a parte do grafo composta

```
procedure rhG_-^RR Ph1(a = (u→,v), w_E, d_V, δ_V, g^SP)
1    if d_u < d_v + w_a then return;
2    if d_u = d_v + w_a then
3        g_a^SP = 1;
4        δ_u = δ_u + 1;
5        return;
6    end if
7    ▽ = d_u − d_v − w_a;
8    d_u = d_u − ▽;
9    Q = {u};
10   degree_u = δ_u − 1;
11   forall u ∈ Q do
12       forall e = (s→,u) ∈ IN(u) do
13           if g_e^SP = 1 then
14               if degree_s = 0 then
15                   d_s = d_s − ▽;
16                   Q = Q ∪ {s};
17                   degree_s = δ_s − 1;
18               end if
19               else degree_s = degree_s − 1;
20           end if
21           else if ▽ = 1 and d_s = d_u + w_e then
22               g_e^SP = 1;
23               δ_s = δ_s + 1;
24           end if
25       end forall
26   end forall
27   forall u ∈ Q do
28       if degree_u > 0 then
29           degree_u = 0;
30           forall e = (u→,v) ∈ OUT(u) do
31               if g_e^SP = 1 and d_u < d_v + w_e then
32                   g_e^SP = 0;
33                   δ_u = δ_u − 1;
34               end if
35           end forall
36       end if
37   end forall
38   if ▽ = 1 return;
39   forall u ∈ Q do
42       forall e = (s→,u) ∈ IN(u) do
41           if g_e^SP = 0 then
42               if d_s = d_u + w_e then
43                   g_e^SP = 1;
44                   δ_s = δ_s + 1;
45               end if
46               else if d_s > d_u + w_e then
47                   d_s = d_u + w_e;
48                   AdjustHeap(H, s, d_s);
49               end if
50           end if
51       end forall
52   end forall
end rhG_-^RR Ph1.
```

Figura 4.17: Pseudo-código do procedimento $rhG_{-}^{RR}Ph1$.

por estes nós e os arcos entrantes e saintes. Além disso, todos nós $s \in U$ com um caminho mínimo ligando o conjunto Q é identificado e se d_s for reduzido, o nó s é inserido na pilha H. A segunda fase atualiza o restante do grafo afetado, isto é, os nós que agora têm um caminho mínimo através de a, que teve seu peso inicialmente decrescido, mas não têm um arco ligando diretamente um nó de Q. Este procedimento é idêntico ao pseudo-código das linhas 9-29 da Figura 4.15.

4.5.2 Especialização do algoritmo G_{-}^{RR} para atualizar uma árvore de caminhos mínimos (T_{-}^{RR})

O algoritmo T_{-}^{RR} é uma especialização do algoritmo do Ramalingam e Reps (G_{-}^{RR}) restrito à atualização de um árvore de caminhos mínimos. Um algoritmo similar foi proposto por Frigioni, Marchetti-Spaccamela e Nanni [FMSN00]. O algoritmo T_{-}^{RR} é descrito na Figura 4.18.

```
procedure T_-^RR(a = (u→,v), w, d, δ, g^SP)
1     if d_u ≤ d_v + w_a then return;
2     d_u = d_v + w_a;
3     t_u^SP = a;
4     InserIntoHeap(H, u, d_u);
5     while HeapSize(H) > 0 do
6           u =FindAndDeleteMin(H, d);
7           forall e = (s→,u) ∈ IN(u) do
8                 if d_s > d_u + w_e then
9                       d_s = d_u + w_e;
10                      t_s^SP = e;
11                      AdjustHeap(H, s, d_s);
12                end if
13          end forall
14    end while
end T_-^RR.
```

Figura 4.18: Pseudo-código do procedimento T_{-}^{RR}.

Considere $a = (\overrightarrow{u,v})$ o arco cujo peso foi decrescido. Caso a distância do nó u não mude, o algoritmo pára (linha 1). Caso contrário, d_u e t_u^{SP} são atualizados nas linhas 2 e 3, respectivamente, e a pilha H é inicializada com o nó u (linha 4). Os nós são removidos um a um da pilha H (linha 6) e cada arco incidente $e = (\overrightarrow{s,u})$ em u é investigado. Se d_s puder diminuir, d_s e t_s^{SP} são atualizados nas linhas 9 e 10, respectivamente e H ajustada (linha 11).

A Figura 4.19 apresenta o pseudo-código para a variante de pilha-reduzida rhT_{-}^{RR}. O algoritmo inicialmente identifica os conjuntos Q e U. A seguir, os nós em U são inseridos na pilha e o restante do grafo é atualizado. Antes de tudo, o valor decremento $\bigtriangledown$ é calculado. Se $\bigtriangledown \leq 0$ o algoritmo

```
procedure rhT_-^RR(a = (u→,v→), w_E, d_V, δ_V, g^SP)
1     ▽ = d_u − d_v − w_a;
2     if ▽ ≤ 0 then return;
3     d_u = d_u − ▽;
4     t_u^SP = a;
5     Q = {u};
6     U = ∅;
7     forall u ∈ Q do
8           forall e = (s→,u→) ∈ IN(u) do
9                 diff = d_s − d_u − w_e;
10                if diff > 0 then
11                      if diff = ▽ then
12                            d_s = d_s − ▽;
13                            t_s^SP = e;
14                            Q = Q ∪ {s};
15                            maxdiff_s = 0;
16                      end if
17                      else if diff > maxdiff_s then
18                            if maxdiff_s = 0 then U = U ∪ {s};
19                            maxdiff_s = diff;
20                            t_s^SP = e;
21                      end if
22                end if
23          end forall
24    end forall
25    if ▽ = 1 then return;
26    forall u ∈ U do
27          if maxdiff_u > 0 then
28                d_u = d_u − maxdiff_u;
29                maxdiff_u = 0;
30                InserIntoHeap(H,u,d_u);
31          end if
32    end forall
33    while HeapSize(H) > 0 do
34          u = FindAndDeleteMin(H,d);
35          forall e = (s→,u→) ∈ IN(u) do
36                if d_s > d_u + w_e then
37                      d_s = d_u + w_e;
38                      t_s^SP = e;
39                      AdjustHeap(H,s,d_s);
40                end if
41          end forall
42    end while
end rhT_-^RR.
```

Figura 4.19: Pseudo-código do procedimento rhT_{-}^{RR}.

pára (linha 2) visto que o valor de $\bigtriangledown$ não é suficiente para decrementar a distância do nó u. Caso contrário, o arco a passa a fazer parte de um novo caminho mínimo do nó u ao destino. A distância d_u e t_u^{SP} são atualizados (linhas 3 e 4, respectivamente), t_u^{SP} atualizado (linha 4) e u é inserido no conjunto Q (linha 5). O conjunto U é inicializado vazio na linha 6.

O vetor *maxdiff* é usado para armazenar o decremento máximo dos nós $u \in U$. O laço das linhas 7-24 identifica os nós do conjunto Q e U. Para cada nó $u \in Q$, todos os arcos saintes $e = (\overrightarrow{s,u})$ são verificados. O total *diff* que d_s diminui quando usando o caminho passando pelo arco e, é calculado na linha 9. Somente valores positivos de *diff* são considerados (linha 10). Se $diff = \bigtriangledown$, então o nó s possui um caminho mínimo alternativo passando pelo arco a e u é identificado como pertencente ao conjunto Q. Uma vez que o nó s é identificado como pertencendo a Q, sua distância é atualizada (linha 12), t_u^{SP} atualizado (linha 13), o nó s é inserido em Q e seu valor de *maxdiff* é zerado (linha 15). Caso contrário, se s não estiver em U, então ele é inserido (linha 18). Se *diff* for maior que $maxdiff_s$, $maxdiff_s$ é atualizado com o valor de *diff* (linha 19) e t_u^{SP} é atualizado. Em caso de decremento unitário, isto é, $\bigtriangledown = 1$, o algoritmo pára na linha 25. Caso contrário, os nós $u \in U$ com valor positivo de *maxdiff* são investigados. Observa-se que os nós u que foram inicialmente adicionados ao conjunto U podem mais tarde ser identificados como pertencentes a Q. Neste caso, o nó u é inserido no conjunto Q, sem ser removido do conjunto U, visto que o esforço para removê-lo pode ser caro computacionalmente e estes nós podem ser identificados mais tarde usando o vetor *maxdiff*. Para cada nó u anteriormente inserido no conjunto U, o algoritmo faz uso da variável

$$maxdiff_u = \begin{cases} 0, & \text{se } u \in U \cap Q; \\ maxdiff_u > 0, & \text{se } u \in U \setminus (U \cap Q), \end{cases}$$

para determinar se o nó $u \in U \setminus (U \cap Q)$. Se $maxdiff_u > 0$, então $maxdiff_u$ contém o valor pelo qual a distância do nó u será reduzida. Nós $u \in U \cap Q$ são descartados no teste da linha 27. Caso $maxdiff_u > 0$, a distância do nó u é atualizada (linha 28), seu valor de *maxdiff* é zerado (linha 29) e o nó u é inserido na pilha H (linha 30). O laço nas linhas 33-42 atualiza t^{SP} e as distâncias dos nós $u \in U$. Nós são removidos um a um da pilha H e cada arco entrante $e = (\overrightarrow{s,u})$ é verificado. Caso d_s possa diminuir, d_s e t_s^{SP} são atualizados (linhas 37 e 38, respectivamente) e H é atualizado (linha 39).

4.5.3 Algoritmo de decremento para atualizar a árvore de caminhos mínimos proposta por King e Thorup (T_-^{KT})

Esta seção introduz um algoritmo para atualização de uma árvore de caminhos mínimos fazendo uso da árvore particular proposta por King e Thorup [KT01]. A diferença com T_-^{RR} é que T_-^{KT} atualiza uma árvore particular de caminhos mínimos. O algoritmo T_-^{KT} armazena, para cada nó, o primeiro arco de sua lista de arcos saintes pertencentes a um caminho mínimo. Devido a esta simples diferença, ambos algoritmos são muito parecidos. Duas variantes são apresentadas: uma utilizando e outra não utilizando a técnica de pilha-reduzida.

O pseudo-código do algoritmo T_-^{KT} é mostrado na Figura 4.20. O algoritmo pára se d_u não diminuir (linha 1). Caso contrário, se o nó u possuir um caminho alternativo mínimo através de a, então t^{SP} é atualizado se necessário (linha 3) e o algoritmo pára (linha 4). As linhas 6-19 são idênticas às linhas 2-14 do pseudo-código do algoritmo T_-^{RR} na Figura 4.20. A linha 17 é necessária para este algoritmo visto que no caso em que houver um caminho mínimo alternativo a partir do nó s, o primeiro arco pertencente ao caminho mínimo da lista de arcos saintes deve ser o arco armazenado em t^{SP}.

```
procedure T_-^KT(a = (u→,v), w_E, d_V, δ_V, g^SP)
1    if d_u < d_v + w_a then return;
2    if d_u = d_v + w_a then
3        if t_u^SP > a then t_u^SP = a;
4        return;
5    end if
6    d_u = d_v + w_a;
7    t_u^SP = a;
8    InserIntoHeap(H,u,d_u);
9    while HeapSize(H) > 0 do
10       u =FindAndDeleteMin(H,d);
11       forall e = (s→,u) ∈ IN(u) do
12           if d_s > d_u + w_e then
13               d_s = d_u + w_e;
14               t_s^SP = e;
15               AdjustHeap(H,s,d_s);
16           end if
17           else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
18       end forall
19   end while
end T_-^KT.
```

Figura 4.20: Pseudo-código do procedimento T_-^{KT}.

O último pseudo-código apresentado nesta seção é a variante de pilha-reduzida rhT_-^{KT} apre-

sentada na Figura 4.21. Seja $a = (\overrightarrow{u,v})$ o arco cujo peso é decrementado. Se $\bigtriangledown < 0$ o algoritmo pára (linha 2). Caso contrário, se o nó u possuir um caminho mínimo alternativo passando por a, a árvore t^{SP} é atualizada caso necessário, isto é, se $t_a^{SP} > v$. (linha 3) e o algoritmo pára (linha 5). A distância d_u é atualizada (linha 7), t_u^{SP} é atualizada (linha 8) e o nó u é inserido no conjunto Q (linha 9). O conjunto U é inicializado vazio na linha 10. Neste algoritmo, o vetor *maxdiff* é usado da mesma forma que foi usado em T_-^{RR}. Em T_-^{KT}, valores não negativos para *diff* são considerados. Caso $diff = 0$ (linha 15), assim como se *diff* = $maxdiff_s$ (linha 29), então o nó s tem um caminho mínimo alternativo. Neste caso, a árvore especial deve ser atualizada. Por isso, a necessidade dessa atualização é testada na linha 49.

4.6 Versões clássicas e de pilha-reduzida dos algoritmos de decremento

Nesta seção, algumas observações são salientadas sobre a comparação entre a versão clássica (`std`) e a variante de pilha-reduzida (`rh`) de cada algoritmo de decremento.

A Figura 4.22 mostra visualmente o tamanho das pilhas usadas pelos algoritmos de decremento discutidos na seção anterior. O objetivo é mostrar visualmente a parte afetada do grafo pelo decremento. A representação do grafo é o mesma usada nas Figuras 4.13 e 4.14 apresentados para os algoritmos de incremento.

Como pode-se observar na figura, para os algoritmos `std` todos os nós $u \in Q$ são inseridos na pilha H, enquanto que para as variantes `rh` nenhum nó $u \in Q$ é inserido na pilha H. Os nós $u \in U$ são inseridos na pilha H para os algoritmos `std` e `rh` para decrementos aleatórios, enquanto que H permanece vazia para decrementos unitários com a variante `rh`.

Em termos de memória, as variantes `rh` precisam de espaço extra em suas implementações. O algoritmo rhG_-^{RR} usa o vetor *degree* com V posições, não utilizado pela versão clássica `std`. Os algoritmos rhT_-^{RR} e rhT_-^{KT} utilizam dois vetores extra, `maxdiff` e U, ambos com V posições.

```
procedure rhT_-^KT(a = (u→,v), w_E, d_V, δ_V, g^SP)
1   ▽ = d_u − d_v − w_a;
2   if ▽ < 0 then return;
3   if ▽ = 0 then
4       if t_a^SP > v then t_a^SP = v;
5       return;
6   end if
7   d_u = d_v − ▽;
8   t_u^SP = a;
9   Q = {u};
10  U = ∅;
11  forall u ∈ Q do
12      forall e = (s→,u) ∈ IN(u) do
13          diff = d_s − d_u − w_e;
14          if diff ≥ 0 then
15              if diff = 0 then
16                  if t_s^SP > e then t_s^SP = e;
17              end if
18              else if diff = ▽ then
19                  d_s = d_s − ▽;
20                  t_s^SP = e;
21                  Q = Q ∪ {s};
22                  maxdiff_s = 0;
23              end if
24              else if diff > maxdiff_s then
25                  if maxdiff_s = 0 then U = U ∪ {s};
26                  maxdiff_s = diff;
27                  t_s^SP = e;
28              end if
29              else if diff = maxdiff_s and t_s^SP > e then t_s^SP = e;
30          end if
31      end forall
32  end forall
33  if ▽ = 1 then return;
34  forall u ∈ U do
35      if maxdiff_u > 0 then
36          d_u = d_u − maxdiff_u;
37          maxdiff_u = 0;
38          InserIntoHeap(H,u,d_u);
39      end if
40  end forall
41  while HeapSize(H) > 0 do
42      u = FindAndDeleteMin(H,d);
43      forall e = (s→,u) ∈ IN(u) do
44          if d_s > d_u + w_e then
45              d_s = d_u + w_e;
46              t_s^SP = e;
47              AdjustHeap(H,s,d_s);
48          end if
49          else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
50      end forall
51  end while
end rhT_-^KT.
```

Figura 4.21: Pseudo-código do procedimento rhT_{-}^{KT}.

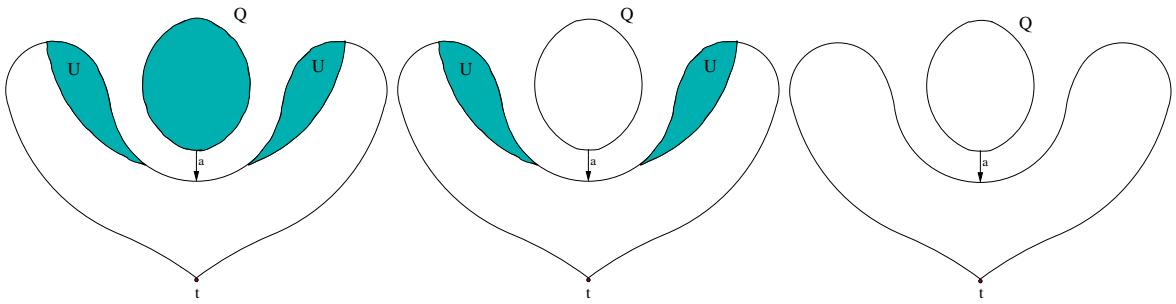

Figura 4.22: Tamanho das pilhas usadas pelos algoritmos de decremento G_{-}^{RR}, T_{-}^{RR} e T_{-}^{KT} e suas variantes de pilha-reduzida. O grafo da esquerda representa os algoritmos `std`. Os grafos do meio e da direita representam a variante `rh` no caso de decrementos aleatórios e unitários, respectivamente.

4.7 Resultados computacionais

Nesta seção os experimentos computacionais são apresentados. Os experimentos foram realizados em um computador 1.7 GHz Intel Pentium IV com 256 MB de RAM, com o sistema operacional RedHat Linux 8.0. Os códigos foram escritos em C e compilados com o `gcc` versão 3.2, usando o parâmetro de otimização `-O3`. Tempos de CPU foram medidos com a função do sistema `getrusage`.

Os experimentos foram realizados com nove classes de instâncias. A primeira classe, `Internet`, é composta pelas instâncias apresentadas no Capítulo 2, advindas de problemas de engenharia de tráfego. Esta classe é composta por quatro subclasses: `att`, `hier`, `rand` e `wax`. Para os experimentos realizados, o peso de cada arco é gerado aleatoriamente no intervalo $[1, \mu]$, sendo μ igual a 20 nos experimentos de mudanças unitárias de peso e 10.000 nos experimentos com mudanças aleatórias dos pesos. Instâncias da classe `Internet` têm múltiplos nós destinos, isto é, uma vez que o peso de um arco mudar, o grafo (ou árvore) para cada nó destino é atualizado.

As outras oito classes foram obtidas de Cherkassky, Goldberg e Radzik [CGR96]. As classes são denominadas de `Grid-SSquare-S`, `Grid-SWide`, `Grid-SLong`, `Grid-PHard`, `Rand-4`, `Rand-1:4`, `Rand-Len` e `Acyc-Pos`. Elas constituem todas as instâncias de Cherkassky, Goldberg e Radzik [CGR96] que somente têm arcos com pesos não-negativos. Os geradores das instâncias de Cherkassky, Goldberg e Radzik está disponível na Internet [1]. Estas instâncias foram originalmente propostas para caminhos mínimos com uma única fonte, mas todos os arcos foram invertidos (nós

[1] `http://www.avglab.att.com/~andrew/soft.html`

origens e destinos trocados) e os nós fontes foram redefinidos como destinos. Ao contrário das instâncias da classe `Internet`, estas instâncias têm apenas um nó destino. As instâncias das classes `Grid-SSquare-S`, `Grid-SWide` e `Grid-SLong` são geradas numa grade quadricular, retangular larga e retangular comprida, respectivamente, com o peso dos arcos selecionados no intervalo [0, 10000]. As instâncias da classe `Grid-PHard` são não-planares e são construídas com uma complexa estrutura de camadas. Nestas instâncias, um caminho entre dois nós com muitos arcos possuem comprimento menor que um caminho com poucos arcos. Os pesos são selecionados de um grande intervalo de inteiros, com alguns destes valores multiplicados por uma função da diferença entre coordenadas x das camadas. As instâncias das classes `Rand-4`, `Rand-1:4` e `Rand-Len` são geradas inicialmente construindo um ciclo Hamiltoniano. Os arcos que compõem o ciclo possuem pesos mínimos, enquanto que os pesos dos demais arcos são escolhidos aleatoriamente no intervalo $[0, \mu]$. O parâmetro μ é fixo em 10.000 para as classes `Rand-4` e `Rand-1:4`, e varia na classe `Rand-Len`. Para as instâncias do grupo `Rand-Len`, o peso do arco é fixo em 1 para a primeira instância da família, e selecionado no intervalo $[0, \mu]$ para as outras instâncias, com μ variando de 10 até 1.000.000. Devido aos arcos do caminho Hamiltoniano terem peso 1, a estrutura da árvore de caminhos mínimos muda conforme μ aumenta. Para valores grandes de μ, os arcos do caminho Hamiltoniano aparecem com mais frequência na árvore de caminhos mínimos e a árvore tende a ser mais longa. A classe `Acyc-Pos` é composta por redes acíclicas. Os pesos dos arcos do caminho Hamiltoniano possuem valor 1 e os pesos dos demais arcos são selecionados no intervalo [0, 10000]. Visto que os algoritmos de caminhos mínimos dinâmicos atualizam um grafo com pesos positivos, os pesos com valor `0` foram mudados para o valor 1.

A seguir é apresentada uma síntese das características das classes de instâncias usadas para testes:

- `Internet`: grafos advindos de problemas de engenharia de tráfego;
  - `att`: *AT&T Worldnet Backbone*;
  - `hier`: grafos de 2 níveis hierárquicos;
  - `rand`: grafos puramente aleatórios;
  - `wax`: nós uniformemente distribuídos num quadrado unitário;

- `Grid-SSquare-S`: nós distribuídos numa grade quadriculada;
- `Grid-SWide`: nós distribuídos numa grade retangular larga;
- `Grid-SLong`: nós distribuídos numa grade retangular comprida;
- `Grid-PHard`: grafos difíceis de grade;
- `Rand-4`: grafos esparsos;
- `Rand-1:4`: grafos densos;
- `Rand-Len`: dependência no peso do arco;
- `Acyc-Pos`: arcos acíclicos.

Cada classe é composta por vários grupos de instâncias, variando de quatro grupos na classe `Rand-4` para 13 grupos na classe `Internet`. Cada grupo consiste de cinco instâncias geradas com sementes diferentes. As sementes usadas para gerar as instâncias de Cherkassky, Goldberg e Radzik, usadas nos experimentos descritos a seguir, são aquelas distribuídas com os geradores. As instâncias de cada grupo de instâncias de grade possuem o mesmo conjunto de arcos, diferindo apenas no que diz respeito aos pesos destes. As instâncias dos outros grupos possuem dimensões idênticas, mas possuem arcos e pesos diferentes. As sementes usadas para gerar as instâncias da classe `Internet` foram $[1001,\ldots,1005]$. As instâncias de cada grupo da classe `Internet` possuem o mesmo conjunto de arcos, diferindo apenas no que diz respeito aos pesos dos arcos.

Para cada instância foram executados 10000 aumentos de pesos usando os algoritmos de incrementos, seguidos de 10000 decrementos de pesos, usando os algoritmos de decrementos, resultando no grafo original. Os tempos são medidos separadamente para os incrementos e decrementos. Os seguintes pares de algoritmos de incremento/decremento foram usados: (G_+^{RR}, G_-^{RR}), (rhG_+^{RR}, rhG_-^{RR}), (T_+^{RR}, T_-^{RR}), (rhT_+^{RR}, rhT_-^{RR}), (T_+^{KT}, T_-^{KT}) e (rhT_+^{KT}, rhT_-^{KT}). Para os algoritmos T_+^D e rhT_+^D somente os 10.000 incrementos de peso foram realizados dado que estes algoritmos são semi-dinâmicos. Além dos algoritmos de caminhos mínimos dinâmicos, resultados são apresentados para o algoritmo Dijkstra. Usou-se um mecanismo local simples para atualizar g^{SP} quando não há mudança no valor de distância dos nós e o algoritmo Dijkstra somente é executado se a distância

de no mínimo um nó muda. Os tempos divulgados para o algoritmo Dijkstra são uma estimativa. O algoritmo foi executado para as primeiras 100 mudanças e estimou-se o tempo para as 10.000 mudanças multiplicando o tempo encontrado por 100. Para certificar os valores das estimativas, alguns testes foram realizados aplicando-se o algoritmo Dijkstra nos 10.000 incrementos e 10.000 decrementos em algumas instâncias de cada classe.

Para cada instância, todos os algoritmos usam a mesma sequência de arcos e valores de mudanças de pesos. Estas sequências de arcos e valores de incremento são geradas uma única vez por um programa em separado e armazenado em arquivo. O processo de geração desse arquivo é descrito a seguir. Seja um grafo de caminhos mínimos g_1^{SP} para uma instância em particular e seja $\bar{w}$ o valor médio dos pesos dos arcos desta instância. O seguinte procedimento é repetido para $k = 1, \ldots, 10.000$ para produzir uma sequência de arcos $a_1, \ldots, a_{10.000}$ e valores de incrementos de pesos $\triangle_1, \ldots, \triangle_{10.000}$ para serem usados nas simulações:

1. selecione um arco a_k aleatoriamente de g_k^{SP};
2. escolha aleatoriamente um valor $\triangle_k$ do intervalo $[1, \bar{w}]$;
3. adicione $\triangle_k$ ao peso atual do arco a_k;
4. recalcule o grafo (ou árvore) de caminhos mínimos g_{k+1}^{SP}.

Os experimentos descritos nesta seção são agrupados em duas categorias: mudanças aleatórias e mudanças unitárias de pesos. A Subseção 4.7.1 apresenta os resultados para as mudanças de pesos em valores aleatórios, enquanto que a Subseção 4.7.2 apresenta os resultados para as mudanças unitárias dos pesos.

4.7.1 Mudanças aleatórias dos pesos

Esta subseção apresenta os experimentos computacionais das mudanças aleatórias dos pesos aplicadas em todas as instâncias de cada grupo das nove classes. O aumento (ou diminuição) do peso é um valor aleatório entre 1 e a média dos pesos dos arcos da instância original, isto é, $\triangle = [1, \bar{w}]$.

As Tabelas 4.1 - 4.27 apresentam, para cada classe, as médias de cinco execuções de cada instância de cada grupo. Cada classe apresenta três tabelas: tempo total de CPU (em segundos)

para as 10.000 atualizações dos algoritmos de incremento, tempo total (em segundos) das 10.000 atualizações dos algoritmos de decremento e o tamanho das pilhas para os dois algoritmos. Para cada instância, o tamanho da pilha é representado pela média do número de nós inseridos nela durante cada uma das 10.000 atualizações. Para cada tabela, a primeira coluna identifica o grupo de instâncias (`Grupo`), as próximas duas colunas indicam o número de nós $|V|$ e arcos $|E|$ de cada grupo. As demais colunas apresentam os resultados para as versões clássicas dos algoritmos G^{RR}, T^{RR} e T^{KT}, enquanto que para o algoritmo T^{D} somente os resultados de incremento são fornecidos.

Visto que os nós inseridos na pilha pelos algoritmos G^{RR}, T^{RR}, e T^{KT} são idênticos, as tabelas que apresentam os tamanhos das pilhas possuem valores iguais para estas entradas. Diferentemente, para as versões de pilha-reduzida destes algoritmos, as entradas não são iguais. Ainda, o tamanho das pilhas para os algoritmos de incremento e decremento G^{RR}, T^{RR} e T^{KT} são idênticos.

As tabelas com valores dos tempos apresentam uma última coluna que contém os resultados para o algoritmo Dijkstra (`Dij`). As tabelas que apresentam os tamanhos das pilhas não apresentam estes valores, pois o algoritmo Dijkstra sempre insere todos os nós do grafo na pilha.

Tabela 4.1: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Internet`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1: att	90	274	0,16	0,09	0,16	0,09	0,16	0,09	0,14	0,06	2,00
GR2: hier50a	50	148	0,29	0,18	0,29	0,17	0,28	0,17	0,26	0,14	3,00
GR3: hier50b	50	212	0,25	0,16	0,25	0,16	0,25	0,16	0,22	0,12	2,60
GR4: hier100	100	280	0,88	0,48	0,87	0,46	0,85	0,44	0,76	0,33	11,40
GR5: hier100a	100	360	0,61	0,38	0,58	0,36	0,58	0,35	0,52	0,28	11,60
GR6: rand50	50	228	0,19	0,13	0,17	0,12	0,18	0,13	0,17	0,11	2,00
GR7: rand50a	50	245	0,17	0,12	0,17	0,13	0,17	0,12	0,16	0,10	3,00
GR8: rand100	100	403	0,52	0,36	0,49	0,33	0,49	0,32	0,45	0,26	11,80
GR9: rand100a	100	410	0,52	0,35	0,49	0,33	0,48	0,32	0,44	0,26	12,00
GR10: wax50	50	169	0,24	0,16	0,23	0,15	0,23	0,15	0,22	0,13	3,00
GR11: wax50a	50	230	0,19	0,13	0,18	0,13	0,18	0,13	0,17	0,11	2,40
GR12: wax100	100	391	0,53	0,34	0,50	0,33	0,49	0,33	0,45	0,27	11,20
GR13: wax100a	100	476	0,47	0,32	0,44	0,31	0,44	0,30	0,40	0,25	11,60

Tabela 4.2: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Internet`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1: att	90	274	0,12	0,08	0,09	0,06	0,09	0,11	2,80
GR2: hier50a	50	148	0,20	0,16	0,15	0,14	0,16	0,22	2,60
GR3: hier50b	50	212	0,17	0,14	0,14	0,12	0,14	0,22	2,40
GR4: hier100	100	280	0,59	0,42	0,48	0,33	0,50	0,61	11,60
GR5: hier100a	100	360	0,44	0,35	0,34	0,29	0,36	0,47	11,40
GR6: rand50	50	228	0,13	0,13	0,11	0,11	0,11	0,17	2,40
GR7: rand50a	50	245	0,13	0,12	0,10	0,11	0,11	0,16	2,60
GR8: rand100	100	403	0,39	0,34	0,30	0,28	0,32	0,42	11,60
GR9: rand100a	100	410	0,39	0,33	0,29	0,28	0,31	0,41	11,80
GR10: wax50	50	169	0,17	0,14	0,14	0,13	0,14	0,19	3,00
GR11: wax50a	50	230	0,13	0,12	0,11	0,11	0,12	0,17	2,60
GR12: wax100	100	391	0,40	0,34	0,30	0,28	0,32	0,42	11,20
GR13: wax100a	100	476	0,35	0,32	0,27	0,26	0,29	0,40	11,00

Tabela 4.3: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Internet`.

				G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo		$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1:	att	90	274	36,98	36,98	1,50	1,49	36,98	1,50
GR2:	hier50a	50	148	70,55	70,55	2,10	2,08	70,55	2,10
GR3:	hier50b	50	212	53,59	53,59	1,94	1,92	53,59	1,94
GR4:	hier100	100	280	228,31	228,31	7,57	7,53	228,31	7,58
GR5:	hier100a	100	360	124,88	124,88	9,37	9,32	124,88	9,37
GR6:	rand50	50	228	30,60	30,60	2,22	2,22	30,60	2,23
GR7:	rand50a	50	245	27,39	27,39	2,23	2,22	27,39	2,23
GR8:	rand100	100	403	89,64	89,64	9,59	9,58	89,64	9,59
GR9:	rand100a	100	410	87,18	87,18	9,76	9,76	87,18	9,76
GR10:	wax50	50	169	47,70	47,70	2,13	2,12	47,70	2,13
GR11:	wax50a	50	230	30,22	30,22	2,30	2,30	30,22	2,30
GR12:	wax100	100	391	92,65	92,65	9,78	9,78	92,65	9,78
GR13:	wax100a	100	476	73,10	73,10	9,45	9,44	73,10	9,45

Tabela 4.4: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Grid-SSquare-S`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	4098	16385	0,35	0,30	0,34	0,31	0,34	0,32	0,30	0,26	34,40
GR2	16386	65537	1,00	0,91	0,95	0,89	0,95	0,92	0,85	0,75	185,60
GR3	65538	262145	2,33	2,10	2,28	2,11	2,30	2,18	1,97	1,73	851,80
GR4	262146	1048577	6,11	5,44	5,99	5,44	6,11	5,63	5,28	4,54	4480,40
GR5	1048578	4194305	15,35	13,57	14,91	13,41	15,53	14,02	13,17	11,37	27690,60

Tabela 4.5: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Grid-SSquare-S`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	4098	16385	0,24	0,21	0,19	0,16	0,19	0,18	35,00
GR2	16386	65537	0,66	0,62	0,48	0,43	0,49	0,47	186,40
GR3	65538	262145	1,46	1,35	1,02	0,94	1,05	1,00	852,20
GR4	262146	1048577	3,35	3,11	2,41	2,20	2,50	2,34	4470,60
GR5	1048578	4194305	7,81	7,09	5,42	4,87	5,63	5,20	27711,80

Tabela 4.6: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Grid-SSquare-S`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	4098	16385	40,67	40,67	31,03	29,50	40,67	31,03
GR2	16386	65537	88,67	88,67	72,56	71,02	88,67	72,57
GR3	65538	262145	171,15	171,15	140,75	137,50	171,16	140,75
GR4	262146	1048577	365,31	365,31	300,31	294,16	365,34	300,35
GR5	1048578	4194305	727,44	727,44	607,92	589,33	727,45	607,93

Tabela 4.7: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Grid-SWide`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	0,10	0,07	0,09	0,07	0,09	0,07	0,09	0,06	71,60
GR2	16385	49152	0,13	0,10	0,12	0,10	0,12	0,10	0,11	0,08	173,20
GR3	32769	98304	0,15	0,12	0,13	0,11	0,13	0,11	0,12	0,10	439,60
GR4	65537	196608	0,15	0,12	0,14	0,12	0,14	0,12	0,13	0,11	1251,00
GR5	131073	393216	0,17	0,15	0,16	0,14	0,16	0,13	0,15	0,12	3526,80
GR6	262145	786432	0,19	0,14	0,17	0,14	0,17	0,14	0,16	0,12	9700,20
GR7	524289	1572864	0,18	0,15	0,17	0,14	0,17	0,13	0,15	0,12	24350,20

Tabela 4.8: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Grid-SWide`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	0,08	0,07	0,05	0,04	0,05	0,05	71,80
GR2	16385	49152	0,10	0,09	0,07	0,06	0,07	0,07	173,60
GR3	32769	98304	0,11	0,10	0,08	0,07	0,08	0,08	440,20
GR4	65537	196608	0,12	0,11	0,08	0,08	0,08	0,08	1250,00
GR5	131073	393216	0,13	0,12	0,09	0,09	0,10	0,09	3527,40
GR6	262145	786432	0,14	0,13	0,10	0,09	0,10	0,10	9703,80
GR7	524289	1572864	0,14	0,13	0,10	0,10	0,10	0,10	24344,80

Tabela 4.9: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Grid-SWide`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8193	24576	11,13	11,13	4,62	4,60	11,13	4,62
GR2	16385	49152	11,25	11,25	4,59	4,56	11,26	4,59
GR3	32769	98304	11,36	11,36	4,81	4,78	11,36	4,81
GR4	65537	196608	11,14	11,14	4,64	4,61	11,15	4,64
GR5	131073	393216	11,49	11,49	4,67	4,64	11,49	4,67
GR6	262145	786432	11,58	11,58	4,50	4,46	11,58	4,50
GR7	524289	1572864	11,57	11,57	4,47	4,43	11,57	4,47

Tabela 4.10: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Grid-SLong`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	3,00	1,91	2,97	1,94	3,03	2,05	2,69	1,56	43,60
GR2	16385	49152	6,53	4,05	6,51	4,09	6,62	4,33	5,78	3,23	87,20
GR3	32769	98304	14,53	9,25	14,53	9,30	14,87	9,87	12,91	7,45	177,40
GR4	65537	196608	30,70	19,28	30,72	19,42	31,44	20,61	27,56	15,65	356,00
GR5	131073	393216	63,63	38,71	63,55	38,87	64,88	41,16	57,38	31,54	710,00
GR6	262145	786432	132,49	80,13	132,36	80,37	134,98	85,09	120,44	66,25	1427,60
GR7	524289	1572864	284,43	173,92	283,07	172,33	289,39	182,47	258,25	143,20	2847,00

Tabela 4.11: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Grid-SLong`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	1,80	1,32	1,34	0,98	1,40	1,31	43,00
GR2	16385	49152	3,75	2,67	2,76	1,93	2,83	2,66	87,20
GR3	32769	98304	8,11	5,87	5,82	4,21	6,01	5,59	176,80
GR4	65537	196608	16,29	11,74	11,78	8,45	12,12	11,27	356,20
GR5	131073	393216	32,30	23,02	23,37	16,54	24,02	22,37	709,80
GR6	262145	786432	64,25	45,69	47,18	33,45	48,49	44,89	1428,80
GR7	524289	1572864	133,12	95,40	96,34	68,43	98,94	91,72	2848,40

Tabela 4.12: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Grid-SLong`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8193	24576	326,31	326,31	159,40	158,68	326,32	159,41
GR2	16385	49152	650,45	650,45	302,50	300,30	650,46	302,50
GR3	32769	98304	1342,16	1342,16	668,65	665,15	1342,34	668,84
GR4	65537	196608	2683,29	2683,29	1319,14	1313,03	2683,30	1319,15
GR5	131073	393216	5347,72	5347,72	2569,13	2563,32	5347,72	2569,13
GR6	262145	786432	10767,60	10767,60	5172,76	5159,84	10767,60	5172,76
GR7	524289	1572864	22094,40	22094,40	10901,09	10844,20	22094,40	10901,10

Tabela 4.13: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Grid-PHard`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	63808	12,08	10,20	12,26	10,52	11,94	10,35	10,05	8,16	122,20
GR2	16385	129344	30,39	25,97	30,94	26,73	30,28	26,47	25,75	21,13	245,20
GR3	32769	260416	71,24	61,04	71,96	62,42	70,62	61,68	60,36	49,42	503,00
GR4	65537	522560	128,67	111,00	127,41	111,15	125,31	110,08	107,81	89,25	774,40
GR5	131073	1046848	286,33	248,09	283,35	247,14	278,38	245,54	239,81	199,27	1528,40
GR6	262145	2095424	587,38	511,89	583,30	514,92	572,67	508,82	496,90	414,58	3131,00

Tabela 4.14: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Grid-PHard`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	63808	7,44	6,46	5,03	4,34	5,33	5,39	122,00
GR2	16385	129344	17,05	15,14	11,72	10,27	12,43	12,90	245,60
GR3	32769	260416	38,99	35,42	24,84	22,13	26,37	28,06	502,80
GR4	65537	522560	62,82	57,97	38,00	35,27	40,46	44,71	776,20
GR5	131073	1046848	134,85	123,28	79,91	75,03	84,81	95,32	1528,20
GR6	262145	2095424	254,48	236,82	166,31	152,96	176,48	192,45	3128,80

Tabela 4.15: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Grid-PHard`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8193	63808	693,42	693,42	532,18	487,34	695,73	534,71
GR2	16385	129344	1699,61	1699,61	1316,92	1196,78	1706,92	1325,28
GR3	32769	260416	3873,01	3873,01	3028,37	2773,73	3895,42	3053,46
GR4	65537	522560	8597,23	8597,23	6842,77	6242,90	8656,73	6907,45
GR5	131073	1046848	18707,41	18707,41	15027,50	13694,50	18817,40	15148,60
GR6	262145	2095424	38572,20	38572,20	31242,00	28564,57	38810,60	31489,00

Tabela 4.16: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Rand-4`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	32768	0,39	0,33	0,38	0,32	0,38	0,32	0,32	0,25	90,40
GR2	16384	65536	0,94	0,83	0,93	0,84	0,97	0,86	0,79	0,68	276,60
GR3	32768	131072	2,00	1,80	2,10	1,94	2,20	1,95	1,73	1,55	817,40
GR4	65536	262144	4,41	4,00	4,76	4,43	4,98	4,40	3,92	3,56	2428,60
GR5	131072	524288	8,16	7,66	8,86	8,48	9,29	8,41	7,37	6,94	6266,60
GR6	262144	1048576	11,14	10,50	11,96	11,46	12,51	11,33	10,09	9,54	14752,40
GR7	524288	2097152	15,02	14,19	15,98	15,33	16,74	15,17	13,53	12,81	33346,60
GR8	1048576	4194304	20,54	19,28	21,42	20,47	22,53	20,23	18,39	17,20	73487,00

Tabela 4.17: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Rand-4`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	32768	0,33	0,28	0,19	0,17	0,19	0,19	90,40
GR2	16384	65536	0,77	0,64	0,48	0,47	0,49	0,48	276,80
GR3	32768	131072	1,56	1,32	0,99	0,99	1,01	1,00	819,00
GR4	65536	262144	3,15	2,70	2,01	2,00	2,04	2,03	2430,60
GR5	131072	524288	5,42	4,64	3,41	3,40	3,45	3,43	6269,00
GR6	262144	1048576	7,29	6,26	4,54	4,50	4,59	4,53	14739,20
GR7	524288	2097152	9,57	8,33	5,97	5,99	6,05	6,06	33415,20
GR8	1048576	4194304	12,84	11,08	7,94	7,82	8,10	7,94	73593,40

Tabela 4.18: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Rand-4`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8192	32768	25,33	25,33	17,62	17,61	25,38	17,66
GR2	16384	65536	41,46	41,46	33,15	33,15	41,50	33,20
GR3	32768	131072	67,19	67,19	55,99	55,98	67,31	56,12
GR4	65536	262144	117,05	117,05	102,34	102,33	117,34	102,63
GR5	131072	524288	184,52	184,52	168,38	168,37	185,22	169,09
GR6	262144	1048576	229,97	229,97	214,25	214,24	231,00	215,28
GR7	524288	2097152	278,77	278,77	268,56	268,55	280,07	269,86
GR8	1048576	4194304	331,87	331,87	309,41	309,41	333,66	311,22

Tabela 4.19: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Rand-1:4`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	512	65536	0,95	0,71	0,72	0,60	0,80	0,64	0,65	0,51	69,20
GR2	1024	262144	4,16	3,40	3,71	3,25	3,89	3,30	3,53	3,03	528,40
GR3	2048	1048576	12,69	10,95	12,03	10,82	12,49	10,85	11,80	10,54	2421,20
GR4	4096	4194304	37,18	32,70	37,02	33,07	38,32	32,96	36,90	33,30	9798,60

Tabela 4.20: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Rand-1:4`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	512	65536	1,06	0,83	0,48	0,47	0,56	0,53	69,40
GR2	1024	262144	4,42	3,49	2,61	2,60	2,73	2,67	528,40
GR3	2048	1048576	12,84	10,14	7,87	7,87	8,08	7,99	2421,00
GR4	4096	4194304	39,26	30,84	23,47	23,10	23,80	23,42	9797,60

Tabela 4.21: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Rand-1:4`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	512	65536	3,98	3,98	2,58	2,58	3,99	2,59
GR2	1024	262144	5,53	5,53	4,04	4,03	5,57	4,08
GR3	2048	1048576	7,22	7,22	5,64	5,63	7,37	5,79
GR4	4096	4194304	9,58	9,58	7,81	7,80	10,08	8,32

Tabela 4.22: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Rand-Len:4`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	131072	524288	0,97	0,68	0,96	0,68	1,01	0,67	0,83	0,55	7127,40
GR2	131072	524288	0,44	0,37	0,44	0,37	0,46	0,37	0,44	0,37	4189,20
GR3	131072	524288	0,56	0,42	0,56	0,43	0,58	0,42	0,49	0,36	6236,20
GR4	131072	524288	0,86	0,60	0,86	0,60	0,90	0,59	0,74	0,49	7138,00
GR5	131072	524288	0,84	0,59	0,83	0,59	0,87	0,59	0,72	0,48	7169,60

Tabela 4.23: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Rand-Len`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	131072	524288	0,73	0,59	0,50	0,41	0,50	0,41	7128,80
GR2	131072	524288	0,34	0,31	0,22	0,21	0,22	0,22	4188,00
GR3	131072	524288	0,43	0,37	0,29	0,26	0,29	0,26	6236,60
GR4	131072	524288	0,65	0,53	0,45	0,36	0,45	0,37	7129,20
GR5	131072	524288	0,63	0,52	0,44	0,36	0,44	0,36	7166,40

Tabela 4.24: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Rand-Len`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	131072	524288	16,04	16,04	6,87	6,87	16,04	6,87
GR2	131072	524288	8,79	8,79	5,04	5,04	10,31	6,56
GR3	131072	524288	11,02	11,02	5,09	5,09	11,20	5,27
GR4	131072	524288	14,61	14,61	6,23	6,23	14,62	6,23
GR5	131072	524288	14,31	14,31	6,30	6,30	14,31	6,30

Tabela 4.25: Tempos de CPU em segundos gastos em 10^4 incrementos aleatórios de pesos nas instâncias da classe `Acyc-Pos`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	131072	1,06	0,82	0,97	0,79	0,96	0,75	0,86	0,65	395,40
GR2	16384	262144	2,07	1,74	2,02	1,75	1,95	1,63	1,78	1,49	998,20
GR3	32768	524288	4,02	3,51	4,25	3,78	4,05	3,44	3,63	3,17	2653,20
GR4	65536	1048576	7,85	6,93	8,66	7,91	8,31	6,89	7,29	6,45	6586,20
GR5	131072	2097152	13,24	12,29	14,87	14,00	14,44	12,29	12,55	11,68	15605,20

Tabela 4.26: Tempos de CPU em segundos gastos em 10^4 decrementos aleatórios de pesos nas instâncias da classe `Acyc-Pos`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	*Dij*
GR1	8192	131072	1,05	0,89	0,60	0,57	0,64	0,60	396,40
GR2	16384	262144	1,97	1,61	1,16	1,14	1,22	1,17	998,20
GR3	32768	524288	3,77	3,04	2,22	2,19	2,29	2,23	2649,80
GR4	65536	1048576	7,00	5,85	4,17	4,16	4,33	4,18	6593,20
GR5	131072	2097152	11,25	9,25	6,66	6,63	6,90	6,67	15596,60

Tabela 4.27: Tamanho da pilha para cada atualização (média de 10^4 mudanças aleatórias de pesos) nas instâncias `Acyc-Pos`.

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	*Incr*	*Decr*	*Incr*	*Decr*	*Incr*	*Incr*
GR1	8192	131072	15,90	15,90	10,90	10,90	15,92	10,92
GR2	16384	262144	25,11	25,11	19,68	19,68	25,15	19,72
GR3	32768	524288	40,52	40,52	33,49	33,49	40,62	33,59
GR4	65536	1048576	66,86	66,86	57,11	57,11	67,00	57,24
GR5	131072	2097152	97,67	97,67	88,01	88,01	97,97	88,31

No restante dessa subseção, discute-se os resultados computacionais realizados com mudanças aleatórias dos pesos.

Melhoramentos obtidos usando-se as variantes de pilha-reduzida

Tabela 4.28: Razão entre os tempos gastos pelos algoritmos *std* e *rh* para 10^4 mudanças aleatórias de pesos.

Classe	G_+^{RR}	G_-^{RR}	T_+^{RR}	T_-^{RR}	T_+^{KT}	T_-^{KT}	T_+^{D}
`Internet`	1,55	1,21	1,55	1,11	1,57	0,74	1,79
`Grid-SSquare-S`	1,13	1,09	1,10	1,12	1,07	1,06	1,15
`Grid-SWide`	1,26	1,08	1,22	1,08	1,22	1,01	1,29
`Grid-SLong`	1,61	1,39	1,60	1,40	1,54	1,07	1,78
`Grid-PHard`	1,16	1,11	1,15	1,11	1,14	0,93	1,21
`Rand-4`	1,10	1,17	1,08	1,02	1,12	1,01	1,11
`Rand-1:4`	1,22	1,27	1,14	1,01	1,18	1,03	1,17
`Rand-Len`	1,37	1,19	1,34	1,17	1,43	1,16	1,42
`Acyc-Pos`	1,17	1,21	1,13	1,02	1,21	1,04	1,17
Média	1,29	1,19	1,26	1,12	1,28	1,01	1,34

A Tabela 4.28 compara os tempos gastos pelos algoritmos clássicos e as variantes de pilha-reduzida de todos os algoritmos de caminhos mínimos dinâmicos descritos neste capítulo. Cada valor da tabela corresponde à razão entre os tempos. Mais precisamente, sendo ts_i e tr_i os tempos computacionais médios para as cinco instâncias do i-ésimo grupo de cada classe, para os algoritmos clássico e de pilha-reduzida, respectivamente. Os valores da tabela são calculados como

$$\frac{\sum_{i=i}^{n_g} \frac{ts_i}{tr_i}}{n_g}, \tag{4.1}$$

onde n_g é o número de grupos na classe. A última linha lista a média de cada algoritmo. Note que os valores de tr_i e ts_i são aqueles apresentados nas tabelas de tempo apresentadas no início da Seção 4.7.1.

Na média, o algoritmo de pilha-reduzida executa mais rápido que o algoritmo clássico correspondente, como pode ser visto na última linha da tabela. Para todas as variantes de incremento, as versões de pilha-reduzida são mais rápidas na média para todas as classes. Essa afirmação também é verdadeira para os algoritmos de decremento, com exceção do algoritmo T_-^{KT}, para o qual a variante de pilha-reduzida levou mais tempo que a versão clássica em duas das nove classes.

A maior média de ganho foi obtida pelo algoritmo T_+^D, cuja média é de 1,79 para a classe `Internet`. Da última linha, observa-se que os algoritmos de incremento se beneficiam mais da idéia de redução de pilha que o de decremento. Isto era esperado, visto que esta idéia aplicada aos algoritmos de decremento requer esforço computacional adicional: os arcos entrantes nos nós do conjunto Q são consultados duas vezes, enquanto que nos algoritmos clássicos eles são consultados uma única vez. O algoritmo clássico de incremento gastou, em média, 29,25% mais tempo que os respectivos algoritmos de pilha-reduzida, enquanto que os algoritmos de decremento gastaram 10,67% mais tempo que seus respectivos algoritmos de pilha-reduzida.

Tabela 4.29: Razão entre os tamanhos das pilhas usadas pelos algoritmos `std` e `rh` para 10^4 mudanças aleatórias de pesos.

	G^{RR}, T^{RR}, T^{KT}		T^D
Classe	Incr	Decr	Incr
`Internet`	17,42	17,50	17,42
`Grid-SSquare-S`	1,23	1,27	1,23
`Grid-SWide`	2,46	2,48	2,46
`Grid-SLong`	2,06	2,07	2,06
`Grid-PHard`	1,27	1,39	1,27
`Rand-4`	1,16	1,16	1,16
`Rand-1:4`	1,35	1,36	1,35
`Rand-Len`	2,17	2,17	2,13
`Acyc-Pos`	1,24	1,24	1,24
Média	3,37	3,40	3,37

A Tabela 4.29 compara a média do tamanho das pilhas. Cada valor da tabela apresenta a razão média do tamanho da pilha entre o algoritmo clássico e a variante de pilha-reduzida. Mais precisamente, sendo hs_i e hr_i a média dos tamanhos das pilhas para as cinco instâncias do i-ésimo grupo de cada classe, para os algoritmos clássico e de pilha-reduzida, respectivamente. Os valores da tabela são calculados como

$$\frac{\sum_{i=i}^{n_g} \frac{hs_i}{hr_i}}{n_g}. \tag{4.2}$$

A última linha lista a média da razão de cada algoritmo. Note que os valores de hr_i e hs_i são aqueles apresentados nas tabelas de tamanho de pilha apresentadas no início da Seção 4.7.1. Para todas as instâncias de todos os grupos e classes, o algoritmo `rh` reduziu o tamanho da pilha usado. Na média, o tamanho da pilha foi reduzido para mais de $\frac{1}{3}$ do tamanho obtido pelo algoritmo clássico.

Comparação de tempo entre calcular o grafo com Dijkstra (`Dij`) ou atualizá-lo com G^{RR}

A Tabela 4.30 apresenta as médias entre o tempo total gasto pelo algoritmo Dijkstra e G^{RR} para cada classe de instâncias. Cada valor na tabela corresponde à média entre a razão dos tempos gastos pelos algoritmos `Dij` e G^{RR} Os valores são calculados de uma forma similar de como foi em (4.1).

Nesta tabela, os tempos de CPU de cada algoritmo é a soma dos tempos gastos nas fases de incremento e decremento. Para as instâncias das classes `Internet`, `Grid-SSquare-S`, `Grid-SWide`, `Rand-4`, `Rand-14` e `Acyc-Pos`, a razão aumenta com o aumento das dimensões da instância, enquanto que o oposto ocorre com as instâncias das classes `Grid-SLong` e `Grid-PHard`. A classe `Rand-Len` é composta por instâncias de mesmo tamanho. A menor razão obtida é de 7,26, para o grupo `GR5` da classe `Grid-PHard`. A maior razão é maior que 149.000, para o grupo `GR7` da classe `Grid-SWide`. O que pode-se obviamente concluir analisando esta tabela é que, em termos de tempos computacionais, os algoritmos de caminhos mínimos dinâmicos devem ser usados no lugar do algoritmo Dijkstra sempre que uma atualização for requerida.

As curvas comparando-se os algoritmos `Dij` e G^{RR} são fornecidas nos gráficos das Figuras 4.23 e 4.24. O gráfico relativo à classe `Rand-Len` não é apresentado, visto que o número de nós e arcos de todas as instâncias dessa classe são os mesmos. Para a maioria das classes, os tempos do algoritmo G^{RR} cresce na mesma taxa em que o algoritmo Dij cresce com o aumento do número de arcos. Mas para a classe `Grid-SWide` os tempos gastos pelos algoritmo G^{RR} crescem numa taxa muito menor que o algoritmo Dij.

A Figura 4.25 apresenta um gráfico ilustrativo do maior e menor ganhos (média por grupo) obtidos pelas nove classes de instâncias usadas para teste.

Tabela 4.30: Razão entre o tempo gasto pelos algoritmos Dij e G^{RR} para atualizar 10^4 mudanças aleatórias em cada grupo de todas as classes.

Grupos	Internet	Grid-SSquare-S	Grid-SWide	Grid-SLong	Grid-PHard	Rand-4	Rand-1 4	Rand-Len	Acyc-Pos
GR1	17,27	117,63	814,77	18,03	12,51	249,72	68,89	8395,88	374,91
GR2	11,43	223,83	1507,83	16,96	10,35	324,77	123,29	10712,53	493,67
GR3	11,79	449,37	3383,85	15,64	9,12	459,40	189,64	12522,89	681,27
GR4	15,65	945,59	9262,96	15,16	8,10	643,26	256,36	9435,98	887,62
GR5	22,03	2391,75	23204,61	14,80	7,26	922,96		9739,13	1274,06
GR6	13,75		60260,87	14,52	7,44	1600,20			
GR7	18,42		149371,17	13,64		2715,88			
GR8	25,88					4406,24			
GR9	26,10								
GR10	14,56								
GR11	15,53								
GR12	24,24								
GR13	27,63								

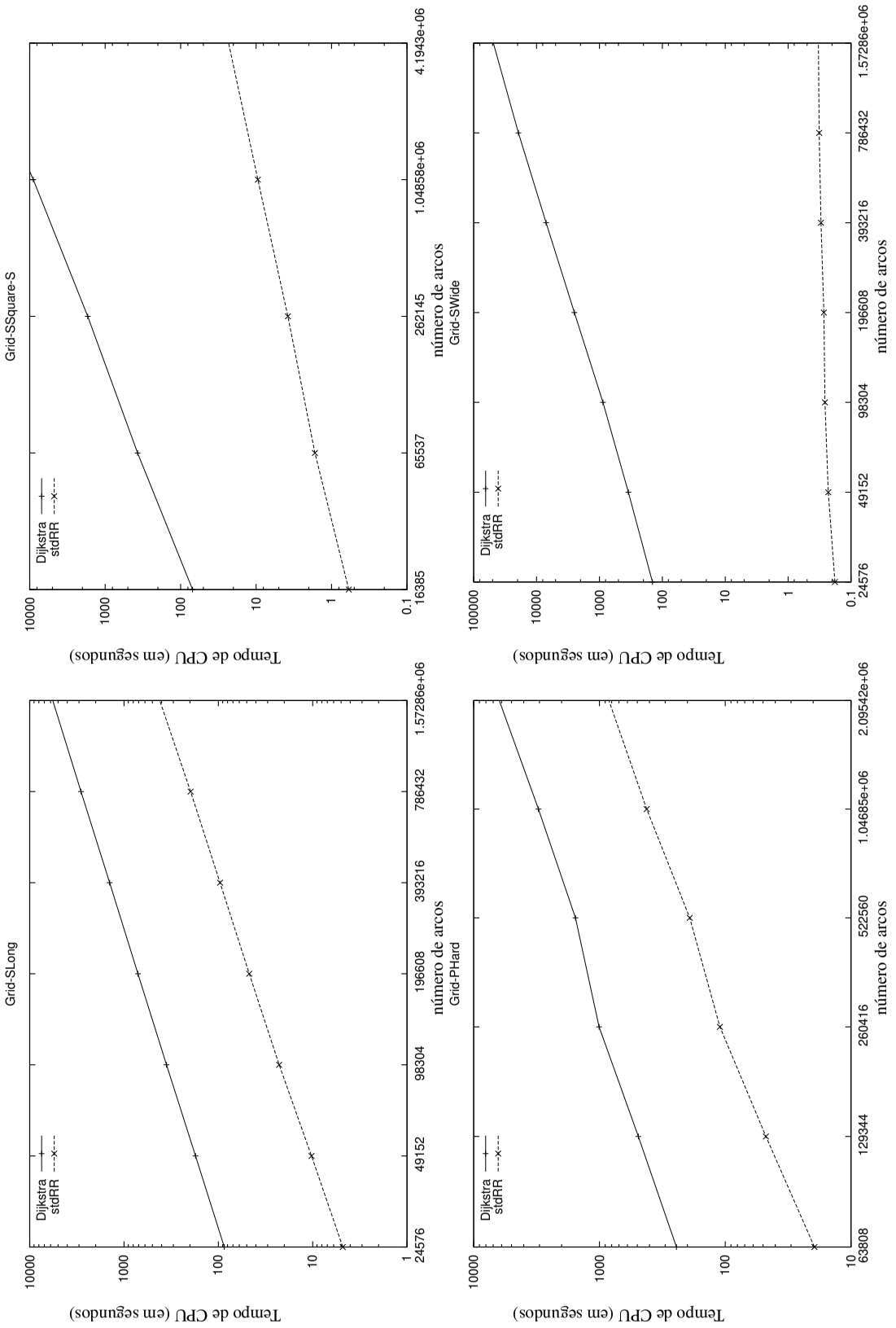

Figura 4.23: Tempo *vs.* número de arcos para as classes `Grid-SSquare-S`, `Grid-Wide`, `Grid-SLong` e `Grid-PHard`.

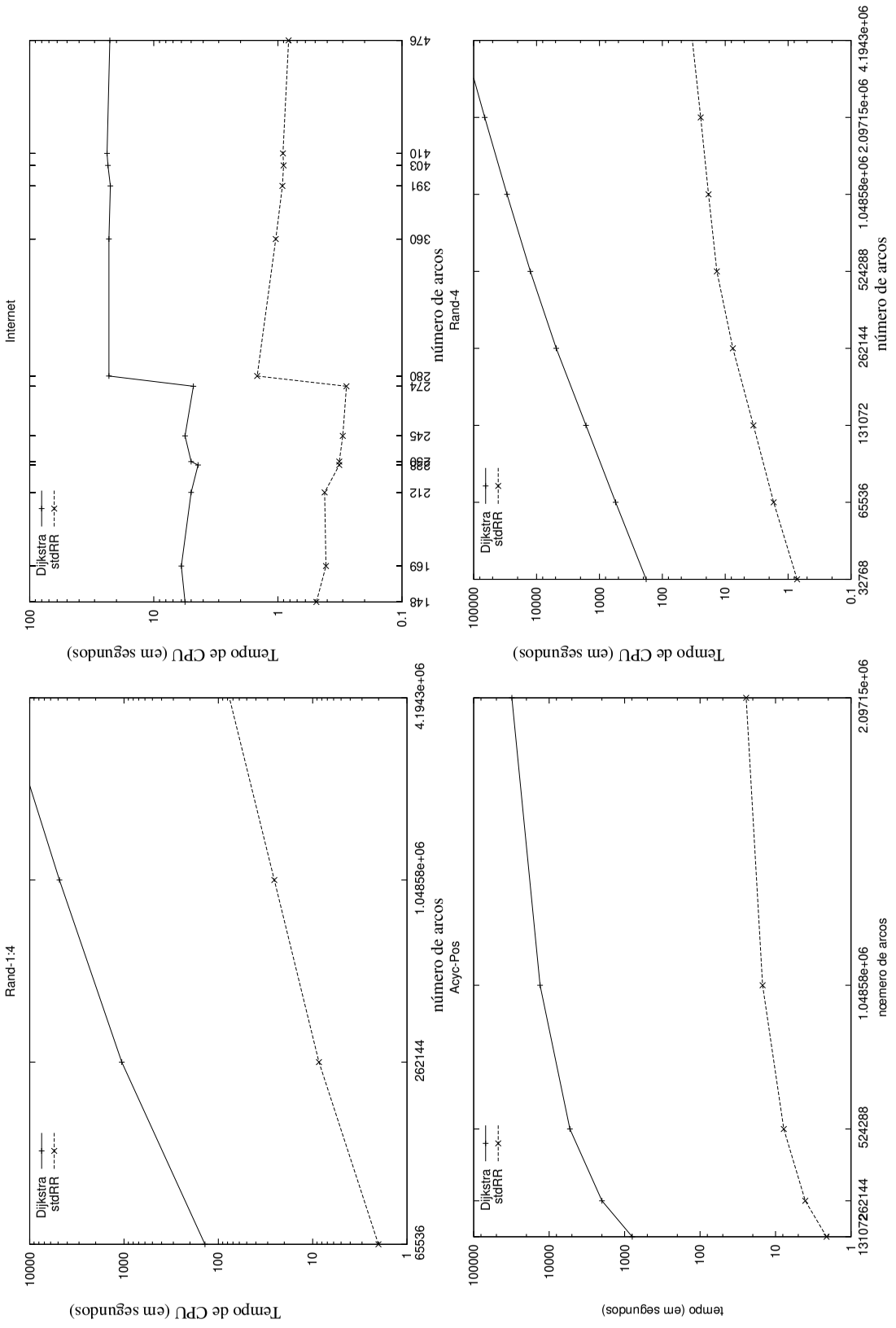

Figura 4.24: Tempo *vs.* número de arcos para as classes `Internet`, `Rand-4`, `Rand-1:4` e `Acyc-Pos`.

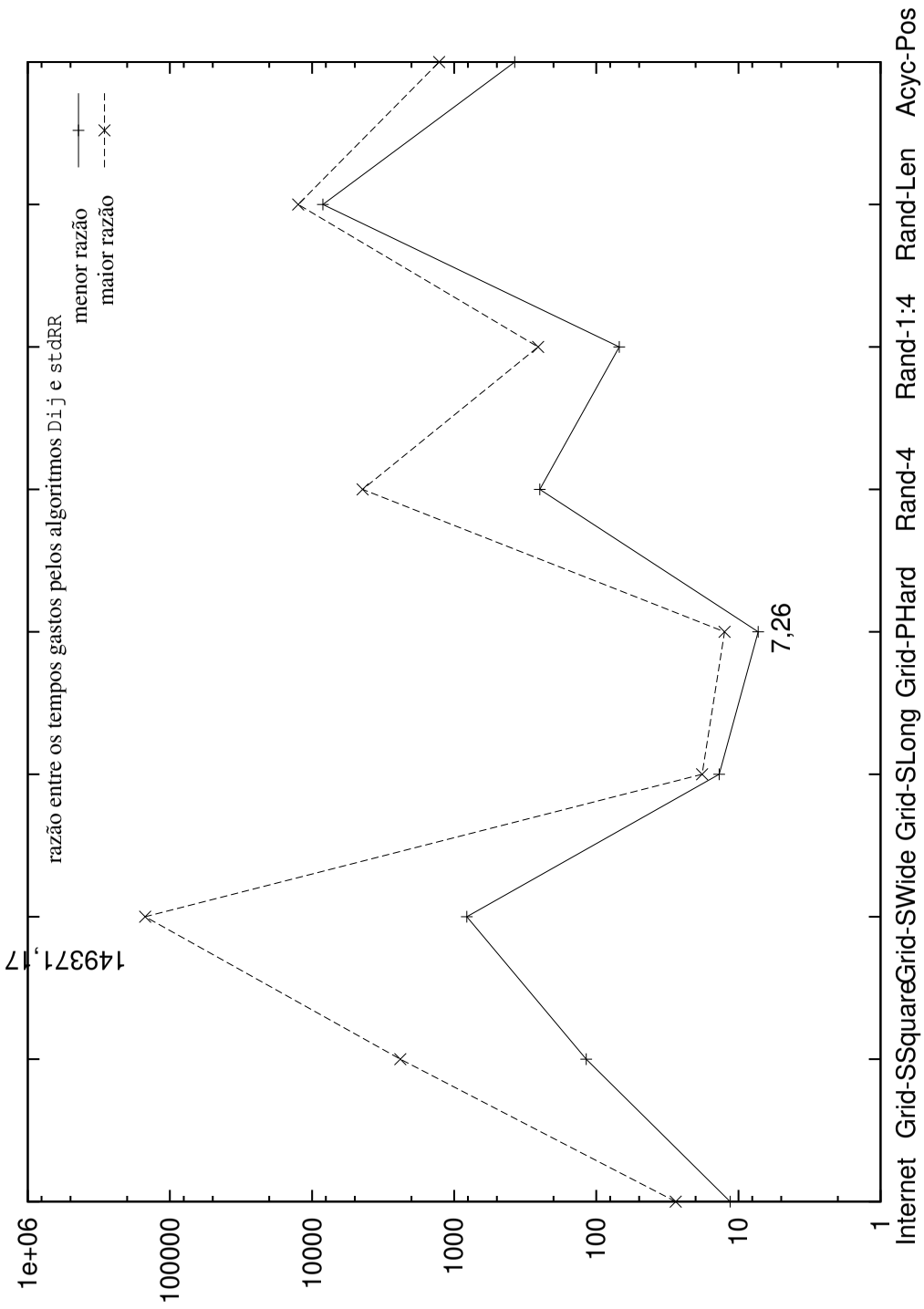

Figura 4.25: Maior e menor ganhos obtidos (média do grupo) por cada uma das nove classes de instâncias usadas para teste.

Comparação de tempo entre T^{KT} e T^{RR}

Nesta subseção, é feita uma comparação entre os algoritmos T^{KT} e T^{RR} de atualização de árvores de caminhos mínimos. O objetivo é saber se computacionalmente é vantajoso armazenar a árvore particular proposta por King and Thorup [KT01] ou se, na prática, é computacionalmente mais caro mantê-la que usufruir de seus benefícios. A Tabela 4.31 apresenta resultados para esta comparação. Cada valor da tabela corresponde à razão média dos tempos de CPU dos algoritmos

Tabela 4.31: Razão entre os tempos gastos pelos algoritmos T^{KT} e T^{RR} para 10^4 mudanças aleatórias de pesos.

	std		rh	
Classe	Incr	Decr	Incr	Decr
`Internet`	1,00	1,06	1,56	1,60
`Grid-SSquare-S`	1,02	1,03	1,09	1,09
`Grid-SWide`	1,00	1,04	1,10	1,11
`Grid-SLong`	1,02	1,03	1,30	1,34
`Grid-PHard`	0,98	1,06	1,22	1,26
`Rand-4`	1,04	1,02	1,02	1,03
`Rand-1:4`	1,06	1,06	1,04	1,05
`Rand-Len`	1,05	1,01	1,01	1,02
`Acyc-Pos`	0,97	1,05	0,99	1,02
Média	1,02	1,04	1,15	1,17

T^{KT} e T^{RR}. Os dados dessa tabela são computados similarmente de como foi para à Fórmula (4.1). A última linha lista a razão média de cada algoritmo.

A segunda coluna da tabela apresenta resultados da comparação entre as implementações clássicas, enquanto que a terceira coluna apresenta comparação entre as variantes de pilha-reduzida. Para ambos os casos, os resultados dos algoritmos de incremento e decremento são também comparados. Na média, tanto para incremento quanto para decremento, para os algoritmos clássicos e de pilha-reduzida, o algoritmo T^{RR} é mais rápido que o T^{KT}. Para os algoritmos clássicos, o desempenho para os algoritmos `std` e `rh` é praticamente igual, enquanto que para a variante de pilha-reduzida, o algoritmo T^{KT}, na média, gasta 15% (*Incr*) e 17% (*Decr*) mais tempo que o T^{RR}.

Comparação de tempo entre os algoritmos T^{RR} e T^{D}

A seguir, o desempenho dos algoritmos T^{RR} e T^{D} é comparada. A Tabela 4.32 apresenta os resultados dessa comparação A segunda coluna da tabela apresenta os resultados da comparação de

Tabela 4.32: Razão entre os tempos gastos pelos algoritmos T^{D} e T^{RR} para 10^4 mudanças aleatórias de pesos.

	Tempo		Tamanho da pilha	
Classe	std	rh	std	rh
`Internet`	0,91	0,79	1,00	1,00
`Grid-SSquare-S`	0,88	0,84	1,00	1,00
`Grid-SWide`	0,93	0,89	1,00	1,00
`Grid-SLong`	0,90	0,81	1,00	1,00
`Grid-PHard`	0,84	0,80	1,01	1,01
`Rand-4`	0,84	0,81	1,00	1,00
`Rand-1:4`	0,96	0,94	1,02	1,03
`Rand-Len`	0,90	0,85	1,04	1,07
`Acyc-Pos`	0,86	0,83	1,00	1,00
Média	0,89	0,84	1,01	1,01

tempo, enquanto a terceira coluna mostra a comparação entre os tamanhos das pilhas. Para ambos, os resultados apresentam comparação entre os algoritmos `std` e `rh`. Como T^{D} possui somente a versão de incremento, resultados para decremento não são apresentados para este algoritmo.

Cada valor da tabela representa a razão média dos tempos de CPU dos algoritmos T^{RR} e T^{D}. Para a segunda coluna, os valores são calculados similarmente a (4.1), enquanto que para a terceira coluna, os valores foram obtidos como em (4.2). A última linha apresenta a razão média de cada algoritmo.

Na média, T_{+}^{RR} consome 10% mais tempo que T_{+}^{D}, para os algoritmos `std` e `rh`. Em se tratando do tamanho da pilha, observa-se que na média o número de nós inseridos na pilha pelo algoritmo T_{+}^{D} é somente 1% maior que a de T_{+}^{RR}. Pode-se concluir que para intervalos grandes de incrementos, embora o tamanho da pilha seja maior, o tempo computacional gasto pelo algoritmo T_{+}^{D} é consideravelmente menor devido ao seu simples mecanismo de identificação dos elementos do conjunto Q.

Comparação de tempo entre os algoritmos de atualização de grafos G^{RR} e árvores T^{RR}

A seguir comparam-se os tempos gastos para atualizar um grafo e para atualizar uma árvore de caminhos mínimos. Os algoritmos usados nesta comparação foram G^{RR} e T^{RR}. Entre os três algoritmos clássicos para atualizar árvores, T^{RR}, T^{KT} e T^{D}, o algoritmo T^{RR} foi escolhido por executar mais rápido, em nossos experimentos, que o algoritmo T^{KT}, e por ser mais previsível que o algoritmo T^{D}.

A Tabela 4.33 apresenta uma comparação dos resultados para as versões `std` e `rh` desses algoritmos. Cada valor da tabela apresenta uma razão média entre tempos dos algoritmos G^{RR} e T^{RR}.

Tabela 4.33: Razão entre os tempos gastos pelos algoritmos G^{RR} e T^{RR} para 10^4 mudanças aleatórias de pesos.

	std		rh	
Classe	Incr	Decr	Incr	Decr
`Internet`	1,04	1,04	1,29	1,18
`Grid-SSquare-S`	1,03	1,00	1,39	1,42
`Grid-SWide`	1,10	1,08	1,45	1,45
`Grid-SLong`	1,00	0,99	1,37	1,38
`Grid-PHard`	1,00	0,99	1,56	1,57
`Rand-4`	0,96	0,94	1,61	1,41
`Rand-1:4`	1,13	1,06	1,80	1,44
`Rand-Len`	1,01	0,99	1,48	1,44
`Acyc-Pos`	0,97	0,94	1,70	1,43
Média	1,03	1,00	1,52	1,41

Estes são calculados de forma similar a (4.1). A última linha apresenta a razão média de cada algoritmo.

A segunda coluna da tabela apresenta resultados comparando as versões clássicas deles, enquanto que a terceira coluna apresenta a comparação entre as variantes de pilha-reduzida. Em ambos os casos, os resultados dos algoritmos de incremento e decremento são comparados. Observa-se que para as implementações clássicas dos algoritmos, atualizar uma árvore é um pouco mais rápido que atualizar um grafo para o caso de incremento, e praticamente o mesmo no caso de decremento. Na comparação dos algoritmos de pilha-reduzida, o tempo para atualizar um grafo é 52% e 41% mais demorado que atualizar uma árvore, para os casos de incremento e decremento, respectivamente.

Comparação entre os algoritmos de incremento de decremento

Nesta subseção, a diferença de tempo entre os algoritmos de incremento e decremento dos algoritmos são comparados. A Tabela 4.34 apresenta os resultados desta comparação. Cada valor

Tabela 4.34: Razão entre os tempos gastos pelos algoritmos *Incr* e *Decr* para 10^4 mudanças aleatórias de pesos.

Classe	G^{RR}	rhG^{RR}	T^{RR}	rhT^{RR}	T^{KT}	rhT^{KT}
`Internet`	1,38	1,08	1,72	1,23	1,62	0,76
`Grid-SSquare-S`	1,67	1,62	2,26	2,30	2,23	2,19
`Grid-SWide`	1,33	1,14	1,75	1,53	1,68	1,39
`Grid-SLong`	1,89	1,63	2,59	2,27	2,57	1,79
`Grid-PHard`	1,95	1,85	3,06	2,94	2,83	2,31
`Rand-4`	1,41	1,51	2,38	2,26	2,45	2,20
`Rand-1:4`	0,94	0,99	1,50	1,33	1,50	1,30
`Rand-Len`	1,32	1,15	1,93	1,68	2,01	1,63
`Acyc-Pos`	1,08	1,13	1,92	1,73	1,78	1,54
Média	1,44	1,34	2,12	1,92	2,07	1,68

da tabela representa a razão média dos tempos dos pares de algoritmos (G^{RR}_{+}, G^{RR}_{-}), (rhG^{RR}_{+}, rhG^{RR}_{-}), (T^{RR}_{+}, T^{RR}_{-}), (rhT^{RR}_{+}, rhT^{RR}_{-}), (T^{KT}_{+}, T^{KT}_{-}) e (rhT^{KT}_{+}, rhT^{KT}_{-}). Os valores das tabelas são calculados da mesma forma que em (4.1). A última linha apresenta a razão média dos algoritmos.

Em quase todos os casos, os algoritmos de decremento são mais rápidos que os de incremento. Este desempenho era esperado, visto que o número de arcos consultados pelos algoritmos de decremento é menor que os consultados pelos algoritmos de incremento.

Ainda, observa-se que para o algoritmo `rh`, os valores são menores que os apresentados para o algoritmo `std`.

4.7.2 Mudanças unitárias dos pesos

Esta seção apresenta os resultados computacionais para o caso de mudanças unitárias dos pesos, considerando todas as instâncias de cada grupo das nove classes. O incremento e o decremento são iguais a *1*, isto é, $\triangle = 1$ e $\bigtriangledown = 1$.

As Tabelas 4.35 - 4.61 apresentam, para cada classe, as médias para as cinco instâncias de cada grupo.

Este processo é o mesmo usado pelas mudanças aleatórias, com a única diferença de que agora o incremento é unitário. Assim como antes, o algoritmo é executado para 10^4 mudanças de pesos. Para este experimento, as instâncias da classe `Internet` tem pesos gerados no intervalo [1, 20]. As instâncias das outras oito classes possuem os pesos originais modificados. A função `mod` foi usada para obter os pesos no intervalo [1, 20]. Cada peso é calculado como $w_a = 1 + (20 * \lfloor w_a \div 20 \rfloor)$.

Visto que os nós inseridos na pilha pelos algoritmos G^{RR}, T^{RR} e T^{KT} são idênticos, nas tabelas que apresentam os tamanhos das pilhas estes valores aparecem uma única vez. Os tamanhos das pilhas são apresentadas somente para a variante de pilha-reduzida rhT^D, visto que é o único algoritmo `rh` que insere nós na pilha no caso de mudança unitária de peso.

Tabela 4.35: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Internet`.

Grupo		$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1:	att	90	274	0,15	0,06	0,15	0,06	0,15	0,06	0,13	0,06	2,00
GR2:	hier50	50	148	0,29	0,14	0,29	0,12	0,29	0,13	0,26	0,13	2,60
GR3:	hier50b	50	212	0,26	0,12	0,26	0,12	0,26	0,12	0,23	0,12	2,40
GR4:	hier100	100	280	0,88	0,35	0,88	0,33	0,87	0,31	0,79	0,34	11,60
GR5:	hier100a	100	360	0,60	0,28	0,59	0,26	0,59	0,26	0,54	0,27	10,60
GR6:	rand50	50	228	0,18	0,10	0,17	0,09	0,17	0,10	0,16	0,10	2,00
GR7:	rand50a	50	245	0,16	0,10	0,16	0,09	0,16	0,10	0,15	0,10	2,20
GR8:	rand100	100	403	0,49	0,25	0,47	0,22	0,46	0,23	0,42	0,24	10,00
GR9:	rand100a	100	410	0,48	0,25	0,46	0,23	0,46	0,23	0,42	0,24	10,80
GR10:	wax50	50	169	0,23	0,12	0,23	0,11	0,22	0,11	0,20	0,12	2,60
GR11:	wax50a	50	230	0,16	0,10	0,17	0,09	0,16	0,10	0,16	0,10	2,60
GR12:	wax100	100	391	0,49	0,25	0,47	0,23	0,47	0,24	0,44	0,25	10,80
GR13:	wax100a	100	476	0,43	0,24	0,41	0,21	0,41	0,22	0,38	0,22	9,60

Tabela 4.36: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Internet`.

Grupo		$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1:	att	90	274	0,11	0,05	0,08	0,06	0,09	0,10	2,00
GR2:	hier50a	50	148	0,20	0,12	0,16	0,13	0,17	0,22	2,40
GR3:	hier50b	50	212	0,18	0,11	0,14	0,12	0,15	0,23	2,40
GR4:	hier100	100	280	0,60	0,31	0,50	0,32	0,53	0,61	11,60
GR5:	hier100a	100	360	0,44	0,26	0,35	0,27	0,37	0,47	10,80
GR6:	rand50	50	228	0,13	0,10	0,10	0,11	0,11	0,16	2,40
GR7:	rand50a	50	245	0,12	0,09	0,09	0,10	0,10	0,15	2,60
GR8:	rand100	100	403	0,37	0,25	0,28	0,26	0,31	0,41	9,60
GR9:	rand100a	100	410	0,37	0,25	0,28	0,26	0,31	0,40	11,00
GR10:	wax50	50	169	0,17	0,11	0,13	0,12	0,14	0,19	2,60
GR11:	wax50a	50	230	0,12	0,09	0,10	0,10	0,10	0,15	2,20
GR12:	wax100	100	391	0,38	0,25	0,29	0,26	0,31	0,41	10,40
GR13:	wax100a	100	476	0,34	0,24	0,25	0,24	0,27	0,39	8,80

Tabela 4.37: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Internet`.

Grupo		$\|V\|$	$\|E\|$	G^{RR}, T^{RR}, T^{KT} Incr	G^{RR}, T^{RR}, T^{KT} Decr	T^D Incr	rhT^D Incr
GR1:	att	90	274	35,77	35,77	36,03	0,28
GR2:	hier50a	50	148	70,05	70,05	70,55	0,50
GR3:	hier50b	50	212	56,07	56,07	56,48	0,42
GR4:	hier100	100	280	230,86	230,86	232,47	1,61
GR5:	hier100b	100	360	125,21	125,21	127,04	1,86
GR6:	rand50	50	228	28,17	28,17	28,66	0,52
GR7:	rand50a	50	245	23,84	23,84	24,30	0,50
GR8:	rand100	100	403	84,72	84,72	86,61	1,93
GR9:	rand100b	100	410	82,02	82,02	83,95	1,98
GR10:	wax50	50	169	45,34	45,34	45,85	0,53
GR11:	wax50a	50	230	26,12	26,12	26,59	0,50
GR12:	wax100	100	391	86,80	86,80	88,75	1,98
GR13:	wax100a	100	476	68,09	68,09	69,85	1,83

Tabela 4.38: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Grid-SSquare-S`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	4098	16385	0,01	0,00	0,00	0,00	0,01	0,01	0,01	0,01	9,00
GR2	16386	65537	0,02	0,02	0,02	0,01	0,02	0,01	0,02	0,01	49,20
GR3	65538	262145	0,03	0,02	0,03	0,02	0,03	0,02	0,03	0,02	188,20
GR4	262146	1048577	0,03	0,02	0,03	0,02	0,03	0,02	0,02	0,02	748,80
GR5	1048578	4194305	0,04	0,03	0,04	0,03	0,04	0,04	0,04	0,03	2927,00

Tabela 4.39: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Grid-SSquare-S`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	4098	16385	0,00	0,01	0,01	0,00	0,00	0,01	7,60
GR2	16386	65537	0,02	0,01	0,01	0,01	0,01	0,01	48,80
GR3	65538	262145	0,03	0,02	0,02	0,02	0,02	0,01	187,80
GR4	262146	1048577	0,03	0,03	0,02	0,02	0,02	0,02	749,40
GR5	1048578	4194305	0,03	0,03	0,02	0,02	0,02	0,02	2927,60

Tabela 4.40: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Grid-SSquare-S`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	4098	16385	0,90	0,90	0,96	0,01
GR2	16386	65537	0,92	0,92	0,96	0,00
GR3	65538	262145	0,97	0,97	0,98	0,00
GR4	262146	1048577	0,99	0,99	0,99	0,00
GR5	1048578	4194305	1,00	1,00	1,00	0,00

Tabela 4.41: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Grid-SWide`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	0,07	0,03	0,06	0,02	0,06	0,03	0,06	0,03	44,00
GR2	16385	49152	0,09	0,05	0,08	0,05	0,09	0,05	0,08	0,05	108,40
GR3	32769	98304	0,11	0,07	0,10	0,06	0,10	0,06	0,10	0,06	298,00
GR4	65537	196608	0,11	0,07	0,10	0,06	0,10	0,06	0,10	0,07	884,80
GR5	131073	393216	0,12	0,08	0,12	0,07	0,11	0,07	0,11	0,07	2388,40
GR6	262145	786432	0,13	0,08	0,12	0,07	0,12	0,07	0,12	0,08	5326,40
GR7	524289	1572864	0,15	0,08	0,14	0,07	0,13	0,08	0,13	0,07	11361,00

Tabela 4.42: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Grid-SWide`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	0,05	0,03	0,03	0,02	0,03	0,04	43,80
GR2	16385	49152	0,08	0,05	0,05	0,04	0,05	0,04	108,60
GR3	32769	98304	0,09	0,06	0,06	0,05	0,06	0,06	298,00
GR4	65537	196608	0,10	0,07	0,07	0,05	0,07	0,06	885,00
GR5	131073	393216	0,10	0,07	0,07	0,06	0,08	0,07	2389,40
GR6	262145	786432	0,11	0,08	0,07	0,06	0,08	0,07	5328,80
GR7	524289	1572864	0,12	0,08	0,08	0,06	0,08	0,07	11351,40

Tabela 4.43: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Grid-PWide`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8193	24576	10,06	10,06	10,73	0,77
GR2	16385	49152	10,20	10,20	10,90	0,78
GR3	32769	98304	10,27	10,27	10,89	0,66
GR4	65537	196608	10,44	10,44	11,05	0,67
GR5	131073	393216	10,57	10,57	11,15	0,60
GR6	262145	786432	10,49	10,49	11,06	0,58
GR7	524289	1572864	10,38	10,38	10,90	0,53

Tabela 4.44: Tempo de CPU em segundos gasto em 10^4 incrementos unitários de peso nas instâncias da classe `Grid-SLong`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	2,03	0,49	2,00	0,45	2,02	0,52	1,92	0,54	29,40
GR2	16385	49152	4,52	1,12	4,51	0,99	4,56	1,25	4,18	1,07	63,00
GR3	32769	98304	10,13	2,45	10,05	2,10	10,18	2,76	9,32	2,36	128,80
GR4	65537	196608	21,29	4,85	21,18	4,18	21,50	5,55	20,14	4,89	251,20
GR5	131073	393216	44,50	9,59	43,94	8,26	44,70	11,01	41,78	9,56	498,00
GR6	262145	786432	91,41	18,74	90,24	16,21	91,52	21,60	86,21	19,22	1012,20
GR7	524289	1572864	258,94	49,06	255,45	42,53	258,06	54,36	242,56	47,26	2637,60

Tabela 4.45: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Grid-SLong`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	1,24	0,45	0,89	0,36	0,94	0,80	29,60
GR2	16385	49152	2,62	1,10	1,85	0,79	1,95	1,66	63,00
GR3	32769	98304	5,63	2,39	3,93	1,74	4,15	3,46	128,60
GR4	65537	196608	11,17	4,82	7,80	3,47	8,24	6,83	251,40
GR5	131073	393216	22,28	9,55	15,59	6,90	16,40	13,58	497,80
GR6	262145	786432	43,53	18,81	30,72	13,61	32,32	26,63	1012,40
GR7	524289	1572864	121,74	46,20	86,58	35,89	91,09	75,18	2637,40

Tabela 4.46: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Grid-SLong`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8193	24576	301,20	301,20	325,95	27,97
GR2	16385	49152	600,38	600,38	643,40	41,89
GR3	32769	98304	1231,78	1231,78	1316,80	90,95
GR4	65537	196608	2439,44	2439,44	2647,14	194,49
GR5	131073	393216	4858,52	4858,52	5211,27	349,32
GR6	262145	786432	9569,81	9569,81	10272,00	708,83
GR7	524289	1572864	19765,20	19765,20	20884,20	1142,01

Tabela 4.47: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Grid-PHard`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	63808	0,16	0,08	0,15	0,07	0,15	0,07	0,18	0,12	55,60
GR2	16385	129344	0,27	0,13	0,25	0,12	0,26	0,13	0,34	0,23	112,40
GR3	32769	260416	0,48	0,22	0,46	0,20	0,46	0,23	0,70	0,49	200,00
GR4	65537	522560	0,71	0,29	0,68	0,28	0,70	0,31	1,06	0,66	438,40
GR5	131073	1046848	1,28	0,48	1,28	0,48	1,30	0,55	2,05	1,29	833,20
GR6	262145	2095424	2,47	0,84	2,52	0,88	2,55	1,00	3,85	2,56	1556,20

Tabela 4.48: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Grid-PHard`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	63808	0,14	0,12	0,08	0,07	0,09	0,08	55,40
GR2	16385	129344	0,23	0,20	0,13	0,12	0,14	0,14	112,80
GR3	32769	260416	0,37	0,34	0,22	0,19	0,24	0,24	199,80
GR4	65537	522560	0,52	0,48	0,30	0,24	0,33	0,32	438,00
GR5	131073	1046848	0,88	0,84	0,49	0,40	0,53	0,55	833,00
GR6	262145	2095424	1,55	1,78	0,85	0,73	0,94	1,01	1555,80

Tabela 4.49: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Grid-PHard`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8193	63808	7,35	7,35	11,45	4,08
GR2	16385	129344	10,80	10,80	17,81	6,83
GR3	32769	260416	18,73	18,73	35,26	16,60
GR4	65537	522560	28,10	28,10	54,84	22,66
GR5	131073	1046848	53,16	53,16	107,66	50,38
GR6	262145	2095424	99,33	99,33	192,10	100,56

Tabela 4.50: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Rand-4`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	32768	0,17	0,06	0,16	0,06	0,16	0,07	0,15	0,07	81,00
GR2	16384	65536	0,26	0,11	0,25	0,11	0,25	0,12	0,24	0,12	262,40
GR3	32768	131072	0,33	0,15	0,33	0,16	0,34	0,17	0,33	0,19	733,60
GR4	65536	262144	0,72	0,31	0,72	0,33	0,74	0,35	0,67	0,34	1926,40
GR5	131072	524288	0,53	0,24	0,55	0,27	0,57	0,29	0,55	0,33	4430,60
GR6	262144	1048576	1,07	0,45	1,09	0,48	1,13	0,53	1,03	0,53	9422,00
GR7	524288	2097152	1,18	0,50	1,20	0,53	1,24	0,58	1,20	0,67	21037,40
GR8	1048576	4194304	0,79	0,35	0,82	0,39	0,83	0,42	0,84	0,49	44635,80

Tabela 4.51: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Rand-4`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	32768	0,14	0,10	0,08	0,06	0,08	0,07	80,60
GR2	16384	65536	0,22	0,17	0,13	0,11	0,14	0,12	261,80
GR3	32768	131072	0,28	0,21	0,17	0,15	0,18	0,16	732,80
GR4	65536	262144	0,52	0,37	0,33	0,28	0,34	0,29	1927,00
GR5	131072	524288	0,41	0,30	0,26	0,22	0,27	0,23	4426,40
GR6	262144	1048576	0,74	0,51	0,47	0,38	0,48	0,40	9425,00
GR7	524288	2097152	0,80	0,56	0,51	0,42	0,52	0,43	20992,60
GR8	1048576	4194304	0,58	0,41	0,35	0,31	0,36	0,32	44661,60

Tabela 4.52: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Rand-4`.

			G^{RR}, T^{RR}, T^{KT}		T^D	rhT^D
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8192	32768	9,90	9,90	11,49	1,64
GR2	16384	65536	10,36	10,36	12,35	1,98
GR3	32768	131072	10,40	10,40	12,45	2,12
GR4	65536	262144	15,42	15,42	17,62	2,18
GR5	131072	524288	12,27	12,27	14,77	2,56
GR6	262144	1048576	18,82	18,82	21,62	2,76
GR7	524288	2097152	19,64	19,64	23,64	4,08
GR8	1048576	4194304	14,22	14,22	17,40	3,15

Tabela 4.53: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Rand-1:4`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	512	65536	0,12	0,03	0,09	0,03	0,10	0,03	0,13	0,09	7,40
GR2	1024	262144	0,15	0,05	0,12	0,05	0,13	0,05	0,30	0,20	28,40
GR3	2048	1048576	0,12	0,05	0,11	0,05	0,10	0,05	0,23	0,16	91,20
GR4	4096	4194304	0,19	0,07	0,16	0,07	0,16	0,08	0,38	0,29	183,80

Tabela 4.54: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Rand-1:4`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	512	65536	0,12	0,11	0,05	0,06	0,07	0,07	7,40
GR2	1024	262144	0,16	0,15	0,09	0,09	0,10	0,10	28,40
GR3	2048	1048576	0,13	0,12	0,08	0,07	0,08	0,08	91,40
GR4	4096	4194304	0,19	0,19	0,11	0,11	0,12	0,12	183,20

Tabela 4.55: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Rand-1:4`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	512	65536	0,43	0,43	0,81	0,35
GR2	1024	262144	0,18	0,18	0,45	0,23
GR3	2048	1048576	0,06	0,06	0,14	0,07
GR4	4096	4194304	0,04	0,04	0,11	0,06

Tabela 4.56: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Rand-Len`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	131072	524288	0,72	0,31	0,72	0,32	0,74	0,34	0,65	0,33	4835,40
GR2	131072	524288	0,42	0,20	0,42	0,21	0,44	0,22	0,42	0,24	4257,60
GR3	131072	524288	0,59	0,27	0,59	0,27	0,61	0,30	0,55	0,30	4812,20
GR4	131072	524288	0,44	0,21	0,44	0,22	0,46	0,23	0,42	0,24	4880,80
GR5	131072	524288	0,62	0,28	0,62	0,28	0,65	0,31	0,57	0,30	4904,00

Tabela 4.57: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Rand-Len`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	131072	524288	0,53	0,34	0,35	0,27	0,36	0,27	4834,00
GR2	131072	524288	0,32	0,23	0,21	0,17	0,22	0,18	4259,00
GR3	131072	524288	0,44	0,30	0,29	0,23	0,30	0,24	4809,00
GR4	131072	524288	0,34	0,24	0,22	0,19	0,23	0,19	4880,20
GR5	131072	524288	0,46	0,31	0,31	0,24	0,32	0,25	4903,40

Tabela 4.58: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Rand-Len`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	131072	524288	12,78	12,78	13,77	0,99
GR2	131072	524288	8,45	8,45	9,91	1,49
GR3	131072	524288	11,29	11,29	12,39	1,16
GR4	131072	524288	9,25	9,25	10,14	0,92
GR5	131072	524288	11,65	11,65	12,65	0,99

Tabela 4.59: Tempos de CPU em segundos gastos em 10^4 incrementos unitários de pesos nas instâncias da classe `Acyc-Pos`.

Grupo	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	131072	0,38	0,12	0,33	0,13	0,33	0,13	0,34	0,16	258,40
GR2	16384	262144	0,47	0,17	0,42	0,17	0,42	0,17	0,45	0,23	663,00
GR3	32768	524288	0,85	0,30	0,85	0,35	0,85	0,33	0,87	0,41	1630,80
GR4	65536	1048576	1,29	0,46	1,38	0,59	1,38	0,54	1,36	0,65	4018,60
GR5	131072	2097152	1,10	0,39	1,19	0,54	1,21	0,51	1,23	0,65	9223,60

Tabela 4.60: Tempos de CPU em segundos gastos em 10^4 decrementos unitários de pesos nas instâncias da classe `Acyc-Pos`.

Grupo	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	131072	0,38	0,35	0,22	0,21	0,24	0,22	258,40
GR2	16384	262144	0,46	0,42	0,28	0,27	0,29	0,28	661,80
GR3	32768	524288	0,82	0,70	0,50	0,48	0,52	0,50	1630,80
GR4	65536	1048576	1,19	0,99	0,74	0,71	0,76	0,72	4018,80
GR5	131072	2097152	1,01	0,84	0,63	0,61	0,65	0,63	9238,40

Tabela 4.61: Tamanho da pilha para cada atualização (média de 10^4 mudanças unitárias de pesos) nas instâncias `Acyc-Pos`.

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Grupo	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8192	131072	5,36	5,36	6,37	1,07
GR2	16384	262144	5,29	5,29	6,33	1,07
GR3	32768	524288	7,73	7,73	9,09	1,44
GR4	65536	1048576	10,10	10,10	11,69	1,62
GR5	131072	2097152	8,06	8,06	9,53	1,48

Nas próximas subseções, discutem-se os resultados para mudanças unitárias do peso. O tipo de comparações são as mesmas realizadas em mudanças aleatórias de pesos e, por isso, os valores são calculados da mesma forma que as comparações correspondentes com mudanças aleatórias.

Melhoramentos obtidos usando-se as variantes de pilha-reduzida

A Tabela 4.62 compara os resultados obtidos pelos algoritmos clássicos e de pilha-reduzida para incremento unitário.

Tabela 4.62: Razão entre os tempos gastos pelos algoritmos *std* e *rh* para 10^4 mudanças unitárias de pesos.

Classe	G_+^{RR}	G_-^{RR}	T_+^{RR}	T_-^{RR}	T_+^{KT}	T_-^{KT}	T_+^{D}
`Internet`	2,00	1,58	2,12	1,15	2,05	0,75	1,83
`Grid-SSquare-S`	1,58	0,91	1,50	1,17	1,43	0,97	1,33
`Grid-SWide`	1,73	1,51	1,87	1,29	1,74	1,05	1,63
`Grid-SLong`	4,50	2,45	5,11	2,32	4,02	1,20	4,21
`Grid-PHard`	2,40	1,07	2,41	1,18	2,19	0,99	1,51
`Rand-4`	2,36	1,39	2,19	1,20	2,10	1,16	1,86
`Rand-1:4`	2,98	1,04	2,38	1,00	2,49	1,01	1,47
`Rand-Len`	2,20	1,46	2,13	1,26	2,04	1,25	1,85
`Acyc-Pos`	2,87	1,15	2,39	1,04	2,52	1,05	2,05
Média	2,51	1,40	2,46	1,29	2,29	1,05	1,97

Na média, para cada classe de instâncias, os algoritmos de pilha-reduzida diminuíram o tempo computacional em quase todas as comparações. Para todos os algoritmos de incremento G_+^{RR}, T_+^{RR} e T_+^{D}, as variantes de pilha-reduzida foram mais rápidas. A razão média para cada grupo variou de 1,33 para o algoritmo T_+^{D} da classe `Grid-SSquare-S` até 5,11 para T_+^{RR} na classe `Grid-SLong`. Para os algoritmos de decremento, somente três razões médias não foram favoráveis para a variante `rh`. Observando a última linha, na média todos os algoritmos foram capazes de reduzir o tempo usando a técnica de pilha-reduzida. A razão média variou de 1,05 para o algoritmoT_-^{KT} até 2,51 para o algoritmo G_+^{RR}.

Comparação de tempo entre calcular o grafo com Dijkstra (`Dij`) ou atualizá-lo com G^{RR}

A Tabela 4.63 apresenta resultados para a comparação entre os algoritmos `Dij` e G^{RR}. Como esperado, o algoritmo `Dij` gastou muito mais tempo que o algoritmo G^{RR}, variando de 10,33 vezes

Tabela 4.63: Razão entre o tempo gasto pelos algoritmos Dij e G^{RR} para atualizar 10^4 mudanças unitárias em cada grupo de todas as classes.

Grupo	Internet	Grid-SSquare-S	Grid-SWide	Grid-SLong	Grid-PHard	Rand-4	Rand-1:4	Rand-Len	Acyc-Pos
GR1	15,38	1383,33	719,67	18,01	362,75	521,29	61,67	7747,92	672,92
GR2	10,33	2722,22	1261,63	17,64	452,21	1092,08	182,05	11416,35	1430,67
GR3	10,91	6962,96	3104,17	16,33	468,15	2427,81	730,40	9340,97	1955,40
GR4	15,59	26753,57	8348,11	15,48	709,06	3117,64	976,06	12482,10	3230,47
GR5	20,58	83637,14	21329,46	14,91	770,68	9422,34		9030,76	8749,76
GR6	14,47		44769,75	15,00	773,36	10424,23			
GR7	17,14		86031,82	13,86		21163,14			
GR8	22,84					65180,58			
GR9	25,41								
GR10	13,27								
GR11	16,67								
GR12	24,31								
GR13	23,96								

para o Grupo GR2 da classe Internet até 83637 vezes para o grupo GR5 da classe Grid-SSquare-S. É interessante observar que as razões mudaram consideravelmente quando comparados com os valores apresentados para incrementos aleatórios. Este comportamento é devido ao intervalo escolhido para testes. Cherkassky, Goldberg e Radzik [CGR96] mostraram experimentalmente que os tempos dependem do intervalo de pesos sob consideração. Observa-se, por exemplo, que os tempos apresentados na Tabela 4.4 e 4.38 são bem diferentes. As instâncias são exatamente as mesmas, somente os pesos foram gerados em intervalos diferentes.

Comparação do desempenho dos algoritmos de caminhos mínimos dinâmicos

As tabelas comparando os algoritmos G^{RR} e T^{RR}, e T^{RR} e T^{KT}, mostram resultados similares aos apresentados nas Tabelas 4.33 e 4.31, respectivamente.

A Tabela 4.64 apresenta resultados comparativos do desempenho dos T_+^{RR} e T_+^D. A segunda

Tabela 4.64: Razão entre os tempos gastos pelos algoritmos T^D e T^{RR} para 10^4 mudanças unitárias de pesos.

	Tempo		Tamanho da pilha	
Classe	std	rh	std	rh
Internet	0,91	1,06	1,02	0,02
Grid-SSquare-S	1,17	1,55	1,03	0,00
Grid-SWide	0,97	1,10	1,06	0,06
Grid-SLong	0,95	1,15	1,07	0,07
Grid-PHard	1,47	2,35	1,83	0,44
Rand-4	0,98	1,15	1,18	0,16
Rand-1:4	2,09	3,39	2,26	0,50
Rand-Len	0,94	1,09	1,11	0,10
Acyc-Pos	1,04	1,21	1,18	0,16
Média	1,17	1,56	1,30	0,17

coluna apresenta resultados para a comparação de tempo, enquanto a terceira coluna compara os tamanhos das pilhas entre os algoritmos. Inicialmente, comparam-se os tamanhos das pilhas dos algoritmos clássicos. Em seguida, comparam-se os algoritmos std e rh do algoritmo T_+^D. Esta tabela apresenta explicitamente alguns exemplos onde o algoritmo T^D não tem bom desempenho quando comparado com os outros algoritmos de caminhos mínimos dinâmicos. Visto que estes experimentos são aplicados para instâncias com pesos gerados num intervalo pequeno, o algoritmo T_+^D trata muitos nós não afetados como se fossem. O pior desempenho deste algoritmo é observado

na classe `Rand-1:4`. Este algoritmo gastou mais que o dobro do tempo que o algoritmo T_+^{RR}, considerando a versão clássica dos algoritmos, e 3,39 vezes mais para o caso de pilha-reduzida. Observando-se o tamanho das pilhas, para os algoritmos clássicos, as pilhas usadas pelo algoritmo T_+^{D} são 2,26 vezes maior, na média, que as pilhas do algoritmo T_+^{RR}. Para as implementações de pilha-reduzida, enquanto o algoritmo rhT_+^{RR} não insere nenhum nó na pilha, o algoritmo T^D insere até 50% dos nós inseridos pela versão clássica.

Comparação entre os algoritmos de incremento e decremento

A Tabela 4.65 apresenta a razão entre os tempos gastos pelos procedimentos de incremento e decremento.

Tabela 4.65: Razão entre os tempos gastos pelos algoritmos *Incr* e *Decr* para 10^4 mudanças unitárias de pesos.

Classe	G^{RR}	rhG^{RR}	T^{RR}	rhT^{RR}	T^{KT}	rhT^{KT}
`Internet`	1,36	1,08	1,72	0,93	1,59	0,59
`Grid-SSquare-S`	2,00	0,91	1,39	1,11	2,43	1,29
`Grid-SWide`	1,21	1,06	1,72	1,20	1,67	1,00
`Grid-SLong`	1,90	1,03	2,67	1,21	2,56	0,77
`Grid-PHard`	1,34	0,61	2,29	1,12	2,13	0,96
`Rand-4`	1,31	0,78	2,14	1,18	2,14	1,19
`Rand-1:4`	0,97	0,35	1,46	0,61	1,35	0,56
`Rand-Len`	1,33	0,89	2,02	1,19	2,04	1,25
`Acyc-Pos`	1,05	0,42	1,68	0,74	1,63	0,68
Média	1,39	0,79	1,90	1,03	1,95	0,92

Para os algoritmos clássicos, os resultados apresentados nesta tabela são similares aos apresentados para incrementos aleatórios. No caso dos algoritmos `rh`, em média os algoritmos de decremento foram mais rápidos para os algoritmos G_-^{RR} e T^{KT}, e um pouco mais lentos para o algoritmo T^{RR}.

4.8 Conclusões

Neste capítulo foi proposta uma técnica para redução do tamanho das pilhas para ser usada pelos algoritmos de caminhos mínimos dinâmicos. A técnica pode ser usada para os algoritmos de incremento, assim como para os de decremento. A idéia principal é fazer atualizações usando

pilhas para um subconjunto dos nós utilizados pelos algoritmos clássicos. Considerando uma mudança do peso do arco $a = (\overrightarrow{u,v})$, e dado que acarreta uma mudança $\triangle$ no valor da distância do nó u ao destino, os algoritmos clássicos inserem na pilha todos os nós cujas distâncias mudaram. Denota-se por Q o conjunto destes nós. As variantes de pilha-reduzida correspondentes inserem na pilha somente o subconjunto de nós de Q cuja distância aumentou no máximo em $\triangle$. Para mudanças unitárias do peso dos arcos, as pilhas são desnecessárias para alguns algoritmos. Resultados computacionais foram apresentados para mudanças aleatórias e unitárias do peso de arcos.

Na média, todas as variantes de pilha-reduzida executaram mais rápido que suas correspondentes versões clássicas. Para mudanças aleatórias de peso, o ganho máximo foi de 1,79 para a classe `Internet` e o algoritmo T_+^D, enquanto que para incrementos unitários foi de 5,11 para a classe `Grid-SLong` e o algoritmo T_+^{RR}.

Comparando-se o algoritmo Dijkstra com o algoritmo `RR`, conclui-se que um algoritmo de atualização deve ser usado sempre que a rede é existente e apenas deseja-se atualizá-la. A diferença no tempo computacional varia entre 7 e 149 mil vezes, dependendo das características da instância, comparando-se os dois algoritmos.

A comparação entre os algoritmos de caminhos mínimos dinâmicos mostrou que atualizar árvores pode ser vantajoso, visto que é apenas um pouco mais rápido que atualizar grafos, tanto para o caso de mudanças aleatórias quanto para mudanças unitárias de pesos. Além disso, comparando-se os algoritmos de atualização de árvores, observou-se que, em média, não há vantagem em se usar o algoritmo T^{KT} em vez do T^{RR}, em termos de tempo computacional. Ainda, o algoritmo T^D pode ser considerado o mais rápido dos três algoritmos de incrementos para atualizar árvores para o caso em que os pesos são escolhidos num intervalo grande, mas no caso de pesos selecionados em um intervalo pequeno, seu desempenho pode ter uma piora considerável.

Como conclusões finais, não existe um único algoritmo de caminhos mínimos dinâmicos que seja o melhor para todos os casos. Entretanto, a escolha de qualquer um deles é melhor que usar o algoritmo de Dijkstra. A idéia de pilha-reduzida é indicada para os algoritmos de incremento e decremento, mesmo que em alguns casos isolados o algoritmo `rh` gastou um pouco mais de tempo que o `std` correspondente. No caso em que a aplicação não precisa de um grafo de caminhos mínimos, a atualização de árvores é um pouco mais rápida. A melhor escolha, considerando os

resultados computacionais apresentados, é a combinação dos algoritmos T_+^D para incremento de pesos e T_-^{RR} no caso de decremento, se a instância for gerada com pesos consistindo em valores grandes. Caso contrário, se o intervalo para geração dos pesos for pequeno, a combinação recomendada é T_+^{RR} e T_-^{RR}. Finalmente, concluiu-se que o desempenho dos algoritmos depende muito da classe e tamanho da instância.

Uma extensão deste trabalho seria a implementação, e comparação com os resultados aqui apresentados, do algoritmo de Even e Shiloach [ES81] para incremento de pesos.

Capítulo 5

Conclusões

Nesta tese foram estudados dois problemas, um de gerenciamento e outro de projeto, oriundos da engenharia de tráfego. Problemas desta natureza têm chamado mais atenção da comunidade acadêmica nos últimos anos pois, até dez anos atrás, eles não existiam e, atualmente, são cruciais no desempenho da rede Internet de uma corporação. Estes problemas têm origem no aumento expressivo do número de usuários, assim como no tamanho e quantidade de dados enviados e serviços oferecidos via Internet.

O roteamento de dados acontece na camada de rede da Internet. O roteamento é basicamente intra ou inter-domínios. Atualmente, o protocolo de roteamento padrão inter-domínios é o BGP, enquanto que o intra-domínio é o OSPF. Pode-se dizer que o protocolo OSPF é mais complexo que os demais. Esta conclusão pode ser obtida comparando-se as regras de cada um deles. O OSPF oferece possibilidades bastante amplas, tais como fluxo balanceado e atribuição de pesos no intervalo [1,65535], sem restrição de tamanho máximo de rota. Mas, o uso de tais flexibilidades oferece uma gama enorme de possibilidades para seu funcionamento. Assim sendo, os problemas dessa procedência são resolvidos através do uso de métodos sofisticados de otimização combinatória, que auxiliarão o administrador manter o funcionamento adequado da rede.

Considerando o gerenciamento da rede, esta tese abordou o problema de designação de pesos (*Weight Setting Problem* - WSP) aos arcos de uma rede Internet intra-domínio, sob roteamento OSPF. O WSP consiste em adequadamente designar pesos aos arcos da rede entre roteadores de modo que o tráfego possa escoar pelos caminhos de peso mínimo, fazendo um melhor uso dos recursos existentes para evitar a ocorrência de congestionamento. O objetivo é minimizar o con-

gestionamento da rede, homogeneizando a carga passante nos arcos. Embora este problema possua aplicação atual e sua otimização represente forte impacto no desempenho das redes provedoras, somente nos 3 últimos anos foi feito um esforço maior da área de otimização para resolvê-lo. Como contribuição desta tese, uma busca local foi proposta para sua inserção num algoritmo genético (*Genetic Algorithm* - GA), originando um algoritmo memético (*Memetic Algorithm* - MA). Os resultados computacionais mostram que o desempenho do MA é significativamente melhor que o do GA. Comparando-se com o método de busca tabu existente, o MA teve desempenho comparável. Ambos encontram-se muito próximos do limitante inferior para o problema, evidenciando o alto desempenho na resolução do problema apresentado por estes dois métodos.

Um problema de projeto também foi abordado. O problema de projeto de redes OSPF é ainda mais recente que o WSP. Visto que nenhuma referência dessa origem foi encontrada, considera-se que seja a primeira vez que um método de resolução é proposto para este problema. Para sua resolução, supõe-se receber a informação da topologia da rede, mas o número de cabos que irão compor cada arco (multiplicidade) deve ser definido. A definição das multiplicidades está intrinsecamente relacionada com custos de projeto, pois custos elevados estão associados à alocação de cada multiplicidade. Por isso, o objetivo do problema é definir a soma mínima das multiplicidades da rede suficiente para rotear a demanda, sem causar sobrecarga de nenhum arco, numa rede Internet intra-domínio sob roteamento OSPF. Várias modificações e adições foram apresentadas, mas as duas com maior impacto para o projeto de uma rede segura é a garantia de manter banda livre ociosa em cada arco e a possibilidade de não sobrecarga de arcos na ocorrência da falha de um arco, roteador ou conjuntos específicos de arcos. Ambas estão relacionadas com Qualidade de Serviço em redes OSPF, o qual objetiva ter redes mais eficientes (seguras, sem atrasos nas entregas de pacotes, etc). A banda ociosa é facilmente disponibilizada apenas considerando-se, nos cálculos que envolvem a definição das multiplicidades, uma capacidade $c_a^{\cdot} < c_a$ de cada arco $a \in E$. Para garantir o funcionamento da rede na ocorrência de falhas, cada falha deve ser simulada. Em cada falha, um ou um conjunto de arcos são desativados, aumentado a carga de outros arcos da rede. As multiplicidades devem ser computadas de forma a considerar estes casos. Estas duas adições ao problema, bem como outras que foram consideradas na abordagem deste problema na tese, acarretam o aumento da soma total das multiplicidades, mas geram uma rede mais segura. Uma proposta

de definição das multiplicidades foi proposta, assim como um algoritmo genético para minimizar a soma total. Ambas são contribuições originais desta tese.

Como já visto, a busca local do MA para o WSP é baseada em mudanças dos pesos de arcos da rede. Ainda, na simulação de falhas na rede, consideradas no problema de projeto, arcos devem ser removidos desta. Nestas duas situações, uma mudança simples ocorreu e deve ser refletida no grafo de caminhos mínimos. Este, por sua vez, não deve ser inteiramente recalculado, e sim atualizado com um algoritmo para caminhos mínimos dinâmicos (*Dynamic Shortest Paths* - DSP). Nesta tese foi apresentado um estudo minucioso e extenso de algoritmos de caminhos mínimos dinâmicos. Para os testes, foram consideradas 9 classes de instâncias, cada classe composta por diversos grupos (60 grupos no total) e cada grupo composto por 5 instâncias (300 instâncias no total). Para cada instância, 10^4 incrementos e 10^4 decrementos foram realizados para cada um dos 14 algoritmos de DSP. As dimensões das instâncias variaram de 50 a 1 milhão de nós, e 148 a 4 milhões de arcos. Para todos os algoritmos clássicos estudados, um algoritmo de pilha-reduzida foi proposto.

Neste estudo, várias conclusões foram obtidas pela comparação de resultados. A primeira conclusão, e a mais óbvia de todas, é que a atualização do grafo de caminhos mínimos é muito mais rápida que recalculá-lo inteiramente. Os ganhos variam de 7 vezes até 149 mil vezes, dependendo da classe e do tamanho das instâncias. Também concluiu-se que, na média, a técnica de pilha-reduzida diminuiu o tempo de todos os algoritmos. Comparou-se o desempenho dos algoritmos para atualizar grafos e árvores de caminhos mínimos e observa-se que os algoritmos de árvores executam um pouco mais rápido. Dos algoritmos de atualização de uma árvore de caminhos mínimos, o algoritmo T^D foi o mais rápido no caso geral, e o T_+^{RR} o mais rápido para o caso de incrementos unitários. A contribuição original deste estudo foi a técnica de pilha-reduzida proposta. Mas, tão importante quanto esta proposta, foi o estudo da aplicação destes algoritmos para instâncias tão diversas, além da comparação entre eles. Pouquíssimos estudos dessa natureza foram realizados anteriormente, sendo que todos consideravam instâncias pouco diversas e de dimensões reduzidas. Por exemplo, a maior diferença publicada entre o tempo computacional dos algoritmos de atualização e computação completa com o algoritmo Dijkstra, foi de 4000 vezes, enquanto nos resultados experimentais apresentados no Capítulo 4 uma diferença de mais de 149000 vezes foi

apresentada.

De forma geral, as heurísticas e algoritmos propostos nesta tese contribuíram significativamente para o desenvolvimento da pesquisa na área de engenharia de tráfego, assim como outras áreas que podem se beneficiar dos estudos aqui realizados. Sugestões de trabalhos futuros foram adicionadas ao final de cada capítulo.

Apêndice A - O Brasil no Contexto da Internet

A título de curiosidade, abaixo encontram-se listados alguns dos mais importantes sistemas autônomos nacionais (18) e estaduais (6) existentes no Brasil no final de 2002 (fonte [Tud01]).

- ASs Nacionais: Embratel, Rede Nacional de Pesquisa (RNP), Unisys, Global-One, Telemar/Pegasus, Brasil Telecom, KDD Nethal, Comsat Brasil, Impsat Comunicações, AT&T, NTT, MetroRED, Diveo do Brasil, CTBC, Mundivox do Brasil, Intelig, Atrium e IBM.
- ASs Estaduais: ANSP (SP), Rede Norte-riograndense de Informática (RN), Rede Pernambuco de Informática (PE), Rede Rio (RJ), Rede Tchê (RS) e REMAV (Redes Metropolitanas de Alta Velocidade).

Cada AS é conhecido mundialmente por um número (*Autonomous System Number* - ASN). Por exemplo, o AS da AT&T nos Estados Unidos é conhecido como `ASN 7018`, enquanto que o da Embratel é `ASN 4230`. Um AS pode possuir mais de um número, porém um número não pode pertencer a mais que um AS. No Brasil, o *Comitê Gestor da Internet no Brasil* [1], criado pelo governo brasileiro em 1990, é responsável pela gerência e administração de nomes de domínio, números IP, tráfego Internet, segurança operacional da rede, entre outras atribuições. A administração dos endereços IP e a definição do número de cada AS na área abrangida pela América Latina e Caribe, é responsabilidade de uma organização chamada LACNIC [2] (*Latin American and Caribbean Internet Addresses Registry*). Segundo dados do LACNIC, atualmente encontram-se registrados 220 ASNs no Brasil, sendo que alguns deles possuem mais de um ASN, enquanto que outros não estão mais em uso.

O primeiro AS estabelecido no Brasil foi o da RNP, em 1990, e o maior é o da Embratel. A Figura 5.1 apresenta um mapa do *backbone* inicial da RNP no Brasil, e a Figura 5.2 apresenta um mapa do *backbone* atual [3].

[1] http://www.cg.org.br/

[2] www.lacnic.net

[3] Estes mapas estão disponíveis em http://www.rnp.br/lang/backbone_en/index.html

Segundo dados obtidos na Embratel [4], atualmente ela dispõe do maior *backbone* Internet da América Latina, tanto em termos de abrangência (atingindo mais de 300 localidades), como em capacidade de circuitos de transmissão de dados. Segundo as notícias da Embratel[5], o OSPF logo será implantado com o protocolo intra-domínio, mas o protocolo vigente não foi claramente especificado (aparentemente existem vários protocolos em operação). O seu *backbone* principal é composto por 27 roteadores, sendo 8 deles localizados no Estado de São Paulo. A Figura 5.3 mostra visualmente onde estes roteadores estão localizados, enquanto que a Figura 5.4 mostra as ligações da Embratel fora do Brasil [6].

Mais detalhes dessas ligações encontram-se resumidas na Tabela 5.1. Esta tabela apresenta 4 colunas contendo informação sobre a administradora do AS, país a que pertence, o número de cabos que compõem a ligação (*multi*) e a capacidade de cada um dos cabos (medida em Mbps - milhões de bits por segundo). As primeiras 5 conexões estão localizadas em São Paulo, enquanto que as demais estão no Rio de Janeiro.

Tabela 5.1: Ligações do AS da Embratel, possuidora do maior AS brasileiro, fora do Brasil.

Administradora	*País*	*Multi*	*Capacidade*
UUNET	EUA	6	155 Mbps
Level3	EUA	4	155 Mbps
Sprint	EUA	2	155 Mbps
Telefônica da Argentina	Argentina	1	512 Kbps
Telecom Internacional Argentina	Argentina	2	1 Mbps
Antel	Uruguai	1	1Mbps
Radio Marconi	Portugal	4	2 Mbps
UUNET	Estados Unidos	11	155 Mbps
Level3	Estados Unidos	2	155 Mbps
Sprint	Estados Unidos	8	155 Mbps
UUNET	Argentina	1	45 Mbps

Atualmente, existem cerca de 11.500 registros de ASs no mundo. Os maiores pertencem a alguns provedores que se estendem por vários continentes, tais como o MCI, Sprint, Level3, AT&T, Qwest, Cable & Wireless (todos são americanos). O *backbone* da MCI é o maior do mundo, segundo Marsan [Mar03], e o `ASN 7018` da AT&T está entre os maiores do mundo. A parte do MCI responsável pela Internet nacional nos EUA e serviços no exterior é chamada de UUNET (ou MCI UUNET). A Figura 5.5 apresenta o *backbone* da UUNET na América Central e do Sul [7], enquanto que a Figura 5.6 apresenta o *backbone* global da MCI [8].

[4] http://www.embratel.net.br/internet/backbone/informacoes-backbone.html
[5] http://www.embratel.net.br/internet/info/og-centro.html
[6] Mapas disponíveis em http://www.embratel.com.br/internet/backbone/informacoes-backbone.html
[7] Disponível em http://global.mci.com/about/network/global_presence/latinamerica/
[8] http://www.servepath.com/why/network.htm#mci

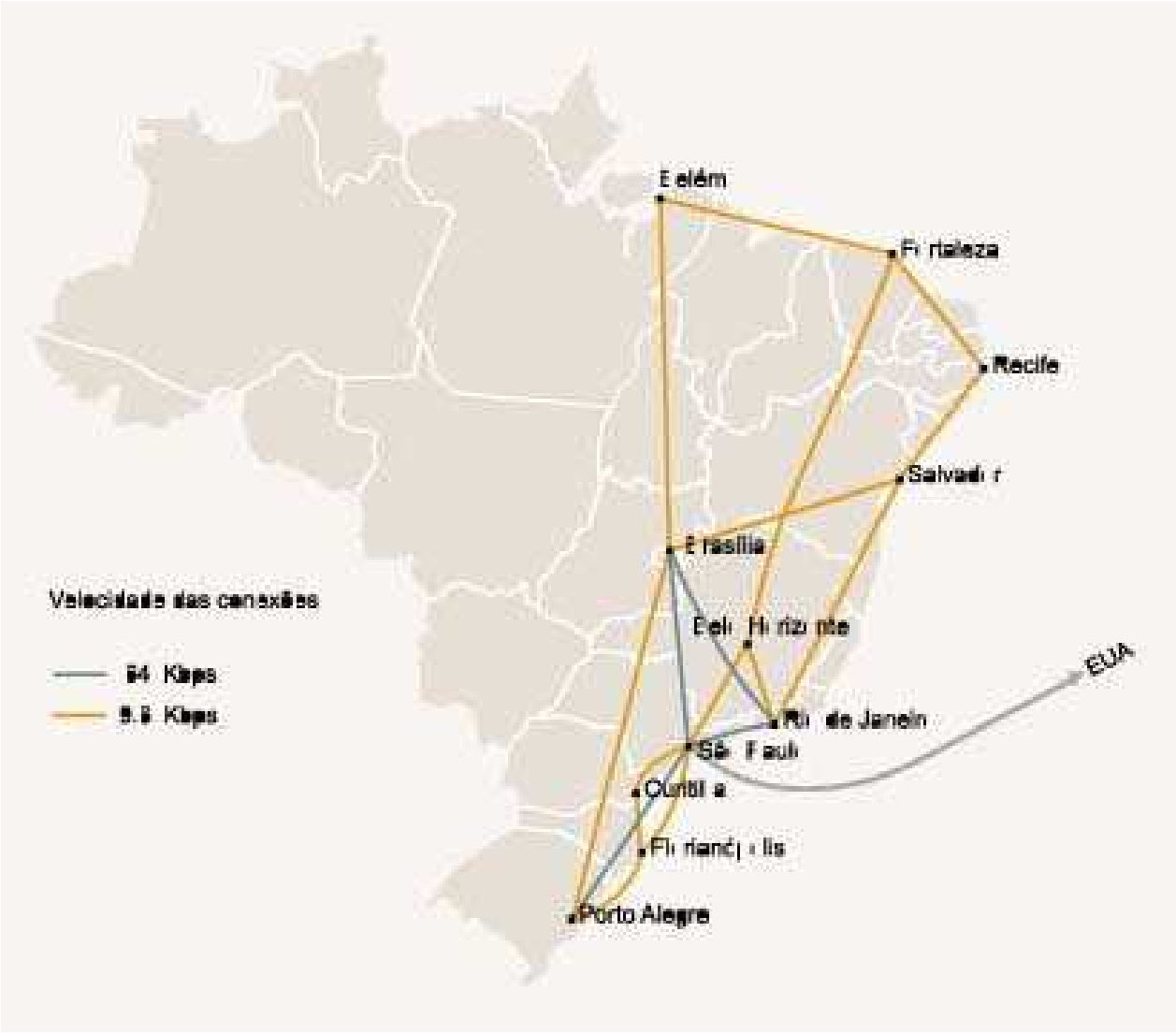

Figura 5.1: Mapa do *Backbone* inicial da rede RNP no Brasil.

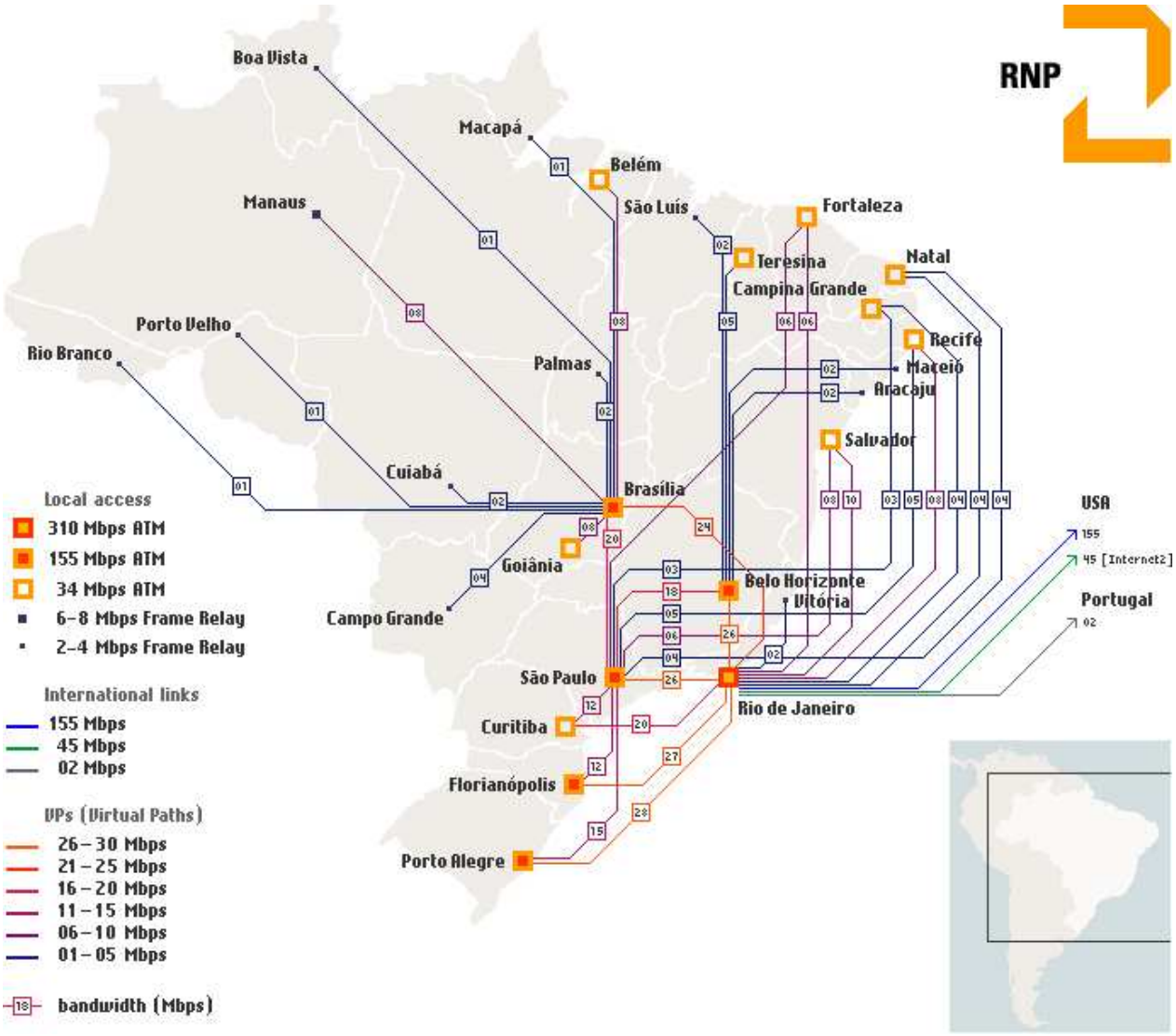

Figura 5.2: Mapa do *backbone* atual da rede RNP no Brasil.

Figura 5.3: Mapa do *Backbone* da Embratel no Brasil.

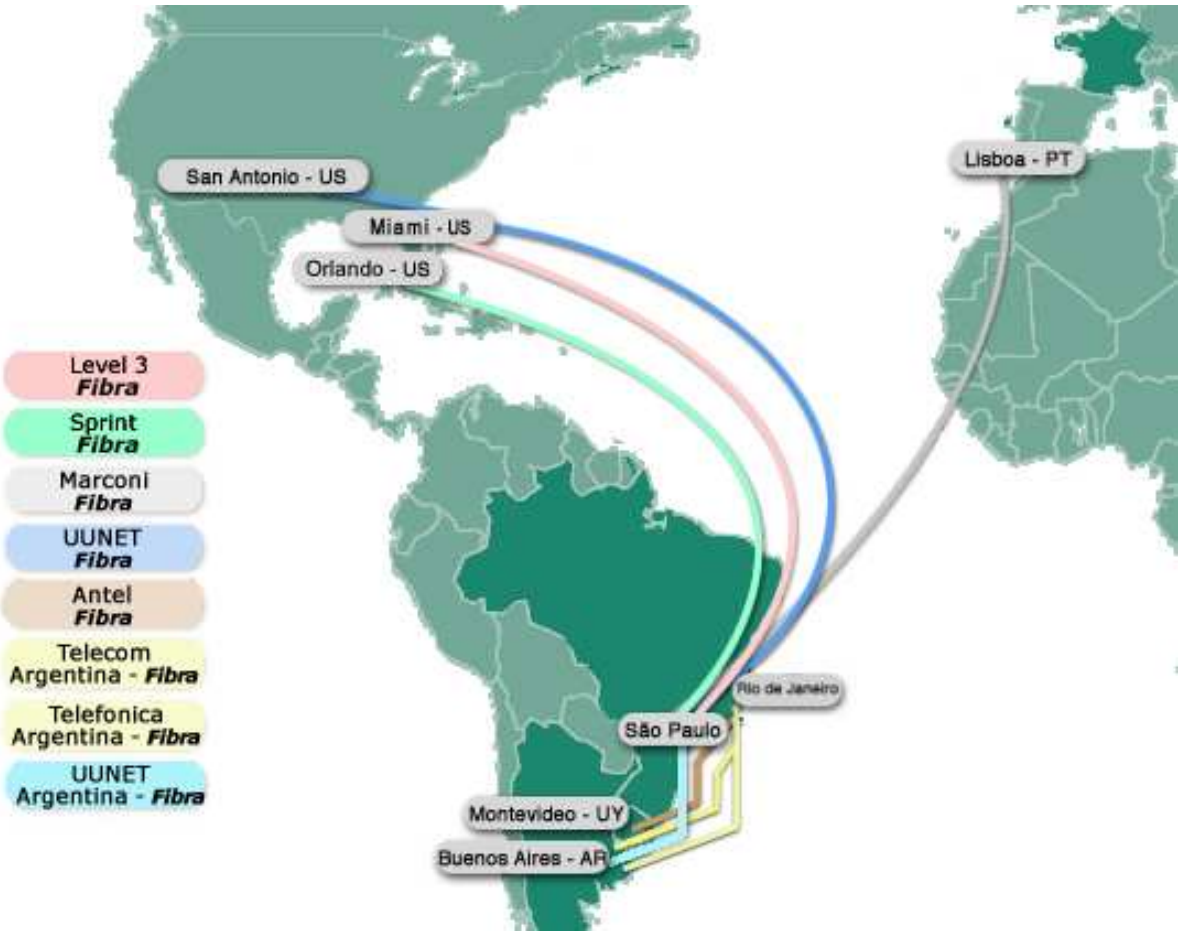

Figura 5.4: Mapa das Ligações da Embratel fora do Brasil.

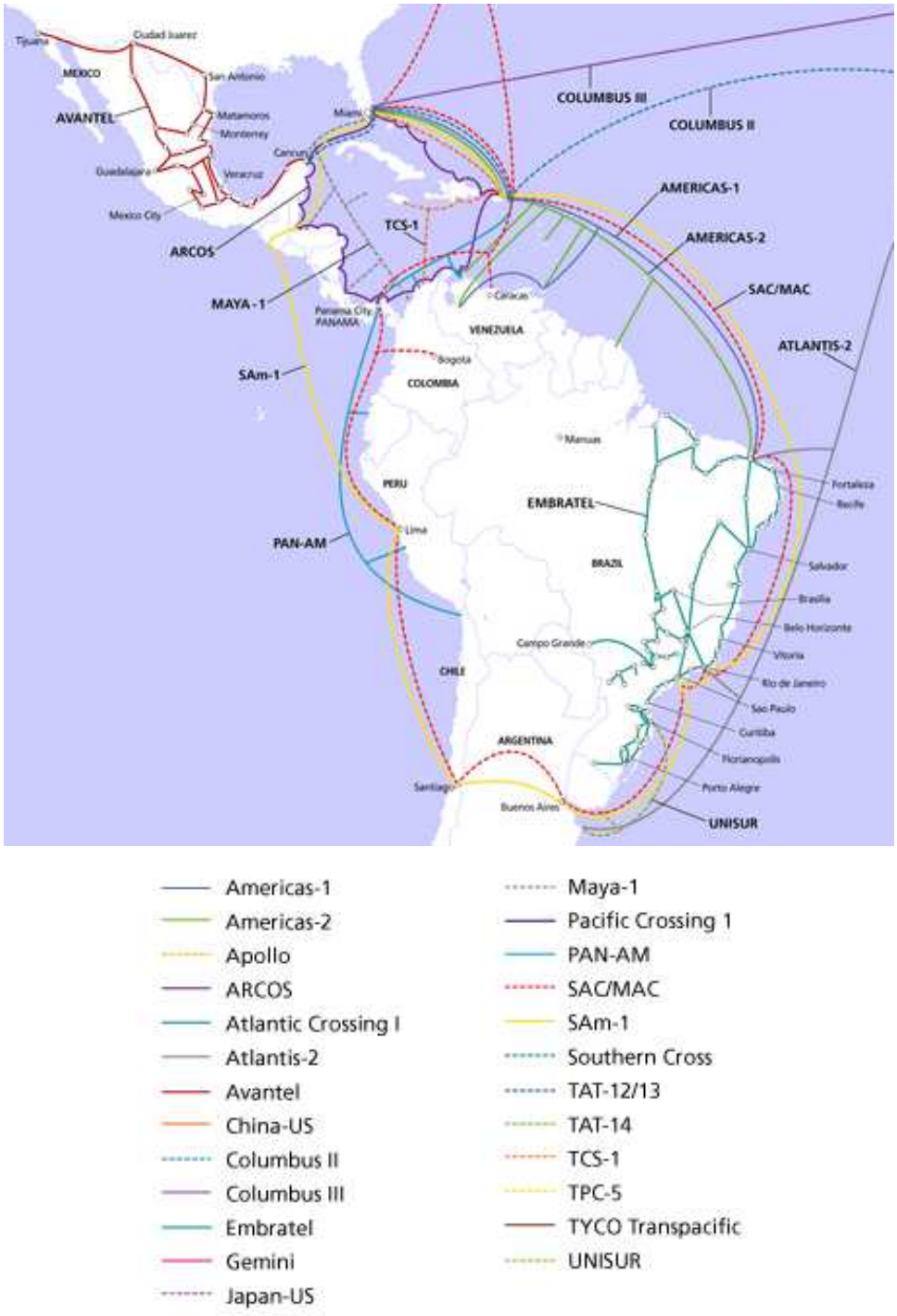

Figura 5.5: Map do *backbone* da MCI na América do Sul.

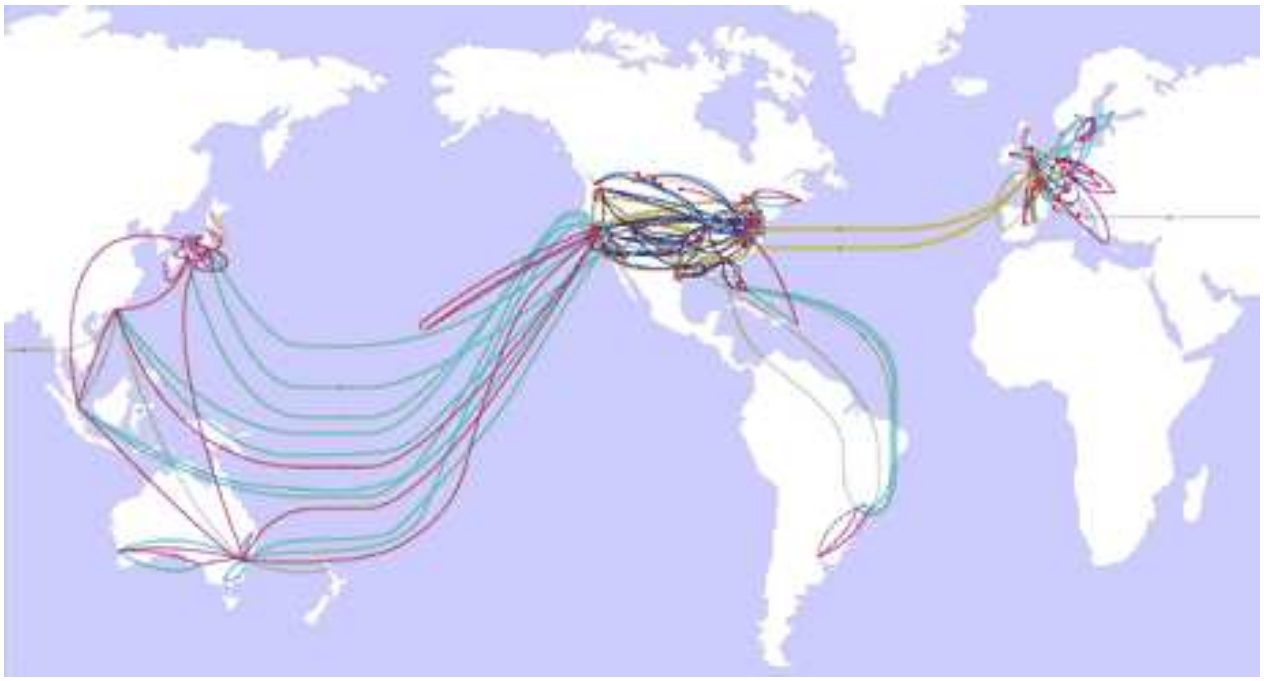

Figura 5.6: Mapa do *backbone* global da MCI.

Referências Bibliográfi cas

[AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.

[ARR02] R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro. Probability distribution of solution time in GRASP: An experimental investigation. *Journal of Heuristics*, 8:343–373, 2002.

[Bea94] J.C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6:154–160, 1994.

[BFRR03] L.S. Buriol, P.M. França, M.G.C. Resende, and C.C. Ribeiro. Otimizando o roteamento do tráfego na internet. In *Anais do XXXV Simpósio Brasileiro de Pesquisa Operacional*, 2003.

[BFRT03] L.S. Buriol, P.M. França, M.G.C. Resende, and M. Thorup. Designing networks for ospf routing. In *Proceedings of the Mathematical Programming in Rio: A Conference in Honour of Nelson Maculan*, 2003.

[BGW98] A. Bley, M. Grötchel, and R. Wessläy. Design of broadband virtual private networks: Model and heuristics for the B-WiN. In *Proceedings of the Fifth DIMACS Workshop on Robust Communication Network and Survivability, AMS-DIMACS Series*, pages 1–16, 1998.

[Bla99] U. Black. *Voice Over IP*. Cisco Press, 1999.

[BRRT03] L.S. Buriol, M.G.C. Resende, C.C. Ribeiro, and M. Thorup. A hybrid genetic algorithm for the weight setting problem in ospf/is-is routing. In *Networks*, 2003. under review.

[CDZ97] K. Calvert, M. Doar, and E.W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35:160–163, 1997.

[CGR96] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[Cis97] Cisco. *Configuring OSPF*. Cisco Press, 1997.

[DEI03] C. Demetrescu, S. Emiliozzi, and G. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Rome, Italy, 2003.

[Dem01] C. Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza", Italy, 2001.

[DFMSN00] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. *Algorithm Engineering*, pages 218–229, 2000.

[DI01] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 260–267, 2001.

[DI03] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 159–166, 2003.

[Dio78] R. Dionne. Etude et extension d'un algorithme de murchland. *INFOR*, 16:132–146, 1978.

[ERP02] M. Ericsson, M.G.C. Resende, and P.M. Pardalos. A genetic algorithm for the weight setting problem in OSPF routing. *Journal of Combinatorial Optimization*, 6:299–333, 2002.

[ES81] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the Association for Computing*, 28:1–4, 1981.

[FGL+01] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Transactions on Networking*, 9:265–279, 2001.

[FINP98] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest path problem. *ACM Journal of Experimental Algorithms*, 3, 1998.

[FMSN96] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem. In *Proceedings of the 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 212–221, 1996.

[FMSN98] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single-source shortest path trees. *Algorithmica*, 22:250–274, 1998.

[FMSN00] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:351–381, 2000.

[FRT02] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional ip routing protocols. *IEEE Communications Magazine*, 10:118–124, 2002.

[FT00] B. Fortz and M. Thorup. Increasing internet capacity using local search. Technical report, AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932 USA, 2000. Preliminary short version of this paper published as "Internet Traffic Engineering by Optimizing OSPF weights," in Proceedings of the IEEE Conf. on Computer Communications (INFOCOM).

[Fuj81] S. Fujishige. A note on the problem of updating shortest paths. *Networks*, 11:317–319, 1981.

[Gal80] G. Gallo. Reoptimization procedures in shortest path problems. *Rivista di Matematica per le Scienze Economiche e Sociali*, 3:3–13, 1980.

[Gro92] P. Gross. Choosing a comomon igp for the ip internet. Technical Report RFC 1371, Recomendação IESG para O IAB, 1992.

[GSV78] S. Goto and A. Sangiovanni-Vincentelli. A new shortest path updating algorithm. *Networks*, 8:341–372, 1978.

[Int94] Internet Engineering Task Force. Ospf version 2. Technical Report RFC 1583, Network Working Group, 1994.

[KT01] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorial Conference (COCOON) in LNCS 2108*, pages 268–277. Springer-Verlag, 2001.

[LW93] F. Lin and J. Wang. Minimax open shortest path first routing algorithms in networks supporting the smds services. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 666–670, 1993.

[Mar03] C.D. Marsan. Mci remains internet's largest backbone. *Network World ISP News Report Newsletter*, 2003.

[MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

[Mos02] P. Moscato. Memetic algorithms. In P.M. Pardalos and M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 157–168. Oxford University Press, 2002.

[Moy98] J.T. Moy. *OSPF, Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1998.

[MS02] A. Martey and S. Sturgess. *IS-IS Network Design Solutions*. Cisco Press, 2002.

[Mur70] J. D. Murchland. *A fixed matrix method for all shortest distances in a directed graph and for the inverse problem*. PhD thesis, Universität Karlsrube, Germany, 1970.

[RR94] M. Rodrigues and K.G. Ramakrishnan. Optimal routing in data networks, 1994. Presentation at International Telecommunication Symposium (ITS).

[RR96a] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1996.

[RR96b] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[RR03] M.G.C. Resende and C.C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.

[SARK02] L. Subramanian, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the internet hierarchy from multiple vantage points. In *Proceedings of the 21st IEEE Conference on Computer Communications (INFOCOM)*, volume 2, pages 618–627, 2002.

[SGD02] A. Sridharan, R. Guérin, and C. Diot. Achieving Near-Optimal Traffic Engineering Solutions for Current OSPF/IS-IS Networks. Sprint ATL Technical Report TR02-ATL-022037, Sprint Labs, February 2002.

[SLC95] L.F.G. Soares, G. Lemos, and S. Colcher. *Redes de Computadores: das LANs, MANs e WANs às Redes ATM*. Campus, segunda edição edition, 1995.

[Sri01] V. Srinivas. *IP Quality of Service*. Cisco Press, 2001.

[Tan97] A.S. Tanenbaum. *Redes de Computadores*. Campus, tradução da terceira edição edition, 1997.

[Tho98] T.M. Thomas II. *OSPF Network Design Solutions*. Cisco Press, 1998.

[Tud01] E. Tude. *Internet no Brasil*. Teleco, 2001. Tutorial disponível na página http://www.teleco.com.br/tutoriais.asp.

[Wax98] B.M. Waxman. Routing of multipoint connections. *IEEE J. Selected Areas in Communications (Special Issue on Broadband Packet Communication)*, 6:1617–1622, 1998.

[WZ93] D.L. Woodruff and E. Zemel. Hashing vectors for tabu search. *Annals of Operations Research*, 41:123–137, 1993.

[ZCB96] E.W. Zegura, K.L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pages 594–602, 1996.

[Zeg96] E.W. Zegura. GT-ITM: Georgia Tech internetwork topology models (software), 1996. `http://www.cc.gatech.edu/fac/Ellen.Zegura/gt-itm/gt-itm.tar.gz`.