**Implementação Eficiente
de Programação Dinâmica:
Análise Teórica e Aplicação
à Designação de Capacidades**

Paulo Regis Zanjacomo

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Carlos Humes Jr.**

*Durante a elaboração deste trabalho, o autor recebeu apoio
financeiro da IBM-Brasil, CNPq e FAPESP.*

-São Paulo, Agosto de 1992-

a M.I.M.

Resumo

Apresentamos um estudo sobre problemas multicritério separáveis e o uso de Programação Dinâmica para resolvê-los, enfatizando a implementação eficiente do algoritmo obtido. Inicia-se a análise com o estudo do custo dos processamentos intermediários das informações associadas ao problema multicritério multiestágio, na busca de soluções não dominadas para cada estágio. Estes custos computacionais são modelados como custos das arestas de um grafo onde a cada nó associa-se um subproblema resolvido. Considerando o custo computacional de Programação Dinâmica como sendo o que resulta na melhor trajetória até o nó associado ao problema global, obtém-se a melhor estratégia de ordenamento das soluções dos subproblemas.

O estudo reafirma a estratégia gulosa como ótima para o processamento intermediário das informações, obtendo o custo computacional exato desta solução. Além disso, prova que uma estratégia de processar subproblemas de dimensão não crescente é ótima para a obtenção dos pontos eficientes do problema multicritério, ao usarmos o algoritmo de Programação Dinâmica. O estudo teórico baseia-se na análise de complexidade de pior caso, pois existem fortes hipóteses sobre eliminação de nós.

Como validação dos resultados em condições menos adversas, esta estratégia foi implementada na resolução do problema de Designação de Capacidades em Redes de Computadores, verificando sua eficiência através de resultados comparativos com outras possíveis estratégias de processamento. Apresentamos, também, uma implementação do algoritmo A^* como alternativa ao esquema de Programação Dinâmica na resolução deste problema de designação de capacidades.

Abstract

We present a study about separable multicriteria problems and the application of Dynamic Programming to solve them, with emphasis on the efficient implementation of the obtained algorithm. The analysis begins studying the cost of intermediate processing of information related to the multicriteria multistage problem, searching for non-dominated solutions in each stage. These computational costs are considered to be arc costs on a graph where each node represents a solved subproblem. The Dynamic Programming approach to determine the best path from origin to the node related to the global problem induces the best ordering strategy for the solution of subproblems.

The study reinforces the greedy strategy as optimal to the intermediate processing of information, leading to the exact computational cost of this solution. Besides, it proves that the strategy of processing subproblems of non-increasing dimension is optimal to obtain efficient points of multicriteria problems, when using the Dynamic Programming algorithm. Due to strong hypothesis about discarding nodes, we adopt the worst case complexity analysis approach.

In order to validate the results in less adverse conditions, this strategy was implemented in the solution of the Capacity Assignment problem for Computer Networks Design and its efficiency was verified through comparative results with other possible strategies of processing. We also present an implementation of the A^* algorithm as an alternative to the Dynamic Programming solution to the Capacity Assignment problem.

Agradecimentos

Ao professor Carlos Humes Jr., por ser tão culpado deste trabalho existir quanto eu;

Aos professores Julio Michael Stern e Celma de Oliveira Ribeiro, pelo auxílio na revisão deste trabalho, decisivo para a sua conclusão;

Ao professor Manuel Valentim Pera, pela sua elegante colaboração na demonstração do lema 3.2.2;

A Sérgio L. Wasserstein, pela sua participação fundamental no desenvolvimento embrionário deste trabalho.

Conteúdo

1	Introdução	2
2	Programação Dinâmica	4
2.1	Problemas Separáveis	4
2.2	Problemas de Decisão Sequencial - uma breve caracterização	5
2.3	A Estratégia de Programação Dinâmica	6
2.3.1	O Princípio de Otimalidade	6
2.3.2	Projeções	7
2.3.3	Um Exemplo Elucidativo	9
3	Problemas de Decisão em Dois Estágios	12
3.1	Sort-Merge Entre Vetores de Mesmo Tamanho	13
3.2	Sort-Merge Entre Vetores de Tamanhos Variados	14
3.3	O Produto Cartesiano entre Dois Vetores	16
4	Problemas de Otimização Multicritério Multiestágio	19
4.1	Uma Breve Descrição	19
4.1.1	Pareto-Otimalidade	20
4.2	Busca de Pontos Eficientes por Programação Dinâmica	21
4.2.1	Caracterização da Busca de Pontos Eficientes	21
4.2.2	Custo da Busca de Pontos Eficientes	22
4.2.3	Estratégia Ótima para Busca de Pontos Eficientes	23
5	Resolução de um Problema Bicritério	29
5.1	Definição do CA	29
5.2	Resolução do CAD através de Programação Dinâmica	34
5.2.1	Experiência Computacional	37
5.3	Resolução do CAD através do A^*	45
5.3.1	Experiência Computacional	47

A Implementações do CAD por Programação Dinâmica	49
A.1 - Implementação com a estratégia gulosa nos sort-merges e a política sequencial ótima para a escolha das tabelas.	49
A.2 - Implementação com a estratégia sequencial para os sort-merges, e a política sequencial ótima para a escolha das tabelas.	70
A.3 - Implementação com a estratégia gulosa para os sort-merges e estratégia gulosa para a escolha das tabelas.	91
B Implementação do CAD pelo A^*	113
Bibliografia	135

Capítulo 1

Introdução

Esta dissertação situa-se no limiar entre os campos de desenvolvimento teórico e aplicações práticas, se é que tal limite realmente existe. Ela também mostra que é falsa a impressão geralmente adquirida nos anos de graduação de que as áreas de conhecimento matemático e computacional são “hermeticamente fechados”, autocontidos ou isolados.

Além de abordar tópicos aparentemente distintos e sem correlação, para depois mostrar como um pode ser usado na resolução do outro, introduzo uma modelagem para a resolução de uma classe definida de problemas, e aplico esta modelagem.

Tudo começou em meados de 1989, quando iniciei o programa de Iniciação Científica. O então problema de Designação de Capacidades em Redes de Computadores pareciam completamente definidos quanto a sua solução, quando implementamos várias versões computacionais envolvendo estratégias distintas no tratamento das tabelas representando estados do problema. Os resultados obtidos foram completamente não intuitivos: agrupar inicialmente as tabelas maiores, deixando as de menor tamanho para o final. Naquele momento, um problema bicritério transformava-se num problema de decisões sequenciais, cuja abordagem de Programação Dinâmica justificava teoricamente os resultados obtidos na prática. Esta transição envolveu diversos estudos, desde análise matemática, passando por teoria das filas, otimização multicritério e concluindo com problemas de decisão, verificando a natureza interdisciplinar dos problemas reais.

Assim, para organizar todo o trabalho executado desde então e introduzir os conceitos à medida do necessário, além de tornar esta dissertação esteticamente mais atraente, apresento no capítulo 2 uma discussão geral sobre Programação Dinâmica, envolvendo a principal estrutura que permite

sua aplicação, a separabilidade. Caracterizo problemas de decisão sequencial que podem ser resolvidos por Programação Dinâmica, que aqui é descrita como projeções, exemplificando, ao final, o uso de Programação Dinâmica num problema simples.

Na resolução de um problema de decisão sequencial com vários estágios, em algum momento necessitaremos resolver um subproblema com dois estágios. No capítulo 3 mostro como isso pode ser feito otimamente, já preparando o caminho para a caracterização a ser feita no capítulo 4 a respeito de problemas multicritério multiestágio. No capítulo 4, além de descrever problemas multicritério, apresento uma forma de buscar seus pontos eficientes através de Programação Dinâmica, usando resultados do capítulo 3.

Finalmente, no capítulo 5 apresento o problema motivador desta dissertação, e mostro como ele é resolvido com as soluções propostas nos capítulos anteriores. Alternativamente ao esquema de Programação Dinâmica, mostro a solução do problema de Designação de Capacidades através do algoritmo A^* , aproveitando as informações subjacentes deste problema. Testes computacionais realizados são analisados neste capítulo.

Seguem-se, nos apêndices, as versões computacionais dos algoritmos implementados, na linguagem PASCAL.

Capítulo 2

Programação Dinâmica

Um dos métodos mais usuais para se obter o ótimo sob algum critério funcional associado a fenômenos econômicos foi formulado na década de 50 por Richard Bellman (embora em 1946, P. Massé tenha feito uso de métodos similares ao proposto por Bellman, este não recebeu a devida atenção - veja P. Massé, “Le Choix des Investissements”). Este método foi alvo de grande interesse principalmente pela sua aplicabilidade a um conjunto mais extenso de problemas e tem fundamental importância na resolução de problemas de decisão sequencial e otimização multiestágio. Nos referimos a este método como “Programação Dinâmica”, e neste capítulo vamos discutir suas idéias básicas, como o Princípio de Otimalidade, além de exemplificar o método.

2.1 Problemas Separáveis

Uma das estruturas fundamentais que permitem a aplicação de programação dinâmica é a “separabilidade”. O conceito de Separação permite dizer que, em certas circunstâncias, o ótimo das partes que compõe um problema corresponde ao ótimo de todo o problema.

Uma modelagem matemática comum é a seguinte (veja Humes [15]):

(i) Dados

$X_i \subset \mathbb{R}^{n_i}$,

$f_i : X_i \rightarrow \mathbb{R}$ e

$g_i : X_i \rightarrow \mathbb{R}^{m_i}$, para $i = 1, \dots, p$,

encontre, se existir, $x^* = (x_1^*, \dots, x_p^*)$ tal que $x^* \in X$ e

$$\sum_{i=1}^p f_i(x_i^*) = \min_{x \in X} \sum_{i=1}^p f_i(x_i),$$

para $X = \{(x_1, x_2, \dots, x_p) : x_i \in X_i \text{ e } g_i(x_i) \leq 0\}$.

Este problema é equivalente a p subproblemas independentes definidos por:

(ii) “minimizar $f_i(x_i)$, sujeito a $x_i \in X_i$ e $g_i(x_i) \leq 0$ ”, ($i = 1, \dots, p$).

Assim, notamos que poderemos resolver o problema dado na forma (i) através da resolução de problemas menores, na forma (ii).

A estrutura de separação será fundamental no decorrer deste texto, para a resolução dos problemas introduzidos nos capítulos 4 e 5.

2.2 Problemas de Decisão Sequencial - uma breve caracterização

Vamos supor que tenhamos um sistema físico S cujo estado num determinado instante t é caracterizado por um vetor p (onde suas componentes podem significar volume, temperatura ou momento de elementos físicos, ou ainda, do ponto de vista econômico, suprimento ou demanda, nível de estoque de um produto, ou até mesmo a distribuição de probabilidade de grandezas envolvidas no sistema S).

No decorrer do tempo, este sistema está sujeito a mudanças de origem determinística ou estocástica, ou seja, as variáveis que descrevem o sistema estão sob transformações. Uma transformação do sistema é, em geral, uma função do tempo, dos ruídos (influências que o sistema pode sofrer em cada estado, dependente de ações ou estados passados ou não) e dos controles (ações que tomamos perante determinado estado do sistema).

Vamos supor, agora, que é possível escolher os controles a serem aplicadas a S em qualquer instante de tempo. A este processo de escolha dos controles em um sistema S denominaremos “Processo de Decisão” (Bellman [2]). Se existir uma única decisão possível a cada instante, temos um processo em um estágio; se houver uma sequência de possíveis decisões, temos um processo de decisão multiestágio. Esta distinção, no entanto, não é rígida: o processo de escolher um ponto no $\mathbb{R}^3$ pode ser visto como um processo de um estágio, onde escolhemos $(x, y, z) \in \mathbb{R}^3$, ou um processo multiestágio, se pudermos escolher $x \in \mathbb{R}$, $y \in \mathbb{R}$ e depois $z \in \mathbb{R}$ (para isto, necessitamos que o nosso sistema tenha as características de *separabilidade*).

Assim, cada decisão pode ser vista como uma “escolha”, se o espaço das possíveis decisões for enumerável, e denominamos “Política” a uma regra para tomada de decisões. Em problemas com caráter estocástico, sob a mesma política, diferentes ruídos conduzem a diferentes sequências de de-

cisões.

Algum critério deve guiar a escolha das decisões, como custos associados a determinadas decisões ou o lucro das mesmas, ou qualquer função que caracterize o objetivo das decisões a serem tomadas. Logo, estamos interessados em descobrir qual é a política que minimize (ou maximize, dependendo do interesse) esta função. Esta política é denominada “Política Ótima”.

Os processos de decisão multiestágios podem ter como característica:

1. Evolução determinística, isto é, o resultado de uma decisão é determinado unicamente por esta decisão;
2. Evolução estocástica, isto é, uma decisão resulta em uma distribuição de transformações, e não só em uma única transformação, em geral;
3. Domínio discreto, onde cada decisão é tomada em um instante discreto determinado de tempo;
4. Domínio Contínuo, onde cada decisão pode ser tomada em cada ponto de um intervalo contínuo.

Neste trabalho, estamos apenas interessados em problemas de decisão multiestágio sequencial discreto determinístico.

2.3 A Estratégia de Programação Dinâmica

O termo Programação Dinâmica aplica-se a um tipo de análise onde, na busca da política ótima para um dado sistema, concentramos nossa atenção nas formas e propriedades das políticas ótimas parciais, isto é, devemos estudar as condições que devam ser satisfeitas para que um processo de decisão seja ótimo em uma parcela do processo, e descobrir como explorar estas condições para determinar a melhor ação global.

2.3.1 O Princípio de Otimalidade

Este é o princípio básico que rege a aplicação de programação dinâmica e é enunciado como

Se J representa a melhor “trajetória” para irmos de um estado A a um estado B , então qualquer que seja o trecho de J , este deve ser a melhor trajetória entre os estados inicial e final deste trecho.

A apresentação matemática tradicional deste princípio é:

Sejam
 $C_t(j)$ = custo associado a atingirmos o estado j do sistema em t instantes de tempo;
 $d(i, j)$ = custo de irmos do estado i do sistema ao estado j em um instante de tempo;
o problema de se obter uma política ótima é descrito recursivamente por

$$C_t(j) = \min_{i \in I} \{C_{t-1}(j), C_{t-1}(i) + d(i, j)\}$$

para I = conjunto dos estados do sistema.

Assim, notamos que podemos obter a política ótima para um sistema se conhecermos o custos das transições entre seus possíveis estados, construindo os trechos ótimos entre estados internos deste sistema.

A grande vantagem em se usar técnicas de programação dinâmica na busca de políticas ótimas é que nem todas as possíveis sequências de decisões deverão ser consideradas nesta busca. Se uma dada sequência de decisões contiver uma subsequência que claramente não é ótima, ela será descartada, segundo o Princípio de Otimalidade.

2.3.2 Projeções

Os problemas dotados de separabilidade podem ser resolvidos pela operação geométrica que o termo “projeção” indica, ou seja, podemos dividir suas variáveis em dois ou mais grupos e resolver os problemas associados a estes subconjuntos de variáveis (veja Humes [15]). Tipicamente, temos:

(P) minimizar $f(x, y)$
sujeito a $x \in X, y \in Y$ e $g(x, y) \leq 0$
para $X \subset \mathbb{R}^p$ não vazio, $Y \subset \mathbb{R}^q$ não vazio, $f : X \times Y \rightarrow \mathbb{R}$ e $g : X \times Y \rightarrow \mathbb{R}^m$,

a operação de projeção sobre o espaço das variáveis y segue naturalmente como

(Py) minimizar $v(y)$
sujeito a $y \in Y \cap V$, onde $V = \{y \in Y : \exists x \in X / g(x, y) \leq 0\}$ e $v : V \rightarrow \mathbb{R} \cup \{-\infty\}$ é definido como $v(y) = \inf \{f(x, y) : x \in X \text{ e } g(x, y) \leq 0\}$.

O problema (P) tem solução se e somente se (Py) também o tiver, e na solução y^* de (Py) existir um $x^* \in X$ tal que $g(x^*, y^*) \leq 0$ e $f(x^*, y^*) = v(y^*)$.

Um dos exemplos mais comuns de projeções é *programação dinâmica*, onde são feitas sucessivas projeções do problema de decisão sequencial multiestágio:

Minimize

$$\sum_{i=0}^{n-1} f_i(x^i, u^i)$$

sujeito a

$$x^{i+1} = g_i(x^i, u^i), i = 0, 1, \dots, n-1,$$

x^0 determinado e

$$u^i \in U^i, i = 0, 1, \dots, n-1,$$

para $U^i \subset \mathbb{R}^m, f_i : \mathbb{R}^q \times \mathbb{R}^m \rightarrow \mathbb{R}, g_i : \mathbb{R}^q \times \mathbb{R}^m \rightarrow \mathbb{R}^q, i = 0, 1, \dots, n-1$.

Se projetarmos as variáveis $(x^i, u^i), i = 0, 1, \dots, n-1$ do problema acima no espaço das variáveis $(x^i, u^i), i = 0, 1, \dots, n-2$, obtemos

Minimize

$$\sum_{i=0}^{n-2} f_i(x^i, u^i) + v_{n-1}(x^{n-1})$$

sujeito a

$$x^{i+1} = g_i(x^i, u^i), i = 0, 1, \dots, n-2,$$

x^0 determinado e

$$u^i \in U^i, i = 0, 1, \dots, n-2,$$

onde $v_{n-1}(x^{n-1}) = \inf\{f_{n-1}(x^{n-1}, u^{n-1}) : u^{n-1} \in U^{n-1}\}$

Novamente, projetando este último problema no espaço das variáveis $(x^i, u^i), i = 0, 1, \dots, n-3$, obtemos

Minimize

$$\sum_{i=0}^{n-3} f_i(x^i, u^i) + v_{n-2}(x^{n-2})$$

sujeito a

$$x^{i+1} = g_i(x^i, u^i), i = 0, 1, \dots, n-3,$$

x^0 determinado e

$$u^i \in U^i, i = 0, 1, \dots, n-3,$$

onde $v_{n-2}(x^{n-2}) = \inf\{f_{n-2}(x^{n-2}, u^{n-2}) + v_{n-1}(g_{n-2}(x^{n-2}, u^{n-2})) : u^{n-2} \in U^{n-2}\}$

Continuando repetidamente (para $0 \leq j \leq n-1$), temos

Minimize

$$\sum_{i=0}^{n-j-1} f_i(x^i, u^i) + v_{n-j}(x^{n-j})$$

sujeito a

$$x^{i+1} = g_i(x^i, u^i), i = 0, 1, \dots, n-j-1,$$

x^0 determinado e

$$u^i \in U^i, i = 0, 1, \dots, n-j-1,$$

$$\text{onde } v_{n-j}(x^{n-j}) = \inf_{u^{n-j} \in U^{n-j}} \{f_{n-j}(x^{n-j}, u^{n-j}) + v_{n-j+1}(g_{n-j}(x^{n-j}, u^{n-j}))\} :$$

As fórmulas

$$v_{n-1}(x^{n-1}) = \inf \{f_{n-1}(x^{n-1}, u^{n-1}) : u^{n-1} \in U^{n-1}\}$$

e

$$v_{n-j}(x^{n-j}) = \inf \{f_{n-j}(x^{n-j}, u^{n-j}) + v_{n-j+1}(g_{n-j}(x^{n-j}, u^{n-j})) : u^{n-j} \in U^{n-j}\}$$

definem as relações de recorrência associadas ao processo recursivo de Programação Dinâmica, fazendo com que um problema de n estágios seja transformado em $(n+1)$ problemas de um estágio e obedecendo o princípio de otimalidade.

Cumpra notar que para problemas combinatórios compatíveis com o problema de decisões sequenciais, as relações acima definem uma enumeração implícita das possíveis combinações deste problema.

2.3.3 Um Exemplo Elucidativo

Suponha que desejamos ir do ponto A ao ponto B da forma mais econômica possível, onde os possíveis caminhos disponíveis estão representados no grafo (extraído de Dreyfus e Law [7]) abaixo, e os valores nas arestas representam o custo de passarmos por ali.

Podemos resolver este problema enumerando todos os possíveis caminhos de A a B , e somando os valores de cada caminho encontrado, para então compararmos os custos obtidos e escolhermos o melhor. Como existem 20 possíveis caminhos de A para B , esta solução obtida na “força bruta” exigiria que fizéssemos 100 somas (cada caminho tem comprimento 5), além de 19 comparações para escolhermos qual caminho é o menos custoso.

Uma forma mais inteligente é usar programação dinâmica para encontrarmos o melhor caminho. A priori, não sabemos como ir diretamente de A a B da melhor forma, mas se soubermos como ir da melhor forma de C a B , e de D a B , então é fácil decidirmos qual caminho tomar: denominando S_X como sendo o custo mínimo de irmos de X a B , para $X \in \{A, B, \dots, P\}$, então escolheremos ir de A a C e depois de C a B se $(1 + S_C) \leq (0 + S_D)$ (pois o custo de ir de A para C é 1 e de A para D é 0), ou ir de A para D e depois de D a B no caso contrário. Isto significa que

$$S_A = \min\{1 + S_C, 0 + S_D\}$$

Assim, só devemos nos preocupar com os caminhos de C a B e de D a B para obtermos nossa solução.

Vamos continuar reduzindo nosso problema a subproblemas menores ao mesmo tempo que aplicamos o princípio de otimalidade (isto é, qualquer trecho do caminho de menor custo entre A e B deve ser o de custo mínimo entre o início e fim destes trechos), e notamos que

$$S_C = \min\{5 + S_E, 4 + S_F\}$$

e

$$S_D = \min\{7 + S_F, 3 + S_G\}$$

onde agora S_E , S_F e S_G são o alvo de nosso estudo, e estes, por sua vez, poderão ser computados se S_H , S_I , S_J e S_K estiverem disponíveis.

Observando o grafo, sabemos de imediato que $S_O = 2$ e $S_P = 1$ (pois existe um único caminho entre O e B e entre P e B). Da mesma forma temos

$$S_L = 5 + S_O = 7 \text{ e } S_H = 3 + S_L = 10;$$

$$S_N = 4 + S_P = 5 \text{ e } S_K = 2 + S_N = 7;$$

$$S_M = \min\{2 + S_O, 8 + S_P\} = 4;$$

$$S_I = \min\{3 + S_L, 4 + S_M\} = 8;$$

$$S_J = \min\{2 + S_M, 2 + S_N\} = 6;$$

$$S_E = \min\{2 + S_H, 1 + S_I\} = 9;$$

$$S_F = \min\{1 + S_I, 2 + S_J\} = 8;$$

$$S_G = \min\{5 + S_J, 4 + S_K\} = 11;$$

$$S_C = \min\{5 + S_E, 4 + S_F\} = 12;$$

$$S_D = \min\{7 + S_F, 3 + S_G\} = 14;$$

e, finalmente

$$S_A = \min\{1 + S_C, 0 + S_D\} = 13.$$

Para obter o caminho de menor custo, é só irmos voltando pelos menores custos calculados, notando quais nós que geraram este custos (ou seja, o nosso caminho é $A-C-F-J-M-O-B$). Com estes cálculos, executamos 24 somas e 9 comparações, bem melhor que tentarmos resolver o problema com enumeração explícita, como descrito anteriormente.

A abordagem de programação dinâmica pode ser encontrada de forma implícita em algoritmos não descritos como a ele pertencentes. É o caso do algoritmo de Dijkstra, apresentado dentro do contexto de teoria dos grafos, para a obtenção de caminhos mais curtos, como no nosso exemplo.

Capítulo 3

Problemas de Decisão em Dois Estágios

Para resolvermos um problema sob a abordagem de Programação Dinâmica, temos que saber resolver eficientemente um problema que envolva somente dois estágios. Na realidade, a resolução de um problema com dois estágios é mais poderosa do que aparenta, pois se um dado problema de decisão sequencial discreta tiver n estágios, e este puder ser resolvido pela partição de n em n_1 e n_2 estágios disjuntos, então as técnicas que veremos aplicar-se-ão a este problema maior.

Fixando as idéias em dois estágios, se a cada um deles associarmos duas tabelas (ou vetores, como denominaremos genericamente), então as possíveis decisões entre estes dois estágios é obtida realizando-se o que denominaremos “Produto Cartesiano” entre os dois vetores representando estes estágios. Vamos considerar a possibilidade de eventuais eliminações de elementos nestas operações (devido a encontrarmos combinações que claramente não interessem ou sejam ineficientes em algum sentido - veja Capítulo 4) e ainda consideraremos os vetores ordenados sob algum critério, facilitando o trabalho de busca da melhor decisão (à melhor decisão estará associada o menor custo como critério de ordenação). A ordenação dos vetores terá implicações mais gerais, que serão vistas nos Capítulos 4 e 5.

Assim, neste capítulo, apresentamos o problema de realizar o produto cartesiano entre dois vetores ordenados, que é uma extensão da operação de “sort-merge” entre vetores, e apresentamos uma estratégia ótima para a solução deste problema. Para isto, vamos inicialmente analisar possíveis subproblemas envolvidos, como o de sort-merge entre vetores de mesmo

tamanho e sort-merge entre vetores de tamanhos variados, problemas estes que possuem soluções conhecidas.

3.1 Sort-Merge Entre Vetores de Mesmo Tamanho

Dados m vetores ordenados de tamanho p cada, queremos obter um vetor final constante com todos os elementos dos m vetores originais, também ordenado. Se $m = 2$, a solução segue facilmente segundo um algoritmo no qual comparamos aos pares um elemento de cada vetor, guardando o menor entre eles:

```

dados  $V_1[1..p]$  e  $V_2[1..p]$  vetores ordenados,
tome  $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$ ;
enquanto  $(i \leq p)$  e  $(j \leq p)$  faça:
    se  $V_1[i] \leq V_2[j]$  então:
         $V_3[k] \leftarrow V_1[i]; k \leftarrow k + 1; i \leftarrow i + 1$ ;
    senão:
         $V_3[k] \leftarrow V_2[j]; k \leftarrow k + 1; j \leftarrow j + 1$ ;
enquanto  $(i \leq p)$  faça:
     $V_3[k] \leftarrow V_1[i]; k \leftarrow k + 1; i \leftarrow i + 1$ ;
enquanto  $(j \leq p)$  faça:
     $V_3[k] \leftarrow V_2[j]; k \leftarrow k + 1; j \leftarrow j + 1$ ;

```

O custo computacional deste algoritmo é de $\mathcal{O}(2p - 1)$, pois no pior caso, cada elemento de cada vetor será comparado duas vezes com algum outro elemento do outro vetor, à exceção do primeiro e último elementos, que só serão comparados uma vez.

Se $m = 2^k$ para $k = 2, 3, \dots$, este problema é resolvido pela estratégia conhecida como “Divide and Conquer”, onde aplica-se o algoritmo acima a pares de vetores, gerando-se 2^{k-1} novos vetores de tamanho $2p$ cada, e sucessivamente reaplica-se esta estratégia aos novos vetores, até obter-se um único vetor ao final. O custo computacional deste novo algoritmo é de $\mathcal{O}(mp \log_2 m) = \mathcal{O}(2^k kp)$ (mais referências, veja Terada [22] e Aho [1]).

No caso de $m > 1$, mas $m \neq 2^k$, pode-se ainda manter a filosofia de Dividir e Conquistar, unindo os vetores aos pares, mesmo que sobre um vetor ao final de cada processo. Neste caso, podemos encontrar em Aho [1] (página 73, ex. 2.30) o fato de que se um vetor de tamanho q é obtido pelo sort-merge de vetores de tamanhos $\lfloor \frac{q}{2} \rfloor$ e $\lceil \frac{q}{2} \rceil$, e assim recursivamente, então

o custo computacional desta estratégia é de $q\lceil\log_2 q\rceil - 2^{\lceil\log_2 q\rceil} + 1$. Para isto, deve-se fazer os sort-merges de forma balanceada.

3.2 Sort-Merge Entre Vetores de Tamanhos Variados

Para realizarmos o sort-merge entre m vetores de tamanhos variados, podemos aproveitar as idéias da última seção. Para $m = 2$, o algoritmo lá descrito não muda, a exceção dos detalhes no controle dos tamanhos dos dois vetores, a fim de que o algoritmo termine corretamente. Para m qualquer, uma boa estratégia é fazer o sort-merge aos pares dos vetores de menor tamanho a cada passo. Esta estratégia é conhecida na literatura em geral como “gulosa”, pois a cada passo realiza-se prioritariamente a operação de menor custo imediato. Vamos mostrar que esta estratégia é a melhor que pode ser feita no nosso caso.

Vamos encarar o problema de fazermos o sort-merge entre m vetores como um problema de formar um conjunto P , representando o sort-merge entre todos os vetores, através de um processo de decisão, onde a cada passo devemos decidir quais são os vetores a serem unidos por um sort-merge.

Consideraremos o custo de unir os vetores R e S como sendo $c(R \cup S) = \alpha(r+s) + \beta$, para $r = \#R$, $s = \#S$ e $(\alpha, \beta) \in \mathbb{R}_{++} \times \mathbb{R}$. Claramente $c(R) = 0$ para $\#R = 1$, e este custo generaliza o custo de um sort-merge comum, onde temos $\alpha = 1$ e $\beta = -1$ para o pior caso. Sem perda de generalidade, vamos considerar $\alpha = 1$.

Notando que $c(R \cup S) > c(R \cup S^0)$ se $R \cup S = R \cup S^0$ e $\#S > \#S^0$, ou seja, o custo de se unir um vetor R a S deve ser maior do que unir R a S^0 , desde que S contenha mais elementos do que S^0 e as uniões nos conduzam a resultados idênticos, podemos obter o custo da política ótima para se formar o conjunto P denotada $T^*(p)$ para $p = \#P$, recursivamente da seguinte forma:

$$T^*(p) = \min_{r+s=p, \substack{r \geq 1 \\ s \geq 1}} \{T^*(r) + T^*(s) + p + \beta\}.$$

Lema 3.2.1 *Para o problema de obter o conjunto P , como descrito acima, o custo ótimo é*

$$(I) \quad T^*(p) = p(\lceil\log_2 p\rceil + 1) - 2^{\lceil\log_2 p\rceil} + \beta(p - 1)$$

e

$$(II)(r, s) \in \operatorname{argmin}_{r+s=p, r \geq s \geq 1} \{T^*(r) + T^*(s) + p + \beta\} \Leftrightarrow \lceil \log_2 r \rceil - \lceil \log_2(s+1) \rceil \leq 0$$

para $r = \#R$ e $s = \#S$ tais que $(R \cup S) = P$.

Prova :

Vamos provar por indução:

Claramente (I) vale para $p = 1, 2$ e (II) vale para $p = 2, 3, 4$. Vamos assumir que (I) vale para todo q , $q < p$ e $p \geq 3$, e vamos provar que (II) vale para p , e isto implica que (I) vale para p .

Vamos considerar

$$D(r, s) = T^*(r) + T^*(s) - T^*(r-1) - T^*(s+1)$$

para todos os pares (r, s) tais que $r + s = p$ e $r - 1 \geq s + 1 \geq 2$ ($D(r, s)$ corresponde a diferença entre os argumentos do minimando na expressão de $T^*(p)$, para (r, s) e $(r-1, s+1)$).

Usando o fato que

$$A(l) = l \lceil \log_2 l \rceil - 2 \lceil \log_2 l \rceil - (l-1) \lceil \log_2(l-1) \rceil - 2 \lceil \log_2(l-1) \rceil = \lceil \log_2 l \rceil,$$

temos $D(r, s) = A(r) - A(s+1) = \lceil \log_2 r \rceil - \lceil \log_2(s+1) \rceil$, e, portanto, (II) vale para p (pois $(r \geq s)$ e $(r + s = p)$).

Segue ainda

$$p \text{ par} \Rightarrow (p/2, p/2) \in \operatorname{argmin}_{r+s=p, r \geq s \geq 1} \{T^*(r) + T^*(s) + p + \beta\}$$

$$p \text{ ímpar} \Rightarrow ((p+1)/2, (p-1)/2) \in \operatorname{argmin}_{r+s=p, r \geq s \geq 1} \{T^*(r) + T^*(s) + p + \beta\}$$

Estas duas verificações, juntas com os fatos de que

$$p \text{ par} \Rightarrow \lceil \log_2(p/2) \rceil = \lceil \log_2 p \rceil - 1$$

$$p \text{ ímpar} \Rightarrow \lceil \log_2((p+1)/2) \rceil = \lceil \log_2 p \rceil - 1$$

implicam na validação de (I) para p . $\square$

Observe que para $p = 2^k$, este resultado reproduz o custo do “divide and conquer” da seção anterior.

Lema 3.2.2 *Uma estratégia ótima para obter o conjunto P , através de uniões aos pares de elementos dados e sob custo linear pode ser recursivamente especificado como:*

- (i) *A partição inicial P^0 de P é formada por seus elementos;*
- (ii) *Dada uma partição P^k de P , escolha os dois elementos de P^k tais que estes sejam os menores em tamanho dentre todos;*
- (iii) *Defina uma nova partição P^{k+1} , computando as uniões dos conjuntos escolhidos;*
- (iv) *Repita o processo até que a partição seja P .*

Prova :

Basta notarmos que qualquer partição P^k de tamanho k , é o resultado da união de dois elementos de uma outra partição ($P^j, j < k$), cujos tamanhos são $a(k)$ e $b(k)$ definidos por

$$\begin{cases} a(k) = 2^{\lceil \log_2 k \rceil} - i_k; \\ b(k) = 2^{\lceil \log_2 k \rceil - 1} + j_k \leq a(k); \\ a(k) + b(k) = k; \\ i_k \geq 0; j_k \geq 0; i_k j_k = 0. \end{cases}$$

Os valores $a(k)$ e $b(k)$ satisfazem a propriedade (II) do lema 3.2.1, isto é,

$$\lceil \log_2 a(k) \rceil - \lceil \log_2 (b(k) + 1) \rceil \leq 0,$$

o que conclui a demonstração. $\square$

A estratégia obtida no lema 3.2.2 é tipicamente uma estratégia gulosa. Os resultados obtidos nesta seção generalizam o a seção anterior, e é com estes resultados que trabalharemos a seguir.

3.3 O Produto Cartesiano entre Dois Vetores

Dados dois vetores, V_1 e V_2 , tais que $\#V_1 = m$ e $\#V_2 = n$ onde cada elemento de $V_i, i = 1, 2$ é da forma $V_i[j] = (x_{ij}^1, x_{ij}^2, \dots, x_{ij}^p)^t$, para p fixo, estamos interessados em obter um terceiro vetor V_3 , $\#V_3 = mn$, tal que cada elemento de V_3 é uma combinação de cada elemento de V_1 com V_2 , isto é, $V_3[i] = V_1[j] + V_2[k]$ para $i = (j - 1)n + k$, $1 \leq j \leq m, 1 \leq k \leq n$, onde a soma acima deve ser interpretada como uma soma componente a componente dos elementos $V_1[j]$ e $V_2[k]$ (ou seja, $x_{3i}^q = x_{1j}^q + x_{2k}^q$).

A obtenção de V_3 desta forma será denominada Produto Cartesiano entre V_1 e V_2 e denotada por $V_3 = V_1 \oplus V_2$.

Nosso especial interesse situa-se no caso de termos vetores V_1 e V_2 ordenados sob algum critério (por exemplo, no caso de termos como componentes $x_{1i}^1 \in \Re$ para $i = 1, \dots, m$, podemos ordenar V_1 por ordem crescente destas componentes), e desejamos obter $V_1 \oplus V_2$ também ordenado segundo este critério. Utilizando resultados das seções anteriores, obtemos o seguinte

Lema 3.3.1 *Dados V_1 e V_2 vetores ordenados, tais que $\#V_1 = m$ e $V_2 = n$ com $m \leq n$, uma estratégia ótima (do ponto de vista do número de operações computacionais) para se obter $V_1 \oplus V_2$ também ordenado é:*

(i) Tome cada elemento de V_1 (menor vetor) e some este elemento a cada elemento de V_2 , em ordem, gerando m novos vetores ordenados;

(ii) Realize o sort-merge destes m vetores segundo a estratégia ótima obtida no lema 3.2.2, isto é, a cada passo faça o sort-merge dos vetores de menor tamanho.

Prova :

A regra (ii) é uma consequência trivial do fato de estarmos sujeitos a custos lineares e do lema 3.2.2.

A regra (i) resulta do fato de que, com a escolha acima para se realizar as “soma parciais”, o custo ótimo é

$$T_0^* = n[m(\lceil \log_2 m \rceil + 1) - 2^{\lceil \log_2 m \rceil} n^{-1}(m - 1)]$$

e a escolha oposta (isto é, fixar o vetor maior) resulta no custo ótimo

$$T_1^* = m[n(\lceil \log_2 n \rceil + 1) - 2^{\lceil \log_2 n \rceil} m^{-1}(n - 1)]$$

e segue que

$$T_0^* \leq T_1^* \Leftrightarrow m \leq n. \quad \square$$

Considerando-se que para obtermos $V_1 \oplus V_2$ geram-se m vetores de tamanho n cada, e que estes foram concatenados da melhor forma (segundo o método guloso, que neste caso replica o “divide and conquer”), isto nos conduz a um custo computacional de $\mathcal{O}(mn \log_2 m)$, se $m \leq n$ (na verdade, gastamos mn somas na obtenção dos m vetores, mais $mn \log_2 m$ comparações para os sort-merges).

Sob a hipótese de que os elementos nos vetores obedeçam a uma distribuição uniforme, temos

Fato 3.3.1 *O custo esperado para o sort-merge de dois vetores com tamanhos p e q sob distribuição uniforme independente de seus elementos é*

$$E(p, q) = p(p + 1)^{-1} q(q + 1)^{-1} (p + q + 2)$$

Prova :

Segue diretamente de:

$$E(p, q) = 1 + p(p + q)^{-1} E(p - 1, q) + q(p + q)^{-1} E(p, q - 1)$$

com $E(1, 1) = 1$. □

A fórmula na demonstração acima apenas reflete o fato de que para fazermos o sort-merge entre dois vetores, comparamos o primeiro elemento de cada vetor (gasta-se 1 comparação), e com probabilidade $p/(p+q)$ o menor elemento é do primeiro vetor e com probabilidade $q/(p+q)$, o menor é do segundo vetor. Os detalhes da resolução da fórmula acima foram omitidos.

Cumpra notar que o caso médio e o pior caso estão muito próximos, ou seja, caso tenhamos $p \approx q$, então

$$E(p, q) \approx 2p^2(p+1)^{-1}$$

e

$$2p-2 \leq 2p^2(p+1)^{-1} \leq 2p-1.$$

Para os custos $2p-2$ e $2p-1$ sabemos que a estratégia gulosa proposta é a melhor, e assim conjecturamos que esta estratégia também é ótima para o custo médio acima.

Finalizando, notamos que a demonstração do lema 3.2.2 mostra a existência de “patamares” de otimalidade em termos de custo computacional para realizarmos o sort-merge (onde o custo não se altera, com pequenas alterações de estratégias). Nestes patamares, os “extremos” correspondem a estratégia gulosa, e o “meio” a estratégia “divide and conquer” para $m \approx 2^k$. Assim, no caso $m = 2^k$, este patamar corresponde a um único ponto, e portanto, a estratégia gulosa coincide com “divide and conquer”.

Capítulo 4

Problemas de Otimização Multicritério Multiestágio

Neste capítulo, vamos abordar uma descrição genérica de problemas multicritério, introduzindo a noção de Pareto-Otimalidade, além de caracterizar um problema multicritério em especial, que permitirá a busca de seus pontos Pareto-ótimos através de Programação Dinâmica, concluindo com uma descrição da estratégia ótima para a busca destes pontos.

4.1 Uma Breve Descrição

Em 1896, no tratado “Cours d’Economie Politique”, Pareto discutiu em termos qualitativos de valores e preços a formação de um método simples para determinar a eficiência entre vários critérios elementares essencialmente não comparáveis de um problema qualquer. Estes critérios elementares podem ser vistos como componentes de um vetor representando todos os estados do problema em questão, e busca-se a solução ótima perante uma relação de ordem entre os vetores.

Para melhor compreender as relações de ordem envolvidas entre vetores, vamos introduzir a seguinte notação:

Dados $x, y \in \mathbb{R}^m$, isto é, $x = (x_1, \dots, x_m)$ e $y = (y_1, \dots, y_m)$, e dadas as relações usuais $>$ e $\geq$ em $\mathbb{R}$, temos

- (i) $x \geq y$ sse $x_i \geq y_i$ para todo $i = 1, \dots, m$;
- (ii) $x \geq y$ sse $x \geq y$ e $x \neq y$;
- (iii) $x > y$ sse $x_i > y_i$ para todo $i = 1, \dots, m$.

Assim, temos que $\geq$ é uma ordem parcial em $\mathbb{R}^m$, enquanto $\geq$ e $>$ são ordens parciais estritas em $\mathbb{R}^m$, assim como $x > y \Rightarrow x \geq y \Rightarrow x \leq y$ mas nenhuma implicação reversa é verdadeira.

Se estamos interessados em problemas de otimização com somente um critério envolvido (ou uma função $f : X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$), temos claramente, pela completividade de $\mathbb{R}$, que este problema terá uma caracterização “absoluta” de seu(s) ponto(s) ótimo(s), isto é, basta encontrar $x \in X$ tal que $f(x) = \inf\{f(y) : y \in X\}$, no caso de problemas de minimização.

Mas, no caso multicritério, vimos que as relações de ordem envolvidas não são completas em $\mathbb{R}^m$, o que inviabiliza a caracterização de um ponto x como sendo ótimo através de $f^i(x) = \inf\{f^i(y) : y \in X \subseteq \mathbb{R}^n\}$, para $i = 1, \dots, m$. Se fosse esta a forma de buscar a solução do problema em questão, vários problemas multicritério em engenharia, economia, estatística e teoria dos jogos, entre outros, simplesmente não teriam solução, pois vários dos critérios envolvidos não possuem qualquer relação de proporcionalidade aos demais, isto é, se $f^i(x) > f^i(y)$, não necessariamente temos $f^j(x) > f^j(y)$, para todo $j \neq i$. Assim, para problemas multicritério, é necessário outra caracterização para os pontos que servirão como solução do problema, e esta caracterização, proposta por Pareto, que basicamente direciona a busca de todos os possíveis pontos que possam ter algum interesse no problema, assumiu vários sinônimos desde sua criação, como “Pareto-Otimalidade”, “eficiência (no sentido de Pareto)”, “admissibilidade”, “não-superioridade (ou não-inferioridade)” e “não-dominância”.

Os problemas multicritério em estudo podem, então, ser modelados da seguinte forma:

Dado $f : X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^m$, encontre, se existir, $x^* \in X$ tal que $\nexists x \in X : f(x) \leq f(x^*)$ para $f(y) = (f^1(y), \dots, f^m(y)) \forall y \in X$, onde $f^i()$, para $i = 1, \dots, m$, são critérios elementares.

Podemos assumir que eventualmente existam outros critérios envolvidos, mas estes m são os mais importantes e nenhum deles tem maior prioridade sobre os demais.

4.1.1 Pareto-Otimalidade

Dado $f = (f^1, \dots, f^m) : X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^m$, um ponto x é dito “Não-Dominado” (ou “eficiente”) se $\nexists y \in X$ tal que $f^i(x) \geq f^i(y), \forall i = 1, \dots, m$ (Lin [18] e Debreu [6]).

Intuitivamente, nota-se que qualquer ponto $y \in X$ dominado por algum

$x \in X$ é um ponto “menos interessante” do que y , pois $f^i(y) \geq f^i(x)$ significa que “ x é pelo menos tão bom quanto y nos critérios i ”, e $f^j(y) > f^j(x)$ afirma que “ x é melhor que y no critério j ”. Logo, y não pode ser um ponto de ótimo, enquanto x , eventualmente, pode.

4.2 Busca de Pontos Eficientes por Programação Dinâmica

4.2.1 Caracterização da Busca de Pontos Eficientes

Dados

$X_i \subseteq \mathbb{R}^{n_i}$, X_i finito para $i = 1, \dots, n$,

$f : X_1 \times \dots \times X_n \rightarrow \mathbb{R}^m$, onde

$f_i^k : X_i \rightarrow \mathbb{R}$, $i = 1, \dots, n$, $k = 1, \dots, m$,

$g_i : X_i \rightarrow \mathbb{R}^{l_i}$ $i = 1, \dots, n$,

Encontre, se existir, o conjunto $E = \{x \in X / \nexists y \in X : f(x) \geq f(y)\}$, onde

$X = \{x = (x_1, \dots, x_n) : x_i \in X_i \text{ e } g_i(x) \leq 0 \text{ } i = 1, \dots, n\}$.

Nestas condições, temos um problema de otimização multicritério onde o conjunto restrição é descrito por uma partição finita, e desta forma podemos determinar o conjunto E através da enumeração dos elementos de X_i , buscando-se os pontos eficientes em uma combinação de pontos de $X_1, \dots, X_n$. O problema de decisão associado corresponde a descrever a ordem de escolha das variáveis a serem agregadas na composição da busca de pontos eficientes, ou seja, o ordenamento dos X_i de tal forma que as composições entre todos os possíveis pontos de X_i para obter o conjunto E custe o menos possível. Esta forma de busca de pontos eficientes (ou construção do conjunto E) só é possível devido à separabilidade das funções f^i (veja Capítulo 2), isto é, $f : X_1 \times \dots \times X_n \rightarrow \mathbb{R}^m$ poder ser calculada como

$$f(x_1, \dots, x_n) = \sum_{i=1}^n f_i(x_i) = \left(\sum_{i=1}^n f_i^1(x_i), \dots, \sum_{i=1}^n f_i^m(x_i) \right)^t.$$

Desta forma, associaremos a pontos $x_I \in X_{i_1} \times \dots \times X_{i_p}$, para $I = \{i_1, \dots, i_p\}$ o valor da função critério f^j como sendo $f_I^j = \sum_{i \in I} f_i^j(x_i)$, e portanto $f_I = \sum_{i \in I} f_i(x_i)$.

Uma forma de modelar o processo de busca do conjunto E através da enumeração dos elementos de X_i é através da construção de um grafo representando esta enumeração. O grafo $G = (VG, AG)$ teria um nó, digamos s ,

tal que ele representa o estado inicial da busca (ou seja, nenhum ponto de nenhum X_i foi sequer analisado). A partir de s , ligam-se por arestas todos os seus nós sucessores, representando cada ponto viável de um X_i específico a ser analisado. A partir de cada nó destes, ligam-se por arestas todos os seus sucessores, que representam todas as combinações deste elemento de X_i com os elementos viáveis de um $X_j, j \neq i$, e assim sucessivamente, vai se construindo uma árvore a partir de s . Ao final, ligam-se todos os nós que foram criados por último (isto é, do último nível) a um nó t , representando todas as possíveis combinações de todos os elementos viáveis de X .

Este grafo G representa, em cada nível a partir de s a t , uma agregação de um novo ponto $x_j \in X_j$ aos pontos do tipo $x_I \in X_{i_1} \times \dots \times X_{i_k}$, onde $I = \{i_1, \dots, i_k\}$, fazendo com que construamos “dinamicamente” o conjunto E . Claramente, este processo pode conduzir à formação de combinações dominadas em relação à função critério f . Assim sendo, estas combinações podem ser eliminadas do grafo, pois sabe-se que elas não nos conduzirão a pontos eventualmente ótimos. Ao final, eliminando-se as combinações ineficientes e agregando-se todos os pontos de X^i , para todo $i = 1, \dots, n$, teremos construído o conjunto E .

4.2.2 Custo da Busca de Pontos Eficientes

O problema da busca de pontos eficientes torna-se especialmente interessante se atribuírmos custos às arestas de G , que simplesmente representam o custo computacional de se agregar um novo elemento a combinação de elementos já calculados. Com isso, a busca de pontos eficientes de um problema multicritério nos moldes descritos neste capítulo replica o problema de se obter o caminho mais curto no grafo G , que resolvido, fornece-nos a melhor estratégia de se agrupar os elementos dos $X_i, \forall i = 1, \dots, n$, na busca do conjunto E das soluções não dominadas do problema dado. Como visto no Capítulo 2, este problema (de obter o conjunto dos pontos não dominados) pode (e será) resolvido por Programação Dinâmica.

Cada nó $n_I \in VG$ pode ser descrito através da seguinte tabela:

$f_I^1(x_I^1)$	$f_I^1(x_I^2)$	$\dots$	$f_I^1(x_I^k)$
$f_I^2(x_I^1)$	$f_I^2(x_I^2)$	$\dots$	$f_I^2(x_I^k)$
$\dots$	$\dots$	$\dots$	$\dots$
$f_I^n(x_I^1)$	$f_I^n(x_I^2)$	$\dots$	$f_I^n(x_I^k)$
x_I^1	x_I^2	$\dots$	x_I^k

onde $k = \#\{x_I : x_i \text{ é eficiente para } f_I()\}$,

$$\text{e } f_I^1(x_I^1) \leq f_I^1(x_I^2) \leq \dots \leq f_I^1(x_I^k).$$

A ordenação por $f^1()$ se dá pelo simples fato de, com isso, podermos identificar mais facilmente elementos dominados, se estes aparecerem, e assim, eliminá-los da tabela, e também para termos pelo menos uma função (ou critério) guiando o processo de busca das soluções eficientes do problema em estudo (note que a ordenação poderia ser feita com qualquer uma das n funções).

Assim, andar por um nível do grafo G a partir do nó s , ou agregar um ponto $x_J \in X_J$ onde $J = \{j_1, \dots, j_l\}$ a um outro ponto $x_I \in X_I$, $I = \{i_1, \dots, i_r\}$ com $I \cap J = \emptyset$, corresponde a realizar-se o produto cartesiano entre as tabelas de x_I e x_J , como descrito no Capítulo 3. Lá foi feita uma análise de pior caso para o custo de produtos cartesianos, e, nas condições descritas acima, dados os nós n_I (associado a x_I) e n_J (associado a x_J) de VG , seja $n_{I \cup J}$ o nó representando a concatenação de x_I e x_J , e com isso, o custo das arestas $(n_I, n_{I \cup J})$ e $(n_J, n_{I \cup J})$ é de $|I||J| \log_2 \min\{|I|, |J|\}$. Ainda sob as condições descritas, cada coluna da tabela associada a $x_{I \cup J}$ seria:

$$\begin{vmatrix} f_I^1(x_I^p) + f_J^1(x_J^q) \\ f_I^2(x_I^p) + f_J^2(x_J^q) \\ \dots \\ f_I^n(x_I^p) + f_J^n(x_J^q) \\ x_{I \cup J} \end{vmatrix}$$

Desta forma, vemos que o problema de otimização multicritério descrito no início desta seção, tratado por Programação Dinâmica pode ter vários custos computacionais, conforme a sequência de pontos escolhidos para uma nova combinação. O problema de obter a melhor estratégia para se fazer o produto cartesiano entre estas tabelas pode também ser analisado sob o enfoque de Programação Dinâmica, o que será feito a seguir.

4.2.3 Estratégia Ótima para Busca de Pontos Eficientes

Seja T_I a tabela associada ao nó $n_I \in VG$, para $I \in 2^{VG}$ (onde 2^{VG} é a notação para o conjunto das partes de VG). Considerando-se já conhecidos os nós $n_{\{1\}}, \dots, n_{\{k\}}$, e portanto, conhecidas as tabelas $T_{\{1\}}, \dots, T_{\{k\}}$, queremos obter T_K para $K = \{1, \dots, k\}$.

Pela notação introduzida no Capítulo 3, temos:

(i) $T_I \oplus T_J$ é a tabela associada a $n_{I \cup J}$, obtida pelo produto cartesiano de T_I e T_J ;

(ii) Para não carregar mais ainda a notação, consideraremos $\{T_1, \dots, T_k\} = \{T_{\{1\}}, \dots, T_{\{k\}}\}$, isto é, $T_i = T_{\{i\}}$ para todo i . Assim, nos referenciaremos às k tabelas representando nós associados aos conjuntos $X_i, i = 1, \dots, k$ apenas por $T_1, \dots, T_k$;

(iii) $T_J \stackrel{\text{def}}{=} \bigoplus_{j=1}^p T_j$ se $T_J = ((T_1 \oplus T_2) \oplus \dots) \oplus T_p$, ou seja, T_J , para $J = \{1, \dots, p\}$ será denotada da forma definida acima, se T_J for obtida do produto cartesiano de T_1 com T_2 , tomando-se o resultado deste produto e obtendo um novo produto cartesiano com T_3 , e assim sucessivamente, até obter $T_J = (((T_1 \oplus T_2) \oplus T_3) \oplus \dots) \oplus T_p$;

(iv) Consideraremos o custo de obter $(T_I \oplus T_J)$ como sendo $\|T_I \oplus T_J\| = t_I \times t_J \times \log_2(t^*) + \|T_I\| + \|T_J\|$, onde t_I é o tamanho da tabela T_I , t_J o tamanho de T_J e $t^* = \min\{t_I, t_J\}$. Este custo advém da análise de pior caso no processamento computacional de produtos cartesianos, feito no Capítulo 3;

(v) Sem perda de generalidade, assumiremos $t_1 \geq t_2 \geq \dots \geq t_k \geq 2$. A ordenação é consistente com a hipótese (ii) acima, e o limite inferior dos tamanhos das tabelas como sendo igual a 2 simplesmente reflete o fato de que se houver alguma tabela $T_W, W \in 2^{V_G}$ tal que $t_W = 1$, então existe um único ponto viável para o nó n_W . Assim, esta tabela em nada acrescentará em termos de custos computacionais na obtenção da melhor estratégia para a busca de pontos eficientes (os produtos cartesianos com esta tabela são feitos à priori, resguardando as informações pertinentes);

(vi) Consideraremos que $\#(T_I \oplus T_J) = t_I t_J$. Isto afirma que, ao gerar-se $n_{I \cup J}$ através de x_I e x_J , este não terá pontos dominados (ou não eficientes). Claramente, esta é uma suposição válida somente para a análise de pior caso. Mais adiante, consideraremos a hipótese de que pontos dominados surjam na dinâmica da estratégia ótima, descrita a seguir:

Teorema 4.2.1 *Sob as hipóteses (i) a (vi) descritas acima, uma política ótima que minimiza o custo computacional para se obter T_K é:*

$$T_N = (((\dots(T_1 \oplus T_2) \oplus T_3) \oplus \dots) \oplus T_n) \stackrel{\text{def}}{=} \bigoplus_{j=1}^n T_j$$

Este teorema nos mostra que fazer sequencialmente como descrito os produtos cartesianos de $T_{\{1\}}, \dots, T_{\{k\}}$ minimiza o custo computacional desta operação.

Vamos provar este resultado concatenando dois lemas. O primeiro deles afirma que se deseja-se realizar os produtos cartesianos em um conjunto de

tabelas, uma de cada vez (isto é, sequencialmente), então a melhor forma de fazê-lo é ordenar as tabelas em ordem não decrescente nos seus tamanhos, desconsiderando deste processamento as tabelas de tamanho unitário (de acordo com (v)):

Lema 4.2.1 *Dado qualquer conjunto de variáveis $J = \{i_1, \dots, i_p\}$, onde $t_{i_j} \geq 2, j = 1, \dots, p$, se restringirmos as possibilidades de obter T_J a estratégias do tipo sequencial, isto é,*

$$T_J = \bigoplus_{k=1}^p T_{j_k}$$

onde $\{j_k\}_{k=1}^p$ é uma permutação de $\{i_k\}_{k=1}^p$,

então obtemos o menor custo computacional através da estratégia de processar os produtos cartesianos em ordem não decrescente no tamanho das tabelas.

Prova :

Observando que a estratégia acima descrita é equivalente a deixarmos por último no processamento dos produtos cartesianos a tabela de menor tamanho (se maior que 1), o resultado segue diretamente se provarmos para $p = 3$. A análise de apenas 3 tabelas é suficiente, pois em qualquer combinação de um número maior de tabelas haverá, em algum momento, uma configuração envolvendo momentaneamente 3 delas.

Vamos considerar, então, sem perda de generalidade que $\{i_j\}_{j=1}^3$ com $t_1 \geq t_2 \geq t_3$. Segue que

$$\|(T_1 \oplus T_2) \oplus T_3\| = t_1 t_2 (\log_2 t_2 + t_3 \log_2 t_3);$$

$$\|(T_1 \oplus T_3) \oplus T_2\| = t_1 t_3 (\log_2 t_3 + t_2 \log_2 t_2);$$

$$\|(T_2 \oplus T_3) \oplus T_1\| = t_2 t_3 (\log_2 t_3 + t_1 \log_2 \bar{t});$$

onde $\bar{t} = \min \{t_2 t_3, t_1\}$.

Como $h(x) = [(x-1)^{-1}(x \log_2 x)]$ é estritamente monotônica crescente para $x > 1$ e o tamanho das tabelas são maiores que 1, temos

$$(i) \|(T_1 \oplus T_2) \oplus T_3\| \leq \|(T_1 \oplus T_3) \oplus T_2\|$$

$$\Leftrightarrow t_1 t_2 \log_2 t_2 + t_1 t_2 t_3 \log_2 t_3 \leq t_1 t_3 \log_2 t_3 + t_1 t_2 t_3 \log_2 t_3$$

$$\Leftrightarrow t_3 \log_2 t_3 (t_2 - 1) \leq t_2 \log_2 t_2 (t_3 - 1)$$

$$\Leftrightarrow \frac{t_3 \log_2 t_3}{t_3 - 1} \leq \frac{t_2 \log_2 t_2}{t_2 - 1}$$

Usando o fato de que

$$t_1 t_2 (\log_2 t_2 + t_3 \log_2 t_3) \leq t_1 t_2 (\log_2 t_1 + t_3 \log_2 t_3) = \alpha,$$

pois $\log_2 t_2 \leq \log_2 t_1$, segue que

$$(ii) t_1 t_2 \log_2 t_1 + t_1 t_2 t_3 \log_2 t_3 \leq t_2 t_3 \log_2 t_3 + t_1 t_2 t_3 \log_2 t_1$$

$$t_1 \log_2 t_1 + t_1 t_3 \log_2 t_3 \leq t_3 \log_2 t_3 + t_1 t_3 \log_2 t_1$$

$$\Leftrightarrow t_3 \log_2 t_3 (t_1 - 1) \leq t_1 \log_2 t_1 (t_3 - 1)$$

$$\Leftrightarrow \frac{t_3 \log_2 t_3}{t_3 - 1} \leq \frac{t_1 \log_2 t_1}{t_1 - 1}, \text{ e}$$

$$(iii) \ t_1 t_2 \log_2 t_2 + t_1 t_2 t_3 \log_2 t_3 \leq t_2 t_3 \log_2 t_3 + t_1 t_2 t_3 \log_2 (t_2 t_3)$$

$$t_1 \log_2 t_2 + t_1 t_3 \log_2 t_3 \leq t_3 \log_2 t_3 + t_1 t_3 \log_2 t_2 + t_1 t_3 \log_2 t_3$$

$$\Leftrightarrow t_1 \log_2 t_2 \leq t_3 \log_2 t_3 + t_1 t_3 \log_2 t_2 \ (t_3 \geq 2).$$

De (i), (ii) e (iii), segue a tese. $\square$

A política acima descrita é denominada Política Sequencial Ótima. A importância desta política se faz sentir no próximo lema, que afirma que, dada qualquer estratégia de se computar os produtos cartesianos de um conjunto de tabelas T_i com $t_i > 1$, sempre existe uma estratégia do tipo sequencial pelo menos tão boa quanto esta:

Lema 4.2.2 *Dada qualquer estratégia para computar T_N com custo C^0 , existe uma estratégia sequencial que computa T_N com um custo menor ou igual a C^0 .*

Prova :

A prova é por indução em $n = \#N$.

Trivialmente para $n = 1, 2, 3$, qualquer estratégia para computar T_N é sequencial. Podemos usar este fato como base da indução, mas para tornar mais claro o processo de prova, vamos mostrar a afirmação para $n = 4$:

Base ($n=4$): Claramente ou T_N é computado em uma forma sequencial (nada a provar) ou

$$T_N = (T_i \oplus T_j) \oplus (T_k \oplus T_l) \quad (1)$$

onde (i, j, k, l) é uma permutação de $(1, 2, 3, 4)$. Assim, vamos assumir, sem perda de generalidade, que

$$t_k t_l \leq t_i t_j \text{ e } t_k \leq t_l \quad (\Rightarrow t_l \leq t_i t_j).$$

Então, o custo de computar T_N , como em (1), é

$$C^0 = \|(T_i \oplus T_j) \oplus (T_k \oplus T_l)\| = \|T_i \oplus T_j\| + \|T_k\| + \|T_l\| + t_l t_k \log_2 t_k + t_i t_j t_k t_l \log_2 (t_k t_l).$$

Considere, agora, T_N computado como em um processo sequencial descrito abaixo:

$$T_N = ((T_i \oplus T_j) \oplus T_k) \oplus T_l \quad (2)$$

Analogamente

$$C^1 = \|((T_i \oplus T_j) \oplus T_l) \oplus T_k\| = \|T_i \oplus T_j\| + \|T_k\| + \|T_l\| + t_i t_j t_l \log_2 t_l + t_i t_j t_l t_k \log_2 t_k.$$

Então,

$$C^0 - C^1 = t_l t_k \log_2 t_k + t_l t_j t_l (t_k - 1) \log_2 t_l > 0$$

(para $t_l \geq t_k > 1$).

Passo de Indução : Vamos assumir que T_N foi computado de qualquer outra maneira com custo C^0 , e que o último produto cartesiano envolveu (I, J) uma partição de N , isto é,

$$T_N = T_I \oplus T_J \quad \text{e}$$

$$C^0 = \|T_I\| + \|T_J\| + t_I t_J \log_2 \min\{t_I, t_J\}.$$

Se $\#I = 1$ ou $\#J = 1$, a indução segue diretamente. Por outro lado, considerando $t_I < t_J$ temos $T_N = T_J \oplus (T_L \oplus T_K)$ e

$$C^0 = \|T_J\| + \|T_K\| + \|T_L\| + t_K t_L t_J \log_2 (t_K t_L) + t_K t_L \log_2 t_K.$$

onde (K, L) é uma partição de I e $t_K \leq t_L$.

Pelo mesmo argumento como em $n = 4$, o custo C^0 é maior do que o custo C^1 , que ocorre quando computamos T_N como $T_N = (T_J \oplus T_L) \oplus T_K$.

Repetindo esta construção até $\#K = 1$ e aplicando o passo de indução para $(T_J \oplus T_L)$, o resultado segue. $\square$

A concatenação dos dois lemas demonstra o Teorema 4.2.1.

Cumpramos notar que o problema multicritério descrito neste capítulo sob a forma de Programação Dinâmica (e reduzido a um problema de caminho mais curto em grafo), tem sua solução obtida sob uma abordagem também de Programação Dinâmica, como se verifica nas demonstrações dos lemas.

Para tentar eliminar a hipótese (vi) acima, isto é, a hipótese de que em um produto cartesiano qualquer não se verificam pontos dominados, vamos substituí-la por

$$(vii) \#(T_I \oplus T_J) = t_I + t_J.$$

A suposição de que um produto cartesiano, que deveria gerar uma tabela de tamanho $t_I t_J$, gere somente $t_I + t_J$ é razoavelmente forte, e é consistente no seguinte sentido: qualquer que seja a estratégia para o processamento dos produtos cartesianos, esperamos que o conjunto de pontos eficientes, ao final, seja exatamente o mesmo, e, portanto, o tamanho da tabela final seja sempre o mesmo. Se supuséssemos que, por exemplo, $\#(T_I \oplus T_J) = \lceil \sigma t_I t_J \rceil$, para $\sigma \in (0, 1)$, simplesmente não haveria mais garantias de que,

independentemente da estratégia adotada, chegaríamos ao final com a tabela do mesmo tamanho, pois estratégias com números diferentes de iterações conduziriam a diferentes tamanhos nesta estimativa, o que não ocorre com a suposição (vii) acima.

Assim, sob a hipótese (vii), resta mostrar a validade dos lemas 4.2.1 e 4.2.2 (e conseqüentemente, do teorema 4.2.1). Para o lema 4.2.1, verificamos, seguindo a sua demonstração, que

$$\begin{aligned} \|(T_1 \oplus T_2) \oplus T_3\| &= t_1 t_2 \log_2 t_2 + (t_1 + t_2) t_3 \log_2 t_3 = \alpha; \\ \|(T_1 \oplus T_3) \oplus T_2\| &= t_1 t_3 \log_2 t_3 + (t_1 + t_3) t_2 \log_2 t_2 = \beta; \\ \|(T_2 \oplus T_3) \oplus T_1\| &= t_2 t_3 \log_2 t_3 + (t_2 + t_3) t_1 \log_2 \bar{t} = \gamma; \\ \text{onde } \bar{t} &= \min \{t_2 t_3, t_1\}, \text{ e } t_1 \geq t_2 \geq t_3 \text{ e segue que} \\ \beta - \alpha &= t_2 t_3 (\log_2 t_2 - \log_2 t_3) \geq 0, \text{ e} \\ \gamma - \alpha &= t_1 t_2 (\log_2 \bar{t} - \log_2 t_2) + t_1 t_3 (\log_2 \bar{t} - \log_2 t_3) \geq 0. \end{aligned}$$

recaindo na mesma conclusão do lema 4.2.1. Quanto ao lema 4.2.2, temos, para $n = 4$:

$$C^0 = \|(T_i \oplus T_j) \oplus (T_k \oplus T_l)\| = \|T_i \oplus T_j\| + \|T_k\| + \|T_l\| + t_l t_k \log_2 t_k + (t_i + t_j)(t_k + t_l) \log_2 (t_k + t_l).$$

e para o processo sequencial

$$C^1 = \|((T_i \oplus T_j) \oplus T_l) \oplus T_k\| = \|T_i \oplus T_j\| + \|T_k\| + \|T_l\| + (t_i + t_j) t_l \log_2 t_l + (t_i + t_j + t_l) t_k \log_2 t_k.$$

(para $t_l \geq t_k > 1$).

e

$$C^0 - C^1 \geq 0$$

A sequência da demonstração segue analogamente.

Capítulo 5

Resolução de um Problema Bicritério

Neste capítulo vamos apresentar um problema que possui dois critérios de interesse, o problema de Designação de Capacidades em Redes de Computadores (denotado “CA”), e mostrar como ele pode ser resolvido pela abordagem do Capítulo 4. Vamos também mostrar como este problema pode ser resolvido por técnicas que envolvem algoritmos informados (mais especificamente, o A^*).

5.1 Definição do CA

Dada uma rede de computadores, que pode ser representada por um grafo $G = (VG, AG)$, com $\#VG = n$, $\#AG = m$, onde os nós representam os computadores de mensagens e as arestas, os canais de comunicação, e sabendo-se previamente os seguintes dados desta rede:

(i) Topologia da rede, isto é, o esquema de ligação entre seus nós através dos canais de comunicação, que são canais “full-duplex” (possuem a capacidade de transmitir informações de qualquer uma de suas pontas (nós) à outra);

(ii) Roteamento das mensagens, isto é, o caminho que cada mensagem percorre do nó fonte da transmissão ao nó destino;

(iii) Demandas na rede, isto é, o número médio de mensagens que trafegam entre cada par de nós (denotado por $\gamma_{ij} \stackrel{\text{def}}{=} \text{número médio de mensagens do nó } i \text{ para o nó } j$, e $\gamma \stackrel{\text{def}}{=} \sum_i \sum_j \gamma_{ij}$, para todo par de nós $(i, j) \in VG \times VG$). Com isso saberemos o fluxo nos canais, isto é, a quantidade de

mensagens média que passa em cada canal, denotada por $f_i \stackrel{\text{def}}{=} \text{fluxo médio no canal } i$;

(iv) O conjunto de capacidades disponíveis no mercado para a instalação na rede, denotada por $\mathcal{C}$;

(v) Custo de instalação da capacidade c_i em algum arco, denotado por $D_j(c_i)$, isto é, $D_j : C^j \rightarrow \mathbb{R}_{++}$ onde $C^j =$ conjunto de capacidades que podem ser instaladas em um canal j . Este custo, em geral, é dado por $D_j(c_i) = d_i c_i^\alpha$, onde $\alpha \in [0, 1]$ e d_i é o custo nominal associado à capacidade c_i . Consideraremos, de maneira geral, que $C^j = \mathcal{C}$;

Estaremos interessados em designar um valor de capacidade para cada canal da rede, tal que este minimize o custo total das instalações, respeitando uma restrição de que o atraso (retardo) médio das transmissões das mensagens na rede não será superior a um certo valor limite (denotado T_{max}), ou então tal que as capacidades instaladas minimizem o atraso (retardo) médio das comunicações, sem ultrapassar o orçamento previsto (denotado D_{max}). A este problema denominamos CA.

Dada a caracterização acima ((i) a (v)), se definirmos $\lambda_i =$ número médio de mensagens por segundo que passa no canal i , e portanto, $\lambda_i = \sum_j \sum_k \gamma_{jk}$ para j e k nós que se utilizam do canal i nas comunicações entre si (note que $\gamma = \sum_{i \in AG} \lambda_i$) e $N_k =$ número médio de mensagens no canal k , temos que $N_k = \lambda_k T_k$, para $T_k =$ atraso médio no canal k .

Assim,

$$\gamma T = \sum_{k=1}^m N_k = \sum_{k=1}^m \lambda_k T_k \Rightarrow T = \frac{1}{\gamma} \sum_{i=1}^m \lambda_i T_i.$$

Claramente $\gamma > 0$, pois temos um número positivo de mensagens na rede (não há o que estudar se não houver mensagens na rede).

Desejamos que T_i dependa somente de suas características associadas a fluxo e velocidade dos canais. Para garantir isto, algumas hipóteses são feitas na modelagem deste problema (veja Kleinrock [16]):

- chegadas externas nos comutadores segundo um processo de Poisson;
- tamanho das mensagens sob uma distribuição exponencial com média $\frac{1}{\mu}$;
- independência entre o tempo de chegada e o tamanho das mensagens;

Pode-se mostrar que, nas hipóteses acima, cada canal representa uma fila M/M/1 com chegadas de Poisson com média λ_i e

$$T = \sum_{i=1}^m \frac{\lambda_i}{\gamma} \left(\frac{1}{\mu c_i - \lambda_i} \right)$$

garantindo a separabilidade da função retardo T . Sabendo-se que $f_i = \frac{\lambda_i}{\mu}$ (fluxo em bits por segundo), segue que

$$T = \frac{1}{\gamma} \sum_{i=1}^m \frac{f_i}{c_i - f_i}.$$

Assim, o problema CA pode ser formalmente descrito como:

(P1)

$$D(C) = \min \sum_{i=1}^m d_i c_i^\alpha, \quad \alpha \in [0, 1]$$

sujeito a

$$\frac{1}{\gamma} \sum_{i=1}^m \frac{f_i}{c_i - f_i} \leq T_{max}$$

$$c_i > f_i, \quad c_i \in \mathcal{C}, \quad i = 1, \dots, m$$

ou também descrito:

(P2)

$$T(C) = \min \frac{1}{\gamma} \sum_{i=1}^m \frac{f_i}{c_i - f_i}$$

sujeito a

$$\sum_{i=1}^m d_i c_i^\alpha \leq D_{max}, \quad \alpha \in [0, 1]$$

$$c_i > f_i, \quad c_i \in \mathcal{C}, \quad i = 1, \dots, m$$

Note que $f_i = 0 \Rightarrow c_i = 0$ e portanto só consideraremos canais com fluxo positivo.

O problema CA é um problema bicritério (isto é, temos dois critérios de interesse: custo e retardo), e como visto acima, pode essencialmente ser resolvido fixando-se um dos critérios em parâmetros satisfatórios, e minimizando-se o outro critério. Assim, de acordo com o que foi descrito no Capítulo 4, nós não estamos encontrando todo o conjunto de pontos eficientes para o CA, mas buscaremos o conjunto de pontos eficientes para estas caracterizações.

Quando ocorre de $\mathcal{C} = \mathfrak{R}_{++}$, este problema resolve-se facilmente da seguinte forma:

(a) Se $\alpha = 0$ (custos lineares):

O problema (P1) tem a seguinte solução:

$$c_i^* = f_i + \frac{1}{\gamma} \left(\frac{\sum_{i=1}^m \sqrt{f_i d_i}}{T_{max}} \right) \sqrt{\frac{f_i}{d_i}} \quad (1 \leq i \leq m)$$

com custo

$$D^* = \sum_{i=1}^m (d_i f_i) + \frac{(\sum_{i=1}^m \sqrt{d_i f_i})^2}{\gamma T_{max}}$$

e o problema (P2) tem solução:

$$c_i^* = f_i + \left(\frac{D_{max} - \sum_{i=1}^m d_i f_i}{\sum_{i=1}^m \sqrt{d_i f_i}} \right) \sqrt{\frac{f_i}{d_i}}$$

com retardo

$$T^* = \frac{1}{\gamma} \left(\frac{(\sum_{i=1}^m \sqrt{d_i f_i})^2}{D_{max} - \sum_{i=1}^m d_i f_i} \right)$$

A demonstração destas fórmulas encontra-se em Kleinrock [16], e baseia-se na obtenção de pontos que satisfaçam as condições de Lagrange. Note que no ótimo a restrição de " $\leq$ " deve ser obedecida com igualdade (senão, cresce-se um pouco mais o retardo decaindo o custo, no caso de (P1), ou cresce-se as capacidades decaindo o retardo para (P2)) e vale a hipótese de regularidade de Lagrange. Mais ainda, como d_i e T_i são diferenciáveis e estritamente monotônicas em c_i , Lagrange é condição necessária e suficiente.

Para demonstrar (P2), seja

$$G = T + \beta \left(\sum_{i=1}^m d_i c_i - D_{max} \right)$$

onde β é um multiplicador a ser determinado. O valor mínimo de G é obtido para $\sum_{i=1}^m d_i c_i = D_{max}$ e como a solução ótima deve satisfazer $\frac{dG}{dc_i} = 0$ para $i = 1, \dots, m$ segue que

$$\left(-\frac{\lambda_i}{\gamma}\right) \frac{\mu}{(\mu c_i - \lambda_i)^2} + \beta d_i = 0$$

ou

$$c_i = \frac{\lambda_i}{\mu} + \frac{1}{\sqrt{\beta\gamma\mu}} \sqrt{\frac{\lambda_i}{d_i}}, \quad i = 1, \dots, m$$

$$\Rightarrow \sum_{i=1}^m d_i c_i = \sum_{i=1}^m \frac{\lambda_i d_i}{\mu} + \frac{1}{\sqrt{\beta\gamma\mu}} \sum_{i=1}^m \sqrt{\lambda_i d_i}$$

Igualando o segundo termo a D_{max} temos

$$\frac{1}{\sqrt{\beta\gamma\mu}} = \frac{D_{max} - \sum_{i=1}^m (\lambda_i d_i / \mu)}{\sum_{i=1}^m \sqrt{\lambda_i d_i}}$$

$$\begin{aligned} \Rightarrow c_i &= \frac{\lambda_i}{\mu} + \left(\frac{D_{max} - \sum_{i=1}^m (\lambda_i d_i / \mu)}{d_i} \right) \left(\frac{\sqrt{\lambda_i d_i}}{\sum_{i=1}^m \sqrt{\lambda_i d_i}} \right) = \\ &= f_i + \left(\frac{D_{max} - \sum_{i=1}^m d_i f_i}{\sum_{i=1}^m \sqrt{d_i f_i}} \right) \sqrt{\frac{f_i}{d_i}}. \end{aligned}$$

Como o mínimo existe e obedece Lagrange, e como Lagrange só é obedecido pela fórmula acima, segue que a solução obtida é ótima.

Para (P1), o resultado segue de forma análoga.

(b) Se $\alpha \in (0, 1]$ (função custo côncava):

Resolve-se (P1) da seguinte forma:

Seja $D_L(c) = \langle \nabla D(\bar{c}), c - \bar{c} \rangle$ a função custo linearizada em torno de $\bar{c}$. Como D_L é uma função linear, esta pode ser resolvida facilmente como visto em (a), e assim busca-se a solução de (P1) através de aproximações sucessivas do problema linearizado, da seguinte forma:

Escolha um conjunto de capacidades viáveis $\bar{c}$;

Repita

$c \leftarrow \bar{c}$;

$\bar{c} \leftarrow$ solução com custos D_L ;

até que c e $\bar{c}$ estejam suficientemente próximos.

Apesar do esquema de linearizações externas para minimizações de funções côncavas não ser muito “natural”, a convergência deste algoritmo é mostrada em Humes [13], e testes computacionais, inclusive alguns por nós realizados, mostraram uma rápida convergência deste para a solução ótima. A solução de (P2) é análoga.

O problema CA, no entanto, em aplicações reais, apresenta $\mathcal{C}$ finito. Isto torna o problema um “problema da mochila”, que até no caso linear é caracterizado como $\mathcal{NP}$ -completo (veja Garey-Johnson [8]). Assim, o problema assume um caráter tipicamente combinatório e de difícil resolução. No entanto, a seguir mostraremos duas formas de resolvê-lo: uma encarando o problema como um processo de decisões sequenciais (onde a cada passo decidiremos que canal vamos estudar e atribuir alguma capacidade), e com isso passível de resolução por programação dinâmica, e outra resolução tirando proveito de informações que podem facilmente ser obtidas do problema, e assim usando um algoritmo informado.

Daqui para frente, em virtude da semelhança entre os problemas (P1) e (P2), consideraremos apenas (P1), com a restrição $\mathcal{C}$ finito (designado CAD).

5.2 Resolução do CAD através de Programação Dinâmica

Como descrito acima, o CA é encarado como um problema bicritério, onde devemos minimizar um dos critérios obedecendo restrições ao outro critério. No nosso caso, ambos os critérios têm tendências opostas, isto é, se procurarmos diminuir o valor do parâmetro D (custo), tenderemos a crescer o valor de T (retardo), e vice-versa. Este problema pode ser tratado sob a ótica descrita no Capítulo 4, onde teremos uma tabela (descrita a seguir) para cada arco da rede, e estamos interessados em fazer o produto cartesiano destas tabelas em busca dos pontos não dominados e da solução ótima do problema (P1).

A cada arco i da rede, associa-se uma tabela do tipo

D_{i_1}	D_{i_2}	$\dots$	D_{i_k}
T_{i_1}	T_{i_2}	$\dots$	T_{i_k}
$\dots$	$\dots$	$\dots$	$\dots$
c_{i_1}	c_{i_2}	$\dots$	c_{i_k}

onde c_{i_j} é uma capacidade viável para o arco i ($c_{i_j} > f_i$), D_{i_j} é o custo associado a instalar-se esta capacidade ($D_{i_j} = d_i c_{i_j}^\alpha$) e T_{i_j} é o retardo associado à instalação desta mesma capacidade ($T_{i_j} = \frac{f_i}{c_{i_j} - f_i}$).

Partindo, então, desta configuração inicial para cada arco da rede em estudo, para se obter as soluções viáveis referente a dois arcos i e j , devemos fazer o produto cartesiano das tabelas referentes a estes dois arcos, isto é,

“ Se cada coluna destas tabelas for representada por uma estrutura de dados conveniente, então

Para cada coluna V da tabela do arco i faça

Para cada coluna W da tabela do arco j faça

Seja Z uma coluna nova;

$T_Z \leftarrow T_V + T_W$;

$D_Z \leftarrow D_V + D_W$;

$c_Z \leftarrow c_V \cup c_W$;

Concatene Z às outras colunas já obtidas anteriormente”

Este processo gera uma terceira tabela contendo todas as possíveis combinações de pontos para os arcos i e j simultaneamente. O mesmo ocorreria se aplicássemos o algoritmo acima para tabelas relacionadas a vários arcos. Evidentemente, para vários arcos ao mesmo tempo, a melhor forma de se guardar quais são as capacidades em c_Z é através de “ponteiros” para tabelas anteriores, evitando repetições de informações.

É claro que este processo pode gerar pontos que claramente não são viáveis, ou porquê violam as restrições impostas, ou então são pontos dominados (veja adiante mais detalhes). Estes pontos devem ser eliminados das tabelas, o que é fácil de ser feito, contando-se com uma estrutura de dados conveniente.

Para facilitar a busca da solução ótima final, podemos inicialmente manter cada tabela ordenada por ordem não decrescente de custo (isto é, em ordem não decrescente de capacidades) e executar todos os produtos cartesianos segundo o algoritmo acima e segundo a análise do Capítulo 3. Assim, ao final de um conjunto de operações de produtos cartesianos, teremos todas as possíveis combinações dos possíveis pontos de interesse do problema, bastando obter o primeiro deste conjunto (isto é, o de menor custo) viável para

ter-se a solução ótima de (P1), recuperando-se as informações associadas ao conjunto de canais designados.

Dado que, como visto no Capítulo 3, o custo computacional de um produto cartesiano é, no pior caso, bem aproximado por $mn \log_2 m$ para $m \leq n$ e $m = \#(\text{tabela } i)$ e $n = \#(\text{tabela } j)$, já sabemos como escolher otimamente a sequência das tabelas que farão parte dos produtos cartesianos (veja Capítulo 4), que serão as de maior tamanho, a cada vez.

Segue, então, um algoritmo para resolver o CAD (P1):

1. Para cada canal i da rede, construa a tabela Tab_i contendo todas as capacidades viáveis para este canal, em ordem não decrescente de custo;
2. Sejam Tab_i e Tab_j as duas tabelas de maior tamanho dentre todas. Faça o produto cartesiano entre elas, de acordo com o algoritmo descrito nesta seção e de acordo com o que foi visto no Capítulo 3;
3. Elimine todos os pontos da nova tabela gerada que não podem conduzir à uma solução ótima;
4. Repita os dois passos acima até que reste uma única tabela. Esta contém todos os pontos não dominados para (P1), e basta escolher a que convier (para minimizar custo, tome o primeiro elemento desta tabela).

As eliminações podem se dar de diversas formas. Uma delas é eliminar colunas das tabelas que sejam dominadas, no sentido de Pareto, onde, se houver duas colunas de uma tabela que são (D_r, T_r, c_r) e (D_s, T_s, c_s) e tais que $D_r \geq D_s$ e $T_r \geq T_s$, então (D_r, T_r, c_r) está dominada em relação à (D_s, T_s, c_s) , e a coluna (D_r, T_r, c_r) deve ser eliminada da tabela. Deve-se eliminar, também, colunas que não respeitem as restrições impostas ao problema, como $T_r > \gamma T_{max}$ para (P1), ou $D_r > D_{max}$ para (P2).

Uma outra forma de buscar limites para os pontos gerados pelo algoritmo e permitindo maior volume de eliminações de pontos que não nos conduzirão à solução ótima é a Relaxação Contínua: dada a rede, resolvemos o problema como se o conjunto das capacidades disponíveis fosse os números reais, isto é, $\mathcal{C} = \mathbb{R}$. É possível adaptar a solução contínua encontrada para obter uma solução discreta, procurando-se a capacidade discreta disponível para

cada canal mais próxima da obtida no problema contínuo, respeitando as restrições do problema. Assim, obtemos uma solução viável do problema completo, e qualquer ponto eventualmente gerado pelo algoritmo de busca das soluções não dominadas, cujo custo seja maior do que o custo calculado naquele ponto obtido do problema “relaxado”, deve ser eliminado (pois é pior que uma solução final já encontrada, mas que pode não ser a ótima).

5.2.1 Experiência Computacional

Aplicamos as estratégias acima descritas na resolução de problemas reais contendo dimensões pequenas ($m \in \{6, 11, 17, 22, 27, 33\}; \#C = 5$), onde m é o número de canais da rede (ou número de estágios do problema na solução por Programação Dinâmica). Estes problemas foram artificialmente replicados, gerando problemas maiores ($6 \leq m \leq 150; \#C \in \{5, 10\}$). Outras estratégias foram testadas, comparativamente à descrita como ótima, sendo que, no total, executamos 704 problemas de tamanhos m variáveis.

Para a validação dos nossos resultados, analisamos as performances das seguintes estratégias:

- (a) estratégia gulosa nos sort-merges e a política sequencial ótima para a escolha das tabelas (esta é a melhor combinação, segundo o que provamos no capítulo 4);
- (b) estratégia sequencial para os sort-merges, e a política sequencial ótima para a escolha das tabelas;
- (c) estratégia gulosa para os sort-merges e estratégia gulosa para a escolha das tabelas.

No apêndice A, apresentamos uma implementação computacional de cada algoritmo segundo cada estratégia (a), (b) e (c).

Alguns resultados comparativos são resumidos no quadro 1, onde mostramos para $\#C = 5$ as diferenças entre as três estratégias acima e no quadro 2, onde executamos a estratégia (a) para $\#C = 5$ e 10.

#C = 5	estrat. (a)			estrat. (b)			estrat. (c)		
<i>m</i>	<i>media</i>	<i>max</i>	<i>min</i>	<i>media</i>	<i>max</i>	<i>min</i>	<i>media</i>	<i>max</i>	<i>min</i>
11	92	156	14	108	186	14	195	332	37
22	366	576	104	441	739	108	665	972	228
33	659	958	280	842	1256	296	1417	2010	421
44	1075	1618	417	1344	2031	440	2056	3162	640
50	1355	2047	439	1754	2603	460	2725	5085	661
100	2104	2730	1286	2440	3812	1384	*	*	*
150	3612	4690	2993	3819	4796	3110	*	*	*

Quadro 1: numero de comparações das estratégias (a) (b) (c).
(* = não computado)

	#C = 5			#C = 10		
<i>m</i>	<i>media</i>	<i>max</i>	<i>min</i>	<i>media</i>	<i>max</i>	<i>min</i>
6	15	42	3	366	618	234
11	36	71	12	1050	1328	863
17	102	189	40	2055	2460	1472
22	191	347	67	2931	3449	2487
27	254	429	103	3843	4535	3113

Quadro 2: estratégia (a) variando #C.

Também executamos o algoritmo de Programação Dinâmica para a obtenção de todos os pontos eficientes (identificado abaixo como EF) e realizando as eliminações segundo os “bounds” obtidos (OPT), como podemos comparar segundo o quadro 3. Neste quadro, mostramos a relação entre o número real de comparações (custo computacional) obtido e o número de comparações esperado na análise de pior caso. Note que o custo de pior caso *não* é o mesmo para as 3 estratégias acima descritas (fazer um sort-merge sequencial entre n tabelas gasta $n - 1$ iterações, e fazer o produto cartesiano total entre n tabelas segundo uma estratégia gulosa gasta, no pior caso, $n - 1$ iterações).

	(OPT)								
	estrat. (a)			estrat. (b)			estrat. (c)		
m	media	max	min	media	max	min	media	max	min
6	7.02	11.11	2.69	3.81	5.66	2.16	13.08	27.77	4.34
11	3.35	9.88	0.29	1.29	3.73	1.7E-1	4.84	14.59	2.8E-1
16	8.7E-1	3.54	7.7E-3	2.6E-1	1.01	3.8E-3	6.5E-1	2.44	6.8E-3
22	5.3E-2	2.7E-1	7.8E-5	1.4E-2	6.9E-2	2.9E-4	6.1E-2	3.4E-1	1.3E-4
27	1.1E-3	4.6E-3	6.0E-7	3.2E-4	1.3E-3	2.2E-7	1.1E-3	4.2E-3	3.3E-7
	(EF)								
	estrat. (a)			estrat. (b)			estrat. (c)		
m	media	max	min	media	max	min	media	max	min
6	13.66	22.22	4.10	7.36	11.95	3.37	26.56	50.0	7.75
11	5.63	13.53	3.5E-1	2.10	4.98	2.2E-1	9.98	30.19	4.2E-1
16	1.12	4.74	7.8E-3	3.4E-1	1.34	3.8E-3	1.47	6.49	7.8E-3
22	7.8E-2	4.2E-1	8.2E-5	1.9E-2	1.0E-1	3.6E-5	1.0E-1	5.6E-1	1.7E-4
27	1.6E-3	6.6E-3	6.2E-7	4.6E-4	1.9E-3	2.5E-7	1.6E-3	6.1E-3	3.9E-7

Quadro 3: Relação Comparações/Estimativa (*100).

Como, na abordagem de Programação Dinâmica, o tamanho da tabela final deve sempre ser o mesmo, notamos que uma estratégia não ótima (b) executa mais eliminações do que o esperado, ou pelo menos, tende a se portar melhor quando o problema cresce em sua dimensão. De qualquer forma, a estratégia (c) mostra-se menos eficientes que as demais, sob qualquer aspecto.

Apresentamos, nas páginas seguintes, alguns gráficos representando os testes computacionais realizados:

No gráfico 1 temos um histograma representando as relações entre as três estratégias descritas, obtido através das médias do número de comparações que cada algoritmo gastou, para o caso $\#C = 5$ e nele podemos notar a melhor eficiência da estratégia (a), como esperaríamos.

No gráfico 2 temos a mesma análise entre as três estratégias para $\#C = 5$, mas aqui apresentamos além das médias, também o desvio do número de comparações gasto por cada um (número mínimo e máximo de todos os testes), descrevendo nas “barras” a variação de cada um. Cumpre esclarecer que em cada teste realizado, a estratégia (a) *sempre* se mostrou melhor que (b), que por sua vez, era melhor que (c).

No gráfico 3, apresentamos uma variãda média do número de comparações ao executarmos a estratégia (a) com $\#C = 5$ e $\#C = 10$.

No gráfico 4, mostramos as relações obtidas através das médias dos número de comparações quando executamos a estratégia (a) nos casos $\#C = 5$ e $\#C = 10$, para obter pontos da forma (EF) e (OPT) descritos anteriormente. Claramente notamos que calcular todos os pontos eficientes (EF) gasta-se mais do que se utilizarmos “bounds” para eliminações (OPT), e ao dobrarmos a variação de $\#C$, mantemos as proporções, mas pioramos o rendimento do algoritmo, concordando com a curva do gráfico 3.

Gráfico 1: histograma relativo às estratégias (a), (b) e (c), para $\#C = 5$.

Gráfico 2: número de comparações das estratégias (a), (b) e (c), para $\#C = 5$.

Gráfico 3: número de comparações da estratégia (a) para $\#C = 5$ e $\#C = 10$.

Gráfico 4: número de comparações da estratégia (a) para $\#C = 5$ e $\#C = 10$ na obtenção de (EF) e (OPT).

5.3 Resolução do CAD através do A^*

Como vimos no Capítulo 4, a caracterização da busca de pontos eficientes em um problema multicritério separável pode ser encarada como o problema de obter o caminho mais curto em um grafo. Diversos algoritmos são tratados na literatura em geral sobre este problema, sendo talvez o mais conhecido, o algoritmo de Dijkstra (veja Aho, Hopcroft e Ulmann [1]), que é essencialmente um algoritmo de Programação Dinâmica.

No entanto, nosso grafo possui uma característica indesejável para qualquer algoritmo em geral: pela característica do problema CAD, e pela forma como é caracterizado em um grafo, este grafo possui um número exponencial de arestas e nós, crescendo em tamanho exponencialmente à medida em que é construído por níveis, dificultando por completo a sua representação explícita. Devemos, assim, trabalhar com este grafo implicitamente, como fizemos na seção anterior, ou somente nos preocuparmos com nós e caminhos no grafo que possam nos conduzir à solução ótima. Para isso, utilizamos algoritmos informados, como o A^* , que, segundo Gonzaga [10], portam-se melhor do que algoritmos que eventualmente trabalhem explicitamente com o grafo obtido.

O algoritmo A^* busca tirar mais informações a respeito do grafo do que as obtidas explicitamente nos nós e arestas. Cada nó do grafo que o A^* manterá deve conter, além das informações referentes ao problema, também um apontador indicando de qual nó ele veio (ou seja, qual estado do problema gerou este novo estado) e uma estimativa de quão promissor é este estado. Na verdade, é necessário que haja um subestimador consistente da distância entre qualquer nó $v \in VG$ ao nó $t \in VG$, final. Por subestimador entendemos como sendo uma função $h : VG \rightarrow \Re$ que calculada em qualquer nó $v \in VG$, retorna um valor menor ou igual ao custo do menor caminho de v a t . Um subestimador $h()$ é dito consistente se $h(x) \leq k(x, y) + h(y)$ para todo $x, y \in NG$, onde $k(x, y)$ = o menor custo dentre todos os caminhos de x a y (veja Pearl [18]). Assim, um subestimador consistente $h()$ porta-se semelhantemente a funções que representem distâncias em grafos e desta forma temos uma idéia de quanto custará ir de v a t , no mínimo, e vamos preferir percorrer caminhos que promissoramente custem menos. Gonzaga ([10] pág 59, teor. 15) mostra que com o uso de subestimadores consistentes, no A^* serão fechados todos os trechos de caminhos que admitem a possibilidade de serem ótimos, ou seja, só abandonamos caminhos que não podem ser ótimos.

Serão necessários duas listas na execução do A^* :

- ABERTOS - nós que foram gerados e ainda não examinados, sendo que esta lista é implementada como uma fila de prioridades, onde os nós mais promissores deverão estar no topo;
- FECHADOS - nós que já foram examinados.

Vamos trabalhar, em termos de prioridade para a lista de ABERTOS, com a soma de um subestimador com o custo acumulado do nó, isto é, se $h()$ é um subestimador para o nosso problema e $g()$ representa o custo para atingirmos um certo nó (ou estado do problema), então a função $f() = g() + h()$ irá controlar os nós nas listas do nosso algoritmo.

Assim, o algoritmo A^* pode ser descrito da seguinte forma geral (veja Rich [19]):

1. Coloque em ABERTOS o nó inicial s (estado inicial do problema) com $f(s) = h(s)$;
2. Enquanto não atingimos um nó final t , a cada passo retire w o primeiro elemento da lista de ABERTOS.
3. Se $w \neq t$, então para todo x um sucessor de w , faça:

se x não está na fila de ABERTOS ou FECHADOS calcule $g(x) = g(w) + \text{custo de ir de } w \text{ a } x$, faça x apontar para w e coloque x em ABERTOS;

se x está em ABERTOS ou FECHADOS, $g(x)$ recebe o valor $\min\{g(x), g(w) + \text{custo de ir de } w \text{ a } x\}$ e x aponta para o nó que resulte no menor custo $g(x)$;

calcule $f(x) = g(x) + h(x)$.

Coloque w na fila FECHADOS.
4. Se $w = t$, então obtivemos a solução ótima, bastando recuperar as informações referentes às designações voltando pelo caminho percorrido no grafo através dos ponteiros dos nós.

Este algoritmo apenas “expande” as configurações mais promissoras do ponto de vista do custo final, configurações estas que podem fazer parte da solução ótima, evitando todas aquelas que não poderão ser ótimas.

No caso da resolução do CAD, o nó inicial s representa o estado de nenhuma designação feita a nenhum canal, e seus sucessores são todos os nós que estão associados a designação de apenas um canal cada. Estes nós

contém as mesmas informações descritas no início da seção anterior, onde caracterizamos as tabelas iniciais para cada arco da rede. Os sucessores de um nó w em geral são todos os nós obtidos a partir da configuração das designações em w , combinadas com todas as possíveis configurações designando-se capacidades a mais um canal.

Ainda com relação ao CAD, facilmente obtemos um subestimador h da seguinte forma: dado um nó $v \in VG$, este nó representa um conjunto de capacidades atribuídas a k canais da rede. A partir destas capacidades designadas aos seus canais, resolve-se o problema de designação contínua para os canais restantes da rede. Este custo calculado é um subestimador consistente com o problema, pois cada capacidade obtida do problema contínuo é menor ou igual ao possível valor inteiro do conjunto de capacidades discretas, e este comporta-se como uma distância num grafo

Assim, temos que $f(w)$ será a soma entre o custo do problema “relaxado” contínuo para os canais ainda não verificados a partir do estado associado a w ($h(w)$) e o custo associado às capacidades já determinadas até este estado, calculado segundo as fórmulas para este custo ($\sum_{i \in w} d_i c_i^\alpha$) ($g(w)$).

Se estamos considerando $s \in VG$ como sendo o nó inicial do grafo (isto é, nenhum canal designado) e $t \in VG$ o nó final (isto é, todos os canais designados), temos $h(s)$ = valor ótimo do problema contínuo para todos os arcos; e $h(t)$ = valor ótimo do problema discreto para todos os arcos.

5.3.1 Experiência Computacional

Utilizando parte dos mesmos dados dos testes para resolução do CAD por Programação Dinâmica, executamos uma versão do algoritmo A^* com duas variações. Em uma delas, ao atingirmos o penúltimo nível do grafo, fazíamos um “look-ahead”, isto é, resolvíamos por completo o problema a partir do estado neste nível, obtendo uma solução viável. Na outra versão, executamos o A^* sem realizarmos um “look-ahead”, ou seja, sem nenhum tipo de melhoria computacional na tentativa de tornar o algoritmo implementado mais eficiente.

Executamos ao todo 75 testes com $\#C = 5$, para $m \in \{11, 22\}$. Os testes foram de pequena dimensão pois o algoritmo exige muita memória para sua execução, e geralmente não havia recursos suficientes¹. Segue um quadro comparativo entre as versões com e sem “look-ahead”, onde notamos o melhor desempenho da versão com “look-ahead”.

¹os teste foram executados em máquinas do tipo PC-AT com 640 KB de memória.

#C = 5	<i>com look-ahead</i>			<i>sem look-ahead</i>		
<i>m</i>	<i>media</i>	<i>max</i>	<i>min</i>	<i>media</i>	<i>max</i>	<i>min</i>
11	97.06	1187	2	103.38	1217	2
22	7749.00	27644	525	7780.64	27644	525

Quadro 4: número de comparações das versões do A^* .

Mostramos no quadro seguinte, a relação #nós eliminados / #nós gerados. Isto torna claro a importância de “bounds” no auxílio da resolução do problema CA utilizando o A^* , pois nota-se a alta porcentagem do número de nós que nem sequer entraram na lista de ABERTOS, diminuindo o esforço computacional total.

#C = 5	<i>com look-ahead</i>		
<i>m</i>	<i>media</i>	<i>max</i>	<i>min</i>
11	0.677542	0.833333	0.514286
22	0.577709	0.691120	0.476510

Quadro 5: % de eliminações.

No apêndice B, apresentamos uma implementação computacional do A^* com “look-ahead”, versão esta que se mostrou mais eficiente nos nossos testes.

✻

Apêndice A

Implementações do CAD por Programação Dinâmica

A.1 - Implementação com a estratégia gulosa nos sort-merges e a política sequencial ótima para a escolha das tabelas.

```
(*1*****  
(*1          DESIGNACAO de CAPACIDADES DISCRETAS          *)  
(*1          *)  
(*1          Recebe uma rede com topologia , roteamento e fluxos nos          *)  
(*1 canais definidos, e devolve uma capacidade para cada canal, com          *)  
(*1 custo minimo possivel, sujeita a um conjunto de capacidades com          *)  
(*1 valores discretos disponiveis.          *)  
(*1          *)  
(*1*****  
(*1  DEFINICOES :          *)  
(*1      Combinacao - juncao de dois canais com capacidades dis-          *)  
(*1                      cretas viaveis.          *)  
(*1          *)  
(*1          *)  
(*1      Possivel Combinacao - Combinacoes de capacidades discretas          *)  
(*1                      viaveis fazendo eliminacao de inefi-          *)  
(*1                      cientes atraves de bounds.          *)  
(*1          *)  
(*1      No - Guarda uma possivel combinacao com custo e retardo          *)
```

```

(*1          associado                                     *)
(*1                                                  *)
(*1      Tabela - Conjunto de "nos" para um determinado conjunto *)
(*1          de canais                                     *)
(*1                                                  *)
(*1                                                  *)
(*1      Alvo - Tabela resultado da juncao de duas tabelas *)
(*1                                                  *)
(*1                                                  *)
(*1*****
(*1      TABELA e um vetor de tabelas, uma para canal, e no decorrer *)
(*1 do programa, todos os elementos em uma tabela estaraõ ordenados *)
(*1 por ordem crescente de retardo, logo por ordem decrescente de *)
(*1 custos, e terao como componentes conjuntos de canais, cada conj. *)
(*1 representando uma capacidade disponivel, onde se determinara *)
(*1 qual capacidade sera atribuida a cada canal. *)
(*1*****

(*****
****
*      TABELAS ORDENADAS EM ORDEM CRESCENTE DE CUSTO      *
*      ESTRATEGIA : PROGDIN em TABELAS e DeC EM SUBTABELAS  *
****
*****
)
program CA_discreto;
uses crt,dos;
const MAX      = 50;
      MAX_CAPS = 5;
      EPSILON  = 10E-5;
type
  apont = ^no;
  conj  = set of 1..max;
  vetconj = array [1..MAX_CAPS] of conj;
  No = record
    ret ,
    custo : real;
    caps : vetconj;
    prox : apont;
  end;

```

```

Item_tabela = record
    link: apont;
    tam : integer;
end;
Item_aprox = record
    capac, ret : real;
    arco : integer;
end;

var
    est : real;
    capacidade : array[1..MAX_CAPS] of integer; { vetor de possiveis
                                                    capacidades }

    GTmax, { gama * Tmax }
    alfa, custo_bound,
    gama,T_max : real;
    Sol, { solucao }
    No_livre : apont; { cabeca da lista livre }
    i,j, { contadores }
    N_canais : integer; { n de canais }
    somas, { n de somas }
    comps : longint; { n de comparacoes }
    tabela : array[1..MAX] of Item_tabela;
    entrada : text; { arquivo de entrada }
    Arq_Ent : string[12]; { string para leitura }
    h,min,s,ds : word; { conta tempo }
    aux,d,dl,f : array [1..MAX] of real;
    vet_sol : array [1..max] of Item_aprox;
    aprox_ini : boolean;

(*2*****)
(*2 Proc. Sol_Inicial : *)
(*2 Proc. que da um chute inicial para CA suj. a Retardo *)
(*2 repartindo os retardos proporcionalmente para cada canal, e ob- *)
(* tendo capacidades iniciais continuas. *)
(*2 E chamado pelo proc. RESOLVE. *)

```



```

procedure Sol_Inicial;
var soma:real;
begin
  soma:=0;
  for i:=1 to N_canais do
    { distribui Gama*Tmax por
      cada capacidade }

    soma:=soma+sqrt(d[i]*f[i]);
  for i := 1 to N_canais do begin
    aux[i] := alfa*d[i];
    { 'aux' calculado no inicio
      e so uma vez }

    vet_sol[i].capac := f[i]*( 1+1/(GTmax*sqrt(d[i]*f[i])/soma) );
    vet_sol[i].arco := i;
  end;
end;

(*2*****
(*2          Proc.      Lineariza :
(*2          Proc. que devolve custos linearizados
(*2  Calcula a derivada da formula fechada para custos, obtendo cus-
(*2 tos linearizados para resolucao do problema, com capacidades
(*2 continuas.
(*2          E chamado pelo proc.  RESOLVE.

procedure Lineariza;
begin
  { obtidos pela derivada dos valores }
  for i := 1 to N_canais do
    { iniciais do custo }
    dl[i] := aux[i] * exp ( (alfa-1)*ln( vet_sol[i].capac ) )
end;

(*2*****
(*2          Proc.      Resolve :
(*2          Proc. que resolve CA sujeito a Retardo
(*2  Calcula capac. otima do problema linearizado usando a formula
(*2 fechada, e reaplica esta solucao iterativamente no problema nao
(*2 linear. A sol. final e de capac. continuas.

```

```

(*2          E chamado pelo proc. SOL_DISCRETA_VIAVEL.          *)

procedure Resolve;
  var va,aux,beta : real;
begin
  j:=0;
  custo_bound := 0;
  Sol_Inicial;      { pega um chute inicial }
  repeat
    j := j+1;          { conta iteracoes }
    va := custo_bound; { va = valor anterior do custo }
    custo_bound := 0;
    Lineariza;         { obtem custos linearizados }
    aux := 0;
    for i := 1 to n_canaais do
      aux := aux + sqrt( f[i]*dl[i] ); { calculos... }
    aux := aux/T_Max ;
    beta := aux*aux/gama;
    aux := aux/gama;
    for i := 1 to n_canaais do begin
      vet_sol[i].capac := f[i] + aux*sqrt(f[i]/dl[i]);
      vet_sol[i].ret := f[i] / ( vet_sol[i].capac - f[i]);
      custo_bound:=custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
    end;

    until ( abs(va - custo_bound)/custo_bound ) <= EPSILON; {diferenca
                                                                relativa }
  end;
                                                                {como condicao
                                                                de parada }

(*2*****
(*2          Proc. SOL_DISCRETA_VIAVEL - resolve o problema para *)
(*2  capacidades continuas e depois obtem uma solucao viavel discre-*)
(*2  ta a partir desta.                                          *)
(*2          E chamado pelo proc. MONTA_TABELA                  *)

procedure Sol_discreta_viaavel;

```

```

var GT, retar : real;
    i,j,k : integer;
    sobe : boolean;
    desce : array [ 1 .. MAX ] of integer;
begin
    Resolve;
    k := 0;
    for i:=1 to N_canaís do
        if capacidade[MAX_CAPS] < vet_sol[i].capac then begin
            k := k+1;
            desce[k] := i;
        end;
    GT := GTmax;
    if k < MAX then begin
        for i := 1 to k do begin
            vet_sol[desce[i]].capac := Capacidade[MAX_CAPS];
            GT := GT - vet_sol[desce[i]].ret;
            Vet_sol[desce[i]].ret:= f[desce[i]] / (capacidade[MAX_CAPS]
- f[desce[i]]);
            GT := GT + vet_sol[desce[i]].ret;
        end;

        aprox_ini := true;

        while aprox_ini and ( (GT > GTMax) or (k=0) ) do begin
            for i:=1 to N_canaís do
                if vet_sol[i].capac < capacidade[MAX_CAPS] then begin
                    j := 1;
                    while capacidade[j] <= Vet_sol[i].capac do j:=j+1;
                    retar := f[i] / ( capacidade[j-1] - f[i]);
                    GT := GT - vet_sol[i].ret;
                    if (capacidade[j-1] > f[i]) and (GTmax-GT-retar>=0)
                        then
                            j:=j-1;
                    retar := f[i] / (capacidade[j] - f[i]);
                    vet_sol[i].capac := capacidade[j];
                    GT := GT + retar;
                    vet_sol[i].ret := retar;
                end;
            end;
        end;
    end;
end;

```

```

        end;
        j:=1; k:=1;
        while (j <= N_canaais ) and (aprox_ini) do begin
            aprox_ini:=capacidade[MAX_CAPS]=vet_sol[j].capac;
            j:=j+1;
        end;
        aprox_ini := j < N_canaais
    end;

    custo_bound := 0;
    for i:=1 to N_canaais do
        custo_bound:=custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
        aprox_ini := GT <= GTMax;
    end

    else aprox_ini := false;
end;

(*2*****
(*2***  Proc.  ALOCA_NO  -  aloca um no para tabela de canais  ***)
(*2*   Mantem  uma  lista de celulas livres para alocao  **)

procedure aloca_no(var P : apont);
var j:integer;
begin
    if No_livre = NIL then
        NEW(p)
    else begin
        P:=No_Livre;
        No_Livre := No_Livre^.prox;
    end;
    with p^ do begin
        ret := 0; custo := 0;
        for j:=1 to MAX_CAPS do caps[j] := [];
        prox := NIL;
    end;
end;
end;

```

```

(*2*****
(*2**      Proc.  DEVOLVE_NO - deixa um no que nao e mais usado  **)
(*2* livre para posterior utilizacao, usando lista livre.          **)

```

```

procedure Devolve_No(var P : apont);
var j:integer;
begin
    P^.prox := No_Livre;
    No_livre := P;
end;

```

```

(*2*****
(*2**      Proc.  LIBERA - Libera todo o espaco ocupado por uma  **)
(*2* tabela de canais, devolvendo todos os nos por ela utilizados **)
(*2* para lista livre.                                           **)

```

```

procedure Libera( num : integer);
var p:apont;
begin
    while tabela[num].link <> NIL do begin
        p:=tabela[num].link;
        tabela[num].link := tabela[num].link^.prox;
        Devolve_No ( p );
    end;
end;

```

```

(*2*****
(*2**      Proc.  EMPURRA - Se alguma tabela no vetor de tabelas **)
(*2* e liberado, este proc. se encarrega de deixar todas as tabelas *)
(*2* restantes adjacentes umas as outras.                          **)

```

```

procedure Empurra(n:integer);
var i:integer;
begin
    for i:=n+1 to max do
        tabela[i-1]:=tabela[i];
    end;

```



```

(*2*****
(*2**      Func.  ELIMINA  -  Determina se dado no'  gerado      **)
(*2*  no MERGE deve ou nao ser descartado, verificando-se Bounds de **)
(*2*  programacao.                                          **)
(*2*      E chamado pelo proc. MERGE.                      **)

```

```

function Elimina(t,c,GamaT:real):boolean;
begin
  Elimina := (t > GamaT) or (c > custo_bound);
end;

```

```

(*2*****
(*2**      Proc.  ELIMINA_INEFICIENTES  -  Dada uma tabela de  **)
(*2*  canais, retira todos os "ineficientes" desta tabela, utili- **)
(*2*  zando a estrategia de Bound 3 (v/ lista Bounds ).          **)
(*2*      E chamado pelo proc. MERGE.                            **)

```

```

procedure Elimina_Ineficientes ( n:integer );
var p,q : apont;
begin
  p := tabela[n].link;
  q:=p;
  if p <> NIL then
    p:=p^.prox;
  while p <> NIL do begin
    if (p^.ret > q^.ret) then begin
      q^.prox := p^.prox;
      Devolve_No ( p );
      tabela[n].tam := tabela[n].tam -1;
    end
    else
      q:=q^.prox;
      p:=q^.prox;
    end;
  end;
end;

```

```

(*2*****
(*2**      Proc.      DeC      -      Toma duas tabelas de canais e mis- **)
(*2* tura-as,gerando uma so tabela ,que e devolvida na primeira **)
(*2* delas ,eliminando-se ineficientes e memoria nao utilizada . **)
(*2* O proc DeC monta todas as sub-tabelas iniciais resultantes da **)
(*2* soma dos elementos de duas tabelas, e as envia para o proc. **)
(*2* MERGE, para juntar todas as subtabelas em uma unica,utilizan- **)
(*2* do a estrategia Divide & Conquer. **)
(*2*****
***
*      Lista de Bounds utilizados :
*      BOUND 1 : dois "nos" com retardos iguais , elimina-se
*                  o de maior custo.
*
*      BOUND 2 : o retardo de um novo "no" criado deve ser
*                  menor que o retardo maximo menos o minimo
*                  dos retardos das tabelas anteriormente
*                  criadas .
*
*      BOUND 3 : se existir dois "nos" , em que retardo e
*                  custo de um sao menores que retardo e
*                  custo do outro , o outro deve ser elimi-
*                  nado.
*
***
*****
(*2*      E chamado pelo Programa Principal , e chama como auxili- **)
(*2* ares os procs. Elimina, Elimina_Ineficientes, Libera, Empurra, **)
(*2* Aloca_No . **)

```

procedure DeC;

var

```

    alvo,
    p, auxp, auxq,          { p aloca nova cel, auxp e auxq percorrem}
    ultcomp,proxcomp,sub_ptr :apont; {tabelas menor e maior respect./}
    i, j,
    p1,p2,
    m,n,                    { tamanho das tabelas }

```

```

tam      : integer;
aux,aux2 : real;
Sub_Tab  : array [ 1 .. MAX_CAPS ] of Item_Tabela;

(*****
***
*   PROCEDURE MERGE : recebe duas sub-tabelas e faz um sort   *
*   merge sobre elas , destaca-se neste procedimento o fato  *
*   de por ser melhor manter as tabelas por ordem crescente de *
*   custo e nao de retardo , os elementos das sub-tabelas sao *
*   comparados por seus custos , afim de manter a eficiencia  *
*   do procedimento ELIMINA INEFICIENTES.                      *
***
*****)
procedure merge ( prim , seg : integer );
  var j : integer;

begin
  Sub_ptr := Sub_Tab[prim].link;
  auxq    := Sub_ptr^.prox;
  auxp    := Sub_Tab[seg].link;
  while ( auxp <> nil ) do begin

    (* PROCURA A POSICAO PARA INSERCAO *)
    while (auxq^.custo < auxp^.custo) and ( auxq <> nil ) do begin
      COMPS := COMPS + 1;
      Sub_Ptr := Sub_Ptr^.prox;
      auxq    := auxq^.prox;
    end;

    (* Terminou o sort merge apenas une o restante das tabelas *)
    if auxq = nil then begin
      Sub_Ptr^.prox := auxp ;
      auxp := nil;
    end
    else begin
      p := auxp^.prox;
    end

    (* custos iguais , permanece o de menor retardo *)

```

```

        if auxq^.custo = auxp^.custo then
            if auxq^.ret > auxp^.ret then begin

                auxq^.ret := auxp^.ret;
                for j := 1 to MAX_CAPS do
                    auxq^.caps[j] := auxp^.caps[j]
                end
            else

                (* Insere novo elemento na posicao *)
                else begin
                    auxp^.prox := auxq;
                    Sub_Ptr^.prox := auxp;
                end;
                auxp := p;
            end
        end;
    end;
end;

(*2***** Procedimento DeC ******)
begin
    tam:=0;
    { Coloca em tabela de p1 a tabela de menor tamanho }
    p1:=1; p2:=2;
    if tabela[2].tam < tabela[1].tam then begin
        n := p1;
        p1:= p2;
        p2:= n;
    end;
    n:=tabela[p1].tam;
    m:=tabela[p2].tam;

    auxp:=tabela[p1].link;
    auxq:=tabela[p2].link;

    for i := 1 to MAX_CAPS do begin
        Sub_Tab[i].link:= nil;
        Sub_Tab[i].Tam := 0;
    end;
end;

```

```

i := 1;
while ( auxp <> Nil ) do begin      { monta as subtabelas }
  while auxq <> NIL do begin
    aux:=auxp^.ret + auxq^.ret;
    aux2 := auxp^.custo + auxq^.custo;
    somas := somas+1;
    if Sub_Tab[i].link <> NIL then begin
      if not Elimina(Aux,Aux2,GTmax) then begin
        ALOCA_NO(p);
        Sub_Tab[i].tam:=Sub_Tab[i].tam+1;
        p^.prox := nil;
        p^.ret := aux;
        p^.custo := aux2;
        for j:=1 to MAX_CAPS do
          p^.caps[j] := auxp^.caps[j]
            + auxq^.caps[j];
        sub_ptr^.prox:=p;
        sub_ptr := p;
      end
    else
      end
    else if not Elimina( aux, aux2, GTmax) then begin
      ALOCA_NO(Sub_Tab[i].link);
      Sub_Tab[i].tam := 1;
      Sub_Ptr := Sub_Tab[i].link;
      Sub_Tab[i].link^.prox := NIL;
      Sub_Tab[i].link^.ret := aux;
      Sub_Tab[i].link^.custo := aux2;
      for j:=1 to MAX_CAPS do
        Sub_Tab[i].link^.caps[j] := auxp^.caps[j]
          + auxq^.caps[j];
      end;
      auxq:=auxq^.prox;
    end; { while interno }
    i := i+1;
    auxp:=auxp^.prox;
    auxq:=tabela[p2].link;
  end;
end;

```



```

end; { while externo }
while Sub_Tab[2].link <> NIL do begin { Mergeia as subtabelas }
  i := 1; { pelo MERGE - Div.& Conq. }
  while Sub_Tab [i+1].link <> nil do begin
    MERGE ( i , i+1 );
    for j := i+2 to MAX_CAPS do
      Sub_Tab [j-1] := Sub_Tab [j];
    Sub_Tab [MAX_CAPS].tam := 0;
    Sub_Tab [MAX_CAPS].link := NIL;
    i := i + 1;
  end;
end;
p:=sub_tab[1].link;
sub_tab[1].tam:=0;
while p<>NIL do begin
  p:=p^.prox;
  sub_tab[1].tam := sub_tab[1].tam + 1;
end;
Libera(1); { cuidados finais }
Libera(2);
tabela[1].link:= Sub_Tab[1].link;
tabela[1].tam := Sub_tab[1].tam;
Empurra(2);
Elimina_Ineficientes(1);
end;

```

```

(*2*****
(*2**      Proc.      SORT - Ordena as tabelas de canais por ordem **)
(*2*  decrescente de tamanho, facilitando posterior MERGE .      **)
(*2*      Versao recursiva do Quick-Sort.                          **)
(*2*      E chamado pelo Monta_Tabela.                            **)

```

```

procedure SORT(l,r : integer);
var i,j : integer;
    x,w : Item_tabela;
begin
  i := l; j := r;
  x := tabela[ (l+r)div 2 ];

```

```

repeat
  while tabela[i].tam > x.tam do
    i := i+1;
  while tabela[j].tam < x.tam do
    j := j-1;
  if i <= j then begin
    w := tabela[i];
    tabela[i] := tabela[j];
    tabela[j] := w;
    i := i+1;
    j := j-1;
  end;
until i > j;
if l < j then
  SORT (l,j);
if i < r then
  SORT (i,r);
end;

procedure estimativa;
var i : integer;
    aux : real;
begin
  est:=0; aux:=tabela[1].tam;
  for i:=2 to n_canaais do begin
    aux:=aux*tabela[i].tam;
    est:=est + aux * ln(tabela[i].tam)/ln(2);
  end;
end;

(*2*****
(*2**      Func.      MONTA_TABELA - Le a rede do arquivo de entrada **)
(*2*  e monta as tabelas de canais, ordena a tabela, verifica via- **)
(*2*  bilidade inicial do problema, e reduz o n de tabelas, descon-**)
(*2*  siderando as que so tem uma capacidade possivel. **)
(*2*      E chamado pelo Programa Principal. **)

function Monta_tabela : boolean;

```

```

var i,j,k :integer;
    p : apont;
    fluxo,cust : real;

begin
    clrscr;
    writeln;writeln;
    writeln('          Designador de Capacidades - Versao Discreta');
    writeln;writeln;
    write(' Arquivo entrada : ');          { leitura }
    readln(arq_ent);
    writeln;
    arq_ent := arq_ent+'.dat';
    assign(entrada,arq_ent);
    {$i-}
    reset(entrada);
    {$i+}
    if IOresult <> 0 then begin
        writeln (' Erro, ',arq_ent,' nao esta no drive ');
        delay(2000);
        halt
    end;

    SETTIME(0,0,0,0);
    {$i-}
    readln(entrada,Gama,T_max,alfa);
    {$i+}
    if IOresult <> 0 then begin
        writeln ('Erro, Incoerencia no formato do arquivo na Linha 1');
        delay(2000);
        halt
    end;

    GTmax := gama * T_max;
    {$i-}
    readln(entrada,N_canaais);
    {$i+}
    if IOresult <> 0 then begin
        writeln ('Erro, Incoerencia no formato do arquivo na Linha 2');

```

```

        delay(2000);
    halt
end;

for i:=1 to N_canais do begin
    {$i-}
    readln(entrada,fluxo,cust);
    {$i+}
    if IOresult <> 0 then begin
        writeln (' Erro, Incoerencia no formato do arquivo na'
                , 'Linha ',i+2);
        delay(2000);
        halt
    end;
    if Fluxo > Capacidade[MAX_CAPS] then begin
        writeln ('Erro,Nao existe capacidade disponivel para fluxo'
                , 'definido na Linha ',i+2,'do arquivo');
        delay(2000);
        halt
    end;

    f[i] := fluxo; d[i] := cust;
    for j:=MAX_CAPS downto 1 do      { monta tabela para canal i }
        if Capacidade[j] > fluxo then begin
            tabela[i].tam := tabela[i].tam + 1;
            ALOCA_NO(p);
            with p^ do begin
                ret := fluxo/(capacidade[j] - fluxo);
                custo := cust * exp(alfa * ln(capacidade[j])) ;
                caps[j] := [i];
                prox := tabela[i].link;
            end;
            tabela[i].link := p;
        end;
    end;

end;

SOL_DISCRETA_VIAVEL;

cust := 0;

```

```

SORT(1,N_canaais);                                { ordena }
if tabela[N_Canaais].tam = 0 then
    monta_tabela := false
else begin

    ESTIMATIVA;

    while (tabela[n_canaais].tam = 1) and (N_canaais > 0) do begin
        GTmax := GTmax - tabela[n_canaais].link^.ret;
        cust  := cust + tabela[n_canaais].link^.custo;
        n_canaais := n_canaais -1;
    end;

    if aprox_ini then CUSTO_BOUND := CUSTO_BOUND - cust;

    Monta_Tabela := (GTMax > 0) and (aprox_ini);
end;
close(entrada);
end;

(*2*****
(*2**   Proc.   INICIA - Inicializa variaveis do programa.   **)
(*2*     E chamado pelo Programa Principal.                  **)

procedure INICIA;
var i:integer;
begin
    No_Livre := NIL;
    capacidade[1] := 300;
    capacidade[2] := 1200;
    capacidade[3] := 2400;
    capacidade[4] := 4800;
    capacidade[5] := 9600;
    for i:=1 to MAX do begin
        tabela[i].tam:=0;
        tabela[i].link:=NIL;
    end;
    somas :=0; comps :=0;

```



```

    aprox_ini := false;
    custo_bound := 1E38;
end;

(*2*****
(*2*      Proc.  IMPRIME_SOL - Imprime a solucao otima do problema **)
(*2*      E chamado pelo proc. Sol_Otima .                               **)

procedure Imprime_sol;
begin
    for i:=1 to N_canais do
        for j:=1 to MAX_Caps do
            if i IN sol^.caps[j] then
                writeln('canal ',i,' com cap. = ',capacidade[j]);
            writeln;
            writeln(' custo = ',sol^.custo);
            writeln(' ret   = ',sol^.ret);
            writeln(' somas = ',somas);
            writeln(' comps = ',comps);
        end;
    end;

(*2*****
(*2*      Proc.  IMPRIME_SOL_2 - Imprime a solucao otima do problema **)
(*2*      E chamado pelo proc. Sol_Otima .                               **)

procedure Imprime_sol_2;
var i:integer;
begin
    for i:=1 to n_canais do
        writeln('canal ',i,' com cap. = ',vet_sol[i].capac:4:0);
        writeln;
        writeln(' custo = ',sol^.custo);
        writeln(' ret   = ',sol^.ret);
        writeln(' somas = ',somas);
        writeln(' comps = ',comps);
    end;
end;

```

```

(*2*****
(*2**      Proc.  SOL_OTIMA  -  Procura a solucao otima na tabela  **)
(*2*  resultante do MERGE total ( que e a ultima ), e incorpora a  **)
(*2*  solucao tabelas deixadas de lado por so terem uma unica capa- **)
(*2*  cidade disponivel.                                          **)
(*2*      E chamado pelo Programa Principal .                      **)

```

```

procedure Sol_Otima;
var i,j : integer;
begin
  if Tabela[1].tam > 0 then begin
    sol:=tabela[1].link;
    i:=2;
    while tabela[i].tam = 1 do begin
      sol^.custo := sol^.custo + tabela[i].link^.custo;
      sol^.ret := sol^.ret + tabela[i].link^.ret;
      for j:=1 to MAX_CAPS do
        sol^.caps[j] := sol^.caps[j]
          + tabela[i].link^.caps[j];
      i:=i+1;
      n_canaais := n_canaais +1;
    end;
    Imprime_Sol;
  end
  else if aprox_ini then begin
    ALOCA_NO(sol);
    sol^.ret := 0;
    sol^.custo := custo_bound;
    i:=2;
    while tabela[i].tam = 1 do begin
      sol^.custo := sol^.custo + tabela[i].link^.custo;
      i:=i+1;
      n_canaais := n_canaais +1;
    end;
    for i:=1 to n_canaais do
      sol^.ret := sol^.ret + vet_sol[i].ret;
    Imprime_Sol_2;
  end
  else

```

```

        writeln(' Problema nao tem solucao viavel ! ');
end;

(*****
(*****          Programa          Principal          *****)
begin
    Inicia;
    if Monta_tabela then begin

        {***-- Prog Din --***}
        i:=1;
        while i <= N_canaais-1 do
            if Tabela[i].tam > 0 then begin
                DEC;
                i:=i+1;
            end
            else
                i:=N_canaais;
            end

        Sol_Otima;
    end
    else
        writeln(' Problema nao tem solucao viavel ! ');

        writeln;
        writeln('razao : ',(comps/est)*100:13:11,' %');
        writeln;
        gettime(h,min,s,ds);
        writeln('Tempo de execucao ',min,':',s,':',ds);
    end.

æ

```

A.2 - Implementação com a estratégia sequencial para os sort-merges, e a política sequencial ótima para a escolha das tabelas.

```
(*1*****
(*1          DESIGNACAO de CAPACIDADES DISCRETAS          *)
(*1                                                         *)
(*1          Recebe uma rede com topologia , roteamento e fluxos nos *)
(*1 canais definidos, e devolve uma capacidade para cada canal, com *)
(*1 custo minimo possivel, sujeita a um conjunto de capacidades com *)
(*1 valores discretos disponiveis. *)
(*1                                                         *)
(*1*****
(*1  DEFINICOES : *)
(*1          Combinacao - juncao de dois canais com capacidades dis- *)
(*1                      cretas viaveis. *)
(*1                                                         *)
(*1                                                         *)
(*1          Possivel Combinacao - Combinacoes de capacidades discretas *)
(*1                      viaveis fazendo eliminacao de inefi- *)
(*1                      cientes atraves de bounds. *)
(*1                                                         *)
(*1          No - Guarda uma possivel combinacao com custo e retardo *)
(*1                  associado *)
(*1                                                         *)
(*1          Tabela - Conjunto de "nos" para um determinado conjunto *)
(*1                  de canais *)
(*1                                                         *)
(*1                                                         *)
(*1          Alvo - Tabela resultado da juncao de duas tabelas *)
(*1                                                         *)
(*1*****
(*1          TABELA e um vetor de tabelas, uma para canal, e no decorrer *)
(*1 do programa, todos os elementos em uma tabela estaraao ordenados *)
(*1 por ordem crescente de retardo, logo por ordem decrescente de *)
(*1 custos, e terao como componentes conjuntos de canais, cada conj. *)
(*1 representando uma capacidade disponivel, onde se determinara qual *)
```

```

(*1 capacidade sera atribuida a cada canal. *)
(*1*****

(*****
****
*          TABELAS ORDENADAS EM ORDEM CRESCENTE DE CUSTO          *
*          ESTRATEGIA : PROGDIN em TABELAS e PROGDIN EM SUBTABELAS *
****
*****
)
program CA_discreto;
uses crt,dos;
const MAX      = 50;
      MAX_CAPS = 5;
      EPSILON  = 10E-5;

type
  apont = ^no;
  conj  = set of 1..max;
  vetconj = array [1..MAX_CAPS] of conj;
  No = record
      ret ,
      custo : real;
      caps : vetconj;
      prox : apont;
  end;
  Item_tabela = record
      link: apont;
      tam : integer;
  end;
  Item_aprox = record
      capac, ret : real;
      arco : integer;
  end;

var
  est : real;
  capacidade : array[1..MAX_CAPS] of integer; { vetor de possiveis
                                                  capacidades }
  GTmax,                                     { gama * Tmax }

```



```

    alfa, custo_bound,
    gama,T_max : real;
    Sol,
    No_livre : apont;
    i,j,
    N_canais : integer;
    somas,
    comps : longint;
    tabela : array[1..MAX] of Item_tabela;
    entrada : text;
    Arq_Ent : string[12];
    h,min,s,ds : word;
    aux,d,dl,f : array [1..MAX] of real;
    vet_sol : array [1..max] of Item_aprox;
    aprox_ini : boolean;

    { solucao }
    { cabeca da lista livre }
    { contadores }
    { n de canais }
    { n de somas }
    { n de comparacoes }
    { arquivo de entrada }
    { string para leitura }
    { conta tempo }

(*2*****
(*2          Proc. Sol_Inicial :
(*2      Proc. que da um chute inicial para CA suj. a Retardo
(*2      repartindo os retardos proporcionalmente para cada canal, e ob-
(*      tendo capacidades iniciais continuas.
(*2          E chamado pelo proc. RESOLVE.

procedure Sol_Inicial;
var soma:real;
begin
    soma:=0;
    for i:=1 to N_canais do
        { distribui Gama*Tmax por
          cada capacidade }

        soma:=soma+sqrt(d[i]*f[i]);
    for i := 1 to N_canais do begin
        aux[i] := alfa*d[i];
        { 'aux' calculado no inicio
          e so uma vez }

        vet_sol[i].capac := f[i]*( 1+1/(GTmax*sqrt(d[i]*f[i])/soma));
        vet_sol[i].arco := i;
    end;
end;

```

```

(*2*****
(*2          Proc.      Lineariza :                               *)
(*2          Proc. que devolve custos linearizados                *)
(*2  Calcula a derivada da formula fechada para custos, obtendo cus- *)
(*2 tos linearizados para resolucao do problema, com capacidades    *)
(*2 continuas.                                                       *)
(*2          E chamado pelo proc.  RESOLVE.                         *)

```

```

procedure Lineariza;
begin
    for i := 1 to N_canais do
        dl[i] := aux[i] * exp ( (alfa-1)*ln( vet_sol[i].capac ) )
    end;

```

```

(*2*****
(*2          Proc.      Resolve :                               *)
(*2          Proc. que resolve CA sujeito a Retardo                *)
(*2  Calcula capac. otima do problema linearizado usando a formula  *)
(*2 fechada, e reaplica esta solucao iterativamente no problema nao *)
(*2 linear. A sol. final e de capac. continuas.                     *)
(*2          E chamado pelo proc.  SOL_DISCRETA_VIAVEL.            *)

```

```

procedure Resolve;
var va,aux,beta : real;
begin
    j:=0;
    custo_bound := 0;
    Sol_Inicial;      { pega um chute inicial }
    repeat
        j := j+1;      { conta iteracoes }
        va := custo_bound; { va = valor anterior do custo }
        custo_bound := 0;
        Lineariza;      { obtem custos linearizados }
        aux := 0;
        for i := 1 to n_canais do

```

```

        aux := aux + sqrt( f[i]*dl[i] );    { calculos... }
    aux := aux/T_Max ;
    beta := aux*aux/gama;
    aux := aux/gama;
    for i := 1 to n_canaais do begin
        vet_sol[i].capac := f[i] + aux*sqrt(f[i]/dl[i]);
        vet_sol[i].ret := f[i] / ( vet_sol[i].capac - f[i]);
        custo_bound:=custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
    end;

    until ( abs(va - custo_bound)/custo_bound ) <= EPSILON; {diferenca
                                                                relativa }
end;                                     { como condicao de parada }

(*2*****
(*2*          Proc. SOL_DISCRETA_VIAVEL - resolve o problema para *)
(*2*  capacidades continuas e depois obtem uma solucao viavel discre-*)
(*2*  ta a partir desta. *)
(*2*          E chamado pelo  proc. MONTA_TABELA *)

procedure Sol_discreta_viaavel;
var GT, retar : real;
    i,j,k : integer;
    sobe : boolean;
    desce : array [ 1 .. MAX ] of integer;
begin
    Resolve;
    k := 0;
    for i:=1 to N_canaais do
        if capacidade[MAX_CAPS] < vet_sol[i].capac then begin
            k := k+1;
            desce[k] := i;
        end;
    GT := GTmax;
    if k < MAX  then begin
        for i := 1 to k do begin
            vet_Sol[desce[i]].capac := Capacidade[MAX_CAPS];

```

```

        GT := GT - vet_sol[desce[i]].ret;
        Vet_Sol[desce[i]].ret:= f[desce[i]] / (capacidade[MAX_CAPS]
- f[desce[i]]);
        GT := GT + vet_sol[desce[i]].ret;
    end;

    aprox_ini := true;

    while aprox_ini and ( (GT > GTMax) or (k=0) ) do begin
        for i:=1 to N_canaais do
            if vet_sol[i].capac < capacidade[MAX_CAPS] then begin
                j := 1;
                while capacidade[j] <= Vet_sol[i].capac do j:=j+1;
                retar := f[i] / ( capacidade[j-1] - f[i]);
                GT := GT - vet_sol[i].ret;
                if (capacidade[j-1] > f[i]) and (GTmax-GT-retar>=0)
                    then
                        j:=j-1;
                retar := f[i] / (capacidade[j] - f[i]);
                vet_sol[i].capac := capacidade[j];
                GT := GT + retar;
                vet_sol[i].ret := retar;

            end;
            j:=1; k:=1;
            while (j <= N_canaais ) and (aprox_ini) do begin
                aprox_ini:=capacidade[MAX_CAPS]=vet_sol[j].capac;
                j:=j+1;
            end;
            aprox_ini := j < N_canaais
        end;

        custo_bound := 0;
        for i:=1 to N_canaais do
            custo_bound:=custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
            aprox_ini := GT <= GTMax;
        end
    else aprox_ini := false;

```

end;

```
(*2*****  
(*2*** Proc. ALOCA_NO - aloca um no para tabela de canais ***)  
(*2* Mantem uma lista de celulas livres para alocao **)
```

```
procedure aloca_no(var P : apont);  
var j:integer;  
begin  
  if No_livre = NIL then  
    NEW(p)  
  else begin  
    P:=No_Livre;  
    No_Livre := No_Livre^.prox;  
  end;  
  with p^ do begin  
    ret := 0; custo := 0;  
    for j:=1 to MAX_CAPS do caps[j] := [];  
    prox := NIL;  
  end;  
end;
```

```
(*2*****  
(*2** Proc. DEVOLVE_NO - deixa um no que nao e mais usado **)  
(*2* livre para posterior utilizacao, usando lista livre. **)
```

```
procedure Devolve_No(var P : apont);  
var j:integer;  
begin  
  P^.prox := No_Livre;  
  No_livre := P;  
end;
```

```
(*2*****  
(*2** Proc. LIBERA - Libera todo o espaco ocupado por uma **)  
(*2* tabela de canais, devolvendo todos os nos por ela utilizados **)  
(*2* para lista livre. **)
```



```

procedure Libera( num : integer);
var p:apont;
begin
    while tabela[num].link <> NIL do begin
        p:=tabela[num].link;
        tabela[num].link := tabela[num].link^.prox;
        Devolve_No ( p );
    end;
end;

(*2*****
(*2**      Proc.  EMPURRA  - Se alguma tabela no vetor de tabelas **)
(*2*   e liberado, este proc. se encarrega de deixar todas as tabelas *)
(*2*   restantes adjacentes umas as outras.                               **)

procedure Empurra(n:integer);
var i:integer;
begin
    for i:=n+1 to max do
        tabela[i-1]:=tabela[i];
    end;

(*2*****
(*2**      Func.  ELIMINA  - Determina se dado no'   gerado          **)
(*2*   no MERGE deve ou nao ser descartado, verificando-se Bounds de **)
(*2*   programacao.                                           **)
(*2*      E chamado pelo proc. MERGE.                          **)

function Elimina(t,c,GamaT:real):boolean;
begin
    Elimina := (t > GamaT) or (c > custo_bound);
end;

(*2*****
(*2**      Proc.  ELIMINA_INEFICIENTES  -   Dada uma tabela de **)

```

```

(*2*  canais, retira todos os "ineficientes" desta tabela, utili- **)
(*2*  zando a estrategia de Bound 3 (v/ lista Bounds ).          **)
(*2*      E chamado pelo proc. MERGE.                            **)

```

```

procedure Elimina_Ineficientes ( n:integer );
var p,q : apont;
begin
    p := tabela[n].link;
    q:=p;
    if p <> NIL then
        p:=p^.prox;
    while p <> NIL do begin
        if (p^.ret > q^.ret) then begin
            q^.prox := p^.prox;
            Devolve_No ( p );
            tabela[n].tam := tabela[n].tam -1;
        end
        else
            q:=q^.prox;
            p:=q^.prox;
        end;
    end;
end;

```

```

(*2*****
(*2**      Proc.      DeC      -      Toma duas tabelas de canais e mis- **)
(*2*  tura-as,gerando uma so tabela ,que e devolvida na primeira **)
(*2*  delas ,eliminando-se ineficientes e memoria nao utilizada . **)
(*2*  O proc DeC monta todas as sub-tabelas iniciais resultantes da **)
(*2*  soma dos elementos de duas tabelas, e as envia para o proc. **)
(*2*  MERGE, para juntar todas as subtabelas em uma unica,utilizan- **)
(*2*  do a estrategia Divide & Conquer.                               **)
(*2*****
***
*      Lista de Bounds utilizados :
*      BOUND 1 : dois "nos" com retardos iguais , elimina-se
*                  o de maior custo.
*
*      BOUND 2 : o retardo de um novo "no" criado deve ser
*

```

```

*          menor que o retardo maximo menos o minimo      *
*          dos retardos das tabelas anteriormente          *
*          criadas .                                       *
*
*          BOUND 3 : se existir dois "nos" , em que retardo e      *
*          custo de um sao menores que retardo e          *
*          custo do outro , o outro deve ser elimi-      *
*          nado.                                           *
*
***
*****
(*2*      E chamado pelo Programa Principal , e chama como auxili- **)
(*2* ares os procs. Elimina, Elimina_Ineficientes, Libera, Empurra, **)
(*2* Aloca_No .                                           **)

procedure DeC;
var
    alvo,
    p, auxp, auxq,      { p aloca nova cel, auxp e auxq percorrem }
    ultcomp,proxcomp,sub_ptr :apont; {tabelas menor e maior respect./}
    i, j,
    p1,p2,
    m,n,                { tamanho das tabelas }
    tam      : integer;
    aux,aux2 : real;
    Sub_Tab  : array [ 1 .. MAX_CAPS ] of Item_Tabela;

(*****
***
*   PROCEDURE MERGE : recebe duas sub-tabelas e faz um sort      *
*   merge sobre elas , destaca-se neste procedimento o fato     *
*   de por ser melhor manter as tabelas por ordem crescente de  *
*   custo e nao de retardo , os elementos das sub-tabelas sao   *
*   comparados por seus custos , afim de manter a eficiencia   *
*   do procedimento ELIMINA INEFICIENTES.                        *
***
*****)
procedure merge ( prim , seg : integer );
    var j : integer;

```

```

begin
  Sub_ptr := Sub_Tab[prim].link;
  auxq    := Sub_ptr^.prox;
  auxp    := Sub_Tab[seg].link;
  while ( auxp <> nil ) do begin

    (* PROCURA A POSICAO PARA INSERCAO *)
    while (auxq^.custo < auxp^.custo) and ( auxq <> nil ) do begin
      COMPS := COMPS + 1;
      Sub_Ptr := Sub_Ptr^.prox;
      auxq    := auxq^.prox;
    end;

    (* Terminou o sort merge apenas une o restante das tabelas *)
    if auxq = nil then begin
      Sub_Ptr^.prox := auxp ;
      auxp := nil;
    end
    else begin
      p := auxp^.prox;

      (* custos iguais , permanece o de menor retardo *)
      if auxq^.custo = auxp^.custo then
        if auxq^.ret > auxp^.ret then begin

          auxq^.ret := auxp^.ret;
          for j := 1 to MAX_CAPS do
            auxq^.caps[j] := auxp^.caps[j]
          end
        else

          (* Insere novo elemento na posicao *)
          else begin
            auxp^.prox := auxq;
            Sub_Ptr^.prox := auxp;
          end;
          auxp := p;
        end
      end
    end
  end
end

```

```

    end;
end;

(*2*****      Procedimento      DeC      *****)
begin
    tam:=0;
    { Coloca em tabela de p1 a tabela de menor tamanho }
    p1:=1; p2:=2;
    if tabela[2].tam < tabela[1].tam then begin
        n := p1;
        p1:= p2;
        p2:= n;
    end;
    n:=tabela[p1].tam;
    m:=tabela[p2].tam;

    auxp:=tabela[p1].link;
    auxq:=tabela[p2].link;

    for i := 1 to MAX_CAPS do begin
        Sub_Tab[i].link:= nil;
        Sub_Tab[i].Tam := 0;
    end;
    i := 1;
    while ( auxp <> Nil ) do begin          { monta as subtabelas }
        while auxq <> NIL do begin
            aux:=auxp^.ret + auxq^.ret;
            aux2 := auxp^.custo + auxq^.custo;
            somas := somas+1;
            if Sub_Tab[i].link <> NIL then begin
                if not Elimina(Aux,Aux2,GTmax) then begin
                    ALOCA_NO(p);
                    Sub_Tab[i].tam:=Sub_Tab[i].tam+1;
                    p^.prox := nil;
                    p^.ret := aux;
                    p^.custo := aux2;
                    for j:=1 to MAX_CAPS do
                        p^.caps[j] := auxp^.caps[j]
                        + auxq^.caps[j];

```



```

                                sub_ptr^.prox:=p;
                                sub_ptr := p;
                                end
                                else
                                end
                                else if not Elimina( aux, aux2, GTmax) then begin
                                    ALOCA_NO(Sub_Tab[i].link);
                                    Sub_Tab[i].tam := 1;
                                    Sub_Ptr := Sub_Tab[i].link;
                                    Sub_Tab[i].link^.prox := NIL;
                                    Sub_Tab[i].link^.ret := aux;
                                    Sub_Tab[i].link^.custo := aux2;
                                    for j:=1 to MAX_CAPS do
                                        Sub_Tab[i].link^.caps[j] := auxp^.caps[j]
                                        + auxq^.caps[j];
                                    end;
                                    auxq:=auxq^.prox;

                                end; { while interno }
                                i := i+1;
                                auxp:=auxp^.prox;
                                auxq:=tabela[p2].link;

                                end; { while externo }
                                while Sub_Tab[2].link <> NIL do begin { Mergeia as subtabelas }
                                    MERGE ( 1 , 2 );
                                    for j := 3 to MAX_CAPS do
                                        Sub_Tab [j-1] := Sub_Tab [j];
                                    Sub_Tab [MAX_CAPS].tam := 0;
                                    Sub_Tab [MAX_CAPS].link := NIL;
                                end;
                                p:=sub_tab[1].link;
                                sub_tab[1].tam:=0;
                                while p<>NIL do begin
                                    p:=p^.prox;
                                    sub_tab[1].tam := sub_tab[1].tam + 1;
                                end;
                                Libera(1);
                                Libera(2);
                                { cuidados finais }

```

```

    tabela[1].link:= Sub_Tab[1].link;
    tabela[1].tam := Sub_tab[1].tam;
    Empurra(2);
    Elimina_Ineficientes(1);
end;

(*2*****
(*2**      Proc.      SORT -  Ordena as tabelas de canais por ordem **)
(*2*  decrescente de tamanho, facilitando posterior MERGE .          **)
(*2*      Versao recursiva do Quick-Sort.                             **)
(*2*      E chamado pelo Monta_Tabela.                               **)

procedure SORT(l,r : integer);
var i,j : integer;
    x,w : Item_tabela;
begin
    i := l; j := r;
    x := tabela[ (l+r)div 2 ];
    repeat
        while tabela[i].tam > x.tam do
            i := i+1;
        while tabela[j].tam < x.tam do
            j := j-1;
        if i <= j then begin
            w := tabela[i];
            tabela[i] := tabela[j];
            tabela[j] := w;
            i := i+1;
            j := j-1;
        end;
    until i > j;
    if l < j then
        SORT (l,j);
    if i < r then
        SORT (i,r);
end;

procedure estimativa;

```

```

var i : integer;
    tam : real;
begin
    est:=0;
    tam:=tabela[1].tam;
    for i:=2 to n_canais do begin
        est:=est + tam * ((tabela[i].tam + 2)*tabela[i].tam/2 - 1);
        tam:=tabela[i].tam * tam;
    end;
end;

```

```

(*2*****
(*2**      Func.      MONTA_TABELA - Le a rede do arquivo de entrada **)
(*2*  e monta as tabelas de canais, ordena a tabela, verifica via- **)
(*2*  bilidade inicial do problema, e reduz o n de tabelas, descon-**)
(*2*  siderando as que so tem uma capacidade possivel. **)
(*2*      E chamado pelo Programa Principal. **)

```

```

function Monta_tabela : boolean;
var i,j,k :integer;
    p : apont;
    fluxo,cust : real;

begin
    clrscr;
    writeln;writeln;
    writeln('      Designador de Capacidades - Versao Discreta ');
    writeln;writeln;
    write(' Arquivo entrada : ');          { leitura }
    readln(arq_ent);
    writeln;
    arq_ent := arq_ent+'.dat';
    assign(entrada,arq_ent);
    {$i-}
    reset(entrada);
    {$i+}
    if IOresult <> 0 then begin

```

```

        writeln (' Erro, ',arq_ent,' nao esta no drive ');
        delay(2000);
        halt
    end;

SETTIME(0,0,0,0);
{$i-}
readln(entrada,Gama,T_max,alfa);
{$i+}
if IOresult <> 0 then begin
    writeln ('Erro, Incoerencia no formato do arquivo na Linha 1');
    delay(2000);
    halt
end;

GTmax := gama * T_max;
{$i-}
readln(entrada,N_canais);
{$i+}
if IOresult <> 0 then begin
    writeln ('Erro, Incoerencia no formato do arquivo na Linha 2');
    delay(2000);
    halt
end;

for i:=1 to N_canais do begin
    {$i-}
    readln(entrada,fluxo,cust);
    {$i+}
    if IOresult <> 0 then begin
        writeln (' Erro, Incoerencia no formato do arquivo na'
            , 'Linha ',i+2);
        delay(2000);
        halt
    end;
    if Fluxo > Capacidade[MAX_CAPS] then begin
        writeln ('Erro,Nao existe capacidade disponivel para fluxo'
            , 'definido na Linha ',i+2,'do arquivo');
        delay(2000);
    end;
end;

```

```

        halt
    end;

    f[i] := fluxo; d[i] := cust;
    for j:=MAX_CAPS downto 1 do      { monta tabela para canal i }
        if Capacidade[j] > fluxo then begin
            tabela[i].tam := tabela[i].tam + 1;
            ALOCA_NO(p);
            with p^ do begin
                ret := fluxo/(capacidade[j] - fluxo);
                custo := cust * exp(alfa * ln(capacidade[j])) );
                caps[j] := [i];
                prox := tabela[i].link;
            end;
            tabela[i].link := p;
        end;
    end;

end;

SOL_DISCRETA_VIAVEL;

cust := 0;
SORT(1,N_canaais);                      { ordena }
if tabela[N_Canaais].tam = 0 then
    monta_tabela := false
else begin

    ESTIMATIVA;

    while (tabela[n_canaais].tam = 1) and (N_canaais > 0) do begin
        GTmax := GTmax - tabela[n_canaais].link^.ret;
        cust := cust + tabela[n_canaais].link^.custo;
        n_canaais := n_canaais -1;
    end;

    if aprox_ini then CUSTO_BOUND := CUSTO_BOUND - cust;

    Monta_Tabela := (GTmax > 0) and (aprox_ini);
end;
close(entrada);

```


end;

```
(*2*****  
(*2**   Proc.   INICIA - Inicializa variaveis do programa.   **)  
(*2*     E chamado pelo Programa Principal.                 **)
```

```
procedure INICIA;  
var i:integer;  
begin  
    No_Livre := NIL;  
    capacidade[1] := 300;  
    capacidade[2] := 1200;  
    capacidade[3] := 2400;  
    capacidade[4] := 4800;  
    capacidade[5] := 9600;  
    for i:=1 to MAX do begin  
        tabela[i].tam:=0;  
        tabela[i].link:=NIL;  
    end;  
    somas :=0; comps :=0;  
    aprox_ini := false;  
    custo_bound := 1E38;  
end;
```

```
(*2*****  
(*2*   Proc.   IMPRIME_SOL - Imprime a solucao otima do problema **)  
(*2*     E chamado pelo proc. Sol_Otima .                        **)
```

```
procedure Imprime_sol;  
begin  
    for i:=1 to N_canais do  
        for j:=1 to MAX_Caps do  
            if i IN sol^.caps[j] then  
                writeln('canal ',i,' com cap. = ',capacidade[j]);  
            writeln;  
            writeln(' custo = ',sol^.custo);  
            writeln(' ret   = ',sol^.ret);  
        end;  
    end;
```

```

        writeln(' somas = ',somas);
        writeln(' comps = ',comps);
end;

(*2*****
(*2*      Proc.  IMPRIME_SOL_2 - Imprime a solucao otima do problema **)
(*2*      E chamado pelo proc. Sol_Otima .                               **)

procedure Imprime_sol_2;
var i:integer;
begin
    for i:=1 to n_canais do
        writeln('canal ',i,' com cap. = ',vet_sol[i].capac:4:0);
    writeln;
    writeln(' custo = ',sol^.custo);
    writeln(' ret   = ',sol^.ret);
    writeln(' somas = ',somas);
    writeln(' comps = ',comps);
end;

(*2*****
(*2**      Proc.  SOL_OTIMA   - Procura a solucao otima na tabela **)
(*2*  resultado do MERGE total ( que e a ultima ), e incorpora a **)
(*2*  solucao tabelas deixadas de lado por so terem uma unica capa- **)
(*2*  cidade disponivel.                                           **)
(*2*      E chamado pelo Programa Principal .                               **)

procedure Sol_Otima;
var i,j : integer;
begin
    if Tabela[1].tam > 0 then begin
        sol:=tabela[1].link;
        i:=2;
        while tabela[i].tam = 1 do begin
            sol^.custo := sol^.custo + tabela[i].link^.custo;
            sol^.ret := sol^.ret + tabela[i].link^.ret;
            for j:=1 to MAX_CAPS do
                sol^.caps[j]:=sol^.caps[j]+tabela[i].link^.caps[j];

```

```

        i:=i+1;
        n_canais := n_canais +1;
    end;
    Imprime_Sol;
end
else if aprox_ini then begin
    ALOCA_NO(sol);
    sol^.ret := 0;
    sol^.custo := custo_bound;
    i:=2;
    while tabela[i].tam = 1 do begin
        sol^.custo := sol^.custo + tabela[i].link^.custo;
        i:=i+1;
        n_canais := n_canais +1;
    end;
    for i:=1 to n_canais do
        sol^.ret := sol^.ret + vet_sol[i].ret;
    end;
    Imprime_Sol_2;
end
else
    writeln(' Problema nao tem solucao viavel ! ');
end;

```

```

(*****)
(*****      Programa      Principal      *****)
begin
    Inicia;
    if Monta_tabela then begin

        {***-- Prog Din --***}
        i:=1;
        while i <= N_canais-1 do
            if Tabela[i].tam > 0 then begin
                DEC;
                i:=i+1;
            end
            else

```

```

        i:=N_canaais;

        Sol_Otima;
    end
    else
        writeln(' Problema nao tem solucao viavel ! ');

        writeln;
        writeln('razao : ',(comps/est)*100:13:11,' %');
        writeln;
        gettime(h,min,s,ds);
        writeln('Tempo de execucao ',min,':',s,':',ds);
    end.

    æ

```

A.3 - Implementação com a estratégia gulosa para os sort-merges e estratégia gulosa para a escolha das tabelas.

```
(*1*****)
(*1      DESIGNACAO de CAPACIDADES DISCRETAS      *)
(*1      *)
(*1      Recebe uma rede com topologia , roteamento e fluxos nos      *)
(*1      canais definidos, e devolve uma capacidade para cada canal, com *)
(*1      custo minimo possivel, sujeita a um conjunto de capacidades com *)
(*1      valores discretos disponiveis.      *)
(*1      *)
(*1*****)
(*1      DEFINICOES :      *)
(*1      Combinacao - juncao de dois canais com capacidades dis-      *)
(*1      cretas viaveis.      *)
(*1      *)
(*1      *)
(*1      Possivel Combinacao - Combinacoes de capacidades discretas      *)
(*1      viaveis fazendo eliminacao de inefi-      *)
(*1      cientes atraves de bounds.      *)
(*1      *)
(*1      No - Guarda uma possivel combinacao com custo e retardo      *)
(*1      associado      *)
(*1      *)
(*1      Tabela - Conjunto de "nos" para um determinado conjunto      *)
(*1      de canais      *)
(*1      *)
(*1      *)
(*1      Alvo - Tabela resultado da juncao de duas tabelas      *)
(*1      *)
(*1      *)
(*1*****)
(*1      TABELA e um vetor de tabelas, uma para canal, e no decorrer *)
(*1      do programa, todos os elementos em uma tabela estaraos ordenados *)
(*1      por ordem crescente de retardo, logo por ordem decrescente de *)
(*1      custos, e terao como componentes conjuntos de canais, cada conj. *)
(*1      representando uma capacidade disponivel, onde se determinara qual *)
```



```

(*1 capacidade sera atribuida a cada canal. *)
(*1*****

(*****
****
*          TABELAS ORDENADAS EM ORDEM CRESCENTE DE CUSTO          *
*          ESTRATEGIA : DeC em TABELAS e DeC EM SUBTABELAS        *
*          e merge ida e volta                                     ****
*****)
program CA_discreto;
uses crt,dos;
const MAX      = 50;
      MAX_CAPS = 5;
      EPSILON  = 10E-5;
type
  apont = ^no;
  conj  = set of 1..max;
  vetconj = array [1..MAX_CAPS] of conj;
  No = record
    ret ,
    custo : real;
    caps  : vetconj;
    prox  : apont;
  end;
  Item_tabela = record
    link : apont;
    tam  : integer;
  end;
  Item_aprox = record
    capac, ret : real;
    arco : integer;
  end;

var
  est : real;
  capacidade : array[1..MAX_CAPS] of integer; { vetor de possiveis
  capacidades }
  GTmax,                                     { gama * Tmax }

```

```

    alfa, custo_bound,
    gama,T_max : real;
    Sol,
    No_livre : apont;
    i,j,cont,
    N_canaais : integer;
    somas,
    comps : longint;
    tabela : array[1..MAX] of Item_tabela;
    entrada : text;
    Arq_Ent : string[12];
    h,min,s,ds : word;
    aux,d,dl,f : array [1..MAX] of real;
    vet_sol : array [1..max] of Item_aprox;
    aprox_ini : boolean;

    { solucao }
    { cabeca da lista livre }
    { contadores }
    { n de canais }
    { n de somas }
    { n de comparacoes }
    { arquivo de entrada }
    { string para leitura }
    { conta tempo }

(*2*****
(*2 Proc. Sol_Inicial : *)
(*2 Proc. que da um chute inicial para CA suj. a Retardo *)
(*2 repartindo os retardos proporcionalmente para cada canal, e ob- *)
(* tendo capacidades iniciais continuas. *)
(*2 E chamado pelo proc. RESOLVE. *)

procedure Sol_Inicial;
var soma:real;
begin
    soma:=0;
    for i:=1 to N_canaais do
        { distribui Gama*Tmax por
        cada capacidade }
        soma:=soma+sqrt(d[i]*f[i]);
    for i := 1 to N_canaais do begin
        aux[i] := alfa*d[i];
        { 'aux' calculado no inicio e so
        uma vez }
        vet_sol[i].capac := f[i]*( 1+1/(GTmax*sqrt(d[i]*f[i])/soma) );
        vet_sol[i].arco := i;
    end;
end;
end;

```

```

(*2*****
(*2          Proc.      Lineariza :                               *)
(*2          Proc. que devolve custos linearizados                *)
(*2  Calcula a derivada da formula fechada para custos, obtendo cus- *)
(*2 tos linearizados para resolucao do problema, com capacidades   *)
(*2 continuas.                                                    *)
(*2          E chamado pelo proc.  RESOLVE.                        *)

```

```

procedure Lineariza;
begin
    { obtidos pela derivada dos
    valores }
    for i := 1 to N_canais do
        { iniciais do custo }
        dl[i] := aux[i] * exp ( (alfa-1)*ln( vet_sol[i].capac ) )
end;

```

```

(*2*****
(*2          Proc.      Resolve :                               *)
(*2          Proc. que resolve CA sujeito a Retardo              *)
(*2  Calcula capac. otima do problema linearizado usando a formula *)
(*2 fechada, e reaplica esta solucao iterativamente no problema nao *)
(*2 linear. A sol. final e de capac. continuas.                  *)
(*2          E chamado pelo proc. SOL_DISCRETA_VIAVEL.           *)

```

```

procedure Resolve;
var va,aux,beta : real;
begin
    j:=0;
    custo_bound := 0;
    Sol_Inicial;    { pega um chute inicial }
    repeat
        j := j+1;          { conta iteracoes }
        va := custo_bound; { va = valor anterior do custo }
        custo_bound := 0;
        Lineariza;          { obtem custos linearizados }
        aux := 0;
    until

```

```

    for i := 1 to n_canaais do
        aux := aux + sqrt( f[i]*dl[i] );    { calculos... }
    aux := aux/T_Max ;
    beta := aux*aux/gama;
    aux := aux/gama;
    for i := 1 to n_canaais do begin
        vet_sol[i].capac := f[i] + aux*sqrt(f[i]/dl[i]);
        vet_sol[i].ret := f[i] / ( vet_sol[i].capac - f[i]);
        custo_bound:=custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
    end;

    until ( abs(va - custo_bound)/custo_bound ) <= EPSILON; {diferenca
        relativa}
end;
{como condicao
de parada}

```

```

(*2*****
(*2*          Proc. SOL_DISCRETA_VIAVEL - resolve o problema para *)
(*2*  capacidades continuas e depois obtem uma solucao viavel discre-*)
(*2*  ta a partir desta. *)
(*2*          E chamado pelo proc. MONTA_TABELA *)

```

```

procedure Sol_discreta_viavel;
var GT, retar : real;
    i,j,k : integer;
    sobe : boolean;
    desce : array [ 1 .. MAX ] of integer;
begin
    Resolve;
    k := 0;
    for i:=1 to N_canaais do
        if capacidade[MAX_CAPS] < vet_sol[i].capac then begin
            k := k+1;
            desce[k] := i;
        end;
    GT := GTmax;
    if k < MAX then begin

```

```

for i := 1 to k do begin
    vet_sol[desce[i]].capac := Capacidade[MAX_CAPS];
    GT := GT - vet_sol[desce[i]].ret;
    Vet_sol[desce[i]].ret := f[desce[i]] / ( capacidade[MAX_CAPS]
- f[desce[i]]);
    GT := GT + vet_sol[desce[i]].ret;
end;

aprox_ini := true;

while aprox_ini and ( (GT > GTMax) or (k=0) ) do begin
    for i:=1 to N_canaais do
        if vet_sol[i].capac < capacidade[MAX_CAPS] then begin
            j := 1;
            while capacidade[j] <= Vet_sol[i].capac do j:=j+1;
            retar := f[i] / ( capacidade[j-1] - f[i]);
            GT := GT - vet_sol[i].ret;
            if (capacidade[j-1]>f[i]) and (GTmax-GT-retar >= 0)
            then
                j:=j-1;
            retar := f[i] / (capacidade[j] - f[i]);
            vet_sol[i].capac := capacidade[j];
            GT := GT + retar;
            vet_sol[i].ret := retar;

            end;
        j:=1; k:=1;
        while (j <= N_canaais ) and (aprox_ini) do begin
            aprox_ini := capacidade[MAX_CAPS] = vet_sol[j].capac;
            j:=j+1;
        end;
        aprox_ini := j < N_canaais
    end;

    custo_bound := 0;
    for i:=1 to N_canaais do
        custo_bound:=custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
    aprox_ini := GT <= GTMax;

```



```

        end
        else aprox_ini := false;
end;

(*2*****
(*2***  Proc.  ALOCA_NO    -  aloca um no para tabela de canais    ***)
(*2*   Mantem  uma  lista de celulas livres para alocao          ***)

procedure aloca_no(var P : apont);
var j:integer;
begin
    if No_livre = NIL then
        NEW(p)
    else begin
        P:=No_Livre;
        No_Livre := No_Livre^.prox;
    end;
    with p^ do begin
        ret := 0; custo := 0;
        for j:=1 to MAX_CAPS do caps[j] := [];
        prox := NIL;
    end;
end;

(*2*****
(*2**   Proc.  DEVOLVE_NO -  deixa um no que nao e mais usado    ***)
(*2*   livre para posterior utilizacao, usando lista livre.      ***)

procedure Devolve_No(var P : apont);
var j:integer;
begin
    if P<>NIL then begin
        with p^ do begin
            ret := 0; custo := 0;
            for j:=1 to MAX_CAPS do caps[j] := [];
        end;
        P^.prox := No_Livre;
        No_livre := P;
    end;
end;

```

```

end;
end;

```

```

(*2*****
(*2**      Proc.  LIBERA  -  Libera todo o espaco ocupado por uma **)
(*2*  tabela de canais, devolvendo todos os nos por ela utilizados **)
(*2*  para lista livre.                                          **)

```

```

procedure Libera( num : integer);
var p:apont;
begin
    while tabela[num].link <> NIL do begin
        p:=tabela[num].link;
        tabela[num].link := tabela[num].link^.prox;
        Devolve_No ( p );
    end;
end;

```

```

(*2*****
(*2**      Proc.  EMPURRA  -  Se alguma tabela no vetor de tabelas **)
(*2*  e liberado, este proc. se encarrega de deixar todas as tabelas *)
(*2*  restantes adjacentes umas as outras.                          **)

```

```

procedure Empurra(n:integer);
var i:integer;
begin
    for i:=n+1 to max do
        tabela[i-1]:=tabela[i];
    end;

```

```

(*2*****
(*2**      Func.  ELIMINA  -  Determina se dado no'  gerado **)
(*2*  no MERGE deve ou nao ser descartado, verificando-se Bounds de **)
(*2*  programacao.                                          **)
(*2*      E chamado pelo proc. MERGE.                      **)

```

```

function Elimina(t,c,GamaT:real):boolean;
begin
    Elimina := (t > GamaT) or (c > custo_bound);
end;

(*2*****
(*2**      Proc.    ELIMINA_INEFICIENTES    -    Dada uma tabela de **)
(*2*  canais, retira todos os "ineficientes" desta tabela, utili- **)
(*2*  zando a estrategia de Bound 3 (v/ lista Bounds ). **)
(*2*      E chamado pelo proc. MERGE. **)

procedure Elimina_Ineficientes ( n:integer );
var p,q : apont;
begin
    p := tabela[n].link;
    q:=p;
    if p <> NIL then
        p:=p^.prox;
    while p <> NIL do begin
        if (p^.ret > q^.ret) then begin
            q^.prox := p^.prox;
            Devolve_No ( p );
            tabela[n].tam := tabela[n].tam -1;
        end
        else
            q:=q^.prox;
            p:=q^.prox;
        end;
    end;
end;

(*2*****
(*2**      Proc.    DeC    -    Toma duas tabelas de canais e mis- **)
(*2*  tura-as,gerando uma so tabela ,que e devolvida na primeira **)
(*2*  delas ,eliminando-se ineficientes e memoria nao utilizada . **)
(*2*  O proc DeC monta todas as sub-tabelas iniciais resultantes da **)
(*2*  soma dos elementos de duas tebelas, e as envia para o proc. **)
(*2*  MERGE, para juntar todas as subtabelas em uma unica,utilizan- **)

```

```

(*2* do a estrategia Divide & Conquer. **)
(*2*****
***
*      Lista de Bounds utilizados :
*      BOUND 1 : dois "nos" com retardos iguais , elimina-se
*                o de maior custo.
*
*      BOUND 2 : o retardo de um novo "no' criado deve ser
*                menor que o retardo maximo menos o minimo
*                dos retardos das tabelas anteriormente
*                criadas .
*
*      BOUND 3 : se existir dois "nos" , em que retardo e
*                custo de um sao menores que retardo e
*                custo do outro , o outro deve ser elimi-
*                nado.
*
***
*****
(*2*      E chamado pelo Programa Principal , e chama como auxili- **)
(*2* ares os procs. Elimina, Elimina_Ineficientes, Libera, Empurra, **)
(*2* Aloca_No . **)

```

```

procedure DeC(prim,seg : integer);
var
    alvo,
    p, auxp, auxq,          { p aloca nova cel, auxp e auxq percorrem}
    ultcomp,proxcomp,sub_ptr :apont; {tabelas menor e maior respect./}
    i, j, k,
    p1,p2,
    m,n,                    { tamanho das tabelas }
    tam      : integer;
    aux,aux2 : real;
    Sub_Tab  : array [ 1 .. MAX ] of Item_Tabela;

(*****
***
*      PROCEDURE MERGE : recebe duas sub-tabelas e faz um sort
*      merge sobre elas , destaca-se neste procedimento o fato

```

```

*   de por ser melhor manter as tabelas por ordem crescente de *
*   custo e nao de retardo , os elementos das sub-tabelas sao  *
*   comparados por seus custos , afim de manter a eficiencia  *
*   do procedimento ELIMINA INEFICIENTES.                        *
***                                                                ***
*****)
procedure merge ( prim , seg : integer );
    var j : integer;

begin
    Sub_ptr := Sub_Tab[prim].link;
    auxq    := Sub_ptr^.prox;
    auxp    := Sub_Tab[seg].link;
    while ( auxp <> nil ) do begin

        (* PROCURA A POSICAO PARA INSERCAO *)
        while (auxq^.custo < auxp^.custo) and ( auxq <> nil ) do begin
            COMPS := COMPS + 1;
            Sub_Ptr := Sub_Ptr^.prox;
            auxq    := auxq^.prox;
        end;
        COMPS := COMPS + 1;
        (* Terminou o sort merge apenas une o restante das tabelas *)
        if auxq = nil then begin
            Sub_Ptr^.prox := auxp ;
            auxp := nil;
        end
        else begin
            p := auxp^.prox;

            (* custos iguais , permanece o de menor retardo *)
            if auxq^.custo = auxp^.custo then
                if auxq^.ret > auxp^.ret then begin
                    auxq^.ret := auxp^.ret;
                    for j := 1 to MAX_CAPS do
                        auxq^.caps[j] := auxp^.caps[j]
                    end
                else

```



```

        (* Insere novo elemento na posicao *)
    else begin
        auxp^.prox := auxq;
        Sub_Ptr^.prox := auxp;
    end;
    auxp := p;
end
end;
end;
end;

```

```

(*2***** Procedimento DeC ******)
begin
    tam:=0;
    { Coloca em tabela de p1 a tabela de menor tamanho }
    p1:=prim; p2:=seg;
    if tabela[seg].tam < tabela[prim].tam then begin
        n := p1;
        p1:= p2;
        p2:= n;
    end;
    n:=tabela[p1].tam;
    m:=tabela[p2].tam;
    auxp:=tabela[p1].link;
    auxq:=tabela[p2].link;

    for i := 1 to MAX_CAPS do begin
        Sub_Tab[i].link:= nil;
        Sub_Tab[i].Tam := 0;
    end;
    i := 1;
    while ( auxp <> Nil ) do begin          { monta as subtabelas }
        while auxq <> NIL do begin
            aux:=auxp^.ret + auxq^.ret;
            aux2 := auxp^.custo + auxq^.custo;
            somas := somas+1;
            if Sub_Tab[i].link <> NIL then begin
                if not Elimina(Aux,Aux2,GTmax) then begin
                    ALOCA_NO(p);

```

```

Sub_Tab[i].tam:=Sub_Tab[i].tam+1;
p^.prox := nil;
p^.ret := aux;
p^.custo := aux2;
for j:=1 to MAX_CAPS do
    p^.caps[j]:=auxp^.caps[j]
    +auxq^.caps[j];
sub_ptr^.prox:=p;
sub_ptr := p;

end
else
end
else if not Elimina( aux, aux2, GTmax) then begin
    ALOCA_NO(Sub_Tab[i].link);
    Sub_Tab[i].tam := 1;
    Sub_Ptr := Sub_Tab[i].link;
    Sub_Tab[i].link^.prox := NIL;
    Sub_Tab[i].link^.ret := aux;
    Sub_Tab[i].link^.custo := aux2;
    for j:=1 to MAX_CAPS do
        Sub_Tab[i].link^.caps[j]:=auxp^.caps[j]
        +auxq^.caps[j];
    end;
    auxq:=auxq^.prox;

end; { while interno }
i := i+1;
auxp:=auxp^.prox;
auxq:=tabela[p2].link;

end; { while externo }

while Sub_Tab[2].link <> NIL do begin {Mergeia as subtabelas}
    i := 0; {pelo MERGE - Div.& Conq.}
    while Sub_Tab [i+1].link <> nil do begin
        i := i + 1;
        MERGE ( i , i+1 );
        for j := i+2 to MAX_CAPS do
            Sub_Tab [j-1] := Sub_Tab [j];

```

```

        Sub_Tab [MAX_CAPS].tam := 0;
        Sub_Tab [MAX_CAPS].link := NIL;
    end;

    for j := i downto 2 do begin
        MERGE ( j-1 , j );
        for k := j+1 to MAX_CAPS do
            Sub_Tab [k-1] := Sub_Tab [k];
        end;
    end;
end;
p:=sub_tab[1].link;
sub_tab[1].tam:=0;
while p <> NIL do begin
    p:=p^.prox;
    sub_tab[1].tam := sub_tab[1].tam + 1;
end;
Libera(prim);                                { cuidados finais }
Libera(seg);
tabela[prim].link:= Sub_Tab[1].link;
tabela[prim].tam := Sub_tab[1].tam;
Empurra(seg);
Elimina_Ineficientes(prim);
end;

(*2*****
(*2**      Proc.      SORT - Ordena as tabelas de canais por ordem **)
(*2*  decrescente de tamanho, facilitando posterior MERGE .          **)
(*2*      Versao recursiva do Quick-Sort.                             **)
(*2*      E chamado pelo Monta_Tabela.                               **)

procedure SORT(l,r : integer);
var i,j : integer;
    x,w : Item_tabela;
begin
    i := l; j := r;
    x := tabela[ (l+r)div 2 ];
    repeat
        while tabela[i].tam > x.tam do

```

```

        i := i+1;
    while tabela[j].tam < x.tam do
        j := j-1;
    if i <= j then begin
        w := tabela[i];
        tabela[i] := tabela[j];
        tabela[j] := w;
        i := i+1;
        j := j-1;
    end;
until i > j;
if l < j then
    SORT (l,j);
if i < r then
    SORT (i,r);
end;

procedure estimativa;
var i,j,fim:integer;
    a : array[1..MAX] of real;
begin
    est:=0; fim:=N_canais;
    for i:=1 to MAX do a[i]:=0;
    for i:=1 to N_canais do a[i]:=tabela[i].tam;
    while a[2]>0 do begin
        i:=1;
        while i< fim do begin
            a[i]:=a[i] * a[i+1];
            est:=est + a[i] * ln(a[i+1])/ln(2);
            for j:=i+1 to fim do a[j]:=a[j+1];
            i:=i+1;
            fim:=fim-1;
        end;
    end;
end;
end;

```

```

(*2*****
(*2**      Func.      MONTA_TABELA  - Le a rede do arquivo de entrada **)
(*2*  e monta as tabelas de canais, ordena a tabela, verifica via- **)
(*2*  bilidade inicial do problema, e reduz o n de tabelas, descon-**)
(*2*  siderando as que so tem uma capacidade possivel.                **)
(*2*      E chamado pelo Programa Principal.                          **)

function Monta_tabela : boolean;
var i,j,k :integer;
    p : apont;
    fluxo,cust : real;

begin
    clrscr;
    writeln;writeln;
    writeln('          Designador de Capacidades  -  Versao Discreta');
    writeln;writeln;
    write(' Arquivo entrada : ');          { leitura }
    readln(arq_ent);
    writeln;
    arq_ent := arq_ent+'.dat';
    assign(entrada,arq_ent);
    {$i-}
    reset(entrada);
    {$i+}
    if IOresult <> 0 then begin
        writeln (' Erro, ',arq_ent,' nao esta no drive ');
        delay(2000);
        halt
    end;

    SETTIME(0,0,0,0);
    {$i-}
    readln(entrada,Gama,T_max,alfa);
    {$i+}
    if IOresult <> 0 then begin
        writeln ('Erro, Incoerencia no formato do arquivo na Linha 1');

```



```

        delay(2000);
        halt
    end;

    GTmax := gama * T_max;
    {$i-}
    readln(entrada,N_canais);
    {$i+}
    if IOresult <> 0 then begin
        writeln ('Erro, Incoerencia no formato do arquivo na Linha 2');
        delay(2000);
        halt
    end;

    for i:=1 to N_canais do begin
        {$i-}
        readln(entrada,fluxo,cust);
        {$i+}
        if IOresult <> 0 then begin
            writeln (' Erro, Incoerencia no formato do arquivo na'
                    , 'Linha ',i+2);
            delay(2000);
            halt
        end;
        if Fluxo > Capacidade[MAX_CAPS] then begin
            writeln ('Erro,Nao existe capacidade disponivel para fluxo'
                    , 'definido na Linha ',i+2,'do arquivo');
            delay(2000);
            halt
        end;

        f[i] := fluxo; d[i] := cust;
        for j:=MAX_CAPS downto 1 do { monta tabela para canal i }
            if Capacidade[j] > fluxo then begin
                tabela[i].tam := tabela[i].tam + 1;
                ALOCA_NO(p);
                with p^ do begin
                    ret := fluxo/(capacidade[j] - fluxo);
                    custo := cust * exp(alfa * ln(capacidade[j])) );
                end;
            end;
        end;
    end;

```

```

        caps[j] := [i];
        prox := tabela[i].link;
    end;
    tabela[i].link := p;
end;
end;

SOL_DISCRETA_VIAVEL;

cust := 0;
SORT(1,N_canaais);                                { ordena }
if tabela[N_Canaais].tam = 0 then
    monta_tabela := false
else begin

    ESTIMATIVA;

    while (tabela[n_canaais].tam = 1) and (N_canaais > 0) do begin
        GTmax := GTmax - tabela[n_canaais].link^.ret;
        cust := cust + tabela[n_canaais].link^.custo;
        n_canaais := n_canaais -1;
    end;

    if aprox_ini then CUSTO_BOUND := CUSTO_BOUND - cust;

    Monta_Tabela := (GTmax > 0) and (aprox_ini);
end;
close(entrada);
end;

(*2*****
(*2**   Proc.   INICIA - Inicializa variaveis do programa.   **)
(*2*     E chamado pelo Programa Principal.                  **)

procedure INICIA;
var i:integer;
begin
    No_Livre := NIL;

```

```

    capacidade[1] := 300;
    capacidade[2] := 1200;
    capacidade[3] := 2400;
    capacidade[4] := 4800;
    capacidade[5] := 9600;
    for i:=1 to MAX do begin
        tabela[i].tam:=0;
        tabela[i].link:=NIL;
    end;
    somas :=0; comps :=0;
    aprox_ini := false;
    custo_bound := 1E38;
end;

(*2*****
(*2*      Proc.  IMPRIME_SOL - Imprime a solucao otima do problema **)
(*2*      E chamado pelo proc. Sol_0tima .                               **)

procedure Imprime_sol;
begin
    for i:=1 to N_canais do
        for j:=1 to MAX_Caps do
            if i IN sol^.caps[j] then
                writeln('canal ',i,' com cap. = ',capacidade[j]);
            writeln;
            writeln(' custo = ',sol^.custo);
            writeln(' ret   = ',sol^.ret);
            writeln(' somas = ',somas);
            writeln(' comps = ',comps);
        end;
    end;

(*2*****
(*2*      Proc.  IMPRIME_SOL_2 - Imprime a solucao otima do problema **)
(*2*      E chamado pelo proc. Sol_0tima .                               **)

procedure Imprime_sol_2;
var i:integer;

```

```

begin
  for i:=1 to n_canais do
    writeln('canal ',i,' com cap. = ',vet_sol[i].capac:4:0);
  writeln;
  writeln(' custo = ',sol^.custo);
  writeln(' ret   = ',sol^.ret);
  writeln(' somas = ',somas);
  writeln(' comps = ',comps);
end;

(*2*****
(*2**      Proc.  SOL_OTIMA   -  Procura a solucao otima na tabela  **)
(*2*  resultante do MERGE total ( que e a ultima ), e incorpora a  **)
(*2*  solucao tabelas deixadas de lado por so terem uma unica capa- **)
(*2*  cidade disponivel.                                           **)
(*2*      E chamado pelo Programa Principal .                      **)

procedure Sol_Otima;
var i,j : integer;
begin
  if Tabela[1].tam > 0 then begin
    sol:=tabela[1].link;
    i:=2;
    while tabela[i].tam = 1 do begin
      sol^.custo := sol^.custo + tabela[i].link^.custo;
      sol^.ret := sol^.ret + tabela[i].link^.ret;
      for j:=1 to MAX_CAPS do
        sol^.caps[j]:=sol^.caps[j]+tabela[i].link^.caps[j];
      i:=i+1;
      n_canais := n_canais +1;
    end;
    Imprime_Sol;
  end
  else if aprox_ini then begin
    ALOCA_NO(sol);
    sol^.ret := 0;
    sol^.custo := custo_bound;
    i:=2;
    while tabela[i].tam = 1 do begin

```

```

        sol^.custo := sol^.custo + tabela[i].link^.custo;
        i:=i+1;
        n_canaais := n_canaais +1;
    end;
    for i:=1 to n_canaais do
        sol^.ret := sol^.ret + vet_sol[i].ret;
    Imprime_Sol_2;
    end
    else
        writeln(' Problema nao tem solucao viavel ! ');
end;

(*****
Programa      Principal      *****)
begin
    Inicia;
    if Monta_tabela then begin
        cont := N_canaais;
        {***-- Div & Conq ----**}
        while Tabela[2].Tam > 1 do begin      { Mergeia as Tabelas }
            i := 0;                          { pelo DEC - Div.& Conq.}
            while (Tabela[i+1].link <> NIL) and (i < cont-1) do begin
                i := i + 1;
                DEC ( i , i+1 );
                cont := cont - 1;
            end;
        end;
        Sol_Otima;
    end
    else
        writeln(' Problema nao tem solucao viavel ! ');

    writeln;
    writeln('razao : ',(comps/est)*100:13:11,' %');
    writeln;
    gettime(h,min,s,ds);

```

```
writeln('Tempo de execucao ',min,':',s,':',ds);  
end.  
æ
```


Apêndice B

Implementação do CAD pelo A*

```
(*1*****  
(*1          DESIGNACAO de  CAPACIDADES  DISCRETAS          *)  
(*1*****  
(*1          Recebe uma rede com topologia , roteamento e fluxos nos      *)  
(*1 canais definidos, e devolve uma capacidade para cada canal, com      *)  
(*1 custo mínimo possível, sujeita a um conjunto de capacidades com      *)  
(*1 valores discretos disponiveis.                                       *)  
(*1*****  
(*1*****  
(*1          TABELA e um vetor de tabelas, uma para canal, e no decorrer *)  
(*1 do programa, todos os elementos em uma tabela estaraõ ordenados      *)  
(*1 por ordem crescente de retardo, logo por ordem decrescente de      *)  
(*1 custos, e terao como componentes conjuntos de canais, cada conj. *)  
(*1 representando uma capacidade disponivel, onde se determinara qual *)  
(*1 capacidade sera atribuida a cada canal.                               *)  
(*1*****  
  
(*1*****  
*****                                     *****  
*          TABELAS ORDENADAS EM ORDEM CRESCENTE DE CUSTO          *  
*          ESTRATEGIA : A*      (com um look ahead)                *  
*****                                     *****
```

```

*****)
program CA_d_Astar;
uses crt,dos,printer;
const MAX      = 50;
      MAX_CAPS = 5;
      EPSILON  = 10E-5;
type
  apont  = ^no;
  conj   = set of 1..max;
  vetconj = array [1..MAX_CAPS] of conj;
  No = record
    ret ,
    custo,
    cstsubs : real;
    nivel   : integer;
    caps    : vetconj;
    prox    : apont;
  end;
  Item_tabela = record
    link      : apont;
    tam,arco  : integer;
  end;
  Item_aprox  = record
    capac, ret : real;
  end;

var
  capacidade : array[1..MAX_CAPS] of integer; { vetor de possiveis
                                                capacidades }
  GTmax,                                           { gama * Tmax }
  alfa, custo_bound,
  gama,T_max : real;
  Sol,                                           { solucao }
  No_livre,                                     { cabeca da lista livre }
  L_abertos : apont;                           { lista de abertos }
  N_canaais : integer;                         { n de canais }
  somas,                                         { n de somas }
  comps : longint;                             { n de comparacoes }

```

```

tabela      : array[1..MAX] of Item_tabela;
entrada     : text;                      { arquivo de entrada }
Arq_Ent     : string[12];                { string para leitura }
h,min,s,ds  : word;                      { conta tempo }
aux,d,dl,f,
vet_sol2    : array [1..MAX] of real;
vet_sol     : array [1..MAX] of Item_aprox;
aprox_ini   : boolean;
uso         : array[1..MAX] of boolean;
contcort    : longint;

(*2*****
(*2      Proc.  Sol_Inicial :                      *)
(*2      Proc. que da um chute inicial para CA suj. a Retardo                      *)
(*2      repartindo os retardos proporcionalmente para cada canal, e ob-          *)
(*      tendo capacidades iniciais continuas.                                  *)
(*2      E chamado pelo proc.  RESOLVE.                                          *)

procedure Sol_Inicial(no:apont);
var GT,soma:real;
    i : integer;
begin
    soma:=0;
    if no=NIL then begin
        for i:=1 to N_canaís do
            { distribui Gama*Tmax por
              cada capacidade }

            soma:=soma+sqrt(d[i]*f[i]);
        for i := 1 to N_canaís do begin
            aux[i] := alfa*d[i];          { 'aux' calculado no inicio e
                                           so uma vez }

            vet_sol[i].capac := f[i]*( 1+1/(GTmax*sqrt(d[i]*f[i]))/soma );
        end;
    end
    else begin
        GT:=GTmax-no^.ret;
        for i:=1 to N_canaís do
            { distribui Gama*Tmax
              por cada capacidade }

            if (tabela[i].arco in no^.caps[1]) then
                uso[i]:=true

```

```

else if (tabela[i].arco in no^.caps[2]) then
  uso[i]:=true
else if (tabela[i].arco in no^.caps[3]) then
  uso[i]:=true
else if (tabela[i].arco in no^.caps[4]) then
  uso[i]:=true
else if (tabela[i].arco in no^.caps[5]) then
  uso[i]:=true
else begin
  soma:=soma+sqrt(d[i]*f[i]);
  uso[i]:=false;
end;
for i := 1 to N_canais do
  if not uso[i] then begin
    aux[i] := alfa*d[i];           { 'aux' calculado no inicio
                                   e so uma vez }
    vet_sol2[i] := f[i]*( 1+1/(GT*sqrt(d[i]*f[i])/soma) );
  end
end
end;

```

```

(*2*****
(*2          Proc.      Lineariza :                               *)
(*2          Proc. que devolve custos linearizados                *)
(*2  Calcula a derivada da formula fechada para custos, obtendo cus- *)
(*2  tos linearizados para resolucao do problema, com capacidades  *)
(*2  continuas.                                                    *)
(*2          E chamado pelo proc.  RESOLVE.                        *)

```

```

procedure Lineariza(no:apont);
var i:integer;
begin
  if no=NIL then begin           { obtidos pela derivada dos valores }
    for i := 1 to N_canais do    { iniciais do custo }
      dl[i] := aux[i] * exp ( (alfa-1)*ln( vet_sol[i].capac ) )
    end
  else begin

```

```

    for i := 1 to N_canais do
        if not uso[i] then
            dl[i] := aux[i] * exp ( (alfa-1)*ln( vet_sol2[i] ) )
        end
    end;
end;

```

```

(*2*****
(*2          Proc.          Resolve :                      *)
(*2          Proc. que resolve CA sujeito a Retardo        *)
(*2  Calcula capac. otima do problema linearizado usando a formula *)
(*2 fechada, e reaplica esta solucao iterativamente no problema nao *)
(*2 linear. A sol. final e de capac. continuas.             *)
(*2          E chamado pelo proc. SOL_DISCRETA_VIAVEL.      *)

```

```

procedure Resolve(no:apont);
var va,aux,beta,TM : real;
    i : integer;
begin
    if no=NIL then begin
        custo_bound := 0;
        Sol_Inicial(no);      { pega um chute inicial }
        repeat
            va := custo_bound;      { va = valor anterior do custo }
            custo_bound := 0;
            Lineariza(no);          { obtem custos linearizados }
            aux := 0;
            for i := 1 to n_canais do
                aux := aux + sqrt( f[i]*dl[i] );  { calculos... }
            aux := aux/T_Max ;
            beta := aux*aux/gama;
            aux := aux/gama;
            for i := 1 to n_canais do begin
                vet_sol[i].capac := f[i] + aux*sqrt(f[i]/dl[i]);
                vet_sol[i].ret := f[i] / ( vet_sol[i].capac - f[i]);
                custo_bound := custo_bound+d[i]*exp(alfa*ln(vet_sol[i].capac));
            end;
        until ( abs(va - custo_bound)/custo_bound ) <= EPSILON; { diferenca

```

```

                                                    relativa }
end                                                    { como condicao de parada }
else begin
  TM:=GTMax - no^.ret;
  no^.cstsubs := no^.custo;
  Sol_Inicial(no);      { pega um chute inicial }
  repeat
    va := no^.cstsubs;          { va = valor anterior do custo }
    no^.cstsubs:= no^.custo;
    Lineariza(no);             { obtem custos linearizados }
    aux := 0;
    for i := 1 to n_canais do
      if not uso[i] then
        aux := aux + sqrt( f[i]*dl[i] );  { calculos... }
      aux := aux/TM ;
    {
      beta := aux*aux/gama;
      aux := aux/gama;    }

    for i := 1 to n_canais do
      if not uso[i] then begin
        vet_sol2[i] := f[i] + aux*sqrt(f[i]/dl[i]);
        no^.cstsubs := no^.cstsubs + d[i]*exp(alfa*ln(vet_sol2[i]));
      end;
    until ( abs(va - no^.cstsubs)/ no^.cstsubs ) <= EPSILON; {diferenca
                                                                relativa }

  end
end;

(*2*****
(*2*          Proc. SOL_DISCRETA_VIAVEL - resolve o problema para *)
(*2*  capacidades continuas e depois obtem uma solucao viavel discre-*)
(*2*  ta a partir desta. *)
(*2*          E chamado pelo  proc. MONTA_TABELA *)

procedure Sol_discreta_viavel;
var GT, retar : real;
    i,j,k : integer;

```



```

    sobe : boolean;
    desce : array [ 1 .. MAX ] of integer;
begin
    Resolve(NIL);
    k := 0;
    for i:=1 to N_canais do
        if capacidade[MAX_CAPS] < vet_sol[i].capac then begin
            k := k+1;
            desce[k] := i;
        end;
    GT := GTmax;
    if k < MAX then begin
        for i := 1 to k do begin
            vet_sol[desce[i]].capac := Capacidade[MAX_CAPS];
            GT := GT - vet_sol[desce[i]].ret;
            Vet_sol[desce[i]].ret:= f[desce[i]] / ( capacidade[MAX_CAPS]
- f[desce[i]]);
            GT := GT + vet_sol[desce[i]].ret;
        end;

        aprox_ini := true;

        while aprox_ini and ( (GT > GTMax) or (k=0) ) do begin
            for i:=1 to N_canais do
                if vet_sol[i].capac < capacidade[MAX_CAPS] then begin
                    j := 1;
                    while capacidade[j] <= Vet_sol[i].capac do j:=j+1;
                    retar := f[i] / ( capacidade[j-1] - f[i]);
                    GT := GT - vet_sol[i].ret;
                    if (capacidade[j-1] > f[i]) and (GTmax-GT-retar>=0)
                    then
                        j:=j-1;
                    retar := f[i] / (capacidade[j] - f[i]);
                    vet_sol[i].capac := capacidade[j];
                    GT := GT + retar;
                    vet_sol[i].ret := retar;

                end;
            j:=1; k:=1;
        end;
    end;
end;

```

```

        while (j <= N_canaís ) and (aprox_ini) do begin
            aprox_ini:=capacidade[MAX_CAPS]=vet_sol[j].capac;
            j:=j+1;
        end;
        aprox_ini := j < N_canaís
    end;

    custo_bound := 0;
    for i:=1 to N_canaís do
        custo_bound := custo_bound +
            d[i]*exp( alfa*ln( vet_sol[i].capac ) );
        aprox_ini := GT <= GTMax;
    end

    else aprox_ini := false;
end;

(*2*****
(*2***  Proc.  ALOCA_NO  -  aloca um no para tabela de canais  ***)
(*2*   Mantem  uma  lista de celulas livres para alocação  **)

procedure aloca_no(var P : apont);
var j:integer;
begin
    if No_livre = NIL then
        NEW(p)
    else begin
        P:=No_Livre;
        No_Livre := No_Livre^.prox;
    end;
    with p^ do begin
        { "zera" o no }
        ret := 0; custo := 0; cstsubs:=0; nivel:=0;
        for j:=1 to MAX_CAPS do caps[j] := [];
        prox := NIL;
    end;
end;
end;

```

```

(*2*****
(*2**      Proc.   DEVOLVE_NO  -  deixa um no que nao e mais usado  **)
(*2*   livre para posterior utilizacao, usando lista livre.          **)
procedure Devolve_No(var P : apont);
var j:integer;
begin
    P^.prox := No_Livre;
    No_livre := P;
    contcort:=contcort+1;
end;

(*2*****
(*2**      Proc.   LIBERA  -  Libera todo o espaco ocupado por uma  **)
(*2*   tabela de canais, devolvendo todos os nos por ela utilizados  **)
(*2*   para lista livre.                                              **)
procedure Libera( p: apont );
var q:apont;
begin
    while p <> NIL do begin
        q:=p;
        p := p^.prox;
        Devolve_No ( q );
    end;
end;

(*2*****
(*2**      Func.   ELIMINA  -  Determina se dado no'   gerado      **)
(*2*   no MERGE deve ou nao ser descartado, verificando-se Bounds de **)
(*2*   programacao.                                                  **)
(*2*   E chamado pelo proc. MERGE.                                    **)
function Elimina(t,c,GamaT:real):boolean;
begin
    Elimina := (t > GamaT) or (c > custo_bound);
end;

(*2*****
(*2**      Proc.   ELIMINA_INEFICIENTES  -  Dada uma tabela de  **)

```

```

(*2* canais, retira todos os "ineficientes" desta tabela.      **)
procedure Elimina_Ineficientes;
var p,q : apont;
begin
    p:=L_Abertos;
    q:=p;
    if p <> NIL then
        p:=p^.prox;
    while p <> NIL do begin
        if (p^.ret > q^.ret) and (p^.custo > q^.custo)
            and (p^.nivel=q^.nivel) then begin
            q^.prox := p^.prox;
            Devolve_No ( p );
        end
        else
            q:=q^.prox;
            p:=q^.prox;
        end;
    end;
end;

(*2*****
(*2**      Proc.   POE_NA_LISTA - Poe um no na lista de abertos  **)
procedure Poe_na_Lista(no:apont);
var p,q:apont;
begin
    if L_Abertos=NIL then
        L_Abertos:=no
    else begin
        p:=L_Abertos;
        q:=p;
        while (p^.cstsubs<no^.cstsubs)and(p<>NIL)and(q^.nivel<N_Canais) do
            begin
                q:=p;  p:=p^.prox; comps:=comps+1;
            end;
        if (q^.nivel < N_canais) then begin
            if p=q then begin
                no^.prox:=L_Abertos;
                L_Abertos:=no;
            end
        end
    end
end

```

```

else begin
    q^.prox:=no;
    no^.prox:=p;
end;
if no^.nivel=N_canaís then begin {apaga todos os abertos piores}
    p:=no^.prox;           {que um que chegou no ultimo nivel}
    no^.prox:=NIL;
    Libera(p);
    custo_bound:=no^.custo;
end
end
end
end;

(*2*****
(*2**      Proc.   TIRA_DA_LISTA - Tira um no da lista de abertos  **)
function Tira_da_Lista(var no:apont):boolean;
begin
{
    no:=L_Abertos;
    while no<>NIL do begin
        writeln(no^.nivel,' ',no^.cstsubs);
        no:=no^.prox;
    end; writeln;
    delay(1000);
}

    no:=L_Abertos;
    if no<>NIL then
        L_Abertos:=no^.prox;
        Tira_da_Lista := (no<>NIL);
end;

(*2*****
(*2**      Proc.   Look_Ahead - Olha adiante um nivel de busca  **)
procedure Look_Ahead(p:apont);
var t,q:apont;
    i:integer;
begin

```

```

q:=Tabela[1].link;

while q<>NIL do begin
    Aloca_No(t);

    t^.ret := p^.ret + q^.ret;
    t^.custo := p^.custo + q^.custo;
    for i:=1 to MAX_CAPS do
        t^.caps[i] := p^.caps[i] + q^.caps[i];
    t^.cstsubs:=0;
    t^.nivel:=p^.nivel+1;
    t^.prox:=NIL;
    somas:=somas+1;

    if t^.ret < GTMax then begin
        Resolve(t);
        if not ELIMINA(t^.ret,t^.cstsubs,GTMax) then
            Poe_na_Lista(t)
        else
            Devolve_No(t);
    end
    else
        Devolve_no(t);

    q:=q^.prox;
end;
end;

(*2*****
(*2**      Proc.      BUSCA_SOL - Procura a solucao do CAD usando o      **)
(*2*  algor. A* .      **)
procedure BUSCA_SOL;
var p,q,t:apont;
    i:integer;
    acabou:boolean;
begin
    p:=Tabela[n_canaais].link;
    acabou:=false;

```



```

while p<>NIL do begin
    Aloca_No(q);

    q^.custo:=p^.custo;
    q^.ret:=p^.ret;
    q^.cstsubs:=p^.cstsubs;
    q^.nivel:=p^.nivel;
    for i:=1 to max_caps do
        q^.caps[i]:=p^.caps[i];
    q^.prox:=NIL;

    Resolve(q);

    if not ELIMINA(q^.ret,q^.cstsubs,GTMax) then
        Poe_na_Lista(q)
    else
        Devolve_No(q);
    p:=p^.prox;
end;
repeat

    ELIMINA_INEFICIENTES;

    if Tira_da_Lista(p) then begin

        writeln('n. ',p^.nivel);

        if p^.nivel < n_canaais then begin

{           q:=Tabela[p^.nivel + 1].link;           }
           q:=Tabela[n_canaais - p^.nivel].link;

        while q<>NIL do begin
            Aloca_No(t);

            t^.ret := p^.ret + q^.ret;
            t^.custo := p^.custo + q^.custo;
            for i:=1 to MAX_CAPS do

```

```

        t^.caps[i] := p^.caps[i] + q^.caps[i];
t^.cstsubs:=0;
t^.nivel:=p^.nivel+1;
t^.prox:=NIL;
somas:=somas+1;

if t^.ret < GTMax then begin
    Resolve(t);
    if not ELIMINA(t^.ret,t^.cstsubs,GTMax) then
        if t^.nivel < n_canais-1 then
            Poe_na_Lista(t)
        else
            Look_Ahead(t)
        else
            Devolve_No(t);
    end
else
    Devolve_no(t);
q:=q^.prox;
end;
end
else begin
    acabou:=true;
    p^.prox:=L_Abertos;
    L_Abertos:=p;
end
end
else acabou:=true;
until (acabou);
end;

```

```

(*2*****
(*2**      Proc.   SORT   -   Ordena as tabelas de canais por ordem **)
(*2*  decrescente de tamanho.                                          **)
(*2*      Versao recursiva do Quick-Sort.                             **)
(*2*      E chamado pelo Monta_Tabela.                                **)

```

```

procedure SORT(l,r : integer);
var i,j : integer;
    x,w : Item_tabela;
    k    : real;
begin
    i := l; j := r;
    x := tabela[ (l+r)div 2 ];
    repeat
        while tabela[i].tam > x.tam do
            i := i+1;
        while tabela[j].tam < x.tam do
            j := j-1;
        if i <= j then begin
            w := tabela[i];
            tabela[i] := tabela[j];
            tabela[j] := w;
            k := d[i];
            d[i] := d[j];
            d[j] := k;
            k := f[i];
            f[i] := f[j];
            f[j] := k;
            i := i+1;
            j := j-1;
        end;
    until i > j;
    if l < j then
        SORT (l,j);
    if i < r then
        SORT (i,r);
end;

```

```

(*2*****
(*2**      Func.      MONTA_TABELA - Le a rede do arquivo de entrada **)
(*2*  e monta as tabelas de canais, ordena a tabela, verifica via- **)
(*2*  bilidade inicial do problema, e reduz o n de tabelas, descon-**)
(*2*  siderando as que so tem uma capacidade possivel.                **)
(*2*      E chamado pelo Programa Principal.                          **)

```

```

function Monta_tabela : boolean;
var i,j,k :integer;
    p : apont;
    fluxo,cust : real;

begin
    clrscr;
    writeln;writeln;
    writeln('Designador de Capacidades - Versao Discreta com A*');
    writeln;writeln;
    write(' Arquivo entrada : ');          { leitura }
    readln(arq_ent);
    writeln;
    arq_ent := arq_ent+'.dat';
    assign(entrada,arq_ent);
    {$i-}
    reset(entrada);
    {$i+}
    if IOresult <> 0 then begin
        writeln (' Erro, ',arq_ent,' nao esta no drive ');
        delay(2000);
        halt
    end;

    SETTIME(0,0,0,0);
    {$i-}
    readln(entrada,Gama,T_max,alfa);
    {$i+}
    if IOresult <> 0 then begin
        writeln('Erro, Incoerencia no formato do arquivo na Linha 1');
        delay(2000);
        halt
    end;

    GTmax := gama * T_max;
    {$i-}
    readln(entrada,N_canais);
    {$i+}

```

```

if IOresult <> 0 then begin
    writeln('Erro, Incoerencia no formato do arquivo na Linha 2');
    delay(2000);
    halt
end;

for i:=1 to N_canaais do begin
    {$i-}
    readln(entrada,fluxo,cust);
    {$i+}
    if IOresult <> 0 then begin
        writeln (' Erro, Incoerencia no formato do arquivo na'
            , 'Linha ',i+2);
        delay(2000);
        halt
    end;
    if Fluxo > Capacidade[MAX_CAPS] then begin
        writeln('Erro,Nao existe capacidade disponivel para fluxo'
            , 'definido na Linha ',i+2,'do arquivo');
        delay(2000);
        halt
    end;
    Tabela[i].arco:=i;
    f[i] := fluxo; d[i] := cust;
    for j:=MAX_CAPS downto 1 do { monta tabela para canal i }
        if Capacidade[j] > fluxo then begin
            tabela[i].tam := tabela[i].tam + 1;
            ALOCA_NO(p);
            with p^ do begin
                ret := fluxo/(capacidade[j] - fluxo);
                custo := cust * exp(alfa * ln(capacidade[j])) ;
                caps[j] := [i];
                cstsubs:=0;
                nivel:=1;
                prox := tabela[i].link;
            end;
            tabela[i].link := p;
        end;
    end;
end;
end;

```

```

SOL_DISCRETA_VIAVEL;

cust := 0;
SORT(1,N_canais);                                { ordena }
if tabela[N_Canais].tam = 0 then
    monta_tabela := false
else begin
    while (tabela[n_canais].tam = 1) and (N_canais > 0) do begin
        GTmax := GTmax - tabela[n_canais].link^.ret;
        cust := cust + tabela[n_canais].link^.custo;
        n_canais := n_canais -1;
    end;

    if aprox_ini then CUSTO_BOUND := CUSTO_BOUND - cust;

    Monta_Tabela := (GTMax > 0) and (aprox_ini);
end;
close(entrada);
end;

(*2*****
(*2**   Proc.   INICIA - Inicializa variaveis do programa.      **)
(*2*     E chamado pelo Programa Principal.                    **)

procedure INICIA;
var i:integer;
begin
    No_Livre := NIL;
    L_Abertos:=NIL;
    capacidade[1] := 300;
    capacidade[2] := 1200;
    capacidade[3] := 2400;
    capacidade[4] := 4800;
    capacidade[5] := 9600;
    for i:=1 to MAX do begin
        tabela[i].tam:=0;
        tabela[i].link:=NIL;

```



```

end;
somas :=0; comps :=0;
contcort:=0;
aprox_ini := false;
custo_bound := 1E38;
end;

(*2*****
(*2*      Proc.  IMPRIME_SOL - Imprime a solucao otima do problema **)
(*2*      E chamado pelo proc. Sol_Otima .                               **)

procedure Imprime_sol;
var i,j:integer;
begin
  for i:=1 to N_canais do
    for j:=1 to MAX_Caps do
      if i IN sol^.caps[j] then
        writeln('canal ',i,' com cap. = ',capacidade[j]);

        writeln;
        writeln(' custo = ',sol^.custo);
        writeln(' ret   = ',sol^.ret);
        writeln(' somas = ',somas);
        writeln(' comps = ',comps);
        writeln(' podas = ',contcort);
      end;
    end;

  (*2*****
  (*2*      Proc.  IMPRIME_SOL_2 - Imprime a solucao otima do problema **)
  (*2*      E chamado pelo proc. Sol_Otima .                               **)

  procedure Imprime_sol_2;
  var i:integer;
  begin
    for i:=1 to n_canais do
      writeln('canal ',i,' com cap. = ',vet_sol[i].capac:4:0);
      writeln;
      writeln(' custo = ',sol^.custo);

```

```

        writeln(' ret    = ',sol^.ret);
        writeln(' somas = ',somas);
        writeln(' comps = ',comps);
        writeln(' podas = ',contcort);
end;

(*2*****
(*2**      Proc.  SOL_OTIMA  -  Procura a solucao otima na lista      **)
(*2*  resultante de abertos. O primeiro no da lista final contem a **)
(*2*  solucao otima.                                              **)
(*2*      E chamado pelo Programa Principal .                      **)

procedure Sol_Otima;
var i,j : integer;
begin
    if L_Abertos <> NIL then begin
        sol:=L_abertos;
        while tabela[n_canaais+1].tam = 1 do begin
            sol^.custo:=sol^.custo+tabela[n_canaais+1].link^.custo;
            sol^.ret := sol^.ret + tabela[n_canaais+1].link^.ret;
            for j:=1 to MAX_CAPS do
                sol^.caps[j] := sol^.caps[j] +
                    tabela[n_canaais+1].link^.caps[j];
            n_canaais := n_canaais +1;
        end;
        Imprime_Sol;
    end
    else if aprox_ini then begin
        ALOCA_NO(sol);
        sol^.ret := 0;
        sol^.custo := custo_bound;
        while tabela[n_canaais+1].tam = 1 do begin
            sol^.custo:=sol^.custo+tabela[n_canaais+1].link^.custo;
            n_canaais := n_canaais +1;
        end;
        for i:=1 to n_canaais do
            sol^.ret := sol^.ret + vet_sol[i].ret;
        Imprime_Sol_2;
    end
end

```

```

        else
            writeln(' Problema nao tem solucao viavel ! ');
end;

(*****)
(*****          Programa      Principal          *****)
begin
    Inicia;
    if Monta_tabela then begin
        Busca_Sol;
        Sol_Otima;
    end
    else
        writeln(' Problema nao tem solucao viavel ! ');

        writeln;
        gettime(h,min,s,ds);
        writeln('Tempo de execucao ',min,':',s,':',ds);
end.

æ

```

Bibliografia

- [1] Aho, A.V., Hopcroft, J.E. e Ullmann, J.D. "The Design of Computer Algorithms", Reading, Addison Wesley, 1974.
- [2] Bellman, R. "Dynamic Programming", Princeton University Press, 1957.
- [3] Bellman, R.E. e Dreyfus, S.R. "Applied Dynamic Programming", Princeton University Press, 1962.
- [4] Bezerra, J.R.M. "Sobre Problemas de Otimização em Redes de Computadores", São Paulo, 1984 (Diss. Mestrado) I.M.E.-U.S.P.
- [5] da Cunha, N.O. e Polak, E. "Constrained Minimization under Vector-Valued Criteria in Finite Dimension Spaces", Journal of Mathematical Analysis and Applications, vol 19, 1967.
- [6] Debreu, G. "Theory of Value - An Axiomatic Analysis of Economic Equilibrium", New Heaven and London, Yale University Press, 1959.
- [7] Dreyfus, S. e Law, A. "The Art and Theory of Dynamic Programming", Academic Press, Mathematics in Science and Engineering, vol. 130, 1977.
- [8] Garey, M. e Johnson, D. "Computers and Intractability", W.H. Freeman and Company, New York, 1979.
- [9] Gerla, M. "The Design of Store and Forward Computer Networks", Los Angeles, 1973, Ph.D. Thesis U.C.L.A.
- [10] Gonzaga, C. "Busca de Caminhos em Grafos e Aplicações", I Reunião de Matemática Aplicada / Gávea, IBM-Brasil, 1978.
- [11] Horowitz, E. "Fundamentals of Computer Algorithms", Computer Science Press, 1978.

- [12] Humes Jr, C. "Tópico de Otimização e Redes de Computadores", (Livre Docência) I.M.E.-U.S.P, 1988.
- [13] Humes Jr, C. "Some Results on Outer Linearizations in the Presence of Concavity", Relatório Técnico Mac-I.M.E.-U.S.P, 1990.
- [14] Humes Jr, C., Wasserstein, S.L. e Zanjacomo, P.R. "A Complexity-Oriented Analysis of Dynamic Programming Implementation: Bicriterion Case", Relatório Técnico Mac-I.M.E.-U.S.P, 1990.
- [15] Humes Jr, C. "Problemas de Grande Porte em Programação Matemática", I Reunião de Matemática Aplicada / Gávea, IBM-Brasil, 1978.
- [16] Kaufmann, A. "Graphs, Dynamic Programming and Finite Games", Academic Press, Mathematics in Science and Engineering, vol. 36, 1967.
- [17] Kleinrock, L., "Queueing Systems", New York, John Wiley, 1975-76, 2º vol.
- [18] Lin, J.G. "Maximal Vectors and Multi-objective Optimization", Journal of Optimization Theory and Applications, vol 18, nº 1, 1976.
- [19] Pearl, J. "Heuristics: Intelligent Search Strategies for Computer Problem Solving", Addison-Wesley Publishing, 1984.
- [20] Rich, E. "Inteligência Artificial", McGraw Hill, 1988.
- [21] Salukvadze, M.E. "Vector Valued Optimization Problems in Control Theory", Academic Press, Mathematics in Science and Engineering, vol. 148, 1979.
- [22] Simonard, M. "Linear Programming", Prentice-Hall, 1966.
- [23] Terada, R. "Desenvolvimento de Algoritmos e Estrutura de Dados", McGraw Hill, Makron books, 1991.
- [24] Wagner, H. "Principles of Operations Research", Prentice-Hall, 2ª ed., 1975.