UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE TELEMÁTICA

ABRIL DE 1993

UM SISTEMA DE SUPORTE A DECISÃO BASEADO EM PROGRAMAÇÃO MULTIOBJETIVO

Por : Fábio Alexandre Gaion Casotti

Orientador : Prof. Dr. Paulo Augusto Valente Ferreira OK

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

À minha família
e aos meus amigos.

Agradecimentos

Primeiramente agradeço a minha família pelo apoio e incentivo dado em todos os momentos.

Agradeço também a todos os amigos, professores e funcionários da FEE que ajudaram direta ou indiretamente no desenvolvimento deste trabalho. Em especial ao Prof. Paulo A. V. Ferreira pela orientação dedicada e eficiente além de sua amizade; ao Prof. Akebo Yamakami; aos Prof. Marcius F. H. Carvalho e Oscar S. S. Filho além dos amigos Walcir Fontanini e Carlos A. Fernandes que foram os primeiros a incentivar o meu ingresso na pós-graduação.

Para evitar reclamações futuras, não posso esquecer de certos **amigos** e **amigas**. Por isso gostaria de agradecer:

- Ao Ronald pela sua amizade e por ter ajudado neste trabalho.
- Luiza, Miguel, Isamara, Pedro, Débora, Karin, Menotti, Reginaldo, Celso, Roberto, Alencar, Alexandre, Trabuco, Julimara, Eli, Gustavo e todos os outros frequentadores do laboratório da Telemática que proporcionaram um ambiente de trabalho alegre e saudável.
- Aos amigos de Ibitinga, os quais não citarei nomes por formarem uma lista muito extensa.
- A Ana que dispensa comentários.

Este trabalho de pesquisa contou com o apoio financeiro da CAPES.

Resumo

Este trabalho apresenta um ambiente para (análise) apoio a tomada de decisão de problemas da vida real que possam ser modelados utilizando o enfoque multicritério. Foi realizado um estudo de técnicas de programação multicritério, softwares de programação matemática e sistemas de suporte a decisão. Baseado nesses estudos, elaborou-se um ambiente para programação matemática baseado em programação multiobjetivo. Neste ambiente, problemas da vida real podem ser modelados utilizando o enfoque multicritério. Além das facilidades proporcionadas para a modelagem dos problemas, o ambiente suporta uma série de facilidades para a edição, resolução e verificação dos dados e resultados obtidos. Especial ênfase foi dada no desenvolvimento da interface com o usuário que foi elaborada num ambiente gráfico padrão (Openwindows), permitindo um rápido aprendizado por parte do usuário bem como flexibilidade na modelagem dos problemas.

Abstract

In this work, an overview of multicriteria optimization techniques, mathematical programming software and decision support systems have been carried out. Based on this overview, a general purpose multiobjective mathematical programming environment has been implemented. Through the software developed, real-world problems can be modeled using a multiobjective framework. Besides furnishing facilities for the modeling of the problem, the environment offers a number of tools for the edition, solution and verification of the data and numerical results obtained. Special emphasis has been put in the development of the user interface, which has been elaborated in a standard graphical environment (Openwindows), allowing a quick apprenticenship from the point of view of the user as well as an efficient approach for multiobjective decision problems.

Índice

Introdução Geral

A Programação Multiobjetivo (PMO) apresenta atualmente uma ampla variedade de métodos e técnicas destinados à resolução de problemas que envolvam vários objetivos. Os métodos que têm apresentado melhores resultados para resolução do PMO, decompõem o problema em dois níveis: *Análise* e *Decisão*. O nível de *Análise* ou inferior está relacionado com aspectos numéricos necessários para se obter uma solução ou um conjunto de soluções para o problema. O nível de *Decisão* ou superior está relacionado com os aspectos subjetivos do problema. Neste nível, o Decisor concentra-se na tarefa de induzir o nível de análise a gerar soluções que venham ao encontro de sua estrutura de preferências.

A descoberta e implementação de algoritmos eficientes em Programação Matemática (PM) tomou muito tempo e esforço. O sucesso das pesquisas na área de PM ao lado da evolução constante do hardware computacional permitiu a resolução de problemas grandes e complexos num tempo reduzido. Contudo a resolução destes problemas de PM normalmente exigem dos usuários uma considerável familiaridade com PM, bons conhecimentos computacionais e um período de aprendizado para lidar com o sistema.

Muitos sistemas de PM requerem dos usuários um conhecimento técnico apurado. Isto deve-se a ênfase dada na elaboração e implementação de algoritmos e técnicas eficientes para resolução de modelos. Atualmente este quadro está se alterando com as várias pesquisas na área da análise, interpretação e uso dos modelos.

Os Sistemas de Suporte a Decisão (SSD) podem ser utilizados na elaboração e resolução de PMs com a vantagem de proporcionar um ambiente menos hostil ao usuário, além de possuir várias outras melhorias se comparados a um software de PM. Um exemplo comum de SSD é uma Planilha de Cálculo (Lotus 123, Quatro Pro) que, através de uma interface amigável, permite ao usuário vários níveis de análise de informações.

Um SSD, normalmente é composto de 3 partes:

- Sistema para gerenciamento de interface;
- Sistema para gerenciamento de dados;
- Sistema para gerenciamento de modelos.

O sistema para gerenciamento da interface consiste de todas as interfaces entre o usuário e o sistema tais como a linguagem usada para definir um problema (modelo) e o modo como os resultados são impressos (gráficos, tabelas, relatórios). O gerenciamento de dados envolve a criação, armazenamento, manipulação e recuperação de dados, o que pode ser conseguido através de um

Sistema Gerenciador de Banco de Dados (SGBD) interligado ao SSD. O sistema para gerenciamento de modelos deve funcionar como um SGBD, porém utilizando modelos ao invés de dados.

Um SSD deve ser suficientemente fácil de ser utilizado por usuários que tenham pouco ou nenhuma experiência em computadores e em programação matemática.

O objetivo de longo prazo do projeto de pesquisa iniciado com esta tese é a implementação de um Sistema de Suporte a Decisão baseado em Programação Multiobjetivo através do uso conjugado de diversas metodologias como engenharia de software, linguagens de programação (inclusive orientação a objetos), análise numérica, gerenciamento de dados e sistemas especialistas.

A estrutura geral do sistema do ponto de vista da hierarquia de programação matemática pode ser vista na Figura i.1.

Implementações intensivas serão realizadas nas camadas 0 e 1, referentes aos aspectos de Decisão e Análise dos métodos multiobjetivos. Uma estratégia de máxima reutilização de software de sistemas abertos nas camadas 2 e 3 deve ser adotada.

O sistema deverá contar com uma linguagem para especificação e definição de problemas de otimização multiobjetivo e para controle de execução de instruções que visem a resolução do problema. Esta linguagem deverá estar associada a recursos gráficos necessários a muitos métodos de programação multiobjetivo.

Baseado nestas premissas foi desenvolvido o software MC++ que tem por objetivo principal auxiliar o usuário na modelagem e na solução de problemas multiobjetivos, fornecendo para isso ferramentas de suporte a decisão, gerenciamento de dados e gerenciamento de modelos.

O conteúdo da tese está dividido em 4 capítulos distribuídos da seguinte maneira.

Capítulo 1 - Programação Multiobjetivo: Uma Análise Baseada no Espaço dos Objetivos

Neste capítulo são descritas as formulações dos Problemas de Programação Multiobjetivo tanto no espaço das variáveis como no espaço dos objetivos. Também é efetuado um estudo sobre a projeção de um PMO no espaço dos objetivos.

Quanto aos métodos de programação multiobjetivo, é realizado um estudo segundo a Classificação Baseada em Estruturas de Preferência. Neste ponto, vários métodos que utilizam tanto a metodologia tradicional baseada no espaço das variáveis de decisão como a metodologia que utiliza o espaço dos objetivos são descritos.

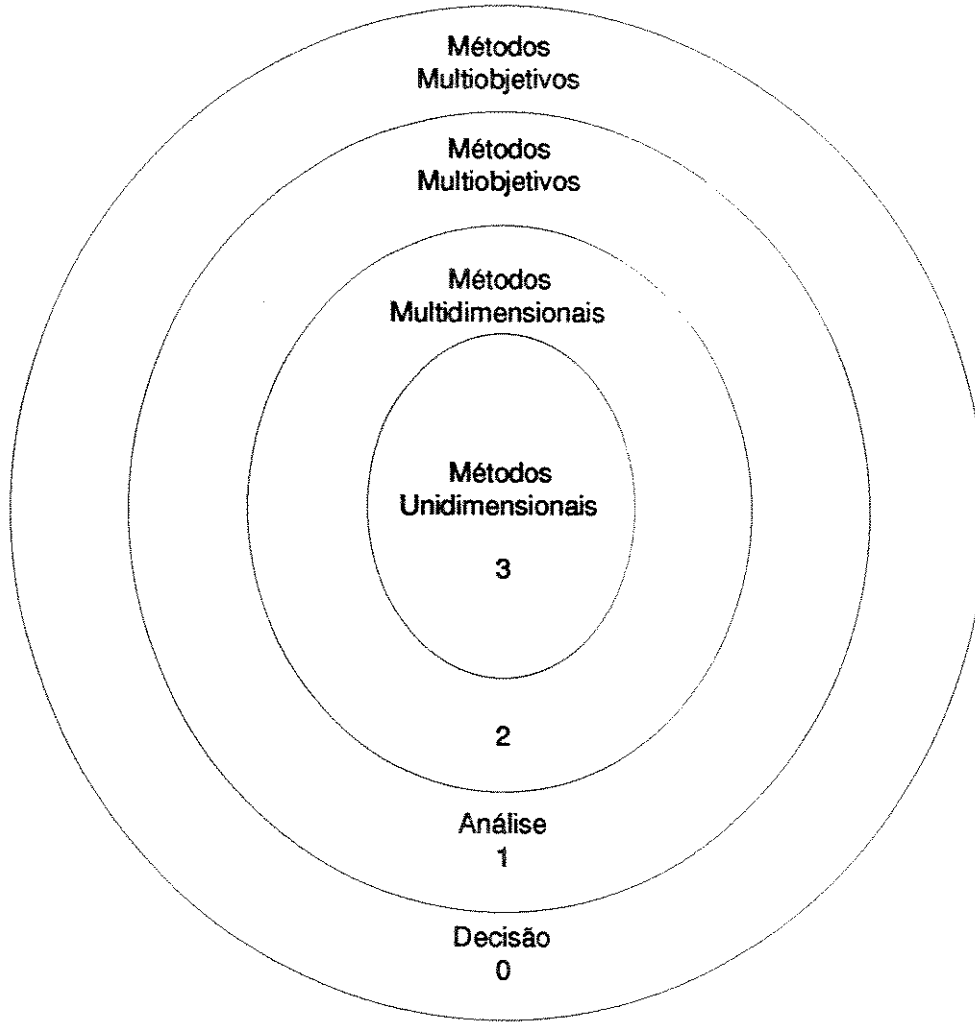


Figura i.1 - Estrutura Geral do Sistema

Capítulo 2 - Softwares para Programação Matemática

Neste capítulo descrevemos as características básicas dos softwares de Programação Matemática e dos Sistemas de Suporte a Decisão. Dedicamos especial ênfase aos Sistemas de Suporte a Decisão, inclusive apresentando um estudo de alguns sistemas existentes.

Capítulo 3 - MC++: Software para Programação Matemática Multiobjetivo

Este capítulo descreve o sistema implementado MC++ fornecendo suas características, principais componentes, estrutura de dados e diagrama hierárquico. Temos aqui uma rica fonte de informações sobre o uso, vantagens e limitações do sistema, sendo um capítulo de considerável importância para o usuário que estiver interessado na sua utilização.

Capítulo 4 - Uma Aplicação MC++ : Problema da Dieta Multiobjetivo

Neste capítulo descrevemos uma aplicação prática através do Problema da Dieta Multiobjetivo, onde é dada a formulação do problema e sua abordagem segundo os métodos multiobjetivos implementados.

O problema é resolvido utilizando o sistema MC++ e os resultados fornecidos pelo software são exibidos e analisados.

Capítulo 1
Programação Multiobjetivo: Uma Análise Baseada no Espaço dos Objetivos

1.1 - Introdução

Processos de decisão usualmente envolvem a consideração de um certo número de objetivos conflitantes e não comensuráveis. A Programação Multiobjetivo (ou Multicritério - PMO) fornece uma base formal para a abordagem destes processos através de técnicas de programação matemática. De acordo com a Programação Multiobjetivo, o problema de decisão é modelado como um problema de otimização vetorial, isto é, envolvendo várias funções objetivos simultâneas que devem, em algum sentido, ser por exemplo minimizadas.

A diferença essencial entre a programação multiobjetivo e os demais ramos da programação matemática está no conceito dado a *solução* do problema. Como a PMO envolve a otimização de um vetor de funções escalares, temos que o espaço das soluções de problemas desta natureza é parcialmente ordenado e portanto existem soluções que não obedecem nenhuma relação de ordem do tipo $\leq$..Com isso o conceito de solução ótima fica totalmente descaracterizado e, em princípio, qualquer elemento do conjunto de soluções eficientes ou dominantes que satisfaça as condições de equilíbrio de Pareto é candidato a solução do PMO. Como nenhuma destas soluções pode ser considerada ótima com respeito a qualquer outra, torna-se evidente a necessidade de critérios (subjetivos) adicionais para se chegar à solução final do problema. Estes critérios adicionais são fornecidos por um decisor que através deles procura encontrar uma solução que proporcione um compromisso adequado entre os seus objetivos.

1.2 - Formulações do PMO

Um PMO genérico com n variáveis, m objetivos e p restrições pode ser formulado de duas maneiras equivalentes.

Formulação no Espaço das Variáveis de Decisão

O PMO assume a forma usual

$$\min_{x \in \chi} U(f(x)) \qquad (1)$$

$$\chi = \{ x : g(x) \leq 0 \}$$

onde

$f(x)' = [f_1(x)\ f_2(x)\ \ldots\ f_m(x)]\ (m \geq 2)$

$f(\cdot) : R^n \rightarrow R^m\ ;\ f_i(\cdot) : R^n \rightarrow R^1$

$g(x)' = [\, g_1(x)\ g_2(x)\ \ldots\ g_p(x)\,]\ ;\ g(\cdot) : R^n \rightarrow R^m\ ;\ g_i(\cdot) : R^n \rightarrow R^1$

$\chi \subseteq \mathrm{R}^n$

Os aspectos de decisão estão representados através da função *utilidade* $U(\cdot) : R^m \rightarrow R$ que possui as características abaixo e é mostrada na Figura 1.1.

- implícita;
- não-decrescente com cada objetivo;
- quasi-convexa.

Assume-se apenas que o *Decisor* é capaz de fornecer informações ordinais a respeito de alternativas de solução. Assim sendo, dadas duas alternativas $f^1 = f(x_1)$ e $f^2 = f(x_2)$ temos:

- $U(f^1) > U(f^2)$: o decisor prefere f^2
- $U(f^1) < U(f^2)$: o decisor prefere f^1
- $U(f^1) = U(f^2)$: o decisor é indiferente

Não é necessário quantificar $U(f)$, isto é, fornecer informação cardinal a respeito das alternativas como em teoria de utilidade multiobjetivo.

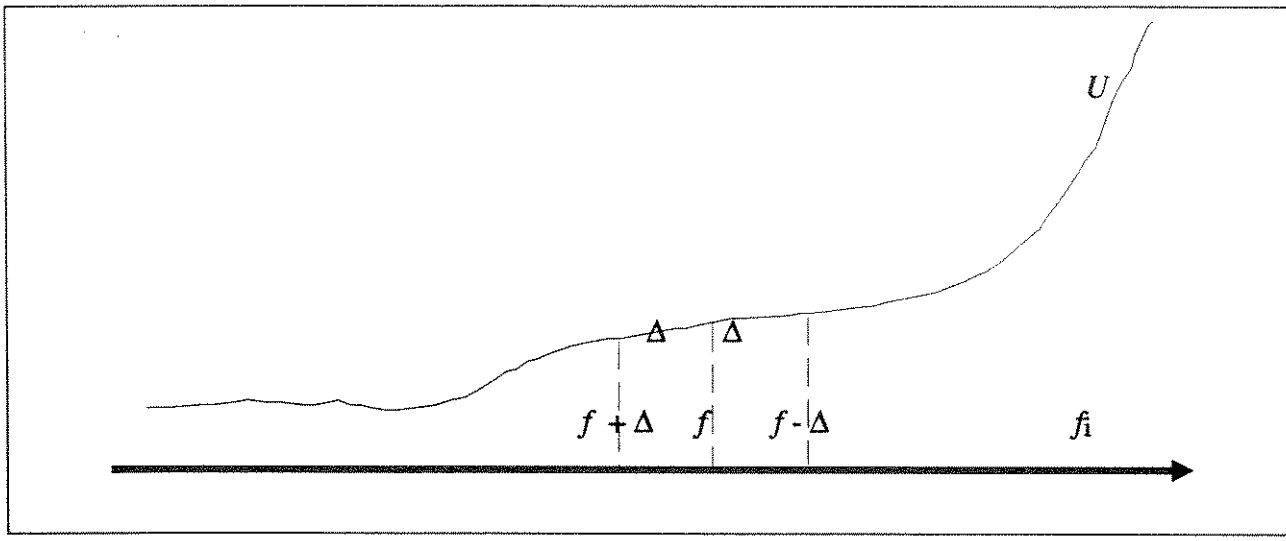


Figura 1.1 - Função Utilidade

Formulação no Espaço dos Objetivos *F*

Definindo-se $F = \{ y : y = f(x), x \in \chi \}$ ou compactamente $F = f(\chi)$, o PMO pode ser formulado como:

$$\min_{y \in F} U(y) \qquad (2)$$

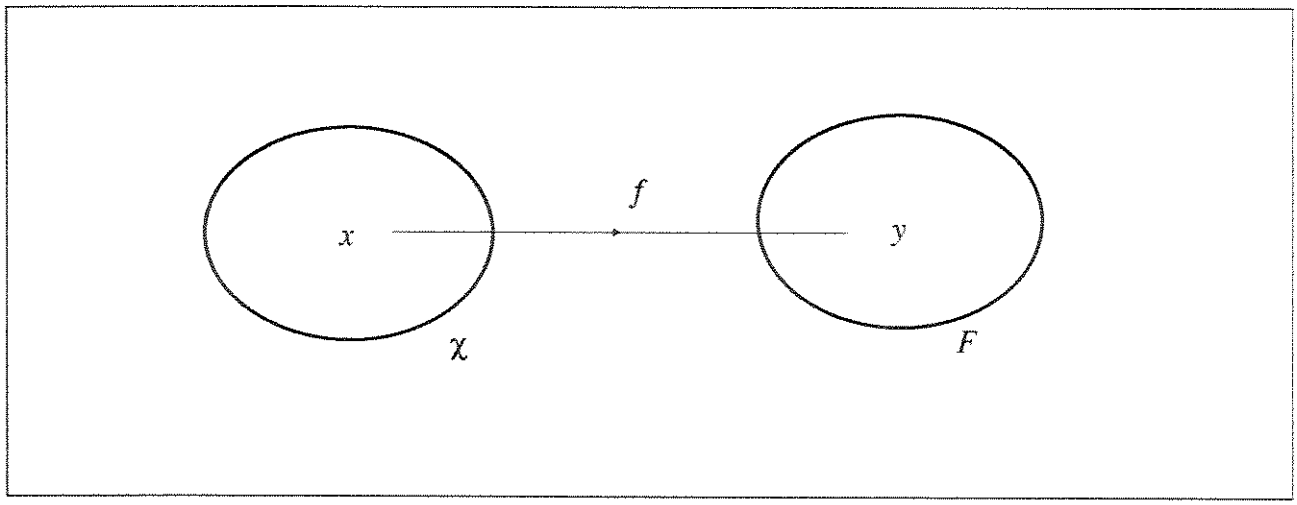


Figura 1.2 - Função no Espaço dos Objetivos

Note que $\chi \subseteq R^n$ e $F \subseteq R^m$ e devido ao fato de, em geral, n >> m, o mapeamento de $F = f(x)$ implica numa drástica redução de dimensionalidade.

Esta formulação possui como problemas o fato de F ser apenas uma abstração e a transformação efetuada não preservar, em geral, a convexidade do problema.

Solução do PMO

A dificuldade básica para a solução de (1) (e portanto de (2)) reside no fato de que $f(\cdot)$ mapeia elementos do R^n no R^m, que por sua vez é um conjunto parcialmente ordenado. Esta característica do problema exclui a possibilidade de obtenção de soluções **"ótimas"** no sentido usual adotado em programação matemática. Substitui-se então o conceito de solução ótima pelo conceito de **solução eficiente**. Uma solução x^0 será eficiente se não existir nenhuma outra decisão factível x $\in \chi$ tal que $f(x) \leq f(x^0)$ e $f(x) \neq f(x^0)$. Como via de regra existem infinitas soluções eficientes associadas ao problema (1), não comparáveis entre si, torna-se necessário adotar critérios adicionais, subjetivos, de maneira a estabelecer uma **solução de compromisso** para (1). Os critérios baseados em Programação Multiobjetivo diferem basicamente quanto aos mecanismos adotados para atingir a solução de compromisso. Cada método estabelece uma forma e nível de interação com um **Decisor**, responsável pela seleção final da solução do problema. Os métodos multiobjetivos mais eficientes operam de acordo com uma estrutura em dois níveis, mostrada abaixo.

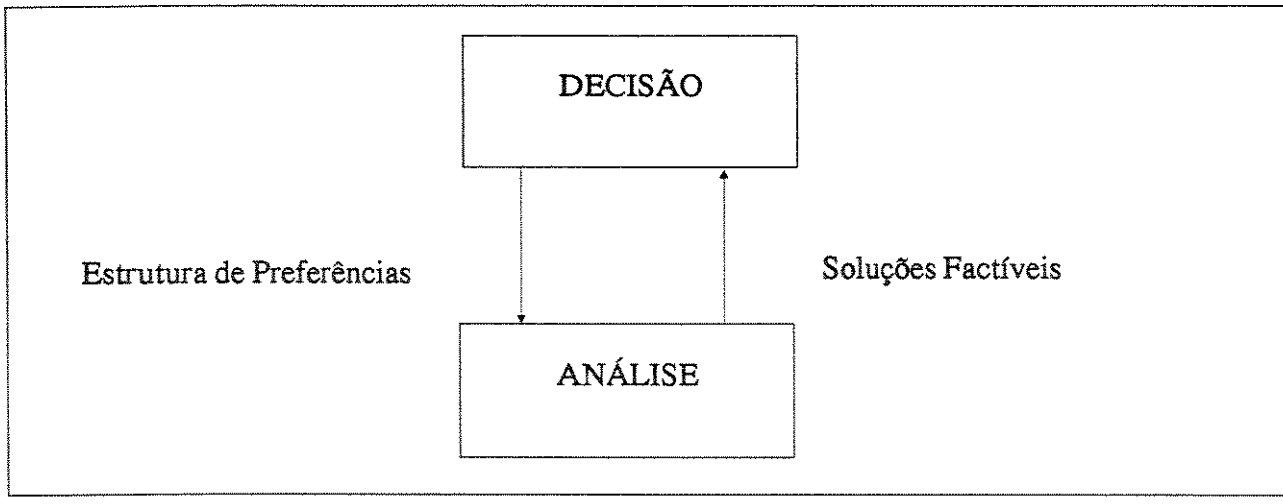


Figura 1.3 - Estrutura em dois níveis

A partir de informações sobre a estrutura de preferências do Decisor indicadas no Nível de Decisão, o Nível de Análise gera soluções que serão avaliadas pelo Decisor. Caso nenhuma destas soluções satisfaça adequadamente as preferências indicadas pelo nível superior, novo ciclo se inicia até a convergência final do processo. Previsivelmente, os métodos multiobjetivos são classificados de acordo com a estrutura de preferências do Decisor (Hwang e Masud, 1979; Ferreira, 1986).

1.3 - Projeção do PMO no Espaço dos Objetivos

Antes de iniciarmos a descrição desta seção, devemos fazer algumas definições formais.

° **Definição 1** - Solução Eficiente (Pareto-Ótima)

Uma solução factível $x^* \in \chi$ é eficiente se não existe qualquer outra solução $x \in \chi$ tal que $f(x) \leq f(x^*)$ e $f(x) \neq f(x^*)$.O conjunto de todas as soluções eficientes será representado por χ^*

Definição 2 - Solução Utópica ou Ideal

A solução ideal $\underline{y}$ do PMO é definida como

$$\underline{y_i} = f_i(x^i), \quad i = 1, \ldots, m$$

onde

$$x^i = \arg \min f_i(x), \; x \in \chi$$

° **Definição 3** - Solução Satisfatória

Um vetor de trade-off's implícitos $y \in R^m$ é dito ser satisfatório se $\exists\, x \in \chi$ tal que $f(x) \leq y$.

° **Definição 4** - Conjunto Γ

Seja $\chi^* \subseteq \chi$ o conjunto de todas as decisões eficientes do PMO, então o conjunto $\Gamma = f(X^*) \subseteq F$ é a imagem de X no espaço dos objetivos

O problema

$$\min_{y \in F} U(y) \tag{3}$$

$$F = \{ y \in R^m / y = f(x), \text{ para algum } x \in \chi \}$$

formulado no espaço dos objetivos apresenta duas características importantes:

- objetivos lineares;
- número de variáveis (objetivos) pequeno - m < n.

A solução de (3) pode ser caracterizada através do conceito de solução *satisfatória.*

Se $\hat{y}$ é satisfatória então

$$U^* \leq U(f(x)) \leq U(\hat{y})$$

O conjunto de todas as soluções satisfatórias do PMO pode ser imaginado como a projeção de $f(x) - \hat{y} \leq 0, x \in \chi$ no R^m.

$$V = \{ y \in R^m / f(x) \leq y \text{ , para algum } x \in \chi \}$$

Resolver o PMO é equivalente agora à resolver

$$\min_{y \in Y^0 \cap V} U(y)$$

onde

$$Y^0 = \{ y \in R^m / y \geq \underline{y} \}$$

e o conjunto V pode ser explicitado a partir do seguinte Teorema:

Teorema da Representação (Lasdon, 1970)

"Sejam $f_i(\cdot)$, $i = 1, 2, \ldots m$ funções convexas definidas sobre um conjunto convexo χ. Então $\hat{y} \in V$ se e somente se $\hat{y}$ satisfaz o sistema infinito (incontável) de desigualdades

$$\min_{x \in \chi} < \lambda, f(x) - \hat{y} > \leq 0, \forall \lambda \in \Lambda"$$

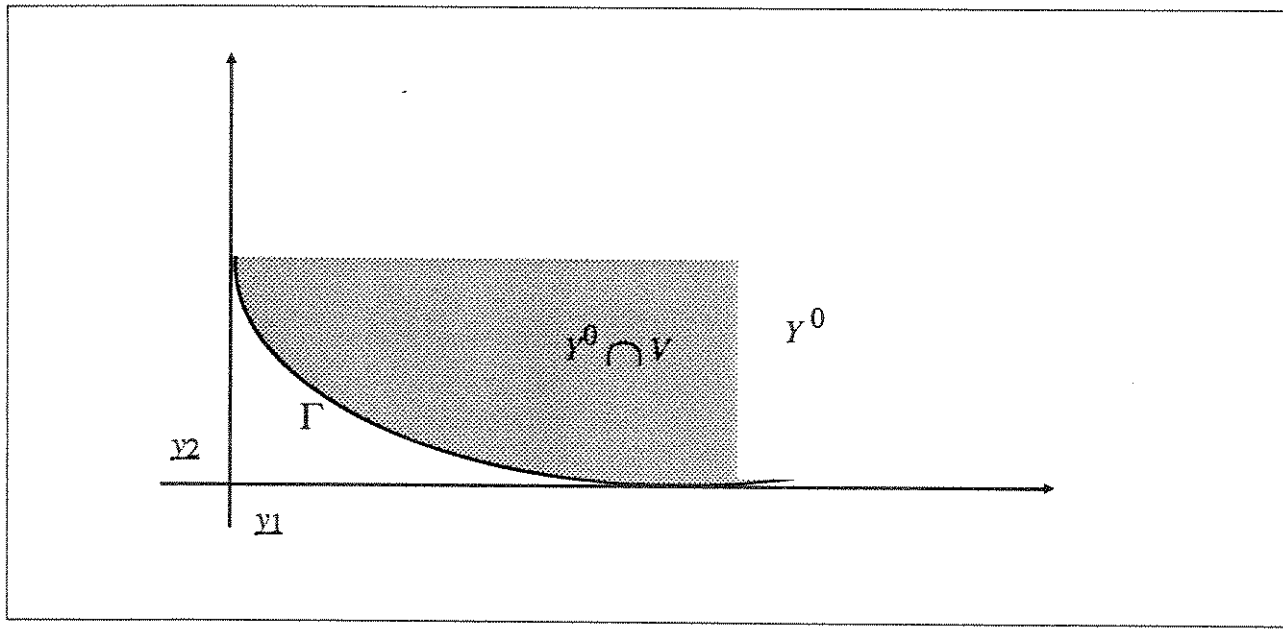


Figura 1.4 - Espaço dos Objetivos

Corolário

" Seja $\theta(\cdot)$: $R^m \rightarrow R$ definida por

$$\theta(y) = \max_{\lambda \in \Lambda} \min_{x \in \chi} < \lambda , f(x) - y >$$

então $\hat{y} \in V$ se e somente se $\theta(\hat{y}) \leq 0$ "

Sempre que $\theta(\hat{y}) > 0$, o vetor $\lambda \in \Lambda$ correspondente estará associado à restrição mais violada de

$$\min_{x \in \chi} < \lambda , f(x) - \hat{y} > \leq 0 , \forall \lambda \in \Lambda$$

Teorema

"Cada desigualdade acima dá origem a um semi-espaço suporte à V"

$$H\lambda = \{ y \in R^m / < \lambda , y > \geq < \lambda , f(x(\lambda)) > \}$$

onde

$$x(\lambda) = \arg \min_{x \in \chi} < \lambda , f(x) - y >$$

Prova: Ferreira, 1986 ; Ferreira e Geromel 1990.

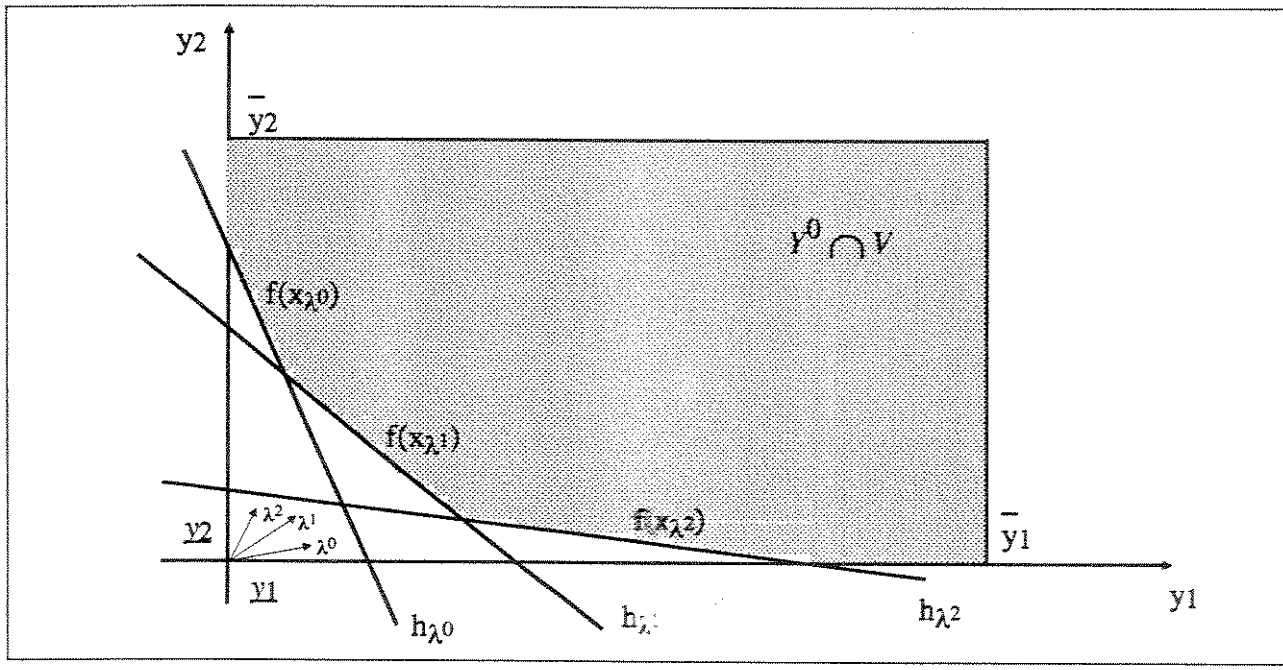


Figura 1.5 - Aproximações de F por Semi-Espaços.

O decisor terá então que resolver

$$\min_y U(y)$$

$$y \in Y^0 \cap H\lambda, \forall \lambda \in \Lambda$$

Esta formulação pode ser viabilizada através de **Relaxação**.

Em geral, os métodos multiobjetivos como enumerados na Tabela II (seção 1.4) alteram fases de Decisão (aspectos subjetivos) e Análise (aspectos numéricos) no espaço das variáveis de decisão $x \in \chi \subseteq R^n$. Recentemente (Ferreira, 1986; Ferreira e Geromel, 1990) foi proposta uma técnica de decomposição do problema multiobjetivo baseada em projeção que, em linhas gerais, permite resolver (1) através do seguinte **Algoritmo Básico**.

Inicialização

Determine $\underline{x}^i = \arg\min_{x \in \chi} f_i(x)$ e faça $\underline{y}_i = f_i(\underline{x}^i)$, $i = 1, \ldots, m$

Se χ é um conjunto compacto é possível ainda obter $\overline{x}^i = \arg\max_{x \in \chi} f_i(x)$ e $\overline{y}_i = f_i(\overline{x}^i)$, $i = 1, \ldots, m$

Senão uma estimativa para $\overline{y}_i$, $i = 1, \ldots, m$ pode ser obtida como em (Geromel e Ferreira, 1991)

Faça $Y^0 = \{ y \in R^m : \underline{y}_i \leq y \leq \overline{y}_i \}, i = 1, \ldots m$ e $k \leftarrow 0$

Decisão

Resolva o problema multiobjetivo relaxado

$$\min_{y \in Y^k} U(y) \qquad (4)$$

onde $U(\cdot)$: $R^m \rightarrow R$ é uma **Função Utilidade Genérica**. Seja y^k a solução encontrada

Análise

Resolva o problema Min-Max

$$\theta(y^k) = \max_{\lambda \in \Lambda} \min_{x \in \chi} (\lambda, f(x) - y^k) \tag{5}$$

onde $\Lambda = \{ \lambda \in R^m : \lambda \geq 0; \Sigma_i \lambda_i = 1 \}$. Seja (λ^k, x^k) a solução encontrada. Se $\theta(y^k) \leq 0$, então x^k resolve o problema original (1). Caso contrário, redefina

$$Y^{k+1} = \{ y \in Y^k : < \lambda, y > \geq < \lambda^k, f(x^k) > \} \tag{6}$$

faça k ← k+1 e volte para o nível de Decisão.

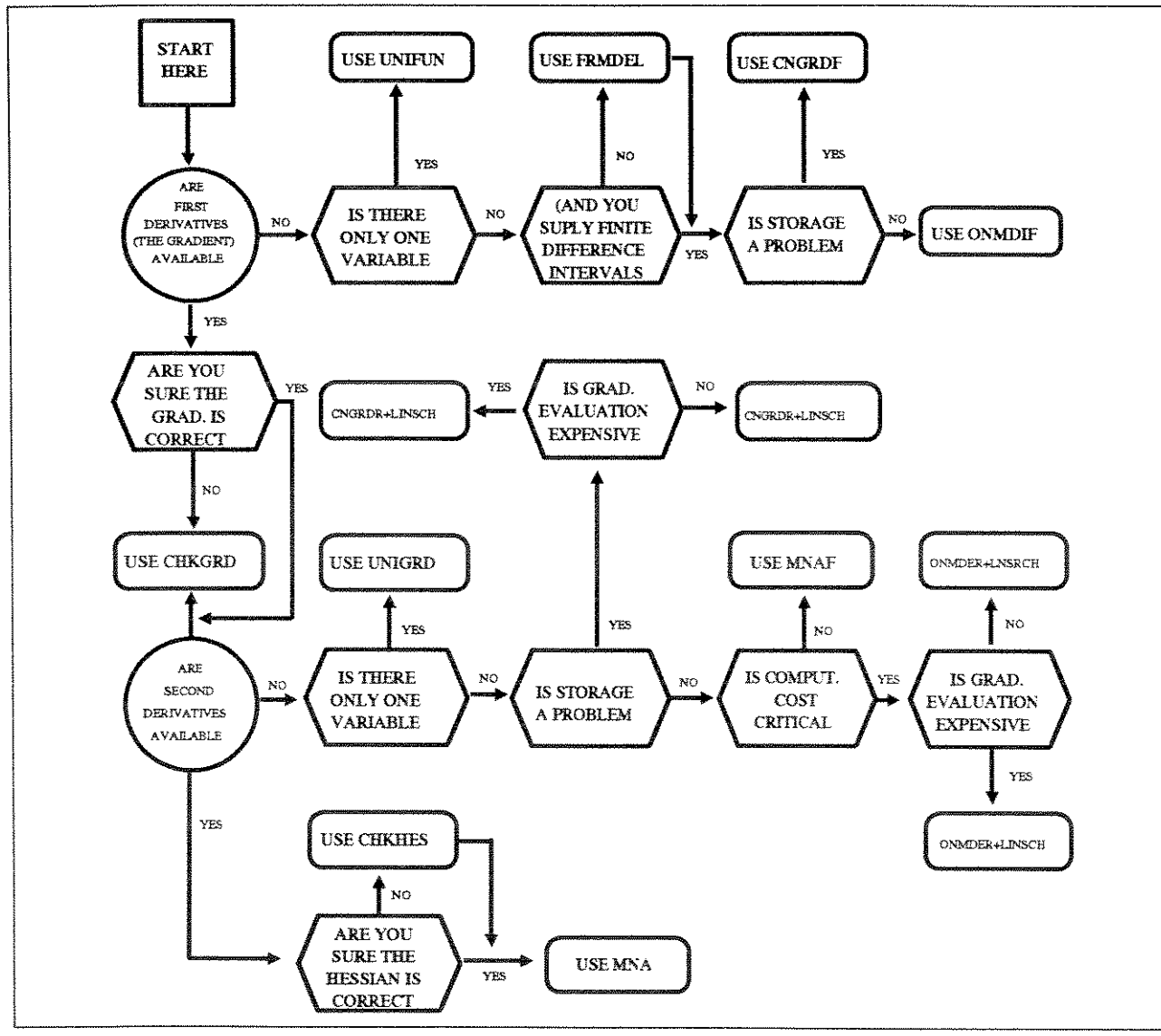


Tabela 1.1 - Gill, Murray e Wright, 1981

Note que embora (4) seja ainda um problema multiobjetivo, algumas características tornam sua solução simplificada.

- i) O número de **variáveis m** (isto é, o número de funções objetivos) é usualmente desprezível se comparado com o número de variáveis do problema original n.
- ii) Os **objetivos** são funções lineares de um tipo muito especial.
- iii) As restrições do problema, representadas pelo politopo Y^k, são sempre lineares.

Observe que estas características independem da natureza do problema original, supostamente convexo, e portanto exibem condições ideais para a implementação de métodos multiobjetivos. Em outras palavras, em se tratando de problemas convexos, a estrutura de decomposição garante que apenas problemas com as características i), ii) e iii) precisam ser considerados.

Esta propriedade torna particularmente adequada a implementação de sistemas de suporte à decisão baseados em Programação Multiobjetivo. O problema (4) garante uniformidade e oferece substanciais facilidades na implementação de qualquer método multiobjetivo. Um exemplo do uso desta técnica pode ser encontrada em (Ferreira e Machado, 1990).

Outra característica interessante é a relativa desagregação das tarefas de Decisão e Análise. Esta última fica praticamente concentrada na resolução do problema Min-Max (5), para o qual algoritmos de Planos de Corte (Luenberger, 1986) podem ser utilizados. Em (5), o problema de minimização interno pode ser reescrito como

$$\min_{x \in X} g(x) \qquad (7)$$

para o qual dispõe-se de algoritmos numéricos, bibliotecas e sistemas de gerenciamento de software eficientes (Dolk, 1986), incluindo algum grau de auxílio ao usuário com relação ao uso do sistema. A Tabela 1.1 exemplifica critérios de seleção de métodos para Otimização Irrestrita.

1.4 - Métodos da Programação Multiobjetivo

Os métodos multiobjetivos são classificados de acordo com a estrutura de preferências do Decisor (Hwang e Masud, 1979; Ferreira, 1986). A Tabela 1.2 apresenta uma classificação de métodos de acordo com diferentes hipóteses sobre a forma de interação do Decisor com o modelo matemático do problema.

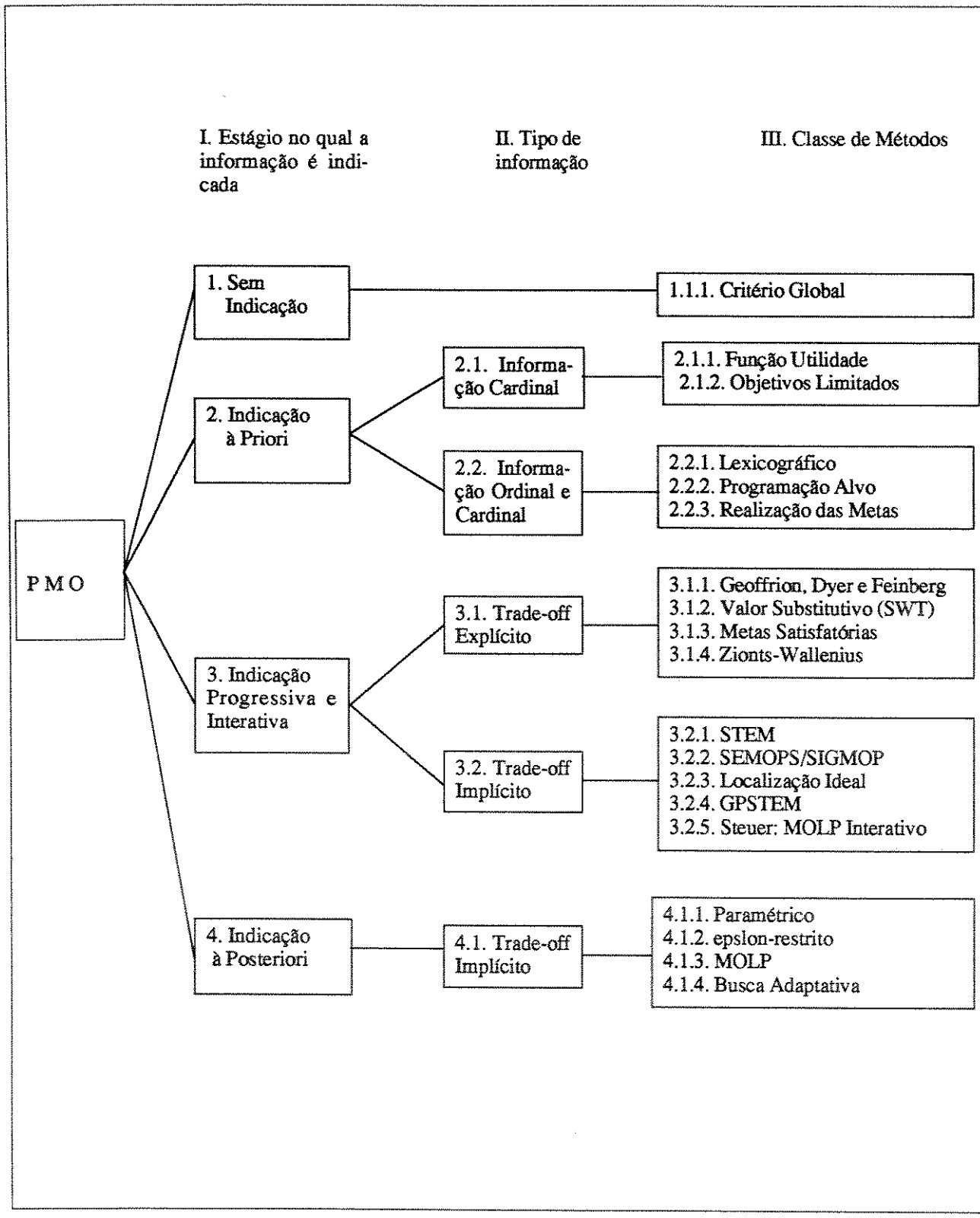


Tabela 1.2 - Hwang e Masud, 1979

1.5 - Descrição de Alguns Métodos: Espaço das Decisões

1.5.1 - Método com Indicação de Preferência À-Posteriori

Método da Ponderação - P_w

As soluções eficientes são encontradas resolvendo-se parametricamente

$$\min_{x \in \chi} \sum_{i=1}^{m} w_i f_i(x) \qquad (P_w)$$

onde

$$w \in W = \{ w \in R^m : w_i \geq 0 \; e \sum_{i=1}^{m} w_i = 1 \}$$

Os pesos w geralmente não refletem a importância relativa dos objetivos no sentido proporcional, sendo apenas parâmetros para a geração de soluções eficientes.

Método das ε-Restrições - P_ε

Neste método, a solução eficiente é gerada a partir do seguinte problema

$$\min_{x \in \chi} f_i(x) \qquad (P_\varepsilon)$$

$$\text{s.a.} \quad f_j(x) \leq \varepsilon_j, j = 1, .., m, \quad (i \neq j)$$

O conjunto de soluções eficientes do problema é gerado a partir da variação dos ε_j's, j = 1,...,m (j ≠ i)

Para um dado ponto $x^* \in \chi$, define-se P_ε^* como o problema P_ε onde $\varepsilon_j = \varepsilon_j^* = f_j(x^*), j \neq i = 1, ..., m.$

1.5.2 - Métodos com Indicação de Preferências À-Priori

Método da Função Utilidade

Uma função $U(\cdot)$ que associa um número real $U(f)$ a cada ponto $f \in F$, é denominada Função Utilidade representativa das preferências do decisor.

Este tipo de método converte o PMO em um problema do tipo

$$\min_{x \in \chi} U(f(x)), \tag{8}$$

onde $f(\cdot) : \chi \rightarrow F$ e $U(\cdot) : F \rightarrow R$ é a Função Utilidade ou Valor que agrega as preferências do Decisor em relação aos m objetivos. Assume-se que $U(\cdot)$ é uma função não-decrescente em relação a cada objetivo. Com esta hipótese é fácil mostrar que no mínimo uma solução de (8) é eficiente (Geoffrion, 1967). A estruturação das preferências do Decisor pode ser obtida através das curvas de nível da sua função utilidade, as quais são também conhecidas como *curvas de indiferença ou isopreferência.* A solução do problema (8) será um ponto onde o conjunto das soluções eficientes for tangente às curvas de indiferença.

Suponha que x^* resolve $\min_{x \in \chi} U(f(x))$. Então sob hipóteses de convexidade, existe $w^* \in W$ tal que $x^* = \arg\min_{x \in \chi} < w^*, f(x) >$ (Ferreira,1986).

Método do Ordenamento Lexicográfico

Neste método, o decisor ordena seus objetivos segundo suas prioridades. Seja a = ($a_1(x)$, ... , $a_m(x)$), o conjunto dos objetivos ordenados pelo Decisor, onde $a_i(x) = f_j(x), j = 1, \ldots m$..O problema fica formulado da seguinte maneira:

$$\min_{x \in \chi_{i-1}} a_i(x), \tag{9}$$

onde

$$\chi_i = \{ x \in \chi_{i-1} : x = \arg \min_{x \in \chi_{i-1}} a_i(x) \}, \ \chi^0 = \chi$$

Para se chegar a solução final, resolve-se (9) seqüencialmente para i=1, ... ,m.

Este método apresenta a desvantagem da solução preferida x^* ser muito sensível ao ordenamento, o que é indesejável se os objetivos tiverem aproximadamente a mesma importância. Com o, propósito de reduzir esta sensibilidade Waltz (1967) propos que os critérios pudessem variar dentro de certas faixas percentuais do valor ótimo a_l^* encontrado nos passos anteriores. Deste modo, o problema (9) é reformulado como

$$\min_{x \in \chi} a_i(x) \tag{10}$$

$$\text{s.a.} \quad a_l(x) \leq a_l^* + \delta_l, \ l = 1, \ldots, i-1,$$

onde $\delta_l > 0$, $l = 1, \ldots, i-1$ são tolerâncias determinadas pelo Decisor.

Método da Programação Alvo

A Programação Alvo (PA) foi inicialmente proposta para resolver problemas lineares, e alcançou um grande desenvolvimento nas últimas décadas.

A formulação do problema (1) através de programação alvo é apresentada a seguir:

$$\min_{x \in \chi} \sum_{i=1}^{m} [(d_i^- + d_i^+)^p]^{1/p} , \; p \geq 1 \tag{11}$$

sujeito a:

$$f_i(x) + d_i^- - d_i^+ = b_i , \; i = 1, \ldots, m \tag{12}$$

$$d_i^- , \; d_i^+ \geq 0 \tag{13}$$

$$d_i^- \; . \; d_i^+ = 0 \tag{14}$$

onde b_i, $i = 1,\ldots,m$ são as metas estabelecidas pelo Decisor para os objetivos $f_i(x)$, $i=1,\ldots,m$, d_i^+ representa quanto o objetivo $f_i(x)$ excedeu a meta estipulada b_i, e d_i^- representa quanto o objetivo $f_i(x)$ ficou abaixo a meta estipulada b_i. O valor de p depende da forma com que os desvios devem ser ponderados. Em geral $p = 1$.

A solução preferida é definida como a que apresenta o *menor desvio total* em relação aos alvos ou metas desejados.

Uma variante para a formulação de PA pode ser conseguida incorporando-se ordenamento lexicográfico aos desvios. Assim, a formulação de PA para o problema (1) passa a ser

$$\min_{\substack{x \in \chi \\ d^+ , d^-}} (a_1 (d^+ , d^-) , \; \ldots , a_m \; (d^+ , d^-)) \tag{15}$$

s.a. (12) — (14)

cuja solução é conseguida resolvendo-se inicialmente

$$\min_{\substack{x \in \chi \\ d^+ , d^-}} a_1 (d^+ , d^-) \tag{16}$$

s.a. (12) — (14)

obtendo-se a_1^*.Em seguida resolve-se para i=2,...,m.

$$\min_{x \in \chi} \; a_i (d^+, d^-) \qquad (17)$$
$$d^+, d^- \geq 0$$

s.a. (12) — (14)

$$a_j (d^+, d^-) \leq a_j^* \;, j = 1, \ldots, i$$

A grande vantagem de Programação Alvo é que o Decisor não precisa atribuir pesos numéricos aos objetivos. Por outro lado, apresenta como desvantagem a sensibilidade à ordenação dos desvios e às metas estipuladas pelo decisor.

1.5.3 - Métodos com Indicação Progressiva de Preferências - Interativos

Esta classe de métodos, geralmente referida como métodos interativos, tem tido nos últimos anos uma grande aceitação por parte dos pesquisadores e usuários. A maioria destes métodos requer que o Decisor forneça *trade-off's locais* (implícitos como níveis de aspiração ou explícitos como taxas marginais de substituição) na vizinhança de uma alternativa viável. Estes trade-off's assumem a função de coeficientes variáveis para algum problema de otimização escalar. Apresenta-se a seguir alguns métodos desta classe:

Método da Geoffrion, Dyer e Feinberg

O estudo desta técnica demonstra que um algoritmo de gradiente pode ser utilizado para resolver um PMO se o Decisor for capaz de fornecer avaliações locais sobre sua função utilidade $U(\cdot)$ (Geoffrion et al., 1972). O método assume que o problema está formulado como segue:

$$\min_{x \in \chi} \; U(f_1(x), \ldots, f_m(x)) \qquad (18)$$

Para permitir a aplicação do algoritmo de Frank-Wolfe à resolução do problema assume-se que a função $U(\cdot)$ e os objetivos $f_i(\cdot)$, $i = 1, \ldots, m$ são diferenciáveis e convexos sobre um subconjunto convexo $\chi \subset R^n$, com $U(\cdot)$ não-decrescente em relação à cada objetivo. Os passos deste algoritmo são apresentados a seguir.

- **Passo 1:** *Determine uma solução inicial factível* $x^0 \in \chi$ *e faça* k=0.
- **Passo 2:** *Obtenha uma solução ótima* ξ^k *para o problema de otimização*

$$\min_{\xi \in \chi} \; < \nabla_x U(f(x^k)) \;, \xi > \qquad (19)$$

faça $d^k = \xi^k - x^k$

- **Passo 3:** *Determine a solução ótima do problema de busca unidimensional*

$$\min_{t \in [0,1]} \; U(f(x^k + t\,d^k))$$

- **Passo 4:** *Faça* $x^{k+1} = x^k + t^k d^k$. *Se* $\| x^{k+1} - x^k \| < \varepsilon$, *para* $\varepsilon > 0$ *arbitrariamente escolhido, pare: a solução* x^{k+1} *resolve (18). caso contrário faça k=k+1 e retorne ao Passo 2.*

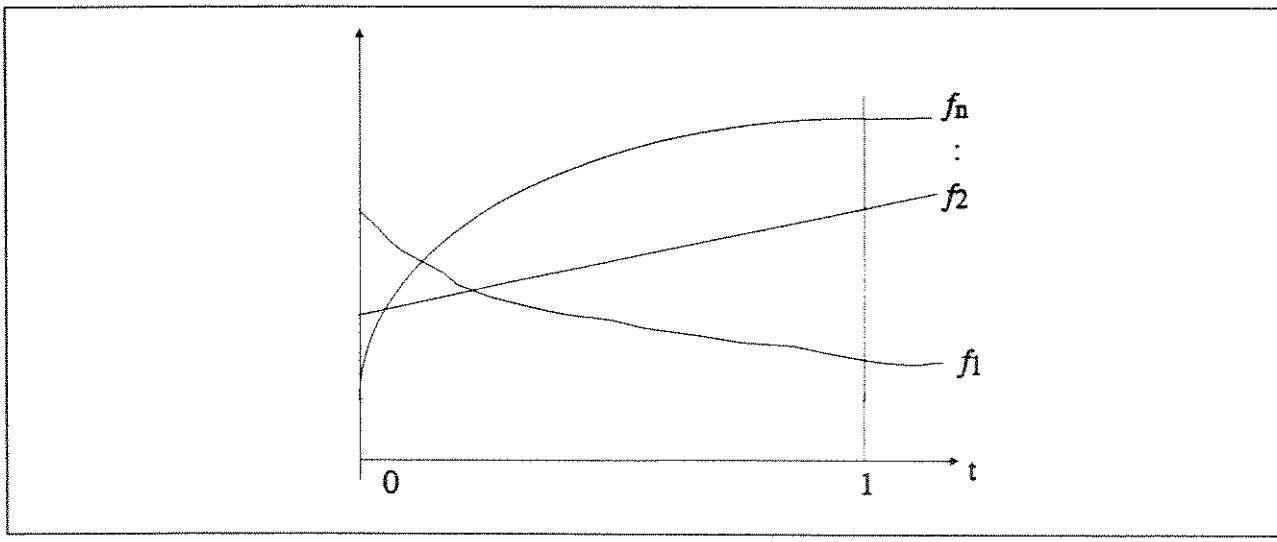


Figura 1.6 - Escolha do passo t

Devido ao fato da função utilidade $U(\cdot)$ geralmente não ser explicitamente conhecida, requer-se o auxílio do Decisor para execução dos Passos 2,3 e 4. Assume-se que o Decisor é capaz de estimar os trade-off's entre dois objetivos quaisquer em uma dada solução do problema. Estes trade-off's podem ser utilizados para encontrar a direção do gradiente de $U(\cdot)$ naquela solução.

O gradiente da função utilidade pode ser expresso a partir da aplicação da regra da cadeia como

$$\nabla_x U(f(x)) = \sum_{i=1}^{m} \frac{\partial U}{\partial f_i} \nabla f_i(x).$$

Como a solução do problema (18) não será afetada pela multiplicação da função objetivo por um escalar positivo, pode-se reescrevê-lo como,

$$\min_{\xi \in \chi} < \sum_{i=1}^{m} \tau_i^k \; \nabla f_i(x^k) \; , \; \xi > \qquad (20)$$

onde

$$\tau_i^k = (\partial U / \partial f_i^k) / (\partial U / \partial f_1^k) \ , \ i = 2, \ldots, m \tag{21}$$

define a *Taxa Marginal de Substituição* do decisor entre o critério i e o critério 1, arbitrariamente escolhido como critério de referência.

Uma forma de se obter τ_i é determinar uma mudança infinitesimal Δf_1 no objetivo de referência que é compensada *exatamente* por uma mudança Δf_i no i-ésimo objetivo, enquanto todos os outros são mantidos constantes. Pode-se então aproximar (21) por,

$$\tau_i^k = - \frac{\Delta f_i^k}{\Delta f_1^k} \qquad i = 1, \ldots, m . \tag{22}$$

Devido a subjetividade na estimação dos trade-off's, o decisor pode ter dificuldade em encontrar valores expressivos. Com o propósito de reduzir esta dificuldade, foi proposto um esquema interativo baseado em comparações ordinais que permite a determinação das taxas marginais de substituição com razoável precisão (Dyer, 1972). Para a execução do Passo 3 resolve-se um problema de busca unidimensional com o auxílio do decisor. A escolha do passo t pode ser conseguida plotando-se todos os m objetivos para $t \in [0,1]$. Desta forma o Decisor poderá determinar qual o melhor passo $t^k \in [0,1]$

A condição teórica de parada do algoritmo (Passo 4) será satisfeita quando as soluções x^k e x^{k+1} forem iguais, o que nem sempre é possível. Em (Hwang e Masud, 1979) apresenta-se um critério de parada alternativo baseado na variação da função utilidade.

O método de Geoffrion apresenta-se como uma ferramenta eficiente para a resolução de problemas multiobjetivos, devido a simplicidade computacional e as fortes propriedades de convergência do algoritmo de Frank-Wolfe, (Geoffrion et al.,1972 ; Dyer,1973). Para analisar a robustez da convergência do algoritmo de Frank-Wolfe (Dyer,1974 ; Ferreira,1986) considera-se a introdução de erros no cálculo do gradiente da função utilidade, reescrevendo-se (19) como,

$$\min_{\xi \in \chi} < \nabla_x U(f(x^k)) + \eta^k \ , \ \xi > \tag{22}$$

Deste modo, pode-se mostrar que a seqüência $[\ \xi^l\ ]_0^\infty$ gerada pelo algoritmo converge para uma solução ótima se $\lim_{k \to \infty} \eta^k \to 0$ .Analisando-se este fato no contexto multiobjetivo, isto indica que o erro na estimação da taxa marginal deve convergir para 0 quando $k \to \infty$, o que pode ser perfeitamente admitido supondo-se que o decisor melhora progressivamente suas estimativas. Assumindo estas condições, o algoritmo sempre convergirá.

Método STEM ("Step Method")

Este método foi proposto para a resolução de problemas multiobjetivos lineares (Benayoun et al., 1971), e permite ao Decisor reconhecer boas soluções de compromisso e a importância relativa

dos objetivos. Esta técnica baseia-se na premissa de que a melhor solução de compromisso tem um desvio mínimo em relação a solução ideal.

A formulação do problema para uma iteração genérica l é dada a seguir:

$$\min_{\delta, x \in \chi^l} \delta \tag{23}$$

$$\text{s.a.} \quad \pi_i (f_i(x) - \underline{y_i}) \leq \delta \ , \ i = 1, \ldots, m, \tag{24}$$

onde χ^l inclui $\chi = \{ x \in R^m : x \geq 0, Ax \leq b \ \}$, $A \in R^{p \times n}$ $(p \leq n)$, $b \in R^p$ e $f_i(x) = < c^i, x >, c^i \in R^n, i = 1, \ldots, m$.O valor de π_i reflete a importância relativa da distância de $f_i(\cdot)$ à solução ideal $\underline{y_i}$, e pode ser calculado através da expressão,

$$\pi_i = \frac{\alpha_i}{\sum_{j=1}^{m} \alpha_j}, \qquad i = 1, \ldots, m, \tag{25}$$

onde

$$\alpha_i = \frac{\overline{y_i} - \underline{y_i}}{\underline{y_i} \ \| c^i \|}, \quad \text{se } \underline{y_i} > 0 \tag{26}$$

$$\alpha_i = \frac{\underline{y_i} - \overline{y_i}}{\underline{y_i} \ \| c^i \|}, \quad \text{se } \underline{y_i} \leq 0 \tag{27}$$

e $\overline{y_i}$ é o *valor pessimista* de f_i, isto é,

$$\overline{y_i} = \max_{1 \leq j \leq m} \{ f_i(x^j) \}$$

onde $x^j = \arg\min_{x \in \chi} f_j(x)$, $j = 1, \ldots, m$.

Pode-se notar que o peso π_i dado a cada diferença depende basicamente da variação $(\overline{y_i} - \underline{y_i})$ de cada objetivo. Se esta variação for pequena, π_i também será, e o objetivo $f_i(\cdot)$não será muito sensível às variações de π_i.

A solução x^l de (23) é apresentada ao Decisor que compara subjetivamente$f_i(x^l)$ com $\underline{y_i}$, e classifica seu valor como satisfatório ou não.

Para algum i tal que f_i^l é satisfatório, o Decisor deverá fornecer uma quantidade $\delta_i > 0$ a qual está disposto a sacrificar, em troca de uma melhora em algum objetivo não satisfatório.

Deste modo, o conjunto de restrições para a iteração l+1 fica sendo

$$\chi^{l+1} = \{ x \in \chi^l : f_i(x) \leq f_i^l + \delta_i \; ; \; f_j(x) \leq f_j^l \text{ , qualquer } j \neq i \} \tag{28}$$

Faz-se $\pi_i = 0$, $l = l$+1 e recalcula-se (23) — (24). A solução final é encontrada quando todos os objetivos forem satisfatórios.

A grande vantagem deste método é que a parte de cálculos pode ser desempenhada através de uma rotina de PL, um *simplex multiobjetivo* por exemplo, o qual permite otimizações sucessivas de várias funções objetivos.

1.6 - Descrição de Alguns Métodos: Espaço dos Objetivos

Seja o problema abaixo:

$$\min_y U(y) \tag{29}$$

$$y \in Y^{k+1} = \{ y \in Y^k : \langle \lambda^k, y \rangle \geq \langle \lambda^k, f(x^k) \rangle \} \tag{30}$$

Como observado em Ferreira (1986), a metodologia baseada no espaço dos objetivos baseia-se, em parte, na habilidade do decisor em escolher a solução que melhor preencha os seus níveis de satisfação através da observação do espaço dos objetivos e dos sucessivos cortes gerados ao longo das iterações. Este procedimento é válido no caso biobjetivo (m = 2), tornando-se difícil se m = 3 e sem sentido se m > 3. Nestes termos será então relevante analisar algumas técnicas de solução de (29) – (30) que eliminem a necessidade de interpretações gráficas.

Estas técnicas são baseadas em algoritmos já existentes em programação multiobjetivo, porém reformulados no sentido de explorar as características do problema relaxado:

- linearidade de objetivos e restrições;
- número de variáveis = número de objetivos = m << n.

Demonstra-se que estas reformulações conduzem às implementações mais eficientes de cada método considerado. Além disso, como normalmente a quantidade de informação a ser manipulada é menor espera-se conseguir avaliações mais precisas da parte do Decisor e reduzir o esforço computacional para resolver o modelo.

Na seção 1.2 formulou-se o problema de otimização multiobjetivo no espaço dos objetivos F como:

$$\min_{y \in F} \quad y \tag{31}$$

onde

$$F \triangleq \{\, y \in R^m : y = f(x), x \in \chi \,\}.$$

Nesta mesma seção 1.2 mostrou-se que através do conceito de solução satisfatória, o problema de programação multiobjetivo pode ser substituído por

$$\min_{y \in Y^{k+1}} \quad y \tag{32}$$

onde Y^{k+1} denota a aproximação de ordem k do espaço de soluções satisfatórias do problema (31).

O problema (32), por sua vez, também possui um conjunto de soluções eficientes $\Gamma^{k+1} \in \partial Y^{k+1}$ que nada mais é do que a aproximação por k hiperplanos para $\Gamma \subset Y^{k+1}$. Lembrando então que (32) é linear, o conjunto Γ^{k+1} pode em princípio ser completamente gerado por meio de técnicas apropriadas para posterior definição, pelo Decisor, da melhor solução de compromisso de (31).

Entretanto este método exaustivo parece ser desnecessário em razão das facilidades que o problema (32) oferece para a utilização de técnicas interativas. No tocante, por exemplo, à geração de soluções eficientes (nível de análise) podem ser empregados um dos procedimentos paramétricos a seguir

$$\min_{y \in Y^{k+1}} \quad < \lambda , y > , \qquad \lambda \in \Lambda \tag{33}$$

$$\min_{y \in Y^{k+1}} \quad y_i \tag{34}$$

$$y_j \leq \varepsilon_j , \qquad j = 1, \ldots, m \quad (j \neq i) \tag{35}$$

Considere o método proposto por Haimes e Hall (1974) que em linhas gerais consiste em interpretar as variáveis duais associadas às ε-*restrições* como *trade-offs* locais entre o critério de referência i e todos os demais. Esta mesma idéia quando transposta ao espaço dos objetivos faria o problema de geração de soluções eficientes recair no problema (34)—(35) com $\varepsilon_j = \underline{y}_i + \overline{\varepsilon}_j$, $\overline{\varepsilon}_j \geq 0$, $j = 1, \ldots, m$ $(j \neq i)$. Contudo ao contrário da versão original do método que exige o emprego de algoritmos especiais para tratamento de possíveis não-linearidades, o problema (34)—(35) oferece condições ideais para o desenvolvimento de análises de sensibilidade via programação linear paramétrica. A Figura 1.7 ilustra como seria obtida a solução de (34)—(35) para o caso biobjetivo.

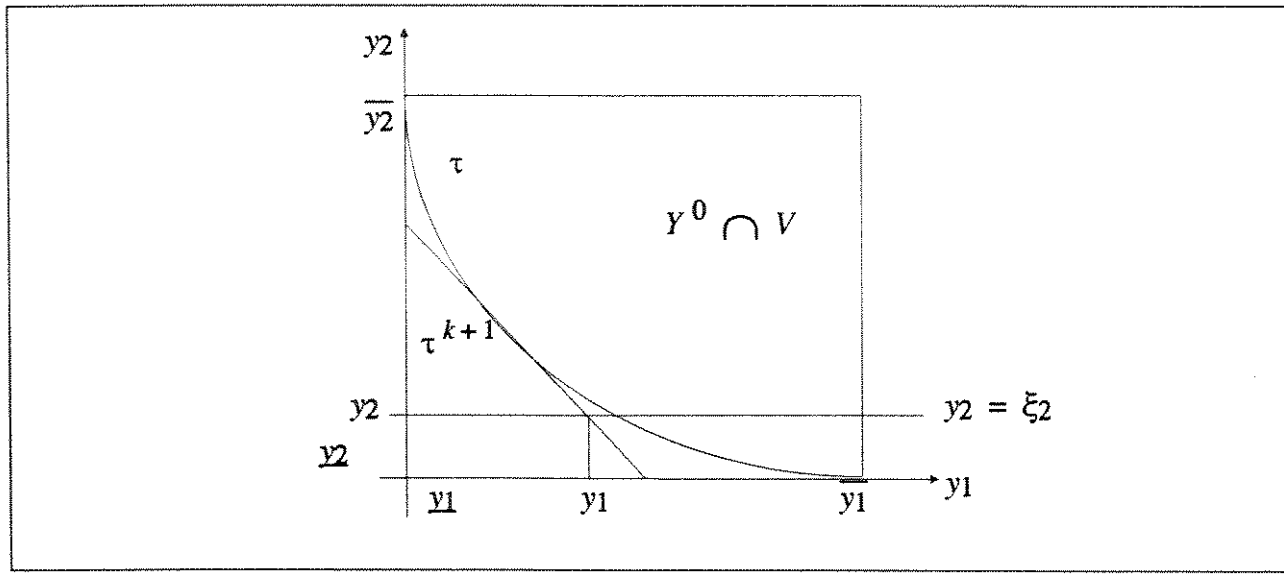


Figura 1.7 - Resolução do ε-restrito

Os demais passos da resolução de (34) através do método de Haimes e Hall seguem analogamente os passos do algoritmo original, isto é, construção da chamada Função do Valor Substituto (*Surrogate Worth Function*) (Haimes et al., 1979) e determinação da faixa de indiferença do Decisor.

A principal desvantagem do método original reside na necessidade de geração de um número muito grande de soluções eficientes. A técnica de relaxação adotada vem então no sentido de reduzir o esforço computacional associado com esta tarefa.

Consideremos agora o método de Geoffrion, Dyer e Feinberg que assume a seguinte formulação para o problema multiobjetivo

$$\min_{x \in \chi} U(f(x)) \tag{36}$$

Além disso, vamos assumir que a função $U(\cdot)$ seja diferenciável e convexa sobre um subconjunto convexo e compacto $\chi \subset R^n$, para assim viabilizar o emprego do método de Frank-Wolfe.

Reescrito nos termos desta nova metodologia, o algoritmo descrito na seção 1.5.3 consistiria dos seguintes passos:

- **Passo 0:** *Determinação de uma solução inicial factível. O cumprimento deste passo é trivial pois por construção, a última solução eficiente gerada pertence a* Y^{k+1}, *isto é,* $f(x^k) \in \Gamma \subset Y^{k+1}$. *Faça* $l = 0$ *e* $y^0 = f(x^k)$.

- **Passo 1:** *Obtenha uma solução ótima* ξ^l *para o problema de otimização*

$$\min_{\xi \in y^{k+1}} < \nabla U(y^l) , \xi > \tag{37}$$

Faça $d^l = \xi^l - y^l$. *Note que* (20) *pode agora ser substituído por*

$$\min_{\xi \in y^{k+1}} < \tau^l , \xi > \tag{38}$$

onde $w_i^l = (\partial U / \partial y_i) / (\partial U / \partial y_1)$, *cujo valor pode ser aproximado por*

$$\tau_i^l = - \frac{y_1 - y_1^l}{y_i - y_i^l} , \qquad i = 1 , \ldots , m \tag{39}$$

e que corresponde à obtenção de uma taxa de ***substituição relaxada***.

- **Passo 2:** *Determine a solução ótima do problema de busca unidimensional*

$$\min_{t \in [0,1]} U(y^l + t d^l) \tag{40}$$

No caso do problema (40), a sugestão para a determinação do passo t^l levaria à construção de um diagrama, como ilustrado na Figura 1.8 . Note que a obtenção deste diagrama não representa na prática nenhum esforço de cálculo, ao contrário da versão original do método em que a determinação de t^l envolve o cálculo do valor de funções no intervalo $t \in [0 , 1]$.

A atualização das variáveis (*objetivos*) e teste de convergência são similares ao passo 3 do algoritmo original. A Figura 1.9 ilustra uma iteração do método de Geoffrion et alli adaptado ao espaço dos objetivos.

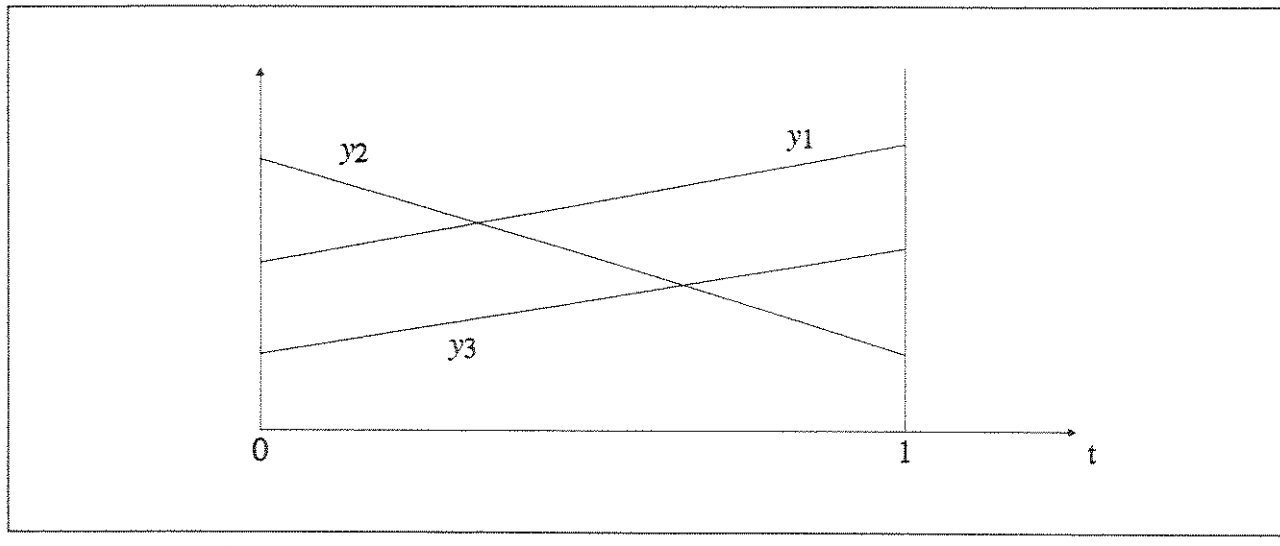


Figura 1.8- Determinação do passo do prob. relaxado

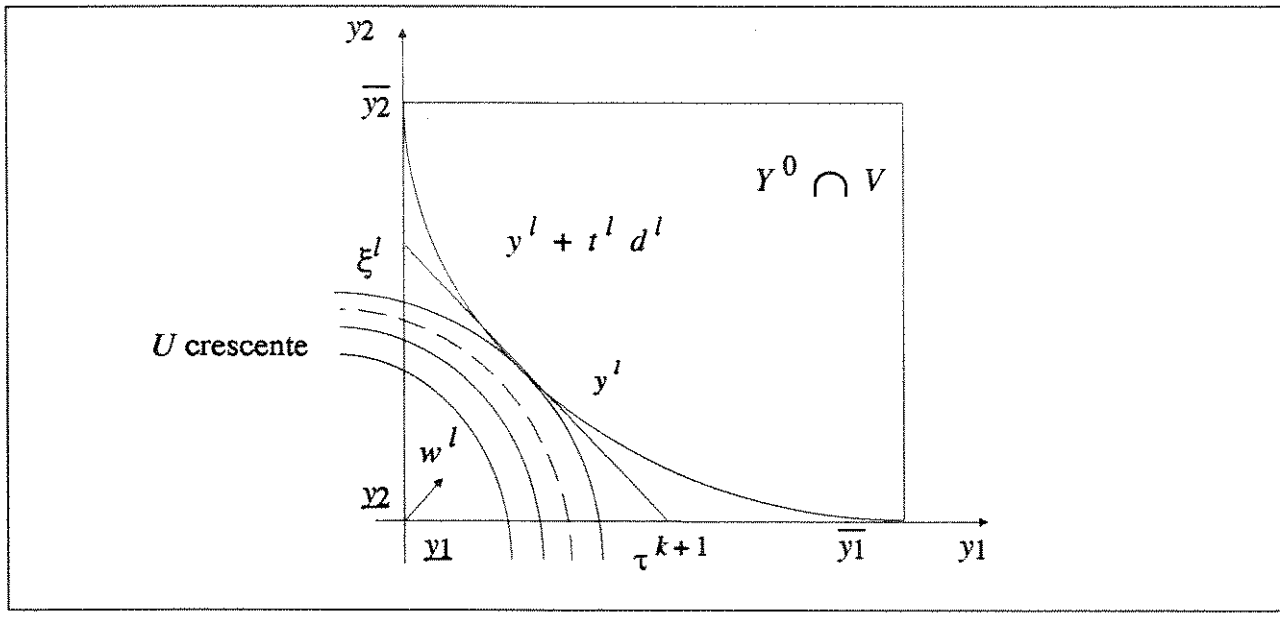


Figura 1.9 - Interação do Método de Geoffrion

1.7 - Conclusão

Neste capítulo abordamos aspectos teóricos básicos relativos a Otimização Multiobjetivo bem como os tipos de formulação para PMO. Foram apresentadas algumas definições e teoremas básicos e descreveu-se a projeção do PMO no espaço dos objetivos. Finalmente foi elaborado um estudo dos métodos multiobjetivos segundo a classificação baseada em estrutura de preferências.

O estudo dos métodos dividiu-se em duas partes: métodos desenvolvidos para o espaço das variáveis de decisão e de suas versões para o espaço dos objetivos. Esta pesquisa foi de essencial importância para o desenvolvimento do Sistema de Suporte à Decisão MC++, pois vários dos métodos descritos foram implementados e testados.

Capítulo 2
Software para Programação Matemática

2.1 - Introdução

O desenvolvimento e implementação de algoritmos eficientes em Programação Matemática (PM) tomou muito tempo e esforço. O sucesso desta atividade ao lado da evolução constante do hardware computacional tem permitido a resolução de problemas grandes e complexos num tempo reduzido. Para efetuar a resolução destes problemas vem sendo desenvolvidos softwares de PM que normalmente exigem dos usuários uma considerável familiaridade com PM, bons conhecimentos computacionais e um período de aprendizado para lidar com o sistema.

Os Sistemas de Suporte a Decisão (SSD) podem ser utilizados na elaboração e resolução de problemas de PM com a vantagem de proporcionar um ambiente menos hostil ao usuário, além de possuir várias outras melhorias se comparados a um software de PM. Um exemplo comum de SSD é uma Planilha de Cálculo (Lotus 123, Quatro Pro) que, através de uma interface amigável, permite ao usuário vários níveis de análise de informações.

Um SSD, normalmente é composto de 3 partes:

- Sistema para gerenciamento de interface;
- Sistema para gerenciamento de dados;
- Sistema para gerenciamento de modelos.

O sistema para gerenciamento da interface consiste de todas as interfaces entre o usuário e o sistema tais como a linguagem usada para definir um problema (modelo) e o modo como os resultados são impressos (gráficos, tabelas, relatórios). O gerenciamento de dados envolve a criação, armazenamento, manipulação e recuperação de dados, o que pode ser conseguido através de um Sistema Gerenciador de Banco de Dados (SGBD) associado ao SSD. O sistema para gerenciamento de modelos deve funcionar como um SGBD, porém utilizando modelos ao invés de dados.

Um SSD deve ser suficientemente fácil de ser utilizado por usuários que tenham pouco ou nenhuma experiência em computadores e em programação matemática. Muitos dos sistemas de PM cometem esta falha, pois requerem dos usuários um conhecimento técnico apurado. Isto deve-se a ênfase dada na elaboração e implementação de algoritmos eficientes e de técnicas para resolução de modelos. Atualmente este quadro está se alterando com as várias pesquisas na área da análise, interpretação e uso dos modelos.

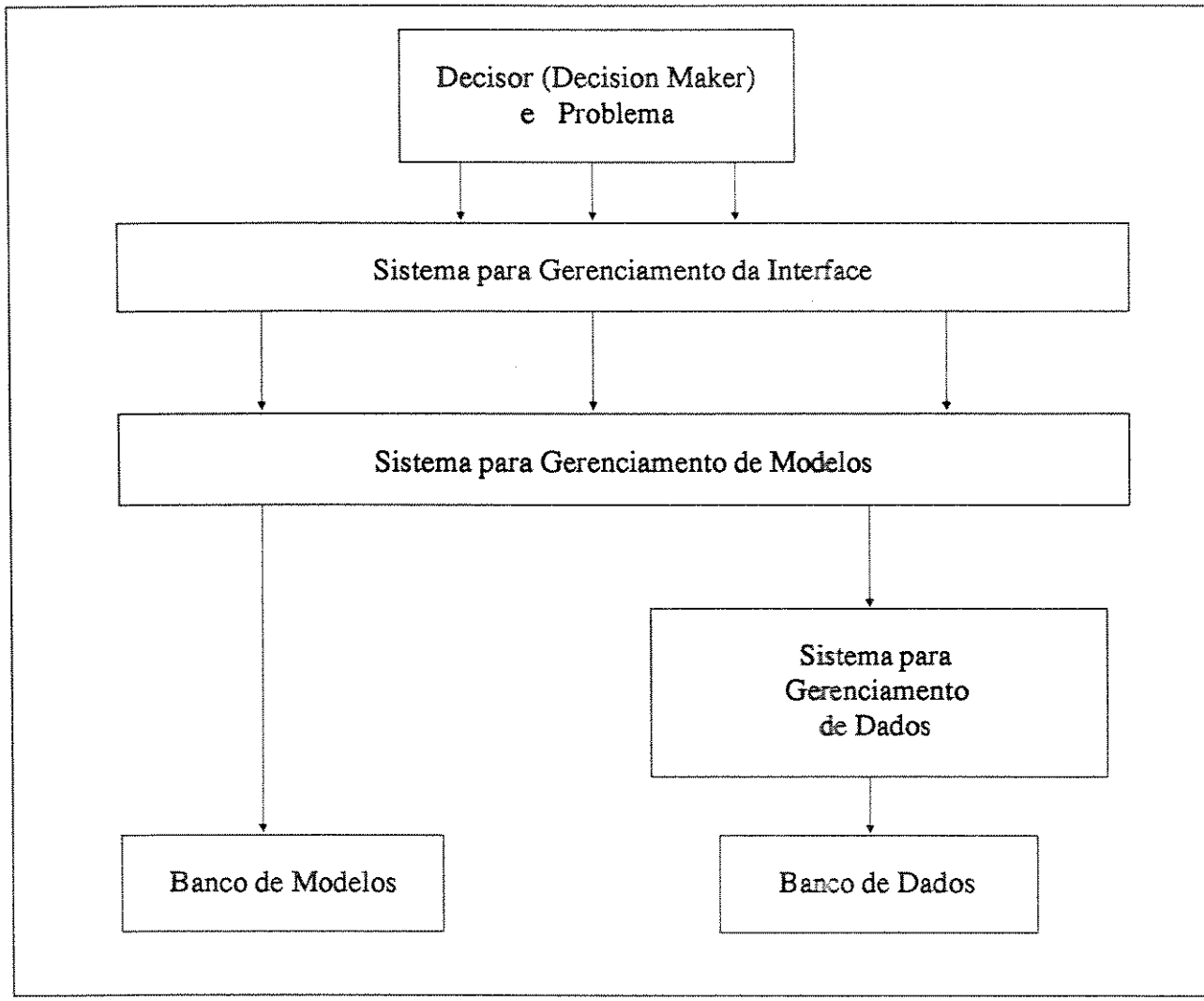


Figura 2.1 - Esquema Geral de um SSD

2.2 - Características Básicas

Historicamente, a maioria dos softwares de PM operava em sua própria base ("standalone"), assumindo que o usuário conhecia o suficiente sobre o seu problema, a ponto de escolher os modelos apropriados e os pacotes requeridos para a sua solução. Mesmo quando o usuário possuía este conhecimento, havia uma certa relutância em utilizar os softwares devido ao tempo gasto no seu aprendizado. Esta associação não é garantida num SSD, onde os usuários (empresários, gerentes, operadores) podem não saber que modelo é apropriado para o seu problema, bem como podem desconhecer as técnicas utilizadas para a resolução do mesmo (Programação Linear, Programação Inteira, etc.).

Portanto, a proposta de um Sistema Gerenciador de Modelos (SGM) é criar um ambiente em que o usuário possa definir e resolver os seus problemas, mesmo não possuindo um conhecimento técnico profundo.

Um SGM deve satisfazer as seguintes condições:

- Manipular modelos do mesmo modo que um SGBD manipula dados;
- Ser independente de algoritmos;
- Suportar múltiplas visões de modelos para o usuário;
- Ser baseado em conhecimento ("Knowledge - based");
- Poder representar uma larga faixa de problemas.

Para tanto requer um software de modelamento, análise e desenvolvimento bem elaborado e mais generalizado.

2.2.1 - Modelos e Dados

Os modelos num SGM são muito importantes e devem ser manejados com muito mais rigor e atenção do que os dados. Portanto um SGM deve possuir as seguintes características.

- descrição de modelos;
- manipulação de modelos;
- controle de modelos.

A descrição de modelos pode ocorrer em diferentes níveis, dependendo da visão de modelo do usuário. Podemos dividir os usuários em dois grandes grupos: os decisores (decision makers) e os modeladores (model builders). Os decisores requerem uma interface de alto nível para o tratamento de seus problemas. Já os modeladores podem utilizar uma linguagem de baixo nível para especificar os detalhes de um modelo.

A manipulação de modelos inclui as funções padrão de um SGBD como CREATE, STORE, DELETE, MODIFY e DISPLAY além de funções específicas para modelos como SOLVE, LINK, AGREGATE, DECOMPOSE. Devemos notar que num SGM, a função SOLVE é apenas uma das várias funções existentes, o que difere dos sistemas anteriores que tinham a resolução do problema como seu objetivo principal.

O controle de modelos envolve características de autorização de acesso, segurança e privacidade, integridade e administração. Estas características também encontradas a um SGBD, possuem uma outra visão no contexto de um SGM. A integridade, por exemplo, de um modelo requer a validade e a formulação correta do mesmo, o que é uma tarefa mais difícil de se realizar comparada a integridade de dados num SGBD.

Os componentes de um modelo têm uma diferença qualitativa com os seus componentes similares num banco de dados (BD), porém é importante enxergar um modelo como um dado pois, assim, podemos empregar os já bem definidos princípios de um SGBD para a construção e desenvolvimento de um SGM.

2.2.2 - Independência de Algoritmos

A modelagem é normalmente desenvolvida em função de um determinado algoritmo. Este fato acarreta um baixo grau de independência de algoritmos, forçando o modelador a utilizar diferentes softwares para resolver os diferentes modelos. Um outro problema é o tempo gasto pelo modelador (usuário) para aprender as características dos vários softwares, além da dificuldade (ou impossibilidade) de *linkar* os vários softwares para se solucionar um modelo de maior complexidade.

Um SGM deve possuir um alto grau de independência de algoritmos, possibilitando que o modelador (usuário) seja capaz de trabalhar com diferentes modelos sem ter de recorrer a outros pacotes de software. Do mesmo modo, os decisores (decision makers) devem acessar e manipular estes modelos sem serem barrados pela sintaxe ou terminologia específica do problema a ser resolvido.

Estas implicações requerem no mínimo duas características de um SGM:

- Um SGM deve possuir uma capacidade de representação de modelos muito mais robusta. As representações precisam ser gerais o suficiente para modelar problemas de vários tipos, utilizando para isto uma linguagem de modelamento.
- A descrição de modelos deve ser separada da solução de modelos. A solução é apenas uma atividade no processo de resolução de um problema, não devendo dominar o sistema.

2.2.3 - Múltiplas Visões de Modelos

Uma característica muito importante num SGM é a capacidade de fornecer a visão do mesmo modelo em diferentes níveis de abstração. Por exemplo, um decisor pode ver um modelo como simplesmente um nome ou um gráfico. Já o modelador pode ver o modelo como um grafo, ou um conjunto de funções (funções objetivos e restrições) ou ainda como uma matriz esparsa. O conjunto de visões de um modelo pode ser conceitualizado como um espaço de abstração com operadores de transformações entre as várias representações dos espaços.

Um SGM deve definir um espaço de abstração para o usuário e realizar as transformações requeridas entre os diferentes níveis de abstração. Isto permite ao usuário interagir com o modelo no nível mais adequado ao seu conhecimento e visão do mesmo.

Espaço de abstração é um conceito poderoso que, novamente, requer uma representação de modelos correspondentemente poderosa. Exige intensivas transformações entre espaços, o que requer muitos operadores de transformação para que a navegação entre os níveis de abstração seja rápida e eficiente. A Figura 2.2 ilustra o conceito de espaço de abstração, indicando as várias transformações necessárias entre as abstrações.

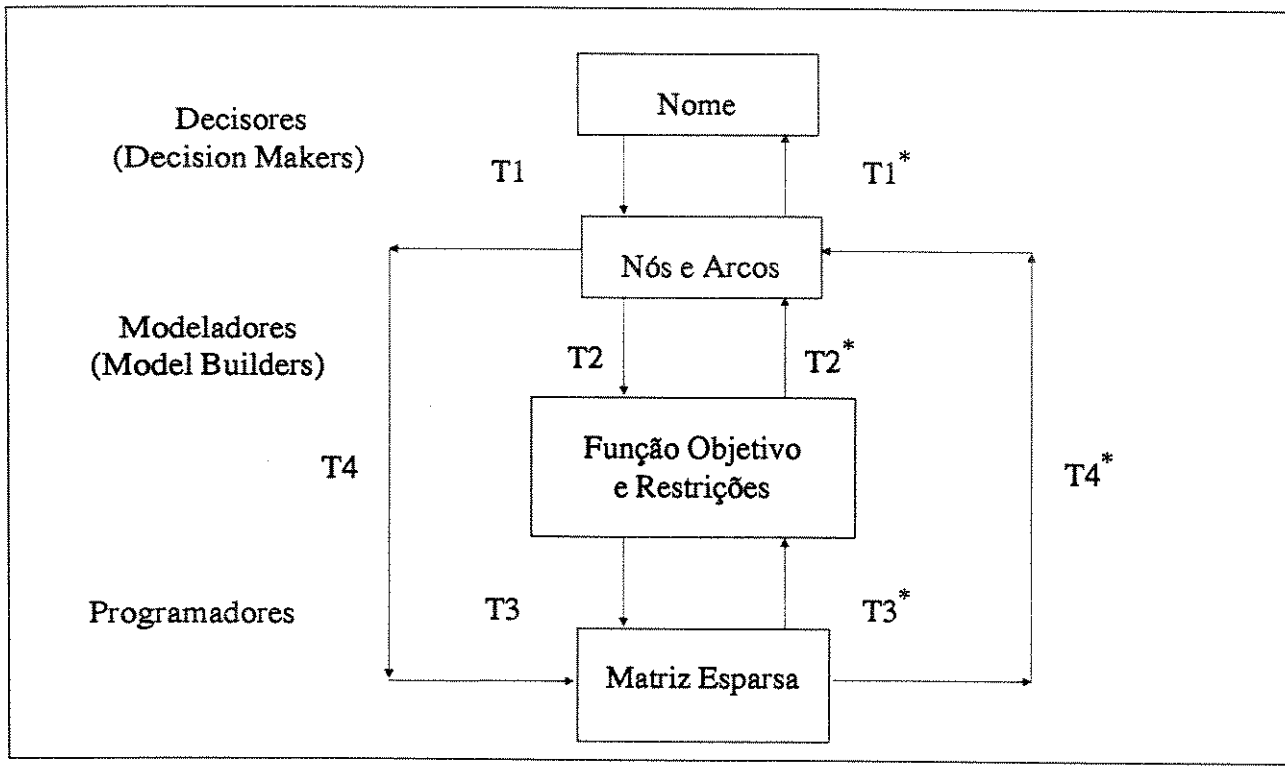


Figura 2.2 - Espaço de Abstração

2.2.4 - Modelamento Baseado no Conhecimento

O avanço da Inteligência Artificial (IA), tornando-se uma teoria mais popular, tem dado crédito à sistemas inteligentes em SSD's. Sistemas inteligentes são simulações em máquina do raciocínio humano dentro de um domínio de aplicação específico. Já existem aplicações de sistemas inteligentes em configuração de hardware, exploração geográfica e diagnóstico médico, entre outras.

Podemos esperar, portanto, que uma tendência crescente seja a utilização de IA com modelos e softwares de modelamento. Neste contexto, os sistemas inteligentes realizariam funções que tentariam auxiliar o usuário na construção, interpretação e entendimento dos modelos. Em grande parte, os sistemas inteligentes serviriam como ferramentas de aprendizado.

Um SGM deve possuir estas funções somadas as suas capacidades de descrições e solução de modelos, isto é, o SGM deve suportar um ambiente de modelamento que permita sistemas inteligentes construídos em conjunção com os modelos. Este ambiente é chamado de Sistema de Modelamento Baseado em Conhecimento (SMBC). O SMBC tenta codificar o seu conhecimento numa base de conhecimento e fazê-lo acessível a todos os usuários do sistema.

Duas importantes características nos sistemas baseados em conhecimento são o conteúdo do conhecimento e a estrutura do conhecimento. O conteúdo trata do que será colocado na base de conhecimento e a estrutura refere-se a maneira como o conhecimento será organizado. Num ambiente de modelamento, no mínimo 3 categorias de conhecimento são relevantes:

- i) Específico do problema: conhecimento do domínio no qual o decisor trabalha;
- ii) Específico do modelo: conhecimento sobre instâncias do modelo e técnicas de solução avaliadas num SGM;
- iii) Específico do mapeamento: conhecimento que ajuda a encaixar descrições de problemas com modelos e algoritmos.

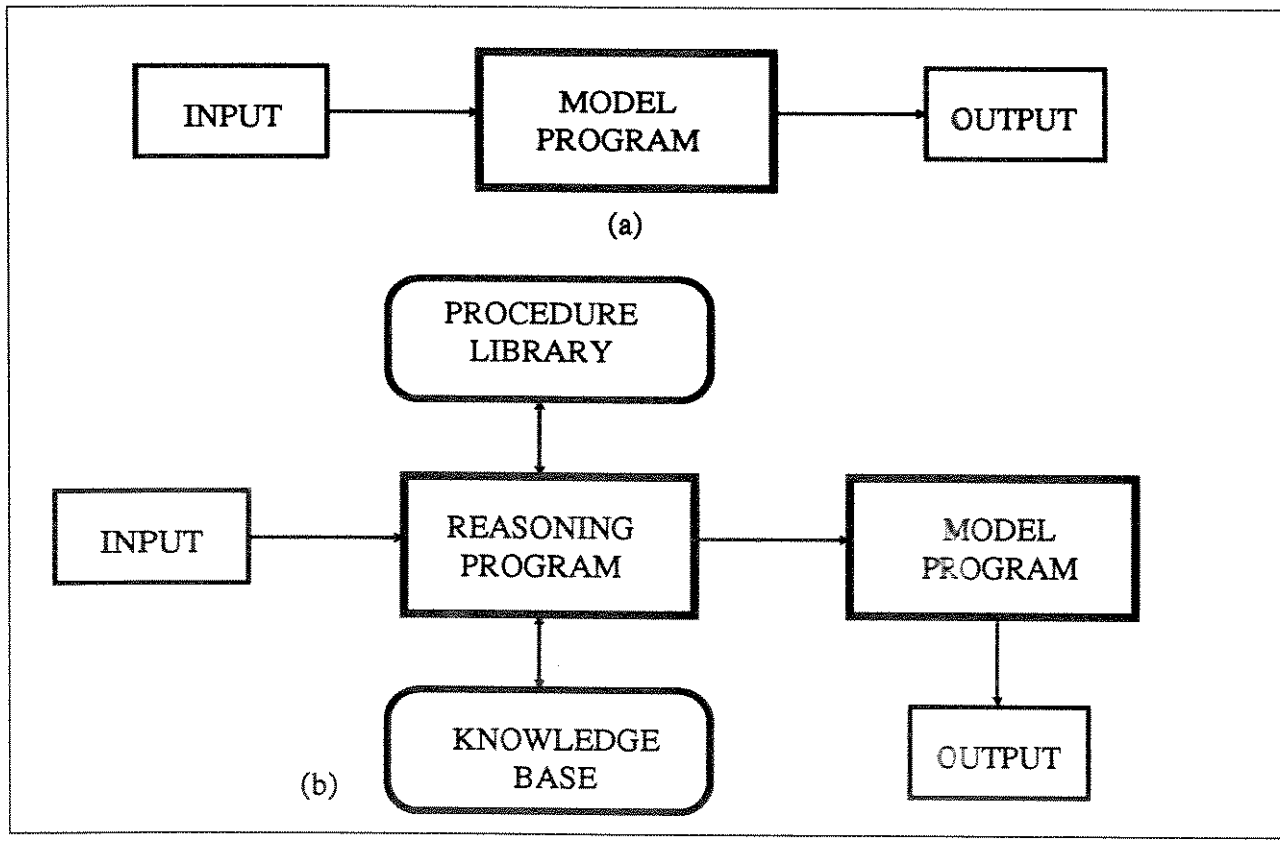


Figura 2.3 - a) Sistema de Modelamento b) Sistema Baseado em Conhecimento

A estrutura do conhecimento trata de como o conhecimento é representado e que tipos de técnicas de inferência podem ser aplicadas. O conhecimento pode ser representado em termos de lógica formal tais como o cálculo de predicados de 1ª ordem, utilizando como técnica de inferência o chamado *princípio de decisão* (resolution principle), por exemplo (Lucas, 1991).

2.2.5 - Representação de Modelos

Cada um dos objetivos de um SGM já descrito, enfatiza a necessidade da criação de um mecanismo genérico de representação de modelos. Em particular, a representação de um modelo deve ser independente de algoritmos e possuir múltiplos níveis de abstração, contudo deve manter a compatibilidade com um SGBD além de características baseadas em conhecimento.

O conceito de abstração é crítico em gerenciamento de modelos, principalmente se considerarmos modelos como abstrações e um SGM como definindo um espaço de abstração.

Para definirmos uma estrutura para representação de modelos devemos especificar as abstrações de maneira tão completa e consistente quanto possível.

As vantagens de representar uma estrutura de dados como abstração são as seguintes:

- uniformidade na maneira de ver as estruturas de dados.
- altos níveis de abstração podem ser construídos sobre abstrações de mais baixo nível.
- pode ser usada por todos os programadores num estilo consistente e correto.

Discutiremos uma representação de modelos que é similar em conceito a abstração de uma linguagem de programação. A abstração de modelo consiste de objetos, procedimentos e regras que são todas descritas em cálculo de predicado de 1ª ordem (Figura 2.4).

Os objetos referem-se as entidades participantes de um modelo. No complexo LP_EQN temos um conjunto de índices, variáveis de decisão, parâmetros e equações. Procedimentos representam operações sobre os objetos e são análogos as sub-rotinas num ambiente de programação com os objetos servindo de argumentos de entrada e saída. As regras representam a base de conhecimento associada com a abstração do modelo sendo descrito. Elas podem estabelecer integridade como, por exemplo, a linearidade requerida no modelo LP_EQN, além de descrever o processo de solução que, neste exemplo, envolve transformações de equação para matriz, aplicação de rotinas de solução e subseqüente extração das variáveis de decisão.

As vantagens de uma abstração de modelo vão de encontro aos objetivos de um SGM:

- i) A caracterização de modelos em termos de objetos, procedimentos e regras permite que modelos sejam descritos independentemente das representações requeridas pelos algoritmos. Isto proporciona um ambiente de modelamento geral onde muitos tipos diferentes de modelos são suportados ao mesmo tempo;

- ii) Abstrações de modelos são desenvolvidas para proporcionar diferentes visões, ou níveis de abstração, de um modelo;

- iii) A abstração de modelos é uma estrutura de dados que permite aos modelos aparecerem como dados em outras abstrações. Esta visão de modelos como dados facilita a implementação dentro do ambiente de um SGBD e garante compatibilidade com o mesmo;

- iv) Abstrações podem ser adaptadas para aplicações baseadas em conhecimento.

Model Abstraction LP-EQN ("Activity_Analysis")

Objects

INDEX_SET(j: activities)
INDEX_SET(i: resources)

DECISION_VARIABLE(X_j: levels)

PARAMETER(C_j: unit_contribution)
PARAMETER(B_i: availability)
PARAMETER(A_{ij}: amt_i_reqd_for_unit_j)

EQUATION(Total_contribution: SUM(j) [$C_j * X_j$]
EQUATION(Consumption_level: SUM(j) [$A_{ij} * X_j$] $\leq B_i$)
EQUATION(Nonnegativity: $X_j \geq 0$)

Procedures

SOLVE(lp-eqn,X_j)
SIMPLEX(lp-matrix) # solve matrix representation
T3(lp-eqn,lp-matrix) # transformation to matrix representation
$T3^*$(lp-matrix,X_j) # extract decision variable from solution matrix
LINEAR(equation,decision_variable) # is equation linear in decision variable ?

Assertions

(1)LINEAR(Total_contribution,X)
(2)LINEAR(Consumption_level,X)
(3)LINEAR(Nonnegativity,X) # all expressions must be linear in the decision variable
(4)SIMPLEX(lp-matrix) → XMAPS & XSLACK & XPRIML # XMP routines to do simplex
(5)SOLVE(lp-eqn: max(total_contribution) subject to Consumption_level & Nonnegativity , X_j) →
T3(lp-eqn,lp-matrix) & SIMPLEX(lp-matrix) & $T3^*$(lp-matrix,Xj)
# must transform to matrix representation before simplex

End LP-EQN

Figura 2.4 - Modelo de Abstração LP-EQN

2.3 - Descrição de Alguns Sistemas

2.3.1 - Gerador de Matrizes

Um sistema gerador de matrizes e gerador de relatórios (matrix-generator and report-writer - MGRW) é de muita importância em indústrias e organizações que utilizam a programação matemática como uma ferramenta de solução. Os MGRW são usados, sem exceção, ao lado de um software de PM.

Para entender o que é necessário para se construir um sistema MGRW, deve-se analisar as tarefas efetuadas pelo gerador de matrizes (matriz-generator - MG) e pelo gerador de relatórios (report-writer - RW). Um sistema típico de modelamento usando um MG, um OTIMIZADOR e um RW, como mostrado na Figura 2.5, trabalha da seguinte maneira:

- Os dados do problema (PROBLEM DATA) são apresentados na forma de tabelas;
- Estes dados são lidos e processados por um programa MG que produz um arquivo de entrada (INPUT FILE) que usualmente contém:
  - a) os nomes das variáveis lógicas ou nomes das linhas;
  - b) os nomes das variáveis estruturais ou nomes das colunas;
  - c) os coeficientes da matriz do problema;
  - d) as informações : RHS, BOUND e RANGES;
  - e) alguma informação sobre a solução inicial.
- Este arquivo é utilizado pelo OTIMIZADOR para obter a solução do problema;
- O sistema RW consulta as informações fornecidas pelo OTIMIZADOR e as apresenta numa forma consistente. O RW também pode acessar as informações do PROBLEM DATA e do INPUT FILE para auxiliar na elaboração do relatório da saída.

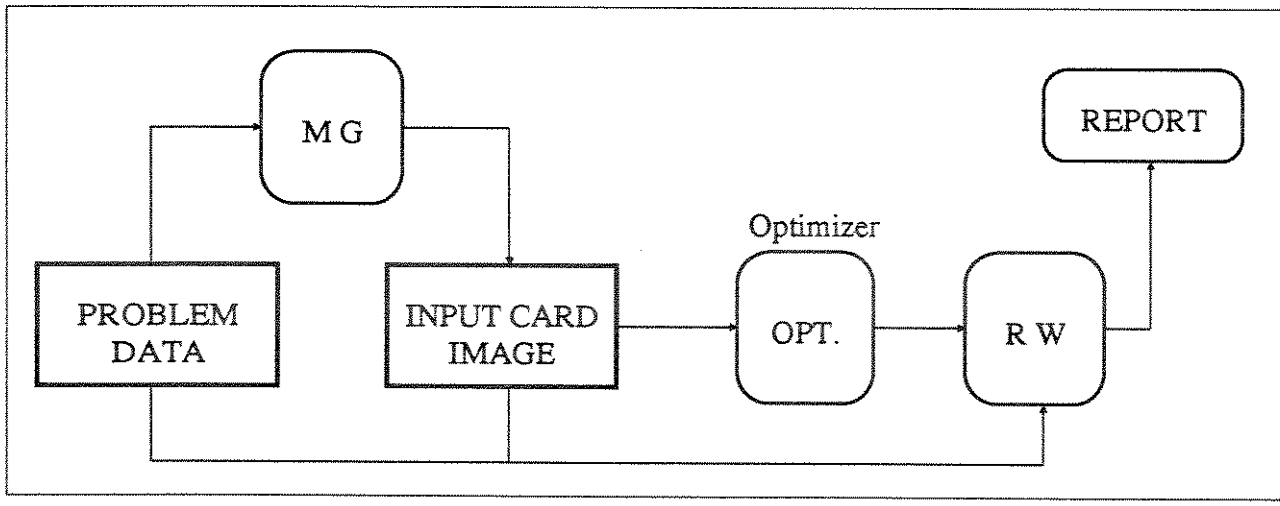


Figura 2.5 - Diagrama de Fluxo

Um sistema MGRW deve possuir, no mínimo, as seguintes características

- (i) Entrada de dados do problema (PROBLEM DATA) numa forma tabular;
- (ii) Construção de nomes de linhas e colunas através de expressões;

- (iii) Uso de constantes ou expressões aritméticas para especificar matrizes, BOUNDS, RHS, etc...;

- (iv) Acessar o arquivo de soluções para obter os valores das mesmas, bem como custos, faixas de atuação das variáveis, etc...;

- (v) Formatar e imprimir tabelas.

Estas características são comuns a todos os sistemas MGRW convencionais.

2.3.2 - Interface para Programação Matemática

Uma Interface para Programação Matemática (IPM) deve possuir, além das características de um MGRW, uma grande ênfase na estruturação dos dados de um problema matemático.

Numa IPM, as táticas de definição de um modelo são alteradas, pois devemos primeiro definir os conjuntos de índices para as variáveis. Feito isso podemos introduzir as variáveis de decisão bem como a matriz de coeficientes, utilizando os índices já definidos.

As estruturas, num IPM, são usadas de uma maneira comparável ao uso de sub-índices e conjuntos. Tendo como ponto de vista o processamento de informações, podemos entender estas estruturas como árvores ou redes. As estruturas são primeiramente introduzidas para um modelo em particular. A partir daí, as tabelas de entrada, as variáveis de decisão (colunas), as variáveis lógicas (linhas), e as tabelas de saída são todas definidas em função destas estruturas. Esta seqüência para a construção de um modelo obriga o modelador a pensar inicialmente na estrutura, deixando para a próxima etapa a definição do modelo.

Uma IPM é basicamente um compilador e executor de uma linguagem de alto nível. A linguagem, limitada na sua generalidade, deve possuir facilidades para a definição e manipulação de dados na forma tabular. Num PL, por exemplo, os elementos de uma estrutura podem ser usados para nomear linhas, colunas, limites, etc...

A seguir, será dada uma especificação para uma linguagem que poderia ser utilizada numa IPM.

2.3.2.1 - Sintaxe Preliminar

As seguintes notações metalinguísticas são introduzidas para definir uma linguagem para uma IPM.

(i) " ::= " é um conectivo que significa *é sintaticamente definido como.*

(ii) Os componentes opcionais da sintaxe estão colocados entre colchetes [...] , enquanto os componentes obrigatórios entre chaves {...} .

(iii) Alternativas podem aparecer entre colchetes e chaves e são escritas uma acima da outra.

(iv) As variáveis podem ser numeradas, caso ocorra uma repetição da mesma (para efeito de explicação das regras semânticas).

(v) Repetições indefinidas de uma variável são indicadas por três pontos dentro de um par de colchetes [...] ou [;...].

2.3.2.2 - Declaradores Simples

Os inteiros, reais, textos e estruturas são declarados pela expressão

LET VARIABLE { *lista de identificadores* } BE TYPE { STRUCTURE INTEGER REAL TEXT }

onde *lista de identificadores* é definida como:

lista de identificadores ::= *identificador* [;...]

e um *identificador* é definido como uma seqüência de caracteres alfanuméricos começando por um caracter alfabético.

Exemplo:

LET VARIABLE I1 BE TYPE INTEGER

LET VARIABLE Q, S1, S2, S3 BE TYPE STRUCTURE

A declaração de uma estrutura tem a forma:

LET STRUCTURE { *str - identificador* } BE { *lista de str - identificador* }

Exemplo:

LET STRUCTURE Q1 BE SA, SB, SC

LET STRUCTURE SA BE P1, P2, P3

LET STRUCTURE THEAD BE

LIST I: = 1 STEP 1 UNTIL 4 / 'TAIL*' , ANOTHER

As declarações acima podem ser representadas pelas 2 estruturas da Figura 2.6

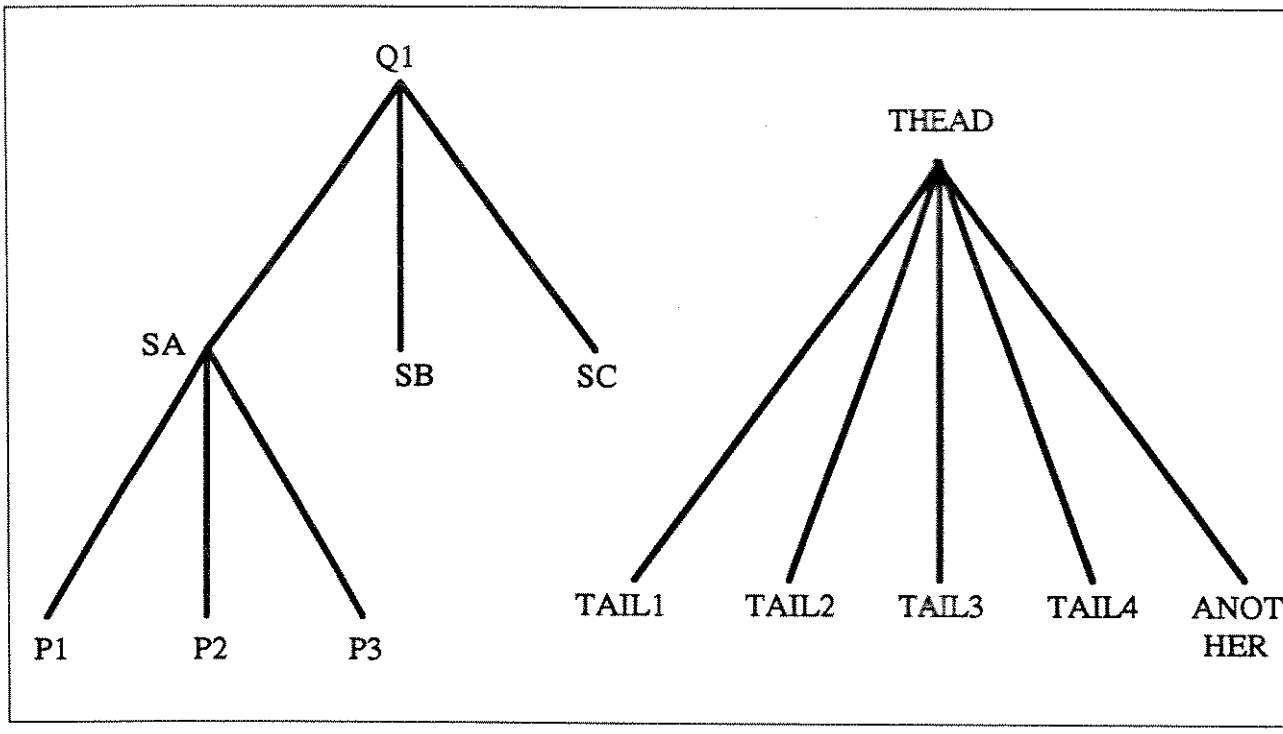


Figura 2.6 - Relacionamento das Estruturas

As seguintes propriedades são associadas com os elementos da estrutura e os identificadores da estrutura:

- (i) Um identificador de estrutura, referenciando-se a cabeça da estrutura contendo um conjunto de elementos na subestrutura inferior. Por exemplo N:THEAD tem o valor de 5.

- (ii) Um elemento tem uma ordinalidade com sua estrutura superior expressa por I:elemento IN identificador. Por exemplo I: SB IN Q1 tem valor 2.

- (iii) Para se associar strings com um elemento ou identificador utilizamos o prefixo T:. Por exemplo: T: SA := 'DOMESTIC'.

- (iv) A string ligada a um elemento ou identificador pode ser obtida colocando o mesmo entre colchetes.

2.3.2.3 - Declaração de Tabelas

A declaração de tabelas é feita do seguinte modo:

LET TABLE { *lista de identificadores* } BE { *estrutura* DOWN *estrutura* ACROSS *estrutura* DOWN BY *estrutura* ACROSS } TYPE { INTEGER REAL TEXT }

Colocando a declaração de tabelas numa forma mais clara temos:

LET TABLE	{ *lista de identificadores* }	BE	{*estrutura* DOWN *estrutura* ACROSS *estrutura* DOWN BY *estrutura* ACROSS }
		TYPE	{ INTEGER REAL TEXT }

Exemplo:

LET TABLE Exemplo BE THEAD DOWN BY Q1 ACROSS TYPE REAL

A declaração acima corresponde a Tabela da Figura 2.7

Exemplo		Q1 SA P1	P2	P3	SB	SC
THEAD	TAIL1					
	TAIL2					
	TAIL3					
	TAIL4					
	ANOTHER					

Figura 2.7 - Tabela THEAD por Q1

2.3.2.4 - Contexto e Enumeração

Para identificar uma determinada célula numa tabela, é necessário estabelecer um esquema para referenciar, através de linhas e colunas, um elemento de uma matriz. Por exemplo, a coluna 3 da tabela da Figura 2.7 é identificada por "P3 IN SA IN Q1" (contexto do elemento).

A sintaxe para definir um elemento num determinado contexto é:

Contexto do elemento ::= { elemento com nome variável tipo estrutura } [IN...]

Podemos efetuar atribuições à variável utilizando, por exemplo, a declaração Q := P3 IN SA IN Q1 , onde Q é uma variável do tipo estrutura definida previamente (através do operador LET, visto na seção 2.3.2.2) e P3, SA e Q1 são nomes de elementos de uma estrutura. Desta maneira, Q pode ser usada como uma referência para a coluna 3 da tabela da Figura 2.7. Portanto o elemento Exemplo(TAIL2 IN THEAD, Q) da Figura 2.7 está localizada na linha 2 e na coluna 3.

As formas enumerativas são utilizadas, geralmente, para especificar repetição. A forma geral para a enumeração de um elemento num contexto é :

Enumeração ::= do contexto de um elemento	*{ variável tipo estrutura*	*IN*	*contexto do elemento*
	variável tipo estrutura	*[IN contexto do elemento]*	*variável tipo estrutura*
	variável tipo estrutura	*[IN contexto do elemento]*	*enumeração do contexto de um elemento}*

Enumeração de elementos é utilizada nos declaradores FOR e SUM que possuem a seguinte forma :

FOR	*enumeração de um elemento no contexto*	*[declaradores em paralelo]*	*[quando declarador]*
SUM	*enumeração de um elemento no contexto*	*[declaradores em paralelo]*	*[quando declarador]*

onde

declaradores em paralelo tem a forma

AND FOR *enumeração do contexto de um elemento*

e

quando declarador tem a forma

WHEN *expressão condicional*

O declarador **FOR** enumera, portanto, a repetição de uma declaração. Um grupo de declarações pode ser repetida usando os declaradores DO, BEGIN e END. Por exemplo:

```
FOR ... DO
    BEGIN
        declaração 1
        declaração 2
            :
            :
        declaração n
    END
```

2.3.2.5 - Forma Linear e em Colunas

Os declaradores para o gerador de linhas e para o gerador de colunas, definidos na seção 2.3.2.6, utilizam a *forma linear* e a *forma em colunas* que são definidas nesta seção.

A *forma linear* é definida da seguinte maneira :

forma linear ::= [{ PLUS MINUS }] termo linear [{ PLUS MINUS } ...]

onde *termo linear* é definido como :

termo linear ::= { termo linear simples | termo linear composto }

Por sua vez, *termo linear simples* e *termo linear composto* são definidos como:

termo linear ::= simples	*{ expressão aritmética* TIMES *referência a uma variável do modelo* *referência a um modelo variável* TIMES *expressão aritmética* *referência a uma variável do modelo*	
termo linear ::= composto	*declaração de enumeração* **SUM**	*{ termo linear colchete esquerdo forma linear colchete direito }*

A *forma em colunas* é sintaticamente similar a *forma linear* e é definida como:

forma em colunas ::= termo coluna [ALSO ... *]*

onde *termo coluna* é definido como:

termo coluna ::= *{ termo coluna simples*
termo coluna composto }

Do mesmo modo, *termo coluna simples* e *termo coluna composto* são definidos da seguinte maneira:

termo coluna ::= simples *{ expressão aritmética* ON *referência a uma restriçõ do modelo*
referência a uma restrição do modelo }

termo coluna ::= composto *declarador de enumeração* **FOR** *{ termo coluna colchete esquerdo termo coluna colchete direito }*

2.3.2.6 - Gerando um Modelo Linear

Um grupo de variáveis e restrições de um modelo linear são definidos por um declarador que possui a seguinte forma:

LET CLASS	*{* MODVAR MODCON *}*	*lista de identificadores*	BE *estrutura -1*
	[BY *estrutura -2]*	*[tipo]*	*[atributo da solução]*

Na declaração acima, *tipo* define as variáveis do modelo ou os tipos de restrição.

Para as variáveis temos:

tipo ::= TYPE { NN MI FX FR }

onde

NN → variáveis não negativas (escolha default)

MI → não positivas, variável

FX → fixas, variável

FR → livres, variável

Para as restrições temos :

tipo ::= TYPE { LE GE EQ }

onde

LE → menor do que ou igual (escolha default)

GE → maior do que ou igual

EQ → igual

O *atributo da solução* tem a seguinte forma :

atributo da solução ::= ATTRIBUTE { X DJ ST }

onde

X → valor da solução

DJ → coeficiente dos custos reduzidos

ST → status da base

Os relacionamentos entre a linha e a coluna atual são especificados por um declarador do gerador de linhas ou um declarador do gerador de colunas.

Os declaradores possuem a forma :

declarador do ::= gerador de linha
ROW *referência a um restrição do modelo*
[tipo] [IS *forma linear] [rhs declarador [...]]*
[declarador de faixas [...]]
[declarador de pivo [...]]

onde

rhs declarador ::= RHS *nome da expressão* VALUE *expressão aritmética*

declarador de faixas ::= RANGE *nome da expressão* VALUE *expressão aritmética*

declarador de pivo ::=
VARIABLE *referência a uma variável do modelo*
PIVOTS *at* ROW [AT { LB UB }]

declarador do ::= gerador de coluna
COLUMN *referência a uma variável do modelo*
[TYPE { PL MI FX FR } *]*
[HAS *forma coluna]*
[declarador de valor de faixa]
[setor declarador de faixa]

onde

declarador ::=	*BOUND nome da expressão {*
de valor	*{ UP LO FX } VALUE expressão aritmética*
de faixa	*MI*
	FR }

setor declarador de faixa ::= IS SET TO { UB LB }

O *declarador de pivo* do gerador de linhas juntamente com o *declarador setor declarador de faixa* do gerador de colunas permitem ao modelador especificar um *ponto inicial* ou uma *base inicial*.

2.3.2.7 - Declarador para Imprimir Tabelas

Este declarador permite ao usuário especificar o formato de saída para os relatórios e possui a seguinte forma :

declarador ::=	PRINT TABLE
de impressão	*{ nome da tabela*
	código da tabela de atributos
	solução da tabela de atributos }

2.3.3 - LPFORM: Uma Interface Gráfica para Programação Linear

Programação linear (PL) é uma técnica de otimização muito utilizada e com diversas aplicações na indústria. Enquanto tem ocorrido um avanço considerável nas técnicas de solução, os métodos para construir e trabalhar com modelos de PL têm mudado pouco nos últimos 30 anos. Atualmente, formular programas lineares é uma arte que requer considerável experiência e cuidadosa atenção a detalhes. Grandes PLs consistem de milhares de variáveis e restrições e muitos sub-modelos. Um pequeno erro pode levar a uma solução ilimitada, a infactibilidades difíceis de se detectar e , pior ainda, a modelos corretos que produzem maus resultados.

A tomada de decisão pode ser dividida em três estágios da solução do problema: *inteligência*, *formulação* e *escolha*. A maioria da pesquisa tem se direcionado no estágio da escolha, pois de fato, a idéia da programação linear é auxiliar o usuário a fazer a *escolha ótima*. Já a pesquisa de métodos que auxiliem na descoberta de problemas e erros (inteligência) e na formulação de problemas têm tido menor desenvolvimento.

O LPFORM (Ma et al., 1989)se preocupa com os estágios da *inteligência* e *formulação* e pode ser descrito como uma interface que utiliza técnicas de inteligência artificial para auxiliar na formulação e manipulação de grandes problemas lineares. Nesta seção discute-se as principais características do LPFORM como forma de ilustrar os vários aspectos do desenvolvimento de uma interface para programação linear.

LPFORM ajuda o usuário através de regras de construção de modelos, desenvolvimento de esquemas de nomeação para linhas e colunas e organização dos dados. Isto pode ser utilizado para qualquer problema linear, mas é especialmente útil para grandes modelos contendo vários sub-modelos. A interface permite aos usuários descrever objetos da vida real e seus relacionamentos numa forma gráfica e definir problemas de PL em termos não matemáticos.

2.3.3.1 - Estágios na Solução de um PL

O processo de solucionar um PL pode ser decomposto em cinco estágios conceituais: *investigação do problema, formulação do modelo, manipulação dos dados, solução algoritmica* e *geração e análise do relatório*. Cada estágio envolve uma translação entre diferentes representações do modelo.

- Investigação do Problema: é um estágio informal devido a riqueza, variedade e ambiguidade do mundo real. Neste estágio a automatização é complicada e provavelmente a maior parte desta tarefa será humana.

- Formulação do Modelo: utiliza a informação do primeiro estágio para criar uma convenção simbólica para conjuntos, variáveis e coeficientes e , então, gerar uma formulação simbólica em notação algébrica e/ou em formato matricial.

- Manipulação de Dados: associa valores numéricos com a formulação algébrica e gera a entrada para a rotina de otimização utilizada. Duas tarefas devem ser realizadas neste estágio: a *coleta de dados* que requer experiência humana e a *geração do problema* que é um processo parcialmente mecanizado.

- Solução Algoritmica: foi o primeiro estágio a ser computadorizado. Corresponde ao *otimizador* , ou seja, a rotina utilizada para solucionar o modelo. Normalmente é um estágio livre de problemas devido aos progressos nesta área e aos avanços na tecnologia dos computadores.

- Geração e Análise do relatório: um sistema de geração de relatórios consulta as informações fornecidas pelo *otimizador* e as apresenta numa forma consistente. Os sistemas de geração e análise de relatórios foram descritos na seção 2.3.1.

A evolução dos softwares de PL podem ser descritos da seguinte maneira:

- Otimizadores utilizam como entrada matrizes dos problemas de PL e fornecem como saída a solução do problema. Podemos utilizar formatos padrão de entrada como o MPSX (IBM Corporation, 1975) ou APEX (Control Data Corporation, 1974.). Estes padrões de entrada exigem a definição dos coeficientes diferentes de zero bem como sua linha e coluna. Enquanto este processo é eficiente em termos computacionais, pode-se dizer que é lento e suscetível a erros se tomarmos o ponto de vista humano.

- O segundo grupo de softwares são os geradores de matrizes que auxiliam na automatização de tarefas como a construção da matriz de entrada para o otimizador. A translação do

problema definido para o gerador de matrizes é feito através de programas computacionais escritos na linguagem do gerador de matrizes. Então, o usuário que desejar trabalhar com um gerador de matriz deve aprender uma linguagem especializada como por exemplo OMNI (Haverly Systems Inc., 1977) ou DATAFORM (Creegan, 1985).

- O terceiro grupo de sistemas possue facilidades de manipulação de dados e/ou uso de representações algébricas dentro de uma linguagem de modelagem. A manipulação de dados deve permitir a edição de dados, recuperação de informações (utilizando uma base de dados), edição de modelos, geração de relatórios e análise de soluções. Exemplos destes sistemas são: AMPL (Fourer e Kernighan, 1987), PAM (Welch, 1987), GAMS (Meeraus, 1984) e PLATAFORM (Palmer et al., 1984).

É ineficiente para um modelador trabalhar com a matriz completa de um problema durante todas as fases da solução de um modelo, pois a matriz não possui um nível de abstração para desenvolver os estágios iniciais da modelagem. Um conjunto de regras algébricas é útil para expressar versões do modelo e para o entendimento da estrutura geral do modelo completo. Além disso, as regras algébricas são consistentes, proporcionam uma boa documentação e são independentes de valores particulares de dados. Entretanto, temos três pontos negativos. Primeiro , a dificuldade de especificar grandes modelos corretamente (devemos garantir, por exemplo, que os índices estão corretos e que todas as restrições foram definidas). Segundo, não temos uma visualização imediata do modelo. Finalmente e mais importante, somente é compreendido por um usuário que esteja familiarizado com PL. Uma interface gráfica representaria entidades do mundo real (estoques, demandas, fluxos) e entidades abstratas (tal como atividades de PL) através de icons. Com isso teremos um sistema auxiliado por computador para modelagem matemática.

2.3.3.2 - Arquitetura do Sistema

A Figura 2.8 mostra a arquitetura do sistema LPFORM. O diagrama mostra os estágios da formulação e solução de problemas de PL descritos na seção anterior.

O sistema é composto de cinco sistemas abertos interligados através de arquivos de comunicação.

- 1. Sistema LPFORM.
- 2. Um gerador de matriz similar ao GAMS.
- 3. Um software para resolução de programação linear e inteira (LINDO - Schrage, 1987).
- 4. Um sistema de gerenciamento de banco de dados similar ao SQL da IBM (Astrahan et al., 1985).
- 5. Um analisador de matrizes e soluções (ANALYZE - Greenberg, 1983).

2.3.3.3 - Interface

A interface do LPFORM fornece as seguintes características:

- Um representação gráfica não matemática para problemas de PL.
- Suporte para diferentes estratégias de solução de problemas que reduzem a complexidade do processo de formulação.
- Capacidades que fornecem uma maior funcionalidade, tais como gerenciador de dados, checagem de inconsistências e gerenciamento de modelos.

Grande parte da experiência humana em formular modelos matemáticos é gasta na translação entre a realidade e os objetos matemáticos. Isto envolve conhecimentos de semântica e sintaxe que podem levar anos para serem adquiridos. A primeira tarefa, portanto, para desenvolver um sistema que auxilie na formulação de PLs é estabelecer um vocabulário (interface de alto nível) para descrever o mundo real.

No LPFORM, a representação dos modelos faz uso de gráficos para descrever as dimensões do problema, bem como se utiliza de icons para representar atividades, estoques, fluxos e outros conceitos. Para definir um problema de PL não é necessário seguir uma seqüência fixa de passos ou identificar objetos matemáticos como variáveis, restrições e funções objetivo. De fato, os usuários podem pensar em termos reais e colocar seus problemas numa forma próxima àquelas usadas por modeladores e matemáticos.

Com o intuito de diminuir o tempo gasto para programar grandes e complexos modelos de PL, o sistema incorpora técnicas de formulação *top-down* e *bottom-up*. A técnica top-down pode ser aplicada quando definimos estruturas hierárquicas permitindo que os objetos sejam especificados com incrementos no nível de detalhes (objetos herdam características de seus pais). Por outro lado, o usuário pode querer estudar formulações de sub-modelos após o modelo completo ter sido definido, entrando, portanto, neste ponto, técnicas de formulação bottom-up.

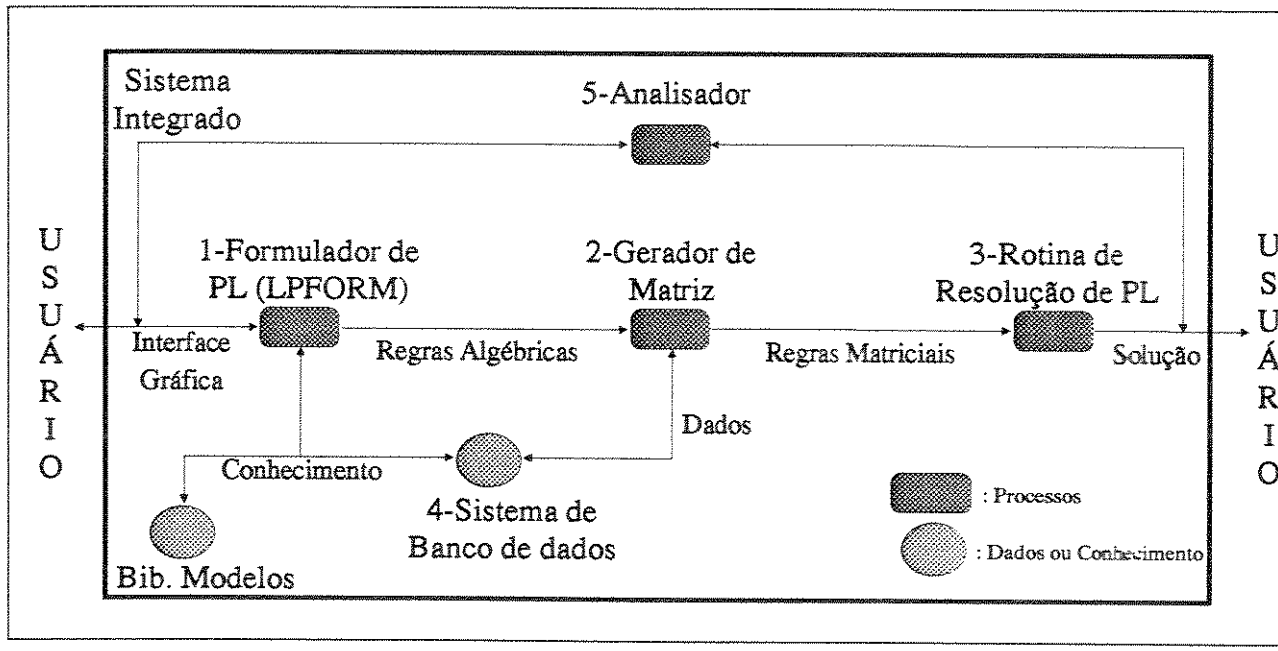


Figura 2.8 - Diagrama Integrado do Sistema

Há várias maneiras de especificar o mesmo problema no LPFORM. Por exemplo, um problema de grande porte pode ser definido em detalhes (*estratégia dos primeiros passos*) desde os seus primeiros passos. Nesta abordagem, o usuário especifica o modelo em termos de objetos reais tais como produção, estoque e atividades de transporte. Uma estratégia alternativa seria utilizar o *planejamento de modelo* no qual o modelo definido através de combinações de outros modelos previamente definidos. Logo, o planejamento de modelos pode ser utilizado para definir a maior parte do novo modelo e a estratégia dos primeiros passos pode ser utilizada para acrescentar novas características ao modelo (restrições, atividades, etc...).

O perigo desta estratégia de definição está no fato dos usuários acharem-na confusa e complicada. Para resolver este impasse deve-se elaborar um sistema suficientemente inteligente que seja capaz de corrigir e inferir as intenções do usuário.

LPFORM aceita uma entrada no formato gráfico e produz uma especificação algébrica como entrada para um gerador de matrizes. Durante este processo, muitas outras representações são geradas a fim de aumentar o entendimento e facilitar a documentação. A primeira é uma lista de cada atividade do modelo e sua respectiva descrição (principalmente entradas e saídas). A segunda consiste numa representação algébrica onde os termos são dispostos em colunas e linhas a fim de mostrar a estrutura do modelo. Finalmente, um dicionário de dados é gerado possibilitando um reconhecimento rápido das componentes do modelo bem como um melhor entendimento para os usuários.

Uma das condições necessárias para qualquer sistema computadorizado é acrescentar novas características (capacidades) e/ou incrementar a eficiência e performance. Com esta premissa em mente, foram acrescentadas ao sistema um banco de dados relacional, uma checagem de inconsistências e um gerenciador de modelos.

- Banco de Dados Relacional: permite um acesso direto aos dados da aplicação e aos conhecimentos a ele inerentes. Os usuários podem aprender sobre os dados do sistema e especificar buscas como parte do processo de formulação do modelo.
- Checagem de Inconsistências: a prática usual utilizada em PL é primeiramente gerar um teste na forma de uma matriz numérica e então analisar os problemas obtidos. O sistema LPFORM utiliza uma abordagem diferente, pois as inconsistências são checadas na própria formulação do LPFORM (a matriz é diretamente dependente desta formulação). Checagem no tablô ainda são necessárias, porém o número de erros será menor, reduzindo portanto, o tempo gasto para se elaborar uma formulação correta.
- Gerenciamento de Modelos: o dicionário de dados de cada modelo é armazenado juntamente com a estrutura matemática, formando uma *base de modelos* que pode ser aumentada e personalizada pelos usuários. A base de modelos permite , por exemplo, que um usuário recupere e inspecione todos os modelos que tenham um dado em comum ou variáveis que utilizam os mesmos recursos.

2.4 - Conclusão

Neste capítulo vimos as características básicas dos ambientes para programação matemática bem como a descrição de alguns sistemas. Como podemos perceber, há uma crescente pesquisa e desenvolvimento nas áreas relativas a interface com o usuário permitindo que este formule, altere e analise um modelo matemático da melhor forma possível. Esta tendência tem base no aumento do número de usuários de programação matemática que não possuem o conhecimento necessário para trabalhar com um *otimizador* , bem como no avanço dos recursos computacionais. Estes sistemas normalmente priorizam a facilidade de uso, utilizando para isto técnicas de gerenciamento de banco de dados, inteligência artificial, gerenciamento de modelos e recursos gráficos. Outra característica é a combinação destes sistemas com outros sistemas abertos afim de fornecer um conjunto final mais eficiente e robusto.

Capítulo 3
MC++: Software para Programação Matemática Multiobjetivo

3.1 - Introdução

O software MC++ pode ser descrito como um **Sistema de Suporte a Decisão Baseado em Programação Multiobjetivo**, no qual o usuário modela problemas da vida real dentro de um ambiente matemático.

É importante salientar que esta versão do MC++ considera problemas lineares. Conforme demonstrado no Capítulo 1, a abordagem de problemas multiobjetivos convexos no espaço dos objetivos torna desnecessária a consideração de modelos não lineares no nível de decisão. Neste nível, o problema é sempre linear, independentemente da natureza do problema original. Entretanto, a solução do nível de análise em geral envolve técnicas de programação não linear que dependem dos modelos considerados e que portanto são de difícil sistematização em um único ambiente. Além disso, a descrição desta classe de modelos requer interfaces com o usuário muito mais sofisticadas. Estes aspectos serão tratados em futuras versões do MC++.

O objetivo principal do software é auxiliar o usuário na modelagem de problemas multiobjetivo, fornecendo para isso ferramentas de suporte a decisão e características de sistemas baseados em conhecimento.

A maioria dos sistemas para programação matemática tem por objetivo principal **somente** a resolução do problema em questão, deixando a desejar quanto a interface e facilidade de uso. Isto provoca uma certa relutância por parte dos interessados em programação matemática, pois estes têm de se adaptar as características de cada software.

O MC++ não prioriza somente a resolução do problema, mas fornece um ambiente de trabalho amigável onde o usuário possa definir, alterar, testar e combinar modelos matemáticos, sendo a resolução do problema apenas uma opção do software.

3.2 - Características

O MC++ foi implementado em C++ e Xwindows utilizando para isto o ambiente gráfico OPENWINDOWS. A máquina utilizada foi uma estação de trabalho SPARC SUN (SUN workstation) rodando o sistema operacional SUN-OS (UNIX).

A linguagem C++ foi escolhida por ser uma linguagem orientada a objetos, o que permite a elaboração de uma abstração de dados (estrutura de dados) mais clara e de fácil manuseio, permitindo portanto um melhor ambiente de programação e manutenção do sistema. Maiores detalhes sobre as classes (objetos) implementados serão discutidos na seção 3.6 e no Apêndice A.

Juntamente com a linguagem C++, foi utilizada a programação Xwindow (Xview, Xlib, Xintrinsics, X11, etc..), que permitiu a elaboração de um ambiente gráfico característico do OPENWINDOWS. Isto facilita o trabalho dos usuários já acostumados com o ambiente OPENWINDOWS pois este segue um padrão pré-estabelecido, além de proporcionar um ambiente amigável para os iniciantes.

Além disso, a programação em Xwindow é orientada a eventos permitindo que uma ação iniciada numa janela gere um evento que será refletido num outro ponto do programa. Estas características serão descritas com mais detalhes na seção 3.6.

O sistema operacional multitarefa (UNIX) também proporciona maiores facilidades como por exemplo a execução de vários processos ao mesmo tempo (Ex: resolução de um modelo enquanto se executa um entrada de dados).

Por ser um sistema de suporte a decisão, o MC++ considera a resolução do problema apenas uma opção dentre as muitas fornecidas, pois o objetivo do software é auxiliar em todas as fases da modelagem de um problema. As definições de índices, variáveis, parâmetros, objetivos, restrições, bem como a visualização, interação e tratamento dos modelos têm a mesma importância que a resolução em si (não significando que os métodos matemáticos de resolução foram relegados a segundo plano).

No tocante a resolução dos problemas matemáticos foi dado enfoque somente aos métodos multiobjetivos, utilizando-se softwares de sistemas abertos para os demais métodos de programação matemática (como exemplo podemos citar o método Simplex).

A estratégia de reutilização de software de sistemas abertos foi utilizada por dar maior robustez na resolução dos problemas matemáticos, bem como permitir uma concentração de esforços no desenvolvimento do sistema de suporte a decisão, que era o objetivo principal a ser atingido.

3.3 - Principais Componentes

3.3.1 - Definições

Antes de iniciar a descrição dos principais componentes do MC++, deve-se conhecer a terminologia básica e as abstrações computacionais utilizadas na implementação do software.

Definições:

Escopo: Faixa de variação para um índice. Pode ser de dois tipos: inteiro ou string (caractere). O tipo inteiro varia entre dois valores numéricos definidos sendo que os limites inferior e superior são dependentes da máquina utilizada . O tipo string varia dentro de uma lista de strings definida, não havendo um limite máximo para o tamanho da lista (o limitante superior é a memória da máquina). Ex: I = [1..10], K = (jan fev mar abr mai jun).

Índice: Todo índice deve ter um escopo associado para indicar a sua faixa de variação. A idéia utilizada é que o índice definido varia dentro de um conjunto, o qual é dado pelo escopo. Os índices são utilizados para definir o número de variáveis e parâmetros. Ex: i com escopo em I implica que o índice i assume valores de 1 a 10, k com escopo K tem interpretação análoga a i.

Variável: Variáveis de decisão, nas quais são armazenados os valores resultantes da solução de um modelo. Toda variável deve ter associada ao menos um índice (uma variável pode ter no máximo 5 índices). Ex: x_i que significa a existência de 10 variáveis denominadas x_1 , x_2 , .. , x_{10}; y_k que implica nas variáveis y_{jan} , y_{fev} , y_{mar} , y_{abr} , y_{mai} , y_{jun}; $z_{i,k}$ que implica numa matriz de 60 elementos como $z_{1,jan}$, $z_{3,fev}$, $z_{10,ago}$, etc...

Parâmetro: Valores que se associam as variáveis para *dosar* o seu significado. Todo parâmetro deve ter ao menos um índice associado (máximo de 5). Ex: $A_{i,k}$, C_i.

Primitivas: Também chamada de primitivas básicas. As primitivas são os elementos básicos que compõem um modelo, ou seja, os escopos, os índices, as variáveis e os parâmetros.

Objetivo: Função matemática formulada utilizando parâmetros e variáveis. Um objetivo pode ser maximizado ou minimizado e sua formulação deve ser linear. Ex: max [SUM_i (C_i x_i)].

Restrição: Função matemática formulada utilizando parâmetros e variáveis. Uma restrição deve ser limitada por sinais de $\leq$, = , $\geq$. Existe dois tipos de restrição: funcional e canalizada, sendo que ambas têm formulação linear. Ex: restrição funcional [SUMi Ai,k zi,k)] $\leq$ 10; restrição canalizada $0.5 \leq x_i \leq 5.3$.

Modelo: Conjunto de objetivos e restrições utilizado para representar matematicamente uma situação real. O modelo matemático pode se apresentar sob duas formas: **simbólica** e **matricial**. A forma simbólica exibe objetivos e restrições em notação algébrica, enquanto que a forma matricial

apresenta as mesmas informações sob o ponto de vista matricial. Esta diferença ficará mais clara na descrição do item 3.3.5.

3.3.2 - Tela Principal

A tela inicial do MC++ (Figuras 3.1 e 3.2) possui um área central destinada a apresentação do modelo matemático nas suas formas simbólicas e matricial. Logo acima temos a **barra de menu** e a **barra de cabeçalho**. Abaixo da área central temos a **barra de rodapé**. A barra de cabeçalho contém o nome do programa enquanto que a barra de rodapé contém a esquerda o nome do modelo corrente e a direita o copyright.

A barra de menu é formada por 9 opções: **Arquivos, Primitivas, Visão, Modelos, Estrutura, Relatório, Opções, Tutorial** e **Copyright**. Estas opções, com exceção de **Tutorial** e **Copyright**, também são apresentadas ao se clicar o botão direito do mouse na área central.

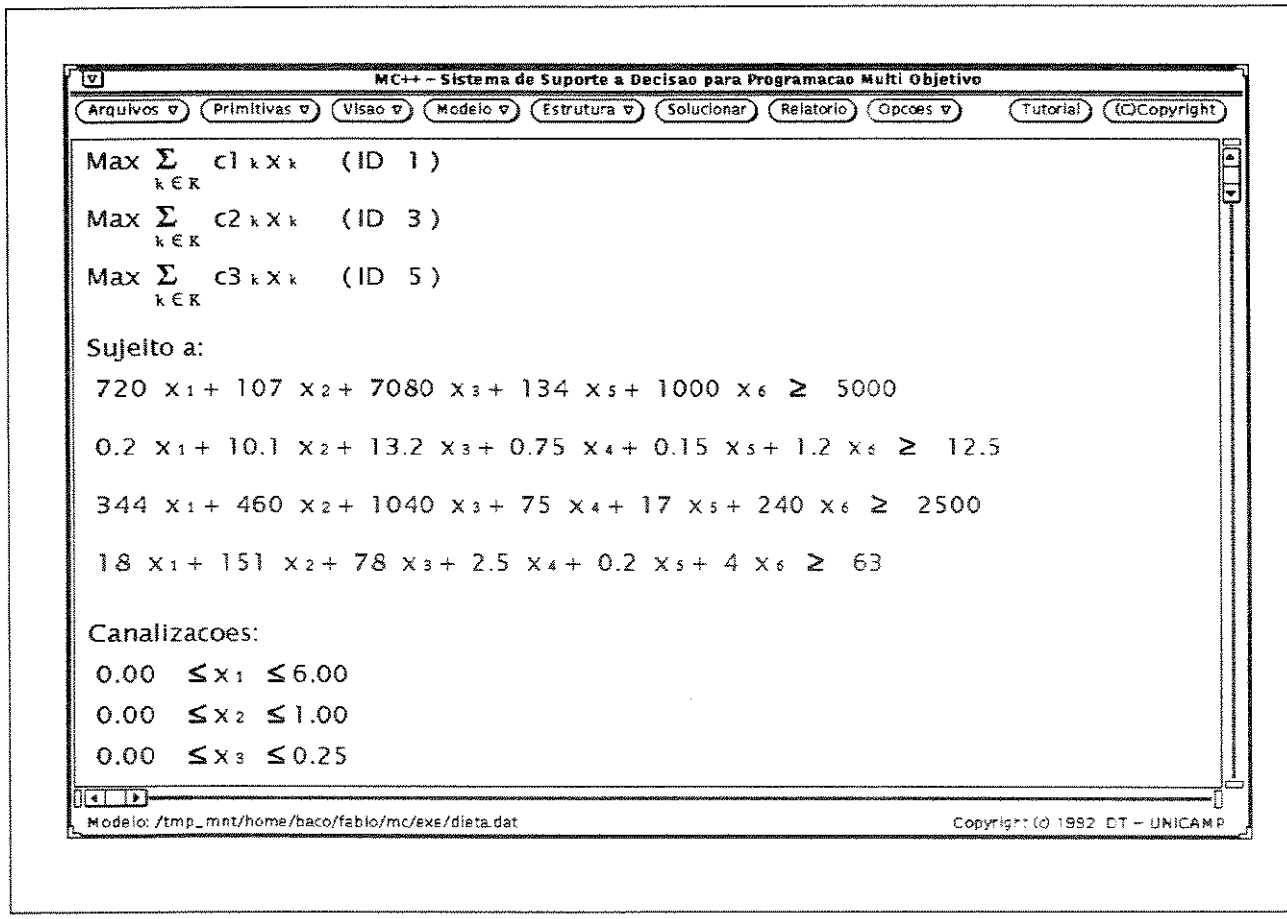


Figura 3.1 - Tela Principal

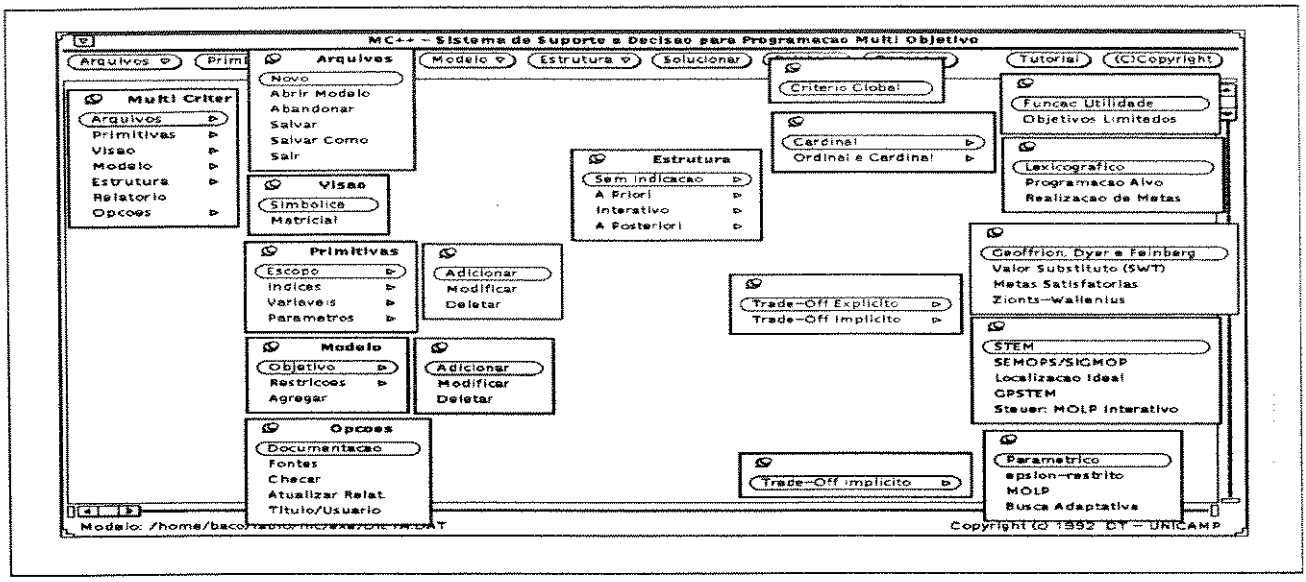


Figura 3.2 - Tela Principal com Menu Alternativo

3.3.3 - Menu Arquivos

O menu **Arquivos** (Figura 3.3) apresenta um menu *pulldown* contendo as seguintes opções: **Novo, Abrir, Salvar, Salvar Como, Abandonar** e **Sair**. A função do menu Arquivos é efetuar as operações de leitura e escrita em disco dos modelos definidos. A seguir será descrita a função de cada opção do menu **Arquivos**.

- **Novo**: retira da memória o modelo corrente e prepara o MC++ para trabalhar com um novo modelo. Ao se descartar o modelo corrente, este é automaticamente substituído pelo modelo *SemNome*. Este é o nome default utilizado pelo MC++ e portanto deve ser substituído por um outro nome através da opção **Salvar Como**.
- **Abrir**: carrega um modelo armazenado em disco para a memória. Esta opção apresenta uma tela de diálogo onde deve ser especificado o diretório e o nome do arquivo que contém o modelo com o qual se quer trabalhar.
- **Salvar**: salva em disco o modelo corrente. O nome do modelo e o diretório onde será armazenado são exibidos na barra de rodapé a esquerda.
- **Salvar Como**: renomeia o modelo corrente e depois salva o seu conteúdo em disco, isto é, salva o modelo corrente com outro nome. Esta opção apresenta uma tela de diálogo onde deve ser especificado o diretório e o nome do arquivo, isto é, o local onde se deseja salvar o modelo.
- **Abandonar**: abandona as alterações efetuadas no modelo corrente a partir da última vez em que este foi salvo. Esta opção fornece um **undo** (retorno a versão anterior) limitado onde se pode abandonar alguma alteração indesejada feita no modelo corrente.
- **Sair**: termina a execução do MC++.

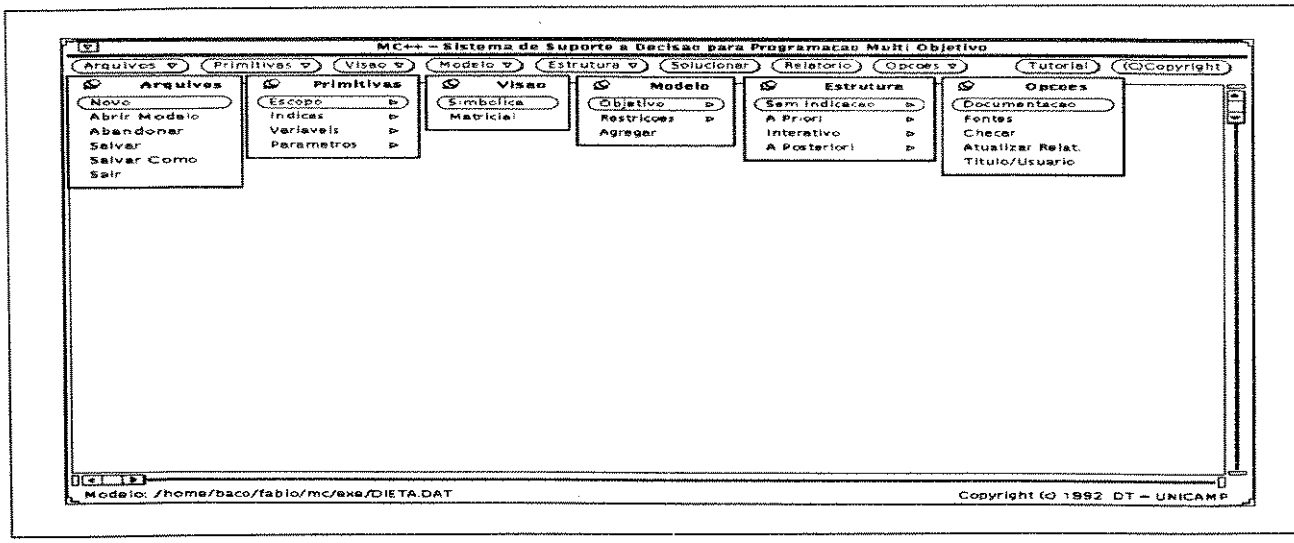


Figura 3.3 - Menu Arquivos

3.3.4 - Menu Primitivas

O menu **Primitivas** apresenta um menu *pulldown* contendo as seguintes opções: **Escopos, Índices, Variáveis** e **Parâmetros**. A função do menu Primitivas é definir (inserir), modificar e apagar escopos, índices, variáveis e parâmetros. Na definição de um modelo novo, este deve ser o primeiro menu a ser visitado pois todo modelo é dependente das primitivas básicas.

Independentemente da opção escolhida, o processo para o tratamento (inclusão, alteração e deleção) das primitivas é o mesmo. Em função do botão do mouse clicado, é apresentado um submenu (tipo *pullright*) com as opções **Inserir, Modificar** e **Deletar** ou uma tela de diálogo para o tratamento das primitivas. O botão esquerdo do mouse mostra a tela de diálogo para **inserção** da primitiva enquanto que o botão direito apresenta o submenu. A opção inserir abre a tela de diálogo para **Inserção** de uma primitiva; a opção modificar abre a tela de diálogo para **Modificação** de uma primitiva; a opção deletar abre a tela de diálogo para **Remoção** de primitiva.

Tela de Diálogo para Escopos

Na tela de diálogo para escopos (Figura 3.4) temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de escopos, enquanto que na barra de rodapé temos o número de escopos existentes.

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir as características do escopo. O escopo possui um nome que deve ser único e um tipo que pode ser numérico ou caractere. Caso o escopo seja do tipo numérico deve-se definir os limites inferior e superior. Se o escopo for do tipo caractere, deve-se definir uma lista de expressões string.

O botão **Inserir** define um novo escopo, o botão **Modificar** altera um escopo existente, o botão **Carregar** carrega os dados de um escopo a partir do seu nome, o botão **Deletar** remove um

escopo existente. Observe que antes de modificar ou deletar um escopo, deve-se carregá-lo ou entrar com o nome do mesmo.

Na inserção de Escopos, os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados. Na modificação de Escopos, os botões **Inserir** e **Deletar** do quadro de diálogo estão desabilitados. Na deleção de Escopos, os botões **Inserir** e **Modificar** do quadro de diálogo estão desabilitados.

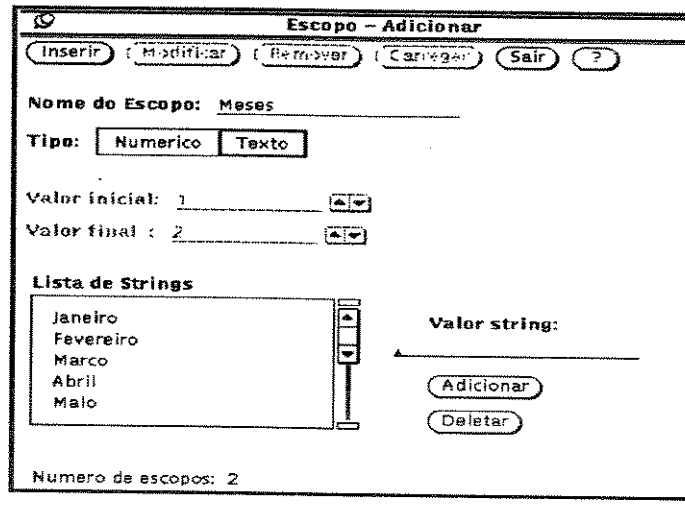


Figura 3.4 - Tela de Diálogo para Escopo (Inserção)

Tela de Diálogo para Índices

Na tela de diálogo para índices (Figura 3.5) temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de índices, enquanto que na barra de rodapé temos o número de índices existentes.

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir as características do índice. Na definição de um índice devemos estabelecer o seu nome (que deve ser único), o escopo associado e a sua documentação.

O botão **Inserir** define um novo índice, o botão **Modificar** altera um índice existente, o botão **Carregar** carrega os dados de um índice a partir do seu nome, o botão **Deletar** remove um índice existente. Observe que antes de modificar ou deletar um índice, deve-se carregá-lo ou entrar com o nome do mesmo.

Deve-se notar que os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados quando efetuamos uma inserção de índice. Na modificação de índices, os botões **Inserir** e **Deletar** do quadro de diálogo estão desabilitados. Já na deleção de índices, temos os botões **Inserir** e **Modificar** do quadro de diálogo desabilitados.

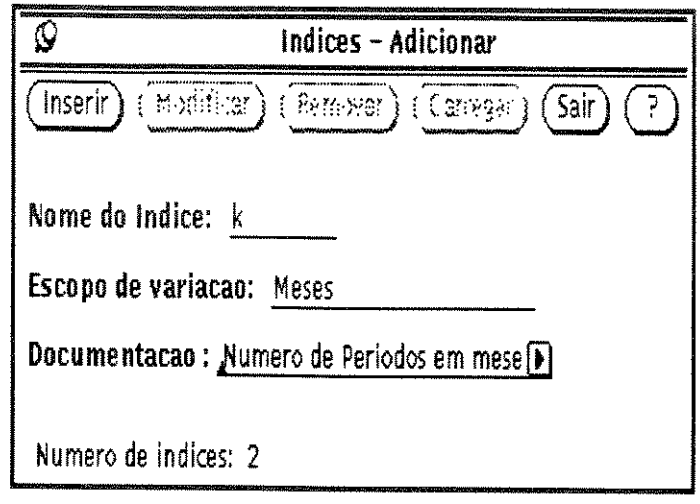


Figura 3.5- Tela de Diálogo para Índices (Inserção)

Tela de Diálogo para Variáveis

Na tela de diálogo para variáveis (Figura 3.6), temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de variáveis, enquanto que na barra de rodapé temos o número de variáveis existentes.

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir as características da variável. Na definição de uma variável devemos estabelecer o seu nome (que deve ser único), os índices associados e a sua documentação. Uma variável deve ter no mínimo um e no máximo cinco índices associados.

O botão **Inserir** define uma nova variável, o botão **Modificar** altera uma variável existente, o botão **Carregar** carrega os dados de uma variável a partir do seu nome, o botão **Deletar** remove uma variável existente. Observe que antes de modificar ou deletar uma variável, deve-se primeiramente carregá-la ou entrar com o nome da mesma.

Na inserção de variáveis, temos os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados. Na modificação de variáveis, temos os botões **Inserir** e **Deletar** do quadro de diálogo estão desabilitados. Finalmente na deleção de variáveis, os botões **Inserir** e **Modificar** do quadro de diálogo estarão desabilitados.

Tela de Diálogo para Parâmetros

Na tela de diálogo para parâmetros (Figura 3.7), temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de parâmetros, enquanto que na barra de rodapé temos o número de parâmetros existentes.

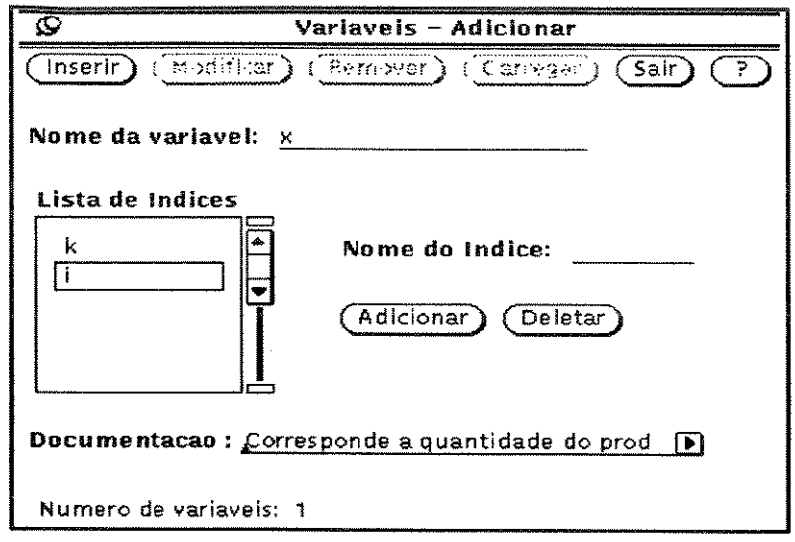


Figura 3.6 - Tela de Diálogo para Variáveis (Ins.)

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir as características do parâmetro. Na definição de um parâmetro devemos estabelecer o seu nome (que deve ser único), os índices associados e a sua documentação. Um parâmetro deve ter no mínimo um e no máximo cinco índices associados.

O botão **Inserir** define um novo parâmetro, o botão **Modificar** altera um parâmetro existente, o botão **Carregar** carrega os dados de um parâmetro a partir do seu nome, o botão **Deletar** remove um parâmetro existente. Observe que antes de modificar ou deletar um parâmetro, deve-se carregá-lo ou entrar com o nome do mesmo.

Na inserção de parâmetros, notamos que os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados. Já na modificação de parâmetros temos os botões **Inserir** e **Deletar** do quadro de diálogo estão desabilitados. Na deleção de parâmetros, os botões **Inserir** e **Modificar** do quadro de diálogo estarão desabilitados.

Devemos notar que nesta tela de diálogo existe a opção **Entrar com os dados ? [Sim] [Não]**. Caso seja escolhida a resposta positiva, uma nova tela de diálogo para a entrada dos valores do parâmetro em questão será apresentada (Figura 3.8).

3.3.5 - Menu Visão

O menu **Visão** apresenta um menu *pulldown* contendo as seguintes opções: **Simbólica** e **Matricial** (Figura 3.2). A função do menu Visão é apresentar o modelo corrente na área principal do MC++. A seguir será descrita a função de cada opção do menu **Visão**.

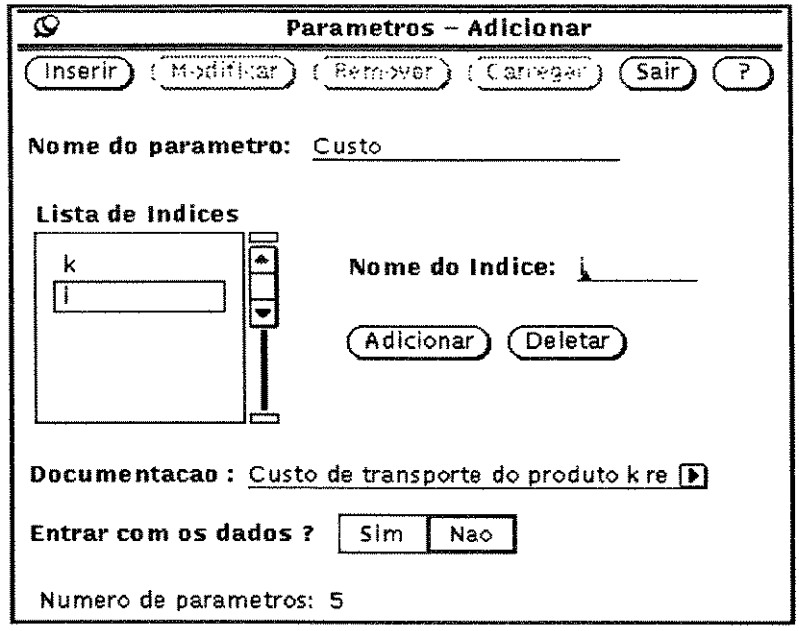


Figura 3.7 - Tela de Diálogo para Parâmetros (Ins.)

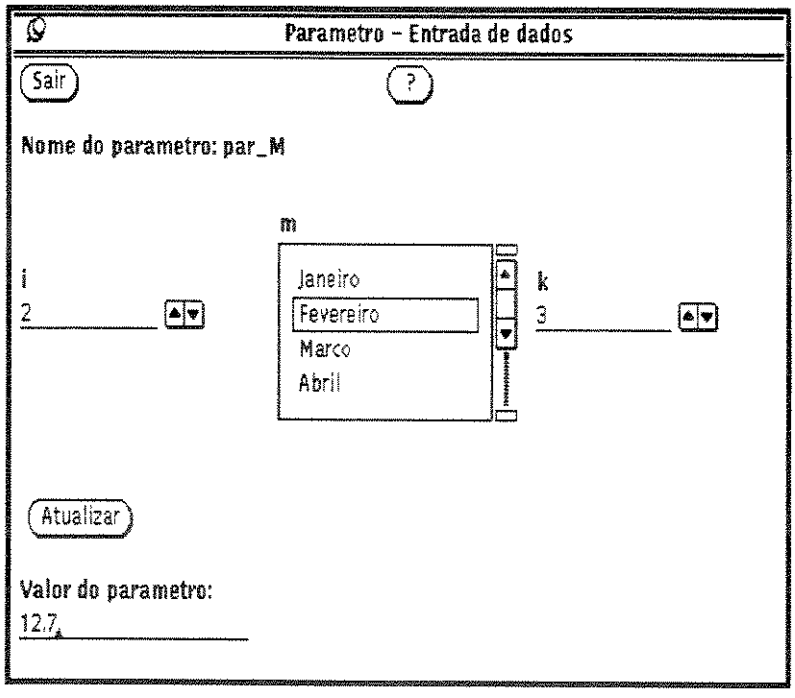


Figura 3.8 - Entrada de valores de Parâmetro.

- **Simbólica**: apresenta o modelo corrente simbolicamente ou seja, como foi definido pelo usuário. A apresentação simbólica é composta por símbolos de somatória, variáveis, parâmetros, índices, etc. Esta visão está mais próxima do entendimento humano pois é formado por expressões matemáticas (Figura 3.9).
- **Matricial**: apresenta o modelo corrente na forma matricial, isto é, através de vetores e matrizes. Esta visão é útil no sentido de verificar se o modelo foi definido corretamente, pois representa a maneira como o problema é tratado pelo computador (Figura 3.10).

$$\text{Max} \sum_{k \in K} c1_k x_k \quad (\text{ID } 1)$$

$$\text{Max} \sum_{k \in K} c2_k x_k \quad (\text{ID } 3)$$

$$\text{Max} \sum_{k \in K} c3_k x_k \quad (\text{ID } 5)$$

Sujeito a:

$$720\,x_1 + 107\,x_2 + 7080\,x_3 + 134\,x_5 + 1000\,x_6 \geq 5000$$

$$0.2\,x_1 + 10.1\,x_2 + 13.2\,x_3 + 0.75\,x_4 + 0.15\,x_5 + 1.2\,x_6 \geq 12.5$$

$$344\,x_1 + 460\,x_2 + 1040\,x_3 + 75\,x_4 + 17\,x_5 + 240\,x_6 \geq 2500$$

$$18\,x_1 + 151\,x_2 + 78\,x_3 + 2.5\,x_4 + 0.2\,x_5 + 4\,x_6 \geq 63$$

Canalizacoes:

$$0.00 \leq x_1 \leq 6.00$$
$$0.00 \leq x_2 \leq 1.00$$
$$0.00 \leq x_3 \leq 0.25$$
$$0.00 \leq x_4 \leq 10.00$$
$$0.00 \leq x_5 \leq 10.00$$
$$0.00 \leq x_6 \leq 4.00$$
$$x \geq 0.0$$

Figura 3.9 - Visão Simbólica.

3.3.6 - Menu Modelo

O menu Modelo apresenta um menu *pulldown* contendo as seguintes opções: **Objetivos, Restrições** e **Agregar**. A função do menu Modelo é definir (inserir), modificar e deletar objetivos e restrições além de agregar modelos. A seguir será descrita a função de cada opção do menu **Modelo**.

- **Objetivos**: dependendo do botão do mouse clicado, é apresentado um submenu (tipo *pullright*) com as opções **Inserir, Modificar** e **Deletar** ou uma tela de diálogo para o tratamento de objetivos. O botão esquerdo do mouse mostra a tela de diálogo para inserção de objetivos enquanto que o botão direito apresenta o submenu. A opção **Inserir** abre a tela de diálogo para inserção de objetivos; a opção **Modificar** abre a tela de diálogo para modificação de objetivos; a opção **Deletar** abre a tela de diálogo para remoção de objetivos (Figura 3.11).
- **Restrições**: dependendo do botão do mouse clicado, é apresentado um submenu (tipo *pullright*) com as opções **Funcional** e **Canalizada** ou uma tela de diálogo para o tratamento

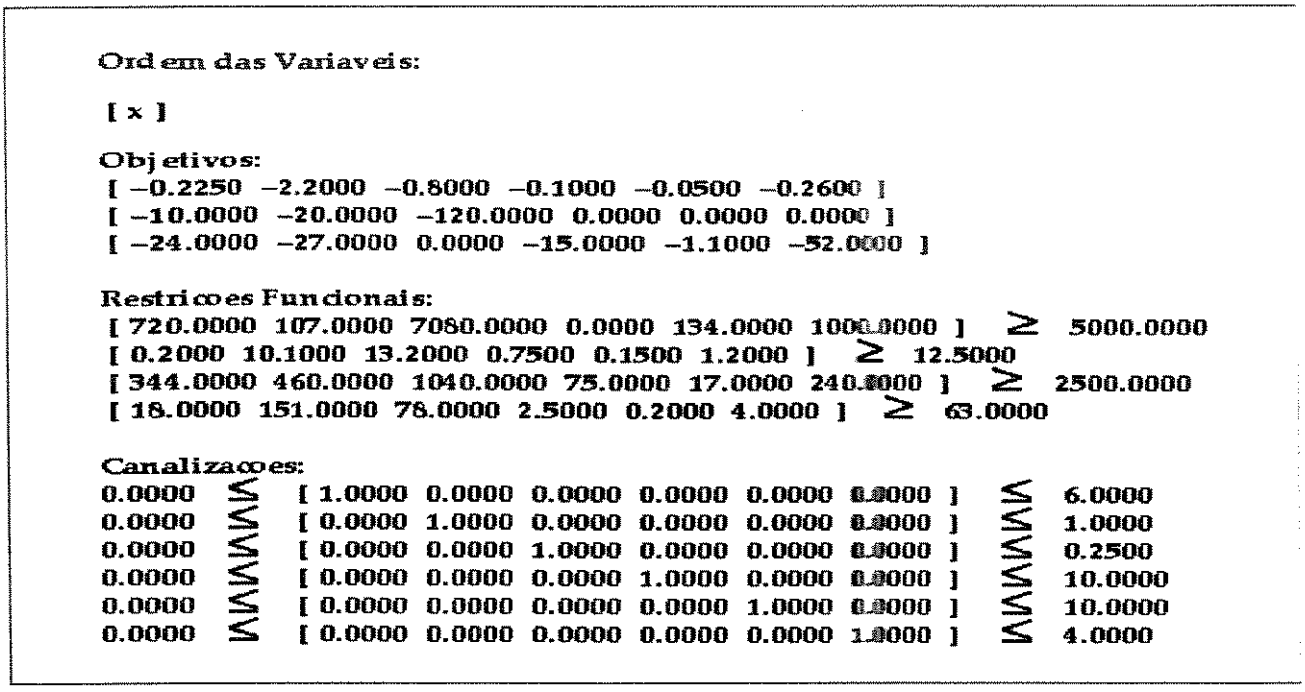


Figura 3.10 - Visão Matricial.

de restrições. O botão esquerdo do mouse mostra a tela de diálogo para inserção de restrições funcionais enquanto que o botão direito apresenta o submenu. Tanto a opção **Funcional** como a **Canalizada** apresentam um novo submenu com as opções **Inserir**. **Modificar** e **Deletar**. A opção **Inserir** abre a tela de diálogo para inserção de restrições (funcional ou canalizada); a opção **Modificar** abre a tela de diálogo para modificação de restrições (funcional ou canalizada); a opção **Deletar** abre a tela de diálogo para remoção de restrições (funcional ou canalizada) (Figuras 3.12 e 3.13).

- **Agregar**: abre uma tela de diálogo que realiza a agregação (*merge*) do modelo corrente com um modelo armazenado em disco. No processo de agregação, é possível escolher os itens a serem agregados. Por exemplo: pode-se agregar ao modelo corrente apenas as variáveis e parâmetros de um outro modelo. Também podemos escolher o tipo de agregação entre as opções **Ignorar, Sobrepor** e **Renomear**. Esta opção é interessante pois podemos construir bibliotecas de primitivas e depois utilizá-las na elaboração dos modelos. Esta tela está melhor detalhada no tópico **Tela de Diálogo para Agregar** (Figura 3.14).

Tela de Diálogo para Objetivos

Na tela de diálogo para objetivos (Figura 3.11) temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de objetivos, enquanto que na barra de rodapé temos o número de objetivos existentes.

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir o objetivo. Inicialmente devemos estabelecer um identificador numérico (ID) para o objetivo. Após ser estabelecido o ID devemos indicar se o objetivo deve ser maximizado ou minimizado e editar o objetivo propriamente dito. A edição do objetivo obedece a uma sintaxe explicada no item 3.3.13.

O botão **Inserir** define um novo objetivo, o botão **Modificar** altera um objetivo existente, o botão **Carregar** carrega os dados de um objetivo a partir do seu ID, o botão **Deletar** remove um objetivo existente. Observe que antes de modificar ou deletar um objetivo, deve-se carregá-lo ou entrar com o ID do mesmo.

Nesta tela de diálogo temos ainda o botão **Objetivo** que desenha o objetivo definido na área de apresentação (nesta área, o objetivo é apresentado com símbolos matemáticos) e o botão **Limpar** que remove todo o conteúdo da área de apresentação. O botão **Objetivo** é útil para verificar se a sintaxe do objetivo está correta, pois antes de se desenhar a expressão matemática esta deve ser submetida ao analisador sintático do MC++.

Na inserção de objetivos, os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados. Na modificação de objetivos, os botões desabilitados serão **Inserir** e **Deletar**. Já na deleção de objetivos, os botões **Inserir** e **Modificar** do quadro de diálogo estarão desabilitados.

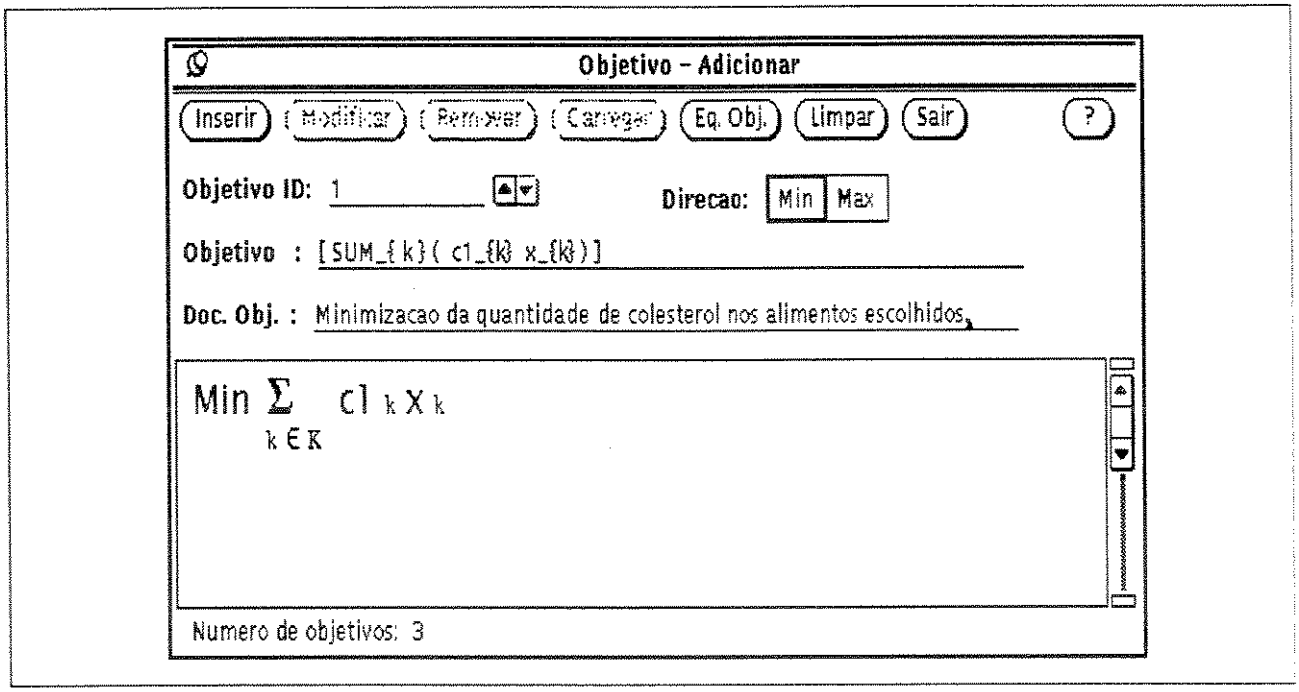


Figura 3.11 -Tela de Diálogo p/ Objetivos (Inserção)

Tela de Diálogo para Restrições Funcionais

Na tela de diálogo para restrições funcionais (Figura 3.12) temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de restrições, enquanto que na barra de rodapé temos o número de restrições existentes.

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir a restrição. Primeiramente devemos estabelecer um identificador numérico inteiro positivo qualquer (ID) para a restrição. Após ser estabelecido o ID devemos editar a restrição. A edição de cada restrição obedece a uma sintaxe explicada no item 3.3.13. Após a edição da restrição devemos estabelecer o lado direito

e o sinal de comparação. O sinal de comparação pode ser ≤ , = ou ≥ , e o lado direito pode ser um número ou um parâmetro.

Além disso, temos ainda a opção **Para todo**, que define os índices nos quais as restrições devem variar. Por exemplo: suponha que temos o índice k que varia entre 1 e 10. Ao definirmos uma restrição $C_{i,k}\ x_i \leq b_k$ para todo k, estamos na verdade definindo 10 restrições (**C** e **b** são parâmetros, **x** é uma variável e **i** é um índice).

O botão **Inserir** define uma nova restrição, o botão **Modificar** altera uma restrição existente, o botão **Carregar** carrega os dados de uma restrição a partir do seu ID, o botão **Deletar** remove uma restrição existente. Observe que antes de modificar ou deletar um restrição, deve-se carregá-la ou entrar com o ID da mesma.

Nesta tela de diálogo temos ainda o botão **Restrição** que desenha a restrição definida na área de apresentação (nesta área, a restrição é apresentada com símbolos matemáticos) e o botão **Limpar** que remove todo o conteúdo da área de apresentação. O botão **Restrição** é útil para verificar se a sintaxe da restrição está correta, pois antes de se desenhar a expressão matemática esta deve ser submetida ao analisador sintático do MC++.

Deve-se notar que os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados na inserção de restrições funcionais. Na modificação de restrições funcionais, notamos que os botões **Inserir** e **Deletar** do quadro de diálogo estão desabilitados. Já na deleção de restrições funcionais, os botões desabilitados serão **Inserir** e **Modificar** .

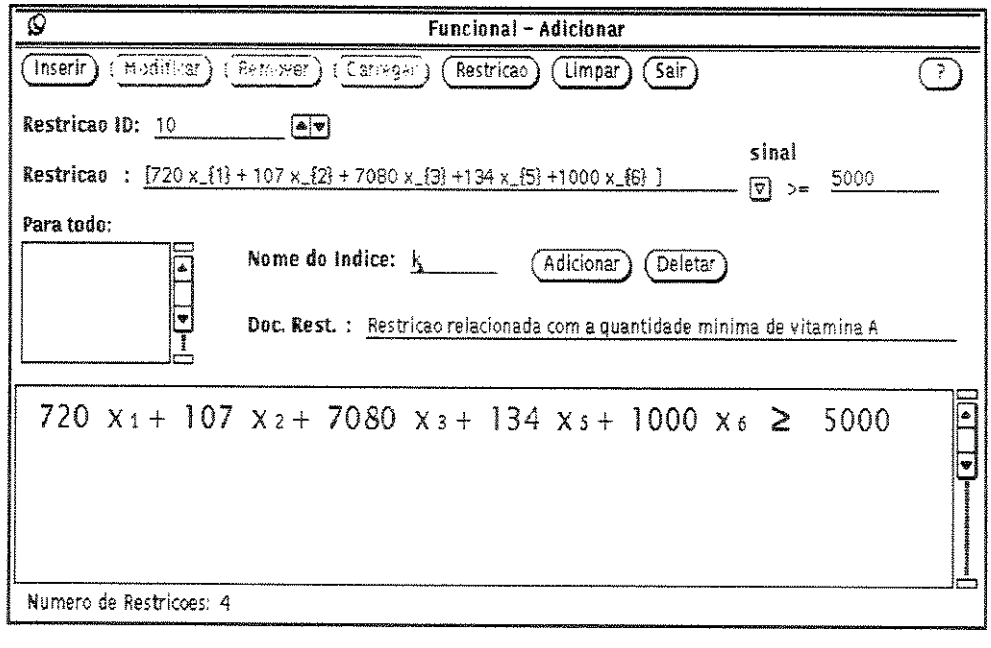


Figura 3.12 - Tela de Diálogo p/ Restrição Funcional

Tela de Diálogo para Restrições Canalizadas

Na tela de diálogo para restrições canalizadas (Figura 3.13) temos acima uma barra de cabeçalho indicando inserção, modificação ou deleção de restrições, enquanto que na barra de rodapé temos o número de restrições canalizadas existentes.

Na área principal (entre as barras de cabeçalho e rodapé) devemos definir a canalização da restrição. Inicialmente todas as variáveis estão canalizadas entre 0 e + ∞. Para canalizarmos uma variável devemos indicar os valores dos índices da mesma e depois entrar com os extremos inferior e superior da canalização.

O botão **Inserir** define uma nova restrição canalizada, o botão **Modificar** altera uma restrição existente, o botão **Carregar** carrega os dados de uma restrição a partir do seu nome, o botão **Deletar** remove uma restrição existente. Observe que antes de modificar ou deletar um restrição, deve-se carregá-la.

Na inserção de restrições canalizadas, os botões **Modificar**, **Deletar** e **Carregar** do quadro de diálogo estão desabilitados. Na modificação de restrições canalizadas, os botões **Inserir** e **Deletar** estarão desabilitados. Finalmente na deleção de restrições canalizadas, os botões desabilitados serão **Inserir** e **Modificar**.

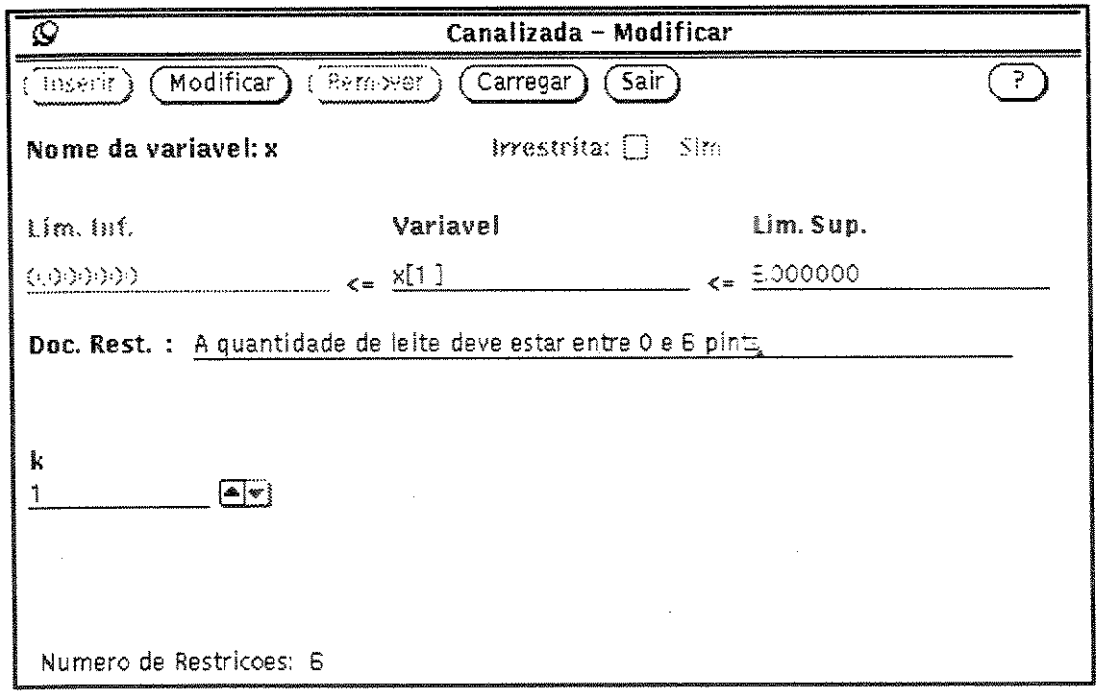


Figura 3.13 - Tela de Diálogo p/ Rest. Canalizada

Tela de Diálogo para Agregar

Esta tela de diálogo (Figura 3.14) permite a agregação (*merge*) do modelo corrente com um ou mais modelos armazenados em disco. Primeiramente deve-se escolher os itens a serem agregados (escopos, índices, variáveis, parâmetros, objetivos, restrições canalizadas e restrições funcionais) e depois deve-se definir o diretório e o nome do modelo a ser agregado. De posse dessas informações devemos escolher o tipo de agregação, ou seja, se desejamos **Ignorar**, **Sobrepor** ou **Renomear** os itens com nomes iguais (podemos ter variáveis com nomes iguais ou objetivos com mesmo ID) e finalmente devemos clicar no botão **Agregar**.

A opção **Ignorar** agrega um modelo ignorando os nomes e IDs idênticos nos dois modelos (neste caso é mantido os dados do modelo corrente). A opção **Sobrepor** agrega um modelo sobrepondo os nomes e IDs idênticos nos dois modelos (neste caso, os dados do modelo corrente são substituídos pelos do modelo agregado). A opção **Renomear** agrega um modelo renomeando os nomes e IDs idênticos nos dois modelos (neste caso, os dados do modelo agregado são alterados a partir do acréscimo de um $ no final de um nome ou da mudança de um ID).

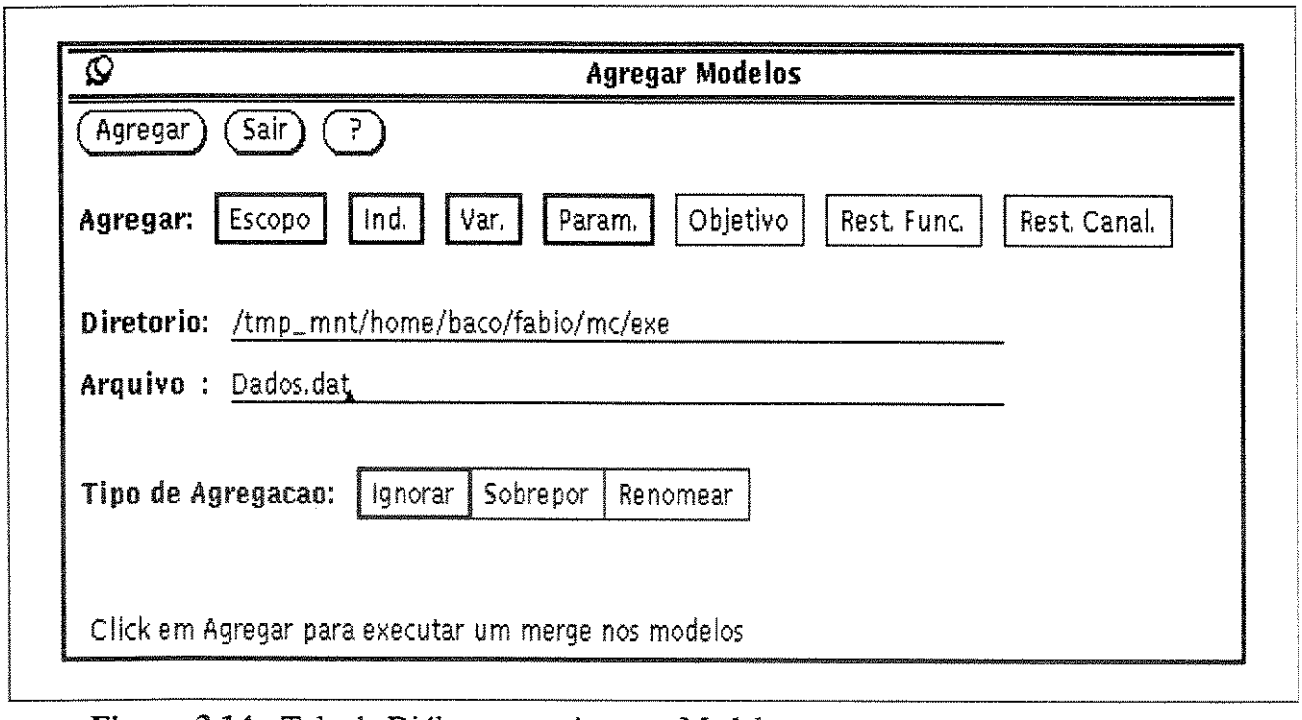


Figura 3.14 - Tela de Diálogo para Agregar Modelos

3.3.7 - Menu Estrutura

O menu Estrutura (Figura 3.15) apresenta um menu *pulldown* contendo as seguintes opções: **Sem Articulação, A Priori, Interativo, A Posteriori**. A função do menu **Estrutura** é definir um método de resolução para o modelo multiobjetivo. Este menu é composto de uma série de outros menus tipo *pullright* que implementam as informações contidas na Tabela 1.2 (Capítulo 1). A seguir será descrita a função de cada opção do menu Modelo.

- **Sem Articulação**: apresenta um menu *pullright* com uma única opção: **Critério Global**. A opção existente abre uma janela de diálogo apresentando os dados necessários para a resolução do problema multiobjetivo através deste método.
- **A Priori**: apresenta um menu *pullright* com duas opções: **Cardinal** e **Ordinal e Cardinal**. A opção **Cardinal**, por sua vez, abre um novo menu *pullright* contendo como opções os métodos **Função Utilidade** e **Objetivos Limitados**. A opção **Ordinal e Cardinal** abre um menu com os métodos **Lexicográfico**, **Programação Alvo** e **Realização de Metas**.

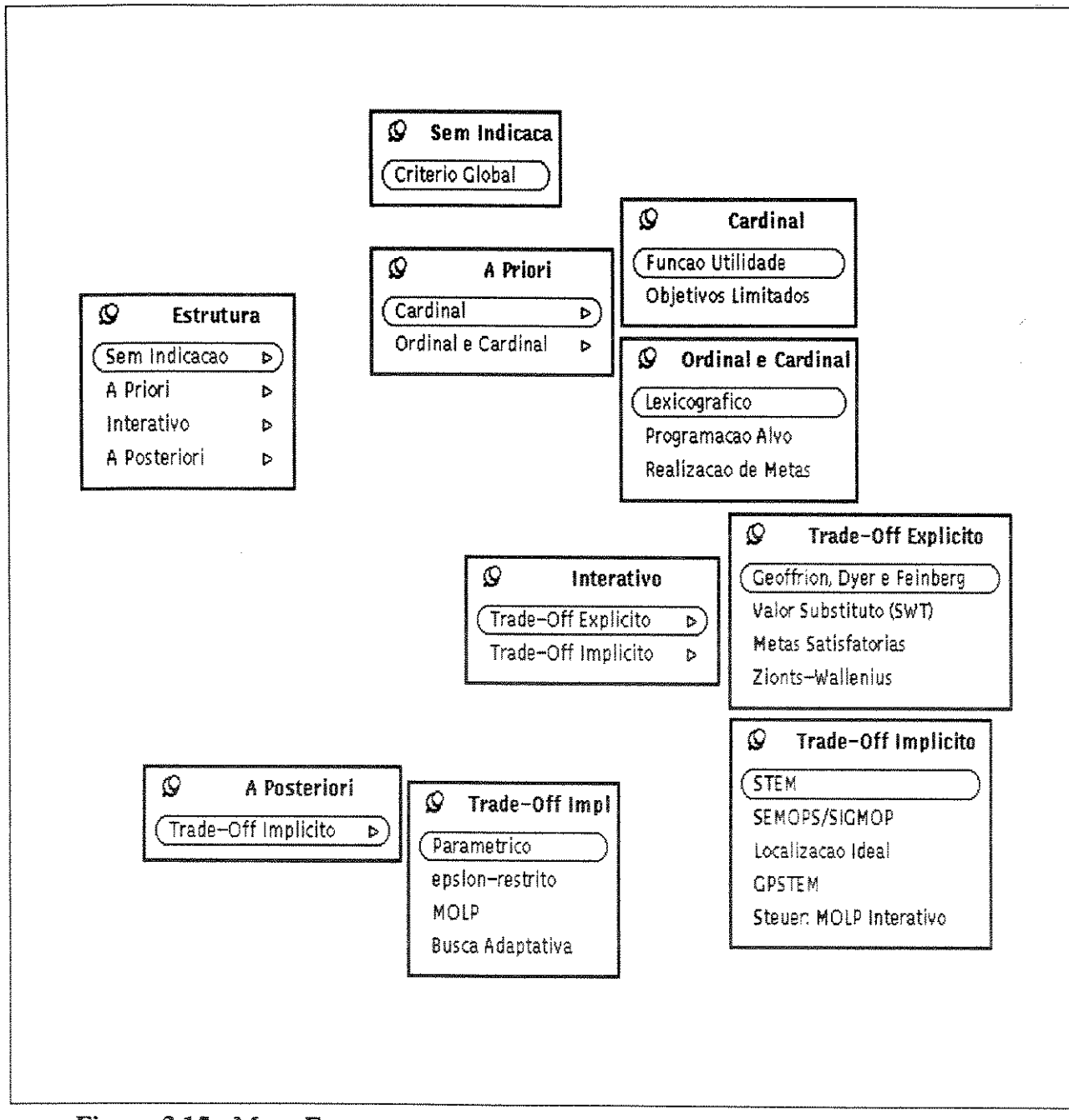


Figura 3.15 - Menu Estrutura.

- **Interativo**: apresenta um menu *pullright* com as opções: **Trade-off Explícito** e **Trade-off Implícito**. A opção **Trade-off Explícito**, por sua vez, abre um novo menu *pullright* contendo como opções os métodos **Geoffrion, Dyer e Feinberg** , **Valor Substituto (SWT)**, **Metas Satisfatórias** e **Zionts-Wallentus**. A opção **Trade-off Implícito** apresenta um menu contendo os métodos **STEM**, **SEMOPS/SIGMOP**, **Localização Ideal**, **GPSTEM** e **Steuer (MOLP Interativo)**.
- **A Posteriori**: apresenta um menu *pullright* com uma única opção: **Trade-off Implícito**. Este por sua vez, apresenta um outro menu *pullright* contendo os métodos **Paramétrico**, **ε-Restrito**, **MOLP** e **Busca Adaptativa** como opções.

Devido ao enorme esforço envolvido no desenvolvimento do MC++, apenas um método representativo das opções do menu **Estrutura** foi implementado. As telas de diálogo relativas a estes métodos são descritas a seguir.

Para os métodos não interativos, o processo para a solução de um modelo é idêntico e consta dos seguintes passos: após preencher os dados referentes a cada método, devemos clicar em **Solucionar**. Esta opção resolve o modelo corrente e apresenta uma tela com o resultado obtido. Se desejarmos que esta solução faça parte do relatório devemos clicar em **Atualizar Relatório**.

Tela de Diálogo para o método Critério Global

Neste método, devemos escolher o valor de **p** que pode ser 1, 2 ou ∞ e depois podemos setar os valores de β cujo *default* é 1.0. O próximo passo é solucionar o modelo (Figura 3.16).

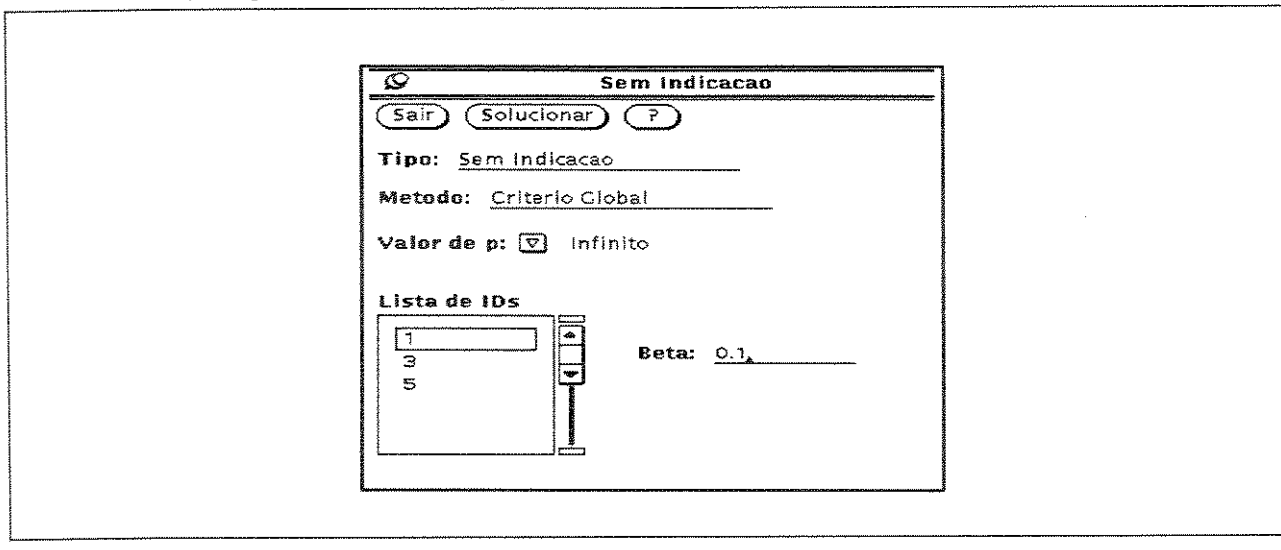


Figura 3.16 - Método Critério Global.

Tela de Diálogo para o Método Objetivos Limitados

Neste método, devemos escolher o objetivo de referência (indicando o ID do objetivo escolhido) e depois devemos estabelecer os limites inferior e superior para todos os outros objetivos (tratados como restrições). Feito isto devemos solucionar o modelo (Figura 3.17).

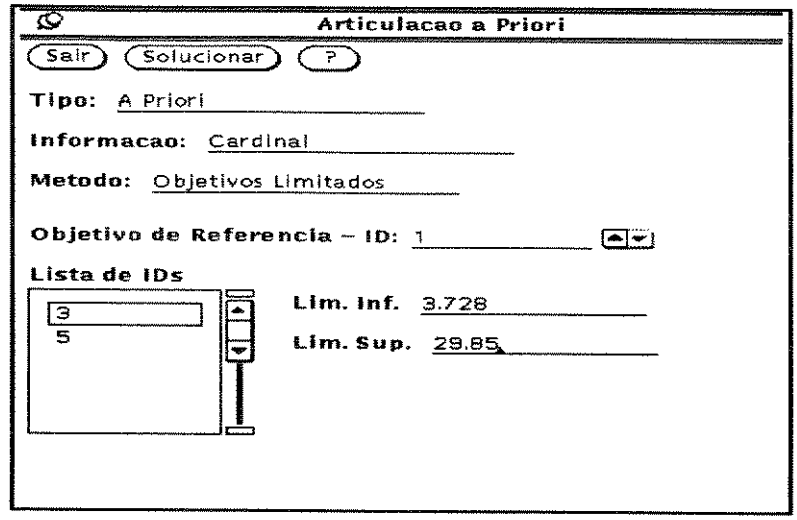


Figura 3.17 - Método Objetivos Limitados.

Tela de Diálogo para o Método Lexicográfico

O método lexicográfico (Figura 3.18) exige que o usuário defina a ordem léxica na qual os objetivos devem ser otimizados. Esta operação é realizada clicando o mouse sobre a lista de IDs apresentada. A medida que escolhemos a ordem dos objetivos, uma nova lista vai sendo construída. Depois de montada a lista com os objetivos em ordem léxica, podemos setar os valores de α que por default são 0.0. Com os dados preenchidos, devemos solucionar o modelo.

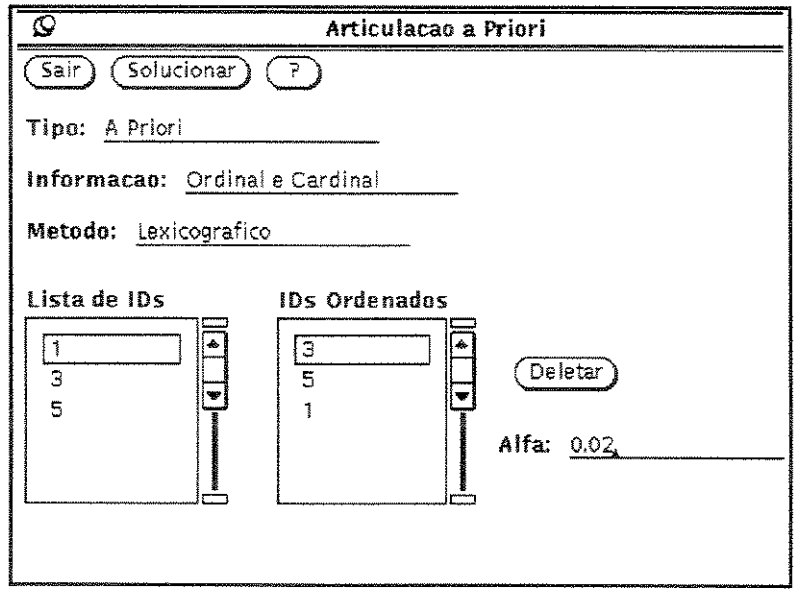


Figura 3.18 - Método Lexicográfico.

Tela de Diálogo para o Método de Geoffrion, Dyer e Feinberg

Este método (Figura 3.19) encaixa-se entre os métodos interativos, e portanto temos uma maior participação do usuário na resolução do problema. Para utilizarmos este método, devemos setar os seguintes valores:

- o número de discretizações do passo do algoritmo (t) que por default é 5 (implica numa tabela de 6 colunas - de 0.0 à 1.0 com passo de 0.2). Este valor é utilizado para montar a tabela com os valores dos objetivos para que o usuário possa definir o passo t ótimo.
- o ID do objetivo de referência
- os trade-offs Δ entre o objetivo de referência e os demais objetivos.

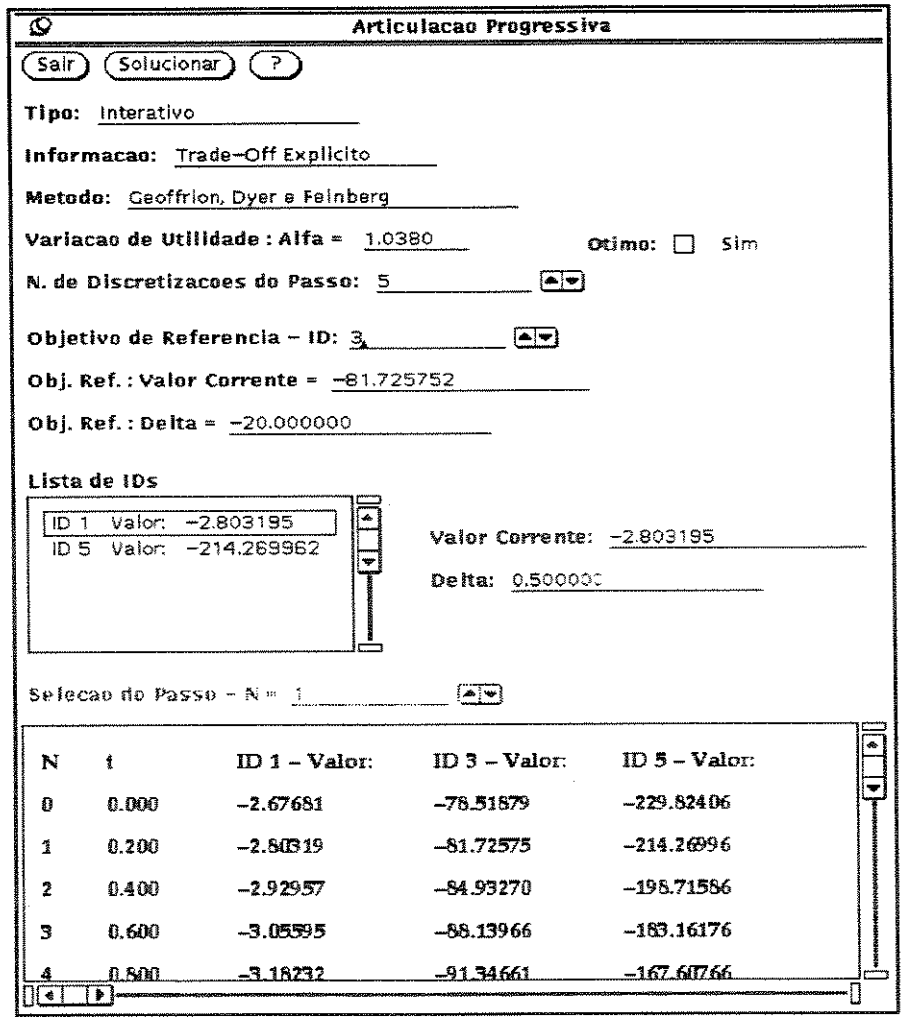


Figura 3.19 - Método Geoffrion, Dyer e Feinberg.

Feito isto, clicamos em **Solucionar** e uma tabela de valores das funções objetivos será apresentada. Escolhemos o passo ótimo **t** e clicamos em **Solucionar** novamente. Com isto, os valores dos objetivos serão atualizados e assim podemos alterar os valores do ID de referência e dos trade-offs, executando assim uma nova iteração. Para finalizar o método clicamos na opção **Ótimo** ou escolhemos um passo **t** igual a 0.0. Após a resolução do problema, uma tela com o resultado obtido será exibida. Se desejarmos que esta solução faça parte do relatório, devemos clicar em **Atualizar Relatório**.

Tela de Diálogo para o Método STEM ("Step Method")

Este método (Figura 3.20) também faz parte dos métodos interativos, e portanto exige uma maior participação do usuário na resolução do problema. Para utilizarmos este método devemos definir quais os objetivos que possuem valores satisfatórios e quais possuem valores insatisfatórios.

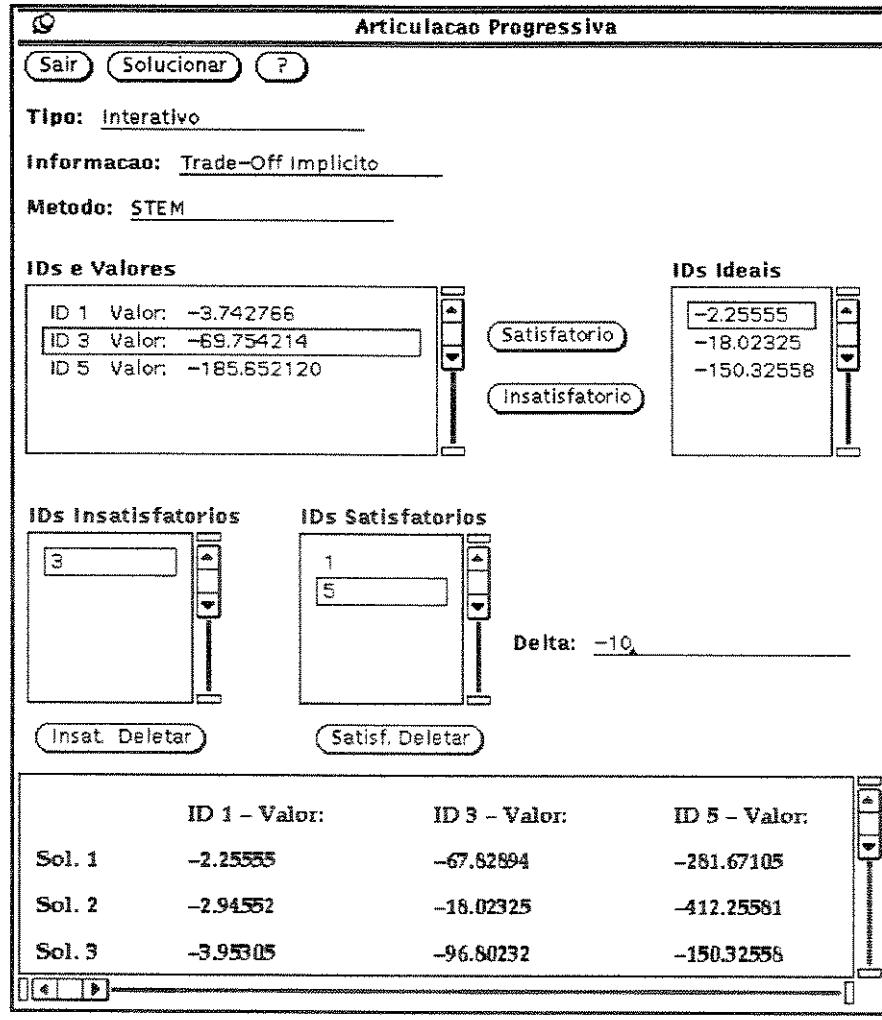


Figura 3.20 - Método STEM ("Step Method")

Este processo é realizado pressionando o botão **Satisfatório** ou **Insatisfatório** para o objetivo corrente (o objetivo corrente é o item selecionado da lista de objetivos). A medida que clicamos em **Satisfatório** ou **Insatisfatório**, duas novas listas vão sendo geradas: uma lista de objetivos satisfatórios e uma de objetivos insatisfatórios. Com estas listas montadas, podemos inicializar os valores de trade-offs Δ para os objetivos satisfatórios. Com os dados preenchidos, devemos clicar em **Solucionar**. Após a resolução desta iteração, os valores dos objetivos serão atualizados e o usuário poderá novamente montar a lista de objetivos satisfatórios e insatisfatórios e clicar em **Solucionar**, executando assim uma nova iteração. A solução ótima será obtida quando todos os objetivos forem satisfatórios, sendo que uma tela com o resultado obtido será exibida quando este fato ocorrer. Se desejarmos que esta solução faça parte do relatório, devemos clicar em **Atualizar Relatório**.

Tela de Diálogo para o Método Paramétrico

Neste método (Figura 3.21), devemos apenas incializar a ponderação de cada objetivo e então clicar em **Solucionar**. Para setarmos a ponderação, escolhemos um determinado objetivo da lista de IDs e depois entramos com o valor da ponderação para este objetivo. O próximo passo é solucionar o modelo.

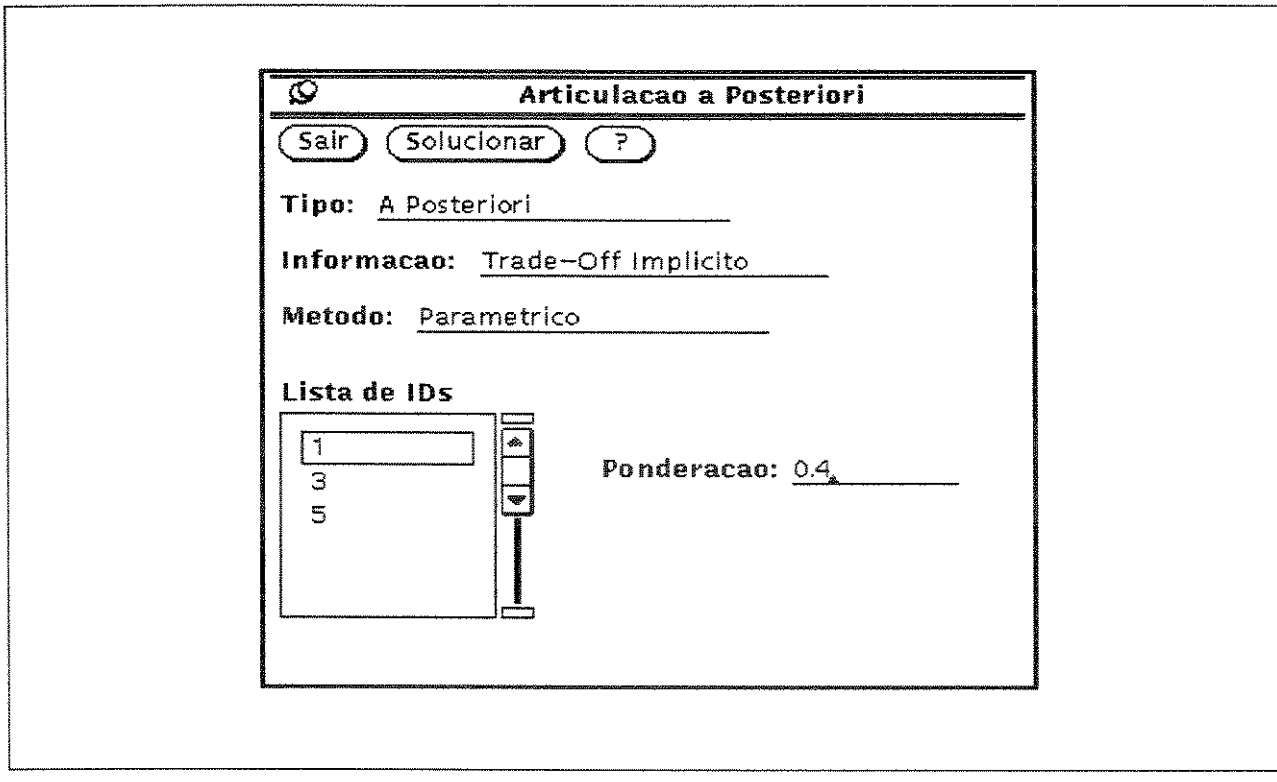


Figura 3.21 - Método Paramétrico

Tela de Diálogo para o Método ε-Restrito

Neste método (Figura 3.22), devemos escolher o objetivo de referência (indicando o ID do objetivo escolhido) e depois devemos estabelecer o lado direito para todos os outros objetivos (tratados como ε-restrições). Com os dados preenchidos, devemos solucionar o modelo.

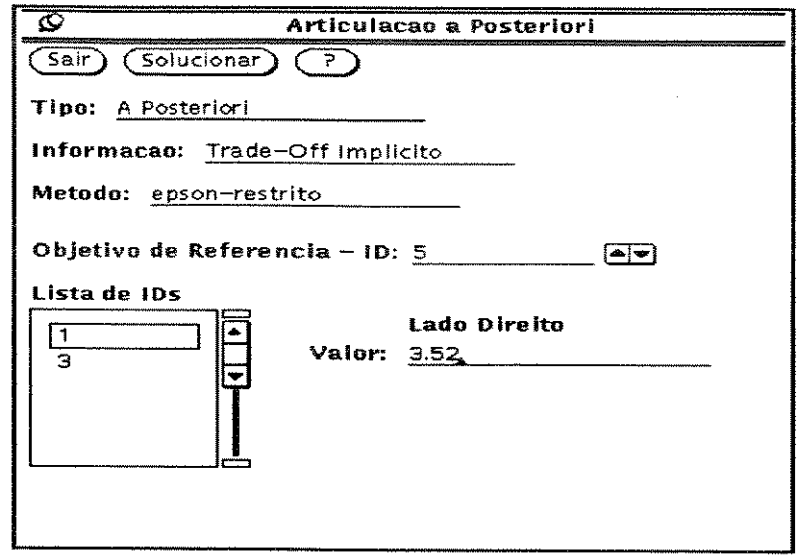


Figura 3.22 - Método ε-Restrito.

3.3.8 - Solucionar

Esta opção resolve matematicamente pelo método escolhido o modelo multiobjetivo corrente. Antes de efetuar a resolução os dados do modelo são checados para verificar possíveis inconsistências. Caso não ocorram inconsistências, o modelo é resolvido e uma tela de diálogo é exibida (Figura 3.23), indicando se houve sucesso na resolução do problema. Esta opção só funcionará se um método já tiver sido escolhido.

3.3.9 - Relatório

A opção Relatório abre uma tela onde são exibidos os dados relativos ao modelo corrente. Esta tela funciona como um manual, onde cada informação a respeito do modelo está numa determinada página. O usuário pode percorrer as páginas do relatório através de menus fornecidos para este devido fim, bem como pode imprimir estas informações.

Este relatório é dividido nas seguintes partes: *Página de Apresentação, Componentes do Modelo, Modelo Simbólico, Modelo Matricial* e *Soluções do Modelo.*

A *Página de Apresentação* contém o nome do modelo, o usuário, o arquivo onde o modelo está armazenado e uma descrição literal simplificada do mesmo. A parte relativa às *Componentes do Modelo* é formada por uma tabela contendo as primitivas que compõem o modelo. A parte relativa ao *Modelo Simbólico* mostra a representação simbólica do modelo, ou seja, a representação utilizando símbolos matemáticos (como a somatória), as variáveis e os parâmetros como estes foram definidos (esta representação é a mesma exibida na tela principal do sistema). A parte relativa ao *Modelo matricial* mostra a representação matricial do modelo, ou seja, os objetivos e restrições são exibidos como vetores e matrizes. Finalmente a parte relativa as *Soluções do Modelo* mostra as

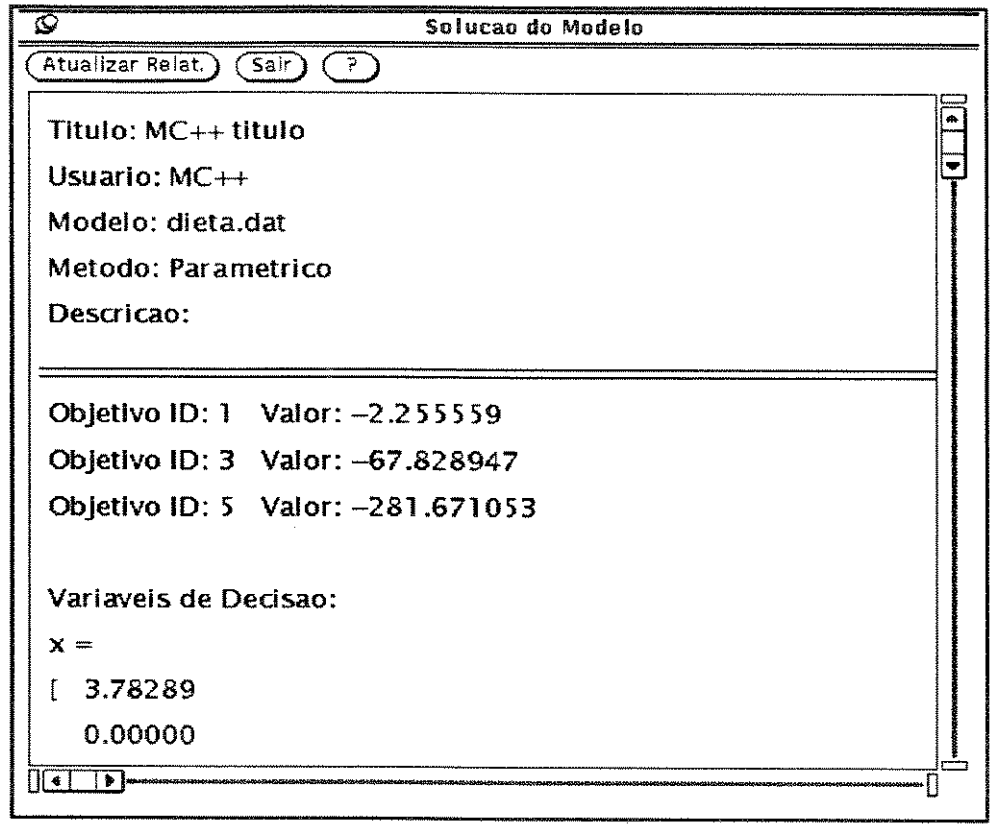


Figura 3.23 - Tela com a solução ótima.

soluções do modelo efetuadas pelos métodos escolhidos. É apresentado o valor das funções objetivos e o valor das variáveis de decisão, bem como outras informações necessárias a interpretação do resultado (Ex: ponderação do objetivos, valor de épsilon, limites, etc...) (Figuras 3.24 à 3.28)

3.3.10 - Menu Opções

O menu Opções apresenta um menu *pulldown* contendo as seguintes opções: **Documentação, Fontes, Checar, Título/Usuário** e **Atualizar Relatório**. A função do menu Opções é fornecer condições de alterar o ambiente de trabalho além de facilidades para a modelagem de um problema. A seguir será descrita a função de cada opção do menu **Opções**.

- **Documentação**: abre uma tela de diálogo (Figura 3.29) contendo um editor de textos onde são exibidas as informações do modelo corrente. Esta opção é interessante no sentido de auxiliar no registro das primitivas, objetivos e restrições já definidas. A diferença desta opção para a opção *Relatório* está na maneira mais simples dos dados serem exibidos, possuindo em contrapartida uma maior liberdade na manipulação dos mesmos;
- **Fontes**: permite a alteração das fontes utilizadas para mostrar as equações matemáticas, os relatórios e os dados (Figura 3.30);

- **Checar**: checa as primitivas, os objetivos e as restrições para verificar se não houve nenhuma inconsistência na elaboração do modelo multiobjetivo. Por exemplo: o usuário pode ter removido uma variável, porém ainda existe objetivos ou restrições que utilizam esta variável (Figura 3.31 e 3.32);
- **Título/Usuário**: tela utilizada na definição do nome do modelo, nome do usuário e descrição do modelo (Figura 3.33);
- **Atualizar Relatório**: inclui no relatório a solução obtida para o modelo pelo último método escolhido. Esta opção é a mesma encontrada nas telas de diálogo para os métodos implementados.

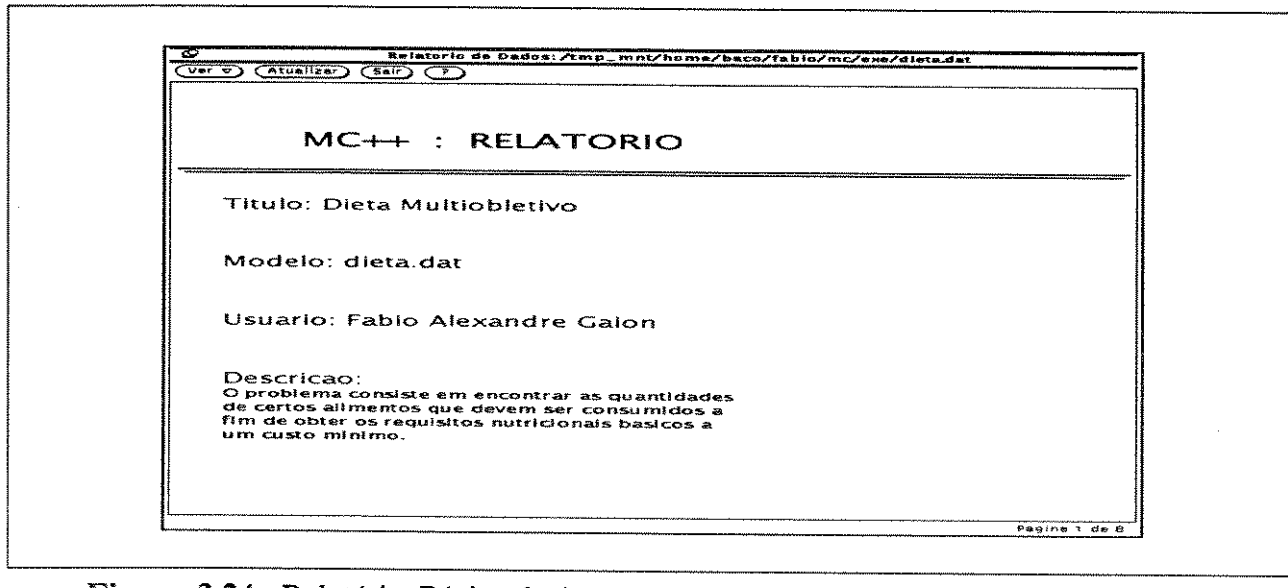


Figura 3.24 - Relatório: Página de Apresentação

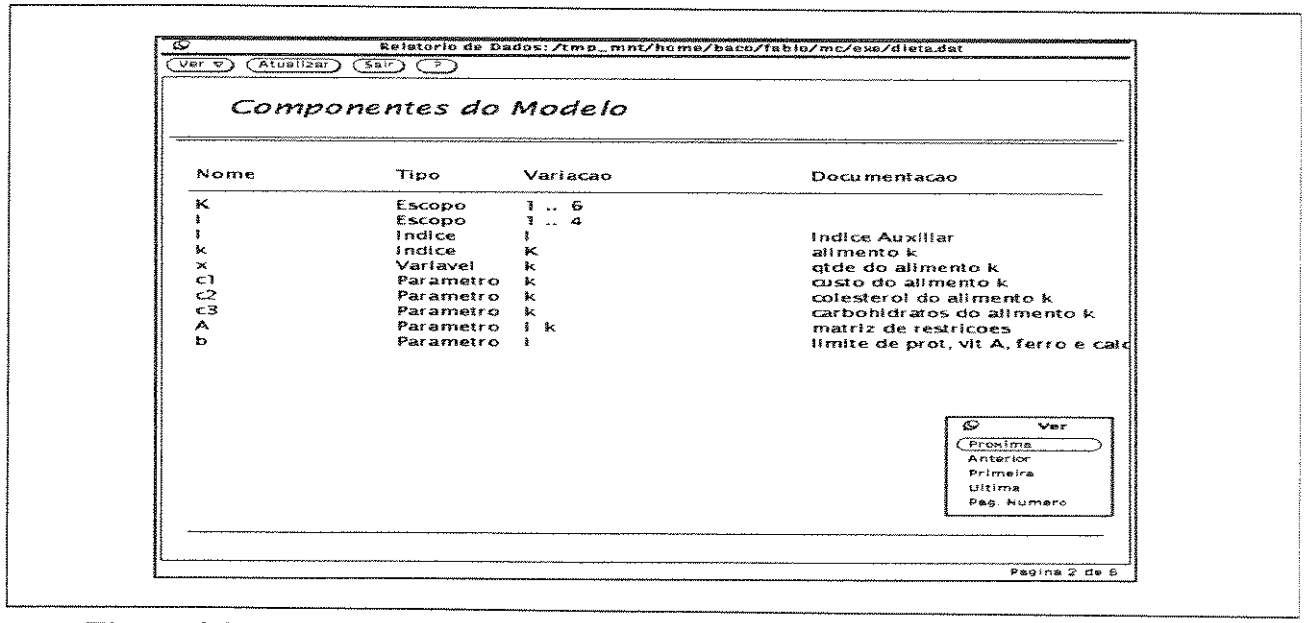


Figura 3.25 - Relatório: Componentes do Modelo.

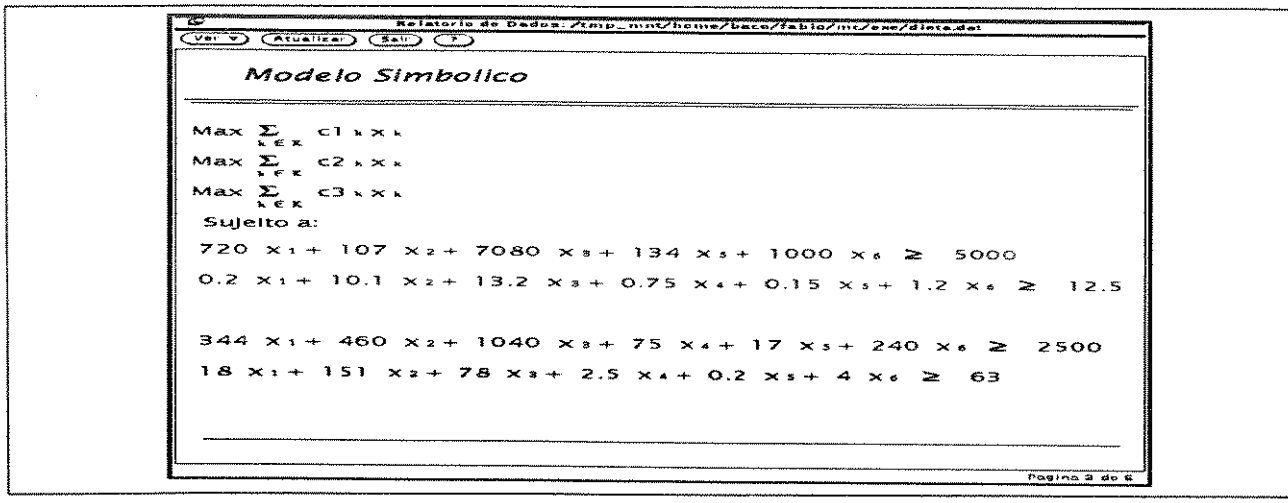


Figura 3.26 - Relatório: Modelo Simbólico

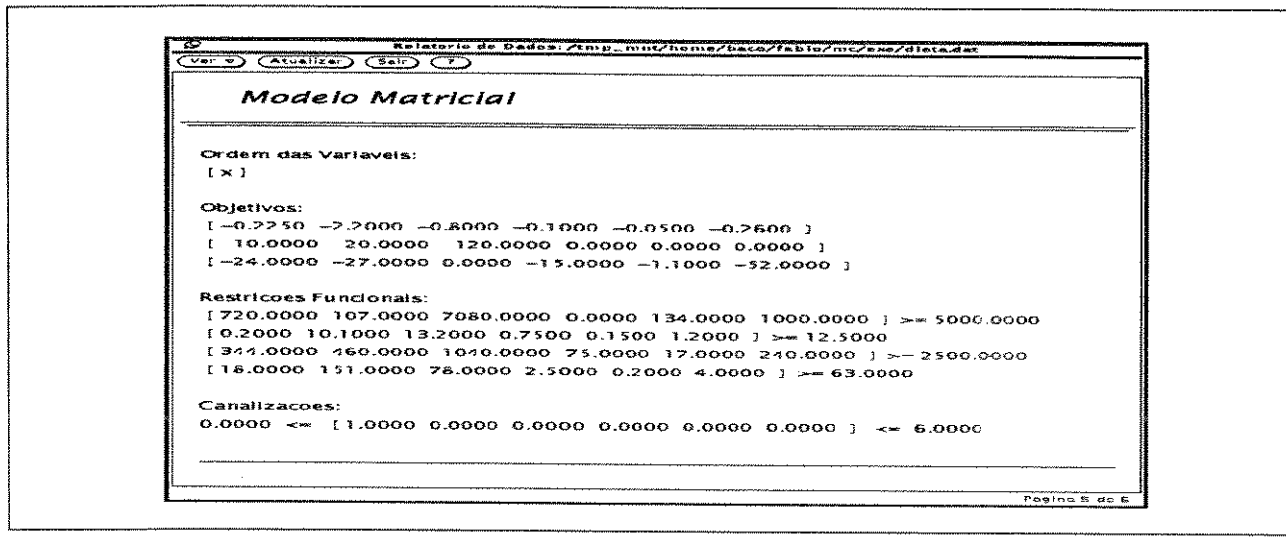


Figura 3.27 - Relatório: Modelo Matricial

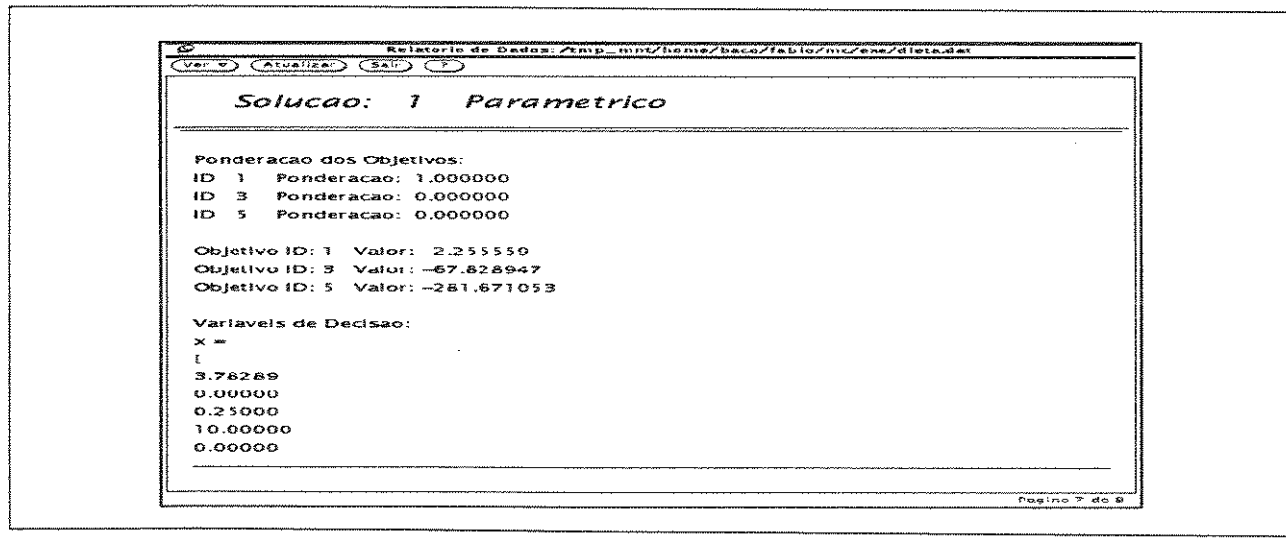


Figura 3.28 - Relatório: Solução

Tela de Diálogo para Dados

Esta janela (Figura 3.29) possui uma barra de cabeçalho contendo o nome do modelo corrente e uma série de opções para a escolha dos dados a serem exibidos. O usuário pode escolher uma combinação de itens entre escopos, índices, variáveis, parâmetros, objetivos, restrições funcionais e restrições canalizadas. Após escolher os itens a serem apresentados, deve-se clicar no botão **Atualizar**. Este botão exibe os dados escolhidos no editor de textos localizado logo abaixo das opções de escolha.

Como os dados são apresentados num editor de texto, podemos usar todos os recursos fornecidos pelo editor para trabalhar com eles. Podemos, por exemplo, usar as funções de procura, de busca e troca, de alteração maiúscula-minúscula, etc. Além disso, podemos editar os dados escrevendo, removendo e alterando trechos do documento. Podemos ainda salvar em disco estas informações para futura utilização em outros programas ou para impressão.

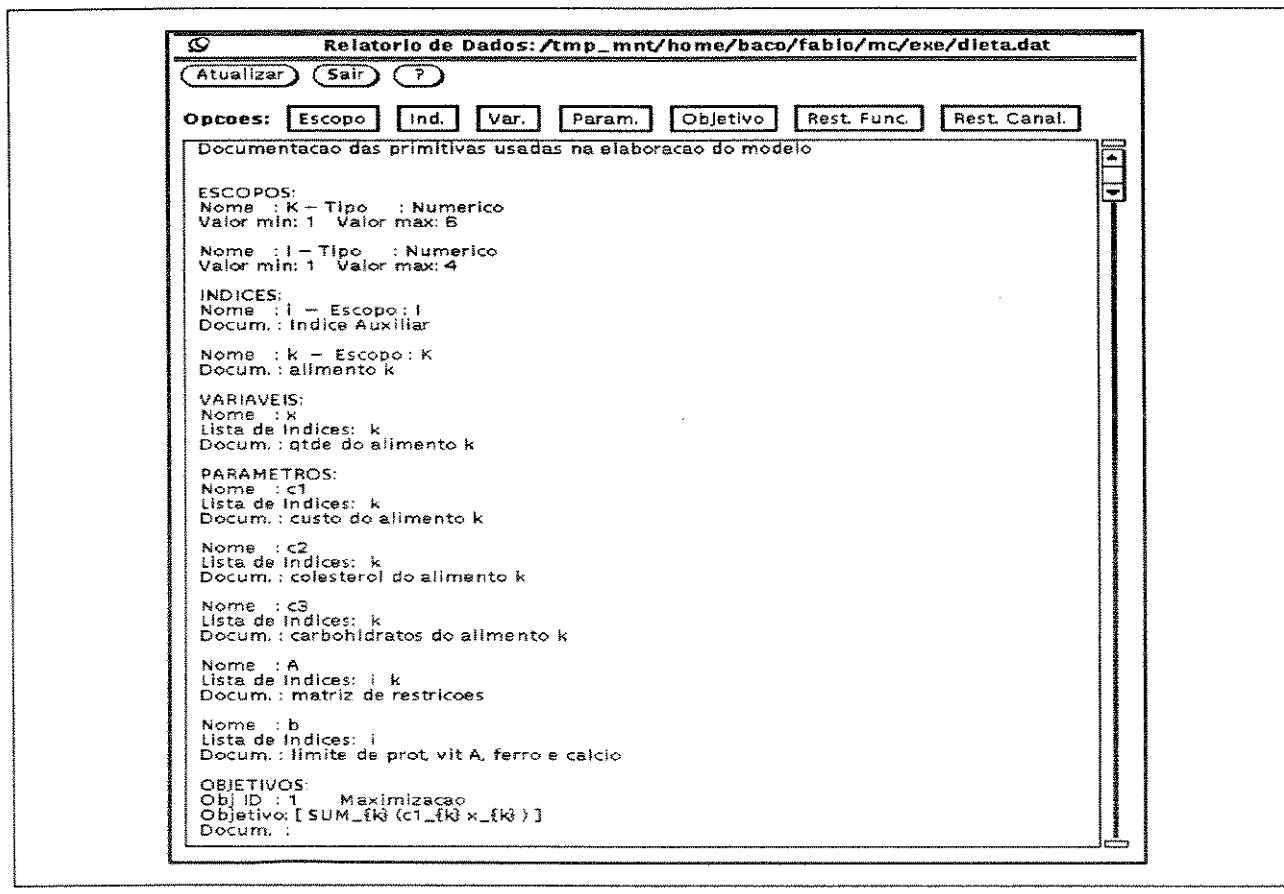


Figura 3.29 - Tela de Apresentação de Dados.

Tela de Diálogo para Fontes

A tela de fontes (Figura 3.30) permite ao usuário alterar as fontes utilizadas na exibição dos dados. Para alterar uma fonte podemos entrar diretamente com o nome da mesma ou escolher entre um conjunto de fontes pré-estabelecidas. Na escolha da nova fonte devemos estabelecer três

informações (não necessariamente as 3, mas pelo menos uma): a *família*, o *tipo* e o *tamanho*. A família é o nome da fonte ou a família a qual a fonte pertence. O tipo pode ser *normal, bold, itálico* ou *bold e itálico* sendo portanto uma informação relativa a inclinação e peso da fonte. O tamanho se relaciona com o número de pontos da fonte.

Após definir a nova fonte, podemos visualizá-la na região ao lado das opções de definição. Uma vez satisfeitos com a aparência da fonte, clicamos em *Redesenhar* para visualizarmos as alterações provocadas no MC++.

O botão *Limpar* restaura a área de apresentação de fontes se esta estiver muito carregada. O botão *Resetar* reestabelece as fontes originais utilizadas pelo MC++. A opção *Modificar* estabelece quais fontes devem ser alteradas: as expressões matemáticas, os índices das expressões ou os dados apresentados pelo opção *Dados* do menu *Opções*.

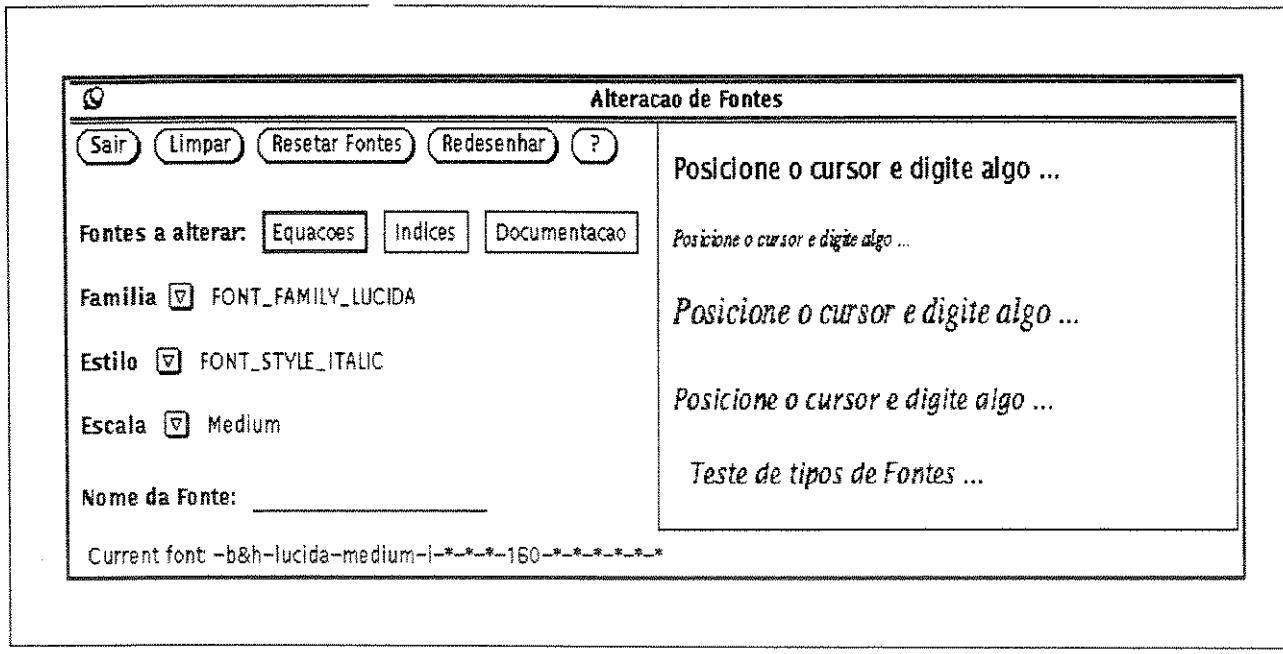


Figura 3.30 - Fontes

Tela de Diálogo para Checar

Esta tela apresenta mensagens indicando se ocorreram inconsistências na definição do modelo. Caso não haja erros, todas as informações serão seguidas por um símbolo indicando. Caso haja alguma inconsistência, esta será exibida com uma *mão* a precedendo. A existência de inconsistências não permite que a modelo seja solucionado até que esta seja corrigida (Figura 3.31 e 3.32).

Tela de Diálogo para Título/Usuário

Esta tela apresenta regiões onde podem ser definidos o nome do modelo, o usuário e a descrição do modelo em questão (FIgura 3.33).

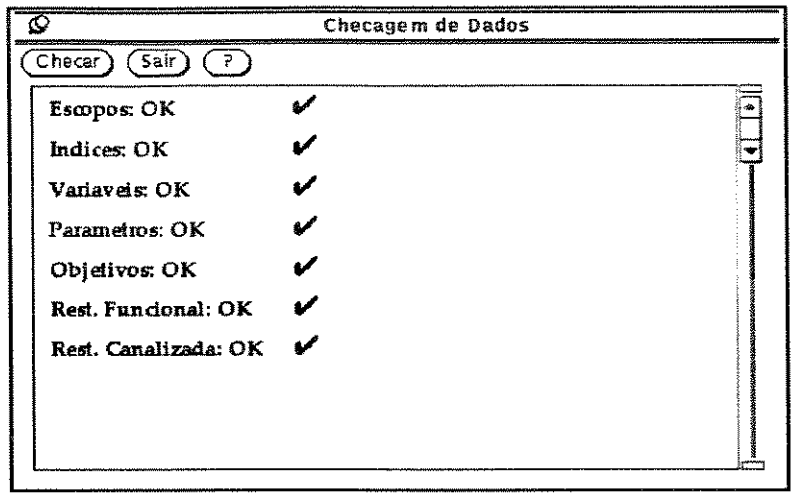


Figura 3.31 - Relatório: Checar sem Erros.

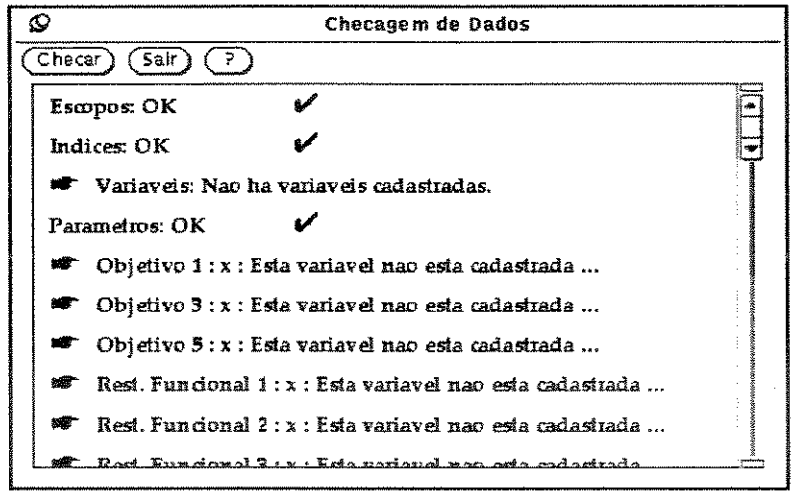


Figura 3.32 - Relatório: Checar com Erros.

3.3.11 - Tutorial

A opção Tutorial abre uma tela (Figura 3.34) onde podemos navegar através de um exemplo com explicações sobre o uso do MC++.

3.3.12 - Copyright

Apresenta informações sobre os direitos autorais (Figura 3.35).

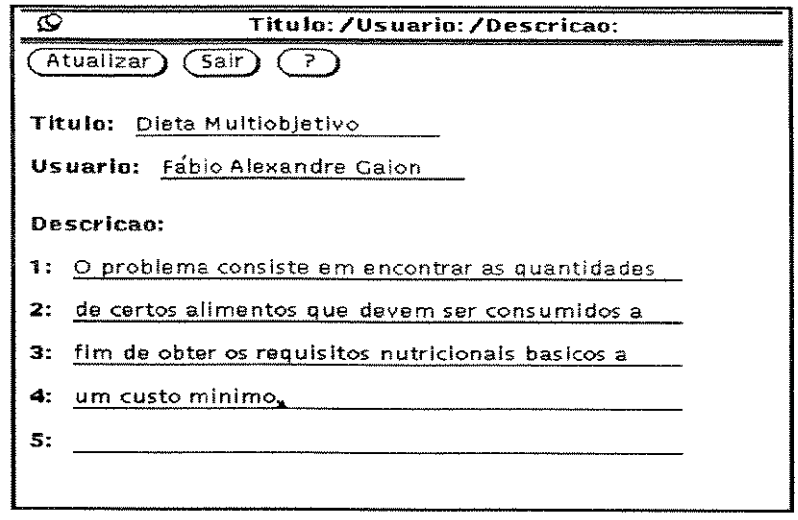


Figura 3.33 - Tela de Diálogo p/ Título/Usuário.

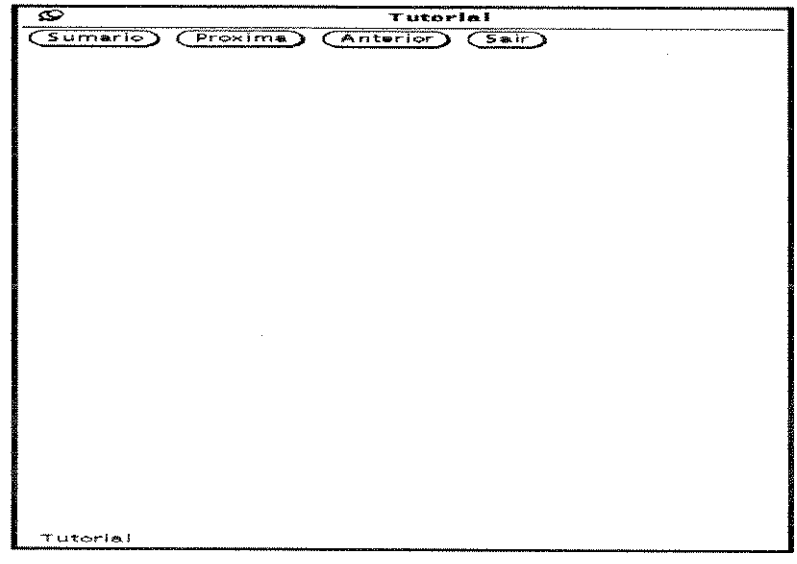


Figura 3.34 - Tutorial.

3.3.13 - Sintaxe dos Objetivos e Restrições

As rotinas que efetuam a análise semântica dos objetivos e restrições foram desenvolvidas com o auxílio dos utilitários **Lex** e **Yacc** (utilitários do sistema operacional UNIX).

O **Lex** (Lesk e Schmidt, 1974) é um programa gerador de código fonte designado para processamento léxico de expressões. Ele aceita a especificação de um problema de reconhecimento de *strings* (*matching*) através de regras de alto nível e gera o código fonte (normalmente em linguagem C) de um programa que efetua este reconhecimento (Figura 3.36).

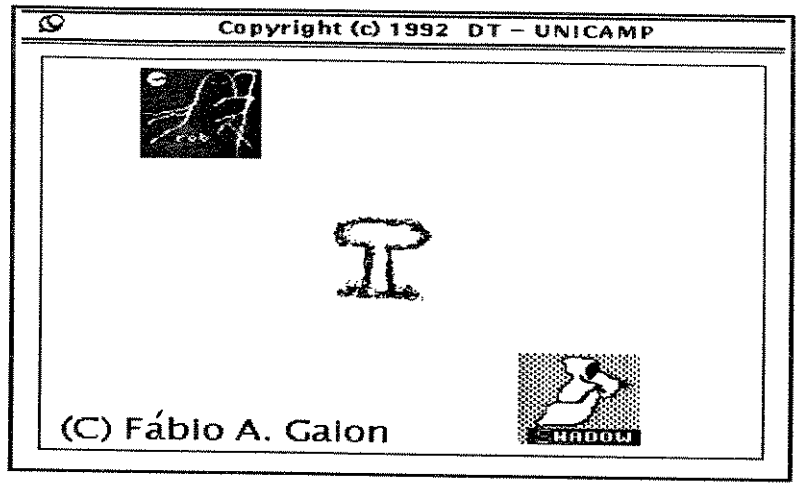


Figura 3.35 - (C) Copyright.

Este utilitário pode ser utilizado para efetuar transformações em arquivos e textos (criptografia por exemplo) ou para segmentar uma entrada de dados (expressão) em termos básicos (*tokens*) a fim de serem utilizados por uma rotina de análise semântica (no caso o **Yacc**). Como podemos notar o **Lex** pode ser utilizado na elaboração de um compilador, pelo menos na parte da compilação que efetua a análise léxica.

Os termos básicos de uma expressão são especificados num arquivo de regras. Neste arquivo, podemos incluir códigos de programação entre as regras a fim de gerar um analisador léxico mais eficiente. Baseado nesse arquivo de regras o **Lex** gera um programa que efetua a análise léxica.

O **Yacc** (Johnson, 1975) é um utilitário gerador de código fonte para efetuar a análise semântica de expressões. Essa análise é baseada num arquivo de regras que definem as expressões

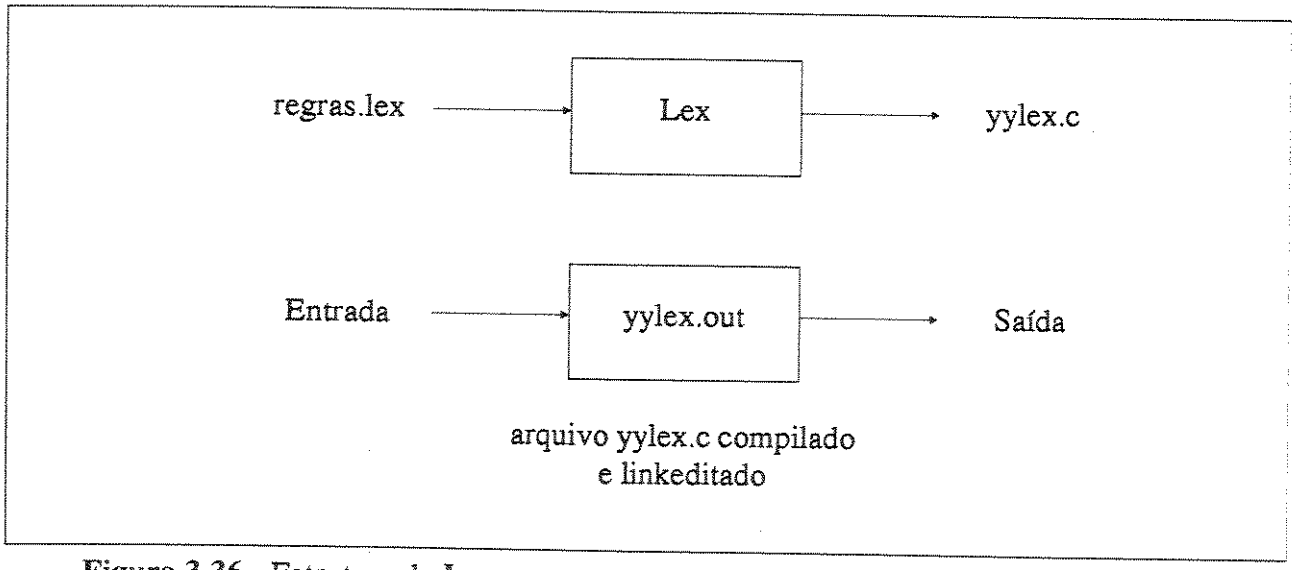


Figura 3.36 - Estrutura do **Lex**

válidas. Ao invocarmos o **Yacc** com um determinado conjunto de regras é gerado automaticamente um programa que efetua a análise semântica de uma expressão.

Este programa exige como entrada os termos básicos (*tokens*) da expressão a ser analisada. De posse de todo o conjunto de *tokens*, o programa procura encaixar os mesmos a fim gerar uma expressão válida ou uma mensagem de erro.

Como podemos notar o **Yacc** e o **Lex** completam um ao outro (Figura 3.37) , pois enquanto o **Lex** quebra uma expresssão em vários *tokens* o **Yacc** verifica se os mesmos formam uma expressão sintaticamente correta. Este fato não é simples coincidência, pois estes dois utilitários foram desenvolvidos justamente com intuito de fornecer uma ferramenta para a elaboração de analisadores sintáticos (cuja implementação não é trivial).

No Apêndice C temos uma descrição completa do analisador utilizado pelo MC++, contendo o arquivo de regras do **Lex**, o arquivo de regras do **Yacc** e a função que controla a chamada do analisador.

Antes de definirmos a *gramática* utilizada na análise dos objetivos e restrições funcionais, precisamos descrever alguns termos básicos (*tokens*) e os nomes das regras sintáticas.

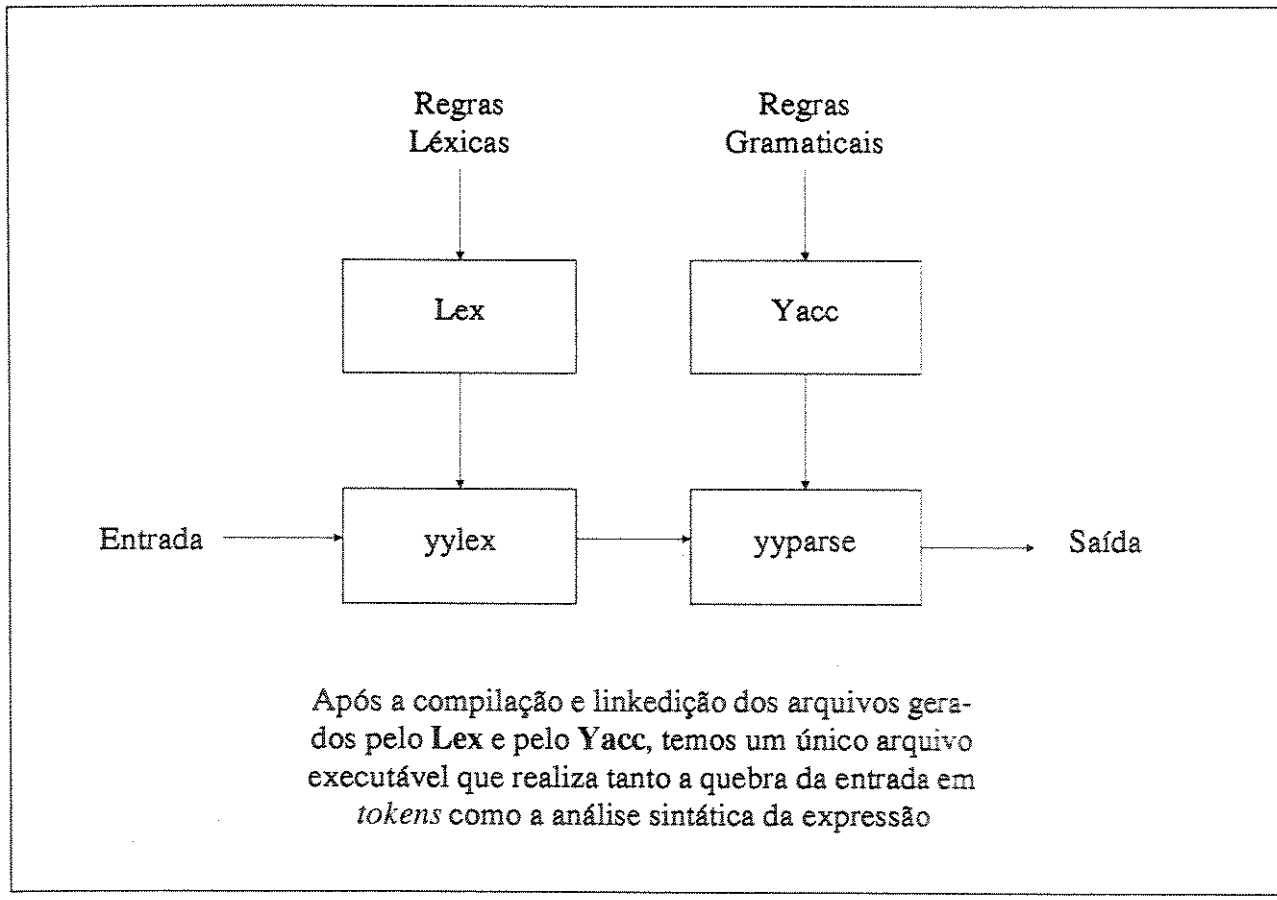


Figura 3.37 - Estrutura conjunta **Lex / Yacc**

Regras e Termos Básicos (*Tokens*)

- **PAR**: parâmetro
- **VAR**: variável;
- **SUM**: símbolo de somatória
- **IND**: string indicando o nome de um índice ou o valor string de um índice
- **IND_NUM**: valor numérico de um índice
- **NUM**: número inteiro ou real
- **LADO_DIR**: lado direito de uma restrição
- **indn**: índice do tipo numérico. Ex: a variável x_{1 , 5} possui índices 1, 5 indicando uma determinada instância da variável.
- **indl**: índice do tipo literal. Ex: a variável y_{i j} possui os índices **i , j** indicando que ela pode variar sobre todo o escopo de **i** e **j**. Também pode representar uma variável ou parâmetro do tipo c_{prod1 junho} onde **prod1** e **junho** são valores literais particulares de dois índice quaisquer.
- **indmis**: índice do tipo misto. Ex: a variável z_{janeiro 3} possui os índices **janeiro** e 3 indicando uma instância particular da variável x. A definição de **indmis** vem da possibilidade dos índices possuírem tipos diferentes (string e numérico). Podemos observar que um indmis engloba tanto um **indn** como um **indl**.
- **ladodir**: corresponde ao lado direito de uma restrição funcional. Pode ser um número ou um valor particular de um parâmetro seguido ou não dos caractere + ou -.
- **termon**: termo numérico contendo números, variáveis, parâmetros e índices mistos. Ex: x_{1 , 3} , C_{ 1 , 1 } y_{ jan , Fase2 } , 12.5 z_{2 , Prod1 , 3} representam uma variável, um parâmetro multiplicando uma variável e um número multiplicando uma variável, respectivamente.
- **termol**: termo literal contendo números, variáveis, parâmetros e índices do tipos literal e misto. O **termol** é utilizado sobretudo em somatórias onde os índices das variáveis ou parâmetros estão presos a variação da somatória. Ex: x_{ i }, C_{ i , j } y_{ i , j } , 3.7 z_{ k } onde **x , y** e **z** são variáveis, **i , j , k** são índices (nome de índice e não valores particulares) e **C** é um parâmetro.
- **exprn**: expressão numérica formada por um termo numérico ou um conjunto de termos numéricos separados pelos sinais de adição e subtração. Ex: x_{ 1 , 2 } + x{ 1 , 3 } - z_{ jan }, onde **x** e **z** são variáveis.

- **exprl**: expressão literal formada por um termo literal ou um conjunto de termos literais separados pelos sinais de adição e subtração. Ex: C_{ i , j } x_{ i , j } - z_{ i , j }, onde **x** e **z** são variáveis, **i** e **j** são índices e **C** é um parâmetro.

- **somat**: somatória (composta pelo símbolo de somatória SUM e pelo símbolo de nome de índice IND) de uma expressão literal, ou somatória de somatórias, ou uma adição/subtração entre somatórias e expressões literais. Ex: SUM_{i} (C_{i} x_{i}) + SUM_{k} (SUM_{j} (A_{ k ,j } y_{ k , j })) onde **C** e **A** são parâmetros, **x** e **y** são variáveis e **i , k e j** são índices.

- **funcpar**: função com parâmetros composta por uma expressão numérica, ou uma expressão literal, ou uma somatória. A função com parâmetros deve estar envolvida entre os caracteres [e]. Ex: [-10.25 SUM_{i} (C_{i} x_{i})] onde **C** é um parâmetro, **x** é uma variável e **i** é um índice.

- **funcao**: função que pode ser composta por uma única **funcpar** ou um conjunto de **funcpar** separadas pelos operadores de adição e subtração. Ex: [-10.25 SUM_{i} (C_{i} x_{i})] + [3 y_{ 1 , jan } - 5.7 z_{ 3 , Prod1 }] onde **C** é um parâmetro, **x , y** e **z** são variáveis , **i** é um índice e **1 , jan , 3** e **Prod1** são valores particulares de algum índice.

- **bala**: pode ser uma **funcao** ou um **ladodir**. Quando chegamos neste ponto, temos um objetivo ou restrição funcional sintaticamente corretos segundo a gramática definida para o MC++.

A sintaxe utilizada na definição dos objetivos e restrições pode ser descrita pela *carta sintática* abaixo.

Carta Sintática

//os carcteres que aparecem entre aspas simples '' são obrigatórios numa expressão

tokens: PAR VAR SUM IND IND_NUM NUM LADO_DIR

bala	: funcao	//função objetivo ou restrição funcional
	| ladodir	//lado direito de uma restrição funcional
	;	
funcao	: funcpar	//função com parâmetros
	| funcao '+' funcpar	//função + função com parâmetros
	| funcao '-' funcpar	//função - função com parâmetros
	;	

```
funcpar     : '[' exprn ']'          //[ expressão numérica ]
            | '[' exprl ']'          //[ uma expressão literal ]
            | '[' somat ']'          //[  somatória ]
            | '[' '+' somat ']'      //[ + somatória ]
            | '[' '-' somat ']'      //[ - somatória ]
            | '[' NUM somat ']'      //[ número vezes somatória ]
            | '[' '+' NUM somat ']'  //[ + número vezes somatória ]
            | '[' '-' NUM somat ']'  //[ - número vezes somatória ]
            ;

somat       : SUM IND '(' exprl ')'  //somatória sobre um índice de uma exprl
            | SUM IND '(' somat ')'  //somatória sobre um índice de uma somat
            | somat '+' exprl        //somatória + expressão literal
            | somat '-' exprl        //somatória  - expressão literal
            | somat '+' somat        //somatória + somatória
            | somat '-' somat        //somatória - somatória
            ;

exprl       : termol                 //termo literal
            | '+' termol             //+ termo literal
            | '-' termol             //- termo literal
            | exprl '+' termol       //expressão literal + termo literal
            | exprl '-' termol       //expressão literal  - termo literal
            ;

exprn       : termon                 //termo numérico
            | '+' termon             //+ termo numérico
            | '-' termon             //- termo numérico
            | exprn '+' termon       //expressão numérica + termo numérico
            | exprn '-' termon       //expressão numérica - termo numérico
            ;

termol      : VAR indl               //variável com índice literal
            | PAR indmis VAR indl    //parâmetro vezes uma variável
            | NUM termol             //número vezes um termo literal
            ;

termon      : VAR indmis             //variável com índice misto
            | PAR indmis VAR indmis  //parâmetro vezes uma variável
            | NUM termon             //número vezes um termo numérico
            ;
```

```
ladodir     : LADO_DIR NUM                  //lado direito é número
            | LADO_DIR '+' NUM              //lado direito é + número (NUM)
            | LADO_DIR '-' NUM              //lado direito é - número (NUM)
            | LADO_DIR PAR indmis           //lado direito é um parâmetro
            | LADO_DIR '+' PAR indmis       //lado direito é + parâmetro
            | LADO_DIR '-' PAR indmis       //lado direito é  - parâmetro
            | LADO_DIR PAR indl             //lado direito é um parâmetro
            | LADO_DIR '+' PAR indl   /     /lado direito é + parâmetro
            | LADO_DIR '-' PAR indl         //lado direito é - parâmetro
            ;

indmis      :                               //índice misto pode ser vazio
            | indmis indl                   //índice misto e um índice literal
            | indmis indn                   //índice misto e um índice numérico
            ;

indl        : IND                           //nome do índice ou valor literal do índice
            | indl IND                      //conjunto de índices (def. recursiva)
            ;

indn        : IND_NUM                       //valor numérico de um índice
            | indn IND_NUM                  //conjunto de valores de índices (recursiva)
            ;
```

Exemplos de Objetivos:

[SUM_{i} (SUM{k} ((A_{i k} z_{i k}))]

[10 x_{1 jan} + 12 x_{2 fev} - 5 x_{3 jan}]

[SUM_{i} (C_{i} x_{i})] - [10 z_{1 jan}]

Exemplos de Restrições Funcionais

[SUM_{i} SUM_{k} ((A_{i k} z_{i k}))] = 100

[10 x_{1 jan} + 12 x_{2 fev} - 5 x_{3 jan}] $\leq$ b_{1}

[SUM_{i} (C_{i} x_{i})] $\geq$ 20

A sintaxe definida foi baseada no modo de edição de fórmulas matemáticas existente no software Tex (Knuth, 1986)

3.4 - Estruturas e Base de Dados

3.4.1 - Estruturas para Manipulação de Dados

Toda a estrutura de dados foi elaborada utilizando-se listas e visando possíveis ampliações e aperfeiçoamentos futuros. A utilização da linguagem C++ no desenvolvimento da estrutura de dados foi responsável por grande parte da generalidade com que foi implementado o MC++.

Como sabemos, C++ é uma linguagem orientada a objetos e baseado nesta característica foi criado um objeto (classe) para cada tipo de dado utilizado no MC++.

Primeiramente foi criada uma classe que implementava uma fila de apontadores para qualquer tipo de dado. Esta classe possui um conjunto de operações gerais pertinentes a qualquer fila. Por exemplo: inserir no começo (*insert*), inserir no fim (*append*), limpar (*clear*), procurar (*find*), etc.

Baseado nesta classe básica e utilizando a hereditariedade permitida pela linguagem C++, foi desenvolvida uma classe para lista de escopos, lista de índices, lista de variáveis, lista de parâmetros, lista de objetivos, lista de restrições funcionais e lista de restrições canalizadas. Cada uma destas listas possuí características próprias além das características básicas dada pela fila geral.

É importante observar que antes de criar as listas de dados, foi desenvolvida uma classe para o dado propriamente dito. Assim sendo, foi criado uma classe para escopo, índice, variável, parâmetro, objetivo, restrição funcional e restrição canalizada.

No Apêndice A são apresentadas as diversas classes ou objetos que compõem o MC++.

3.4.2 - Estruturas para Tratamento de Janelas

A interface com o usuário do MC++ foi implementada utilizando-se as bibliotecas Xwindows. As bibliotecas mais utilizadas foram o Xview, o Xlib e o Xintrinsics. Estas bibliotecas fornecem funções que possibilitam a construção de programas com a interface do ambiente OPENWINDOWS.

Antes de descrevermos as estruturas utilizadas, devemos nos familiarizar com alguns termos utilizados em progamação Xwindow.

- *Frame*: objeto principal onde são associados todos os objetos Xwindows criados.
- *Panel*: área onde são criados os objetos das telas de diálogo, isto é, os objetos para entrada de dados e apresentação de mensagens de aviso ao usuário.
- *Canvas*: área de desenho onde são apresentados elementos gráficos.
- *Menu*: objeto relacionado com a criação de menus *pulldown* e *pullright*.

Na implementação do MC++ foi criado um frame básico principal. Sobre este frame foram criados todos os outros objetos que formam o software.

Cada tela de diálogo é associada a um subframe (frame de comando) que por sua vez pode conter vários objetos do tipo panel ou canvas. Na realidade, a maioria das telas de diálogos são um frame de comando contendo um panel

Existe também um panel principal associado ao frame básico que contém os botões **Arquivos, Primitivas, Visão, Modelo, Estrutura, Relatório, Opções, Tutorial** e **Copyright** já descritos na seção 3.3.

Um menu principal foi igualmente associado ao frame básico contendo as opções **Arquivos, Primitivas, Visão, Modelo, Estrutura, Relatório** e **Opções**. Este menu proporciona ao usuário uma outra via de acesso às telas de diálogo.

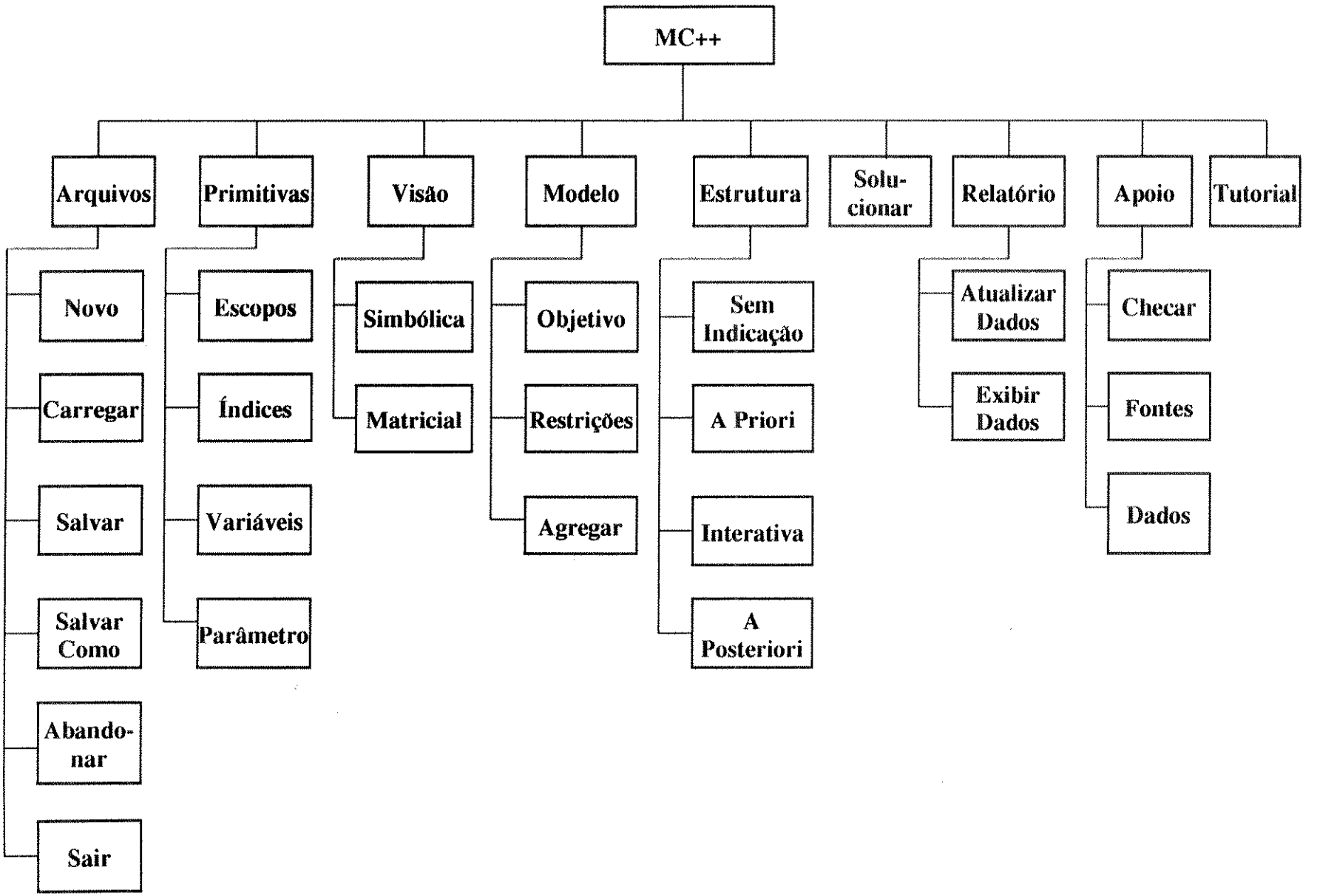
Finalmente, foram criados alguns panels para a apresentação dos dados. Um *panel* principal toma conta da tela básica do MC++ apresentando o modelo corrente nas formas simbólica e matricial. Outros *panels* foram criados para apresentação do relatório de dados e apresentação dos objetivos e restrições.

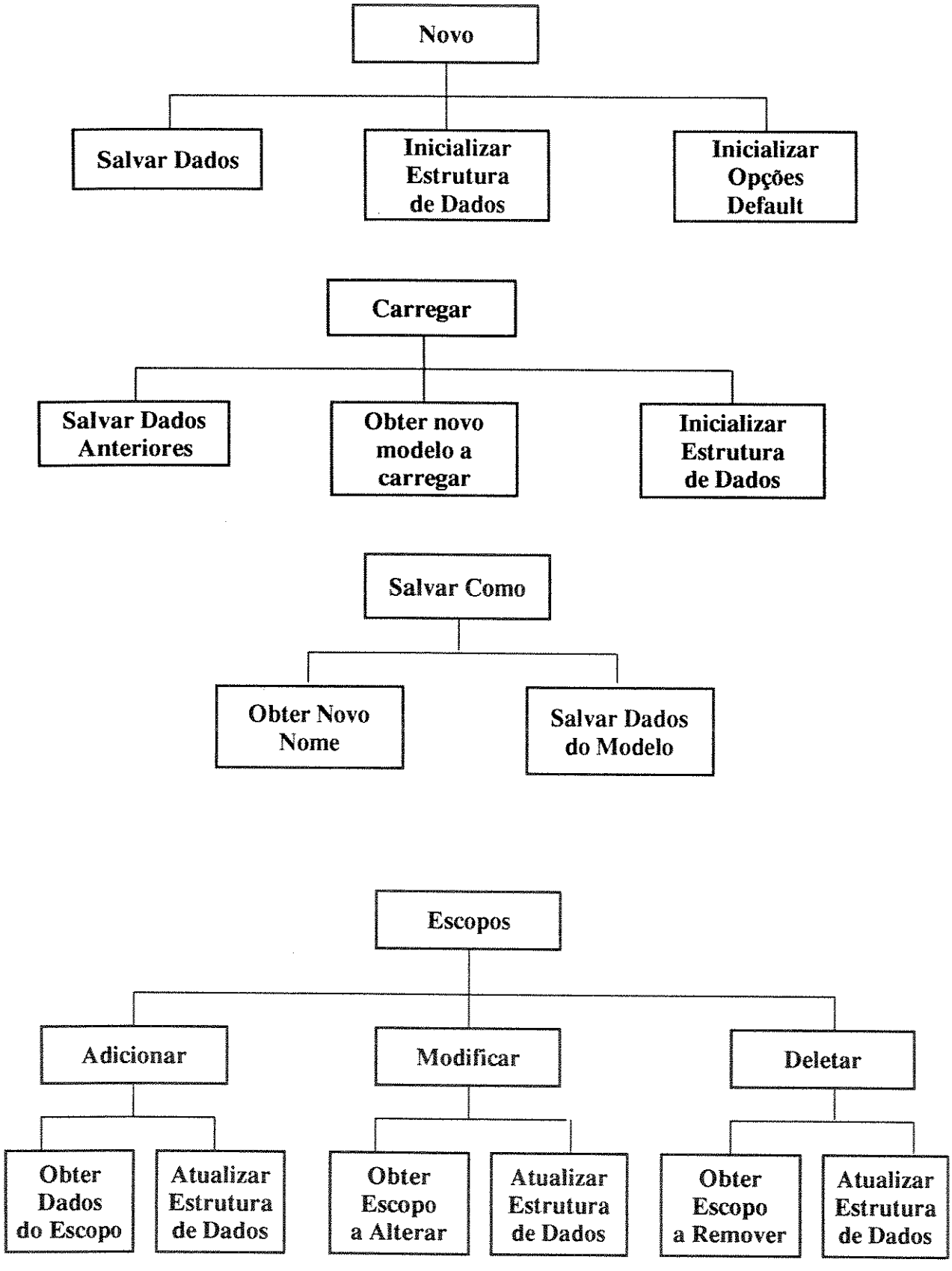
No Apêndice B são apresentadas as estruturas para tratamento de janelas através da utilização de um dicionário de dados.

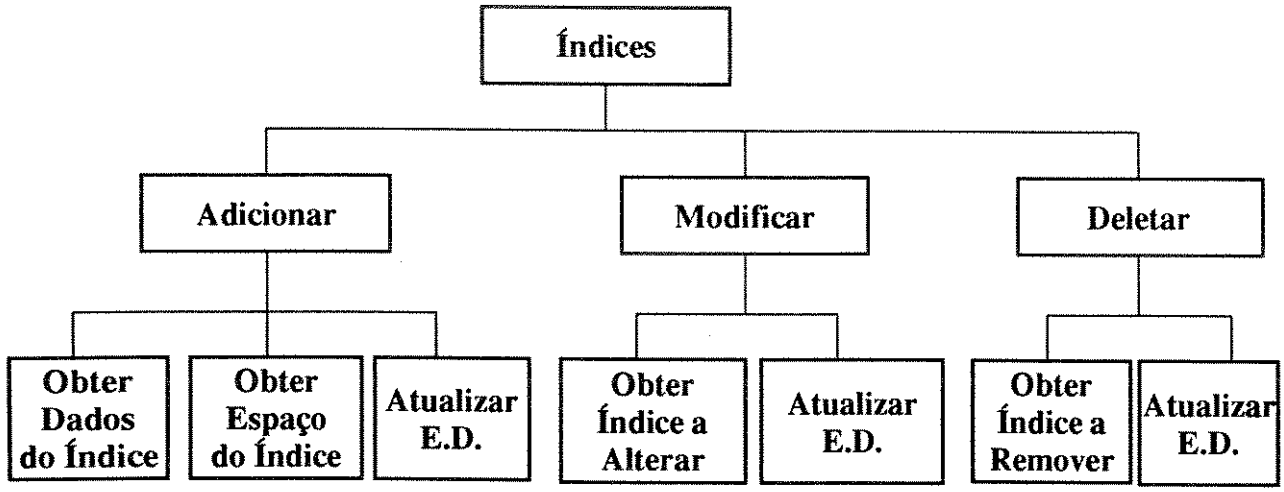
3.5 - Diagrama Hierárquico de Funções

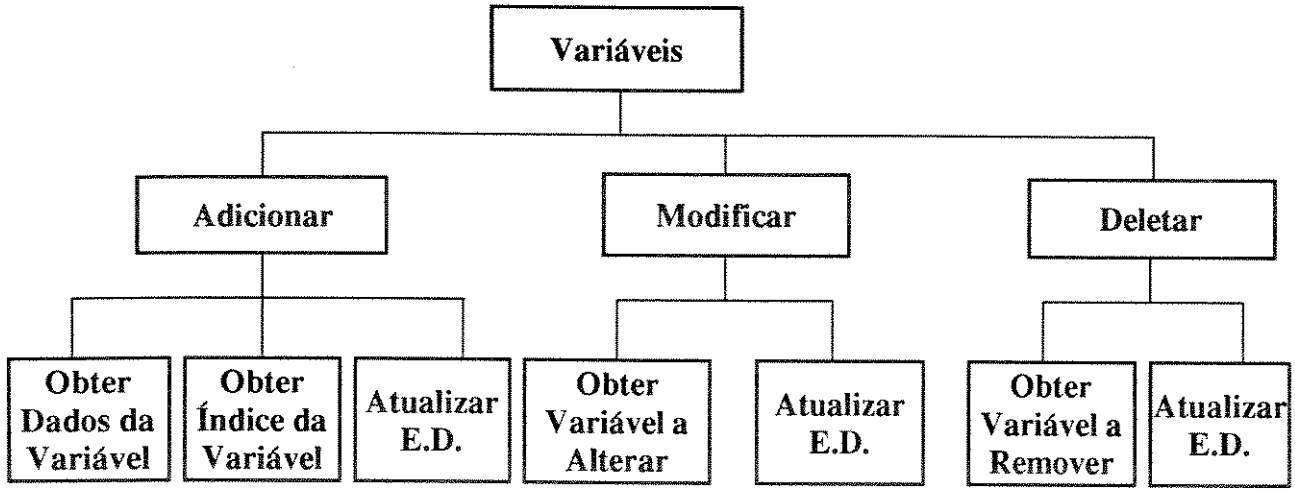
A seguir serão apresentados os diagramas hierárquicos de funções para o sistema MC++. Estes diagramas são úteis no sentido de melhorar a compreensão do fluxo de dados e características do software, bem como por representar uma fonte de informações para futuras alterações e manutenções no sistema.

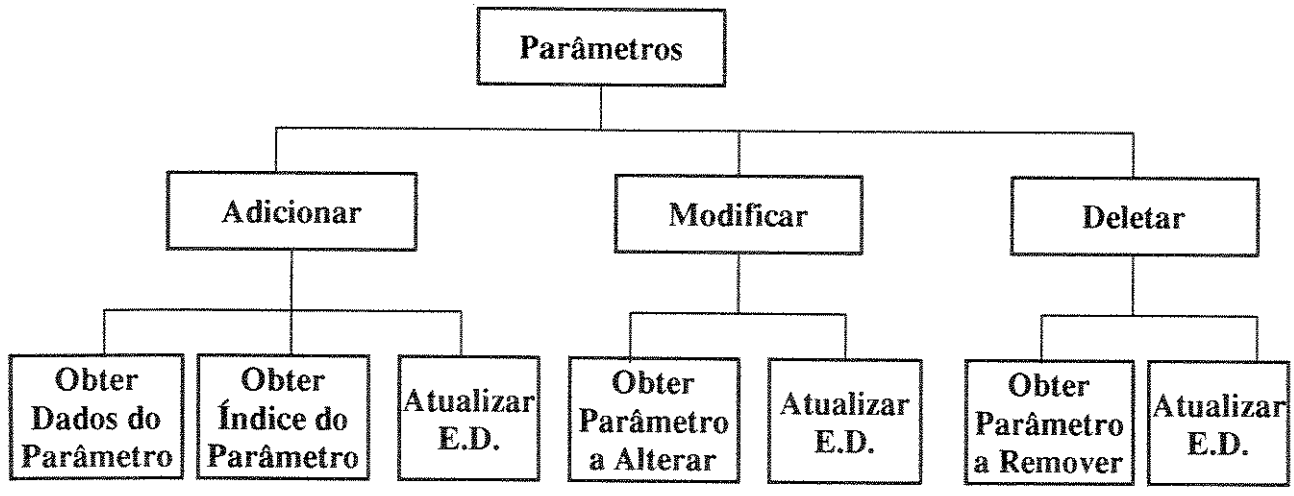
Os diagramas são em número de oito e procuram retratar o software seguindo uma linha hierárquica do tipo *top-down*.

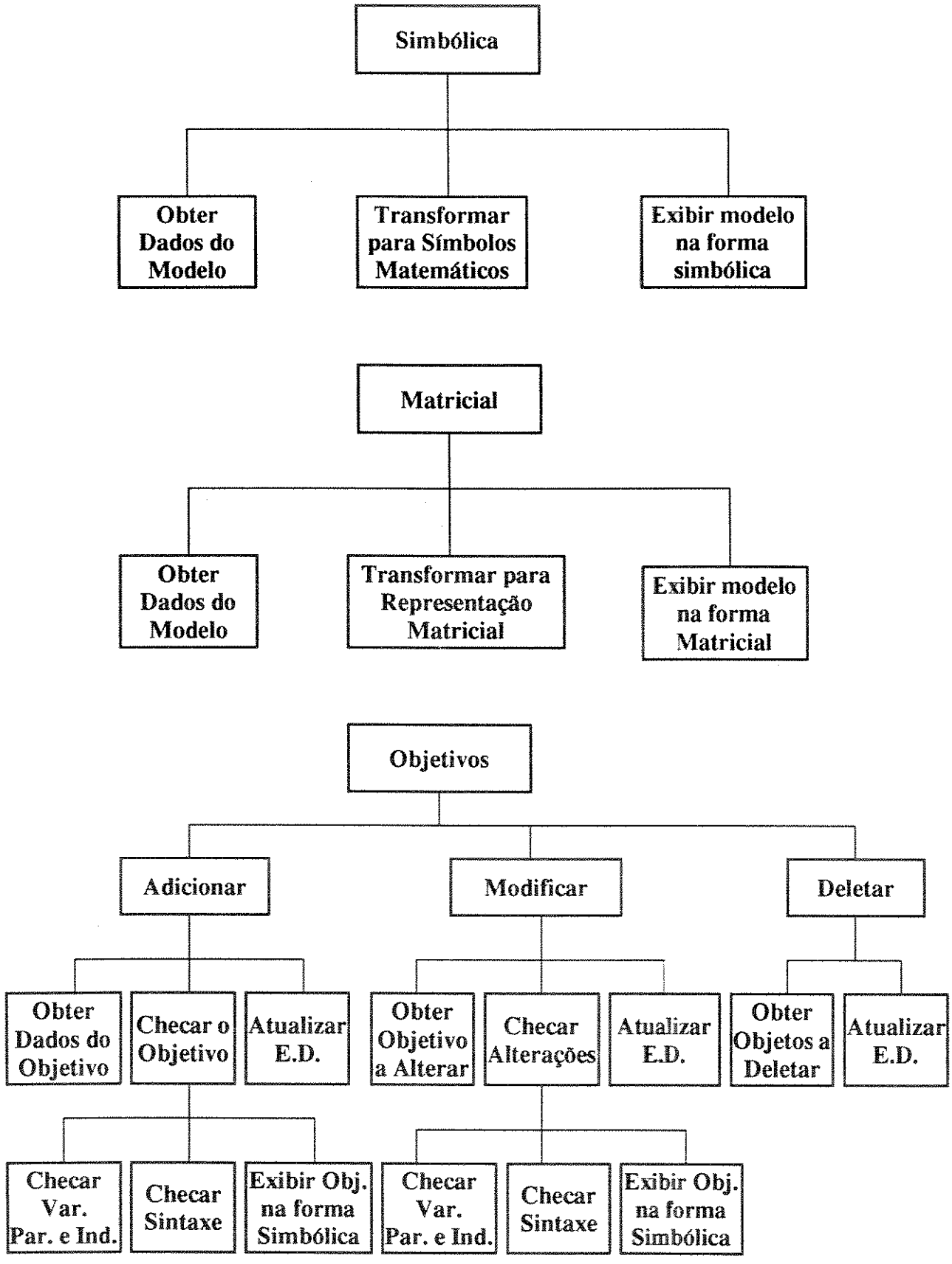

MC++
Arquivos
Primitivas
Visão
Modelo
Estrutura
Solu-cionar
Relatório
Apoio
Tutorial
Novo
Carregar
Salvar
Salvar Como
Abando-nar
Sair
Escopos
Índices
Variáveis
Parâmetro
Simbólica
Matricial
Objetivo
Restrições
Agregar
Sem Indicação
A Priori
Interativa
A Posteriori
Atualizar Dados
Exibir Dados
Checar
Fontes
Dados

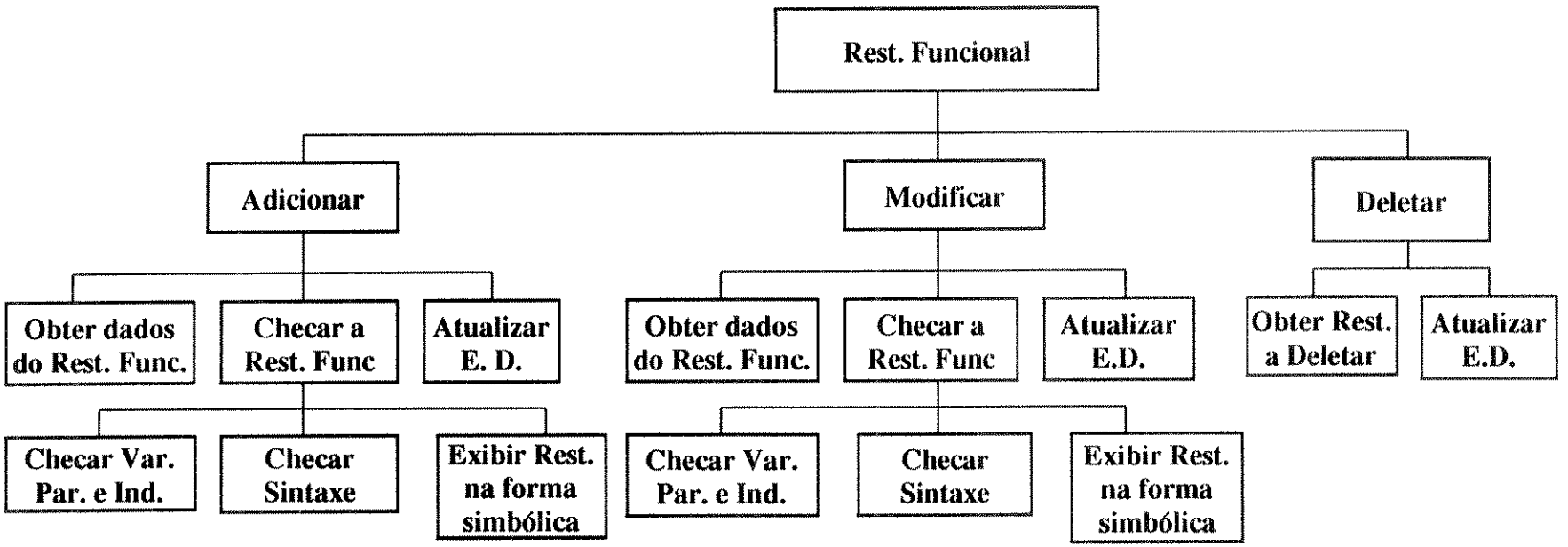

Novo
Salvar Dados
Inicializar Estrutura de Dados
Inicializar Opções Default
Carregar
Salvar Dados Anteriores
Obter novo modelo a carregar
Inicializar Estrutura de Dados
Salvar Como
Obter Novo Nome
Salvar Dados do Modelo
Escopos
Adicionar
Modificar
Deletar
Obter Dados do Escopo
Atualizar Estrutura de Dados
Obter Escopo a Alterar
Atualizar Estrutura de Dados
Obter Escopo a Remover
Atualizar Estrutura de Dados

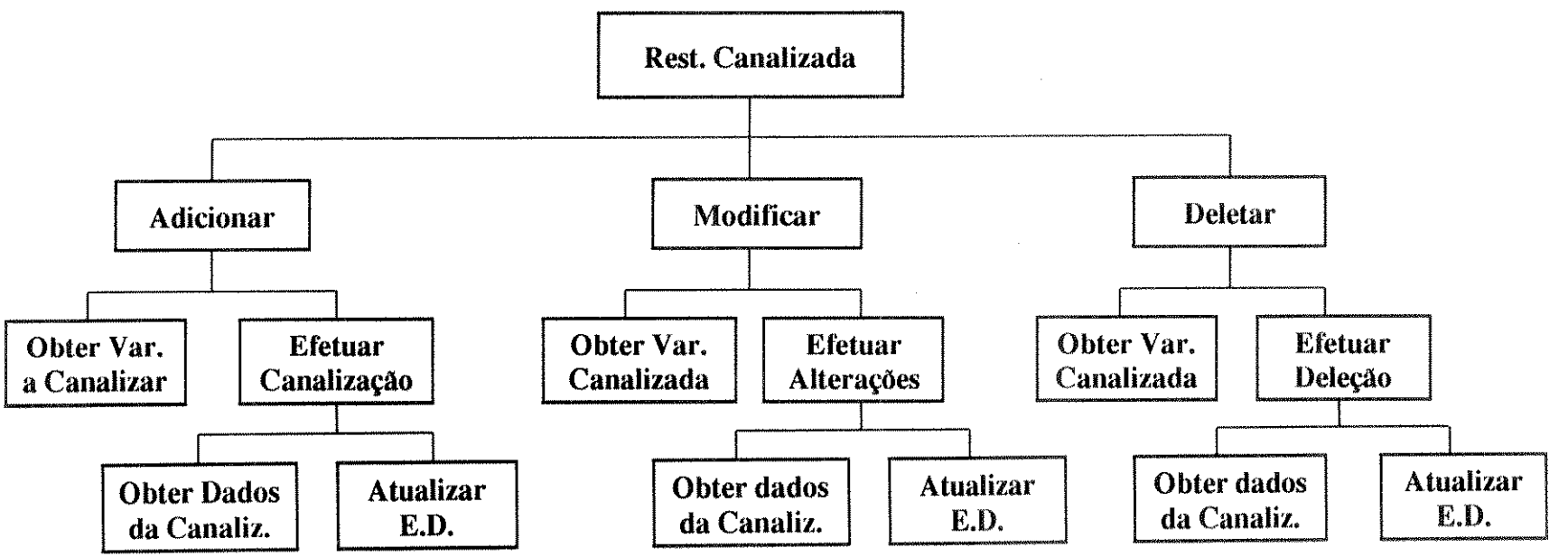

Índices
Adicionar
Modificar
Deletar
Obter Dados do Índice
Obter Espaço do Índice
Atualizar E.D.
Obter Índice a Alterar
Atualizar E.D.
Obter Índice a Remover
Atualizar E.D.

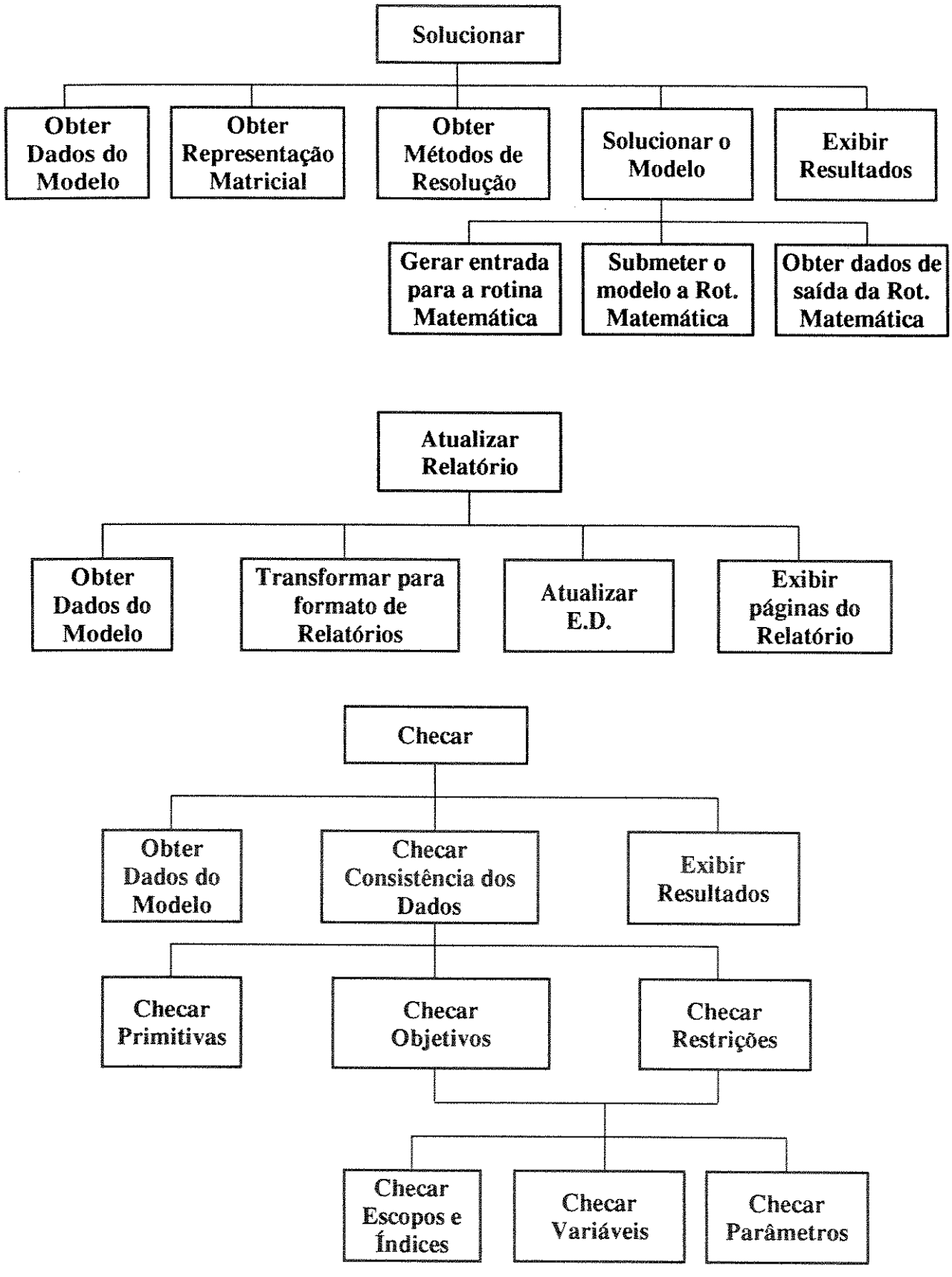

Variáveis
Adicionar
Modificar
Deletar
Obter Dados da Variável
Obter Índice da Variável
Atualizar E.D.
Obter Variável a Alterar
Atualizar E.D.
Obter Variável a Remover
Atualizar E.D.

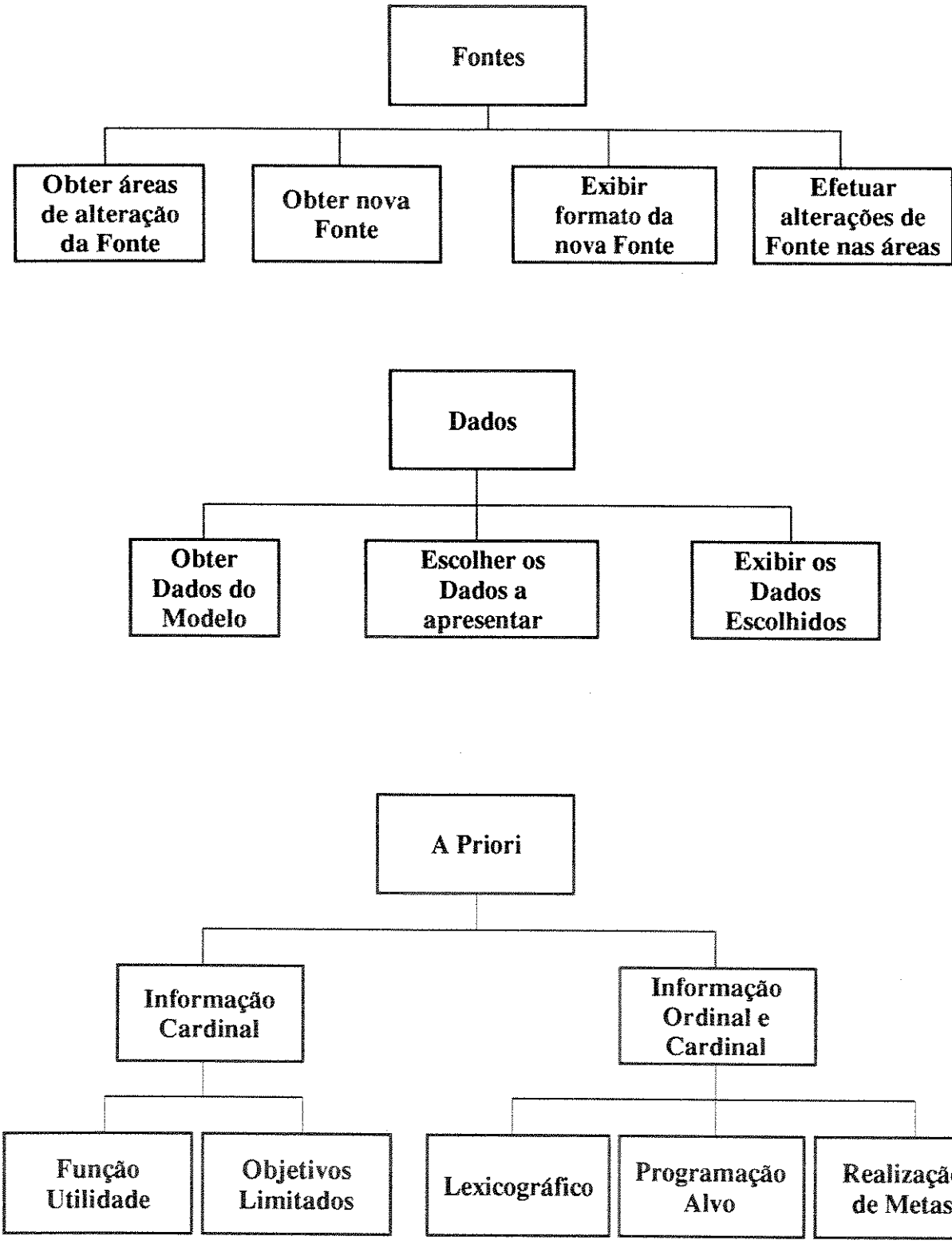

Parâmetros
Adicionar
Modificar
Deletar
Obter Dados do Parâmetro
Obter Índice do Parâmetro
Atualizar E.D.
Obter Parâmetro a Alterar
Atualizar E.D.
Obter Parâmetro a Remover
Atualizar E.D.

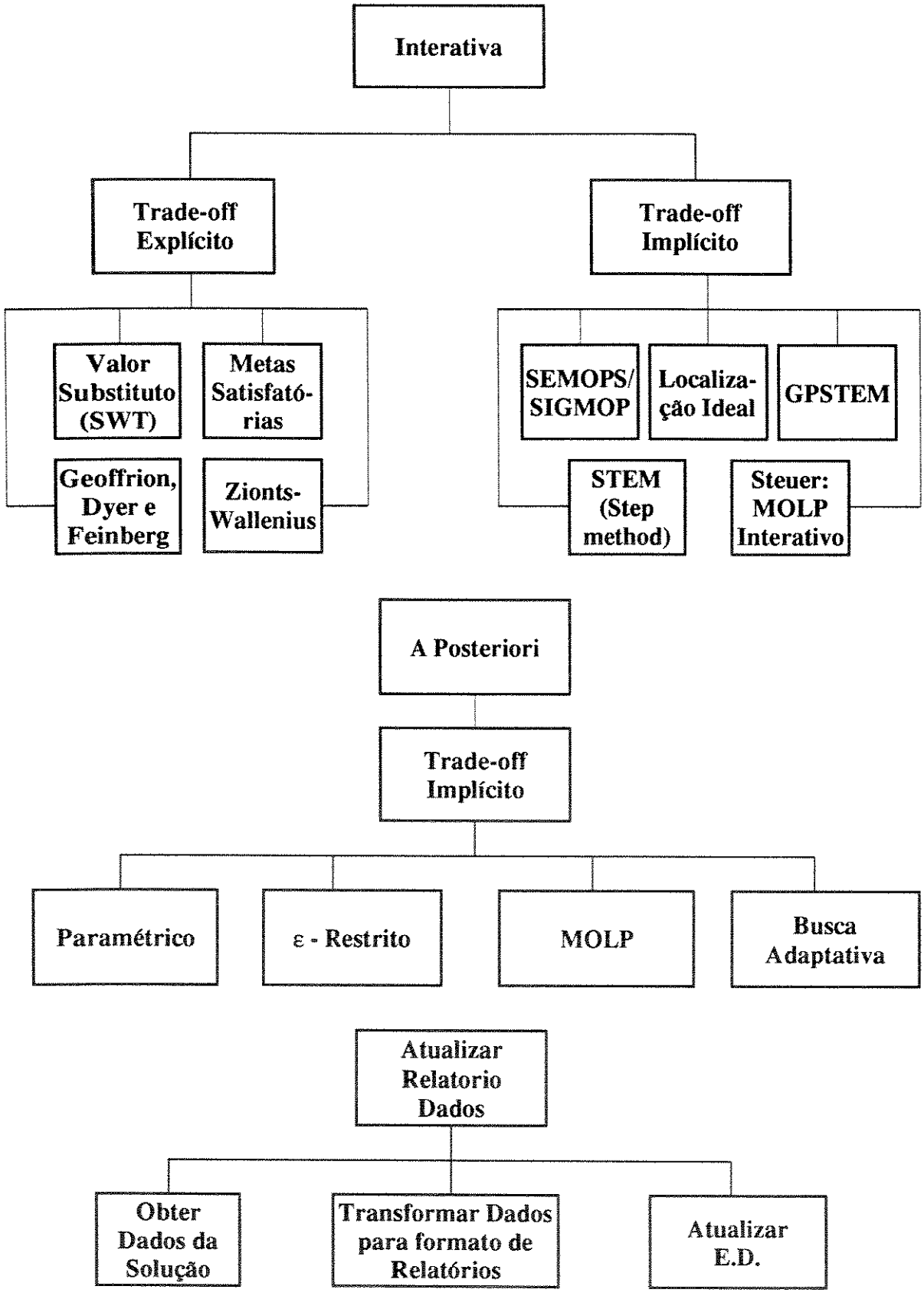

Simbólica
Obter Dados do Modelo
Transformar para Símbolos Matemáticos
Exibir modelo na forma simbólica
Matricial
Obter Dados do Modelo
Transformar para Representação Matricial
Exibir modelo na forma Matricial
Objetivos
Adicionar
Modificar
Deletar
Obter Dados do Objetivo
Checar o Objetivo
Atualizar E.D.
Obter Objetivo a Alterar
Checar Alterações
Atualizar E.D.
Obter Objetos a Deletar
Atualizar E.D.
Checar Var. Par. e Ind.
Checar Sintaxe
Exibir Obj. na forma Simbólica
Checar Var. Par. e Ind.
Checar Sintaxe
Exibir Obj. na forma Simbólica


Rest. Funcional
Adicionar
Modificar
Deletar
Obter dados do Rest. Func.
Checar a Rest. Func
Atualizar E. D.
Obter dados do Rest. Func.
Checar a Rest. Func
Atualizar E.D.
Obter Rest. a Deletar
Atualizar E.D.
Checar Var. Par. e Ind.
Checar Sintaxe
Exibir Rest. na forma simbólica
Checar Var. Par. e Ind.
Checar Sintaxe
Exibir Rest. na forma simbólica


Rest. Canalizada
Adicionar
Modificar
Deletar
Obter Var. a Canalizar
Efetuar Canalização
Obter Var. Canalizada
Efetuar Alterações
Obter Var. Canalizada
Efetuar Deleção
Obter Dados da Canaliz.
Atualizar E.D.
Obter dados da Canaliz.
Atualizar E.D.
Obter dados da Canaliz.
Atualizar E.D.


Solucionar
Obter Dados do Modelo
Obter Representação Matricial
Obter Métodos de Resolução
Solucionar o Modelo
Exibir Resultados
Gerar entrada para a rotina Matemática
Submeter o modelo a Rot. Matemática
Obter dados de saída da Rot. Matemática
Atualizar Relatório
Obter Dados do Modelo
Transformar para formato de Relatórios
Atualizar E.D.
Exibir páginas do Relatório
Checar
Obter Dados do Modelo
Checar Consistência dos Dados
Exibir Resultados
Checar Primitivas
Checar Objetivos
Checar Restrições
Checar Escopos e Índices
Checar Variáveis
Checar Parâmetros


Fontes
Obter áreas de alteração da Fonte
Obter nova Fonte
Exibir formato da nova Fonte
Efetuar alterações de Fonte nas áreas
Dados
Obter Dados do Modelo
Escolher os Dados a apresentar
Exibir os Dados Escolhidos
A Priori
Informação Cardinal
Informação Ordinal e Cardinal
Função Utilidade
Objetivos Limitados
Lexicográfico
Programação Alvo
Realização de Metas


Interativa
Trade-off Explícito
Trade-off Implícito
Valor Substituto (SWT)
Metas Satisfatórias
Geoffrion, Dyer e Feinberg
Zionts-Wallenius
SEMOPS/ SIGMOP
Localização Ideal
GPSTEM
STEM (Step method)
Steuer: MOLP Interativo
A Posteriori
Trade-off Implícito
Paramétrico
ε - Restrito
MOLP
Busca Adaptativa
Atualizar Relatorio Dados
Obter Dados da Solução
Transformar Dados para formato de Relatórios
Atualizar E.D.

3.6 - Aspectos da Implementação

O objetivo desta seção é mostrar o esforço computacional empregado na implementação do software, bem como definir o escopo de funcionamento do MC++, indicando suas características e limitações.

3.6.1 - Esforço Computacional

O esforço computacional empregado na implementação do MC++ (Versão 1.0) pode ser analisado através dos seguintes itens:

- tempo gasto no aprendizado das ferramentas utilizadas na elaboração do software;
- problemas computacionais encontrados no decorrer do desenvolvimento do software;
- enumeração de alguns parâmetros de tempo de programação como número de linhas de código fonte, tempo de compilação, linkedição e execução, etc...

3.6.1.1 - Ferramentas Utilizadas

Toda a estrutura de dados do MC++ foi elaborada utilizando a técnica de orientação a objetos fornecida pela linguagem C++. A estruturação dos dados através de classes implica numa maior segurança e facilidade de manipulação dos mesmos (encapsulamento, hereditariedade, etc...) além de proporcionar um código fonte mais compacto e elegante.

A elaboração das classes (objetos) exige uma familiaridade com o conceito de programação orientada a objetos. A solidificação destes conceitos demanda um certo grau de estudo e desenvolvimentos computacionais intensos nesse sentido.

As classes elaboradas para o tratamento da estruturas de dados no MC++ são em número de dezesseis (16), sendo que cada classe é composta por um arquivo com extensão **.h** (arquivo de cabeçalho) e um arquivo com extensão .C (arquivo de implementação). No apêndice A estão descritos as classes (somente os **.h**) principais utilizadas no MC++.

A seguir temos uma listagem dos nomes das classes implementadas e uma breve descrição das mesmas.

- **1) slistas.h e slistas.C**: tratamento para uma lista genérica de elementos utilizando uma lista ligada de apontadores;
- **2) vlistas.h e vlistas.C**: tratamento para uma lista genérica de elementos utilizando vetor de apontadores;
- **3) cadeia.h e cadeia.C**: tratamento de strings;

- **4) listcadeia.h e listcadeia.C**: tratamento de listas de strings;
- **5) vetores.h e vetores.C**: tratamento de vetores unidimensionais;
- **6) matn.h e matn.C**: tratamento para matrizes de n dimensõoes;
- **7) mc.escopo.h e mc.escopo.C**: tratamento de escopos e lista de escopos;
- **8) mc.indice.h e mc.indice.C**: tratamento de índices e lista de índices;
- **9) mc.decisao.h e mc.decisao.C**: tratamento de variáveis de decisão e lista de variáveis de decisão;
- **10) mc.param.h e mc.param.C**: tratamento de parâmetros e lista de parâmetros;
- **11) mc.valores.h e mc.valores.C**: tratamento dos valores assumidos pelos parâmetros e lista destes valores;
- **12) mc.objetivo.h e mc.objetivo.C**: tratamento das funções objetivo e lista de funções objetivos;
- **13) mc.restfunc.h e mc.restfunc.C**: tratamento de restrições funcionais e lista de restrições funcionais;
- **14) mc.restcanal.h e mc.restcanal.C**: tratamento de restrições canalizadas e lista de restrições canalizadas;
- **15) mc.equacao.h e mc.equacao.C**: tratamento das equações (objetivos e restrições funcionais) do ponto de vista matricial (numérico).
- **16) mc.list_eq.h e mc.list_eq.C**: tratamento das listas de equações. Utilizada juntamente com a classe de equações para gerar as matrizes necessárias aos métodos de otimização (Ex: Método Simplex).

A interface com o usuário foi elaborada utilizando o ambiente gráfico OPENWINDOWS, através do uso das bibliotecas fornecidas juntamente com o ambiente. A escolha deste ambiente baseou-se na crescente padronização do mesmo e, sobretudo, na sua portabilidade (o OPENWINDOWS funciona num bom número de equipamentos diferentes).

Basicamente as bibliotecas (*toolkits*) utilizadas foram o XVIEW e o XLIB. O XVIEW (que tem como base o XLIB) é útil na criação e tratamento dos tipos de dados de mais alto nível como *frames*, *menus*, *panels* e *canvas* entre outros (seção 3.4.2). Já o XLIB fornece funções de mais baixo nível utilizadas para elaboração de desenhos, alteração e apresentação de fontes e produção de gráficos.

O MC++ possui 41 *frames*, sendo um *frame* principal e outros 40 *subframes* ligados ao principal. Cada *frame* deve possuir uma rotina para a criação, manipulação e ativação/desativação do mesmo.

Normalmente quando criamos *frames*, devemos associar ao mesmos um *panel*, um *canvas* ou ambos. Os *panels* contêm as estruturas utilizadas para interfacear com o usuário como por exemplo *menus, botões, listas de elementos, campos de aquisição de dados*, etc... Já os *canvas* são utilizados para a apresentação de desenhos, tabelas, gráficos, etc...

Portanto, as rotinas responsáveis pela criação dos *frames* devem também cuidar da criação dos *panels* e/ou *canvas* associados. Isto implica, no caso dos *panels*, no posicionamento exato dos *botões*, *menus*, *listas* e demais estruturas dentro do espaço a eles reservado. No caso dos *canvas*, devemos estabelecer a sua localização e tamanho dentro do *frame*, bem como fornecer rotinas que atuem sobre o mesmo atualizando o seu conteúdo (redesenhando quando necessário).

Do mesmo modo que estão estabelecidas as rotinas que atualizam os *canvas*, devem também ser fornecidas as rotinas para o tratamento dos itens de um *panel*. Por exemplo: cada *botão* de um *panel* deve ter uma rotina associada que será chamada quando clicamos o mouse no mesmo, ou ainda, quando entramos com um dado devemos ter uma rotina de validação associada ao item do *panel* que adquiriu o dado.

Este tipo de enfoque de programação é chamada de programação orientada a eventos, pois quando clicamos num *botão* ou acessamos um item de um *menu* estamos gerando um evento que será refletido em algum outro ponto do programa.

Além do 41 *frames* já citados, o MC++ possui ainda 41 *panels* (cada *subframe* possui por default um *panel*), 10 *canvas*, 15 tipos de fontes diferentes e 10 *menus* com nomes (vários menus sem nomes foram criados e associados a um *botão*).

Um outro aspecto que deve ser comentado é a apresentação dos objetivos e restrições na forma simbólica/matricial e a apresentação do relatório do modelo.

Estas exibições de dados são realizadas utilizando a estrutura *canvas*, ou seja, cada elemento apresentado nesta estrutura deve ser posicionado e desenhado individualmente. O programador deve estar ciente do tamanho do *canvas* (altura e largura), das especificações de cada fonte (número de pontos, espaço entre linhas, altura, largura, etc..) e, de posse dessas informações, elaborar rotinas que atualizem os *canvas* respeitando os seus tamanhos e sendo genéricas o suficiente para permitir que uma alteração no tipo da fonte não prejudique a apresentação dos dados.

Finalmente, a análise sintática dos objetivos e restrições desenvolvida através dos utilitários **lex** e **yacc** (seção 3.3.13) foi responsável por considerável parte do esforço computacional envolvido no projeto.

Além do processo de aprendizado dos utilitários, surgiram problemas de comunicação entre o MC++ e o código gerado pelo **lex/yacc**. Por default, o programa produzido pelo **lex/yacc** utiliza a entrada e a saída padrão do sistema operacional (teclado/vídeo ou arquivos), além disso, era

interessante passar os nomes das variáveis, parâmetros, índices e seus escopo de variação para o analisador sintático a fim de realizar checagens preliminares.

Este problema foi resolvido através do estudo do código fonte gerado pelo **lex/yacc**. O fonte foi alterado de maneira a se tornar uma subrotina que recebia como entrada uma expressão a ser analisada e uma lista contendo os nomes das variáveis, parâmetros, índices e escopos. Como saída, é gerada uma mensagem de erro ou uma cópia da expressão analisada num formato mais compreensível para o computador (situação onde não ocorrem erros).

3.6.1.2 - Problemas Computacionais

Alguns problemas de implementação merecem ser comentados a fim de dimensionar o esforço envolvido na implementação desta versão do MC++.

Além dos problemas já discutidos no item anterior (classes, interface com o usuário e análise léxica), podemos citar como outros problemas as transformações entre os espaços de abstração (Capítulo 2, Figura 2.2), o tratamento de matrizes com elementos no espaço de dimensão **n** e a checagem da consistência de um modelo.

A transformação de um modelo de um espaço de abstração para outro (Figura 3.38) implica numa série de checagens e validações que devem ser feitas a fim de montar os vetores correspondentes aos objetivos e restrições de uma maneira correta.

A transformação de uma restrição funcional definida como uma somatória e tendo um ou mais índices do tipo **Para todo** implica na geração de vários vetores correspondentes àquela restrição.

Por exemplo:

$$\sum_{i=1}^{10} C_{i,j,k}\ x_i \quad \text{para} \quad j = 1, \ldots, 5 \ \text{e} \ k = 1, \ldots, 3$$

implica na geração de quinze restrições, ou seja, quinze vetores contendo os valores de $C_{i,j,k}$ correspondentes.

Como o MC++ trata problemas multiobjetivos, temos aqui uma nova transformação acarretada por este enfoque dado aos problemas de otimização. Cada método multiobjetivo implementado possui características próprias que exigem que os dados no formato matricial sejam alterados segundo as suas particularidades e exigências. Assim sendo, cada método implementado tem associado uma rotina de transformação de dados.

Após a modificação processada no modelo matricial pelo método de resolução, temos ainda que realizar uma nova transformação para que o modelo possa ser entendido pelas rotinas de otimização (Método Simplex por exemplo). Ou ainda, caso seja definida uma rotina de otimização externa ao MC++, temos que gerar arquivos de comunicação entre essa rotina e o MC++ (obs: esta característica ainda não foi acrescentada ao MC++).

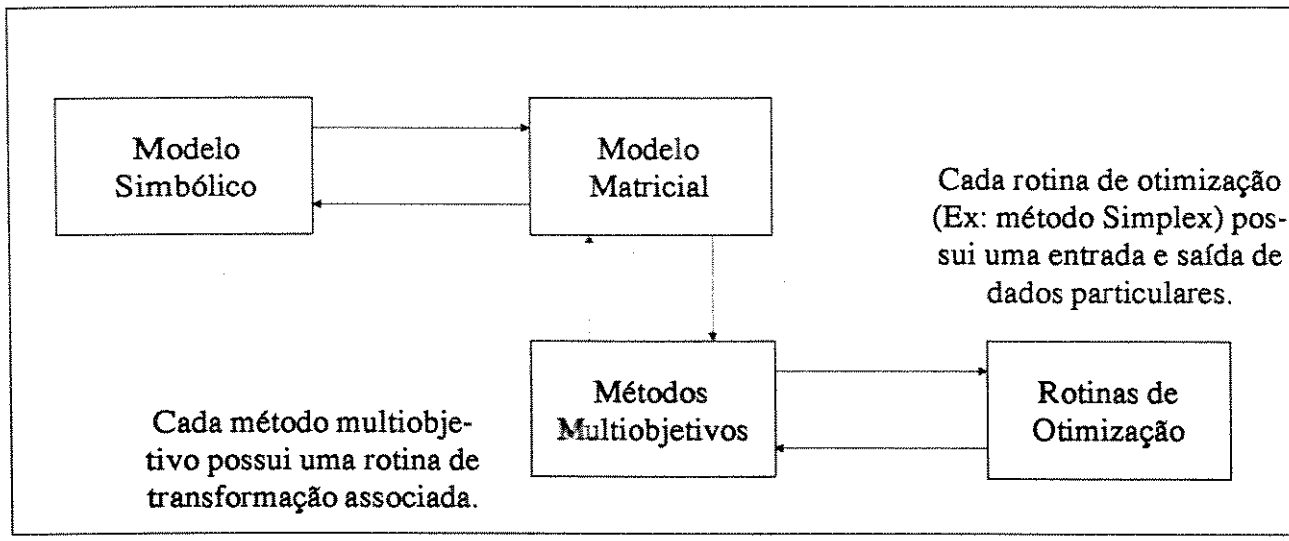


Figura 3.38 -Transformacões nos Espaços de Abstração

Nesta vesão do MC++, as variáveis e parâmetros podem ter até cinco índices associados, ou seja, podemos ter variáveis do tipo $x_{i,j,k,l,m}$. Do ponto de vista simbólico, este fato não acarreta maiores problemas, porém do ponto de vista de armazenamento dos dados (matricial), temos alguns problemas a serem resolvidos.

Um parâmetro ou uma variável pode ser visto como uma matriz possuindo elementos na dimensão **n** , onde a dimensão é dada pelo número de índices. Computacionalmente esta matriz de elementos na dimensão **n** deve ser tratada (transformada) como um vetor. Da mesma forma este vetor deve ser submetido a transformações inversas de modo a obtermos, em algum momento, um valor particular da matriz. Estes processos são utilizados tanto nas rotinas de transformação entre espaços de abstração como nas rotinas para armazenamento e tratamento de variáveis e parâmetros.

Uma classe para o tratamento dessas matrizes foi especialmente elaborada. O nome desta classe é **matn** (matn.h e matn.C) e possui várias rotinas que manipulam este tipo de estrutura de dados.

A checagem da consistência de um modelo matemático foi um dos problemas que mais consumiram tempo. Este processo consta da verificação e validação de todas as primitivas, objetivos e restrições com o intuito de estabelecer se um modelo é consistente e se pode ser submetido a um dos métodos de resolução.

No processo de validação, as sintaxes dos objetivos e restrições devem ser reavaliadas, pois várias alterações no modelo podem ter ocorrido no período decorrido entre a definição dos objetivos/restrições e da resolução do modelo.

Além disso, o analisador semântico realiza apenas checagens iniciais quando nos referimos aos nome e valores de parâmetros, índices e variáveis. Devido a este fato, uma análise de erros mais apurada deve ser realizada a fim de eliminarmos possíveis erros de sintaxe. Por exemplo: a expressão [Sum_{i} (C_{k , j} x_{ j })] está sintaticamente correta do ponto de vista do analisador sintático

(**lex/yacc**), porém notamos claramente que ocorre uma inconsistência entre o índice da somatória e os índices do parâmetro e da variável (**i** , **j** , **k** são índices, C é um parâmetro e **x** é uma variável).

3.6.1.3 - Parâmetros de Implementação

A seguir serão enumerados alguns parâmetros que podem ajudar na determinação do nível de computação envolvido na implementação do sistema.

- O programa possui aproximadamente 30.000 linhas de código fonte;
- Os códigos fontes estão distribuídos entre 94 arquivos, sendo 23 arquivos de inclusão (.h), 16 arquivos de implemetação de classes (.C), 50 arquivos que controlam o MC++ (.C) e 5 arquivos relativos ao analisador semântico (.lex , .yacc e .c);
- O arquivo de definição de constantes (mc.defines.h) possui aproximadamente 650 constantes. Todos os valores de comparação e mensagens utilizadas no MC++ são definidos como constantes, pois isto facilita a manutenção do programa e torna as alterações mais simples (a alteração do MC++ para uma outra língua implicaria somente na mudança do arquivo de definições);
- O tempo médio para compilar os 50 arquivos que controlam o MC++ está entre 20 e 30 minutos (utilizando uma Sparc Sun do tipo IPX);
- O tempo médio para linkedição do programa é de aproximadamente 70 segundos;
- O tempo médio para compilação de um módulo é de aproximadamente 30 segundos;
- O arquivo executável do MC++ tem 819 Kbytes;
- Os 94 arquivos que constituem o software ocupam 500 Kbytes de espaço em disco;
- O tempo médio para carregar o programa na memória é de aproximadamente 3 minutos;
- É interessante que a máquina onde esteja sendo executado o MC++ tenha mais de 8 Mbytes de RAM, pois isto o torna mais rápido.

Um dado interessante de ser calculado seria uma estimativa do tempo gasto em todas as compilações e linkedições realizadas até gerar esta versão do MC++. Esta estimativa poderia ser feita multiplicando o tempo médio de linkedição mais o tempo médio para compilar um único módulo pelo número de vezes que este processo foi realizado (uma estimativa deste número é claro). Este tempo refletiria o período que eu fiquei olhando para o monitor enquanto a CPU trabalhava. Achei melhor não realizar esta estatística em prol do meu bem estar físico e mental.

Os números relativos aos tempos de compilação, linkedição e execução apresentados acima dependem do quão carregada está a rede no momento da execução do comando e da máquina com que se está trabalhando.

3.6.2 - Limitações

Alguns aspectos de implementação do MC++ ainda não alcançaram um ponto considerado satisfatório. A seguir serão descritas algumas características **ainda** não implementadas e algumas outras que necessitam ser aperfeiçoadas.

A Tabela 1.2 do Capítulo 1 exibe 19 métodos diferentes para a resolução de problemas multiobjetivos (sem considerar os métodos que utilizam lógica nebulosa). Dos 19 métodos, **apenas** 7 foram implementados (no mínimo um método de cada classe). Futuras versões do MC++ incluirão os outros métodos além de aperfeiçoar os já existentes.

Métodos implementados nesta versão.

- **Critério Global**: sem articulação;
- **Objetivos Limitados**: informação cardinal com articulação a priori;
- **Lexicográfico**: informação ordinal e cardinal com articulação a priori;
- **Geoffrion, Dyer e Feinberg**: trade-off explícito com articulação interativa;
- **STEM**: trade-off implícito com articulação interativa;
- **Paramétrico**: trade-off implícito com articulação a posteriori;
- **ε-Restrito**: trade-off implícito com articulação a posteriori;

O método do Critério Global implementado corresponde a obter a solução do problema resolvendo

$$\min_{x \in \chi} \quad \left(\sum_{i=1}^{m} \beta_i \left(f_i(x) - \underline{y_i} \right)^p \right), \; p \geq 1$$

onde β_i são constantes positivas que poderam os desvios das funções em relação aos seus valores ideais.

Internamente existe uma rotina implementando o Método Simplex que é utilizada como suporte para os métodos multiobjetivos. Uma rotina deste tipo está sempre em evolução, pois novas características podem e devem ser acrescentadas a fim de torná-la mais genérica e eficaz (esparsidade, análise de sensibilidade, variáveis livres, etc...). Uma outra facilidade que será incluída em versões futuras é a possibilidade do usuário escolher sua própria rotina de otimização. Desta maneira, seria estipulado um programa externo ao MC++ que se comunicaria através de arquivos (o padrão MPSX da IBM poderia ser utilizado como padrão de comunicação - Capítulo 2, seção 2.3.3.1).

O tratamento para exibição das telas de *HELP* está todo implementado, porém as mensagens ainda não foram incluídas e o modo de exibição das mesmas também não foi definido. Este trabalho foi deixado para o final pois era mais importante a elaboração de uma versão inicial que fechasse o ciclo definição, modelagem , resolução e apresentação de resultados.

A opção **Tutorial** ainda não foi implementada, visto que a sua falta não compromete o sistema como um todo. O objetivo do **Tutorial** é fornecer explicações e exemplos para que um usuário inexperiente possa se adaptar mais facilmente as características e particularidades do software.

Uma das características do MC++ que precisa ser melhorada é justamente aquela que foi mais trabalhosa computacionalmente: a análise sintática das expressões e o tratamento das inconsistências de um modelo. Como podemos notar, estas duas operações estão intimamente relacionadas e poderiam ser tratadas conjuntamente a fim de produzir um resultado mais eficiente que o atual.

Com o objetivo de elaborar um versão inicial, várias simplificações foram feitas na gramática e o tratamento de erros de consistências se tornou muito rígido na sua análise. Assim sendo, algumas expressões que são válidas do ponto de vista matemático não são aceitas pelo analisador. É claro que estas expressões sempre podem ser manipuladas a fim de produzir uma expressão que o analisador entenda, porém isto exige que o usuário tenha um trabalho extra executado fora do ambiente do MC++.

3.7 - Conclusão

Este capítulo descreve todas as características do Software de Programação Matemática Multiobjetivo MC++. Temos aqui uma descrição dos principais componentes do sistema, a estrutura de dados, a base de dados bem como um diagrama hierárquico de funções. Com isso pretende-se dar uma visão dos recursos oferecidos pelo sistema no tocante a modelagem, resolução de modelos e análise dos resultados obtidos.

Do Capítulo 1, aproveitou-se toda a discussão sobre programação multiobjetivo, sobretudo a parte relativa aos métodos que estavam diretamente ligados a implementação do sistema.

Do Capítulo 2, aproveitou-se várias idéias e técnicas para a elaboração de um sistema que fosse amigável, eficiente e robusto. É claro que um sistema deste tipo está sempre em desenvolvimento e, portanto, existem muitas outras características que podem ser incluídas no futuro para que o mesmo se torne cada vez melhor.

A maneira como foi concebido o sistema permite esta expansão, inclusive com a possibilidade de utilização de outros softwares (sistema abertos).

Desta maneira, temos em mãos um sistema que permite a modelagem, resolução e análise de problemas de programação multiobjetivo (problemas de um único objetivo também são permitidos) funcionando sobre uma plataforma gráfica.

Capítulo 4
Uma Aplicação MC++: Problema da Dieta Multiobjetivo

4.1 - Introdução

O problema da dieta multiobjetivo (Hwang & Massud, 1979) consiste em encontrar as quantidades de certos alimentos que devem ser consumidos a fim de obter os requisitos nutricionais básicos e ao mesmo tempo minimizar os objetivos da dieta (Tabela 4.1).

Neste capítulo modelaremos o problema da dieta multiobjetivo dentro do ambiente MC++ e, utilizando os métodos implementados, solucionaremos o problema em questão. A fim de proporcionar um melhor entendimento dos métodos interativos, o método de Geoffrion, Dyer e Feinberg será executado passo a passo.

4.2 - Descrição do Problema

Para efeito de ilustração do problema, serão escolhidos 6 alimentos: leite, carne, ovos, pão, salada e suco de laranja.

As quantidades máximas de cada alimento a serem consumidas diariamente são:

- 6 pints de leite (1 pint corresponde a aproximadamente meio litro)
- 1 pound de carne (1 pound = 453,6g)
- 0.25 dúzias de ovos (3 ovos)
- 10 ounces de pão (1 ounce = 28,35g)
- 10 ounces de salada
- 4 pints de suco de laranja

Tabela 4.1 - Informações e Custos Nutricionais

----- ----- -----	Leite (pint)	Carne (pound)	Ovos (dúzias)	Pão (ounce)	Salada (ounce)	Suco de Laranja (pint)	Qtde. Recomendad (Adultos)
Vit. A (U.I.)	720	107	7080	0	134	1000	5000
Calorias (cal)	344	460	1040	75	17.4	240	2500
Colesterol (unidade)	10	20	120	0	0	0	minimizar
Proteínas (g)	18	151	78	2.5	0.2	4	63
Carboidratos (g)	24	27	0	15	1.1	52	minimizar
Ferro (mg)	0.2	10.1	13.2	0.75	0.15	1.2	12.5
Custo ($)	0.225	2.2	0.8	0.1	0.05	0.26	minimizar

Devemos encontrar as quantidades destes alimentos que satisfaçam:

- (i) um mínimo diário de vitamina A e ferro
- (ii) uma quantidade balanceada de calorias e proteínas;
- (iii) um mínimo diário de colesterol;
- (iv) um mínimo diário de carboidratos;
- (v) um custo mínimo.

Vamos definir as quantidades diárias de leite (em pints), carne (em pounds), ovos (em dúzias), pão (em ounces), salada (em ounces) e suco de laranja (em pints) através das variáveis x_1, x_2, x_3, x_4, x_5 e x_6 respectivamente. Então o problema pode ser modelado da seguinte forma:

Min $f_1(x) = 0.225\ x_1 + 2.2\ x_2 + 0.8\ x_3 + 0.1\ x_4 + 0.05\ x_5 + 0.26\ x_6$ (Custo)

Min $f_2(x) = 10\ x_1 + 20\ x_2 + 120\ x_3$ (Colesterol)

Min $f_3(x) = 24\ x_1 + 27\ x_2 + 15\ x_4 + 1.1\ x_5 + 52\ x_6$ (Carboidratos)

sujeito a:

$g_1(x) = 720\ x_1 + 107\ x_2 + 7080\ x_3 + 134\ x_5 + 1000\ x_6 \geq 5000$ (Vitamina A)

$g_2(x) = 0.2\ x_1 + 10.1\ x_2 + 13.2\ x_3 + 0.75\ x_4 + 0.15\ x_5 + 1.2\ x_6 \geq 12.5$ (Ferro)

$g_3(x) = 344\ x_1 + 460\ x_2 + 1040\ x_3 + 75\ x_4 + 17.4\ x_5 + 240\ x_6 \geq 2500$ (Calorias)

$g_4(x) = 18\ x_1 + 151\ x_2 + 78\ x_3 + 2.5\ x_4 + 0.2\ x_5 + 4\ x_6 \geq 63$ (Proteínas)

$g_5(x) = x_1 \leq 6.0$

$g_6(x) = x_2 \leq 1.0$

$g_7(x) = x_3 \leq 0.25$

$g_8(x) = x_4 \leq 10.0$

$g_9(x) = x_5 \leq 10.0$

$g_{10}(x) = x_6 \leq 4.0$

$x_i \geq 0 \quad i = 1,2,3,4,5,6$

4.3 - Definição do Problema no MC++

O primeiro passo a ser tomado será a definição de um índice e seu escopo de variação. De posse destas definições, criaremos uma variável de decisão e finalmente poderemos definir os objetivos e restrições.

Definição de Escopos

Para definir um escopo, devemos selecionar a opção **Primitivas** do menu principal e depois acessar a subopção **Escopo-adicionar**.

Após estas seleções, uma tela será apresentada para a definição do escopo. Chamaremos este escopo de **I**, sendo do tipo **numérico** com variação entre 1 e 6 (Figura 4.1).

Definição de Índices

Para definir um índice, devemos selecionar a opção **Primitivas** do menu principal e depois acessar a subopção **Índice-adicionar**.

Após estas seleções, uma tela será apresentada para a definição do índice. Chamaremos este índice de **i**, sendo **I** o seu escopo de variação (Figura 4.2).

Definição de Variáveis

Para definir uma variável, devemos selecionar a opção **Primitivas** do menu principal e depois acessar a subopção **variável-adicionar**.

Após estas seleções, uma tela será apresentada para a definição da variável. Chamaremos esta variável de **x**, sendo **i** o seu índice de variação. Com esta definição, teremos criado as variáveis x_1, x_2, ... ,x_6 que correspondem aos alimentos leite, bife, ovos, pão, salada e suco de laranja, respectivamente (Figura 4.3).

Definição de Objetivos

Para definir um objetivo, devemos selecionar a opção **Modelo** do menu principal e depois acessar a subopção **Objetivo-adicionar**.

Após estas seleções, uma tela será apresentada para a definição dos objetivos. Nesta tela daremos um identificador numérico (ID) ao objetivo, escolheremos entre maximização e minimização, definiremos a equação a ser otimizada e a documentação do objetivo em questão. Estas definições devem ser realizadas para cada um dos três objetivos do problema da dieta (Figura 4.4).

Definição de Restrições Funcionais

Para definir uma restrição funcional, devemos selecionar a opção **Modelo** do menu principal e depois acessar a subopção **Restrição Funcional-adicionar**.

Após estas seleções, uma tela será apresentada para a definição das restrições funcionais. Nesta tela daremos um identificador numérico (ID) a restrição, definiremos a equação, o sinal de limitação ($\leq$, $=$ e $\geq$), o lado direito, a documentação e ,se houver, um índice para a criação de várias restrições. Estas definições devem ser realizadas para cada uma das quatro restrições do problema da dieta (Figura 4.5).

Definição de Restrições Canalizadas

Para definir uma restrição canalizada, devemos selecionar a opção **Modelo** do menu principal e depois acessar a subopção **Restrição Canalizada-adicionar**.

Após estas seleções, uma tela será apresentada para a definição das restrições canalizadas. Nesta tela devemos escolher o limite inferior e superior de cada variável a ser canalizada. Estas definições devem ser realizadas para cada uma das seis canalizações do problema da dieta (Figura 4.6).

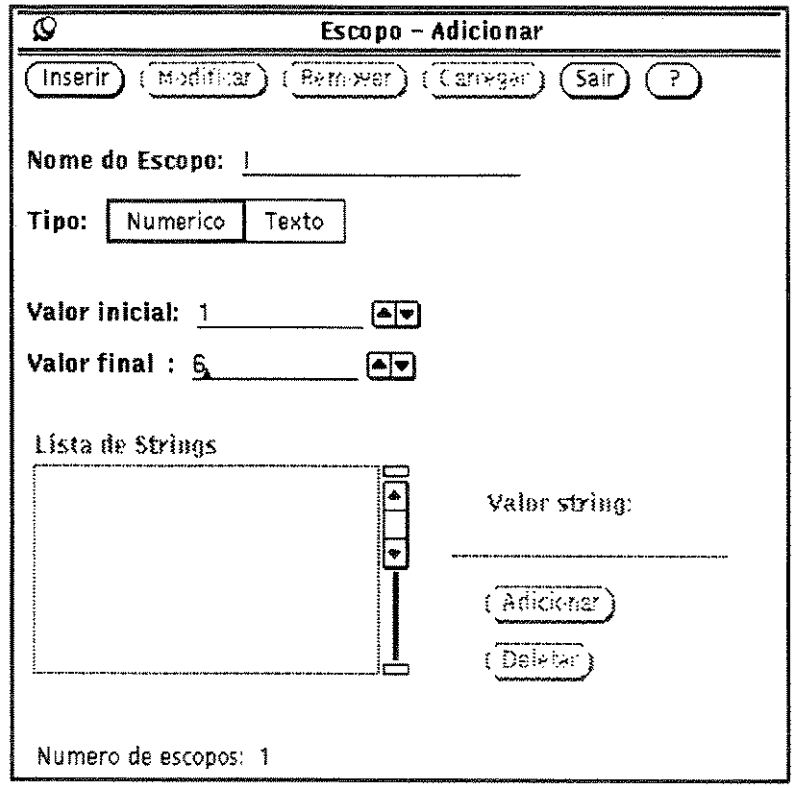


Figura 4.1 - Definição de Escopos

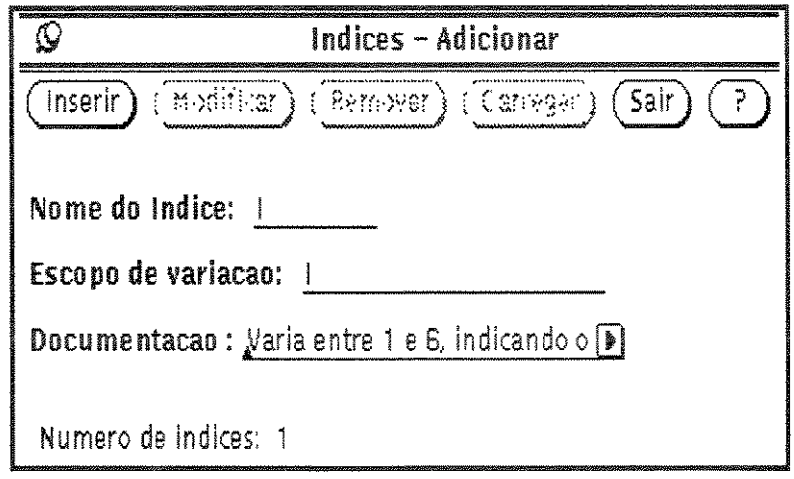


Figura 4.2 - Definição de Índices

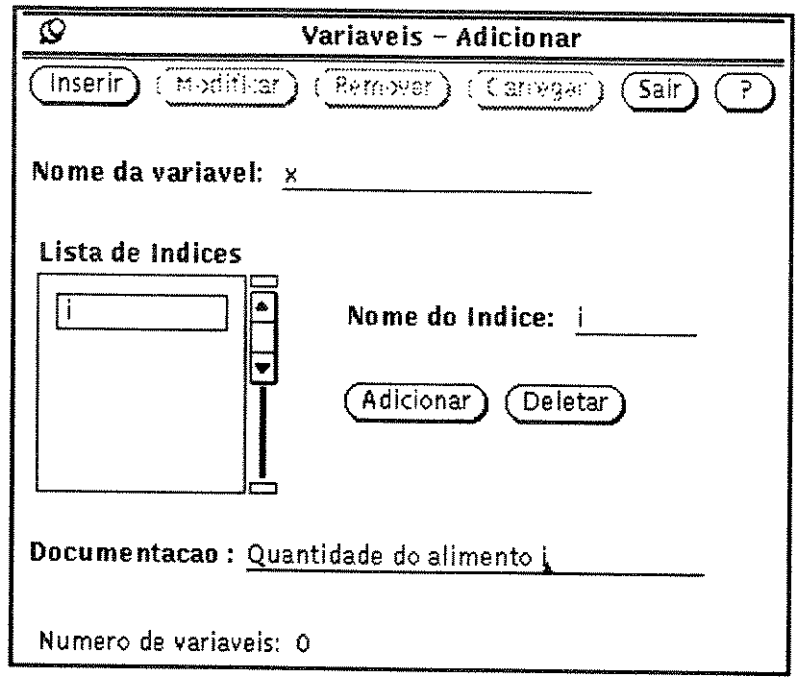


Figura 4.3 - Definição de Variáveis

Salvando o Modelo Definido

Após a definição do modelo, devemos salvá-lo em disco para uso futuro. Esta operação é efetuada através da opção **Arquivos** do menu principal. Ao selecionar esta opção, um menu para operações de escrita e leitura em disco será exibido. Devemos escolher a opção **Salvar-Como** (Figura 4.7) para salvar o modelo com um novo nome ou escolher a opção **Salvar** para salvar o modelo com o nome corrente (o nome do modelo é apresentado, entre outros locais, na linha de rodapé da tela principal).

Uma outra maneira de definir o problema seria através da criação de parâmetros que refletissem os vetores de custos e a matriz de restrições. Para utilizar esta abordagem definiremos um outro escopo de nome **K** que varia entre 1 e 4 (número de restrições funcionais) e um índice **k** variando sobre o escopo **K**

Com o novo índice definido podemos criar os parâmetros **C1** , **C2** e **C3** que serão utilizados na definição dos três objetivos, o parâmetro **A** que será utilizada na definição das quatro restrições funcionais e o parâmetro **b** que será o lado direito das restrições funcionais. Os parâmetros **C1**, **C2** e **C3** variam sobre o índice **i**, o parâmetro **A** varia sobre os índices **k** e **i** e o parâmetro **b** varia sobre o índice **k**. A definição dos parâmetros, objetivos e restrições funcionais sob este novo enfoque é mostrada nas Figuras 4.8 , 4.9, 4.10, 4.11 e 4.12

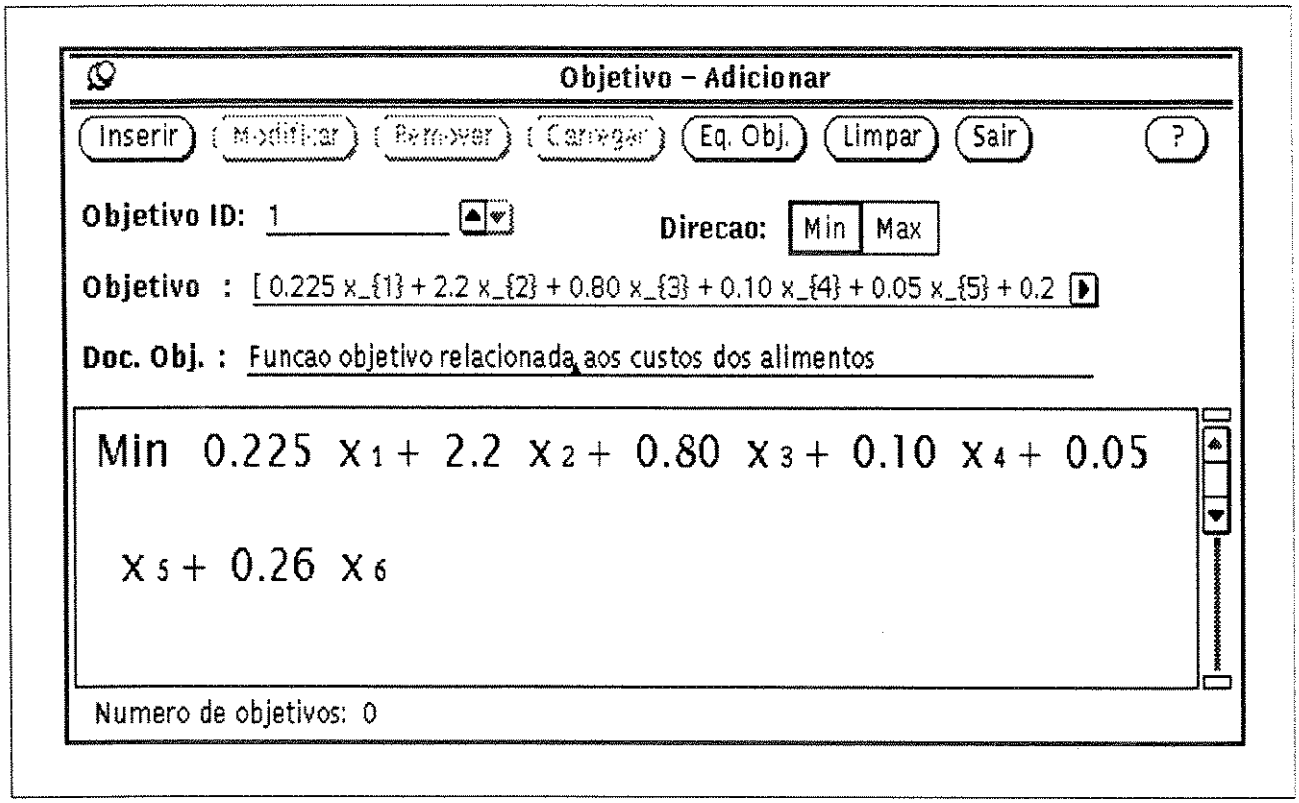


Figura 4.4 - Definição de Objetivos

4.3 - Solucionando o Problema

Para efetuar a resolução de um modelo devemos escolher um método dentro os oferecidos pelo sistema. A escolha do método é feita através da opção **Estrutura** do menu principal. Esta opção implementa a Tabela 1.2 Capítulo 1, onde os métodos são agrupados em 4 classes. Estas classes são divididas segundo o estágio onde a informação é necessária e são denominadas: *Sem Articulação*, *Articulação a Priori*, *Interativo* e *Articulação a Posteriori*.

Resolveremos o problema da dieta utilizando todos os métodos implementados, que são: **Critério Global** (Sem Articulação), **Objetivos Limitados** (A Priori - Informação Cardinal), **Lexicográfico** (A Priori -Iinformação Ordinal e Cardinal), **Paramétrico** (A Posteriori -Ttrade-off Implícito), ε-**Restrito** (A Posteriori - Trade-off Implícito), **Geoffrion, Dyer e Feinberg** (Interativo - Trade-off Explícito) e **STEM** (Interativo - Trade-off Implícito).

Critério Global

Este método (Figuras 4.15 e 4.16) é selecionado através da opção **Critério Global** do menu **Estrutura**. A tela apresentada para este método possui o valor de **p** que é a norma escolhida para a resolução do método e os valores β para cada objetivo. Devemos escolher o valor de **p** (1, 2 ou infinito) e setar os valores de β que por default são 1.0. Após definir estes parâmetros clicamos em

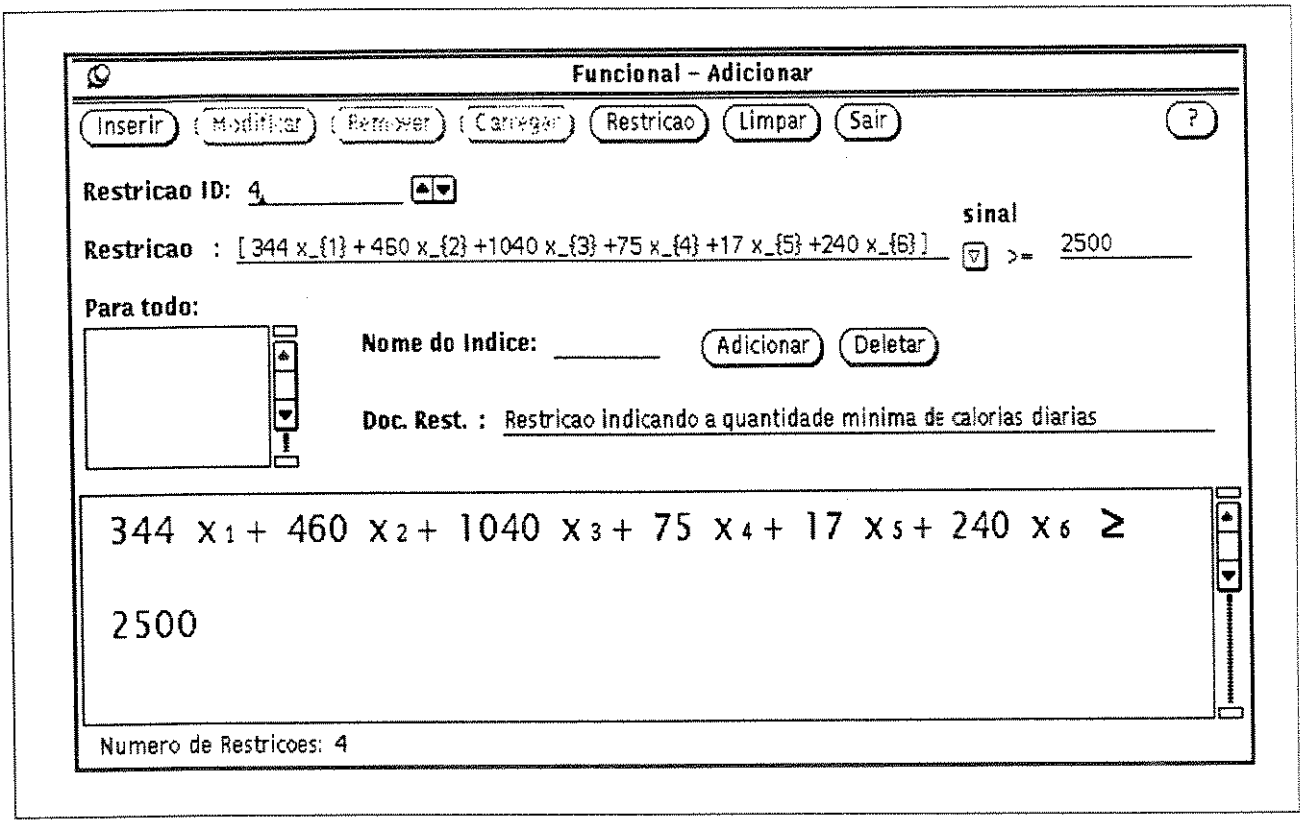


Figura 4.5 - Definição de Restrições Funcionais

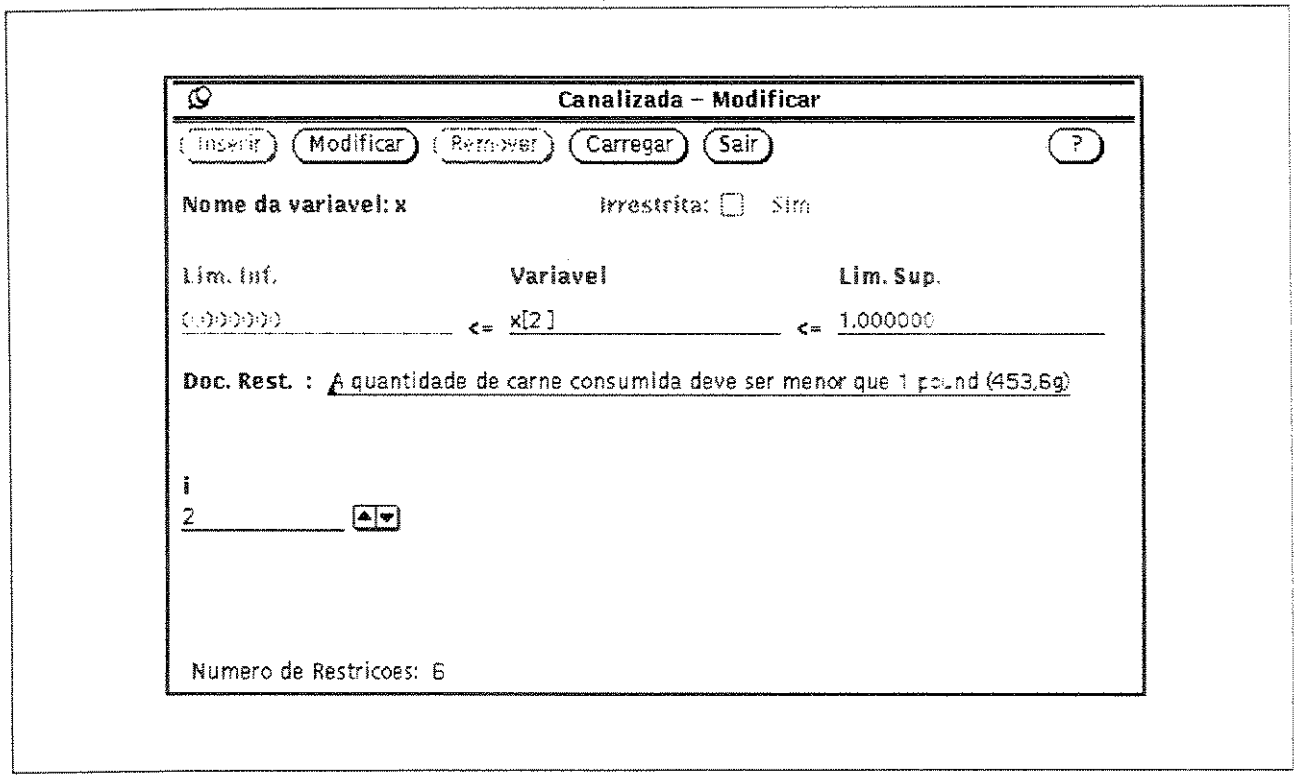


Figura 4.6 - Definição de Restrições Canalizadas

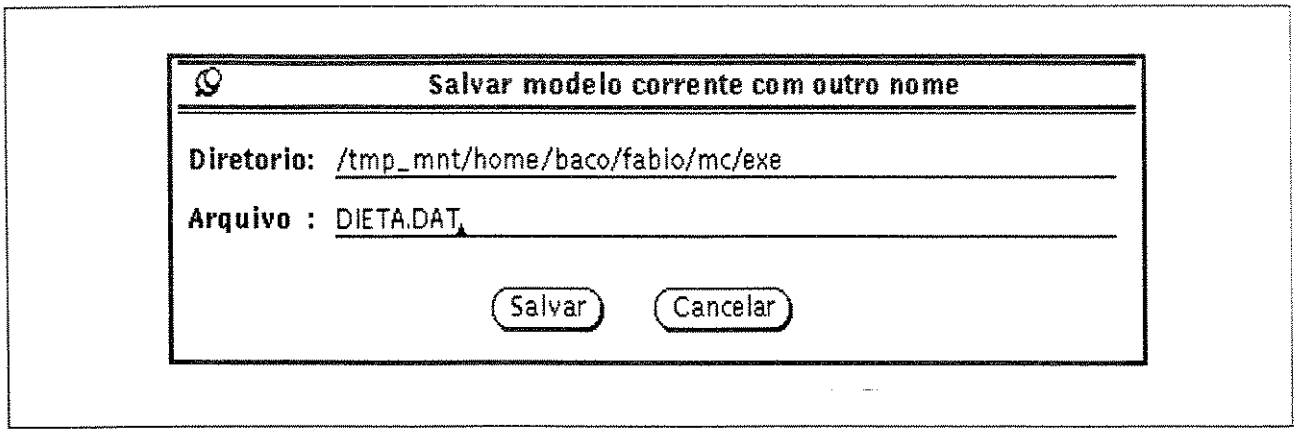


Figura 4.7 - Salvando o Modelo

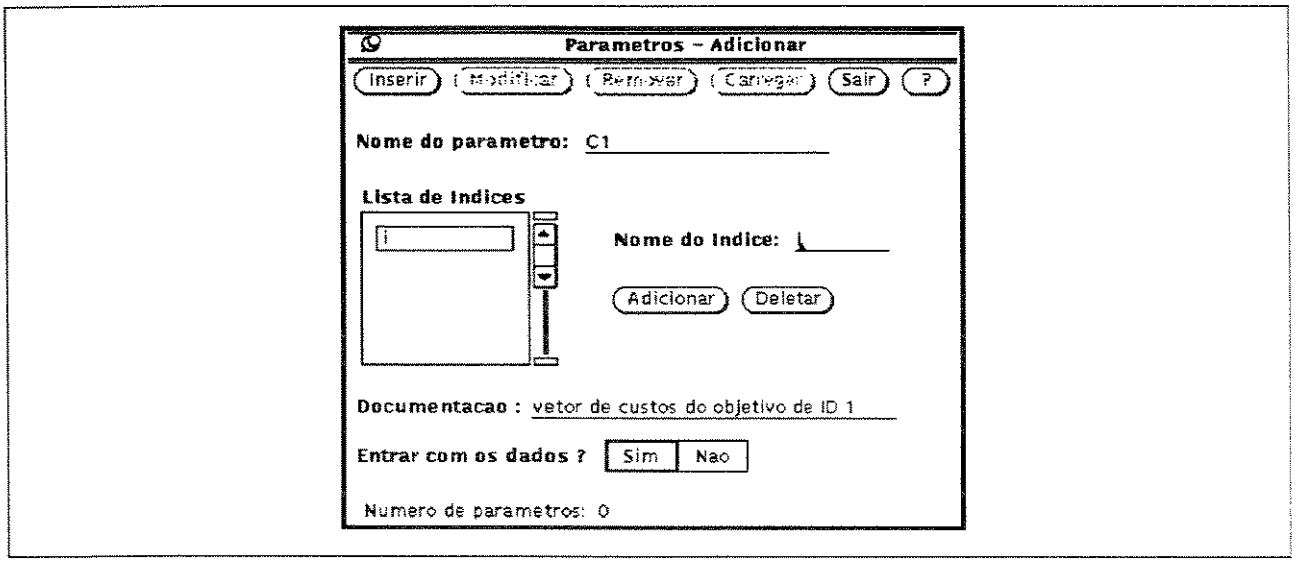


Figura 4.8 - Definição do Parâmetro C1

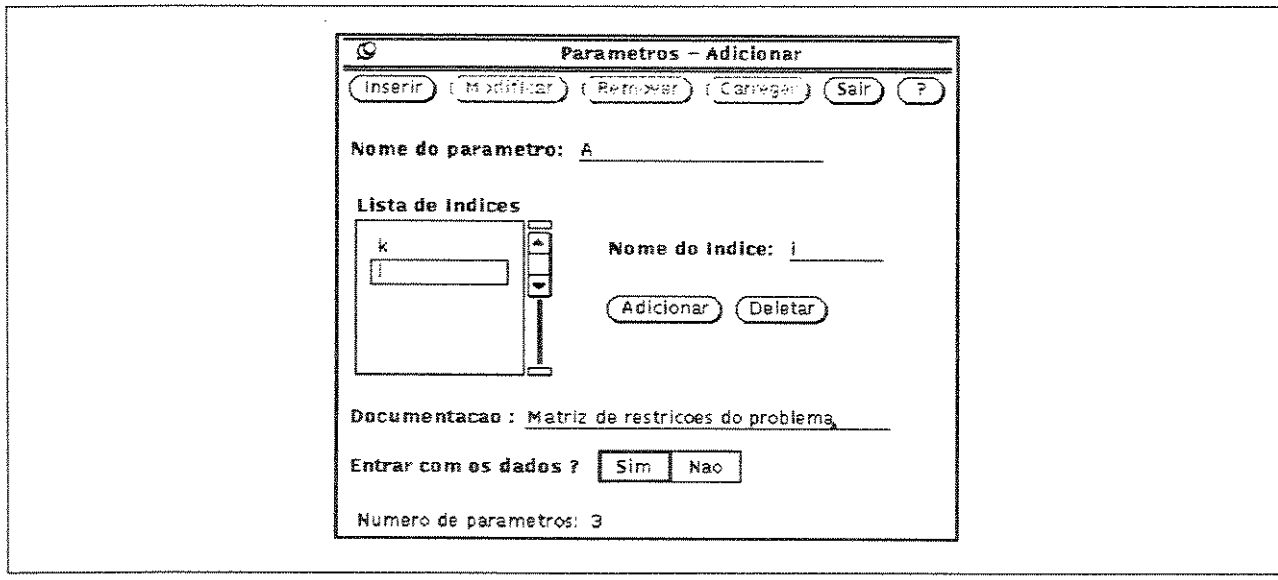


Figura 4.9 - Definição do Parâmetro A

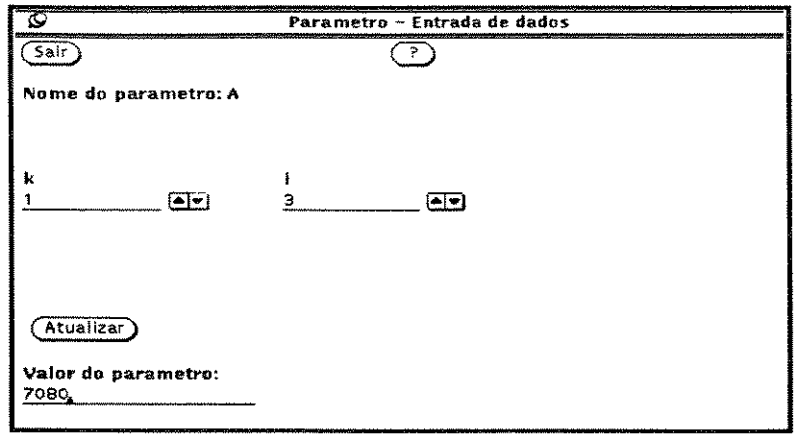


Figura 4.10 - Entrada de dados do parâmetro A

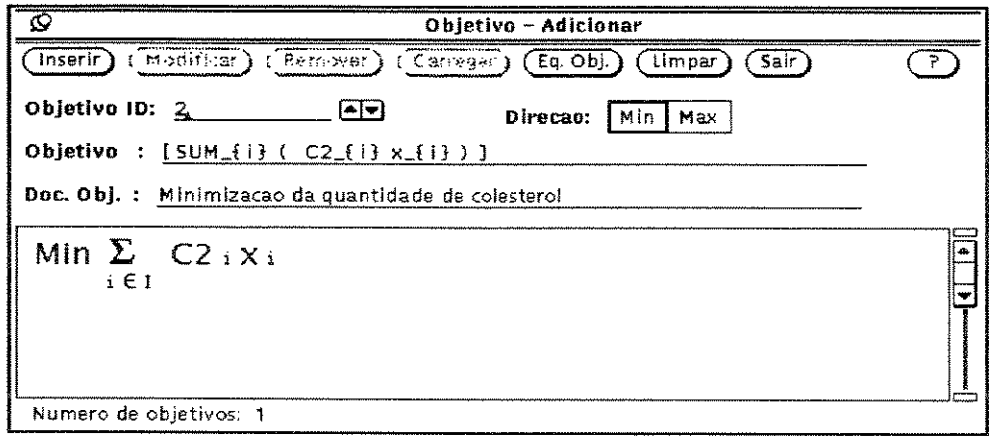


Figura 4.11 - Definição de Objetivos (Parâmetros)

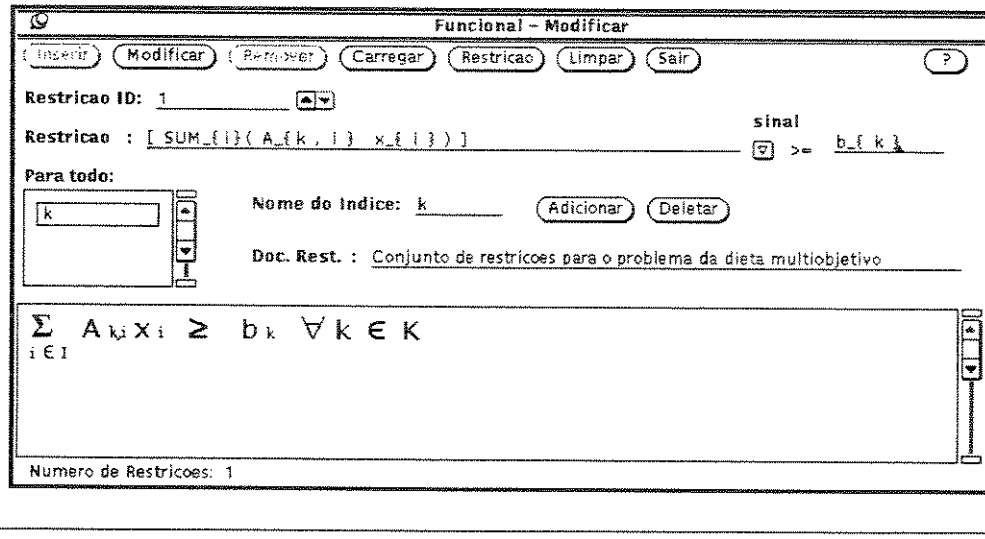


Figura 4.12 - Definição de Restrições (Parâmetros)

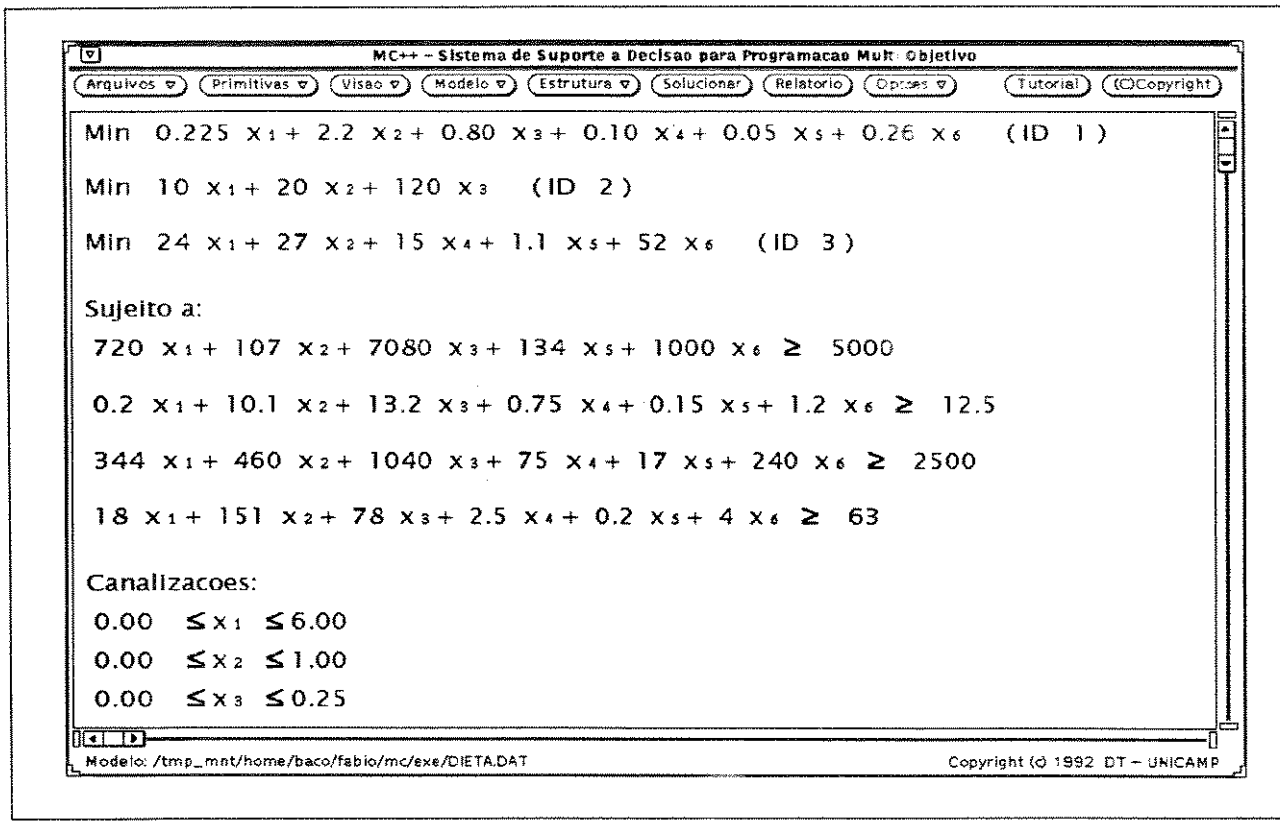


Figura 4.13 - Modelo definido sem parâmetros

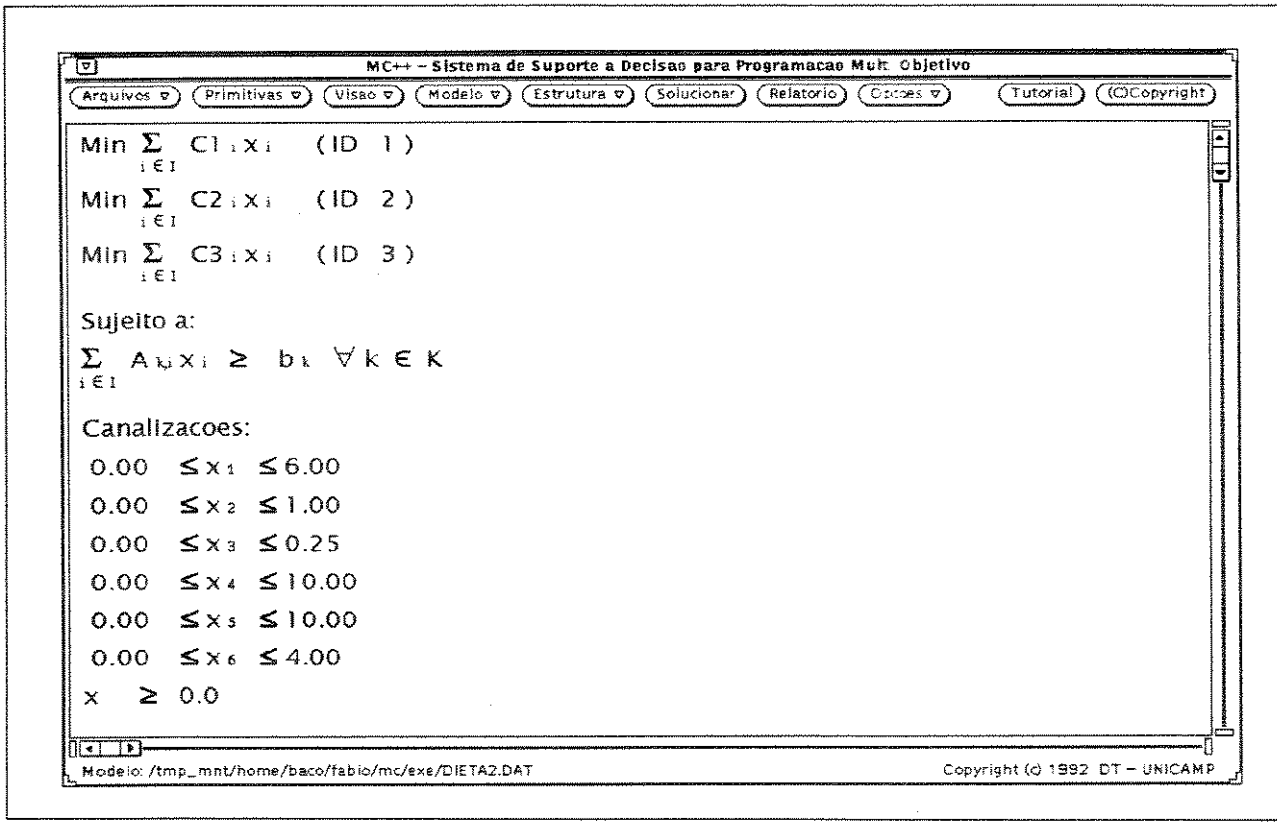


Figura 4.14 - Modelo definido com parâmetros

solucionar para efetuarmos a resolução do problema.O método do Critério Global implementado corresponde a obter a solução do problema resolvendo

$$\min_{x \in \chi} \quad \left(\sum_{i=1}^{m} \beta_i \left(f_i(x) - \underline{y_i} \right)^p \right), \; p \geq 1$$

Objetivos Limitados

Este método (Figuras 4.17 e 4.18) é selecionado através da opção **Objetivos-Limitados** do menu **Estrutura**. A tela apresentada para este método possui o ID do objetivo de referência e os limites inferior e superior para os outros objetivos (tratados como restrições). Devemos definir os limites inferior e superior para cada objetivo que não seja de referência, escolhendo o ID do mesmo e setando os valores limites. Após definir estes parâmetros clicamos em solucionar para efetuarmos a resolução do problema.

Lexicográfico

Este método (Figuras 4.19 e 4.20) é selecionado através da opção **Lexicográfico** do menu **Estrutura**. A tela apresentada para este método possui uma lista com os IDs dos objetivos e uma outra lista a ser definida com a ordem léxica dos mesmos. Para montar esta lista com a ordem de prioridade dos objetivos basta clicar sobre os IDs da lista original de objetivos. Com a lista de objetivos montada lexicograficamente, podemos definir o valor α para cada um dos objetivos (por default os valores de alfa são 0.0). Este parâmetro α representa uma porcentagem de variação num determinado objetivo na tentativa de melhorar os demais. Após definir estes parâmetros clicamos em solucionar para efetuarmos a resolução do problema.

Paramétrico

Este método (Figuras 4.21 e 4.22) é selecionado através da opção **Paramétrico** do menu **Estrutura**. A tela apresentada para este método possui uma lista com os IDs dos objetivo e o valor de ponderação dado a cada um deles. Para definirmos o valor da ponderação de cada objetivo, basta clicarmos sobre o ID do objetivo a ser ponderado e depois inicializar o valor da ponderação. As ponderações podem assumir qualquer valor positivo pois serão normalizadas durante a solução do problema. Após definir estes parâmetros clicamos em solucionar para efetuarmos a resolução do problema.

ε-Restrito

Este método (Figuras 4.23 e 4.24) é selecionado através da opção **ε-Restrito** do menu **Estrutura**. A tela apresentada para este método possui o ID do objetivo de referência, uma lista com os objetivos que não são de referência (tratados como ε-restrições) e o valor do lado direito para esses objetivos. Para definirmos o valor do lado direito devemos clicar sobre o ID de um objetivo da lista e depois definir o valor. Após definidos estes parâmetros, clicamos em solucionar para efetuarmos a resolução do problema.

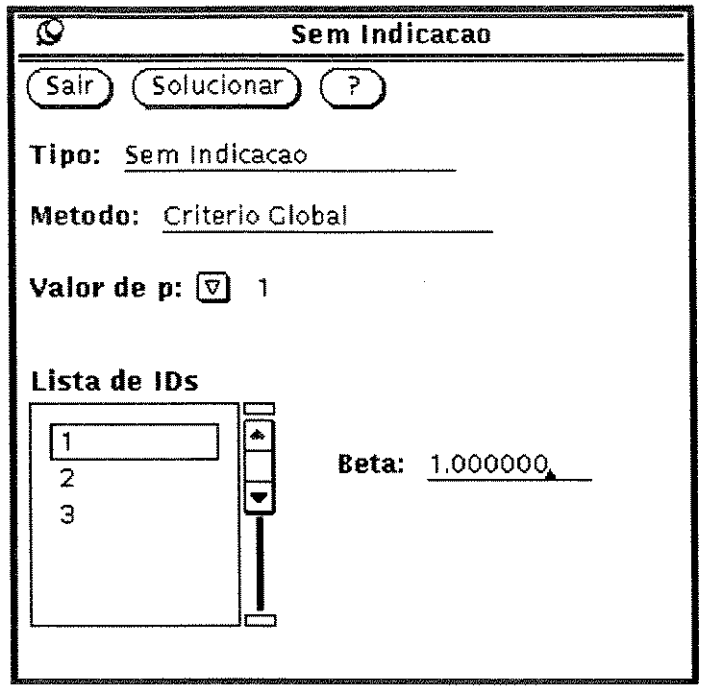


Figura 4.15 - Critério Global

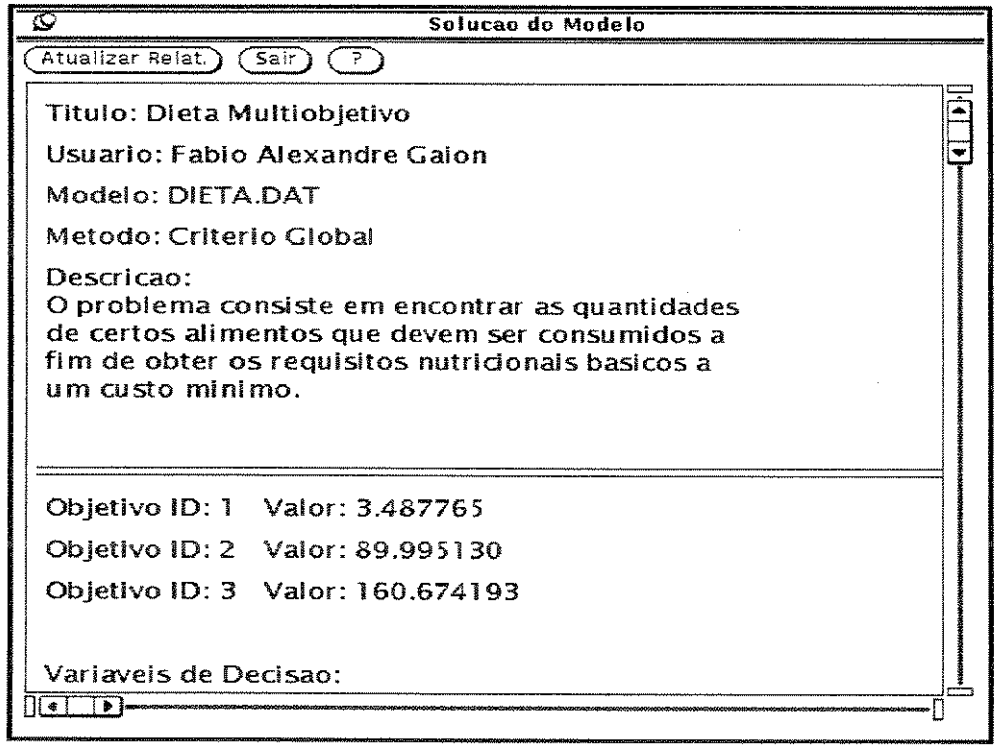


Figura 4.16 - Solução p/ Critério Global com p = 1

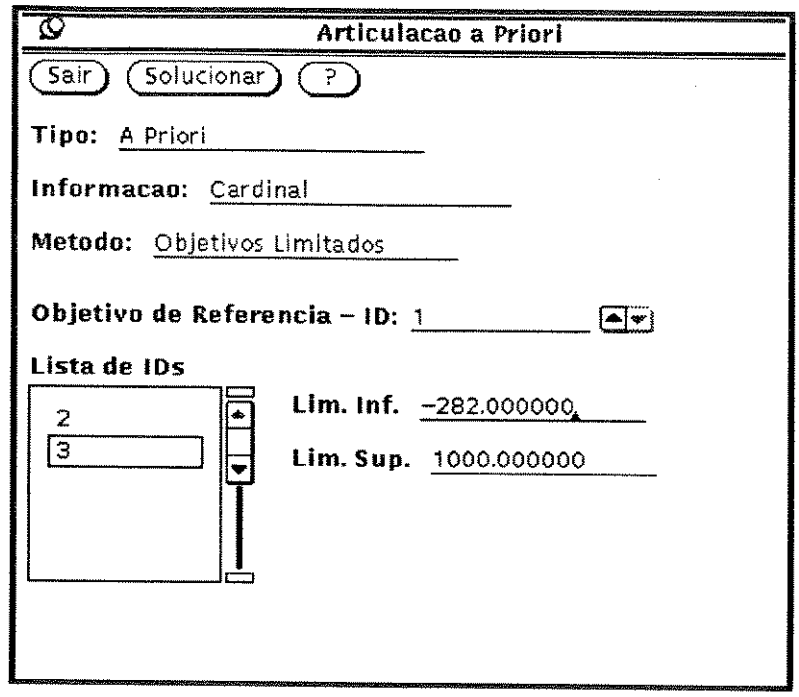


Figura 4.17 - Objetivos Limitados

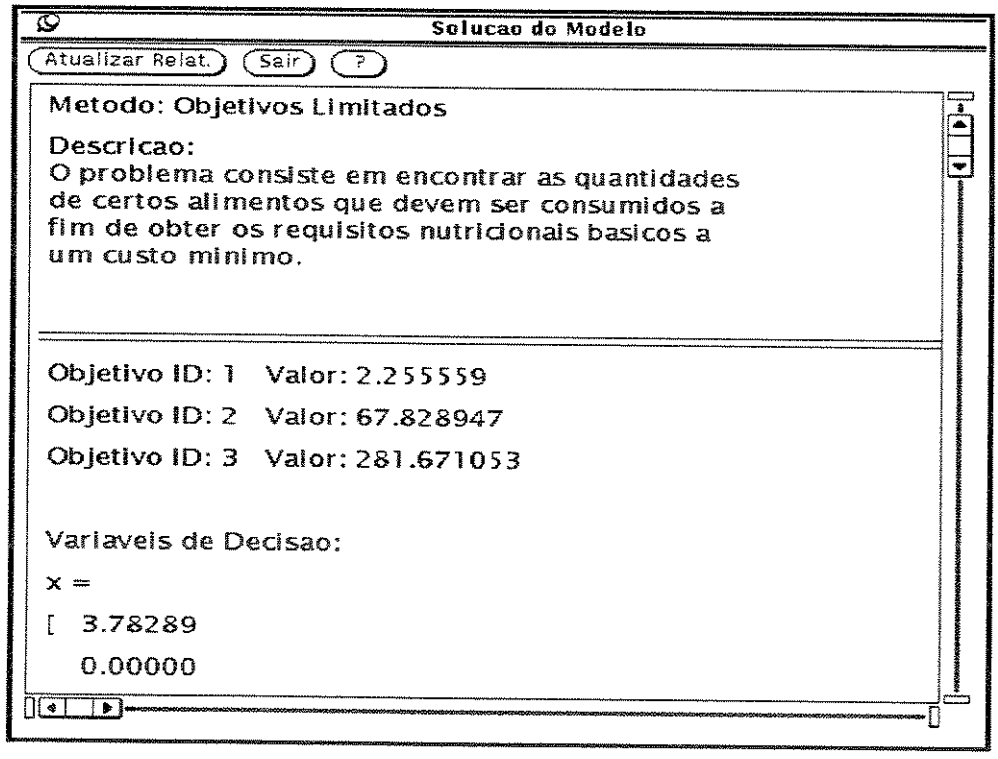


Figura 4.18 - Solução p/ o Método Obj. Limitados

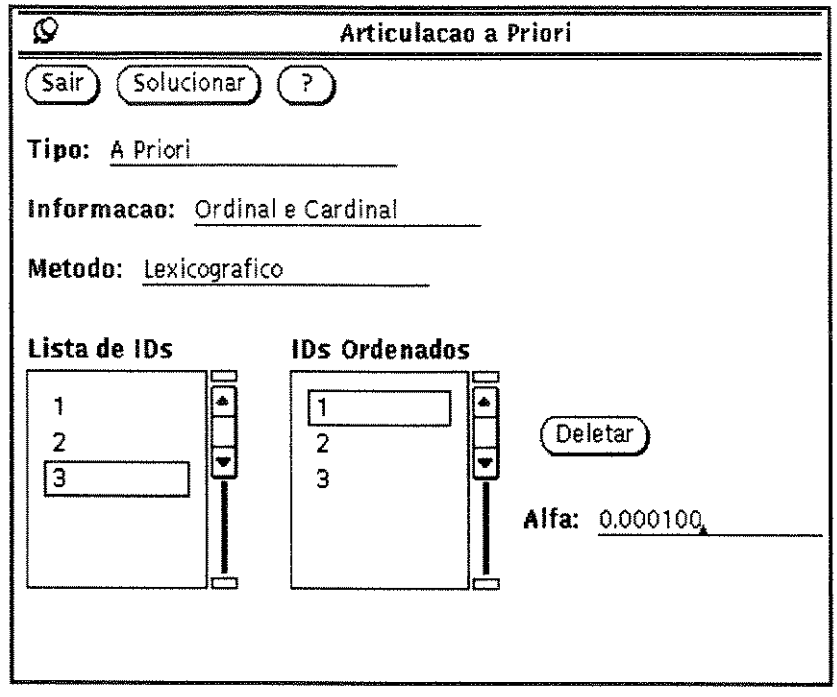


Figura 4.19 - Método Lexicográfico

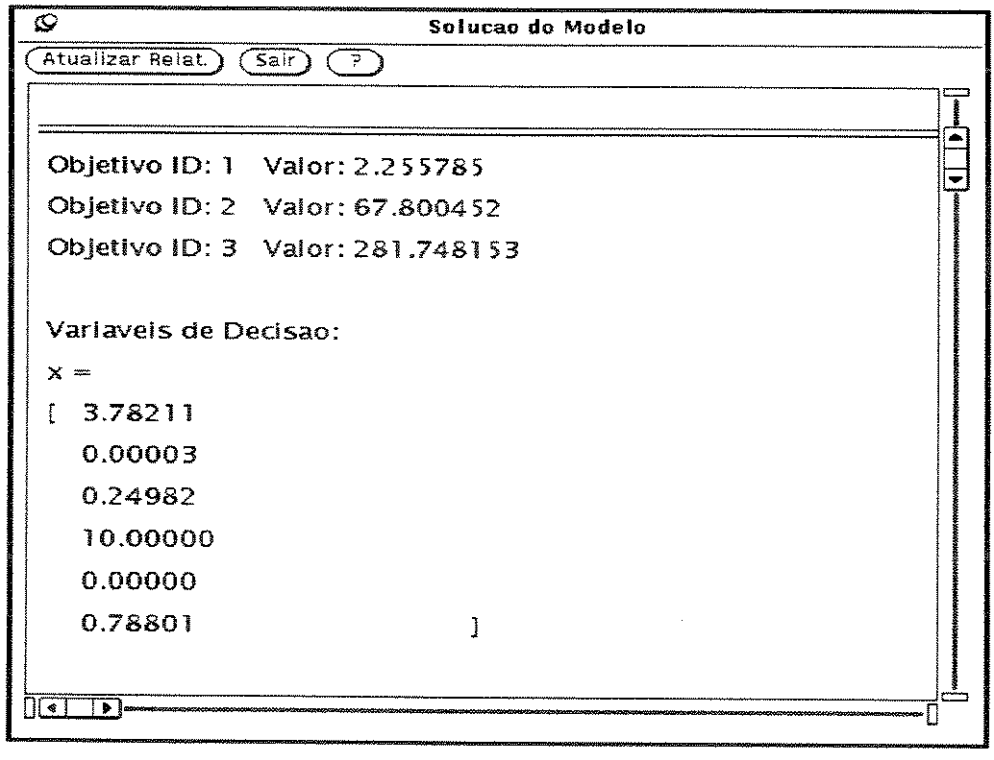


Figura 4.20 - Solução para o Método Lexicográfico

Figura 4.21 - Método Paramétrico

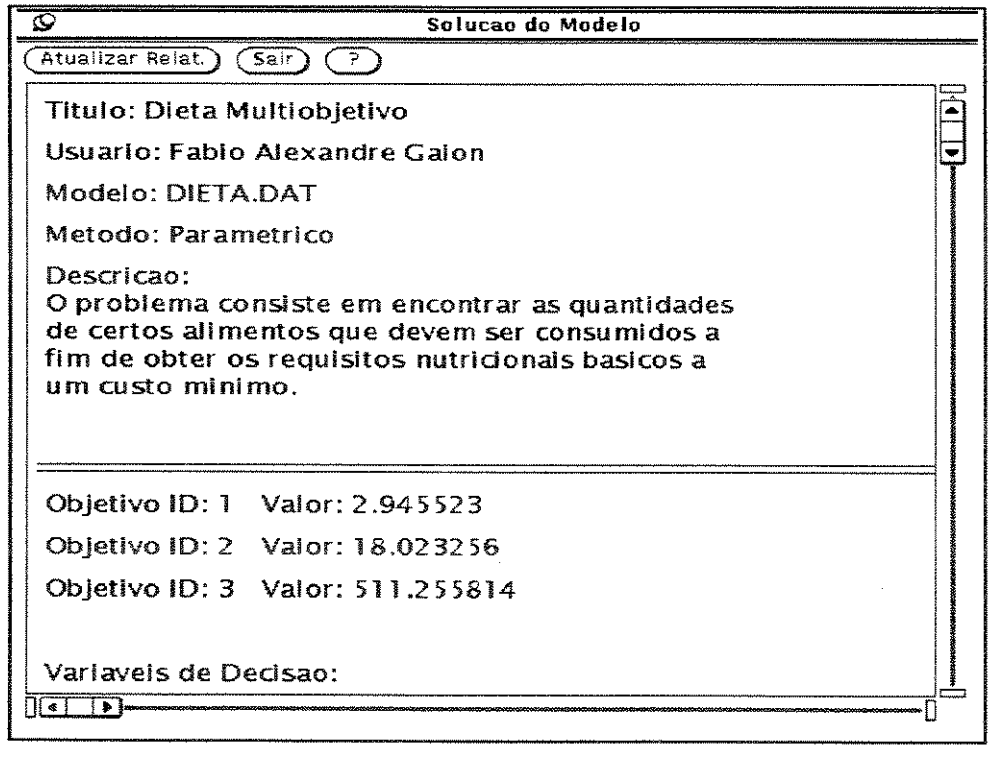


Figura 4.22 - Solução para o Método Paramétrico

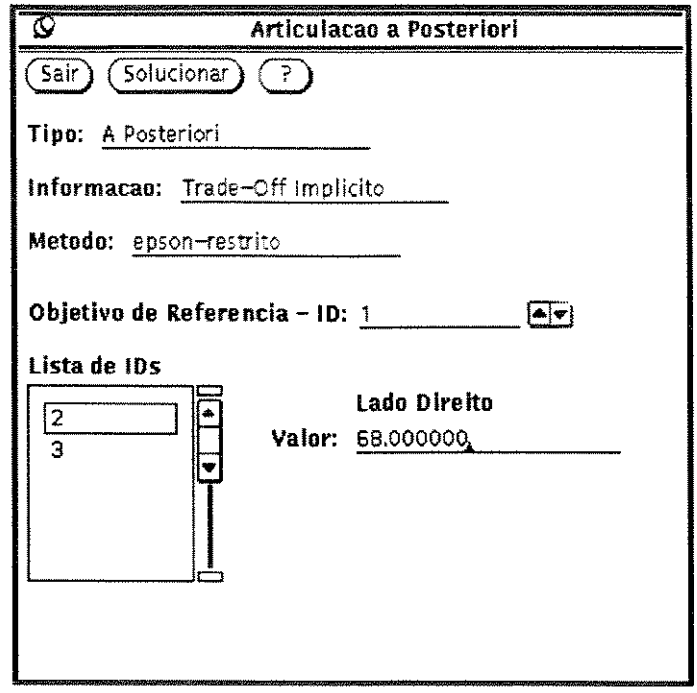


Figura 4.23 - Método ε-Restrito

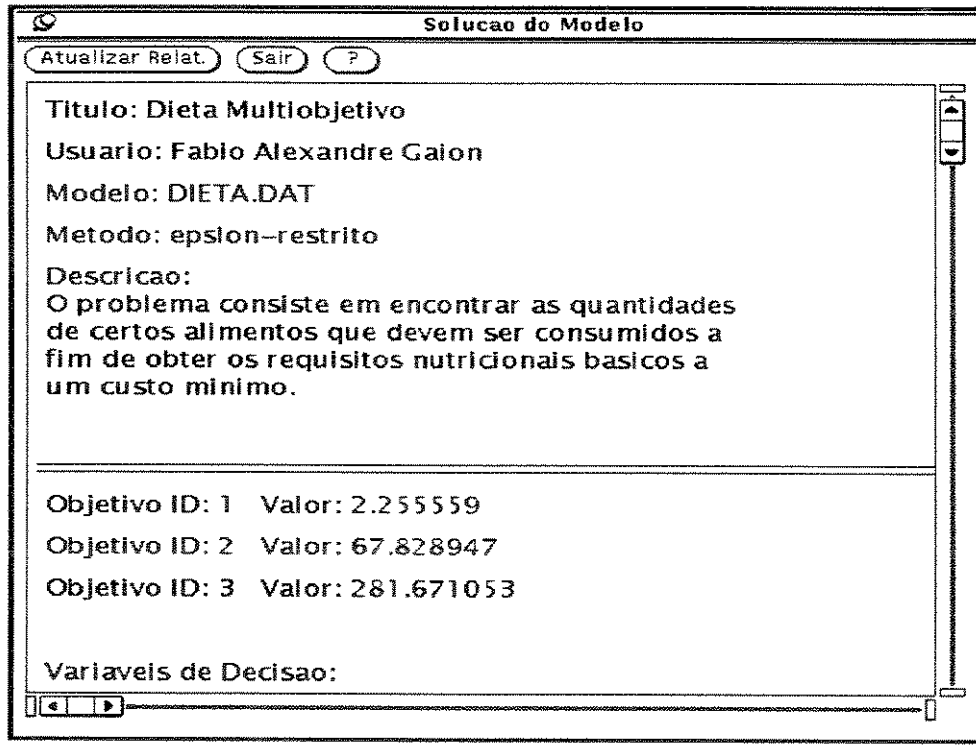


Figura 4.24 -Sol. p/ o Método ε-Restrito

Geoffrion, Dyer e Feinberg

Este método (Figuras 4.25 à 4.30) é selecionado através da opção **Geoffrion** do menu **Estrutura**. A tela apresentada para este método exibe a cada iteração o ID do objetivo de referência, uma lista dos objetivos que não são de referência, os valores de trade-off (Δ's), o passo ótimo, uma estimativa da variação da função utilidade do decisos (α) e o número de discretizações. Neste método devemos inicializar os valores de Δ e depois clicar em solucionar. Com isso será exibida uma tabela com os valores de cada função objetivo. Esta tabela terá tantas colunas quantos forem os objetivos e as linhas são dadas pelo campo **Número de discretizações:**.

De posse desta tabela escolhemos o passo ótimo e clicamos novamente em **Solucionar** .Com isso a solução corrente, exibida na tela do método, será atualizada. Se esta solução for ótima podemos clicar na opção **Ótimo** para encerrarmos o método, caso contrário devemos setar novamente os valores de Δ e efetuar uma nova iteração. Uma outra maneira de encerrar o método com uma solução ótima em mãos é escolhendo o passo ótimo como sendo 0.

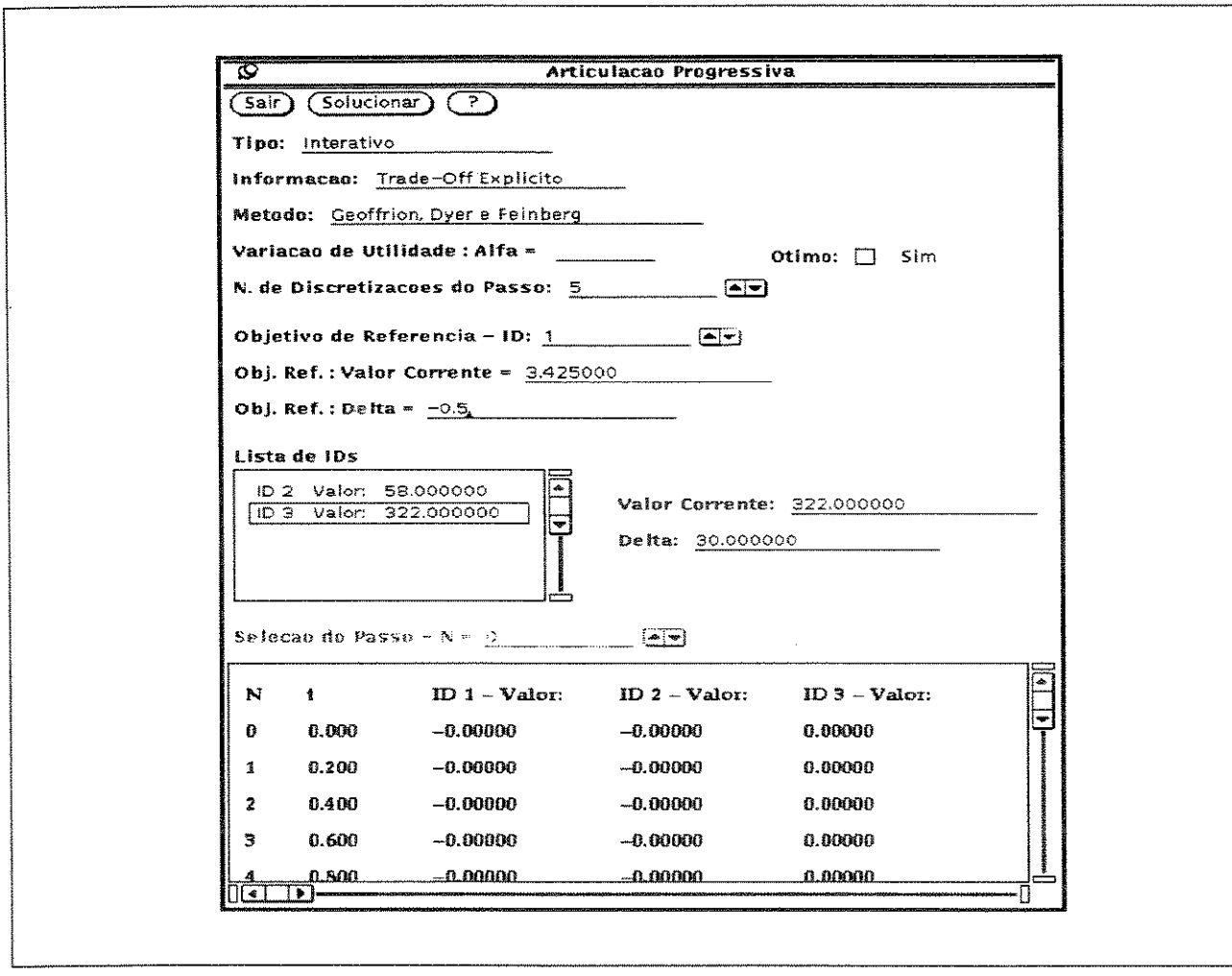


Figura 4.25 -Método de Geoffrion, Dyer e Feinberg

Passo 1: Tomando $x^0 = (3.0, 0.5, 0.15, 5.0, 5.0, 3.0)$ como solução inicial para o problema, temos que as funções objetivos assumem os valores $f^0 = (3.425, 58.0, 322.0)$. Assumiremos também que $\alpha < 0.15$ representa uma solução ótima para o problema da dieta.

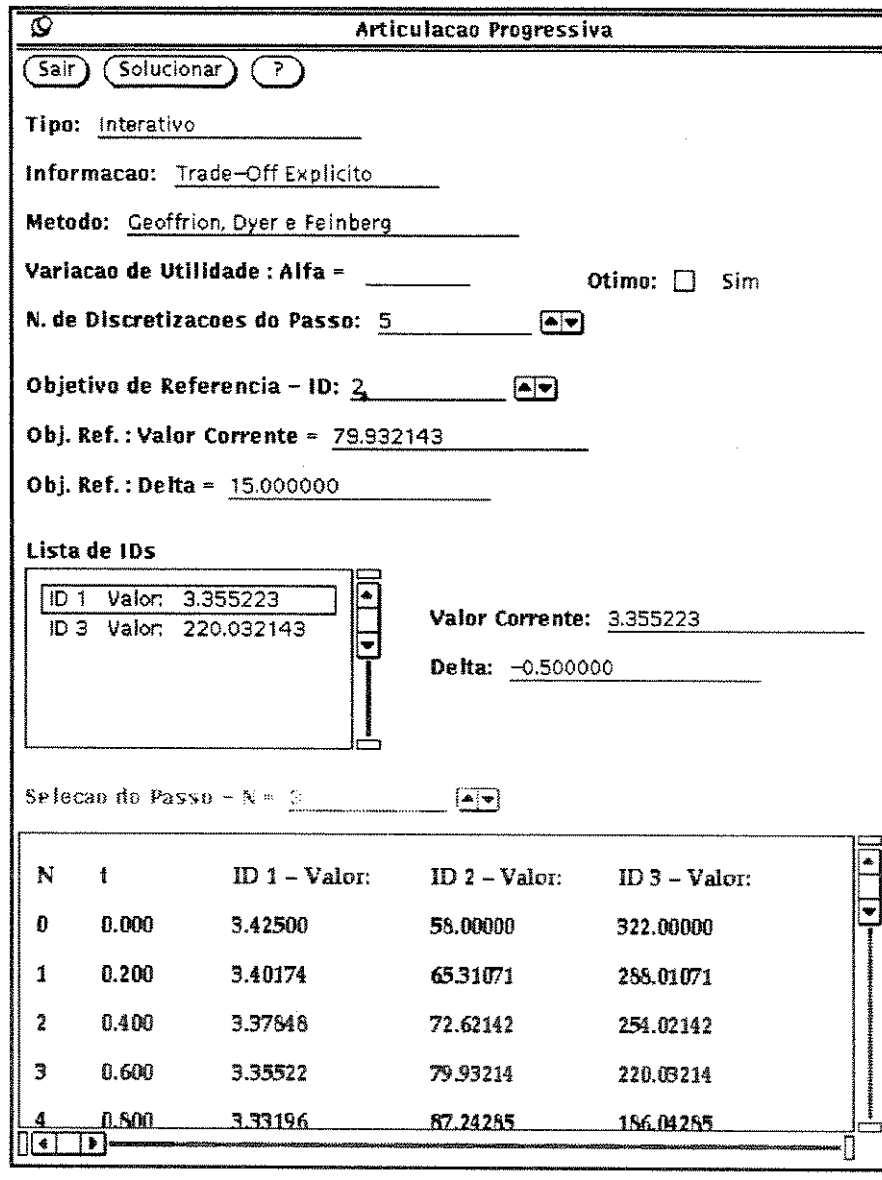


Figura 4.26 -Método de Geoffrion, Dyer e Feinberg

Passo 2: Como observamos na Figura 4.25, o objetivo de referência escolhido foi o de ID 1 com $\Delta = -0.5$ e os trade-offs para os objetivos de ID 2 e ID 3 são respectivamente 20 e 30. Resolvendo esta iteração, obtemos a Figura 4.26 que mostra os novos valores para a funções objetivos segundo uma discretização com passo $\mathbf{t} = 0.2$. Baseado na tabela da Figura 4.26 escolhemos $\mathbf{t} = 0.6$ ($\mathbf{N} = 3$) como passo ótimo.Essa escolha de passo gera uma nova solução f^1 = (3.355 , 79.932 , 220.032). Neste ponto escolhemos o objetivo de ID 2 como objetivo de refência com $\Delta = 15$. Para os objetivos de ID 1 e ID 3 fazemos Δ igual a -0.5 e 20 respectivamente.Uma vez setados os valores dos trade-offs devemos clicar em Solucionar para inicializar uma nova iteração.

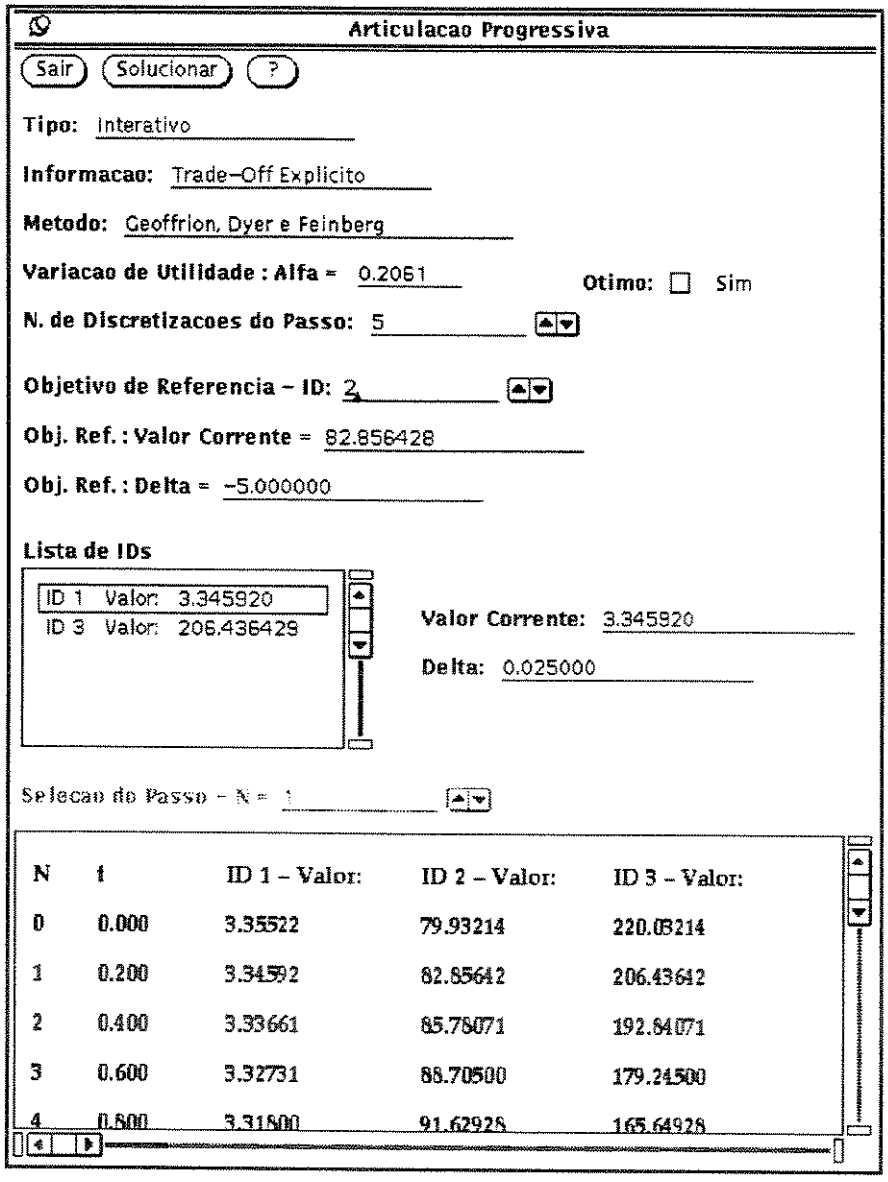


Figura 4.27 -Método de Geoffrion, Dyer e Feinberg

Passo 3: Nesta iteração temos uma nova tabela mostrando os valores das funções objetivos segundo uma discretização com passo t = 0.2.Baseado nesta tabela, escolhemos t = 0.2 (N = 1) como passo ótimo. Esta escolha gera uma nova solução f^2 = (3.345 , 82.856 , 206.436). A partir desta solução escolhemos o objetivo de ID 2 como referência e fazemos seu Δ = -5. Os trade-offs para os objetivos de ID 1 e ID 3 são respectivamente 0.025 e 10. Com os valores dos trade-offs inicializados procedemos a uma nova iteração.

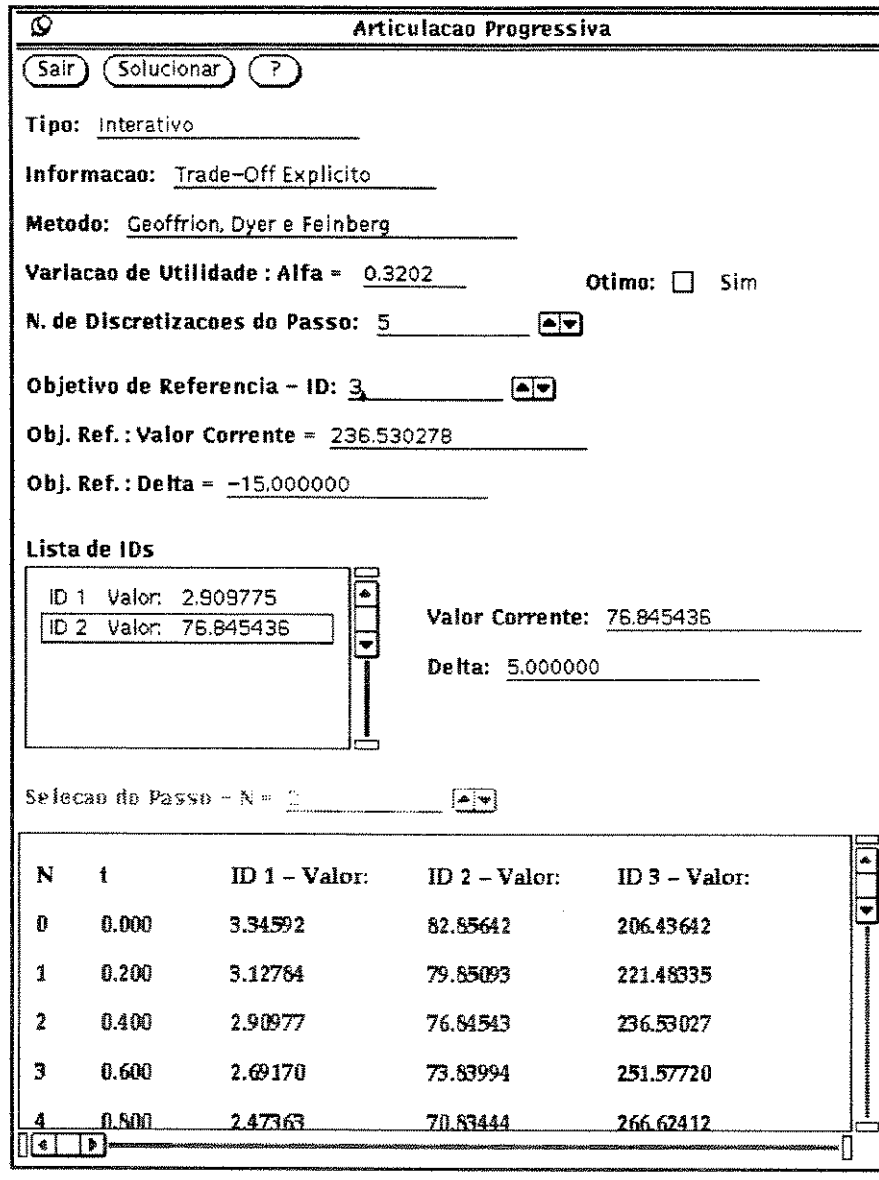


Figura 4.28 -Método de Geoffrion, Dyer e Feinberg

Passo 4: Nesta iteração temos uma nova tabela mostrando os valores das funções objetivos segundo uma discretização também com passo $t = 0.2$.Baseado nesta tabela, escolhemos $t = 0.4$ ($N = 2$) como passo ótimo. Esta escolha gera uma nova solução $f^3 = (2.909 , 76.845 , 236.530)$. A partir desta solução escolhemos o objetivo de ID 3 como referência e fazemos seu $\Delta = -15$. Os trade-offs para os objetivos de ID 1 e ID 3 são respectivamente 0.3 e 5. Com os valores dos trade-offs inicializados procedemos a uma nova iteração.

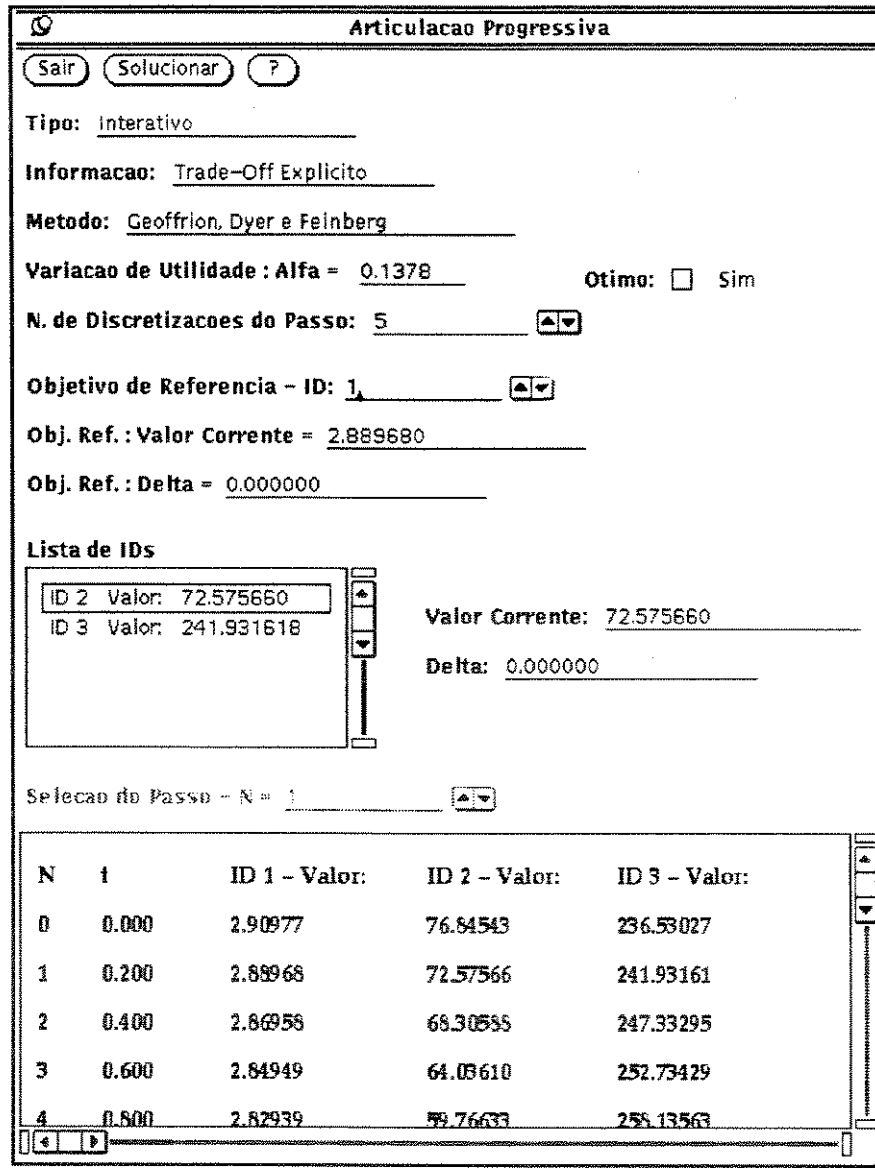


Figura 4.29 -Método de Geoffrion, Dyer e Feinberg

Passo 4: Nesta iteração temos $\alpha = 0.1378$, isto é, conseguimos obter uma estimativa da variação percentual da função utilidade que está dentro dos parâmetros a serem atingidos. Os valores das funções objetivos nesta iteração são $f^4 = (2.889 , 72.575 , 241.931)$. Para obtermos a solução completa do problema devemos clicar na opção **Ótimo**. Ao escolhermos esta opção, será exibida uma nova tela com os resultados obtidos (Figura 4.30).

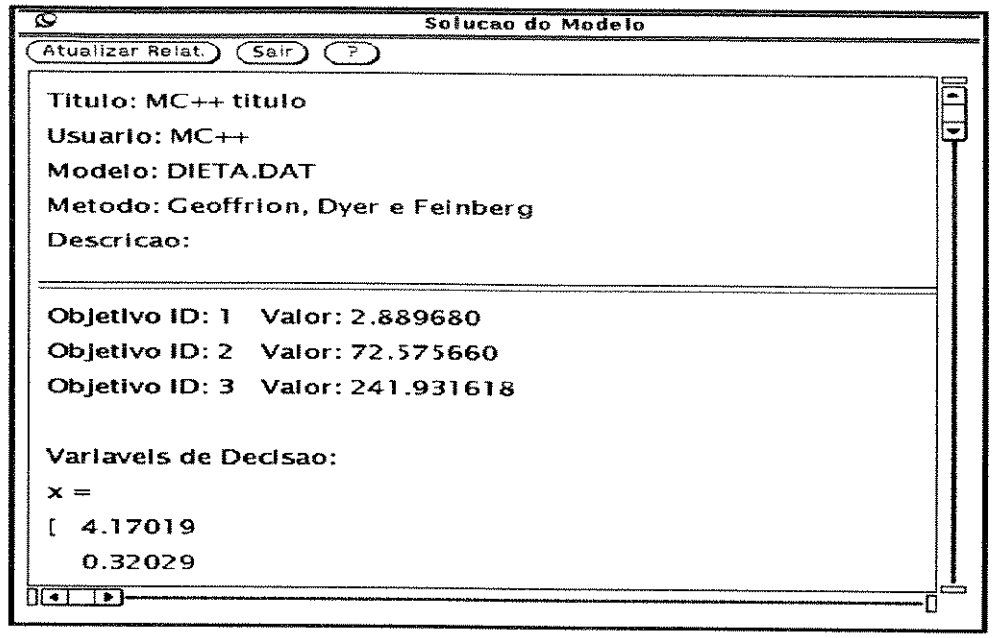


Figura 4.30 -Solução para o Método Geoffrion, Dyer e Feinberg

STEM

Este método (Figuras 4.31 e 4.32) é selecionado através da opção **STEM** do menu **Estrutura**. A tela apresentada para este método possui uma lista com os IDs, o valor corrente de cada função objetivo e dois botões denominados **Satisfatório** e **Insatisfatório**. A medida que vamos definindo os objetivos como satisfatórios ou insatisfatórios duas outras listas vão sendo elaboradas: uma lista com o conjunto de objetivos satisfatórios e outra lista com o conjunto de objetivos insatisfatórios. Após definir estes dois conjuntos, devemos setar o valores de Δ (relaxação dos objetivos satisfatórios). Estes valores indicam perdas nos objetivos satisfatórios em prol dos objetivos insatisfatórios. Com estes parâmetros setados, devemos clicar em solucionar para efetuarmos a solução do problema. A solução ótima é obtida quando a lista de objetivos insatisfatórios for vazia.

4.4 - Relatório do Problema da Dieta Multiobjetivo

O relatório é obtido através da opção **Relatório** do menu principal. Ao se escolher esta opção, um relatório contendo o descrição do modelo é apresentado. Este relatório é dividido nas seguintes partes: *Página de Apresentação, Componentes do Modelo, Modelo Simbólico, Modelo Matricial* e *Soluções do Modelo*.

Página de Apresentação

Esta página contém o nome do modelo, o usuário, o arquivo onde o modelo está armazenado e uma descrição do mesmo (Figura 4.33).

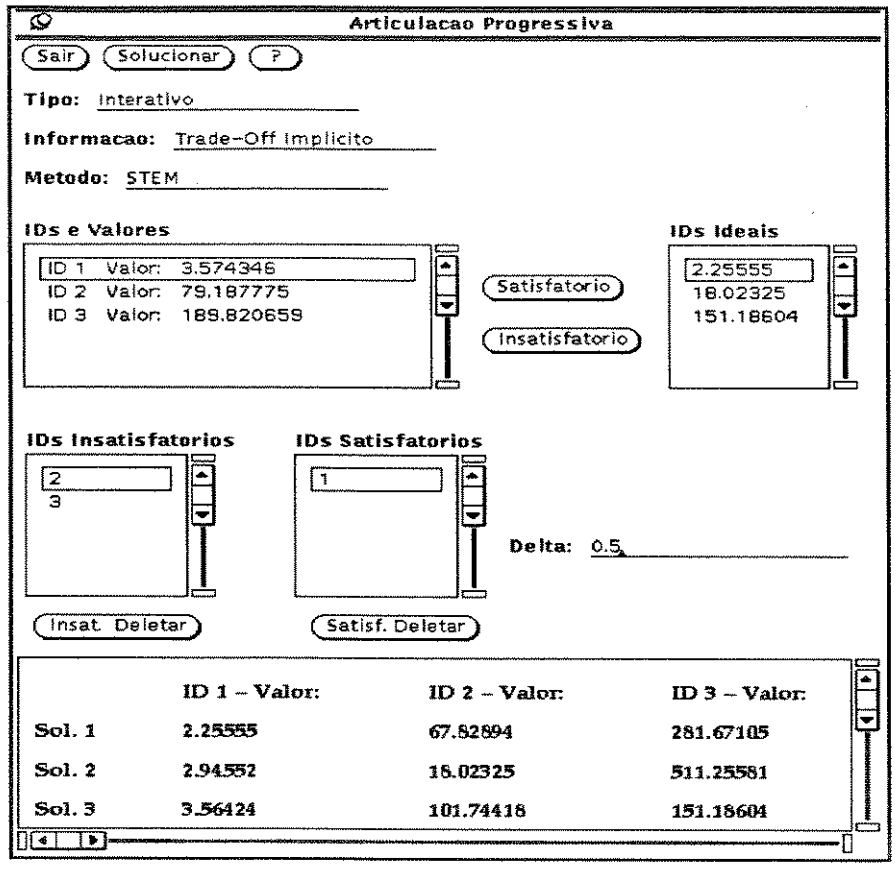


Figura 4.31 - Método STEM (Step Method)

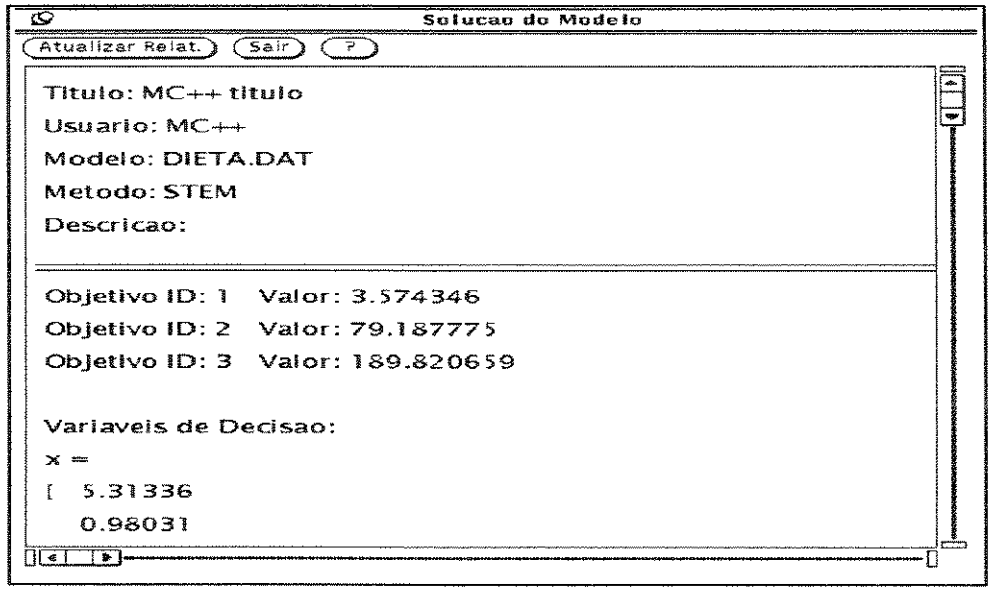


Figura 4.32 - Solução para o Método STEM

Componentes do Modelo

Esta parte do relatório é formada por uma tabela contendo as primitivas que compõem o modelo. Esta tabela possui uma relação dos escopos, índices, variáveis e parâmetros bem como suas variações e documentação (Figura 4.34).

Modelo Simbólico

Esta parte do relatório mostra a representação simbólica do modelo, ou seja, a representação utilizando símbolos matemáticos (como a somatória), as variáveis e os parâmetros como estes foram definidos. Esta representação é a mesma exibida na tela principal do sistema (Figura 4.35).

Modelo Matricial

Esta parte do relatório mostra a representação matricial do modelo, ou seja, os objetivos e restrições são exibidos como vetores e matrizes (Figura 4.36).

Soluções do Modelo

Neste ponto são exibidas as soluções do modelo efetuadas pelos métodos escolhidos (Figuras 4.37 à 4.39). É apresentado o valor das funções objetivos e o valor das variáveis de decisão, bem como outras informações necessárias a interpretação do resultado (Ex: ponderação do objetivo, valor de ε, limites, etc...)

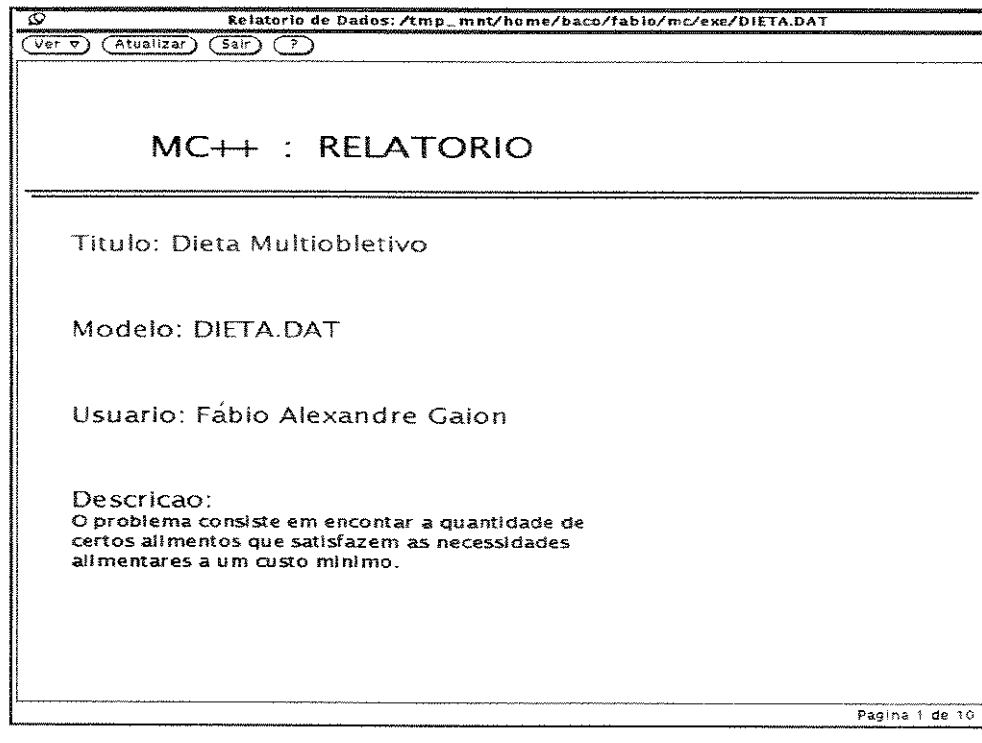


Figura 4.33 - Tela de Apresentação

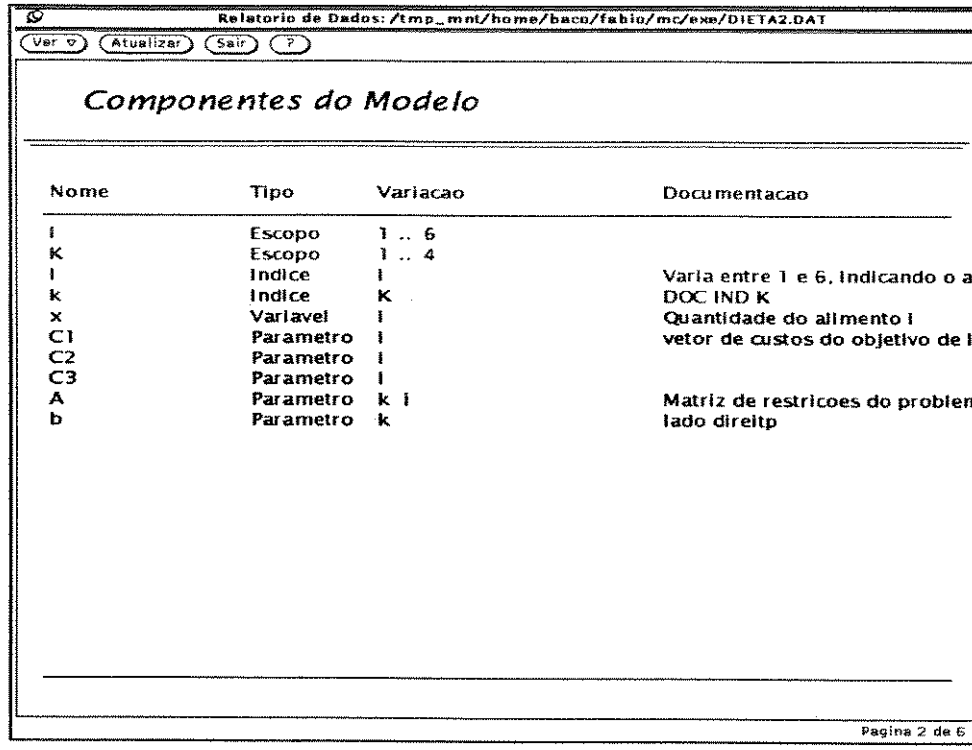


Figura 4.34 - Componentes do Modelo

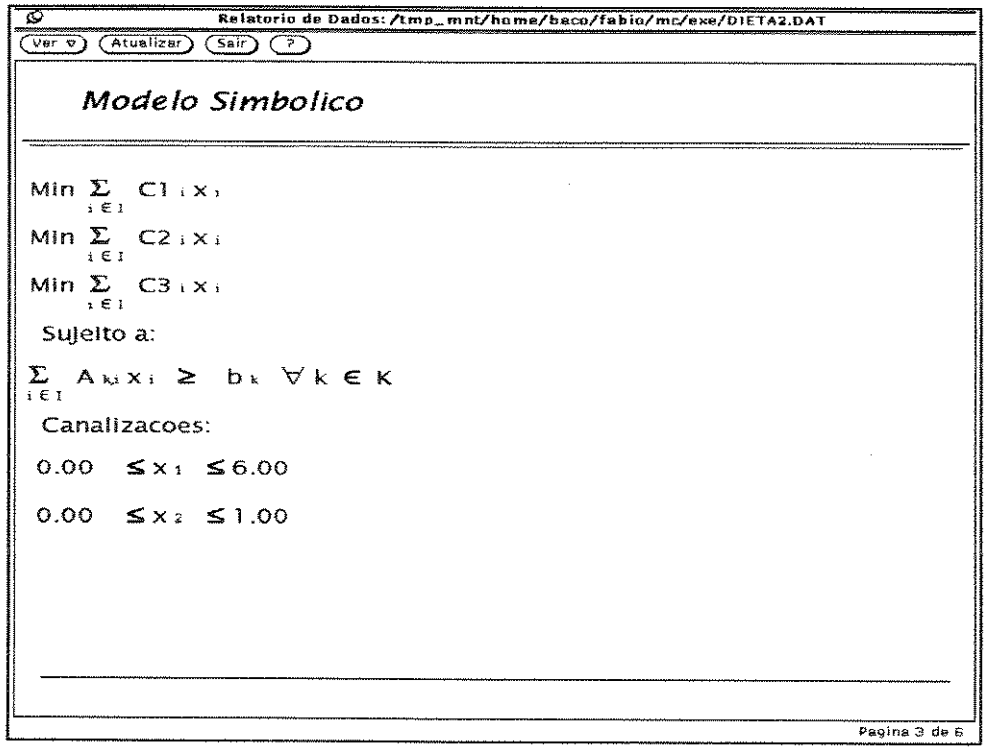


Figura 4.35 - Modelo Simbólico

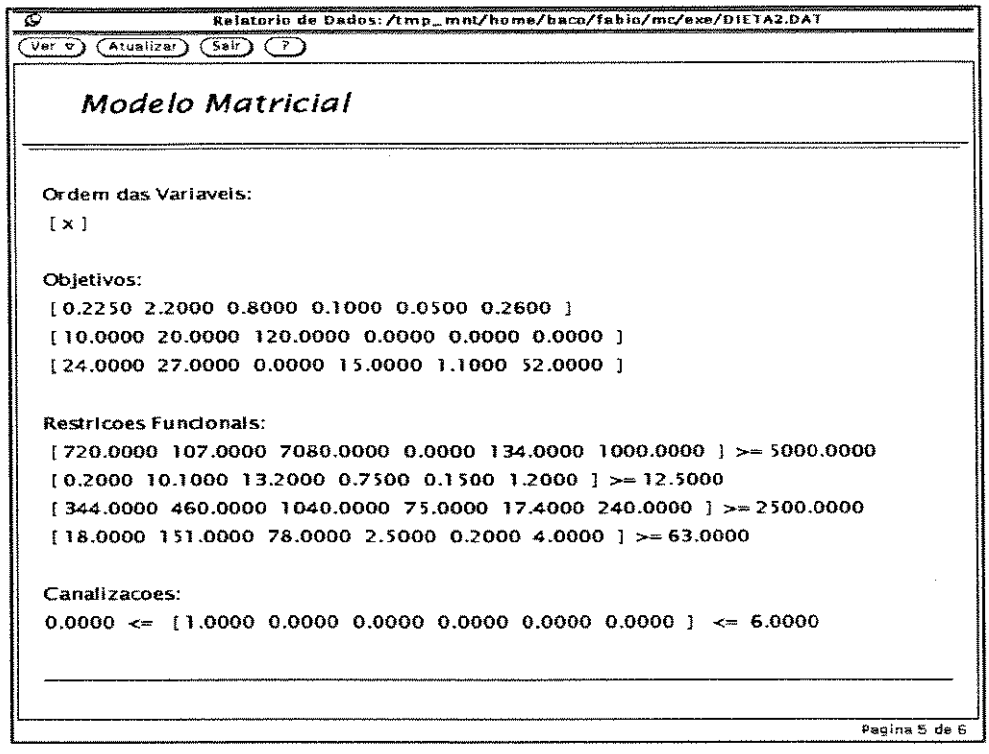


Figura 4.36 - Modelo Matricial

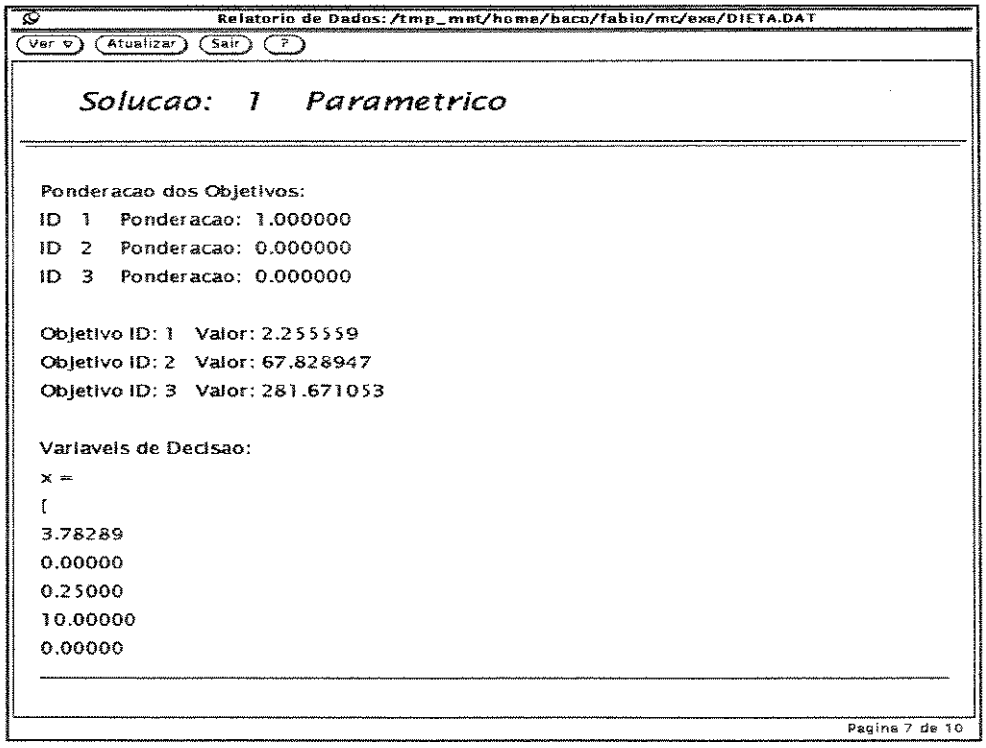


Figura 4.37 - Solução 1

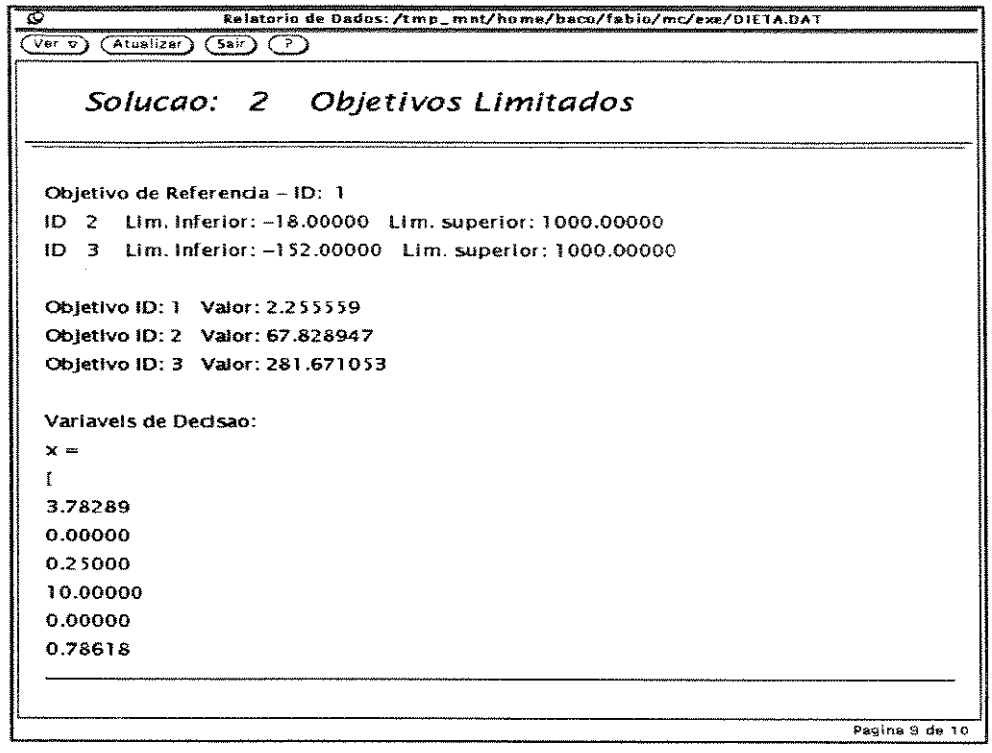


Figura 4.38 - Solução 2

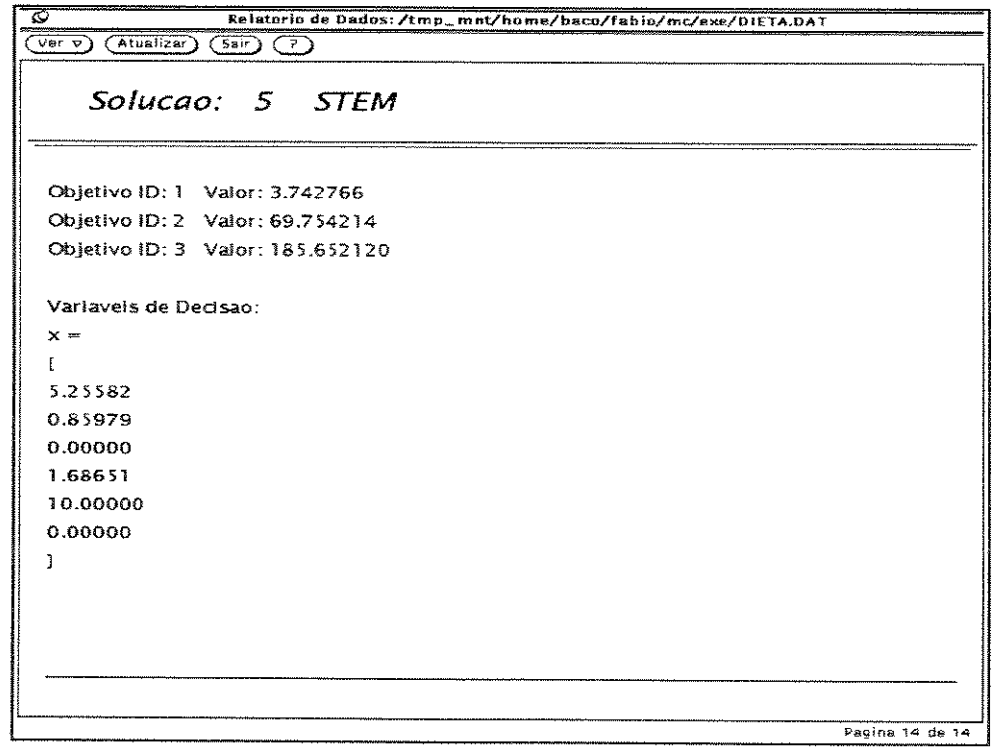


Figura 4.39 - Solução 3

4.5 - Conclusão

Este capítulo ilustra o ambiente matemático proporcionado pelo MC++ através da modelagem do problema da dieta multiobjetivo. Todo o processo de definição de primitivas, objetivos e restrições é descrito exibindo as diferentes maneiras de se modelar um problema. Após modelado, o problema foi solucionado por todos os métodos implementados e suas soluções foram apresentadas. O método interativo Geoffrion, Dyer e Feinberg foi executado passo a passo com o intuito de exibir as interações do usuário com o MC++.

Conclusão Geral

Neste trabalho, foi implementado um Sistema de Suporte a Decisão Baseado em Programação Multiobjetivo, o MC++. O software visa a modelagem, resolução e análise de resultados de problemas de otimização com enfoque multiobjetivo e fornece várias facilidades e recursos do ponto de vista computacional.

Foi dado especial enfoque no desenvolvimento da interface com o usuário que foi elaborada a partir da utilização de recursos gráficos. Esta importância dada a interface teve como objetivo fornecer um ambiente amigável de maneira a facilitar a modelagem e interação com os problemas de otimização.

A concepção do MC++ seguiu duas linha básicas: o estudo de problemas e métodos de programação multiobjetivo e o estudo de softwares de programação matemática e sistemas de suporte a decisão.

O estudo das formulações de Problemas de Programação Multiobjetivo tanto no espaço das variáveis como no espaço dos critérios bem como o estudo dos métodos de programação multiobjetivo (classificados segundo estruturas de preferência) serviram de base para a implemetação dos métodos suportados por esta versão do MC++.

Na implementação do MC++ aproveitou-se várias idéias e técnicas descritas em sistemas de suporte a decisão e softwares de programação matemática já existentes. Porém é importante salientar que nenhum dos sistemas estudados proporcionavam um enfoque multiobjetivo no tratamento dos problemas de otimização.

Também é importante observar que o estudo destes sistemas levou-nos a concepção de um sistema que permitisse expansões futuras através de uma estratégia de implementação modularizada.

A implementação do MC++ foi fruto de muito trabalho tanto teórico como computacional. Todavia este esforço foi de grande valia, pois as dificuldades e situações enfrentadas no decorrer do seu desenvolvimento proporcionaram um amadurecimento e enriquecimento pessoal que, com certeza, será utilizado no aperfeiçoamento do MC++ e na elaboração de novos sistemas.

Com esta versão inicial desejávamos fechar o ciclo definição, modelagem, resolução e apresentação de resultados a fim de termos um sistema que pudesse ser analisado, testado e criticado. Baseado nestas análises e críticas poderemos gerar novas versões cada vez mais completas e robustas.

Como todo software, o MC++ está sempre em evolução, sendo que idéias para a sua melhoria não faltam.

Além das caraterísticas ainda não implementadas (novos métodos, *Help*, tutorial, melhoria no analisador semântico, etc..) que serão incluídas num futuro próximo, temos novas idéias que tornariam o MC++ mais genérico e útil.

Até o momento as possíveis alterações propostas para o MC++ são:

- Desenvolvimento de uma linguagem matemática mais genérica que permita a elaboração de expressões mais complexas. Esta linguagem possibilitaria a definição de expressões algébricas (como o Matlab) e também expressões de otimização.
- Tratamento de algumas classes de problemas não lineares.
- Introdução de mais alguns níveis de abstração como por exemplo a definição de um modelo através de sua descrição verbal ou através de um desenho formado por *ícones* e *relacionamentos* entre os *ícones*.
- Utilização de técnicas de Inteligência Artificial afim de desenvolver um sistema especialista baseado em conhecimento.
- Unir o MC++ com um software para manipulações algébricas (similar ao Matlab) a fim de proporcionar um ambiente matemático mais completo.
- Incluir técnicas de conjuntos nebulosos. Estas técnicas podem ser utilizadas tanto na implementação de novos métodos multiobjetivos como para o controle lógico do MC++.
- Proporcionar uma comunicação do MC++ com outros softwares externos, sobretudo com pacotes utilizados para otimização.

É interessante colocar que já existe um software para manipulações algébricas em desenvolvimento no Departamento de Telemática chamado **Convex** (Gapski e Geromel, 1992). O mais importante a ser observado é que existe uma vontade comum em unir os softwares MC++ e Convex (além de promover várias melhorias) a fim de propoporcionar um sistema mais abrangente e robusto.

Estas idéias preliminares bem como muitas outras que surgirão no decorrer da implementação de novas versões farão parte de um projeto de pesquisa para a elaboração de um ambiente matemático mais completo.

Bibliografia

APEX II Reference Manual. Control Data Corporation, 1974. Publication nº 5915810.

AT&T C++ Language Reference Manual, AT&T, 1989.

Astrahan, M.M. e Chamberlin, D.D. Implementation of a Structured English Query Language, *Communications of de ACM* 18,10 , outubro, 1985, 580-588.

Bazaraa, M.S. e C.M. Shetty. *Nonlinear Programming Theory and Algorithms*, John Wiley & Sons, 1979.

Bazaraa, M.S. e J.J. Jarvis. *Linear Programming Theory and Network Flows*, John Wiley & Sons, 1977.

Collins, W.J. *Programação Estruturada com Estudos de Casos em Pascal*, McGraw Hill, 1988.

Dolk, D.R. A Generalized Model Management System for Mathematical Programming, *ACM Trans. on Mathematical Software*, Vol 12, nº 2, 1986.

Dyer, J.C. (1972), Interactive Goal Programming, *Management Science*, Vol. 19, nº 1.

Dyer, J.C. (1974), The Effects of Errors in the Estimation of the Gradient of the Frank-Wolfe Algorithm, with Application for Interactive Programming, *Operations Research*, Vol. 22.

Ellison, E.F.D. e G. Mitra. UIMP: User Interface For Mathematical Programming, *ACM Trans. on Mathematical Software*, Vol 8, nº 3, setembro, 1982, 229-255.

Ferreira, P.A.V. Otimização Multiobjetivo: Desenvolvimento de um Método Interativo Baseado em Projeção, *Tese de Doutorado*, FEE/UNICAMP, Novembro, 1986.

Ferreira, P.A.V. e J.C. Geromel. An Interactive Projection Method for Multicriterio Optimization Problem, *IEEE Trans. on Systems Man and Cibernetics*, Vol SMC-20, nº 3, 1990.

Ferreira, P.A.V. e Menotti A.S. Machado. A Relaxation Procedure for Multiobjetive Decision Making, *29º Conference on Decision and Control*, Honolulu, 1990.

Fourer, R.D.M. Gay e B.W. Kernighan. *AMPL: A Mathematical Programming Language*. AT&T Bell'Laboratories, Murray Hill, N.J., 1987.

Gapski, P.B. e J.C. Geromel. *Relatório de Iniciação Científica submetido a FAPESP*, Proc. 91/2612-2, 1992.

Geoffrion, A.M. (1967), Solving Bicriteria Mathematical Programs, *Operations Research*, Vol. 15, n º 1.

Geoffrion, A.M. (1968), Proper Efficiency and the Theory of Vector Optimisation, *Journal of Mathematical Analysis and Applications*, Vol. 22, n º 3.

Geoffrion, A.M. (1971), Duality in Nonlinear Programming: A Simplified Aplications-Oriented Development, *SIAM Review*, vol. 13

Geoffrion, A.M. (1972), Generalized Benders Decomposition, *Journal of Optimisation Theory and Applications*, vol. 10, nº 4.

Geoffrion, A.M., J.S. Dyer e A. Feinberg (1972), An Interactive Approach for Multi-Criterion Optimisation, with an Application to the Operation of an Academic Department, *Management Science*, Vol. 19, n º 4 (Part I).

Geromel, J.C. e P.A.V. Ferreira. An Upeer Bound on Property Efficient Solution in Multiobjective Optimization, *Operations Research Letters*, Vol 10, nº 1, 1991.

Gill, P.E., Murray, W. e M.H. Wright. *Practical Optimization*, Academic Press, 1981.

Greegan, J.B. Jr. *Dataform: A Model Management System.* Ketron, Inc., Arlington, Virginia, Novembro, 1985.

Greenberg harvey J. A Functional description of ANALIZE: A Computer-Assisted Analysis System for Linear Programming Models. *ACM Transactions on mathematical Software* 9,1 , março, 1983, 18-56.

Haimes, Y.Y. e W.A. Hall (1974), Multiobjectives in Water Resources Analyses: The Surrogate Worth Trade-off Method, *Water Resources Research*, vol 10.

Haimes, Y.Y., Hall, W.A. e H.T. Freedman (1975), *Multiobjective Optimization in Water Resources Systems*, Elsevier Scientific Publishing Co., Amsterdam.

Horowitz, E. e S. Sahni. *Fundamentos de Estruturas de Dados*, Ed. Campus, 1987.

Hwang, C.L. e A.S. Md Masud. *Multiple Objective Decision-Making Methods and Applications*, Springer - Verlag, 1979.

IBM Mathematical Programming Language Extended S70 (MSPX/S70), Program Reference Manual, SH19-1095. IBM Corporation, Paris, França, 1975.

Johnson, S.C. Yacc: Yet Another Compiler Compiler, *Computing Science Technical Report* nº 32, 1975, Bell Labotories, Murray Hill, NJ.

Kernighan B.W. e D.M. Ritchie. *The C Programming Language*, Prentice-Hall, N.J. , 1978.

Knuth, Donald E. *The Art of Computer Programming*, Addison Wesley, 1969.

Knuth, Donald E. *The TEXbook*, Addison Wesley, 1986.

Kurator, W.G. e R.P. O'Neill. PERUSE: AN Interactive System for Mathematical Programs. *ACM Trans. on Mathematical Software*, Vol 6, nº 4, dezembro, 1980, 489-509.

Lasdon, L.S. (1970), *Optimization Theory for Large Systems*, MacMillan, New York.

Lesk, M.E. e E. Schmidt. *Lex - A Lexical Analyser Generator*, Bell Laboratories, Murray Hill, New Jersey, 1974.

Lippman, Stanley. *C++ Primer*, Addison Wesley, 1992.

Lucas, G.A.A.G. *Principles of Expert Systems*, Addison Wesley, 1991.

Luenberger, D., *Linear and Nonlinear Programming*, Addison Wesley, 1984.

Ma, Pai-chun. An Intelligent Approach Towards Formulating Linear Programs. *Ph.D. Th. Stern School of Business*, New York University, 1986.

Machado, M.E.S. Projeção em Programação Multiobjetivo: Algoritmos e Experiências Numéricas. *Tese de Mestrado*, FEE UNICAMP, 1991.

MATLAB User's Guide, The Math Works, Inc. , 1991.

Meeraus, A. *General Algebraic Modeling Systems (GAMS): User's Guide, Version 1.0*. Development research center, World Bank, 1984.

Murphy, F.H. e E. A. Stohr. An Intelligent System for Formulating Linear Programs. *Decision Support Systems* 2,1, janeiro-fevereiro, 1986.

Nilsson, N.J. *Principles of Artificial Inteligence*, Morgan Kanfruann, 1980.

OMNI Linear Programming System: User manual and Operating manual. Haverly Systems Inc. Denville, N.J., 1977.

Palmer, K.H., N.K. Boudwin, H.A. Patton, A.J. Rowland, J.D. Sammes, e D.M. Smith. *A Model-Management Framework for Mathematical Programming*. John Wiley & Sons, new York, 1984.

Routo, T. *Desenvolvimento de Algoritmos e Estrutura de Dados*, Makron Books, 1991.

Sagie, Ilke. Computer-Aided Modeling and Planning (CAMP), *ACM Trans. on Mathematical Software*, Vol 12, nº 3, setembro, 1986, 225-248.

Schrage, Linus. *Linear, Integer, and Quadratic Programming with LINDO*. The Scientific Press, Palo Alto, 1987.

Stroustrup, B. *The C++ Programming Language*, Prentice Hall, 1992.

Sun C++ Programmer's Guide, Sun Microsystems, Inc , 1989.

Welch, james S. Jr. PAM - A Practitioner's Approach to Modeling. *Management Science* 33.5, maio, 1987, 610-625.

Wirth, N. *Algorithms and Data Structures*, Prentice Hall, 1976.

Xlib Programming Manual, O'Reilly & Associates, 1990.

Xview Programming Manual, An OPEN LOOK Toolkit for X11, O'Reilly & Associates, 1990.

Apêndice A
Classes ou Objetos

Lista Genérica

Esta classe é a base de todas as outras classes tipo lista, sendo responsável pela inserção, remoção, alteração, busca e outras tarefas realizadas nas estruturas de dados. Ela implementa uma lista de apontadores que apontam para qualquer tipo de elemento, assim sendo, podemos montar listas com todos os tipos de dados e estruturas.

Lista genérica

```
const int NVEC = 128;
typedef void* ent;
typedef int (*CMPENT)(ent, ent); //apontador para função de comparação entre elementos
typedef void (*F_ERROR)(int,char *); //apontador para função de erros
extern F_ERROR vlist_handler;
extern F_ERROR set_vlist_handler(F_ERROR);
class vlist
{
private:
friend class vlist_iterator;
  int tam;//tamanho da lista
  ent *e;//apontador para vetor de ent
  int next;//next é o próximo índice @PROG = livre
public:
  int insert(ent a);//adiciona na cabeça da lista
  int append(ent a);//adiciona no final da lista
  ent get();//retorna e remove a cabeça da lista
  ent get(int x);//retorna o x-ésimo elemento sem remove-lo
  ent getscroll();//retina a cabeça da lista e faz o scroll
  void clear();//limpa a lista
  int ordena(CMPENT);//ordena segundo a fução CMPENT
  int insordem(ent, CMPENT);//insere na ordem segundo a função CMPENT
  ent recupera(ent, CMPENT);//recupera da lista sem remover segundo CMPENT
  ent remove(ent);//remove ent da lista
  vlist();
  vlist(ent);
  ~vlist()    { clear(); }
```

```
 int vazia()  { return (!next); }
 int tamanho() {return next; }
};
class vlist_iterator
{
private:
 vlist *pv;
 int corrente;
public:
 vlist_iterator(vlist& v) { pv = &v; corrente =0; }
 ent operator()()
 {
  ent ret = pv-e[corrente] ? (pv-e[corrente++]) : 0;
  return ret;
 }
};
```

Escopo e Lista de Escopos

A classe **Escopo** é responsável pela representação computacional de um escopo de índice. Nesta classe armazenamos o tipo, variação e valores dos escopos, além de possuirmos funções para o tratamento dos mesmos. A classe **Lista de Escopos** é baseada na classe **Lista Genérica** e implementa uma lista de escopos.

Escopo e Lista de Escopos

```
extern F_ERROR escopo_handler;
extern F_ERROR set_escopo_handler(F_ERROR);
class escopo: public string
{
private:
  string nome;//nome do escopo
  int min;//valor mínimo do escopo (tipo numérico)
  int max;//valor máximo do escopo (tipo numérico)
  sstringlist *lstr;//lista de strings (tipo caracter)
  string doc;//docimentação
public:
  //Construtores
  escopo():nome(),min(0),max(0),lstr(0),doc() {}
  escopo(char *s): nome(s),min(0),max(0),lstr(0),doc() {}
  escopo(char *,int,int,sstringlist*,char *);
  escopo(escopo &);
  //Acessar os dados da classe
  char* enome() { return nome.cadeia(); }
```

```
int   emin() { return min; }
int   emax() { return max; }
sstringlist* estrlist() { return lstr; }
char* edoc()  { return doc.cadeia(); }
//Setar os dados
void enome(char *s) { nome = s; }
void emin(int vmin) { min = vmin; }
void emax(int vmax) { max = vmax; }
void estrlist(sstringlist *ss) { lstr = ss; }
void edoc(char *s)  { doc = s; }

int tipo()  { return(lstr ? 2 : 1); }//1 é numérico e 2 é do tipo string
int dentro_do_escopo(int v) { return( (min  v && v  max) ? 1 : 0); }
int dentro_do_escopo(char *s) { return( lstr-find(lstr,s) ); }
int dentro_do_escopo(string s) { return( lstr-find(lstr,s.cadeia()) ); }
int tam_escopo() { return( lstr ? lstr-tamanho() : max-min+1); }

//posicao relativa a partir do 0 de um deleterminado valor de escopo
// Ex: escopo variando de 12 a 25. A pos. rel. de 18 e' 6 (comeca de 0 )
int pos_relativa(int x) { return( dentro_do_escopo(x) ? x-min : -1 ); }
int pos_relativa(char *s);
int pos_relativa(string s);
int valor_num(int pos_rel) { return(pos_rel+min  max? pos_rel+min : -1); }
char* valor_str(int pos_rel);
//Operadores
escopo& operator=(char *s) { nome = s; return *this;}
escopo& operator=(escopo&);
~escopo() { delete lstr; }//destrutor
friend ostream& operator<<(ostream&, escopo&);
//friend istream& operator>(istream&, escopo&);
friend int operator==(escopo &x, char* s) {return !strcmp(x.nome.cadeia(),s);}
friend int operator==(escopo&, escopo&);
friend int operator!=(escopo &x, char *s) {return strcmp(x.nome.cadeia(),s);}
friend int operator!=(escopo&, escopo&);
};

class vescopolist : public vlist
{
public:
 void insert(escopo *a) { vlist::insert(a); }
 void append(escopo *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(escopo *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 escopo* recupera(escopo *a,CMPENT cmp) {return (escopo*) vlist::recupera(a,cmp);}
 void ordena(CMPENT cmp) { vlist::ordena(cmp); }
 escopo* remove(escopo *a) { return (escopo*) vlist::remove(a); }
```

```
    escopo* get()       { return (escopo*) vlist::get();}
    escopo* get(int x)  { return (escopo*)vlist::get(x);}
    escopo *getscroll()  { return (escopo*) vlist::getscroll(); }
    vescopolist()       {}
    vescopolist(escopo *a) : (a) {}
    int vazia() { return (int) vlist::vazia(); }
    int tamanho() { return (int) vlist::tamanho(); }
    int find(vescopolist *, escopo *);   //retorna 0 se nao achou o elemento
    int find(vescopolist *, char *);
    escopo* remove(vescopolist *, char *);
    escopo* recupera(vescopolist *, char *);
    void print_list(vescopolist *, char* );
};
```

Índices e Lista de Índices

A classe **Índices** é responsável pela representação computacional de um índice de parâmetro ou variável. Nesta classe armazenamos o nome, o escopo e a documentação dos índices, além de possuirmos funções para o tratamento dos mesmos. A classe **Lista de Índices** é baseada na classe **Lista Genérica** e implementa uma lista de índices.

Índices e Lista de Índices

```
extern F_ERROR indice_handler;
extern F_ERROR set_indice_handler(F_ERROR);
class indice : public string
{
private:
  string nome;//nome do índice
  escopo *esc;//escopo associado ao índice
  string doc;//documentação
public:
  //Construtores
  indice():nome(),esc(0),doc() {}
  indice(char *s): nome(s),esc(0),doc() {}
  indice(string s):nome(s),esc(0),doc() {}
  indice(char *s1,escopo *ee,char *s2): nome(s1),esc(0),doc(s2) { esc = ee; }
  indice(string s1,escopo *ee,string s2): nome(s1),esc(0),doc(s2) { esc = ee; }
  indice(indice &);
  //Acessar os dados da classe
  char* inome() { return nome.cadeia(); }
  escopo* iesc()  { return esc; }
```

```
char* idoc() { return doc.cadeia(); }
//Setar os dados
void inome(char *s) { nome = s; }
void iesc(escopo* e) { esc = e; }
void idoc(char *s) { doc = s; }
int tam_escopo() { return( esc ? esc-tam_escopo() : 0); }
int tipo() { return( esc ? esc-tipo() : 0); }
int dentro_do_escopo(int v) { return esc-dentro_do_escopo(v); }
int dentro_do_escopo(char *s) { return esc-dentro_do_escopo(s); }
int dentro_do_escopo(string s) { return esc-dentro_do_escopo(s); }
int prim_valor_int() { return(esc-emin()); }
char *prim_valor_str() { return(esc-estrlist()-get(0)-cadeia() ); }
int pos_relativa(int x) { return(esc-pos_relativa(x)); }
int pos_relativa(char *s) { return(esc-pos_relativa(s)); }
int pos_relativa(string s) { return(esc-pos_relativa(s)); }
int valor_num(int pos_rel) { return(esc-valor_num(pos_rel)); }
char* valor_str(int pos_rel) { return(esc-valor_str(pos_rel)); }
//Operadores
indice& operator=(char *s) { nome = s; return *this;}
indice& operator=(indice&);
~indice() { delete esc; }//destrutor
friend ostream& operator<(ostream&, indice&);
// friend istream& operator>(istream&, indice&);
friend int operator==(indice &x, char* s) {return !strcmp(x.nome.cadeia(),s);}
friend int operator==(indice&, indice&);
friend int operator!=(indice &x, char *s) {return strcmp(x.nome.cadeia(),s);}
friend int operator!=(indice&, indice&);
};

class vindicelist : public vlist
{
public:
 void insert(indice *a) { vlist::insert(a); }
 void append(indice *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(indice *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 indice* recupera(indice *a,CMPENT cmp) {return (indice*) vlist::recupera(a,cmp);}
 void ordena(CMPENT cmp) { vlist::ordena(cmp); }
 indice* remove(indice *a) { return (indice*) vlist::remove(a); }
 indice* get()      { return (indice*) vlist::get();}
 indice* get(int x)   { return (indice*)vlist::get(x);}
 indice *getscroll()  { return (indice*) vlist::getscroll(); }
 vindicelist()      {}
 vindicelist(indice *a) : (a) {}
 int vazia() { return (int) vlist::vazia(); }
 int tamanho() { return (int) vlist::tamanho(); }
```

```
 int find(vindicelist *, indice *);   //retorna 0 se não achou o elemento
 int find(vindicelist *, char *);
 indice* remove(vindicelist *, char *);
 indice* recupera(vindicelist *, char *);
 void print_list(vindicelist*, char* );
};
```

Variável de decisão e Lista de Variáveis

A classe **Variável** é responsável pela representação computacional de uma variável de decisão. Nesta classe armazenamos o nome, os índices e a documentação das variáveis, além de possuirmos funções para o tratamento dos mesmos. A classe **Lista de Variáveis** é baseada na classe **Lista Genérica** e implementa uma lista de variáveis de decisão.

Variável de decisão e Lista de Variáveis

```
extern F_ERROR decisao_handler;
extern F_ERROR set_decisao_handler(F_ERROR);
class decisao : public string
{
private:
  string nome;//nome da variável
  vindicelist *ind;//lista de índices
  string doc;//documentação
public:
  //Construtores
  decisao(): nome(),ind(0),doc() {}
  decisao(char *s): nome(s),ind(0),doc() {}
  decisao(string s): nome(s),ind(0),doc() {}
  decisao(char *s1,vindicelist *v,char *s2):nome(s1),ind(0),doc(s2) { ind = v;}
  decisao(string s1,vindicelist *v,string s2):nome(s1),ind(0),doc(s2) {ind = v;}
  decisao(decisao &);
  //Acessar dados da classe
  char* dnome() { return nome.cadeia(); }
  vindicelist* dind()  { return ind; }
  char* ddoc()  { return doc.cadeia(); }
  //Setar dados da classe
  void dnome(char *s) { nome = s; }
  void dnome(string s) { nome = s; }
  void dind(vindicelist* e) { ind = e; }
  void ddoc(char *s)  { doc = s; }
  void ddoc(string s) { doc = s; }
  //verifica se existe um determinado índice
  int existe_indice(char *s) { return(ind? ind-find(ind,s) : 0); }
  int existe_indice(string s) { return(ind? ind-find(ind,s.cadeia()) : 0); }
  //verifica se existe o valor de um determinado índice
```

```
int existe_valor_indice(char *,int);
int existe_valor_indice(char *, char *);
int existe_valor_indice(char *, string);
int num_indices() { return(ind? ind->tamanho(): 0); }
//Operadores
decisao& operator=(char *s) { nome = s; return *this;}
decisao& operator=(decisao&);
~decisao() { delete ind; }//Destrutor
friend ostream& operator<<(ostream&, decisao&);
// friend istream& operator>>(istream&, decisao&);
friend int operator==(decisao &x, char* s){return !strcmp(x.nome.cadeia(),s);}
friend int operator==(decisao&, decisao&);
friend int operator!=(decisao &x, char *s) {return strcmp(x.nome.cadeia(),s);}
friend int operator!=(decisao&, decisao&);
};

class vdecisaolist : public vlist
{
public:
 void insert(decisao *a) { vlist::insert(a); }
 void append(decisao *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(decisao *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 decisao* recupera(decisao *a,CMPENT cmp) {return (decisao*) vlist::recupera(a,cmp);}
 void ordena(CMPENT cmp) { vlist::ordena(cmp); }
 decisao* remove(decisao *a) { return (decisao*) vlist::remove(a); }
 decisao* get()       { return (decisao*) vlist::get(); }
 decisao* get(int x)   { return (decisao*) vlist::get(x); }
 decisao* getscroll()  { return (decisao*) vlist::getscroll(); }
 vdecisaolist()        {}
 vdecisaolist(decisao *a) : (a) {}
 int vazia() { return (int) vlist::vazia(); }
 int tamanho() { return (int) vlist::tamanho(); }
 int find(vdecisaolist *, decisao *);   //retorna 0 se não achou o elemento
 int find(vdecisaolist *, char *);
 decisao* remove(vdecisaolist *, char *);
 decisao* recupera(vdecisaolist *, char *);
 void print_list(vdecisaolist*, char* );
};
```

Parâmetros e Lista de Parâmetros

A classe **Parâmetros** é responsável pela representação computacional de um parâmetro. Nesta classe armazenamos o nome, os índices, a documentação e os valores dos parâmetros, além de possuirmos funções para o tratamento dos mesmos. A classe **Lista de Parâmetros** é baseada na classe **Lista Genérica** e implementa uma lista de parâmetros.

Parâmetros e Lista de Parâmetros

```
extern F_ERROR param_handler;
extern F_ERROR set_param_handler(F_ERROR);
class param : public string
{
private:
  string nome;//nome do parâmetro
  vindicelist *ind;//lista de índices
  vvaloreslist *listval;//lista de valores de parâmetros
  string doc;//documentação
public:
  //Construtores
  param():nome(),ind(0),listval(0),doc() { listval=new vvaloreslist;}
  param(char *s):nome(s),ind(0),listval(0),doc() {listval=new vvaloreslist;}
  param(string s):nome(s),ind(0),listval(0),doc() {listval=new vvaloreslist;}
  param(char *s1,vindicelist *vi,vvaloreslist *vs,char *s2);
  param(string s1,vindicelist *vi,vvaloreslist *vs,string s2);
  param(param &);
  //Acessar dados da classe
  char* pnome() { return nome.cadeia(); }
  vindicelist* pind()  { return ind; }
  vvaloreslist* plistval()  { return listval; }
  char* pdoc()  { return doc.cadeia(); }
  //Setar dados da classe
  void pnome(char *s) { nome = s; }
  void pnome(string s) { nome = s; }
  void pind(vindicelist* e) { ind = e; }
  void plistval(vvaloreslist *lv) { listval = lv; }
  void pdoc(char *s)  { doc = s; }
  void pdoc(string s)  { doc = s; }
  //verifica se existe um determinado índice
  int existe_indice(char *s) { return(ind? ind-find(ind,s) : 0); }
  int existe_indice(string s) { return(ind? ind-find(ind,s.cadeia()) : 0); }
  //verifica se existe o valor de um determinado índice
  int existe_valor_indice(char *,int);
  int existe_valor_indice(char *, char *);
  int existe_valor_indice(char *, string);
  int num_indices() { return(ind? ind-tamanho(): 0); }
  //Operadores
  param& operator=(char *s) { nome = s; return *this;}
  param& operator=(param&);
  ~param();
  friend ostream& operator<(ostream&, param&);
  // friend istream& operator>(istream&, param&);
```

```
 friend int operator==(param &x, char* s) {return !strcmp(x.nome.cadeia(),s);}
 friend int operator==(param&, param&);
 friend int operator!=(param &x, char *s) {return strcmp(x.nome.cadeia(),s);}
 friend int operator!=(param&, param&);
};

class vparamlist : public vlist
{
public:
 void insert(param *a) { vlist::insert(a); }
 void append(param *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(param *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 param* recupera(param *a,CMPENT cmp) {return (param*) vlist::recupera(a,cmp);}
 void ordena(CMPENT cmp) { vlist::ordena(cmp); }
 param* remove(param *a) { return (param*) vlist::remove(a); }
 param* get()      { return (param*) vlist::get(); }
 param* get(int x)   { return (param*) vlist::get(x); }
 param* getscroll()  { return (param*) vlist::getscroll(); }
 vparamlist()      {}
 vparamlist(param *a) : (a) {}
 int vazia() { return (int) vlist::vazia(); }
 int tamanho() { return (int) vlist::tamanho(); }
 int find(vparamlist *, param *);   //retorna 0 se não achou o elemento
 int find(vparamlist *, char *);
 param* remove(vparamlist *, char *);
 param* recupera(vparamlist *, char *);
 void print_list(vparamlist*, char* );
};
```

Objetivos e Lista de Objetivos

A classe **Objetivos** é responsável pela representação computacional de uma função objetivo. Nesta classe armazenamos o ID (nº identificador), a direção (maximização ou minimização), a função objetivo propriamente dita e a documentação dos objetivos, além de possuirmos funções para o tratamento dos mesmos. A classe **Lista de Objetivos** é baseada na classe **Lista Genérica** e implementa uma lista de objetivos.

Objetivos e Lista de Objetivos

```
extern F_ERROR objetivo_handler;
extern F_ERROR set_objetivo_handler(F_ERROR);
class objetivo: public string
{
private:
 string obj;//objetivo digitado pelo usuário
```

```
 string obj2;//objetivo em outro formato
 int   id;//ID do objetivo
 int   max_min;//0 é minimizar e 1 é maximizar
 string doc;//documentação
public:
 //Construtores
 objetivo(): obj(),obj2(),id(0),max_min(0),doc() { }
 objetivo(char *s): obj(s),obj2(),id(0),max_min(0),doc() { }
 objetivo(int n, char *s1, char *s2): obj(s1),obj2(),id(n),max_min(0),doc(s2) { }
 objetivo(objetivo &);
 //Acessar dados da classe
 char* oobj() { return obj.cadeia(); }
 char *oobj2() { return obj2.cadeia(); }
 int  oid()  { return id; }
 int  omax_min() { return max_min; }
 char* odoc()  { return doc.cadeia(); }
 //Setar dados da classe
 void oobj(char *s) { obj = s; }
 void oobj(string s) { obj = s; }
 void oobj2(char *s) { obj2 = s; }
 void oobj2(string s) { obj2 = s; }
 void oid(int ii) { id = ii; }
 void omax_min(int n) { max_min = ( (n==0||n==1) ? n : 0); }
 void odoc(char *s)  { doc = s; }
 void odoc(string s)  { doc = s; }
 //Operadores
 objetivo& operator=(char *s) { obj = s; return *this;}
 objetivo& operator=(int ii) { id = ii; return *this;}
 objetivo& operator=(objetivo&);
 //~objetivo();
 friend ostream& operator<<(ostream&, objetivo&);
 // friend istream& operator>>(istream&, objetivo&);
 friend int operator==(objetivo &x, char* s){return !strcmp(x.obj.cadeia(),s);}
 friend int operator==(objetivo&, objetivo&);
 friend int operator!=(objetivo &x, char *s) {return strcmp(x.obj.cadeia(),s);}
 friend int operator!=(objetivo&, objetivo&);
};

class vobjetivolist : public vlist
{
public:
 void insert(objetivo *a) { vlist::insert(a); }
 void append(objetivo *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(objetivo *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 objetivo* recupera(objetivo *a,CMPENT cmp) {return (objetivo*) vlist::recupera(a,cmp);}
```

```
void ordena(CMPENT cmp) { vlist::ordena(cmp); }
objetivo* remove(objetivo *a) { return (objetivo*) vlist::remove(a); }
objetivo* get() { return (objetivo*) vlist::get(); }
objetivo* get(int x) { return (objetivo*) vlist::get(x); }
objetivo *getscroll() { return (objetivo*) vlist::getscroll(); }
vobjetivolist() { }
vobjetivolist(objetivo *a) : (a) { }
int vazia() { return (int) vlist::vazia(); }
int tamanho() { return (int) vlist::tamanho(); }
int find(vobjetivolist *, objetivo *);   //retorna 0 se nao achou o elemento
int find(vobjetivolist *, char *);
int find(vobjetivolist *, int);
objetivo* remove(vobjetivolist *, char *);
objetivo* recupera(vobjetivolist *, char *);
objetivo* remove(vobjetivolist *, int);
objetivo* recupera(vobjetivolist *, int);
void print_list(vobjetivolist*, char* );
};
```

Restrição Funcional e Lista de Restrições Funcionais

A classe **Restrição Funcional** é responsável pela representação computacional das restrições funcionais de um problema de otimização. Nesta classe armazenamos o ID (nº identificador), a função da restrição, o lado direito da função, o sinal de comparação da função ($\geq$, = , $\leq$), os índices de variação e a documentação das restrições, além de possuirmos funções para o tratamento das mesmas. A classe **Lista de Restrições** é baseada na classe **Lista Genérica** e implementa uma lista de restrições funcionais.

Restrição Funcional e Lista de Restrições Funcionais

```
extern F_ERROR rest_func_handler;
extern F_ERROR set_rest_func_handler(F_ERROR);
class rest_func : public string
{
private:
  int id;//ID da restrição
  string rest;//restrição digitada pelo usuário
  string rest2;//restrição em outro formato
  string sinal;//sinal de comparação
  string lado_dir;//lado direito digitado pelo usuário
  string lado_dir2;//lado direito em outro formato
  vstringlist *qq;//lista de índices do tipo "para todo"
  string doc;//documentação
public:
  //Construtores
  rest_func():id(0),rest(),rest2(),sinal(),lado_dir(), lado_dir2(),qq(0),doc() { }
```

```
rest_func(char *s): id(0),rest(s),rest2(),sinal(), lado_dir(), lado_dir2(), qq(0),doc() { }
rest_func(string s):id(0),rest(s),rest2(),sinal(), lado_dir(),lado_dir2(),qq(0),doc() { }
rest_func(int n,char *s1,char *s2,char *s3,char *s4):id(n),rest(s1),rest2(),sinal(s2),lado_dir(s3),lado_dir2(),qq(0),doc(s4){ }
rest_func(int n,string s1,string s2,string s3,string s4): id(n),rest(s1),rest2(),sinal(s2), lado_dir(s3),lado_dir2(), qq(0),doc(s4){ }
rest_func(rest_func &);
//Acessar dados da classe
int   rid()  { return id; }
char* rrest() { return rest.cadeia(); }
char* rrest2() { return rest2.cadeia(); }
char* rsinal() { return sinal.cadeia(); }
char* rlado_dir() { return lado_dir.cadeia(); }
char* rlado_dir2() { return lado_dir2.cadeia(); }
vstringlist* rqq() { return qq; }
//Setar dados da classe
char* rdoc()  { return doc.cadeia(); }
void rid(int ii) { id = ii; }
void rrest(char *s) { rest = s; }
void rrest2(char *s) { rest2 = s; }
void rsinal(char *s) { sinal = s; }
void rlado_dir(char *s) { lado_dir = s; }
void rlado_dir2(char *s) { lado_dir2 = s; }
void rdoc(char *s)  { doc = s; }
void rqq(vstringlist *vv) { qq = vv; }
void rrest(string s) { rest = s; }
void rrest2(string s) { rest2 = s; }
void rsinal(string s) { sinal = s; }
void rlado_dir(string s) { lado_dir = s; }
void rlado_dir2(string s) { lado_dir2 = s; }
void rdoc(string s)  { doc = s; }
void rqq(vstringlist vv) { *qq = vv; }
int existe_indice() { return(qq? 1 : 0); }
//Operadores
rest_func& operator=(char *s) { rest = s; return *this;}
rest_func& operator=(string s) { rest = s; return *this;}
rest_func& operator=(int ii) { id = ii; return *this;}
rest_func& operator=(rest_func&);
//~rest_func();
friend ostream& operator<<(ostream&, rest_func&);
// friend istream& operator>>(istream&, rest_func&);
friend int operator==(rest_func &x,char *s){return!strcmp(x.rest.cadeia(),s);}
friend int operator==(rest_func&, rest_func&);
friend int operator!=(rest_func &x,char *s){return strcmp(x.rest.cadeia(),s);}
friend int operator!=(rest_func&, rest_func&);
};

class vrest_funclist : public vlist
```

```
{
public:
 void insert(rest_func *a) { vlist::insert(a); }
 void append(rest_func *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(rest_func *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 rest_func* recupera(rest_func *a,CMPENT cmp) {return (rest_func*) vlist::recupera(a,cmp);}
 void ordena(CMPENT cmp) { vlist::ordena(cmp); }
 rest_func* remove(rest_func *a) { return (rest_func*) vlist::remove(a); }
 rest_func* get() { return (rest_func*) vlist::get(); }
 rest_func* get(int x) { return (rest_func*) vlist::get(x); }
 rest_func *getscroll() { return (rest_func*) vlist::getscroll(); }
 vrest_funclist() {}
 vrest_funclist(rest_func *a) : (a) {}
 int vazia() { return (int) vlist::vazia(); }
 int tamanho() { return (int) vlist::tamanho(); }
 int find(vrest_funclist *, rest_func *);   //retorna 0 se não achou o elemento
 int find(vrest_funclist *, char *);
 int find(vrest_funclist *, int);
 rest_func* remove(vrest_funclist *, char *);
 rest_func* recupera(vrest_funclist *, char *);
 rest_func* remove(vrest_funclist *, int);
 rest_func* recupera(vrest_funclist *, int);
 void print_list(vrest_funclist*, char* );
};
```

Restrição Canalizada e Lista de Restrições Canalizadas

A classe **Restrição Canalizada** é responsável pela representação computacional das canalizações nas variáveis de um problema de otimização. Nesta classe armazenamos o nome da variável a ser canalizada, a instância da variável (os valores dos índices), os limites inferior e superior e a documentação das canalizações, além de possuirmos funções para o tratamento das mesmas. A classe **Lista de Restrições Canalizadas** é baseada na classe **Lista Genérica** e implementa uma lista de restrições canalizadas.

Restrição Canalizada e Lista de Restrições Canalizadas

```
extern F_ERROR rest_canal_handler;
extern F_ERROR set_rest_canal_handler(F_ERROR);
class rest_canal : public string
{
private:
 string var;//nome da variável
  vval_indlist *list_ind;//lista de índices
  double min;//valor mínimo
  double max;//valor máximo
```

```
 string doc;//documentação
public:
 //Construtores
 rest_canal():var(),list_ind(0),min(0), max(d_INFINITO),doc() { list_ind = new vval_indlist;}
 rest_canal(char *s):var(s),list_ind(0),min(0), max(d_INFINITO), doc() { list_ind = new vval_indlist;}
 rest_canal(string s):var(s),list_ind(0),min(0), max(d_INFINITO), doc(){ list_ind=new vval_indlist;}
 rest_canal(char* s,double a,double b):var(s), list_ind(0), min(a),max(b),doc() { list_ind=new vval_indlist;}
 rest_canal(string s,double a,double b):var(s),list_ind(0), min(a),max(b),doc() { list_ind=new vval_indlist;}
 rest_canal(char*,vval_indlist*,double,double,char*);
 rest_canal(string,vval_indlist*,double,double,string);
 rest_canal(rest_canal &);
 //Acessar dados da classe
 char* rcvar() { return var.cadeia(); }
 vval_indlist* rclist_ind() { return list_ind; }
 double rcmin() { return min; }
 double rcmax() { return max; }
 char* rcdoc() { return doc.cadeia(); }
 //Setar dados da classe
 void rcvar(char *s) { var = s; }
 void rcvar(string s) { var = s; }
 void rclist_ind(vval_indlist *v) { list_ind = v; }
 void rcmin(double menor) { min = menor; }
 void rcmax(double maior) { max = maior; }
 void rcdoc(char *s) { doc = s; }
 void rcdoc(string s) { doc = s; }
 //Operadores
 rest_canal& operator=(char *s) { var = s; return *this;}
 rest_canal& operator=(rest_canal&);
 ~rest_canal() {delete list_ind; }
 friend ostream& operator<<(ostream&, rest_canal&);
 // friend istream& operator>>(istream&, rest_canal&);
 friend int operator==(rest_canal &x,char* s){return !strcmp(x.var.cadeia(),s);}
 friend int operator==(rest_canal&, rest_canal&);
 friend int operator!=(rest_canal &x,char *s){return strcmp(x.var.cadeia(),s);}
 friend int operator!=(rest_canal&, rest_canal&);
};
class vrest_canallist : public vlist
{
public:
 void insert(rest_canal *a) { vlist::insert(a); }
 void append(rest_canal *a) { vlist::append(a); }
 void clear() { vlist::clear(); }
 void insordem(rest_canal *a,CMPENT cmp) { vlist::insordem(a,cmp); }
 rest_canal* recupera(rest_canal *a,CMPENT cmp) {return (rest_canal*) vlist::recupera(a,cmp);}
 void ordena(CMPENT cmp) { vlist::ordena(cmp); }
 rest_canal* remove(rest_canal *a) { return (rest_canal*) vlist::remove(a); }
```

```
rest_canal* get()        { return (rest_canal*) vlist::get(); }
rest_canal* get(int x)   { return (rest_canal*) vlist::get(x); }
rest_canal *getscroll()  { return (rest_canal*) vlist::getscroll(); }
vrest_canallist()        { }
vrest_canallist(rest_canal *a) : (a) { }
int vazia() { return (int) vlist::vazia(); }
int tamanho() { return (int) vlist::tamanho(); }
int find(vrest_canallist *, rest_canal *);//retorna 0 se não achou o elemento
int find(vrest_canallist *, char *);
int find(vrest_canallist *, char *, vval_indlist *);
rest_canal* remove(vrest_canallist *, rest_canal *);
rest_canal* remove(vrest_canallist *, char *);
rest_canal* remove(vrest_canallist *, char *, vval_indlist *);
rest_canal* recupera(vrest_canallist *, rest_canal *);
rest_canal* recupera(vrest_canallist *, char *);
rest_canal* recupera(vrest_canallist *, char *, vval_indlist *);
void print_list(vrest_canallist*, char* );
};
```

Apêndice B
Dicionário de Dados

O software MC++ possui muitas variáveis utilizadas para o tratamento da interface com o usuário, para o tratamento dos problemas de otimização e para a análise léxica e semântica das funções (objetivos e restrições funcionais). Devido ao volume da estrutura de dados é essencial a elaboração de um dicionário de dados, contendo o nome da variável e uma descrição suscinta das suas características e escopo de utilização. Este apêndice contém um dicionário com os principais dados utilizados e uma breve descrição dos mesmos.

- frame: frame básico ou principal.
- sf_arq: frame de comando para o opção Arquivos.
- sf_escopo: frame de comando para escopos.
- sf_indice: frame de comando para índices.
- sf_decisao: frame de comando para variáveis de decisão.
- sf_parametro: frame de comando para parâmetros.
- sf_par_dados: frame de comando para valores dos parâmetros.
- sf_reldados: frame de comando para apresentação do dados.
- sf_help: frame de comando para as telas de help.
- sf_objetivo: frame de comando para objetivos.
- sf_restricao: frame de comando para as restrições funcionais.
- sf_restc1,sf_restcanal: frames de comando para as restrições canalizadas.
- sf_agregar: frame de comando para agregar modelos.
- sf_font: frame de comando para tratamento de fontes.
- sf_tutorial: frame de comando para as telas de tutorial.
- sf_copyright: frame de comando para copyright.
- sf_checar: frame de comando para checagem dos dados.
- sf_relat, sf_relat_page: frames de comando para relatório.
- sf_solv: frame de comando para as telas de solução.
- sf_metodo_111: frame de comando para o método Global Criterion.
- sf_metodo_211: frame de comando para o método Utility Function.

- sf_metodo_212: frame de comando para o método Bounded Objective.
- sf_metodo_221: frame de comando para o método Lexicographic.
- sf_metodo_222: frame de comando para o método Goal Programming.
- sf_metodo_223: frame de comando para o método Goal Atainment.
- sf_metodo_311: frame de comando para o método Geoffrion.
- sf_metodo_312: frame de comando para o método Surrogate Worth Trade-off.
- sf_metodo_313: frame de comando para o método Satisfactory Goals.
- sf_metodo_314: frame de comando para o método Zionts-Wallenius.
- sf_metodo_321: frame de comando para o método STEM.
- sf_metodo_322: frame de comando para o método SEMOPS and SIGMOP.
- sf_metodo_323: frame de comando para o método Displaced Ideal.
- sf_metodo_324: frame de comando para o método GPSTEM.
- sf_metodo_325: frame de comando para o método Steuer.
- sf_metodo_411: frame de comando para o método Parametric.
- sf_metodo_412: frame de comando para o método epson-constraint.
- sf_metodo_413: frame de comando para o método MOLP.
- sf_metodo_414: frame de comando para o método Adaptive Search.
- canvas1: canvas principal onde é apresentado o modelo corrente nas formas simbólica e matricial.
- canvas_solv: canvas onde é apresentado o resultado da resolução de um modelo.
- canvas_help: canvas onde é apresentado mensagens de help.
- canvas_obj: canvas associado a tela de diálogo para objetivos. Neste canvas é apresentado o objetivo que acaba de ser definido.
- canvas_rest: canvas associado a tela de diálogo para restrições funcionais. Neste canvas é apresentado a restrição funcional que acaba de ser definida.
- canvas_font: canvas onde são apresentadas as alterações efetuadas nas fontes.
- canvas_chec: canvas onde é apresentado os resultados de checagem do modelo corrente.
- canvas_relat: canvas associado a opção relatório. Neste canvas são apresentadas as páginas do relatório sobre o modelo corrente.
- canvas_geof: canvas onde é apresentada a tabela de valores de função objetivo do método Geoffrion, Dyer e Feinberg
- canvas_stem: canvas onde é apresentada a tabela de valores de função objetivo do método STEM
- panel: panel principal associado ao frame básico.
- sub_panel: panel utilizado para montar as telas de diálogo.
- menu: menu principal associado ao frame básico.
- menu_var: menu associado ao botão Primitivas.
- menu_arq: menu associado ao botão Arquivos.
- menu_visao: menu associado ao botão Visão.

- menu_mod: menu associado ao botão Modelos.
- menu_mod_rest: menu associado ao botão Restrições do menu Modelos.
- menu_op: menu associado ao botão Opções.
- menu_estru: menu associado ao botão Estrutura.
- cur_eq1: contém um objetivo ou restrição funcional a ser submetida ao analisador léxico.
- cur_eq2: contém o objetivo ou restrição funcional passado em cur_eq1 porém em formato diferente. Este formato é útil na apresentação do modelo nos canvas.
- erro_parser: contém uma mensagem de erro caso o analisador léxico encontre algum.
- list_par: vetor contendo os parâmetros definidos. Este vetor é passado para o analisador léxico a fim de que este possa checar a consistência de um objetivo ou restrição funcional.
- list_var: vetor contendo as variáveis definidas. Este vetor é passado para o analisador léxico a fim de que este possa checar a consistência de um objetivo ou restrição funcional.
- list_ind: vetor contendo os índices definidos. Este vetor é passado para o analisador léxico a fim de que este possa checar a consistência de um objetivo ou restrição funcional.
- tam_par: inteiro contendo o tamanho do vetor de parâmetros.
- tam_var: inteiro contendo o tamanho do vetor de variáveis de decisão.
- tam_ind: inteiro contendo o tamanho do vetor de índices.
- modelo_corrente: nome do modelo corrente.
- arq_ent_simplex: nome do arquivo de entrada para uma rotina externa que implementa o método simplex.
- arq_sai_simplex: nome do arquivo de saída gerado pala rotina externa que implementa o método simplex.
- titulo: título dado ao modelo corrente para efeitos de apresentação do relatório.
- usuário: nome do usuário que está utilizando o sistema.
- dir_corrente: diretório onde está armazenado o modelo corrente.
- metodo_def: método a ser utilizado na resolução do problema multiobjetivo.
- indicacao_def: tipo de indicação utilizada na resolução de um modelo. Pose ser *sem indicação, a priori, a posteriori* ou *interativa.*
- tip_info_def: tipo de informação utilizada na resolução de um modelo. Pode ser *cardinal, ordinal e cardinal, trade-off implícito* ou *trade-off explícito.*
- alterar_relatorio: inteiro que indica se o relatório de dados deve ser refeito.
- alterado_mod_corrente: inteiro que indica se o modelo corrente foi modificado.
- ultima_visao: indica se a última visão escolhida pelo usuário foi a simbólica ou a matricial.
- pagina: página do relatório a ser apresentada.
- num_pag: número de páginas do relatório.
- num_solucao: número de soluções a serem apresentadas no relatório.
- id_obj_refer: ID do objetivo de referência.
- erro_simplex: inteiro indicando se houve erro no método simplex (infactível, ilimitado, matriz singular, máximo de iterações).
- pond_param: vetor com as ponderações de cada função objetivo. Utilizado no método Paramétrico.

- crit_global_p: valor do p para o método Critério Global. Esta variável pode assumir os valores 1, 2 e 3 indicando a norma para p valendo 1, 2 e infinito respectivamente.
- crit_global_ideal: vetor com os valores ideais de cada função objetivo. Utilizado no método Critério Global.
- crit_global_ci: vetor contendo os coeficientes de todas as funções objetivos. Utilizado no método Critério Global.
- crit_global_beta: vetor contendo os valores de beta para o método Critério Global.
- crit_global_grad: vetor contendo o gradiente. Utilizado no método Critério Global.
- crit_global_d: vetor contendo a direção entre a uma dada solução e a solução obtida com o gradiente. Utilizado no método Critério Global.
- epson_lado_dir: vetor contendo o lado direito de uma restrição. Utilizado no método Epsolon-Restrito.
- limite_inf: vetor contendo os limitantes inferiores para o método Objetivos Limitados.
- limite_sup: vetor contendo os limitantes superiores para o método Objetivos Limitados.
- id_lexic: vetor contendo os IDs dos objetivos segundo uma ordem. Utilizado no método Lexicográfico.
- lex_lado_dir: vetor contendo o lado direito de uma restrição. Utilizado no método Lexicográfico.
- lex_obj_id_corrente: ID do objetivo a ser otimizado quando utilizamos o método Lexicográfico.
- alfa_lexic: vetor contendo os valores de alfa para o método Lexicográfico.
- stem_ideal: vetor com os valores ideais de cada função objetivo. Utilizado no método STEM.
- stem_pessimo: vetor com os valores pessimistas de cada função objetivo. Utilizado no método STEM.
- stem_ci: vetor contendo os coeficientes das funções objetivos. Utilizado no método STEM.
- stem_tabela: vetor contendo a tabela dos valores das funções objetivo para o método STEM.
- stem_id_sat: vetor contendo a lista de IDs dos objetivos satisfatórios para o método STEM.
- stem_delta: vetor contendo os valores de delta para o método STEM.
- stem_iteracao: indica o número da iteração para o método STEM.
- geof_delta: vetor contendo os valores de delta para o método Geoffrion, Dyer e Feinberg.
- geof_list_id: Lista dos IDs dos objetivos para o método Geoffrion, Dyer e Feinberg.
- geof_peso: vetor contendo os pesos atribuídos aos objetivos não de referência no método Geoffrion, Dyer e Feinberg.
- geof_sol: vetor contendo a solução corrente para o método Geoffrion, Dyer e Feinberg.
- geof_ci: vetor contendo os coeficientes das funções objetivo. Utilizado no método Geoffrion, Dyer e Feinberg.
- geof_tabela: vetor contendo a tabela dos valores das funções objetivo para o método Geoffrion, Dyer e Feinberg.
- geof_discret: número de discretizações para a apresentação da tabela de valores no método Geoffrion, Dyer e Feinberg.
- alt_max_min: inteiro indicando se o objetivo foi alterado de maximização para minimização.
- val_xs: vetor contendo a solução ótima do modelo.
- val_dual: vetor contendo a solução dual ótima.
- val_obj: vetor contendo os valores das funções objetivo.
- min_obj: vetor indicando se o objetivo é de minimização ou maximização.

- A: matriz utilizada pela rotina simplex.
- xs: vetor de solução utilizado pela rotina simplex.
- bo: vetor contendo o lado direito das restrições para a rotina simplex.
- ci: vetor contendo os coeficientes da função objetivo da rotina simplex.
- u: vetor contendo os limitantes superiores das variáveis para a rotina simplex.
- w: vetor contendo a solução dual utilizado pela rotina simplex.
- sig: vetor contendo os sinais das restrições (=, =, =) para a rotina simplex.
- li: número de linhas da matriz de restrições para a rotina simplex.
- co: número de colunas da matriz de restrições para a rotina simplex.
- TipoErro: conjunto contendo os valores *infactível, ilimitado, iteração* e *singular* utilizado pela rotina simplex.
- lesc: lista de escopos.
- lind: lista de índices.
- lpar: lista de parâmetros
- ldec: lista de variáveis de decisão.
- lobj: lista de objetivos.
- lrestf: lista de restrições funcionais.
- lrestc: lista de restrições canalizadas.
- lvet_obj: lista contendo os objetivos no formato matricial.
- lvet_restf: lista contendo as restrições funcionais no formato matricial.
- lvet_restc: lista contendo as restrições canalizadas no formato matricial.
- lcadeia: lista de strings para uso geral.
- report: vetor de listas de strings contendo as linha do relatório de dados.
- rep_sol: vetor de listas de strings contendo as soluções a serem acrescentadas no relatório.
- list_desc: lista de strings contendo a descrição modelo.

Apêndice C
Analisador Sintático

Arquivo que controla a chamada do analisador sintático - mc.eq.c

```
#include <stdio.h>
#include <ctype.h>
#include <malloc.h>
#include <string.h>
#include "mc.defines.h"
int poseq=0;
static int erro;
int cont_ap=0,cont_fp=0, cont_ac=0, cont_fc=0;
int cont_chaves=0,nind_sum=0,chave=0;
extern char *cur_eq1;
extern char *cur_eq2;
extern char *erro_parser;
extern char *list_par[];
extern char *list_var[];
extern char *list_ind[];
extern int tam_par,tam_var,tam_ind;
extern int yyparse();
int nome_na_lista(nome,lista,num_elem)
char *nome;
char *lista[];
{
  int i=-1;
  while ( ++i  num_elem)
    if ( !strcmp(nome,lista[i]) ) return 1;
  return -1;
}
int yyerror(msg)
char *msg;
{
  printf("\nDentro de yyerror - poseq = : %d\n",poseq);
  fprintf(stderr,"*** Erro: %s\n",msg);
  if (!erro) strcpy(erro_parser,msg);
  erro = 1;
  /*Os comandos abaixo forcam yyparse a retornar um erro de sintaxe
```

```
 poseq=0;
 cur_eq1[0] = '\0';*/
}
int main_parser()
{
 int res=0;
 poseq = 0; erro = 0;
 cont_ap=0,cont_fp=0, cont_ac=0, cont_fc=0;
 cont_chaves=0,nind_sum=0,chave=0;
 res = yyparse();
 if ( erro ) res = erro;
 return(res);
}
```

Regras do Analisador Léxico (arquivo mc.regras.lex)

```
extern int nome_na_lista(), yyerror();
extern char *list_par[], *list_var[], *list_ind[];
extern int tam_ind,tam_par,tam_var;
extern char *cur_eq1,*cur_eq2;
extern int poseq;
/*O comando return(';') e' utilizado para forcar um erro de sintaxe
  no parser pois o caracter ';' e' ilegal na sintaxe em questao  */
extern int cont_ap,cont_fp, cont_ac, cont_fc;
/*contadores abre parenteses e colchetes e fecha parent e col*/
extern int cont_chaves, nind_sum, chave;
static char msg_erro[80];
%%
[ \t]+;
\;
\#\$\@\_ldir\_\@\$\#{ return(LADO_DIR); }
[Ss][Uu][Mm][\t]*\_{
 nind_sum++;
 yyless(yyleng-1);
 strcat(cur_eq2,d_ID_SUM);
 /*strcat(cur_eq2,yytext);*/
 strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
 return(SUM);
}

[A-Za-z][A-Za-z0-9!@#$%^&_]*
{
 if(yytext[yyleng-1] == '_' ) yyless(yyleng - 1);
 if ( chave )
 {
  if(nome_na_lista(yytext,list_ind,tam_ind) != -1)
```

```
 {
  strcat(cur_eq2,d_ID_IND);
  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return(IND);
 }
 else
 {
  strcat(cur_eq2,d_ID_IND_LIT);
  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return(IND);
 }
}
else
{
 if (nome_na_lista(yytext,list_par,tam_par) != -1)
 {
  strcat(cur_eq2,d_ID_PAR);
  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return(PAR);
 }
 else
 if (nome_na_lista(yytext,list_var,tam_var) != -1)
 {
  strcat(cur_eq2,d_ID_VAR);
  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return(VAR);
 }
 else
 {
  sprintf(msg_erro,"%s : %s",yytext,d_PARSER_E002);
  yyerror(msg_erro);
  return(';');
 }
 }
}
[0-9]+{
 if (chave)
 {
  strcat(cur_eq2,d_ID_IND_NUM);
  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return(IND_NUM);
```

```
 }
 else
 {
  strcat(cur_eq2,d_ID_NUM);
  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return(NUM);
 }
}
[0-9]+"."[0-9]*([eE][+-]?[0-9]+)?{
 strcat(cur_eq2,d_ID_NUM);
 strcat(cur_eq2,yytext);
 strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
 return(NUM);
}
[0-9]*"."[0-9]+([eE][+-]?[0-9]+)?{
 strcat(cur_eq2,d_ID_NUM);
 strcat(cur_eq2,yytext);
 strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
 return(NUM);
}
[0-9]+([eE][+-]?[0-9]+){
 strcat(cur_eq2,d_ID_NUM);
 strcat(cur_eq2,yytext);
 strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
 return(NUM);
}
\_[\t]*\{{  cont_chaves++; chave = 1; }
\({  cont_ap++; strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return('(');
}
\+{  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return('+');
}
\-{  strcat(cur_eq2,yytext);
  strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
  return('-');
}
\*;
\[{  cont_ac++; return('['); }
\_[\t]*[^{]{
  yyerror(d_PARSER_E003);
  return(';');
}
```

```
\{{
 yyerror(d_PARSER_E004);
 cont_chaves++;
 return(';');
}
\}{
 if ( nind_sum  1 )
 {
  yyerror(d_PARSER_E005);
  return(';');
 }
 nind_sum = 0;
 chave = 0;
 cont_chaves--;
}
\){
 cont_fp++;
 if (cont_fp  cont_ap )
 {
  yyerror(d_PARSER_E006);
  return(';');
 }
 strcat(cur_eq2,yytext); strcat(cur_eq2,d_SEP_TOKEN);/*ECHO;*/
 return(')');
}
\]{
 cont_fc++;
 if (cont_fc  cont_ac )
 {
  yyerror(d_PARSER_E007);
  return(';');
 }
 return(']');
}
\n{
 if ( cont_chaves != 0 || (cont_fp - cont_ap) != 0
|| (cont_fc - cont_ac) != 0 )
 {
  yyerror(d_PARSER_E008);
  return(';');
 }
 cont_chaves = cont_fp = cont_ap = 0;
 return(0);
}
[^=\\]*{
 sprintf(msg_erro,"%s : %s",yytext,d_PARSER_E009);
```

```
 yyerror(msg_erro);
 return(';');
}
[\:\;\?\^\~\'\"\%\,]*{
 sprintf(msg_erro,"%s : %s",yytext,d_PARSER_E009);
 yyerror(msg_erro);
 return(';');
}
```

Regras do Analisador Sintático (arquivo mc.regras.yacc)

```
%{
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include "mc.defines.h"
extern char *cur_eq1,*cur_eq2;
extern int yyerror();
extern int nome_na_lista();
%}
/* %start funcao */
%token PAR VAR SUM IND IND_NUM NUM LADO_DIR
%{
# include "lex.yy.c"
%}
%left '+' '-'
%left '*'
%%
bala: funcao
| ladodir
;
funcao: funcpar
| funcao '+' funcpar
| funcao '-' funcpar
;
funcpar: '[' exprn ']'
| '[' exprl ']'
| '[' somat ']'
| '[' '+' somat ']'
| '[' '-' somat ']'
| '[' NUM somat ']'
| '[' '+' NUM somat ']'
| '[' '-' NUM somat ']'
;
somat: SUM IND '(' exprl ')'
```

```
| SUM IND '(' somat ')'
| somat '+' exprl
| somat '-' exprl
| somat '+' somat
| somat '-' somat
;
exprl: termol
| '+' termol
| '-' termol
| exprl '+' termol
| exprl '-' termol
;
exprn: termon
| '+' termon
| '-' termon
| exprn '+' termon
| exprn '-' termon
;
termol: VAR indl
| PAR indmis VAR indl
| NUM termol
;
termon: VAR indmis
| PAR indmis VAR indmis
| NUM termon
;
ladodir: LADO_DIR NUM
| LADO_DIR '+' NUM
| LADO_DIR '-' NUM
| LADO_DIR PAR indmis
| LADO_DIR '+' PAR indmis
| LADO_DIR '-' PAR indmis
| LADO_DIR PAR indl
| LADO_DIR '+' PAR indl
| LADO_DIR '-' PAR indl
;
indmis:
| indmis indl
| indmis indn
;
indl: IND
| indl IND
;
indn: IND_NUM
| indn IND_NUM
;
```