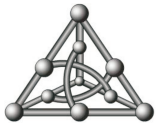


UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACULDADE DE COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

ROBERTO ARAGY XAVIER JUNIOR

Algoritmo BSP/CGM para o Problema do Fluxo Máximo em Redes

Campo Grande, MS - Brasil
Dezembro de 2010

Aragy Xavier Jr, Roberto, 1980 -

ALGORITMO BSP/CGM PARA O PROBLEMA DO FLUXO MÁXIMO EM REDES/Roberto Aragy Xavier Jr. - 2010

64 f.: il.

Orientador: Dr. Marco Aurélio Stefanes

Dissertação(Mestrado)- UFMS/Faculdade de Computação/Mestrado em Ciência da Computação, 2010

Referências Bibliográficas: f.56-57

1. Fluxo Máximo 2. Algoritmos Paralelo 3. Modelo BSP/CGM - Dissertação. I. Stefanes, Marco Aurélio. II. Universidade Federal de Mato Grosso do Sul, FACOM, Mestrado em Ciência da Computação. III. Algoritmo BSP/CGM para o Problema do Fluxo Máximo em Redes.

ALGORITMO BSP/CGM PARA O PROBLEMA DO FLUXO MÁXIMO EM REDES

Roberto Aragy Xavier Junior

Dissertação apresentada como parte dos requisitos para obtenção de grau de Mestre em Ciência da Computação no curso de Pós-Graduação em Ciência da Computação, Faculdade de Computação, Fundação Universidade Federal de Mato Grosso do Sul

Orientador: Prof. Dr. Marco Aurélio Stefanes

Resumo

Neste trabalho estudamos o Problema do Fluxo Máximo sob a ótica do paradigma do paralelismo. O objetivo geral desta dissertação é discutir os métodos sequenciais e paralelos para o Problema do Fluxo Máximo em Redes. Uma das contribuições deste trabalho é produzir um texto em português que trate dos principais algoritmos para o problema. Outra contribuição relevante é que propomos um novo algoritmo paralelo BSP/CGM que gasta $O(p)$ rodadas de comunicação para duas classes especiais de grafos. Nos resultados dos testes realizados em uma máquina paralela tipo Beowulf de 12 nós, observamos *speed-ups* superlineares de 1,85 até 107 com uso de classes de grafos especiais.

Palavras-chave: fluxo máximo, computação paralela, modelo BSP/CGM

Abstract

We study the maximum flow problem from the perspective of the paradigm of parallelism. The objective of this thesis is to discuss the sequential and parallel methods to the Problem of Maximum Flow in Networks. A contribution of this work is to produce a text in Portuguese, that addresses the main algorithms for the problem. Another important contribution is to propose a new algorithm parallel BSP/CGM spending $O(p)$ communication rounds for two special graphs' class. The results of tests performed on a Beowulf type parallel machine with 12 nodes, we observed superlinear speed-ups of 1.85 to 107 with the use of special graphs' class.

Keywords: max-flow, parallel computing, BSP/CGM model

Agradecimentos

Agradeço ao meu orientador Prof. Dr. Marco Aurélio Stefanes da FACOM/UFMS pela paciência e compreensão. Agradeço aos professores Prof. Dr. Alfredo Goldman Vel Lejbman da IME-USP e Profa. Dra. Edna Ayako Hoshino da FACOM/UFMS, que participam da banca desta defesa.

Agradeço também a coordenadoria do CTEI e seus colaboradores com o auxílio nos testes realizados da implementação, e aos professores e colegas da FACOM/UFMS pelo apoio e orientação, em especial aos Profs. Marcelo Henriques de Carvalho e Said Sadique Adi que participaram da banca de qualificação deste trabalho.

Agradeço a minha família em especial a minha esposa e filho pela força que me transmitem nas horas difíceis e a Deus pelas pessoas que coloca em meu caminho.

Conteúdo

Introdução p. 1

Introdução p. 1

1 Computação Paralela p. 3

1.1 Modelo PRAM - Parallel Random-Access Machine . . . . . p. 3

1.2 Modelo BSP . . . . . p. 4

1.2.1 O Modelo . . . . . p. 4

1.2.2 Custo . . . . . p. 5

1.3 Modelo CGM . . . . . p. 6

2 Problema de Fluxo Máximo em Redes p. 7

2.1 Aplicações do Problema de Fluxo Máximo . . . . . p. 7

Problema do Fluxo Viável . . . . . p. 7

Fluxo Máximo Dinâmico . . . . . p. 8

A guerra Fria e o abastecimento Ferroviário do Leste Europeu . . . . . p. 9

Emparelhamento Máximo em Grafo Bipartido . . . . . p. 9

2.2 P-Completo . . . . . p. 9

2.3 Definições do Problema . . . . . p. 11

2.4 Fluxos e Cortes . . . . . p. 12

3 Métodos Sequenciais para o Problema do Fluxo Máximo p. 15

3.1 Método de Caminhos Aumentantes . . . . . p. 15

3.1.1 Algoritmo de Ford-Fulkerson . . . . . p. 15
3.1.2 Algoritmo de Edmonds-Karp . . . . . p. 16
3.1.3 Algoritmo de Dinic . . . . . p. 20
3.2 Método *Push-Relabel* . . . . . p. 24

4 Algoritmos para o Problema do Fluxo Máximo em Redes para Computação Paralela p. 29
4.1 Algoritmo de Anderson-Setubal . . . . . p. 29
4.2 Algoritmo Autoestabilizante para o Problema de Fluxo Máximo . . . . p. 33

5 Algoritmo para o Problema do Fluxo Máximo em Redes para modelo CGM p. 44
5.1 Classe de Grafos para Entrada . . . . . p. 44
5.2 Estrutura de Dados . . . . . p. 45
5.3 Descrição do Algoritmo . . . . . p. 46
5.4 Cluster . . . . . p. 50
5.5 MPI(Message Passing Interface) . . . . . p. 50
5.6 Testes e Resultados . . . . . p. 51

6 Conclusão p. 55

Referências Bibliográficas p. 55

Referências p. 56

Introdução

Neste trabalho estudaremos um problema clássico da Teoria dos Grafos: o problema do Fluxo Máximo em Redes. A primeira noção que se pode ter ao se falar em fluxo em redes pode ser o abastecimento de água através de um sistema de encanamento, sistemas de abastecimento de mercadorias em um conjunto de localidades, ou até mesmo do montante de dados que percorre uma rede de computadores. O Problema do Fluxo Máximo em Redes corresponde a calcular uma quantidade máxima de unidades de um produto que passam por esses sistemas. Chamamos de redes um grafo orientado, ou seja, um conjunto de pontos, que são os vértices, interligados por arestas orientadas ou arcos. A cada um desses arcos é atribuído um valor inteiro denominado capacidade do arco. Esta capacidade é usada para representar restrições de transferência de produto. Podemos definir fluxo como sendo o montante de algum produto que é transferido de um determinado ponto a outro.

Apesar de ser um problema fundamental da Teoria de Grafos, o estudo do problema de fluxo em redes sob a ótica da computação paralela é pouco abordado, tendo ainda menos exemplos para o modelo realístico de computação BSP/CGM. Devido a variedade e importância das aplicações do problema associada a ascendente demanda do uso do paralelismo para incrementar o poder computacional de sistemas de processamento de dados, é crescente a necessidade de encontrar novas formas de resolver tanto esse como outros problemas correlatos.

O problema do Fluxo em Redes vem sendo estudado há muito tempo dentro da Teoria dos Grafos. Ford e Fulkerson [8] apresentaram o primeiro trabalho resolvendo o problema com o Algoritmo de Caminhos Aumentantes. Esse algoritmo apresenta complexidade pseudopolinomial de $O(nmU)$, para um grafo de n vértice e m arcos, onde U é o valor da maior capacidade de um arco do grafo. Além disso, neste trabalho foram descritos a maior parte dos conceitos utilizados pelos estudos que vieram depois. Posteriormente, Edmonds e Karp [7] propuseram uma implementação polinomial alterando o algoritmo proposto por Ford e Fulkerson de forma que a complexidade ficasse em $O(nm^2)$. Dinic [6] mostrou que era possível melhorar o tempo de resolução do problema, escolhendo o caminho com o menor número de arcos para enviar fluxo em uma rede rotulada pela distância dos vértices.

Esse algoritmo tem complexidade $O(n^2m)$. Karzanov [17] introduziu uma nova forma de calcular o fluxo máximo através de envio de fluxo por caminhos parciais e demonstrou que esse método é ainda mais eficiente que os apresentados até então, obtendo um algoritmo de $O(n^3)$. Cheriyan e Maheshwari em [3] mostraram que com esse método escolhendo o vértice de maior distância do sorvedouro pode-se obter um algoritmo com a complexidade de $O(n^2\sqrt{m})$.

Discutiremos, neste trabalho, os algoritmos acima e analisaremos métodos para paralelizar os algoritmos existentes do Problema do Fluxo Máximo em Rede. Este problema está entre aqueles de mais difícil paralelização, uma vez que os algoritmos conhecidos possuem características intrinsecamente sequenciais. Em termos de Teoria de Complexidade, este problema se encontra na classe dos problemas **P-completo**, isto equivale a dizer que todos os problemas da classe **P** podem ser reduzidos a ele. Uma vez encontrado uma paralelização eficiente para ele, podemos paralelizar com eficiência todos os problema contidos em **P**. Entre os estudos anteriores de algoritmos paralelos para o problema temos: Shiloach e Vishkin [22] em 82 escreveram um algoritmo PRAM que roda em tempo $O(n^2 \log n)$ com $O(n)$ processadores. Goldberg [24] apresentou em 1991 um algoritmo PRAM com estruturas de dados mais eficientes e usando menos memória. Esse algoritmo roda em $O(n^2 \log n)$ com $O(n)$ processadores. Mais recentemente, em 2005, Tabirca e Tabirca [24] descreveram um algoritmo de tempo $O(n^2)$ usando $O(m)$ processadores. O objetivo geral desta dissertação é discutir os métodos sequenciais e paralelos para o Problema do Fluxo Máximo em Redes. Como objetivos específicos descreveremos um novo algoritmo paralelo BSP/CGM que gasta $O(p)$ rodadas de comunicação para duas classes especiais de grafos. Até onde conhecemos esse é o primeiro algoritmo paralelo no modelo BSP/CGM para o problema. Nos testes experimentais que realizamos num cluster com 16 nós obtivemos *speed-up* superlineares de 1,85 até 107, para as classes especiais de grafos escolhidos.

Este trabalho está disposto da seguinte forma: no primeiro capítulo comentaremos sobre o modelo de computação paralela PRAM e BSP/CGM. No capítulo seguinte introduziremos conceitos e notações necessárias para a definição e apresentação dos algoritmos que seguem nos capítulos seguintes. No terceiro capítulo, demonstraremos o funcionamento dos algoritmos Ford-Fulkerson, Edmonds-Karp, Dinic e o *push-relabel*. Em seguida, no capítulo 4 vemos as propostas de Anderson-Setubal e Gosh *et. al.* de paralelismo do problema, e então, no capítulo 5 descreveremos um novo algoritmo para resolução do problema no modelo BSP/CGM.

1 Computação Paralela

Na primeira década do atual século, a computação sequencial encontrou uma barreira que impede o crescimento exponencial da quantidade de cálculos que até então acontecia, a barreira física do tamanho dos transistores que a tecnologia atual permite. Mas esse paradigma começou a ser quebrado com estudos de paralelismo computacional iniciados na década de 80. A computação paralela é o modo de realizar cálculos simultâneos, dividindo problemas grandes em subproblemas menores que são resolvidos paralelamente. Computador paralelo é o conjunto de processadores organizados sob um determinado modelo de interconexão que os permitam a coordenação das atividades e trocas de informações. Os modelos de computação paralela são classificados de acordo com o nível de paralelismo de instruções e de dados. Vamos ver alguns desses modelos a seguir.

1.1 Modelo PRAM - Parallel Random-Access Machine

O modelo PRAM [16] uma extensão natural do modelo de computação sequencial de Von Neumann. Neste modelo temos uma grande quantidade de processadores, cada um com sua memória local, executando um programa local e toda a comunicação é feita através de troca de dados por uma unidade de memória global compartilhada. Cada processador tem um identificador único, acessível localmente. No modelo PRAM todos os processadores operam de modo síncrono sob o controle de um *clock* comum.

Um algoritmo desenvolvido para o modelo PRAM devem ser do tipo *single instruction multiple data* (SIMD), ou seja, todos os processadores executem o mesmo programa a cada unidade de tempo, mas sobre dados diferentes. Além do mais, o modelo permite que diferentes programas possam ser carregados nas memórias locais do processador, contanto que os processadores possam operar sincronamente.

Existem três variações do modelo PRAM baseados na forma de permissão de leitura e escrita na memória compartilhada. O *exclusive read exclusive write* (EREW) PRAM que

não permite acesso simultâneo a um único local de memória. O *concurrent read exclusive write* (CREW) PRAM que permite acesso simultâneo apenas para leitura de instruções. O *concurrent read concurrent write* (CRCW) PRAM permite acesso simultâneo de leitura e escrita.

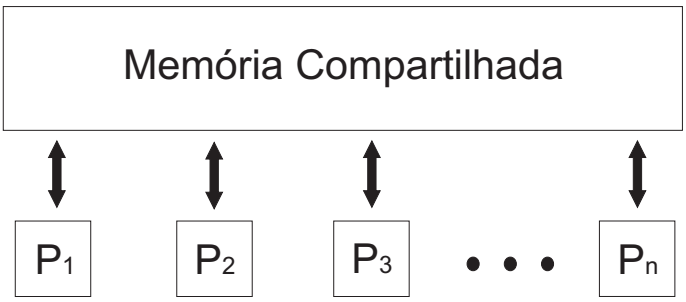


Figura 1: Modelo PRAM

1.2 Modelo BSP

O modelo BSP (Bulk Synchronous Parallel Model), como proposto por Valiant [25], tem como objetivo principal servir de modelo ponte entre as necessidades de hardware e software na computação paralela.

1.2.1 O Modelo

O BSP consiste de uma máquina com p processadores com memória local, comunicando-se entre si através de uma rede local gerenciados por um roteador, e com facilidade de sincronização. O algoritmo BSP consiste em uma sequência de superpassos globais. Cada superpasso consiste em três estágios:

Computação Concorrente : Neste estágio todos os processadores executam instruções localmente, processando os dados armazenados na memória local;

Comunicação : Neste estágio, ocorre a troca de informações entre os processadores. A comunicação no modelo BSP é realizado em massa, considerando que todas as ações de comunicação de cada superpasso como única e assumindo que todas as mensagens têm o mesmo tamanho;

Barreira de Sincronização : É o estágio onde os processadores só passam para o próximo estágio se todos o atinge, fazendo com que eles iniciem o próximo estagio de computação concorrente ao mesmo tempo.

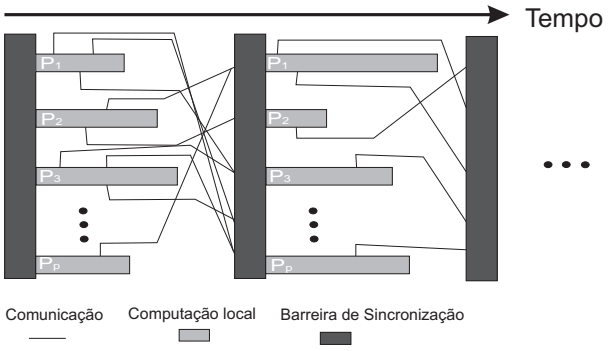


Figura 2: Modelo BSP

1.2.2 Custo

O custo da computação de um algoritmo BSP é determinado por três fatores: o custo do processamento mais longo de computação local, o custo da comunicação entre os processadores e o custo da barreira de sincronização.

O número máximo de mensagens chegando ou saindo do processador i é denotado po h. O tempo de entrega da mensagem através da rede é representado por g. Portanto hg é o custo do processador entregar h mensagens de tamanho 1. Sendo m o tamanho da maior mensagem transmitida entre os processadores, que obviamente leva um tempo maior para ser entregue do que uma mensagem de tamanho 1, podemos expressar que o custo de comunicação é limitado por mgh.

O custo do estágio da barreira de sincronização, que é representado por l, depende da variação de tempo de conclusão da computação local entre os processadores e o custo para se atingir um estado de consistência global em todos os processadores, mas na prática é determinado empiricamente.

Sendo w_i o tempo de conclusão da computação local no processador i, temos que o cálculo do custo do superpasso é obtido na seguinte fórmula:

$$max^{1\leq i\leq p}(w_i) + max^{1\leq i\leq p}(m_i g h_i) + l \tag{1.1}$$

Dada a Equação 1.2.2 expressado um superpasso e considerando S a quantidade de superpassos, temos que o custo de um algoritmo BSP é limitado por:

$$\sum_{s=1}^{S}(w_s) + g\sum_{s=1}^{S}(m_s h_s) + Sl \tag{1.2}$$

1.3 Modelo CGM

O modelo CGM (Coarse Granularity Multicomputer), proposto por Dehne *et.al.* [5], consiste de vários processadores com memória local, conectados por um meio qualquer de intercomunicação, rede de interconexão ou memória compartilhada. Um algoritmo CGM alterna computação local com rodadas de comunicação global. O termo *"granulosidade grossa"* é originado no fato de que o tamanho do problema n ser consideravelmente maior que o número de processadores p ($n/p \gg p$).

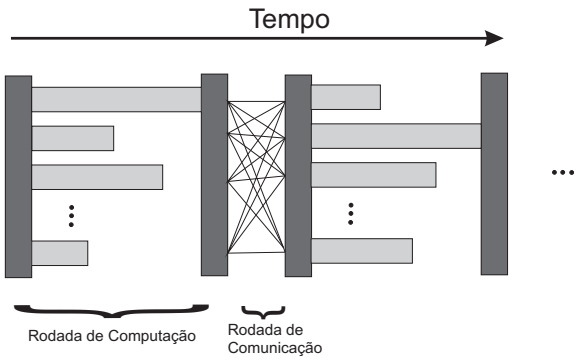


Figura 3: Modelo CGM

Uma rodada de computação junto com uma rodada de comunicação no modelo CGM equivale a um superpasso no modelo BSP. O custo de um algoritmo CGM é a soma dos tempos obtidos em termos do número total de rodadas de computação local e de comunicação. Um algoritmo CGM tem o objetivo de resolver um problema executando a menor quantidade possível de rodadas de comunicação mantendo-se a razão entre o tempo do melhor algoritmo sequencial e do algoritmo paralelo proporcional a p.

No universo de pesquisa de computação paralela existem vários outros modelos, mas por estarem fora do escopo do estudo deste trabalho eles não foram citados.

2 *Problema de Fluxo Máximo em Redes*

O Problema de Fluxo Máximo (PFM) [8] é um problema de otimização combinatória cujas aplicações práticas podem ser muito vastas. O fluxo em uma rede, intuitivamente, é a taxa na qual os produtos se movem. Muitos exemplos podem ser tirados de nosso dia a dia. Rede de telefonia ou rede elétrica, as conexão dos *backbones* da internet, o fluxo de tráfego de autoestradas e ferrovias e também os clássicos encanamentos de água, esgoto e gás são bons exemplos onde podemos observar a aplicação deste problema.

Neste capítulo comentaremos sobre aplicações e exemplos na primeira seção e na seção seguinte abordaremos superficialmente os conceitos básicos do problema.

2.1 Aplicações do Problema de Fluxo Máximo

O Problema do Fluxo Máximo pode ser aplicado em muitas situações. Quando se imagina um problema que trata da otimização ou quantificação de fluxo por uma rede, muito provavelmente estaremos tratando do problema de Fluxo Máximo em Redes ou de uma variação deste problema. Ilustramos em seguida alguns exemplos de aplicações.

Problema do Fluxo Viável

Suponha um sistema de portos onde navios fazem a transferência de mercadorias entre eles através de rotas pré-determinadas. Suponha que exista um tipo de mercadoria que esteja disponível em alguns portos deste sistema. É conhecida a quantidade disponível de mercadoria no estoque de cada porto, o montante requerido por outros portos e a quantidade máxima que pode ser transportado em cada rota. Considere que o montante de toda a mercadoria que sai de cada porto i menos o que entra em cada porto i seja a demanda deste porto representado por $b(i)$, deseja-se saber se é possível atender as demandas de cada porto usando o suprimento disponível.

Para resolver o problema vamos assumir o seguinte cenário. Representemos o problema em um grafo $G = (V, A)$, onde V serão os portos e A as rotas possíveis entre esses portos. Sendo assim, assuma que $\sum_{i \in V} b(i) = 0$. Vamos acrescentar ao conjunto V dois vértices, um vértice fonte s e um sorvedouro t. Para cada porto com $b(i) > 0$ vamos acrescentar ao conjunto de arcos A um novo arco (s, i) com capacidade $b(i)$, e para cada porto com $b(i) < 0$ vamos acrescentar um arco (i, t) com capacidade $-b(i)$. Portanto agora teremos uma rede $N = (V', A', s, t, c)$ onde V' e A' são os conjuntos V e A com seus respectivos acréscimos. Então se calcula um fluxo máximo para a rede N. Se todo $(s, i) \in A'$ e todo $(i, t) \in A'$ são saturados nesse processo então existe um fluxo viável em G.

Fluxo Máximo Dinâmico

O problema de fluxo máximo dinâmico é uma variante do Problema do Fluxo Máximo onde pode ser tirado o seguinte cenário. Em uma guerra entre os países A e B, suponha que o general do exército de A tenha decidido lançar um ataque maciço nas 24 horas a seguir, usando suas melhores unidades de infantaria baseadas na localização s, contra o inimigo localizado em t. O general gostaria de enviar o maior número de unidades do local s até t no período de 24 horas, obedecendo a capacidade dos acesso por período de tempo.

No problema do fluxo dinâmico, maximiza-se o número total de unidades de fluxo que podem ser enviados de s a t no tempo p enquanto satisfaz a capacidade do arco $c(i, j)$ e do arco de tempo transversal $p(i, j)$.

Dada uma rede $N = (V, A)$, podemos montar a rede N^p da seguinte forma. Faça p cópias de cada vértice i. O vértice i_k na rede de tempo expandido representa o vértice i original no tempo k. Inclui-se um arco (i_k, j_l) de capacidade $c(i, j)$ na rede de tempo expandido sempre que $(i, j) \in A$ e $l - k = p(i, j)$. O arco (i_k, j_l) na rede N^p representa o potencial movimento das unidades militares entre o vértice i até o vértice j no intervalo de tempo de k a l. Ainda pode-se reduzir as múltiplas fontes e múltiplos sorvedouros com o acréscimo de uma superfonte s^* e um supersorvedouro t^*. Resolvendo o fluxo máximo em N^p resolve-se o problema do Fluxo dinâmico em N.

A guerra Fria e o abastecimento Ferroviário do Leste Europeu

Durante os anos de Guerra Fria, no inicio da década de 50 foi formulado um problema devido ao interesse que o exército dos Estados Unidos tinha sobre a malha ferroviária do Leste Europeu e Oeste da União Soviética. T.E. Harris formulou o seguinte: "Considere uma rede ferroviária conectando duas cidades através de um número de cidades intermediárias, onde cada ligação entre cada cidade tenha associado um valor de capacidade de transporte. Encontre um fluxo maximal de uma dada cidade até outra". T.E. Harris em conjunto com Gal. F. S. Ross formulou um relatório ([14]) a partir de estudos da região e fotos de satélite sobre as capacidades operacionais de abastecimento da região através dessa malha ferroviária e formas de interromper o abastecimento dessa região pelos trilhos. Foi este relatório que inspirou as idéias de "Fluxo Máximo Corte Mínimo"de Ford-Fulkerson, onde ao descobrir o corte mínimo da malha ferroviária poderia minimizar os esforços da forças armadas dos Estados Unidos para desabastecer a região, no caso de um possível embate com o exército vermelho. Por questões política o memorando foi apresentado apenas em 1955.

Emparelhamento Máximo em Grafo Bipartido

O problema do Fluxo Máximo pode ser usado na resolução de muitos outros problemas matemáticos e de combinatória. Um dos exemplos é o Emparelhamento Máximo em grafos Bipartidos. A solução se daria da seguinte forma. Insere-se no grafo dois novo vértices, o vértice fonte s e o sorvedouro t, onde s tem um arco que o conecta a cada um dos vértices do primeiro grupo V_1 enquanto que t tem um arco conectando-o cada vértice do segundo grupo V_2. Atribui-se como capacidade de uma unidade para cada arco dessa rede formada. Então é calculado o fluxo máximo nessa rede. Os arcos que ligam os vértices de V_1 aos de V_2 que forem saturados, vão formar o emparelhamento máximo.

2.2 P-Completo

Em complexidade computacional, a classe de complexidade **P-completo** é um conjunto de problemas de decisão de grande valia para analisar quais problemas podem ser resolvidos eficientemente em máquinas paralelas. A classe de complexidade **NC** é o conjunto de problemas que podem ser decididos por uma máquina paralela PRAM em tempo $O(\log^c n)$ utilizando $O(n^k)$ processadores, onde c e k são constantes.

Um problema de decisão é **P-completo** se está em **P** e todo problema de **P** pode ser reduzido a ele em tempo polilogaritmico em uma máquina paralela PRAM usando um número polinomial de processadores. Em outras palavras, um problema A está em **P-completo** se para todo problema B em **P**, existem constantes c, e k tais que B pode ser reduzido a A em tempo $O(\log^c n)$ utilizando $O(n^k)$ processadores.

A classe **P** contém os problemas solucionáveis por uma máquina de Turing, ou seja uma máquina sequencial, e contém a classe NC, que contém os problemas que podem ser resolvidos eficientemente por uma máquina paralela. Isto é fácil de ver, uma vez que as máquinas paralelas podem ser simuladas por máquinas sequenciais. Uma questão em aberto é se **NC=P**. Da mesma forma que se acredita que **P** é diferente de **NP**, acredita-se também que as classes **NC** e **P** são distintas.

A classe **P-completo**, é composta do problemas provavelmente inerentemente sequenciais, ou seja dos problemas não paralelizáveis. Esta classe é utilizada para estudar a questão **NC=P**. Se for encontrada uma paralelização eficiente de um problema **P-completo** então teríamos que **NC=P**.

Entre os vários problemas que podem ser provados ser **P-completos** podemos citar:

- Problema do Valor do Circuito (CVP): dado um circuito, com as entradas do circuito, e uma porta saída no circuito, saber se a saída dessa porta é 1.
- Programação Linear: dada uma matriz A de inteiros de dimensão $n \times d$, um vetor b de inteiros de tamanho $n \times 1$ e um vetor c de inteiros de tamanho $1 \times d$, deseja-se encontrar o vetor rotacional x de tamanho $d \times 1$ no qual $Ax \leq b$ e cx está maximizado.
- Problema do Fluxo Máximo: dado um grafo dirigido $N = (V, A, s, t, c)$, onde a cada arco a associamos uma capacidade inteira positiva $c(a)$, dois vértices s, e t e um inteiro k, queremos saber se k-ésimo bit do valor de um fluxo máximo da fonte s ao sorvedouro t é 1.

Uma vez que o Problema do Fluxo Máximo é **P-completo**, fica claro a dificuldade da paralelização deste problema, evidenciando as razões de existirem apenas algoritmos experimentais para sua solução paralela, nem mesmo existindo nenhum estudo conhecido de um algoritmos BSP/CGM para esse problema.

2.3 Definições do Problema

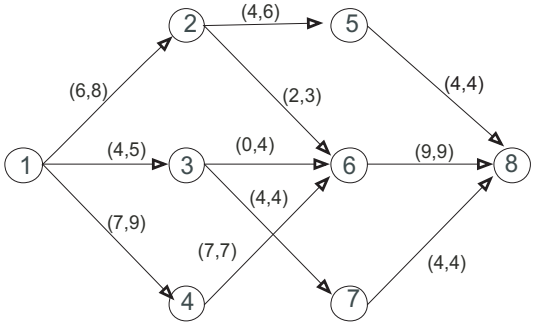
Seja uma rede orientada $N = (V, A, s, t, c)$, onde cada arco $(i, j) \in A$ está associado a uma função de capacidade $c(i, j) > 0$. Além disso, considera-se dois vértices especiais, a fonte s e o sorvedouro t. Um fluxo viável é uma função $f : V \times V \to \mathbb{R}$ onde as seguintes propriedades são satisfeitas:

$$\sum_{j:(j,i)\in A} f(j,i) - \sum_{j:(i,j)\in A} f(i,j) = \begin{cases} -z & \text{para } i = s \\ 0 & \text{para todo } i \in V - \{s \text{ e } t\} \\ z & \text{para } i = t \end{cases} \tag{2.1}$$

$$f(i,j) \leq c(i,j) \text{ para cada } (i,j) \in A \tag{2.2}$$

A restrição 2.1 é a restrição de balanceamento de fluxo e quer dizer que, com exceção da fonte e do sorvedouro, todo fluxo que entra em um vértice é igual ao que sai dele. Enquanto que a restrição 2.2, que a restrição de capacidade, diz que o fluxo que atravessa um arco não pode exceder sua capacidade.

O Problema do Fluxo Máximo consiste em encontrar um fluxo viável de forma que z tenha valor máximo. Chamamos o valor z de Fluxo na Rede N e abusando da notação definimos que $f(s, t) = z$. Na figura 4 temos um exemplo de fluxo máximo em uma Rede onde $f(s, t) = 17$. Associado a cada arco temos os valores entre parênteses separados por vírgula onde o primeiro representa o fluxo que passa pelo arco e o segundo a capacidade máxima do arco.



(a)

Figura 4: Exemplo de Fluxo Máximo. Nos arcos - (f(i,j),c(i,j))

Neste trabalho vamos supor as seguintes limitações para o conjunto de grafos sobre os quais trabalharemos:

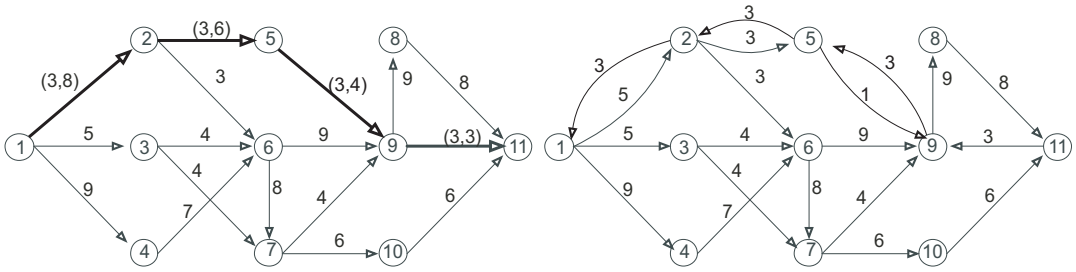
- O grafo é orientado;
- Todas as capacidades são de valores inteiros não negativos;
- O grafo não contém caminhos diretos compostos apenas de arcos de capacidades infinitas;
- Sempre que um arco (i, j) pertencer a A, o arco (j, i) também pertencerá;
- O grafo não conterá arcos paralelos.

2.4 Fluxos e Cortes

Os primeiros estudos sobre fluxo apresentados pelo o método de Ford-Fulkerson, e introduzidos em [8], tinham como base três conceitos importantes, não apenas para o método, mas também para muitos algoritmos de fluxos que vieram depois. Estes conceitos são descritos aqui, a saber: redes residuais, caminhos aumentantes e cortes.

Uma vez que a rede N é atravessada por um montante de fluxo, este ocupará parcialmente ou totalmente a capacidade dos arcos da rede. A diferença entre a capacidade do arco e a capacidade ocupado pelo fluxo é dado por $r(i, j) = c(i, j) - f(i, j) + f(j, i)$, onde $r(i, j)$ é denominado **capacidade residual** do arco (i, j). Quando $r(i, j) = 0$, diz-se que (i, j) é um arco saturado e se $r(i, j) > 0$ diz-se que (i, j) é um arco **insaturado**. A rede obtida de N e de um fluxo f, associando um valor $r(i, j)$ a cada arco, é chamado **rede residual**. Dado a rede N mostrada na figura 5a, para calcular a rede residual N_f a partir de um fluxo no valor 3 passando pelo caminho $P = (1, 2), (2, 5), (5, 9), (9, 11)$, substituiremos os arcos pertencentes a P por um arco de retorno com capacidade no valor do fluxo que passa pelos arcos do caminho P e um arco no mesmo sentido ao pertencente a P mas com capacidade igual a capacidade residual do arco. Como no exemplo da Figura 5b, o arco $(1, 2)$ será substituído pelos arcos $(1, 2)$ com capacidade 5 e $(2, 1)$ com capacidade 3. O mesmo ocorre para todos os arcos de P. O arco cuja capacidade residual é 0 não será representado no grafo residual, constando apenas o arco de retorno com a capacidade no valor do fluxo. A rede residual N_f pode conter ciclos. Onde em A está associado um função $c(i, j)$, aos arcos de A_f estão associados a função $r(i, j)$ conforme definido acima.

Seja P um conjunto de arcos pertencentes a A_f que formam um caminho de s a t. Seja x o menor valor da capacidade residual dentre os arcos pertencentes a P. O caminho P é dito **caminho aumentante** se $x > 0$. Dada $N_f = (V, A_f, s, t, r)$ uma rede residual



(a) Rede com fluxo (b) Rede residual

Figura 5: Exemplo de Rede Residual com $f(s,t) = 3$

derivada de N, onde N é uma rede acíclica como definida anteriormente, $|A_f|$ pode variar conforme o fluxo nos arcos de A seja aumentado.

Formalmente, seja S um subgrafo de N tal que $s \in V(S)$ e $t \notin V(S)$. Considere $\bar{S}$ um subgrafo de N tal que $s \notin V(\bar{S})$ e $t \in V(\bar{S})$. Definimos um corte s-t, denotado por $[S, \bar{S}]$, como sendo o conjunto de arcos (i, j) tais que $i \in V(S)$ e $j \in V(\bar{S})$. Veja Figura 6. Definimos também que a **capacidade do corte *s-t*** ($c[S, \bar{S}]$) por $c[S, \bar{S}] = \sum_{(i,j)\in[S,\bar{S}]} c(i, j)$, que é o montante máximo de fluxo que pode passar do subgrafo S para o sub-grafo $\bar{S}$. A capacidade residual do corte s-t é $r[S, \bar{S}] = \sum_{(i,j)\in[S,\bar{S}]} r(i, j)$. Consequentemente teremos que o fluxo sobre o corte s-t é o somatório do fluxo que sai do subgrafo S em direção a $\bar{S}$, menos o fluxo que sai de $\bar{S}$ em direção a S, ou seja, $f[S, \bar{S}] = \sum_{(i,j)\in[S,\bar{S}]} f(i, j) - \sum_{(j,i)\in[\bar{S},S]} f(j, i)$. Tendo as definições acima pode fazer algumas observações quanto ao fluxo na rede.

Lema 2.1 *Existindo um fluxo na rede, $f(s,t)$ estará limitado ao valor da capacidade de qualquer corte que possa ser feito em N.*

Esse lema é simples de ser provado, pois para manter as restrições da capacidade dos arcos, o fluxo $f(s,t)$ também não pode exceder a capacidade do corte.

Neste ponto chegamos à nossa última definição desta seção, o **corte mínimo** é um corte s-t onde a soma das capacidades de seus arcos é menor ou igual a qualquer outro corte s-t existente em N. Portanto temos o seguinte teorema.

Teorema 2.1 ***Teorema do Fluxo Máximo Corte Mínimo**[1] Se fluxo $f(s,t)$ em N é máximo, as seguintes condições são atendidas:*

1. *A rede residual N_f não possui caminhos aumentantes;*
2. *$f(s,t) = c[S, \bar{S}]$, onde $[S, \bar{S}]$ é o corte mínimo de N.*

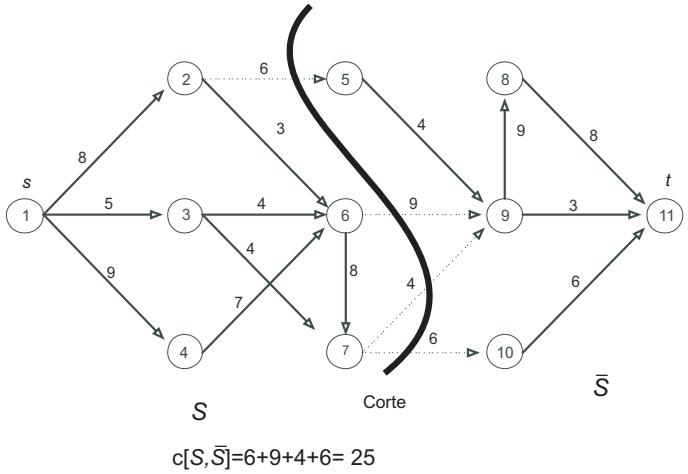


Figura 6: Corte $s - t$ e capacidade do corte

Prova:

1. Se existir um caminho aumentante em N_f então claramente $f(s,t)$ pode ser aumentado no valor da menor capacidade residual dos arcos deste caminho.

2. Este item precisa ser provado em duas partes:

   Conforme o lema 2.1, o corte mínimo será o limitante superior do fluxo $f(s,t)$.

   Para efeito de contradição, vamos supor que existe pelo menos um arco $(i,j) \in [S,\bar{S}]$ que não tenha a capacidade completamente ocupada em uma rede residual ao ser encontrado o fluxo máximo. Três situações podem ocorrer. Não existe nenhum caminho na rede residual onde s alcance i. Neste caso há pelo menos um arco (u,v) onde v alcança i na rede residual ou $v = i$ na qual a capacidade é totalmente ocupada por fluxo, o que significa que $c(u,v) < c(i,j)$ o que é uma contradição pois senão $(u,v) \in [S,\bar{S}]$ e não (i,j). No segundo caso é o mesmo do primeiro mas do lado de j não alcançar t, onde caso exista pelo menos um arco (u,v), onde j alcança u ou $j = u$ e possui a capacidade totalmente ocupada por fluxo,impedindo que j alcance t na rede residual. Isso significa que $c(u,v) < c(i,j)$ que também é uma contradição pois senão $(u,v) \in [S,\bar{S}]$. O terceiro caso é se existe um caminho P onde i é alcançado por s e j alcança t na rede residual. Isso significa que o fluxo pode ser aumentado pelo caminho P, o que é uma contradição.

□

3 Métodos Sequenciais para o Problema do Fluxo Máximo

O problema do Fluxo Máximo é estudado desde os anos 50 e são conhecidos algoritmos sequenciais eficientes para resolução do problema. Esses algoritmos são de dois tipos. Os algoritmos de Caminhos Aumentantes que mantém as restrições de balanceamento de fluxo para todos os vértices, com exceção de s e t, a cada passo de execução; e os algoritmos de *Preflow-Push* que inunda a rede fazendo com que alguns vértices tenham excessos e aos poucos o algoritmo encontra o balanceamento de fluxo até maximizá-lo.

3.1 Método de Caminhos Aumentantes

Os métodos de caminhos aumentantes foram os primeiros estudados para resolver o Problema do Fluxo Máximo. Cronologicamente, o algoritmo de caminhos aumentantes foi apresentado por Ford e Fulkerson em 1962 em [8], mas o algoritmo não é polinomial, ou seja, ele resolve o problema em tempo de $O(nmU)$. Em 1970, Y. Dinic publicou um algoritmo polinomial para resolução do problema que roda no tempo de $O(n^2m)$ e de forma independente, em 1972, Edmonds e Karp descrevem seu algoritmo que roda em $O(nm^2)$ apresentando melhoria no método de Ford-Fulkerson. Esses algoritmos serão vistos a seguir.

3.1.1 Algoritmo de Ford-Fulkerson

O método de Ford-Fulkerson lida iterativamente com a idéia de procurar caminhos aumentantes na rede residual. A cada iteração é encontrado um caminho aumentante de s a t para que se escolha o valor da menor capacidade residual dentre os arcos deste caminho e aumenta-se o fluxo de todos os arcos do caminho nesse valor. Repete-se essa operação até que se encontra uma rede residual onde não haja um caminho aumentante ligando s a t.

Algoritmo 1 Método de Ford-Fulkerson

Entrada: Rede N
Saída: Fluxo máximo para f(s,t) e o fluxo em cada arco de N

```
f(s,t) ← 0;
Enquanto existir caminho aumentante P que ligue s a t faça
    f(s,t) = min{r(i,j) | (i,j) ∈ P}
    Para Cada arco (i,j) de P faça
        atualize a capacidade residual
    fim Para
fim Enquanto
```

O tempo de execução do algoritmo de Ford-Fulkerson depende de como o caminho aumentante é escolhido. A análise se dá da seguinte forma. Para que seja escolhido o caminho aumentante, vamos supor que o algoritmo utilize uma estrutura eficiente que represente a rede residual da rede N, e nesta seja feita ou uma busca em largura ou profundidade. O tempo para encontrar um caminho na rede residual é $O(n+m) = O(m)$ onde $|V| = n$ e $|A| = m$. Sendo $U = f(s,t)$ onde $f(s,t)$ é máximo, o laço *enquanto* é executado no máximo U vezes, pois o valor do fluxo aumenta em pelo menos uma unidade a cada iteração. A atualização dos arcos dentro do laço *enquanto* é $O(n)$. Portanto a execução do método tem o tempo pseudopolinomial de $O(nmU)$.

3.1.2 Algoritmo de Edmonds-Karp

O tempo de execução do Algoritmo 1 pode ser melhorado escolhendo em cada iteração o menor caminho aumentante entre s e t como apresentado por Edmonds e Karp [7]. Dessa forma, pode-se obter um algoritmo de tempo polinomial de $O(nm^2)$.

Com o intuito de enviar fluxo por caminhos aumentantes mínimos, identifica-se a distância dos vértices com relação ao sorvedouro. A distância é, portanto, uma função $d : V \rightarrow \mathbb{Z}^+ \cup \{0\}$. A função de distância é valida, desde que obedeça às seguintes restrições:

$$d(t) = 0 \tag{3.1}$$

$$d(i) = d(j) + 1 | c(i,j) > 0 \text{ para } (i,j) \in A \tag{3.2}$$

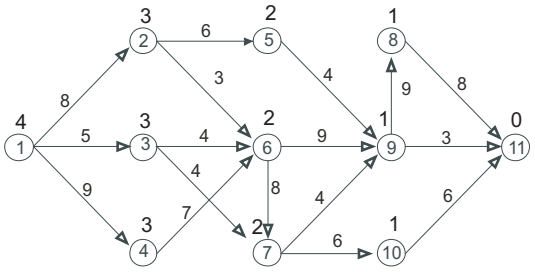


Figura 7: Rede residual rotulada com função de distância $d(i)$

Tendo os valores de distância das extremidades de um arco (i, j), temos que um arco é dito **arco admissível** se a restrição 3.1.2 é atendida, caso contrário, o arco (i, j) é um **arco inadmissível**.

O Algoritmo 2 de Caminhos Aumentantes mais Curtos extraído de [1] usa a função de distância para descobrir o caminho aumentante mais curto entre s e t para aumentar o fluxo máximo do grafo .

Primeiramente, o algoritmo calcula as distâncias de cada vértice em relação ao sorvedouro usando um algoritmo de busca em largura. Então o algoritmo, a partir das informações de distância e com o auxílio de uma estrutura que se comporte como uma pilha, tenta percorrer um caminho partindo de s, avançando para o vértice vizinho através de arcos admissíveis, tomando o cuidado de guardar a informação dos vértices anteriores no caminho na estrutura de pilha. Quando chega a um vértice que não possua arcos admissíveis, então realiza retorno, que aumenta o valor da distância. Se for encontrado o vértice sorvedouro, então se realiza o aumento do fluxo calculando os novos arcos da rede residual e reinicia-se a busca a partir de s de um outro caminho na rede residual calculada. Quando o vértice fonte for rotulado com valor igual ao número de vértices do grafo, o algoritmo termina.

Algoritmo 2 Caminhos Aumentantes Mais Curtos

Entrada: Rede N

Saída: Fluxo x e a Rede N com x de fluxo o atravessando

```
x ← 0; // x é o fluxo inicial
Calcular a função d(i) de cada vértice;
i ← s
Enquanto d(s) < n faça
  Se i possui arco admissível então
    avanca(i)
    Se i = t então
      aumentar_fluxo()
      i ← s
    fim Se
  Senão
    recuar(i)
  fim Se
fim Enquanto
```

1 $avancar(i)$

```
selecionar um arco (i, j) admissível
pred(j) ← i
i ← j
```

2 $recuar(i)$

```
d(i) ← min{d(j) + 1 : dentre os arcos (i, j) ∈ A com r(i, j) > 0}
Se i ≠ s então
  i ← pred(i)
fim Se
```

3 $aumentar\_fluxo()$

```
identificar a partir dos índices de precedência o caminho aumentante
y ← min{r(i, j), dentre os arcos do caminho identificado}
x ← x + y
atualiza a rede residual conforme o aumento de fluxo
```

O exemplo da figura 8a mostra a instância inicial do grafo, onde os números sobre os arcos representam suas respectivas capacidades e os números sobre os vértices representam d. A variável i aponta para o vértice 1. O algoritmo procura um arco admissível saindo do vértice 1, arco $(1, 2)$ no caso. Então i avança para apontar para o vértice 2 e atribui a $pred(2)$, 1(Fig. 8b). Repete-se o procedimento para os próximos vértices, 5 e 8(Fig. 8c e d), quando alcança o vértice t. Então se faz o aumento do fluxo e atualiza o grafo residual e atribui 1 a variável i, ficando como na figura 8e. Novamente repete-se o procedimento de avanço para os vértices 2 e 5 (Fig. 8f), até que não seja mais possível avançar, pois 5 não tem mais arcos admissíveis, sendo necessário fazer a operação de recuo, onde é alterado

o valor do rótulo de 5 para a menor distância de um vizinho alcançável acrescido em 1 (Fig. 9a). É atribuído 2 a variável i. O algoritmo encontra apenas o arco $(2, 6)$ como admissível, portanto i avança para o vértice 6 e em seguida para o vértice 8 (Fig. 9b), quando é realizado o aumento e atualização do grafo residual(Fig. 9c). E assim, com avanços e recuos, o grafo realiza os aumentos e atualizações dos grafos residuais até que em dado momento é alterado o valor do vértice fonte para maior que o número de vértices no grafo, terminando o algoritmo.

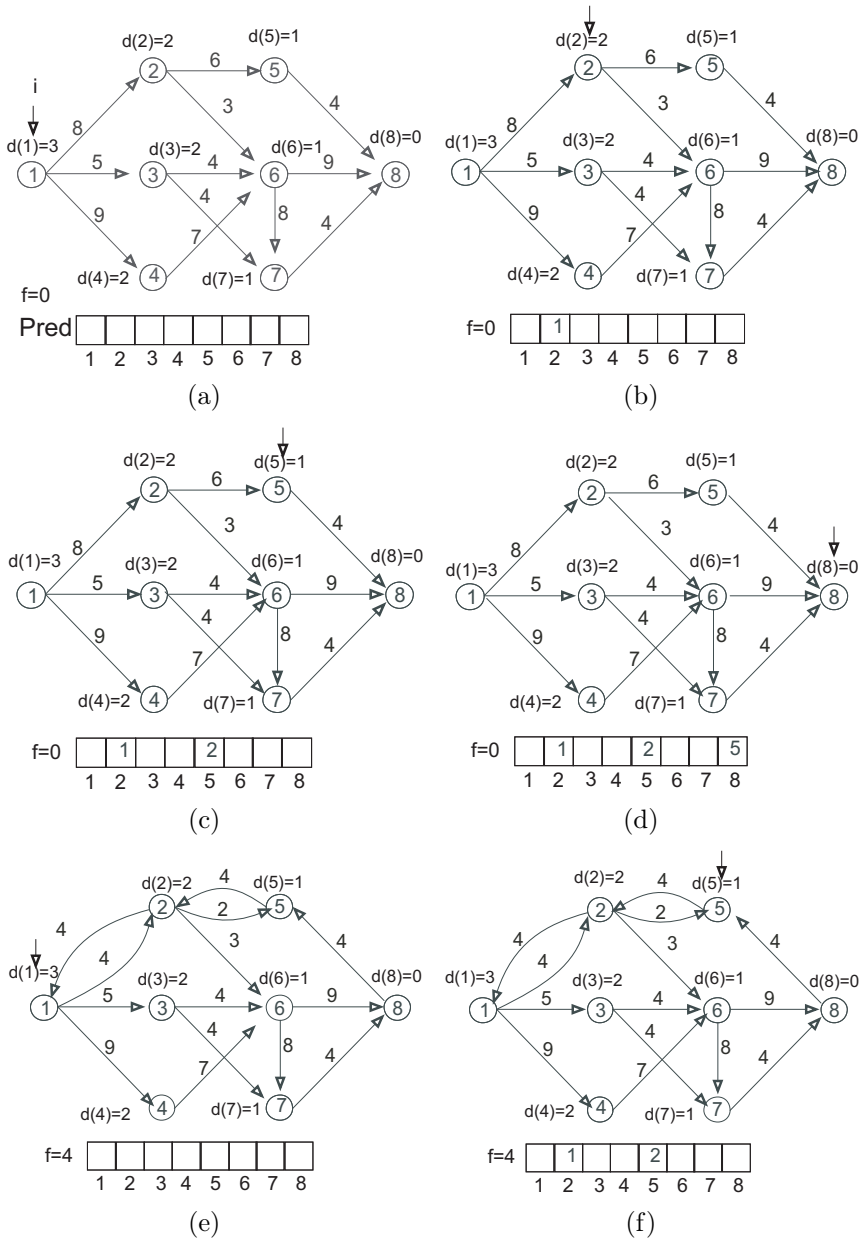


Figura 8: Algoritmo de Caminhos Aumentantes Mais Curto

Para a análise do tempo de execução deste algoritmo vamos supor que em uma rede residual N_f com um fluxo f, um arco $(i, j) \in N_f$ é **crítico** em um caminho aumentante

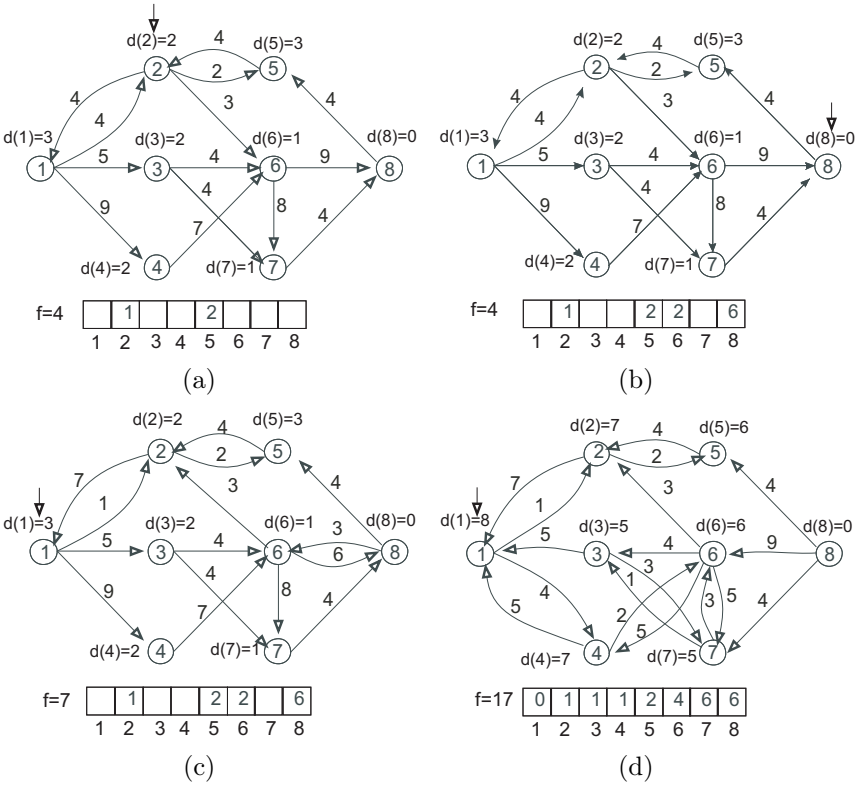


Figura 9: Algoritmo de Caminhos Aumentantes Mais Curto

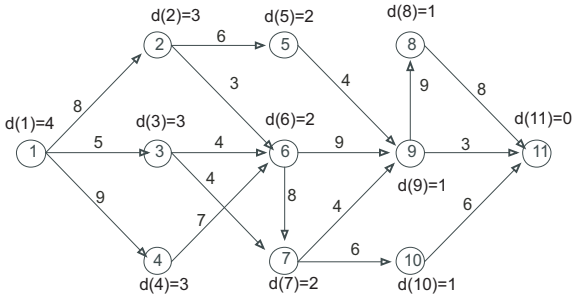
P se a capacidade residual $r(i, j)$ é a menor dentre os arcos de P. A cada aumento pelo menos um arco em cada caminho deve ser crítico, e esse caminho escolhido é sempre o mais curto em N_f. Depois que há um aumento de fluxo por P, o arco (i, j) deixa de existir em N_f até que em outra iteração um caminho aumentante P' tenha $(j, i) \in P'$. Para que (i, j) volte a existir na rede residual, a distância $d(j)$ deve ser aumentada. Se no primeiro momento $d(i) = d(j) + 1$, para o arco (j, i) pertencer a P' então as distâncias deverão atender $d(j) = d_f(i) + 1$. Isso implica que a distância de i aumenta a cada vez em, no mínimo, duas unidades. Para se tornar inalcançável pela fonte em uma rede residual, cada vértice pode alterar a distância em até $n/2$ vezes. Como temos no máximo $m - n$ arcos que podem se tornarem críticos para alterar a distância de um vértice, temos que existirão no máximo $O(nm)$ caminhos aumentantes. Como a busca por caminhos aumentantes é por busca em largura ou profundidade, que tem o tempo de $O(m)$, temos que o tempo total do algoritmo é de $O(nm^2)$.

3.1.3 Algoritmo de Dinic

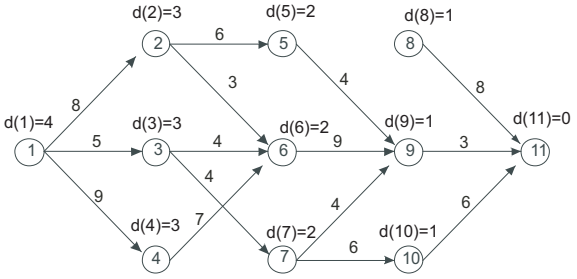
Apresentado por Dinic, o método de construir redes em camadas também usa o conceito de distância. Enquanto que o algoritmo de Edmonds-Karp realiza uma busca em

largura para encontrar o caminho mais curto a cada iteração, o algoritmo de Dinic consiste em construir uma rede apenas com caminhos mais curtos possíveis no momento, o qual é chamado de rede em camadas, e enviar fluxo até que não seja mais possível enviar, para então construir uma rede N' de caminhos mais curtos. O algoritmo de Dinic não cria arcos reversos em N', como acontece com o algoritmo de Edmonds-Karp. O algoritmo alterna entre a construção da rede e envio do fluxo até que a fonte não esteja mais na rede em camada.

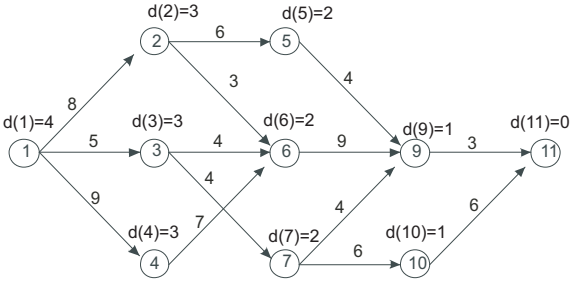
Dada uma rede $N = (V, A, s, t, c)$ de entrada, temos inicialmente N', uma rede em camada, que será uma cópia de N. A partir de uma busca em largura em N', o algoritmo calcula a exata distância entre o sorvedouro e os demais vértices (fig. 10a). Posteriormente são eliminados de N' os arcos não admissíveis (fig. 10b). A seguir verifica-se quais vértices não fazem parte de algum caminho entre s e t e os elimina de N'(fig. 10c). Neste momento G' torna-se de fato a rede em camadas, onde cada vértice pode ser agrupado em um conjunto onde a função de distância os rotula com a exata distância do grupo até o sorvedouro. Calculada a rede em camadas, o algoritmo começa a enviar fluxos da fonte em direção ao sorvedouro.



(a) Grafo inicial



(b) Arcos Inadmissíveis são excluídos do grafo



(c) Rede em Camadas é gerada pela exclusão dos vértices que não são alcançados pela fonte

Figura 10: Rede em Camadas

Algoritmo 3 Algoritmo de Dinic

Entrada: Rede N
Saída: Fluxo x e a Rede N com x de fluxo o atravessando

```
x ← 0
Calcule a rede em camadas
Enquanto For possível calcular distância para s faça
  i ← s
  Enquanto s ≠ bloqueado faça
    Se i possui arco admissível então
      avancar(i)
      Se i = t então
        aumentar_fluxo()
        i ← s
      fim Se
    Senão
      recuar(i)
    fim Se
  fim Enquanto
  Atualize a rede residual conforme a rede em camadas;
  Calcule a nova rede em camadas a partir da nova rede residual
fim Enquanto
```

1 $avancar(i)$

```
selecionar um arco (i, j) admissível
pred(j) ← i
i ← j
```

2 $recuar(i)$

```
estado[i] ← bloqueado
Se i ≠ s então
  i ← pred(i)
fim Se
```

3 $aumentar\_fluxo()$

```
identificar a partir dos índices de precedência o caminho aumentante
y ← min{r(i, j), dentre os arcos do caminho identificado}
x ← x + y
atualiza a rede residual conforme o aumento de fluxo
```

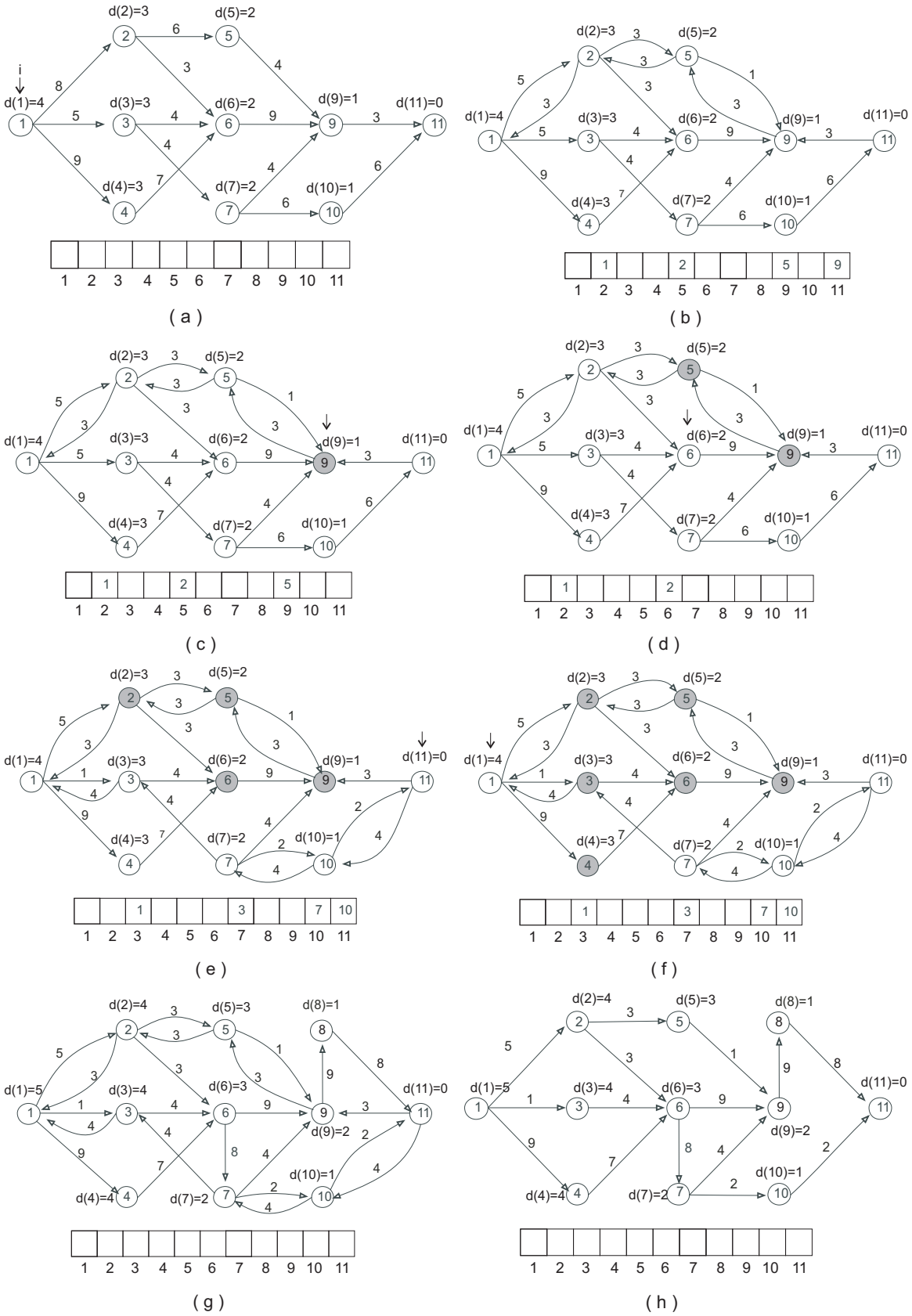


Figura 11: Algoritmo de Dinic: Os vértices sombreados representam vértices bloqueados

O primeiro passo deste algoritmo é criar a rede em camadas (Fig.10). A partir deste momento, é semelhante ao Algoritmo de Caminhos Aumentantes. A variável i é apontada para o vértice s (Fig. 11a). Então é realizada uma operação de avanço sobre um arco admissível. E assim segue-se até que se avance até o vértice t (Fig. 11b), ou até que se avance até um vértice que não possua arcos admissíveis (Fig.11c), onde neste momento este vértice é bloqueado e realiza um recuo para um precedente. Como neste algoritmo o arco (i, j) somente é admissível se j não é bloqueado, serão realizados tantos recuos quanto o necessário para encontrar um outro caminho, e a cada recuo é realizado outro bloqueio de vértice i corrente (Fig. 11d e Fig. 11e). No momento que o algoritmo encontra uma situação semelhante a figura 11f, onde não é mais possível enviar fluxo nesta rede em camadas, então recalcula-se uma nova rede em camadas com atualização da rede residual dos arcos dos caminhos aumentados na rede em camada (Figs. 11g e h).

Analisando o algoritmo para verificar a complexidade temos, o algoritmo faz busca em largura na rede, como já visto antes, em tempo de $O(m)$. A cada vez em que é calculada uma nova rede em camadas, a distância entre s e t deve aumentar em pelo menos uma unidade da rede calculada anteriormente. Essa distância pode aumentar no máximo n vezes. A cada nova rede em camadas, no máximo n vértices poderão ser bloqueados, o que significa que haverão no máximo n aumentos em N'. Portanto temos que o algoritmo roda no tempo $O(n^2m)$.

3.2 Método *Push-Relabel*

Em contraste com os algoritmos de caminhos aumentantes, o algoritmo de *Push-Relabel* [10] não encontra caminhos diretos entre s e t para enviar fluxo, mas envia o fluxo de vértice em vértice até chegar ao sorvedouro. Desta forma, a restrição de balanceamento de fluxo, $\sum_{j:(i,j)\in A} f(i,j) - \sum_{j:(j,i)\in A} f(j,i) = 0$ para todo $i \in N - \{s \text{ e } t\}$, pode não ser obedecida nas fases intermediárias, voltando a ser atendida no final.

Para este algoritmo, alguns novos conceitos são utilizados. Estes são:

Fluxo excedente do vértice *i e(i)* é o montante de fluxo recebido de um vértice que realizou uma operação *push*;

Nós Ativos são vértices i que possuam $e(i) > 0$;

Relabel operação de alteração do rótulo do vértice;

Push operação de envio de fluxo excedente através de um arco admissível para outro vértice.

O algoritmo começa saturando os arcos que saem do vértice s, ativando seus vizinhos. Posteriormente enquanto houver nós ativos, um nó ativo será selecionado e um empurrão de fluxo é realizado, enviando fluxo excedente para vértices vizinhos através de um arco admissível, ou *relabel* caso o nó ativo selecionado não possua arcos admissíveis. O algoritmo termina quando não houver mais nós ativos.

Algoritmo 3 Algoritmo Push-relabel

Entrada: Rede N
Saída: Fluxo x e a Rede N com x de fluxo o atravessando
Preprocessamento
Enquanto Existir nós ativos **faça**
Selecionar um nó ativo i
Se vértice i possui arco admissível (i, j) **então**
$push(i)$;
Senão
$relabel(i)$;
fim Se
fim Enquanto

1 *Preprocessamento*

$e(j) \leftarrow 0$, para todo $j \in V$;
Calcular a exata distância $d(i)$ em relação a t;
Para todo arco $(s, j) \in A$ **faça**
$y \leftarrow c(s, j)$
$e(j) \leftarrow e(j) + y$
$e(s) \leftarrow e(s) - y$
fim Para
$d(s) \leftarrow n$;

2 $push(i)$

$y \leftarrow min\{e(i), r(i, j)\}$;
$e(j) \leftarrow e(j) + y$;
$e(i) \leftarrow e(i) - y$;

3 $relabel(i)$

$d(i) \leftarrow min\{d(j) + 1 : (i, j) \in A(i) \text{ e } r_{ij} > 0\}$;

O Algoritmo *Push-relabel* inicia fazendo um pré-processamento sobre o grafo, que consiste em calcular as distâncias $d(i)$ de cada vértice até o sorvedouro. Atribui ao rótulo da

fonte a quantidade de vértices do grafo, e os vértices vizinhos da fonte recebem incremento no fluxo excedente no valor da capacidade do arco, ativando-os, ao passo que a fonte tem o fluxo excedente subtraído na mesma proporção (fig. 12b). Então é selecionado um vértice ativo e é verificado se existe pelo menos um arco admissível. Existindo, é selecionado um para realizar um *push* para o vizinho na outra ponta do arco, como na figura 12c, enviando fluxo excedente do vértice 2 para o vértice 5. Neste caso está sendo realizado um *push* saturante sobre o arco (2,5). Então seleciona-se novamente um vértice ativo, no caso o vértice 2, verificando novamente se possui um arco admissível (fig. 12d) e realiza um *push* insaturante sobre o arco (2,6), tornando o vértice 2 inativo. E continua, selecionando o vértice 4, realizando um *push* saturante sobre o arco (4,6), mas continuando ativo. Novamente é selecionado o vértice 4, mas desta vez não possui arcos admissíveis, portanto é realizado um *relabel* do vértice 4 (fig.12e). Após alguns *pushs* e *relabels* o grafo assume a forma igual ao da Figura 12f. Note que o algoritmo já encontrou o fluxo máximo, mas ele continua até que não exista mais nós ativos.

Analisando a complexidade da implementação genérica deste algoritmo, é preciso levar em conta a quantidade de *relabel* e de *pushs*. Observando isso podemos tirar algumas conclusões.

Lema 3.1 *Cada vértice $i \in V$, $d(i) < 2n$.*

Prova: Considere o menor caminho p onde o vértice i alcança o vértice fonte na rede residual. Inicialmente o vértice s é rotulado com n para a distância de t. Considere ainda que i não pode alcançar t na rede residual. O tamanho de p estará limitado à quantidade de vértices do conjunto $V(N)$ que é n, ou seja, $|p| < n$. Portanto $d(i)$ estará limitado a $d(s) + |p|$, ou seja, $d(i) < 2n$.□

Sendo que a cada vez que o algoritmo altera a distância do vértice i, $d(i)$ aumenta em pelo menos 1 unidade. Temos a seguinte conclusão.

Lema 3.2 *Cada rótulo de distância aumenta no máximo $2n$ vezes. Consequentemente o número de operação de relabel é no máximo $2n^2$.*

Prova: Aumentando pelo menos uma unidade por incremento, temos a partir do Lema 3.1 que cada vértice realiza no máximo $2n$ *relabels*, e como há n vértices, o número de operações de *relabel* é no máximo $2n^2$.□

Lema 3.3 *O algoritmo realiza no máximo nm operações de push saturante.*

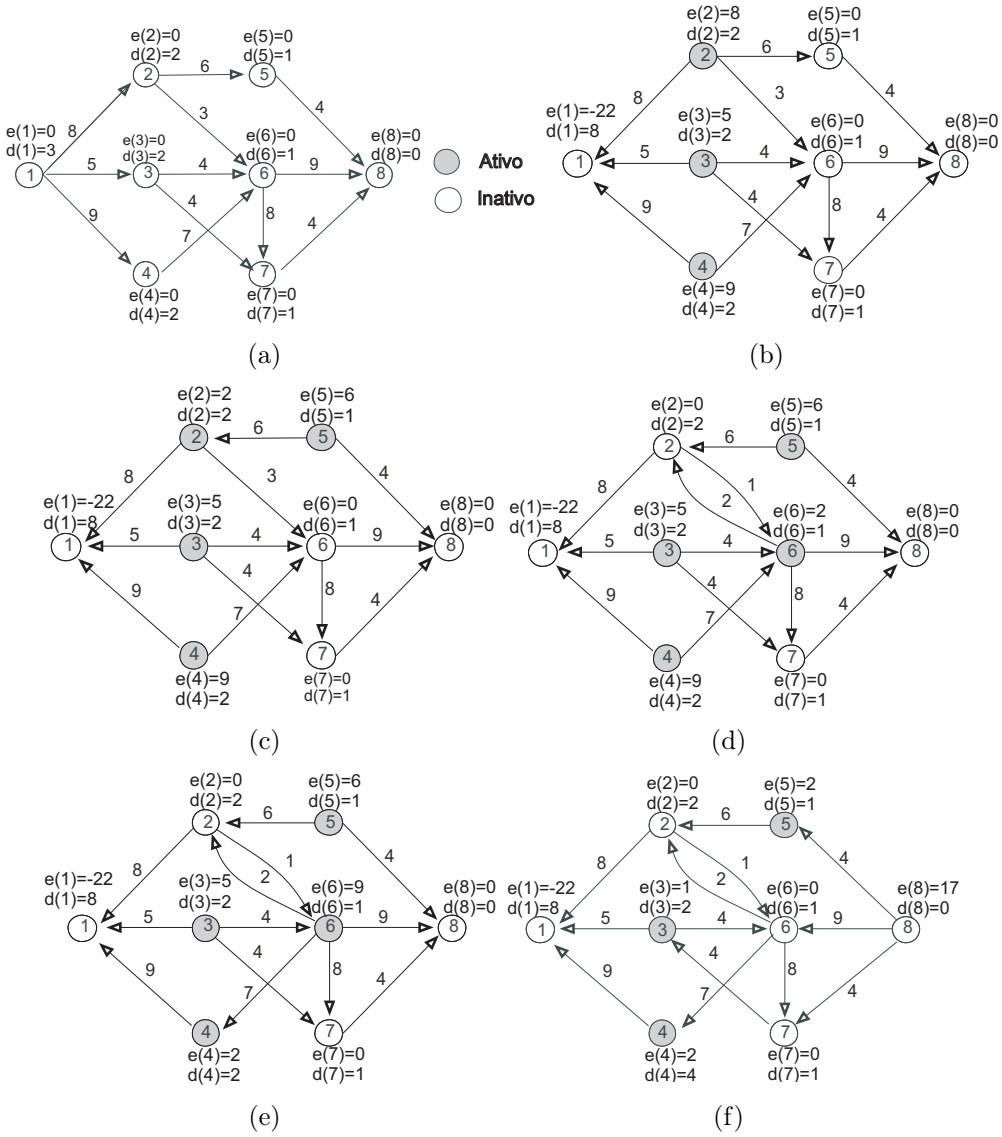


Figura 12: Algoritmo Push-Relabel

Prova: Para um arco $(i, j) \in A$, considere um empurrão saturante de i para j. Após um *push* saturante, $r(i, j) = 0$, e outro *push* não pode ocorrer até que $d(j)$ aumente em pelo menos duas unidades.Ocorrendo um *push* saturante em (j, i) após o aumento de $d(j)$, para ocorrer outro empurrão saturante em (i, j), $d(i)$ deverá também aumentar em pelo menos duas unidades. Pelo Lema 3.2, a quantidade de vezes que a distância dos vértices podem aumentar é no máximo $2n$, que neste caso aumentando de duas em duas é $2n/2$, ou seja n vezes. Como existem m arcos que podem sofrer empurrões saturantes temos que o número máximo é de nm empurrões saturantes.□

Falta agora contar o número de empurrões insaturantes.

Lema 3.4 *O número de empurrões não saturantes é no máximo* $4n^2 + 2n^2m$.

Prova: Para esta prova utilizaremos a **função potencial** que é definida como $\Phi = \sum_{i \in I} d(i)$ onde I é o conjunto de vértices ativos. Inicialmente, após a fase de preprocessamento $\Phi < 2n^2$ e o valor pode mudar após cada elevação de distância, empurrão saturante e empurrão insaturante. No término do algoritmo $\Phi = 0$. Durante as operações de *push* e *relabel* dois casos podem acontecer.

Caso 1 O algoritmo não consegue encontrar um arco admissível no qual possa enviar fluxo a um vizinho. Neste caso, a distância do vértice i deve aumentar em α unidades onde $\alpha \geq 1$. Esta operação aumenta Φ em no máximo α unidades. Sendo que o total de aumentos que $d(i)$ sofre através da execução do algoritmo é limita a $2n$, o total de incremento de Φ devido ao aumento dos rótulo de distância está limitada a $2n^2$.

Caso 2 O algoritmo consegue identificar um arco no qual pode enviar fluxo a um vizinho realizando um *push* saturante ou insaturante. Um push saturante sobre o arco(i, j) pode gerar um novo vértice com excedente positivo de fluxo o que aumenta o número de vértices ativos em 1 e incrementa Φ em $d(j)$, que pode ser até $2n$ por *push* saturante, o que significa $2n^2m$ para todos os pushs saturantes. Por outro lado, um *push* insaturante sobre (i, j) não aumenta $|I|$. O *push* insaturante vai decrementar Φ em $d(i)$, tornando i inativo, ao mesmo tempo que incrementa Φ em $d(j) = d(i) - 1$ se o *push* ativar j, sendo o total de decremento de Φ de pelo menos 1.

Totalizando Φ temos então no máximo $2n^2$ inicial e durante o processamentos pode haver um incremento de mais $2n^2 + 2n^2m$. Como cada *push* insaturante decrementa Φ em pelo menos uma unidade e ao término do algoritmo $\Phi = 0$, portanto haverá no máximo $4n^2 + 2n^2m$ pushs insaturantes.□

Sumarizando a complexidade temos que em sua implementação mais genérica, o tempo requerido para a execução do algoritmo é de $O(n^2m)$. Mas alterando a forma em que se escolhe a ordem dos vértices que sofrerão *push-relabel*, pode-se alterar o tempo de execução. Processando os vértices ativos em uma ordem FIFO, a complexidade do algoritmo passa para $O(n^3)$. Enquanto que processando primeiramente os vértices ativos com maior distância, passa-se a ter uma complexidade de $O(n^2\sqrt{m})$.

4 *Algoritmos para o Problema do Fluxo Máximo em Redes para Computação Paralela*

4.1 Algoritmo de Anderson-Setubal

O algoritmo de Anderson-Setubal [2] utiliza um modelo de paralelismo de memória compartilhada. Esse algoritmo apresenta uma heurística de rotulação global que possibilita bons resultados práticos. Os *speed-ups* alcançados foram de 4.1 a 7.2 com 16 processadores em vários tipos de grafos. Este algoritmo é baseado no algoritmo sequencial de *push-relabel*.

Relembrando o algoritmo de *push-relabel*, ele encontra o fluxo máximo enviando certas quantidades de fluxos de vértice em vértice, da fonte em direção ao sorvedouro. O fluxo é enviado de um vértice a outro através de arcos admissíveis. Em determinados momentos da execução do *push-relabel*, as operações de *push* não podem enviar fluxos excedentes para um vértice adiante, em direção ao sorvedouro, e precisam voltar alguns vértices para irem para outra seção do grafo, ou voltar à fonte. Tendo esse contexto em mente, o algoritmo de Anderson-Setubal propõe que a execução do algoritmo seja parada em um determinado momento para recalcular os valores dos rótulos dos vértices da rede residual daquele instante. Desta forma pode melhorar o tempo de execução do algoritmo, evitando muitas idas e vindas de fluxo que estão excedentes no grafo. Essa heurística foi definida como rotulação global.

A rotulação global é realizada em dois passos: Primeiro faz-se a busca em largura a partir do sorvedouro e atribui-se ao valor do rótulo de cada vértice a distância em arcos dele até o sorvedouro, depois é realizado a busca em largura a partir da fonte, atribuindo ao rótulo de cada vértice não alcançado pelo passo anterior o valor da distância em arcos dele até a fonte, acrescido em n. O tempo de execução da rotulação global torna proibitiva sua execução com muita frequência. Portanto para melhorar o tempo total do *push-relabel*,

foi sugerido que a rotulação global iniciaria a cada $2n$ *pushs*. Essa estratégia foi proposta para resolução do problema em uma máquina de memória compartilhada, mas ela pode também melhorar o desempenho de algoritmos sequenciais.

A execução do *push-relabel* com a estratégia da rotulação global precisa exercer um controle cuidadoso na execução paralela das operações. Ou seja, para ser realizada a operação *relabel* ou a rotulação global, é preciso que o processador p realize um bloqueio do vértice no qual a operação será realizada. Caso seja realizado um *push*, o processador precisa bloquear os dois vértices pontas do arco em que o fluxo será empurrado. O processador p deve escolher se realizará uma rotulação global ou um *push-relabel* em um vértice i. O bloqueio é uma instrução atômica para garantir a exclusividade de manipulação de um trecho de memória.

Apenas com o bloqueio descrito acima em uma execução concorrente pode ocorrer erros na rotulação. Os vértices das extremidades do arco em que ocorre um *push* podem ter sofridos quantidades diferentes de rotulações globais, permitindo que o fluxo tome um caminho incorreto, possibilitando que o algoritmo termine antes de encontrar o fluxo máximo. Como no exemplo da figura 13, o processador P_i está fazendo *global relabel* na direção G_1. O processador P_j está enviando fluxo f pelo caminho C_1(Fig. 13A). P_i pode criar um caminho alternativo C_2 que leve o fluxo f até a fonte(13C). Para resolver esse problema, o conceito de onda foi criado. Onda significa quantas vezes a rotulação global foi realizado em um vértice. Cada vértice possui um valor local para identificar a onda na qual está, e existe também um valor global para a onda corrente. Com isso, cada operação de *push* apenas poderá ocorrer quando os vértices pontas de um arco estiverem na mesma onda, ou seja, possuírem os valores locais de onda iguais.

Os processadores modernos têm associados certa quantidade de memória *cache* para busca rápida de dados na memória. Se o processador tem parte da estrutura de dados compartilhada guardado nesta *cache* e o atualiza, outro processador que também tem os mesmo dados em sua *cache* é invalidado e precisa obter uma nova cópia atualizada desses dados. Isso pode causar invalidação desnecessária de dados não compartilhados que ocupam o mesmo bloco que foi invalidado, gastando tempo na recuperação destes dados. Considerando que a atualização de vértices ativos é relativamente frequente em vários processadores, isso pode se tornar um problema. Para evitar isso, pode-se usar a estratégia de separar uma parte da memória principal como uma *cache* local a cada processador. Nesta *cache* estarão alocadas duas filas de vértices, *in-queue* e *out-queue*, enquanto no restante da memória compartilhada ainda existirá uma fila global de vértices. O *in-queue*

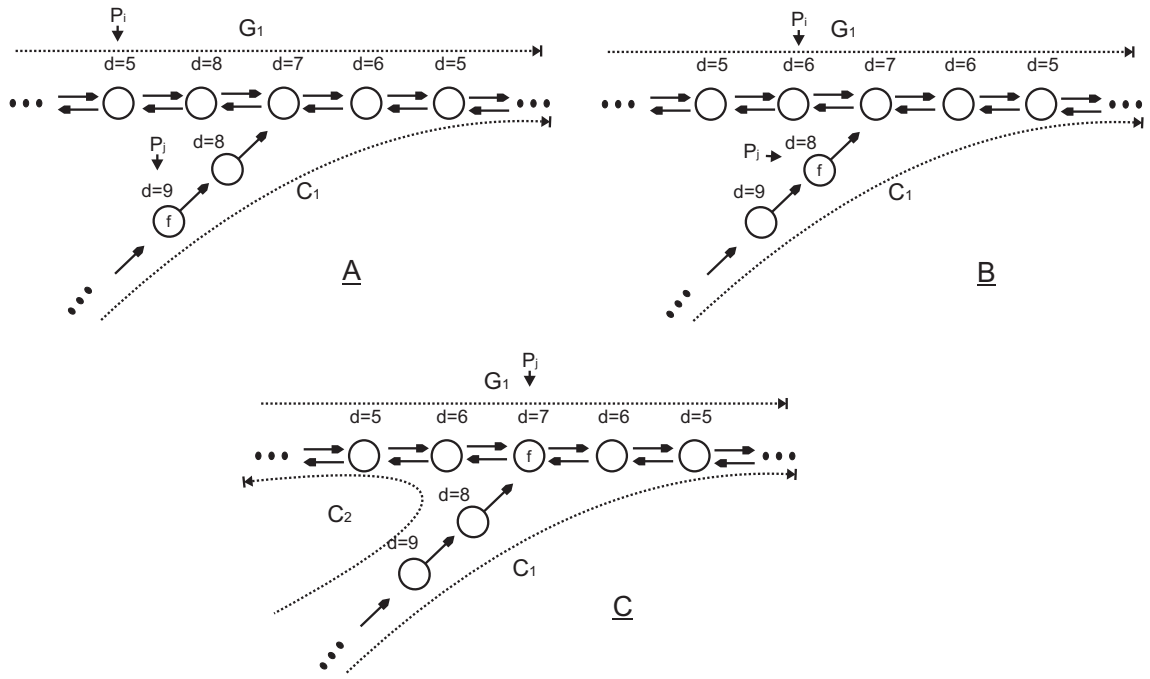


Figura 13: Rotulação Global vai interferir no envio de fluxo

de cada processador é alimentado com os primeiros b vértices da fila global, enquanto o *out-queue* é alimentado com os vértices ativados pelas operações de *push* realizados no processador. Toda vez que acaba os vértices do *in-queue*, este é realimentado com outros b vértices. O *out-queue* é "esvaziado"mandando seus vértices para a fila global, quando atinge a quantidade de b vértices. Para a operação de esvaziar o *out-queue* é preciso tomar o cuidado de verificar se o vértice já esta na fila global ou não, afim de evitar duplicidade dos vértices. Para isso, o processador tem de bloquear a fila global para garantir que nenhum processador carregue dados incorretos dos vértices. Para carregar os vértices no *in-queue*, o processador vai bloquear a fila global, garantindo que somente ele vai carregar aquele conjunto de b vértices.

Nos testes realizado pelos autores dos estudos foram usados um Sequent Symmetry S81 com 20 processadores 80386 de 16Mhz e 32 MB de memória, rodando DYNIX 3.0. Nesta máquina a largura do barramento de comunicação entre o processador e a memória é de 64 bits. Cada processador tem 64 KB de memória *cache*. Com as entradas testadas foi observado que o algoritmo rodando uma instância sequencialmente completa a tarefa em 74.9 segundos, com dezesseis processadores esse tempo cai para 10.4 segundos. Um *speed-up* de 7.2. Em outra instância o tempo sequencial foi de 176.4 segundos enquanto que a execução paralela completou em 43.3 segundos, com um *speed-up* de 4.1. mais detalhes da implementação e resultado podem ser vistos em [2].

Algoritmo 1 $Push - Relabel_i$

$npushs \leftarrow 0$
Enquanto Houver vértices ativos **faça**
  Enquanto $inqueue \neq \emptyset$ e $outqueue \neq$ FULL **faça**
    $v \leftarrow$ primeiro vértice do $inqueue$;
    Se v possui arco admissível (v, w) **então**
      $Push_i(v, w)$;
      $npushs \leftarrow npushs + 1$
      Se $npushs$ é igual a duas vezes o números de de vértices **então**
        ativar o global Relabel
        $npushs \leftarrow 0$
      fim Se
    Senão
      $Relabel_i(v)$
    fim Se
  fim Enquanto
  Se $inqueue = \emptyset$ **então**
    $inqueue$ recebe parte da fila de ativos globais;
  fim Se
  Se $outqueue = FULL$ **então**
    $outqueue$ é concatenada ao final da fila de ativos globais;
  fim Se
fim Enquanto

2 $Push_i(v, w)$

Se $wave(v) = wave(w)$ **então**
  Processador i bloqueia os vértices v e w
  $y \leftarrow min\{e(v), r(v, w)\}$;
  $e(w) \leftarrow e(w) + y$;
  $e(v) \leftarrow e(v) - y$;
fim Se

3 $Relabel_i(v)$

Processador i bloqueia o vértice v
$y \leftarrow min\{d(w) + 1 | (v, w) \in A_f\}$;
Se $y > d(v)$ **então**
  $d(v) \leftarrow y$;
fim Se

4 *Global* $Relabel_i(v)$

Se $wave(v) < CurrentWave$ **então**
  Processador i bloqueia o vértice v
  Se $d(v) < CurrentLevel$ **então**
    $d(v) \leftarrow CurrentLevel\}$;
    $wave(v) \leftarrow CurrentWave$;
  fim Se
fim Se

4.2 Algoritmo Autoestabilizante para o Problema de Fluxo Máximo

O Algoritmo autoestabilizante para Fluxo Máximo é uma abordagem baseada nos algoritmos de caminhos aumentantes e *push-relabel*. Este algoritmo descrito por Gosh *et al.*([11]) trata o problema em um modelo de computação paralela de memória distribuída de granulosidade fina, onde o problema é resolvido em grafos dirigidos acíclicos em uma rede tolerante a falhas transientes, ou seja, falhas passageiras que impedem a comunicação dos nós por um breve período de tempo.

O modelo computacional descrito define que para cada vértice $i \in V(N) \setminus \{s\}$ de uma rede N há um processador chamado de processador i que executa o programa assincronamente. Cada arco $(i, j) \in A(N)$ corresponde a um *link* físico bidirecional entre os processadores i e j. Dessa forma a rede de processos distribuídos tem a mesma forma da rede N. Cada processador i tem um número fixo de variáveis locais. Essas variáveis podem ser lidas pelo processador i e por seus vizinhos, mas só podem ser escritas por i. Todos os processadores executam programas idênticos, exceto o processador s que é sempre inativo. Cada processador executa certas ações A_k, $1 \leq k \leq 4$, que para serem ativadas aguardam que determinadas condições sejam observadas nas respectivas funções de guarda F_k. Cada função de guarda F_k é uma função booleana cujo valor é definido a partir das variáveis locais de um processador ou de seus vizinhos em um estado S_k. Em cada ação A_k o processador atualiza algumas variáveis locais. A ordem de execução das ações é arbitrária, sendo executadas apenas quando a função de guarda correspondente assuma o valor verdadeiro.

A partir deste momento assumiremos as definições a seguir. Considere uma rede $N = (V, A, s, t, c)$, onde V é o conjunto de vértices de N e A é o conjunto de arcos do mesmo grafo, c é a função de capacidade atribuído a cada arco pertencente a A e s e t são os vértices fonte e sorvedouro. Sendo um par de vértices $\{i, j\} \in V(N)$ e um fluxo f na rede N, de acordo com a definição no capítulo 2, temos que a capacidade residual $r(i, j)$ é igual a diferença $c(i, j) - f(i, j)$, onde $c(i, j)$ e a capacidade máxima do arco (i, j) se $(i, j) \in A(N)$ ou 0 caso contrario, e $f(i, j)$ é o fluxo que atravessa esse mesmo arco. Para algum fluxo f em N, o grafo residual de $G_f = (V_f, A_f)$, onde $V_f = V(N)$ e $(i, j) \in A_f$ se e somente se $r(i, j) > 0$, e a cada arco (i, j) associamos um valor positivo de $r(i, j)$. É válido observar que se a rede N é acíclica, o grafo residual resultante, G_f, pode ainda assim ter arcos paralelos entre pares de vértices.

Para efeito de simplicidade, assumimos que para cada arco (i, j) temos uma variável $f(i, j)$ que armazena o fluxo que o percorre na direção de i para j, onde os processadores i e j podem ler e escrever. Não se trata de descumprir o modelo computacional, pois $f(i, j)$ pode ser o resultado da diferença de duas variáveis, uma em cada processador. Ademais para cada processador temos a variável $d(i)$ que é o valor que o processador i "acredita"ser o tamanho do menor caminho de s a i em G_f.

De uma forma geral, o algoritmo realiza em cada processador a execução de ações para atualizar $f(i, j)$ e $d(i)$, sendo três tipos que atualizam $f(i, j)$ e um que atualiza $d(i)$.

Para cada nó $i \in V_f(G_f) \setminus \{s, t\}$ temos a $demanda(i) = O_f(i) - I_f(i)$, onde $O_f(i) = \sum f(i, j)$ e $I_f(i) = \sum f(j, i)$. O processador t se comporta diferentemente dos demais processadores uma vez que ao longo da execução, teremos $demanda(t) = \infty$. Em cada nó $i \neq s$, tenta-se reestabelecer a restrição de conservação de fluxo onde $demanda(i) = 0$, seja reduzindo o fluxo entrante caso $demanda(i) < 0$, ou incrementando o fluxo entrante ou reduzindo o fluxo que sai do vértices se $demanda(i) > 0$. De forma simplificada, cada vértice com demanda positiva tenta puxar fluxo através de um caminho mais curto vindo de s até ele no grafo residual, ou caso o vértice "acredite"não haver mais um caminho de s a até ele, então ele empurra fluxo excedente através de um arco que sai do vértice.

Neste algoritmo temos quatro funções de guarda com suas respectivas ações associadas:

Função de Guarda S1: Cada vértice i calcula sua distância $d(i)$ examinando as distâncias de seus vizinhos $d(j)$ no grafo residual para todo $(j, i) \in A_f$, escolhendo aquele de menor valor de distância. Se o vértice escolhido k tem o valor de distância menor que o número de vértices do grafo então $d(i) = d(k) + 1$, caso contrário $d(i)$ receberá como valor o número de vértices do grafo, que significa que i acredita não haver mais um caminho entre s e i no grafo residual. Por notação definimos que $IN(i) = \{j|(j, i) \in A_f\}$ e $D(i) = min\{d(p) + 1|p \in IN(i)\}$.

Função de Guarda S2: Para qualquer vértice i, onde $i \neq s$, se temos $demanda(i) < 0$, então diminui-se o valor do fluxo entrante em i no montante do valor absoluto de $demanda(i)$, independentemente do valor das distância do vértice predecessor.

Função de Guarda S3: Para qualquer vértice i, onde $i \neq s$, se temos $demanda(i) > 0$ e $d(i) < n$, sendo n o número total de vértices na rede N, então i tenta buscar fluxo através de algum arco fornecedor $(j, i) \in A_f$ no qual aparente pertencer ao menor caminho de s a i no grafo residual G_f. Ou seja, identifica-se um arco $(j, i) \in A_f$ onde $d(j) = d(i) - 1$,

incrementa-se o fluxo através desse arco no valor referente ao mínimo entre a $demanda(i)$ e $r(j,i)$. Resumindo através do predicado $pull(j,i)$ as condições de incremento de fluxo em (j,i) temos: $pull(j,i) = (demanda(i) > 0) \bigwedge (d(i) < n) \bigwedge (d(j) = d(i) - 1)$.

Função de Guarda S4: Para qualquer vértice i, onde $i \neq s$, se temos $demanda(i) > 0$ e $d(i) = n$, acredita-se que não há caminho de s a i em G_f e portanto a demanda excedente não pode ser atendida, sendo necessário diminuir o fluxo que sai do vértice. Resumindo através do predicado $push(i)$ as condições de decremento de fluxo em um arco que sai do vértice de i temos: $push(i) = (demanda(i) > 0) \bigwedge (d(i) = n) \bigwedge (i \neq t)$.

Em resumo, temos as seguintes definições:

- $demanda(i) = O_f(i) - I_f(i)$
- $IN(i) = \{j | (j,i) \in A_f\}$
- $D(i) = \{d(p) + 1 | p \in IN(i)\}$
- $pull(j,i) = (demanda(i) > 0) \wedge (d(i) < n) \wedge (d(j) = d(i) - 1)$
- $push(i) = (demanda(i) > 0) \wedge (d(i) = n) \wedge (i \neq t)$.

Para cada estado temos as seguintes funções de guarda e suas respectivas ações:

S1 $d(i) \neq min(D(i) \cup n) \longrightarrow d(i) := min(D(i) \cup n)$;

S2 $demanda(i) < 0 \longrightarrow Reduce\_InFlow(i)$;

S3 $\exists j \in IN(i) : pull(j,i) \longrightarrow f(j,i) := f(j,i) + min(demanda(i), r(j,i))$;

S4 $push(i) \longrightarrow Reduce\_Outflow(i)$;

E os seguintes procedimentos:

1 $Reduce\_InFlow(i)$

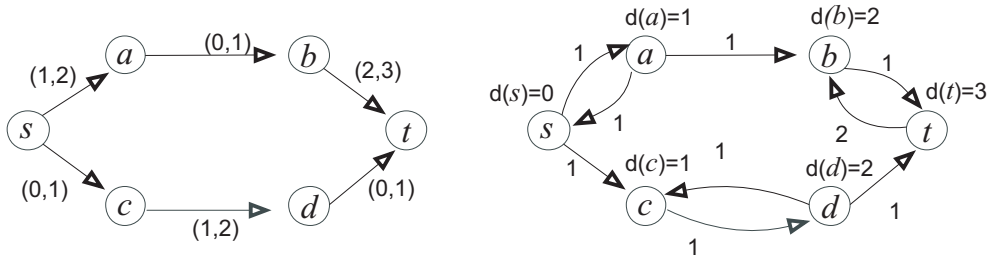
Encontre $(k,i) \in E$ dado que $f(k,i) > 0$;
$f(k,i) := f(k,i) - min(-demanda(i), f(k,i))$;

2 $Reduce\_Outflow(i)$

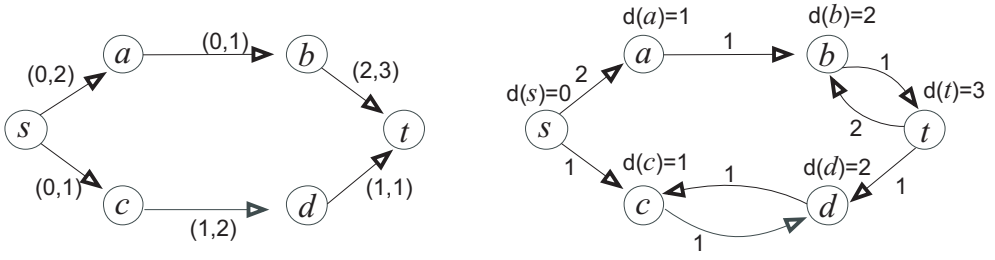
Encontre $(i,k) \in E$ dado que $f(i,k) > 0$;
$f(i,k) := f(i,k) - min(demanda(i), f(i,k))$;

É válido lembrar que o processador s fica sempre inativo e o processador t executa o mesmo algoritmo que os demais processadores, contudo, o valor de $O_f(t)$ é fixado em ∞.

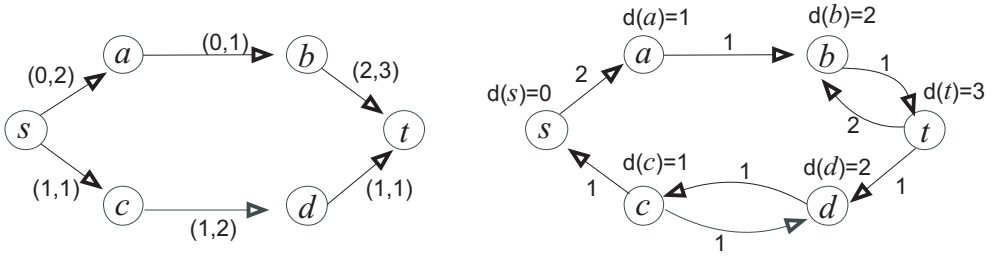
Nas Figuras 14 e 15 temos um exemplo da execução do algoritmo. Temos na Figura 14a um estado inicial arbitrário com fluxo inicial $f(s,a) = 1$, $f(a,b) = 0$, $f(b,t) = 2$, $f(s,c) = 0$, $f(c,d) = 1$, $f(d,t) = 0$. Começando pelo vértice t estando no estado $S3$, puxa uma unidade de fluxo do vértice d. Com o vértice a está no estado $S2$, a rejeita o fluxo vindo de s, resultando desses movimentos o grafo da Figura 14b. O vértice c possui demanda positiva, sendo alcançável por s, este puxará fluxo de s, veja Figura (14c). Após esse movimento, c está em estado $S1$, com a distância ao qual estava de s no grafo residual diferente de antes, necessitando atualizar a distância, mas isso fará com que d também fique em estado $S1$. Dessa forma, após algumas atualizações, as distâncias de c e d assumem respectivamente 5 e 4 como pode ser visto na Figura 15a. A seguir $S3$ passa a ser observado em b, fazendo com que o vértice puxe fluxo de a, como pode ser visto na Figura 14e. Dessa forma tornam-se inatingíveis por s os vértices b, c, d e t, ou seja entrando em estado $S1$, tendo que realizar vários movimentos de alteração de distância até atingirem o valor n, como pode ser visto na Figura 15a. O vértice a puxa fluxo de s por estar em estado $S3$(15b) e por fim b reduz o fluxo que sai do vértice em direção a t por estar em $S4$, vide Figura 15c.



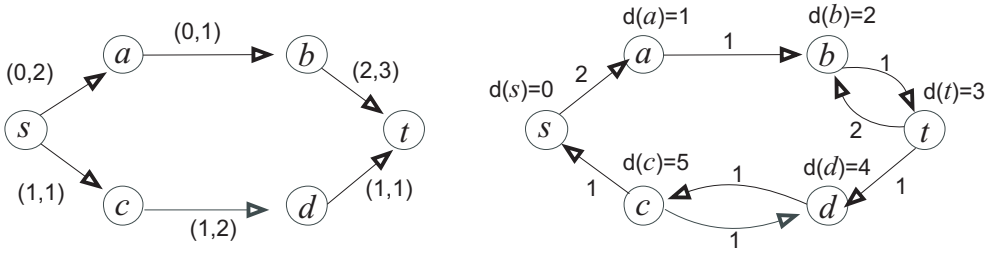
(a) Fluxo inicial $f(s,a) = 1$, $f(a,b) = 0$, $f(b,t) = 2$, $f(s,c) = 0$, $f(c,d) = 1$, $f(d,t) = 0$



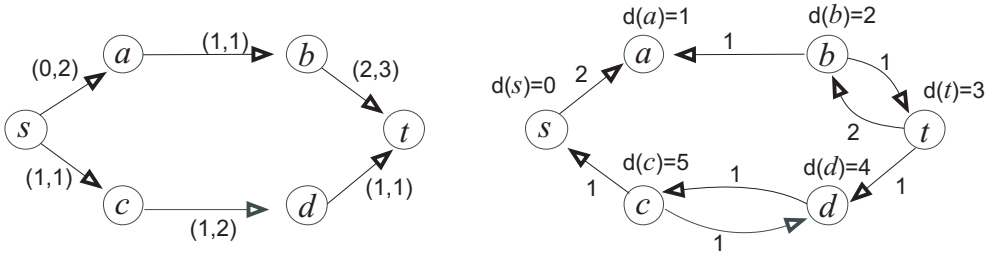
(b) S3 em t e S2 em a



(c) S3 em c

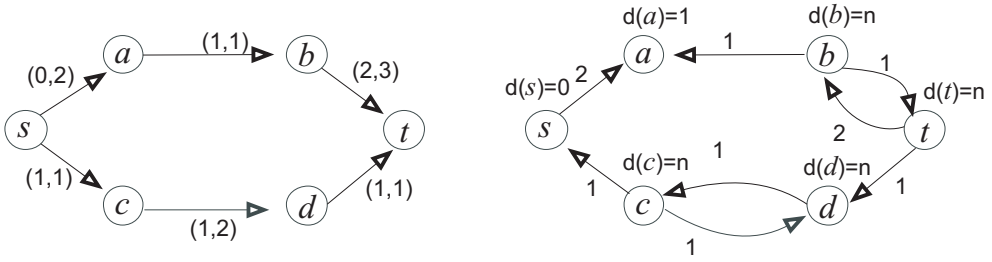


(d) S1 em c e d

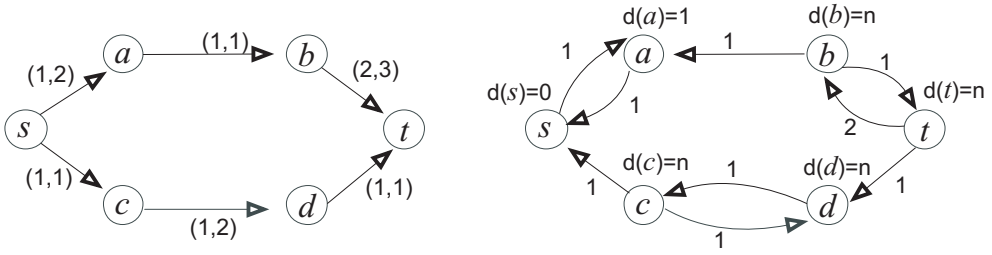


(e) S3 em b

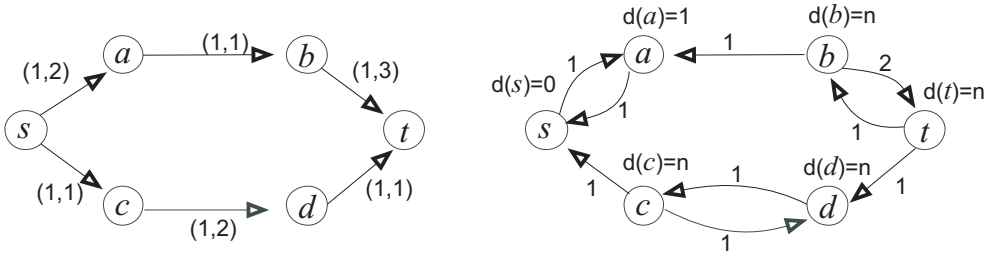
Figura 14: Algoritmo Autoestabilizante



(a) S1 em b, t, c e d



(b) S3 em a



(c) S4 em b

Figura 15: Algoritmo Autoestabilizante

Para verificarmos que o algoritmo está correto, dividimos a prova em duas partes. Na primeira parte mostraremos que quando a rede está em um estado final, então os valores de fluxo nos arcos constituem um fluxo máximo. Na segunda parte mostraremos que a quantidade de movimentos é finita, ou seja, que o algoritmo termina.

Definimos como um estado do algoritmo, um conjunto de valores de fluxos atribuídos nos arcos e de distâncias atribuídos aos vértices. O estado final é o estado onde todas as funções de guarda devolvam valor lógico falso. Para mostrar a correção parcial do algoritmo, é necessário mostrar que se o algoritmo está em um estado final então os valores de fluxo nos arcos constituem um fluxo máximo.

Antes de fazermos esta demonstração, precisamos da definição a seguir. Definimos que um grafo residual G_f está em estado desejável se:

$$demanda(i) = 0 \text{ para todo } i \in V \setminus \{s, t\} \tag{4.1}$$

Para todo $i \in V \setminus t$

$$d(i) = \begin{cases} \text{o comprimento de um caminho mínimo entre } s \text{ e } i \text{ caso exista um em } G_f \\ n \text{ caso contrário} \end{cases} \tag{4.2}$$

$$d(t) = n \tag{4.3}$$

Lema 4.1 *Se o algoritmo está em um estado desejável, então os valores de fluxo dos arcos constituem o fluxo máximo em N.*

Prova: Pela condição 4.1 temos que no estado desejável o fluxo em N é viável. Pelas condições 4.2 e 4.3 juntas, temos que não há mais caminhos aumentantes em G_f. Como já foi visto antes, os fluxos nos arcos pertencentes ao grafo residual, que já não possui mais algum caminho aumentante, constituem um fluxo máximo em N. □

Lema 4.2 *Se o algoritmo está em um estado final, então os valores dos fluxos nos arcos implicam que $demanda(i) = 0$ para todo vértice $i \in V \setminus \{s, t\}$.*

Prova: Consideremos que qualquer vértice $i \in V(N) \setminus \{s, t\}$. Se a função de guarda de $S2$ retorna falso, então $demanda(i) \geq 0$ para todo i. Se $d(i) = n$, desde que a função

de guarda de $S4$ retorne falso, teremos $demanda(i) \leq 0$. Consequentemente, para todos os vértices i com $d(i) = n$, temos $demanda(i) = 0$. Se $d(i) < n$, uma vez que $S3$ retorna falso, ou $demanda(i) \leq 0$ ou não existe $j \in IN(i)$ no qual $d(j) = d(i) - 1$. Mas, neste último caso $S1$ deve ser verdadeiro para todo vértice i. Como $S1$ é falso, teremos $demanda(i) \leq 0$ ainda que $d(i) < n$. Consequentemente, para todo vértice i com $d(i) < n$, $demanda(i) = 0$.□

Lema 4.3 *Se o algoritmo está em um estado final, então $d(i)$ é o valor do menor caminho em arcos desde s até i, caso exista algum em G_f. Caso não exista um caminho de s até i então $d(i) = n$.*

Prova: Vamos considerar que f seja um fluxo em uma rede quando todas as funções de guarda devolvem falso. Considere ainda que o fluxo na rede tem como resultante o grafo G_f. Dois casos são possíveis.

- No primeiro caso temos a inexistência de um caminho de s a i no grafo residual. Suponhamos que $d(i) \neq n$. Encontramos uma sequência maximal de vértices $r_0,\ r_1,\ ...,\ r_k$ de forma que $r_k = i,\ r_j \in IN(r_{j+1})$, e $d(r_j) = d(r_{j+1} - 1)$ para todo $0 \leq j \leq k$. Uma vez que i não é alcançável por um caminho a partir de s no grafo residual, nenhum vértice r_j, $0 \leq j \leq k$ são alcançáveis por s. Duas situações são possíveis:

  $\boldsymbol{IN(r_0) = \emptyset}$: Se $d(i) < n$ e $d(r_0) < d(i)$, temos $d(r_0) < n$. Portanto $S1$ é verdadeiro para o vértice r_0. Portanto temos uma contradição.

  $\boldsymbol{IN(r_0) \neq \emptyset}$: Seja q o vértice com o menor valor de $d(q)$ entre todos os vértices em $IN(r_0)$. Como a sequência $r_0,\ r_1,\ ...,\ r_k$ é maximal, $d(q) \neq d(r_0) - 1$. Portanto também uma contradição, pois $S1$ devolveria um valor lógico verdadeiro para r_0 pois $d(r_0) < n$.

  Portanto temos que qualquer vértice i não alcançável por s deve ter $d(i) = n$.

- No segundo caso temos i alcançável por s. Seja k o tamanho do menor caminho entre s e i. Por indução em k vamos mostrar que $d(i) = k$.

  Base: Tendo $k = 0$, é verdadeiro por definição, pois $d(s) = 0$.

  Passo de indução: Suponha que todos os vértices que estão a uma distância de s menor ou igual a l tenham o valor de distância estabelecido corretamente. Considere que o vértice i está a uma distância $l + 1$ de s. Assuma que $d(i) \neq l + 1$. Duas situações são possíveis.

$d(i) > l + 1$ Se existe algum vértice $j \in IN(i)$ com $d(j) = l$, por indução $S1$ vai devolver verdadeiro, sendo uma contradição.

$d(i) < l + 1$ Encontrando uma sequência maximal de vértices $R = r_0,\ r_1,\ ...,\ r_{k-1},\ r_k$ no qual $r_k = i$, $r_j \in IN(r_{j+1})$ e $d(r_j) = d(r_{j+1})-1$ para todo $0 \leq j < k$, teremos duas situações:

[$r_0 = s$] Neste caso existe um caminho de comprimento $d(i)$ de s até i. Uma vez que $d(i) < l+1$, portanto $d(i)$ está corretamente estabelecido, o que implica em uma contradição, pois por hipótese de indução, i está a uma distância de $l + 1$ de s.

[$r_0 \neq s$] Uma vez que $d(i) < l + 1$ e $d(r_0) < d(i)$, temos que $d(r_0) \neq n$. Então r_0 deve ser alcançável por s, caso contrario $d(r_0) = n$. Como a sequência R é maximal e r_0 é alcançável por s, temos uma contradição pois $S1$ devolveria verdadeiro.

Portanto, $d(i) = l + 1$. Com isso completamos o passo de indução que mostra que para todo vértice i alcançável por s, tendo o menor caminho de s a i de tamanho k, teremos $d(i) = k$.

□

Lema 4.4 *Se o algoritmo está em um estado final então $d(t) = n$.*

Prova: Suponha que $d(t) < n$. Uma vez que $S1$ devolve falso para t, então existe um $w \in IN(t)$ no qual $d(w) = d(t) - 1$. Uma vez que $O_f(t) = \infty$, temos que $S3$ devolve verdadeiro para t, o que é uma contradição. Portanto, $d(t) = n$.□

Os Lemas 4.2, 4.3 e 4.4 mostram que o algoritmo está em estado final, portanto está também em estado desejável. Além do mais, o lema 4.1 mostra que o algoritmo estando em estado desejável os valores dos fluxo nos arcos constituem o fluxo máximo em N. Portanto temos:

Teorema 4.1 *Quando o algoritmo atinge o estado final, então os valores dos fluxos nos arcos constituem o fluxo máximo em N.*

A quantidade de movimentos realizados pelo algoritmo é difícil de quantificar, uma vez que existe certa dependência dupla entre os tipos de movimentos. O estudo empírico

apresentado em [11] mostra que a quantidade de movimentos é da ordem de $O(n^2)$. A seguir daremos um esboço de que a quantidade movimentos é finito.

Se considerarmos que o algoritmo executa os movimentos em uma ordem arbitrária que terminam em tempos distintos, e colocarmos esses movimentos em uma sequência ordenada pelo momento de término, em certo momento a função de guarda $S2$ é executado pela última vez. Podemos garantir que não pode ocorrer mais execução de $S2$ após o momento x, pois $S3$ e $S4$ executam em vértices onde a demanda é positiva diminuindo-a de forma a zerá-la, mas nunca a torná-la negativa, como pode ser visto no algoritmo 4.2, enquanto que $S1$ altera apenas $d(i)$. A quantidade de vezes que $S2$ é executada é finita, pois em primeiro lugar, a quantidade de vértices com $demanda(i) < 0$ é finito uma vez que a quantidade de vértices em N também é, e a quantidade de vértices que podem vir a ter demanda negativa também é finito, por que cada vértice i com $demanda(i) < 0$ empurra fluxo em direção a s passando por um vértice j predecessor a ele num caminho que vai de s a i, podendo tornar $demanda(j) < 0$. Como o comprimento do caminho é limitado pela quantidade de vértices na rede N, e a quantidade de caminhos em que isso pode acontecer também é limitada, como já discutido por Ford-Fulkerson em [8], podemos concluir que o número de vezes que $S2$ tornará um vértice com demanda negativa é finita, onde podemos concluir que $S2$ é executado em um número finito de vezes.

A partir do momento x, quando $S2$ foi executado pela última vez, teremos uma sequência onde serão executados apenas as funções de guarda $S1$, $S3$ e $S4$. Neste momento o grafo residual possui vértices com $demnda(i) \neq 0$ que precisam aumentar o fluxo que entra ($S3$) ou diminuir o fluxo que sai ($S4$). Para os vértices do caso $S4$, estes já possuem o valor de distância $d(i) = n$, não executando mais $S1$. Para todo o vértice i com $demanda(i) > 0$ e $d(i) < n$, o algoritmo encontra capacidade residual nos arcos entrante em i suficiente para zerar sua demanda, mas deixando a demanda excedente para o predecessor de i no caminho de s a i, e sucessivamente até que se puxe fluxo de s.

Primeiro note que cada vértice i pode deixar a demanda positiva de um ou mais vizinhos j através de arcos $(j, i) \in G_f$, mas para efeito de simplicidade, vamos considerar que é tornado positivo apenas um nó por vez. Seguiremos o mesmo raciocínio da prova de término de $S2$, primeiro para o caso de $S3$. Para cada vértice i com $S3$ devolvendo verdadeiro, temos um caminho de s a i. Se todos os arcos que formam esse caminho tem capacidade maior que a $demanda(i)$, então os vértices desse caminho assumem sucessivamente os estados de ativação da função $S3$ na ordem do percurso de i a s. Se todos os casos forem assim, podemos concluir que a quantidade de execução de $S3$ é finita,

pois primeiro a quantidade de vértices com demanda positiva é menor que o número de vértices do grafo, portanto finito, e o número de caminho da fonte a esses vértices também é finito, conforme já visto nos parágrafos acima, e por fim a quantidade de vértices nesses caminhos também é limitado ao número de vértices no grafo. Concluindo, nessa situação $S3$ é executado em um número finito de vezes. Mas no caso de que algum arcos (k, l) pertencente ao caminho de s a i tenha capacidade menor que a $demanda(i)$, (para efeito de simplicidade está sendo considerado que os vértices do caminho de l a i não tenha outro arco de entrada além dos que formam o caminho) os vértices do caminho entre l e i passarão a "acreditar"que há sucessivas mudanças de comprimento de caminho até s e assumirão $d(v) = n$ para o v pertence ao caminho de l a i. Uma vez que $d(l) = n$ então $S4$ passa a devolver verdadeiro em l. E assim sucessivamente no caminho de l a i. Uma vez que $S3$ não é mais executado, não haverá mais mudanças das distâncias dos vértices, portanto se $S3$ é finito, $S1$ também será.

Uma vez que $S3$ não é mais executado, com o comportamento parecido com o dos vértices com demanda negativa, a execução de $S4$ é finita, pois a quantidade de vértices i com $demanda(i) > 0$ e $d(i) = n$ é finito e a quantidade de vértices nos caminhos de i a t também é. Como a ação de $S4$ não altera $d(i)$ do vértice i, $S4$ somente poderá gerar em outro vértice um estado em que apenas $S4$ devolverá verdadeiro.

Portanto, a partir dessas observações temos que:

Teorema 4.2 *Começando de um estado arbitrário, o algoritmo atinge o estado final em um número finito de movimentos.*

Uma demonstração detalhada pode ser encontrada em [11].

Os algoritmos apresentados neste capítulo não se adaptam diretamente ao modelo CGM, pois o primeiro usa multiprocessamento com memória compartilhada enquanto que o segundo trata o problema com processamento paralelo de granulosidade fina.

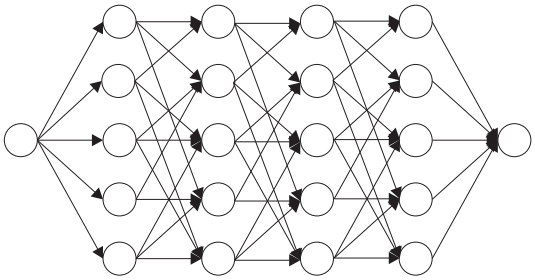
5 *Algoritmo para o Problema do Fluxo Máximo em Redes para modelo CGM*

O algoritmo paralelo BSP/CGM desenvolvido neste trabalho foi baseado no algoritmo autoestabilizante de Gosh *et al.* apresentado no capítulo 4. O algoritmo satura todos os arcos da rede e todos os vértices com demanda diferente de zero são colocados em uma fila. Cada processador retirará um vértice do início da fila atualizará a distância conforme informação local e estabilizará o vértice com a operação referente ao estado em que se encontre. As alterações de fluxo nos arcos cujos vértices pontas estão em processadores diferentes são informadas através de mensagens que esses processadores trocam entre si. O algoritmo terminará assim que não houver mais vértice com demanda diferente de zero nas filas dos processadores.

5.1 Classe de Grafos para Entrada

Nos algoritmos para o modelo de processamento BSP/CGM vamos utilizar algumas classes de grafos usadas por Anderson e Setubal em [2] e Gosh *et. al.* em [11]. Os grafos estão descritos a seguir:

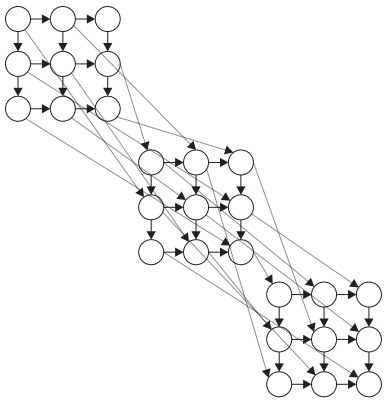
***Random Level Graphs*(RLG)** Os vértices estão dispostos em forma de *grid* retangular, onde cada vértice de cada linha está ligado por um arco a outros três vértices da linha seguinte. A fonte e o sorvedouro estão fora desse *grid*. A fonte tem um arco ligando a cada vértice da primeira linha do *grid*, tal como cada vértice da ultima linha do *grid* tem um arco ligando-os ao sorvedouro. Esses grafos podem ter instâncias largas, com número muito grande de vértices em cada linha com uma quantidade menor de colunas, ou compridas com quantidade de colunas muito maior do que a quantidade de vértices em cada linha. Na figura 16 ilustramos um exemplo de grafo RLG.



(a)

Figura 16: RLG

***Rmf Graphs*(RMF)** Também descrito em [2]. É composto por uma sequência de *frames* conectados entre si em sequência, onde cada *frame* nada mais é que um *grid* quadrado de $l \times l$ vértices. A fonte é um vértice do canto do primeiro *frame*, enquanto que o sorvedouro é o vértice do canto do ultimo *frame*. Cada vértice está conectado aos seus vizinhos no *grid* do *frame* e a um vértice escolhido aleatoriamente no *frame* seguinte. Esse grafo pode ter instâncias largas, onde o valor de l é grande em relação à quantidade de *frames*, ou compridas, onde a quantidade de *frames* é muito maior do que o valor de l. Na figura 17 ilustramos um exemplo de grafo RMF.



(a)

Figura 17: RMF

5.2 Estrutura de Dados

Nesta seção apresentamos o modelo de dados na qual o algoritmo executa. Estão sendo utilizadas estruturas do tipo registro para representar os vértices, os arcos e a rede. Definimos como **arcos de fronteira** os arcos cujos vértices pontas estão em processadores diferentes, ou seja pelos arcos de fronteira os fluxos passam de um processador a outro.

Da mesma forma, **vértice de fronteira** é aquele que possui um arco de fronteira em sua lista de arcos.

O registro para representar os vértices é composto por um inteiro que é a representação numérica do vértice, um inteiro que representa o grau do vértice, um inteiro que representa a demanda de fluxo, um inteiro para representar o índice de onde está o primeiro arco no vetor de arcos e um ponteiro para um nó de fila que pode apontar para um nó na fila de demanda.

O registro que representa os arcos é composto por quatro inteiros onde um inteiro representa a origem do arco, um para o destino do arco, um para representar a capacidade e um para representar o fluxo que passa pelo arco.

O registro que representa a rede é composto de por um ponteiro para um vetor de registro de vértices e um ponteiro para vetor de registros de arcos. Também tem um inteiro que representa o vértice inicial e outro o vértice final do conjunto de vértices contidos no processador. Além disso, neste registro um inteiro irá representar o tamanho do conjunto de vértices total e outro representará o tamanho do conjunto local de vértices no processador. Outros dois inteiros farão o mesmo papel para o numero total local e local de arcos.

Existe também um registro de fila que é composta por um ponteiro que aponta para o inicio da fila e um que aponta para o fim. Os ponteiros destes registros apontam para um registro de nó de fila em que é composto por um ponteiro para um vértice e um ponteiro para o próximo nó da fila. Um vetor de inteiros será usado para representar as distâncias dos vértices locais mais os vértices vizinhos aos vértices de fronteira que estão em outros processadores.

Cada processador possuirá um registro de rede, um de fila para os vértices com demanda não nula, além de dois vetores de inteiros para transferência de alteração de fluxo e um vetor de inteiros que representam as distâncias dos vértices.

5.3 Descrição do Algoritmo

Os processadores estarão logicamente organizados em sequência, lado a lado, onde um processador i só pode enviar mensagens para os processadores $i+1$ e $i-1$ da sequência, com exceção dos processadores 0 e p que só enviarão e receberão mensagem de uma direção apenas.

A entrada contém a descrição do tipo de grafo, RLG ou RMF, o número de vértices da rede n e o número de arcos m. Também será informado o número de linhas para a rede do tipo RLG ou tamanho do quadro para a rede do tipo RMF que representaremos por l e o número de colunas para a rede do tipo RLG ou quantidade de frames para a rede do tipo RMF que está representado por c. Cada processador possuirá localmente lc/p vértices se a rede for o RLG ou l^2c/p vértices se for RMF. Os arcos estão representados em duplicidade neste algoritmo estando associados os arcos (i, j) ao vértice i e os arcos (j, i) associados ao arco j. Ambos possuirão as mesmas informações de capacidade e fluxo. O vetor direção do arco na rede original N é sempre $i \rightarrow j$ quando índice de i menor do que o de j. Conceitualmente, um arco da rede residual tem associado a ele o valor da capacidade residual. Em nosso algoritmo o valor da capacidade residual pode ser facilmente observada calculando $r(i, j) = c(i, j) - f(i, j)$ se $i < j$, ou $r(i, j) = f(i, j)$ se $i > j$.

Todos os arcos serão inicialmente saturados. Após a saturação inicial é calculado o valor da demanda de fluxo dos vértices que é o somatório do fluxo dos arcos que saem menos o somatório do fluxo dos arcos que entram. A partir desse resultado, caso a demanda seja diferente de zero será criado um nó de fila apontando para esse vértice e inserido em uma fila de vértices no qual o algoritmo irá processar.

Estando todas as demandas dos vértices calculadas é hora de estabilizá-los, ou seja, igualar o fluxo que entra com o que sai. Retira-se o primeiro nó da fila e localiza o vértice correspondente. É atualizado o valor da distância deste vértice conforme as informações locais de distância. Verifica-se em qual estado o vértice encontra-se e executa a ação correspondente. As ações e os estados são os mesmos apresentados no capítulo anterior.

Após o término do processamento de todos os vértices da fila, é feita a troca das informações referente aos valores dos fluxos nos arcos de fronteira entre os processadores vizinhos. Cada processador enviará e receberá as mensagens nas duas direções, mas em rodadas alternadas. Dessas mensagens recebidas, cada processador manterá os valores de fluxo nos arcos de fronteira o valor recebido na mensagem e realizará a alteração da demanda e do fluxo nos vértices e arcos locais. Os vértices que ficarem com demanda diferente de zero serão inseridos na fila de demanda. Nesta mesma troca de mensagem, também são atualizadas as distâncias dos vértices de fronteira. Atualiza-se a distância de todos os vértices a partir da fronteira que teve as distâncias atualizadas na mensagem.

Após a atualização dos fluxos nos arcos de fronteira e consequentemente das demandas decorrentes das trocas de mensagens, será verificado se existem vértices que estão

novamente com demanda diferente de zero. Caso exista, então será repetida a fase de processamento da fila de demanda com a estabilização dos vértices com demanda diferente de zero. O algoritmo termina quando não houver nenhum vértice na fila de vértices com demanda diferente de zero em todos os processadores.

Algoritmo 3 Algoritmo CGM para PFM

Distribua os vértices entre os processadores;
Sature todos os arcos da Rede;
$demanda(i) = \sum_{(i,j)\in A} f(i,j) - \sum_{(j,i)\in A} f(j,i)$;
$Fim := FALSO$;
Enquanto $Fim = FALSO$ para todos os processadores **faça**
  Para todo $i \mid demanda(i) \neq 0$ **faça**
    Se $demanda(i) < 0$ **então**
      Encontre $(k,i) \in E$ dado que $f(k,i) > 0$;
      $f(k,i) := f(k,i) - min(-demanda(i), f(k,i))$;
    Senão
      Se $demanda(i) > 0) \wedge (d(i) < n) \wedge (d(j) = d(i) - 1$ **então**
        $f(j,i) := f(j,i) + min(demanda(i), r(j,i))$;
      Senão
        Se $demanda(i) > 0 \wedge d(i) = n$ **então**
          Encontre $(i,k) \in E$ dado que $f(i,k) > 0$;
          $f(i,k) := f(i,k) - min(demanda(i), f(i,k))$;
        fim Se
      fim Se
    fim Se
    $d(i) := min(D(i) \cup n)$;
  fim Para
  Envie o valor dos fluxos nos arcos de fronteira ao vizinho anterior e os valores das distâncias dos vértices de fronteira para os dois vizinhos;
  Atualize os valores das distância dos vértices da fronteira vizinha com os valores recebidos;
  $x := f_{local}(i,j) - f_{vizinho}(i,j)$;
  $demanda(i) = x$;
  $f_{local}(i,j) := f_{vizinho}(i,j)$;
  Se $\nexists i \mid demanda(i) \neq 0$ **então**
    $Fim := VERDADEIRO$;
  Senão
    $Fim := FALSO$;
  fim Se
fim Enquanto

Nas figuras 18 estão um exemplo simples do algoritmo. O algoritmo calcula a demanda de cada vértice no processador. Na figura 18a todos os vértices tem demanda igual a -1. Todos os vértices terão que diminuir o fluxo que entra, onde obtemos a figura 18b. Terminado o processamento local, os processadores irão informar os valores dos fluxo dos

arcos de fronteira aos vizinhos correspondentes. No caso P_j informa a P_i que $f(b, c) = 1$, figura 18c, alterando o fluxo em (b, c) que está em P_i, tornando novamente a demanda do vértice b negativa, fazendo com que haja uma nova rodada de processamento local para estabilizar os vértices em P_i como visto na figura 18e.

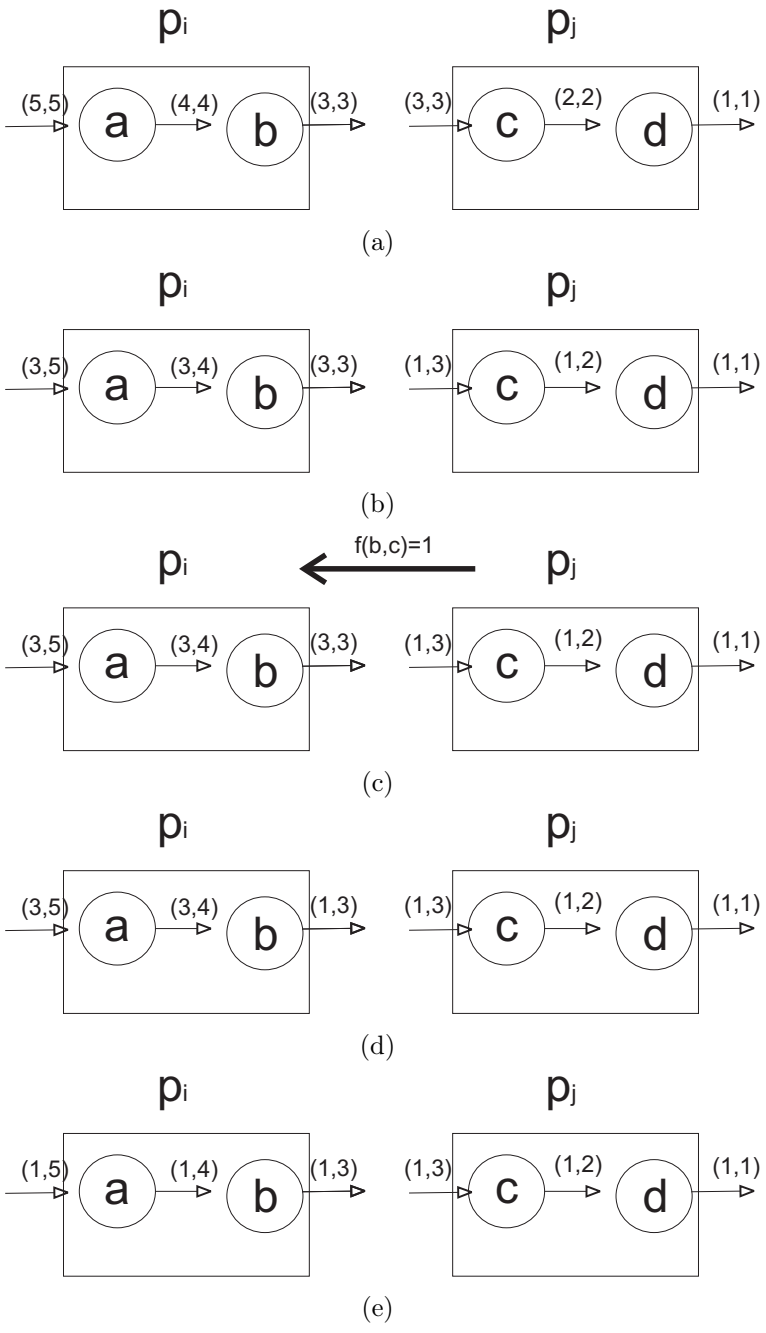


Figura 18: Exemplo da execução do algoritmo BSP/CGM

A correção deste algoritmo deriva da correção do algoritmo autoestabilizante de Gosh *et. al.* apresentado no capítulo anterior.

5.4 Cluster

Um cluster é um conjunto de dois ou mais computadores interligados por uma rede, que trabalham juntos com um propósito computacional em comum como fossem um único computador. O cluster pode ser formado por computadores e equipamentos de redes comuns, mas para melhora de desempenho pode-se adotar soluções de comunicação e computação específicas.

Cada computador de um cluster é denominado nó. Todos devem ser interconectados através de uma rede de qualquer topologia. O sistema operacional usado nos computadores deve ser de um mesmo tipo, isso porque existem particularidades em cada sistema operacional que poderiam impedir o funcionamento do cluster. Independente do sistema operacional usado é preciso usar um software que permita a montagem do cluster em si. Esse software vai ser responsável principalmente pela distribuição do processamento. O software trabalha de forma que erros e defeitos sejam detectados, oferecendo meios de providenciar reparos, mas sem interromper as atividades do cluster.

O cluster utilizado neste trabalho é um Cluster Beowulf de 16 nós, onde cada um é um IBM System x3200 M2, com processador SMP Intel Xeon X3320 com 4 núcleos de 2,5 GHz, com cada núcleo possuindo memória *cache* L2 de 3 MB, e memória de 1 GB de RAM, um adaptador Myrinet 10G-PCIE-8B-QP e um adaptador Etherner Onboard. O cluster também possui um Host com processador Intel Dual E2160 com 2 núcleos 1,80 GHZ, onde cada núcleo possui memória *cache* L2 de 1 MB, e 2 adaptadores Ethernet Offboard. A interconexão dos nós do Cluster para processamento é feita através de uma rede Myrinet de 10 Gbps na topologia Fat Tree, e a administração e o monitoramento são realizados por uma rede Ethernet de 100Mbps. Estão instalados o SO Linux/Debian Lenny 2.6.26 versão de 64-bit, o FAI, Open PBS, o MPICH [23], MPICH2[23] e o OpenMPI[12].

5.5 MPI(Message Passing Interface)

O MPI é uma especificação para bibliotecas de passagem de mensagem que implementam a comunicação entre os processadores de um cluster. Foi desenvolvido por um fórum internacional aberto consistindo de representantes da indústria, acadêmicos e laboratórios de governos. Foi rapidamente e amplamente aceito por ter sido cuidadosamente especificado para permitir máximo desempenho em uma grande variedade de sistemas. As especificações foram definidas para programas em C/C++ e Fortran. O MPI tem como

vantagem a portabilidade, praticidade, eficiência e flexibilidade.

As aplicações que utilizam a biblioteca do MPI, precisam iniciá-lo através da função *MPI_Init* e encerrá-lo com *MPI_Finalize*. O MPI ao ser inicializado cria um canal comunicador entre os processadores, identificando-os no processo. As funções utilizadas para isso são *MPI_Comm_rank* e *MPI_Comm_size*. A comunicação entre os processadores através do MPI é feito por chamada de funções que "envelopam"as informações que precisam ser enviadas a outros processadores que por sua vez "desenvelopam"os dados e os utilizam. As funções de envio de mensagem podem ser bloqueantes ou não bloqueantes. As funções de envio de mensagem são *MPI_Send* e *MPI_Isend*. A primeira é o envio bloqueante e a segunda é o não bloqueante. As funções de recebimento de mensagem são *MPI_Recv* e *MPI_Irecv*. Também o primeiro é o recebimento bloqueante e o segundo é o não bloqueante. Essas são os principais comandos dentre as centenas de funções que a biblioteca tem implementada.

5.6 Testes e Resultados

Foram realizados os teste para 2, 4, 8 e 12 processadores em instâncias onde a quantidade de vértices de 2^{12}, 2^{13}, 2^{14} e 2^{15} vértices. A quantidade de arcos em cada instância é de aproximadamente três vezes o número de vértices. Nos testes as classes de grafos testadas foram restringidas de forma que o corte mínimo estivesse no processador de maior identificador no cluster e as capacidades dos arcos que entram nos vértices são maiores que a capacidade dos arcos que saem dele, de forma que o a rede tenha sucessivos cortes de capacidade crescente. Essa restrição, permite com que o algoritmo apenas reduza o fluxo que entra nos vértices.

Durante a implementação observamos dificuldades com relação a escolha do melhor caminho devido a forte dependência que os algoritmos deste problema carregam. A dificuldade era tanto na escolha de qual caminho o algoritmo iria enviar o fluxo com também a imprecisão das informações das distâncias exata dos vértices de fronteira a um referencial na rede, o que nos levou a restringir às classes de grafos escolhidas. Com a restrição de classes de grafos, tentamos levantar o custo que as recorrentes trocas de fluxos entre vértices de fronteira e alteração de distâncias oneram no desempenho.

Realizamos os testes com três instâncias para cada tamanho de grafo onde realizamos quatro testes para cada instância de onde tiramos a média entre o tempo dos testes. O tempo observado nos teste estão descritos na 5.6 das instâncias do grafo RLG e 5.6 das

	2^{12}	2^{13}	2^{14}	2^{15}
1	130	390	1130	3400
2	50	140	380	1230
4	70	210	160	450
8	40	140	190	230
12	30	110	110	160

Tabela 1: Tabela de tempo (em décimo de segundos) dos testes dos grafos RLG - número de vértices X número de processadores

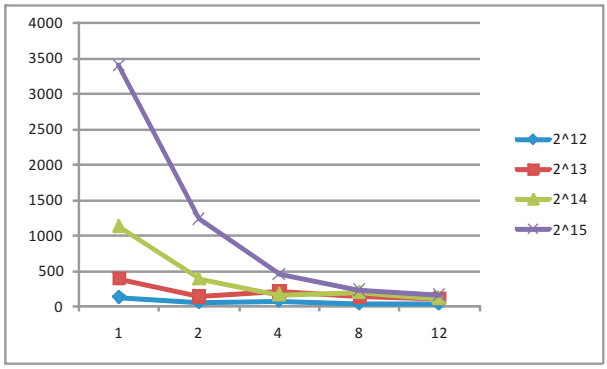


Figura 19: Gráfico de tempo para o grafo RLG - Tempo (em décimos de segundos) × Número de Processadores

instâncias do grafo RMF. Nos gráficos 19 e 21 podemos ver que o tempo estão melhorando a medida que se aumenta o número de processadores. O *speed-up* foi calculado pela razão entre o tempo de execução para determinado número de processadores pelo tempo de execução da versão sequencial do algoritmo em um processador. Podemos observar o *speed-up* crescente como pode ser visto nos gráficos 20 e 22. O *speed-up* observado nos causa estranhamento por ser superlinear, o que nos leva a uma análise da quantidade de operações que ocorrem.

A primeira vez que cada vértice é estabilizado, o algoritmo empurra fluxo de volta à fonte em cada um dos arcos que entra no vértice. Portanto temos no algoritmo sequencialmente nk empurrões de volta, onde k é a média do grau dos vértice dividido por dois. Mas cada vértice pode ser desestabilizado novamente se um vértice vizinho mais

	2^{12}	2^{13}	2^{14}	2^{15}
1	320	3030	5090	11830
2	90	770	1290	2880
4	30	210	350	760
8	20	70	120	230
12	20	40	70	110

Tabela 2: Tabela de tempo (em décimo de segundos) dos testes para instâncias do grafo RMF

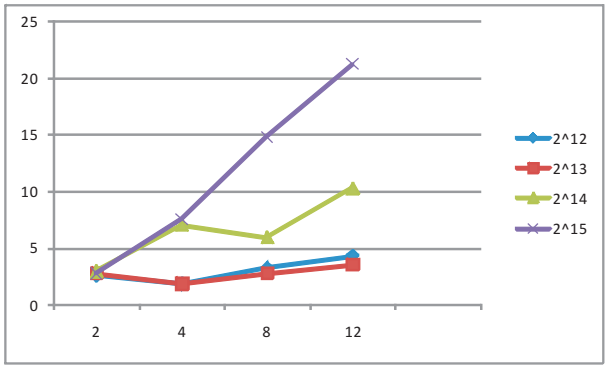


Figura 20: Gráfico de Speedup para o grafo RLG - Speed-up × Número de Processadores

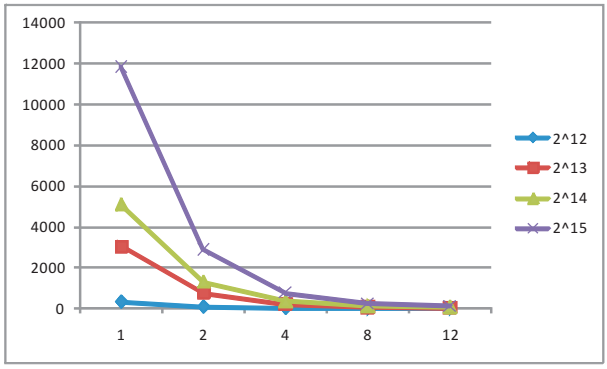


Figura 21: Gráfico de tempo para o grafo RMF - Tempo (em décimos de segundos) × Número de Processadores

próximo ao sorvedouro empurrar fluxo para ele, podendo ocorrer até n/k vezes. Portanto poderá ocorrer $O(n^2)$ empurrões. Mas se olharmos na execução paralela, serão realizados na primeira rodada de computação $O((n/p)^2)$ empurrões. A cada uma das demais rodadas de computação subsequentes, temos que cada vértice não vai mais receber fluxo de outro vizinho até a próxima rodada de computação. Portanto teremos apenas $O(n/p)$ empurrões em cada uma nas rodadas seguintes. Acontecerão p rodadas de computação para que o fluxo vindo dos vértices do último processador alcance a fonte. Com essa restrição de grafos temos que os tempo de computação e mais de um processador é de $O((n/p)^2) + O((n/p)(p-1))$, que significa $O((n/p)^2)$ operações.

Foi observado também que à medida que os grafos diminuem no número de vértices, o tempo de execução começa a divergir do comportamento dos grafos com a execução de vários processadores. Isso pode ser bem observado na instância de 2^{12} vértices do grafo RMF nos gráficos 21 e 22, onde o crescimento do *speed-up* não acompanha ao das demais instâncias medidas. Esse mesmo comportamento também é observada em instâncias menores das que foram medidas em ambas as classes de grafos. Outra anomalia que pode ser observada no comportamento do algoritmo pode ser evidenciada na instância de 2^{14}

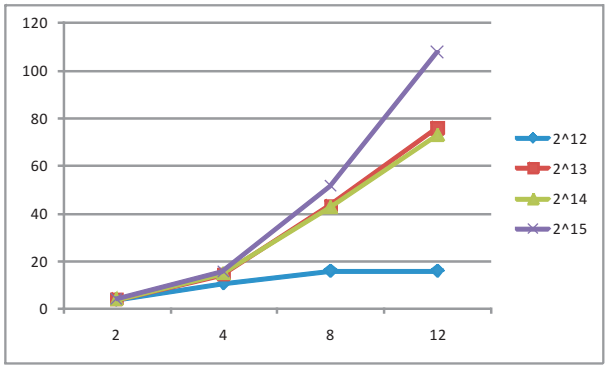


Figura 22: Gráfico de Speedup para o grafo RMF - Speed-up × Número de Processadores

vértices do grafo RLG vista nos gráfico 19 e 20. O *speed-up* observado no teste de 8 processadores diminui em relação ao teste com 4 processadores e volta a aumentar comparando o teste com 12 processadores ao de 8. Esse mesmo comportamento é observado, mas de forma mais branda, nas instâncias de 2^{12} e 2^{13} vértices do grafo RLG no comparativo dos teste de 2 e 4 processadores.

6 Conclusão

Neste trabalho tínhamos o intuito de estudar o problema do fluxo máximo paralelizado para o modelo BSP/CGM. Para isso estudamos o modelo de computação paralela PRAM e o modelo realístico BSP/CGM. Estudamos ainda o conceito do problema do Fluxo Máximo e suas aplicações. Vimos a dificuldade da implementação paralela pelo fato do problema **P-completo**. Estudamos a implementação dos algoritmos de Ford-Fulkerson, Edmond-Karps e Dinic que são baseados no método de caminhos aumentantes, e estudamos também o algoritmo do método de *Push-Relabel*. Estudamos ainda o algoritmo de Anderson-Setubal para a solução do problema em paralelo em uma máquina no modelo PRAM. E por fim estudamos também o algoritmo de Gosh *et. al.* para tratar o problema do fluxo máximo em uma máquina paralela distribuída de granulosidade fina.

Realizamos a implementação de um algoritmo BSP/CGM para tratar o problema do fluxo máximo em uma máquina paralela. Observamos a dificuldade na definição do critério de escolha do arco para realizar a operação, o que nos levou a restringir nossos estudos a duas classes de grafos, para as quais realizamos testes e obtivemos *speed-up* superlinear. Os *speed-ups* obtidos foram de 1,85 até 107 para as classes de grafos escolhidas.

Sob essa observação podemos direcionar futuros estudos em algoritmos que evitam o uso da distância como critério de seleção de arco. Nesse estudo sugiro o estudo do método citado em [24], apresentado por Tabirca e Tabirca. Também podemos orientar trabalhos futuros na direção proposta em [18] sobre fluxo em grafos planares.

Referências

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

[2] Anderson, R., and Setubal, J. C. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *Journal of Parallel and Distributed Computing 29*, 1 (1995), 17–26.

[3] Cheriyan, J., and Maheshwari, S. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal of Computing*, 18 (1989), 1057 – 1086.

[4] Cormen, T. H., Leiserson, C. E., and R. L. Rivest, C. S. *Algoritmos - Teoria e Prática*. Elsevier, 2002.

[5] Dehne, F., Fabri, A., and Rau-Chaplin, A. Scalable parallel geometric algorithms for coarse grained multicomputers. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry* (New York, NY, USA, 1993), ACM Press, pp. 298–307.

[6] Dinic, E. A. Algorithm for solution of problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady 11* (1970), 1277–1280.

[7] Edmonds, J., and Karp, R. M. Theorical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM 19* (1972), 248–264.

[8] Ford Jr, L. R., and Fulkerson, D. *Flows in Network*. Princeton University Press, 1962.

[9] Goldberg, A. V. Processor-efficient implementation of a maximum flow algorithm. *Information Processing Letters 38*, 4 (1991), 179–185.

[10] Goldberg, A. V., and Tarjan, R. E. A new approach to the maximum flow problem. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1986), ACM Press, pp. 136–146.

[11] Gosh, S., Gupta, A., and Pemmaraju, S. V. A self-stabilizing algorithm for the maximun flow problem. In *Computers and Communications. Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on* (March 1995), IEEE, pp. 8–14.

[12] Graham, R. L., Shipman, G. M., Barrett, B. W., Castain, R. H., Bosilca, G., and Lumsdaine, A. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks* (Barcelona, Spain, September 2006).

[13] Greenlaw, R., Hoover, H. J., and Ruzzo, W. *Limits To Parallel computation: P-Completeness Theory*. Oxford University Press, New York, NY, USA, 1995.

[14] Harris, T. E., and Ross, F. S. Fundamentals of a method for evaluating rail net capacities. Research Memorandum RM-1573, The RAND Corporation, US Army, Santa Monica, California, 1955.

[15] Hong, B. A lock-free multi-threaded algorithm for the maximum flow problem. In *IEEE International Parallel and Distributed Processing Symposium 2008* (2008), IEEE Computer Society, pp. 1–8.

[16] JáJá, J. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[17] Karzanov, A. V. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15 (1974), 434–437.

[18] Miller, A., and Naor, J. Flow in planar graphs with multiple sources and sinks. *Siam Journal of Computing 34*, 5 (1995).

[19] Pacheco, P. S. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[20] Reif, J. H. *Synthesis of Parallel Algorithms*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[21] Schrijver, A. Flows in railway optimization. *Nieuw Archief voor Wiskunde 2* (September 2008), 126–131.

[22] Shiloach, Y., and Vishkin, U. An $o(n^2 logn)$ parallel max-flow algorithm. *Journal of Algorithms 3*, 2 (1982), 128 – 146.

[23] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI - The Complete Reference*, vol. 1. The MIT Press, 1997.

[24] Tabirca, S., and Tabirca, T. An $o(n^2)$ parallel algorithm for the maximum flow problem. *Concurrent Information Processing and Computing* (2005), 295 – 300.

[25] Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (1990), 103–111.