UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

PROBLEMA DE COBERTURA DE CONJUNTOS – UMA COMPARAÇÃO NUMÉRICA DE ALGORITMOS HEURÍSTICOS.

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA, PARA A OBTENÇÃO DO GRAU DE "MESTRE EM ENGENHARIA"

NILOMAR VIEIRA DE OLIVEIRA

FLORIANÓPOLIS
SANTA CATARINA – BRASIL
1999

PROBLEMA DE COBERTURA DE CONJUNTOS – UMA COMPARAÇÃO NUMÉRICA DE ALGORITMOS HEURÍSTICOS.

NILOMAR VIEIRA DE OLIVEIRA

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA A OBTENÇÃO DO TÍTULO DE "**MESTRE EM ENGENHARIA**", ESPECIALIZADA EM ENGENHARIA DE PRODUÇÃO E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO.

Prof. Ricardo Miranda Barcia, Ph.D.
COORDENADOR

BANCA EXAMINADORA:

Sérgio Fernando Mayerle, Dr.
ORIENTADOR

Prof.a. Mirian Gonçalves Buss, Dra.

Prof. Antônio Sérgio Coelho, Dr.

Dedico este trabalho a meus pais
Clarício Pinto de Oliveira e
Doralice Vieira de Oliveira.

RESUMO

Neste trabalho faz-se um estudo comparativo de algoritmos heurísticos para a resolução do Problema de Cobertura de Conjuntos (PCC), que é um problema de Programação Linear Inteira, com diversas aplicações práticas, comprovadamente NP-Difícil. O estudo foi conduzido sobre algoritmos heurísticos que utilizaram um conjunto de dados padrão. Foi feito uma revisão bibliográfica do PCC, considerando-se os métodos atuais mais eficientes na resolução do PCC, conforme a literatura: Relaxação Lagrangeana e otimização por subgradientes e Algoritmos Genéticos. São apresentadas as variações de Heurísticas Gulosas para PCC mais utilizadas. Adicionalmente ao estudo comparativo sobre o conjunto de dados padrão, gerou-se um conjunto de problemas com dados diferentes. Verificou-se, com isto, que os algoritmos considerados mais eficientes para a solução do PCC são muito sensíveis a mudanças nos conjuntos de dados do problema. Baseado nestes resultados foram feitas sugestões para aprimorar estes algoritmos.

ABSTRACT

The aim of this work is a comparative study of heuristics algorithms to Set Covering Problem (SCP). This problem is an *Integer Linear Programming* problem with several practical applications, proved to be NP-hard. The study was conducted on heuristics algorithms that were applied on a standard data set. It was made a bibliographic review to the SCP, considering the most recent effective methods, according to literature: *Lagrangian relaxation, subgradient optimization* and *Genetic Algorithms*. It is presented the most used *Greedy Heuristics* variations to the SCP. In addition to the comparative study on the standard data set, it was generated a different data set problems and it was verified that algorithms considered the most efficient to solution to the SCP are very sensitive to changes in the data set problem. Suggestions are done to improve these algorithms.

AGRADECIMENTOS

Ao Prof. Mayerle pela paciência, solicitude e idéias que serviram de orientação em toda etapa desta dissertação.

Aos amigos Nilson Rodrigues Barreiros, João Caldas do Lago Neto, Marco Antônio Pereira Rodrigues e Francisco Paiva Canidé pelas contribuições para a realização deste trabalho.

A todos membros do Departamento de Matemática da Universidade Federal do Amazonas que, de alguma forma, viabilizaram este trabalho.

ÍNDICE

LISTA DE FIGURAS

LISTAS DE QUADROS

CAPÍTULO I

1. INTRODUÇÃO

1.1. Considerações Gerais

A modelagem por Programação Inteira (PI) é adequada para representação de uma grande classe de problemas de otimização discreta. A maior barreira enfrentada na solução de muitos problemas modelados por PI e também do mundo real (como por exemplo o planejamento de escalas de tripulação) é a chamada "explosão combinatorial", pois a maioria destes problemas pertencem a classe chamados de Não Polinomial (NP). (Vide [GAR79]) .

Dentre essas classes de problemas PI existe um caso particular que tem despertado especial interesse: o Problema de Cobertura de Conjuntos (PCC), devido a enorme quantidade de problemas teóricos e práticos, e cujo aumento no tamanho do problema implica em sérias complicações computacionais, tornando imprescindível o desenvolvimento de algoritmos eficientes para a sua resolução.

Estes problemas podem ser resolvidos por métodos convencionais de programação inteira 0-1. A experiência de vários autores indica que são, em geral, mais fáceis de resolver do que outros problemas de programação inteira 0-1. Apesar disto, os tipos de PCC que surgem na prática costumam ser com muitas variáveis e restrições, criando a necessidade de algoritmos que tirem vantagens dessas particularidades.

Atualmente, existem várias formas de atacar problemas deste tipo. No entanto, não existem estudos comparativos e sistematizados dos métodos atualmente mais utilizados. Os poucos

trabalhos que existem neste sentido o fazem considerando um conjunto de dados para teste com características limitadas, e não oferecem uma base para conclusão do desempenho dos algoritmos em situações diversas. Portanto, seria excelente se os algoritmos aproximados conhecidos como os de melhor performance, pudessem ser avaliados também, em situações em que se variasse algumas das características do conjunto analisado.

1.2. Objetivos

Pretende-se com este trabalho, apresentar uma revisão bibliográfica do PCC, enfocando o desenrolar histórico de alguns métodos utilizados no desenvolvimento de algoritmos, unificando a nomenclatura e notação, e que foram reportados na literatura como mais eficientes.

Como objetivo principal, pretende-se fazer a comparação dos diversos algoritmos aproximados e exatos para o PCC, pela compilação direta de resultados observados em trabalhos publicados, considerando o conjunto de testes da *OR Library*. As peculiaridades desses testes, instigam a necessidade de perturbação de alguns parâmetros dessas instâncias de problemas, na tentativa de inferir se ainda apresentarão boa performance computacional. Em particular, esta investigação se dará sobre dois dos mais eficientes algoritmos heurísticos reportados na literatura. Dos resultados destas análises, serão sugeridas, finalmente, algumas modificações nestes algoritmos que podem ajudar a melhorar suas performances, tanto numérica quando em tempo de processamento.

1.3. Importância do Trabalho

O PCC e sua variação mais próxima, o Problema de Particionamento de Conjuntos (PPC), a ser definido no capítulo 2, é um problema com muitas aplicações práticas, dentre as quais se pode destacar: geração de escalas de tripulação (ônibus, aérea), problema de Steiner, localização de facilidades, entre outros.

Karp (Vide [KAR72]) demonstrou que o problema de cobertura de conjuntos é um problema NP-difícil, significando que para grandes instâncias é pouco provável de ser

resolvido através de algoritmos que requeiram um número de passos limitados por uma função polinomial. Este fato, geralmente, já é suficiente para justificar o estudo e o desenvolvimento de algoritmos heurísticos eficientes.

Mesmo nos artigos que apresentam algoritmos relativamente eficientes para resolução do PCC, não existe uma descrição mais detalhada que possa servir de orientação para o desenvolvimento de novos algoritmos. Através da bibliografia pesquisada constatou-se que não existe, ou pelo menos não se conhece até o presente, um trabalho mais abrangente na tentativa de comparação dos métodos que se mostraram mais eficientes. Esta comparação deverá ser feita com instâncias de problemas que apresentarem características distintas daqueles usualmente utilizados e que se encontram na *OR Library*.

1.4. Limitações

Em uma avaliação inicial, neste trabalho, devido a dificuldade de análise de algoritmos sobre conjuntos de testes distintos, serão considerados apenas os algoritmos heurísticos e exatos mais recentes e eficientes na resolução do PCC, testados sobre o conjunto de dados da *OR Library de Beasley* (Veja [BEA90a]).

1.5. Estrutura do Trabalho

No capítulo seguinte apresenta-se, além de uma Revisão Histórica do Problema, alguns resultados importantes para o desenvolvimento de algoritmos para o PCC, a saber: as Reduções mais comuns para o PCC; os Métodos Heurísticos de Busca para o PCC; as Relaxações Lagrangeanas e Otimização por Subgradientes de uma forma geral e, especificamente para o PCC; e uma metodologia relativamente recente: os Algoritmos Genéticos, de forma mais geral, e também as suas variações mais comuns para o PCC; resumidamente, os métodos exatos clássicos e outros que representam atualmente o estado-da-arte; além destas serão apresentados outras metodologias heurísticas que estão despontando com possibilidade de aperfeiçoamentos. Esses algoritmos aproximados mais

utilizados para o PCC podem ser aprimorados em outros algoritmos heurísticos mais eficientes ou utilizados em um método exato de *branch and bound*. No Capítulo 3, faz-se uma comparação de alguns desses algoritmos heurísticos e algumas modificações sobre o conjunto de testes para verificar quão eficientes são os algoritmos aproximados que representam o estado–da–arte . Finalmente, no capítulo 4, apresenta-se as conclusões e recomendações para estudos posteriores.

CAPÍTULO II

2. REVISÃO BIBLIOGRÁFICA BÁSICA

2.1. Considerações Iniciais

Na atualidade, os algoritmos para a resolução do PCC baseiam-se em Técnicas de Enumeração, Combinações de Técnicas Heurísticas, Relaxações Lineares, Relaxações Lagrangeanas, ou algoritmos baseados em novos paradigmas desenvolvidos recentemente como Algoritmos Genéticos, Redes Neurais Artificiais (para o caso unicusto [GRO95]) e *Simulated Annealing*.

A seguir é apresentada uma revisão histórica do Problema de Cobertura de Conjuntos. Na seção 2.3 serão consideradas algumas definições básicas que auxiliarão no entendimento do problema proposto, incluindo a notação adotada freqüentemente. Em particular, apresentam-se as duas definições equivalentes e mais comuns utilizadas nos PCC. Na seção 2.4 são estabelecidas algumas reduções que podem ser utilizadas. A partir da seção 2.5 passa-se a descrever os métodos que se mostraram mais efetivos na resolução do PCC.

2.2. Uma Revisão Histórica do Problema

A estrutura da revisão histórica que se apresenta abaixo inicia com os algoritmos exatos que se tornaram clássicos na resolução do PCC. Depois, são apresentados outros importantes estudos para o desenvolvimento de algoritmos heurísticos eficientes. Finalmente, são apresentados os algoritmos exatos e heurísticos mais atuais, testados no mesmo conjunto de dados da *OR Library* (vide [BEA90a]).

Balinski (vide [BAL65]) apresenta os métodos clássicos utilizados para resolução de *Programação Linear Inteira* (PLI), possibilitando uma visão geral dos métodos, dos usos e das experiências computacionais da época. O PCC é discutido, principalmente, como aplicação à Teoria dos Grafos, no caso *unicusto*. Também são reportados os problemas de *Matching*, e de *Particionamento* de Conjuntos, relacionados com o PCC. Através desse artigo, e na bibliografia que contém, é possível conhecer algumas das aplicações práticas mais interessantes do PCC e de outros problemas de programação inteira .

Para entendimento do estado-da-arte do PCC até meados da década de 70 , três artigos são clássicos. São eles: *Garfinkel* e *Nemhauser* (vide [GAR69]), *Lemke*, *Salkin* e *Spielberg* (vide [LEM72]), e *Cristofildes* e *Korman* (vide[CHR75]). O primeiro artigo é uma pesquisa dos diversos métodos para resolução do PCC (planos cortantes, enumeração implícita e heurísticas) e também os problemas de Particionamento e *Matching*; o segundo, trata exclusivamente da enumeração implícita; o terceiro, é uma pesquisa dos métodos para a resolução do PCC até cerca de 1975. Nele, o autor tenta estabelecer a eficiência e méritos de alguns métodos conhecidos até então, testando cinco algoritmos distintos. O primeiro e o segundo artigo apresentam as reduções fundamentais para PCC, que podem, eventualmente, possibilitar um ganho computacional. Essas reduções são apresentadas na seção 2.4.

Para o Problema de Partição de Conjuntos, *Marsten* (vide [MAR74]) apresentou um algoritmo com bons resultados em problemas de larga escala (vide [FIS90]), e uma técnica de ramificação eficiente. As características da técnica de ramificação de *Marten* foram posteriormente adotadas no algoritmo para o PCC proposto por *Etcheberry* (vide [ETC77]).

Até 1976, pelo que se sabe, nenhum algoritmo para resolução do PPC utilizou métodos diferentes dos consagrados até então: busca em árvore (enumeração implícita) ou planos cortantes (substituição de restrições). Basicamente eram métodos exatos, e quando não encontravam a solução ótima (o que era comum nos problemas de grande escala), estavam quase sempre distante dela. Devido ao método *Simplex* para resolver o problema original e os subproblemas resultantes de um *branch and bound*, os algoritmos exatos apresentavam um alto custo computacional.

Etcheberry (vide [ETC77]), pelo que se sabe, foi o primeiro a utilizar o método de otimização por *subgradientes* em PCC, para obter limites inferiores para o problema original. Embora seu algoritmo não representasse um ganho computacional significativo em comparação com os algoritmos existentes na época, foi importante o tratamento diferenciado dos demais. Pelo que se sabe, todos utilizavam o método *Simplex Primal* ou *Dual*, ou variações deles para encontrar soluções viáveis.

Chvàtal (vide [CHV79]) apresenta uma heurística gulosa para o problema de cobertura de conjuntos e faz um estudo para o pior caso de algoritmos baseados nessa heurística. Este resultado influenciou o desenvolvimento de novas idéias e algoritmos heurísticos gulosos mais eficientes, em análise de pior caso. Na seção 2.5. discute-se mais detalhadamente essa e outras heurísticas. (Vide [BAL80], [BAK81], [HO82],[VAS84])

Balas e *Ho* (vide [BAL80]) testaram várias técnicas de relaxações Lagrangeanas, por eles desenvolvidas, utilizando otimização por subgradientes. Algumas destas incluem planos cortantes, e introduziram diversas heurísticas primais e duais, combinando com técnicas de fixação de variáveis, e novas regras de ramificação. Tais estudos serviram como ponto de referência para o desenvolvimento de vários algoritmos para o PCC nos anos 80 subsequentes. Por exemplo, *Hall* e *Hochbaum* (vide [HAL83]) estenderam esta abordagem para problemas de coberturas mais gerais (lado direito maior que um). Da mesma forma, *Vasko* e *Wilson* (vide [VAS84] e [VAS86]) introduziram uma versão eficiente aleatória de heurística gulosa.

Baesley (vide [BEA87]) implantou um algoritmo exato, usando a estrutura de *Balas e Ho* (vide [BAL80]), mas resolvendo subproblemas de programação linear e usando algumas novas regras de fixação de variáveis. *Beasley* (vide [BEA90]) introduziu a heurística baseada em *Relaxação Lagrangeana* e *Otimização por Subgradientes* que gera uma cobertura após cada iteração do subgradiente. *Beasley e Jörnsten* (vide [BEA92]) acrescentam planos cortantes a abordagem de *Beasley* (vide [BEA87]), melhorando sua performance.

Jacob e Brusco (vide [JAC93]) descreveram uma heurística baseada em *Simulated Annealing* para o PCC, posteriormente aperfeiçoada (vide [JAC95] e [JAC96]). Pelo que se sabe, nesse momento começaram as tentativas em buscas de novas abordagens de *Meta-Heurísticas* para resolução de problemas de cobertura de conjuntos.

Lorena e Lopes ([vide LOR94]) apresentam um algoritmo para PCC baseado em *Relaxação Surrogate*, explicado sucintamente na seção 2.8.2. *Thompsom e Harshe* (vide [THO94]) descreveram um método exato para resolução dos Problema de Cobertura de Conjuntos, Empacotamento e Partição de Conjuntos Ponderados, baseado no Método Clássico de Subtração de Colunas. Esta nova metodologia tem o mérito de evitar o degenerência massiva típica destes problemas.

Grossman e Wool (vide [GRO95]) desenvolveram um estudo comparativo de algoritmos aproximados para o PCC, no caso unicusto, utilizando *Redes Neurais Artificiais* (RNA). Embora não tenha apresentado resultados comparáveis aos do conjunto de teste que se tornou clássico para o PCC, por ser unicusto, mostrou-se bastante robusto trabalhando com os testes gerados. No entanto, para um conjunto de testes de aplicações práticas, esse algoritmo não obteve boa performance.

Caprara, Fischetti e Toth (vide [CAP95]) desenvolveram um algoritmo para PCC que foi utilizado na resolução de um problema real, numa ferrovia na Itália, e quando aplicado ao conjunto de testes da *OR Library*, mostrou-se amplamente favorável se comparado com alguns algoritmos existentes. A comparação desse algoritmo com alguns outros pode ser encontrada no Capítulo 3.

Beasley e Chu (vide [BEA96]) utilizaram *Algoritmos Genéticos* (AG) para o Problema de Cobertura de Conjuntos; neste mesmo ano Al-Sultan *et al.* (vide [ALS96]) também os utilizaram. Balas e Carrera (vide [BAL96]) apresentaram um algoritmo *branch and bound* para o PCC, cujo ponto central era um procedimento que integrava limite superior e limite inferior chamado por eles de "otimização do subgradiente dinâmico". Este novo procedimento, aplicado ao Dual Lagrangeano em todo nó da árvore de busca, combinava o

método do subgradiente padrão com heurísticas primais e duais que interagiam para mudar os multiplicadores de Lagrange, ajustar os limites superiores e inferiores , e fixar variáveis. Este método obteve performance muito boa em comparação com outros algoritmos heurísticos e exatos existentes; e juntamente com algoritmo de ramificação e avaliação sucessiva avançou bastante o estado da arte na resolução do problema de cobertura de conjuntos.

Lorena e Lopes (vide [LOR97]) utilizaram um Algoritmo Genético aplicado a Problemas de Cobertura de Conjuntos computacionalmente difíceis, cujos testes em matrizes de incidência de Steiner, obteve as soluções ótimas conhecidas e os melhores resultados para alguns casos que não se conhece os valores ótimos.

2.3. Definição

O *Problema de Cobertura de Conjuntos (PCC)* é um problema de programação inteira 0-1, definido matematicamente como:

$$\text{(PCC)} \qquad \textit{Minimizar} \sum_{j=1}^{n} c_j x_j \qquad (1)$$

$$\textit{Sujeito a} \sum_{j=1}^{n} a_{ij} x_j \geq 1, \quad i = 1,..., m \qquad (2)$$

$$x_j \in \{0, 1\}, \quad j = 1, ..., n \qquad (3)$$

Nesta formulação, $x_j = 1$ se a coluna j está na solução e $x_j = 0$, em caso contrário. Se todos os c_j são iguais, o problema é chamado de *Problema de Cobertura Unicusto de Conjuntos.* A inequação (2) encerra que cada linha da matriz (a_{ij}) é coberta por ao menos uma coluna e a (3) é a restrição de integridade. Se a restrição (2) for substituída por:

$$\sum_{j=1}^{n} a_{ij} x_j = 1, \qquad i = 1,..., m \qquad (2')$$

o problema de programação inteira zero-um resultante será chamado de *Problema de Partição de Conjuntos (PPC).*

De uma forma abreviada, numa notação matricial, o Problema de Cobertura de Conjuntos (PCC) e Particionamento de Conjuntos (PPC) podem ser formulados como abaixo, onde $A = (a_{ij})$ é uma matriz $m \times n$ 0-1, e um vetor coluna de m *1's*, e c um vetor linha $1 \times n$ de pesos inteiros positivos.

$$\begin{array}{lll} \text{(PCC)} & \textit{Minimizar} & cx \\ & \textit{Sujeito a} & Ax \geq e \\ & & x \in \{0,1\}^n \end{array}$$

e o *Problema de Particionamento de Conjuntos (PPC)* é definido como

$$\begin{array}{lll} \text{(PPC)} & \textit{Minimizar} & cx \\ & \textit{Sujeito a} & Ax = e \\ & & x \in \{0,1\}^n. \end{array}$$

Dessa forma, o PCC é o problema equivalente a cobertura das m-linhas da matriz $A = (a_{ij})$, por um subconjunto de suas n-colunas com custo mínimo, onde c_j (> 0) é o custo associado a coluna j. No caso do PPC, o interesse está na união menos onerosa de conjuntos disjuntos (colunas) da matriz $A = (a_{ij})$ que a cobrem.

Desta maneira, considera-se neste trabalho $M = \{1,...,m\}$ e $N = \{1,...,n\}$, como o conjunto das linhas e das colunas, respectivamente, da matriz $A = (a_{ij})$. Diz-se que uma coluna $j \in N$ cobre a linha $i \in M$ se $a_{ij} = 1$. Assim, a solução do PCC busca por um subconjunto $S \subseteq N$ de colunas com um custo mínimo, de forma que cada linha $i \in M$ seja coberta, ao menos, por uma coluna $j \in S$. Assim, $x_j = 1$ se $j \in S$, e $x_j = 0$, em caso contrário.

Por conveniência de notação, para cada linha $i \in M$ seja $N_i = \{j \in N;\ a_{ij} = 1\}$. Analogamente, para cada coluna $j \in N$ seja $M_j = \{i \in M;\ a_{ij} = 1\}$. Além disso, $q = \sum_{i \in M} \sum_{j \in N} a_{ij}$ denotará o número de entradas não-nulas da matriz A, que geralmente é muito menor que o número mn.

2.4. Reduções

Devido às características próprias do PCC é possível fazer algumas reduções bem conhecidas. Outras reduções podem ser encontradas, por exemplo, nas referências [GAR69], [LEM72]) e [CHR75a].

(a) Não viabilidade

Se para alguma linha $i^* \in \boldsymbol{M}$, $i^* \notin M_j$, para todo $j \in \boldsymbol{N}$, então a linha i^* não pode ser coberta e o problema não tem solução.

(b) Coluna nula

Se para alguma coluna $j^* \in \boldsymbol{N}$ tem-se $M_{j^*} = \phi$ então ela deve ser eliminada do problema, visto que a coluna j^* não cobre nenhuma linha.

(c) Inclusão de coluna

Se existe alguma linha $i^* \in \boldsymbol{M}$ tal que $i^* \in M_{j^*}$ e $i^* \notin M_k$, para todo $k \neq j^*$, então j^* deve estar em todas as soluções, e as linhas cobertas por M_{j^*} podem ser eliminadas. O problema deve ser reduzido fazendo-se $\boldsymbol{M} = \boldsymbol{M} - M_{j^*}$ e, considerando-se o conjunto $\boldsymbol{N}$ de todas as colunas como $\boldsymbol{N} = \boldsymbol{N} - \{j^*\}$.

(d) Domínio de Linha

Considere N_i como definido na seção anterior. Se existirem linhas $i^*, r^* \in \boldsymbol{M}$ com $N_{i^*} \subseteq N_{r^*}$ então a linha r^* deve ser eliminada, uma vez que a restrição i^* é mais forte que a r^*. Diz-se, neste caso, que a linha r^* *é dominada* pela linha i^*.

(e) Domínio de Coluna

Considere M_j como definido na seção anterior. Se existirem colunas $p^*, q^* \in \boldsymbol{N}$ com $M_{p^*} \subseteq M_{q^*}$ e $c_{p^*} \leq c_{q^*}$, então a coluna q^* deve ser eliminada, uma vez que o conjunto M_{p^*} que for coberto pela coluna p^* também cobrirá o conjunto M_{q^*}, com um custo menor ou igual. Diz-se, neste caso, que a coluna q^* é *dominada* pela coluna p^*.

(f) Reunião de Colunas menos Onerosa em Comparação com uma dada Coluna

Se, para algum $S^* \subset N$ tem-se $\bigcup_{j^* \in S^*} M_{j^*} \supseteq M_{k^*}$, com $\sum_{j^* \in S^*} c_{j^*} \leq c_{k^*}$ e qualquer coluna $k^* \in N - S^*$, então a coluna k^* pode ser eliminada porque é dominada pela reunião $\bigcup_{j^* \in S^*} M_{j^*}$.

2.5. Método Heurístico de Construção

Vários métodos heurísticos foram utilizados para encontrar soluções viáveis para o PCC. As heurísticas que tentam encontrar uma solução viável para o PCC no problema primal são chamadas de heurísticas primais. Aquelas que tentam encontrar soluções duais viáveis para o PCC são chamadas de heurísticas duais. Na seção 2.5.5. descreve-se o método genérico para construção de heurísticas primais eficientes. Antes, porém, descreve-se abaixo as idéias originais dos algoritmos gulosos, que são os métodos mais utilizados de construção de soluções viáveis primais próximas a solução ótima.

2.5.1. A Heurística Gulosa de Chvàtal (vide [CHV79])

Esta *heurística* é baseada no fato de que aumentam as chances de uma determinada coluna j^* estar na solução quando a razão entre o custo da referida coluna j^* pela cardinalidade do conjunto M_{j^*} é menor em relação a outras colunas $j \in N$.

Assim, para uma dada iteração t,

Se $\frac{c_{j^*}(t)}{|M_{j^*}(t)|} < \frac{c_j(t)}{|M_j(t)|}$, com $|M_{j^*(t)}|, |M_{j(t)}| \neq 0, \forall j \in N - \{j^*\}$ então $j^* \in S^*$, onde $|.|$ denota a cardinalidade dos conjuntos $M_{j^*(t^*)}$ e $M_{j(t^*)}$, e S^* a conjunto das colunas que podem estar na solução ótima.

O pseudocódigo para o Algoritmo Guloso de Chvàtal está descrito abaixo:

Leia: c, M, N, M_j, N_i *(como definidos na seção 2.3)*;

Inicialize: $R := M$ e $S^* = \phi$ (onde R é conjunto das linhas não cobertas e S^* é o conjunto das colunas na solução) ;

Em cada iteração t

Escolha j^* tal que

$$f(c_{j^*}, k_{j^*}) = min\ \{f(c_j, k_j) \mid k_j > 0, \forall j \in N\}, \text{ onde } k_j = |M_j \cap R| \text{ e } f(c_j, k_j) = \frac{c_j}{k_j};$$

Faça $R := R - M_{j^*}$;

$S^* = S^* \cup \{j^*\}$;

Repita até $R = \phi$;

Escreva S^* .

Uma das deficiências do algoritmo heurístico guloso de Chvàtal (como de outras heurísticas) é a não garantia de uma cobertura primal não redundante, significando que ao final pode-se ter soluções que não são mínimas. Portanto, na utilização destes algoritmos, deve-se ao final retirar as colunas redundantes.

2.5.2. A Heurística de Balas e Ho (vide [BAL80])

Uma outra forma prática para melhorar o resultado da heurística de Chvàtal descrita acima é utilizar alternadamente diversas funções de avaliação $f(c_j, k_j)$, ao invés de uma única. *Balas e Ho* consideram as cinco funções seguintes :

(1) c_j ;

(2) $\frac{c_j}{k_j}$;

(3) $\frac{c_j}{log_2 k_j}$;

(4) $\frac{c_j}{k_j log_2 k_j}$;

(5) $\frac{c_j}{k_j \ln k_j}$;

Nos casos (3) e (4) $\log_2 k_j$ deve ser substituído por 1 quando $k_j = 1$ e no caso (5), $\ln k_j$ deve ser substituído por 1 quando $k_j = 1$ ou 2.

A função (1) inclui na cobertura S^* a cada iteração, a coluna de menor custo. A função (2) minimiza o custo unitário de cobertura de uma linha não coberta (note que se tem, neste caso, a Heurística Gulosa de *Chvàtal* descrita acima) . As funções (3) , (4) e (5) selecionam a mesma coluna que (2) sempre que $c_j = 1$ ($j = 1,...n$). Caso contrário, (3) assinala menos peso ao número k_j de linhas cobertas , enquanto (5) atribui mais e (4) mais ainda em relação ao custo c_j quando comparadas com (2). Os testes de *Balas e Ho* mostraram que nenhuma das funções é significativamente melhor que outra. A melhor solução encontrada por qualquer das cinco funções não desviou o valor ótimo mais que 10,8%.

A heurística de Balas e Ho (vide [BAL80]) ainda inclui ao algoritmo guloso um passo adicional ao final do algoritmo, para remover as colunas redundantes da cobertura, como segue:

$Z = Z + c_j$, onde Z é o custo total da cobertura.
Ordenar a cobertura de forma que
$S = \{j_1, j_2, ...j_t\}$ e $c_{j1} \geq c_{j2} \geq .....\geq c_{jt}$;
Para $j = 1$ **até** t **fazer**
  Se $S^* - \{j\}$ é cobertura **então**
    $S^* := S^* - \{j\}$
  Caso contrário $Z := Z + c_j$;
  Escrever $\{S$ e $Z\}$;

2.5.3. A Heurística de Vasko e Wilson (vide [VAS84])

Vasko e Wilson utilizam na heurística gulosa que chamaram de SCHEURI, além das 5 funções acima descritas, utilizadas por Balas e Ho (vide [BAL80]), duas outras *funções de avaliação* heurísticas, que valorizam ainda mais o número de linhas cobertas :

(6) $\frac{c_j}{k_j^2}$

(7) $\frac{c_j^{1/2}}{k_j^2}$

Vasko e Wilson, além da remoção das colunas redundantes da solução viável S^*, também acrescentam um passo adicional ao algoritmo básico, consistindo em realizar uma busca 1-opt numa vizinhança de S^* (vide [ROT69]) , trocando uma coluna de S^* com outra não pertencente a S^*. Se uma solução viável vizinha com menor custo for encontrada, a permutação é realizada sobre o conjunto S^*.

Como *Ho* [HOA82] demonstrou que essas funções não melhoram a performance da heurística primal no *pior caso*, quando utilizadas separadamente, *Vasko e Wilson* modificaram SCHEURI e chamaram de SCFUNC1TO7, de forma que todas as sete funções pudessem gerar uma única solução para o PCC, da seguinte maneira: a função $f(c_j,k_j)$ é determinada aleatoriamente toda vez que uma coluna entra na solução S^*. Isto é, toda vez que uma coluna for selecionada para entrar na solução, é gerado um número aleatório de 1 a 7, correspondente às funções de avaliação de Balas e Ho (vide [BAL80]) e Vasko e Wilson (vide [VAS84]), e usado o número da dessa função na ocasião da chamada da função de avaliação no algoritmo SCHEURI.

Segundo Vasko e Wilson, a melhor solução obtida por SCFUNC1T07 foi superior ou igual que a melhor de qualquer uma das cinco soluções das funções de (1) a (5) usadas isoladamente no algoritmo SCHEURI. A melhor solução encontrada por SCFUNC1TO7 foi estritamente melhor em 50% dos problemas testados.

2.5.4. Outras Heurísticas Gulosas

Ainda nessa mesma linhagem de heurísticas gulosas primais que exploram diferentes funções de avaliações, Baker (vide [BAK81]) propôs a obtenção de diversas soluções viáveis através da aplicação da heurística primal com funções diferentes, combinando as heurísticas na tentativa de melhorar o valor da solução.

A Heurística dual de Fisher e Kedia (vide [FIS90]) é semelhante a regra heurística gulosa de *Chvàtal* (vide [CHV79]), exceto pelo critério de seleção da coluna j, em que c_j foi substituído por $c_j - \sum_{i \in M_j} u_i$, chamado *de Custo Lagrangeano*, onde u é a solução dual determinada pela heurística dual. Esta mudança pretende capturar melhores impactos das restrições na seleção de variáveis, e tem como resultado uma melhora no resultado da performance computacional.

2.5.5. Estratégias Heurísticas para o PCC

As heurísticas gulosas apresentadas nas seções acima se caracterizam por obterem soluções com tempos computacionais consideravelmente pequenos e poucas iterações do algoritmo guloso apresentado por Chvàtal. As soluções encontradas são, no entanto, de qualidade regular, se comparadas com heurísticas de programação matemática mais sofisticada.

Conforme os algoritmos apresentados acima, os métodos heurísticos fornecem "boas" soluções viáveis ao PCC, mas sem garantia de otimalidade. Muitas heurísticas para o PCC são por natureza construtivas. Primeiramente, nenhuma coluna é representada no vetor solução, isto é, $x_j = 0$ $(j = 1,...n)$. Iterativamente, baseado em algum critério, escolhe-se um índice r e faz-se $x_r = 1$. Esse critério é usado repetidamente até que uma solução viável seja encontrada. Em seguida, a solução pode ser melhorada removendo-se todas as colunas redundantes. Uma busca por vizinhança também pode ser utilizada para obter um ótimo local.

2.6. Relaxação Lagrangeana

2.6.1. Conceito e Evolução

Uma das idéias computacionais mais úteis dos anos 70 é a observação que muitos problemas difíceis podem ser visto como problemas fáceis, complicados por um número relativamente pequeno de restrições. A dualização das restrições difíceis, isto é, o acréscimo destas à função objetivo através de um vetor de multiplicadores, chamados de *multiplicadores de Lagrange*, e eliminadas em seguida do conjunto de restrições deve produzir um problema Lagrangeano que é fácil de resolver e cujo valor da solução ótima é um limite inferior (para problemas de minimização) para o valor ótimo do problema original. O *Problema Lagrangeano* pode, portanto , ser usado no lugar de um problema de *Relaxação Linear* para produzir limites num algoritmo de busca do tipo *branch and bound.* Além disso, com base nesse limite inferior, é possível estimar quão próxima está a solução viável disponível da solução ótima. Pode-se verificar que a abordagem da relaxação Lagrangeana oferece um número de vantagens importantes sobre a programação linear (Vide [FIS81].

Embora tenham existido uma série de incursões anteriores aos anos 70 no uso dos métodos de Relaxação Lagrangeana, tanto em problemas teóricos quanto em práticos, os trabalhos de Held e Karp (vide [HEL70] e [HEL71]) são considerados marcos fundamentais. No entanto, o nome "Relaxação Lagrangeana" foi cunhado definitivamente por Geoffrion (vide [GEO74]).

2.6.2. Definição do Problema Primal

Considere um problema de otimização combinatorial formulado como programação inteira da seguinte forma :

(P) *Minimizar* $f(x)$

Sujeito a $g_i(x) \leq 0 \quad (i = 1,...,m), \qquad (1)$

$x \in S, \qquad (2)$

$x = (x_1, x_2,..., x_n)^T \in Z_+^n \qquad (3)$

onde $f(x)$ e $g_i(x)$ $(i = 1,...,m)$ são funções arbitrárias, S é um *conjunto discreto finito* $S = \{x^{(1)},...,x^{(r)}\}$ e $x^{(k)}$ $(k = 1,\ldots,r)$ é uma solução de (P).

S é chamado *o conjunto das soluções* do problema. Uma *solução viável* $x \in S$ é uma solução que também satisfaz as restrições (1), (2) e (3).

Muitos problemas combinatoriais difíceis de resolver encaixam-se na estrutura do problema (P).

A cada restrição $g_i(x) \leq 0$, associa-se um número real $u_i \geq 0$, chamado *Multiplicador de Lagrange* e define-se a *função Lagrangeana* como sendo

$F(x, u) = f(x) + u\, g(x)$,

onde $u = (u_1,..., u_m)$ e $g(x) = [g_1(x),...,g_m(x)]^T$.

Como o problema de programação inteira pode ser colocado de várias maneiras na forma de uma função Lagrangeana (vide [FIS81]), deve-se assumir que a *função Lagrangeana* acima satisfaz a condição de que para todo $u \in \Re_+^m$ existe um bom algoritmo para computar $L(u) = \underset{x \in S}{Min} \{ F(x,u) \}$.

2.6.3. Definição do Problema Dual e suas Propriedades

Por definição, o problema dual de *(P)* é :

$$\underset{u \in \Re_+^m}{Max} \; \underset{x \in S}{Min} \; \{ F(x,u) \} \qquad (D)$$

ou também pela introdução da função $L(u) = \underset{x \in S}{Min} \; \{ F(x,u) \}$, chamada de *função dual,*

$$\underset{u \in \Re_+^m}{Max} \; L(u) \qquad (D).$$

A função dual de um problema de programação inteira tem um número de importantes propriedades:

Propriedade 1 - Limite inferior

Para qualquer $u \in \Re_+^m$, e para todas as soluções viáveis x de (P), tem-se

$$L(u) \leq f(x).$$

Em particular, $L(u) \leq f(x^*)$, onde x^* é um ótimo global de (P).

Demonstração:

Por definição, $L(u) \leq f(x) + \sum_{i=1}^{m} u_i g_i$, $\forall u \geq 0$, $\forall x \in S$. Como $u_i \geq 0$ e $g_i \leq 0$, para todas as soluções viáveis x de (P), temos $L(u) \leq f(x)$.

Em particular, para x^*, um mínimo absoluto de (P), e se u^* é um ótimo de (D),

$$L(u) \leq L(u^*) \leq f(x^*).$$

Uma conseqüência imediata da Propriedade 1 é a seguinte: se o dual (D) tem um ótimo ilimitado (∞), então o primal (P) não tem solução.

Definição 1

O vetor $h = (h_1, \ldots, h_m)^T$ é um subgradiente de L em u se, para todo $u^* \in \Re^n$, temos

$L(u^*) \leq L(u) + (u^* - u)h$.

Definição 2

O conjunto dos subgradientes de L em u é um conjunto convexo denotado por $\partial L(u)$ e chamado de *subdiferencial* de L em u.

De fato, segue da definição acima que se h^1 e h^2 são dois subgradientes em u, então $\theta h^1 + (1 - \theta) h^2$ é também um subgradiente, para todo $\theta \in [0,1]$.

Propriedade 2 – Subgradientes

Para $u \in \Re_+^m$, seja $H(u) = \{x^{(k)} \in S \mid f(x^{(k)}) + u\, g(x^{(k)}) = L(u)\}$.

Então

$\forall\, x^{(k)} \in H(u)$, $g(x^{(k)}) \in \partial L(u)$. Em palavras: $g(x^{(k)})$ é um *subgradiente* de L em u.

Demonstração:

Por definição de L, para todo u^* em $\Re_+^m$, tem-se

$$L(u^*) \leq f(x) + u^* g(x)$$

Em particular, para $x^{(k)} \in H(u)$,

$$L(u^*) \leq f(x^{(k)}) + u^* g(x^{(k)})$$

e por definição de $H(u)$,

$$L(u) = f(x^{(k)}) + u\, g(x^{(k)}).$$

Segue por subtração que

$$L(u^*) - L(u) \leq (u^* - u)\, g(x^{(k)}), \qquad \forall\, u^* \in \Re_+^m.$$

Portanto, tem-se $g(x^{(k)}) \in \partial L(u)$.

Além dessas duas propriedades, é possível provar que $L(u)$ é uma função linear por partes e côncava (vide [FIS81]), o que facilita a aplicação de um método adaptado do método gradientes para resolver o problema (D).

2.6.4. Otimização do Problema Lagrangeano por Subgradientes

A função $L(u)$ é côncava, de forma que todo ótimo local é também um ótimo global. Como esta função não é diferenciável em toda parte, os métodos clássicos de programação não-linear (método do gradiente, método do gradiente conjugado, etc.) não podem ser aplicados a ela. Contudo, pode-se tentar generalizar os métodos clássicos dos gradientes tomando, em cada passo, a direção do gradiente, se a função é diferenciável no ponto u corrente ou qualquer subgradiente $g(x) \in \partial L(u)$ se L não é diferenciável em u.

Tal método tem ao menos o mérito de ser simples, porque a computação de $L(u)$, em tal ponto, fornece como conseqüência um subgradiente de L em u. Além disso, o método é justificado pelo fato que $h(u)$ é a direção de decréscimo estrito da função $d(u)$, a distância de u do conjunto Q dos pontos ótimos
$Q = \{u \in \Re^m \mid L(u) = L(u^*)\}$. (vide [HEL71])

No entanto, somente o movimento na direção de $h(u)$ não assegura um aumento na função L por si só. Por esta razão, não é útil tentar otimizar L na direção de $h(u)$ simplesmente, e devem ser dados valores pré-determinados de *tamanhos de passos* t_j.

Pode ser mostrado (vide [POL67]) que a seqüência $L(u^j)$ converge para o valor ótimo $L(u^*)$ com as simples condições de que

$$t_j \to 0 \ (j \to \infty), \text{ e } \sum_{j=0}^{\infty} t_j \to \infty,$$

mas nada pode ser dito sobre a velocidade de convergência. Por outro lado, a seqüência de valores $L(u)$ obtidas é, geralmente, crescente não monitonicamente.

Se em toda iteração j, escolhe-se t_j de acordo com a fórmula abaixo

$$t_j = p_j \frac{L(u^*) - L(u^j)}{\left\| h(u^j) \right\|^2} ,$$

onde o coeficiente de relaxação p_j é sujeito a $a < p_j \leq 2$, $\forall j$, $a > 0$ fixo, então a convergência é geométrica (vide [POL69]).

Este resultado parece ser principalmente de interesse teórico, porque $L(u^*)$ não é conhecido. Mas se na fórmula acima, troca-se $L(u^*)$ por um estimador inferior $\underline{L} < L(u^*)$, então pode ser mostrado (vide [POL69]) que ou a seqüência $L(u_j)$ converge para $\underline{L}$, ou se obtém (depois de um número finito de passos) um ponto u_j para o qual vale a desigualdade $\underline{L} \leq L(u_j) \leq L(u^*)$. Isto acontece, em particular, quando $p_j = 2$, $\forall j$.

Os resultados acima podem ser derivados dos trabalhos de Agmon (vide [AGM54]) e de Motzkin e Schoenberg (vide [MOT54]) sobre a solução de sistema de inequações lineares.

Em vista da grande quantidade de escolhas para o coeficiente p_j , Held, Karp e Crowder (vide [HEL74]) mostraram que $L(u^*)$ pode ser substituído na equação acima por um estimador superior $\bar{L} \geq L(u^*)$ sem essencialmente afetar a convergência do algoritmo. Neste caso, a condição $t_j \rightarrow 0$ (j $\rightarrow \infty$) faz necessário a escolha de $p_j \rightarrow 0$ ($j \rightarrow \infty$).

Na prática, freqüentemente se escolhe para $\bar{L}$ o valor de $f(x)$ que corresponde a melhor solução para o problema (P), obtida nos estágios iniciais de computação.

Para escolha de coeficiente de relaxação p_j várias estratégias podem ser adotadas. Held *et al.* (vide [HEL74]) mencionam uma experiência satisfatória, utilizando a seguinte regra: p = 2 durante $2m$ iterações (m é o número de variáveis u_i); então divida por 2 ambos p e o número de iterações até que um limite q (fixado com antecedência) de iterações for alcançada; finalmente, divida o valor de p por 2 a cada q iterações até que t_j seja

suficientemente pequeno (<ε, fixado). Este procedimento viola a condição $\sum_{j=0}^{\infty} t_j \rightarrow \infty$, sendo possível a convergência para um ponto que não pertence ao conjunto ótimo, embora tenham reportado que esse fato quase nunca tenha acontecido. Particularmente, pode-se apontar que a escolha do tamanho do passo é uma área que não é perfeitamente compreendida na atualidade.

2.6.5. O Algoritmo Básico

O Algoritmo básico para encontrar u melhorado é o seguinte:

(a) Na inicialização, começa-se de um ponto $u^0 \in \Re_+^m$ e define-se uma seqüência de números t_j tais que $t_j \rightarrow 0$ quando $\sum_{j=0}^{\infty} t_j = \infty$.

(b) Na iteração j, se está no ponto u^j.
Compute $L(u^j) = f(x^j) + u^j g(x^j) = \underset{x \in S}{Min} \{f(x) + u^j g(x)\}$;
$g(x^j)$ é um subgradiente de L em u^j.

(c) Encontre u^{j+1} através da fórmula

$$u^{j+1} = u^j + t_j g(x^j)$$

Se $u^{j+1} \notin \Re_+^m$ então projete u^{j+1} sobre $\Re_+^m$ e retorne para (b).

A principal vantagem do método de relaxação Lagrangeana com otimização do subgradiente sobre o método simplex é o seu baixo custo computacional. É experiência comum, dos usuários desta abordagem, que o número de iterações requeridas para a convergência não depende do tamanho do problema. Além disso, não são bem compreendidos os fatores dos quais, depende exatamente o método do subgradiente, além da própria formulação do problema Lagrangeano para a sua convergência (vide [BAL96]).

2.6.6. Formulação do Problema Lagrangeano Dual para o PCC

Uma das formas que se tornaram clássicas para a formulação do Problema Lagrangeano Dual para o PCC é a apresentada abaixo (adaptado de [BAL96]) :

$$\max_{u \geq 0} \; L(u) = ue + (c - uA)x$$

Sujeito a $u \in \Re_{+}^{m}$.

A solução $x_j(u)(j = 1,...,n)$ para o problema $L(u)$ pode ser obtida fazendo $x_j(u) = 1$ se $(c_j - ua_j) \leq 0$; $x_j(u) = 0$ caso contrário. A cada iteração é resolvido o Dual Lagrangeano (D) associado a L(u)

Assim, para o Problema de Cobertura de Conjunto tem-se o seguinte algoritmo utilizando o método de Relaxação Lagrangeana e otimização do subgradiente padrão:

procedimento *SUBGRADIENTE* ;
entrada : $c, N, M, N_0, f_{sup}, f_{inf}$; , K, D, E, I
c – vetor dos custos;
M – conjuntos das linhas
N – conjuntos das colunas
f_{sup} – limite superior
f_{inf} – limite inferior
u – vetor dos multiplicadores de Lagrange
p^t – parâmetro do tamanho do passo
P^t – tamanho do passo
K – número de iterações sem melhora
E – número que regula o tamanho do passo
D - número que regula a norma do gradiente
I – número que regula a quantidade de iterações (número máximo de iterações)
a_j - coluna j da matriz A

saída: f_{inf}, u_t {*melhorado*}

começo

$N^0 := N$, $t := 1$, $u^t := u$, {*inicialize*}

enquanto $f_{sup} > f_{inf}$ **faça** {*resolva o problema de minimização* $L(u^t)$ *e defina* u^{t+1}}

para $j \in N^0$ **faça** {*assinale valores para os outros* $x_j(u^t)$}

se $c_j \leq u^t a_j$ **então** $x_j(u^t) := 1$

caso contrário $x_j(u^t) := 0$

fimse

fimpara

$L(u^t) := u^t e + (c - u^t A)x(u^t)$

$f_{inf} := max\ \{f_{inf}, L(u^t)\}$

para $i \in M$ **faça** {*compute o subgradiente*}

$g_i(u_t) := 1 - \Sigma(x_j(u_t): j \in N_i)$

fimpara

$p^0 := 2$;

se f_{inf} *não mudar por K iterações* **então**

$p^t := p^t/2$ {*para a t-ésima iteração*}

fimse;

$P^t := p^t (f_{sup} - f_{inf})/\|g(u^t)\|^2$ {*compute o tamanho do passo* P^t}

se $P^t < E$ **ou** $\|g(u^t)\| < D$ **ou** $t > I$ **então**

pare

fimse

para $i \in M$ **faça** {*atualize* u_i}

$u_i^{t+1} := max\ \{0, u_i^t + P^t g_i(u^t)\}$

fimpara

fimenquanto

fim.

Simultaneamente ao procedimento do subgradiente pode-se utilizar uma das heurísticas descritas na seção anterior, ou associações delas, com o objetivo de encontrar soluções

viáveis melhoradas, ou seja, limites superiores que serão aplicados no algoritmo acima. Por sua vez, o método genérico descrito abaixo é facilmente adaptado ao PCC e outros problemas que podem ser colocados na forma (P). Surge, dessa maneira, uma heurística que tem se mostrado extremamente eficiente.

2.6.7. Variações de Estratégias de Busca

O método dos subgradientes tem a finalidade de maximizar o limite inferior obtido pelo problema relaxado, através do ajuste de multiplicadores.

Geralmente, as heurísticas de relaxação com otimização por subgradientes são caracterizadas pelos seguintes passos:

(1) utilizam uma relaxação do problema original para definir um limite inferior para o (P);
(2) tentam maximizar o limite inferior via otimização de subgradientes;
(3) produzem soluções viáveis a partir das soluções do problema relaxado; o custo da solução viável define um limite superior f_{sup} para o (P);
(4) através dos limites superior e inferior, tentam identificar as colunas e linhas que podem ser removidas do problema.

O passo (3) é fundamental para se conseguir um limite superior viável para o problema original. Nesse ponto, deve-se utilizar um método heurístico conveniente para acelerar a convergência da Relaxação Lagrangeana. Quanto ao limite inferior, pode-se modificar vários parâmetros do procedimento do subgradiente padrão para conseguir melhorá-lo.

Entre os diversos testes de parada do algoritmo, pode-se citar os critérios de parada por satisfação das seguintes condições:

(1) subgradiente nulo;
(2) $f_{sup} = f_{inf}$;
(3) $f_{sup} - f_{inf} \leq 1$;

(4) o tamanho de passo t_j é muito pequeno para que haja uma alteração significativa no valor de f_{inf} de uma iteração para outra;

(5) o valor de f_{inf} não apresenta melhorias significativas durante um certo número de iterações consecutivas;

(6) um número máximo de iterações é atingido.

Ao final da heurística, f_{sup} é o valor da melhor solução viável encontrada e f_{inf} é o valor do melhor limite inferior encontrado para o valor ótimo do problema original. A qualidade da solução final obtida pode ser avaliada pela expressão:

$$\frac{f_{sup} - f_{inf}}{f_{inf}}$$

O pseudocódigo abaixo descreve uma heurística geral utilizando o método dos subgradientes (para um caso de minimização):

1. Inicialize $f_{sup} := +\infty$; $f_{inf} := -\infty$;
2. Defina os multiplicadores iniciais, tal que $u_i > 0$, $i = 1,...,m$;
3. **Enquanto** (testes de parada = FALSO) **faça**
4. Resolva (D) e obtendo a solução $x(u)$ com custo $f_{x(u)}$;
5. Construa uma solução viável x^* para o (P) usando $x(u)$;
6. Faça $f_{x^*} := \sum_{j=1}^{n} c_j x_j^*$
7. Faça $f_{sup} := \min \{f_{sup}, f_{x^*}\}$;
8. Faça $f_{inf} := max \{f_{inf}, f_{x(u)}\}$;
9. Execute os testes de redução do problema;
10. Faça $g(u_i) := e_i - \sum_{j=1}^{n} a_{ij} x_j(u)$, $i = 1,...,m$;
11. Calcule o novo tamanho do passo $t := p \dfrac{f_{sup} - f_{inf}}{\|g(u)\|^2}$
12. Faça $u_i := max \{0, u_i + tg(u_i)\}$, $i = 1,...,m$;
13. **fim enquanto.**

2.7. Algoritmos Genéticos

2.7.1. Conceitos Básicos

A fundamentação teórica dos algoritmos genéticos (AGs) foi originalmente desenvolvida por Holland (vide [HOL75]). As idéias de AGs estão baseadas no processo evolucionário de organismos biológicos na Natureza. Durante o curso da evolução, populações naturais evoluem de acordo com os princípios de seleção natural e "sobrevivência do mais adaptado". Indivíduos que são mais bem sucedidos na adaptação ao seu ambiente terão melhor chance de sobreviver e se reproduzir, enquanto indivíduos que são menos apropriados serão eliminados. Isto significa que os genes dos indivíduos altamente apropriados se propagarão a um número crescente de indivíduos em cada geração sucessiva. A combinação de boas características de ancestrais, altamente adaptados, deverão produzir uma prole ainda mais ajustada. Desta maneira, as espécies evoluem para se tornar mais e mais bem adaptadas ao seu ambiente.

Um algoritmo genético (AG) pode ser aplicado a uma variedade de problemas de otimização combinatorial (vide [GRE85] e [VIG91]). AGs utilizam escolhas aleatórias como uma ferramenta para guiar a busca em direção a regiões do espaço com prováveis melhorias. Contudo, o modo como um AG faz uso de probabilidade lhe confere uma característica que o distingue de uma mera busca aleatória.

O *AG* simula o processo de evolução tomando uma população inicial de indivíduos e aplicando operadores genéticos em cada reprodução. Em termos de otimização, cada indivíduo da população é codificado em um *string* ou *cromossomo* que representa a solução possível para um dado problema. O *fitness* (adaptação) de um determinado indivíduo é avaliado com respeito a uma dada função objetivo. Aos indivíduos (ou soluções altamente ajustadas) são dados a oportunidade de se reproduzirem pela troca de pedaços de suas informações genéticas, num procedimento de *crossover*. Este procedimento produz uma nova "prole" de soluções (ou seja, filhos), que dividem as mesmas características tomadas de ambos pais. A *mutação* é freqüentemente aplicada após o cruzamento para alterar alguns

genes na fileira. A prole pode repor a população completa (abordagem geracional) ou repor indivíduos menos ajustados (abordagem do estado firme). Este ciclo de avaliação-seleção-reprodução é repetido até que uma solução satisfatória seja encontrada. Os passos básicos de um AG simples são mostrados ao final desta subseção.

Estabelecendo uma analogia entre o processo de evolução simulado por um AG e um processo de busca conduzido na solução de um problema de otimização de função, pode-se perceber que ambos envolvem uma busca através de um conjunto de equivalências:

ESTRUTURA ↔ SOLUÇÃO

FITNESS ↔ VALOR DA FUNÇÃO OBJETIVO

EVOLUÇÃO ↔ BUSCA

Dada uma transformação apropriada, a adaptação de uma estrutura pode ser interpretada como uma aproximação da função objetivo original.

As principais diferenças entre os AGs e os procedimentos de busca tradicionais são descritas a seguir:

(a) AGs trabalham com uma codificação do conjunto de variáveis x em vez de trabalhar com as variáveis individuais. O objetivo da busca genética é encontrar o conjunto de variáveis (estruturas) que obtenha o melhor valor possível para a função objetivo do problema. Já os métodos de otimização tradicionais trabalham diretamente com as variáveis individuais, trocando seus valores de acordo com uma regra de transição particular.

(b) AGs realizam uma busca a partir de uma população de pontos, e não um único ponto no espaço de soluções. Enquanto a maioria das técnicas seguem um mecanismo de busca por ponto, um AG mantém uma população de pontos a serem explorados. Nos métodos de busca convencionais, o processo de busca é originado a partir de um ponto único (uma busca por vizinhança) a cada iteração, utilizando alguma regra de transição. Pouco é

aprendido durante o processo de busca, mesmo que informações importantes sobre o perfil da função possam ser recuperadas a partir do espaço de busca já explorada. Um AG vai além de uma mera busca seqüencial de cada ponto (estrutura) na população. Seu funcionamento baseia-se na identificação e exploração de blocos de construção comuns de boas estruturas na população e na verificação vários picos em paralelo. Esses blocos de construção correspondem a regiões no espaço onde boas soluções são prováveis de serem encontradas.

Três operadores básicos formam o núcleo da maioria das implementações de um AG: (1) *operador de reprodução* (2) *operador de cruzamento* e (3) *operador de mutação*. Outros operadores têm sido propostos, mas são ou derivados dos operadores citados ou operadores específicos projetados para um problema particular. Os operadores atuam no sentido de promover a melhoria da qualidade global das soluções a cada geração.

A busca genética termina quando um certo critério de parada é atendido. Entre os critérios mais utilizados podemos destacar alguns, os quais implicam na satisfação das seguintes condições:

(1) um nível pré-determinado de qualidade de solução é atingido (por exemplo, adaptação média da população);
(2) a convergência é observada;
(3) um número máximo de geração é atingido.

Algoritmo Básico AG
{Algoritmo genético para otimização de uma função f}
G_t = população da t-ésima geração

início

$t = 0$;

inicializar G_t;

avaliar G_t;

enquanto não (condição de término) **faça**

$t = t + 1$;

selecione G_t de G_{t-1}; {*operador de reprodução*}

recombine G_t ; {*operadores de cruzamento e mutação*}

avalie G_t ;

fimenquanto

fim

2.7.2. Estruturas Básicas

2.7.2.1. Estrutura Básica do Cromossomo

Informalmente, um problema pode ter a forma de uma questão ou uma tarefa. Matematicamente, um *problema (busca)* é um subconjunto P de $S^* \times S^*$, onde S é um *alfabeto*[1]. O problema matemático correspondente então a um dado *string* $z \in S^*$, encontrar um *string* y tal que $(z,y) \in P$, ou decidir que tal fileira y não exista.

Aqui o *string* z é chamado de uma *instância* ou a *entrada* do problema, e y é a *solução* ou a *saída.*

Em um AG, uma solução x (estrutura) é codificada como um *string* de comprimento k ($k > 0$) sobre um conjunto alfabeto V. O espaço de estruturas é definido como o conjunto de *strings* pertencente a V^k. Logo, o comprimento do espaço de estruturas é $| V |^k$, onde $| V |$ é o número de símbolos em V. Uma escolha comum para V é o conjunto binário {0, 1}.

---

1 - Um *alfabeto* S é um conjunto finito (freqüentemente $S = \{0,1\}$). Seus elementos são chamados *símbolos* ou *letras*. Uma seqüência ordenada de símbolos de S é chamado de uma *"string" (de símbolos)* ou *uma palavra*. S^* significa a coleção de todas a s fileiras de símbolos de S.

2.7.2.2 – Função de Adaptação

A adaptação de uma estrutura é medida por uma função $\mu : S^* \rightarrow \Re^+$, onde S^* é o conjunto de todas as estruturas (isto é, V^k) e $\Re^+$ o conjunto dos números reais não negativos. Se a função f do problema de otimização subjacente é sempre positiva, então f pode ser usada diretamente como μ . Caso contrário, μ será uma transformação de f. A transformação dependerá da função objetivo original (se f é uma função de minimização ou maximização) e do mecanismo de seleção utilizado.

AGs processam populações de estruturas. Naturalmente, a estrutura de dados primária para um AG é uma população de *strings*. Uma população pode ser implementada como um vetor de N estruturas, onde cada estrutura contém o genótipo (*o cromossomo artificial representado* por um string de bits), o fenótipo (a decodificação do genótipo) e um valor de *fitness* (função objetivo), juntamente com outras informações auxiliares. O esquema que pode ser utilizado para armazenar uma população é ilustrado na tabela 2.1.

Número da estrutura	Genótipo (String de bits)	Fenótipo (Custo)	Função de Adaptação μ (x)
1	0101...0110	67	31
2	1100...1001	15	137
.	.	.	.
N	1001...0101	8	645

Tabela 2.1: Esquema para armazenamento de uma população de estruturas em um AG.

Os valores da função de adaptação, em conjunto com as similaridades entre as estruturas de uma população, são utilizados para dirigir o processo de busca.

2.7.2.3 – Operador de seleção

O operador de seleção designa a cada estrutura da população G_t uma chance de ser selecionada para permanecer na próxima população G_{t+1} proporcional à adaptação dessa estrutura. Esse operador atribui a cada estrutura uma taxa de amostragem $T(s, t)$, definida como o número esperado de descendentes a serem gerados a partir dessa estrutura na geração t. Para coincidir com a teoria da evolução, dadas duas estruturas s' e s'', se $\mu(s') > \mu(s'')$, então $T(s', t) > T(s'', t)$. A definição mais adotada para a taxa de amostragem é $T(s, t) = \mu(s) / \overline{\mu}(s)$, onde o numerador é a adaptação de s e o denominador é a adaptação média da população G_t. Naturalmente, estruturas com adaptação acima da média possuem uma maior probabilidade de sobrevivência do que aquelas com adaptação abaixo da média.

2.7.2.4. Operador de *Crossover*

Se apenas o operador de reprodução atuar, a população tenderá a se tornar mais homogênea a cada geração. O operador de cruzamento é incluído em um AG por dois motivos: primeiro, ele introduz novas estruturas recombinando estruturas já existentes; segundo, ele tem um efeito de seleção, eliminando os esquemas de baixa adaptação.

Um dos operadores de cruzamento mais simples, o operador de cruzamento por um único ponto, que opera da seguinte forma: dadas duas estruturas s' e s'', elas trocam um substring de acordo com um ponto de cruzamento para formar duas novas estruturas. Supondo as estruturas s' e s'' como sendo 011|11 e 101|00 respectivamente, com um ponto de cruzamento indicado por |. Trocando os substrings à direita do ponto de cruzamento, duas novas estruturas 01100 e 10111 são criadas.

O operador de cruzamento por dois pontos também é muito utilizado na literatura e promove a troca dos substrings compreendidos entre dois pontos escolhidos aleatoriamente. O operador pode ser melhor ilustrado pela figura 2.1:

Fig. 2.1 : Operador de cruzamento por dois pontos

Nem todas as estruturas em uma nova população são geradas pelo operador de cruzamento.

2.7.2.5. Operador de Mutação

Uma das maneiras mais utilizadas para o operador de mutação é a introdução de mudanças aleatórias às estruturas em uma população trocando um símbolo em uma estrutura com uma probabilidade (ou taxa de mutação) p_m. Por exemplo, se p_m é 0.01, então a cada geração existe uma chance de 1% que uma estrutura da geração seja alterada pela troca de um de seus símbolos. Esse operador possui o efeito de aumentar a diversidade da população, ou seja, evita que as estruturas tornem-se muito homogêneas. O aumento nas diversidade das estruturas permite reduzir a possibilidade de convergência prematura, isto é, a obtenção de uma solução de mínimo local.

2.7.3. Variações de Estruturas de Busca para o PCC

Nessa seção, apresenta-se algumas variações de estruturas de busca utilizando AG para o Problema de Cobertura de Conjuntos. Estas estruturas são básicas de busca para os principais algoritmos genéticos para o PCC. Descreve-se ao final desta seção as principais diferenças entre os algoritmos implantados por Beasley e Chu (vide [BEA96]), Lorena e Lopes (vide [LOR97]). Outros algoritmos baseados em AG para PCC são encontrados na literatura (vide [ALS96] e [WRE95]).

2.7.3.1. Representação do Cromossomo e *fitness*

O primeiro passo para implementar com sucesso um algoritmo genético para um problema particular é conceber um esquema de representação e usá-lo para um mapeamento entre o espaço de soluções e o espaço de estruturas, de modo a preservar o significado do problema original.

A representação binária 0-1 é uma escolha usual e óbvia para o PCC por representar as variáveis básicas inteiras 0-1. Usa-se um *string* de n-dígitos binários como estrutura de cromossomo onde n é o número de colunas no PCC. O valor de 1 para dígito implica que a coluna i está na solução. A representação binária de um cromossomo do indivíduo (solução) para o PCC é ilustrado na Fig. 2.2. O ajustamento de um indivíduo está diretamente relacionado com o valor de sua função objetivo. Com a representação binária, o ajustamento f_i de um indivíduo i é calculado simplesmente por

$$f_i = \sum_{j=1}^{n} c_j s_{ij}$$

onde s_{ij} é o valor do *j-ésimo* dígito (coluna) na fileira correspondente ao *i-ésimo* indivíduo e c_j é o custo da coluna j.

coluna (gene)	1	2	3	4	5	...	...	n-1	n
Dígito fileira	0	1	0	1	0	...	...	1	0

Fig. 2.2: Representação binária de um cromossomo de um indivíduo

Um resultado importante acerca do uso da representação binária é que ao aplicar operadores genéticos aos *strings* binários, as soluções resultantes não tem garantias de serem viáveis. Existem duas maneiras de lidar com soluções inviáveis. Uma maneira é aplicar uma função

penalidade para o ajustamento de soluções inviáveis sem distorcer a paisagem de ajuste (vide [RIC89]). A outra maneira é criar operadores heurísticos que transformem soluções inviáveis em soluções viáveis. Beasley e Chu (vide [BEA96]) e Lorena e Lopes (vide [LOR97]) escolheram a segunda forma de representação pela dificuldade freqüente em determinar uma boa função de penalidade.

O problema de manter soluções viáveis deve também ser resolvido pelo uso de uma representação não-binária. Uma possível representação é ter o tamanho de cromossomo igual ao número de linhas no PCC. Nesta representação, a locação de cada gene corresponde a uma linha no PCC e o valor codificado de cada gene é uma coluna que cobre aquela linha (veja Fig. 2.3). Como a mesma coluna deve ser representada em mais que uma locação de gene, um método de codificação modificado para avaliação do *fitness* é usado por extrair somente o único conjunto de colunas que o cromossomo representa (isto é, colunas repetidas são contadas somente uma vez).

linha (gene)	1	2	3	4	5	...	*m*-1	*m*
String	12	8	12	313	6	...	43	5

Fig. 2.3: Representação não-binária de um cromossomo de um indivíduo.

Com esta representação, viabilidade pode geralmente ser mantida através do procedimento de cruzamento e mutação. Mas a avaliação do ajustamento pode tornar-se ambíguo porque a mesma solução pode ser representada de diferentes formas e cada forma pode dar um ajustamento diferente dependendo de como o *string* é representado. Uma experiência computacional limitada leva a acreditar que a performance do AG usando esta representação não-binária foi inferior ao que usa AG com representação binária.

2.7.3.2. Técnica de seleção dos pais

A seleção dos pais é a tarefa de designar oportunidades de reprodução para cada indivíduo na população. Existe um número de métodos largamente usados incluindo seleção proporcional (um método conhecido como roleta ponderada) e seleção por torneio.

No primeiro método cria-se uma roleta ponderada, de modo que cada estrutura da população corrente ocupe um "setor" da roleta em proporção à sua adaptação. O operador de reprodução atua fazendo "girar" a roleta e selecionando a estrutura correspondente ao setor sorteado. A cada giro da roleta, uma estrutura da população é selecionada com probabilidade proporcional à sua adaptação. Dessa forma, as estruturas com maiores valores de adaptação têm chance de possuir um número maior de descendentes na próxima geração. Esta representação foi utilizada por Lorena e Lopes (vide [LOR97]).

O método de seleção por torneio trabalha pela formação de duas parcelas de indivíduos, cada uma consistindo de T indivíduos extraídos da população aleatoriamente. Dois indivíduos com o melhor ajustamento, cada um tomado de uma das duas partes são escolhidos para cruzar. Usando um valor grande para T tem-se o efeito de aumentar a pressão de seleção em indivíduos mais convenientes. Este método foi utilizado por Beasley e Chu (vide [BEA96]).

2.7.3.3. Operadores de *Crossover*

Num AG tradicional, os operadores de cruzamento mais utilizados são o operador de ponto único e o operador de cruzamento por dois pontos. Estes operadores de cruzamento trabalham aleatoriamente gerando um ou mais pontos de cruzamento(s) e então permutando segmentos dos dois cromossomos pais para produzir dois cromossomos filhos, os quais promovem a troca de informações entre duas estruturas de acordo com pontos escolhidos aleatoriamente .

Uma técnica de cruzamento comumente usada é a de operador de cruzamento uniforme. Cada gene na solução-filho é criado pela cópia do gene correspondente de um dos pais, escolhidos de acordo com o gerador binário de números aleatórios {0,1}. Se o número aleatório é 0, o gene é copiado do primeiro pai, se é 1, o gene é copiado do segundo pai.

2.7.3.4. Taxa de Mutação

A mutação é aplicada a cada filho após o cruzamento. Ela funciona pela inversão de cada dígito na solução com alguma probabilidade pequena. Uma das principais dificuldades dos AGs (e da maioria dos algoritmos de busca) é a ocorrência de convergência para uma solução sub-ótima. Foi observado que esse problema está intimamente ligado à perda de diversidade na população. A mutação é vista geralmente como um gerador de base que fornece uma pequena porção de procura aleatória. Ela também ajuda a proteger contra a perda de informações genéticas valiosas pela reintrodução de informação perdida resultante da convergência prematura e desse modo expandindo o espaço de procura.

Back (vide [BAC93]) sugeriu uma taxa de mutação de $1/n$ como um limite inferior na taxa de mutação ótima, onde n é o comprimento do cromossomo. Este limite inferior é equivalente a mutar um dígito aleatoriamente escolhido por cromossomo.

É experiência de alguns autores (vide [BEA96] e [LOR97]) a utilização de uma taxa de mutação variável ao invés de uma fixa. Esta taxa variável deve depender da taxa de convergência do AG. Inicialmente, em uma execução, uma taxa de mutação baixa pode ser proveitosa para evitar que o trabalho do cruzamento e da busca local sejam destruídos. Entretanto, mais adiante na execução, a população tende a convergir, tornando-se muito homogênea. Nesse ponto uma taxa de mutação maior pode ajudar a introduzir diversidade na população, expandindo assim o espaço de busca e permitindo escapar de possíveis mínimos locais. Quando o algoritmo genético finalmente convergir, a taxa de mutação também se tornará estável em alguma taxa constante. A taxa que o AG converge depende do método de reposição de população.

2.7.3.5. Operador de Viabilidade Heurística

As soluções geradas pelos operadores de *crossover* e mutação podem violar as restrições do problema, isto é, algumas linhas podem não ser cobertas. Para fazer cada solução viável, deve-se identificar todas as linhas não cobertas e adicionar colunas à solução gerada, tal que estas linhas passem a ser cobertas. Basicamente, um mesmo operador de viabilidade é usado nos AG para o PCC, baseado na heurística gulosa de Chvàtal. Ou seja, a busca destas colunas perdidas é baseada na razão:

$$\frac{\text{custo uma coluna}}{\text{número de linhas não cobertas que ela cobre}}$$

Uma vez que as colunas são adicionadas e uma solução se torna factível, um passo de otimização local é aplicado para remover qualquer coluna redundante na solução. Uma coluna redundante é aquela que pode ser eliminada sem que a solução se torne inviável.
O algoritmo é como segue.

Seja

$\boldsymbol{M}$ = o conjunto de todas as linhas,

$\boldsymbol{N}$ = o conjunto de todas as colunas,

N_i = o conjunto das colunas que cobrem a linha i, $i \in \boldsymbol{M}$.

M_j = o conjunto das linhas cobertas pela coluna j, $j \in \boldsymbol{N}$,

S = o conjunto de colunas numa solução

R = o conjunto de linhas descobertas

n_i = o número de colunas que cobrem a linha i, $i \in \boldsymbol{M}$ em S.

(1) Inicialize $n_i := |S \cap N_i|, \forall i \in \boldsymbol{M}$.

(2) Inicialize $R := \{i \mid n_i = 0, \forall i \in \boldsymbol{M}\}$

(3) Para cada linha i em R (em ordem crescente de i):

(a) encontre a coluna j (em ordem crescente de j) em N_i que minimize $c_j / |R \cap M_j|$,

(b) adicione j a S e estabeleça $n_i := n_i + 1, \forall i \in M_j$.

Estabeleça $R := R - M_j$

(4) Para cada coluna j em S (em ordem decrescente de c_j), se $n_i \geq 2$, $\forall_i \in M_j$, estabeleça $S := S - j$ e faça $n_i := n_i - 1$, $\forall i \in M_j$.

(5) S é agora uma solução viável para o PCC que não contém colunas redundantes.

Os passos (1) e (2) identificam as linhas descobertas. Os passos (3) e (4) são heurísticas "gulosas" no sentido que no passo (3) as colunas com baixa razão de custo são consideradas primeiro e no passo (4) as colunas com alto custo são retiradas primeiro sempre que possível.

2.7.3.6. Modelo de Reposição da População

Uma vez que uma nova solução-filho viável foi gerada, o filho irá repor um membro aleatoriamente escolhido (geralmente aquele com uma média baixa de ajustamento) na população. Note que uma média baixa de ajustamento significa menos *fitness*. Este tipo de método de reposição é chamado de reposição incremental ou estado firme. Outro método comumente usado é a reposição geracional, onde uma nova população de filhos é gerada e os pais da população são substituídos completamente.

As vantagens da reposição pelo método do estado firme são que as melhores soluções são sempre mantidas na população e a solução gerada recentemente está imediatamente disponível para seleção e reprodução. Portanto, o AG que usa o método do estado firme de reposição tende a convergir mais rápido que o método de reposição geracional (vide [BEA96].

2.7.4. Algoritmo Genérico Utilizando AG para PCC

Abaixo, apresentam-se os passos para uma AG modificada para PCC:

(1) Gere uma população inicial de N soluções aleatórias. Estabeleça $t := 0$.

(2) Selecione duas soluções P_1 e P_2 da população de soluções, usando *torneio binário, roleta ponderada,* etc.

(3) Combine P_1 e P_2 para formar uma nova solução C usando operador de cruzamento (*um ponto, dois pontos, fusão*).

(4) Mute k genes aleatoriamente selecionados em C, onde k é determinado por uma escolha de mutação *fixa* ou *variável.*

(5) Faça C viável e remova as suas colunas redundantes em C pela aplicação de *operador de viabilidade heurístico*, caso a escolha para a estrutura do cromossomo seja a binária. (Se a escolha for não-binária a solução será viável, veja a seção 2.7.3.1.).

(6) Se C é idêntico a qualquer uma das soluções na população, vá para o passo (2); caso contrário, estabeleça $t := t + 1$ e vá para o passo (7).

(7) Reponha uma solução escolhida aleatoriamente com média de *fitness* baixa na população por C (método do *estado firme de reposição*) ou toda a população (método *geracional*).

(8) Repita os passos (2) a (7) até que (*uma condição de parada seja alcançada*). A melhor solução encontrada é aquela que apresenta o melhor *fitness* na população.

A tabela 2.2 apresenta as diferenças entre dois principais algoritmos genéticos para o PCC.

	[LOR97]	[BEA96]
Seleção	Roleta ponderada	Torneio de seleção
Crossover	Um ponto	Fusão
Mutação	Variável	Variável
Inicialização	Chvàtal [CHV79] + busca local	Aleatória
Tamanho da População	100	100

Tabela 2.2 : Resumo das características de dois AG

2.8. Outros Métodos de Resolução

2.8.1. Relaxação *Surrogate*

A relaxação Surrogate, como a relaxação Lagrangeana, tem o objetivo de criar um problema mais fácil de ser resolvido, fornecendo um limite para a solução ótima do problema original.

Uma alternativa para a relaxação Lagrangeana é a *Relaxação Surrogate*, que é usada por Lorena e Lopes (vide [LOR94]). Para um vetor de multiplicadores $v \in \Re_+^m$, a relaxação Surrogate para o PCC tem a seguinte forma:

$$S(u) = min \quad \sum_{j \in N} c_j x_j$$

Sujeito a

$$\sum_{j \in N} \left(\sum_{i \in M_j} v_i \right) x_j \geq \sum_{i \in M} v_i$$

$x_j \in \{0,1\}, j \in N,$

Este problema relaxado é um *problema da Mochila*, que ainda é NP- difícil, mas solúvel em tempo Pseudo-Polinomial, e, na prática, bastante eficiente (vide [MAR90]). Apesar disso, Lorena e Lopes relaxaram a restrição de integridade e resolveram o problema com um algoritmo de complexidade O(n). Embora a última relaxação seja ainda equivalente à relaxação LP do PCC, os resultados experimentais reportados por Lorena e Lopes sugerem que o uso da Relaxação Surogate, ao invés da Relaxação Lagrangeana com otimização do subgradiente, permite obter multiplicadores próximos ao ótimo num tempo computacional mais curto. Também é reportado por Lorena e Lopes (vide [LOR94]) que o Dual Surrogate, mesmo para o problema relaxado, é normalmente mais difícil de ser resolvido do que o Dual Lagrangeano.

2.8.2. *Simulated Annealing*

As abordagens que se encontram na literatura para o PCC utilizando *Simulated Annealing* foram desenvolvidas por Jacobs e Brusco (vide [JAC95]) e Jacobs, Brusco e Thompson (vide [JAC96]).

No primeiro artigo [JAC95], uma solução inicial S é gerada pela heurística gulosa que, em cada iteração, seleciona aleatoriamente uma linha descoberta. Em seguida adiciona à solução a coluna com menor índice que cobre esta linha. Depois dessa adição, colunas possivelmente redundantes são eliminadas, e o processo prossegue até que uma solução viável (mínima) é determinada. Então, um número de iterações é efetuada no qual d colunas são removidas aleatoriamente da solução corrente S, a qual é completada de forma a obter uma nova solução mínima S' por meio de um algoritmo guloso, de maneira análoga à heurística de Beasley (vide [BEA90]). S' torna-se a solução corrente se for melhor que S; caso contrário S é trocado por S' com uma probabilidade que decresce exponencialmente com a diferença entre os valores de S e S', e com o número de iterações que é executada.

Jacobs, Brusco e Thompson (vide [JAC96]) apresentaram dois principais aperfeiçoamentos sobre a abordagem de Jacobs e Brusco (vide [JAC95]). Antes de tudo, a seleção das colunas que devem ser removidas da solução corrente S é geralmente feita aleatoriamente (como antes) a cada três iterações, enquanto que nas duas iterações restantes remove-se as colunas cuja remoção deixa menor o número de linhas descobertas. Depois, para cada coluna j na solução tem-se uma lista chamada *morphs,* que são colunas "similares" a j. Depois das colunas serem removidas da solução corrente, cada coluna na solução é substituída por um de seus *morphs* se isto melhorar a razão entre o custo e o número de linhas cobertas na solução parcial.

2.8.3. Heurística Gulosa Probabilística

Para o caso unicusto, a heurística probabilística apresentada por Feo e Resende (vide [FEO88]) é uma variação não determinística da abordagem gulosa de Chvàtal (vide [CHV79]), realizando uma busca em "boas" vizinhanças para as suas melhores soluções. O método distingue-se do anterior onde o índice $f(c_j,k_j)$ é selecionado. Em vez de solucionar o índice f correspondente à coluna de maior razão $| M_j | / c_j$, a seleção é feita aleatoriamente de um conjunto de índices candidatos que possuem razão $| M_j | / c_j$ igual ou superior a $\alpha \times \max \{| M_j | / c_j ; 1 \leq j \leq n \}$, onde $0 \leq \alpha \leq 1$. Além disso, os elementos supérfluos são eliminados da cobertura parcial $S^0 \cup \{f\}$.

O algoritmo recebe como entrada os n conjuntos discretos $M_1,...M_n$, o vetor custo c, o parâmetro α e o número de repetições I, e retorna uma cobertura S^*. O método de Chvátal é executado uma vez, enquanto a versão probabilística é repetida I vezes no *loop* **repita-até** (veja a seção 2.5.1). A heurística gulosa de Chvàtal é um caso especial do procedimento probabilístico, onde $\alpha = 1.0$ e $I = 1$.

Feo e Resende testaram sua heurística utilizando a classe de problemas de triplas de Steiner e compararam os resultados obtidos à heurística de Chvátal. A heurística probabilística provê as soluções ótimas para esses problemas, dadas as instâncias para as quais a solução ótima é conhecida. Além disso, provê as melhores soluções conhecidas para outras instâncias testadas . O desempenho apresentado pela heurística probabilística, em relação à heurística de Chvátal, pode ser explicado da seguinte forma: primeiro, a heurística de Feo e Resende sempre produz soluções minimais ao PCC, pela eliminação dos elementos supérfluos das coberturas parciais. Segundo, a introdução de um critério de seleção probabilístico dos elementos do conjunto determinado por $f(c_j,k_j)$ a serem incluídos na cobertura parcial permite evitar a obtenção de soluções de mínimo local.

2.8.4. Métodos exatos

Algoritmos exatos foram desenvolvidos por Salkin e Koncal (vide [SAL73]), Etcheberry (vide [ETC77]), Balas e Ho (vide [BAL80]), Beasley (vide [BEA87]), Beasley e Jörnsten (vide [BEA92]), Balas e Carrera (vide [BAL96]).

Os métodos mais conhecidos utilizados para a solução do PCC de forma exata são os métodos enumerativos e os de planos de corte. Nestes métodos é realizada uma enumeração, de forma implícita ou explícita, das soluções viáveis para o problema. Algoritmos enumerativos mantêm um caminho de enumeração e utilizam um algoritmo de busca em lista ou em árvore. Através das restrições do problema e da integridade das variáveis, é possível abandonar a busca em certos nós que não conduzem à soluções melhores, de forma a promover a enumeração implícita de um grande número de nós. Algoritmos enumerativos foram propostos por Lemke et. al (vide [LEM72] e Etcheberry (vide [ETC77]).

Algoritmos de planos de corte deduzem desigualdades suplementares às restrições do PCC a partir das próprias restrições e da condição de integridade das variáveis. Estas desigualdades "cortam" parte da região viável do problema de programação linear correspondente, enquanto deixam a região viável do problema original intacta. Quando um número suficiente de hiperplanos forem gerados, o problema original terá a mesma solução ótima do problema de programação linear correspondente. Em geral, uma abordagem por planos de corte envolve a solução de uma seqüência de problemas. Dado um PCC com uma matriz 0-1 A^p, inicialmente encontra-se um vetor solução x^p, de custo c^p. Então, se uma nova desigualdade (ou corte) pode ser obtida tal que:

(a) todas as soluções com custo menor que c^p a satisfazem e
(b) x^p não a satisfaz,

então uma nova matriz A^{p+1} é obtida pela inclusão da desigualdade à matriz A^p . Se em qualquer iteração *p,* tal desigualdade não pode ser deduzida, o procedimento termina com

uma solução ótima x^* com custo c^*, onde $c^* = \min_{0 \le k \le p^*} \{c^k\}$. Bellmore e Ratliff (vide [BEL71]) e Balas e Ho (vide [BAL80]) descrevem métodos de planos de corte para solução do PCC.

Em seguida, descreve-se o estado-da-arte dos algoritmos exatos para o PCC. O algoritmo proposto por Beasley (vide [BEA87]) resolve a relaxação PL do PCC encontrando a solução linear ótima para o nó da raiz da árvore de ramificação. Nos outros nós, um limite inferior é computado usando relaxação Lagrangeana com otimização por subgradientes, começando com o vetor de multiplicadores associado ao nó principal. Em cada nó, em particular, multiplicadores de Lagrange próximo ao ótimo são computados através de um procedimento de dual ascendente seguido por um procedimento de subgradiente. Então o problema é reduzido de tamanho pela aplicação da fixação de preços Lagrangeanos, e o PL do problema reduzido é resolvido ao ótimo pelo método dual simplex. A solução do PL relaxado não tira nenhuma vantagem da informação obtida da computação dos multiplicadores de Lagrange próximo ao ótimo. Aparentemente, a única razão para a computação destes multiplicadores é o PL menor que tem de ser resolvido depois da redução. Soluções viáveis são computadas pelo uso da heurística gulosa simples, aplicando-se diversos procedimentos de dominância para a redução dos números de linhas e de colunas. A regra de ramificação seleciona uma linha descoberta com o maior multiplicador de Lagrange. Em seguida gera dois novos problemas, fixando em 0 e 1 a variável que cobre a linha selecionada, e que apresenta o menor custo Lagrangeano.

Um aperfeiçoamento do algoritmo de Beasley (vide [BEA87]) é dado por Beasley e Jörnsten (vide [BEA92]). A principal diferença com respeito ao algoritmo original está na adição de restrições ao problema. Estas restrições são manuseadas pelo procedimento de otimização do subgradiente, exceto no nó inicial. Outros melhoramentos com relação a [BEA87] são o uso da relaxação Lagrangeana de [BEA90] e uma nova estratégia de ramificação. Nesta estratégia, computa-se, em cada nó, a solução Lagrangeana correspondente ao melhor vetor de multiplicadores u, determinando a linha i com o máximo $|s_i(u)u_i|$, onde $s_i(u)$ é o subgradiente associado com a linha i. A ramificação é feita sobre a

variável associada com a coluna $j \in N_i$ tal que $x(u_i) = 1$ e cujo valor de x_j é o máximo na solução do PL depois da adição dos cortes de Gomory.

O mais recente algoritmo exato para PCC apresentado na literatura é o de Balas e Carrera (vide [BAL96]). Este algoritmo é baseado num procedimento denominado de *"subgradiente dinâmico"* em todo nó da arvore de busca dentro de um esquema *branch and bound*. Dois esquemas de ramificação baseados na informação dual associada com os multiplicadores de Lagrange são usados. Observa-se que o melhor vetor u de multiplicadores de Lagrange computado pelo método de Balas e Carrera sempre corresponde a uma solução viável dual (e maximal), isto é, não existem colunas com custos Lagrangeanos negativos. Este novo procedimento, aplicado ao Dual Lagrangeano em todo nó da árvore de busca, combina o método do subgradiente padrão com heurísticas primais e duais que interagem para mudar os multiplicadores de Lagrange e ajustar os limites superiores e inferiores, fixar variáveis, e periodicamente revisar o problema Lagrangeano. Este método obteve performance muito boa em comparação com outros algoritmos exatos existentes; juntamente com algoritmo de ramificação e avaliação sucessiva avançou bastante o estado-da-arte na resolução do PCC.

Embora garantam uma solução ótima, os métodos exatos são, em geral, caros do ponto de vista computacional, principalmente se aplicados a problemas de grande porte.

CAPÍTULO III

3. COMPARAÇÃO NUMÉRICA DOS ALGORITMOS

3.1. Considerações Iniciais

Os resultados obtidos pela aplicação dos algoritmos mais recentes e eficientes serão comparados neste capítulo. Adicionalmente, apresenta-se um estudo mais detalhado de comparação entre as principais vantagens e desvantagens de algoritmos: abordagens que utilizam relaxação Lagrangeana e otimização por subgradientes e algoritmos genéticos. Nestes dois casos, procurou-se concentrar maior atenção na captação de informações que podem ser utilizadas para desenvolvimento de algoritmos mais eficientes.

Inicialmente, fez-se a comparação tomando por base um conjunto de testes padrão, os quais são apresentados resumidamente na tabela 3.1. Após, escolheu-se duas dessas metodologias que se mostraram mais eficientes para uma comparação mais detalhada. Para essas duas classes de algoritmos, criou-se um conjunto de testes, com custos mais homogêneos, não necessariamente inteiros, na faixa de 80 a 100.

3.2. Problemas da *OR Library*

A maior parte dos algoritmos eficientes para a resolução do Problema de Cobertura de Conjuntos, com poucas exceções, foram testados sobre as instâncias da *OR Libra*ry de Beasley. Resumidamente, apresenta-se na tabela 3.1, algumas características deste conjunto de testes, disponibilizado pelo autor, e que pode ser obtido através de transferência eletrônica de arquivos de uma biblioteca de problemas, disponibilizado na rede do *Imperial*

College, Inglaterra. Para maior compreensão da estrutura destas instâncias deve-se consultar o trabalho de J. E. Beasley (vide [BEA90a]). Não se comparará algoritmos que não foram testados nestes problemas.

Conjunto de Problemas	Número de linhas	Número de colunas	Densidade (%)	Variação dos Custos
4	200	1000	2	1-100
5	200	2000	2	1-100
6	200	1000	5	1-100
A	300	3000	2	1-100
B	300	3000	5	1-100
C	400	4000	2	1-100
D	400	4000	5	1-100
E	500	5000	10	1-100
F	500	5000	20	1-100
G	1000	10000	2	1-100
H	1000	10000	5	1-100

Tabela 3.1: Conjunto de Testes da *OR Library*

A densidade acima é definida como o valor percentual de e/mn, onde e é o número entradas não-nulas na matriz $A(m \times n)$, m linhas e n colunas.

Apresenta-se, a seguir, as comparações entre os algoritmos mais eficientes sobre os problemas da tabela 3.1. Estes algoritmos foram, em sua maioria, implementados em computadores diferentes. Caprara, Fischetti e Toth (vide [CAP98]) utilizaram os resultados reportados em J. J. Dongarra (vide [DON96]) para aproximar as performances desses computadores a um padrão comum. Mesmo levando em conta essa aproximação *grosseira*, as tabelas 3.2, 3.3, 3.4 e 3.5 dão, na maioria dos casos, uma indicação mais clara da performance dos algoritmos pesquisados.

3.2.1. Avaliação do Desempenho de Tempo e de Erro Numérico

Para fins de avaliação do desempenho, foram considerados algoritmos heurísticos e exatos. Entre os algoritmos heurísticos, foram estudados duas categorias distintas. A primeira delas consiste na categoria de algoritmos heurísticos baseados em relaxação lagrangeana e

surrogate, com otimização por subgradiente. Nesta classe estão os algoritmos de Beasley (vide [BEA90]), Balas e Carrera (vide [BAL96]), Haddadi (vide [HAD97]) e Caprara, Fischetti e Toth (vide [CAP95]), todos estes desenvolvidos a partir da idéia de relaxação lagrangeana. Além destes estudou-se o algoritmo de Lorena e Lopes (vide [LOR94]), que por sua vez foi desenvolvido a partir do conceito de relaxação surrogate. A segunda classe é composta pelos algoritmos meta-heurísticos. Nesta classe estão o algoritmo genético desenvolvido por Beasley e Chu (vide [BEA96]), os algoritmo de Jacob e Brusco (vide [JAC95]) e Jacob, Brusco e Thomson (vide [JAC96]), ambos baseados em *simulated annealing*.

Entre os algoritmos exatos foram estudados os algoritmos de Beasley (vide [BEA87]), Beasley e Jörnsten (vide [BEA92]) e Balas e Carrera (vide [BEA96]), todos baseados em processo de enumeração implícita e busca em árvores.

A tabela 3.2 reporta os resultados obtidos pelos algoritmos heurísticos, para as instâncias das Classes de Problemas A, B, C e D da OR Library, cujas soluções ótimas são conhecidas. Em particular, para os algoritmos de Beasley (vide [BEA90]), coluna "*Be*", Lorena e Lopes (vide [LOR94]), coluna "*LL*", Balas e Carrera (vide [BAL96]), coluna "*BaCa*", Beasley e Chu (vide [BEA96]), coluna "*BeCh*", Haddadi (vide [HAD97]), coluna "*Ha*", e Caprara, Fischetti e Toth (vide [CAP95]), coluna "*CFT*", são reportados os valores das melhores soluções encontradas (coluna "*Solução*"), assim como também o tempo de processamento total gasto para obter esta solução (coluna "*Tempo*"), com exceção dos algoritmos de "*BaCa*" e "*Ha*" para os quais são dados o tempo de execução para a heurística completa. Ceria, Nobili e Sassano (vide [CER95]), Jacob e Brusco (vide [JAC95]) e Brusco, Jacob e Thompson (vide [JAC96]) não reportaram nenhum resultado para estas instâncias de problemas, e portanto não são apresentados.

Para todas as instâncias, os algoritmos "*BeCh*" e "*CFT*" encontraram o ótimo. Os algoritmos "*Be*", "*LL*", "*BaCa*" e "*Ha*" apresentam soluções ligeiramente piores, na média. A média de tempo de computação é de 4 segundos para o algoritmo "*LL*", 30 segundos para "Ha", 40 segundos para "Be", 45 segundos para "*BaCa*", 50 segundos para "*CFT*", e 180 segundos para "*BeCh*". Portanto, o algoritmo "*CFT*" é muito mais rápido que o algoritmo

"*BeCh*", e a sua velocidade é comparável com "*Be*", "*BaCa*" e "*Ha*". Quanto ao "*LL*", ele é, sem dúvida, o mais rápido algoritmo, embora os tempos de computação reportados em Lorena e Lopes (vide [LOR94]) não incluam os tempos para as reduções iniciais. Em 15 das 20 instâncias de problemas testados, contudo, este algoritmo não é capaz de encontrar a solução ótima. O gráfico 3.1 fornece os desvios percentuais médios do ótimo para essas classes de algoritmos estudados.

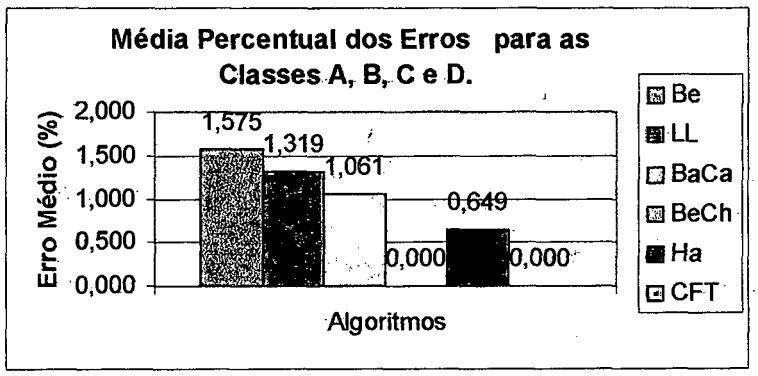

Gráfico 3.1: Média Percentual dos Erros para as classes A, B, C e D.

A tabela 3.3 fornece os resultados para as grandes instâncias da *OR Library* (Classes de Problemas *E*, *F*, *G* e *H*), para as quais a solução ótima é conhecida somente em poucos casos. As colunas "*Melhor*" e "*LI*" provêem, respectivamente, as melhores soluções e os limites inferiores conhecidos. Lorena e Lopes (vide [LOR94]) não é incluído como na tabela 3.2, porque não reportam resultados para estas instâncias. Entretanto, são incluídos os algoritmos de Jacobs e Brusco (vide [JAC95]), coluna "*JaBr*", Brusco, Jacobs e Tompson (vide [JAC96]), coluna "*BJT*", e Ceria, Nobili e Sassano (vide [CER95]), coluna "*CNS*". Os últimos autores apresentam somente resultados para as classes G e H. Além disso, Jacob e Brusco (vide [JAC95]) e Ceria, Nobili e Sassano (vide [CER95]) apresentam somente o tempo limite para a execução de seus algoritmos, que são reportados na coluna "Tempo". Para todas estas instâncias, o algoritmo "CFT" resulta na melhor solução conhecida na literatura. Os algoritmos "BeCh" e "BJT" encontraram as mesmas soluções que as do algoritmo "CFT" com três e duas exceções, respectivamente. Os algoritmos "CNS" e "JaBr" apresentam soluções que são, em média, ligeiramente piores que os

algoritmos "BeCh", "BJT", e "CFT", porém melhores que os algoritmos "Be", "BaCa" e "Ha". A média de tempo de computação é cerca de 200 segundos para os algoritmos "Be" e "BaCa", 250 segundos para "BJT", 400 segundos para "CFT", e 7500 segundos para "BeCh". Nenhuma boa comparação pode ser feita com os tempo de computação dos algoritmos "CFT" e "JaBr".

O gráfico 3.2 apresenta os erros percentuais médios cometidos por esses algoritmos. Devido às dificuldade das instâncias analisadas, os erros percentuais médios encontrados são bem maiores. Os algoritmos de maior robustez são "BeCh", "BJT" e "CFT", e apresentam erros na faixa de 8%.

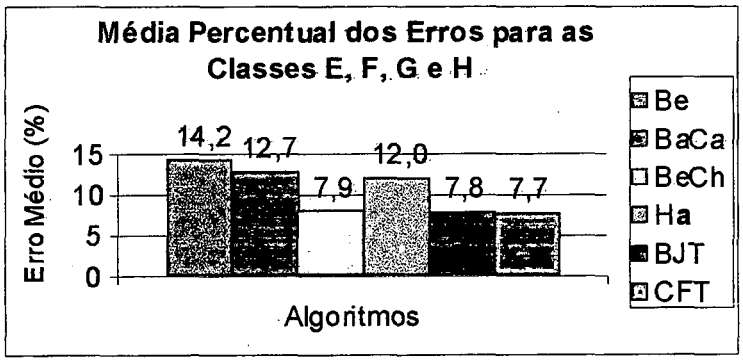

Gráfico 3.2 : Média Percentual dos Erros para as classes E, F, G e H.

Em seguida faz-se a comparação dos algoritmos exatos sobre as classes de problemas da *OR Library* (Classes A, B, C e D) que são comprovadamente solúveis em um tempo computacional razoável. Na tabela 3.4 reporta-se o tempo de computação (coluna "Tempo") e um número de nós explorados através da ramificação e avaliação sucessiva (coluna "Nós") pelos algoritmos de Beasley (vide [BEA87]), coluna "Be", Beasley e Jörnsten (vide [BAL92]), coluna "BeJö", e Balas e Carrera (vide [BAL96]), coluna "BaCa*".

		Be		LL		BaCa	
Nome	Ótimo	Sol	Tempo	Sol	Tempo	Sol	Tempo*
A.1	253	255	36,0	254	2,7	258	39,0
A.2	252	256	44,2	255	2,9	254	40,9
A.3	232	234	28,1	234	2,6	237	28,6
A.4	234	235	33,5	234	2,4	235	36,3
A.5	236	237	19,0	238	2,2	236	26,2
B.1	69	70	28,4	70	3,0	69	29,0
B.2	76	77	40,8	76	4,0	76	29,0
B.3	80	80	25,4	81	4,4	81	35,1
B.4	79	80	37,0	81	4,3	79	29,0
B.5	72	72	26,0	72	4,1	72	32,6
C.1	227	230	42,4	227	4,0	230	116,2
C.2	219	223	66,0	222	4,1	220	56,1
C.3	243	251	75,1	251	4,9	248	61,7
C.4	219	224	63,4	224	5,4	224	68,1
C.5	215	217	39,9	216	4,1	217	64,6
D.1	60	61	40,9	60	4,8	61	36,6
D.2	66	68	52,7	68	3,5	67	46,6
D.3	72	75	55,8	75	5,8	74	47,2
D.4	62	64	36,5	63	4,8	63	39,8
D.5	61	62	36,7	62	4,5	61	36,2

		BeCh		Ha		CFT	
Nome	Ótimo	Sol	Tempo	Sol	Tempo	Sol	Tempo
A.1	253	253	222 4	254	22,1	253	82 0
A.2	252	252	327,9	253	20,2	252	116,2
A.3	232	232	127,0	234	21,7	232	249,9
A.4	234	234	45 5	234	16,8	234	4,7
A.5	236	236	23,7	236	17,4	236	80,0
B.1	69	69	20,0	69	26,4	69	4,0
B.2	76	76	11,6	76	28,3	76	6,1
B.3	80	80	7097	81	28,0	80	18,0
B.4	79	79	29,9	79	29,7	79	6,3
B.5	72	72	5,3	72	26,6	72	3,3
C.1	227	227	187,9	228	32,7	227	74,0
C.2	219	219	40,7	223	34,0	219	64,2
C.3	243	243	541,3	245	35,9	243	70,2
C.4	219	219	144,6	223	32,8	219	61,6
C.5	215	215	80,6	216	30,2	215	60,3
D.1	60	60	13,8	61	45,9	60	23,1
D.2	66	66	198,6	66	49,3	66	22,0
D.3	72	72	785,3	73	48,6	72	22,6
D.4	62	62	73,5	62	48,0	62	8,3
D.5	61	61	79,8	62	42,5	61	10,3

Tabela 3.2: Algoritmos Heurísticos para as Classes *A, B, C, D.* * Sobre todos os tempo de execução do algoritmo heurístico.

			Be		BaCa		CNS		BeCh	
Nome	Melhor	LI	Sol	Tempo	Sol	Tempo *	Sol	Tempo º	Sol	Tempo
E.1	29	29	29	72,6	29	55,3	x	x	29	38,2
E.2	30	28	32	92,7	32	76,0	x	x	30	14647,7
E.3	27	27	28	92 7	28	80,9	x	x	27	28360,2
E 4	28	28	30	100,3	29	77,5	x	x	28	539,9
E.5	28	28	28	80,8	28	61,6	x	x	28	35,0
F.1	14	14	15	43,9	14	67,5	x	x	14	76,4
F.2	15	15	16	102,6	15	88,6	x	x	15	78,1
F.3	14	14	15	124,7	15	76,5	x	x	14	266,8
F.4	14	14	15	118,2	15	74,8	x	x	14	209,7
F.5	13	13	14	129,3	14	62,2	x	x	13	13192,6
G.1	176	165	184	287,8	183	325,6	176	4905,5	176	30200,0
G.2	154	147	163	204,9	161	370,1	155	4905,5	155	360,5
G.3	166	153	174	318,2	175	378,6	167	4905,5	166	7841,6
G.4	168	154	176	292,0	176	332,2	170	4905,5	168	25304,7
G 5	168	153	175	277,5	172	262,6	169	4905,5	168	549,3
H.1	63	52	68	317,7	68	488,4	64	4905,5	64	1682,1
H.2	63	52	66	293,9	67	380,7	64	4905,5	64	530,3
H.3	59	48	65	325,1	63	443,1	60	4905,5	59	1803,5
H.4	58	47	63	333,5	62	354,7	59	4905,5	58	27241,8
H.5	55	46	60	303,0	58	321,3	55	4905,5	55	449,6

			JaBr		Ha		BJT		CFT	
Nome	Melhor	LI	Sol	Tempo º	Sol	Tempo	Sol	Tempo	Sol	Tempo
E.1	29	29	29	408,0	29	187,1	29	0,8	29	26,0
E.2	30	28	30	408,0	32	199,8	30	23,7	30	408,0
E.3	27	27	27	408,0	28	198,1	27	31,1	27	94,2
E 4	28	28	28	408,0	29	191,3	28	4,0	28	26,3
E.5	28	28	28	408,0	28	181,6	28	1,4	28	36,6
F.1	14	14	14	408,0	14	654,1	14	2,4	14	33,2
F.2	15	15	15	408,0	15	646,8	15	1,9	15	31,2
F.3	14	14	14	408,0	15	665,3	14	7,2	14	248,5
F.4	14	14	14	408,0	15	665,5	14	4,0	14	31,0
F.5	13	13	14	408,0	14	684,7	13	180,7	13	201,1
G.1	176	165	179	408,0	181	191,3	176	135,3	176	147,0
G.2	154	147	158	408,0	160	216,5	155	112,7	154	783,4
G.3	166	153	170	408,0	174	193,1	166	1755,1	166	978,0
G.4	168	154	172	408,0	177	184,0	168	1792,3	168	378,5
G 5	168	153	168	408,0	173	192,0	168	160,7	168	237,2
H.1	63	52	64	408,0	66	479,9	64	44,6	63	1451,1
H.2	63	52	64	408,0	67	462,3	63	643,7	63	887,0
H.3	59	48	61	408,0	62	462,4	59	255,4	59	1560,3
H.4	58	47	59	408,0	61	458,0	58	139,2	58	237,6
H.5	55	46	55	408,0	56	452,3	55	23,4	55	155,4

Tabela 3.3: Algoritmos Heurísticos para as Classes E, F, G, H.

* Sobre todos os tempo de execução do algoritmo heurístico.

º Tempo Limite,

		Be		BeJö		BaCa*	
Nome	Ótimo	Tempo	nós	Tempo	nós	Tempo	nós
A.1	253	395,3	515	565,2	373	180,6	95
A.2	252	534,4	809	555,6	357	49,4	28
A.3	232	564,5	857	482,4	377	68,6	41
A.4	234	158,4	63	126,6	17	18,8	10
A.5	236	144,7	79	94,8	15	7,9	3
B.1	69	418,2	411	413,2	137	108,3	81
B.2	76	2021,4	3273	1578,6	1655	1301,6	939
B.3	80	762,1	1383	774,8	671	357,6	288
B.4	79	2873,2	4381	2505,4	3157	2137,1	1508
B.5	72	461,0	661	459,8	355	97,4	69
C.1	227	1036,2	1265	1038,4	549	89,2	34
C.2	219	576,2	275	1242,0	619	366-9	145
C.3	243	10405,2	15757	5 950,8	5957	687,3	291
C.4	219	662 3	493	1063,0	515	305,5	116
C.5	215	548,2	375	1233,4	1001	207,6	91
D.1	60	2990,4	3393	1351,8	1015	617,3	337
D.2	66	5380,6	6739	6718,4	7179	5478,3	2771
D.3	72	15137,3	19707	9439,0	10455	11426,4	5120
D.4	62	9923,3	12389	7535,6	8687	6399,9	2931
D.5	61	387,7	147	454,4	63	38,7	18

Tabela 3.4: Algoritmos Exatos para as Classes A, B, C e D

Também são considerados os sistemas computacionais de propósitos gerais para programação inteira linear, conhecidos por CPLEX 4.0.8 e MINTO 2.3, cujos resultados são apresentados na tabela 3.5. Com estes sistemas são executadas as entradas originais das instâncias (as colunas "CPLEX" e "MINTO") e as instâncias pré-processadas pelas heurísticas de Caprara, Fischetti e Toth (1995), sem pós-otimização, e o método de Caprara, Fischetti e Toth (1998) para a computação da relaxação de programação linear, e uma relaxação PL através de custo reduzidos. Este procedimento fornece uma entrada para os sistemas de programa linear inteira (PLI) consideravelmente menor que os originais, e com uma solução inicial próxima ao ótimo do PCC.

		CPLEX		MINTO		Imp. CPLEX		Imp. MINTO	
Nome	Ótimo	Tempo	nós	Tempo	nós	Tempo	nós	Tempo	nós
A.1	253	137,6	44	66,3	55	63,4	29	44,1	55
A.2	252	56,1	18	78,9	85	53,8	27	55,4	97
A.3	232	91,3	37	66,5	71	45,6	25	44,8	71
A.4	234	26,7	10	25,0	13	8,4	6	11,7	13
A.5	236	22,4	10	24,6	13	13,7	14	14,1	13
B.1	69	165,1	40	141,1	67	72,0	32	69,1	67
B.2	76	759,9	173	470,1	301	631,5	206	255,4	301
B.3	80	252,7	66	232,2	261	153,4	69	152,4	261
B.4	79	1245,1	301	1419,6	1403	741,3	298	1034,8	1403
B.5	72	220,4	56	119,7	93	112,2	57	74,4	93
C.1	227	169,4	38	90,0	39	108,6	46	45,4	45
C.2	219	308,6	61	204,1	91	246,0	71	119,2	92
C.3	243	683,5	129	599,3	489	238,2	73	512,3	485
C.4	219	18 1,6	38	7 1,7	27	151,2	47	38,5	27
C.5	215	63,5	13	111,8	47	119,9	34	52,9	47
D.1	60	1484,8	210	965,4	324	326,3	84	404,0	305
D.2	66	3638,7	575	4046,3	2791	1858,9	496	3084,1	2791
D.3	72	2179,1	324	3117,6	2055	1504,5	358	2188,7	2055
D.4	62	2252,4	315	3148 9	1957	1131,0	272	2078,2	1955
D.5	61	336,9	43	79,8	13	106,4	28	31,4	13

Tabela 3.5: Sistemas comerciais aplicados às Classes A, B, C e D

De acordo com as tabelas 3.4 e 3.5, a média de tempo computacional para o algoritmo "*Be*" é de 3000 segundos, 2000 segundos para "*BeJö*", 1500 segundos para "*BaCa**", 700 segundos para "CPLEX", 750 segundos para "MINTO", 500 segundos para "Imp. MINTO", e 400 segundos para "Imp. CPLEX". Isto mostra que os programas de computadores considerados o estado-da-arte são competitivos com os melhores métodos exatos apresentados na literatura e suas performances podem ser melhoradas sensivelmente por procedimento de pré-processamento.

Na tabela 3.6, comparou-se os algoritmos heurísticos da tabela 3.2, com exceção do algoritmo "LL", que sofreu pré-processamento dos dados, com os algoritmos exatos mais recentes "Be", "BeJö" e "BaCa*", da tabela 3.4, quanto ao tempo de execução em segundos. Verifica-se que os heurísticos são muito mais rápidos que os exatos. Observa-se que o algoritmo heurístico "Ha" é quase 48 vezes mais rápido que "BaCa*", que obteve o menor tempo de execução entre todos os algoritmos exatos.

	Algoritmos Heurísticos					Algoritmos Exatos		
	Be	BaCa	BeCh	Ha	CFT	Be	BeJö	BaCa*
Médias	41,39	44,94	183,455	31,855	49,355	2769,03	2179,16	1497,22

Tabela 3.6 : Médias de tempo para as Classes de Problemas A, B, C e D da *OR Library* para Métodos Exatos e Heurísticos.

3.2.2. Avaliação Conjunta (Erro Numérico × Desempenho Computacional)

Nas tabela 3.7 e 3.8 estabeleceu-se como medida de eficiência um índice normalizado no intervalo de 0 a 100%, caracterizando o grau de pertinência dos algoritmos heurísticos, apresentados nas tabelas 3.2 e 3.3, ao *conjunto dos algoritmos eficientes*. Quanto mais próximo de 100% for este índice, maior será a aceitação do algoritmo em questão como sendo um exemplo de algoritmo eficiente. Por outro lado, algoritmos com índice de eficiência próximos de zero, serão descartados do conjunto de alternativas eficientes.

Para a obtenção deste índice considerou-se, separadamente, os tempos e os valores numéricos obtidos pelos algoritmos. Para cada uma destas medidas, estabeleceu-se entre os diversos algoritmos, para o pior e o melhor resultado, respectivamente, os índice 0 e 100%. Para as valores intermediários destas medidas, foram estabelecidos índices proporcionais dentro do intervalo [0,100]. Finalmente, para cada algoritmo, tomou-se o menor entre os índices de eficiência calculados (tempo, valor) para formar as tabelas 3.7 e 3.8 (vide [Def.3 Anexo]). Essa metodologia foi aplicada em conformidade com a teoria de Conjuntos Difusos desenvolvida por Zadeh (vide Anexo).

Para problemas de pequeno e médio porte, pode-se verificar através dos índices de performance computacional apresentados na tabela 3.7, que os algoritmos derivados a partir do conceito de relaxação lagrangena tendem a ser mais eficientes. Em particular, o algoritmo "CFT" é o que apresentou o maior índice de eficiência médio (68,8%) seguido do algoritmo "Ha" (42,05%), ambos baseados na técnica de relaxação lagrangeana. Os demais algoritmos obtiveram um índice baixo, com exceção do algoritmo "LL" (39,37%), que é baseado em relaxação *surrogate*, e cujo tempo de pré-processamento não foi computado.

Para problemas maiores, verifica-se na tabela 3.8, que os algoritmos "CFT" e "BJT" são muito superiores em performance computacional aos demais, com índices médios de 91,88% e 96,685%, respectivamente. Dos demais, somente os algoritmos "BeCh" e "JaBr" apresentam índices de performance computacional razoáveis para esse conjunto de testes das Classes E, F, G e H da *OR Library*.

	Beasley	LL	BaCa	BeCh	Ha	CFT
A.1	60	80	0	0	80	63,9
A.2	0	25	50	0	75	65,1
A.3	60	60	0	49,7	60	0
A.4	0	100	0	0	66,6	94,7
A.5	50	0	69,2	72,4	80,5	0
Média A	34	53	23,84	24,42	72,42	44,74
B.1	0	0	0	34,6	10	96,2
B.2	0	100	32,1	79,3	34	94,3
B.3	89,7	0	0	0	0	98,1
B.4	27,8	0	24,5	21,7	22,3	93,9
B.5	78,4	97,3	0	93,2	20,5	100
Média B	39,18	39,46	11,32	45,76	17,36	96,5
C.1	0	100	0	0	66,7	61,9
C.2	0	25	16	40,9	0	2,9
C.3	0	0	37,5	0	75	87,8
C.4	0	0	0	0	20	59,6
C.5	0	50	0	0	50	26,5
Média C	0	35	10,7	8,18	42,34	47,74
D.1	0	100	0	78,1	0	55,5
D.2	0	0	50	0	76,5	90,5
D.3	0	0	33,3	0	66,7	97,8
D.4	0	50	49,1	0	37,1	94,9
D.5	0	0	57,9	0	0	92,3
Média D	0	30	38,06	15,62	36,06	86,2
Média	18,3	39,37	20,98	23,5	42,05	68,8

Tabela 3.7: Eficiência dos Algoritmos pela Análise do Conjunto (tempo, valor) para as classes A, B, C e D

	Be	BaCa	CNS	BeCh	JaBr	Ha	BJT	CFT
E.1	82,4	86,6		90,8	0	54,2	100	93,8
E.2	0	0		0	97,4	0	100	97,4
E.3	0	0		0	98,7	0	100	99,8
E.4	0	50		0	24,6	50	100	95,8
E.5	80,5	85,2		91,7	0	55,7	100	91,3
Média E	32,58	44,36		36,5	44,14	31,98	100	95,62
F.1	0	90		88,6	37,8	0	100	95,3
F.2	0	86,6		88,2	37	0	100	95,5
F.3	0	0		60,6	39,1	0	100	63,3
F.4	0	0		68,9	38,9	0	100	95,9
F.5	0	0		0	0	0	99,1	98,9
Média F	0	35,32		61,26	30,56	0	99,82	89,78
G.1	0	12,5	84,1	0	62,5	37,5	100	100
G.2	0	22	0	89	56	33	89	86
G.3	11	0	38,4	0	56	11	79,6	100
G.4	11	11	78	0	56	0	93,6	99,2
G.5	0	43	0	91,8	94,8	29	100	98,4
Média G	4,4	17,7	40,1	36,16	65,06	22,1	92,44	96,72
H.1	0	0	0	66,3	80	40	80	71,1
H.2	25	0	0	75	75	0	92,4	87,1
H.3	0	33	0	66,7	67	50	100	71,9
H.4	0	20	80	0	80	40	100	99,6
H.5	0	40	0	91,3	92,1	80	100	97,3
Média H	5	18,6	16	59,86	78,82	42	94,48	85,4
Média	10,495	28,995	14,025	48,445	54,645	24,02	96,685	91,88

Tabela 3.8: Eficiência dos Algoritmos pela Análise do Conjunto (tempo, valor) para as classes E, F, G e H

3.3. Outros Problemas

3.3.1. Implantação e Validação de Algoritmos

Com base nos resultados acima, em conformidade com o conjunto de testes da *OR Library* de *Beasley*, os algoritmos heurísticos mais eficientes para resolução do Problema de Cobertura de Conjuntos são aqueles que utilizaram a abordagens de Relaxação Lagrangeana com otimização por subgradientes. Em vista desse fato, resolveu-se implementar computacionalmente essa metodologia de otimização para o PCC e analisar mais detalhadamente a sua robustez quando se aumenta a densidade das matrizes ou proporciona-se uma variação menor nos custos da função objetivo.

Resolveu-se, também, testar um algoritmo genético apresentado na literatura, que não foi testado no conjunto de dados da *OR Library*, mas que apresentou um bom desempenho em problemas reais (vide [WRE95]).

Essas duas abordagens algorítmicas, relaxação Lagrangeana com otimização por subgradientes e algoritmo genético, foram programados no ambiente *Delphi* 4.0, e executados em microcomputador (Pentium 233 Mhz, 64 Mb RAM). Esses dois algoritmos foram implantados e validados, comparando seus resultados com os apresentados na literatura. Conforme as tabelas 3.9 e 3.10, os resultados obtidos por esses algoritmos tanto para valores quanto para tempo, mostram-se consistentes.

Relaxação Lagrangeana				
Problemas	**Ótimo**	**Valor**	**Tempo(s)**	**Erro Max. (%)**
4.1	429	429	13,18	1
4.2	512	512	19,83	1
4.3	516	516	23,78	1
4.4	494	495	12,47	1
4.5	512	512	11,59	1
4.6	560	560	11,32	1
4.7	430	430	10,10	1
4.8	492	492	29,16	1
4.9	641	641	69,70	1
4.10	514	514	7,30	1
5.1	253	255	184,83	2
5.2	302	304	62,45	2
5.3	226	226	6,76	1
5.4	242	244	48,50	1
5.5	211	211	11,92	1
5.6	213	213	11,04	1
5.7	293	295	42,56	2
5.8	288	289	39,11	2
5.9	279	279	21,15	2
5.10	265	265	9,12	2
6.1	138	141	122,27	5
6.2	146	149	474,06	5
6.3	145	145	18,29	5
6.4	131	131	22,03	5
6.5	161	166	786,70	6

Tabela 3.9 : Validação do algoritmo baseado em Relaxação Lagrangeana.

Algoritmo Genético					
Problemas	**Dens.(%)**	**Tamanho**	**Melhor Valor (*)**	**Valor Obtido**	**Tempo(s)**
Wren-01	8,17	200×539	7865 (**)	7856	18,73
Wren-02	3,33	222×5.522	14218	13790	130,33
Wren-03	3,27	219×4.990	30 (***)	30	40,59
Wren-04	1,07	798×4.569	109 (***)	113	64,05

(*) Solução obtida com o auxílio do IMPACS, por pesquisadores da University of Leeds.

(**) Solução publicada em [WRE95].

(***) Problemas de cobertura de conjunto com custos unitários.

Tabela 3.10 - Validação do algoritmo baseado em Algoritmo Genético

Nas implementações realizadas não se fez o pré-processamento das reduções por dominância de linhas e colunas para melhorar a performance dos algoritmo em questão, conforme descrito com maiores detalhes do Capítulo 2.

3.3.2. Problemas Gerados Aleatoriamente

A maior preocupação nessa etapa é a robustez do método, ou seja, a exatidão das soluções quando se variam os custos e a esparsidade da matriz A ($m \times n$), visto que para o conjunto de dados analisado, os algoritmos baseados nessas abordagens obtiveram uma boa média de tempo computacional em comparação com os demais.

A variação dos custo da *OR Library* é de 1 a 100, enquanto que nos testes gerados é na faixa de 80 a 100, conforme descrito na Tabela 3.11. Estes problemas permitem avaliar o Problema de Cobertura de Conjuntos para casos mais próximos ao unicusto, que é um caso mais difícil computacionalmente e comum nas aplicações práticas.

Nome	Dens.(%)	Amplitude	Tamanho
T1	2	80-100	200 x 1000
T2	2	80-100	200 x 1000
T3	2	80-100	200 x 1000
T4	2	80-100	200 x 1000
T5	2	80-100	200 x 1000
T6	2	80-100	200 x 2000
T7	2	80-100	200 x 2000
T8	2	80-100	200 x 2000
T9	2	80-100	200 x 2000
T10	2	80-100	200 x 2000
T11	5	80-100	200 x 1000
T12	5	80-100	200 x 1000
T13	5	80-100	200 x 1000
T14	5	80-100	200 x 1000
T15	5	80-100	200 x 1000
T16	5	80-100	200 x 2000
T17	5	80-100	200 x 2000
T18	5	80-100	200 x 2000
T19	5	80-100	200 x 2000
T20	5	80-100	200 x 2000

Tabela 3.11: Conjunto de Teste Gerados Aleatoriamente

Na tabela 3.12, verificam-se para os problemas gerados, os resultados obtidos com a aplicação dos dois algoritmos testados. Para cada um destes algoritmos, são apresentados o valor da função objetivo (coluna "Solução"), o tempo computacional (coluna "Tempo") e o erro máximo avaliado considerando o limite inferior calculado pelo método da relaxação Lagrangeana.

Pode-se observar que o algoritmo de Relaxação Lagrangeana com otimização por subgradientes é muito sensível com um erro médio (50,3%) muito alto quando se diminui a variação dos custos ou aumenta-se a densidade da matriz *A(mxn)*. Com relação ao algoritmo genético implementado, observa-se que o erro cometido é ligeiramente inferior (40,6%), indicando que sua aplicação é mais favorável para problemas com as características dos testes gerados (tabela 3.11).

			RELAXAÇÃO LAGRANGEANA			ALGORITMO GENÉTICO		
Nome	Melhor	Lim. Inferior	Solução	Tempo	Erro Max (%)	Solução	Tempo	Erro Max (%)
T1	3.219,77	3.219,77	4.210,94	491,0	30,8%	3.798,24	68,9	18,0%
T2	2.526,90	2.526,90	3.279,74	421,6	29,8%	2.997,62	81,2	18,6%
T3	2.928,20	2.928,20	3.893,57	781,2	33,0%	3.800,96	76,8	29,8%
T4	3.247,22	3.247,22	4.119,55	177,7	26,9%	3.807,85	70,0	17,3%
T5	3.187,85	3.187,85	4.189,49	294,1	31,4%	3.869,83	61,4	21,4%
T6	2.950,47	2.950,47	3.950,94	772,2	33,9%	3.601,19	165,5	22,1%
T7	2.666,00	2.244,71	2.928,55	1.114,9	30,5%	2.666,00	109,3	18,8%
T8	2.927,92	2.927,92	3.893,57	2.604,0	33,0%	3.555,77	134,5	21,4%
T9	2.944,37	2.944,37	3.944,62	880,6	34,0%	3.589,85	151,9	21,9%
T10	2.929,19	2.929,19	3.929,26	766,3	34,1%	3.541,46	155,4	20,9%
T11	1.365,98	1.365,98	2.439,16	313,0	78,6%	2.168,01	158,2	58,7%
T12	1.377,89	1.377,89	2.392,81	184,1	73,7%	2.257,46	196,5	63,8%
T13	1.378,77	1.378,77	2.374,62	106,3	72,2%	2.203,51	106,3	59,8%
T14	1.366,89	1.366,89	2.435,13	1.346,0	78,2%	2.243,46	146,5	64,1%
T15	1.365,52	1.365,52	2.295,44	667,8	68,1%	2.202,75	150,4	61,3%
T16	1.377,89	1.377,89	2.392,81	293,0	73,7%	2.116,61	170,8	53,6%
T17	1.025,61	1.025,61	1.709,26	325,0	66,7%	1.628,10	153,5	58,7%
T18	1.354,55	1.354,55	2.158,33	163,6	59,3%	2.138,37	186,9	57,9%
T19	1.348,30	1.348,30	2.192,04	168,2	62,6%	2.166,79	339,0	60,7%
T20	1.355,35	1.355,35	2.108,07	100,8	55,5%	2.198,94	261,8	62,2%
ERRO MËDIO					50,3%			40,6%

Tabela 3.12 : Comparação algoritmo de relaxação Lagrangeana com otimização por subgradientes e algoritmos genéticos.

3.4. Considerações Finais

Conforme os resultados reportados, pode-se ressaltar os seguintes fatos mais relevantes:

Para o relaxamento Lagrangeano, a média dos erros tende a manter-se constante quando se fixa a densidade e o tamanho da matriz A. Verificou-se que os erros tendem a crescer a medida que se aumentam o tamanho ou densidade da matriz A. A razão da divergência para esses testes gerados pode estar no fato da convergência precoce da *função Lagrangeana*. Dessa forma, a heurística gulosa não apresenta melhora porque depende das variáveis duais, que não serão mais alteradas. Quando se homogeneiza os custos ou aumenta-se a densidade da matriz A , a heurística gulosa (conforme definida no capítulo 2) não consegue captar as diferenças entre a relação custo reduzido e número de colunas não cobertas numa dada iteração. Ou seja, essa heurística gulosa torna-se praticamente inoperante. Convém, portanto, aprimorar essa heurística ou simplesmente utilizar uma outra mais efetiva, mais adequada para o caso próximo ao unicusto. No caso unicusto, podem ser utilizadas heurísticas que não sejam tão sensíveis a essas mudanças, como por exemplo, os discutidos em Grossman e Wool (vide [GRO96]).

Para o algoritmo genético implementado, a média dos erros (40,6%) mostra uma maior adequação aos problemas com as características dos testes gerados. Seu comportamento é, ligeiramente, melhor para problemas que os custos são mais homogêneos, e com uma densidade maior na matriz A. Entretanto, para problemas com custos na função objetivo mais heterogêneos (como os testes da OR Library), esse algoritmo mostra-se inadequado. Alguns melhoramentos podem ser feitos para torná-lo mais eficiente. Por exemplo, a sua transformação num algoritmo misto, guloso-genético, agregando nele informações adicionais que outras heurísticas conseguem captar com maior facilidade (por exemplo, a relação de custo por linhas não cobertas nas heurísticas gulosas ou variantes).

CAPÍTULO IV

4. CONCLUSÕES E RECOMENDAÇÕES

4.1. Conclusões

Esse trabalho compreende uma revisão bibliográfica do Problema de Cobertura de Conjuntos e dos algoritmos reportados como os mais eficientes na literatura, métodos heurísticos e exatos, para resolvê-lo. É feita uma revisão histórica do problema, visando os métodos clássicos e de outras que se mostram na atualidade mais eficientes, como a Relaxação Lagrangeana com Otimização por Subgradientes, Algoritmos Genéticos e *Simulated Annealing*, sendo que os dois primeiros receberam uma atenção especial, em face dos resultados obtidos em problemas de otimização combinatorial.

Dentre os algoritmos estudados neste trabalho, os que se revelaram mais eficientes em tempo de execução e robustez, de uma forma geral para o conjunto de testes descritos no capítulo 3, foram os algoritmos que utilizaram a abordagem da Relaxação Lagrangeana com otimização por subgradientes. Devido a este fato, implementou-se computacionalmente um algoritmo baseado neste método para se estudar seu comportamento quando se aumenta a densidade da matriz A e diminui-se a variação dos custos.

Testou-se, também uma versão adaptada do algoritmo genético de Wren (vide [WREN95]) para PCC, com a finalidade de verificar a adaptação desse algoritmo ao conjunto de testes da *OR Library* e o resultado é que ele é altamente desfavorável a esse conjunto de testes,

embora tenha obtido bons resultados para aplicações práticas e resultados ligeiramente melhores para o conjunto de teste gerados.

Os resultados demonstraram que os algoritmos para PCC baseados em relaxação Lagrangeana e otimização por subgradientes são altamente sensíveis a alteração da densidade e esparsidade da matriz A dos coeficientes da restrições, mostrando-se altamente ineficiente para matrizes pouco esparsas ou com maior densidade. No entanto, para matrizes que apresentam pouca esparsidade e variação no custo de 1 a 100, a metodologia mostra-se amplamente favorável.

De uma forma geral, pode-se concluir que os algoritmos heurísticos são muito mais rápidos em tempo de execução que os exatos, embora a sua precisão seja inferior na maioria dos casos. E entre os algoritmos heurísticos, obtiveram melhores performances computacional aqueles que utilizaram algoritmos gulosos, associados ao subgradiente para encontrar limites inferior melhorados na Relaxação Lagrangeana.

Ainda relativamente a algoritmos heurísticos, existe uma classe muito promissora, a dos Algoritmos Genéticos, que embora apresentem custo computacional maior que as que utilizaram métodos de relaxações, devem apresentar uma melhora sensível se forem incorporadas técnicas mistas dos algoritmos analisados e descritos. O algoritmo genético de Beasley e Chu (vide [BEA96]), por exemplo, é altamente robusto, porém, caro computacionalmente. Heurísticas mais específicas devem ser agregadas aos algoritmos genéticos para ganho de velocidade de tempo de execução. Outra possibilidade seria a utilização de métodos mistos.

4.2. Recomendações

Existem uma série de modificações que se incorporadas aos algoritmos baseados em relaxação Lagrangeana poderão aumentar a sua performance. O principal modificação é utilização de heurísticas diferentes das heurísticas comumente usadas, isto é, as gulosas.

Deve-se incorporar heurísticas ao modelo capazes de captar a relação custo das colunas por número de linhas que elas podem cobrir de uma forma menos gulosa e mais direcionada. Uma forma para atingir esse objetivo é utilizar heurísticas que observam os custos reduzidos Lagrangeanos em relação ao número de linhas não cobertas que uma dada coluna pode cobrir, ou adaptações delas, como as heurísticas descritas por Fisher e Kedia [vide [FIS90]) e Caprara et. al. [CAP95]).

Outra questão não esclarecida é o tamanho do passo. Até a atualidade não se sabe ao certo os critérios mais corretos para a tomada dos tamanhos de passos. Uma tentativa de melhora nos modelos de algoritmos baseados em relaxação Lagrangeana seria a utilização de tamanhos de passos que se modificassem dinamicamente, de acordo com algum critério.

Quanto ao modelo de algoritmo genético implantado, baseado no trabalho de Wren (vide [WEN95]) verificou-se que para problemas que gozam de estrutura mais homogênea, o algoritmo tem maior facilidade de encontrar soluções heurísticas melhoradas. O passo de reprodução dos pais, feito de uma maneira mais aleatória, deve ser modificado para modelos que visam captar mudanças mais sensíveis na relaxação custo por número de linhas não cobertas.

BIBLIOGRAFIA

[AGM54] AGMON, S. (1954). "The relaxation method for linear inequalities". *Canad. J. Math.* 6, 382-392.

[ALS96] AL-SULTAN, K.S., HUSSAIN, M.F. e NIZAMI, J.S. (1996). "A Genetic Algorithm for the Set Covering Problem". *Journal of the Operational Research Society,* 47: 702-709.

[AOU94] AOURID, M.; KAMINSKA, B. (1994). "Neural Networks for the Set Covering Problem: An Application to the Test Vector Compaction". IEEE: 4645-4649.

[BAC93] BÄCK, T.(1993). "Optimal mutation rates in genetic search", in: S. Forrest, ed., *Proc. Fifth International Conference on Genetic Algorithm.,* Morgan Kaufmann, San Mateo, CA, 2-9.

[BAK81] BAKER, E.K. (1981). "Heuristic Algorithms for the Weighted Set Covering Problem", *Computers and Operations Research*, 8(4), 303-310.

[BAL65] BALINSKI, M.L. (1965). "Integer programming: methods, uses, computation". *Management Science,* 12(3): 253-313.

[BAL70] BALAS, E.; PADBERG, M.W. (1970). "On the Set-Covering Problem". *Mathematical Programming,* 1152-1161.

[BAL80] BALAS, E.; HO, A. (1980). "Set covering algorithms using cutting planes, heuristics and subgradient optimization: a computational study". *Mathematical Programming* Study 12: 37-60.

[BAL80a] BALAS, E. (1980). "Cutting planes from conditional bounds: a new approach to set covering". *Mathematical Programming Study* 12: 19-36.

[BAL80b] BALAS, E.; ZEMEL, E. (1980). "An algorithm for large zero-one knapsack problems". *Operations Research,* 28(5): 1130- 1154.

[BAL96] BALAS, E.; CARRERA, M.C. "A dynamic subgradient-based branch-and-bound procedure for set covering". *Operations Research,* 44(6): 875-890.

[BEA87] BEASLEY, J.E. (1987). "An algorithm for set covering problem". *European Journal of Operational Research* , 31: 85-93.

[BEA90] BEASLEY, J.E. (1990)."A Lagrangian heuristic for set covering problems". *Naval Research Logistics,* 37: 145-164.

[BEA90a] BEASLEY, J.E. (1990). "OR-Library: Distributing tests problems by electronic mail". *Journal of Operations Research Society.* London, Imperial College, Management School.

[BEA92] BEASLEY, J.E ; JORNSTEN, K. (1992). "Enhancing an algorithm for set covering problems". *European Journal of Operational Research,* 58: 293-300, 1992.

[BEA96] BEASLEY, J.E. ; CHU, P.C. (1996). "A genetic algorithm for the set covering problem". *European Journal of Operational Research,* 94: 392-404.

[BEL71] BELMORE, M. ; RATLIFF, H.D.(1971). "Set covering and involuntary bases". *Management Science,* 18: 194-206.

[CAP95] CAPRARA, A.; FISCHETTI, M.; TOTH, P. (1995). "A Heuristic Method for the Set Covering Problem". *Technical Report OR-95-8*, DEIS, Univesity of Bologna.

[CAP98] CAPRARA, A.; FISCHETTI, M.; TOTH, P. (1998). "Effective Solution of the LO Relaxação of Set Covering Problems". *Technical Report*, DEIS, University of Bologna.

[CER95] CERIA, S.; NOBILI, P.; SASSANO, A. (1998). "A Lagrangian-based heuristic for large-scale set covering problems". *Mathematical Programming,* 81: 215-228.

[CHR75] CHRISTOFIDES, N.; KORMAN, S. (1975). "A computational survey of methods for the set covering problem". *Management Science,* 21(5): 591-599.

[CHR75a] CHRISTOFIDES, N. (1975). *Graph Theory: An Algorithm Approach.* Academic Press. Inc..

[CHV79] CHVÁTAL, V. (1979). "A greedy heuristic for the set covering problem". *Mathematics of Operations Research,* 4: 233-235.

[DON96] DONGARRA, J. J. (1996). "Performance of Various Computers Using Standard Linear Equations Software". Technical Reports No. CS-89-85, Computer Science Department, University of Tennessee, January 1996.

[DUD87] DUDZINSKI, K.; WALUKIEWICZ, S. (1987). "Exact methods for the knapsack problem and its generalizations". *European Journal of Operational Research,* 28: 3-21.

[ELD92] EL-DARZI, E.; MITRA, G. (1992). "Solution of Set-Covering and Set-Partitioning Problems Using Assignment Relaxations". *Journal of the Operational Research Society,* 43(5): 483-493.

[ELD95] EL-DARZI, E.; MITRA, G. (1995). "Graph theoretic relaxations of set covering and set partitioning problems". *European Journal of Operational Research,* 87: 109-121.

[ETC77] ETCHEBERRY, J. (1977) "The set covering problem: a new implicit enumeration algorithm". *Operational Research,* 25: 760-772.

[FEO89] FEO, T.A. ; RESENDE, M.G.C. (1989). "A probabilistic heuristic for a computationally difficult set covering problem". *Operations Research Letters,* 8: 67-71.

[FIS81] FISHER, M. L. (1981) "The Lagrangian relaxation method of solving integer programming problems. *Management Science,* 27(1): 1-18.

[FIS85] FISHER, M.L (1985)."An Applications Oriented Guide to Lagrangian Relaxation". *Interfaces,* 15(2): 10-21.

[FIS89] FISHER, M.L. ; ROSENWEIN, M. B. (1989) "An interactive optimization system for bulk cargo ship scheduling". *Naval Research Logistics Quarterly,* 36: 27-42.

[FIS90] FISHER,M.L.; KEDIA, P. (1990). "Optimal solution of set covering/ partitioning problems using dual heuristics". *Management Science,* 36(6): 674-688.

[GAR69] GARFINKEL, R.S.; NEMHAUSER, G.L. (1969). "The set partitioning problem: set covering with equality constraints". *Operations Research,* 17: 848-856.

[GAR70] GARFINKEL, R.S. ; NEMHAUSER, G.L. (1970). "Optimal political districting by implicit enumeration techniques". *Management Science,* 16: B495-B508.

[GAR72] GARFINKEL, R.S. ; NEMHAUSER, G.L. (1972). *Integer Programming.* New York, John Wiley.

[GAR72] GARFINKEL, R.S.; NEMHAUSER, G.L.(1972). "Optimal Set Covering: A Survey". *Perspective in Optimization: A Collection of Expository Articles* (ed.: Geoffrion), Reading, Mass., Addison-Wesley.

[GAR79] GAREY, M.R.; JOHNSON, D.S. (1979). *Computer and intractability – a guide to the theory of NP-completeness.* San Francisco, W.H. Freeman and Company

[GEO69] GEOFFRION, A.M. (1969). "An improved implicit enumeration approach for integer programming". *Operations Research,* 17: 437-454.

[GEO74] GEOFFRION, A.M. (1974). "Lagrangian Relaxation end its Uses in Integer Programming". Math. Programming Study, Vol. 2 , 82-114.

[GOL89] GOLDBERG, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning,* Adison-Wesley.

[GRE85] GREFENSTETTE, J. J.; GOPAL R.; ROSMAITA, B.J.; GUCHT D.V. (1985). "Genetic Algorithms for the travelling salesman problem". In Grefenstette (ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Lawrence Erlbaum, Hillside, NJ, 160-168.

[HAD97] HADDADI, S. (1997). "Simple Lagrangian heuristic for the set covering problem". *European Journal of Operational Research,* 97: 200-204.

[HAL83] HALL, N. G; HOCHBAUM D. S. (1983). "A Fast Approximation Algorithm for Multicovering Problem". *Discrete Appl. Math.* 15, 35-40.

[HAR94] HARCHE, F.; THOMPSON, G.L. (1994). "The column subtraction algorithm: an exact method for solving weighted set covering, packing and partitioning problems". *Computers Ops Res.,*21(6): 689-705.

[HEL70] HELD, M. ; KARP, R.M. (1970). "The travelling salesman problem and minimum spanning trees". *Mathematical Programming,* 18: 1138-1162.

[HEL71] HELD, M. ; KARP, R.M. (1971). "The travelling salesman problem and minimum spanning trees: part II". *Mathematical Programming,* 1: 6-25.

[HEL74] HELD, M.; WOLF, P.; CROWDER, H.P. "Validation of subgradient optimization". *Mathematical Programming,* 6: 62-88.

[HOA82] HO, A. C. (1982). "Worst Case Analysis of a Class of Set Covering Heuristics", Mathematical Programming, 23(2), 170-180.

[HOL75] HOLLAND, H. J. (1975). "Adaption in Natural and Artificial Systems", MIT Press, Cambridge, MA.

[JAC93] JACOB L. W.; BRUSCO M. J. (1993). "A simulated annealing based heuristic for the set-covering problem", Working paper, Operations Management and Information Systems Department, Northern Illinois University, Dekalb, IL.

[JAC95] JACOB L. W.; BRUSCO M. J. (1995). "A Local Search for Large Set-Covering Problems", *Naval Research Logistics* 52, 1129-1140.

[JAC96] JACOB L. W.; BRUSCO M. J; THOMPSON G. M. (1996). "A Morphing Procedure to Supplement a Simulated Annealing Heuristic for Cost and Coverage–Correlated Weighted Set–Covering Problems", Working paper, Operations Management and Information Systems Department, Northern Illinois University.

[LEM72] LEMKE, C.E.; SALKIN, H. M.; SPIELBERG, K. (1972). "Set covering by single branch enumeration with linear programming subproblems". *Operations Research,* 19: 998-1022.

[LOR94] LORENA, L.; LOPES, F.B. (1994). "A surrogate heuristic for set covering problems". *European Journal of Operational Research,* 79: 138-150.

[LOR97] LORENA, L.; LOPES, L. S. (1997). "Genetic algorithms applied to computationally difficult set covering problems". *Journal of the Operational Research Society,* 48: 440-445.

[MAR74] MARSTEN, R.E. (1974). "An algorithm for large set partitioning problems". *Management Science* , 20(5): 774-787.

[MAR90] MARTELLO, S; TOTH, P. (1990). "Knapsack Problems: Algorithms and Computer Implementations", J. Wiley and Sons

[MOT54] MOTZKIN, T.; SCHOENBERG, I. J.(1954). "The relaxation method for linear inequalities". *Canad. J. Math.* 6, 393-404.

[POL67] POLYAK, B.T. (1967). "A general method for solving extremal problemas". *Soviet. Math. Dokl.* 8, 593-597.

[POL69] POLYAK, B.T. (1969). "Minimization of unsmooth functionals". USSR *Comput. Math. Math. Phys.* 9, 509-521.

[RIC89] RICHARDSON, J.; PALMER M.; LIEPINS, G.; HILIARD M. (1989). "Some guidelines for genetic algorithms with penalty funções". In: J. Schaffer, ed.., *Proc. Third International Conference on Genetic Algorithms*, 191-197.

[ROT69] ROTH, R. (1969). "Computer Solutions to Minimum-Cover Problems" , *Operations Research*, 17(3), 455-465

[VAS84] VASKO, F.J.; WILSON, G.R.. (1984). "An efficient heuristic for large set covering problems". *Naval Research Logistics Quarterly,* 31: 163-171.

[VAS86] VASKO, F.J.; WILSON, G.R. (1986) "Hybrid heuristics for Minimum Cardinality Set Covering Problems". *Naval Research Logistics Quarterly,* 33.

[VIG91] VIGNAUX G. A; MICHALEWICX Z. (1991). "A genetic algorithm for the linear transportation problem". *IEEE Trans. Systems, Man and Cybernetics*, 21, 445-452.

[WRE95] WREN, A. ; WREN, D. O. (1995). "A Genetic Algorithm for Public Transport Driver Scheduling". *Computers Ops. Res.*, Vol. 22, No. 1, 101–110.

[ZAH65] ZADEH, L. A (1965). Fuzzy sets; *Inf. Control* 8, 338-353.

[ZAH75] ZADEH, L. A. (1975). "The concept of linguistic variable and its application to approximate reasoning". Parts 1 e 2, *Inf. Science*, 8, 199-249; Part 3, *Inf. Science,* 9, 43-80.

APÊNDICE:

CÓDIGO EM *OBJECT* PASCAL DOS ALGORITMOS IMPLEMENTADOS

Unit USetCov

```
Unit UsetCov;
Interface
Uses Classes,StdCtrls,SysUtils,Dialogs,Forms;

Type
  PTElem = ^TElem;
  TElem = object
    Lin,Col                  : Integer;
    Valor                    : Single;
    NextElemCol,NextElemLin  : PTElem;
    Constructor Init (ILin,ICol : Integer; IVal : Single);
    Destructor Done;
  end;

  PTLinha = ^TLinha;
  TLinha = Record
    B,Dual,Grad,SobreCob  : Single;
    First                 : PTElem;
  end;

  PTColuna = ^TColuna;
  TColuna = Record
    Numero,NElem    : Integer;
    ScoreFix,
    C,CReduz        : Single;
    EstaNaSolOtima,
    EstaNaSolucao   : Boolean;
    First           : PTElem;
  end;

  PTSetCov = ^TSetCov;
  TSetCov = object
    Linha       : Array of PTLinha;
    Coluna      : Array of PTColuna;
    Ordem       : Array of Integer;
    Memo        : TMemo;
    kPenal,N_1,
    M_1,
    M,N,NElem   : Integer;
    ZSup,ZInf   : Single;
    Constructor InitLendo (NomeArq : String);
    Destructor Done;
    Procedure Incluir (i,j : Integer; UmValor : Single);
    Procedure ApresentarColuna (Var Arq : TextFile; j : Integer);
    Procedure ApresentarTodasColunas (Var Arq : TextFile; EhSolucaoOtima : Boolean);
    Procedure ApresentarLinha (Var Arq : TextFile; i : Integer);
    Procedure ApresentarTodasLinhas (Var Arq : TextFile);
    Procedure SubGradiente (InitMemo : TMemo);
    Procedure HeuristicaGulosa;
    Procedure Ordena;
    Procedure CalculaScore;
    Procedure InicializaDuais;
    Procedure EncontrarSolucaoNaoRedundante;
    Procedure CalculaSobrecobertura;
    Function SolucaoMelhorou : Boolean;
    Procedure CalculaCustoReduzido;
    Function CalculaLagrange : Single;
    Function CalculaNormaSubGradiente : Single;
  end;

Var
  SCP                                  : PTSetCov;
  IterModMax,IterMax                   : Integer;
  Perturbacao,Lambda,ErroPercMax,Epsilon : Single;
  Stop                                 : Boolean;
  UsaFatorAleatorio                    : Boolean;

Implementation
```

```
{-------- Objeto TElem ------------------------------------}

Constructor TElem.Init (ILin,ICol : Integer; IVal : Single);
begin
  Lin := ILin - 1;
  Col := ICol - 1;
  Valor := IVal;
end;

Destructor TElem.Done;
begin
end;

{-------- Objeto TSetCov -----------------------------------}

Constructor TSetCov.InitLendo (NomeArq : String);
Var
  Arq          : TextFile;
  NN,MM,
  i,j          : Integer;
  UmCusto      : Single;
  LIXO         : String [7];
begin
  AssignFile (Arq,NomeArq);
  Reset (Arq);
  Readln (Arq,LIXO,M);
  Readln (Arq,LIXO,N);
  Readln (Arq);
  MM := M;
  NN := N;

  SetLength(Linha,M);
  SetLength(Ordem,N);
  SetLength(Coluna,N);
  For i := 0 to M-1 do
  begin
    New (Linha [i]);
    Linha [i].First := Nil;
    Linha [i].B     := 1;
  end;
  For j := 0 to N-1 do
  begin
    New (Coluna [j]);
    Coluna [j].First := Nil;
    Coluna [j].Numero := j + 1;
    Coluna [j].C     := 0;
  end;

  NElem := 0;
  While not Eof (Arq) do
  begin
    Read (Arq,j,UmCusto);
    If j <= N then Coluna [j-1].C := UmCusto;
    If j > NN then NN := j;
    While not Eoln (Arq) do
    begin
      Read (Arq,i);
      If (i <= M) and (j <= N) then Incluir (i,j,1);
      If i > MM then MM := i;
      NElem := NElem + 1;
    end;
    Readln (Arq);
  end;

  CloseFile (Arq);
  N_1 := N - 1;
  M_1 := M - 1;
  M := MM;
  N := NN;
```

```
end;

Destructor TSetCov.Done;
Var
  NextElem,UmElem : PTElem;
  i               : Integer;
begin
  For i := 0 to M_1 do
  begin
    UmElem := Linha [i].First;
    While UmElem <> nil do
    begin
      NextElem := UmElem.NextElemLin;
      Dispose (UmElem,Done);
      NElem := NElem - 1;
      UmElem := NextElem;
    end;
  end;
  Linha := Nil;
  Coluna := Nil;
  N := 0;
  M := 0;
end;

Procedure TSetCov.Incluir (i,j : Integer; UmValor : Single);
Var
  UmElem : PTElem;
begin
  New (UmElem,Init(i,j,UmValor));

  UmElem.NextElemCol := Coluna [j-1].First;
  Coluna [j-1].First := UmElem;

  UmElem.NextElemLin := Linha [i-1].First;
  Linha [i-1].First  := UmElem;
end;

Procedure TSetCov.ApresentarColuna (Var Arq : TextFile; j : Integer);
Var
  UmElem      : PTElem;
begin
  Write (Arq,Coluna[j-1].Numero:5,Coluna [j-1].C:8:2,' ',Coluna [j-1].CReduz:8:2,' ',
            Coluna [j-1].ScoreFix:8:2,' ');
  UmElem := Coluna [j-1].First;
  While UmElem <> Nil do
  begin
    Write (Arq,(UmElem^.Lin + 1):5);
    UmElem := UmElem^.NextElemCol;
  end;
  Writeln (Arq);
end;

Procedure TSetCov.ApresentarTodasColunas (Var Arq : TextFile; EhSolucaoOtima : Boolean);
Var
  j         : Integer;
begin
  If EhSolucaoOtima
  then For j := 0 to N_1 do
       begin
         If Coluna[j].EstaNaSolOtima then ApresentarColuna (Arq,j+1);
       end
  else For j := 0 to N_1 do ApresentarColuna (Arq,j+1);
end;

Procedure TSetCov.ApresentarLinha (Var Arq : TextFile; i : Integer);
Var
  UmElem      : PTElem;
begin
  Write (Arq,i:5,Linha[i-1].B:8:2,' ');
  UmElem := Linha [i-1].First;
```

```
  While UmElem <> Nil do
  begin
    Write (Arq,(UmElem^.Col + 1):5);
    UmElem := UmElem^.NextElemLin;
  end;
  Writeln (Arq);
end;

Procedure TSetCov.ApresentarTodasLinhas (Var Arq : TextFile);
Var
  i         : Integer;
begin
  For i := 1 to M do ApresentarLinha (Arq,i);
end;

Procedure TSetCov.CalculaScore;
Var
  j       : Integer;
  UmElem  : PTElem;
begin
  For j := 0 to N_1 do
  begin
    Coluna[j].NElem := 0;
    UmElem := Coluna[j].First;
    While UmElem <> Nil do
    begin
      Coluna[j].NElem := Coluna[j].NElem + 1;
      UmElem := UmElem.NextElemCol;
    end;
    Coluna[j].ScoreFix := Coluna[j].C / Coluna[j].NElem;
  end;
end;

Procedure TSetCov.InicializaDuais;
Var
  i,j     : Integer;
  UmElem  : PTElem;
begin
  For i := 0 to M_1 do
  begin
    UmElem := Linha[i].First;
    Linha [i].Dual := 1E+99;
    While UmElem <> Nil do
    begin
      j := UmElem.Col;
      If Coluna[j].ScoreFix <= Linha [i].Dual then Linha [i].Dual := Coluna[j].ScoreFix;
      UmElem := UmElem.NextElemLin;
    end;
  end;
end;

Procedure TSetCov.EncontrarSolucaoNaoRedundante;
Var
  j,i                : Integer;
  PodeSerEliminada   : Boolean;
  UmElem             : PTElem;
begin
  For j := 0 to N_1 do If Coluna [Ordem[j]].EstaNaSolucao then
  begin
    PodeSerEliminada := True;
    UmElem := Coluna[Ordem[j]].First;
    While UmElem <> Nil do
    begin
      i := UmElem.Lin;
      PodeSerEliminada := PodeSerEliminada and (UmElem.Valor <= Linha[i].SobreCob);
      UmElem := UmElem.NextElemCol;
    end;
    If PodeSerEliminada then
    begin
      Coluna [Ordem[j]].EstaNaSolucao := False;
```

```
      UmElem := Coluna[Ordem[j]].First;
      While UmElem <> Nil do
      begin
        i := UmElem.Lin;
        Linha[i].SobreCob := Linha[i].SobreCob - UmElem.Valor;
        UmElem := UmElem.NextElemCol;
      end;
    end;
  end;
end;

Procedure TSetCov.CalculaSobrecobertura;
Var
  i,j                : Integer;
  UmElem             : PTElem;
begin
  { Calcular a sobrecobertura de cada linha e identificar linhas descobertas }
  For i := 0 to M_1 do
  begin
    Linha[i].SobreCob := - Linha[i].B;
    UmElem := Linha[i].First;
    While UmElem <> Nil do
    begin
      j := UmElem.Col;
      If Coluna [j].EstaNaSolucao
      then Linha[i].SobreCob := Linha[i].SobreCob + UmElem.Valor;
      UmElem := UmElem.NextElemLin;
    end;
  end;
end;

Function TSetCov.SolucaoMelhorou : Boolean;
Var
  Soma   : Single;
  j      : Integer;
begin
  Result := False;
  Soma := 0;
  For j := 0 to N_1 do If Coluna [j].EstaNaSolucao then Soma := Soma + Coluna [j].C;
  If Soma < ZSup then
  begin
    ZSup := Soma;
    Result := True;
    For j := 0 to N_1 do Coluna [j].EstaNaSolOtima := Coluna [j].EstaNaSolucao;
  end;
end;

Procedure TSetCov.CalculaCustoReduzido;
Var
  j,i         : Integer;
  UmElem      : PTElem;
begin
  For j := 0 to N_1 do
  begin
    Coluna [j].CReduz := Coluna [j].C;
    UmElem := Coluna [j].First;
    While UmElem <> Nil do
    begin
      i := UmElem.Lin;
      Coluna [j].CReduz := Coluna [j].CReduz - Linha[i].Dual * UmElem.Valor;
      UmElem := UmElem.NextElemCol;
    end;
  end;
end;

Function TSetCov.CalculaLagrange : Single;
Var
  i,j      : Integer;
begin
  Result := 0;
```

```
  For j := 0 to N_1 do
  begin
    Coluna [j].EstaNaSolucao := (Coluna [j].CReduz <= 0);
    If Coluna [j].EstaNaSolucao then Result := Result + Coluna [j].CReduz;
  end;
  For i := 0 to M_1 do Result := Result + Linha [i].Dual * Linha [i].B;
end;

Function TSetCov.CalculaNormaSubGradiente : Single;
Var
  i,j         : Integer;
  UmElem      : PTElem;
begin
  Result := 0;
  For i := 0 to M_1 do
  begin
    Linha [i].Grad := Linha [i].B;
    UmElem := Linha [i].First;
    While UmElem <> nil do
    begin
      j := UmElem.Col;
      If Coluna[j].EstaNaSolucao then Linha [i].Grad := Linha [i].Grad - UmElem.Valor;
      UmElem := UmElem.NextElemLin;
    end;
    Result := Result + Linha [i].Grad * Linha [i].Grad;
  end;
end;

Procedure TSetCov.SubGradiente (InitMemo : TMemo);
Var
  Iter,IterMod,i,j,k   : Integer;
  Norma2,Passo,
  Dif,ErroPerc,
  Lagrange,PiorLagr    : Single;
  Fim,Inicio           : TDateTime;
  IndiceStr,
  Texto                : String;
begin
  Inicio := Now;
  Stop := False;
  Memo := InitMemo;
  Memo.Clear;
  { Inicialização }
  ZInf   := 0;
  ZSup := +1E+30;
  Lambda := 2;
  Iter := 0;
  IterMod := 0;
  For j := 0 to N_1 do Coluna [j].EstaNaSolucao := True;
  CalculaScore;
  Ordena;
  InicializaDuais;
  CalculaSobrecobertura;
  EncontrarSolucaoNaoRedundante;

  { Passo iterativo }
  Repeat
    Iter := Iter + 1;
    IterMod := IterMod + 1;
    If SolucaoMelhorou then
    begin
      ErroPerc := ((Zsup - Zinf) / ZSup);
      Memo.Lines.Add ('ZSup = ' + IntToStr (Iter) + '  ' +
                                  FloatToStrF (ZSup,ffNumber,10,2) + '  ' +
                                  FloatToStrF (ZInf,ffNumber,10,2) + '  ' +
                                  FloatToStrF (ErroPerc * 100,ffNumber,10,2) + '% ');
    end;
    CalculaCustoReduzido;
    Lagrange := CalculaLagrange;
```

```
  { Atualiza o limite inferior }
  If IterMod = 1 then PiorLagr := Lagrange
                 else If PiorLagr > Lagrange then PiorLagr := Lagrange;
  If Lagrange > ZInf then begin
                            ZInf := Lagrange;
                            IterMod := 0;
                          end;

  If (Iter mod 50) = 0
  then Memo.Lines [Memo.Lines.Count-1] := Memo.Lines [Memo.Lines.Count-1] + ':'
  else If (Iter mod 10) = 0
       then Memo.Lines [Memo.Lines.Count-1] := Memo.Lines [Memo.Lines.Count-1] + '-';

  If (Iter mod 200) = 0 then
  begin
    ErroPerc := ((Zsup - Zinf) / ZSup);
    Memo.Lines.Add ('ZInf = ' + IntToStr (Iter) + '  ' +
                                FloatToStrF (ZSup,ffNumber,10,2) + '  ' +
                                FloatToStrF (ZInf,ffNumber,10,2) + '  ' +
                                FloatToStrF (ErroPerc * 100,ffNumber,10,2) + '% ');
  end;

  Norma2 := CalculaNormaSubGradiente;

  { Atualiza Lambda }
  If IterMod >= IterModMax
  then begin
         Dif := (ZInf - PiorLagr) / ZInf;
         If Dif > 0.01 then Lambda := Lambda / 2;
         If Dif < 0.001 then Lambda := Lambda * 1.5;
         IterMod := 0;
       end;

  { Calcula o passo na direção do sub-gradiente e Re-calcula o valor da variável Dual }
  Passo := Lambda * (Perturbacao * Zsup - ZInf) / Norma2;
  For i := 0 to M_1 do
  begin
    Linha [i].Dual := Linha[i].Dual + Passo * Linha[i].Grad;
    If Linha[i].Dual < 0 then Linha[i].Dual := 0;
  end;

  HeuristicaGulosa;
  Application.ProcessMessages;
  ErroPerc := ((Zsup - Zinf) / ZSup);
Until (ErroPerc < ErroPercMax) or
      (Iter > IterMax) or Stop;

Fim := Now;
ErroPerc := ((Zsup - Zinf) / ZSup);
Memo.Lines.Add ('------------------------------------------------------------');
Memo.Lines.Add ('FIM =  ' + IntToStr (Iter) + '  ' +
                            FloatToStrF (ZSup,ffNumber,10,2) + '  ' +
                            FloatToStrF (ZInf,ffNumber,10,2) + '  ' +
                            FloatToStrF (ErroPerc * 100,ffNumber,10,2) + '%');
Memo.Lines.Add ('------------------------------------------------------------');
Memo.Lines.Add ('Melhor Solucao Encontrada');
Memo.Lines.Add ('------------------------------------------------------------');
Texto := '';
k := 0;
For j := 0 to N_1 do
begin
  If Coluna [j].EstaNaSolOtima
  then begin
         k := k + 1;
         Str (j+1:6,IndiceStr);
         Texto := Texto + IndiceStr;
         If k = 10 then begin
                          Memo.Lines.Add (Texto);
                          k := 0;
                          Texto := '';
```

```
                        end;
        end;
  end;
  If k > 0 then Memo.Lines.Add (Texto);
  Memo.Lines.Add ('------------------------------------------------------------');
  Memo.Lines.Add ('Tempo = '+ FloatToStrF(24*3600*(Fim-Inicio),ffNumber,10,2) + '
segundos');
end;

Procedure TSetCov.HeuristicaGulosa;
Var
  jMin,
  i,j,NumLinDesc    : Integer;
  CobAdic,Falta,
  ScoreMin,Score    : Single;
  UmElem            : PTElem;
  EhNegativo,
  EhPositivo        : Boolean;
  FatorAleatorio    : Real;
begin
  For j := 0 to N_1 do Coluna [j].EstaNaSolucao := False;
  CalculaSobreCobertura;
  NumLinDesc := 0;
  For i := 0 to M_1 do If Linha[i].SobreCob < 0 then NumLinDesc := NumLinDesc + 1;
  While NumLinDesc > 0 do
  begin
    { Calcular o Score mínimo e identificar coluna }
    jMin := -1;
    ScoreMin := +1E+30;
    For j := 0 to N_1 do If not Coluna[j].EstaNaSolucao then
    begin
      CobAdic := 0;
      UmElem := Coluna [j].First;
      While UmElem <> Nil do
      begin
        i := UmElem.Lin;
        Falta := -Linha [i].SobreCob;
        If Falta > UmElem.Valor then Falta := UmElem.Valor;
        If Falta > 0 then CobAdic := CobAdic + Falta;
        UmElem := UmElem.NextElemCol;
      end;
      If CobAdic > 0 then
      begin
        If Coluna [j].CReduz < 0
        then Score := Coluna [j].CReduz * CobAdic
        else Score := Coluna [j].CReduz / CobAdic;

        If jMin < 0
        then begin
               ScoreMin := Score;
               jMin := j;
             end
        else begin
               If (Score < ScoreMin) then
               begin
                 ScoreMin := Score;
                 jMin := j;
               end;
             end;
      end;
    end;
    If jMin >= 0
    then begin
           { Incluir coluna na solução }
           Coluna [jMin].EstaNaSolucao := True;
           UmElem := Coluna [jMin].First;
           While UmElem <> Nil do
           begin
             i := UmElem.Lin;
             EhNegativo := Linha[i].SobreCob < 0;
```

```
            Linha [i].SobreCob := Linha [i].SobreCob + UmElem.Valor;
            EhPositivo :=  Linha[i].SobreCob >= 0;
            If EhNegativo and EhPositivo then NumLinDesc := NumLinDesc - 1;
            UmElem := UmElem.NextElemCol;
          end;
        end
    else begin
           MessageDlg ('Não existe solução para o problema',mtInformation,[mbOK],0);
           Halt;
         end;
  end;
  EncontrarSolucaoNaoRedundante;
end;

Procedure TSetCov.Ordena;
Var
  Trocou              : Boolean;
  k,OrdAux,j          : Integer;
begin
  For j := 0 to N_1 do
  begin
    Ordem [j] := j;
    k := j;
    Trocou := True;
    While (k > 0) and Trocou do
    begin
      If Coluna [Ordem [k]].ScoreFix > Coluna [Ordem [k-1]].ScoreFix
      then begin
             OrdAux := Ordem [k];
             Ordem [k] := Ordem [k-1];
             Ordem [k-1] := OrdAux;
           end
      else Trocou := False;
      k := k - 1;
    end;
  end;
end;

end.
```

Unit Uconst

```
Unit UConst;
Interface

Const
  Mmax         =    800;
  Mred         =    650;
  Nmax         =   6000;
  CompLabel    =      5;
  ProbMutacao  =   0.02;

Type
  PTVetReal = ^TVetReal;
  TVetReal = Array [1..Nmax] of Real;

  PTVetInt = ^TVetInt;
  TVetInt = Array [1..Mmax] of Integer;

  PTVetIntRed = ^TVetIntRed;
  TVetIntRed = Array [1..Mred] of Integer;

Implementation

end.
```

Unit UElem

```
Unit UElem;
Interface
Uses UConst;

Type
  PTElem = ^TElem;
  TElem = object
    Lin,Col : Integer;
    LastElemLin,LastElemCol,NextElemCol,NextElemLin : PTElem;
    Constructor Init (InitLin,InitCol : Integer);
    Destructor Done;
    Procedure SetNextElemCol (UmElem : PTElem);
    Procedure SetNextElemLin (UmElem : PTElem);
    Procedure SetLastElemCol (UmElem : PTElem);
    Procedure SetLastElemLin (UmElem : PTElem);
    Function GetNextElemCol : PTElem;
    Function GetNextElemLin : PTElem;
    Function GetLastElemCol : PTElem;
    Function GetLastElemLin : PTElem;
    Function GetLin : Integer;
    Function GetCol : Integer;
  end;

  PTVetElem = ^TVetElem;
  TVetElem = Array [1..Nmax] of PTElem;

Implementation

Constructor TElem.Init (InitLin,InitCol : Integer);
begin
  Lin := InitLin;
  Col := InitCol;
  NextElemLin := Nil;
  NextElemCol := Nil;
  LastElemLin := Nil;
  LastElemCol := Nil;
end;

Destructor TElem.Done;
begin
end;

Procedure TElem.SetNextElemCol (UmElem : PTElem);
begin
```

```
  NextElemCol := UmElem;
end;

Procedure TElem.SetNextElemLin (UmElem : PTElem);
begin
  NextElemLin := UmElem;
end;

Procedure TElem.SetLastElemCol (UmElem : PTElem);
begin
  LastElemCol := UmElem;
end;

Procedure TElem.SetLastElemLin (UmElem : PTElem);
begin
  LastElemLin := UmElem;
end;

Function TElem.GetNextElemCol : PTElem;
begin
  GetNextElemCol := NextElemCol;
end;

Function TElem.GetNextElemLin : PTElem;
begin
  GetNextElemLin := NextElemLin;
end;

Function TElem.GetLastElemCol : PTElem;
begin
  GetLastElemCol := LastElemCol;
end;

Function TElem.GetLastElemLin : PTElem;
begin
  GetLastElemLin := LastElemLin
end;

Function TElem.GetLin : Integer;
begin
  GetLin := Lin;
end;

Function TElem.GetCol : Integer;
begin
  GetCol := Col;
end;

end.
```

Unit ULabel

```
Unit ULabel;
Interface
USes UConst;

Type

  TLabel = String [CompLabel];

  PTListaLabel = ^TListaLabel;
  TListaLabel = object
    Nome : Array [1..Nmax] of TLabel;
    N : Integer;
    Constructor Init;
    Destructor Done;
    Function GetIndice (UmLabel : TLabel) : Integer;
    Function GetNome (Indice : Integer) : TLabel;
    Function GetN : Integer;
    Procedure Apresentar;
    Procedure SetNome (i : Integer; UmNome : TLabel);
  end;

Implementation
```

```
Constructor TListaLabel.Init;
begin
  N := 0;
end;

Destructor TListaLabel.Done;
begin
end;

Function TListaLabel.GetIndice (UmLabel : TLabel) : Integer;
Var
  i,Indice : Integer;
begin
  While Length (UmLabel) < CompLabel do UmLabel := UmLabel + ' ';
  i := 1;
  Indice := 0;
  While (i <= N) and (Indice = 0) do
    If UmLabel = Nome[i] then Indice := i
                         else i := i + 1;
  If Indice = 0 then begin
                       N := N + 1;
                       Nome[N] := UmLabel;
                       Indice := N;
                     end;
  GetIndice := Indice;
end;

Function TListaLabel.GetNome (Indice : Integer) : TLabel;
begin
  GetNome := Nome [Indice];
end;

Function TListaLabel.GetN : Integer;
begin
  GetN := N;
end;

Procedure TListaLabel.Apresentar;
Var
  i : Integer;
begin
  Writeln ('Tamanho da Lista = ',N);
  For i := 1 to N do Writeln (i:3,'. ',Nome[i]);
end;

Procedure TListaLabel.SetNome (i : Integer; UmNome : TLabel);
begin
  Nome [i] := UmNome;
  If i > N then N := i;
end;

end.
```

Unit UMatBin

```
Unit UMatBin;
Interface
Uses UElem,UConst;

Type
  PTMatBin = ^TMatBin;
  TMatBin = Object
    FirstElemLin,FirstElemCol : PTVetElem;
    Constructor Init;
    Destructor Done;
    Procedure Incluir (i,j : Integer);
    Procedure Excluir (i,j : Integer);
    Function GetFirstElemCol (j : Integer) : PTElem;
    Function GetFirstElemLin (i : Integer) : PTElem;
  end;

Implementation

Constructor TMatBin.Init;
Var
```

```
  i : Integer;
begin
  New (FirstElemLin);
  New (FirstElemCol);
  For i := 1 to Nmax do begin
                          FirstElemLin^[i] := Nil;
                          FirstElemCol^[i] := Nil;
                        end;
end;

Destructor TMatBin.Done;
Var
  Next,UmElem : PTElem;
  j           : Integer;
begin
  For j := 1 to Nmax do
  begin
    UmElem := FirstElemCol^[j];
    While UmElem <> Nil do
    begin
      Next := UmElem^.GetNextElemCol;
      Dispose (UmElem,Done);
      UmElem := Next;
    end;
  end;
  Dispose (FirstElemCol);
  Dispose (FirstElemLin);
end;

Procedure TMatBin.Incluir (i,j : Integer);
Var
  UmElem : PTElem;
begin
  New (UmElem,Init(i,j));
  UmElem^.SetNextElemCol (FirstElemCol^[j]);
  If FirstElemCol^[j] <> Nil then FirstElemCol^[j]^.SetLastElemCol (UmElem);
  FirstElemCol^[j] := UmElem;
  UmElem^.SetNextElemLin (FirstElemLin^[i]);
  If FirstElemLin^[i] <> Nil then FirstElemLin^[i]^.SetLastElemLin(UmElem);
  FirstElemLin^[i] := UmElem;
end;

Procedure TMatBin.Excluir (i,j : Integer);
Var
  UmElem,Next,Last : PTElem;
begin
  UmElem := FirstElemCol^[j];
  While (UmElem <> Nil) and (UmElem^.GetLin <> i)
    do UmElem := UmElem^.GetNextElemCol;
  If UmElem <> Nil then
  begin
    Next := UmElem^.GetNextElemCol;
    Last := UmElem^.GetLastElemCol;
    If Last <> Nil then Last^.SetNextElemCol (Next)
                   else FirstElemCol^[j] := Next;
    If Next <> Nil then Next^.SetLastElemCol (Last);
    Next := UmElem^.GetNextElemLin;
    Last := UmElem^.GetLastElemLin;
    If Last <> Nil then Last^.SetNextElemLin (Next)
                   else FirstElemLin^[i] := Next;
    If Next <> Nil then Next^.SetLastElemLin (Last);
    Dispose (UmElem,Done);
  end;
end;

Function TMatBin.GetFirstElemCol (j : Integer) : PTElem;
begin
  GetFirstElemCol := FirstElemCol^[j];
end;

Function TMatBin.GetFirstElemLin (i : Integer) : PTElem;
begin
  GetFirstElemLin := FirstElemLin^[i];
end;

end.
```

Unit USetCov

```
Unit USetCov;
Interface
Uses UMatBin,UElem,ULabel,UConst;

Type
  PTSetCov = ^TSetCov;
  TSetCov = object (TMatBin)
    LabLin,LabCol : PTListaLabel;
    Custo,CustoUnit : PTVetReal;
    Constructor Init;
    Destructor Done;
    Procedure Ler (NomeArq : String);
    Procedure LerNumerico (NomeArq : String);
    Procedure Apresentar (Var Arq : TextFile);
    Procedure ApresentarColuna (Var Arq : TextFile; k : Integer);
    Function GetNLin : Integer;
    Function GetNCol : Integer;
    Function GetCusto (j : Integer) : Real;
    Function GetCustoUnit (j : Integer) : Real;
  end;

Var
  SCP : PTSetCov;

Implementation

Constructor TSetCov.Init;
begin
  TMatBin.Init;
  New (LabLin,Init);
  New (LabCol,Init);
  New (Custo);
  New (CustoUnit);
end;

Destructor TSetCov.Done;
begin
  TMatBin.Done;
  Dispose (LabLin,Done);
  Dispose (LabCol,Done);
  Dispose (Custo);
  Dispose (CustoUnit);
end;

Procedure TSetCov.Ler (NomeArq : String);
Var
  Arq          : TextFile;
  Coluna,Linha : TLabel;
  i,j          : Integer;
  UmCusto      : Real;
  Lixo2        : String[2];
  Texto        : String;
  NElem,Count  : LongInt;
begin
  AssignFile (Arq,NomeArq);
  Reset (Arq);
  Readln (Arq,Texto);
  Readln (Arq,Texto);
  Readln (Arq,Texto);
  NElem := 0;
  While not Eof (Arq) do
  begin
    Read (Arq,Coluna,UmCusto,Lixo2);
    j := LabCol^.GetIndice(Coluna);
    Custo^[j] := UmCusto;
    Count := 0;
    While not Eoln (Arq) do
    begin
      Read (Arq,Linha);
      i := LabLin^.GetIndice(Linha);
      Incluir (i,j);
      NElem := NElem + 1;
      Count := Count + 1;
    end;
    Readln (Arq);
```

```
    CustoUnit^[j] := UmCusto/Count;
  end;
  CloseFile (Arq);
  Writeln ('Linhas    = ',LabLin^.GetN);
  Writeln ('Colunas   = ',LabCol^.GetN);
  Writeln ('Elementos = ',NElem);
end;

Procedure TSetCov.LerNumerico (NomeArq : String);
Var
  Arq          : TextFile;
  Coluna,Linha : TLabel;
  i,j          : Integer;
  UmCusto      : Real;
  Texto        : String;
  NElem        : LongInt;
begin
  AssignFile (Arq,NomeArq);
  Reset (Arq);
  Readln (Arq,Texto);
  Readln (Arq,Texto);
  Readln (Arq,Texto);
  NElem := 0;
  While not Eof (Arq) do
  begin
    Read (Arq,j,UmCusto);
    Str (j:CompLabel,Coluna);
    LabCol^.SetNome (j,Coluna);
    Custo^[j] := UmCusto;
    While not Eoln (Arq) do
    begin
      Read (Arq,i);
      Str (i:CompLabel,Linha);
      LabLin^.SetNome (i,Linha);
      Incluir (i,j);
      NElem := NElem + 1;
    end;
    Readln (Arq);
  end;
  CloseFile (Arq);
  Writeln ('Linhas    = ',LabLin^.GetN);
  Writeln ('Colunas   = ',LabCol^.GetN);
  Writeln ('Elementos = ',NElem);
end;

Procedure TSetCov.Apresentar (Var Arq : TextFile);
Var
  N,j        : Integer;
begin
  N := LabCol^.GetN;
  For j := 1 to N do ApresentarColuna (Arq,j);
end;

Procedure TSetCov.ApresentarColuna (Var Arq : TextFile; k : Integer);
Var
  UmElem      : PTElem;
begin
  Write (Arq,LabCol^.GetNome(k),Custo^[k]:8:2,'  ');
  UmElem := GetFirstElemCol (k);
  While UmElem <> Nil do
  begin
    Write (Arq,LabLin^.GetNome(UmElem^.GetLin));
    UmElem := UmElem^.GetNextElemCol;
  end;
  Writeln (Arq);
end;

Function TSetCov.GetNLin : Integer;
begin
  GetNLin := LabLin^.GetN;
end;

Function TSetCov.GetNCol : Integer;
begin
  GetNCol := LabCol^.GetN;
end;
```

```
Function TSetCov.GetCusto (j : Integer) : Real;
begin
  GetCusto := Custo^[j];
end;

Function TSetCov.GetCustoUnit (j : Integer) : Real;
begin
  GetCustoUnit := CustoUnit^[j];
end;

end.
```

Unit UCromo

```
Unit UCromo;
Interface
Uses USetCov,UConst,UElem;

Type
  PTCromo = ^TCromo;
  TCromo = object
    N : Integer;
    Gene : PTVetIntRed;
    Fitness : Real;
    Constructor Init;
    Constructor CrossOver (Cromo1,Cromo2 : PTCromo);
    Destructor Done;
    Procedure CalculaFitness;
    Procedure PrimeCover1;
    Procedure PrimeCover2;
    Procedure Mutacao;
    Procedure Apresentar (NomeArq : String);
    Procedure ApresentaFitness;
    Function GetFitness : Real;
    Function GetGene (j : Integer) : Integer;
    Function GetN : Integer;
  end;

Implementation

Constructor TCromo.Init;
Var
  E,Cand                    : PTVetInt;
  p,i,NLin,NE,NCand         : Integer;
  UmElem                    : PTElem;
begin
  NLin := SCP^.GetNLin;                    {Inicialização}
  New (Gene);
  N := 0;
  New (E);
  New (Cand);
  For i := 1 to NLin do E^[i] := 0;
  NE := 0;
  While NE < NLin do
  begin
    p := 0;
    i := 1;                                {Escolher uma linha}
    While (i <= NLin) and (p = 0) do If E^[i] = 0 then p := i
                                                  else i := i + 1;
    UmElem := SCP^.GetFirstElemLin (p);
    NCand := 0;
    While UmElem <> Nil do begin           {Identificar colunas}
                             NCand := NCand + 1;
                             Cand^[NCand] := UmElem^.GetCol;
                             UmElem := UmElem^.GetNextElemLin;
                           end;
    p := Cand^[Random (NCand) + 1];        {Escolhe uma coluna}
    N := N + 1;
    Gene^[N] := p;
    UmElem := SCP^.GetFirstElemCol (p);
    While UmElem <> Nil do begin           {Atualiza cobertura}
                             i := UmElem^.GetLin;
                             E^[i] := E^[i] + 1;
                             If E^[i] = 1 then NE := NE + 1;
                             UmElem := UmElem^.GetNextElemCol;
```

```
                              end;
  end;
  Dispose (E);
  Dispose (Cand);
  CalculaFitness;
end;

Constructor TCromo.CrossOver (Cromo1,Cromo2 : PTCromo);
Var
  j,N1,N2,k,UmGene : Integer;
begin
  New (Gene);
  N1 := Cromo1^.GetN;
  N2 := Cromo2^.GetN;
  N := N1 + N2;
  For j := 1 to N1 do Gene^[j] := Cromo1^.GetGene (j);
  For j := (N1+1) to (N1+N2) do Gene^[j] := Cromo2^.GetGene (j-N1);

  If Random < ProbMutacao then Mutacao;

  For j := N downto 1 do
  begin
    k := Random(j)+1;
    UmGene := Gene^[k];
    Gene^[k] := Gene^[j];
    Gene^[j] := UmGene;
  end;

  If Random < 0.5 then PrimeCover2
                  else PrimeCover2;
  CalculaFitness;
end;

Destructor TCromo.Done;
begin
  Dispose (Gene);
end;

Procedure TCromo.CalculaFitness;
Var
  j : Integer;
begin
  Fitness := 0;
  For j := 1 to N do Fitness := Fitness + SCP^.GetCusto (Gene^[j]);
end;

Procedure TCromo.PrimeCover1;
Var
  E                             : PTVetInt;
  i,j,NLin,k                    : Integer;
  UmElem                        : PTElem;
begin
  NLin := SCP^.GetNLin;
  New (E);
  For i := 1 to NLin do E^[i] := 0;
  For j := 1 to N do
  begin
    UmElem := SCP^.GetFirstElemCol (Gene^[j]);
    While UmElem <> Nil do begin
                             i := UmElem^.GetLin;
                             E^[i] := E^[i] + 1;
                             UmElem := UmElem^.GetNextElemCol;
                           end;
  end;
  j := 1;
  While (j <= N) do
  begin
    k := j;
    UmElem := SCP^.GetFirstElemCol (Gene^[j]);
    While (UmElem <> Nil) and (k <> 0) do
    begin
      i := UmElem^.GetLin;
      If E^[i] <= 1 then k := 0;
      UmElem := UmElem^.GetNextElemCol;
    end;
    If k <> 0
    then begin
```

```
          Fitness := Fitness - SCP^.GetCusto (Gene^[k]);
          UmElem := SCP^.GetFirstElemCol (Gene^[k]);
          While UmElem <> Nil do begin
                                   i := UmElem^.GetLin;
                                   E^[i] := E^[i] - 1;
                                   UmElem := UmElem^.GetNextElemCol;
                                 end;
          Gene^[k] := Gene^[N];
          N := N - 1;
        end
    else j := j + 1;
  end;
  Dispose (E);
end;

Procedure TCromo.PrimeCover2;
Var
  E                              : PTVetInt;
  i,jOt,j,jMin,NLin,GeneTemp     : Integer;
  UmElem                         : PTElem;
  Redundante                     : Boolean;
  CustoUnitMin                   : Real;
begin
  NLin := SCP^.GetNLin;
  New (E);
  For i := 1 to NLin do E^[i] := 0;
  For j := 1 to N do
  begin
    UmElem := SCP^.GetFirstElemCol (Gene^[j]);
    While UmElem <> Nil do begin
                             i := UmElem^.GetLin;
                             E^[i] := E^[i] + 1;
                             UmElem := UmElem^.GetNextElemCol;
                           end;
  end;

  jMin := 1;
  While (jMin <= N) do
  begin
    CustoUnitMin := SCP^.GetCustoUnit (jMin);
    jOt := jMin;
    For j := jMin to N do
    begin
      If SCP^.GetCustoUnit (j) < CustoUnitMin
      then begin
             CustoUnitMin := SCP^.GetCustoUnit (Gene^[j]);
             jOt := j;
           end;
    end;

    GeneTemp := Gene^[jOt];
    Gene^[jOt] := Gene^[jMin];
    Gene^[jMin] := GeneTemp;

    Redundante := True;
    UmElem := SCP^.GetFirstElemCol (Gene^[jMin]);
    While (UmElem <> Nil) and Redundante do
    begin
      i := UmElem^.GetLin;
      If E^[i] <= 1 then Redundante := False;
      UmElem := UmElem^.GetNextElemCol;
    end;

    If Redundante
    then begin
           Fitness := Fitness - SCP^.GetCusto (Gene^[jMin]);
           UmElem := SCP^.GetFirstElemCol (Gene^[jMin]);
           While UmElem <> Nil do begin
                                    i := UmElem^.GetLin;
                                    E^[i] := E^[i] - 1;
                                    UmElem := UmElem^.GetNextElemCol;
                                  end;
           Gene^[jMin] := Gene^[N];
           N := N - 1;
         end
    else jMin := jMin + 1;
  end;
```

```
  Dispose (E);
end;

Procedure TCromo.Mutacao;
Var
  k,NTemp,NSCP : Integer;
begin
  NTemp := N div 4;
  NSCP := SCP^.GetNCol;
  For k := 1 to NTemp do
  begin
    N := N + 1;
    Gene^[N] := Random (NSCP) + 1;
  end;
end;

Procedure TCromo.Apresentar (NomeArq : String);
Var
  k     : Integer;
  Arq   : TextFile;
begin
  AssignFile (Arq,NomeArq);
  Rewrite (Arq);
  Writeln (Arq,'COLUNA  CUSTO   COBERTURA     ');
  Writeln (Arq,'============================');
  For k := 1 to N do SCP^.ApresentarColuna (Arq,Gene^[k]);
  Writeln (Arq,'============================');
  Writeln (Arq,'SOMA',Fitness:9:2);
  Writeln (Arq,'DUTIES  = ',N);
  Writeln (Arq,'LINHAS  = ',SCP^.GetNLin);
  Writeln (Arq,'COLUNAS = ',SCP^.GetNCol);
  CloseFile (Arq);
end;

Procedure TCromo.ApresentaFitness;
begin
  Writeln (Fitness:10:2,N:5);
end;

Function TCromo.GetFitness : Real;
begin
  GetFitness := Fitness;
end;

Function TCromo.GetGene (j : Integer) : Integer;
begin
  GetGene := Gene^[j];
end;

Function TCromo.GetN : Integer;
begin
  GetN := N;
end;

begin
  Randomize;
end.
```

Unit UPop

```
Unit UPop;
Interface
Uses UCromo,USetCov,UConst,stdctrls,SysUtils,Forms;

Const
  MaxPop  = 1600;

Type
  PTPop = ^TPop;
  TPop = object
    Indiv : Array [1..MaxPop] of PTCromo;
    N : Integer;
    Constructor Init (TamPop : Integer);
    Destructor Done;
    Procedure Run (NIter : Integer; Memo : TMemo);
```

```
    Procedure Incluir (UmIndiv : PTCromo);
    Procedure ExcluirPior;
    Function EscolhaAleatoria : Integer;
  end;

Var
  DeveParar : Boolean;

Implementation

Constructor TPop.Init (TamPop : Integer);
Var
  UmIndiv : PTCromo;
begin
  Writeln ('População = ',TamPop:5);
  N := 0;
  While N < TamPop do
  begin
    New (UmIndiv,Init);
    Incluir (UmIndiv);
    {

    GotoXY (1,WhereY);
    Write ('Gerando   = ',N:5);

    }

  end;
  {
  Writeln;
  }
end;

Destructor TPop.Done;
Var
  i : Integer;
begin
  For i := 1 to N do Dispose (Indiv[i],Done);
  N := 0;
end;

Procedure TPop.Run (NIter : Integer; Memo : TMemo);
Var
  i          : Integer;
  k          : LongInt;
  In1,In2    : Integer;
  UmIndiv    : PTCromo;
  Diferenca  : Real;
begin
  k := 0;
  Memo.Clear;
  Memo.Lines.Add ('Iteração    = ' + IntToStr(k*NIter));
  Memo.Lines.Add ('Fitness MAX = ' + FloatToStr(Indiv[N]^.GetFitness));
  Memo.Lines.Add ('Fitness MIN = ' + FloatToStr(Indiv[1]^.GetFitness));
  Diferenca := (Indiv[N]^.GetFitness - Indiv[1]^.GetFitness)/ Indiv[1]^.GetFitness;
  Memo.Lines.Add ('Diferença % = ' + FloatToStr(100*Diferenca));

  Repeat
    For i := 1 to NIter do
    begin
      In1 := EscolhaAleatoria;
      In2 := EscolhaAleatoria;
      New(UmIndiv,CrossOver(Indiv[In1],Indiv[In2]));
      Incluir (UmIndiv);
      ExcluirPior;
    end;
    k := k + 1;
    Memo.Clear;
    Memo.Lines.Add ('Iteração    = ' + IntToStr(k*NIter));
    Memo.Lines.Add ('Fitness MAX = ' + FloatToStr(Indiv[N]^.GetFitness));
    Memo.Lines.Add ('Fitness MIN = ' + FloatToStr(Indiv[1]^.GetFitness));
    Diferenca := (Indiv[N]^.GetFitness - Indiv[1]^.GetFitness)/ Indiv[1]^.GetFitness;
    Memo.Lines.Add ('Diferença % = ' + FloatToStr(100*Diferenca));
    Application.ProcessMessages;
  Until DeveParar or (Diferenca < 0.01);
end;
```

```
Procedure TPop.Incluir (UmIndiv : PTCromo);
Var
  i,k : Integer;
begin
  k := 1;
  While (k <= N) and (UmIndiv^.GetFitness > Indiv[k]^.GetFitness) do
  begin
    k := k + 1;
  end;
  For i := N downto k do Indiv [i+1] := Indiv [i];
  Indiv[k] := UmIndiv;
  N := N + 1;
end;

Procedure TPop.ExcluirPior;
begin
  Dispose (Indiv[N],Done);
  N := N - 1;
end;

Function TPop.EscolhaAleatoria : Integer;
begin
  EscolhaAleatoria := Random (N) + 1;
end;

end.
```

ANEXO:

TEORIA DOS CONJUNTOS DIFUSOS

Teoria dos Conjuntos Difusos

Definições Básicas (veja [ZAH65])

Definição 1

Seja X um conjunto clássico de objetos, chamado de universo, cujo os elementos genéricos são denotados por x. A função de pertinência de um elemento em um conjunto clássico $A \subseteq X$ é uma função característica $\mu_A(x)$: $x \rightarrow \{0,1\}$, tal que:

$$\mu_A(x) = \begin{cases} 0 & sse \quad x \notin A \\ 1 & sse \quad x \in A \end{cases}$$

onde $\{0,1\}$ é chamado de conjunto de avaliação . Se o conjunto de avaliação for o intervalo real $[0,1]$, então A é um *conjunto difuso*.

Nesta definição é importante ressaltar que:

a) quanto mais próximo do valor 1 for o valor de $\mu_A(x)$, maior é a pertinência de x em relação ao conjunto difuso A;
b) o conjunto universo X nunca é difuso.

Para caracterizar o conjunto difuso, poderá ser utilizada a seguinte notação:

$$A = \{(x, \mu_A(x)) \mid x \in X\}.$$

Zadeh, em 1972 propôs uma notação alternativa para a representação de conjuntos difusos. Quando X é um conjunto discreto $\{x_1, x_2, ..., x_n\}$, o conjunto difuso $A \subseteq X$ poderá ser escrito como:

$$A = \mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + ...+ \mu_A(x_n)/x_n = \sum_{i=1}^{n} \mu_A(x_i)/x_i$$

onde os elementos com grau de pertinência nulo podem ser omitidos. Quando X é um conjunto contínuo, esta expressão é a seguinte:

$$A = \int_X \mu_A(x) / x$$

Definição 2

Nos termos da definição 1, tem-se:

a) o suporte de A é um subconjunto ordinário de X caracterizado por

supp $A = \{ x \in X \mid \mu_A(x) > 0 \}$;

b) o conjunto A é dito normalizado se , e somente se, $\exists\, x \in X$ tal que $\mu_A(x) = 1$, e $\forall x \in X$ tem-se $0 \leq \mu_A(x) \leq 1$;

c) ϕ é conjunto vazio, definido como tendo $\mu_\phi(x) = 0$, $\forall x \in X$;

d) dois conjuntos difusos A e B são iguais, e denota-se por $A = B$, sse $\mu_A(x) = \mu_B(x)$, $\forall x \in X$.

e) a altura de um conjunto difuso é definida por:

$$hgt(A) = \sup_{x \in X} \mu_A(x)$$

ou seja, é o limite superior de $\mu_A(x)$.

Definição 3

Sejam A e B conjuntos difusos definidos em um universo X. Então:

a) o conjunto união, denotado por $A \cup B$ é definido pela seguinte função de pertinência:

$$\mu_{A \cup B}(x) = max\,(\mu_A(x), \mu_B(x)), \forall x \in X.$$

b) o conjunto interseção, denotado por $A \cap B$, tem a seguinte função de pertinência:

$$\mu_{A \cap B}(x) = min\ (\mu_A(x), \mu_B(x)), \forall x \in X.$$

c) o complemento $\overline{A}$ de um conjunto difuso é definido pela função de pertinência:

$$\mu_{\overline{A}}(x) = 1 - \mu_A(x), \forall x \in X$$

Para os operadores acima, seguem-se algumas das propriedades válidas:

i) Comutatividade

$A \cup B = B \cup A$

$A \cap B = B \cap A$

ii) Associatividade

$A \cup (B \cup C) = (A \cup B) \cup C$

$A \cap (B \cap C) = (A \cap B) \cap C$

iii) Idempotência

$A \cup A = A$

$A \cap A = A$

iv) Distributividade

v) Identidade

$A \cup \phi = A$

$A \cap X = A$

vi) $A \cup X = X$

vii) $A \cap \phi = A$

viii) Absorção

$A \cup (A \cap B) = A$

$A \cap (A \cup B) = A$

ix) Involução

$\overline{\overline{A}} = A$

Estes operadores não satisfazem, entretanto, a segunda lei, válida somente para os conjuntos ordinários:

i) $A \cap \overline{A} = \phi$

ii) $A \cup \overline{A} = X$

Definição 4

Seja A um conjunto difuso definido sobre o conjunto universo de discurso X, e seja $\alpha \in (0,1]$. O conjunto ordinário A_α , definido por $A_\alpha = \{x \in X \mid \mu_A(x) \geq \alpha\}$ é denominado de conjunto de nível α.

Definição 5

Um conjunto difuso A é convexo se, e somente se, seus conjuntos de nível α são todos convexos, ou $\mu_A(\lambda x_1 + (1-\lambda)x_2) \geq min(\mu_A(x_1), \mu_A(x_2))$, $\forall x_1, x_2 \in X$ e $\forall \lambda \in [0,1]$.

Esta definição de conjunto difuso convexo não implica no fato de que μ_A seja uma função convexa de x. Note-se, ainda, que se A e B forem conjuntos convexos, então $A \cap B$ também será um conjunto difuso convexo.

Definição 6

Um número difuso é um conjunto convexo normalizado A, definido sobre o conjunto de números reais R, que satisfaz as seguintes condições:

i) $\exists!\ x_0 \in R$, chamado de valor mais provável, tal que $\mu_A(x_0) = 1$;

ii) μ_A contínua por partes.

Definição 7

Sejam n universos, denotados por $X_1, X_2, ..., X_n$. Uma relação difusa de ordem n em $X_1 \times X_2 \times ... \times X_n$ é um conjunto difuso em $X_1 \times X_2 \times ... \times X_n$.

Princípio da Extensão (vide [ZAH75])

Definição 8

Seja X o produto cartesiano de universos, $X = X_1 \times X_2 \times ... \times X_r$, e seja $A_1, A_2, ..., A_r$, r conjuntos difusos em $X_1, X_2, ..., X_r$, respectivamente. O produto cartesiano $A_1 \times A_2 \times ... \times A_r$ é definido como:

$$A_1 \times A_2 \times ... \times A_r = \int_{X_1 \times ... \times X_r} min\,(\mu_{A_1}(x_1),..., (\mu_{A_r}(x_r)/(x_1,...,x_r)$$

Definição 9

Seja um mapeamento de $X_1 \times X_2 \times ... \times X_r$ para o universo Y, tal que $y = f(x_1,...,x_r)$. Então os r conjuntos difusos, denotados por A_i, induzem um conjunto difuso B em Y, através do mapeamento f, tal que:

$$\mu_B(y) = \begin{cases} \sup\limits_{\substack{x_1,...,x_r \\ y=f(x_1,...,x_r)}} mim(\mu_{A_1}(x_1),...,\mu_{A_r}(x_r)) \\ 0 \qquad se\ f^{-1}(y)=\phi \end{cases}$$

onde $f^{-1}(y)$ é a imagem inversa de y, e $\mu_B(y)$ é o grau de pertinência do valor y no conjunto difuso B definido em Y.

Adição de Números Difusos

A adição é uma operação crescente, isto é, dados x1 > x2, e y1 > y2, então x1 + y1 > x2 + y2. Sua extensão difusa também é crescente, e é definida através do princípio da extensão como:

$$\mu_{A \oplus B} = \sup_{\substack{z=x+y \\ x \in A \\ y \in B}} min\,(\mu_A(x), \mu_B(x))$$

onde $A \oplus B$ é o número difuso resultante da soma dos números difusos A e B definido sobre os reais.

Esta operação tal como foi definida, satisfaz as seguintes propriedades:

a) Comutatividade: $A \oplus B = B \oplus A$

b) Associatividade: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$

c) Identidade: $A \oplus 0 = A$ (zero não difuso)

d) Elemento Simétrico: $\mu_{-M}(x) = \mu_{M}(-x)$

Ressalta-se, contudo, que $\oplus$ não tem elemento simétrico no sentido da estrutura de grupo. Em outras palavras, a propriedade $M \oplus (-M) = 0$, $\forall M$ não se verifica.

Máximos e Mínimos entre Números Difusos

Dados os números difusos $M_1, M_2, \ldots, M_n$, o operador máximo difuso é definido pelo princípio da extensão, como:

$$\mu_{\tilde{max}(M_1,\ldots,M_n)}(z) = \sup_{\substack{z = max(x_1,\ldots,x_n) \\ x_i \in M_i}} min\ (\mu_{M_1}(x_1),\ldots,\mu_{Mn}(x_n))$$

Para o operador mínimo, tem-se:

$$\mu_{\tilde{mix}(M_1,\ldots,M_n)}(z) = \sup_{\substack{z = min(x_1,\ldots,x_n) \\ x_i \in M_i}} min\ (\mu_{M_1}(x_1),\ldots,\mu_{Mn}(x_n))$$

As seguintes propriedades são satisfeitas para estes operadores:

a) Comutatividade

$$\tilde{max}\ (A,B) = \tilde{max}\ (B,A)$$

$$\tilde{min}\ (A,B) = \tilde{min}\ (B,A)$$

b) Associatividade

$$\tilde{max}(A, \tilde{max}(B,C)) = \tilde{max}(\tilde{max}(A,B),C)$$

$$\tilde{min}(A, \tilde{min}\ (B,C)) = \tilde{min}\ (\tilde{min}\ (A,B),C)$$

c) Distributividade

$$\tilde{min}(A, \tilde{max}(B,C)) = \tilde{max}(\tilde{min}\ (A,B), \tilde{min}(\mathrm{A},C))$$

$$\tilde{max}\ (A, \tilde{min}\ (B,C)) = \tilde{min}\ (\tilde{max}\ (A,B), \tilde{max}\ (\mathrm{A},C))$$

$$A \oplus \tilde{max}(B,C) = \tilde{max}(A \oplus B, A \oplus C)$$

$$A \oplus \tilde{min}\ (B,C) = \tilde{min}\ (A \oplus B, A \oplus C)$$

d) Absorção

$$\tilde{max}\ (A, \tilde{min}\ (A,B)) = A$$

$$\tilde{min}\ (A, \tilde{max}\ (A,B)) = A$$

Ressalta-se que, enquanto a operação de máximo (ou mínimo) definida para os números ordinários determina o maior (ou menor) argumento da lista, o mesmo não acontece com estes operadores quando definidos para os números difusos.

Comparação de Números Difusos

Dados dois número difusos, M e N, a comparação entre eles consiste em determinar o grau de possibilidade de $M \geq N$, denotado por $\mu\ (M \geq N)$, e definido pelo princípio da extensão como:

$$\mu\ (M \geq N) = \sup_{\substack{x \geq y \\ x \in M \\ y \in N}} \ min\ (\mu_M (x), \mu_N (x))$$

Observa-se, entretanto, que o fato de $\mu\ (M \geq N) = 1$ não implica necessariamente que $\mu\ (M < N)$ seja nulo. De fato, $\mu\ (M < N)$ poderá ser diferente de zero, porém, quanto mias próximo de zero for, maiores serão as razões para se supor que M é maior que N.