UNIVERSIDADE FEDERAL DO MARANHÃO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

INTERFACE DE ANÁLISE DA INTERCONEXÃO EM UMA LAN USANDO CORBA

Milson Silva Monteiro

São Luís

2002

CDU 004.514

CDD 001.642 5

INTERFACE DE ANÁLISE DA INTERCONEXÃO EM UMA LAN USANDO CORBA

Dissertação de Mestrado submetida à Coordenação do
Programa de Pós-Graduação em Engenharia de Eletricidade da UFMA
como parte dos requisitos para obtenção do título de
mestre em Ciência da Computação.

Por

Milson Silva Monteiro

Junho

2002

INTERFACE DE ANÁLISE DA INTERCONEXÃO EM UMA LAN USANDO CORBA

MILSON SILVA MONTEIRO

Dissertação aprovada em 07 de junho de 2002.

Prof. Dr. Zair Abdelouahab
(Orientador)

Prof. Dr. Francisco de Assis Magalhães Gomes
(Membro da Banca Examinadora)

Prof. Dr. Edson Nascimento
(Membro da Banca Examinadora)

INTERFACE DE ANÁLISE DA INTERCONEXÃO EM UMA LAN USANDO CORBA

MESTRADO EM ENGENHARIA DE ELETRICIDADE

ÁREA DE CONCENTRAÇÃO: CIÊNCIA DA COMPUTAÇÃO

MILSON SILVA MONTEIRO

Orientador: Prof. Dr. Zair Abdelouahab

DEDICATÓRIA

A Deus,

A Srª Maria Auta Silva Monteiro,

Ao Profº Dr. Antônio Pinto Neto.

AGRADECIMENTOS

A Deus, pois sem Ele nada poderia ser feito.

A minha família, pelo constante incentivo ao meu desenvolvimento profissional e por me ensinar o significado do amor gratuito.

Ao meu orientador, Prof. Dr. Zair Abdelouahab pela orientação.

Ao Prof. Dr. Antônio Pinto Neto e família por sua amizade, acompanhamento, incentivo, idéias e co-orientação nas pesquisas.

À Coordenadora do Mestrado Prof. Dr^{a}. Maria da Guia por seu apoio constante.

Ao CNPQ pelo apoio financeiro.

Ao Profº Dr. Fernando Pantoja por sua gentileza em colaborar e esclarecer tópicos de pesquisa operacional.

Aos Professores do Programa de Pós-Graduação em Engenharia de Eletricidade da UFMA que me incentivaram.

Aos membros do Departamento Acadêmico de Informática do CEFET – Ma, pela colaboração, incentivo e credibilidade.

A Mary Jane, Diógenes Aquino, Deni e todo o pessoal da Infolivros por sua amizade.

Aos colegas de mestrado Márcia Cristina e Raimundo pela amizade.

A Maria Teresa por sua amizade.

A Sr^{a} Conceição do Departamento de Física da UFMA pela amizade.

A Isabel Mousinho, Ione Amorim e Violeta por sua delicadeza pela qual eu sempre fui recebido na coordenação do mestrado.

Aos verdadeiros amigos, por me conhecerem tão bem, e ainda assim, continuarem sendo meus amigos.

RESUMO

O objeto de estudo deste trabalho é o desenvolvimento de um software (interface gráfica do usuário) que possibilita analisar a interconexão de uma LAN (Local Area Network) usando CORBA (Common Object Request Broker Architecture) em ambientes distribuídos e heterogêneos entre diversas máquinas periféricas. Este trabalho apresenta os paradigmas da teoria de grafos: menor caminho (Dijkstra, Ford-Moore-Belman), fluxo máximo (Edmonds-Karp) e fluxo de custo mínimo (Busacker-Gowen) para formalizar o desenvolvimento da interface. Discorremos sobre a teoria de grafos e fluxos em redes que são relevantes para garantir o embasamento teórico.

Palavras-Chave: Teoria de Grafos; Fluxos em Redes; Problema do Menor Caminho; Problema do Fluxo Máximo; Problema do Fluxo de Custo Mínimo; Algoritmos Paralelos.

ABSTRACT

This works concern software development (graphical user interface) that makes possible to analyze the interconnection in a LAN (Local Area Network) using CORBA (Common Object Request Broker Architecture) on distributed and heterogeneous environment among several outlying machines. This works presents paradigms of graphs theory: shortest paths problems (Dijkstra-Ford-Moore-Belman), maximum flow problems (Edmonds-Karp) and minimum cost flow problems (Busacker-Gowen) to formalize the interface development. We discoursed on the graphs theory and networks flows that are essentials to guarantee theoretical insight.

Keywords: Graph Theory; Networks Flows; Shortest Path Problem; Maximum Flow Problem; Minimum Cost Flow Problem; Parallel Algorithms.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

1.1 Motivação

A rápida disseminação de microcomputadores e estações de trabalho, aumento de processamento computacional e o surgimento de redes de comunicação com grande largura de banda possibilitou aumentar a demanda pelo desenvolvimento de componentes relacionados com a tecnologia de objetos distribuídos [9] [13] [23] [24] [25].

Estes componentes apresentam vantagens advindas da distribuição [7] [9] [23] [29] [36], tais como: disponibilidade, desempenho, otimização de custos, etc. Estes componentes apresentam outras vantagens [27] [28] [29] [30], que são: afastamento, concorrência, assincronismo, heterogeneidade, autonomia, evolução e mobilidade.

Diversas arquiteturas distribuídas foram desenvolvidas com o objetivo de facilitar a implementação de aplicações que apresentam as vantagens descritas anteriormente [13] [25]. A característica de integração [23] [25], impõe a necessidade de especificações abertas, com interfaces padronizadas, possibilitando o desenvolvimento de middlewares abertos [13] [23] [24] [25] [27] [28] [29] [30]. O middleware é uma camada de software intermediária que permite a comunicação entre aplicações cliente/servidor heterogêneas, fornecendo abstrações de alto nível com o objetivo de facilitar a programação distribuída. Estas abstrações apresentam uma visão uniforme na utilização de recursos heterogêneos existentes nas camadas de sistema operacional e redes [7] [35] [36].

A arquitetura CORBA [13] [23] [24] [25] [27] [28] é uma solução baseada em padrões abertos para computação distribuída. A utilização desta arquitetura nesta dissertação é a portabilidade da aplicação em um ambiente distribuído e heterogêneo. Outra característica essencial é a possibilidade que as aplicações clientes e servidores possam ser implementadas em diversas linguagens de programação compatíveis com este padrão. Em [9] [23] [29] [30] e [40] são descritos diversos benefícios da utilização de objetos distribuídos. Por exemplo, sua

utilização ajuda a lidar com a heterogeneidade de alguns sistemas, pois, os serviços fornecidos são separados de suas implementações.

O desenvolvimento da teoria de grafos estuda as relações entre os elementos de conjuntos discretos [1] [4] [12] [17] [21]. O seu intenso desenvolvimento foi motivado por aplicações a problemas de otimização discreta e combinatorial [1] [12] [17] [31]. Esta teoria possui aplicabilidade em diversas áreas do conhecimento humano [1] [17].

O estudo da teoria de grafos é uma área de pesquisa que apresenta um crescimento constante. A justificativa para este crescimento é a utilização dos computadores na resolução de problemas de otimização de larga escala. Estes problemas podem ser modelados e solucionados por algoritmos fornecidos pela teoria de grafos. Esta abordagem fornece modelos de aplicabilidade geral e importância econômica.

Antes de iniciar qualquer formalização é importante delimitarmos nossa área de trabalho, pois, os problemas de grafos ou fluxos em redes são uma área muito abrangente e em constante elaboração em seus aspectos mais sofisticados sob a pressão das necessidades práticas. Para alcançarmos tal objetivo, norteamos a delimitação com as seguintes restrições:

– As situações de tráfego urbano, especialmente aquelas que envolvem tráfego em situações mais críticas, como passagem em túneis ou engarrafamentos não possui aplicabilidade nas hipóteses aqui feitas, o que exige prudência no uso da teoria nesses casos.

– No estudo destes problemas estamos considerando apenas fluxos lineares independentes do tempo (estáticos ou constantes).

– Na resolução consideraremos apenas problemas de unifluxo onde apenas um tipo de recurso é considerado.

– As características relacionadas com fluxos em redes objetivam resolver problemas com vistas à seleção de técnicas de resolução fundamentado na teoria de grafos ou fluxos em redes, possibilitando selecionar abordagens algorítmicas modeláveis por grafos.

– Todos os dados (comprimento, custo, fornecimento/demanda e capacidade) são valores inteiros.

– As redes são finitas, direcionadas e não possuem laços ou arestas paralelas. Ou seja, os modelos de redes estudados excluem vértices isolados (vértices que não são pontos finais

de qualquer aresta), laços (arestas em que os pontos finais coincidem) e arestas paralelas (arestas que possui pontos finais em comum).

A Figura 1.1, mostra as restrições que são aplicadas aos modelos de grafos e redes nesta dissertação.

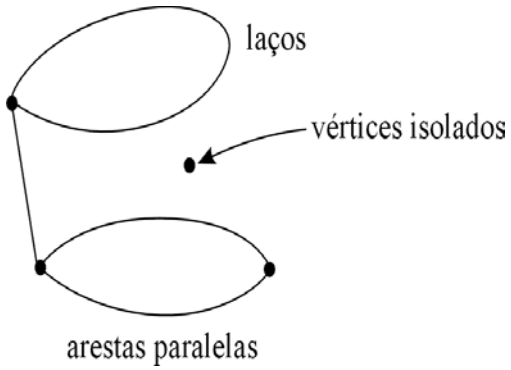


Figura 1.1 Laços, Vértices Isolados e Arestas Paralelas.

1.2 O problema a ser estudado

Em um grafo de topologia arbitrária, desejamos desenvolver uma interface gráfica do usuário que possibilite analisar a interconexão paralela e distribuída entre dois vértices distintos s e t em uma LAN, de modo a não saturar as suas interconexões. Existem diversos fatores que podem saturar a interconexão em uma rede [2], que são: latência (enviar uma mensagem em uma rede, pagando o menor custo pelo fluxo), throughput (quantidade máxima de informação que a rede pode transportar), conectividade (quantos vizinhos imediatos cada nó possui, também conhecido como o grau do nó), custo do hardware (fração do custo total do hardware que a rede representa), integridade (caminhos redundantes), funcionalidade (funções adicionais realizadas pela rede, tais como, combinação de mensagens e tolerância à falhas).

Nesta dissertação, os parâmetros de análise da interconexão são delimitados apenas a latência e ao throughput. A conexão entre os vértices da rede possui uma capacidade

máxima e mínima de fluxo e um custo associado ao transporte do fluxo através destas arestas (conexões).

A Figura 1.2 mostra o problema do throughput. Onde os colchetes expressam os valores mínimos (l_{ij}) e máximos (L_{ij}) para o fluxo em cada arco.

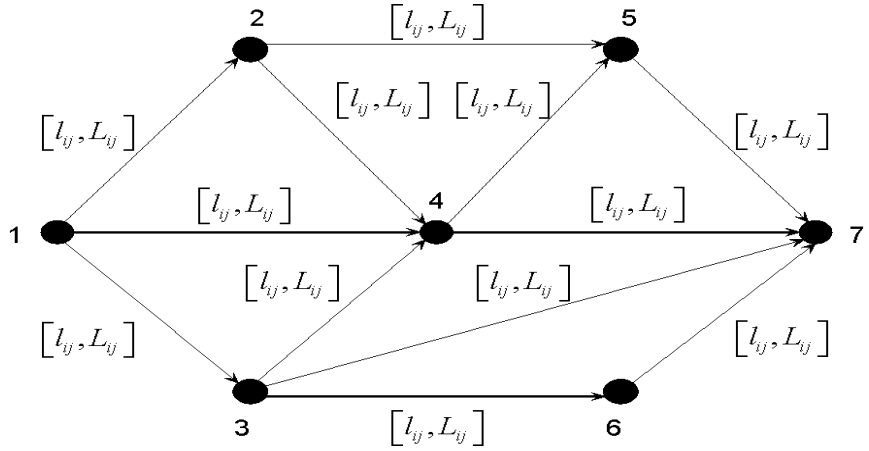


Figura 1.2 Problema do Throughput.

A Figura 1.3 mostra o problema da latência. Onde os colchetes expressam os valores mínimos (l_{ij}), máximos (L_{ij}) e o custo (c_{ij}) associado ao fluxo em cada arco.

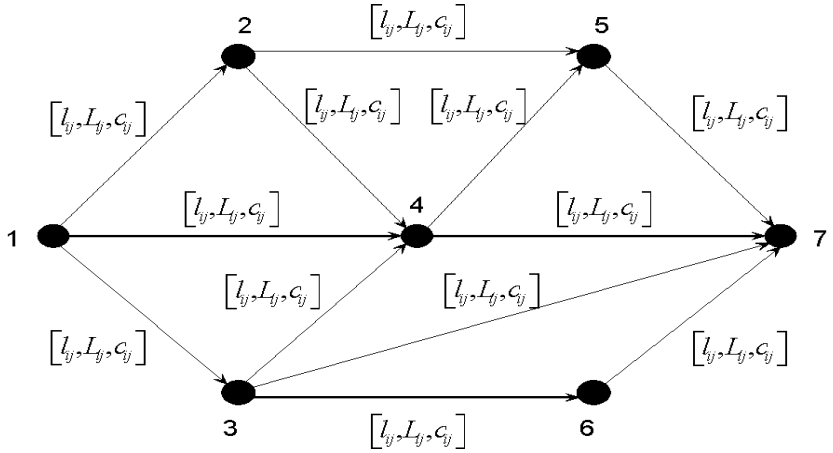


Figura 1.3. Problema da Latência.

Normalmente, o valor do fluxo a ser transportado é conhecido e existem restrições para a quantidade máxima e mínima do fluxo admissível nos arcos e um custo associado ao transporte em cada arco, por unidade de fluxo. Na resolução deste problema, utilizamos uma metodologia conhecida como teoria de grafos. A justificativa para a escolha desta metodologia é a correlação intrínseca existente com o problema.

Durante a fase de seleção dos parâmetros (throughput e latência) observamos que estes poderiam ser modelados através do problema do fluxo máximo (throughput) e do problema do fluxo de custo mínimo (latência) existente na literatura da teoria de grafos.

A solução que apresentamos para percurso em grafos utilizada na implementação dos algoritmos é BFS (Breadth First Search). O algoritmo BFS é um dos algoritmos mais simples para pesquisar em um grafo [1] [6] [12] [17].

A implementação de algoritmos paralelos envolvendo a teoria de grafos ou fluxos em redes não é uma tarefa trivial [2] [11] [19] [20] [22] [33] [41]. A utilização destes algoritmos corrobora para fundamentar o modelo teórico proposto. A abordagem paralela que utilizamos é fundamentada no modelo XPRAM [26]. Este modelo possibilita desenvolver o projeto e a implementação de um algoritmo paralelo baseado em um computador paralelo de propósito geral. No Capítulo 5, fornecemos detalhes do modelo XPRAM.

A justificativa para a paralelização dos algoritmos [2] [8] [10] [11] [16] [18] [20] [22] [22] [38] [40] é executar em um ambiente heterogêneo e distribuído (LAN), a simulação do modelo teórico real. A Figura 1.4 mostra o ambiente heterogêneo e distribuído.

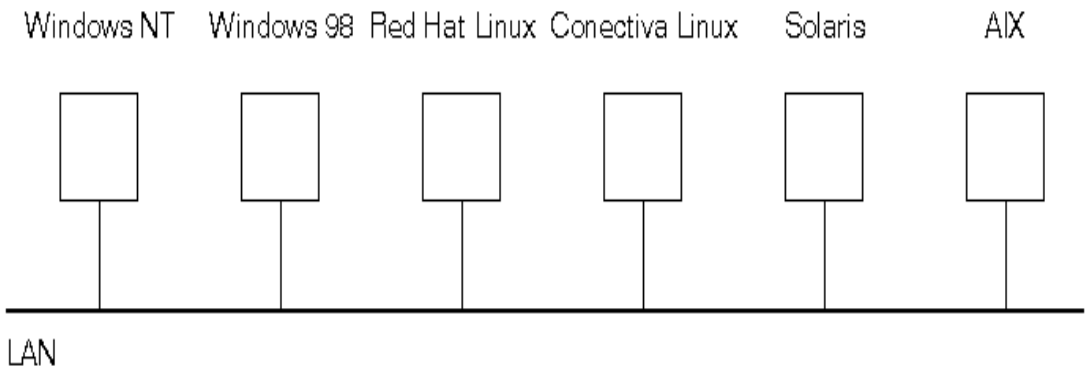


Figura 1.4. Ambiente Heterogêneo e Distribuído.

Nesta dissertação, utilizamos uma sistemática de uma rede de computador (LAN), ou seja, os vértices (nós) no modelo teórico são representados por computadores e as arestas por interconexões (topologia da rede).

1.3 A estrutura da dissertação

Este trabalho está organizado nos seguintes capítulos:

O Capítulo 2 fornece uma fundamentação teórica relacionada ao problema do menor caminho em um grafo. Assim, inicialmente caracteriza-se o conceito de grafo, dígrafo e rede. Em seguida apresenta-se o princípio de Bellman que ocupa uma posição central no desenvolvimento de algoritmos seqüenciais de menor caminho: algoritmo de Dijkstra e algoritmo de Ford-Moore-Bellman. Neste capítulo apresentamos a pesquisa em grafos utilizando BFS (Breadth First Search) e descrevemos a implementação do algoritmo de Dijkstra e Ford-Moore-Bellman para o problema do menor caminho utilizando BFS.

O Capítulo 3 fornece a fundamentação teórica para o problema do fluxo máximo. Neste capítulo apresentamos os conceitos básicos e a seguir descrevemos dois algoritmos para solucionar a instância do problema, que são: algoritmo de Ford-Fulkerson e o algoritmo de Edmonds-Karp.

O Capítulo 4 apresenta a fundamentação teórica necessária para a compreensão do problema do fluxo de custo mínimo sendo posteriormente descrito o algoritmo de Busacker-Gowen que visa solucionar esta classe de problema. Este algoritmo utiliza o algoritmo de menor caminho (Ford-Moore-Bellman) e o algoritmo de fluxo máximo (Edmonds-Karp).

O Capítulo 5 apresenta o conceito de algoritmo paralelo. Detalhamos a abordagem utilizada para implementar os algoritmos baseados no modelo XPRAM. Em seguida descrevemos

os algoritmos paralelos para resolver os problemas de menor caminho, fluxo máximo e fluxo de custo mínimo utilizando o modelo XPRAM.

O Capítulo 6 descreve a arquitetura e os componentes de software utilizados para implementar a interface de análise da interconexão em uma LAN. Em seguida justificamos a escolha do ambiente integrado de desenvolvimento que foi utilizado para implementá-la.

Finalmente, o Capítulo 7 apresenta as conclusões obtidas durante o desenvolvimento desta dissertação.

2 PROBLEMA DO MENOR CAMINHO

Este capítulo apresenta a fundamentação teórica relacionada ao problema do menor caminho em um grafo e os algoritmos seqüenciais originais de Dijkstra e Ford-Moore-Bellman para solucionar este problema. Apresentamos dois algoritmos que implementamos modificando os algoritmos seqüenciais originais (Dijkstra e Ford-Moore-Bellman) e introduzimos a pesquisa em amplitude para pesquisar o menor caminho em um grafo.

2.1 Conceitos Básicos

Um grafo $G = (V, E)$ é formado por um conjunto V de vértices $v_1, v_2, ..., v_n$,(depois simplesmente denotado por $1, 2, ..., n$) e um conjunto E de arestas $e_1, e_2, ..., e_m$, onde cada aresta conecta dois vértices.

Um dígrafo [1] [4] [6] [12] [17] [32] [34] [39] é um grafo no qual cada aresta possui uma direção (indicada por uma seta). Para implementar algoritmos usando grafos e dígrafos é necessário à utilização de matrizes ou listas.

Uma rede [1] [4] [6] [12] [17] [34] [39] é um dígrafo onde cada aresta (i, j) possui um valor numérico associado (tipicamente, custos, capacidades). Cada rede possui dois vértices distintos, que são: o vértice fonte e o vértice sumidouro. Os vértices fonte e sumidouro são representados respectivamente por s e t. Nesta dissertação utilizamos os termos "grafos" e "redes" simultaneamente representando o mesmo domínio de contexto.

2.2 Caminho

Entendemos por um caminho [1] [17] $v_1 \rightarrow v_k$ de um vértice v_1 até um vértice v_k em uma rede N como uma seqüência de arestas em que todos os nós visitados são distintos

$$(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k), \tag{2.1}$$

de acordo com as suas direções em N. A Figura 2.1 mostra um caminho.

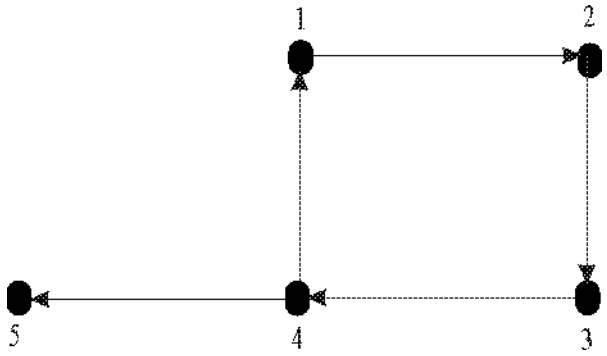


Figura 2.1. Caminho.

Na Figura 2.1, o percurso 1 -> 2 -> 3 -> 4 -> 5 forma um caminho.

2.3 Menor Caminho

Depois que definimos o que é um caminho, podemos definir o que entendemos como menor caminho [1] [6] [17] [32] [34] [39] em uma rede $N = (V, E)$. Para isto, cada aresta (v_i, v_j) em N pode ter uma determinado valor (comprimento, capacidade, custo, peso) $l_{ij} > 0$.

Então o menor caminho $v_1 \rightarrow v_k$ (com v_1 e v_k fixos) é um caminho, tal que a soma dos comprimentos das suas arestas é igual a:

$$l_{12} + l_{23} + l_{34} + ... + l_{k-1,k} \tag{2.2}$$

l_{12} (igual ao comprimento de (v_1, v_2), etc) é mínimo (menor o quanto possível entre todos os caminhos de v_1 até v_k). Semelhantemente, o caminho máximo $v_1 \rightarrow v_k$ é aquele no qual a soma dos comprimentos das suas arestas é máximo.

Problemas que envolvem maior e menor caminho estão entre os mais importantes problemas de otimização. Aqui, o comprimento l_{ij} pode ser um comprimento

medido em metros, mas pode ser também qualquer outro tipo de medida. Por exemplo, escolhendo a "rota mais lucrativa" $v_1 \rightarrow v_k$, um vendedor pode desejar maximizar $\sum l_{ij}$, onde l_{ij} é a sua comissão esperada menos suas despesas para ir da cidade i para a cidade j. Em um problema de investimento, i pode ser o dia em que um investimento é feito, j o dia pagável e l_{ij} o lucro resultante, e o operador esboça uma rede considerando as diversas possibilidades de investimento e re-investimento através de um determinado período de tempo.

2.4 Princípio da Otimalidade de Bellman – Algoritmo de Dijkstra

Na maioria das aplicações as arestas (i, j) podem ter qualquer comprimento $l_{ij} > 0$. Escrevemos $l_{ij} = \infty$ para qualquer aresta (i, j) que não existe em N (fixando $\infty + a = \infty$ para qualquer número a). Consideramos o problema de encontrar o menor caminho de um dado vértice, designado por 1 e denominado origem, para todos os outros vértices 2, 3, ..., n de N. Observamos que L_j significa o comprimento do menor caminho $1 \rightarrow j$ em N.

O princípio da otimalidade de BELLMAN apud KREYSZIG (1993, 1021) é mostrado a seguir:

Se $P: 1 \rightarrow j$ *é o caminho mínimo de 1 até* j *em* N *e* (i, j) *é a última aresta de* P, *então* $P_i: 1 \rightarrow i$ *(é obtido eliminando* (i, j) *de* P*) é o menor caminho de* $1 \rightarrow i$.

A Figura 2.2, mostra o princípio da otimalidade de Bellman nos caminhos P *e* P_i.

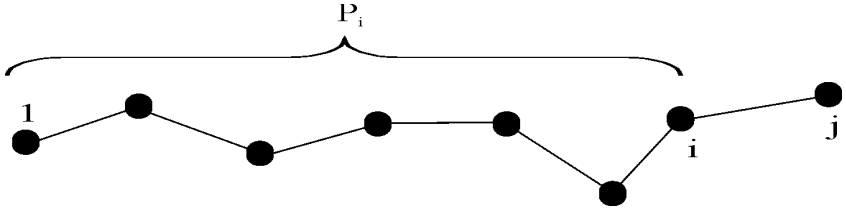


Figura 2.2. Princípio da Otimalidade de Bellman.

Do princípio de Bellman [1] [17], podemos derivar as equações a seguir. Para j fixo podemos obter vários caminhos $1 \rightarrow j$ obtendo os menores caminhos P_i para vários i para os quais existe em N uma aresta (i, j), e adicionando (i, j) ao P_i correspondente.

Estes caminhos obviamente têm comprimento $L_i + l_{ij}$ (L_i = comprimento de P_i). Podemos agora obter o menor de todos os i, ou seja, escolha um i, tal que, $L_i + l_{ij}$ seja mínimo. Pelo princípio de Bellman, este é o menor caminho. Ele possui o comprimento:

$$L_i = 0.$$

$$L_j = \min_{i \neq j} (L_i + l_{ij}), \qquad j = 2, ..., n.$$

Estas são as equações de Bellman. Uma vez que $l_{ii} = 0$ por definição, ao invés de $\min_{i \neq j}$ podemos simplesmente escrever $\min_i$. Estas equações sugerem a idéia de um dos algoritmos mais conhecidos para o problema do menor caminho denominado Algoritmo de Dijkstra [1] [6] [12] [17] [32] [34].

2.5 Formulação do Problema do Menor Caminho

Podemos formular [1] [3] [6] o problema do menor caminho como um problema de programação matemática da seguinte forma:

Minimizar $z = \sum_{(i,j) \in A} l_{ij} x_{ij}$ (2.3)

sujeito a:

$$\sum_{\{j:(i,j) \in A\}} x_{ij} - \sum_{\{j:(j,i) \in A\}} x_{ki} = \begin{cases} n-1 & para\ i = s \\ -1\ para\ todos\ i \in N - \{s\} \end{cases} \quad (2.4)$$

$$x_{ij} \geq 0\ para\ todo\ (i, j) \in A \quad (2.5)$$

Onde:

x_{ij} representa o fluxo circulando no arco $(i, j) \in A$.

Os vértices s e t representam os vértices fonte e sumidouro da rede.

2.6 Algoritmo de Dijkstra

A Tabela 2.1 apresenta o algoritmo de Dijkstra. Este é um procedimento de rotulamento.

ALGORITMO DIJKSTRA [$N = (V, E)$, $V = \{1, ..., n\}$, l_{ij} para todas as arestas (i, j) em E]

Dado uma rede $N = (V, E)$ com vértices 1, ..., n e arestas (i, j) possuindo comprimento $l_{ij} > 0$, este algoritmo determina os comprimentos do menor caminho do vértice 1 para os vértices 2, ..., n.

ENTRADA: Número de vértices n, arestas (i, j), e comprimentos.

SAÍDA: O comprimento L_j do menor caminho $1 \rightarrow j$, $j = 2$, ..., n.

1. passo inicial

   o vértice 1 obtém RP: $L_1 = 0$.

   o vértice j (= 2, ..., n) obtém RT: $\tilde{L}_j = l_{1j}$ (= ∞ se não existe aresta (i, j) em G);

   fixar $RP = \{1\}$, $RT = \{2, 3, ..., n\}$.

2. fixando um rótulo permanente

   encontre um k entre os RT´s para o qual $\tilde{L}_k$ seja mínimo, fixar $L_k = \tilde{L}_k$. Pegue o menor k se existir vários. Elimine o k de RT e inclua-o em RP;

   se $RT = \varnothing$ (isto é, RT está vazio) então

   SAÍDA $L_2, ..., L_n$. Pare.

senão continue (isto é, retorne para o Passo 3). 3. atualizando os rótulos temporários. para todo j com RT, fixe $\tilde{L}_j = \min_k \{\tilde{L}_j,\ L_k + l_{kj}\}$. retornar para o Passo 2. FIM DIJKSTRA.

Tabela 2.1. Algoritmo de Dijkstra.

Em cada estágio de execução, cada vértice v obtém um rótulo é do tipo:

(RP) rótulo permanente = comprimento L_v do menor caminho $1 \rightarrow v$.

ou

(RT) rótulo temporário = limite superior $\tilde{L}_v$ para o comprimento do menor caminho $1 \rightarrow v$.

O algoritmo possui um passo inicial no qual o vértice é 1 e obtém o rótulo permanente $L_1 = 0$ e os outros vértices possuem vértices temporários, então o algoritmo alterna entre os passos 2 e 3. No passo 2 a idéia é fixar k "minimamente". No passo 3, a idéia é que os limites superiores em geral melhorariam (diminuiriam) e seriam atualizados adequadamente. A complexidade [1] [6] [17] [21] [31] [39] do algoritmo de Dijkstra é $O(n^2)$.

Este algoritmo não é capaz de encontrar menor caminho na presença de arcos com pesos negativos, uma vez que a cada iteração, o vértice examinado com menor distância acumulado é fechado [1] [6] [17] [32].

2.7 Algoritmo BFS (Breadth First Search)

O algoritmo BFS [1] [6] [32] é um dos algoritmos mais simples para pesquisar em um grafo. Este algoritmo expande a amplitude entre vértices descobertos e vértices não descobertos uniformemente ao longo da amplitude de fronteira.

Este algoritmo descobre todos os vértices à distância k a partir de s, antes de descobrir quaisquer outros vértices à distância $k+1$. O algoritmo BFS foi desenvolvido por Moore [17].

A Tabela 2.2 apresenta o algoritmo BFS.

```
ALGORITMO BFS (u : vértice)
        {u é o vértice onde a pesquisa inicia}
        {Q é uma estrutura de dados do tipo fila, onde, esta fila está
vazia no inicio e todos os vértices ainda não foram visitados}
        inserir o vértice u em Q;
        marcar o vértice u como "visitado";
        enquanto (Q ≠ ∅) faça
            remover o vértice u que é o primeiro de Q;
            para cada vértice v adjacente a u faça
                se (o vértice v ainda não foi "visitado") então
                    inserir o vértice v em Q;
                    marcar o vértice v como "visitado";
FIM BFS.
```

Tabela 2.2. Algoritmo BFS.

Neste algoritmo , parte-se de um nó u escolhido e visita-se este nó. Em seguida , considera-se cada um dos nós v adjacentes a u.

Para cada um dos nós v:

visita-se o nó v;

coloca-se o nó v na fila Q;

Ao terminar de visitar os nós v, toma-se o nó que estiver na frente da fila Q e repete-se o processo visitando cada um dos nós adjacentes a ele e colocando-os em Q.

Quando um nó é visitado ele é "marcado" , de modo que se for encontrado novamente no percurso não será visitado outra vez nem colocado na fila Q.

2.8 Algoritmo de Dijkstra usando BFS

Na Tabela 2.3, apresentamos uma implementação que desenvolvemos do algoritmo de Dijkstra utilizando BFS. O BFS é considerado como o método mais eficiente para percorrer um grafo e é bem adaptado para o paralelismo.

ALGORITMO DIJKSTRA BFS[$N = (V, E)$, $V = \{1, ..., n\}$, l_{ij} para todas as arestas (i, j) em E]

Dado uma rede $N = (V, E)$ com vértices 1, ..., n e arestas (i, j) possuindo comprimento $l_{ij} > 0$, este algoritmo determina os comprimentos do menor caminho do vértice 1 para os vértices 2, ..., n.

ENTRADA: Número de vértices n, arestas (i, j), e comprimentos.

SAÍDA: O comprimento L_j do menor caminho $1 \rightarrow j$, $j = 2$, ..., n.

1. instanciar as variáveis.

   para $i, u, v = 1$ até n faça

   $pai_i = 0$;

   $d_u = \infty$;

   $d_v = 0$;

   $Q = \varnothing$;

   inserir o vértice fonte s em Q;

   marcar s como "visitado";

2. enquanto ($Q \neq \varnothing$) faça

   remova o primeiro vértice de Q que denotamos por u.

   para cada vértice v adjacente a u faça

   se ($N_{uv} \neq 0$) então

   $d_v = d_u + N_{uv}$;

se ($d_v < d_u$) então

$d_u = d_v$;

se (o vértice v ainda não foi visitado) então

$pai_v = u$;

inserir o vértice v em Q ;

marque o vértice v como "visitado";

senão

se (o vértice v já foi visitado e ($d_u < d_{u-1}$))

então

$pai_v = v$;

se ($pai_v = v$) então

$pai_v = v + 1$;

3. executar Backtrack;

[Regra de Backtracking. Usando a numeração dos vértice de n até 1 (não os rótulos), em cada passo, se um vértice rotulado é encontrado, vá para o próximo vértice que possui o menor número (não o rótulo) entre todos os vértices rotulados $i - 1$.]

se (existe caminho de s para t) então

SAÍDA L_2 , ..., L_n .

senão

imprima("Não existe caminho entre os vértices s e t ");

pare.

FIM DIJKSTRA BFS.

Tabela 2.3. Algoritmo de Dijkstra usando BFS.

Em cada estágio de execução, cada vértice v é marcado como visitado. Este algoritmo realiza um percurso completo no espaço de soluções.

O algoritmo usando BFS que apresentamos somente funciona se a matriz de adjacência for triangular superior, ou seja, não existem ciclos negativos.

2.8.1 Caso Teste

A Figura 2.3 mostra a rede N_1. Nesta rede desejamos encontrar o menor caminho entre os vértices $s(1)$ e $t(12)$.

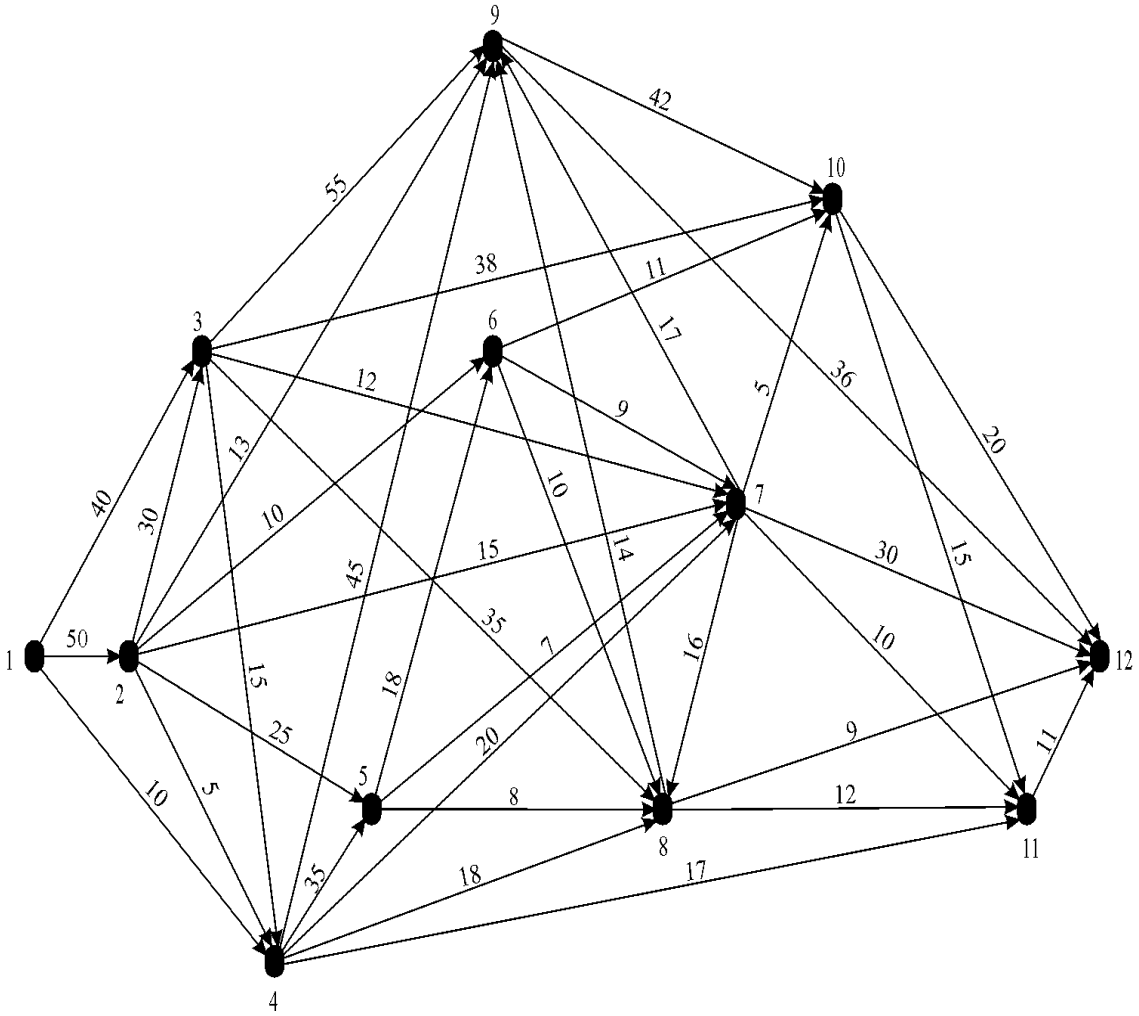


Figura 2.3. Rede N_1.

A matriz de adjacência a seguir representa os pesos (distância) associados às arestas da rede da Figura 2.3.

$$\begin{bmatrix}
0 & 50 & 40 & 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 30 & 5 & 25 & 10 & 15 & 0 & 13 & 0 & 0 & 0 \\
0 & 0 & 0 & 15 & 0 & 0 & 12 & 35 & 55 & 38 & 0 & 0 \\
0 & 0 & 0 & 0 & 35 & 0 & 20 & 18 & 45 & 0 & 17 & 0 \\
0 & 0 & 0 & 0 & 0 & 18 & 7 & 8 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 9 & 10 & 0 & 11 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 17 & 5 & 10 & 30 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 12 & 9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 42 & 0 & 36 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15 & 20 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

O menor caminho encontrado na rede N_1 de $s(1)$ - $t(12)$ = 1 -> 4 -> 8 -> 12 e a sua distância é igual a 37.

2.9 Algoritmo de Ford-Moore-Bellman

O algoritmo de Ford-Moore-Bellman [1] [6] [12] [17] evita o fechamento de um nó a cada iteração e examina todos os nós até que não seja mais possível melhoria. Podendo com isso, aceitar arestas com pesos negativos.

O critério de parada está associado à não modificação de todos os rótulos em uma iteração. A idéia básica da iteração é de que, se um caminho de um vértice s para um conjunto j contém k arestas, um caminho melhor de s para j conterá, no máximo, $k+1$ arestas. Denominamos de:

$l(s, j) \equiv$ comprimento de um caminho entre s e j.

$l_s \equiv$ comprimento do caminho associado ao nó s.

$l_{j_k}^k \equiv$ comprimento do menor caminho P_{sj}^k usando no máximo k arcos, em que $P_{sj}^k \subset A$.

A Tabela 2.4 apresenta o algoritmo de Ford-Moore-Bellman.

ALGORITMO FORD-MOORE-BELLMAN [$N = (V, E)$, $V = \{1, ..., n\}$, l_{ij} para todas as arestas (i, j) em E]

Dado uma rede $N = (V, E)$ com vértices 1, ..., n e arestas (i, j) possuindo comprimento l_{ij}, este algoritmo determina os comprimentos do menor caminho do vértice 1 para os vértices 2, ..., n.

ENTRADA: Número de vértices n, arestas (i, j), e comprimentos.

SAÍDA: O comprimento L_j do menor caminho $1 \rightarrow j$, $j = 2, ..., n$.

1. passo inicial

   $k \leftarrow 1$;

   $l_s \leftarrow l_s^0 \leftarrow 0$;

   $l_j \leftarrow 0, j = 1,...,n-1$;

   $l_s^1 \leftarrow l(s, j)\, j = 1,...,n-1$;

   $l(i, j) \leftarrow \infty, se(l(i, j)) \notin A$;

2. encontrando o Menor Caminho

   enquanto($l_j^{k-1} \neq l_j^k, \forall j \in N$) faça

   $l_j^{k+1} \leftarrow \min\left\{l_j^k, \min_i\left[l_i^k + l(i, j)\right]\right\} i \neq j, s; j = 1,...,n-1$

   $k \leftarrow k + 1$;

3. executar Backtrack;

   { O comprimento L_j do menor caminho $1 \rightarrow j$, $j = 2, ..., n$.}

   pare;

FIM FORD-MOORE-BELLMAN.

Tabela 2.4. Algoritmo de Ford-Moore-Bellman.

Se as arestas forem inspecionadas em uma ordem previamente definida (como por exemplo, a lexicográfica) a complexidade [1] [6] [17] [21] [31] [34] [37] [39] deste algoritmo é O (mn).

2.10 Algoritmo de Ford-Moore-Bellman usando BFS

Na Tabela 2.5 apresentamos uma implementação que desenvolvemos do algoritmo de Ford-Moore-Bellman usando BFS.

ALGORITMO FORD-MOORE-BELLMAN BFS[$N = (V, E)$, $V = \{1, ..., n\}$, l_{ij} para todas as arestas (i, j) em E]

Dado uma rede $N = (V, E)$ com vértices 1, ..., n e arestas (i, j) possuindo comprimento l_{ij}, este algoritmo determina os comprimentos do menor caminho do vértice 1 para os vértices 2, ..., n.

ENTRADA: Número de vértices n, arestas (i, j), e comprimentos.

SAÍDA: O comprimento L_j do menor caminho $1 \rightarrow j$, $j = 2, ..., n$.

1. instanciar as variáveis.

   para $i, u, v = 1$ até n faça

   $pai_i = 0$;

   $d_{u_i} = \infty$;

   $d_v = 0$;

   $Q = \varnothing$;

   inserir o vértice fonte s em Q;

   $d_{u_1} = 0$;

   marcar o vértice s como "visitado";

2. enquanto ($Q \neq \varnothing$) faça

remover o primeiro vértice de Q que denotamos por u;

para cada vértice v adjacente ao vértice u faça

se ($N_{uv} \neq 0$) então

$d_v = d_{u_u} + N_{uv}$;

se (($d_{v_v} < d_{u_v}$) e o vértice v ainda não foi "visitado") então

$d_{u_v} = d_v$;

$pai_v = u$;

inserir o vértice v em Q;

marcar o vértice v como "visitado";

se (($d_v < d_{u_v}$) e o vértice v já foi "visitado") então

$du_v = dv_v$;

$pai_v = u$;

se (($d_v = 0$) e o vértice v já foi "visitado") então

se ($d_v < d_{u_v}$) então

$d_{u_v} = d_v$;

$pai_v = u$

3. executar Backtrack

{ O comprimento L_j do menor caminho $1 \rightarrow j$, $j = 2, ..., n$.}

Pare;

FIM FORD-MOORE-BELLMAN BFS.

Tabela 2.5. Algoritmo de Ford-Moore-Bellman usando BFS.

Em cada estágio de execução, cada vértice v é marcado como "visitado". Este algoritmo assim como na sua versão original, evita o fechamento de um nó a cada iteração, e

examina os nós até que não seja mais possível otimizar o percurso, podendo, com isso, aceitar arestas com pesos negativos.

2.10.1 Caso Teste

A Figura 2.4 mostra a rede N_2. Nesta rede desejamos encontrar o menor caminho entre os vértices $s(1)$ e $t(12)$.

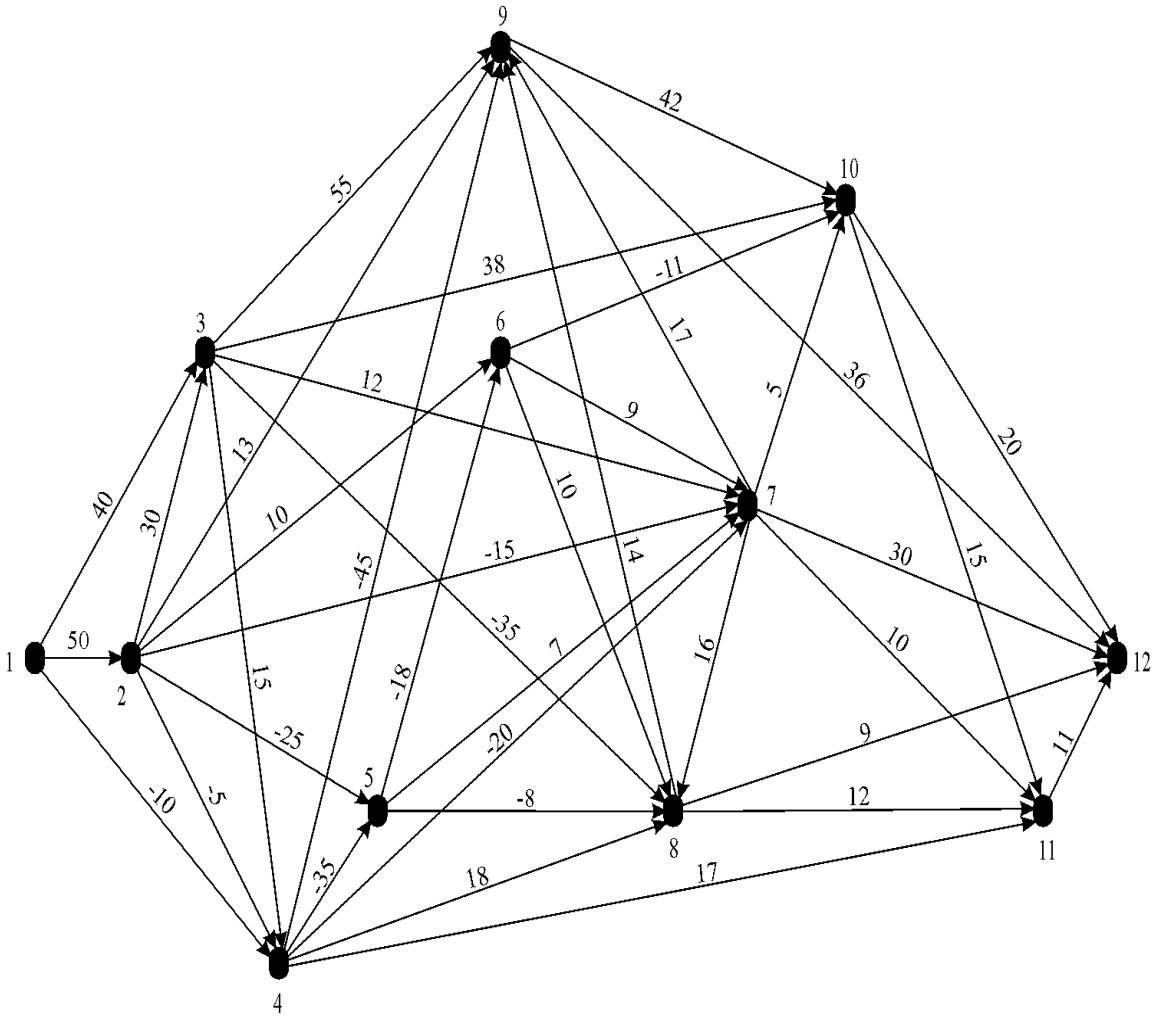


Figura 2.4. Rede N_2.

A matriz de adjacência a seguir representa os pesos (distância) associados às arestas da rede da Figura 2.4.

$$\begin{bmatrix}
0 & 50 & 40 & -10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 30 & -5 & -25 & 10 & -15 & 0 & 13 & 0 & 0 & 0 \\
0 & 0 & 0 & 15 & 0 & 0 & 12 & -35 & 55 & 38 & 0 & 0 \\
0 & 0 & 0 & 0 & -35 & 0 & -20 & 19 & -45 & 0 & 17 & 0 \\
0 & 0 & 0 & 0 & 0 & -18 & 7 & -8 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 9 & 10 & 0 & -11 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 17 & 5 & 10 & 30 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 12 & 9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 42 & 0 & 36 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15 & 20 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

O menor caminho encontrado na rede N_2 de $s(1)$ - $t(12)$ = 1 -> 4 -> 5 -> 6 -> 10 -> 12 e a sua distância é igual a - 54.

3 PROBLEMA DO FLUXO MÁXIMO

Este capítulo tem por objetivo apresentar uma fundamentação teórica relacionada ao problema do fluxo máximo em uma rede, bem como descrever dois algoritmos seqüenciais que possibilitam solucionar este tipo de problema. Os respectivos algoritmos seqüenciais descritos são: o algoritmo de rotulação de Ford-Fulkerson baseado no grafo de aumento de fluxo e o algoritmo de Edmonds-Karp baseado no grafo de escalonamento de capacidade.

3.1 Fluxo

Denotamos o fluxo ao longo de uma aresta direcionada (i, j) por f_{ij} e impomos duas condições:

1. Para cada aresta $e \in E$ em N o fluxo não excede a capacidade c_{ij},

   $0 \leq f_{ij} \leq c_{ij}$ ("condição da aresta"). (3.1)

2. Para cada vértice i, diferente de s ou t,

$$\sum_{k} f_{ki} - \sum_{j} f_{ij} = \begin{cases} 0 \text{ se o vértice } i \neq s, i \neq t, \\ -f \text{ na fonte } s, \\ f \text{ no sumidouro } t \end{cases} \quad (3.2)$$

A segunda condição é denominada regra de conservação ou Lei de Gustav Kirchhoff [17].

3.2 Tipos de Redes

Podemos classificar as redes em dois tipos [1] [4] [6] [12] [17], que são:

- Redes Capacitadas.
- Redes Limitadas.

Uma rede é capacitada se para cada estágio o fluxo é limitado apenas pelo seu limite superior. O limite superior (L) está relacionado a sua capacidade máxima. O fluxo f então satisfaz a seguinte condição:

Para cada aresta $e \in E$, $0 \leq f(e) \leq L$. (3.3)

Em uma rede capacitada, todos os limites inferiores das arestas constituintes da rede possuem valor zero.

Em uma rede limitada, adicionalmente aos limites superiores das arestas, o fluxo também é delimitado por um limite inferior (l) em suas respectivas arestas. O fluxo f então satisfaz a seguinte condição:

Para cada $e \in E$, $l \leq f(e) \leq L$. (3.4)

Observar que podemos encontrar em uma rede limitada no mínimo uma aresta e na qual $l(e) \neq 0$. Senão, (se $l(e) = 0$, para todas as arestas da rede), a rede é capacitada.

3.3 Folga de um Arco

Denominamos folga do arco (x, y) aos seguintes valores:

$$\bar{\xi}(x,y) = L(x,y) - f(x,y) \quad se \quad f(x,y) < L(x,y). \tag{3.5}$$

$$\tilde{\xi}(x,y) = f(x,y) - l(x,y) \; se \quad f(x,y) > l(x,y). \tag{3.6}$$

Um arco (x, y) pode ter simultaneamente $\bar{\xi}$ como $\tilde{\xi}$.

3.4 Caminho de Aumento de Fluxo

Nosso objetivo será maximizar o fluxo da fonte s para o sumidouro t em uma rede. A idéia então é encontrar um caminho [1] [17] $P: s \rightarrow t$ em N que, em relação ao fluxo corrente, possui uma folga que pode ser calculada por $\xi = Min\{\bar{\xi}, \tilde{\xi}\}$. A conservação

dos fluxos nos vértices não será afetada se aumentarmos de ξ o fluxo nos arcos diretos e subtrairmos de ξ o fluxo nos arcos reversos ao caminho de aumento de fluxo.

3.5 Conjunto de Corte

Considerando a rede N, cujo grafo de substrato é $G=(V,E)$ e $(X,\bar{X})$ uma partição de V tal que $X\cup\bar{X}=V$ e $X\cap\bar{X}=\varnothing$, dizemos que $(X,\bar{X})$ define um corte $s-t$ em $G=(V,E)$.

A Figura 3.1 exemplifica um corte $s-t$, em que $X=\{s,1,2,3\}$, $\bar{X}=\{4,t\}$.

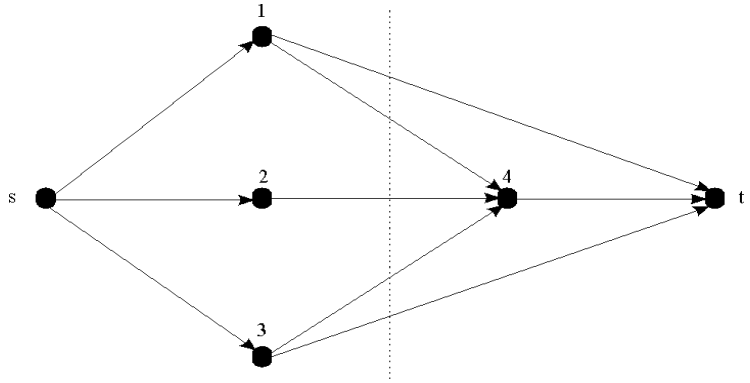


Figura 3.1. Topologia da Rede e Conjunto de Corte.

Pelo teorema da circulação [1] [4] [12], a existência de um fluxo implica em:

$$L(X,\bar{X})\geq l(X,\bar{X}),\ \ \forall X,\bar{X}\subseteq V. \tag{3.7}$$

onde $l(X,\bar{X})$ e $L(X,\bar{X})$ representam os limites inferior e superior do fluxo total entre o conjunto de nós $X,\bar{X}$. As conclusões fornecidas pelo teorema da circulação são:

1. O valor do fluxo máximo f_0^* na rede $s-t$ deve ser menor ou igual a capacidade líquida de qualquer corte $s-t$, ou seja:

$$f_0^*\leq L(X,\bar{X})-l(X,\bar{X})\ \ \forall corte\ \ s-t. \tag{3.8}$$

2. Todo fluxo f^* na rede $s-t$, tal que:

$$f_0^* = \min_{corte\ s-t(X,\bar{X})} \{L(X,\bar{X}) - l(X,\bar{X})\} \text{ é máximo.} \quad (3.9)$$

3.6 Viabilidade do Fluxo

Um fluxo f é viável se e somente se:

Para cada $e \in E$, $l \leq f(e) \leq L$. (3.10)

Da equação (3.10), em uma rede capacitada não existe problema em calcular um fluxo viável. De fato, um fluxo viável inicial igual a zero é suficiente.

Uma vez que o fluxo viável tenha sido determinado, obtemos um caminho de aumento de fluxo da fonte para o sumidouro, então aumentamos o fluxo através das arestas do caminho. Este processo é repetido até que não exista caminho de aumento de fluxo na rede. Este é o método geral aplicado para obter o fluxo máximo em uma rede.

O problema de encontrar o fluxo viável em uma rede limitada é mais complexo em relação a uma rede capacitada. Aqui a complexidade [1] [6] [12] [17] [19] [21] [31] [34] [41] é que o fluxo viável inicial não é igual a zero uma vez que o fluxo pode satisfazer a seguinte condição:

Para cada $e \in E$ $l \leq f(e) \leq L$. (3.11)

3.7 Modificação de uma Rede N

O problema da obtenção de uma solução viável aparece associado à questão mais geral de saber se, de fato, existe algum fluxo viável em uma rede $N' = (V', E')$ com vetores limite de capacidade $l \neq 0$ e c $\geq l$. Não sendo $f = 0$ em geral uma solução viável, é preciso que se disponha de uma técnica capaz de encontrar um fluxo viável, se este existir.

Para isso utilizaremos [1] [3] [4] [6] [12] [17] [33] [39] uma rede auxiliar $N' = (V', E', f^*)$ no qual se procuram levar os limites inferiores a zero, definindo-se:

$$f_k^* = f_k - l_k \quad \forall u_k \in E' \tag{3.12}$$

Em geral não se pode garantir que o novo "fluxo" f satisfaça a lei da conservação, então teremos que incluir nesta uma correção para cada vértice,

$$\sum_{u_j \in \omega_i^-} f_j^* - \sum_{u_j \in \omega_i^+} f_j^* + \varepsilon_i = 0 \quad \forall i \in V' \tag{3.13}$$

onde:

ω_i^- e ω_i^+, significa retirar ou adicionar fluxos aos arcos artificiais $(\bar{s}, \bar{t})$ da rede auxiliar.

$$\varepsilon_i = \sum_{u_j \in \omega_i^-} b_j - \sum_{u_j \in \omega_i^+} b_j \quad \forall i \in V' \tag{3.14}$$

Para satisfazermos à lei de conservação, precisaremos então de um arco auxiliar para cada vértice com ε_i não nulo, através do qual se possa:

- Introduzir essa diferença de fluxo, se $\varepsilon_i > 0$ (caso em que teremos subtraído uma quantidade maior do fluxo na entrada do que na saída do vértice, logo faltará fluxo).
- Retirar essa diferença, se $\varepsilon_i < 0$ (caso em que, ao contrário, teremos subtraído uma quantidade maior na saída do que na entrada, logo sobrará fluxo).

Estes arcos, de capacidade ε_i, que deverão ser adjacentes a uma fonte fictícia $\bar{s}$ (no primeiro caso) ou a um sumidouro fictício $\bar{t}$ (no segundo caso), são chamados arcos terminais. Consideraremos no processo, um arco de retorno $(\bar{t}, \bar{s})$.

Evidentemente um fluxo viável somente poderá existir se os arcos terminais forem saturados, visto que as diferenças ε_i terão sido satisfeitas e o novo fluxo obedecerá a lei de conservação.

A rede $N' = (V', E', f^*)$ será então definida como:

$$V' = V \cup \left\{\bar{s}, \bar{t}\right\} \tag{3.15}$$

$$E' = E' \cup \mathbb{U} \left\{\left(\bar{t}\right), \left(\bar{s}\right)\right\} \tag{3.16}$$

Onde $\mathbb{U} = \{u_i\}$, sendo os u_i da forma ($\bar{s}$, i) ou (i, $\bar{t}$) conforme o sinal de ε_i. As capacidades dos u_i são iguais a ε_i e as capacidade dos u_i (arcos originais) a $c_i - l_i$. Todos os limites inferiores são nulos. Uma vez obtido um fluxo f, a solução inicial será obtida explicitando-se os f_k em:

$$f_k = f_k + l_k \tag{3.17}$$

É importante observar que o total dos ε_i positivos será sempre igual ao total dos ε_i negativos, visto que o fluxo adicional introduzido terá sempre que ser igual ao que é retirado. O problema está em se conseguir saturar os arcos terminais.

3.8 Formulação do Problema do Fluxo Máximo

Podemos formular [1] [3] [6] [12] [17] o problema do fluxo máximo como um problema de programação matemática da seguinte forma:

Maximizar v (3.18)

sujeito a:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v & para\ i = s \\ 0 & para\ todo\ i \in N - \{s\ e\ t\} \\ -v & para\ i = t \end{cases} \tag{3.19}$$

$$0 \le x_{ij} \le u_{ij} \quad para\ cada\ (i,j) \in A \tag{3.20}$$

Onde:

x_{ij} representa o fluxo circulando no arco $(i,j) \in A$.

Os vértices s e t representam os vértices fonte e sumidouro da rede.

3.9 Algoritmo de Ford-Fulkerson

Os caminhos de aumento de fluxo são utilizados como a ferramenta básica no algoritmo de Ford-Fulkerson [1] [17], no qual um dado fluxo (por exemplo, fluxo zero em

todas as arestas) é aumentado até que ele seja máximo. O algoritmo acompanha o aumento através de uma construção passo a passo de caminhos de aumento de fluxo, um de cada vez até que nenhum caminho possa ser construído, o que acontece precisamente quando o fluxo é máximo. A Tabela 3.1 mostra o Algoritmo de Ford-Fulkerson.

ALGORITMO FORD-FULKERSON [$N = (V, E)$, vértices $1(=s), ..., n(=t)$, arestas $(i, j), c_{ij}$]

{Este algoritmo calcula o fluxo máximo em uma rede N com fonte s, sumidouro t e capacidades $c_{ij} > 0$ das arestas (i, j).}

ENTRADA: n, $s = 1$, $t = n$, arestas (i, j) de N, c_{ij}

SAÍDA: O fluxo máximo f em N

1. fixar um fluxo inicial f_{ij} {por exemplo, $f_{ij} = 0$ para todas as arestas}, calcular f.
2. rotular s com $\varnothing$. Marque os outros vértices como "não rotulados".
3. encontre um vértice rotulado, porém não examinado i efetuando o exame da seguinte forma:

   para todos os vértices j adjacentes não rotulados, se $c_{ij} > f_{ij}$, calcule

   $$\Delta_{ij} = c_{ij} - f_{ij} \quad e \quad \Delta_j = \begin{cases} \Delta_{1j} & se \quad i = 1 \\ \min(\Delta_i, \Delta_{ij}) & se \quad i > 1 \end{cases}$$

   e rotule j com um "rótulo avançar" (i^+, Δ_j); *ou se* $f_{ji} > 0$, calcule

   $$\Delta_j = \min(\Delta_i, f_{ji})$$

   e rotule j com um "rótulo de retrocesso" (i^-, Δ_j).

   se nenhum j existe então SAÍDA. Pare

   [f é o fluxo máximo.]

senão continue (isto é, vá para o passo 4).

4. repita o Passo 3 até que t seja encontrado.
   {Isto dá um caminho de aumento de fluxo $P: s \to t.$}
   se é impossível chegar a t então SAÍDA f. Pare
   [f é o fluxo máximo.]
   senão
   vá para o passo 5.
5. backtrack o caminho P, usando os rótulos.
   {Regra de Backtracking. Usando a numeração dos vértice de n até 1 (não os rótulos), em cada passo, se um vértice rotulado é encontrado, vá para o próximo vértice que possui o menor número (não o rótulo) entre todos os vértices rotulados $i - 1.$}
6. usando P, aumente o fluxo existente por Δ_t. Ajuste $f = f + \Delta_t.$
7. remova todos os rótulos dos vértices 2, ..., n. vá para o Passo 3.

FIM FORD-FULKERSON.

Tabela 3.1. Algoritmo de Ford-Fulkerson.

No Passo 1, um fluxo inicial pode ser fornecido. No Passo 3, um vértice j pode ser rotulado se existe uma aresta (i, j) com i rotulado e $c_{ij} > f_{ij}$ ("arco direto") ou se existe uma aresta (j, i) com i rotulado e $f_{ji} > 0$ ("arco de retorno").

Para examinar um vértice rotulado i significa rotular todos os vértices não rotulados j adjacentes a i que podem ser rotulados. Antes de localizar um vértice i rotulado, localize todos os vértices que foram rotulados antes de i. Esta estratégia utilizando BFS (Breadth First Search) [1] [17] foi sugerida por Edmonds-Karp apud CORMEN et al. (2002, p. 523). Ela tem o efeito de obter o menor caminho de aumento de fluxo possível.

3.10 Algoritmo de Edmonds-Karp

A Tabela 3.2 apresenta uma implementação do algoritmo de Edmonds-Karp [5] [21].

ALGORITMO EDMONDS-KARP [$N = (V, E)$, vértices $1(=s), ..., n(=t)$, arestas $(i, j), c_{ij}$]

{Este algoritmo é utilizado para calcular o fluxo máximo em uma rede N com fonte s, sumidouro t e capacidades c_{ij} das arestas (i, j)}.

ENTRADA: n, $s = 1$, $t = n$, arestas (i, j) de N, c_{ij}

SAÍDA: fluxo máximo da rede N

1. fixar um fluxo inicial f_{ij} (por exemplo, $f_{ij} = 0$ para todas as arestas).
2. instanciar as variáveis
   para $i = 1$ até n faça
   $valor_i = \infty$;
   $pai_i = 0$;
   $Q = \varnothing$;
   inserir o vértice fonte s em Q;
3. enquanto ($Q \neq \varnothing$) faça
   remover o primeiro vértice de Q que denotamos por u;
   para cada vértice v adjacente a u faça
   se ($N_{uv} > 0$) então
   $\xi = N_{uv} - f_{uv}$
   senão
   se (a rede é limitada) então
   $\xi = f_{vu}$
   senão

$$\xi = f_{vu} + N_{vu}$$

se (o vértice v ainda não foi "visitado") então

marque o vértice v como "visitado";

inserir o vértice v em Q;

se ($valor_v < \xi$) então

$\xi = valor_u$;

se ($valor_v < \xi$) então

$valor_v = \xi$;

$pai_v = u$;

4. executar Backtracking;

{Isto dá um caminho de aumento de fluxo $P : s \to t.$}

mínimo $= \infty$

$y = n$;

$x = pai_n$;

$residual = 0$;

enquanto ($x \neq 0$) faça

se (a $aresta_{xy}$ está avançando) então

$residual = N_{xy} - f_{xy}$;

senão

se (a rede é auxiliar) então

$residual = f_{xy}$;

senão

$residual = f_{yx} + N_{xy}$;

se ($pai_n \neq 0$) então

para cada aresta e do caminho faça

$f_e = f_e +$ mínimo ;

5. backtrack o caminho P, usando os rótulos dos vértices pai_j visitados.

[Regra de Backtracking. Usando a numeração dos vértice

de n até 1 (não os rótulos), em cada passo, se um vértice rotulado é encontrado, vá para o próximo vértice que possui o menor número (não o rótulo) entre todos os vértices rotulados $i - 1$.]

se (é impossível chegar a t) então SAÍDA f. Pare

[f é o fluxo máximo.]

senão continue (isto é, vá para o passo 6).

6. atualiza o fluxo

$f = f +$ mínimo.;

vá para o Passo 2;

FIM EDMONDS-KARP.

Tabela 3.2. Algoritmo de Edmonds-Karp.

No Passo 1, um fluxo inicial pode ser dado. No Passo 2, as estruturas de dados que são utilizadas no algoritmo são instanciadas. No Passo 3, o algoritmo procura por um caminho de aumento de fluxo utilizando BFS. No Passo 4 e 5, o algoritmo faz o backtrack para recuperar o caminho de aumento de fluxo. No Passo 6, o fluxo é incrementado pelo valor do fluxo ao longo do caminho de aumento de fluxo que foi encontrado. O algoritmo executa sucessivas vezes à procura de caminhos de aumento de fluxo. Quando não existir mais caminhos de aumento de fluxo, então, o algoritmo é finalizado.

3.10.1 Casos Testes

Mostramos aqui dois exemplos de redes. A primeira é uma rede capacitada e a segunda é uma rede limitada. Em seguida mostramos os caminhos de aumento de fluxos encontrados através da execução seqüencial do algoritmo.

Exemplo 1.

A Figura 3.2 mostra a rede capacitada N_3.

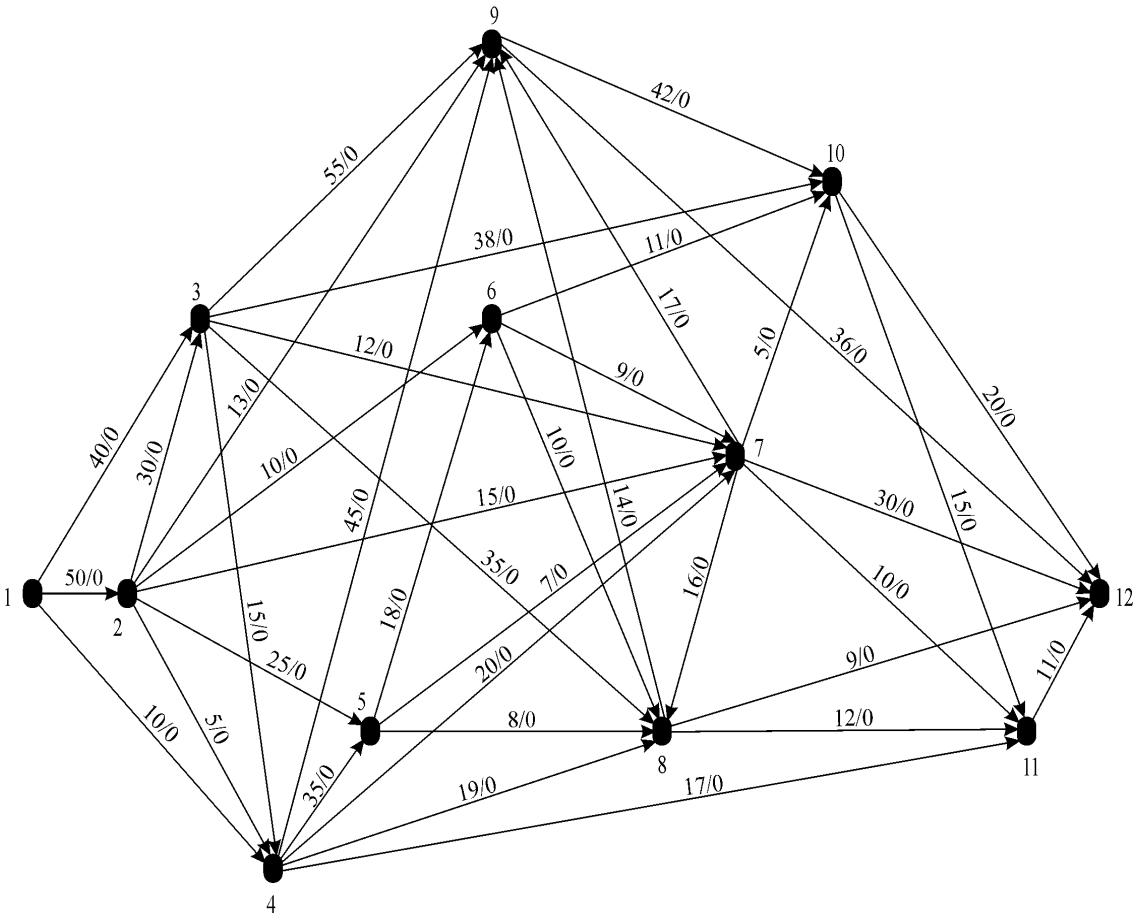


Figura 3.2. Rede Capacitada N_3

A matriz de adjacência a seguir representa a rede capacitada da Figura 3.2.

$$\begin{bmatrix}
0 & 50 & 40 & 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 30 & 5 & 25 & 10 & 15 & 0 & 13 & 0 & 0 & 0 \\
0 & 0 & 0 & 15 & 0 & 0 & 12 & 35 & 55 & 38 & 0 & 0 \\
0 & 0 & 0 & 0 & 35 & 0 & 20 & 19 & 45 & 0 & 17 & 0 \\
0 & 0 & 0 & 0 & 0 & 18 & 7 & 8 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 9 & 10 & 0 & 11 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 17 & 5 & 10 & 30 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 12 & 9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 42 & 0 & 36 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15 & 20 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Os caminhos de aumento de fluxo encontrados foram:

1 -> 2 -> 7 -> 12 = 15.

1 -> 3 -> 7 -> 12 = 12.

1 -> 4 -> 7 -> 12 = 3.

1 -> 3 -> 8 -> 12 = 9.

1 -> 2 -> 9 -> 12 = 13.

1 -> 3 -> 9 -> 12 = 19.

1 -> 4 -> 9 -> 12 = 4.

1 -> 2 -> 6 -> 10 -> 12 = 10.

1 -> 2 -> 5 -> 6 -> 10 -> 12 = 1.

1 -> 4 -> 7 -> 10 -> 12 = 3.

1 -> 2 -> 5 -> 7 -> 10 -> 12 = 2.

1 -> 2 -> 5 -> 7 -> 9 -> 10 -> 12 = 4.

1 -> 2 -> 5 -> 7 -> 11 -> 12 = 1.

1 -> 2 -> 5 -> 6 -> 7 -> 11 -> 12 = 4.

O fluxo máximo encontrado é igual a 100.

Exemplo 2.

A Figura 3.3 mostra a rede limitada N_4.

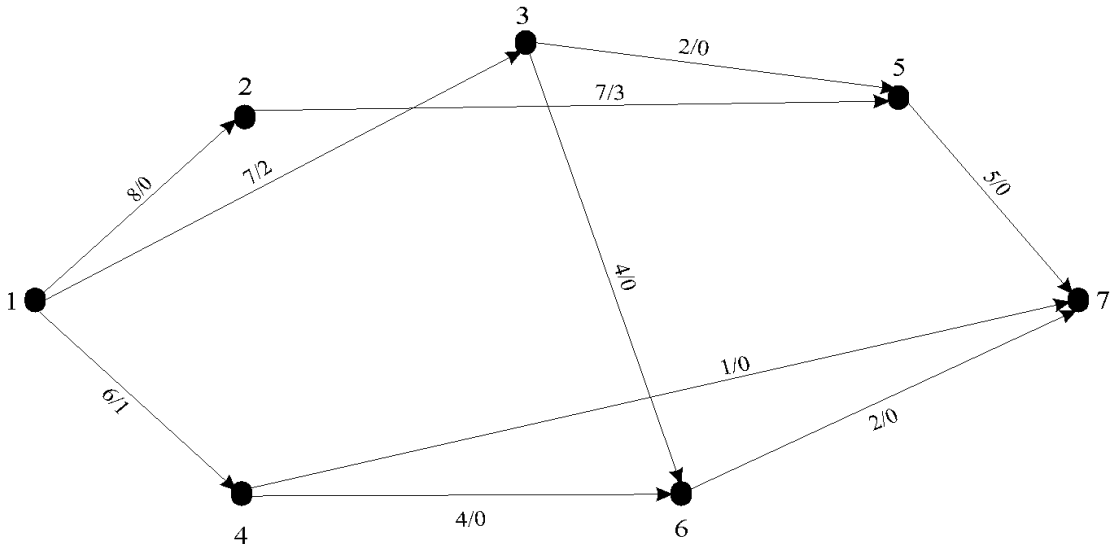


Figura 3.3. Rede Limitada N_4.

A matriz de adjacência a seguir representa os pesos associados ao fluxo da rede limitada da Figura 3.3.

$$\begin{bmatrix} 0 & 8 & 7 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 \\ -2 & 0 & 0 & 0 & 2 & 4 & 0 \\ -1 & 0 & 0 & 0 & 0 & 4 & 1 \\ 0 & -3 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A Figura 3.4 mostra a rede auxiliar N_5.

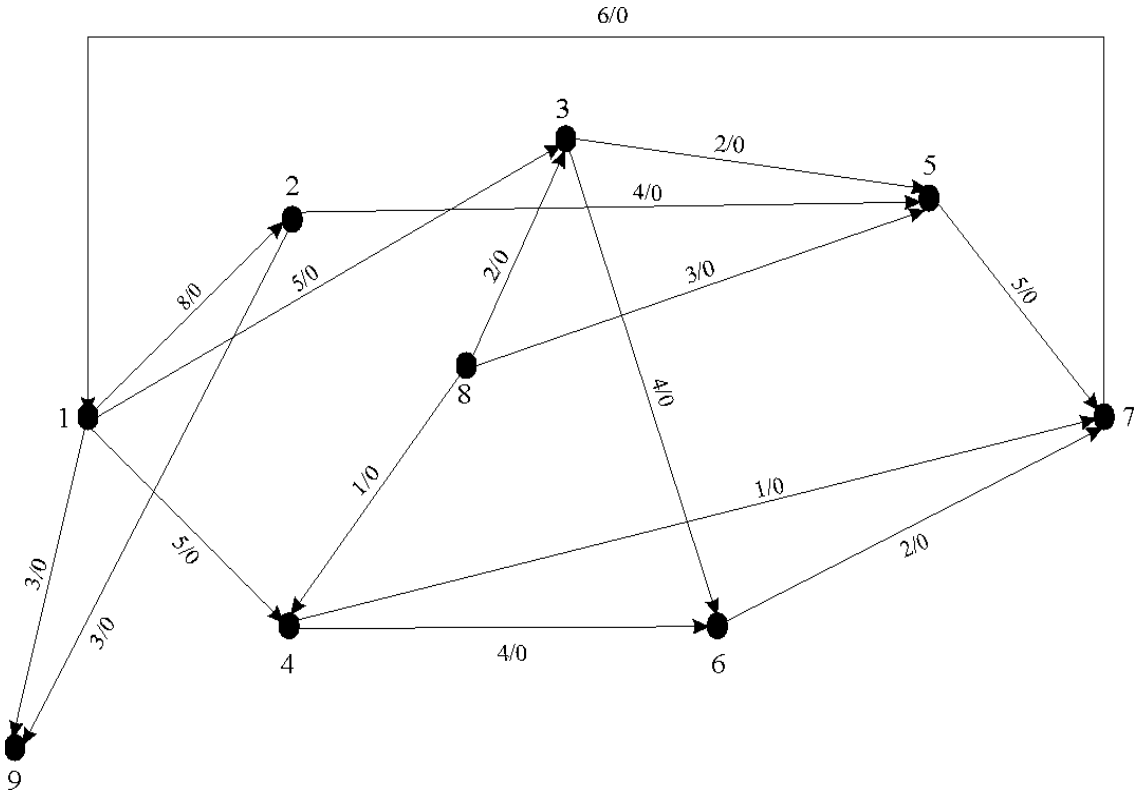


Figura 3.4. Rede Auxiliar N_5.

A matriz de adjacência a seguir representa os pesos associados ao fluxo da rede auxiliar da Figura 3.4.

$$\begin{bmatrix} 0 & 8 & 5 & 5 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 2 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ \infty & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Os caminhos de aumento de fluxo encontrados foram:

8 -> 5 ->7 -> 1 -> 9 = 3.

8 -> 4 -> 7 -> 1 -> 2 -> 9 = 1.

8 -> 3 -> 6 -> 7 -> 1 -> 2 -> 9 = 2.

1 -> 2 -> 5 -> 7 = 2.

O fluxo máximo encontrado é igual a 8.

4 PROBLEMA DO FLUXO DE CUSTO MÍNIMO

Este capítulo tem como objetivo apresentar uma fundamentação teórica para o problema do fluxo de custo mínimo. Neste problema tal como aqui considerado utiliza-se o custo sob qualquer forma racional como critério de otimalidade.

Os dados disponíveis são apenas custos unitários, o que implica em linearidade para que eles tenham sentido. Em princípio, não existe uma relação especificada entre a quantidade transportada e a quantidade máxima possível (ou seja, o fluxo máximo). Portanto, estamos interessados em circular um fluxo em uma rede pagando o menor custo.

4.1 Formulação do Problema do Fluxo de Custo Mínimo

Estamos falando de um problema linear, portanto é interessante mostrar sua formulação no âmbito da programação linear [1] [3]. Estamos considerando que um dado valor ϕ de fluxo, onde queremos minimizar o custo da sua passagem pelo grafo. Podemos formular o problema do fluxo de custo mínimo como um problema de programação matemática da seguinte forma:

$$\text{Minimizar } z(x) = \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{4.1}$$

sujeito a:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} \phi & para\ i = s \\ 0 & para\ todo\ i \in N - \{s,t\} \\ -\phi & para\ i = t \end{cases} \tag{4.2}$$

$$0 \le x_{ij} \le u_{ij} \quad para\ todo\ (i,j) \in A \tag{4.3}$$

Onde os vértices s e t representam os vértices fonte e sumidouro da rede.

Esta formulação, que não considera o arco de retorno, apresenta a vantagem de permitir a especificação do valor ϕ do fluxo envolvido. A especificação do valor do fluxo ϕ

deve ter um valor menor que o fluxo máximo admissível pelo grafo, senão, o problema não terá solução.

Podemos resolver o problema partindo de uma solução inicial de forma a mais econômica, até atingir o valor do fluxo especificado que seria de custo mínimo. Esta abordagem se beneficia naturalmente dos algoritmos de menor caminhos existentes.

Na resolução de problemas de fluxo de custo mínimo devemos fazer uma advertência de que é necessário usar algoritmos que admitam trabalhar com arcos de valor negativo [1] [4]. Um exemplo da técnica que utiliza esta abordagem é o algoritmo de BUSACKER-GOWEN apud BOAVENTURA NETTO (1996, 237).

4.2 Algoritmo de Busacker-Gowen

O algoritmo de Busacker-Gowen [4], considera o conjunto $\mathbb{P}$ de caminhos de menor custo entre s e t em G^f e utiliza um deles para incrementar o valor do fluxo.

A forma aqui apresentada visa a obtenção do menor custo para circular uma quantidade especificada de fluxo ϕ em um grafo. Esta abordagem se beneficia naturalmente dos algoritmos de menor caminhos existentes, os quais são aplicados ao grafo de aumento de fluxo.

A Tabela 4.1 mostra o Algoritmo de Busacker-Gowen para o problema do fluxo de custo mínimo.

ALGORITMO BUSACKER-GOWEN [$N = (V, E)$, vértices $1(=s)$,..., $n(=t)$, arestas $(i, j), c_{ij}, custo_{ij}$]
{Este algoritmo calcula o fluxo de custo mínimo em uma rede N com fonte s, sumidouro t, capacidades $c_{ij} > 0$ e custo do fluxo($custo_{ij}$) das arestas (i, j).}
ENTRADA: n, $s = 1$, $t = n$, arestas (i, j) de N, $c_{ij}, custo_{ij}$
SAÍDA: O menor custo pago pelo tráfego de um

determinado fluxo ϕ em N

1. fixar a quantidade de fluxo ϕ que será transportado através da rede .

   $f = 0$;{é a variável que representa o valor do fluxo corrente}

   $VF = 0$; {é a variável que representa o valor do sistema de fluxos construído até o momento}

   $\xi = 0$;{resíduo de fluxo}

   G^f = FluxoMaximo.Algoritmo ();

2. enquanto($\mathbb{P} \neq \varnothing$) faça

   {determina-se o caminho μ_{st} de menor custo de s a t na rede G^f }

   $\mathbb{P}$ = MenorCaminho.Algoritmo ();

   { incremento de fluxo através de μ_{st} }

   f = Incremento ();

   $VF = VF + f$;

   $\xi = \xi - f$;

   se ($VF <$ *fluxo máximo*) então

   se ($VF < \phi$) então

   atualiza ();

   senão

   $f = \phi - \xi$;

   atualiza ();

   pare;

   senão

   imprima (‘o problema não possui solução’);

   pare;

FIM ALGORITMO-BUSACKER-GOWEN.

Tabela 4.1. Algoritmo de Busacker-Gowen.

No Passo 1, o valor do fluxo ϕ é especificado e o algoritmo de fluxo máximo é executado. No Passo 2, o algoritmo procura caminhos de menor custo entre os vértices s e t até a quantidade especificada de fluxo ϕ seja alcançada. Se o valor do fluxo ϕ possuir um valor maior que a quantidade máxima de fluxo que a rede pode transportar, então, o problema não possui solução.

4.2.1 Caso Teste

A Figura 2.3 mostra a rede N_6. Nesta rede desejamos saber qual é o menor custo para circular uma determinada quantidade de fluxo entre os vértices 1 e 7.

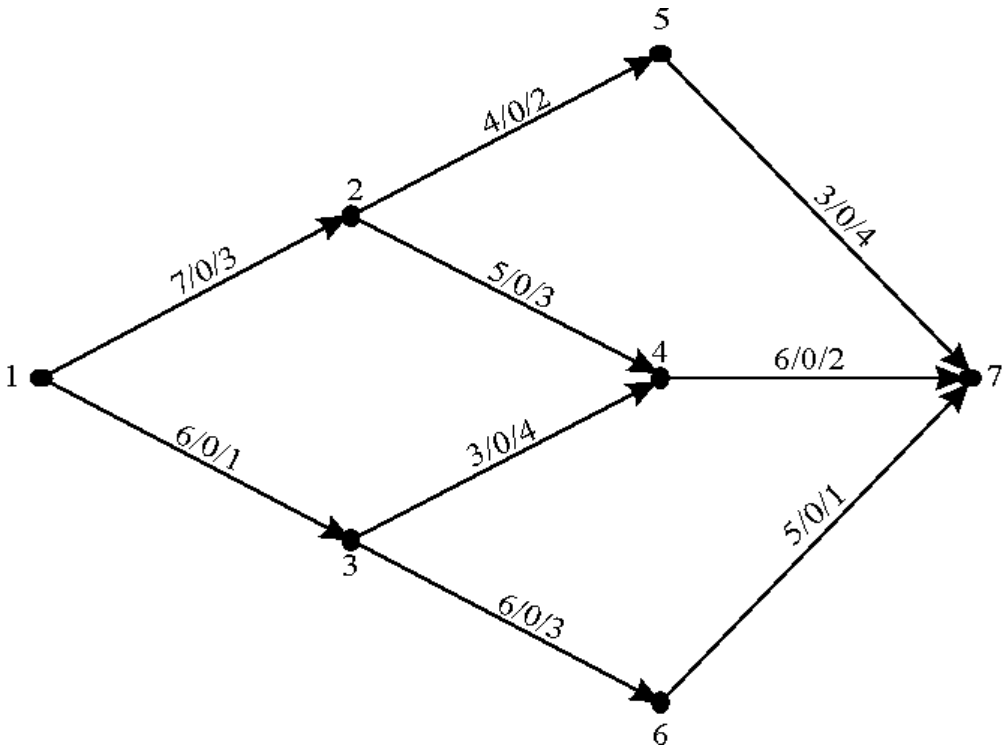


Figura 4.1. Rede N_6.

A matriz de adjacência a seguir representa a rede da Figura 4.1.

$$\begin{bmatrix} 0 & 7 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 4 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A matriz de adjacência a seguir representa os custos associados à rede da Figura 4.1.

$$\begin{bmatrix} 0 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Desejamos enviar $\phi = 12$ (quantidade de fluxo) de $s(1)$ para $t(7)$ pagando o menor custo.

Os caminhos de fluxo de custo mínimo encontrado foram:

O caminho encontrado = ->1->3->6->7.
O fluxo encontrado neste caminho = 5.
O caminho encontrado = ->1->3->4->7.
O fluxo encontrado neste caminho = 1.
O caminho encontrado = ->1->2->4->7.
O fluxo encontrado neste caminho = 5.
O caminho encontrado = ->1->2->5->7.
O fluxo encontrado neste caminho = 1.
O fluxo de custo mínimo na rede = 81.

5 ALGORITMOS PARALELOS

Neste capítulo apresentamos o conceito de algoritmo paralelo, a máquina BSP de Valiant e o modelo de programação paralela XPRAM. Estes algoritmos paralelos possibilitarão analisar a interconexão entre os vértices (nós) em uma LAN.

5.1 Introdução

Os algoritmos seqüenciais relacionados nos Capítulos 1, 2 e 3 são difíceis de paralelizar devido a sua natureza puramente seqüencial. Porém, muitas tarefas computacionais podem ser divididas em subtarefas que são executadas seqüencialmente. Quando estas subtarefas podem ser executadas simultaneamente dizemos que elas são paralelas. Algoritmos paralelos [2] [8] [10] [11] [16] [18] [40] possibilitam a distribuição da computação e, portanto exploram o paralelismo. Os algoritmos paralelos apresentados neste capítulo possibilitarão analisar a interconexão em um ambiente heterogêneo e distribuído fundamentado no modelo XPRAM.

5.2 A Máquina BSP

Valiant apresentou a máquina BSP [26] [38] como um modelo para um computador paralelo de propósito geral. A máquina essencialmente é formada por um conjunto de processadores (P_i), memória (M_i) e pares (ou nós) que se comunicam através de uma rede de passagem de mensagem ponto a ponto. A Figura 5.1, mostra a máquina BSP.

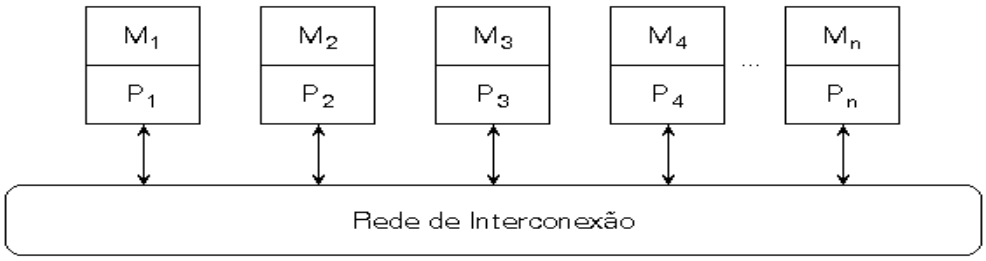


Figura 5.1. Máquina BSP.

Nesta máquina os processadores compartilham dados através de uma memória comum, implementada pela dispersão dos dados através da memória local dos nós. A leitura e a escrita dos dados comuns acontece assíncronamente de forma que pontos de sincronização explícita são necessários em qualquer algoritmo para garantir que novos dados estejam disponíveis para leitura. O modelo de memória é expressamente fracamente coerente, isto é, uma vez que dados comuns somente estariam em um estado coerente em determinados pontos do algoritmo.

Na máquina de Valiant isto acontece somente nos pontos da barreira de sincronização que são capazes de sincronizar todos os processadores ou um subconjunto destes. Esta metodologia divide o algoritmo em um número de superpassos, possibilitando que o código executado esteja entre duas barreiras.

Um algoritmo escrito para aquela máquina específica pode garantir que esta não leia uma variável comum no mesmo superpasso como aquela que estava inicialmente escrita, visto que o resultado estaria indefinido. O modelo de memória é similar ao modelo CREW PRAM [2], mas com a sincronização da extensão do PRAM para superpassos.

A latência de acesso associada a dados comuns é efetivamente escondida através do uso da relaxação paralela de Valiant [26] no algoritmo. Usando este método, um algoritmo é desenvolvido para executar em uma máquina virtual com um número de processadores mais velozes que os disponíveis atualmente. Desta forma os múltiplos processadores virtuais podem ser multiplexados dentro de cada processador físico de modo que estes pode estar aguardando ativo, enquanto uma requisição de dado remota esteja completando. Isto resulta na execução ótima de um algoritmo na máquina.

A leitura concorrente de uma variável comum pode ser executada na máquina BSP ou através de um hardware de interconexão de rede ou usando técnicas de software. O primeiro método de software ordena as requisições concorrentes dentro de grupos [26] (para a

mesma variável) no inicio de cada superpasso, e então usa a operação de prefixo paralelo [2] [26] [8] para enviar uma requisição para cada grupo. Os resultados são distribuídos para todos os membros.

Este método necessita que o algoritmo conheça quais as variáveis são necessárias para o próximo superpasso, isto produz um grande overhead devido à necessidade de ordenar as requisições dentro dos grupos. Mas garante que qualquer requisição combinada será realmente combinada.

O uso de software combinante pode também aumentar o número de barreiras em um algoritmo (reduzindo o comprimento de alguns superpassos) se os processadores simplesmente têm que sincronizar a barreira para ordenar a combinação destas requisições quando elas normalmente seriam executadas assíncronamente.

5.3 O Modelo XPRAM

O modelo XPRAM deseja fornecer um conjunto bem definido de operações escaláveis com um modelo de performance associado. Cada processador XPRAM suporta múltiplos processos, possibilitando o compartilhamento de dados locais e o suporte para os requisitos de relaxação paralela de Valiant.

O paralelismo no modelo XPRAM é baseado na criação de grupos de processos idênticos que podem ser colocados no mesmo processador ou distribuídos através da máquina.

O modelo de memória XPRAM possibilita colocar dados localmente em cada processador (que podem ser compartilhados entre os processos residentes), bem como cada processo tendo acesso a dados privados.

O modelo XPRAM suporta a coerência fraca de memória comum da máquina BSP, pois esta possui uma latência de acesso associada que aumenta com o tamanho da máquina. Devemos enfatizar que a distribuição de todos os tipos de dados comuns, locais e privados existentes no mesmo espaço de endereço compartilhado único, reduz a complexidade do algoritmo resultante. Assim existe somente a semântica de acesso associada

a uma variável particular (isto é, privada, local ou comum) que define a sua semântica de acessibilidade.

A Figura 5.2 mostra o modelo de memória.

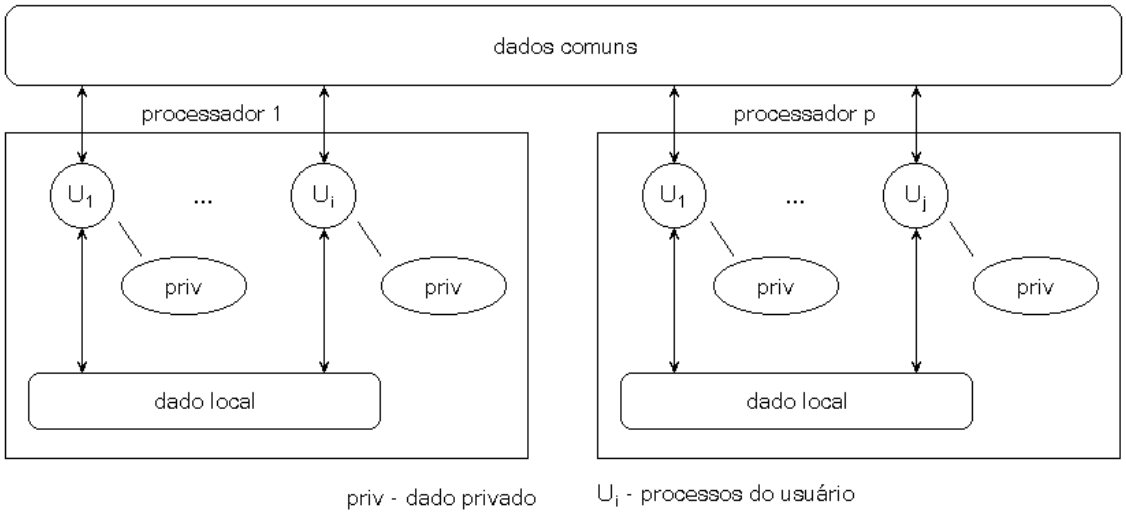


Figura 5.2. Modelo de Memória.

A sincronização dos processos no modelo XPRAM compreende diversas formas. A primeira é a criação de um grupo de processos garantindo a visibilidade dos dados produzidos através da criação de um grupo de processos, para os processos que estão naquele grupo. Outra forma possível é o bloqueio de um processo até que todos os processos em um grupo especificado tenham terminado. Isto pode ser generalizado para o primeiro processo do grupo de um conjunto de grupos. O modelo também corrobora para que um grupo de processo possa sincronizar a barreira como na máquina BSP.

5.4 Metodologia utilizada na Implementação dos Algoritmos Paralelos

A metodologia utilizada na implementação dos algoritmos paralelos é fundamentada no modelo XPRAM. Neste modelo utilizamos a pesquisa BFS em paralelo para percorrer um grafo, pois, esta é considerada o método mais rápido para pesquisar em grafos. Este método de pesquisa pode ser facilmente adaptado para o paralelismo.

Os algoritmos utilizam o método de transformar uma rede qualquer em uma rede de camadas paralelas. A Figura 5.3 mostra como uma rede de camada é construída.

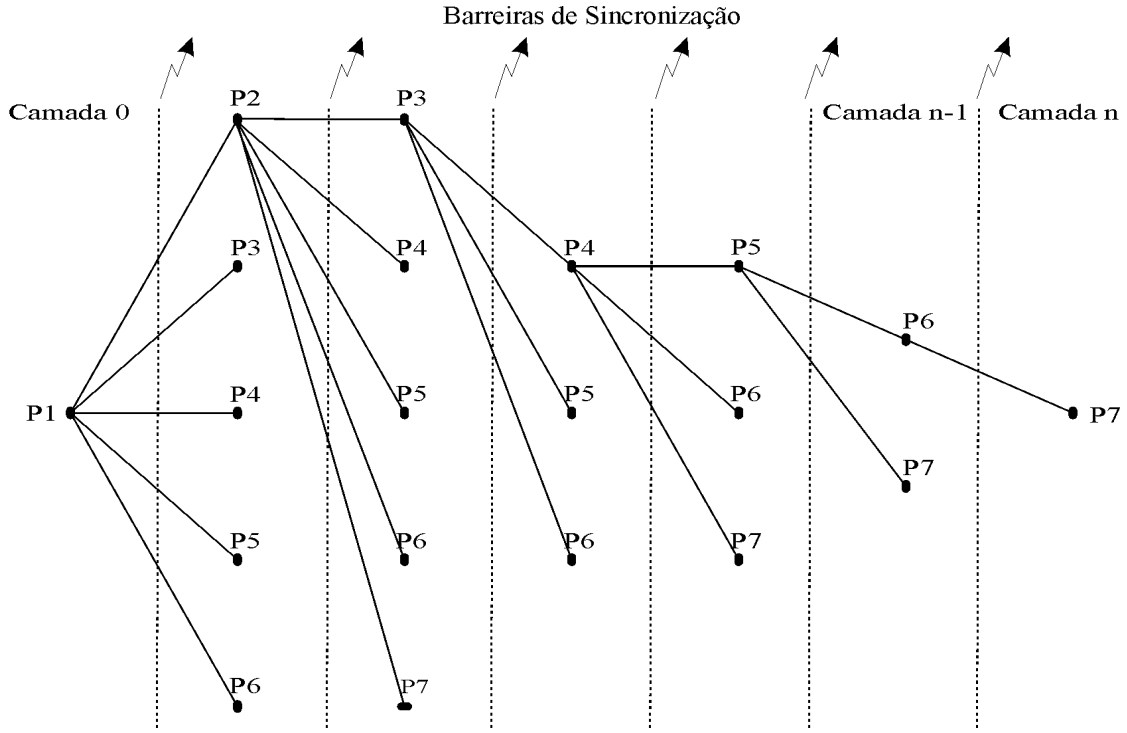


Figura 5.3. Rede de Camadas.

Esta abordagem possui o efeito de substituir um problema de grafos em vários subproblemas, sendo uma boa escolha e é mais fácil que o método seqüencial original.

Para uma rede com n vértices os algoritmos paralelos solucionam a instância do problema de forma mais eficiente e rápida. Iniciamos com a fonte s que contém a camada 0.

A primeira camada é constituída por todos os vértices v tal que existe uma aresta de s para v. Da mesma forma a camada n é produzida da camada $(n-1)$ conectando os vértices da camada $(n-1)$ para a próxima camada como aresta. Estas arestas podem ser diretas ou reversas.

Observe que não existe vértice que se conecte com outro vértice na mesma camada. Obviamente, a última camada somente conterá um vértice que denominamos de sumidouro t. Cada vértice na rede corresponde a um processo preemptivo na rede local.

O método aplicado eficientemente para obter a rede de camada é BFS. De fato, uma pesquisa inicia na fonte revelando a primeira camada 0. Da mesma forma, a camada n é revelada executando uma pesquisa da camada $(n-1)$.

Naturalmente, a BFS aplicada é executada em paralelo. Por exemplo, conforme mostra a Figura 5.3:

- Na camada 0 somente o processo P_1 executaria;
- Na camada 1 os processos P_2, P_3, P_4, P_5 e P_6 executariam;
- Na camada n somente o processo P_7 executaria;

Quando ocorre a passagem de uma camada para outra, utilizamos a barreira de sincronização.

Neste momento ocorre a atualização das estruturas de dados do processo coordenador com as estruturas de dados dos processos escravos que pertencem ao grupo de processos escravos.

O paralelismo e a distribuição que implementamos baseado no modelo XPRAM é baseado na criação de um grupo de processos idênticos (processos escravos) que podem ser colocados no mesmo processador ou distribuídos através de uma LAN.

Na Figura 5.4 apresentamos a aplicação do modelo XPRAM com os algoritmos paralelos que implementamos.

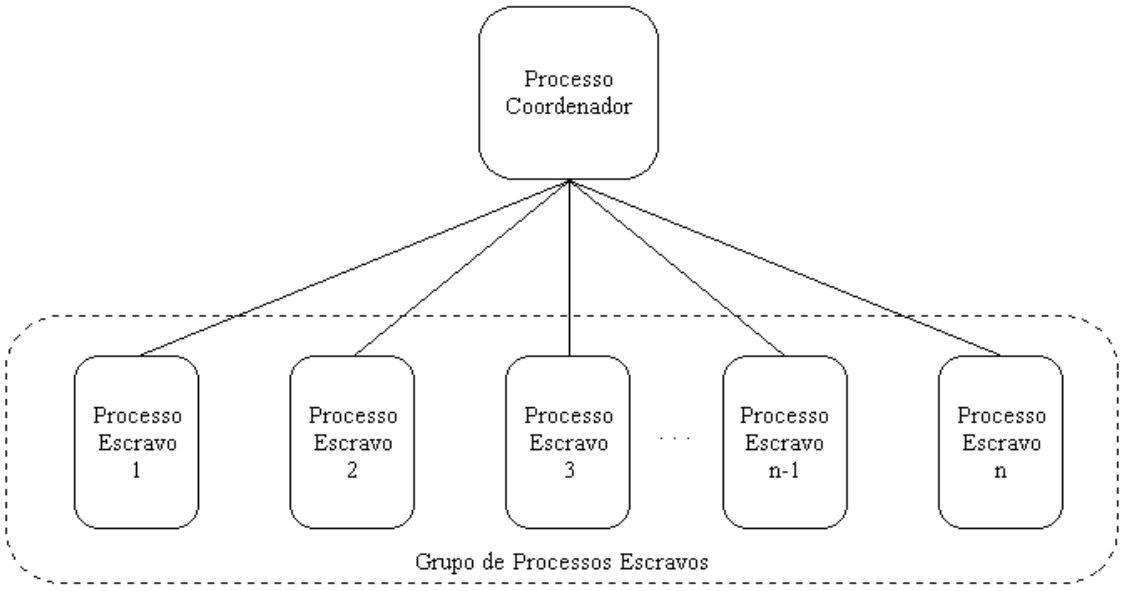


Figura 5.4. Aplicação do Modelo XPRAM relacionado aos Algoritmos Paralelos.

Neste modelo criamos um grupo de processos escravos idênticos. Posteriormente, implementamos um processo coordenador que será responsável em implementar uma política de escalonamento entre os processos que fazem parte do grupo de processos escravos. A política de escalonamento entre os processos é possível, porque o processo coordenador que é o responsável por efetivar o escalonamento entre os processos escravos utiliza duas variáveis *tag´s* .

A primeira variável *tag* é utilizada para implementar a barreira de sincronização do grupo de processos escravos que estão na mesma camada. Esta variável (barreira de sincronização) possibilita que todos os processos escravos que estejam em uma determinada camada mantenham a consistência de suas estruturas de dados com as estruturas de dados do processo coordenador. Conforme podemos observar, este algoritmo paralelo é formado de *n* superpassos como no modelo XPRAM original.

A Figura 5.5 mostra a primeira variável tag controlando as barreiras de sincronização das Camadas.

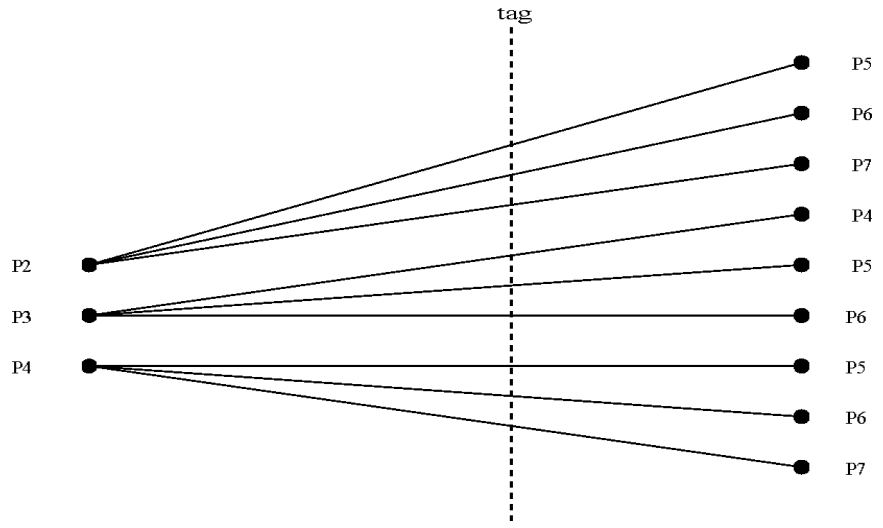


Figura 5.5. Variável *tag* Controlando as Barreiras de Sincronização das Camadas.

A segunda variável *tag* é utilizada pelo processo coordenador para efetivar o sincronismo dos processos escravos que estão executando na mesma camada (nível). Isto possibilita que todos os processos escravos que estão em uma camada específica realizem o mesmo trabalho paralelamente.

A Figura 5.6 mostra a segunda variável *tag* sincronizando a execução paralela do grupo de processos escravos que estão na mesma camada (nível).

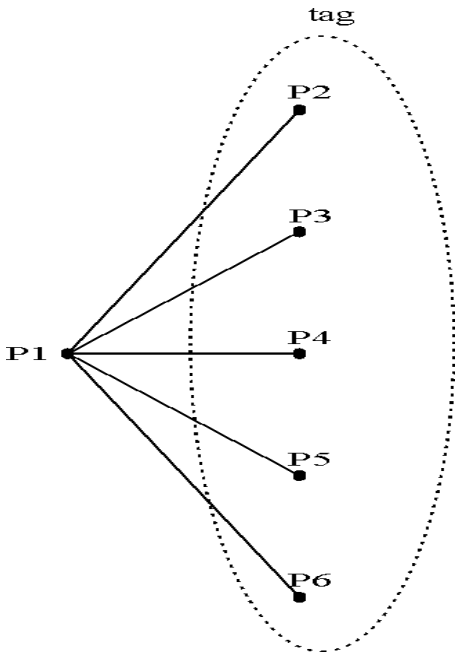


Figura 5.6. Variável *tag* Controlando o Grupo de Processos Escravos presentes na mesma Camada.

5.5 Algoritmo Paralelo (Problema do Menor Caminho)

A implementação do algoritmo paralelo do problema do menor caminho (Dijkstra , Ford-Moore-Bellman) utiliza n processadores.

A Tabela 5.1 descreve o algoritmo paralelo do processo coordenador para o problema do menor caminho.

ALGORITMO PARALELO-MENOR-CAMINHO
$tag1 = 0$; {variável utilizada para sincronizar a barreira de sincronização das camadas}
$tag2 = 0$; {variável utilizada para sincronizar os processos escravos que

executam na mesma camada (nível)}

$key = 0$;

criar grupo de Processos_i escravos;

$Q = \varnothing$; {fila do algoritmo}

$Q_{processoprontoexecução} = \varnothing$; {fila de processos escravos prontos para execução}

inserir s em Q;

inserir o $\text{Pr}ocesso_1$ em $Q_{processoprontoexecução}$;

marcar Processo_1 escravo como "visitado";

enquanto ($Q_{processoprontoexecução} \neq \varnothing$) faça

se ($Q_{processoprontoexecução} \neq \varnothing$) então

$tag1$ =recebe o índice do primeiro processo escravo de $Q_{processoprontoexecução}$;

$tag2 = tag1$;

enquanto ($tag1 = tag2$) faça

key =remover o $\text{Pr}ocesso_{tag1}$ de $Q_{processoprontoexecução}$;

executar o método $A\lg oritmo(Q, pai_v, d_v, d_u, \text{novisitado}_u)$ do Processo_{key} escravo;

BarreiraSincronização ();

se ($val_p \neq 0$)então

executar o método $Backtrack()$ do Processo_p escravo;

senão

imprima('não existe menor caminho entre s e t');

FIM PARALELO-MENOR-CAMINHO.

Tabela 5.1. Algoritmo Paralelo do Processo Coordenador para o Problema do Menor Caminho.

A barreira de sincronização descrita na Tabela 5.1 possibilita sincronizar a atualização em paralelo das estruturas de dados $(Q, pai_v, d_v, d_u, \text{novisitado}_u)$ do processo coordenador com as estruturas de dados $(Q, pai_v, d_v, d_u, \text{novisitado}_u)$ dos processos escravos que estavam presentes na fila de processos de pronto para execução ($Q_{processopronto\text{execução}}$).

Depois que a atualização estiver concluída, ocorre o que denominamos no modelo XPRAM, o superpasso para a próxima camada. Os processos escravos utilizam o algoritmo de Dijkstra usando BFS seqüencial que apresentamos na Tabela 2.3, para implementar o algoritmo de Dijkstra paralelo.

A implementação paralela utiliza a mesma abordagem descrita no algoritmo paralelo de Foord-Moore-Bellman usando BFS descrito na Tabela 5.1.

5.5.1 Caso Teste

A Figura 5.7 mostra a rede N_7.

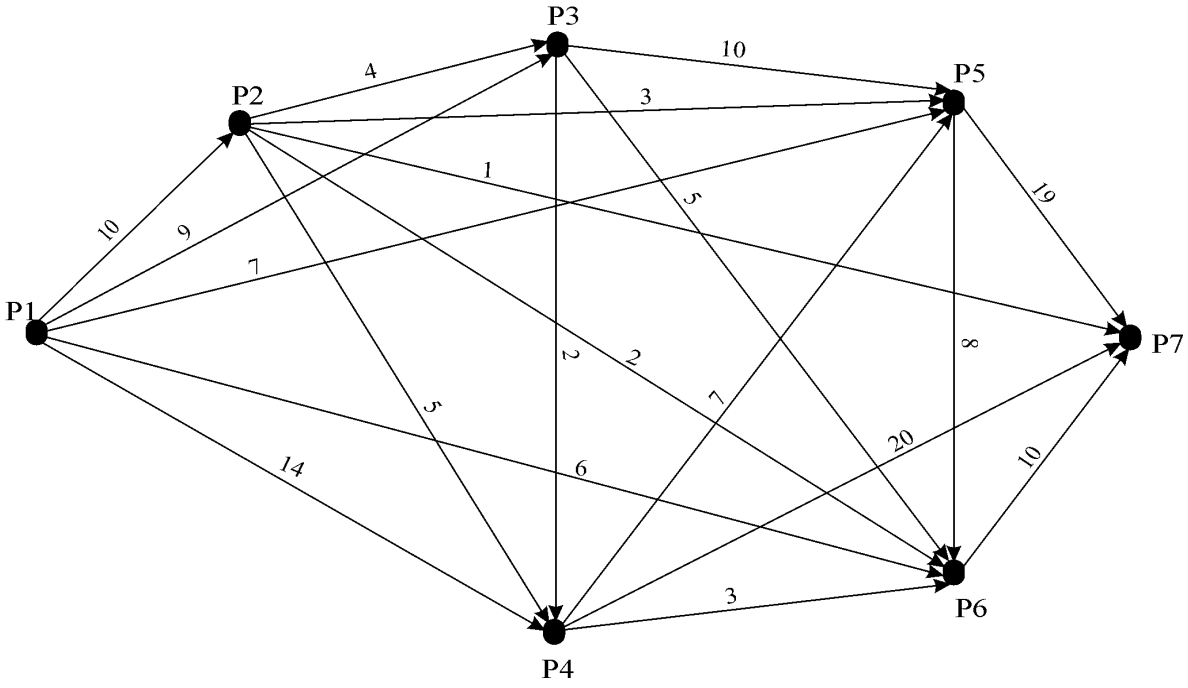


Figura 5.7. Rede N_7.

A matriz de adjacência a seguir representa os pesos (distância) associados às arestas da rede da Figura 5.7.

$$\begin{bmatrix} 0 & 10 & 9 & 14 & 7 & 6 & 0 \\ 0 & 0 & 4 & 5 & 3 & 2 & 1 \\ 0 & 0 & 0 & 2 & 10 & 5 & 0 \\ 0 & 0 & 0 & 0 & 7 & 3 & 20 \\ 0 & 0 & 0 & 0 & 0 & 8 & 19 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A execução do algoritmo paralelo apresenta como resultado:

O menor caminho encontrado na rede N_7 de $s(1)-t(7)$ = 1-> 2 -> 7.

O valor do custo encontrado = 11.

5.6 Algoritmo Paralelo (Problema do Fluxo Máximo)

A implementação do algoritmo paralelo do problema do fluxo máximo (Edmonds-Karp) utiliza n processadores para as redes capacitadas e $n+2$ processadores para as redes limitadas.

Na Tabela 5.2 descrevemos o algoritmo paralelo do processo coordenador para solucionar o problema do fluxo máximo.

ALGORITMO PARALELO-FLUXO-MÁXIMO
$tag1 = 0$; {variável utilizada para sincronizar a barreira de sincronização das camadas}
$tag2 = 0$; {variável utilizada para sincronizar os processos escravos que executam na mesma camada (nível)}
$key = 0$;
criar grupo de $Processos_i$ escravos;

$Q = \varnothing$; {fila do algoritmo}

$Q_{processoprontoexecução} = \varnothing$; {fila de processos escravos prontos para execução}

faça

inserir s em Q;

inserir o Processo_1 escravo em $Q_{processoprontoexecução}$;

marcar Processo_1 escravo como "visitado";

enquanto ($Q_{processoprontoexecução} \neq \varnothing$) faça

se ($Q_{processoprontoexecução} \neq \varnothing$) então

$tag1$ =recebe o índice do primeiro processo escravo de $Q_{processoprontoexecução}$;

$tag2 = tag1$;

enquanto ($tag1 = tag2$) faça

key =remover o $Processo_{tag1}$ de $Q_{processoprontoexecução}$;

executar o método $Algoritmo(Q, pai_v, d_v, d_u, visitado_u)$ do Processo_{key} escravo;

BarreiraSincronização ();

se ($val_p \neq 0$)então

executar o método $Backtrack()$ do Processo_p escravo;

sincronizar a atualização das estruturas de dados ($fluxo_{max imo}$, Q, $fluxo_{ij}$, $valor_i$, $nosvisitados_i$, pai_i) do processo coordenador em paralelo com as estruturas de dados dos processos escravos que estavam em $Q_{processoprontoexecução}$;

para ($i = 1$ *até* n) faça em paralelo

{Atualizar as estruturas de dados do grupo de processos escravos ($fluxo_{max imo}$, Q, $fluxo_{ij}$, $valor_i$, $nosvisitados_i$, pai_i)}

atualizar as estruturas de dados do Processo_i escravo;

enquanto ($val_p \neq 0$);

FIM ALGORITMO PARALELO-FLUXO-MÁXIMO

Tabela 5.2. Algoritmo Paralelo do Processo Coordenador para o Problema do Fluxo Máximo para Rede Capacitada N.

A barreira de sincronização descrita na Tabela 5.2 possibilita sincronizar a atualização em paralelo das estruturas de dados ($fluxo\max imo$, Q, $fluxo_{ij}$, $valor_i$, $nosvisitados_i$, pai_i) do processo coordenador com as estruturas de dados ($fluxo\max imo$, Q, $fluxo_{ij}$, $valor_i$, $nosvisitados_i$, pai_i) dos processos escravos que estavam presentes na fila de processos pronto para execução ($Q_{processoprontoexecução}$).

Na implementação do algoritmo paralelo utilizando a rede limitada, é necessário utilizar a abordagem da rede modificada que descrevemos no Capítulo 3, ou seja, deverão ser feitas algumas modificações no algoritmo paralelo do processo coordenador descrito na Tabela 5.2. Além disso, devemos adicionar dois novos processos ao grupo de processos escravos.

5.6.1 Casos Testes

Mostramos aqui dois exemplos de redes. A primeira é uma rede capacitada e a segunda é uma rede limitada. A execução deste algoritmo possibilita encontrar o fluxo máximo em uma rede utilizando uma abordagem paralela e distribuída. Em seguida mostramos os caminhos de aumento de fluxos encontrados através da execução do algoritmo.

Exemplo 1:

A Figura 5.8 mostra a rede capacitada N_8.

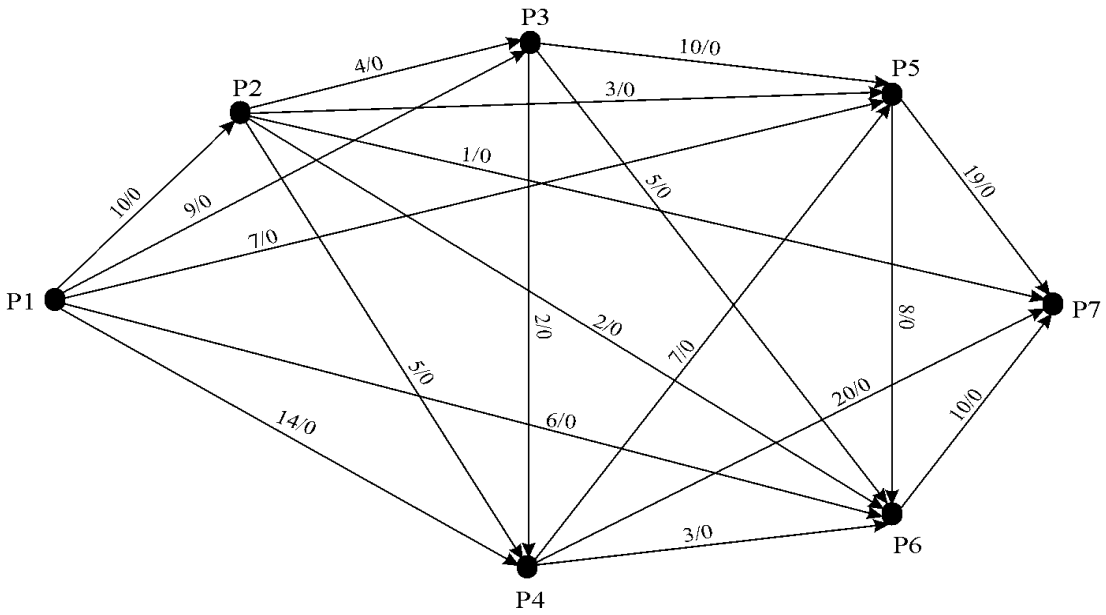


Figura 5.8. Rede Capacitada N_8.

A matriz de adjacência a seguir representa a rede capacitada da Figura 5.8.

$$\begin{bmatrix} 0 & 10 & 9 & 14 & 7 & 6 & 0 \\ 0 & 0 & 4 & 5 & 3 & 2 & 1 \\ 0 & 0 & 0 & 2 & 10 & 5 & 0 \\ 0 & 0 & 0 & 0 & 7 & 3 & 20 \\ 0 & 0 & 0 & 0 & 0 & 8 & 19 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Os caminhos de aumento de fluxo encontrados foram:

1 -> 2 -> 7 = 1.
1 -> 4 -> 7 = 14.
1 -> 5 -> 7 = 7.
1 -> 6 -> 7 = 6.
1 -> 2 -> 4 -> 7 = 5.
1 -> 2 -> 5 -> 7 = 3.
1 -> 2 -> 6 -> 7 = 1.

1 -> 3 -> 4 -> 7 = 1.

1 -> 3 -> 5 -> 7 = 8.

O fluxo máximo encontrado é igual a 46.

Exemplo 2:

A Figura 5.9 mostra a rede limitada N_9.

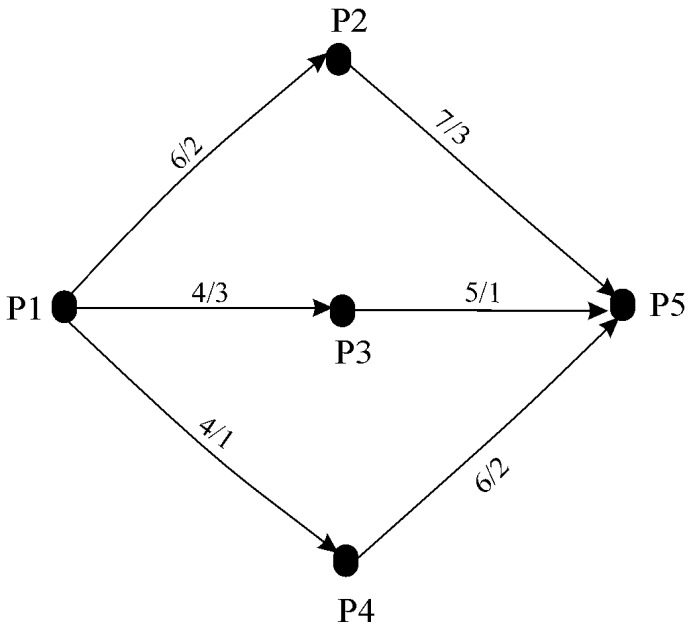


Figura 5.9. Rede Limitada N_9

A matriz de adjacência a seguir representa a rede limitada da Figura 5.9.

$$\begin{bmatrix} 0 & 6 & 4 & 4 & 0 \\ -2 & 0 & 0 & 0 & 7 \\ -3 & 0 & 0 & 0 & 5 \\ -1 & 0 & 0 & 0 & 6 \\ 0 & -3 & -1 & -2 & 0 \end{bmatrix}$$

A Figura 5.10 mostra a rede auxiliar N_{10}.

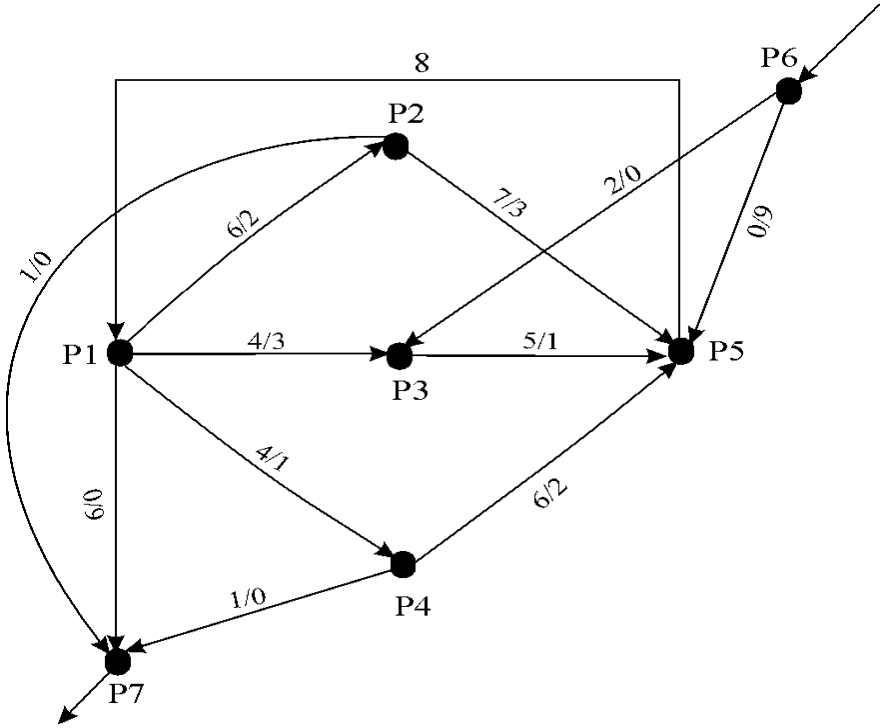


Figura 5.10. Rede Auxiliar N_{10}.

A matriz de adjacência a seguir representa a rede auxiliar da Figura 5.10.

$$\begin{bmatrix} 0 & 4 & 1 & 3 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 1 \\ \infty & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Os caminhos de aumento de fluxo encontrados foram:

6 -> 5 -> 1 -> 7 = 6.

6 -> 3 -> 5 -> 1-> 2 -> 7 = 1.

6 -> 3 -> 5 -> 1 ->4 -> 7 = 1.

1 -> 2 -> 5 = 3.

1 -> 3 -> 5 = 1.

1 -> 4 -> 5 = 2.

O fluxo máximo encontrado é igual a 14.

5.7 Algoritmo Paralelo (Problema do Fluxo de Custo Mínimo)

A paralelização do problema do fluxo de custo mínimo utiliza o algoritmo paralelo descrito Na Tabela 5.1 (algoritmo paralelo para o problema do menor caminho) e o algoritmo paralelo descrito na Tabela 5.2 (algoritmo paralelo para o problema do fluxo máximo). Os processos escravos utilizam a implementação dos algoritmos seqüenciais apresentados nas Tabelas 2.5 e 3.2.

A Figura 5.11 mostra a metodologia utilizada para implementar este algoritmo.

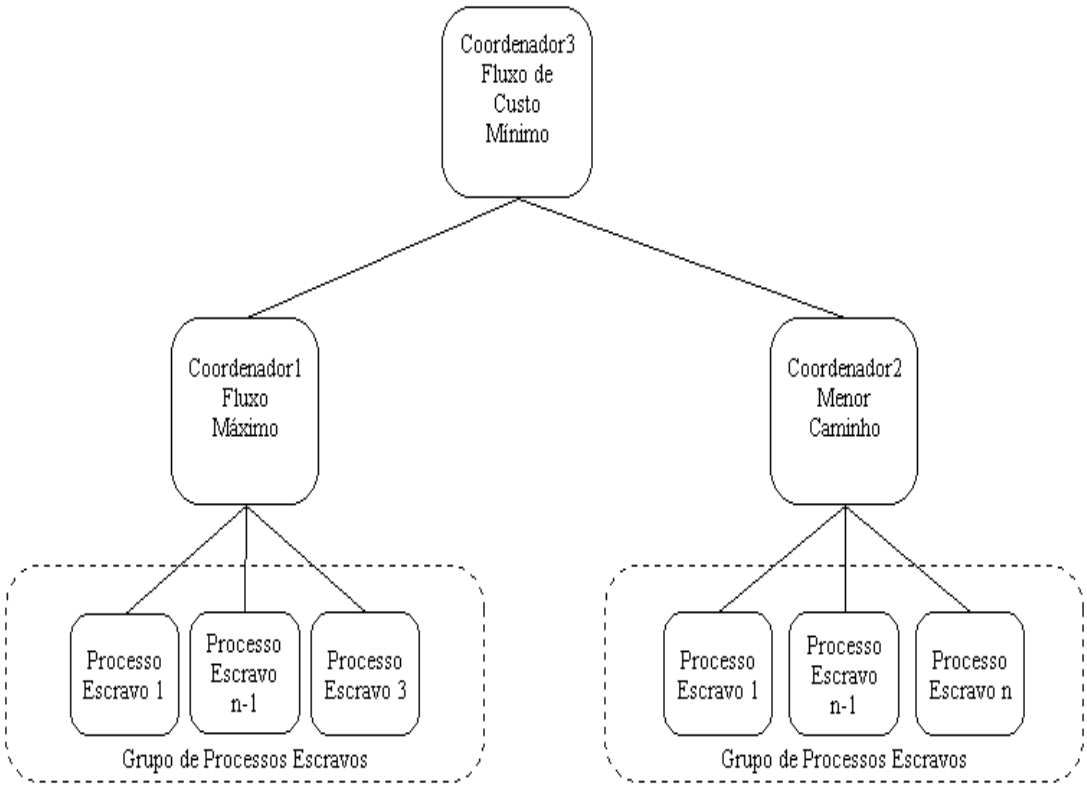


Figura 5.11. Metodologia utilizada para implementar o Algoritmo Paralelo do Processo Coordenador do Fluxo de Custo Mínimo.

O processo coordenador do fluxo de custo mínimo controla os processo coordenador do fluxo máximo e o processo coordenador do menor caminho descritos nas Tabelas 5.1 e 5.2 respectivamente.

A Tabela 5.3 mostra o algoritmo paralelo do processo coordenador para solucionar o problema do fluxo de custo mínimo (Busacker-Gowen).

ALGORITMO PARALELO-DE FLUXO DE CUSTO MÍNIMO
ALGORITMO-PARALELO-FLUXO-MÁXIMO Coordenador1;
ALGORITMO-PARALELO-MENOR-CAMINHO Coordenador2;

1. fixar a quantidade ϕ que será transportado através da rede;

criar o grupo de processos escravos para o problema do menor caminho;

criar o grupo de processos escravos para o problema do fluxo máximo;

instanciar o processo Coordenador1 paralelo para o problema do fluxo máximo;

instanciar o processo Coordenador2 paralelo para o problema do menor caminho;

$f = 0$; {é a variável que representa o valor do fluxo corrente}

$VF = 0$; {é a variável que representa o valor do sistema de fluxos construído até o momento}

$\xi = 0$; {resíduo de fluxo}

$G^f = Coordenador1.Algoritmo(N_{ij})$;

2. enquanto($\mathbb{P} \neq 0$) faça

{Determina-se o caminho μ_{st} de menor custo de s a t na rede G^f}

$\mathbb{P} = Coordenador2.Algoritmo(custo_{ij})$;

$f = \text{Incremento}()$;{ incremento de fluxo através de μ_{st} }

$VF = VF + f$;

```
                    ξ = VF − f ;
                se (VF <  fluxo máximo  )então
                   se (ξ ≤ ϕ ) então
                       atualiza ();
                       senão
                            f = ϕ − ξ ;
                            atualiza ();
                            pare;
                senão
                    imprima (´problema não possui solução´);
                    pare;
FIM ALGORITMO-PARALELO-FLUXO-CUSTO-MÍNIMO .
```

Tabela 5.3. Algoritmo Paralelo do Processo Coordenador para o problema do Fluxo de Custo Mínimo.

5.7.1 Caso Teste

A Figura 5.12 mostra a rede N_{11}.

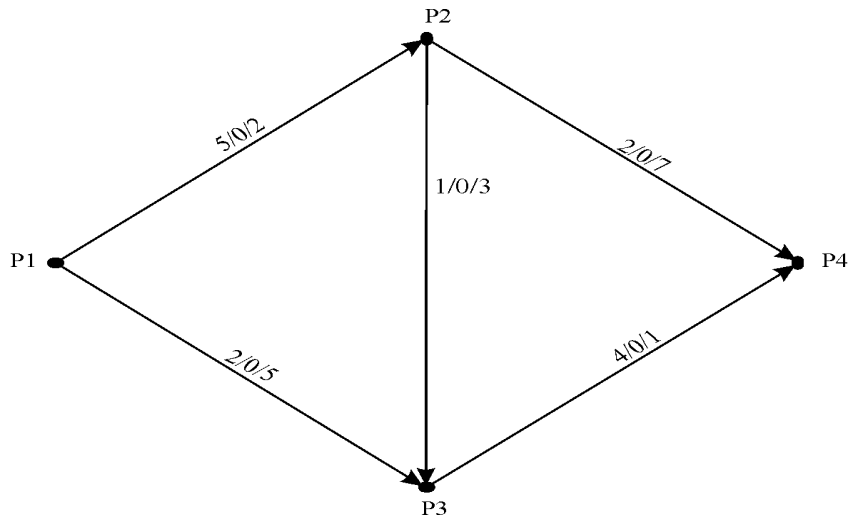


Figura 5.12. Rede N_{11}.

A matriz de adjacência a seguir representa a rede da Figura 5.12.

$$\begin{bmatrix} 0 & 5 & 2 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

A matriz de adjacência a seguir representa os custos associados às arestas da rede da Figura 5.12.

$$\begin{bmatrix} 0 & 2 & 5 & 0 \\ 0 & 0 & 3 & 7 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Desejamos enviar $\phi = 4$ (quantidade de fluxo) de $s(1)$ para $t(4)$ pagando o menor custo.

Os caminhos de fluxo de custo mínimo encontrado foram:

O caminho encontrado = ->1->3->4.
O fluxo encontrado neste caminho = 2
O caminho encontrado = ->1->2->3->4.
O fluxo encontrado neste caminho = 1
O caminho encontrado = ->1->2->4.
O fluxo encontrado neste caminho = 1
O fluxo de custo mínimo = 27

A implementação de algoritmos seqüenciais ou paralelos envolvendo a teoria de grafos ou fluxos em redes não é uma tarefa trivial [1] [2] [6] [11] [19] [20] [21] [22] [31] [33] [37] [39] [41].

A utilização dos algoritmos paralelos corrobora para fundamentar o modelo teórico proposto.

Nesta dissertação, utilizamos uma sistemática de uma rede de computador (LAN), ou seja, os vértices (nós) no modelo teórico são representados por computadores e as arestas por interconexões (topologia da rede).

6 DESENVOLVIMENTO DA INTERFACE

A interface de análise da interconexão em uma LAN foi desenvolvida para executar em um ambiente heterogêneo e distribuído para as seguintes plataformas de sistemas operacionais: Windows [95-98-NT-2000], Unix (Solaris, AIX), Linux (Conectiva, Red Hat) utilizando uma memória compartilhada e distribuída conforme descrito no Capítulo 5. Por isso, utilizamos a linguagem de programação Java [9] [14] [15] adotamos a especificação CORBA para a criação de objetos distribuídos e interoperáveis. O padrão CORBA fornece interoperabilidade entre clientes e servidores heterogêneos, permitindo a comunicação entre aplicações escritas em diferentes linguagens de programação.

Neste capítulo descrevemos os componentes de software utilizados no desenvolvimento da interface de análise da interconexão em uma LAN.

6.1 Arquitetura de Agente de Requisição de Objeto Comum (Common Object Request Broker Architecture – CORBA)

A Arquitetura CORBA [13] [23] [24] [25] [27] [28] é um middleware (camada de software intermediária) que permite a comunicação entre aplicação cliente/ servidor heterogêneas. CORBA possibilita que aplicações escritas em diferentes linguagens de programação se comuniquem usando uma linguagem de definição de interface (IDL) e um Object Request Broker [13] [23] [27].

A IDL possibilita que um objeto distribuído [9] [13] [27] descreva uma interface para os serviços que fornece. A utilização da IDL possibilita separar a definição de objeto da implementação, ou seja, um cliente visualiza um objeto servidor somente através da interface IDL. A IDL é uma linguagem neutra e puramente declarativa. Isto é essencial para garantir a interoperabilidade do padrão CORBA. As interfaces escritas em IDL podem ser mapeadas para interfaces em qualquer linguagem de programação. Em outras palavras, um cliente escrito em Java pode estabelecer comunicação com um servidor escrito em C++, Delphi, etc. O objeto servidor implementa a interface especificada pela IDL.

O ORB é o alicerce de CORBA, pois, possibilita que os clientes invoquem métodos em objetos distribuídos [13] e aceitem valores de retorno. Existe um ORB no cliente e no servidor. O protocolo utilizado para a comunicação entre os ORB´s é o Internet InterORB Protocol [23] [27].

Quando uma aplicação cliente solicita uma referência para um objeto remoto é responsabilidade do ORB do cliente localizar o objeto remoto no sistema distribuído [9] [13]. O ORB do cliente também é responsável por rotear às invocações de método remoto para o servidor apropriado e por aceitar os resultados do servidor.

O ORB do servidor permite que o servidor registre novos objetos CORBA e também é responsável por aceitar pedidos de ORB´s de clientes para chamar métodos no servidor. O ORB do servidor também devolve valores de retorno para o cliente.

Um sistema CORBA funciona da seguinte maneira: assim que um cliente tiver uma referência a um objeto remoto, qualquer invocação de método para esse objeto é feita através do stub cliente. Esse stub utiliza o ORB do cliente para se comunicar com o servidor. Quando o ORB do servidor recebe um pedido do cliente para invocar um método, ele chama o skeleton apropriado, que, por sua vez, invoca a implementação do método remoto no servidor. Os valores de retorno são devolvidos pelo mesmo caminho. A Figura 6.1 mostra a invocação de método remoto.

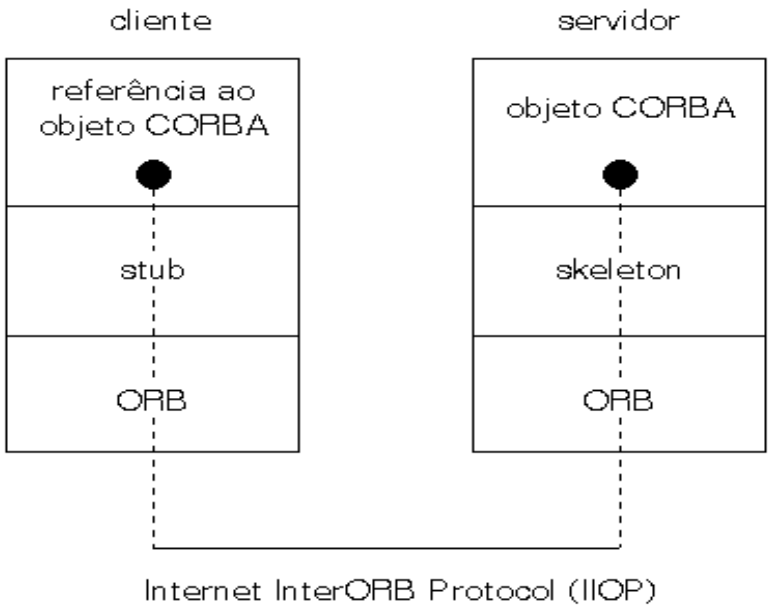


Figura 6.1. Invocação de Método Remoto.

O padrão CORBA fornece interoperabilidade entre clientes e servidores heterogêneos, permitindo a comunicação entre aplicações escritas em diferentes linguagens de programação. CORBA é um sistema de objetos distribuídos [9] [13], ou seja, os objetos [5] [32] são distribuídos em uma rede de computador [7] [35] [36] e as aplicações podem se comunicar com estes objetos.

Um recurso importante deste padrão é o serviço de registro de objeto, que possibilita o registro do objeto. As aplicações que desejam utilizar um objeto distribuído obtêm uma referência ao objeto através deste serviço.

Em um sistema CORBA, o ORB no servidor é responsável por fornecer o serviço de registro. A função do ORB é unificar o acesso a serviços de aplicação.

6.2 O Protocolo Internet Inter-ORB (IIOP)

A implementação do ORB é específica do fabricante. O IIOP foi introduzido para facilitar a interoperabilidade do ORB. Este possibilita que os objetos ORB de um fabricante se comuniquem com objetos ORB de outro através de uma rede TCP/IP. O IIOP possibilita usar a internet como backbone para os ORB´s comunicarem-se.

6.3 Ambiente de Desenvolvimento Integrado (Integrated Development Environment – IDE)

O IDE é um pacote de software que possui uma grande variedade de recursos de compilação, execução e depuração, que possibilita ao engenheiro de software desenvolver sua aplicação CORBA.

Na implementação da Interface de Análise da Interconexão em uma LAN usando CORBA utilizamos o IDE Borland App Server 4.5.

No IDE Borland App Server 4.5, temos o Visibroker que é um ORB CORBA Versão 2.3. Este ORB suporta o desenvolvimento e o gerenciamento de aplicações de objetos

distribuídos. Três componentes adicionais acompanham o Borland AppServer 4.5, que são: o serviço de nome, o serviço de evento e o gatekeeper.

O serviço de nome associa um ou vários nomes lógicos com a implementação de um objeto e armazena estes nomes em um espaço de nomes. Ele também permite que aplicações clientes usem este serviço para obter uma referência a um objeto usando o nome lógico atribuído àquele objeto.

O serviço de evento apresenta facilidades que separa a comunicação entre objetos. Ele fornece um modelo de comunicação produtor-consumidor possibilitando que múltiplos objetos fornecedores enviem dados assíncronamente para múltiplos objetos consumidores, através de um canal de eventos.

O gatekeeper executa em um servidor web e possibilita que programas clientes localizem e usem objetos que não residem no servidor web e recebam chamadas de retorno, até mesmo quando firewalls estão sendo utilizados. Ele pode ser utilizado como um daemon HTTP, eliminando a exigência assim de um servidor de HTTP separado durante a fase de desenvolvimento da aplicação.

6.4 A Interface

No desenvolvimento da Interface de Análise da Interconexão, utilizamos a tecnologia de objetos distribuídos CORBA e no desenvolvimento da interface gráfica do usuário utilizamos o swing [14] [39] do Java. O IDE que utilizamos para o desenvolvimento das aplicações distribuídas foi o Borland App Server 4.5.

Os algoritmos utilizados no desenvolvimento da interface para analisar a interconexão conforme proposto pelo modelo teórico foram: o algoritmo do fluxo máximo (throughput) e o algoritmo de fluxo de custo mínimo (latência).

A Figura 6.2 mostra a Interface de Análise da Interconexão em uma LAN usando CORBA.

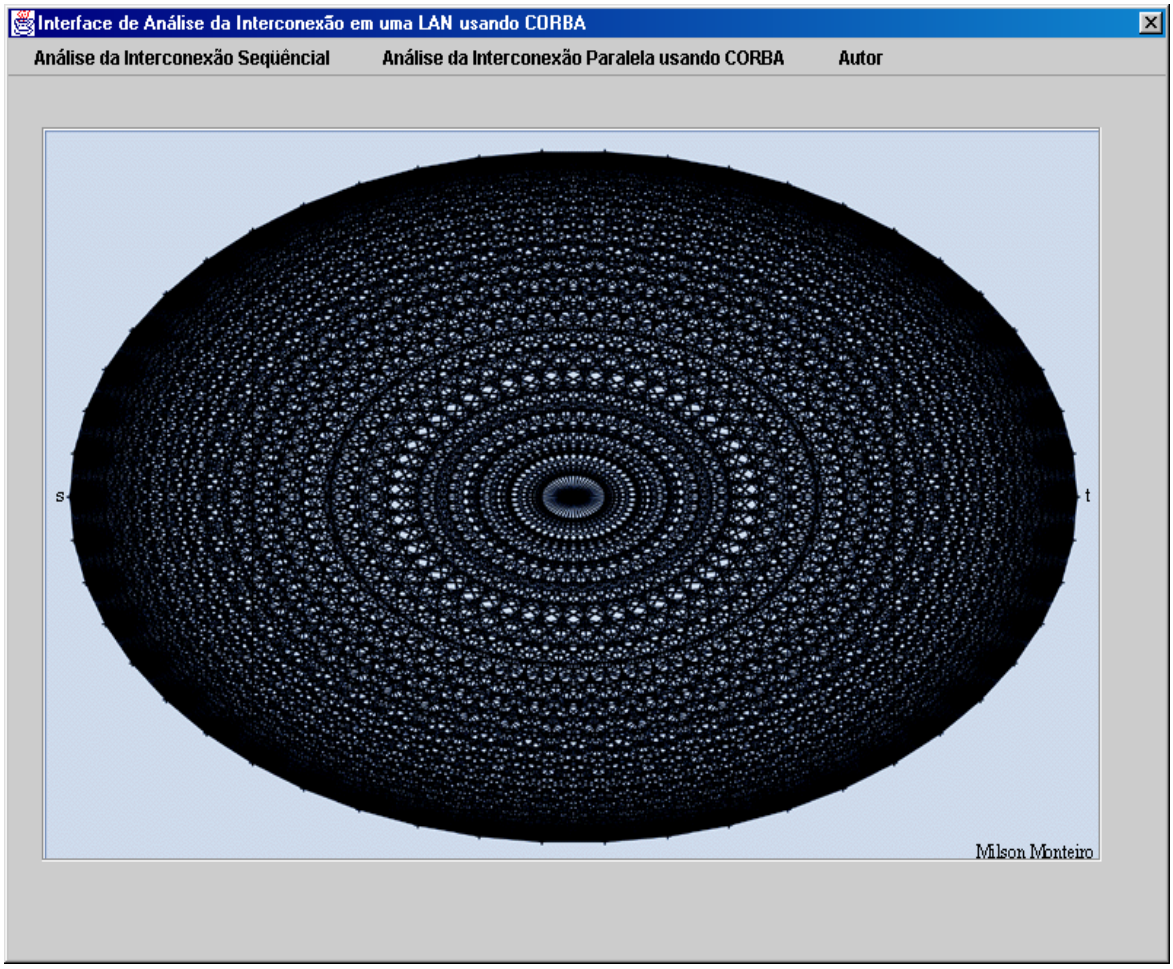


Figura 6.2. Interface de Análise da Interconexão em uma LAN usando CORBA.

A interface de análise da interconexão em uma LAN usando CORBA possibilita que o analista selecione o modelo de análise que deseja realizar.

A interface disponibiliza duas formas de análise, que são: a análise seqüencial e a análise paralela.

A análise seqüencial possibilita analisar três tipos diferentes de problemas da teoria de grafos, que são: menor caminho (Dijkstra e Ford-Moore-Bellman), fluxo máximo (Edmonds-Karp) e fluxo de custo mínimo (Busacker-Gowen).

A análise paralela, possibilita analisar os problemas descritos nos Capítulo 2, 3 e 4 utilizando um modelo de programação seqüencial e paralela. A programação paralela é fundamentada no modelo XPRAM que foi apresentado no Capítulo 5.

A Figura 6.3 mostra a interface de análise seqüencial da melhor rota. Esta interface utiliza o algoritmo de Dijkstra usando BFS que apresentamos no Capítulo 2.

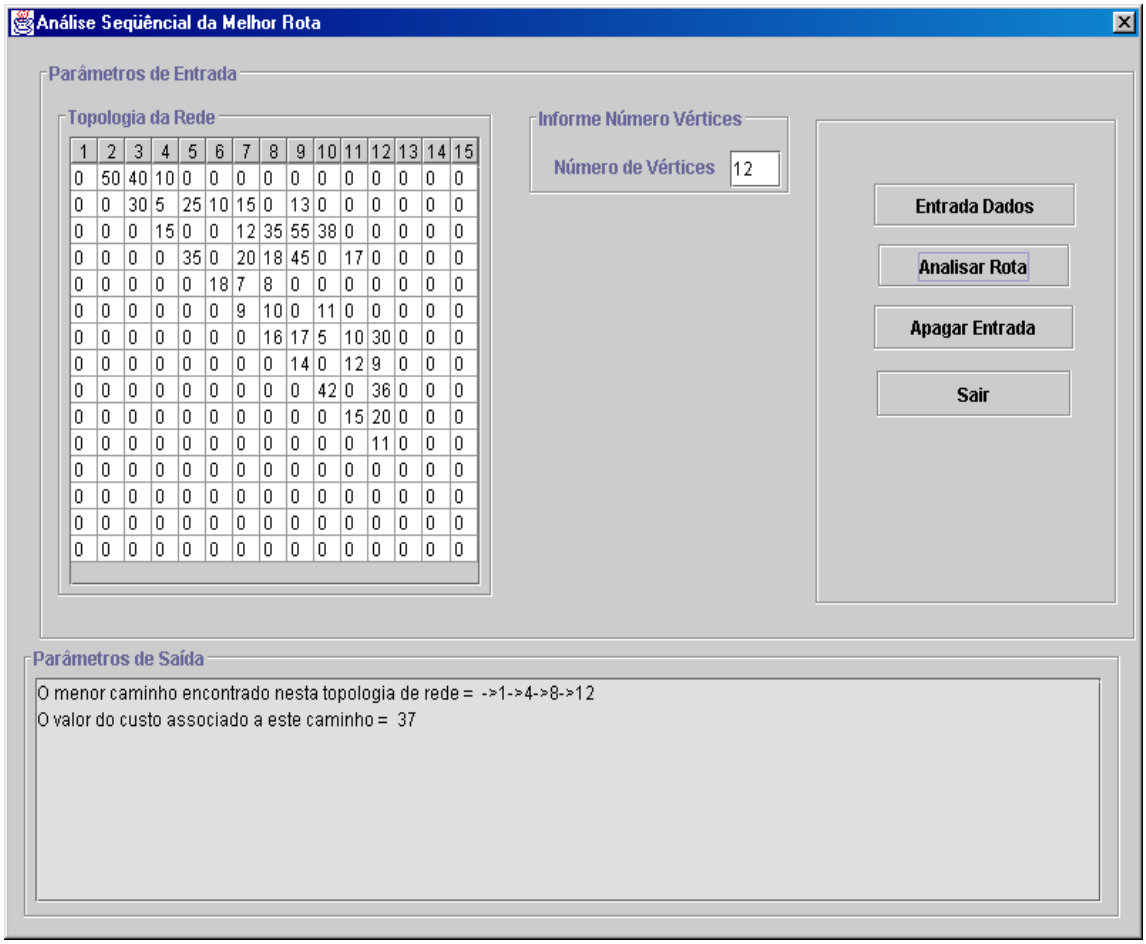


Figura 6.3. Interface de Análise Seqüencial da Melhor Rota.

Esta interface possui uma matriz de incidência que permite que o analista insira a topologia da rede que será analisada.

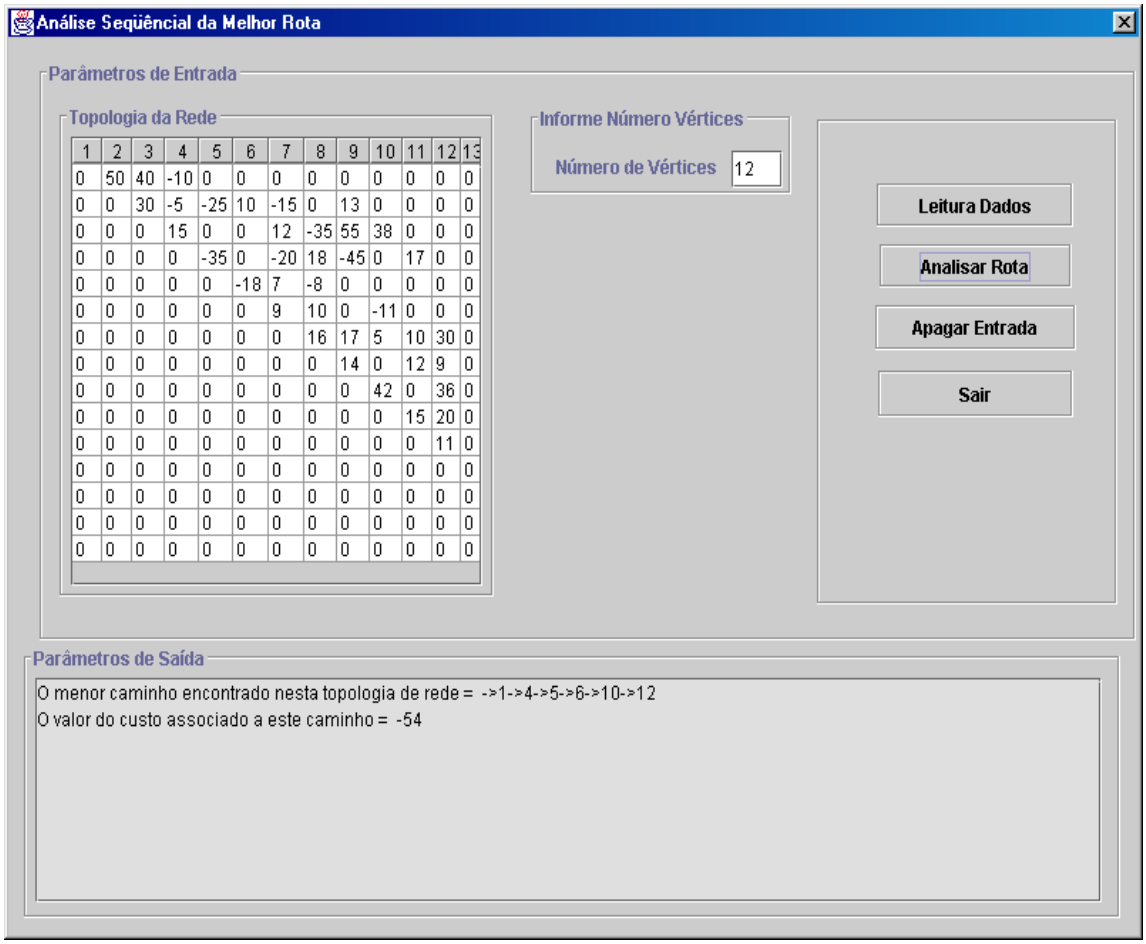
O número de vértices da rede deverá também ser informado. O botão Entrada de Dados efetua a entrada das informações para o algoritmo.

O botão Analisar Rota executa o algoritmo e mostra o resultado.

O botão Apagar Entrada elimina os parâmetros de entrada e saída da interface. O botão Sair retorna para a interface principal.

Na implementação desta interface utilizamos o algoritmo de Ford-Moore-Bellman usando BFS que apresentamos no Capítulo 2.

A Figura 6.4 mostra a interface de análise seqüencial da melhor rota.



1	2	3	4	5	6	7	8	9	10	11	12	13
0	50	40	-10	0	0	0	0	0	0	0	0	0
0	0	30	-5	-25	10	-15	0	13	0	0	0	0
0	0	0	15	0	0	12	-35	55	38	0	0	0
0	0	0	0	-35	0	-20	18	-45	0	17	0	0
0	0	0	0	0	-18	7	-8	0	0	0	0	0
0	0	0	0	0	0	9	10	0	-11	0	0	0
0	0	0	0	0	0	0	16	17	5	10	30	0
0	0	0	0	0	0	0	0	14	0	12	9	0
0	0	0	0	0	0	0	0	0	42	0	36	0
0	0	0	0	0	0	0	0	0	0	15	20	0
0	0	0	0	0	0	0	0	0	0	0	11	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 6.4. Interface de Análise Seqüencial da Melhor Rota.

Esta interface possui uma matriz de incidência que permite que o analista insira a topologia da rede que será analisada.

O número de vértices da rede deverá também ser informado. O botão Entrada de Dados efetua a entrada das informações para o algoritmo.

O botão Analisar Rota executa o algoritmo e mostra o resultado.

O botão Apagar Entrada elimina os parâmetros de entrada e saída da interface. O botão Sair retorna para a interface principal.

Na implementação desta interface utilizamos o algoritmo de Edmonds-Karp que apresentamos no Capítulo 3.

A Figura 6.5 mostra a interface de análise seqüencial do throughput.

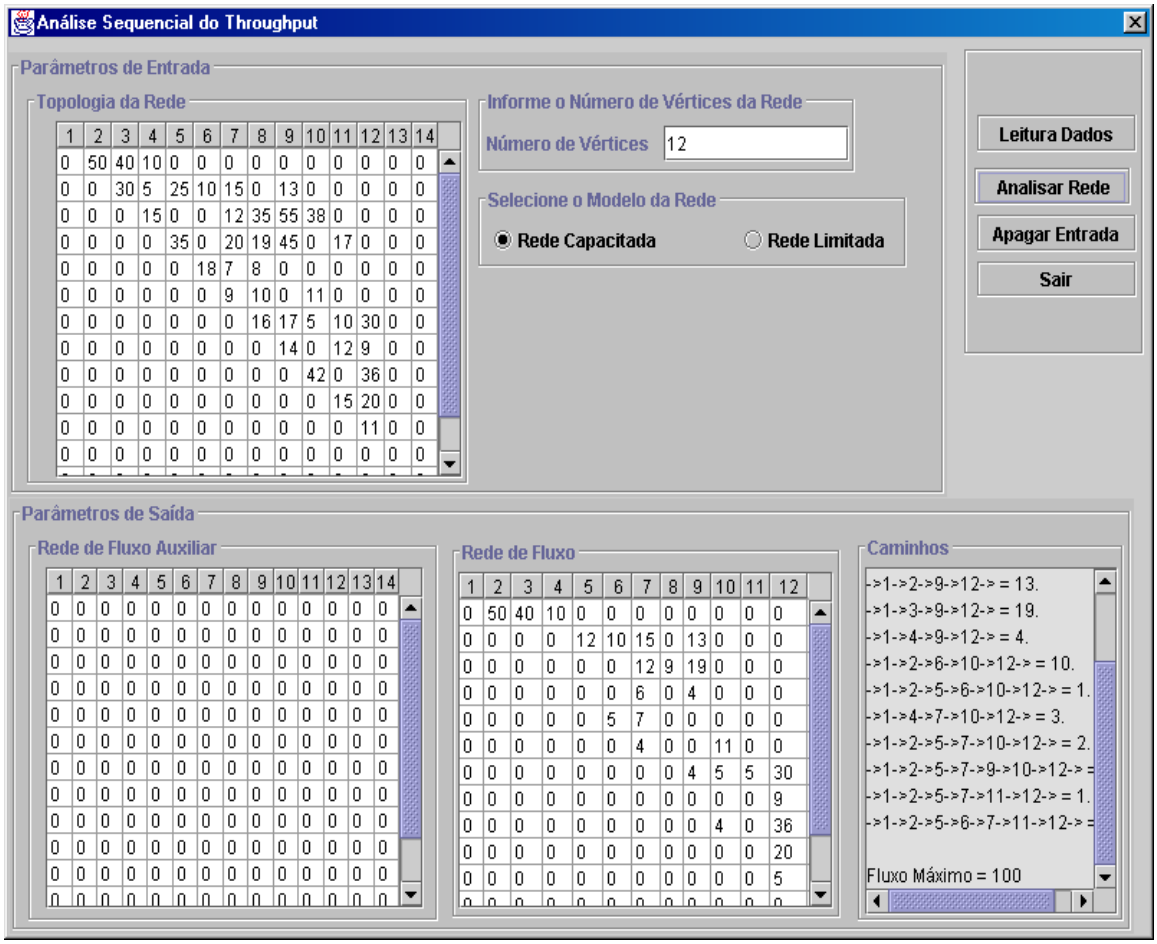


Figura 6.5. Interface de Análise Seqüencial do Throughput.

Esta interface possui uma matriz de incidência que permite que o analista insira a topologia da rede que será analisada.

O número de vértices da rede e o seu modelo (capacitada ou limitada) deverá também ser informado.

O botão Entrada de Dados efetua a entrada das informações para o algoritmo. O botão Analisar Rede executa o algoritmo e mostra o resultado.

O botão Apagar Entrada elimina os parâmetros de entrada e saída da interface. O botão Sair retorna para a interface principal.

Na implementação desta interface utilizamos o algoritmo de Busacker-Gowen que apresentamos no Capítulo 4.

A Figura 6.6 mostra a interface de análise seqüencial da latência.

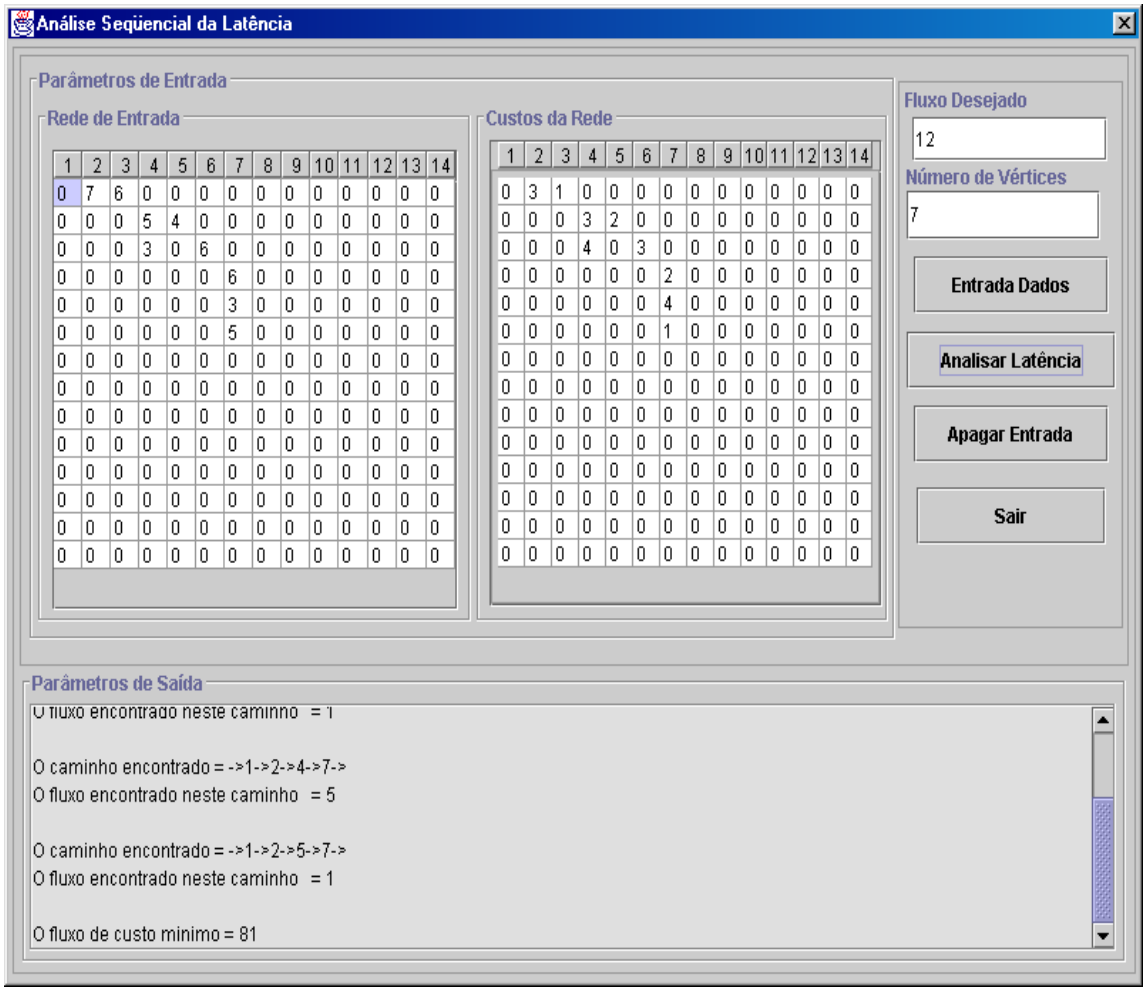


Figura 6.6. Interface de Análise Seqüencial da Latência

Esta interface possui duas matrizes. A primeira matriz é uma matriz de incidência que permite que o analista insira a topologia da rede que será analisada. A segunda matriz permite inserir o custo das arestas da rede.

O número de vértices da rede deverá também ser informado. O botão Entrada de Dados efetua a entrada das informações para o algoritmo.

O botão Analisar Latência executa o algoritmo e mostra o resultado. O botão Apagar Entrada elimina os parâmetros de entrada e saída da interface.

O botão Sair retorna para a interface principal.

Na implementação desta interface utilizamos o algoritmo paralelo de Ford-Moore-Bellman usando BFS que apresentamos no Capítulo 5.

A Figura 6.7 mostra a interface de análise paralela da melhor rota.

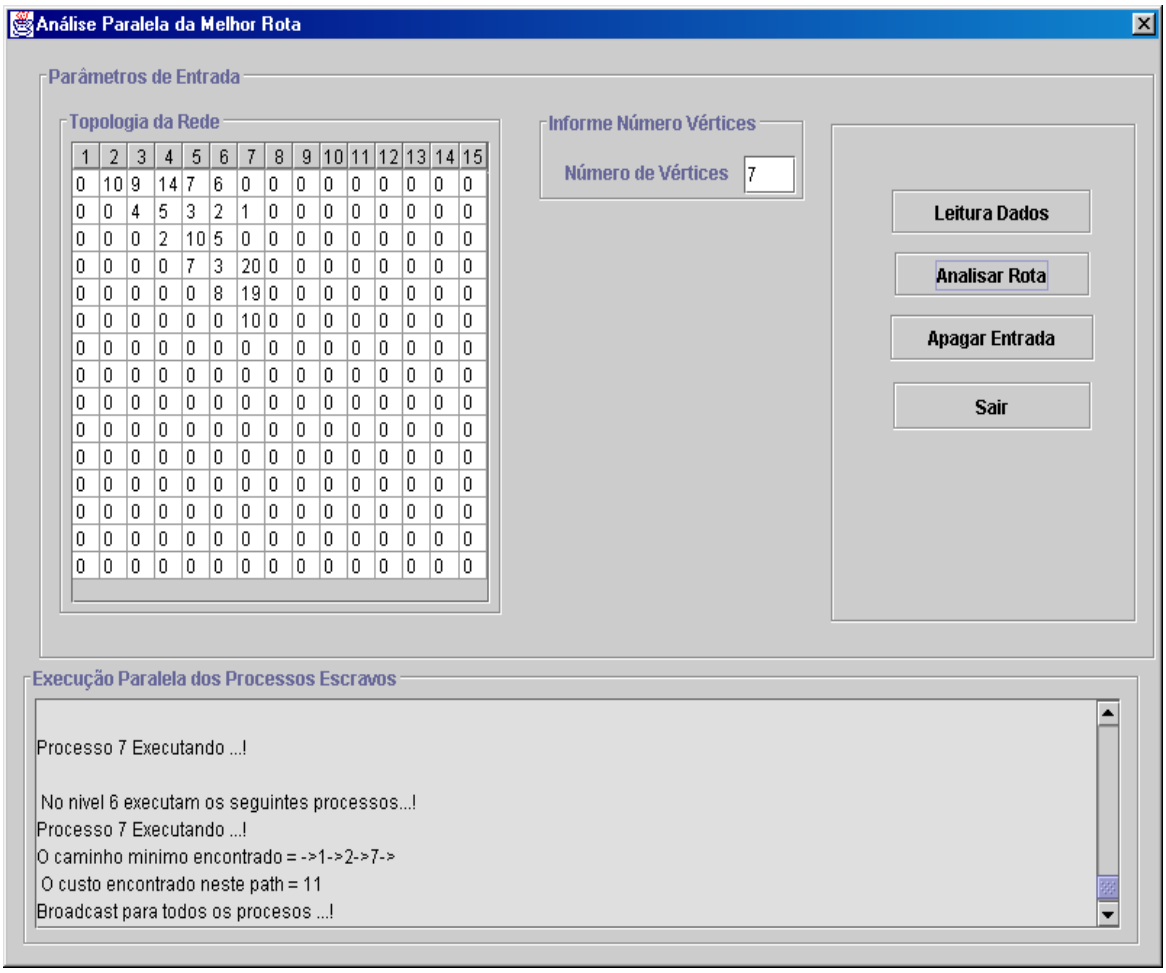


Figura 6.7. Interface de Análise Paralela da Melhor Rota .

Esta interface possui uma matriz de incidência que permite que o analista insira a topologia da rede que será analisada.

O número de vértices da rede deverá também ser informado. O botão Entrada de Dados efetua a entrada das informações para o algoritmo.

O botão Analisar Rota executa o algoritmo em paralelo e mostra o resultado.

O botão Apagar Entrada elimina os parâmetros de entrada e saída da interface.

O botão Sair retorna para a interface principal.

Na implementação desta interface utilizamos o algoritmo paralelo de Edmonds-Karp que apresentamos no Capítulo 5.

A Figura 6.8 mostra a interface de análise paralela do throughput.

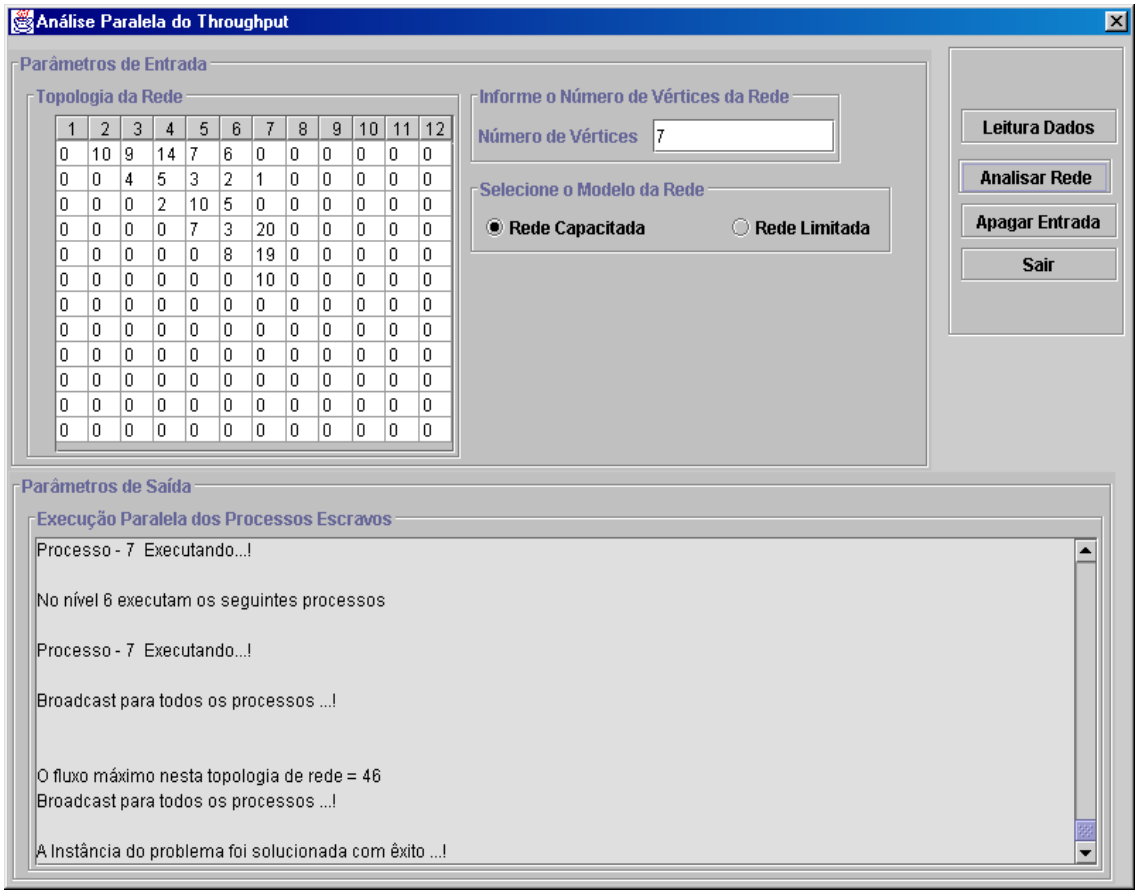


Figura 6.8. Interface de Análise Paralela do Throughput .

Esta interface possui uma matriz de incidência que permite que o analista insira a topologia da rede que será analisada.

O número de vértices da rede deverá também ser informado.

O botão Entrada de Dados efetua a entrada das informações para o algoritmo. O botão Analisar Rede executa o algoritmo em paralelo e mostra o resultado.

O botão Apagar Entrada elimina os parâmetros de entrada e de saída da interface.

O botão Sair retorna para a interface principal.

7 CONCLUSÃO

Neste capítulo apresentamos as conclusões relacionadas à pesquisa que desenvolvemos. Posteriormente, apresentamos sugestões que poderão contemplar pesquisas futuras.

7.1 Introdução

Este trabalho apresentou uma proposta para analisar a interconexão em uma LAN em um ambiente heterogêneo e distribuído segundo um modelo teórico proposto. A interface resultante apresenta ao analista uma interface gráfica e orientada a objetos.

O desenvolvimento desta interface disponibiliza uma ferramenta de análise que corrobora para solucionar diversos modelos teóricos relacionados à teoria de grafos e fluxos em redes, ou seja, classes de problemas seqüenciais ou paralelas.

De uma forma geral, o fator que motiva a escolha de um algoritmo paralelo depende da facilidade de implementação e do custo associado com hardware, software e mão de obra especializada. Não existe método universal para o projeto de algoritmos paralelos.

Os algoritmos seqüenciais (monolíticos) da teoria de grafos ou fluxos em redes algum grau de paralelismo existe. Exploramos o paralelismo dos algoritmos seqüenciais, dividindo algumas tarefas em subtarefas.

7.2 Contribuição

Nesta dissertação apresentamos as seguintes contribuições:

- Fizemos a extensão do algoritmo BFS para implementar alguns algoritmos da teoria de grafos ou fluxos em redes.

- Desenvolvemos dois algoritmos seqüenciais para o problema do menor caminho com as seguintes características:
  - O primeiro algoritmo que desenvolvemos diferencia-se do algoritmo original de Dijkstra porque utilizamos na sua implementação o método BFS para percorrer o grafo;
  - O segundo algoritmo que desenvolvemos diferencia-se do algoritmo original de Ford-Moore-Bellman porque utilizamos na sua implementação o método BFS para percorrer o grafo;
- Desenvolvemos o algoritmo de Edmonds-Karp.
- Desenvolvemos um algoritmo para o problema do fluxo de custo mínimo. Este algoritmo diferencia-se do algoritmo original de Busacker-Gowen, porque utilizamos na implementação do problema do menor caminho, o algoritmo de Ford-Moore-Bellman usando BFS.
- Desenvolvemos dois algoritmos paralelos para o problema do menor caminho, que são: o algoritmo de Dijkstra usando BFS e o algoritmo de Ford-Moore-Bellman usando BFS. No desenvolvimento destes algoritmos aplicamos o modelo de programação paralela baseada no modelo XPRAM e paralelizamos o algoritmo BFS para percorrer o grafo.
- Desenvolvemos um algoritmo paralelo para o problema do fluxo máximo baseado na implementação que fizemos do algoritmo de Edmonds-Karp. No desenvolvimento deste algoritmo, aplicamos o modelo de programação paralela baseada no modelo XPRAM e paralelizamos o algoritmo BFS para percorrer o grafo.
- Desenvolvemos um algoritmo paralelo para o problema do fluxo de custo mínimo baseado na implementação que fizemos do algoritmo de Busacker-Gowen. No desenvolvimento deste algoritmo, aplicamos o modelo de programação paralela baseada no modelo XPRAM e paralelizamos o algoritmo BFS para percorrer o grafo.

7.3 Trabalhos Futuros

Durante o desenvolvimento do trabalho, identificamos diversas questões que poderão ser desenvolvidas em trabalhos futuros como extensão deste. Enumeramos a seguir algumas possíveis extensões:

- Estudar funções assintóticas de algoritmos seqüenciais e paralelos com melhores tempos de execução relacionados à teoria de grafos ou fluxos em redes;
- Estudar outros problemas envolvendo a teoria de grafos que possam ser paralelizados e distribuídos;
- Ampliar as funcionalidades da interface, para garantir mecanismos de segurança em caso de falhas de processos presentes no grupo de processos escravos.

BIBLIOGRAFIA

[1] AHUJA, Ravindra K; MAGNANTI, T. L.; ORLIN, James B. *Network flows: theory, algorithms and applications*. New Jersey: Prentice-Hall, 1993. 846 p.

[2] ALMASI, George S.; GOTTLIEB, Allan. *Highly Parallel Computing*. California: Benjamin/Cummings, 1991. 689 p.

[3] BAZARAA, Mokhtar S.; JARVIS, John J.; SHERALI, Hanif D. *Linear Programming and Network Flows*. New York: Wiley, 1990. 684 p.

[4] BOAVENTURA NETTO, Paulo O. *Grafos, Teoria, Modelos, Algoritmos*. São Paulo. Edgard Blücher, 1996. 418 p.

[5] BOOCH, Grady. *Object Oriented Design with Applications*. California: Benjamin/Cummings, 1994. 589 p.

[6] CORMEN, Thomas H; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: teoria e prática*. Rio de Janeiro: Campus, 2002. 916 p.

[7] COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. *Distributed Systems Concepts and Design*. Massachusetts: Addison Wesley, 1994. 644 p.

[8] DAVY, John R.; DEW, Peter M. Abstract machine models for higly parallel computers. New York: Oxford University Press, 1995. 337 p.

[9] FARLEY, Jim. *Java Distributed Computing*. Canada: O'Reilly, 1998. 368 p.

[10] GIBBONS, Alan; RYTTER, Wojciech. *Efficient Parallel Algorithm*. Cambridge: Cambridge University Press, 1988. 259 p.

[11] GOLDBERG, Andrew. V. *Efficient Graph Algorithms for Sequential and Parallel Computers.* Massachusetts, 1987. Tese (Doutourado em Ciência da Computação) - Department of Electrical Engineering and Computer Science, MIT, Massachusetts, 1987.

[12] GONDRAN, Michel; MINOUX, Michel. *Graphs and Algorithms*. New York: Wiley, 1984. 650 p.

[13] HENNING, Michi; VINOSKI, Steve. *Advanced CORBA Programming with C++*. Massachusetts: Addison-Wesley, 1999. 1083 p.

[14] HORSTMANN, Cay S.; CORNELL, Gary. *Core JAVA 2 Volume I – Fundamentals*. New Jersey: Sun Microsystems Press, 2000. 832 p.

[15] HORSTMANN, Cay.; CORNELL, Gary. *Core JAVA 2 Volume II - Advanced Features*. New Jersey: Sun Microsystems Press, 2001. 1232 p.

[16] HWANG, K.; BRIGGS, F. A. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984. 846 p.

[17] KREYSZIG, Erwin. *Advanced Engineering Mathematics*. New York: John Wiley & Sons, 1993. 1271 p.

[18] KWO, Steve; MOLDOVAN, Dan. *The State of Art in Parallel Production Systems*. Journal of Parallel and Distributed Computing. v. 15, p. 1-26, 1992.

[19] KRUSKAL, Clyde P.; RUDOLPH, Larry; SNIR, Marc. *A Complexity Theory of Efficient Parallel Algorithms*. IBM Research Report RC 13572, 1988.

[20] KUMAR, Vipin; SINGH, Vineet. *Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem*. Journal of Parallel and Distributed Computing, v. 13, p. 124-138, 1991.

[21] LEWIS, Harry R.; PAPADIMITRIOU, Christos H. *Elements of the Theory of Computation*. New Jersey: Prentice-Hall, 1981. 361 p.

[22] MATETI, Prabhaker; DEO, Narsingh. *Parallel Algorithms for the Single Source Shortest Path Problem.* Computing, v. 29, p. 31-49, 1982.

[23] MOWBRAY, Thomas J.; RUTH, William A. *Inside CORBA distributed objects standards and applications*. New York: John Wiley & Sons, 1996. 376 p.

[24] MOWBRAY, Thomas J.; MALVEAU, Raphael C. *CORBA design patterns*. New York: Wiley Computer, 1997. 333 p.

[25] MOWBRAY, Thomas J.; ZAHAVI, Ron. *The essential CORBA : systems integration using distributed objects*. New York: Wiley, 1995. 316 p.

[26] NASH, Jonathan M. et al. *Parallel Algorithm design on The XPRAM Model*. In: ABSTRACTS MACHINES WORKSHOP, 2.,1993. Inglaterra.

[27] ORFALI, Robert; HARKEY, Dan. *Client/Server programming with Java and Corba*. New York: Wiley, 1998. 1022 p.

[28] ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri. *Instant CORBA*. New York: Wiley, 1997. 313 p.

[29] ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri. *The essential distributed objects survival guide*. New York: Wiley, 1996. 604 p.

[30] ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri. *The essential client/server survival guide*. New York: Wiley, 1996. 676 p.

[31] PAPADIMITRIOU, Christos H.; STEIGLITZ, Kenneth. *Combinatorial Optimization: Algorithms and Complexity*. New Jersey: Prentice-Hall, 1998. 496 p.

[32] PREISS, Bruno R. *Data Structures and Algorithms: with object-oriented design patterns in Java*. New York: John Wiley, 1999. 635 p.

[33] SHILOACH, Yossi; VISHKIN, Uzi. *An $O(n^2 \log n)$ parallel max-flow algorithm*. Journal of Algorithms, v. 3, p. 128-146, 1982.

[34] SKIENA, Steve S. *The Algorithm design manual*. TELOS: Canada, 1998. 486 p.

[35] TANENBAUM, Andrew S. *Computer Networks*. New Jersey: Prentice-Hall, 1996. 813 p.

[36] TANENBAUM, Andrew S. *Modern Operating Systems*. New Jersey: Prentice-Hall, 2001. 951 p.

[37] TARJAN, Robert E. *Complexity of Combinatorial Algorithms*. SIAM Review, v. 20, p. 457-491, 1978.

[38] VALIANT, Leslie G. *A scheme for fast parallel communication*. SIAM Journal on Computing, v. 11, n. 2, p. 350-361, 1982.

[39] WILF, Herbert S. *Algorithms and Complexity*. New Jersey: Prentice-Hall, 1986. 231 p.

[40] WILKINSON, Barry; ALLEN, Michael. *Parallel programming: techniques and applications using networked workstations and parallel computers*. New Jersey: Prentice-Hall, 1999. 431 p.

[41] WYLLIE, James C. *The Complexity of Parallel Computations*. Cornell, 1979. Tese (Doutourado em Ciência da Computação) - Department of Computer Science, Cornell University, Cornell, 1979.