Universidade Federal Fluminense

RAFAELLI DE CARVALHO COUTINHO

Algoritmos Distribuídos para o Problema de Atribuição de Clientes a Servidores em Redes de Distribuição de Conteúdos

NITERÓI

2011

Universidade Federal Fluminense

RAFAELLI DE CARVALHO COUTINHO

Algoritmos Distribuídos para o Problema de Atribuição de Clientes a Servidores em Redes de Distribuição de Conteúdos

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de Concentração: Algoritmos Distribuídos

Orientadora:

Lúcia Maria Assumpção Drummond

NITERÓI

2011

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

C871 Coutinho, Rafaelli de Carvalho
Algoritmos distribuídos para o problema de atribuição de clientes a servidores em redes de distribuição de conteúdos / Rafaelli de Carvalho Coutinho. – Niterói, RJ : [s.n.], 2011.
96 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2011.
Orientador: Lúcia Maria Assumpção Drummond.

1. Algoritmo distribuído. 2. Rede de distribuição de conteúdos. 3. Heurística. 4. Simplex de transporte. 5. Simplex de redes. I. Título.

CDD 005.136

Algoritmos Distribuídos para o Problema de Atribuição de Clientes a Servidores em Redes de Distribuição de Conteúdos

Rafaelli de Carvalho Coutinho

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de Concentração: Algoritmos Distribuídos

Aprovada por:

Profa. Lúcia M. A. Drummond / IC-UFF(Presidente)

Prof. Yuri Frota/ IC-UFF

Prof. Lucidio Cabral / UFPB

Niterói, 29 de agosto de 2011.

À minha família.

Agradecimentos

A Deus, primeiramente.

Aos meus pais por todo amor, esforço e dedicação empenhados na minha formação e na minha vida.

Ao meu namorado Ubiratam por todo apoio, ajuda e paciência durante o desenvolvimento deste trabalho.

A todos os meus familiares, em especial meus irmãos Ronaldo e Rodrigo, por estarem sempre presentes.

À minha orientadora Lúcia M. A. Drummond pelos ensinamentos, conselhos e incentivos, que forneceram condições para realização deste trabalho.

Aos professores Yuri Frota, Luidi Simonetti e Eduardo Uchoa, pelas sugestões que contribuíram para este trabalho.

Aos demais professores do IC-UFF, pelo conhecimento recebido no curso de mestrado.

Aos colegas do LOGIS e do IC-UFF, pela amizade e momentos de descontração.

À CAPES pelo apoio financeiro concedido.

Resumo

Uma Rede de Distribuição de Conteúdo (RDC) é uma rede sobreposta que mantém réplicas de conteúdos em servidores com o objetivo de diminuir o atraso, a carga dos servidores e o congestionamento da rede, melhorando a Qualidade de Serviço (QoS) percebida pelos clientes. Para implementar o serviço de uma RDC, diversos problemas podem ser considerados, como o Problema de Atribuir Clientes a servidores (PAC). Neste trabalho, o PAC é estudado como um Problema de Transporte e um algoritmo distribuído é proposto para solucioná-lo. O algoritmo é composto por uma heurística distribuída, chamada DistPAC, baseada nos métodos Canto Noroeste e Custo Mínimo, que são tradicionalmente usados para a obtenção de solução inicial viável para o Simplex de Transporte, e do algoritmo distribuído do Simplex de Transporte, chamado DistST. Os experimentos realizados sobre um conjunto de instâncias da literatura mostraram que o DistPAC obtém soluções próximas as ótimas e que o tempo de execução do DistST é comparável com a versão centralizada.

Palavres-chave: Problema de Transporte, Simplex de Transporte, Algoritmos Distribuídos, Redes de Distribuição de Conteúdos

Abstract

A Content Distribution Network (CDN) is an overlay network that replicates contents in servers with the objective of reducing the delay, server load and network congestion, improving the Quality of Service perceived by clients. To implement the service of a CDN, many problems can be considered as the Problem of Assigning clients to servers (PAC). In this work, the PAC is studied as a transportation problem and a distributed algorithm is proposed to solve it. The algorithm is composed of a distributed heuristic, called DistPAC, based on Northwest Corner and Minimum Cost methods, which are traditionally used to obtain initial feasible solution for the Transportation Simplex method, and the Transportation Simplex distributed algorithm, called DistST. Experiments executed on a set of instances of the literature showed that the DistPAC obtains solutions close to the optimum and that the DistST execution time is comparable with the centralized version.

Key words: Transportation Problem, Transportation Simplex Method, Distributed Algorithms, Content Distribution Network

Sumário

Lista de Figuras viii

Lista de Tabelas ix

1 Introdução 10

1.1 Objetivos e Contribuições do Trabalho 11

1.2 Organização do Trabalho 12

2 Descrição do Problema e Formulação Matemática 13

2.1 Descrição do Problema 13

2.1.1 Formulação Matemática 14

3 Trabalhos Relacionados 16

3.1 Literatura sobre Redes de Distribuição de Conteúdos 16

3.2 Literatura sobre Simplex de Redes 19

4 Algoritmo Sequencial 23

4.1 Solução Inicial 24

4.2 Segunda Etapa do Simplex de Transporte 25

4.3 Análise de Complexidades 29

4.3.1 Solução Inicial 30

4.3.2 Simplex de Transporte 32

4.4 Exemplo 34

5 Algoritmo Distribuído **39**

5.1 Algoritmo DistPAC 40

5.1.1 Variáveis 40

5.1.2 Descrição do Algoritmo 41

5.2 Algoritmo DistST 45

5.2.1 Variáveis 45

5.2.2 Descrição do Algoritmo 47

5.3 Análise de Complexidades 68

5.3.1 Solução Inicial 68

5.3.2 Simplex de Transporte 69

6 Resultados Computacionais **73**

6.1 DistPAC 74

6.2 DistST 76

7 Conclusões e Trabalhos Futuros **80**

Referências **82**

Apêndice A – Pseudocódigos **84**

A.1 Construção da árvore de solução dual 84

A.2 Seleção de Ciclo 86

A.3 Cancelamento de Ciclo 89

A.4 Reconstrução da árvore de solução 92

Lista de Figuras

2.1 (a) Grafo bipartido completo (b) Atribuições cliente-servidor 14

4.1 Ciclo relativo à variável que entra na solução: Passo Linha e Passo Coluna 29

5.1 Exemplo de solução 44

5.2 Envio das mensagens VARV e VARU 49

5.3 (a) Solução degenerada (b) Propagação das variáveis duais parada 51

5.4 (a) Resolvendo o caso de degeneração (b) Envio das mensagens ATENDCOMPLETO e ATENDCOMPREQ 52

5.5 (a) Envio das mensagens ATENDCOMPLETO e ATENDCOMPREQ (b) Árvore da solução 53

5.6 Envio da mensagem CREDUZIDO 56

5.7 (a) Propagação da mensagem PCICLO (b) Propagação da mensagem ATUALIZACICLO 56

5.8 (a) Propagação da mensagem PCICLO do servidor 1; (b) Propagação da mensagem PCICLO do servidor 3; (c) Propagação da mensagem CANCELACICLO e CANCELACICLOF; (d) Continua a propagação da mensagem PCICLO do servidor 3 61

5.9 (a) Envio da mensagens ATUALIZAPAI e RECONSTROI (b) Nova propagação das mensagens VARU e VARV 64

5.10 Envio da mensagem ATUALIZAU 66

Lista de Tabelas

3.1 Literatura sobre RDC 18

3.2 Literatura sobre Simplex 21

4.1 Tabela de Transporte 23

4.2 Tabela de Transporte com as variáveis duais 25

4.3 Exemplo para o Simplex de Transporte 34

4.4 Requisição Artificial 34

4.5 Selecionando células de menor custo 36

4.6 Solução Inicial Encontrada 36

4.7 Construção da solução dual correspondente 37

4.8 Calculando as variáveis duais da primeira linha 37

4.9 Calculando as variáveis duais da segunda linha 37

4.10 Calculando as variáveis duais da terceira linha 37

4.11 Calculando os custos reduzidos 37

4.12 Selecionando o ciclo da variável que entra na solução 38

4.13 Selecionando a variável que sai da solução 38

4.14 Obtenção de uma nova solução 38

4.15 Calculando as variáveis duais novamente 38

4.16 Nenhum custo reduzido negativo é encontrado: Solução Ótima 38

6.1 Solução Inicial: Sequencial x Distribuído 75

6.2 Simplex de Transporte: Sequencial x Distribuído 77

6.3 Número de Mensagens e Tempo Global: DistPAC e DistST 79

Capítulo 1

Introdução

Com a popularização da internet, o número de usuários aumentou bastante nos últimos anos, e consequentemente, a demanda por conteúdos na rede. Assim, pesquisas têm sido realizadas para conseguir atender, com Qualidade de Serviço (QoS), o crescente número de requisições por conteúdos. As pesquisas se concentram na utilização de Redes de Distribuição de Conteúdos (*Content Distributed Network*) para o fornecimento dos conteúdos aos usuários finais (clientes).

Uma Rede de Distribuição de Conteúdo (RDC) é um sistema de computadores que possui cópias de conteúdos posicionadas em vários pontos de uma rede com o objetivo de maximizar a utilização da largura de banda (i.e. o acesso a estes dados) por clientes em toda a rede. Por exemplo, é preferível um cliente acessar a cópia de um conteúdo através do servidor mais próximo a ele do que buscar esta informação no servidor central, provocando a sobrecarga deste. Dentre os diversos tipos de conteúdos pode-se citar: objetos da Web, objetos disponíveis para download (arquivos de mídia, software, documentos) e aplicações *streaming* de mídia em tempo real.

Com o objetivo de implementar o serviço de uma RDC, uma série de problemas precisam ser considerados. Um deles é o de encontrar as melhores localizações em uma rede para posicionar os servidores. Este problema é denominado Problema de Posicionamento de Servidor (PPS) [5]. Um outro problema inerente a RDC é o Problema de Posicionamento de Réplicas (PPR) que consiste em encontrar os melhores servidores dentro da rede para colocar as réplicas dos conteúdos, e desta forma, atender as requisições [18]. Para isto, é necessário o conhecimento das posições dos servidores, o tamanho dos conteúdos e as informações sobre as requisições. O Problema de Atribuir Clientes a servidores (PAC) consiste em, dadas as requisições dos clientes e os posicionamentos das réplicas dos conteúdos, encontrar as atribuições cliente-servidor (i.e. que servidor irá atender que

cliente) considerando a disponibilidade dos conteúdos, a carga nos servidores e o custo de comunicação dos enlaces. É importante notar que as requisições dos clientes podem se modificar com o tempo, assim as atribuições cliente-servidor precisam ser atualizadas de forma coordenada e eficaz para garantir uma melhor QoS aos clientes da rede.

Neste trabalho, o PAC foi estudado e uma modelagem baseada no Problema de Transporte [1] foi proposta. Este problema é muito estudado na área de fluxo de redes e pode ser resolvido de forma exata por programação linear através de um algoritmo especializado conhecido como Simplex de Transporte [8]. O Simplex de Transporte requer que uma solução inicial viável, de preferência de boa qualidade, seja calculada. Tipicamente, os métodos do Canto Noroeste ou do Custo Mínimo são utilizados para encontrar esta solução inicial. Note que em uma RDC as informações de localização de conteúdos e requisições estão distribuídas entre os servidores e podem sofrer modificações no decorrer do tempo. Assim, o custo para coletar, armazenar e atualizar estas informações em um nó central pode ser proibitivo. Neste contexto, a utilização de algoritmos distribuídos executados em servidores que possuem um conhecimento limitado sobre a RDC pode ser uma alternativa atraente para a solução do PAC com eficiência.

1.1 Objetivos e Contribuições do Trabalho

Este trabalho tem como objetivo principal o desenvolvimento de um algoritmo eficiente para resolver o PAC utilizando a abordagem do Problema de Transporte através do algoritmo do Simplex de Transporte de maneira distribuída.

As principais contribuições desse trabalho se resumem nos seguintes pontos:

- modelagem do PAC como um Problema de Transporte;
- adaptação dos algoritmos sequenciais do Canto Noroeste e do Custo Mínimo para solução do PAC;
- proposta da heurística distribuída baseada nos algoritmos sequenciais do Canto Noroeste e do Custo Mínimo, DistPAC;
- proposta de um algoritmo distribuído para o Simplex de Transporte adaptado para o PAC, DistST, e avaliação experimental sobre um conjunto de instâncias da literatura.

1.2 Organização do Trabalho

O trabalho está dividido em 7 capítulos incluindo a introdução.

O Capítulo 2 apresenta a descrição e a formulação matemática para o problema abordado neste trabalho, o Problema de Atribuir Clientes a Servidores em uma Rede de Distribuição de Conteúdos.

No Capítulo 3, os trabalhos relacionados ao problema abordado presentes na literatura são apresentados e descritos.

No Capítulo 4 é apresentado o algoritmo sequencial do Simplex de Transporte modificado para resolver o PAC com as heurísticas Canto Noroeste e Custo Mínimo e a análise de complexidade do algoritmo.

O Capítulo 5 apresenta o algoritmo proposto, que consiste de uma heurística distribuída baseada nos métodos Canto Noroeste e Custo Mínimo, denominada DistPAC, do algoritmo distribuído para o Simplex de Transporte distribuído, denominado DistST, e a análise de complexidades do algoritmo.

No Capítulo 6 são apresentados os resultados computacionais e no Capítulo 7 são apresentadas as conclusões e os possíveis trabalhos futuros.

Capítulo 2

Descrição do Problema e Formulação Matemática

Neste capítulo é apresentada a descrição e a formulação matemática baseada no Problema de Transporte para o problema abordado.

2.1 Descrição do Problema

O PAC consiste em atribuir as requisições de clientes aos servidores de maneira que todas as requisições sejam atendidas. Algumas considerações feitas sobre a RDC são: a rede possui conteúdos pré-posicionados; podem existir cópias de conteúdos em servidores distintos e os servidores possuem largura de banda limitada.

Neste trabalho, o modelo matemático foi baseado no Problema de Transporte, que é um problema de otimização, em que itens devem ser transportados de pontos de origem para pontos de destino, as origens possuem ofertas limitadas do item e os destinos, demandas pré-determinadas. O objetivo é minimizar o custo de transporte. Este problema pode ser definido sobre um grafo bipartido completo $G = (V, A)$, em que o conjunto de vértices V está associado aos n pontos de origem e m pontos de destino, e o conjunto de arcos A possui um arco para cada par origem-destino. Assim, para cada vértice origem i, b_i indica seu limite de oferta, para cada vértice destino j, d_j é a sua demanda e para cada arco entre a origem i e o destino j existe um custo c_{ij} associado. O total de ofertas das origens deve ser igual ao total de demandas dos destinos, constituindo um sistema equilibrado. A Figura 2.1(a) ilustra este grafo.

Este problema pode ser utilizado para modelar o PAC da seguinte maneira. Os servidores constituem os vértices de origem e as requisições dos clientes, os vértices de

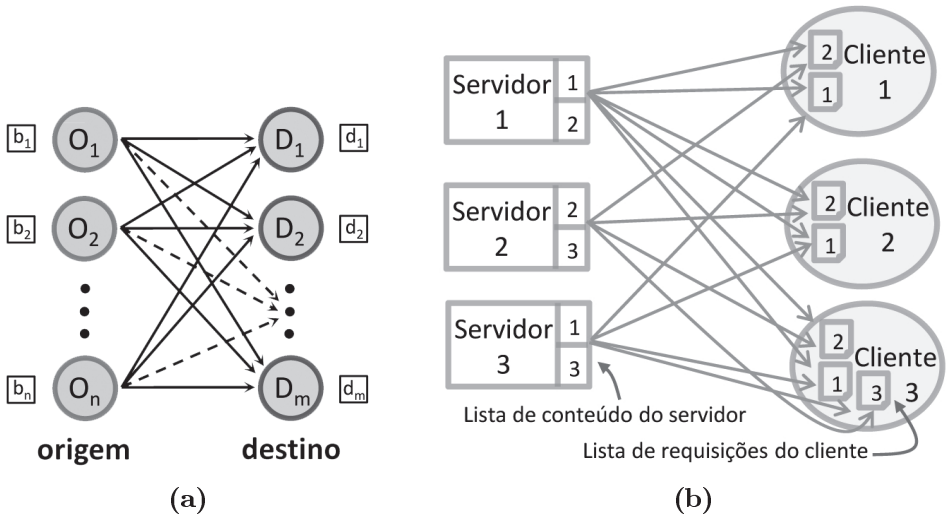

Figura 2.1: (a) Grafo bipartido completo (b) Atribuições cliente-servidor

destino. As demandas e as ofertas são, respectivamente, a largura de banda do conteúdo (requisito de QoS do conteúdo) e a largura de banda do servidor. Por exemplo, na Figura 2.1(b), existem 3 servidores. Um servidor possui cópia dos conteúdos 1 e 2, outro possui cópia dos conteúdos 2 e 3 e um outro possui cópia dos conteúdos 1 e 3. Existem também, 7 requisições por conteúdos de 3 clientes distintos. Os clientes 1 e 2 possuem requisições dos conteúdos 1 e 2 e o cliente 3 possui requisições dos conteúdos 1, 2 e 3. No entanto, os vértices e os arcos que formam o PAC (Figura 2.1(b)) não representam um grafo bipartido completo como no Problema de Transporte, pois nem todos os servidores possuem cópias de todos os conteúdos. Assim, para modelar o PAC baseando-se no Problema de Transporte foi necessário inserir arcos artificiais com custos de comunicação infinitos entre vértices origem e destino não adjacentes. Estes custos infinitos indicam que o conteúdo da requisição não está presente no servidor associado. Os demais arcos têm o custo correspondente ao atraso do enlace. Assim, o objetivo é atribuir clientes a servidores de forma a minimizar o custo de comunicação.

2.1.1 Formulação Matemática

Nesta seção, a formulação para o PAC como um Problema de Transporte é apresentada formalmente. A notação abaixo foi utilizada na formulação do problema:

Conjuntos

N	Conjunto de servidores, indexado por i
M	Lista de requisições, indexada por r

K	Conjunto de clientes, indexado por j
C	Lista de conteúdos, indexada por c
R_j	Lista de requisições do cliente j
C_i	Lista de conteúdos do servidor i

Dados

$c(r)$	Conteúdo da requisição r
$j(r)$	Cliente da requisição r
d_r	Banda do conteúdo da requisição r
b_i	Largura de banda disponível no servidor i
c_{ij}	Custo de comunicação do enlace (i, j)

Variável

$x^r_{ij(r)}$	Tráfego da requisição r que passa pelo enlace $(i, j(r))$

O custo de comunicação dos enlaces ij é infinito quando o conteúdo $c(r)$ da requisição r de j não pertence a C_i. O objetivo do problema é determinar o tráfego de cada requisição que passa por cada enlace minimizando o custo de comunicação (Equação 2.1). Para isso, o tráfego total de uma requisição por um conteúdo tem que ser equivalente a largura de banda deste conteúdo (Equação 2.2) e o tráfego de todas as requisições atendidas por um servidor não pode exceder a sua banda (Equação 2.3). Assim, a formulação é escrita da seguinte maneira:

$$min \sum_{i \in N} \sum_{j \in K} \sum_{r \in R_i} c_{ij} x^r_{ij} \tag{2.1}$$

sujeito a:

$$\sum_{i \in N} x^r_{ij} = d_r, \forall j \in K, r \in R_j \tag{2.2}$$

$$\sum_{j \in K} \sum_{r \in R_j} x^r_{ij} \leq b_i, \forall i \in N \tag{2.3}$$

$$x^r_{ij} \geq 0, \forall i \in N, j \in K, r \in R_j \tag{2.4}$$

Capítulo 3

Trabalhos Relacionados

A estratégia de solução apresentada neste trabalho utiliza o Simplex de Transporte para resolver um problema em RDC, o que permite a divisão dos trabalhos relacionados em dois grupos distintos: RDC e método Simplex de Rede.

3.1 Literatura sobre Redes de Distribuição de Conteúdos

Nesta seção, uma breve descrição de alguns trabalhos existentes sobre o PAC em RDC é apresentada e para auxiliar a leitura, um resumo dos mesmos foi colocado na Tabela 3.1.

Shah et al. abordam o PAC, atribuindo clientes a servidores de forma a minimizar os atrasos de comunicação [16]. Eles consideram uma RDC formada por um conjunto de fontes, servidores e clientes, onde fontes e servidores cooperam entre si para disseminação dos conteúdos. Neste sistema, as mudanças são passadas das fontes para os servidores e dos servidores para os clientes. Os clientes e os servidores associam requisitos de coerência com um conteúdo d, denotando o desvio máximo permitido sobre conhecimento do conteúdo d neles em relação a fonte. Consideram também que os conteúdos estão posicionados nos servidores e que para um conteúdo d, tem-se um conjunto de n requisições e r servidores que as atendem. Assim, dada uma lista <conteúdo,coerência> de servidores e uma lista <cliente, conteúdo, coerência> de requisições, o objetivo é encontrar as atribuições cliente-servidor que maximizem o grau do requisito de coerência conhecido pelos clientes. Para isto, dois algoritmos foram propostos e são descritos a seguir. O primeiro é baseado em um algoritmo para o problema de *matching*, utilizando fluxo máximo de custo mínimo. No sistema considerado, cada cliente pode obter o conteúdo requisitado de um servidor diferente. Assim, atribuir clientes a servidores, neste caso, é um problema de atribuição

de muitos para um, que pode ser resolvido usando fluxo em rede de custo mínimo. O custo de uma atribuição é uma função do atraso de comunicação, requisito de coerência do cliente e do servidor, e da carga dos servidores. Como a entrada do problema de fluxo em rede de custo mínimo é uma rede com vértices e arestas, os vértices são representados pelos servidores e pelas requisições <cliente, conteúdo, coerência>, existe um vértice fonte onde o fluxo origina e um vértice sumidouro (*sink*) onde o fluxo é absorvido e existe, também, uma aresta do vértice fonte para cada vértice requisição, uma aresta de cada vértice requisição para todo vértice servidor e por último, uma aresta de cada vértice servidor para o vértice *sink*. A garantia de que cada requisição é atribuída a, no máximo, um servidor, é assegurada através da capacidade atribuída às arestas entre o vértice fonte e os vértices de requisição e às arestas entre os vértices de requisição e os vértices de servidor que as atendem. O segundo algoritmo é baseado em um método para o problema de *stable-marriages*. O problema de *stable-marriages* considera conjuntos de n homens e n mulheres em que cada um classifica os membros do sexo oposto por preferência. O problema tem como objetivo encontrar um casamento estável entre os conjuntos, formando pares de homens com mulheres, nos quais não exista nenhum homem m e nenhuma mulher w que m prefira w como sua parceira atual e que w prefira m como seu parceiro atual. Uma vez que os pares com somente homens e com somente mulheres são excluídos, o problema pode ser representado por um grafo bipartido. O algoritmo utilizado pelos autores para o problema *stable-marriages* é um algoritmo iterativo, conhecido como Gale-Shapley. Associando o problema *stable-marriages* com o PAC, tem-se que os clientes que requisitam um conteúdo representam os homens e os servidores que atendem as requisições representam as mulheres. Como consiste de um problema de atribuição de muitos para um, o casamento é poliândrico [1]. Assim, os servidores e as requisições representam um grafo bipartido em que as requisições <cliente, conteúdo, coerência> escolhem servidores usando uma função de preferência baseada no atraso de comunicação, carga dos servidores e requisito de coerência. Então, se existe n servidores, pode-se atribuir n requisições a estes servidores usando o algoritmo *stable-marriages*.

Em [12] são apresentados uma formulação matemática baseada no problema de transporte para o problema PPS, e dois protocolos de redes para o PAC. A formulação é usada para encontrar a alocação ótima entre servidores e populações de clientes com o objetivo de formar *clusters* de servidores. Em seguida, dois protocolos realizam a manipulação das requisições nos *clusters* de servidores. Um protocolo utiliza um servidor central para distribuir as requisições de maneira uniforme no *cluster* e o outro protocolo, a manipulação

[1] Poliândrica: sociedade onde a mulher possui vários maridos

é feita de forma totalmente distribuída.

Bektas et al. propuseram abordagens heurísticas centralizadas e exatas para os problemas PPS, PPR e PAC com o objetivo de minimizar o custo total de distribuição [5]. A seguir, em [4], duas novas abordagens são apresentadas: uma baseada em decomposição de Benders e outra baseada em relaxação Lagrangeana e decomposição, para a solução do problema.

Tabela 3.1: Literatura sobre RDC

Problema	**Ano**	**Método**	**Contribuição**
Os clientes são atribuídos a servidores minimizando os atrasos de comunicação [16]	2005	Heurística Centralizada	Adaptações em duas soluções da literatura para o problema de *matching*: o fluxo máximo de custo mínimo e *stable-marriages*.
PPS para aplicações multimídias na Internet.[12]	2006	Exato e Protocolos de Redes	Formulação matemática baseada no Problema de Transporte para resolver o PPS e dois protocolos de rede para atender as requisições.
Resolver os problemas PPS, PPR e PAC com objetivo de minimizar o custo de comunicação. [5]	2007	Exato e Heurística Centralizada	Solução simultânea dos três problemas.
Replicar os conteúdos nos servidores e rotear requisições a um servidor minimizando o custo total de distribuição. [4]	2008	Heurística Centralizada	Dois algoritmos, um baseado na decomposição de Benders e outro baseado em relaxação Lagrangiana e decomposição.
Resolver os problemas PPR e PAC. [13]	2010	Exato e Heurística Centralizada	Formulação matemática para a versão *off-line* e uma heurística híbrida para versão *on-line* dos problemas.
Rede de Distribuição de Conteúdos da Akamai. [14]	2010	Heurística Distribuída	Plataforma de alto desempenho para aplicações da Internet.

Em [13], é apresentada uma formulação matemática para a versão *off-line* do PPR e PAC, chamada FD, assim como uma heurística híbrida para a versão *on-line* destes problemas. A formulação FD consiste em resolver o PPR juntamente com o PAC tendo o conhecimento total das requisições. No entanto, a heurística híbrida consiste em resolver o PAC através de uma formulação matemática simplificada extraída da formulação FD e utilizar uma heurística centralizada para resolver o PPR. A estratégia centralizada tenta inserir no servidor com capacidade disponível o conteúdo mais requisitado e remover

o conteúdo menos requisitado do servidor que não tem capacidade a cada período de tempo. Esta estratégia foi empregada em duas heurísticas: a Heurística Construtiva Híbrida (HC), que utiliza estimativas das requisições futuras e a Heurística Construtiva com Conhecimento Futuro (HCFK), que possui o conhecimento das requisições do próximo período de tempo. Sendo que a segunda não é empregada na prática, ela é usada somente como base de comparação de *gaps* da solução. Outra heurística para comparar os *gaps*, a Heurística de Solução por Período (PSH), também foi construída resolvendo o PAC e o PPR, através da formulação FD, a cada período de tempo individualmente. E para comparar o trabalho com o algoritmo da Akamai, o PAC foi resolvido da maneira descrita anteriormente e o PPR, através de uma versão mais sofisticada que a ideia empregada pela Akamai.

Em [14], a plataforma geral do funcionamento da distribuição de conteúdos da empresa Akamai é descrita. A Akamai foi a pioneira no conceito de RDCs há mais de uma década atrás. Por se tratar de uma empresa comercial, a Akamai não fornece detalhes de implementação de sua rede, mas é possível ter uma visão ampla do serviço fornecido. O algoritmo de posicionamento de réplicas consiste, basicamente, em posicioná-las o mais próximo possível do usuário. Assim, um usuário requisita o conteúdo ao servidor mais próximo. As requisições são, então, tratadas pelos servidores que as receberam. Se o servidor não possui o conteúdo requisitado, o conteúdo é replicado localmente para atender a requisição. No entanto, caso o servidor não possua recursos para a replicação, o esquema *Least Recent Used (LRU)* é utilizado para descartar réplicas e liberar recursos. Nenhuma restrição de número de réplicas, custo de replicação ou de transporte é relatada. Assim, toda replicação de conteúdo é realizada sobre a rede de entrega da Akamai, que é uma rede virtual construída sobre a Internet.

3.2 Literatura sobre Simplex de Redes

Além dos trabalhos relacionados a RDC, um estudo sobre o método Simplex para o problema de redes foi realizado com o objetivo de compreender as abordagens distribuídas e paralelas existentes. Um resumo sobre os trabalhos existentes também foi feito e está ilustrado na Tabela 3.2.

Em [6], um algoritmo paralelo do Simplex primal para redes é proposto no qual uma decomposição do problema original em subproblemas é apresentada. Os autores utilizaram um subconjunto de processadores, denominado SP, que executam o Simplex sequencial

em certos subproblemas. Os subproblemas são problemas locais criados particionando os nós da rede entres os membros SP. Os arcos que conectam nós de um mesmo subproblema são chamados de arcos locais e os arcos que conectam nós entre subproblemas distintos, de arcos *across*. Três variantes paralelas deste algoritmo foram apresentadas: método SP Puro, método PP/SP e método Híbrido. No método SP Puro, todos os processadores estão em SP e quando os problemas locais obtém a solução ótima, nós são trocados entre as soluções básicas dos problemas locais para formar um novo problema local. Os arcos *across* são ditos não básicos (não pertencem à solução) e quando um arco desses é escolhido para entrar na solução, a solução básica de um subproblema é movida para o outro subproblema, aumentando o problema local. A solução ótima é encontrada quando todos os problemas locais são ótimos e não existem arcos *across* para entrar na solução.

No método PP/SP, um outro conjunto de processadores também é utilizado, denominado PP, que são os processadores que realizam as operações de *pricing* do Simplex (i. e. cálculo do custo reduzido das variáveis que não pertencem a solução). Um processador do conjunto PP é escolhido, chamado de *mp* (processador *master*), este processador realiza o *pricing* dos arcos *across* ou direciona outro membro PP para fazer, e também mantém o número e o tamanho das soluções básicas movidas para balancear a carga. A solução ótima é encontrada quando todos os problemas locais são ótimos e o processador *mp* não encontra mais arcos *across* candidatos a entrar na solução.

Por fim, o método Híbrido, mantém ambos os conjuntos PP e SP, mas permite que um membro de SP interrompa o processador *mp* assim que tenha alcançado seu ótimo local, para saber sobre os nós dos arcos *across* do seu problema local. No entanto, quando um arco *across* é escolhido por *mp* para entrar na solução, um processador SP que compõe este arco *across* interrompe o processador oposto a ele para que ele transfira a solução básica ao outro processador.

Comparando os três métodos descritos, os autores relatam que o método SP Puro trata todos os processadores igualmente e é mais paralelo que os outros métodos propostos, mas pode ser "míope" na sua busca por bons arcos *across*. Para o método PP/SP, eles descrevem que o método possui um passo no qual todos os processadores trocam informação e que isto, para memória distribuída, pode resultar em comunicação excessiva. No entanto, o uso do *mp* deve produzir melhores arcos *across* para entrar na base, uma vez que a seleção é feita pesquisando todos os arcos possíveis. Já sobre o método Híbrido, eles argumentam que ele combina as características boas dos outros dois métodos. Os teste foram realizados no multi-computador CRYSTAL e maior eficiência foi observada

em problemas de redes multi-período onde obteve um *speedup* aproximadamente linear em número de processos.

Em [17], um algoritmo paralelo do Simplex dual para redes com memória compartilhada é proposto. Este algoritmo é uma tentativa de realizar pivoteamento concorrente. O método dual do Simplex tradicional não oferece muitas possibilidades para paralelização, porque ele se move de uma solução básica dual viável para outra realizando uma operação de pivoteamento de cada vez. Assim, Thulasiraman et al. propuseram um método dual do Simplex modificado para a realização de pivoteamento concorrente. Durante a operação de pivoteamento, somente um pequeno subgrafo do grafo dado pode estar envolvido. Esta operação consiste em percorrer a árvore geradora do grafo das folhas para raiz, identificando os *clusters* que podem realizar o pivoteamento concorrente. Cada processo, consiste de um nó do grafo e a comunicação entre os processos é feita por memória compartilhada. Os testes foram realizados na máquina Butterfly BBN e para calcular o *speed-up* paralelo, o algoritmo foi testado para um número diferente de processos. Assim, foi observado que o tempo computacional inicialmente diminui com o aumento do número de processos e estabiliza após atingir um mínimo.

Tabela 3.2: Literatura sobre Simplex

Problema	Ano	Método Usado	Contribuição
Algoritmo paralelo do Simplex primal para problemas de redes. [6]	1988	Algoritmo Paralelo	Dividir os problemas em subproblemas.
Algoritmo paralelo do Simplex dual para redes com memória compartilhada. [17]	1993	Algoritmo Paralelo	Tentativa de realizar o pivoteamento concorrente.
Resolver o problema de fluxos de rede e testar seu desempenho em problemas médios e grandes. [3]	1994	Algoritmo Paralelo	Implementação paralela do método simplex primal para problemas de fluxo de redes de custo mínimo, dividido entre pivoteamento e operações de de *pricing*.
Paralelizações do método Simplex.[9]	2010	*Survey*	Revisão das tentativas de paralelização do método simplex.

Em [3], uma abordagem paralela para o método primal do Simplex para o problema de fluxo em rede de custo mínimo, dividida entre operações de pivoteamento e *pricing*, foi proposto. Nesta abordagem, as operações de *pricing* e pivoteamento são executadas simultaneamente. Um grau maior de paralelismo é alcançado decompondo estas operações.

O conceito de monitor foi utilizado para sincronizar os processos paralelos. O monitor escalona uma tarefa a um processo, essas tarefas estão divididas em: (i) selecionar um arco e realizar o pivoteamento; (ii) atualizar as variáveis duais; e (iii) realizar a operação de *pricing*. Os testes foram realizados em uma máquina MIMD fortemente acoplada, Sequent Symmetry S81, com 20 processadores e 32 Mbytes de memória compartilhada. Para as instâncias testadas, foi alcançado um *speedup* médio de $8,26$ reduzindo o tempo de solução.

Em [9], é mostrada uma revisão dos métodos de paralelização do método Simplex em geral. Esta revisão consiste dos estudos de diversas tentativas de paralelização do método Simplex em relação a técnicas eficientes do Simplex sequencial e a natureza dos problemas práticos de programação linear (PL). Para problemas de PL grandes e esparsos, não existe paralelização do método Simplex que ofereça significante melhora no desempenho sobre uma boa implementação sequencial. Entretanto, algumas implementações de resolvedores paralelos obtém sucesso para problemas de PL que são densos ou que possuam propriedades estruturais particulares.

Note que os trabalhos apresentados na seção anterior resolvem o PAC juntamente com outros problemas de RDC, através de métodos matemáticos ou por heurísticas sequenciais, e que apesar do algoritmo da Akamai possuir uma abordagem distribuída, este não oferece muitos detalhes sobre sua implementação por ser um produto comercial. Além disso, não foram encontradas abordagens paralelas ou distribuídas específica para o método Simplex de Transporte.

Capítulo 4

Algoritmo Sequencial

Neste capítulo, o algoritmo sequencial para o Simplex de Transporte adaptado para o problema do PAC é apresentado.

O Problema de Transporte pode ser representado por uma Tabela de Transporte. Assim, o PAC modelado como um Problema de Transporte também pode se representado por uma tabela, como a Tabela 4.1. Nesta tabela, as linhas representam os servidores ($S_i ... S_n$) e as colunas, as requisições dos clientes ($R_i ... R_m$) por um conteúdo específico. As células são formadas pelo custo de transporte associado aos índices linha e coluna, ou seja, o custo de transporte $c_{ij(r)}$ do servidor linha i para o cliente $j(r)$ da requisição coluna r. Cada célula também possui uma variável $x^r_{ij(r)}$ associada ao tráfego da requisição r entre o servidor i e o cliente $j(r)$. As linhas e colunas possuem ofertas e demandas associadas, respectivamente. A oferta de cada servidor linha i é representada pela largura de banda disponível no servidor (b_i) e a demanda de cada requisição coluna r é representada pela largura de banda do conteúdo $c(r)$ requisitado pela requisição (d_r).

Serv \ **Req**	R_1	R_2	$\cdots$	R_m	**bandas**
S_1	$c_{1j(1)}$ $x^1_{1j(1)}$	$c_{1j(2)}$ $x^2_{1j(2)}$	$\cdots$ $\cdots$	$c_{1j(m)}$ $x^m_{1j(4)}$	b_1
S_2	$c_{2j(1)}$ $x^1_{2j(1)}$	$c_{2j(2)}$ $x^2_{2j(2)}$	$\cdots$ $\cdots$	$c_{2j(m)}$ $x^m_{2j(4)}$	b_2
$\vdots$	$\cdots$ $\cdots$	$\cdots$ $\cdots$	$\cdots$ $\cdots$	$\cdots$ $\cdots$	$\vdots$
S_n	$c_{n3j(1)}$ $x^1_{nj(1)}$	$c_{nj(2)}$ $x^2_{nj(2)}$	$\cdots$ $\cdots$	$c_{nj(m)}$ $x^m_{nj(m)}$	b_n
bandas conteúdos	d_1	d_2	$\cdots$	d_m	

Tabela 4.1: Tabela de Transporte

4.1 Solução Inicial

O método do Canto Noroeste [1] para encontrar uma solução inicial viável para o Simplex de Transporte aplicado em RDC utilizando a Tabela de Transporte pode ser dividido em três passos:

Passo Inicialização: Faça $x^r_{ij(r)} := \text{NIL}\ \forall i \in N$ e $\forall r \in M$.

Passo 1: Identificar a célula do canto superior esquerdo da tabela (i, r) e atribuir o valor máximo possível a variável $x^r_{ij(r)} (x^r_{ij(r)} = min(b_i, d_r))$.

Passo 2: Seja b'_i, o valor da banda disponível no servidor i e d'_r, o valor da banda do conteúdo da requisição r ainda não alocada. Inicialmente $b'_i = b_i$ e $d'_r = d_r$. Se

- $d'_r > b'_i$, a largura de banda do servidor S_i está esgotada, mas a requisição do cliente R_r não é totalmente atendida, $d'_r = d'_r - b_i$ e $b'_i = 0$. Mover, se possível, para a segunda linha $(i + 1, r)$ e atribuir a $x^r_{i+1j(r)}$ o $min(b'_{i+1}, d'_r)$, reduzindo a tabela com a exclusão do servidor S_i.

- $b'_i > d'_r$, a requisição do cliente R_r é totalmente satisfeita, mas a largura de banda do servidor S_i não é totalmente utilizada, $b'_r = b'_r - d'_r$. e $d'_r = 0$. Mover, se possível, para a próxima coluna $(i, r + 1)$ e atribuir a $x^{r+1}_{ij(r+1)}$ o $min(b'_i, d'_{r+1})$, reduzindo a tabela com a exclusão da requisição R_r.

- $d'_r = b'_i$, a largura de banda do servidor S_i está esgotada e a requisição do cliente R_r é totalmente atendida, $d'_r = b'_i = 0$. A tabela é reduzida, excluindo o servidor S_i e a requisição R_r. Caso ainda haja requisições e servidores para compor a tabela, retorne ao passo 1. Caso contrário, terminar o algoritmo.

Passo 3: Caso ainda haja requisições e servidores para compor a tabela, retorne ao passo 2. Caso contrário, terminar o algoritmo.

O método do Custo Mínimo é similar ao do Canto Noroeste. Eles diferem na escolha das células. O Canto Noroeste escolhe sempre a célula mais à esquerda superior e o Custo Mínimo, a célula que possui o menor custo de comunicação.

4.2 Segunda Etapa do Simplex de Transporte

A segunda etapa do Simplex de Transporte [8] consiste em analisar a solução atual, verificando se esta pode ser melhorada. Para isto, é necessário construir a solução dual correspondente.

Cada servidor linha i está associado a uma variável dual u_i e cada requisição coluna r, a uma variável dual v_r, conforme a Tabela 4.2. Assim, a construção de uma solução dual implica no cálculo destas variáveis duais respeitando, para cada variável básica pertencente a solução atual, a restrição dual $u_i + v_r = c_{ij(r)}$. Uma variável básica é definida como uma variável $x^r_{ij(r)}$ que pertence a solução atual do problema.

Portanto, cada variável básica $x^r_{ij(r)}$ está associada a uma restrição do problema dual. Isto constitui, então, um sistema de equações, conhecido como Sistema de *Dantzig* [8]:

$$\boxed{x^r_{ij(r)} \text{ é básica} \longrightarrow u_i + v_r = c_{ij(r)}}$$

Para que uma variável se torne básica é preciso esgotar a largura de banda do conteúdo de uma requisição ou a largura de banda de um servidor. Assim, o Problema de Transporte deve esgotar a largura de banda de n servidores e a largura de banda dos conteúdos de m requisições. Isto significa que toda vez que uma variável se torna básica através do método de obtenção de uma solução inicial viável, uma das n ou m bandas é esgotada, o que constituiria $n + m$ variáveis básicas. No entanto, note que ao esgotar a banda do conteúdo da última requisição, também é esgotado a banda do último servidor. Sendo assim, o Problema de Transporte constitui, apenas, $n + m - 1$ variáveis básicas.

Req / **Serv**	R_1	R_2	$\cdots$	R_m	
S_1	$x^1_{1j(1)}$ $c_{1j(1)}$	$x^2_{1j(2)}$ $c_{1j(2)}$	$\cdots$ $\cdots$	$x^m_{1j(4)}$ $c_{1j(m)}$	$\boldsymbol{u_1}$
S_2	$x^1_{2j(1)}$ $c_{2j(1)}$	$x^2_{2j(2)}$ $c_{2j(2)}$	$\cdots$ $\cdots$	$x^m_{2j(4)}$ $c_{2j(m)}$	$\boldsymbol{u_2}$
$\vdots$	$\cdots$ $\cdots$	$\cdots$ $\cdots$	$\cdots$ $\cdots$	$\cdots$ $\cdots$	$\vdots$
S_n	$x^1_{nj(1)}$ $c_{n3j(1)}$	$x^2_{nj(2)}$ $c_{nj(2)}$	$\cdots$ $\cdots$	$x^m_{nj(m)}$ $c_{nj(m)}$	$\boldsymbol{u_n}$
	$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\cdots$	$\boldsymbol{v_m}$	**duais**

Tabela 4.2: Tabela de Transporte com as variáveis duais

Portanto, o sistema de equações da solução dual possui $n + m - 1$ equações e $m + n$ variáveis duais, pois para cada uma das $n + m - 1$ variáveis básicas, uma restrição dual é satisfeita, e para cada um dos n servidores e m requisições, uma variável dual é associada. Como este sistema possui mais variáveis do que equações, a solução desse sistema é obtida

a partir da atribuição de um valor arbitrário a uma das variáveis duais u (é comum fazer $u_1 = 0$). Depois disso é possível resolver o restante do sistema, calculando as demais variáveis duais a partir desta.

Atribuindo $u_1 = 0$, percorre-se a linha através das células correspondentes às variáveis básicas da tabela de transporte, para obter os v_r. Uma vez obtidos estes, percorre-se a coluna para obter os u_i.

Obtidos os valores das variáveis duais, calcula-se os custos reduzidos das variáveis não-básicas (variáveis que não fazem parte da solução).

$$\boxed{custo\_reduzido = c_{ij(r)} - u_i - v_r}$$

Se todos os custos reduzidos forem não-negativos, a solução é ótima. Se existirem custos reduzidos negativos, isto significa que a solução pode ser melhorada. Para melhorar a solução, uma variável não-básica é selecionada para entrar na solução e uma variável básica é escolhida para deixar a solução. A variável que entra na solução é a variável cujo custo reduzido negativo tem o maior valor absoluto. A introdução de uma nova variável na solução ocasiona uma reação em cadeia para compensar as restrições de linha (largura de banda do servidor) e coluna (largura de banda do conteúdo da requisição). O valor da variável que entra deve ser o maior valor possível, sem tornar nenhuma variável básica negativa. A variável básica que tiver seu valor anulado em consequência da variável que entra será a variável que sai da solução.

Assim, a segunda etapa do Simplex de Transporte pode ser dividida em quatro passos:

Passo 1: Critério de Otimalidade

Determinar a solução dual

1. Atribuir um valor arbitrário a uma das variáveis. Comumente é utilizado $u_s = 0$, onde $s = 1$.

2. Percorre-se esta linha do servidor s. Para cada requisição r que $x^r_{sj(r)}$ faz parte da solução, calcula-se $v_r = c_{sj(r)} - u_s$.

3. Para cada v_r calculado anteriormente, percorre-se a coluna da requisição r, calculando o valor $u_i = c_{ij(r)} - v_r$, caso $x^r_{ij(r)}$ faça parte da solução corrente.

4. Para cada u_i calculado, retorne ao sub-passo 2 com $s = i$.

Verificar otimalidade

1. Para cada variável $x^{r}_{ij(r)}$ que não faz parte da solução, calcula-se

   $custo\_reduzido = c_{ij(r)} - u_i - v_r$.

2. Se todos os custos reduzidos são não-negativo, a solução é ótima. Fim do algoritmo.

Passo 2: Critério de Entrada

1. A variável que entra na solução é a variável cujo custo reduzido negativo tenha o maior valor absoluto:

   $max\{u_i + v_r - c_{ij(r)} : u_i + v_r - c_{ij(r)} > 0\}$

Passo 3: Critério de Saída

Selecionar o ciclo relativo à variável que entra

Seja $x^{req}_{sj(req)}$ a variável associada ao servidor s e a requisição req do cliente $j(req)$ que entra na solução. Nos sub-passos seguintes, um processo em cadeia é iniciado para garantir que as restrições de banda dos servidores e banda dos conteúdos das requisições continuem sendo satisfeitas, atribuindo operações de subtração ($\ominus$) e adição ($\oplus$), alternadamente, às variáveis que pertencem ao ciclo relativo à variável que entra, para que estas variáveis sejam atualizadas de um valor θ (menor valor entre as variáveis do ciclo).

1. **Inicialização:** Faça $i := s$, $r := req$, $i_{ant} :=$ NIL, $r_{ant} :=$ NIL, $\theta := \infty$, célula $(a, b) :=$ $não\_visitado$ e $x^{b}_{aj(b)} :=$ operação NIL $\forall a \in N$ e $\forall b \in M$.

2. **Passo Linha:** A partir da célula (i, r), caminhar na linha do servidor i até a célula (i, r'), tal que $x^{r'}_{ij(r')}$ é uma variável básica e a célula (i, r') não foi visitada.

   - Caso (i, r') exista, faça:
     - $i_{ant} := i$;
     - $r_{ant} := r$;
     - $r := r'$;

  - célula $(i, r') :=$ *visitado*;
  - $x^{r'}_{ij(r')} :=$ operação $\ominus$;
- Caso contrário:
  - $x^{r}_{ij(r)} :=$ operação NIL;
  - $i := i_{ant}$;
  - $r := r_{ant}$;
  - Vá para o passo *coluna*;

3. Passo Coluna: A partir da célula (i, r), caminhar na coluna da requisição r até a célula (i', r), tal que $x^{r}_{i'j(r)}$ é uma variável básica e a célula (i', r) não foi visitada.

- Caso (i', r) exista, faça:
  - $i_{ant} := i$;
  - $r_{ant} := r$;
  - $i := i'$;
  - célula $(i', r) :=$ *visitado*;
  - $x^{r}_{i'j(r)} :=$ operação $\oplus$;
- Caso contrário:
  - $x^{r}_{ij(r)} :=$ operação NIL;
  - $i := i_{ant}$;
  - $r := r_{ant}$;
  - Vá para o passo *linha*;

4. Teste Final: Caso $i = s$ e $r = req$, termina o processo de identificação de ciclo. Caso contrário, vá para o passo *linha*.

Selecionar a variável que sai da solução

1. Seja $x^{r'}_{s'j(r')}$, a variável, que possui o menor valor dentre todas as variáveis marcadas com a operação de subtração ($\ominus$) no processo em cadeia do passo anterior e está associada ao servidor s' e a requisição r' que sai da solução.

2. O valor de θ é igual a $x^{r'}_{s'j(r')}$, logo:

   $\theta = \{x^{r}_{ij(r)}$ marcada com a operação de subtração $\in$ ciclo relativo à variável que entra na solução $\}$

   - Se θ é o valor de mais de uma variável, escolhe-se arbitrariamente somente uma para deixar a solução.

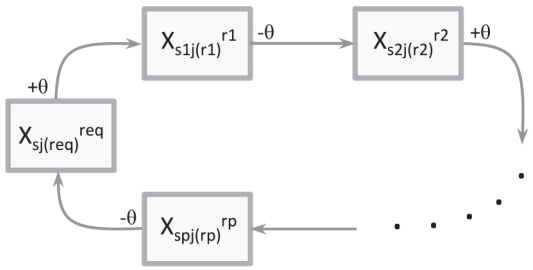

Figura 4.1: Ciclo relativo à variável que entra na solução: Passo Linha e Passo Coluna

Passo 4: Obtenção de uma nova solução

1. A nova solução é obtida adicionando e subtraindo θ das variáveis que formam o ciclo da variável que entra na solução de acordo com as operações atribuídas a elas (Figura 4.1). Retorne ao Passo 1.

Pode existir casos em que a quantidade total de largura de banda nos servidores é maior que a quantidade total da largura de banda de conteúdos requisitados. Neste caso, o sistema é dito como não equilibrado. Assim, requisições artificiais são adicionadas a solução para que a largura de banda disponível possa ser totalmente utilizada e o custo de transporte para cada requisição artificial seja igual a zero.

Além disso, a solução inicial encontrada pode não atender às demandas de todos os clientes por conteúdos. Isto significa, que a solução inicial não é viável. No entanto, para o problema de RDC considerado, as instâncias sempre têm pelo menos uma solução viável. Para obter uma solução viável e prosseguir com o Simplex de transporte, um servidor artificial é adicionado ao problema para atender as requisições dos clientes por conteúdos que não foram atendidas com um custo de comunicação infinito.

4.3 Análise de Complexidades

Nesta seção, a complexidade de tempo para o algoritmo sequencial é apresentada.

4.3.1 Solução Inicial

Nesta subseção, as complexidades dos métodos Canto Noroeste e Custo Mínimo são apresentadas.

Como descrito anteriormente, o Problema de Transporte pode ser representado por uma tabela, como por exemplo a Tabela 4.1. Assim, a célula do canto superior esquerdo da tabela (i, r) é escolhida para iniciar o método Canto Noroeste para obter uma solução inicial para o Simplex de Transporte. O valor máximo possível é atribuído a variável $x^r_{ij(r)}$ ($x^r_{ij(r)} = min(b_i, d_r)$). Com isso, a largura de banda do servidor S_i é esgotada e/ou a requisição do cliente R_r é totalmente satisfeita. O servidor esgotado e/ou a requisição atendida são desconsiderados e a tabela é reduzida. Uma nova célula é escolhida na tabela reduzida e o algoritmo continua até que as larguras de banda de todos os servidores são esgotadas e todas as requisições são satisfeitas.

Ao atribuir um valor máximo a variável $x^r_{ij(r)}$, a largura de banda do servidor S_i é esgotada e/ou a requisição do cliente R_r é totalmente satisfeita, conforme descrito anteriormente. Isto pode ser dividido em três casos: (i) largura de banda do servidor é esgotada e a requisição do cliente não é totalmente satisfeita; (ii) largura de banda do servidor não é esgotada e a requisição do cliente é totalmente satisfeita; (iii) largura de banda de um servidor é esgotada e as requisições atendidas por ele são totalmente satisfeitas;

No caso (i), quando o servidor não teve sua banda esgotada, a nova célula escolhida será referente a este servidor e a próxima requisição por um conteúdo, que o servidor possui, não atendida. Como as requisições são por conteúdos diversos e os servidores não possuem todos os conteúdos, para encontrar esta próxima requisição é necessário percorrer, no pior caso, todas as m requisições (m é número de requisições) para encontrá-la.

Em (ii), quando a requisição não é totalmente satisfeita, a nova célula escolhida será referente a esta requisição e ao próximo servidor que possui o conteúdo requisitado e que não teve sua banda esgotada ainda. Como nem todos os servidores possuem todos os conteúdos, para encontrar este próximo servidor é necessário percorrer, no pior caso, todos os n servidores para encontrá-lo (n é o número de servidores).

Com este dois casos, observa-se que para mudar de linha (servidor) na tabela, no pior caso, deve-se percorrer as m requisições para encontrar uma requisição não atendida totalmente por um servidor (caso i). Encontrada esta requisição, no pior caso, os n servidores possuem o conteúdo requisitado e todos eles são percorridos (caso ii). Nestes

casos, para mudar de linha, $O(m+n)$ células são percorridas.

Por último, no caso (iii), quando a banda de um servidor s é esgotada e as requisições atendidas por ele são totalmente satisfeitas, a nova célula escolhida será referente ao próximo servidor $pServ$ que não teve sua banda esgotada que possui o conteúdo de uma das requisições atendidas por s. Com isso, para cada requisição atendida por s, no pior caso, os n servidores possuem o conteúdo da requisição e todos eles são percorridos para encontrar o servidor $pServ$ ($O(max_r.n)$, onde max_r é o maior número de requisições atendidas por um servidor).

Assim, a cada escolha de célula, a largura de banda de um servidor é esgotado e/ou uma requisição é satisfeita e para cada mudança de linha (servidor) da tabela, percorre-se $m+n$ (casos i e ii) ou $m+max_r.n$ (caso iii) células. Como as bandas de todos os servidores devem ser esgotadas e a cada banda esgotada $O(m+max_r.n)$ células são percorridas, a complexidade de tempo do método Canto Noroeste é $n.(m+max_r.n) \approx O(n.m)$, onde $max_r << m$.

O método do Custo Mínimo é similar ao do Canto Noroeste. Eles diferem na escolha das células. O método do Custo Mínimo escolhe sempre a célula que possui o menor custo de comunicação para continuar a construção da solução inicial. Por exemplo, no caso (i), onde se percorre, no pior caso, todas as requisições para a escolha da célula, o método do Custo Mínimo também percorre, no pior caso, todas as requisições, mas estas são percorridas a partir das requisições dos servidores mais próximo (com menores custos de comunicação). Para isto, cada servidor ordena os servidores por menor custo em relação a ele ($O(n.logn)$).

Assim, no pior caso, quando for necessário percorrer as requisições (caso (i)), percorre-se as requisições dos servidores mais próximo até as requisições dos servidores mais distante, quando for necessário percorrer os servidores (caso (ii)), percorre-se do servidor mais próximo ao mais distante e quando é necessário percorrer os servidores e as requisições (caso (iii)), percorre-se do servidor mais próximo ao mais distante e as requisições dos servidores mais próximos até as requisições dos servidores mais distante. Com isso, antes de iniciar o método do Custo Mínimo é necessário que cada servidor ordene os servidores por menor custo. A ordenação por todos os servidores tem complexidade de tempo $O(n^2.logn)$.

Portanto, a complexidade de tempo do método do Custo Mínimo é $O(n^2.logn) + O(n.m) \approx O(n.m)$.

4.3.2 Simplex de Transporte

Nesta seção, a complexidade do Simplex de Transporte é apresentada.

Com a solução obtida pelos métodos Canto Noroeste ou Custo Mínimo, calcula-se os valores das variáveis duais dos servidores (linha da Tabela 4.1) e requisições (coluna da Tabela 4.1). Para isto, a variável dual de um servidor s é iniciada com valor zero. A partir da linha deste servidor s, percorre-se esta linha, calculando as variáveis duais das requisições r, na qual $x^r_{sj(r)}$ faz parte da solução. Para cada requisição r que o valor da variável dual foi calculada anteriormente, percorre-se a coluna da requisição r, calculando as variáveis duais ainda não calculadas dos servidores i, nas quais $x^r_{ij(r)}$ faz parte da solução. Este processo de percorrer linhas e colunas da tabela calculando as variáveis duais é feito até que todas as variáveis duais são calculadas. Assim, a complexidade de tempo para o cálculo das variáveis duais é $O(n+m)$.

Com as variáveis duais calculadas, para cada variável $x^r_{ij(r)}$ que não faz parte da solução em que o servidor i possui o conteúdo requisitado de r, o custo reduzido é calculado. Com isso, para cada requisição, se, no pior caso, todos os servidores possuem o conteúdo requisitado e a requisição é atendida por um único servidor, $n-1$ custos reduzidos são calculados. Portanto, a complexidade de tempo do cálculo do custo reduzido é $O(n.m)$.

Com os custos reduzidos calculados, o menor custo reduzido é encontrado ($O(n.m)$). Se o custo não é negativo, a solução ótima foi encontrada. Caso contrário, a variável $x^{req}_{sj(req)}$ responsável por este custo entra na solução. A introdução desta variável origina um ciclo na solução, ocasionando uma reação em cadeia para compensar as restrições de banda do servidor e banda do conteúdo da requisição. Esta reação em cadeia consiste em percorrer o ciclo atribuindo operações de adição e de subtração, alternadamente, às variáveis pertencentes ao ciclo.

A partir da variável $x^{req}_{sj(req)}$ que entra na solução, move-se na linha do servidor s até encontrar uma variável $x^{r'}_{sj(r')}$ básica. A partir desta variável, move-se na coluna da requisição r' até encontrar uma variável $x^{r'}_{i'j(r')}$ básica. Quando não é possível encontrar uma variável básica na linha de um servidor, retrocede-se e encontra-se outro servidor, o mesmo é feito quando não é possível encontrar uma variável básica na coluna de uma requisição, retrocede-se e escolhe-se uma outra requisição. Este processo é feito, até chegar na requisição req relacionada a variável que entra.

Suponha que o ciclo é formado por todas variáveis $x^r_{ij(r)}$ que pertencem à solução (as $n+m$ variáveis básicas, ver Seção 4.2). Assim, no pior caso, para cada servidor percorre-se

as m requisições para encontrar uma variável básica e para cada requisição, percorre-se os n servidores. A complexidade de tempo do processo de reação em cadeia é $O(n.m)$.

Para descobrir a variável que sai da solução, percorre-se a tabela para descobrir a variável com menor valor atribuído a operação de subtração no processo reação em cadeia ($O(n.m)$). Este menor valor, denominado de θ, é o valor com que as variáveis do ciclo são atualizadas, de acordo com as operações atribuídas a elas. Para isto, percorre-se a tabela atualizando as variáveis ($O(n.m)$). Com isso, a variável que tinha o valor igual a θ, é atualizada para zero e sai da solução. Assim, uma nova solução é obtida e recomeça o cálculo das variáveis duais até que nenhum custo reduzido negativo seja encontrado.

A complexidade de tempo de uma passada do Simplex de Transporte sequencial é a complexidade do cálculo do custo reduzido ($O(n.m)$), do processo de reação em cadeia ($O(n.m)$), da descoberta da variável que sai ($O(n.m)$) e atualização das variáveis ($O(n.m)$). Isto se resume em:

$$O(n+m) + O(n.m) + O(n.m) + O(n.m) \approx O(n.m) \tag{4.1}$$

Logo, a complexidade de tempo do Simplex de Transporte sequencial é $O(p.n.m)$, onde p é número de passos do Simplex de Transporte.

4.4 Exemplo

Nesta seção é apresentado um exemplo de execução do algoritmo sequencial para o Simplex de Transporte adaptado para o problema do PAC descrito anteriormente.

Os dados de entrada do exemplo estão ilustrados na Tabela 4.3. Estes dados consistem da largura de banda de 3 servidores, da largura de banda dos conteúdos de 7 requisições e do custo de transporte associado a cada par servidor-requisição. Note que o exemplo constitui um sistema não equilibrado pois a soma das larguras de banda dos conteúdos das requisições ($10+20+10+20+10+20+15=105$) é menor que a soma das larguras de banda dos servidores ($50+40+20=110$). Assim, uma requisição artificial é criada para utilizar a banda de servidor que está sobrando ($110-105=5$). Isto esta ilustrado na Tabela 4.4.

Req / Serv	R_1	R_2	R_3	R_4	R_5	R_6	R_7	**bandas**
S_1	0	0	5	5	5	5	5	50
S_2	5	5	0	0	10	10	10	40
S_3	5	5	10	10	0	0	0	20
bandas conteúdos	10	20	10	20	10	20	15	

Tabela 4.3: Exemplo para o Simplex de Transporte

Req / Serv	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	**bandas**
S_1	0	0	5	5	5	5	5	0	50
S_2	5	5	0	0	10	10	10	0	40
S_3	5	5	10	10	0	0	0	0	20
bandas conteúdos	10	20	10	20	10	20	15	5	

Tabela 4.4: Requisição Artificial

Utilizando o método do Custo Mínimo, a tabela é percorrida a partir das células com os menores custos de transporte. Como no exemplo existe várias células com o menor custo, que é zero, uma destas é escolhida para iniciar. Sendo assim, considere que o processo de construção da solução se inicia percorrendo a tabela a partir da célula $(1,1)$, conforme ilustrado na Tabela 4.5. O valor máximo possível é atribuído à variável $x_{1j(1)}$ ($x^1_{1j(1)} := 10$) e as larguras de banda do servidor 1 e do conteúdo da requisição 1 são atualizadas ($b_1 = 50-10 = 40$ e $d_1 = 10-10 = 0$, respectivamente). Depois disso,

move-se para a próxima célula de menor custo depois da célula $(1,1)$, que no caso, é uma célula também de custo zero. Assim, suponha que se move para a célula $(1,2)$. O valor máximo possível também é atribuído à variável $x^2_{1j(2)}$ ($x^2_{1j(2)} := 20$) e as larguras de banda do servidor 1 e do conteúdo da requisição 2 são atualizadas ($b_1 = 40 - 20 = 20$ e $d_2 = 20 - 20 = 0$, respectivamente). Este processo continua até todas as larguras de banda estarem esgotadas (i. e. são iguais a zero). Quando isto acontece, uma solução inicial é obtida, conforme ilustrado na Tabela 4.6.

Depois de obtida a solução inicial, a solução dual correspondente deve ser também encontrada. Para isto, conforme descrito na Seção 4.2, uma variável dual é associada a cada servidor e a cada requisição (Tabela 4.7). Inicialmente, o valor zero é atribuído à variável dual u_1. A partir desta atribuição, percorre-se a linha do servidor 1, calculando as variáveis duais v de cada requisição r em que a variável $x^r_{1j(r)}$ faz parte da solução (i. e. $x^2_{1j(2)} \neq$ NIL). Assim, as variáveis duais v_1, v_2, v_6 e v_7 são calculadas na Tabela 4.8. Para cada variável dual v calculada, percorre-se a coluna da requisição r associada, calculando as variáveis duais u dos servidores i, em que $x^r_{ij(r)}$ faz parte da solução. Para as requisições 1 e 2, que já tiveram suas variáveis duais v calculadas, nenhuma variável dual u é calculada. Para a requisição 6, a variável dual u do servidor 2 é calculada (pois $x^6_{2j(6)}$ pertence a solução). Com isso a linha do servidor 2 é percorrida, calculando as variáveis duais v de cada requisição r em que a variável $x^r_{2j(r)}$ pertence à solução. Assim, as variáveis duais v_3 e v_4 são calculadas na Tabela 4.9.

Por fim, para a requisição 7, que também teve sua variável dual v calculada, a variável dual u do servidor 3 é calculada (pois $x^7_{3j(7)}$ pertence a solução). Com isso a linha do servidor 3 é percorrida, calculando as variáveis duais v de cada requisição r em que a variável $x^r_{3j(r)}$ pertence à solução. Assim, as variáveis duais v_5 e v_A são calculadas na Tabela 4.10. Neste momento, a solução dual foi, totalmente, construída.

Com as variáveis duais calculadas, os custos reduzidos de cada variável que não pertence a solução são calculados. Na Tabela 4.11, os custos reduzidos foram calculados e colocados ao lado dos custos em cada célula. Note que dois custos reduzidos negativos são encontrados (das variáveis $x^A_{1j(A)}$ e $x^A_{2j(A)}$). A variável referente ao custo reduzido negativo com maior valor absoluto é escolhida para entrar na solução, que é a variável $x^A_{2j(A)}$, com custo reduzido -10.

Selecionada a variável que entra, deve-se, então, selecionar o ciclo formado pela entrada desta variável. Para isto, a partir da célula $(2, A)$, percorre-se a linha do servidor 2 até encontrar uma variável básica, no caso, a variável $x^6_{2j(6)}$, conforme ilustrado na Tabela

4.12. A operação de subtração é atribuída a esta variável para que ela seja atualizada de um valor θ posteriormente. Depois percorre-se a coluna da requisição 6 até encontrar uma variável básica, no caso, a variável $x^6_{1j(6)}$, atribuindo a operação de adição a ela. Percorre-se, agora, a linha do servidor 1, encontrando a variável básica $x^7_{1j(7)}$ e atribuindo a operação de subtração a ela. Este processo continua até encontrar a variável $x^A_{2j(A)}$ que entra na solução. Neste momento, o ciclo da variável que entra foi selecionado.

O próximo passo é escolher uma variável básica pertencente ao ciclo selecionado para deixar a solução. A menor variável básica marcada com a operação subtração é a escolhida para sair. No caso do exemplo, a variável $x^A_{3j(A)}$ é a escolhida e está marcada com $*$ na Tabela 4.13 e o valor desta variável é atribuído a θ.

Com o valor de θ, as variáveis que foram marcadas com alguma operação, adição ou subtração, são atualizadas, obtendo uma nova solução, coma a ilustrada na Tabela 4.14. Para verificar se esta solução é a ótima, a solução dual desta nova solução é calculada (Tabela 4.15). Após isso, os custos reduzidos são calculados (Tabela 4.16). Observe que nenhum custo negativo é encontrado, sendo assim, esta solução é a ótima e o algoritmo termina.

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	**bandas**
S_1	10 [0]	20 [0]	[5]	[5]	[5]	[5]	[5]	[0]	~~50~~ 20
S_2	[5]	[5]	[0]	[0]	[10]	[10]	[10]	[0]	40
S_3	[5]	[5]	[10]	[10]	[0]	[0]	[0]	[0]	20
bandas conteúdos	~~10~~ 0	~~20~~ 0	10	20	10	20	15	5	

Tabela 4.5: Selecionando células de menor custo

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	**bandas**
S_1	10 [0]	20 [0]	[5]	[5]	[5]	10 [5]	10 [5]	[0]	~~50~~ 0
S_2	[5]	[5]	10 [0]	20 [0]	[10]	10 [10]	[10]	[0]	~~40~~ 0
S_3	[5]	[5]	[10]	[10]	10 [0]	[0]	5 [0]	5 [0]	~~20~~ 0
bandas conteúdos	~~10~~ 0	~~20~~ 0	~~10~~ 0	~~20~~ 0	~~10~~ 0	~~20~~ 0	~~15~~ 0	~~5~~ 0	

Tabela 4.6: Solução Inicial Encontrada

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	5	5	5	5 / 10	5 / 10	0	u_1
S_2	5	5	0 / 10	0 / 20	10	10 / 10	10	0	u_2
S_3	5	5	10	10	0 / 10	0	0 / 5	0 / 5	u_3
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_A	duais

Tabela 4.7: Construção da solução dual correspondente

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	5	5	5	5 / 10	5 / 10	0	$u_1 = 0$
S_2	5	5	0 / 10	0 / 20	10	10 / 10	10	0	u_2
S_3	5	5	10	10	0 / 10	0	0 / 5	0 / 5	u_3
	$v_1 = 0$	$v_2 = 0$	v_3	v_4	v_5	$v_6 = 5$	$v_7 = 5$	v_A	duais

Tabela 4.8: Calculando as variáveis duais da primeira linha

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	5	5	5	5 / 10	5 / 10	0	$u_1 = 0$
S_2	5	5	0 / 10	0 / 20	10	10 / 10	10	0	$u_2 = 5$
S_3	5	5	10	10	0 / 10	0	0 / 5	0 / 5	u_3
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	v_5	$v_6 = 5$	$v_7 = 5$	v_A	duais

Tabela 4.9: Calculando as variáveis duais da segunda linha

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	5	5	5	5 / 10	5 / 10	0	$u_1 = 0$
S_2	5	5	0 / 10	0 / 20	10	10 / 10	10	0	$u_2 = 5$
S_3	5	5	10	10	0 / 10	0	0 / 5	0 / 5	$u_3 = -5$
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	$v_5 = 5$	$v_6 = 5$	$v_7 = 5$	$v_A = 5$	duais

Tabela 4.10: Calculando as variáveis duais da terceira linha

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	**10** 5	**10** 5	**0** 5	5 / 10	5 / 10	**−5** 0	$u_1 = 0$
S_2	**0** 5	**0** 5	0 / 10	0 / 20	**0** 10	10 / 10	**0** 10	**−10** 0	$u_2 = 5$
S_3	**10** 5	**10** 5	**20** 10	**20** 10	0 / 10	**0** 0	0 / 5	0 / 5	$u_3 = -5$
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	$v_5 = 5$	$v_6 = 5$	$v_7 = 5$	$v_A = 5$	duais

Tabela 4.11: Calculando os custos reduzidos

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	**10** / 5	**10** / 5	**0** / 5	5 / 10 $+\theta$	5 / 10 $-\theta$	**−5** / 0	$u_1 = 0$
S_2	**0** / 5	**0** / 5	0 / 10	0 / 20	**0** / 10	10 / 10 $-\theta$	**0** / 10	**−10** / 0 $+\theta$	$u_2 = 5$
S_3	**10** / 5	**10** / 5	**20** / 10	**20** / 10	0 / 10	**0** / 0	0 / 5 $+\theta$	0 / 5 $-\theta$	$u_3 = -5$
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	$v_5 = 5$	$v_6 = 5$	$v_7 = 5$	$v_A = 5$	**duais**

Tabela 4.12: Selecionando o ciclo da variável que entra na solução

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	**10** / 5	**10** / 5	**0** / 5	5 / 10 $+\theta$	5 / 10 $-\theta$	**−5** / 0	$u_1 = 0$
S_2	**0** / 5	**0** / 5	0 / 10	0 / 20	**0** / 10	10 / 10 $-\theta$	**0** / 10	**−10** / 0 $+\theta$	$u_2 = 5$
S_3	**10** / 5	**10** / 5	**20** / 10	**20** / 10	0 / 10	**0** / 0	0 / 5 $+\theta$	* / 0 5 $-\theta$	$u_3 = -5$
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	$v_5 = 5$	$v_6 = 5$	$v_7 = 5$	$v_A = 5$	**duais**

Tabela 4.13: Selecionando a variável que sai da solução

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	5	5	5	5 / **15**	5 / **5**	0	u_1
S_2	5	5	0 / 10	0 / 20	10	10 / **5**	10	0 / **5**	u_2
S_3	5	5	10	10	0 / 10	0	0 / **10**	0	u_3
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_A	**duais**

Tabela 4.14: Obtenção de uma nova solução

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	5	5	5	5 / 15	5 / 5	0	$u_1 = 0$
S_2	5	5	0 / 10	0 / 20	10	10 / 5	10	0 / 5	$u_2 = 5$
S_3	5	5	10	10	0 / 10	0	0 / 10	0	$u_3 = -5$
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	$v_5 = 5$	$v_6 = 5$	$v_7 = 5$	$v_A = -5$	**duais**

Tabela 4.15: Calculando as variáveis duais novamente

Serv \ Req	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_A	
S_1	0 / 10	0 / 20	**10** / 5	**10** / 5	**0** / 5	5 / 15	5 / 5	**5** / 0	$u_1 = 0$
S_2	**0** / 5	**0** / 5	0 / 10	0 / 20	**0** / 10	10 / 5	**0** / 10	0 / 5	$u_2 = 5$
S_3	**10** / 5	**10** / 5	**20** / 10	**20** / 10	0 / 10	**0** / 0	0 / 10	**10** / 0	$u_3 = -5$
	$v_1 = 0$	$v_2 = 0$	$v_3 = -5$	$v_4 = -5$	$v_5 = 5$	$v_6 = 5$	$v_7 = 5$	$v_A = 5$	**duais**

Tabela 4.16: Nenhum custo reduzido negativo é encontrado: Solução Ótima

Capítulo 5

Algoritmo Distribuído

Neste capítulo, a descrição do algoritmo distribuído baseado no método do Custo Mínimo e Simplex de Transporte são realizadas, assim como, a análise de complexidades do algoritmo.

Nas próximas seções é feita a descrição da heurística distribuída baseada no método do Custo Mínimo, DistPAC, e do algoritmo distribuído do Simplex de Transporte, DistST. Para isto, algumas considerações são feitas e algumas notações são usadas.

Como é preferível que um cliente acesse a cópia de um conteúdo através do servidor mais próximo, as requisições dos clientes foram, então, colocadas nestes servidores. Com isso, a figura do cliente foi retirada e toda a responsabilidade da distribuição de requisições foi dada aos servidores, o que de fato acontece em uma RDC. Assim, cada servidor possui uma lista de requisições e as informações das posições das réplicas dos conteúdos em todos os servidores através de um serviço similar ao DNS. Os servidores também conhecem os custos de comunicação com todos os outros servidores. O custo de comunicação para um servidor que não possui um conteúdo requisitado é infinito.

Como o Problema de Transporte constitui um sistema equilibrado, a largura de banda total dos conteúdos requisitados deve ser equivalente a largura de banda total disponível nos servidores. Entretanto, nem sempre esta equivalência ocorre, pois a largura de banda total dos conteúdos requisitados pode superar a banda total disponível nos servidores. Quando isto ocorre, o PAC é considerado inviável. No entanto, se a largura de banda total disponível nos servidores supera a largura de banda total dos conteúdos requisitados, uma requisição artificial é criada para utilizar as bandas não alocadas e permitir o equilíbrio do sistema. Assim, um servidor artificial é criado para ser o responsável por esta requisição.

Alguns pseudocódigos de algoritmos referenciados nas próximas seções (a partir do

Algoritmo 33) estão apresentados no Apêndice A. As notações das variáveis utilizadas pelos algoritmos estão divididas entre as duas próximas seções, onde os algoritmos DistPAC e DistST são descritos.

5.1 Algoritmo DistPAC

O algoritmo DistPAC consiste de um procedimento distribuído para encontrar uma solução inicial para o Simplex de Transporte baseado no método do Custo Mínimo. As variáveis utilizadas neste algoritmo são descritas na próxima seção.

5.1.1 Variáveis

As notações das variáveis de cada servidor associadas a ele, as suas requisições (requisições locais) e aos conjuntos utilizados pelo DistPAC são apresentadas a seguir:

(i) **Servidor**

$meuId$	id associado ao servidor
$idMenor$	servidor mais próximo (com menor custo de comunicação)
$servArtificial$	servidor artificial
$banda$	largura de banda disponível no servidor
$atendeu_{ic}$	quantidade de banda alocada pelo servidor $meuId$ devido ao pedido do servidor i pelo conteúdo c

(ii) **Requisição**

req_r	requisição r pertencente a $lReq$
$cont_r$	identificação do conteúdo da requisição r
$bandCont_r$	tamanho da banda exigida pelo conteúdo da requisição r
$quantAlocada_r$	quantidade de banda alocada para requisição r
$atendimento_{ri}$	quantidade de banda para requisição r atendida pelo servidor i
$idReqArt$	id da requisição artificial atendida pelo servidor artificial

(iii) **Conjuntos**

S	Conjunto de Servidores
$lCont$	Lista de todos os Conteúdos
$lReq$	Lista de Requisições
$nReq$	Número de Requisições
$lReqNalocadas$	lista de requisições não alocadas do servidor
$lAcabou$	lista de servidores que suspeitam que acabaram

5.1.2 Descrição do Algoritmo

Nesta primeira parte do algoritmo, os servidores enviam mensagens uns aos outros com objetivo de encontrar uma solução inicial para a execução do Simplex de Transporte. Para isto, cada servidor, inicialmente, tenta alocar cada uma das suas requisições locais. Se não for possível alocar uma requisição local, uma mensagem $ALOCA$ é enviada ao servidor de menor custo (Algoritmos 1 e 2). Um servidor ao receber a mensagem $ALOCA$ (Algoritmo 3), verifica a disponibilidade de banda. Se for possível atender a requisição, uma mensagem ACK é enviada ao servidor que requisitou o conteúdo, caso contrário, uma mensagem $NACK$ é enviada.

Um servidor ao receber um ACK (Algoritmos 4 e 6), desconta a quantidade que já foi atendida. Se esta quantidade não representar a banda total do conteúdo, uma mensagem $ALOCA$ é enviada a outro servidor que ainda não foi verificado (i.e. o próximo servidor com menor custo). No entanto, um servidor ao receber um $NACK$ (Algoritmo 5), apenas envia $ALOCA$ a outro servidor não verificado.

Esta primeira parte do algoritmo, que consiste da obtenção da solução inicial, termina quando todos os servidores conseguirem atribuir todas as suas requisições a um servidor.

Algoritmo 1: Inicialização

```
1 servInicial := 0; se meuId = servidor artificial então
2     idReqArt := 0;
3 para r de 1 até nReq faça
4     se cont_r ∈ lCont então
5         tentaAlocar(r, bandCont_r, cont_r)
6     senão
7         idMenor← id do servidor mais próximo ainda não verificado;
8         envia ALOCA(r, bandCont_r, cont_r) para idMenor;
9 verificaTermino();
```

Para terminação desta parte, o servidor verifica se sua lista $lReqNalocadas$ está vazia (linha 1 do Algoritmo 7), no momento em que ele recebe a confirmação de alocação (i.e. recebimento da mensagem ACK). Se estiver, uma mensagem FIM_SOL é enviada a todos os servidores, ou seja, esta mensagem indica que o servidor suspeita que terminou o algoritmo de obtenção da solução inicial (Algoritmo 7). Assim, um servidor, ao receber FIM_SOL, armazena o id do servidor que enviou a mensagem na lista $lAcabou$ e verifica se a lista $lAcabou$ possui todos os servidores, exceto do servidor artificial. Se possuir, isto indica que todos os servidores suspeitam que terminaram. Neste momento, o servidor verifica se ele possui banda disponível. Se possuir, uma mensagem $REQARTIFICIAL$ é enviada ao servidor artificial com o valor da banda disponível, indicando que este valor

Algoritmo 2: tentaAlocar (r, tam, c)

```
1  se banda > 0 então
2      se tam <= banda então
3          banda := banda − tam;
4          atendeu_meuIdc := tam;
5          atendimento_rmeuId := tam;
6          quantAlocada_r := quantAlocada_r + tam;
7      senão
8          atendeu_meuIdc := banda;
9          atendimento_rmeuId := banda;
10         quantAlocada_r := quantAlocada_r + banda;
11         banda := 0;
12 se (quantAlocada_r < bandCont_r) e (∃ servidor não verificado que possui c) então
13     idMenor− id do servidor mais próximo ainda não verificado que possui c;
14     envia ALOCA(r, bandCont_r − quantAlocada_r, c) para idMenor;
```

Algoritmo 3: Ao receber uma mensagem $ALOCA(r, tam, c)$ do servidor i

```
1  se banda > 0 então
2      se tam <= banda então
3          banda := banda − tam;
4          atendeu_ic := tam;
5          envia ACK(r, tam, c) para servidor i;
6      senão
7          atendeu_ic := banda;
8          envia ACK(r, banda, c) para servidor i;
9          banda := 0;
10 senão
11     envia NACK(r, tam, c) para servidor i;
```

Algoritmo 4: Ao receber uma mensagem $ACK(r, tam, c)$ do servidor i

```
1  procedimentoAck(r, tam, c, i);
```

Algoritmo 5: Ao receber uma mensagem $NACK(r, tam, c)$ do servidor i

```
1  se ∃ servidor não verificado que possui c então
2      idMenor− id do servidor mais próximo ainda não verificado que possui c;
3      envia ALOCA(r, bandCont_r − quantAlocada_r, c) para idMenor;
4  senão
5      envia ALOCA(r, bandCont_r − quantAlocada_r, c) para servArtificial;
6  verificaTermino();
```

Algoritmo 6: procedimentoAck(r, tam, c, i)

```
1  atendimento_ri := tam;
2  quantAlocada_r := quantAlocada_r + tam;
3  se quantAlocada_r < bandCont_r então
4      se ∃ servidor não verificado que possui c então
5          idMenor− id do servidor mais próximo ainda não verificado ≠ i que possui c;
6          envia ALOCA(r, bandCont_r − quantAlocada_r, c) para idMenor;
7      senão
8          envia ALOCA(r, bandCont_r − quantAlocada_r, c) para servArtificial;
9  verificaTermino();
```

deve ser utilizado pela requisição artificial. Entretanto, se não possuir banda disponível, uma mensagem *ACABOU* é enviada ao servidor artificial (Algoritmo 8).

Algoritmo 7: verificaTermino()

```
1 se lReqNalocadas == {} então
2     envia FIM_SOL() para todos os servidores, exceto para o servidor artificial;
```

Algoritmo 8: Ao receber uma mensagem $FIM\_SOL()$ do servidor i

```
1 Insere i na lista lAcabou;
2 se lAcabou possui os ids de todos os servidores, exceto do servidor artificial então
3     se banda > 0 então
4         envia REQARTIFICIAL(banda, c) para servArtificial, onde meuId possui c ;
5         atendeu_{servArtificialc} := banda;
6         banda := 0;
7     senão
8         envia ACABOU() para servArtificial;
```

O servidor artificial ao receber $REQARTIFICIAL$, utiliza o valor da banda disponível no servidor que enviou a mensagem para atender a requisição artificial e armazena o *id* do servidor na lista *lAcabou* (Algoritmo 9). Quando o servidor artificial recebe $ACABOU$, verifica se a lista *lAcabou* possui todos os servidores, se possuir, a mensagem *ACABOU* é enviada aos demais servidores. Assim, quando um servidor qualquer recebe $ACABOU$, verifica se a lista *lAcabou* possui todos os servidores, se possuir, indica que o servidor acabou a etapa de obtenção de solução inicial e o Simplex de Transporte pode ser iniciada (Algoritmo 10).

Algoritmo 9: Ao receber uma mensagem $REQARTIFICIAL(tam, c)$ do servidor i

```
1 atendimento_{idReqArti} := tam;
2 idReqArt := idReqArt + 1;
3 Insere i na lista lAcabou;
```

Uma solução é ilustrada na Figura 5.1 (exemplo descrito na Seção 2.1). As requisições são representadas por círculos com a identificação do servidor onde estão alocadas e do conteúdo requisitado. Os servidores estão ilustrados por retângulos com as identificações dos conteúdos disponíveis e, ao lado, um cilindro indica a largura de banda disponível. As atribuições indicam o fluxo transportado dos servidores que possuem os conteúdos para os servidores responsáveis pelas requisições com um custo de transporte associado. Nesta solução, é necessário que uma requisição artificial tenha sido criada, pois o servidor 2 possui largura de banda igual a 40 e somente 35 está sendo utilizado, o que não constitui um sistema de solução equilibrado, onde toda requisição demandada é atendida e toda banda ofertada pelos servidores é utilizada. Sendo assim, uma mensagem $REQARTIFICIAL$

Algoritmo 10: Ao receber uma mensagem $ACABOU()$ do servidor i

```
1  Insere i na lista lAcabou;
2  se lAcabou possui os ids de todos os servidores então
3      /*termina solução inicial*/;
4      se meuId = servidor artificial então
5          envia ACABOU() para todos os servidores;
6      passo := 1;
7      idCicloServ := −1;
8      servCicloInicado := {};
9      se meuId = servInicial então
10         dualU := 0;
11         paiServDU := −1;
12         reqServDU := −1;
13         caminhoTam := 0;
14         caminho := {};
15         para cada servidor s faça
16             para cada conteúdo c faça
17                 se atendeu_sc > 0 então
18                     se s! = meuId então
19                         envia VARV(custo_s − dualU, c, atendeu_sc, caminho, caminhoTam, −1)
                           para o servidor s;
20                     senão
21                         procedimentoAtendimento(s, c, atendeu_sc, caminho, caminhoTam, −1);
```

deve ter sido enviada ao servidor artificial indicando que a banda disponível será utilizada pela requisição artificial. Na Figura 5.2, é possível observar esta requisição artificial, onde o servidor artificial é representado por um servidor 4 e a requisição artificial está alocada neste servidor e requisita um conteúdo 4 (considere que todos os servidores possuam este conteúdo).

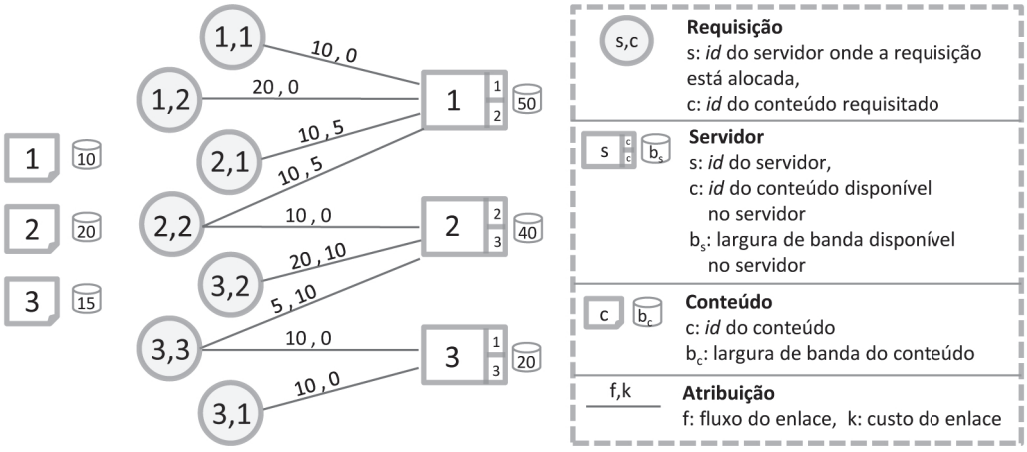

Figura 5.1: Exemplo de solução

5.2 Algoritmo DistST

Conforme descrito na Seção 4.2, o Simplex de Transporte consiste em analisar a solução encontrada, verificando se esta pode ser melhorada. Nesta seção, um algoritmo distribuído para o Simplex de Transporte, o DistST, é descrito, que utiliza a solução inicial obtida pelo DistPAC. As variáveis utilizadas por esse algoritmo estão descritas na próxima seção.

5.2.1 Variáveis

As notações das variáveis de cada servidor associadas a ele, as suas requisições (requisições locais) e aos conjuntos utilizados pelo DistST são apresentadas a seguir:

(i) **Servidor**

$servInicial$	id do servidor que inicializa o segundo passo do simplex
$dualU$	valor da variável dual u
$servPaiDU$	servidor Pai da variável dual u
$reqPaiDU$	requisição Pai da variável dual u
$custo_{is}$	distância entre o servidor i e o servidor s
$passo$	número do passo do Simplex de Transporte
$caminhoServ$	caminho na árvore de solução T desde da raiz sT até o servidor (cada posição corresponde a um servidor $\{s\}$ ou a uma $3-upla$ <servidor, requisição, $cont_{requisicao}$ > - $\{s, r, c\}$)
$caminhoServTam$	tamanho do caminho na árvore de solução T desde da raiz até o servidor sT
$menorEscolhido$	servidor com menor id de uma requisição alcançada pela construção da árvore de solução e escolhida para continuar a construção
$reqEscolhida$	requisição escolhida para continuar a construção da árvore
$servRaiz$	1, para o servidor raiz da árvore de propagação das variáveis duais
$numCiclo$	número de ciclos descobertos em um passo
$idCicloServ$	id do ciclo que o servidor pertence
$cicloServ$	caminho do ciclo na qual o servidor pertence
$cicloServTam$	tamanho do ciclo na qual o servidor pertence
$posCicloServ$	posição que o servidor está no ciclo que ele pertence
$crServ$	custo reduzido do ciclo que o servidor pertence
$funcServ(c_p)$	função que retorna o servidor da posição p do caminho (ciclo) c
$teta$	valor do θ do ciclo que o servidor pertence
$recAtualizaU_s$	1, se o servidor recebeu atualização da variável dual u referente ao ciclo de s e 0, caso contrário

(ii) **Requisição**

$dualV_r$	valor da variável dual v associado à requisição r
$servPaiDV_r$	servidor pai da variável dual v associado à requisição r
$participa_{rs}$	1, se a atribuição da requisição r ao servidor s pertence a solução, e 0, caso contrário
$caminhoReq^r$	caminho na árvore de solução T da raiz sT até a requisição r
$caminhoReqTam_r$	tamanho do caminho na árvore de solução T da raiz sT até a requisição r
c_i	posição i do caminho c
$idCicloReq_r$	id do Ciclo que a requisição r pertence
$cicloReq_r$	caminho do ciclo na qual a requisição r pertence
$cicloReqTam_r$	tamanho do ciclo na qual a requisição r pertence
$posCicloReq_r$	posição que a requisição r está no ciclo que ela pertence
$crReq_r$	custo reduzido do ciclo que a requisição r pertence
$funcReq(c_p)$	função que retorna a requisição da posição p do caminho c, (se a posição corresponder a de um servidor, o valor -1 é retornado)
$funcCont(c_p)$	função que retorna o conteúdo da requisição da posição p do caminho ou ciclo c (se a posição corresponder a de um servidor, o valor -1 é retornado)
$tetaReq_r$	valor do θ do ciclo que requisição r pertence
$recAtualizaUReq_{rs}$	1, se a requisição r recebeu atualização da variável dual u referente ao ciclo de s e 0, caso contrário

(iii) **Conjuntos**

$servAtingidos$	conjunto dos servidores atingidos pelas mensagens de cálculo das variáveis duais da construção da árvore de solução
$servCicloIniciado$	conjunto dos servidores que iniciaram o processo de descoberta de ciclo
$varSainte$	conjunto relacionado a variável que sai da solução, $\{req$, $servReq, serv$, $paiReq, ciclo, r, pos\}$ em que req é a requisição relacionada à atribuição que sai da solução, $servReq$ é o servidor na qual req pertence, $serv$ é o servidor relacionado à atribuição que sai da solução, $paiServ$ é o nó pai de $reqT(req, serv)$ na árvore T, $ciclo$ é o id do ciclo que originou a saída da variável, r é a requisição responsável por $ciclo$ e pos é a posição de req no $ciclo$
$servVaru$	lista dos servidores que enviaram o valor da variável dual u
$varu_s$	conjunto de valores duais u dos servidores, onde $varu_s$ é o valor u de s

5.2.2 Descrição do Algoritmo

Para verificar se uma solução obtida pelo DistPAC pode ser melhorada é necessário a construção da solução dual correspondente. O Simplex de Transporte é, então, iniciado por um servidor. Este servidor inicia o processo de construção da solução dual calculando os valores duais dos servidores e das requisições a partir das variáveis que pertencem a solução (i. e. as atribuições requisição-servidor que pertencem a solução). Assim, o valor zero é atribuído a variável dual u associada a este servidor. Com esta atribuição, ele calcula as variáveis duais v das requisições atendidas por ele e envia esses valores aos servidores responsáveis por estas requisições. Cada servidor destas requisições, por sua vez, calcula a variável dual u dos outros servidores que também as atendem e envia para eles. Cada servidor ao receber o valor da sua variável dual u, envia este valor aos demais servidores para o cálculo dos custos reduzidos. Quando todos esses valores duais tiverem sido calculados, a árvore da solução dual T é obtida.

Portanto, com os valores das variáveis duais obtidos, os custos reduzidos das variáveis (atribuições) que não pertencem a solução são calculados. Ao invés de selecionar o custo reduzido negativo com maior valor absoluto dentre todos os servidores e requisições, como é feito na versão sequencial, cada servidor, seleciona a variável de custo reduzido negativo com maior valor absoluto entre suas requisições locais. Cada custo reduzido selecionado indica a introdução da variável associada na solução. Ao introduzir esta variável, um ciclo é formado na árvore da solução dual. Assim, cada servidor, que selecionou uma variável, envia uma mensagem para o servidor responsável pelo custo reduzido desta variável. Este servidor inicia o processo de seleção de ciclo, propagando a informação de participação de ciclo aos demais componentes do ciclo. Durante esta propagação uma variável que pertence ao ciclo é escolhida para sair. Isto ocasiona o recálculo das variáveis duais.

Na versão sequencial do Simplex de Transporte descrita no Capítulo 4, quando uma variável que pertence a solução é escolhida para sair, todas as variáveis duais são recalculadas. No entanto, foi observado que nem todas as variáveis duais precisam ser recalculadas, somente as variáveis duais dos servidores e das requisições da subárvore filha da atribuição (variável) que entrou na solução. Assim, somente parte da árvore de solução dual T é reconstruída.

Como cada servidor pode selecionar uma variável de custo reduzido negativo, uma quantidade de ciclos igual ao número de servidores pode ser formada com a adição dessas variáveis. O intuito de selecionar mais de uma variável de custo reduzido negativo, formando mais de um ciclo, é que estes ciclos sejam processados em paralelo. Entretanto,

como cada ciclo resulta na reconstrução de alguma parte da árvore de solução T e estas partes podem ter intersecções, a seleção de todos esses ciclos pode ocasionar incoerências na árvore. Portanto, durante a propagação do ciclo, quando esta atinge um componente que participa de outro ciclo, o ciclo com menor custo reduzido é escolhido para prosseguir e outro é cancelado.

Quando todos os ciclos forem finalizados ou cancelados, um novo passo do Simplex pode ser iniciado. Ao iniciar um novo passo, é necessário que as variáveis duais dos servidores e das requisições de parte da árvore que deve ser reconstruída sejam recalculadas. Com isso, a variável dual filha da atribuição que entrou na solução é calculada, permitindo que outros valores duais também sejam calculados e que estes sejam enviados aos servidores destas variáveis duais. Cada servidor ao receber o valor da sua variável dual u recalculada, envia este valor aos demais servidores para o cálculo dos custos reduzidos. Portanto, quando todos os valores duais dos servidores e das requisições são obtidos, os custos reduzidos das variáveis que não pertence a solução são calculados. Esse processo continua até que nenhum custo reduzido negativo seja encontrado.

Nas próximas subseções, o algoritmo DistST é descrito detalhadamente. Para isto, a descrição é dividida de acordo com cada etapa que constitui um passo do Simplex de Transporte: construção da árvore de solução dual inicial; cálculo dos custos reduzidos; seleção de ciclo; cancelamento de ciclo; mudança de passo; e reconstrução da árvore de solução dual.

5.2.2.1 Construção da árvore de solução dual

Como falado anteriormente, o Simplex de Transporte é iniciado por um servidor, chamado de *servInicial*. Este servidor inicia o processo de construção da árvore T da solução obtida pelo algoritmo inicial. A variável dual u referente ao *servInicial* é iniciada com o valor zero e o seu pai com o valor -1. Isto significa que o *servInicial* é a raiz da árvore de solução T. Com a variável dual u do *servInicial* obtida, é possível determinar as variáveis duais v de cada requisição atendida por ele. Assim, para cada servidor s atendido por ele referente a um conteúdo c, uma mensagem $VARV$ é enviada com o valor da variável dual v relacionada a s e c (ou seja, o valor do custo de comunicação para s menos o valor da variável dual u do *servInicial*), além disso, o caminho em T responsável pelo valor da variável dual v também é enviado na mensagem. Neste caso, o caminho somente contém o *servInicial*, pois este caminho consiste do trajeto percorrido da raiz de T até a requisição para o cálculo da variável dual v associada a requisição.

O *servInicial* também pode ter atendido requisições dele próprio por um conteúdo, quando isso acontece, estas requisições são selecionadas, as variáveis duais v referentes a elas são calculadas e os caminhos em T para cálculo das variáveis são preenchidos (linhas 3 a 9 do Algoritmo 12). Se estas requisições são atendidas por outros servidores, uma mensagem $VARU$ é enviada aos outros servidores que atenderam as requisições, com o valor dual u destes servidores e o caminho para o cálculo deste valor dual (linhas 12 a 17 do Algoritmo 12).

A Figura 5.2 ilustra o envio das mensagens $VARV$ e $VARU$ de acordo com a solução da Figura 5.1, onde as linhas tracejadas indicam que a requisição é uma requisição local do servidor correspondente e que na verdade não ocorre envio de mensagens, apenas o cálculo da variável dual v, no caso da ilustração do envio da mensagem $VARV$, e da dual u, no caso da ilustração do envio da mensagem $VARU$.

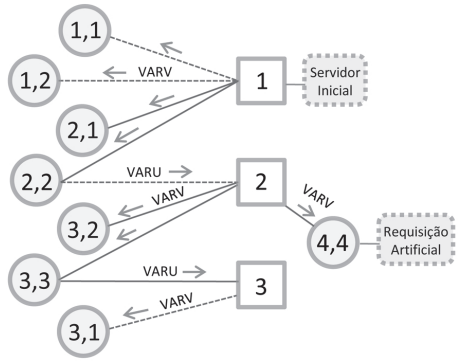

Figura 5.2: Envio das mensagens VARV e VARU

Sendo assim, quando um servidor recebe a mensagem $VARV$ (Algoritmo 11), o servidor realiza um procedimento similar ao descrito anteriormente de selecionar as requisições atendidas pelo servidor que enviou a mensagem, tais que o valor dual v destas requisições não tenha sido calculado (primeiro passo do Simplex), ou que o servidor que enviou $VARV$ seja pai das requisições em T (demais passos) (linha 3 do Algoritmo 12). Esta verificação de passo é necessária para que a cada recebimento da mensagem $VARV$, o valor dual v não seja sempre recalculado e, por consequência, não resulte no cálculo de outros valores duais, fazendo com que o algoritmo não termine (i.e. entre em *loop*). Além disso, os valores das variáveis duais v destas requisições são guardados e o servidor que enviou $VARV$ é atribuído como o pai das requisições em T. Se estas requisições são atendidas por outros servidores, uma mensagem $VARU$ é enviada aos outros servidores que atenderam as requisições, da mesma forma descrita nos parágrafos anteriores.

Um servidor ao receber a mensagem $VARU$ (Algoritmo 13), realiza algumas verificações. Se ele não recebeu o seu valor dual u (primeiro passo do Simplex), ou se quem enviou a mensagem é o seu servidor pai em T (demais passos), então o servidor atribui o valor dual da sua variável u e preenche o caminho em T para o cálculo dessa variável (linhas 2 a 6 do Algoritmo 14). Com isso, o servidor está apto a calcular as variáveis duais v das requisições que ele atendeu. Então, para cada servidor s atendido por esse servidor referente a um conteúdo c, uma mensagem $VARV$ é enviada com o valor da variável dual v relacionada a s e c, juntamente com o caminho em T responsável pelo valor da variável dual v (linhas 9 a 15 do Algoritmo 14). Assim, uma mensagem $VARNBU$ é enviada aos demais servidores (linha 16 do Algoritmo 14) com o valor de sua variável dual u para que este valor seja usado no cálculo dos custos reduzidos das variáveis não básicas (i. e. variáveis das atribuições servidor-requisição que não pertencem a solução). O cálculo dos custos reduzidos será explicado posteriormente. Entretanto, se seu valor dual u já foi calculado (e está primeiro passo do Simplex), uma mensagem $URECUSADO$ é enviada ao servidor que enviou a mensagem $VARU$. Por fim, o servidor verifica se pode mudar de passo (Algoritmo 26) e se recebeu a mensagem $VARNBU$ de todos os servidores (Algoritmo 15).

Ao receber a mensagem $URECUSADO$, um outro servidor ainda não escolhido é selecionado para enviar a mensagem $VARU$ (Algoritmo 40). É necessário enviar $VARU$ para outro servidor para que este possa continuar a construção da árvore de solução T.

Solução Degenerada

No entanto, pode acontecer de todas as requisições do servidor serem atendidas completamente por um único servidor (i. e. 100% do conteúdo da requisição atendido por um servidor) e este único servidor só atender estas requisições, ou seja, a construção da árvore de solução T não pode continuar e o algoritmo para. A Figura 5.3(a) ilustra este caso, que é chamado de degeneração. O servidor 1 atende completamente as requisições dele e do servidor 2 pelos conteúdos 1 e 2. Assim, a construção da árvore com a mensagem $VARV$ pararia nas requisições (Figura 5.3(b)).

Para solucionar isto, é necessário escolher um servidor para continuar. No entanto, esta escolha deve ser feita de maneira inteligente, pois a escolha aleatória de um servidor pode causar a construção de um árvore de solução errada. Portanto, um mecanismo de *convergecast* é realizado a partir das requisições folhas que já pertencem a T para um servidor raiz. Com isso, este servidor tem a informação de quais servidores já pertencem a T e pode decidir, corretamente, qual requisição folha de T deve con-

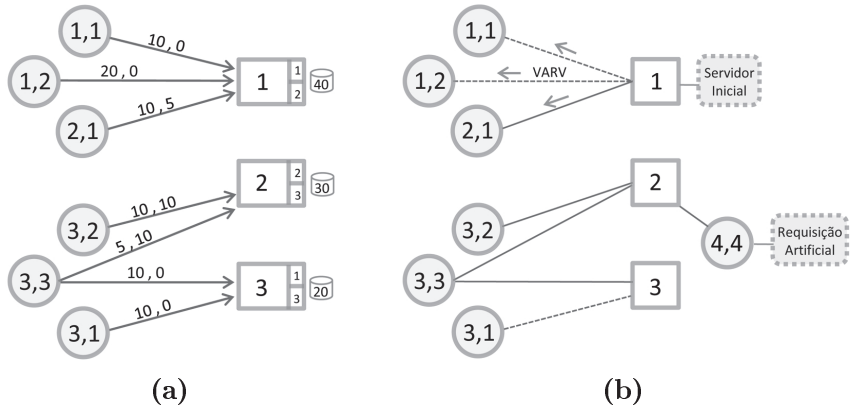

Figura 5.3: (a) Solução degenerada (b) Propagação das variáveis duais parada

tinuar a construção da árvore para um servidor que ainda não foi alcançado e, consequentemente, não está em T. Este mecanismo é realizado somente no primeiro passo do Simplex, quando um servidor recebe $VARV$ e não envia $VARU$ a outro servidor. Assim, uma mensagem $ATENDCOMPREQ$ é enviada ao servidor pai da requisição associada a mensagem $VARV$ com o conjunto de servidores atingidos para o cálculo da variável dual v desta requisição (linhas 19 a 25 do Algoritmo 12). Esta mensagem $ATENDCOMPREQ$ resulta no envio de outras mensagens. Pois quando um servidor recebe $ATENDCOMPREQ$ de todos os servidores que foram atendidos por ele, este envia uma mensagem $ATENDCOMPLETO$ para o servidor da requisição pai dele (Algoritmo 33). De forma similar, quando um servidor de uma requisição recebe $ATENDCOMPLETO$ de todos os servidores que a atendeu, este envia $ATENDCOMP-REQ$ para o seu servidor pai. Esta propagação continua até atingir o servidor raiz (Algoritmo 35). Quando o servidor raiz é atingido, este envia uma mensagem $CONTINUA$ ao servidor de menor identificação de uma requisição r atingida e deixa de ser o servidor raiz (Algoritmo 35). O servidor que recebe $CONTINUA$, por sua vez, envia uma mensagem $DEGENERACAO$ a um servidor s não atingido e marca que a atribuição $r - s$ participa da árvore de solução (Algoritmo 37). Ao receber $DEGENERACAO$, o servidor continua a propagação das variáveis duais e passa a ser o novo servidor raiz para a subárvore que será construída com a continuação da construção de T (Algoritmo 39). A Figura 5.4(a) ilustra a resolução do caso de degeneração, quando um servidor raiz recebe $ATENDCOMPREQ$ de todos os servidores que ele atendeu e envia a mensagem $CONTINUA$ para o servidor de uma requisição alcançada, que, consequentemente, envia $DEGENERACAO$ a um servidor não alcançado, que continua a propagação das variáveis duais para a construção de T. Esta propagação resulta no envio de mais mensagens $ATENDCOMPLETO$ e $ATENDCOMPREQ$ para verificar se todos os servi-

dores foram alcançados e fazem parte de T (Figura 5.4(b)).

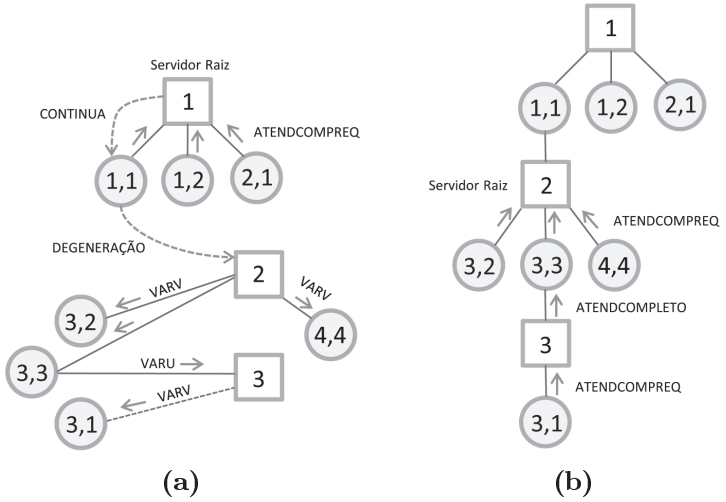

Figura 5.4: (a) Resolvendo o caso de degeneração (b) Envio das mensagens ATENDCOMPLETO e ATENDCOMPREQ

Algoritmo 11: Ao receber uma mensagem $VARV(variavel, c, atendimentoServidor, caminho, caminhoTam, idCiclo)$ do servidor i

1 $procedimentoAtendimento(i, c, atendimentoServidor, caminho, caminhoTam, idCiclo, variavel)$;

O exemplo de propagação das variáveis duais ilustrado na Figura 5.2, também ocorre envios das mensagens $ATENDCOMPLETO$ e $ATENDCOMPREQ$ para verificar se todos os servidores foram alcançados e se todos fazem parte da árvore de solução da mesma maneira explicada anteriormente para resolver o caso da degeneração, mas sem envio das mensagens $CONTINUA$ e $DEGENERACAO$, visto que todos os servidores já foram alcançados e pertencem a T. Estes envios estão ilustrados na Figura 5.5(a). Com isso, obtém-se a árvore de solução com as variáveis duais calculadas (Figura 5.5(b)).

5.2.2.2 Cálculo dos Custos Reduzidos

Como descrito na seção anterior, os servidores ao receberem os valores duais u enviam a mensagem $VARNBU$ aos outros servidores. Isto é necessário, para que cada servidor calcule os custos reduzidos das atribuições locais requisição-servidor que não pertencem a solução.

Sendo assim, ao receber a mensagem $VARNBU$, o servidor inclui o id do servidor que enviou a mensagem na lista $servVaru$, armazena o valor dual u recebido com a mensagem e verifica se recebeu a mensagem $VARNBU$ de todos os servidores (Algoritmo 15).

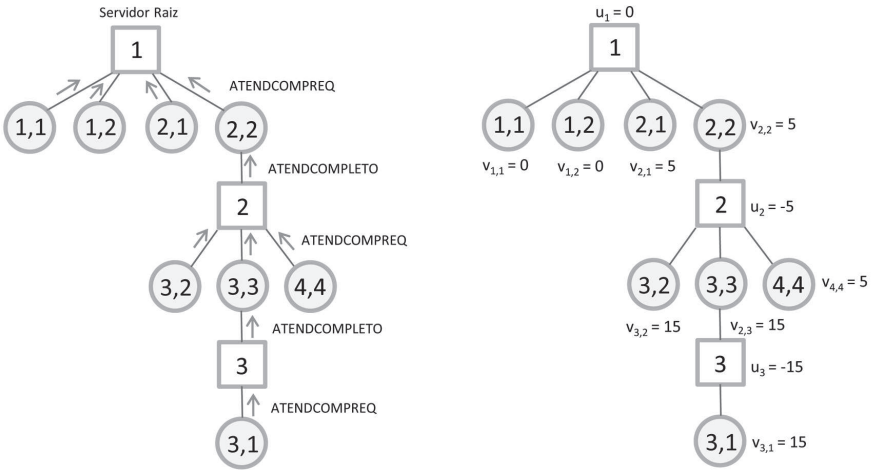

Figura 5.5: (a) Envio das mensagens ATENDCOMPLETO e ATENDCOMPREQ (b) Árvore da solução

Quando o servidor verifica se recebeu $VARNBU$ de todos os servidores, este deve ter recebido $VARNBU$ o número de vezes igual ao número de ciclos encontrados que podem ser processados paralelamente no passo anterior do Simplex. Esta condição é necessária, pois, para cada ciclo que pode ser processado em paralelo, uma reconstrução de parte da árvore de solução T referente ao ciclo é iniciada e cada reconstrução gera o envio da mensagem $VARNBU$ para cada servidor. Assim, este servidor calcula o valor do custo reduzido referente a cada requisição para cada servidor que não atendeu a requisição e seleciona o servidor com o menor custo reduzido negativo (linhas 9 a 16 do Algoritmo 16). Uma mensagem $CREDUZIDO$ é enviada ao servidor responsável por este menor custo juntamente com o caminho referente ao cálculo da variável dual v da requisição. Se este servidor responsável for ele próprio, nenhuma mensagem é enviada e o procedimento do recebimento da mensagem $CREDUZIDO$ é realizado (Algoritmo 18).

A Figura 5.6 ilustra o envio da mensagem $CREDUZIDO$. Neste momento, todos os servidores já receberam a mensagem $VARNBU$ de todos os outros e o custo reduzido de cada requisição local para todo servidor que não a atendeu já foi calculado. No exemplo, o servidor 3 encontrou dois custos reduzidos negativos, como os valores são iguais, um é escolhido e o servidor 3 envia $CREDUZIDO$ ao servidor responsável pelo custo negativo escolhido, o servidor 1.

Quando um servidor recebe a mensagem $CREDUZIDO$ (Algoritmo 17), se este não participa de nenhum ciclo ($idCicloServ = -1$), então o ciclo referente ao caminho da variável dual v recebido com a mensagem e ao caminho da variável dual u deste servidor é

Algoritmo 12: *procedimentoAtendimento*$(i, c, atendimentoServidor, caminho, caminhoTam, idCiclo, variavel)$

```
1  aux := 0; /*variável auxiliar*/
2  escolheuServidor := 0; /*variável auxiliar*/
3  enquanto aux != atendimentoServidor faça
4      Selecione uma req_r tal que cont_r = c e se (passo = 1, servPaiDV_r = {} e
       atendimento_ri > 0) ou se (passo > 1, i = servPaiDV_r e participa_ri = 1) ;
5      dualV_r := variavel;
6      paiServDV_r := i;
7      participa_ri := 1;
8      caminhoReqTam_r := caminhoTam + 1;
9      caminhoReq^r := caminho + {i};
10     se idCiclo != -1 então
11         recAtualizaUReq_ridCiclo = 1;
12     para cada servidor s != i que se (passo = 1, atendimento_rs > 0) ou se (passo > 1,
       participa_rs = 1) faça
13         escolheuServidor := 1;
14         se s != meuId então
15             envia VARU(custo_s - dualV_r, r, caminhoReq^r, caminhoReqTam_r, idCiclo) para o
                   servidor s;
16         senão
17             procVaru(meuId, custo_s - dualV_r, r, c, caminhoReq^r, caminhoReqTam_r, idCiclo);
18     aux := aux + atendimento_ri > 0;
19 se passo = 1 e escolheuServidor = 0 então
20     Selecione uma req_r tal que cont_r = c e i = servPaiDV_r;
21     atingidos := todos os servidores pertencentes a caminhoReq^r;
22     se i != meuId então
23         envia ATENDCOMPREQ(req_r, meuId, atingigos) para o servidor i;
24     senão
25         procAtendeuCompletoReq(r, s, atingidos);
```

Algoritmo 13: Ao receber uma mensagem $VARU(variavel, r, caminho, caminhoTam, idCiclo)$ do servidor i

```
1  procVaru(i, variavel, r, caminho, caminhoTam, idCiclo);
```

encontrado (linha 2 do Algoritmo 18, chamada do procedimento *encontraCiclo*()). Uma mensagem $CICLOINICIADO$ é enviado para cada servidor com o *id* do servidor que enviou $CREDUZIDO$, que é o *id* do ciclo encontrado. Assim, todos os servidores têm o controle de quantos ciclos foram iniciados. Para iniciar o processo de descoberta de ciclo e avisar aos componentes do ciclo encontrado que eles participam de um ciclo, uma mensagem $PCICLO$ é enviada ao primeiro servidor participante do ciclo. Com a mensagem também é enviado um valor infinito, para que este sirva com base de comparação para a seleção do enlace com menor fluxo que pode sair da solução.

O procedimento *encontraCiclo*() (Algoritmo 41) consiste em, dados o caminho de uma variável dual v de uma requisição req e o caminho da variável dual u do próprio servidor, retirar a intersecção entre os dois caminhos para encontrar o caminho entre a requisição req e o próprio servidor. No exemplo do ciclo formado na Figura 5.6, a única

Algoritmo 14: $procVaru(i, variavel, r, caminho, caminhoTam, idCiclo)$

```
1  se (passo = 1 e paiServDU = {}) ou (passo > 1 e meuId = paiServDU) então
2      dualU := variavel;
3      paiServDU := i;
4      reqServDU := r;
5      caminhoServTam := caminhoTam + 1;
6      caminhoServ := caminho + {i, r};
7      se idCiclo! = −1 então
8          recAtualizaU_idCiclo := 1;
9      para cada servidor s faça
10         para cada conteúdo c faça
11             se atendeu_sc > 0 então
12                 se s! = i então
13                     envia VARV(custo_s −
                       dualU, c, atendeu_sc, caminhoServ, caminhoServTam, idCiclo) para o
                       servidor s;
14                 senão
15                     procedimentoAtendimento(s, c, caminhoServ, caminhoServTam, idCiclo,
                       custo_s − dualU);
16     envia VARNBU(dualU, caminhoServ, caminhoServTam) para todos os servidores s, onde
       s! = meuId;
17     servVaru := servVaru + {meuId};
18     varu_meuId := dualU;
19 senão
20     envia URECUSADO(variavel, r) para o servidor i;
21 verificaPasso();
22 verificaRecVarnbu(valor, caminho, caminhoTam);
```

Algoritmo 15: Ao receber uma mensagem $VARNBU(valor, caminho, caminhoTam)$ do servidor i

```
1  servVaru := servVaru + {i};
2  varu_i := valor;
3  verificaRecVarnbu(valor, caminho, caminhoTam);
```

intersecção é o servidor 1.

5.2.2.3 Seleção do Ciclo

A etapa da seleção de ciclo consiste do processo iniciado na etapa anterior de propagação da informação de participação de ciclo aos componentes de um ciclo. Assim, a mensagem $PCICLO$ é enviada ao componente do ciclo posterior ao servidor responsável pelo custo reduzido negativo que formou o ciclo. Esta mensagem é propagada aos demais componentes para avisar que eles participam do ciclo.

Portanto, ao receber a mensagem $PCICLO$ (Algoritmo 19 e 42), o servidor marca que ele ou que a requisição dele pertence ao ciclo da mensagem. Ao marcar uma requisição dele, o servidor verifica se o fluxo entre ele e o servidor que enviou $PCICLO$ é menor que o valor recebido com a mensagem, caso positivo, atualiza o valor de θ. O menor

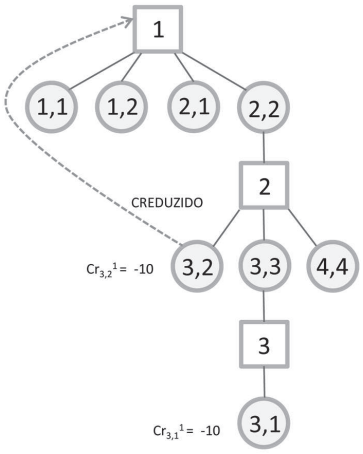

Figura 5.6: Envio da mensagem CREDUZIDO

valor, chamado θ, é referente ao enlace que poderá sair da solução. Se quem foi marcado (servidor ou requisição dele) não é o último componente do ciclo, a mensagem $PCICLO$ com o valor θ é enviada para o próximo servidor do ciclo. Se for o último do ciclo, isto indica que o menor θ foi encontrado. Com isso, uma mensagem $ATUALIZACICLO$ com valor θ é enviada ao primeiro componente do ciclo, para que esta mensagem seja propagada aos demais componentes do ciclo e, assim, estes possam atualizarem os fluxos do ciclo em relação a θ. A Figura 5.7(a) ilustra a propagação da mensagem $PCICLO$ e o envio da mensagem $ATUALIZACICLO$. No entanto, se o servidor ou a requisição dele que está no ciclo da mensagem $PCICLO$ recebida já pertence a um outro ciclo, um processo de cancelamento de ciclo é iniciado. Este processo será explicado posteriormente.

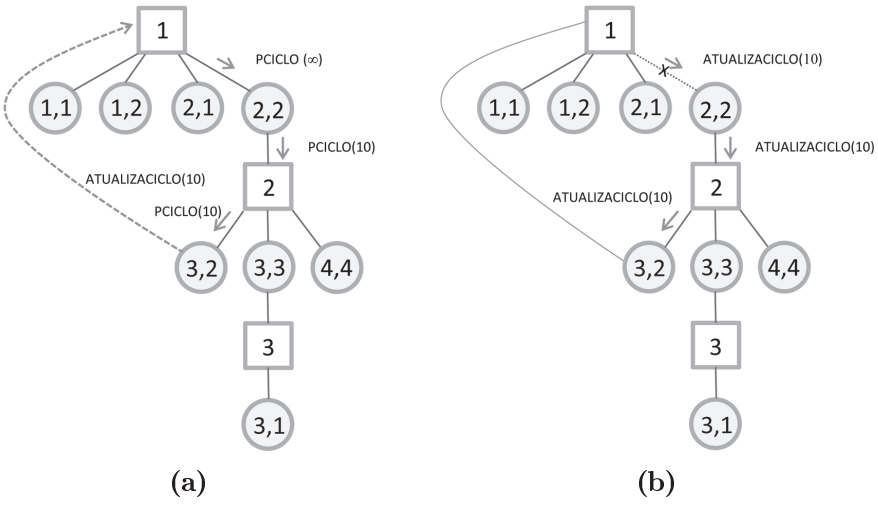

Figura 5.7: (a) Propagação da mensagem PCICLO (b) Propagação da mensagem ATUALIZACICLO

Algoritmo 16: $verificaRecVarnbu(valor, caminho, caminhoTam)$

```
1  se cada servidor aparece numCiclo vezes em servVaru então
2      servVaru := {};
3      numCiclo := 0;
4      custoReduzido := 0; /*variável auxiliar*/
5      varnbMenor := 0; /*variável auxiliar*/
6      varnbReq := −1; /*variável auxiliar*/
7      varnbServ := −1; /*variável auxiliar*/
8      se meuId! = servArtificial então
9          para cada requisição r faça
10             para cada servidor s faça
11                 se atendimento_rs = 0 e paticipa_rs = 1 e s possui cont_r então
12                     custoReduzido := custo_s − varu_s − dualV_r;
13                     se custoReduzido < varnbMenor então
14                         varnbMenor := custoReduzido;
15                         varnbReq := i;
16                         varnbServ := j;
17         se varnbMenor < 0 então
18             se varnbServ! = meuId então
19                 envia
                   CREDUZIDO(varnbMenor, cont_varnbReq, varnbReq, caminhoReq^varnbReq,
                   caminhoReqTam_varnbReq) para o servidor varnbServ;
20             senão
21                 procReduzido(meuId, varnbMenor, cont_varnbReq, varnbReq, caminhoReq^varnbReq,
                   caminhoReqTam_varnbReq);
```

Algoritmo 17: Ao receber uma mensagem $CREDUZIDO(cr, c, r, caminho, caminhoTam)$ do servidor i

```
1  procReduzido(i, cr, c, r, caminho, caminhoTam);
```

Quando um servidor recebe a mensagem $ATUALIZACICLO$ (Algoritmo 20 e 43), o servidor verifica se é ele ou é uma requisição dele que participa do ciclo, atualizando o fluxo correspondente de θ. Se é ele próprio que participa, atualiza o fluxo dele para o servidor que enviou a mensagem com $+\theta$ e o fluxo dele para o próximo componente do ciclo com $-\theta$. Isto significa, que o servidor passa a atender com uma quantidade maior o servidor que enviou a mensagem e menor o próximo componente do ciclo. No entanto, se for uma requisição dele que participa do ciclo, o fluxo da requisição para o servidor que enviou a mensagem é atualizado com $-\theta$ e da requisição para o próximo componente do ciclo é atualizado com $+\theta$. Isto significa, que a requisição será atendida em menor quantidade pelo servidor que enviou a mensagem e maior pelo próximo componente do ciclo. Assim, se o servidor ou a requisição dele não for o último componente do ciclo, a mensagem $ATUALIZACICLO$ é enviada para o próximo componente do ciclo. A Figura 5.7(b) ilustra a propagação da mensagem $ATUALIZACICLO$. Entretanto, se for o último, uma mensagem $CICLOTERMINADO$ é enviado para todos os servidores com o id do ciclo. Com isso, todos os servidores têm o controle de quais ciclos foram

Algoritmo 18: $procReduzido(i, cr, c, r, caminho, caminhoTam)$

```
1  se idCicloServ = −1 então
2      cicloServ := encontraCiclo(r, i, caminho, caminhoTam);
3      idCicloServ := i;
4      pos := 0; /*variável auxiliar*/
5      valor := 'infinito'; /*variável auxiliar*/
6      envia CICLOINICIADO(i) para todos os servidores s, onde s! = meuId;
7      se cicloServ_pos! = {meuId} então
8          envia PCICLO(cr, valor, pos, idCicloServ, cicloServ, cicloServTam) para o servidor de
           cicloServ_pos;
9      senão
10         procCiclo(cr, valor, meuId, pos, idCicloServ, cicloServ, cicloServTam);
```

Algoritmo 19: Ao receber uma mensagem $PCICLO(cr, valor, pos, idCiclo, ciclo, cicloTam)$ do servidor i

```
1  procCiclo(cr, valor, i, pos, idCiclo, ciclo, cicloTam);
```

terminados. Por fim, o servidor verifica se pode mudar de passo (Algoritmo 26).

Algoritmo 20: Ao receber uma mensagem $ATUALIZACICLO(cr, achou, valor, pos, idCiclo, ciclo, cicloTam, variavelSm)$ do servidor i

```
1  atualizaCiclo(cr, achou, valor, i, pos, idCiclo, ciclo, cicloTam, variavelS);
```

5.2.2.4 Cancelamento de Ciclo

Esta etapa é necessária para cancelar o ciclo de maior custo reduzido quando um servidor ou uma requisição de um servidor, que já pertence a um ciclo, recebe a mensagem $PCICLO$ de um outro ciclo. Sendo assim, o processo de cancelamento de um ciclo (Algoritmo 44) consiste em desfazer a marcação de ciclo, feita anteriormente, atribuída a um servidor ou a uma requisição de uma posição *pos* do ciclo.

Quando o procedimento de cancelamento é chamado (no Algoritmo 42), verifica-se qual ciclo deve ser cancelado, se é o ciclo na qual o servidor ou a requisição já pertence ($cicloExistente$) ou é o ciclo da mensagem $PCICLO$ que desencadeou o cancelamento ($cicloMensagem$). Assim, o ciclo com maior custo reduzido é escolhido. Se o ciclo escolhido para ser cancelado é o $cicloExistente$, é necessário avisar aos demais componentes deste ciclo sobre o cancelamento, então um processo de cancelamento é iniciado em direção aos componentes antecessores do *componenteCancelado* (servidor ou a requisição da posição *pos* do $cicloMensagem$) e outro em direção aos sucessores. Para isto, é preciso selecionar o ciclo da requisição ou do servidor do *componenteCancelado*. Se algum enlace já tiver sido selecionado para sair deste ciclo, esta seleção é desfeita. Uma mensagem $CICLOCANCELADO$ é enviada a todos os servidores com o *id* do

cicloExistente para que eles tenham o controle de quais ciclos foram cancelados. Se existir um componente sucessor ao *componenteCancelado* no ciclo ($pos + 1$), o processo de cancelamento dos componentes sucessores é iniciado com o componente imediatamente sucessor ao *componenteCancelado* (procedimento $cancelaFrente()$). Após este procedimento, o processo para cancelar os componentes antecessores é realizado a partir do *componenteCan− celado* (procedimento $cancelaAtras()$). Com isso, o $cicloMensagem$ que desencadeou o cancelamento de ciclo pode, agora, continuar sua propagação (procedimento $procCiclo()$).

No entanto, se o ciclo escolhido de menor custo reduzido não foi o ciclo no qual o servidor ou a requisição já pertencia, e sim o ciclo referente a mensagem $PCICLO$ responsável pelo cancelamento, o $cicloMensagem$, uma mensagem $CICLOCANCELADO$ é enviada a todos os servidores com o *id* do $cicloMensagem$ para que eles tenham o controle de quais ciclos foram cancelados. Se algum enlace já tiver sido selecionado para sair deste ciclo, esta seleção também é desfeita. Como a propagação do $cicloMensagem$ ainda estava sendo realizada, nenhum componente sucessor ao *componenteCancelado* tinha sido atingido por esta propagação. Assim, somente o processo de cancelamento dos componentes antecessores ao *componenteCancelado* é realizado (procedimento $cancelaAtras()$). Por fim, o servidor verifica se pode mudar de passo (procedimento $verificaPasso()$).

O procedimento $cancelaFrente()$ (Algoritmo 45) consiste em desmarcar todos os componentes sucessores de um servidor ou de uma requisição (*componenteCancelado*) em um ciclo. As informações da posição do componente que deve ser cancelado (*pos*), do *id* do ciclo (*idCiclo*) e do ciclo são passadas quando este procedimento é chamado. Assim, o componente da posição *pos* é verificado, se este for diferente do *meuId* do servidor, uma mensagem $CANCELACICLOF$ é enviada ao servidor do componente da posição *pos*. Caso contrário, o cancelamento de ciclo do componente da posição *pos* é realizado (procedimento *procCancelaCicloFrente*). No procedimento *procCancelaCicloFrente* (Algoritmo 46), a requisição *req* do componente da posição *pos* do ciclo é verificado. Se existe requisição ($req! = -1$) e ela, realmente, pertence ao ciclo *idCiclo*, então esta é desmarcada e não pertence mais ao ciclo, e se, também o valor θ do ciclo tiver sido encontrado ($tetaReq! = -1$), a atualização dos fluxos feita pela propagação da mensagem $ATUALIZACICLO$ é desfeita. No entanto, caso não exista requisição ($req = -1$, o componente é o próprio servidor) e o servidor pertence ao ciclo *idCiclo*, então ele é desmarcado e não pertence mais ao ciclo, e se, também o valor θ do ciclo tiver sido encontrado ($teta! = -1$), a atualização dos fluxos feita pela propagação da mensagem $ATUALIZACICLO$ é desfeita. Se, depois de ter cancelado o componente, ainda possuem

componentes sucessores no ciclo para serem cancelados, o *cancelaFrente*() é novamente chamado com a posição $pos + 1$.

O procedimento *cancelaAtras*() (Algoritmo 47) consiste em desmarcar todos os componentes antecessores de um servidor ou de uma requisição (*componenteCancelado*) em um ciclo. Este procedimento é similar ao *cancelaFrente*, eles diferem quanto aos enlaces que têm seus fluxos atualizados. Com isso, as informações da posição do *componenteCancelado* (*pos*), do *id* do ciclo (*idCiclo*) e do ciclo são passadas quando este procedimento é chamado. Assim, a requisição da posição *pos* é verificada, se existe requisição ($req! = -1$) e ela pertence ao ciclo *idCiclo*, então esta é desmarcada e não pertence mais ao ciclo, e se, também o valor θ do ciclo tiver sido encontrado ($tetaReq! = -1$), a atualização dos fluxos feita pela propagação da mensagem $ATUALIZACICLO$ é desfeita. No entanto, caso não exista requisição ($req = -1$, o componente é o próprio servidor) e o servidor pertence ao ciclo *idCiclo*, então ele é desmarcado e não pertence mais ao ciclo, e se, também o valor θ do ciclo tiver sido encontrado ($teta! = -1$), a atualização dos fluxos feita pela propagação da mensagem $ATUALIZACICLO$ é desfeita. Se, depois de ter cancelado o componente, ainda existirem componentes antecessores no ciclo para serem cancelados, uma mensagem $CANCELACICLO$ com $pos - 1$ é enviada ao antecessor de *componenteCancelado*, se este não é *meuId*, caso contrário, o *cancelaAtras*() é novamente chamado com a posição $pos - 1$.

Para ilustrar um cancelamento de ciclo, considere que o servidor 1 envia $CREDUZIDO$ para o servidor 4 (Figura 5.8(a)), que inicia a propagação da mensagem $PCICLO$ referente ao ciclo do servidor 1. Considere também, que o servidor 3 envia $CREDUZIDO$ para o servidor 1, que inicia a propagação da mensagem $PCICLO$ referente ao ciclo do servidor 3 (Figura 5.8(b)). Suponha que a mensagem $PCICLO$ do ciclo 1 alcance o servidor 2 primeiro que a do ciclo 3. Assim, quando a mensagem $PCICLO$ do ciclo 3 alcança o servidor 2, um dos ciclos, 1 ou 3, deve ser cancelado. É cancelado o ciclo com maior custo reduzido. Seja o ciclo 3, o de menor custo reduzido, então é necessário cancelar o ciclo 1. Uma mensagem $CANCELACICLO$ é enviada ao antecessor do servidor 2 no ciclo 1 e uma mensagem $CANCELACICLOF$ é enviada ao sucessor (Figura 5.8(c)). Depois disso, o ciclo 3 pode continuar a propagação da mensagem $PCICLO$ (Figura 5.8(d)).

Ao receber a mensagem $CANCELACICLOF$ (Algoritmo 21), o servidor realiza o procedimento *procCancelaCicloFrente* descrito anteriormente, com a posição *pos* do componente do ciclo *idCiclo* proveniente da mensagem. E ao receber a mensagem $CAN-$

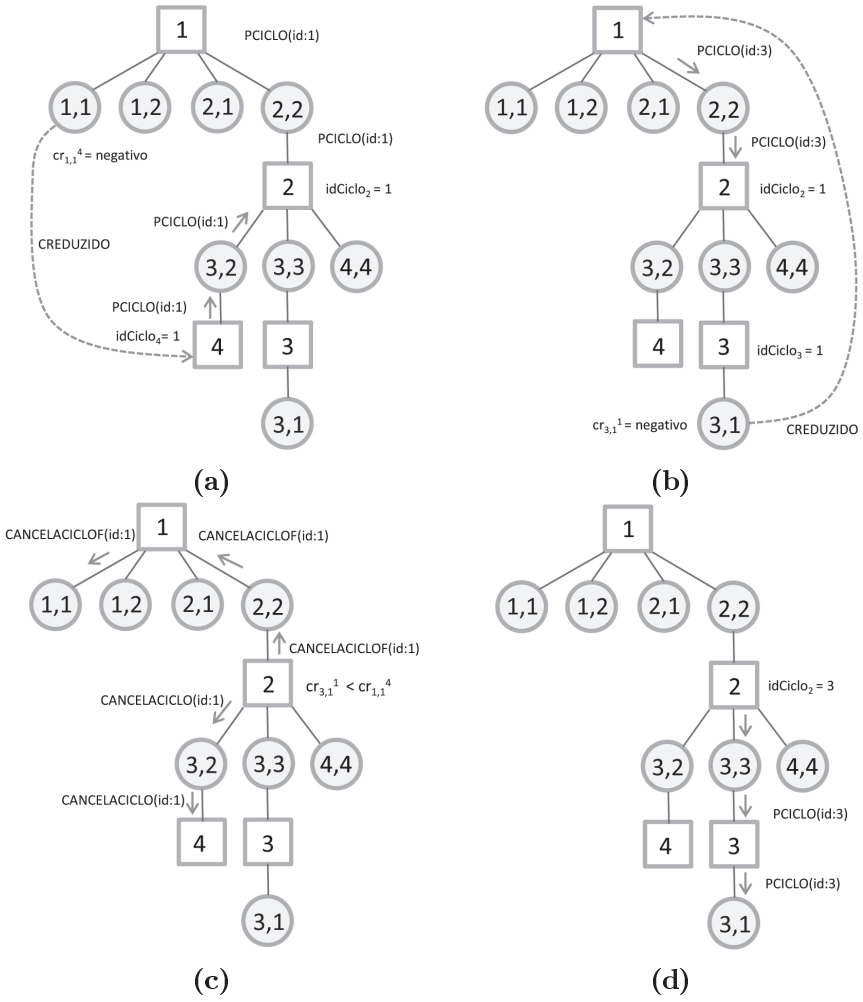

Figura 5.8: (a) Propagação da mensagem PCICLO do servidor 1; (b) Propagação da mensagem PCICLO do servidor 3; (c) Propagação da mensagem CANCELACICLO e CANCELACICLOF; (d) Continua a propagação da mensagem PCICLO do servidor 3

$CELACICLO$ (Algoritmo 22), o servidor realiza o procedimento $cancelaAtras()$ também descrito anteriormente, com a posição pos do componente do ciclo $idCiclo$ proveniente da mensagem.

Algoritmo 21: Ao receber uma mensagem $CANCELACICLOF(pos, idCiclo, ciclo, cicloTam)$ do servidor i

1 $procCancelaCicloFrente(pos, idCiclo, ciclo, cicloTam)$

Um servidor ao receber a mensagem $CICLOINICIADO$ (Algoritmo 23), ele adiciona o id do servidor recebido com a mensagem no conjunto de servidores $servCicloIniciado$ que iniciaram o processo de descoberta do ciclo e realiza a verificação de mudança de passo (Algoritmo 26).

Quando um servidor recebe a mensagem $CICLOCANCELADO$ ou $CICLOTER-$

Algoritmo 22: Ao receber uma mensagem $CANCELACICLO(pos, idCiclo, ciclo, cicloTam)$ do servidor i

1 $cancelaAtras(pos, idCiclo, ciclo, cicloTam)$;

$MINADO$ (Algoritmos 24 e 25), ele retira o id do servidor recebido com a mensagem no conjunto de servidores $servCicloIniciado$ que iniciaram o processo de descoberta do ciclo e realiza a verificação de mudança de passo (Algoritmo 26).

Algoritmo 23: Ao receber uma mensagem $CICLOINICIADO(id)$ do servidor i

1 $servCicloIniciado := servCicloIniciado + \{id\}$;
2 $verificaPasso()$;

Algoritmo 24: Ao receber uma mensagem $CICLOCANCELADO(id)$ do servidor i

1 $servCicloIniciado := servCicloIniciado - \{id\}$;
2 $verificaPasso()$;

5.2.2.5 Mudança de Passo

Esta etapa é para verificar se um servidor pode mudar de passo. Um servidor está apto a mudar de passo se ele possui a informação de que todos os ciclos descobertos iniciados já terminaram ou foram cancelados.

Assim, o procedimento $verificaPasso()$ (Algoritmo 26), chamado em outros procedimentos, consiste do servidor verificar se o conjunto $servCicloIniciado$ está vazio. Se estiver vazio, isto indica que todos os servidores que iniciaram o processo de descoberta de ciclo já terminaram e, então, o servidor pode mudar de passo. Assim, verifica-se também, o conjunto $varSainte$ relacionado a variável que sai da solução. Se o conjunto $varSainte$ não estiver vazio, indica que um processo de descoberta de ciclo referente a uma requisição do próprio servidor conseguiu terminar, sendo assim, é necessário que a parte da árvore de solução T relacionada ao ciclo seja atualizada. Para isto, o processo de atualização dos pais dos componentes do ciclo é iniciado (linha 4 do Algoritmo 26, chamada do procedimento $procAtualizaPai()$). Seguindo o exemplo da Figura 5.7(b), o servidor 3 inicia o processo de atualização dos pais dos componentes do ciclo (Figura 5.9).

5.2.2.6 Reconstrução da árvore de solução

A reconstrução da árvore de solução se inicia ao mudar de passo, começando com a atualização dos pais dos servidores e/ou requisições que foram modificados com a adição

Algoritmo 25: Ao receber uma mensagem $CICLOTERMINADO(id)$ do servidor i

```
1 servCicloIniciado := servCicloIniciado − {id};
2 verificaPasso();
```

Algoritmo 26: $verificaPasso()$

```
1 se servCicloIniciado = {} então
2     passo := passo + 1;
3     se varSainte! = {} então
4         procAtualizaPai(varSainte);
5         varSainte := {};
```

das variáveis de custos reduzidos negativos que formaram os ciclos que terminaram. Após esta atualização, pode-se recalcular os valores duais dos servidores e das requisições das partes da árvore de solução T que devem ser reconstruídas.

Sendo assim, o procedimento *procAtualizaPai*() (Algoritmo 27), chamado na mudança de passo, consiste em determinar o sentido no qual a atualização dos pais do componente do ciclo terminado será feita e iniciá-la. O servidor, então, verifica em seu conjunto *varSainte*, se o elemento *serv* é igual ao elemento *paiServ*, se for igual, indica que a atualização dos pais é realizada a partir do servidor referente ao elemento *servReq* de *varSainte* até a última posição do ciclo (sentido 1). Assim, se *servReq* não for o próprio servidor, uma mensagem *ATUALIZAPAI* é enviada a *servReq* com sua posição no ciclo, e se ele for, o procedimento *atualizaPai*() é realizado (linha 1 a 7 do Algoritmo 27). A Figura 5.9(a) ilustra este caso, em que o servidor 3 envia a mensagem *ATUALIZAPAI* ao servidor 2 da requisição pertencente a atribuição que saiu a solução (onde *serv* de *varSainte* é o servidor 1, *paiServ* é também o servidor 1 e *servReq* é o servidor 2). No entanto, se o elemento *serv* é diferente do elemento *paiServ*, a atualização dos pais é feita a partir do servidor *serv* até a primeira posição do ciclo (sentido 2) e uma mensagem *ATUALIZAPAI* é enviada ao servidor *serv* com sua posição no ciclo, se este não for o próprio servidor, e se *serv* for o próprio servidor, o procedimento *atualizaPai*() é realizado (linha 8 a 14 do Algoritmo 27).

Ao receber a mensagem *ATUALIZAPAI* (Algoritmo 28), o servidor realiza o procedimento *atualizaPai*() (Algoritmo 50). Neste procedimento, o servidor verifica o componente da posição *pos* (relativa a mensagem *ATUALIZAPAI* recebida). Se o componente não for o primeiro nem o último do ciclo referente a mensagem *ATUALIZAPAI*, a requisição *req* deste componente é verificada. Se a requisição existe ($req! = -1$) e o sentido de atualização dos pais é 2, o servidor pai de *req* é atualizado e passa ser o servidor do componente da posição $pos - 1$ e a mensagem *ATUALIZAPAI* é enviada ao atual servidor

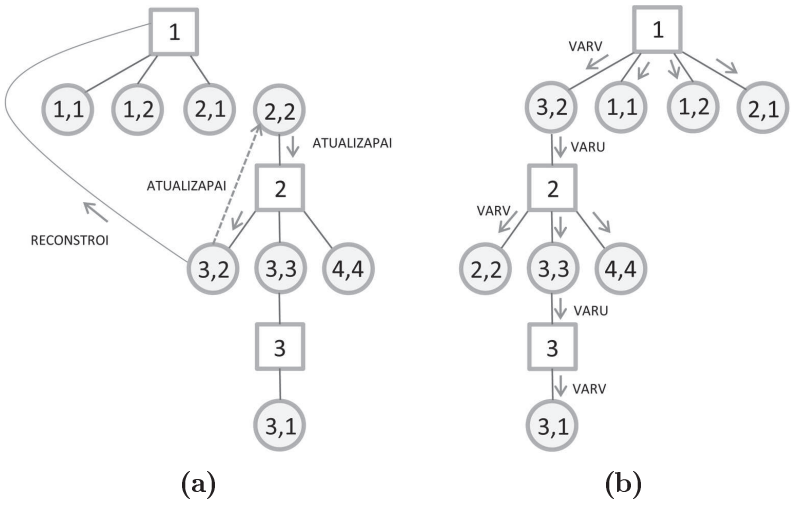

Figura 5.9: (a) Envio da mensagens ATUALIZAPAI e RECONSTROI (b) Nova propagação das mensagens VARU e VARV

Algoritmo 27: *procAtualizaPai*(req)

```
1  se varSainte.serv = varSainte.paiReq então
2      pos := funcServCiclo(CicloReq_req, varSainte.servReq, req);
3      sentido := 1; /*Sentido para percorrer o ciclo*/
4      se varSainte.servReq! = meuId então
5          envia ATUALIZAPAI(pos, idCicloReq_req, cicloReq_req, cicloReqTam_req, sentido) para
               varSainte.servReq;
6      senão
7          atualizaPai(pos, idCicloReq_req, cicloReq_req, cicloReqTam_req, meuId, sentido);
8  senão
9      pos := funcServCiclo(CicloReq_req, varSainte.serv, −1);
10     sentido := 2; /*Sentido para percorrer o ciclo*/
11     se varSainte.serv! = meuId então
12         envia ATUALIZAPAI(pos, idCicloReq_req, cicloReq_req, cicloReqTam_req, sentido) para
               varSainte.serv;
13     senão
14         atualizaPai(pos, idCicloReq_req, cicloReq_req, cicloReqTam_req, meuId, sentido);
```

pai de req. Porém, se a requisição existe ($req! = -1$) e o sentido de atualização dos pais é 1, o servidor pai de req é atualizado e passa a ser o servidor do componente da posição $pos + 1$ e a mensagem $ATUALIZAPAI$ é enviada para o servidor pai req. No entanto, se a requisição não existe ($req = -1$, o componente é o próprio servidor) e o sentido de atualização dos pais é 2, a requisição pai do próprio servidor é atualizada e passa a ser a requisição do componente da posição $pos - 1$ e a mensagem $ATUALIZAPAI$ é enviada para o servidor da requisição pai. E por fim, se a requisição não existe ($req = -1$) e o sentido de atualização dos pais é 1, a requisição pai do próprio servidor é atualizada e passa a ser a requisição do componente da posição $pos+1$ e a mensagem $ATUALIZAPAI$ é enviada para o servidor da requisição pai.

Entretanto, se o componente da posição *pos* for o primeiro ou o último do ciclo, isto significa que a atualização dos pais terminou e que reconstrução da subárvore alterada por esta atualização pode ser iniciada. No entanto, a reconstrução não é iniciada imediatamente, pois se existir outras reconstruções decorrentes de outros ciclos, estas devem ser iniciadas após as atualizações de pais de todos os ciclos estejam concluídas. Assim, uma mensagem $ACABOUATUALIZA$ é enviada a todos os servidores. Se o componente da posição *pos* for o primeiro, atualiza-se a requisição pai do servidor, caso o componente da posição *pos* for o último, atualiza-se o servidor pai da requisição req. Depois é feita uma verificação para saber se todas as atualizações dos pais foram concluídas (Algoritmo 30). Se já foram concluídas e o componente *pos* é o primeiro, uma mensagem $RECONSTROI$ é enviada ao servidor da requisição pai do servidor do componente *pos*. Já se o componente *pos* é o último, uma mensagem $RECONSTROI$ é enviada ao servidor pai da requisição do componente *pos*. Na Figura 5.9(a), considere que a mensagem $ACABOUATUALIZA$ já foi enviada a todos os servidores, concluindo a atualização dos pais. Com isso, o servidor 3 envia a mensagem $RECONSTROI$ ao servidor 1 que é o servidor pai da requisição do componente *pos*, última posição, do ciclo.

Algoritmo 28: Ao receber uma mensagem $ATUALIZAPAI(pos, idCiclo, ciclo, cicloTam, sentido)$ do servidor i

```
1 atualizaPai(pos, idCiclo, ciclo, cicloTam, sentido);
```

Ao receber a mensagem $ACABOUATUALIZA$ (Algoritmo 29), o número de atualizações de pais concluídas ($nAtualiza$) é incrementada e a posição do último componente referente ao ciclo proveniente da mensagem é armazenado. Esta informação é necessária para saber por onde se deve iniciar a reconstrução (Algoritmo 48). Depois é feita uma verificação para saber se todas as atualizações dos pais foram concluídas (Algoritmo 30). Este procedimento foi descrito anteriormente.

Algoritmo 29: Ao receber uma mensagem $ACABOUATUALIZA(id, pos)$ do servidor i

```
1 procAcabouAtualiza(id, pos);
2 verificaAcabouAtualiza();
```

Ao receber a mensagem $RECONSTROI$ (Algoritmo 31), o servidor realiza o procedimento $reconstroi()$ (Algoritmo 49). Se nenhuma requisição req é recebida com a mensagem ($req = -1$), isto significa que a reconstrução é feita nas subárvores filhas do próprio servidor em T. Assim, uma mensagem $ATUALIZAU$ é enviada ao servidor da requisição pai do servidor que recebeu $RECONSTROI$(com a informação de conteúdo $= -1$) , para informar que a requisição não será reconstruída (i. e. a sua variável dual v

não será recalculada) pelo ciclo referente a mensagem $RECONSTROI$ e para que esta informação seja propagada a outros servidores. É necessário avisar os servidores que eles não serão reconstruídos, para que estes propaguem seus valores duais u, pois todo servidor espera receber esses valores referentes a cada reconstrução de ciclo terminado de cada servidor para o cálculo dos custos reduzidos. A reconstrução é, então, iniciada e para cada requisição r atendida pelo servidor que recebeu $RECONSTROI$, a mensagem $VARV$ é enviada ao servidor da requisição r. Nesse momento, a mensagem $VARNBU$ também é enviada aos demais servidores com o valor de sua variável dual para que este valor seja usado no cálculo dos custos reduzidos. No entanto, se uma requisição req é recebida com a mensagem $(req! = -1)$, isto significa que a reconstrução é feita nas subárvores filhas da requisição req em T. Com isso, uma mensagem $ATUALIZAU$ é enviada ao servidor pai de req(com a informação de conteúdo $= -2$) , para informar que ele não será reconstruído pelo ciclo referente a mensagem $RECONSTROI$ recebida, ou seja, notifica que a variável dual dele não será modificada. Para cada servidor s que atendeu a requisição req, em que a atribuição de req a s pertence a T, a mensagem $VARU$ é enviada a s. Por fim, o servidor verifica se recebeu a mensagem $VARNBU$ de todos os servidores para o cálculo dos custos reduzidos(Algoritmo 16). Como no exemplo da Figura 5.9(a), o servidor 1, raiz da árvore, é quem recebeu $RECONSTROI$, não é necessário o envio da mensagem $ATUALIZAU$ e somente a reconstrução da subárvore filha alterada é iniciada com o envio da mensagem $VARV$, conforme pode ser visto na Figura 5.9(b). No entanto, considere agora o caso da Figura 5.10, a mensagem $RECONSTROI$ é recebida pelo servidor 2. Desse modo, além de iniciar a reconstrução da subárvore filha, é necessário também enviar a mensagem $ATUALIZAU$ ao servidor da requisição pai dele, para informar que ele não será reconstruído pelo ciclo em questão.

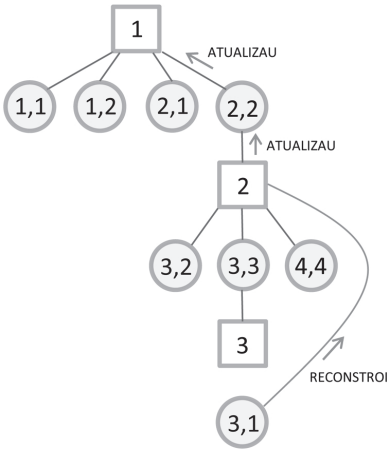

Figura 5.10: Envio da mensagem ATUALIZAU

Um servidor ao receber a mensagem $ATUALIZAU$ (Algoritmo 32) realiza o procedimento $procAtualizaU()$ (Algoritmo 51). Com a mensagem, é recebida as informações de conteúdo, da requisição req e do id do ciclo ($idCiclo$) que resultou no envio da mensagem $ATUALIZAU$. Se não houver informação de conteúdo (conteúdo $= -1$ ou $= -2$), isto indica que se o servidor ainda não recebeu $ATUALIZAU$ referente ao ciclo $idCiclo$ (para o conteúdo $= -2$), a mensagem $VARNBU$ é enviada aos demais servidores com o valor de sua variável dual e para cada requisição r atendida pelo servidor, a mensagem $ATUALIZAU$ é enviada ao servidor da requisição r com o conteúdo $cont_r$. Já se o conteúdo é igual a -1 e a requisição req ainda não recebeu $ATUALIZAU$ referente ao ciclo $idCiclo$, para cada servidor s que atendeu req, a mensagem $ATUALIZAU$ é enviada a s com conteúdo igual a -2. No entanto, se houver informação de conteúdo (conteúdo diferente de -1 e -2) e o servidor ainda não recebeu $ATUALIZAU$ referente ao ciclo $idCiclo$, então para cada requisição $r! = req$ atendida pelo servidor em que o $cont_r$ é igual ao conteúdo, a mensagem $ATUALIZAU$ é enviada ao servidor da requisição req com conteúdo igual a -2 e requisição r.

No final desta etapa, os servidores devem ter recebido os valores duais u modificados e os que não foram alterados também para iniciar a etapa do cálculo dos custos reduzidos. Se nenhum custo reduzido negativo for encontrado o algoritmo termina, caso contrário, prossegue o algoritmo com a seleção de ciclos.

Algoritmo 30: $verificaAcabouAtualiza()$

```
1  se nAtualiza = numCiclos então
2      nAtualiza = −1;
3      se varSainte.req! = −1 então
4          req := varSainte.r;
5          ciclo := cicloReq_req;
6          cicloTam := cicloReqTam_req;
7          pos := varSainte.pos;
8          idCiclo := varSainte.ciclo;
9          se pos = cicloTam − 1 então
10             s := servPaiDV_req;
11             se s! = meuId então
12                 envia RECONSTROI(cont_req, −1, idCiclo) para s;
13             senão
14                 reconstroi(cont_req, −1, idCiclo, meuId);
15         senão
16             se pos = 0 então
17                 s := servPaiDU;
18                 se s! = meuId então
19                     envia RECONSTROI(cont_req, reqPaiDU), idCiclo) para s;
20                 senão
21                     reconstroi(cont_req, reqPaiDU), idCiclo, meuId);
```

Algoritmo 31: Ao receber uma mensagem $RECONSTROI(c, req, idCiclo)$ do servidor i

```
1 reconstroi(c, req, idCiclo, i);
```

Algoritmo 32: Ao receber uma mensagem $ATUALIZAU(pos, idCiclo, ciclo, cicloTam)$ do servidor i

```
1 procAtualizaU(pos, idCiclo, ciclo, cicloTam);
```

5.3 Análise de Complexidades

Nesta seção, os limites superiores para o número de mensagens trocadas, para o tempo e para memória são apresentados.

5.3.1 Solução Inicial

Para obter uma solução inicial viável para o Simplex de Transporte através da heurística DistPAC, cada servidor tenta alocar cada requisição sua localmente. Caso não seja possível alocar localmente, no pior caso, uma mensagem de tentativa de alocação é enviada a todos os outros servidores. Para este caso, n mensagens são enviadas, onde n é o número de servidores. Se não for possível alocar a requisição, uma mensagem é enviada a um servidor artificial, indicando que o conteúdo requisitado será atendido pelo servidor origem do conteúdo. Logo, para o atendimento de todas as requisições, o número total de mensagens enviadas é da ordem de $n.m$, onde m é o número de requisições.

A complexidade de tempo global em modelos distribuídos assíncronos assume que as computações locais não gastam tempo, e que o tempo para enviar uma mensagem para um nó vizinho é $O(1)$. A complexidade de tempo global é então definida como o número de mensagens na maior cadeia causal do tipo "recebeu uma mensagem e enviou uma mensagem como consequência" ocorrida durante a execução do algoritmo [2]. Ao tentar alocar uma requisição, suponha que seja necessário enviar mensagens de tentativa de alocação a todos os servidores, resultando no envio de, no pior caso, $O(n)$ mensagens. Então, uma requisição, no pior caso, é atendida pelo último servidor ou é atendida pelo servidor artificial, resultando em uma cadeia de mensagens de comprimento $O(n)$.

A complexidade de tempo local refere-se ao tempo de realizar computações locais em um nó entre o recebimento de duas mensagens consecutivas. Assim, o tempo local gasto por cada nó i (servidor i) é $O(r_i)$, onde r_i é número de requisições do servidor i (requisições dos clientes mais próximos ao servidor).

A complexidade de memória refere-se ao espaço de memória para armazenar as infor-

mações necessárias para realizar as computações locais em um servidor. As informações necessárias armazenadas por cada servidor são as requisições dos clientes mais próximos, logo, $O(r_i)$.

A heurística DistPAC tem, no pior caso, complexidades de $O(n.m)$ mensagens, de $O(n)$ tempo global, de $O(r_i)$ tempo local e de $O(r_i)$ de memória.

5.3.2 Simplex de Transporte

A partir da solução obtida pelo DistPAC, um único servidor inicia a construção da árvore T desta solução para o cálculo das variáveis duais u dos servidores (sT) e v de todas as requisições ($reqT$). Esta construção atinge, então, todos os nós servidores e todos nós requisições, resultando no envio de $n.m$ mensagens. No entanto, para evitar que o algoritmo pare no caso de uma degeneração, toda vez que um nó requisição $reqT(ri)$ não continua a construção da árvore no primeiro passo do Simplex, mensagens em *convergecast* são enviadas ao nó servidor raiz em questão, resultando no envio de mais $n+m$ mensagens. Como este *convergecast* só é realizado em uma passada do Simplex, este número de mensagens não é considerado na complexidade. Quando esta construção atinge um nó servidor $sT(i)$, este propaga o valor de sua variável dual u para todos os servidores, o que resulta no envio $O(n^2)$ mensagens. Cada servidor $sT(i)$ ao receber o valor das variáveis duais u dos outros servidores, encontra localmente uma atribuição requisição-servidor $aT(sT, reqT)$ que não pertence a árvore de solução com menor custo reduzido negativo para entrar na solução, originando um ciclo. Este ciclo, no pior caso, conterá os n nós servidores sT e, consequentemente, n nós requisições $reqT$. O ciclo é identificado pelo id do servidor sT_i que o originou.

Ao encontrar um ciclo, o servidor $sT(i)$ propaga a informação de ciclo entre os componentes do ciclo e encontra a atribuição requisição-servidor que sairá da solução, resultando no envio de $2.n$ mensagens. Ao terminar esta propagação, uma propagação para atualização dos fluxos das atribuições pertencentes ao ciclo é iniciada, resultando no envio de mais $2.n$ mensagens. Entretanto, duas ou mais propagações de ciclo podem se encontrar. Quando isto acontece, as propagações relacionadas aos ciclos com maiores custos reduzidos (em caso de empate, os com maiores id's) são canceladas. Para cancelar um ciclo, uma propagação de cancelamento de ciclo é iniciada, o que resulta, no pior caso, no envio de $2.n$ mensagens. Os ciclos não cancelados, chamados de ciclos terminados, iniciam a reconstrução da árvore de solução T com a atualização dos pais dos servidores ($paisT$) e requisições ($paireqT$) pertencentes a subárvore t de solução enraizada pela a atribuição

requisição-servidor ($aT(sT, reqT)$) que entrou na solução até a atribuição que saiu da solução. A atualização dos pais resulta, no pior caso, $2.n$ mensagens enviadas. Como no máximo n ciclos são descobertos (um por servidor), no pior caso, $O(n^2)$ mensagens são enviadas para propagação da informação de ciclo, $O(n^2)$ mensagens para atualização de fluxos de ciclo, $O(n^2)$ mensagens para cancelamento de ciclo e $O(k.n)$ mensagens para atualização de pais, onde k é o número de ciclos terminados e $k < n$.

Quando a atualização dos pais termina, a reconstrução dos nós que foram atualizados na árvore T inicia, ou seja, o cálculo das variáveis duais dos servidores sT e requisições $reqT$ que tiveram seus pais atualizados na árvore T é iniciado. O nó (servidor $sT(i)$ ou requisição $reqT(ri)$) que originou a atualização dos pais também inicia uma propagação de aviso aos nós servidores sT que estão acima dele na árvore T para notificar que os valores duais destes nós servidores não foram modificados e que estes podem propagar o valor de suas variáveis duais para os demais servidores (i. e. esses nós não precisaram ser reconstruídos). Esta propagação de aviso e o cálculo das variáveis duais modificadas resultam, no pior caso, no envio de $2.n$ e $n.m$ mensagens, respectivamente. Cada servidor $sT(i)$ ao receber o valor de todas as variáveis duais u dos outros servidores referente a cada ciclo válido, inicia o processo de descoberta de ciclo, continuando o método Simplex de Transporte até que nenhum servidor encontre mais um ciclo.

A complexidade de mensagens do Simplex de Transporte distribuído, DistST, é o número de mensagens enviadas em cada passada do Simplex de Transporte para a construção da árvore solução T, a propagação do valor da variável u, a propagação de informação de ciclo, a atualização dos fluxos, o cancelamento de ciclo, a atualização dos pais e propagação de notificação de variável dual não modificada. Para cada iteração do Simplex de Transporte, a complexidade se resume em:

$$O(n.m) + O(n^2) + O(n^2) + O(n^2) + O(n^2) + O(k.n) + O(k.n) \tag{5.1}$$

Logo, a complexidade de mensagens é $O(p.n.m)$, onde p é número de iterações do Simplex de Transporte.

A construção da árvore de solução T para o cálculo das variáveis duais resulta em uma árvore com altura de comprimento $2.n$, no pior caso. Pois como a construção atinge todos os nós servidores e todos os nós requisições, o caminho desde o nó servidor $sT(i)$ que iniciou a construção até a requisição folha $reqT(ri)$ de maior profundidade pode percorrer todos os servidores sT e para percorrer cada servidor $sT(i)$, necessariamente, uma requisição $reqT(ri)$ também é percorrida e essas requisições são diferentes. Assim,

a maior cadeia de mensagens até o término da construção da árvore T, no pior caso, terá comprimento $2.n$.

No entanto, quando um servidor $sT(i)$ é atingido durante a construção da árvore T, o valor da sua variável u é propagado para os demais servidores sT. Quando um servidor $sT(i)$ recebe este valor propagado de todos os outros servidores, um ciclo pode ser encontrado e uma propagação de informação de ciclo pode ser iniciada. Suponha que um ciclo é encontrado e que este seja do servidor raiz sR à requisição folha $reqT(ri)$ de maior profundidade, percorrendo todos os servidores sT, ou seja, o ciclo tem comprimento $2.n$. Assim, a propagação de informação de ciclo resulta em uma cadeia de mensagens de comprimento $2.n$. Quando esta propagação termina, a atualização dos fluxos das atribuições aT pertencentes ao ciclo inicia-se, resultando novamente em uma cadeia de $2.n$ mensagens. Se a propagação de informação de ciclo ou a atualização dos fluxos encontrarem com outro ciclo de custo reduzido menor, o cancelamento de um ciclo é feito e resulta, no pior caso, em uma cadeia de $2.n$ mensagens. O ciclo que não foi cancelado inicia a atualização dos pais dos servidores sT e das requisições $reqT$ que terão suas variáveis duais modificadas, resultando, no pior caso, uma cadeia de mensagens de comprimento $2.n$. Ao terminar a atualização dos pais, a propagação de notificação das variáveis duais não modificadas (cadeia de mensagens de comprimento $2.n$, no pior caso) e a reconstrução da árvore T são iniciadas.

A complexidade de tempo global do DistST relativa a uma passada do Simplex é a soma do comprimento da cadeia de mensagens gerada a construção da árvore solução T $(2.n)$, a propagação do valor da variável u (1), a propagação de informação de ciclo $(2.n)$, a atualização dos fluxos $(2.n)$, a atualização dos pais $(2.n)$ e propagação de notificação de variável dual não modificada$(2.n)$. Isto resume-se em:

$$2.n + 1 + 2.n + 2.n + 2.n + 2.n = 11.n + 1 \approx O(n) \tag{5.2}$$

Logo, a complexidade de tempo global é $O(p.n)$.

Durante a construção da árvore de solução T, o tempo gasto de computação entre o recebimento e o envio de uma mensagem é o tempo para armazenar o valor da variável dual $(O(1))$ e selecionar um servidor $sT(i)$ $(O(n))$ ou uma requisição $reqT(ri)$ $(O(r_{atendidas_i}))$, onde $r_{atendidas_i}$ é o número de requisições atendidas pelo servidor i) para continuar a construção.

Cada servidor $sT(i)$ ao receber os valores das variáveis duais de todos os outros servi-

dores realiza o procedimento de encontrar ciclo. Para isto, é necessário percorrer o caminho das variáveis duais do servidor $sT(i)$ e da requisição $reqT(ri)$ responsáveis pelo ciclo (da raiz da árvore sR até o servidor $sT(i)$ e a requisição $reqT(ri)$). Como este caminho tem, no pior caso, comprimento $2.n$, o tempo para realizar este procedimento é $O(n)$.

Na propagação de informação de ciclo, o tempo gasto de computação entre o recebimento e o envio de uma mensagem é o tempo para armazenar o caminho do ciclo ($O(n)$).

Durante a atualização dos fluxos, a atualização dos pais e propagação de notificação de variável dual não modificada, o tempo gasto de computação entre o recebimento e o envio de uma mensagem é o tempo de armazenar informações necessárias e continuar a propagação ($O(1)$).

A complexidade de tempo local do DistST se resume em:

$$O(r_{atendidas_i}) + O(n) + O(n) + O(1) \approx O(r_{atendidas_i}) \tag{5.3}$$

Logo, a complexidade de tempo local é $O(r_{atendidas_i}) > O(n)$.

Para realizar as computações locais durante o DistST é necessário que cada processo servidor i armazene as seguintes informações: as requisições dos clientes mais próximos (r_i), as requisições atendidas($r_{atendidas_i}$), o caminho do ciclo de cada requisição dos clientes mais próximos ($r_i.n$) e a quantidade atendida de cada servidor para cada conteúdo ($n.c$), onde c é o número de conteúdos disponíveis localmente). Assim, a demanda de memória de cada servidor para Simplex de Transporte distribuído é $r_i + r_{atendidas_i} + r_i.n + n.c$.

O algoritmo DistST tem, no pior caso, complexidades de $O(p.n.m)$ mensagens, de $O(p.n)$ tempo global, de $O(r_{atendidas_i})$ tempo local e demanda de memória $= r_i + r_{atendidas_i} + r_i.n + n.c$.

Uma alternativa para resolver o Simplex de Transporte seria a eleição de um nó líder para resolver localmente o problema e depois propagar a informação. A propagação de informação neste caso possui complexidade de mensagens de $O(n^2)$ (com tamanho correspondente as listas de requisições) e $O(n)$ de tempo global. Além disso, a complexidade de tempo do algoritmo sequencial do Simplex de Transporte é $O(p.n.m)$. Considerando essas complexidades e a demanda de memória do nó líder, que é igual a demanda de memória dos n servidores, a abordagem distribuída proposta é uma alternativa atraente para o problema.

Capítulo 6

Resultados Computacionais

Para validar o algoritmo proposto, os testes foram realizados no *cluster* OSCAR [15] que possui 42 máquinas com 2 processadores Intel Xeon QuadCore 2.66GHz, sendo que 40 máquinas de processamento possuem 16GBytes de memória, e 2 máquinas de login e manutenção possuem 8GBytes. Todas as máquinas são interligadas com *switch* Gigabit Ethernet. O sistema operacional do *cluster* é o *Red Hat Enterprise Linux Server* versão 5.3. O algoritmo foi implementado em ANSI C e MPICH2 para simulação distribuída com troca de mensagens. Como o método do Custo Mínimo apresenta melhores soluções, em média, do que o método do Canto Noroeste, somente os resultados do algoritmo DistPAC baseado no método do Custo Mínimo são apresentados.

As instâncias utilizadas nos testes foram obtidas de [13] e estão disponíveis em [11]. As instâncias são divididas em três classes: fáceis de resolver, intermediárias e difíceis. Para cada classe, há 20 instâncias, 5 para cada número de servidores, 10, 20, 30 e 50. Os números de requisições totais das instâncias com 10, 20, 30 e 50 servidores são, em média, 550, 1200, 2050 e 3300, respectivamente. Além disso, a largura de banda dos conteúdos considerada foi 15% do tamanho do conteúdo associado a requisição. As instâncias consideram que as requisições dos clientes estão nos servidores mais próximos a ele. Assim, cada servidor possui uma lista de requisições.

Como descrito no Capítulo 5, as posições das réplicas são dados de entrada para o algoritmo. Estes dados não estão disponíveis nessas instâncias. Assim, esses dados foram gerados através do posicionamento ótimo obtido por formulação matemática com o uso do CPLEX [10] versão 11.2. Cada instância foi executada 10 vezes e calculada a média das execuções para as soluções encontradas e a média do tempo de CPU das execuções. Para a execução de cada instância foi utilizado a quantidade de processos igual ao número

de servidores da instância.

Os resultados para obtenção da solução inicial estão apresentados na Tabela 6.1. Esta tabela possui uma coluna com a identificação das instâncias, $X\_\ Y$, onde X é a quantidade de servidores e Y é a identificação da classe da instância. Os valores Y de 1 a 5 identificam as instâncias da classe fácil de resolver, os de 6 a 10, da classe intermediária e, por fim, os de 11 a 15, da classe difícil. A tabela também possui outras informações como o número de requisições, a solução ótima, as soluções iniciais obtidas pelo método do Custo Mínimo sequencial e pelo DistPAC. Na Tabela 6.2, estão expostos os resultados do Simplex de Transporte, em que são apresentados a primeira solução viável o tempo de CPU gasto pela versão sequencial do Simplex e pelo DistST, o número de passos do executados pelo Simplex sequencial e pelo DistST. Uma outra tabela também é utilizada para expor os resultados, a Tabela 6.3. Nesta são apresentados o tempo global e o número de mensagem trocadas na simulação das instâncias.

A descrição dos resultados está dividida em duas seções. A primeira relata os resultados encontrados durante a simulação do DistPAC, ou seja, a solução inicial obtida por ele. E a segunda parte descreve os resultados obtidos durante a execução do DistST, ou seja, a etapa do Simplex de Transporte distribuído.

6.1 DistPAC

As soluções iniciais obtidas pelo método do Custo Mínimo sequencial e pela heurística DistPAC baseada neste método são apresentadas na Tabela 6.1.

Nas instâncias consideradas fáceis, observa-se o valor de solução igual a zero. Isto ocorre devido à existência de réplicas de todos os conteúdos em todos os servidores, o que permite que para estas instâncias, independendo do número de servidores, os algoritmos do método de Custo Mínimo sequencial e do DistPAC encontrem sempre o valor ótimo.

A partir das instâncias intermediárias, é possível observar valores altos para as soluções iniciais, tanto para o método sequencial, quanto para a heurística distribuída. Isto ocorre devido aos custos infinitos atribuídos aos enlaces de requisições por um conteúdo a um servidor que não possui este conteúdo, pois estes algoritmos não consideram a disponibilidade de conteúdos nos servidores. A distinção entre os servidores que possuem um conteúdo e os que não possuem é feita utilizando custos infinitos para servidores que não possuem um determinado conteúdo, conforme descrito na Seção 2. Com isso, nada impede esses algoritmos de utilizarem estas atribuições com custo infinito para encontrar um

Tabela 6.1: Solução Inicial: Sequencial x Distribuído

Instâncias	Nº de Requisições	Ótimo	Solução Inicial Sequencial	Solução Inicial Distribuída	Desvio Padrão (%)
10_1	613	0.0	0.0	0.0	0.0
10_2	734	0.0	0.0	0.0	0.0
10_3	736	0.0	0.0	0.0	0.0
10_4	798	0.0	0.0	0.0	0.0
10_5	634	0.0	0.0	0.0	0.0
10_6	627	471763.2	471763.2	471763.2	0.0
10_7	675	306226.7	12055280.7	306226.7	0.0
10_8	641	378230.8	380742.9	379416.4	0.0
10_9	659	461161.2	360134314.9	66494292.1	57.5
10_10	649	501081.9	516505.8	516538.4	1.1
10_11	627	471763.2	471763.2	471763.2	0.0
10_12	675	316065.0	104204534.4	93844995.3	7.6
10_13	641	378230.8	380742.9	379416.4	0.0
10_14	659	461161.2	360134314.9	104635498.3	54.3
10_15	649	501081.9	516505.8	519472.7	0.8
20_1	1915	0.0	0.0	0.0	0.0
20_2	1830	0.0	0.0	0.0	0.0
20_3	2100	0.0	0.0	0.0	0.0
20_4	1660	0.0	0.0	0.0	0.0
20_5	1695	0.0	0.0	0.0	0.0
20_6	1289	1434569.8	298622393.5	69422807.0	106.5
20_7	1356	916305.8	974232.0	950868.2	0.6
20_8	1314	1158373.3	252787669.9	186742182.3	34.6
20_9	1352	1160990.9	814687624.4	542353658.5	18.8
20_10	1367	853256.2	864901.9	864244.9	0.1
20_11	1289	1434569.8	298622393.5	90560889.4	80.4
20_12	1356	916305.8	974232.0	951266.7	0.7
20_13	1314	1158373.3	252787669.9	181488041.3	19.7
20_14	1352	1160990.9	814687624.4	481505793.2	22.6
20_15	1367	839075.6	845560.6	845502.6	0.2
30_1	2765	0.0	0.0	0.0	0.0
30_2	2296	0.0	0.0	0.0	0.0
30_3	2450	3192.4	3192.4	3192.4	0.0
30_4	2422	0.0	0.0	0.0	0.0
30_5	2961	0.0	0.0	0.0	0.0
30_6	2007	1443886.8	1452805.4	1471336.8	0.3
30_7	1963	1280967.4	1299447.3	1311361.4	0.2
30_8	2021	1132308.9	1187075.7	1168246.4	1.6
30_9	1991	1169034.7	1219235.1	1226494.9	0.4
30_10	1998	1150970.7	111066243.5	87874821.9	50.7
30_11	2007	1458770.6	1467689.2	1487350.1	0.2
30_12	1963	1280967.4	1299447.3	1308424.3	0.4
30_13	2021	1132308.9	1187075.7	1168002.0	1.5
30_14	1991	1182624.5	1232757.0	1241409.1	0.3
30_15	1998	1150970.7	111066243.5	68044773.5	113.0
50_1	4536	0.0	0.0	0.0	0.0
50_2	4620	23162.4	23162.4	23162.4	0.0
50_3	4800	34877.9	34877.9	34877.9	0.0
50_4	3646	0.0	0.0	0.0	0.0
50_5	4884	49511.3	49511.3	49511.3	0.0
50_6	3391	2223511.6	232991437.3	2326645.0	0.9
50_7	3329	1655211.8	1674253.3	1670466.6	0.2
50_8	3214	3065126.1	81444219.6	100397234.8	68.0
50_9	3303	3029900.9	3111283.4	3117664.5	0.4
50_10	3295	2196471.3	2250426.7	2227283.9	0.5
50_11	3391	2223511.6	232991437.3	2333774.6	0.7
50_12	3329	1655211.8	1674253.3	1668489.5	0.2
50_13	3314	3065126.1	81444219.6	162122967.7	35.3
50_14	3303	3011459.5	163208074.1	14219737.9	126.4
50_15	3295	2196403.0	2223242.6	2217503.0	0.2
Média	2018	868251.1	76845373.0	38040322.8	13.5

solução inicial. No entanto, estas soluções obtidas são consideradas viáveis teoricamente, mas na prática não são. Assim, é possível observar desvios padrão com porcentagens acima de 100, devido a presença de inviabilidades práticas na solução, o que pode gerar soluções muito altas. Por isso, é necessário o uso do Simplex de Transporte para obter uma solução viável na prática para o problema.

Apesar desses valores altos serem observados para as duas abordagens, sequencial e distribuída, as soluções iniciais obtidas pela heurística DistPAC são, em média, 50.5% melhores do que o método sequencial. Pois, o envio de mensagens para alocação de requisições no DistPAC é não-determinístico. Isto pode ocasionar na obtenção de soluções melhores devido as ordens de alocações poderem ser diferentes, fato que não acontece na abordagem sequencial, onde a ordem de alocação é sempre a mesma.

6.2 DistST

Na Tabela 6.2, o tempo de CPU da execução do Simplex sequencial e do Simplex distribuído estão apresentados na quinta (Tempo) e na décima primeira coluna (Tempo), respectivamente. Este tempo corresponde somente o tempo de CPU da etapa do Simplex de Transporte, não é considerado o tempo gasto para obter uma solução inicial para o problema. Na sexta (Nº. de Passos) e na última coluna (Nº. de Passos), o número de passos realizados pelo Simplex sequencial e pelo DistST também são apresentados, respectivamente. Como soluções não viáveis são encontradas inicialmente, a primeira solução viável, o tempo gasto e o número de passos necessários para obter esta solução, também, são apresentados na tabela. Estas informações estão, respectivamente, na segunda (Solução), terceira (Tempo) e quarta (N.º de Passos) coluna para a versão sequencial e, respectivamente, na sétima (Solução), nona (Tempo) e décima (N.º de Passos) para a versão distribuída

Nesses experimentos, foi observado que em alguns passos do DistST, o algoritmo não encontra ciclos distintos entre si que possam ser processados paralelamente. Assim, a quantidade máxima observada de ciclos processados em paralelos em um passo do Simplex foi 3. Contudo, em média, o algoritmo, quando encontra ciclos em paralelos, encontra 2. Isto ocorre, pois os ciclos encontrados com custo reduzido menor são compostos de muitos servidores e requisições, o que dificulta a obtenção de ciclos com componentes distintos na árvore de solução.

Para as instâncias fáceis, observa-se um tempo similar nas duas abordagens, sequencial

Tabela 6.2: Simplex de Transporte: Sequencial x Distribuído

Instâncias	Sequencial					Distribuído					
	Solução Viável			Solução Final		Solução Viável				Solução Final	
	Solução	Tempo (s)	N.º de Passos	Tempo (s)	Nº de Passos	Solução	Desvio Padrão (%)	Tempo (s)	Nº de Passos	Tempo (s)	Nº de Passos
10_1	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
10_2	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
10_3	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
10_4	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
10_5	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
10_6	471763.2	0.00	1	0.00	1	471763.2	0.0	0.00	1	0.00	1
10_7	306302.9	0.00	2	0.00	3	306226.7	0.0	0.00	1	0.00	1
10_8	380742.9	0.00	1	0.00	5	379416.4	0.0	0.00	1	0.00	2
10_9	469533.0	0.00	19	0.01	24	473679.5	2.2	0.00	3	0.08	21
10_10	516505.8	0.00	1	0.00	11	516538.4	1.1	0.00	1	0.06	18
10_11	471763.2	0.00	1	0.00	1	471763.2	0.0	0.00	1	0.00	1
10_12	316443.0	0.00	6	0.00	7	316065.0	0.0	0.00	3	0.01	4
10_13	380742.9	0.00	1	0.00	5	379416.4	0.0	0.00	1	0.00	2
10_14	469533.0	0.00	19	0.01	24	475190.2	2.4	0.00	5	0.09	23
10_15	516505.8	0.00	1	0.00	11	519472.7	0.8	0.00	1	0.07	22
20_1	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
20_2	0.0	0.00	1	0.00	1	0.0	0.0	0.10	1	0.10	1
20_3	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
20_4	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
20_5	0.0	0.00	1	0.00	1	0.0	0.0	0.00	1	0.00	1
20_6	1465555.5	0.03	18	0.09	53	1480731.9	0.6	0.00	6	0.32	47
20_7	974232.0	0.00	1	0.15	87	950868.2	0.6	0.00	1	0.55	78
20_8	1165684.8	0.02	13	0.05	27	1168902.4	0.2	0.01	13	0.22	33
20_9	1162230.5	0.10	58	0.13	75	1173135.2	0.7	0.29	44	0.55	87
20_10	864901.9	0.00	1	0.03	18	864244.9	0.1	0.00	1	0.12	22
20_11	1465555.5	0.03	18	0.09	53	1477462.1	0.4	0.00	7	0.31	46
20_12	974232.0	0.00	1	0.16	87	951266.7	0.7	0.00	1	0.54	80
20_13	1165684.8	0.02	13	0.05	27	1179776.5	2.8	0.14	12	0.24	36
20_14	1162230.5	0.10	58	0.13	75	1173851.7	0.7	0.30	44	0.52	82
20_15	845560.6	0.01	1	0.03	13	845502.6	0.2	0.00	1	0.07	13
30_1	0.0	0.01	1	0.01	1	0.0	0.0	0.01	1	0.01	1
30_2	0.0	0.01	1	0.01	1	0.0	0.0	0.01	1	0.01	1
30_3	3192.4	0.01	1	0.01	1	3192.4	0.0	0.01	1	0.01	1
30_4	0.0	0.01	1	0.01	1	0.0	0.0	0.01	1	0.01	1
30_5	0.0	0.02	1	0.02	1	0.0	0.0	0.01	1	0.01	1
30_6	1452805.4	0.01	1	0.18	34	1471336.8	0.3	0.00	1	0.43	43
30_7	1299447.3	0.00	1	0.18	37	1311361.4	0.2	0.01	1	0.49	50
30_8	1187075.7	0.00	1	0.33	64	1168246.4	1.6	0.01	1	0.57	61
30_9	1219235.1	0.00	1	0.39	79	1226494.9	0.4	0.01	1	0.97	95
30_10	1178768.2	0.05	10	0.24	47	1176347.8	0.7	0.16	7	0.60	54
30_11	1467689.2	0.01	1	0.17	34	1487350.1	0.2	0.00	1	0.44	43
30_12	1299447.3	0.01	1	0.18	37	1308424.3	0.4	0.01	1	0.48	48
30_13	1187075.7	0.01	1	0.33	64	1168002.0	1.5	0.01	1	0.57	61
30_14	1232757.0	0.00	1	0.38	77	1241409.1	0.3	0.01	1	0.92	97
30_15	1178768.2	0.04	10	0.23	47	1176740.3	1.0	0.15	9	0.54	50
50_1	0.0	0.04	1	0.04	1	0.0	0.0	0.02	1	0.02	1
50_2	23162.4	0.05	1	0.05	1	23162.4	0.0	0.09	1	0.19	3
50_3	34877.9	0.04	1	0.04	1	34877.9	0.0	0.02	1	0.02	1
50_4	0.0	0.03	1	0.03	1	0.0	0.0	0.02	1	0.02	1
50_5	49511.3	0.05	1	0.05	1	49511.3	0.0	0.02	1	0.02	1
50_6	2351414.6	0.63	26	4.26	171	2326645.0	0.9	0.02	1	3.37	150
50_7	1674253.3	0.04	1	0.80	33	1670466.6	0.2	0.02	1	0.55	30
50_8	3149847.0	0.18	8	2.85	122	3161766.5	2.1	0.50	13	3.83	144
50_9	3111283.4	0.03	1	2.61	109	3117664.5	0.4	0.02	1	2.40	104
50_10	2250426.7	0.04	1	1.45	61	2227283.9	0.5	0.02	1	1.07	47
50_11	2351414.6	0.64	26	4.28	171	2333774.6	0.7	0.02	1	3.39	150
50_12	1674253.3	0.03	1	0.80	33	1668489.5	0.2	0.02	1	0.50	28
50_13	3149847.0	0.17	8	2.85	122	3311723.0	8.7	0.69	14	3.78	155
50_14	3078388.5	0.27	12	2.99	125	3090349.7	0.3	0.02	2	3.17	110
50_15	2223242.6	0.03	1	0.92	39	2217503.0	0.2	0.02	1	0.85	41
Média	889564.8	0.05	6	0.46	36	892456.6	0.6	0.05	2	0.55	37

e distribuída, visto que a solução inicial obtida nessas instâncias é ótima, sendo assim, o Simplex somente é utilizado para verificar a otimalidade da solução.

Para as demais classes de instâncias, alguns casos são ressaltados e descritos em particular. O primeiro caso pode ser exemplificado pela instância *50_14*. Neste caso, o tempo do Simplex sequencial é um pouco menor do que o algoritmo DistST, mesmo este realizando um número menor de passos. O número de passos menor é justificado pelo fato da solução inicial distribuída ser melhor do que a sequencial. Já o tempo maior gasto pela abordagem distribuída é porque o DistST encontra ciclos paralelos em uma quantidade pequena de passos. Essa quantidade não é suficiente para melhorar o tempo da abordagem distribuída, em relação à sequencial, por causa do tempo gasto com as trocas de mensagens do DistST.

Um segundo caso pode ser exemplificado pela instância *50_12*. Neste caso, o tempo de CPU gasto pelo DistST é menor do que o Simplex sequencial e o número de passos também. Como as soluções iniciais são similares, onde a solução inicial do DistPAC é apenas 0.3% melhor do que a solução inicial do método sequencial, pode-se concluir que a quantidade de passos em que o DistST obteve ciclos paralelos foi suficiente para diminuir o tempo de execução do Simplex de Transporte, considerando o tempo das trocas de mensagens. No entanto, existe também o caso observado na instância *50_11*, em que o tempo e o número de passos da abordagem distribuída são ainda muito menores do que os da abordagem sequencial. Esta diferença maior se deve a solução inicial muito melhor alcançada pelo DistPAC. Um outro caso, observado na instância *30_14*, é que tanto o tempo quanto o número de passos são maiores no DistST. Isto é justificado pela solução inicial pior do DistPAC. No entanto, a solução inicial da instância *30_15* obtida pelo DistPAC é melhor do que a solução inicial do método sequencial, e o tempo gasto e o número de passos executado pelo DistST são maiores. Isto ocorre porque a solução inicial melhor obtida por DisPAC pode convergir mais lentamente para a solução ótima do que uma solução inicial pior.

Ainda na Tabela 6.2, pode-se observar que a maioria das instâncias encontra uma solução viável realizando poucos passos do Simplex de Transporte, sendo esta de boa qualidade e ficando, em média, 1.6% do ótimo, tanto para a versão sequencial quanto para a distribuída.

Na Tabela 6.3, são apresentadas o número de mensagens e o tempo global (conforme descrito na Seção 5.3.2) da heurística DistPAC em conjunto com o algoritmo DistST. Apesar do número de mensagens ser alto, este número é menor do que a complexidade

teórica de mensagens ($O(p.n.m)$). E o tempo global encontrado é o mesmo esperado pela complexidade de tempo global teórica ($O(p.n)$).

Portanto, os resultados mostraram que o algoritmo DistST obteve a solução ótima em tempos compatíveis, na média, com a versão sequencial, mesmo com algumas instâncias não alcançando um grau de paralelismo desejado. Mas devido a natureza do problema ser distribuída, onde os dados estão dispersos em uma rede, o algoritmo se mostra uma alternativa atraente para resolver o problema de forma satisfatória. Além disso, o algoritmo pode ser utilizado não só para obter a melhor solução, a ótima, mas como também para encontrar uma solução de boa qualidade realizando apenas alguns passos do Simplex.

Tabela 6.3: Número de Mensagens e Tempo Global: DistPAC e DistST

Instâncias	Nº de Mensagens	Tempo Global	Instâncias	Nº de Mensagens	Tempo Global
10_1	511	31	20_1	1821	25
10_2	511	34	20_2	1821	41
10_3	511	32	20_3	1821	36
10_4	511	35	20_4	1821	32
10_5	511	29	20_5	1821	41
10_6	953	33	20_6	88059	1946
10_7	809	31	20_7	129604	4663
10_8	1325	87	20_8	56851	1804
10_9	10696	745	20_9	144346	3541
10_10	9061	564	20_10	34759	806
10_11	953	32	20_11	86359	1914
10_12	2290	137	20_12	129986	4807
10_13	1326	87	20_13	60097	1878
10_14	11399	844	20_14	137740	3344
10_15	10645	665	20_15	21606	428
30_1	3931	50	50_1	10551	67
30_2	3931	53	50_2	26947	116
30_3	3993	40	50_3	11227	64
30_4	3931	52	50_4	10551	69
30_5	3931	53	50_5	11521	74
30_6	157277	2057	50_6	1349196	7450
30_7	173295	2665	50_7	250870	1061
30_8	197913	2768	50_8	1325242	8135
30_9	325215	4653	50_9	950912	4865
30_10	187902	3269	50_10	414222	1991
30_11	159776	2083	50_11	1350529	7461
30_12	165810	2502	50_12	236214	964
30_13	199496	2738	50_13	1414197	8449
30_14	327944	4603	50_14	1018436	5764
30_15	174515	2892	50_15	358956	1681

Capítulo 7

Conclusões e Trabalhos Futuros

Este trabalho aborda o Problema de Atribuir Clientes (PAC) a servidores em uma Rede de Distribuição de Conteúdos (RDC). O problema foi modelado como um Problema de Transporte, que é um problema muito estudado na área de fluxo de redes e pode ser resolvido de forma exata por programação linear através de um algoritmo especializado conhecido como Simplex de Transporte.

Dada a natureza distribuída do problema, em que as informações de localização de conteúdos e requisições estão distribuídas entre os servidores de uma RDC, e com isso, o custo da coleta, do armazenamento e da atualização destas informações em um nó central pode ser proibitivo, um algoritmo distribuído é proposto e executado em servidores que possuem um conhecimento limitado sobre a rede.

O algoritmo combina uma heurística distribuída baseada nos métodos sequenciais do Canto Noroeste e do Custo Mínimo para obtenção de uma solução inicial viável para o Simplex de Transporte, denominada DistPAC, e um algoritmo distribuído para resolver o Simplex de Transporte aplicado ao PAC, denominado DistST.

Por se tratar de um algoritmo distribuído para o Simplex de Transporte, sua aplicação não se limita ao PAC, podendo, também, ser utilizado para resolver, de maneira geral, qualquer problema modelado como um Problema de Transporte, cujos dados estão geograficamente distribuídos.

O algoritmo resolve o PAC com informações de requisições por conteúdos de um dado instante de tempo. Sendo assim, pode também ser utilizado para a avaliação da rede. As requisições são extraídas durante um intervalo de tempo, permitindo obter a estatística do número de requisições por um determinado conteúdo. A estatística pode, então, compor os dados de entrada do problema. Assim, é possível obter informações importantes da rede,

como a carga nos servidores e o custo total de transporte dos conteúdos, possibilitando melhorias na QoS da rede.

Para validar o algoritmo proposto, testes foram realizados em instâncias existentes na literatura para problemas em RDC. Assim, a heurística DistPAC apresentou resultados, em média, melhores do que os métodos sequenciais e o algoritmo DistST resolveu o PAC em tempos compatíveis com a versão sequencial.

Os resultados obtidos pela heurística DistPAC juntamente com um mecanismo de trocas de atribuições que melhora a solução obtida estão no artigo aceito para publicação no XLIII Simpósio Brasileiro de Pesquisa Operacional [7].

Como trabalho futuro, propõe-se um estudo mais aprofundando nos algoritmos distribuídos propostos para melhorar o grau de paralelização e a construção de instâncias com quantidades maiores de servidores e de requisições que explorem melhor a realidade das RDCs e que favoreçam os processamentos paralelos do algoritmo.

Referências

[1] Ahuja, R. K., Magnanti, T. L., Orlin, J. B. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Fevereiro 1993.

[2] Barbosa, V. C. *An Introduction to Distributed Algorithms*. The MIT Press, Cambridge, Massachusetts, Fevereiro 1996.

[3] Barr, R. S., Hickman, B. L. Parallel simplex for large pure network problems: Computational testing and sources of speedup. *Operations Research 42*, 1 (1994), 65–80.

[4] Bektas, T., Cordeau, J.-F., Erkut, E., Laporte, G. Exact algorithms for the joint object placement and request routing problem in content distribution networks. *Comput. Oper. Res. 35*, 12 (2008), 3860–3884.

[5] Bektas, T., Oguz, O., Ouveysi, I. Designing cost-effective content distribution networks. *Comput. Oper. Res. 34*, 8 (2007), 2436–2449.

[6] Chang, M. D., Engquist, M., Finkel, R., Meyer, R. R. Parallel algorithm for generalized networks. *Ann. Oper. Res. 14* (Junho 1988), 125–145.

[7] Coutinho, R. C., Drummond, L. M. A., Frota, Y., Simonetti, L. Uma heurística distribuída para o problema de atribuição de clientes a servidores em redes de distribuição de conteúdos. Em *XLIII Brazilian Symposium of Operational Research - A ser publicado* (2011).

[8] Dantzig, G. Application of the simplex method to a transportation problem. Em *Activity analysis of production and allocation*, T. Koopmans, Ed. J. Wiley, New York, 1951, p. 359–373.

[9] Hall, J. Towards a practical parallelisation of the simplex method. *Computational Management Science 7* (2010), 139–170.

[10] ILOG, S. A. Cplex 11 user's manual.

[11] LABIC. World wide web. Em *http://labic.ic.uff.br/* (2005).

[12] Mundur, P., Arankalle, P. Optimal server allocations for streaming multimedia applications on the internet. *Computer Networks 50*, 18 (2006), 3608 – 3621.

[13] Neves, T. A., de A. Drummond, L. M., Ochi, L. S., Albuquerque, C., Uchoa, E. Solving replica placement and request distribution in content distribution networks. *Electronic Notes in Discrete Mathematics 36* (2010), 89–96.

[14] Nygren, E., Sitaraman, R. K., Sun, J. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev. 44* (2010), 2–19.

[15] OSCAR. Computação de alto desempenho no cluster oscar. Em *http://suporte.ic.uff.br/index.php/servicos/posgrad/oscarusar* (2010).

[16] Shah, S., Ramamritham, K., Ravishankar, C. Client assignment in content dissemination networks for dynamic data. Em *Proceedings of the 31st international conference on Very large data bases* (2005), p. 673–684.

[17] Thulasiraman, K., Chalasani, R., Comeau, M. Parallel network dual simplex method on a shared memory multiprocessor. Em *Proceedings of the Fifth IEEE Symposium on* (Dezembro 1993), p. 408 –415.

[18] Zhou, X., Xu, C.-Z. Efficient algorithms of video replication and placement on a cluster of streaming servers. *J. Netw. Comput. Appl. 30* (Abril 2007), 515–540.

APÊNDICE A – Pseudocódigos

Neste apêndice, alguns pseudocódigos referenciados no Capítulo 5 são apresentados para o leitor que busca mais detalhes sobre os procedimentos. Eles estão divididos de acordo com cada etapa do passo do Simplex de Transporte.

A.1 Construção da árvore de solução dual

Os procedimentos desta seção representam o mecanismo de escolha de um servidor para continuar a construção da árvore de solução inicial quando acontece o caso de degeneração na solução.

Algoritmo 33: Ao receber uma mensagem $ATENDCOMPREQ(r, s, atingidos)$ do servidor i

```
1 procAtendeuCompletoReq(r, s, atingidos);
```

Algoritmo 34: $procAtendeuCompletoReq(r, s, atingidos)$

```
1  servAtingidos := servAtingidos ∪ atingidos;
2  se s < menorEscolhido então
3      menorEscolhido := s;
4      reqEscolhido := r;
5  se recebeu ATENDCOMPREQ de todos os servidores que meuId atendeu então
6      se servRaiz = 1 então
7          servRaiz := 0;
8          se serAtingidos não possui todos os servidores então
9              se menorEscolhido! = meuId então
10                 envia CONTINUA(reqEscolhido, servAtingidos) para o servidor
                   menorEscolhido;
11             senão
12                 procContinua(reqEscolhido, servAtingidos);
13     senão
14         se paiServDU! = meuId então
15             envia ATENDCOMPLETO(reqEscolhido, menorEscolhido, servAtingidos) para
               o servidor paiServDU;
16         senão
17             procAtendeuCompleto(reqEscolhido, menorEscolhido, servAtingidos);
```

Algoritmo 35: Ao receber uma mensagem $ATENDCOMPLETO(r, s, atingidos)$ do servidor i

```
1 procAtendeuCompleto(r, s, atingidos);
```

Algoritmo 36: $procAtendeuCompleto(r, s, atingidos)$

```
1 servAtingidos := servAtingidos ∪ atingidos; se s < menorEscolhido então
2     menorEscolhido := s;
3     reqEscolhido := r;
4 se recebeu ATENDCOMPLETO de todos os servidores que atenderam r então
5     se servPaiDV_r ! = meuId então
6         envia ATENDCOMPLETO(reqEscolhido, menorEscolhido, servAtingidos) para o
          servidor servPaiDV_r;
7     senão
8         procAtendeuCompleto(reqEscolhido, menorEscolhido, servAtingidos);
```

Algoritmo 37: Ao receber uma mensagem $CONTINUA(r, atingidos)$ do servidor i

```
1 procContinua(r, atingidos);
```

Algoritmo 38: $procContinua(r, atingidos)$

```
1 servAtingidos := servAtingidos ∪ atingidos; Selecione um servidor s que não pertença a
  serAtingidos;
2 participa_rs := 1;
3 se s! = meuId então
4     envia DEGENERACAO(custo_s − dualV_r, r, cont_r, caminho, caminhoTam) para o servidor
      s;
5 servRaiz := 1;
6 procVaru(s, custo_s − dualV_r, r, cont_r, caminhoReq^r, caminhoReqTam_r, 1);
```

Algoritmo 39: Ao receber uma mensagem $DEGENERACAO(variavel, r, caminho, caminhoTam)$ do servidor i

```
1 servRaiz := 1;
2 procVaru(i, variavel, r, caminho, caminhoTam, idCiclo);
```

Algoritmo 40: Ao receber uma mensagem $URECUSADO(variavel, r, idCiclo)$ do servidor i

```
1 participa_ri := 0;
2 Selecione um servidor s' não escolhido que atendimento_rs' > 0 ;
3 se s'! = meuId então
4     envia VARU(variavel, r, caminhoReq^r, caminhoReqTam_r, idCiclo) para o servidor s';
5 senão
6     procVaru(s', custo_s' − dualV_r, r, caminhoReq^r, caminhoReqTam_r, idCiclo);
```

A.2 Seleção de Ciclo

Os algoritmos desta seção consistem do procedimento que encontra um ciclo na árvore de solução dado dois caminhos, do mecanismo de descoberta de ciclo através da propagação da mensagem $PCICLO$ e da atualização dos fluxos de um ciclo.

Algoritmo 41: $encontraCiclo(r, s, caminho, caminhoTam)$

```
1  se caminhoServTam > 0 então
2      aux1 := 0; /*variável auxiliar*/
3      aux2 := 0; /*variável auxiliar*/
4      enquanto aux1 < caminhoTam e aux2 < caminhoServTam faça
5          se caminho_aux1 = caminhoServ_aux2 então
6              aux1 := aux1 + 1;
7              aux2 := aux2 + 1;
8          senão
9              exit;
10     cicloTam := (caminhoTam − aux1) − (caminhoServTam − aux2) + 3;
11     se caminhoServ_aux2 = {s, r} então
12         cicloTam := (caminhoTam − aux1) − (caminhoServTam − aux2) + 1;
13     senão
14         se caminho_aux1 = {meuId} então
15             cicloTam := (caminhoTam − aux1) − (caminhoServTam − aux2) + 1;
16             aux2 := aux2 + 1;
17     ciclo_0 := {meuId};
18     para k = 1 até caminhoServTam − 1 faça
19         ciclo_k := caminhoServ_caminhoServTam−k;
20     se ciclo_k−1 ! = {s, r} então
21         se aux1 = axu2 então
22             ciclo_k := caminhoServ_caminhoServTam−k;
23             aux1 := aux1 − 1;
24         para k = caminhoServTam − aux1 + 1 até cicloTam − 1 faça
25             ciclo_k := caminhoServ_aux2;
26             aux2 := aux2 + 1;
27         ciclo_cicloTam−1 := {s, r};
28 senão
29     cicloTam := caminhoTam + 1; para k = 0 até caminhoTam − 1 faça
30         ciclo_k := caminhoServ_caminhoServTam−k;
31     ciclo_cicloTam−1 := {s, r};
32 retorna ciclo;
```

Algoritmo 42: $procCiclo(cr, valor, i, pos, idCiclo, ciclo, cicloTam)$

```
1  req := funcReq(ciclo_pos);
2  se ((req! = −1) e (idCicloReq_req = −1)) ou ((req = −1) e (idCicloServ = −1)) então
3      se (req! = −1) e (idCicloReq_req = −1) então
4          idCicloReq := idCiclo;
5          cicloReq_req := ciclo;
6          cicloReqTam_req := cicloTam;
7          posCicloReq_req := pos;
8          crReq_req := cr;
9          se atendimento_ri < valor então
10             valor := atendimento_ri;
11     senão
12         se (req = −1) e (idCiclo = −1) então
13             idCicloServ := idCiclo;
14             cicloServ := ciclo;
15             cicloServ := cicloTam;
16             posCicloServ := pos;
17             crServ := cr;
18     se pos < cicloTam − 1 então
19         se funcServ(ciclo_pos+1! = meuId) então
20             envia PCICLO(cr, valor, pos + 1, idCiclo, ciclo, cicloTam) para o servidor de
               funcServ(ciclo_pos+1);
21         senão
22             procCiclo(cr, valor, meuId, pos + 1, idCiclo, ciclo, cicloTam);
23     senão
24         se req! = −1 então
25             pos := 0;
26             achouVarSainte := 0;/*variável auxiliar*/
27             varSainte := {};
28             se funcServ(ciclo_pos! = meuId) então
29                 envia
                   ATUALIZACICLO(cr, achouVarSainte, valor, pos, idCiclo, ciclo, cicloTam,
                   varSainte) para o servidor de funcServ(ciclo_pos);
30             senão
31                 atualizaCiclo(cr, achouVarSainte, valor, meuId, pos, idCiclo, ciclo, cicloTam,
                   varSainte);
32 senão
33     cancelamento(cr, req, pos, valor, i, pos, idCiclo, ciclo, cicloTam);
```

Algoritmo 43: $atualizaCiclo(achou, valor, i, pos, idCiclo, ciclo, cicloTam, variavelS)$

```
1  req := funcReq(ciclo_pos);
2  se ((req! = −1) e (idCicloReq_req = idCiclo)) ou ((req = −1) e (idCicloServ = idCiclo)) então
3      se req = −1 então
4          se pos = 0 então
5              conteudoA := funcCont(ciclo_cicloTam−1);
6          senão
7              conteudoA := funcCont(ciclo_pos−1);
8          se pos = cicloTam − 1 então
9              conteudoD := funcCont(ciclo_0);
10             prox := funcServ(ciclo_0);
11         senão
12             conteudoD := funcCont(pos + 1);
13             prox := funcServ(ciclo_pos+1);
14         se atendeu_iconteudoA > 0 então
15             atendeu_iconteudoA := atendeu_iconteudoA + valor;
16         senão
17             atendeu_iconteudoA := valor;
18         θ := valor;
19         atendeu_proxconteudoD := atendeu_proxconteudoD − valor;
20         se (achou = 0)e(atendeu_proxconteudoD = 0) então
21             atendeu_proxconteudoD := −1;
22     senão
23         se req! = 1 então
24             atendimento_reqi := atendimento_reqi − valor;
25             se pos = cicloTam − 1 então
26                 prox := funcServ(ciclo_0);
27             tetaReq_req := valor;
28             se (achou = 0)e atendimento_ri = 0 então
29                 participa_reqi := 0;
30                 achou := 1;
31                 variavelS := {req, meuId, i, paiReq_r, idCiclo, r, −1};
32                 atendimento_rprox := atendimento_prox + valor;
33     se pos < cicloTam − 1 então
34         se funcServ(ciclo_pos+1! = meuId) então
35             envia
               ATUALIZACICLO(achou, valor, pos + 1, idCiclo, ciclo, cicloTam, variavelS) para
               o servidor de funcServ(ciclo_pos+1;
36         senão
37             atualizaCiclo(achou, valor, meuId, pos + 1, idCiclo, ciclo, cicloTam, variavelS);
38     senão
39         envia CICLOTERMINADO(idCiclo) para todos os servidores s, onde s! = meuId;
40         servCicloIniciado := servCicloIniciado − {i};
41         prox := funcServ(ciclo_0);
42         participa_reqprox := 1;
43         varSainte := variavelS;
44 verificaPasso();
```

A.3 Cancelamento de Ciclo

Os pseudocódigos desta parte apresenta o mecanismo de cancelamento de um ciclo com o processo de cancelar os sucessores e os antecessores de um dado componente do ciclo.

Algoritmo 44: $cancelamento(cr, req, valor, i, pos, idCiclo, ciclo, cicloTam)$

```
1  se ((req! = −1) e (idCicloReq_req! = −1)) ou ((req = −1) e (idCicloServ! = −1)) então
2      se ((req! = −1) e (idCicloReq_req! = −1) e ((cr < crReq_req) ou ((cr = crReq_req) e
       (idCiclo < idCicloReq_req)))) ou ((req = −1) e (idCicloServ! = −1) e ((cr < crServ) ou
       ((cr = crServ)e (idCiclo < idCicloServ)))) então
3          se (req! = −1) e (idCicloReq_req! = −1) então
4              antPos := posCicloReq_req;
5              se antPos < cicloReqTam_req − 1 então
6                  proxPos := pos + 1;
7              senão
8                  proxPos := −1;
9              cicloAux := cicloReq_req;
10             cicloAuxTam := cicloReqTam_req;
11         senão
12             se (req = −1) e (idCicloServ! = −1) então
13                 antPos := posCicloServ;
14                 se antPos < cicloServTam − 1 então
15                     proxPos := pos + 1;
16                 senão
17                     proxPos := −1;
18             cicloAux := cicloServ;
19             cicloAuxTam := cicloServTam;
20         se varSainte! = {} e varSainte.ciclo = idCiclo então
21             varSainte := {};
22         envia CICLOCANCELADO(idCiclo) para todos os servidores s, onde s! = meuId;
23         servCicloIniciado := servCicloIniciado − {i};
24         se proxPos! = −1 então
25             cancelaFrente(proxPos, cicloAux, cicloAuxTam);
26         cancelaAtras(antPos, cicloAux, cicloAuxTam);
27         procCiclo(cr, valor, i, pos, idCiclo, ciclo, cicloTam);
28     senão
29         se ((req! = −1) e (idCicloReq_req! = −1) e ((cr > crReq_req) ou ((cr = crReq_req) e
           (idCiclo > idCicloReq_req)))) ou ((req = −1) e (idCicloServ! = −1) e ((cr > crServ) ou
           ((cr = crServ)e (idCiclo > idCicloServ)))) então
30             envia CICLOCANCELADO(idCiclo) para todos os servidores s, onde
               s! = meuId;
31             servCicloIniciado := servCicloIniciado − {i};
32             se varSainte! = {} e varSainte.ciclo = idCiclo então
33                 varSainte := {};
34             cancelaAtras(pos, ciclo, cicloTam);
35 verificaPasso();
```

Algoritmo 45: $cancelaFrente(pos, idCiclo, ciclo, cicloTam)$

```
1  se funcServ(ciclo_pos)! = meuId então
2      envia CANCELACICLOF(pos, idCiclo, ciclo, cicloTam) para funcServ(ciclo_pos);
3  senão
4      procCancelaCicloFrente(pos, idCiclo, ciclo, cicloTam)
```

Algoritmo 46: $procCancelaCicloFrente(pos, idCiclo, ciclo, cicloTam)$

```
1  entrei := falso;
2  req := funcReq(ciclo_pos);
3  se (req! = −1) e (idCicloReq_req = idCiclo) então
4      idCicloReq_req := −1;
5      cicloReq_req := {};
6      entrei := verdade;
7      se tetaReq_req! = −1 então
8          se pos = cicloTam − 1 então
9              prox := funcServ(ciclo_0);
10         senão
11             prox := funcServ(ciclo_pos+1);
12         valor := tetaReq_req;
13         tetaReq_req := −1;
14         participa_reqmeuId := 1;
15         atendimento_reqmeuId := atendimento_reqmeuId + valor;
16         atendimento_reqprox := atendimento_reqprox − valor;
17 senão
18     se (req = −1) e (idCicloServ = idCiclo) então
19         idCicloServ := −1;
20         cicloServ := {};
21         entrei := verdade;
22         se teta! = −1 então
23             valor := teta;
24             teta := −1;
25             se pos = 0 então
26                 conteudoA := funcCont(ciclo_cicloTam−1);
27             senão
28                 conteudoA := funcCont(ciclo_pos−1);
29             se pos = cicloTam − 1 então
30                 conteudoD := funcCont(ciclo_0);
31                 prox := funcServ(ciclo_0);
32             senão
33                 conteudoD := funcCont(pos + 1);
34                 prox := funcServ(ciclo_pos+1);
35             atendeu_meuIdconteudoA := atendeu_meuIdconteudoA − valor;
36             atendeu_proxconteudoD := atendeu_proxconteudoD + valor;
37 se pos < cicloTam − 1 e entrei = verdade então
38     cancelaFrente(pos + 1, idCiclo, ciclo, cicloTam)
```

Algoritmo 47: $cancelaAtras(pos, idCiclo, ciclo, cicloTam)$

```
1  req = funcReq(ciclo_pos);
2  se (req! = −1) e (idCicloReq_req = idCiclo) então
3      idCicloReq_req := −1;
4      cicloReq_req := {};
5      entrei := verdade;
6      se tetaReq_req! = −1 então
7          se pos = cicloTam − 1 então
8              prox := funcServ(ciclo_0);
9              participa_reqprox := 0;
10         senão
11             prox := funcServ(ciclo_pos+1);
12         se pos = 0 então
13             ant := funcServ(ciclo_cicloTam−1);
14         senão
15             ant := funcServ(ciclo_pos−1);
16         valor := tetaReq_req;
17         tetaReq_req := −1;
18         atendimento_reqprox := atendimento_reqprox − valor;
19         atendimento_reqant := atendimento_reqant + valor;
20 senão
21     se (req = −1) e (idCicloServ = idCiclo) então
22         idCicloServ := −1;
23         cicloServ := {};
24         entrei := verdade;
25         se teta! = −1 então
26             valor := teta;
27             teta := −1;
28             se pos = cicloTam − 1 então
29                 conteudoA := funcCont(ciclo_0);
30                 ant := funcServ(ciclo_0);
31             senão
32                 conteudoA := funcCont(pos + 1);
33                 ant := funcServ(ciclo_pos+1);
34             se pos = 0 então
35                 conteudoD := funcCont(ciclo_cicloTam−1);
36                 prox := funcServ(ciclo_cicloTam−1);
37             senão
38                 conteudoD := funcCont(ciclo_pos−1);
39                 prox := funcServ(ciclo_pos−1);
40             atendeu_antconteudoA := atendeu_antconteudoA + valor;
41             atendeu_proxconteudoD := atendeu_proxconteudoD − valor;
42 se (pos > 0) e (entrei = verdade) então
43     se funcServ(ciclo_pos−1)! = meuId então
44         envia CANCELACICLO(pos − 1, idCiclo, ciclo, cicloTam) para funcServ(ciclo_pos−1);
45     senão
46         cancelaAtras(pos − 1, idCiclo, ciclo, cicloTam);
```

A.4 Reconstrução da árvore de solução

Por fim, nesta última seção, os algoritmos da atualização dos pais dos componentes de um ciclo e da reconstrução da árvore de solução alterada por propagações de ciclos são apresentados.

Algoritmo 48: $procAcabouAtualiza(id, pos)$

```
1  se nAtualiza = −1 então
2      nAtualiza := 1;
3  senão
4      nAtualiza := nAtualiza + 1;
5  se id = meuId então
6      varSainte.pos := pos;
```

Algoritmo 49: $reconstroi(c, req, idCiclo, i)$

```
1  se req = −1 então
2      se (paiServ.serv! = meuId) e (paiServ.serv! = −1) então
3          envia ATUALIZAU(−1, paiServ.req, idCiclo) para paiServ.serv;
4      senão
5          procAtualizaU(−1, paiServ.req, idCiclo, meuId);
6      recAtualizaU_idCiclo := 1;
7      para cada servidor s faça
8          para cada conteúdo c faça
9              se atendeu_sc > 0 então
10                 se s! = meuId então
11                     envia
                       VARV(dualU, c, atendeu_sc, caminhoServ, caminhoServTam, idCiclo) para
                       o servidor s;
12                 senão
13                     procedimentoAtendimento(s, c, caminhoServ, caminhoServTam, idCiclo);
14     envia VARNBU(dualU, caminhoServ, caminhoServTam) para todos os servidores s, onde
       s! = meuId;
15     servVaru := servVaru + {meuId}; varu_meuId := dualU;
16 senão
17     se req! = −1 então
18         se paiReq_req! = meuId então
19             envia ATUALIZAU(−2, req, idCiclo) para paiReq_req;
20         senão
21             procAtualizaU(−2, req, idCiclo, meuId);
22         para cada servidor s! = i que se passo = 1, atendimento_reqs > 0 ou se passo > 1,
           participa_reqs = 1 faça
23             recAtualizaUReq_reqidCiclo := 1;
24             se s! = meuId então
25                 envia VARU(custo_s − dualV_r, r, idCiclo) para o servidor s;
26             senão
27                 procVaru(i, variavel, r, caminho, caminhoTam, idCiclo)
28 verificaRecVarnbu(valor, caminho, caminhoTam);
```

Algoritmo 50: $atualizaPai(pos, idCiclo, ciclo, cicloTam, sentido)$

```
1  req = funcReq(ciclo_pos);
2  se (pos > 0) e (pos < cicloTam − 1) então
3      se req! = −1 então
4          se sentido = 2 então
5              servPaiDV_req := funcServ(ciclo_pos−1);
6              se pos > 0 então
7                  se servPaiDV_req! = meuId então
8                      envia ATUALIZAPAI(pos − 1, idCiclo, ciclo, cicloTam, sentido) para
                       servPaiDV_req;
9                  senão
10                     atualizaPai(pos − 1, idCiclo, ciclo, cicloTam, sentido);
11         senão
12             se (sentido = 1) então
13                 servPaiDV_req := funcServ(ciclo_pos+1);
14                 se pos < cicloTam − 1 então
15                     se servPaiDV_req! = meuId então
16                         envia ATUALIZAPAI(pos + 1, idCiclo, ciclo, cicloTam, sentido) para
                           servPaiDV_req;
17                     senão
18                         atualizaPai(pos + 1, idCiclo, ciclo, cicloTam, sentido);
19     senão
20         se req = −1 então
21             se (sentido = 2) então
22                 servPaiDU := funcServ(ciclo_pos−1);
23                 reqPaiDU := funcReq(ciclo_pos−1);
24                 se pos > 0 então
25                     se servPaiDU! = meuId então
26                         envia ATUALIZAPAI(pos − 1, idCiclo, ciclo, cicloTam, sentido) para
                           servPaiDU;
27                     senão
28                         atualizaPai(pos − 1, idCiclo, ciclo, cicloTam, sentido);
29             senão
30                 se (sentido = 1) então
31                     servPaiDU := funcServ(ciclo_pos+1);
32                     reqPaiDU := funcReq(ciclo_pos+1);
33                     se pos > 0 então
34                         se servPaiDU! = meuId então
35                             envia ATUALIZAPAI(pos + 1, idCiclo, ciclo, cicloTam, sentido)
                               para servPaiDU;
36                         senão
37                             atualizaPai(pos + 1, idCiclo, ciclo, cicloTam, sentido);
38 senão
39     envia ACABOUATUALIZA(idCiclo, pos) para todos os servidores s, onde s! = meuId;
40     procAcabouAtualiza(idCiclo, pos);
41     se pos = cicloTam − 1 então
42         s := funcServ(ciclo_0);
43         servPaiDV_req := s;
44         participa_reqs := 1;
45     senão
46         se pos = 0 então
47             s := funcServ(ciclo_cicloTam−1);
48             servPaiDU := s;
49             reqPaiDU := funcReq(ciclo_cicloTam−1);
50     verificaAcabouAtualiza();
```

Algoritmo 51: *procAtualizaU*$(conteudo, req, idCiclo, i)$

```
1  se conteudo = −2 então
2      se recAtualizaU_idCiclo = 0 então
3          recAtualizaU_idCiclo := 1;
4          envia VARNBU(dualU, caminhoServ, caminhoServTam) para todos os servidores s,
           onde s! = meuId;
5          servVaru := servVaru + {meuId};
6          para cada servidor s faça
7              para cada conteúdo c faça
8                  se atendeu_sc > 0 então
9                      se s! = i então
10                         req = −5;
11                     se s! = meuId então
12                         envia ATUALIZAU(c, req, idCiclo) para s;
13                     senão
14                         procAtualizaU(c, req, idCiclo, meuId);
15 senão
16     se conteudo = −1 então
17         se recAtualizaUReq_reqidCiclo = 0 então
18             recAtualizaUReq_reqidCiclo := 1;
19             para cada servidor s faça
20                 se (s! = i) e (participa_reqs = 1) então
21                     se s! = meuId então
22                         envia ATUALIZAU(−2, req, idCiclo) para s;
23                     senão
24                         procAtualizaU(−2, req, idCiclo, meuId);
25     senão
26         se conteudo! = −2 então
27             para cada requisição req_r faça
28                 se (req_r! = req) e (cont_r = conteudo) e (participa_req_ri = 1) então
29                     se recAtualizaUReq_req_ridCiclo = 0 então
30                         recAtualizaUReq_req_ridCiclo = 0;
31                         para cada servidor s faça
32                             se (s! = i) e (participa_req_rs = 1) então
33                                 se s! = meuId então
34                                     envia ATUALIZAU(−2, req_r, idCiclo) para s;
35                                 senão
36                                     procAtualizaU(−2, req_r, idCiclo, meuId);
```