Um estudo sobre projeto de redes com baixas restrições de conectividade

Luciana Ramos

Trabalho Final de Mestrado Profissional

Um estudo sobre projeto de redes com baixas restrições de conectividade

LUCIANA RAMOS

TRABALHO FINAL
DE
MESTRADO PROFISSIONAL APRESENTADO
AO
INSTITUTO DE COMPUTAÇÃO DA UNICAMP

PARA OBTENÇÃO DO GRAU
DE
MESTRE EM COMPUTAÇÃO

ÁREA DE CONCENTRAÇÃO: ENGENHARIA DA COMPUTAÇÃO

ORIENTADOR: FLÁVIO KEIDI MIYAZAWA

Julho 2003

Instituto de Computação
Universidade Estadual de Campinas

Um estudo sobre projeto de redes com baixas restrições de conectividade

Luciana Ramos

Banca Examinadora:

- Prof. Dr. Flávio Keidi Miyazawa (Orientador)
  Instituto de Computação - UNICAMP
- Prof. Dr. Vinícius Amaral Armentano
  Faculdade de Engenharia Elétrica e de Computação – UNICAMP
- Prof. Dr. Ricardo Dahab
  Instituto de Computação - UNICAMP
- Prof. Dr. Arnaldo Vieira Moura (Suplente)
  Instituto de Computação - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ramos, Luciana

R147e Um estudo sobre projeto de redes com baixas restrições de conectividade / Luciana Ramos -- Campinas, [S.P. :s.n.], 2003.

Orientador : Flavio Keidi Miyazawa

Dissertação (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Otimização combinatória. 2. Programação inteira. 3. Analise de redes (Planejamento). I. Miyazawa, Flavio Keidi. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Um estudo sobre projeto de redes com baixas restrições de conectividade

Este exemplar corresponde à redação final do Trabalho de Conclusão do curso de Mestrado Profissional em Computação devidamente corrigida e defendida por Luciana Ramos e aprovada pela Banca Examinadora.

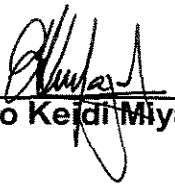
Campinas, 28 de agosto de 2003.

Prof. Dr. Flávio Keidi Miyazawa (Orientador).

Trabalho Final apresentado ao Instituto de Computação, Unicamp, para obtenção do título de Mestre em Computação na área de Engenharia de Computação.

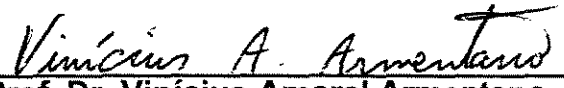
TERMO DE APROVAÇÃO

Tese defendida e aprovada em 28 de agosto de 2003, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Flavio Keidi Miyazawa
IC - UNICAMP

Prof. Dr. Ricardo Dahab
IC - UNICAMP



Prof. Dr. Vinícius Amaral Armentano
FEEC - UNICAMP

Resumo

Na atualidade não são raras as vezes que se fazem necessários estudos para se obter resultados melhores para um determinado problema. De fato, situações como estas são encontradas em uma variedade imensa de problemas, desde a necessidade de um trabalhador encontrar o melhor caminho para chegar ao seu local de trabalho até situações mais complexas como projetar uma rede de computadores ou de telecomunicações, interligando cidades de um país, com o menor custo possível - tema estudado neste trabalho.

Nesta dissertação, consideramos o problema de projetar uma rede através da conexões entre pares de pontos, sob certas condições de conectividade. Tais condições garantem que a ocorrência de falha em uma conexão continue garantindo ligações alternativas entre pontos importantes da rede. Consideramos que estas falhas podem indisponibilizar apenas as conexões entre pontos, não consideramos falhas que indisponibilizam os pontos. O custo de uma rede é a soma dos custos das conexões existentes entre seus pontos. Neste problema o objetivo é obter uma rede com estas características de conectividade e que seja de custo mínimo.

Para atingir esse objetivo, apresentamos algumas técnicas que são utilizadas para resolver esse tipo de problema. Descrevemos um algoritmo exato e algumas heurísticas para obter soluções viáveis de maneira eficiente. Por fim, apresentamos os resultados computacionais obtidos pela aplicação de teste em instâncias obtidas em repositórios públicos, algumas adaptadas para este problema.

Abstract

At present, it is not rare the study of a problem to obtain better results. In fact, this type of situation is found in many problems like the problem a laborer have to find the shortest path between his home and his office or the problem to design a network connecting the cities of a country, which is considered in this work.

In this dissertation, we consider the network design problem under certain connectivity constraints. These constraints guarantee that the occurrence of a fail in a link does not disconnect important nodes of the network. We consider that a fail may occur only in the links, and we do not consider fails that affect nodes. In this problem, the objective is to obtain a network with these connectivity constraints minimizing the total cost of its links.

To attain this objective, we present some techniques used to solve problems of this type. We describe an exact algorithm and some heuristics to obtain feasible solutions with reduced cost efficiently. At last, we present some computational results applied on instances obtained in public repositories, adapted for this problem.

Agradecimentos

Agradeço a Deus por ter-me dado mais uma vez a oportunidade de experimentar a dádiva do crescimento espiritual e por guiar-me em todos os momentos desta existência.

Agradeço aos meus pais Ana Amália e André por existirem, por terem me apoiado em todos os momentos de minha vida, por fazerem de mim o que hoje sou, por me amarem, serem meus pais e principalmente por serem responsáveis pelo sentimento eterno de amor que existe dentro do meu coração.

Agradeço aos meus irmãos Adriana e André pela paciência nos meus momentos de mau humor em que eu considerei mais importante o estudo e o trabalho em detrimento do relacionamento familiar.

Agradeço às minhas amigas Valesca, Daniela e Solange pelo apoio e pela força nos momentos de desanimo em que eu estava decidida a desistir de mais esta conquista.

Agradeço ao meu amigo Maciste pelo apoio e paciência em todos o momentos em que precisei de ajuda.

Agradeço especialmente aos meus amigos Áurea e Flávio pelo apoio, pelas inúmeras noites de sexta feira que passei em Campinas, por todas as coisas que fizeram para possibilitar a conclusão deste trabalho e principalmente pelo carinho recíproco que existe em nossos corações.

Agradeço a meu orientador e amigo Flávio pelas inúmeras horas gastas durante esta orientação, pela paciência, dedicação e pelo seu entusiasmo contagiante pela pesquisa, nativo de sua personalidade.

Sumário

Lista de Tabelas e Exemplos

Lista de Figuras

Lista de Algoritmos

Capítulo 1

1. Introdução

Problemas de otimização são vastamente estudados devido a necessidade de resolução de problemas com menor custo e melhores resultados. Na atualidade não são raras as vezes que se fazem necessários estudos para maximizar ou minimizar os resultados de um determinado problema.

Nesta dissertação estudamos o problema de projeto de redes com baixas restrições de conectividade visando minimizar o seu custo. Nossa preocupação será apenas na topologia, ou seja, consideramos apenas links que ligam pontos na rede. O objetivo deste trabalho consiste em estudar apenas a disponibilidade dos caminhos; não consideramos possibilidade de indisponibilidade dos pontos envolvidos.

Problemas como este já foram estudados por vários pesquisadores, dentre os quais destacamos o trabalho de Stoer [1]. No nosso caso, o objetivo principal é projetar redes que interligam cidades e estados de um determinado país com o menor custo e possuam algumas restrições de segurança, como garantir que exista pelo menos um caminho alternativo para comunicação entre pontos considerados especiais em uma rede.

Em [1], Stoer cita problemas de telecomunicações que paralisam os sistema de telefonia ou que isolam cidades causando transtornos ou implicando na perda de milhões de dólares. Porém não é preciso ir tão longe para nos deparamos com problemas do mesmo grau de complexidade; no mundo dos negócios, situações como estas são muito comuns: imagine uma grande companhia, com por exemplo a Ambev, que comercializa seus produtos por todo o país. É estratégico para a empresa interligar todas as suas distribuidoras e revendedoras de forma que a matriz possa distribuir as metas de negócios, controlar o desempenho de seus vendedores e coletar informações sobre a sua dominância no mercado e a concorrência enfrentada nos diversos estados brasileiros. Certamente, para resolver este problema, uma empresa como a Ambev contratará os serviços de uma empresa especializada em telecomunicações, porém o paralelo aqui é feito como forma de exemplificar a extensão do problema no mundo dos negócios.

Para resolver problemas como estes, de forma eficiente, faz-se necessário a utilização de algoritmos e técnicas de otimização para determinar o projeto de rede de menor custo para ligar os vários escritórios, distribuídos entre as várias cidades do país, e ainda garantir que pontos estratégicos possuam rotas alternativas.

No decorrer do trabalho, consideramos diversas técnicas de otimização e, aplicamos estas técnicas ao problema de construção de redes com baixas restrições de conectividade:

a. programação linear e programação linear inteira;
b. planos de corte e o método de *Branch & Cut*;
c. algoritmos de aproximação;
d. heurísticas construtivas e de melhoria.

Neste capítulo, apresentamos o problema e suas motivações. O capítulo 2 apresenta as técnicas de otimização que foram utilizadas. O capítulo 3 apresenta a

formulação matemática que descreve o problema proposto e as desigualdades consideradas. O capítulo 4 apresenta as heurísticas utilizadas neste trabalho e que serviram como apoio na determinação de uma solução para o problema bem como na busca pela solução ótima. O capítulo 5 apresenta a aplicação de teste construída no decorrer deste trabalho. O capítulo 6 apresenta os resultados computacionais obtidos pela aplicação das técnicas implementadas e aplicadas em instâncias obtidas em repositórios públicos e gerados aleatoriamente. Por fim, o capítulo 7 apresenta as conclusões obtidas como fruto deste estudo.

1.1. Definição do Problema

O problema estudado neste trabalho visa construir redes com características de sobrevivência em situações onde links de comunicação tornam-se indisponíveis entre pontos importantes da rede. Assim, o objetivo é construir redes que permaneçam operantes, através de rotas alternativas, mesmo em caso de falha de links.

Durante este trabalho, consideramos que um link de comunicação é uma seqüência de ligações entre pares vértices v_1 e v_2.

A denominação em inglês para problemas como este é dada pelo termo "*Design of Survivable Networks*", porém a tradução exata deste termo para a língua portuguesa resulta em um título pouco intuitivo, a saber, "Projeto de redes sobreviventes". Dessa forma, neste trabalho, denominamos o problema como "*Problemas de Projeto de Redes Resilientes*".

Por questões de simplicidade, no decorrer deste texto, os problemas de projeto de redes são denotados por PRR.

Dizemos que dois pontos da rede possuem um link de conexão se existe uma forma de comunicação entre eles. Dizemos que dois pontos possuem um link alternativo se após a queda de um link de comunicação ainda existe outro link de comunicação entre estes pontos.

Posto isso, podemos dizer que o PRR pode ser descrito informalmente como a necessidade de interligar diversos pontos de forma a garantir algumas propriedades. Uma definição mais precisa, usando a linguagem de teoria dos grafos, será apresentada posteriormente. As propriedades são:

- Pontos considerados críticos aos negócios requerem pelo menos um link de conexão alternativo entre si.
- Pontos considerados sem importância, podem ou não estar interligados na rede.
- Os demais pontos devem possuir pelo menos um link de conexão entre si.

Stoer [1] nomeia esses pontos, conforme descrito a seguir:

- Pontos críticos são chamados de "especiais".
- Pontos considerados sem importância são chamados de "opcionais".
- Os demais pontos são chamados de "ordinários".

No decorrer deste trabalho, denotamos tais pontos como críticos, opcionais e ordinários.

Segundo Stoer, o estudo sobre a disponibilidade de redes abrange aspectos muito mais amplos, como por exemplo, garantir que existam três ou mais links de conexão interligando pontos críticos, considerações sobre o tráfego da rede, equipamentos, etc. Este trabalho visa estudar apenas redes com baixas restrições de conectividade, ou seja, será exigido que existam, no máximo, duas rotas alternativas ligando pontos críticos da rede conforme descrito no início desta seção.

Assumindo-se estas condições, o problema consiste em encontrar o menor custo para construção de uma rede, por exemplo de fibra ótica, que obedeça as condições impostas acima. Vale observar que o problema não está restrito a redes de fibra ótica, porém para interligar cidades de um país, por exemplo, a fibra ótica é uma opção interessante a ser considerada.

A seguir, apresentamos uma formulação para o problema considerado. Usamos termos bastante conhecidos da área de teoria dos grafos e de algoritmos. Porém, caso o leitor não esteja familiarizado com os conceitos da área de teoria dos grafos e de algoritmos, recomendamos a leitura do livro de Cormen et al. [15].

Seja $G = (V,E)$ um grafo não orientado, que representa o conjunto de pontos e possíveis arestas de conexões entre eles, onde V representa o conjunto de pontos e E o conjunto de arestas do grafo ou, mais especificamente no nosso caso, o conjunto de possíveis links de conexões entre os pontos.

Para cada conexão $e=(u,v) \in E$, entre dois pontos u e v, existe um custo fixo associado c_e, que representa o custo de construção do link de conexão entre os pontos u e v.

Considere que para cada ponto $s \in V$, existe um número r_s associado tal que:

- $r_s = 2$, se s é um ponto crítico da rede;
- $r_s = 1$, se s é um ponto ordinário da rede;
- $r_s = 0$, se s é um ponto opcional da rede.

Neste cenário, as condições de disponibilidade ficam satisfeitas se $\forall\, s, t \in V$, $s \neq t$, a rede possui pelo menos $r(s,t) := \min \{r_s, r_t\}$ caminhos disjuntos nas arestas ligando s a t.

Dessa forma, uma solução que não fere nenhuma das restrições de disponibilidade do problema, também chamada de *solução viável*, é dada por um subgrafo $G' = (V, E')$, onde $E' \subseteq E$, sendo que para cada par de vértices disjuntos $s, t \in V$, existem, pelo menos, $r(s,t)$ caminhos disjuntos entre eles.

Dado uma solução viável $G' = (V,E')$, seu custo é dado por $\sum_{e \in E'} c_e$. Vamos denotar este custo por $c(G')$.

Uma solução viável $N=(V,F)$, $F \subseteq E$ é dita ser uma *solução ótima* se seu custo é mínimo.

A seguir, são apresentadas as questões que motivaram o estudo de problemas de construção de redes como as características definidas aqui.

1.2. Motivação

Problemas de otimização são encontrados em uma variedade imensa de situações, desde a necessidade de um trabalhador encontrar o melhor caminho para chegar a seu local de trabalho até situações mais complexas como projetar uma rede de computadores ou de telecomunicações, interligando cidades de um país, com o menor custo possível - tema estudado neste trabalho.

Encontrar soluções ótimas ou mesmo aproximadas para problemas desse tipo é um desafio nem sempre fácil de ser vencido. Conforme visto no Capitulo 2, existem diversas técnicas que podem ser aplicadas para resolução desses problemas, porém o tempo computacional exigido pode tornar-se um problema sério a ser enfrentado.

A sociedade moderna, desde a indústria, as finanças, o comércio, a medicina, a educação, a polícia, as instituições governamentais até a agricultura, depende de maneira quase que crucial da disponibilidade de redes de comunicações. Cada uma dessas entidades possui um certo grau de tolerância quanto a falhas na rede. Em operações militares, por exemplo, faz-se necessário um mecanismo de tolerância a falhas altamente sofisticado. Falhas de comunicação entre pontos estratégicos pode significar o fracasso de uma operação militar. Já em outras áreas como a educação, o nível de tolerância pode ser relaxado sendo exigido apenas que existam conexões entre pontos importantes.

O tema associado à sobrevivência de redes surgiu pioneiramente no campo militar uma vez que em tempos de guerra as forças armadas precisavam da garantia de que a comunicação entre seus exércitos não seria interrompida em caso do ataque inimigo destruir alguns dos links de comunicação entre suas frentes militares ou mesmo alguns de seus quartéis.

Atualmente, o tema vem sendo vastamente utilizado por empresas de telecomunicações tanto para disponibilizar comunicação telefônica como para viabilizar a transferência de dados entre empresas.

Conforme citado na introdução deste trabalho, existe uma variedade de grandes empresas que precisam manter links de conexão entre seus escritórios ou clientes por questões estratégicas, sejam elas para troca ou centralização de informação ou mesmo para distribuição das diretrizes de seus negócios.

Na maioria das vezes, tais empresas possuem alguns escritórios ou clientes que são considerados de maior importância em relação a outros. É natural pensar, por exemplo, que escritórios localizados nas grandes capitais possuem maior importância quando comparados a escritórios localizados em pequenas cidades. Dessa forma, é também intuitivo desejar que tais localizações possuam maior tolerância a falhas no que diz respeito a comunicação entre tais pontos. Ou seja, em caso de falha de um link de conexão entre esses pontos, é desejado que exista uma caminho alternativo que viabilize a conexão.

Em cenários como estes, as empresas de telecomunicações, no intuito de satisfazer tais critérios, elegem os chamados pontos críticos e ordinários e projetam a rede de custo mínimo que viabiliza a conexão entre esses pontos.

O grande desafio para as empresas de telecomunicações é que existem milhares de pontos e milhares de alternativas de ligação entre eles. Isto aumenta bastante a dificuldade computacional para encontrar uma solução ótima deste problema.

O problema de projeto de redes com baixas restrições de conectividade é um problema típico de Otimização Combinatória. Problemas deste tipo são comuns no mundo dos negócios e na sociedade atual.

No próximo capítulo apresentamos um resumo das técnicas que consideramos neste trabalho para resolver o PRR.

Capítulo 2

2. Técnicas

Embora existam vários métodos para resolver problemas de otimização, nem sempre eles são eficientes. Por exemplo, a complexidade computacional pode variar de problemas resolvidos em tempo polinomial a problemas resolvidos em tempo exponencial.

Segundo Ferreira e Wakabayashi [2], para alguns desses problemas são conhecidos métodos eficientes e rápidos para resolvê-los; para outros não são esperados métodos que produzam soluções ótimas em tempo polinomial. Para muitos só são conhecidos algoritmos de tempo exponencial.

Para exemplificar a complexidade do problema, suponha que uma instrução de computador seja executada em (10^{-10}) segundos. A tabela 2.1 ilustra o tempo computacional necessário considerando funções de tempos polinomiais e exponenciais:

f(n)	n =10	n = 20	n = 30	n = 40	n = 50	n = 60
n	0,000000001 segundos	0,000000002 segundos	0,000000003 segundos	0,000000004 segundos	0,000000005 segundos	0,000000006 segundos
n^2	0,00000001 segundos	0,00000004 segundos	0,00000009 segundos	0,00000016 segundos	0,00000025 segundos	0,00000036 segundos
n^3	0,0000001 segundos	0,0000008 segundos	0,0000027 segundos	0,0000064 segundos	0,0000125 segundos	0,0000216 segundos
n^5	0,00001 segundos	0,00032 segundos	0,00243 segundos	0,01024 segundos	0,03125 segundos	0,07776 segundos
2^n	$1,024x10^{-7}$ segundos	0,000104858 segundos	0,107374182 segundos	1,83251938 minutos	1,303124892 dias	3,645901338 anos
3^n	$5,9049 \times 10^{-6}$ segundos	0,34867844 segundos	0,238299921 dias	38,44637175 anos	2270,219805 milênios	134054209,3 milênios

Tabela 2.1: comparação entre tempos polinomiais e exponenciais

Por outro lado, muitos destes são problemas práticos de grande importância que necessitam de uma solução, mesmo que aproximada, ou que demore bastante tempo para ser obtida. Para viabilizar a busca dessa solução desejada, existem vários métodos que são aplicados para a obtenção de soluções de custo reduzido para estes problemas. Alguns deles consistem em enumeração implícita, relaxação, métodos de planos-de-corte e heurísticas.

A complexidade computacional de muitos problemas computacionais pode ser compreendida usando a teoria de NP-completude. Para isto, os problemas foram simplificados para problemas de decisão, onde a resposta para o problema é simplesmente sim ou não, mas que ainda conservam sua dificuldade computacional.

O estudo da complexidade computacional destes problemas levou a uma classificação dos problemas que podemos resumir da seguinte maneira:

- P (*Polynomial*): corresponde à classe de problemas de decisão que podem ser resolvidos por um algoritmo determinístico polinomial;
- NP (*Non Deterministic Polynomial*): corresponde ao conjunto dos problemas de decisão que podem ser resolvidos por um algoritmo não determinístico polinomial;
- NP-Difícil (*NP-Hard*): um problema x é considerado NP-Difícil se todo problema de NP se reduz polinomialmente a x;

  Informalmente, redução polinomial de um problema P a outro problema P^* corresponde a um algoritmo de tempo polinomial que transforma uma instância $x \in P$ a uma instância $y \in P^*$, tal que $P(x)$ = "SIM" se e somente se $P^*(y)$ = "SIM" [14].
- NP-Completo(*NP-Complete*): um problema é NP-Completo se está na classe NP e é NP-Difícil.

Para obter definições mais precisas, recomendamos a leitura do livro de Cormen et al. [15] ou Garey e Johnson [13], ao leitor que não estiver habituado a esses conceitos.

Embora ainda não tenha sido provado matematicamente, acredita-se que a figura 2.1, apresentada a seguir, representa a categorização destas classes:

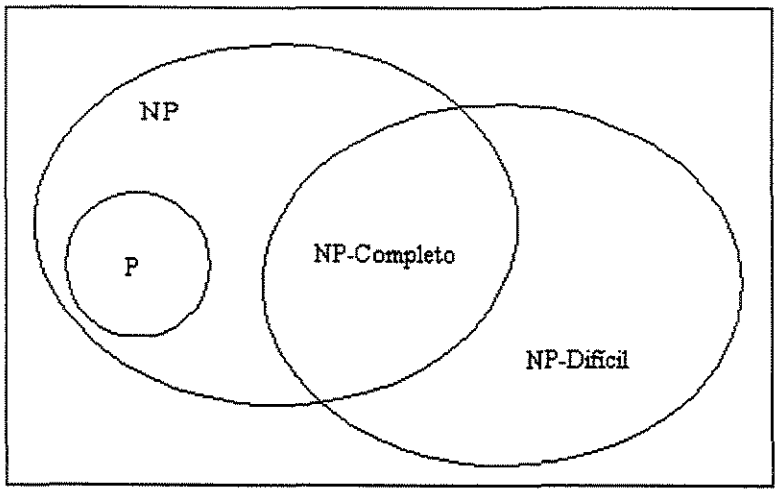


Figura 2.1: possível categorização de classe de problemas P, NP, NP-completo e NP-difícil

As classes NP, P e NP-Completo são aplicadas para problemas de decisão, ou seja, aqueles problemas que assumem como resposta "sim" ou "não". Já a classe NP-Difícil pode incluir problemas que não são de decisão, além dos problemas de decisão. A grande questão em aberto na área de ciência da computação é se $P \stackrel{?}{=} NP$. A maioria dos pesquisadores acredita que essa igualdade não é válida e só existam algoritmos super-polinomiais para os problemas NP-difíceis.

Associados a todos esses conceitos, existem técnicas matemáticas que auxiliam na resolução de problemas reais como o estudado aqui.

O PRR é um problema de otimização NP-Difícil. Um caso particular bem conhecido é o problema da árvore de Steiner para o qual não há pontos críticos e já foi provado ser NP-difícil [13].

As seções a seguir apresentam os conceitos básicos envolvidos em cada técnica utilizada para resolver o problema em questão.

2.1. Programação Linear

A programação linear é um modelo matemático utilizado para resolver problemas de otimização que consiste em encontrar uma solução ótima para um determinado problema, caso ela exista.

A programação linear busca solucionar problemas de maximização ou minimização de funções lineares considerando-se um conjunto de restrições lineares previamente estabelecido.

No âmbito dos negócios, a programação linear é vista como uma ferramenta extremamente importante que é utilizada em casos onde é desejado encontrar o lucro máximo ou o custo mínimo para situações nas quais existem diversas restrições [3]. A utilização desse método é considerada uma das principais técnicas para obtenção de soluções ótimas e aproximadas para problemas de otimização dessa natureza.

Em outras palavras, a programação linear é um modelo usado para modelagem de problemas reais em linguagem matemática e a vantagem de utilizar esse tipo de estratégia é que existem algoritmos rápidos para sua resolução. O mais conhecido é o Simplex, que apesar de ter complexidade de tempo exponencial no pior caso, é polinomial no caso médio.

Outros métodos que têm obtido excelentes resultados práticos são os métodos de pontos interiores, alguns destes com desempenho superior ao do algoritmo Simplex.

Em termos matemáticos, um problema de programação linear pode ser definido da seguinte forma:

Dados:

- uma matriz $A = (a_{ij}) \in Q^{n \times m}$
- vetores $c = (c_i) \in Q^m$ e $b = (b_i) \in Q^n$.

Encontrar vetor $x = (x_i) \in Q^m$ (se existir) tal que:

- minimize $c_1x_1 + c_2x_2 + \ldots + c_mx_m$
- e esteja sujeito às seguintes restrições:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m \leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m \geq b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m = b_n. \end{cases}$$

Graficamente, podemos representar um problema de programação linear como sendo um poliedro, conforme exemplificado pela figura 2.2:

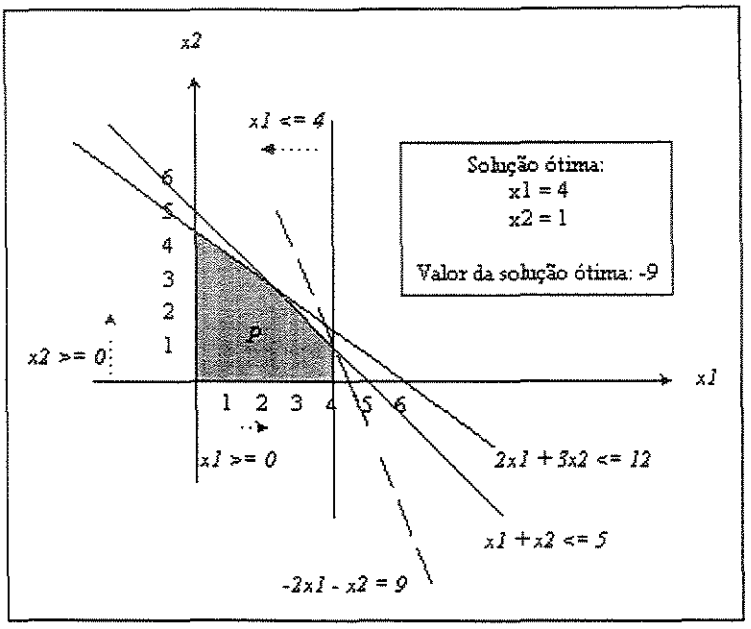


Figura 2.2: exemplo gráfico de programação linear

Na figura 2.2, o problema de programação linear consiste em:

- minimizar $-2x_1 - x_2$

- sujeito a:

$$\begin{cases} x_1 + x_2 \leq 5 \\ 2x_1 + 3x_2 \leq 12 \\ x_1 \leq 4 \\ x_1 \geq 0 \\ x_2 \geq 0 \end{cases}$$

Graficamente, é intuitivo ver que a solução está restrita ao poliedro P. No caso do exemplo da figura 2.2, a solução ótima é $x_1 = 4$ e $x_2 = 1$, sendo que o resultado da solução ótima é $-9 = -2x_1 - x_2$.

Atualmente, existem vários softwares livres e comerciais que resolvem problemas de programação linear de maneira satisfatória. Dentre eles, podemos citar:

- Pacotes comerciais:
  - CPLEX / ILOG,
  - XPRESS/Dash Optimization,
  - OSL / IBM.

- Pacotes livres:
  - GLPK / Gnu,
  - SoPlex / Zib.

Segundo Ferreira e Wakabayashi [2], a comunidade de otimização combinatória tem optado pelo uso do CPLEX que é considerado um pacote eficiente e robusto e vem apresentando ótimos resultados. O CPLEX tem sua biblioteca de funções (Cplex Callable Library) projetada sob medida para a utilização no contexto de algoritmos *Branch & Cut* que serão abordados posteriormente neste trabalho.

Neste estudo, nós utilizamos a biblioteca XPRESS que nos últimos anos tem apresentando um bom desempenho e, recentemente, estudos através de benchmarks mostram que tem obtido desempenho superior ao do CPLEX.

2.2. Programação Linear Inteira

A programação linear inteira é uma restrição da programação linear onde as variáveis assumem apenas valores inteiros.

Em termos matemáticos, um problema de programação linear inteira pode ser definido da seguinte forma:

Dados:

- uma matriz $A = (a_{ij}) \in Q^{n \times m}$
- vetores $c = (c_i) \in Q^m$ e $b = (b_i) \in Q^n$.

Encontrar vetor $x = (x_i) \in Z^m$ (se existir) tal que:

- minimize $c_1x_1 + c_2x_2 + \ldots + c_mx_m$
- e esteja sujeito às seguintes restrições:
  - $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m \leq b_1$
  - $a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m \geq b_2$
  - ...
  - $a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m = b_n$.

Problemas de programação linear inteira podem ser relaxados a problemas de programação linear se excluirmos a condição de integralidade do vetor x, ou seja, considerarmos a condição $x = (x_i) \in Q^m$.

A grande diferença entre programação linear e programação linear inteira é quanto à complexidade computacional: o problema de programação linear inteira é um problema NP-difícil, mesmo quando o sistema linear contém apenas uma restrição e as variáveis só podem receber valores 0 ou 1. Um exemplo de problema que pode ser modelado desta forma é o problema da mochila binária [13].

2.3. Branch & Bound

A técnica de otimização *Branch & Bound* é considerada uma técnica de "enumeração" e "poda" uma vez que a sua estratégia é construir uma árvore com todas as soluções viáveis de um determinado problema. A técnica consiste em percorrer a árvore de *Branch & Bound* evitando percorrer ramos que levem a soluções piores que a melhor solução já encontrada ou que não levam a soluções viáveis. Sempre que um ramo represente uma solução de valor melhor que as anteriores, esta é guardada. Ou seja, a idéia é tentar eliminar vários ramos da árvore. Assim, ao percorrer a árvore que enumera o conjunto de soluções do problema, sempre que for encontrado um ramo que não é viável, ou que contém uma solução pior que aquela que já temos, este é podado e não é preciso percorrê-lo.

Dessa forma, o problema original é subdividido em vários outros problemas: os vários ramos da árvore construída. Supondo que é conhecida uma solução para o problema, a busca por soluções ótimas limita-se a percorrer a árvore na procura do menor (ou maior) valor limitante.

No pior caso, um algoritmo *Branch & Bound* tem a mesma complexidade de uma enumeração e a busca na árvore obedece à seguinte propriedade:

Seja x uma solução conhecida. Suponha que deseja-se encontrar uma solução de custo mínimo de um determinado problema . A cada nó da árvore está associado um conjunto de restrições, portanto cada nó possui um limitante inferior. Considere um determinado nó da árvore de decisão:

- se o valor do limitante inferior para este nó for maior ou igual ao da solução x conhecida, então certamente não há solução nesta sub-árvore com valor menor que x. Assim, podemos podar esta sub-árvore, podando todos os descendentes da busca;
- se o valor do limitante inferior para este nó for menor que o valor do limitante superior do nó em questão, então pode ser que a solução ótima esteja nesta sub-árvore. O ramo não é descartado e a procura pela solução ótima continua;
- sempre que encontrarmos uma solução y de menor valor que x neste ramo, então a solução x conhecida é desconsiderada e y passa a ser considerado com a solução conhecida atual e, então, a busca continua.

De acordo com as regras acima, o problema descrito pode ser considerado um problema de minimização e, o importante no método *branch & bound* é que nenhum ramo deixará de ser analisado: um ramo será "podado" se nos levar a soluções inviáveis ou que não melhoram a solução que já conhecemos.

Um dos grandes problemas deste método é que na maioria dos problemas, a enumeração gera uma quantidade exponencial de soluções e, portanto, a escolha do limitante inferior passa a ser decisiva na solução do problema. A escolha de bons limites inferiores e superiores garantem uma grande redução no número de nós que serão percorridos.

No caso do nosso estudo, consideramos árvores de *Branch & Bound* binárias juntamente com programação linear uma vez que as variáveis do problema assumirão apenas valores 0 ou 1. Isto se deve à necessidade de encontrarmos soluções inteiras para

o problema, ou seja, não nos interessa encontrar uma solução que viabilize a construção de apenas parte do link de comunicação entre dois pontos.

A figura 2.3, apresentada a seguir, ilustra a técnica *Branch & Bound* para o nosso problema:

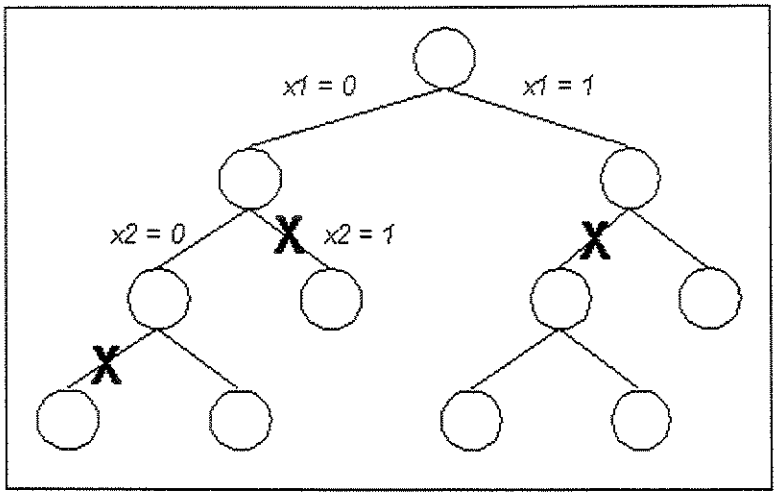


Figura 2.3: técnica *Branch & Bound*

2.4. Planos de Corte

Os métodos chamados "planos de corte" foram criados para resolver problemas de programação linear inteira. A idéia do método é "relaxar" a condição de integralidade e sistematicamente eliminar as soluções fracionárias encontradas através da inclusão de novas restrições (inequações). Essas novas inequações que são acrescidas ao modelo são chamadas de planos de corte.

Sabe-se que o processo de inclusão de planos de corte é um processo finito que leva a uma solução inteira para o problema de programação linear inteira proposto. Embora finito, o número de inequações necessárias para chegar-se à solução inteira procurada, pode crescer exponencialmente.

Nesta estratégia, o problema é resolvido através de um poliedro inicial que contenha o problema original porém descrito por poucas desigualdades e, repetidamente, encontra uma solução ótima para o novo problema. Se esta solução não resolver o problema original, inserimos uma nova desigualdade válida que separa esta solução sem perder nenhuma solução do problema original.

Mesmo reduzindo o número de inequações, como já foi dito anteriormente, uma vez que vários problemas de programação linear inteira são NP-Difíceis, não é esperado que existam algoritmos eficientes para resolver problemas desse tipo.

Quando estudamos planos de corte, outra questão a ser resolvida é como obter inequações válidas que separam as soluções que não nos interessam. Este problema é chamado de *problema da separação*. O problema da separação consiste em determinar se um ponto x é uma solução viável ou, caso não seja, encontrar uma desigualdade que separe este ponto do problema estudado.

Formalmente, podemos definir o problema da separação da seguinte forma[8]:

Seja

- $P \subseteq R^n$ um conjunto convexo
- $y \in R^n$

Determinar

- se $y \in P$
- caso contrário, encontrar uma desigualdade $ax \leq b$ tal que

$$P \subseteq \{x : ax \leq b\}$$

$$ay > b$$

A figura 2.4 exemplifica o problema da separação:

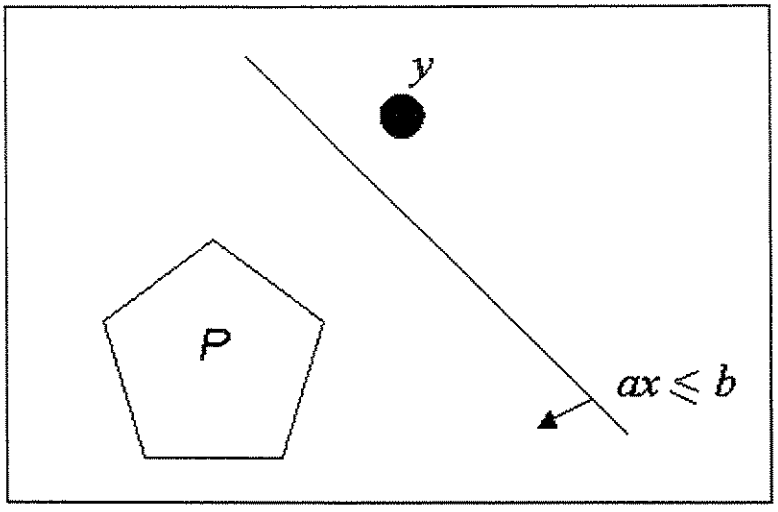


Figura 2.4: exemplo de problema da separação.

O problema da separação é um problema importante para o nosso estudo uma vez que Grotschel, Lovász, Schrijver provaram que é possível resolver um problema de otimização associado a um programa linear em tempo polinomial se o problema de separação descrito acima pode ser resolvido em tempo polinomial.

Gomory (1958, 1960, 1963) e Chvátal (1973) [6] foram os primeiros a propor métodos automatizados e genéricos para criar as inequações de corte e tiveram um papel importante neste ponto desenvolvendo algoritmos que abordaram a questão. Segundo descrito por Aardal [6], a contribuição de Gomory e Chvátal foi muito importante teoricamente porém, na prática, suas técnicas foram pouco efetivas para resolver problemas NP-difíceis em tempo satisfatório. A vantagem destes métodos é que são gerais, permitindo a geração de planos de corte para qualquer problema de programação linear inteira.

A principal dificuldade para usar os métodos de Gomory e Chvátal é que as melhorias obtidas pela inserção de planos de corte são, em geral, pequenas e, portanto, levam a um custo computacional alto devido a grande quantidade de desigualdades que são necessárias para se resolver um problema.

Assim, pesquisadores começaram a estudar métodos automatizados para encontrar planos de corte particulares para cada problema. Esse tipo de abordagem têm dado bons resultados uma vez que o problema é visto de forma mais específica.

Ao leitor interessado em estudar com mais detalhes este tópico, recomendamos o artigo de Aardal [6].

Existem muitas formas de implementar algoritmos de planos de corte. Como forma de exemplificar a técnica, descrevemos, a seguir, o algoritmo 2.1 de Dantzig, Fulkerson e Johnson que ilustra o método de planos de cortes[8]:

```
var P;        //P é o poliedro do problema;
var Q;        // Q é o poliedro fracionário inicial;
var c;        // função objetivo;

Q ← {poliedro inicial};
y ← solução ótima do programa linear (Q,c);
Enquanto y pode ser separado de Q faça
   Seja uma desigualdade ax ≤ b que separa P de y
   Q ← Q ∩ {x : ax ≤ b};
   y ← solução ótima do programa linear (Q,c);
Se Q = ϕ, devolva ϕ
Senão, devolva y
```

Algoritmo 2.1: algoritmo planos de corte

Seja (P, c) um programa linear de minimização onde P é um poliedro do problema e c é a função objetivo. A idéia do algoritmo 2.1 é considerar inicialmente um poliedro Q, tal que $P \subseteq Q$ descrito por poucas desigualdades e, a cada iteração, inserir desigualdades de forma a separar y de P sem perder nenhuma solução de P. A figura 2.5 exemplifica o processo.

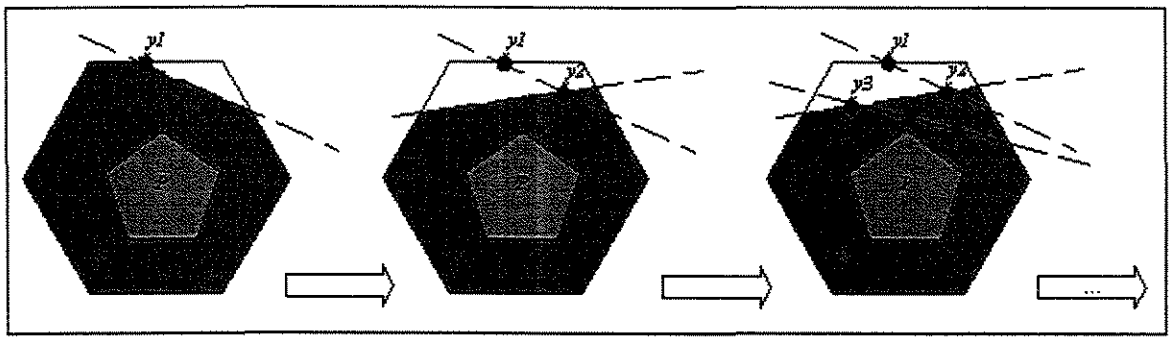


Figura 2.5: exemplo do processo de inclusão de planos de cortes

Para o caso de geração de planos de corte para programação linear inteira, a idéia é semelhante, porém antes relaxamos a condição de integralidade e repetidamente inserimos planos de corte.

Como já vimos anteriormente, como forma de otimizar o tempo computacional necessário para resolução do problema original, é importante tentarmos incluir planos de cortes particulares ao nosso problema. Planos de cortes considerados bons são aqueles que induzem facetas. Neste caso os planos de corte tocam o poliedro do problema.

Nos últimos anos, as técnicas de plano de cortes tem sido usadas juntamente com os algoritmos de *Branch & Bound* e são comumente chamados de *Branch & Cut*, trazendo resultados satisfatórios em termos computacionais.

2.5. Branch & Cut

O método *Branch & Cut* combina os métodos de planos de corte com os métodos de *branch & bound* para resolver problemas de otimização.

A idéia do método consiste em utilizar o método de planos de corte em cada um dos nós da árvore de decisão obtida pelo método de *Branch & Bound*. O sucesso do método está intimamente ligado à geração de bons limitantes inferiores em cada nó, ou seja, é preciso encontrar inequações (planos de cortes) válidas de boa qualidade.

Embora o processo pareça simples, é importante pensar em estratégias que limitem o crescimento da árvore de decisão e delimitem o problema, como, por exemplo, utilizando:

- estratégias de separação
- heurísticas (veja capítulo 4 para mais detalhes)
- métodos de ramificação
- pré-processamento em cada nó
- gerenciamento de desigualdades válidas.

O algoritmo 2.2 apresenta um exemplo simplificado de implementação do método *Branch & Cut*:

```
var P';          //programa linear relaxado
var P;           //programa linear inteiro
var UB;          //limitante superior
var ativos;      //guarda o conjunto de nós ativos no problema
x* ← uma solução heurística sobre P';      // pode ser ϕ se não encontrar
                                           //solução viável
UB ← val(x*)   // val (ϕ) = +∞;
ativos ← {P'};
Enquanto houver nós ativos faça
      Escolha P ∈ ativos;
      Remova P de ativos;
      Insira planos de corte em P;
      x ← solução ótima de P;              // pode ser ϕ se P é inviável
      se val(x) ≥ UB então ramo é podado;
      se val(x) < UB então
            se x é inteiro então
                  UB ← val(x);
                  x* ← x;
            senão
                  crie dois subproblemas P1 e P2 a partir de P;
                  ativos ← ativos ∪ {P1, P2};
devolva x;
```

Algoritmo 2.2: algoritmo *branch & cut* para problema de minimização

Como já vimos anteriormente, muitas vezes é difícil encontrar inequações que satisfaçam as condições necessárias para planos de corte, para um determinado problema. Dessa forma, torna-se conveniente utilizar-se de mecanismos como o *pool* de desigualdades, descrito a seguir:

2.5.1. Pool de desigualdades

Para minimizar a dificuldade em se obter as inequações descritas acima, é utilizado o conceito de "*pool*" de desigualdades. Ou seja, as inequações que são encontradas pelos algoritmos de separação são armazenadas para utilização futura considerando-se que é muito mais fácil e rápido percorrer uma lista de inequações disponíveis a executar um algoritmo de separação para encontrar inequações apropriadas.

Alem de servir como depósito de desigualdades, o *pool* de desigualdades é usado também para gerenciar desigualdades que não são mais usadas por nenhum nó da árvore de *Branch & Cut*. Neste caso, tais desigualdades são removidas do sistema.

Dessa forma, se por um lado é interessante manter um depósito de inequações para facilitar as buscas, por outro, esta ação pode comprometer a capacidade de

armazenamento de dados do sistema. Assim, é importante pensar em mecanismos de descarte de inequações que não são utilizadas por algum tempo pré-estabelecido. Este processo é chamado, em geral, de coletor de lixo e é executado periodicamente no sistema.

A figura 2.6, apresentada a seguir, exemplifica o conceito de desigualdades "velhas" no pool de desigualdades:

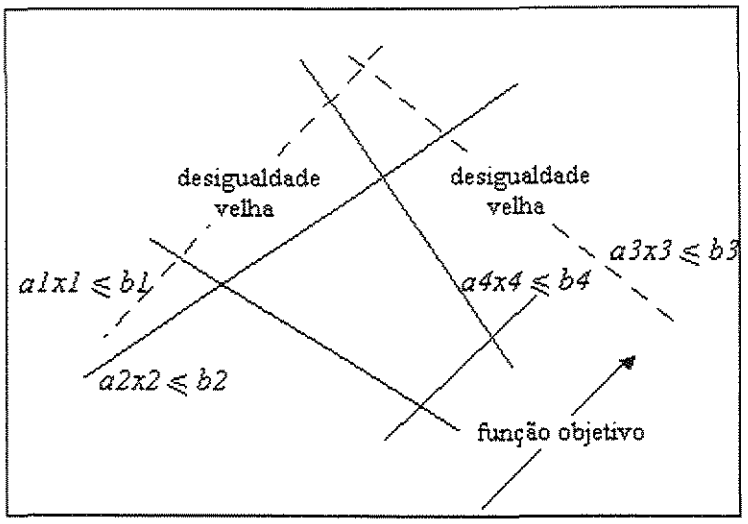


Figura 2.6: exemplo de desigualdades velhas do pool de desigualdades

Na figura 2.6, as linhas tracejadas, representadas por $a_1x_1 \leq b_1$ e $a_3x_3 \leq b_3$, são desigualdades velhas uma vez que as desigualdades representadas $a_2x_2 \leq b_2$ e $a_4x_4 \leq b_4$ limitam melhor o problema. Assim, essas desigualdades podem ser removidas do pool pelo coletor de lixo na próxima vez que este for executado.

2.6. Algoritmos de Aproximação

Algoritmos de aproximação surgiram para auxiliar na resolução de problemas de otimização combinatória uma vez que muitos desses problemas são NP-Difíceis e, portanto, de difícil resolução em termos computacionais.

A idéia básica consiste em abrir mão da solução ótima por uma aproximação de boa qualidade que é obtida de forma eficiente. Conforme descrito em [7], esse compromisso entre perda de otimalidade e ganho em eficiência é o paradigma dos algoritmos de aproximação.

Embora a idéia seja simples, muitos algoritmos de aproximação envolvem provas sofisticadas e encontrar uma solução aproximada pode ser tão difícil quanto encontrar uma solução ótima.

É importante porém observar que apesar de sacrificar a otimalidade, a aproximação deve ser feita de forma que ainda possamos dar boas garantias sobre o valor da solução obtida, procurando ganhar o máximo em termos de eficiência computacional

O estudo de algoritmos de aproximação, apesar de ser anterior à teoria de NP-completude, ganhou força nos últimos anos com o uso de técnicas de caráter mais geral. Dentre esta, podemos citar as seguintes técnicas e métodos:

- métodos combinatórios;
- métodos usando programação linear;
- programação semidefinida;
- métodos probabilísticos;
- relaxação lagrangeana.

Recomendamos ao leitor interessado no assunto a leitura do livro [7].

Dado um algoritmo de aproximação A, denotamos por $A(I)$ o valor da solução gerada pelo algoritmo A sobre a instância I. Denotamos por $OPT(I)$ o valor de uma solução ótima de I. Dizemos que A é α-aproximado para o PRR se $A(I) \leq \alpha\ OPT(I)$ para toda instância I.

Agora que já apresentamos as técnicas matemáticas que são utilizadas para resolver problemas de otimização, vamos descrever matematicamente o PRR, proposto no início deste trabalho.

Para tanto, o próximo capítulo apresenta a formulação matemática do problema e as desigualdades que são construídas para viabilizar a sua resolução.

Capítulo 3

3. Algoritmo Exato

Conforme discutido por Stoer [1], para problemas como os estudados neste trabalho, onde só é exigido disponibilidade de links de conexão (arestas), os principais algoritmos que tem sido utilizados na prática, são os algoritmos de planos de corte combinados com as técnicas de *branch & bound*. Um exemplo de algoritmo que utiliza esses conceitos é o algoritmo de Christofides e Whitlock [1].

O algoritmo de Christofides e Whitlock trata de casos mais gerais que o nosso, considerando requisitos de conectividade de arestas de grau N, podendo N ser superior a dois, porém a idéia básica do algoritmo de Christofides e Whitlock consiste em utilizar apenas desigualdades de corte e inequações triviais. Segundo Christofides e Whitlock, esse algoritmo, quando combinado com as técnicas de *branch & bound*, permite resolver problemas com mais de uma centena de nós.

A seguir, apresentamos o nosso problema em termos matemáticos, utilizando programação linear inteira.

3.1. Formulação em programação linear inteira

Conforme já descrito anteriormente, estamos interessados em encontrar uma rede de custo mínimo que satisfaça as seguintes condições de conectividade:

- entre pontos críticos deve haver, pelo menos, dois links de conexão disjuntos nas arestas;
- entre pontos ordinários deve haver, pelo menos, um link de conexão;
- os demais pontos não precisam estar, necessariamente, ligados por links de conexão.

A figura 3.1, apresentada a seguir, exemplifica uma rede nessas condições:

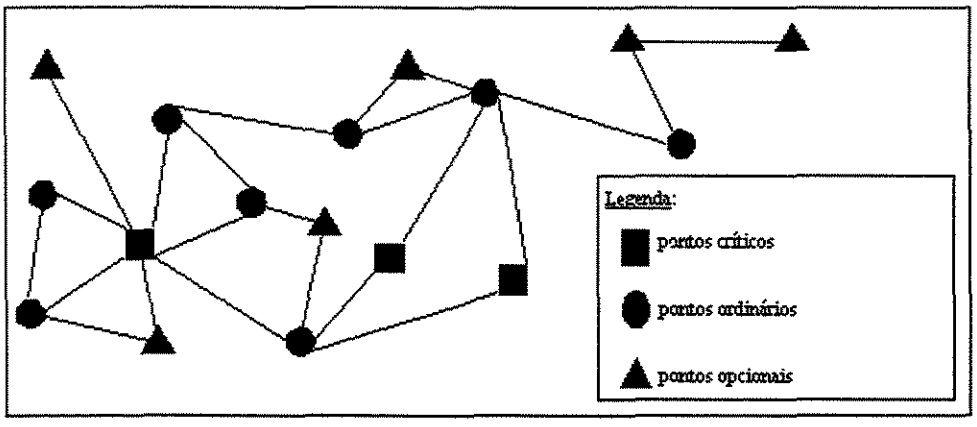


Figura 3.1: exemplo de rede com pontos críticos, ordinários e opcionais.

Já a figura 3.2, apresentada a seguir, representa uma solução possível de configuração da rede para a rede exemplificada na figura 3.1:

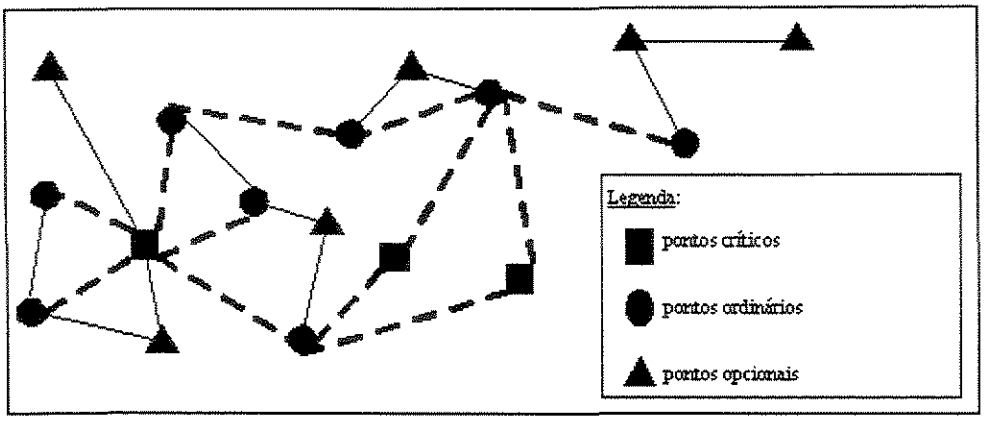


Figura 3.2: exemplo de uma solução para o PRR proposto na figura 3.1

Na figura 3.2, as linhas tracejadas representam uma configuração possível para o PRR exemplificado na figura 3.1. Note que os pontos críticos possuem pelo menos dois links de conexão entre si. Analogamente, os pontos ordinários possuem pelo menos um link de conexão entre si.

Dado um grafo de entrada $G=(V,E)$ para o PRR, denotaremos por V_C e V_O o conjunto de vértices críticos e ordinários do conjunto V, respectivamente.

Sem perda de generalidade, vamos considerar que se há vértices críticos, haverá pelo menos dois deles.

A modelagem matemática deste problema pode ser descrita na forma de programação linear inteira conforme segue:

Dados:

$$\begin{cases} \text{um grafo } G = (V,E), \\ \text{uma função de custo nas arestas } c\text{: } E \rightarrow R^+, \\ \text{uma função } r\text{: } V \rightarrow \{0,1,2\}. \end{cases}$$

Encontrar x que

$$\text{minimize} \sum_{e \in E} c_e x_e \qquad (1)$$

sujeito a:

$$\begin{cases} \sum_{e \in \delta(S)} x_e \geq f(S), \ \forall S \subseteq V : S \text{ separa algum conjunto de vértices,} & (2) \\ x_e \in \{0,1\}, & (3) \end{cases}$$

onde:

$$f(S) = \max \{\min\{r_u, r_v\}, u \in S, v \in V \setminus S\}, \qquad (4)$$

$$\delta(S) = \{\{u,v\} \in E \mid u \in S, v \in V \setminus S\}.$$

Na formulação acima, (1) representa a função objetivo para construção da rede representada pelo grafo $G=(V,E)$ que satisfaz as restrições de conectividade desejadas. As desigualdades de corte (ou restrições de conectividade) são representadas por (2). A integralidade da solução é representada por (3) sendo que, dados $u, v \in V$:

- $x_{uv} = 1$, indica que a aresta que liga u a v pertence à solução
- $x_{uv} = 0$, indica que a aresta que liga u a v não pertence à solução

Observe que a restrição (2) impõe que todo corte que separa u e v deve ter pelo menos duas arestas ligando dois pontos críticos e pelo menos uma aresta ligando pontos ordinários. A função f indica a quantidade mínima de arestas necessárias em qualquer corte $(S, V \setminus S)$.

A formulação descrita a seguir representa a relaxação do programa linear inteiro descrito acima:

(5)

Dados:

- um grafo $G = (V,E)$,
- custos nas arestas c: $E \rightarrow R^+$,
- uma função r: $V \rightarrow \{0,1,2\}$.

Encontrar x que

minimize $\sum_{e \in E} c_e x_e$

sujeito a:

- $\sum_{e \in \delta(S)} x_e \geq f(S)$, $\forall S \subseteq V$: S separa algum conjunto de vértices,
- $x_e \in [0,1]$,

onde:

$f(S) = \max \{\min\{r_u, r_v\}, u \in S, v \in V \setminus S\}$,

$\delta(S) = \{\{u,v\} \in E \mid u \in S, v \in V \setminus S\}$.

Na modelagem (5), abrimos mão da integralidade fazendo com que $x_e \in [0,1]$ ao invés de assumir apenas 0 ou 1.

A figura 3.3 apresenta uma rede contendo pontos críticos e ordinários e um corte S separando dois pontos críticos:

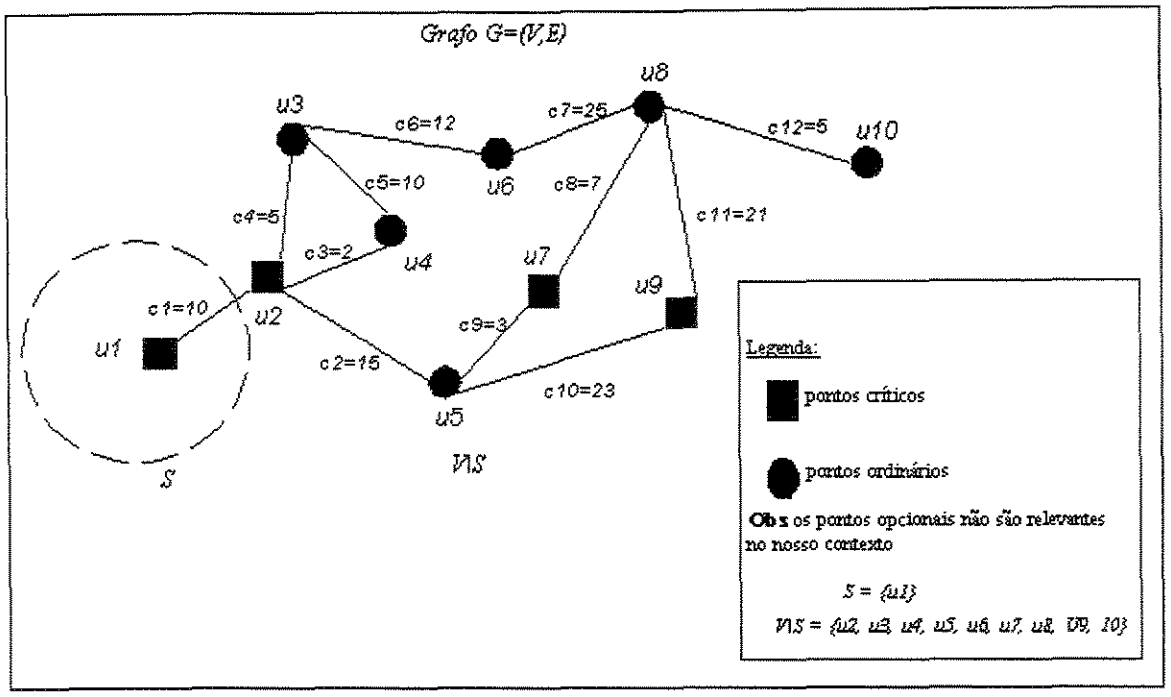


Figura 3.3: exemplo do PRR contendo corte violado S separando pontos

No exemplo, o Grafo $G=(V,E)$ é representado pelo conjunto de vértices $V = \{u_1, ..., u_{10}\}$ e pelo conjunto de arestas $E = \{(u_1, u_2), (u_2,u_4), (u_2,u_3), (u_2,u_5), (u_3,u_4), (u_3,u_6), (u_6,u_8), (u_8,u_{10}), (u_8,u_9), (u_8,u_7), (u_7,u_5), (u_9,u_5), (u_5,u_2)\}$.

O conjunto de pontos críticos e ordinários, no exemplo, é formado por:

$V_C = \{u_1, u_2, u_7, u_9\}$;

$V_O = \{ u_3, u_4, u_5, u_6, u_8, u_{10}, u_{11}\}$;

S, um corte tal que $u_1 \in S$ e $u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10} \in V \setminus S$ e

$\delta(S) = \{(u_1,u_2)\}$.

O exemplo apresentado na figura 3.3 não satisfaz as condições impostas pela restrição (2) uma vez que $u_1, u_2 \in V_C$ e S é um corte que viola a restrição de que pontos críticos devem possuir pelo menos dois links de conexão.

Em redes como a exemplificada, não existirá uma solução viável. Porém, em condições adequadas, certamente existem muitas formas de interligar os pontos de uma rede, de forma a preservar as restrições desejadas. Nosso objetivo é tentar encontrar não só uma rede viável, mas uma rede viável e de custo mínimo.

Vale observar que existem várias formulações, através de planos de cortes, para o problema, porém, por questões de simplicidade, foi escolhida a formulação acima, também apresentada em [1]. Além dos motivos de simplicidade, existem várias rotinas disponíveis publicamente que auxiliam na separação.

Considerando que o problema de redes com baixas restrições de conectividade pode ser expresso pelo modelo de programação linear inteiro definido aqui, vamos agora analisar um conjunto de desigualdades iniciais representadas em (2).

3.2. Desigualdades Iniciais

Conforme descrito anteriormente, na seção 2.4, o número de inequações de corte pode crescer exponencialmente. Assim, uma estratégia bastante utilizada para minimizar o problema é reduzir o número de inequações e tentar resolver o problema incluindo novas inequações conforme a necessidade.

Nesta seção apresentamos o conjunto inicial de desigualdades que são consideradas para começar a procura pela solução ótima do problema.

A idéia consiste em relaxar o programa linear inteiro descrito em (1)-(4) e começar com um pequeno número de restrições. O problema inicial a ser resolvido passa a ser representado por:

$$
(6)\quad \begin{cases} \text{Encontrar } x \text{ que} \\ \quad \text{minimize } \sum_{e \in E} c_e x_e \\ \\ \text{sujeito a:} \\ \quad \sum_{e \in \delta(v)} x_e \geq 2, \ \forall\, v \in V_C, \\ \\ \quad \sum_{e \in \delta(u)} x_e \geq 1, \ \forall\, u \in V_O, \\ \\ \quad 0 \leq x_e \leq 1, \ \forall\, e \in E. \qquad (7) \end{cases}
$$

Na formulação acima, a condição de integralidade foi relaxada para a condição em (7).

A idéia básica do algoritmo que resolve o problema consiste em analisar os pontos do grafo $G=(V,E)$ e inserir todas as respectivas desigualdades que garantem as restrições de conectividade.

Considere, inicialmente, um ponto $v \in V_C$. Seja $\{e_1, e_4, e_5, e_7, e_{10}\}$ as arestas incidentes em v. Como v é um ponto crítico da rede, a primeira inequação que é inserida ao problema será: $x_{e1} + x_{e4} + x_{e5} + x_{e7} + x_{e10} \geq 2$. Isto é, pelo menos duas arestas incidentes ao vértice v devem pertencer a solução. De forma análoga, todas as inequações correspondente aos pontos $v \in V_C$, são inseridas ao problema.

Analogamente, seja $u \in V_O$ e considere $\{e_3, e_9, e_{15}\}$ as arestas incidentes em u. Como u é um ponto ordinário da rede, a inequação $x_{e3} + x_{e9} + x_{e15} \geq 1$ deverá ser inserida ao problema. Novamente, todas as inequações correspondentes aos pontos $u \in V_O$ são inseridas ao problema.

A figura 3.4, representa graficamente os pontos u e v e as respectivas inequações associadas a eles:

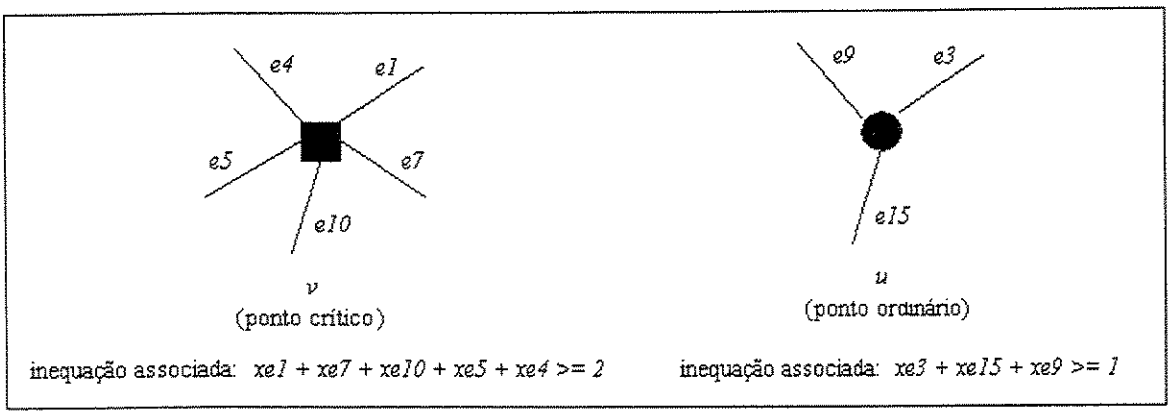


Figura 3.4: exemplo de desigualdades iniciais de pontos críticos e ordinários

O processo de inclusão de inequações deste tipo é feito para todos os vértices críticos e ordinários do grafo G. Note que estas restrições são casos particulares do conjunto de restrições apresentado em (2). Depois de concluído este processo, o algoritmo resolve o problema e verifica se a solução obtida é viável.

Se a solução encontrada não for viável, o próximo passo é incluir as inequações de corte. O processo de verificação de viabilidade é feito através do algoritmo para encontrar cortes mínimos.

3.3. Desigualdades de Corte

O problema agora consiste em determinar se um ponto x qualquer da nossa rede satisfaz realmente as desigualdades definidas em (2). Ou seja, como podemos garantir que dois vértices u e v dados estão separados por um corte que viola a desigualdade de corte.

Vale observar que, conforme apresentado em [2], o número de desigualdades desse tipo é exponencial já que em um grafo $G=(V,E)$ com $|V| = n$ teremos 2^{n-1} cortes. Dessa forma, torna-se muito difícil inserir todas essas inequações no problema a fim de resolvê-lo com o apoio de uma aplicação computacional. Sabemos, porém, que embora o número de cortes seja exponencial, podemos separar as inequações correspondentes em tempo polinomial. Conseqüentemente, podemos obter a solução ótima do programa linear fracionário em tempo polinomial. Para resolver esta questão, utilizamos o chamado algoritmo de corte mínimo.

O problema do corte mínimo consiste em encontrar o corte de menor peso em uma determinada rede. O peso de um corte pode ser definido como a soma dos pesos das arestas que compõem este corte.

Matematicamente, podemos dizer que dados:

$$\begin{cases} \text{um grafo } G=(V,E), \\ \text{uma função de custo } c\text{: } E \rightarrow Q^{+} \text{ e} \\ \text{vértices } s \text{ e } t \in V, \end{cases}$$

o problema do corte mínimo consiste em encontrar subconjuntos disjuntos V_1 e V_2 tal que $V = V_1 \cup V_2$, que minimiza a soma dos custos das arestas com extremos em subconjuntos disjuntos.

Existem várias maneiras de se aplicar o algoritmo de corte mínimo, como por exemplo utilizar uma estratégia que testa todos os pares de vértices. Neste caso, a busca por um plano de corte pode ser encontrada em no máximo n^2 chamadas da rotina para encontrar corte mínimo.

Para fins deste trabalho, utilizamos o algoritmo para gerar árvores de corte de Gomory-Hu que gera n-1 cortes mínimos para representar todos os cortes entre quaisquer dois vértices do grafo.

O conceito da árvore de Gomory-Hu consiste em construir uma "árvore de cortes" a partir do grafo original $G=(V,E)$. Cada aresta da árvore de Gomory-Hu representa um corte. O corte representado pela aresta $e \in E$ é definido pelas duas componentes quando e é removido. O corte mínimo entre dois vértices u e v é dado pelo corte representado pela aresta de peso mínimo no caminho que liga u a v na árvore de Gomory-Hu.

A figura 3.5, representa um exemplo da "Árvore de Cortes de Gomory-Hu".

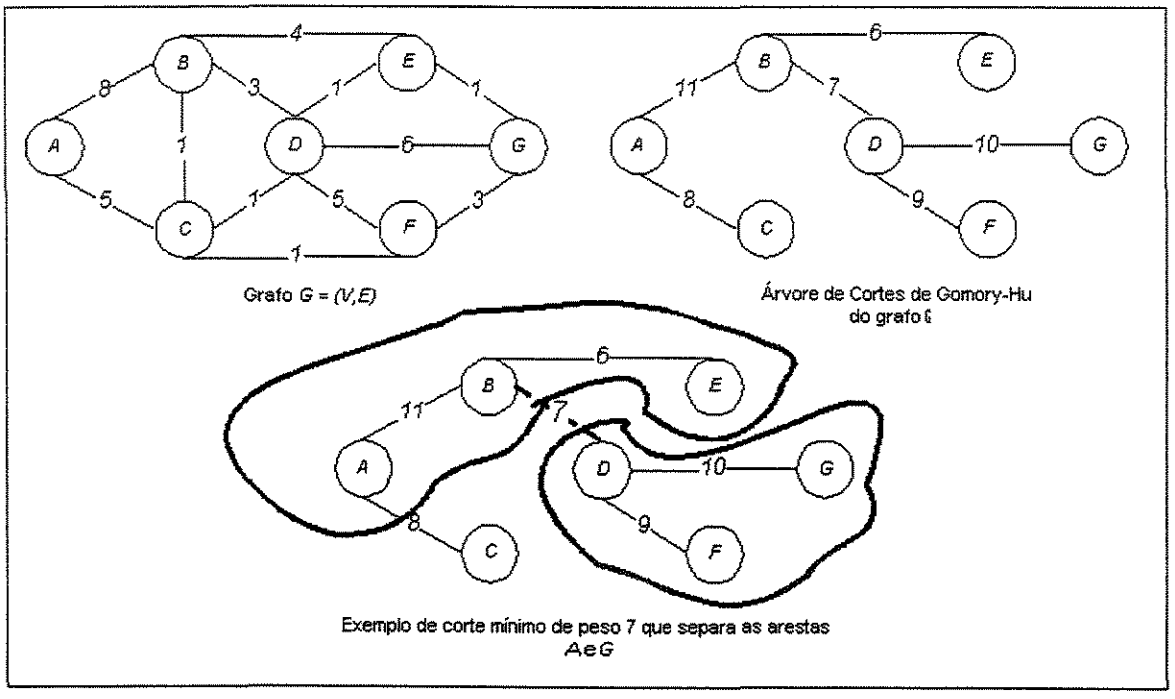


Figura 3.5: exemplo da árvore de Gomory-Hu

No exemplo apresentado acima, o corte mínimo que separa os vértices A e G tem peso 7 uma vez que 7 é o mínimo entre {11,7,10}.

A idéia geral consiste em encontrar as arestas da árvore de Gomory e Hu que representam cortes violados. Assim, a cada iteração construímos a árvore de Gomory-Hu, determinamos o corte mínimo que viola a desigualdade de corte e inserimos a desigualdade de corte no problema.

Dessa forma, a cada iteração, uma nova desigualdade de corte é inserida no problema.

O algoritmo apresentado em 3.1 verifica se uma solução é violada utilizando a árvore de Gomory-Hu:

var x; //estrutura que armazena uma solução
var G; //estrutura que armazena grafo
var V; //estrutura que armazena vértices
var E, C; //estrutura que armazena arestas
var D; //estrutura que armazena desigualdades
var x_e; //armazena os custos nas arestas $e \in E$
var T, T', T''; //estruturas que armazenam arvores

$x \leftarrow$ solução fracionária de (6) recebida como parâmetro;
Seja $G = (V,E)$ e custos x_e nas arestas $e \in E$;;
Seja $T = (V,F)$ uma árvore de cortes de Gomory-Hu com pesos nas arestas $gh(e)$;
$D \leftarrow \phi$; //conjunto de desigualdades
Para cada aresta $e \in F$ *faça*
    Seja T' e T'' as sub-árvores obtidas de T pela remoção da aresta e;
    Seja C o conjunto de arestas de G ligando vértices de T' e T'';
    Seja $f_e = \max \{\min \{r_u, r_v: u \in T', v \in T''\}\}$
    Se $f_e > gh(e)$ // encontramos um corte violado
        $D \leftarrow D \cup \{ \sum_{d \in C} x_d \geq f_e \}$

Devolva D;

Algoritmo 3.1: exemplo de algoritmo para achar cortes violados utilizando a árvore de Gomory-Hu

Além das desigualdade de corte, outro tipo de desigualdades que são inseridas no problema são as desigualdades de partição que são apresentadas a seguir.

3.4. Desigualdades da Partição

Nesta seção apresentamos as classes de desigualdades de partição para o nosso problema de disponibilidade de rede.

Desigualdades de partição generalizam as desigualdades de corte e aparecem como inequações válidas para problemas de disponibilidade de redes, estudados neste documento.

As desigualdades de partição, sob condições adequadas, são facetas limitantes para o problema de Steiner e para o nosso problema de disponibilidade de redes, conforme citado por Magnanti e Raghavan [10].

Assim sendo, as desigualdades de partição tornam-se ferramentas importantes para o estudo objeto deste documento. Em geral, desigualdades de partição são usadas

junto com as desigualdades de corte e as desigualdades iniciais no algoritmo de *Branch & Cut* para resolver problemas como os estudados neste documento.

As desigualdades de partição consideradas no desenvolvimento deste trabalho foram definidas como:

Dados:

$$\begin{cases} \text{grafo } G=(V,E), \\ W=(V_1,\ldots,V_k) \text{ uma partição de } V, \\ \delta(W) = \{\{uv\} \in E \mid u \in V_i, v \in V_j \text{ e } i \neq j, \text{ onde } V_i \cap (V_C \cup V_O) \neq \phi\}. \end{cases}$$

As desigualdades apresentadas em (8) são desigualdade de partição:

$$(8) \quad \begin{cases} \sum_{e \in \delta(W)} x_e \geq k & \text{se } \exists\, u, v \in V_C,\ u \neq v, \text{ tal que, } u \in V_i,\ v \in V_j,\ i \neq j \\ \sum_{e \in \delta(W)} x_e \geq k-1 & \text{caso contrário} \end{cases}$$

As duas inequações acima indicam que se existem duas ou mais partições contendo pontos críticos, então a rede deve possuir pelo menos k arestas entre essas partições; caso contrario, a rede possui pelo menos (k-1) arestas entre as partições.

A figura 3.6, apresentada em [2], representa um exemplo de desigualdade da partição:

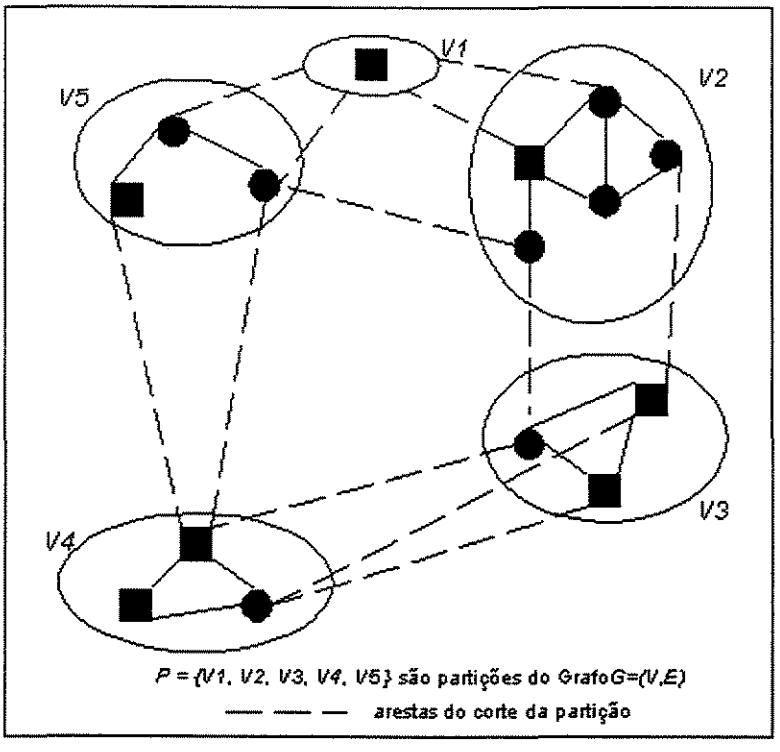


Figura 3.6: exemplo de desigualdade de partição

No exemplo acima, o problema é particionado em 5 conjuntos sendo que as linhas tracejadas representam as arestas do corte da partição.

As heurísticas de separação para esse tipo de inequação tem sido bastante estudadas e Grötschel, Monma e Stoer [2], provaram que o problema da separação para a classe das partições de Steiner é NP-Difícil.

O problema agora é determinar as partições cuja desigualdade correspondente esteja violada por x tal que $x \in R^E$. Existem vários métodos heurísticos utilizados para encontrar as partições desejadas, desde métodos gulosos até métodos mais otimizados. Para fins deste trabalho não implementamos nenhum método para encontrar desigualdades de partição. Porém estudos mais direcionados a este tópico devem implementar algoritmos que determinem desigualdades de partições como exemplificado em [2] e descrito a seguir:

var P; // estrutura que armazena partições
var V; // estrutura que armazena os vértices do grafo $G=(V,E)$
var C; //estrutura que armazena corte mínimo
$P \leftarrow \{V\}$;
Enquanto $x\,(\delta(P)) \geq K_e(P)$ **faça**:
    Encontre $C \in P$ tal que $|C \cap (V_C \cup V_O)| \geq 2$;
    Se não encontrou **então** retornar erro;
    Seja $u, v \in C$, $r(u) \geq 1$ e $r(v) \geq 1$;
    Encontre o corte mínimo $(S, C \setminus S)$ em $G[C]$ separando u e v;
    $P \leftarrow (P \setminus C) \cup \{S, C \setminus S\}$;

// Onde $K_e(P)$ é definido da seguinte forma:
// Seja:
// $K_{eO}(P) = |\{C \in P : V_O \cap C \neq \phi\}|$
// $K_{eC}(P) = |\{C \in P : V_C \cap C \neq \phi\}|$

$$// K_e(P) = \begin{cases} K_{eO}(P) + K_{eC}(P) - 1 & \text{se } K_{eC}(P) \leq 1 \\ K_{eO}(P) + K_{eC}(P) & \text{caso contrário} \end{cases}$$

Algoritmo 3.2: heurística para determinar desigualdades de partição

No algoritmo 3.2, apresentamos uma heurística para determinação de inequações de partição. Esta heurística é apresentado por Ferreira e Wakabayashi em [2] e é conhecida como "heurística de cortes mínimos". Este método geralmente apresenta bons resultados, porém exige alto custo computacional.

Neste algoritmo, as partições podem ser determinadas utilizando-se a árvore de cortes de Gomory-Hu.

Apresentamos nas seções 3.2, 3.3 e 3.4 vários tipos de inequações que são inseridas ao nosso problema de programação linear (1). Porém, se por um lado é uma boa estratégia

inserir o maior número de desigualdades antes de iniciar a fase de *branching* - já que nesta fase criamos dois novos problemas a partir de um existente - por outro, quanto maior o número de desigualdades analisadas, maior o custo computacional necessário.

Para resolver esse paradigma, faz-se necessário selecionar de forma criteriosa quais desigualdades devemos adicionar ao nosso problema. Não existe uma fórmula exata para escolha das desigualdades a serem consideradas; existem apenas recomendações que têm trazido bons resultados, como, por exemplo, escolher desigualdades de classes diferentes. Outra boa prática quando nossa missão é procurar inequações violadas, é manter um pool de desigualdades, conforme descrito na seção 2.6.

Como resultado de todas essas considerações podemos concluir que implementar um algoritmo bom para a resolução de problemas NP-difíceis não é uma missão fácil e envolve muitas sofisticações. Assim, precisamos analisar cuidadosamente a estratégia que vamos seguir para alcançar nosso objetivo de maneira satisfatória.

Muitas vezes, a utilização de heurísticas é uma boa estratégia que podemos utilizar para minimizar custos computacionais ou mesmo para encontrar soluções viáveis iniciais que sirvam de base para procura das soluções ótimas.

O próximo capítulo apresenta com mais detalhes os algoritmos e as heurísticas que foram utilizadas neste trabalho para encontrar soluções ótimas para problemas de construção de redes com as características apresentadas no decorrer deste texto.

Capítulo 4

4. Algoritmos e Heurísticas

Neste capítulo apresentamos algumas abordagens heurísticas para obter soluções de custo reduzido para o PRR.

Existem vários métodos heurísticos que podem ser utilizados para construir uma rede viável inicial ou para reduzir o custo de projeto de redes existentes preservando a sua viabilidade. Abordagens heurísticas tem sido muito utilizadas para resolver uma variedade de problemas de otimização como o problema do caixeiro viajante e o problema de Steiner.

Dividimos a apresentação das heurísticas em duas partes. Na seção 4.1 apresentamos as heurísticas de construção e na seção 4.2 apresentamos as heurísticas de melhoria.

4.1. Heurísticas de construção

Nesta seção apresentamos as chamadas heurísticas de construção, ou seja, as heurísticas que viabilizam uma forma para determinar uma solução inicial para o problema.

Por questões de simplicidade, abordamos o algoritmo desenvolvido por Jain [11] para obter uma solução inteira do problema de programação linear apresentado em (3.1) e propomos uma adaptação do mesmo para estudar o problema em questão.

Conforme apresentado por Jain [11], arredondamentos baseados em algoritmos de aproximação usam soluções fracionárias ótimas do programa linear relaxado - ou seja, programação linear sem a exigência de integralidade - para obter uma solução inteira para o problema original.

O processo empregado por tais algoritmos consiste em primeiro encontrar uma solução ótima para o problema de programação linear relaxado e, posteriormente, aplicar processos de arredondamento para converter a solução fracionária obtida, provavelmente fracionária, em uma solução inteira. Estes arredondamentos são realizados até que uma solução viável seja obtida.

Entre os vários algoritmos de aproximação existentes, escolhemos o algoritmo proposto por Jain [11] devido a sua simplicidade de implementação e por ser o que apresenta o melhor fator de aproximação conhecido para este problema. Outros algoritmos de aproximação podem ser encontrados em [19][11][20][12].

4.1.1. O Algoritmo de Jain

Jain [11] apresenta uma generalização da técnica de arredondamento para o PRR resolvendo primeiro o programa linear de forma a obter uma solução fracionária e depois arredondando iterativamente essa solução.

Para este propósito, Jain provou em [11] que qualquer solução básica, não inteira, do programa linear relaxado tem pelo menos uma aresta e, tal que x_e é pelo menos ½.

Com base neste resultado, Jain considera o conjunto de todas as arestas que possuem x_e pelo menos ½ e arredonda seu valor para x_e=1 obtendo um novo programa linear. Após este arredondamento, obtemos um grafo residual G_{res} para o qual a estratégia de arredondamento se repete. A formulação do novo programa é descrita a seguir.

Dados:

$$\begin{cases} \text{o grafo } G = (V,E), \\ \text{o grafo } G_{res} = G - E_{1/2}, \text{ onde } E_{1/2} = \{e \in E \mid x_e \text{ é pelo menos } 1/2\}. \end{cases}$$

Encontrar x que

$$\text{minimize} \sum_{e \in E(G_{res})} c_e x_e$$

Sujeito a:

$$\begin{cases} \sum_{e \in \delta_{Gres(S)}} x_e \geq f(S) - \sum_{e \in E_{1/2} \cap \delta(S)} 1 & \text{para } \forall S \subseteq V, \\ 0 \leq x_e \leq 1, \text{ para } \forall e \in E(G_{res}), \end{cases}$$

onde:

$$f(S) = \max \{\min\{r_u, r_v\}, u \in S, v \in V \setminus S\},$$

$$\delta_{Gres}(S) = \{\{u,v\} \in E \mid u \in S, v \in V \setminus S\}$$

Se o problema proposto tem solução, o algoritmo de Jain sempre retorna uma solução viável. O seguinte teorema é valido:

Teorema: O algoritmo de Jain é uma 2-aproximação para o PRR [11].

4.1.2. A adaptação do algoritmo

Para fins deste estudo, implementamos uma adaptação do algoritmo de Jain arredondando para 1 as arestas e tal que x_e é pelo menos ½. Caso não existam tais arestas, arredondamos para 1 as arestas de maior valor fracionário.

Optamos por fazer uma adaptação do algoritmo de Jain, obtendo a solução fracionária através do método de planos de cortes. Assim, não podemos garantir que existirão arestas e tal que x_e é pelo menos ½.

Essa implementação está representada pelo algoritmo (4.1), apresentado a seguir:

```
var x;          //estrutura que guarda uma solução do problema
var fixo;       //estrutura que armazena o conjunto de arestas maiores que 0.5
var r;          //estrutura que armazena custos

x ← uma solução ótima para o problema (possivelmente fracionária)
Enquanto x não representa uma solução viável faça
        r ← max {x_e: e ∈ E, x_e < 1 };
        se r > 0.5 então r ← 0.5;
        fixo ← { e ∈ E : x_e ≥ r};
        x ← uma solução ótima do problema com a restrição adicional x_e = 1 ∀ e ∈ fixo
```

Algoritmo 4.1: adaptação do algoritmo de Jain

4.1.3. Heurística de Redução

As heurísticas de redução são heurísticas de construção que consistem em selecionar algumas arestas promissoras e posteriormente aplicar o algoritmo de *Branch & Cut* para o grafo reduzido formado pelas arestas escolhidas.

Para selecionar as arestas promissoras, o algoritmo resolve o programa linear fracionário e escolhe todas as arestas com valor $x_e > 0$.

As arestas escolhidas são removidas e novas arestas são escolhidas repetindo-se o mesmo processo.

Vamos denotar por k-redução a heurística de redução que escolhe as arestas promissoras após k iterações.

O algoritmo 4.2, apresentado a seguir, representa a implementação da heurística k-redução que foi utilizada para fins deste trabalho:

```
var P;          // estrutura que armazena o programa linear relaxado
var i;          // auxiliar
var x;          // estrutura que armazena soluções de P
var k;          // número de iterações
Seja P o programa linear relaxado;
i ← 1; E_res ← ϕ;
Enquanto (i ≤ k) e (P é viável) faça
        Seja x uma solução ótima fracionária de P;
        E_i ← {e ∈ E : x_e > 0};
        E_res ← E_res ∪ E_i;
        P ← P ∪ { x_e = 0 ∀ e ∈ E_i };
        i ← i + 1;
Devolva solucão ótima de (G[E_res]).
```

Algoritmo 4.2: Algoritmo k-redução

4.1.4. Heurística arredondamento probabilístico

A heurística de arredondamento probabilístico é uma heurística simples, e bastante efetiva e pode ser aplicada em cada nó ativo. A heurística fixa algumas variáveis do nó em 1 e, em seguida, o programa linear é resolvido. Esse processo é repetido até que uma solução inteira seja encontrada ou até que o programa linear torne-se inviável. A escolha das variáveis que serão arredondadas é feita de forma probabilística.

Quando utilizamos a heurística de arredondamento probabilístico, existem algumas questões que devem ser levadas em consideração como quantas variáveis fixar e quantas arredondar, quando fazer o arredondamento probabilístico, quantas iterações fazer, etc.

Existem várias regras que podem ser implementadas para resolver essas questões. Por exemplo, em cada iteração, podemos analisar o número de variáveis fracionárias dos nós em questão. Se este número é baixo, o arredondamento probabilístico tem boas chances de chegar em uma solução com valor próximo do fracionário. Outra estratégia utilizada é quanto ao tempo de execução, ou seja, o algoritmo é repetido até ser atingido um tempo máximo de execução.

O algoritmo 4.3 apresenta a implementação que fizemos para uma chamada da heurística de arredondamento probabilístico:

var P; //estrutura que armazena poliedros
var x: //estrutura que armazena soluções
var E, E'; //estrutura que armazena arestas do poliedro P
var p_e, p_f; // estrutura que armazena probabilidades
$P \leftarrow$ poliedro referente ao nó que está sendo otimizado;
Insira planos de corte em P;
Enquanto P é viável e não encontrou solução inteira **faça**
    Seja x solução ótima fracionária de P;
    Seja $E' \leftarrow \{e \in E: 0 < x_e < 1\}$;
    Seja $p_e = \dfrac{x_e}{\sum_{x_e \in E'} x_e}$
    Escolha uma aresta f em E', cada aresta $e \in E'$ com probabilidade p_e de ser escolhida;
    $P \leftarrow P \cup \{ x_f = 1\}$;
    Insira planos de corte em P;

Algoritmo 4.3: heurística arredondamento probabilístico

A seguir, apresentamos as heurísticas de melhorias que podem ser utilizadas pelo leitor interessado no assunto para otimizar uma rede já existente.

4.2. Heurísticas de melhoria

Nesta seção apresentamos as técnicas heurísticas estudadas por Monma e Shallcross [9] onde são apresentadas as formas como o problema de projeto de rede de custo mínimo foi resolvido por eles. A abordagem apresentada por estes autores é bastante ampla e apresenta, também, heurísticas para vértices e arestas.

Utilizamos, neste trabalho, apenas as heurísticas designadas às arestas. O objetivo de tais heurísticas é reduzir o custo de uma rede já existente, preservando a sua viabilidade, utilizando-se de transformações locais que são rápidas e executadas em tempo satisfatório.

4.2.1. Heurísticas k-OPT

O objetivo das heurísticas k-OPT, $k \neq 1$, é diminuir o custo local dos circuitos de um determinado grafo G que representa a rede original. Para atingir o objetivo proposto, são trocadas k arestas de um circuito C, obtido aleatoriamente do grafo G, por outras k arestas de menor custo, de forma a obter um circuito C' de custo inferior ao custo de C.

Monma e Shallcross [9] estudaram casos particulares das heurísticas k-OPT, as heurísticas 2-OPT e 3-OPT, cujo objetivo é diminuir o custo de uma rede trocando, respectivamente, 2 e 3 arestas por outras de custo inferior.

No exemplo 2-OPT da figura 4.1, são consideradas duas arestas $x=(u_1,v_1)$ e $y=(u_2,v_2)$ de um circuito C de uma determinada rede que deseja-se otimizar. Aplicando-se a heurística 2-OPT, as arestas x e y são substituídas pelas arestas $x'=(u_1,v_2)$ e $y'=(v_1,u_2)$ com o objetivo de se obter um novo circuito C' com custo menor que o do circuito C e, conseqüentemente, obtendo um custo menor para toda a rede:

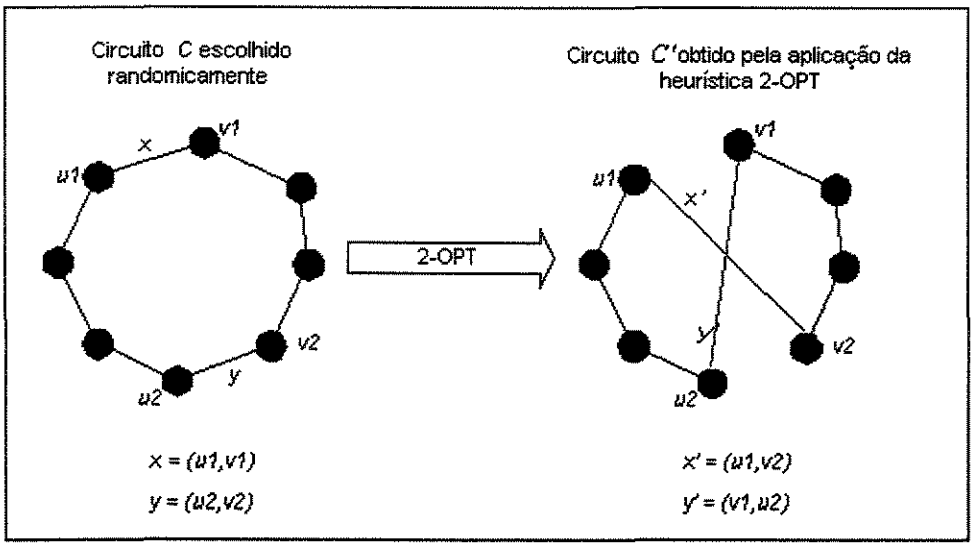


Figura 4.1: heurística 2-OPT

De forma análoga, no exemplo 3-OPT da figura 4.2 são consideradas três arestas $x=(u_1,v_1)$ e $y=(u_2,v_2)$, e $z=(u_3,v_3)$ de um circuito C de uma determinada rede que deseja-se otimizar. Aplicando-se a heurística 3-OPT, as arestas x, y e z são substituídas pelas

arestas $x'=(u_1,u_3)$, $y'=(v_1,v_2)$, $z'=(u_2,v_3)$ com o objetivo de se obter um circuito C' com custo menor que o do circuito C e, conseqüentemente, obtendo um custo menor para toda a rede:

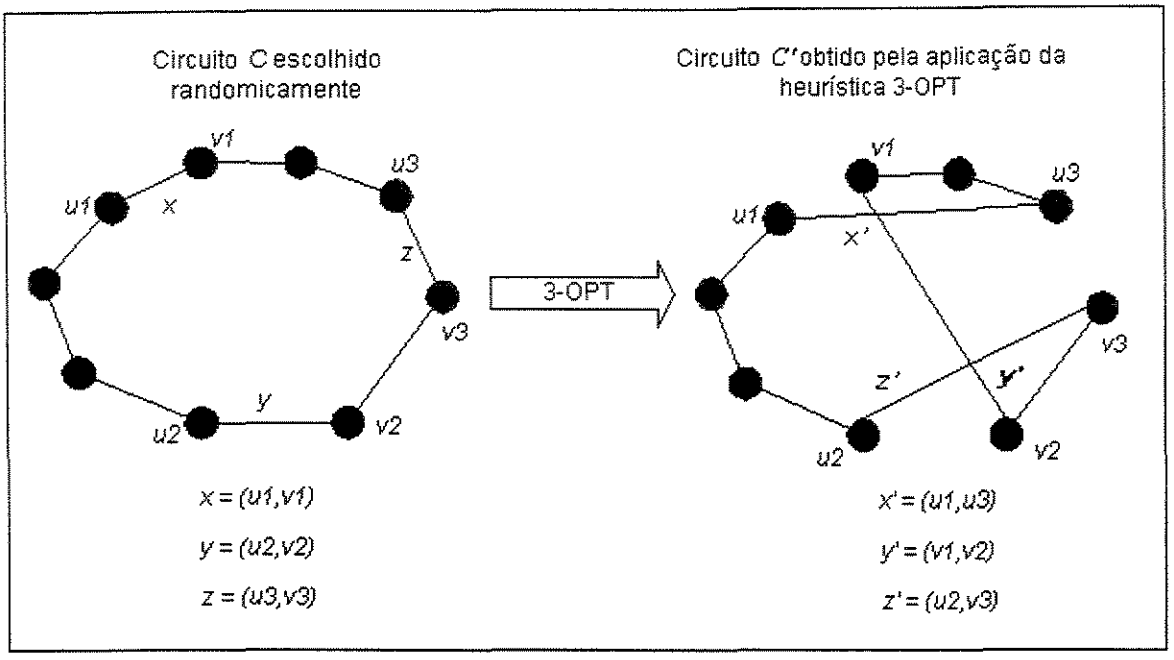


Figura 4.2: heurística 3-OPT

Sem perda de generalidade, as heurísticas k-opt são similares às heurísticas 2-OPT e 3-OPT, onde o objetivo é trocar k arestas de um circuito C de forma a obter um circuito C' de custo menor.

A implementação de heurísticas desse tipo pode ser feita por algoritmos similares ao algoritmo 4.4 apresentado a seguir:

```
var G;        //estrutura que armazena o grafo G que representa a rede
var C, C';    //estruturas que armazenam os circuitos do grafo G

Enquanto (é possível otimizar a rede) faça
        C = escolha randomicamente um circuito de G;
        C'= troca heuristicamente k arestas de C por outras de menor custo
        G = substitua o circuito C por C' no grafo G
```

Algoritmo 4.4: heurística k-OPT

Apresentamos até aqui as heurísticas k-OPT para $k\neq1$. A heurística 1-opt é um caso especial das heurísticas k-OPT, proposto por Monma & Shallcross[9], que considera a rede inteira ao invés de trabalhar com partes dela. O objetivo desta heurística é tentar trocar uma aresta do grafo G que representa a rede por outra de menor custo que não estava no grafo G, porém conservando a viabilidade na nova solução e reduzindo seu custo.

A figura 4.3 apresenta um exemplo da heurística 1-OPT:

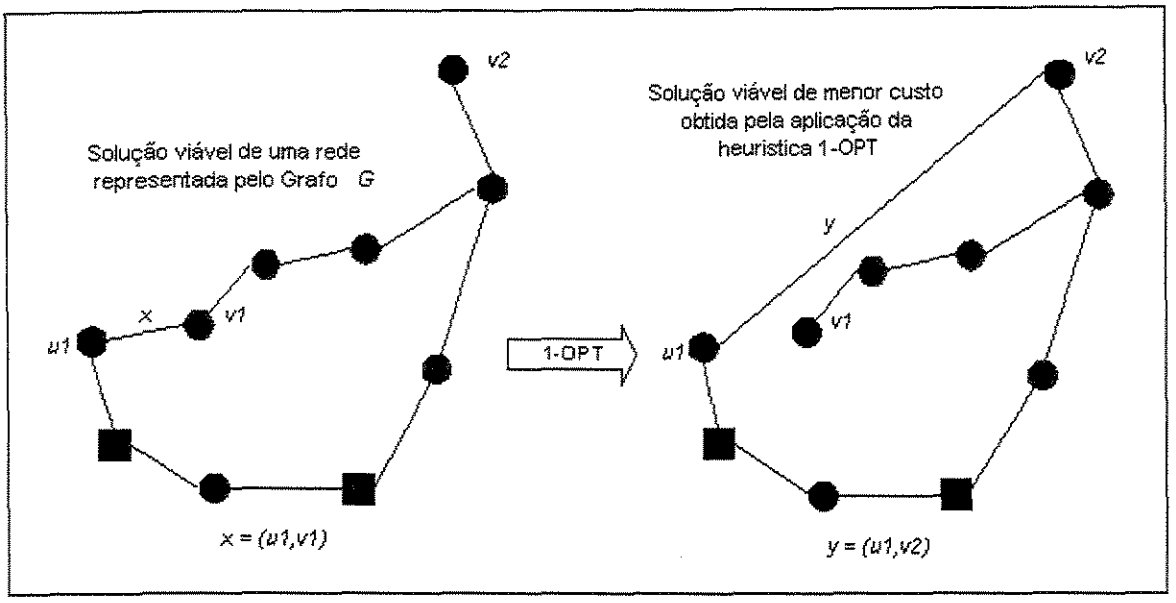


Figura 4.3: heurística 1-OPT

No exemplo da figura (4.3), a aresta $x = (u_1,v_1)$ é substituída pela aresta $y = (u_1,v_2)$ de forma que a nova rede obtida é viável e de menor custo.

A implementação de heurísticas desse tipo pode ser feita por algoritmos similares ao algoritmo 4.5 apresentado a seguir:

```
var G, G';          //estrutura que armazena o grafo G que representa a rede
var x, y;           //estrutura que armazena arestas
Para (cada nó u) faça
    Para (todas as arestas incidentes em u) faça
        x = tome (u,v1) aresta incidente em u;
        y = considere (u, v2) de custo menor, uma aresta que não estava
            contida na solução viável original;
        G' = troque a aresta x pela aresta y em G;
        Se G' é viável e custo(G) < custo(G') então
            G = G'    //G' passa a ser a melhor solução viável conhecida
```

Algoritmo 4.5: heurística 1-OPT

Já a viabilidade da nova solução pode ser testada pelo algoritmo 4.6, descrito a seguir:

```
var H;          //estrutura que armazena uma solução do problema
var V_H;        //estrutura que armazena os vértices da solução H
var E_H;        //estrutura que armazena as arestas da solução H
var X, Y;       //estrutura que armazenam grafos

H ← uma solução do problema recebida como parâmetro;
Seja H_1, ... ,H_k as componentes conexas de H;
Se existe H_i e H_j, i ≠ j, tal que v ∈ Hi e u ∈ H_j, r_u > 0 e r_v > 0
        Devolva (falso)
Senão
        Seja H' a componente de H contendo os vértices {v ∈ V_H: r_v > 0};
        Se existe aresta de corte e ∈ E_H tal que H' – e é desconexo, digamos em
                subgrafos X e Y tal que X ∩ V_C ≠ ϕ e Y ∩ V_C ≠ ϕ
                Devolva (falso)
        Senão
                Devolva (verdadeiro)
```

Algoritmo 4.6: algoritmo para testar viabilidade de uma solução

Por motivos computacionais, Monma & Shallcross [9] fizeram a escolha do ponto v_2 considerando a distância do ponto v_2 ao ponto u, dentro de um espaço delimitado que eles chamaram de uma janela de tamanho pré-definido.

As heurísticas k-OPT, $k \neq 1$, visam trocar k arestas por outras k arestas de menor custo, porém existem outras formas heurísticas que visam minimizar o custo de uma rede onde a troca de arestas é feita de forma assimétrica, ou seja, são trocadas k arestas por k' arestas de menor custo. Estas heurísticas são apresentadas a seguir.

4.2.2. Heurísticas Pretzel e Quetzel

Na heurística Pretzel, o objetivo é diminuir o custo local de circuitos de um determinado grafo G, que representa a rede original, trocando uma aresta do circuito C por outras duas cuja soma possui custo inferior à aresta que foi substituída.

No exemplo da figura 4.4, a aresta $x = (u_1,v_1)$ do circuito C, é substituída pelas arestas $y = (u_1,v_2)$ e $z = (v_1, u_2)$ de forma a obter um circuito C' de menor custo:

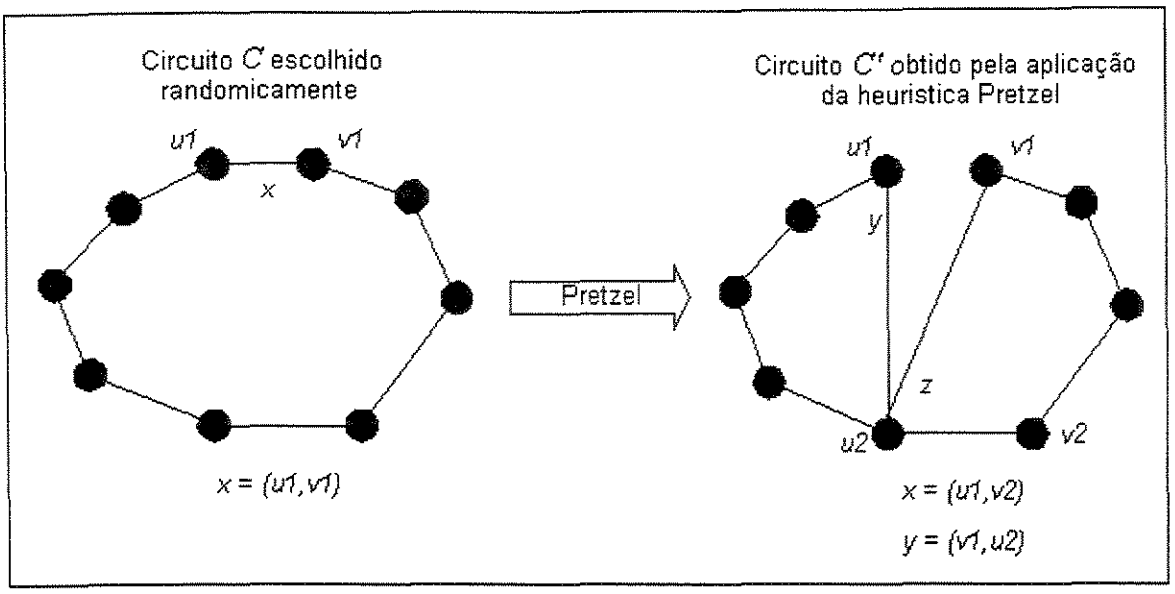


Figura 4.4: heurística Pretzel

A implementação de heurísticas desse tipo pode ser feita por algoritmos similares ao algoritmo 4.7 apresentado a seguir:

```
var G;          //estrutura que armazena o grafo G que representa a rede
var C, C';      //estruturas que armazenam os circuitos do grafo G

Enquanto (é possível otimizações da rede) faça
        C = escolha randomicamente um circuito de G;
        C'= troca heuristicamente uma aresta de C por duas outras de menor custo
        G = substitua o circuito C por C' no grafo G        //a solução deve ser viável
```

Algoritmo 4.7: heurística Pretzel

Ao contrário das heurísticas k-OPT, a heurística Pretzel é considerada uma transformação uma vez que a estrutura fundamental do circuito C original não é mantida após a troca das arestas.

Análoga à heurística Pretzel, a heurística Quetzel é uma transformação que age de forma inversa à heurística Pretzel trocando duas arestas por uma aresta de menor custo.

O exemplo da figura 4.5, é uma aplicação da transformação Quetzel que substitui as arestas $y = (u_1,v_2)$ e $z = (v_1, u_2)$ pela aresta $x = (u_1,v_1)$ obtendo um circuito de menor custo:

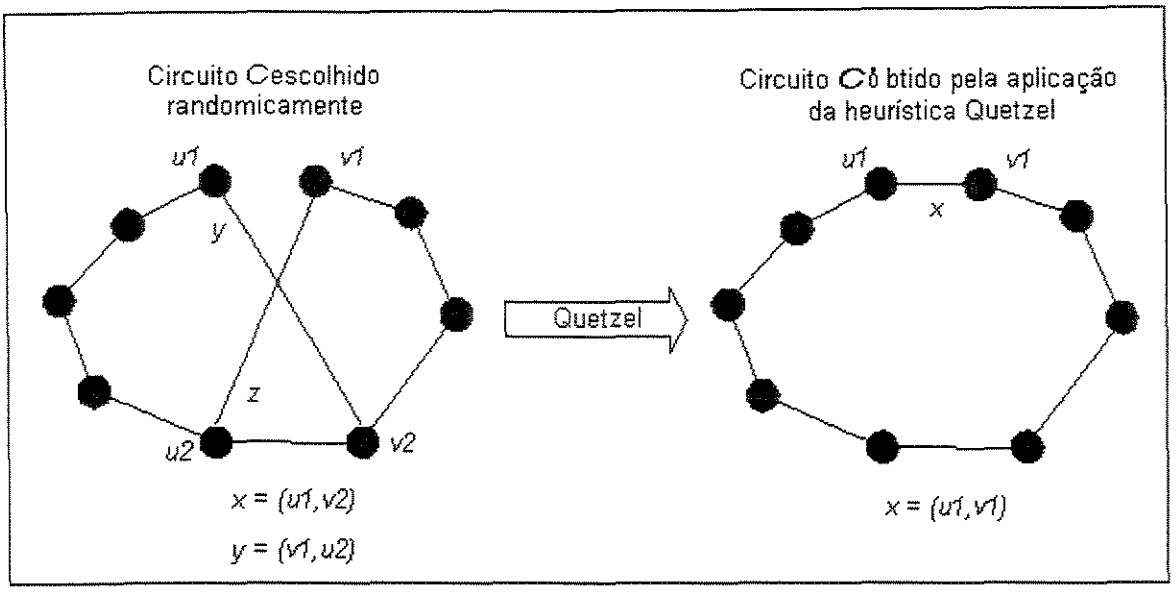


Figura 4.5: heurística Quetzel

A implementação de heurísticas desse tipo pode ser feita por algoritmos similares ao algoritmo 4.8 apresentado a seguir:

```
var G;          //estrutura que armazena o grafo G que representa a rede
var C, C';      //estruturas que armazenam os circuitos do grafo G

Enquanto (é possível otimizações da rede) faça
        C = escolha randomicamente um circuito de G;
        C'= troca heuristicamente duas aresta de C uma outra de menor custo
        G = substitua o circuito C por C' no grafo G   //a solucão final deve ser viável
```

Algoritmo 4.8: heurística Quetzel

As heurísticas k-opt, pretzel e quetzel são intuitivas e conceitualmente simples. Existem outras heurísticas mais complexas que também podem ser utilizadas para minimizar o custo de uma rede. Um exemplo desse tipo de heurística é a heurística Degree, porém esta só é válida para problemas que satisfazem a desigualdade triangular.

Não entraremos em detalhes quanto a esta heurística por se tratar de um caso particular. Porém sugerimos ao leitor interessado no assunto, a leitura do artigo de Monma & Shallcross [9] e referências lá mencionadas.

Para fins deste trabalho, implementamos como heurística de melhoria apenas a heurística 2-OPT, conforme apresentado na seção 5. Não implementamos nenhuma das outras heurísticas apresentadas por Monma e Shallcross [9]. Porém conforme o estudo realizado por eles no artigo [9], essas heurísticas tem apresentado bons resultados tanto quando aplicadas em casos reais, como em redes geradas randomicamente para estudos teóricos do problema.

4.3. Heurísticas de construção nos nós da árvore de Branch & Cut

Como vimos anteriormente, ao construir um algoritmo de *Branch & Cut* é importante investir em cada nó para evitar o crescimento da árvore e delimitar o problema.

Para atingir tais objetivos existem várias heurísticas que podem ser implementadas. Para fins deste trabalho implementamos a heurística de arredondamento probabilístico e a adaptação do algoritmo de Jain, conforme descrito a seguir.

4.3.1. Arredondamento probabilístico

Neste trabalho, nós utilizamos a heurística arredondamento probabilístico, apresentada anteriormente, em 20% dos nós.

4.3.2. Adaptação do algoritmo de Jain

Esta adaptação simplesmente executa o algoritmo de Jain, considerando as especificações atribuídas ao nó em questão.

A seguir, apresentamos as implementações que fizemos para apoio a este trabalho bem como os resultados computacionais obtidos por ela.

Capítulo 5

5. Aplicação de Teste

No decorrer deste trabalho desenvolvemos uma aplicação de teste com a finalidade de analisar os tópicos estudados nesta dissertação. A aplicação de teste foi construída de forma simples, apresentando de maneira básica as linhas mestras que podem ser seguidas por empresas, ou pessoas interessadas no assunto, que enfrentam problemas similares ao exposto neste trabalho.

Nossa aplicação não tem a pretensão de ser uma implementação ideal para o problema. Porém, no decorrer do processo, apresentamos conceitos teóricos que podem ser implementados em trabalhos futuros, por pessoas interessadas, dependendo da necessidade de aprofundamento de determinado tópico.

Apresentaremos a seguir a nossa implementação da aplicação de teste e, em seguida, os resultados computacionais obtidos pela execução da mesma.

5.1. Sobre a aplicação de teste

A nossa aplicação de teste implementa a estrutura do algoritmo de *Branch & Cut* com heurísticas simples que são apresentadas nessa seção.

A aplicação foi construída de tal forma que pudesse ser compilada para execução com alguns *solvers* de programas lineares distintos. O alvo principal foi o pacote XPRESS e para flexibilizar a utilização de diferentes *solvers*, usamos a biblioteca COIN/OSI, desenvolvida por pesquisadores da IBM [4].

A execução da aplicação consiste em executar a seguinte linha de comando:

sndp.exe *<arquivo descritivo da rede>* *<identificação das restrições>*

O parâmetro *<arquivo descritivo da rede>* corresponde ao arquivo contendo o grafo de entrada que descreve a rede no que diz respeito a quantidade de nós, arestas e custos de cada aresta. O arquivo de mesmo nome porém com extensão *<identificação das restrições>* contém as restrições de conectividade associadas à rede definida pelo arquivo descritivo. Dessa forma, poderíamos usar diferentes restrições para a mesma rede.

O exemplo a seguir descreve uma execução da aplicação de teste executada em uma amostragem de dados:

sndp.exe gr_50 a

No exemplo, o arquivo *gr_50* contém o grafo de entrada e possui o seguinte formato:

```
50 121                  (indica que o grafo possui 50 nós e 121 arestas)
50
49                      (nomeia cada um dos nós)
48
...
4  2        0.8486700
1  36       0.497370
1  44       0.879000    (atribui custos a cada uma das arestas)
37 28       0.340000
...
```

Exemplo 5.1: Exemplo de arquivo descritivo do grafo de entrada

Já o arquivo *gr_50.a* contêm as restrições de cada nó conforme exemplificado a seguir:

```
50      (indica que o grafo possui 50 nós)
1  1    (indica que o nó 1 é um ponto ordinário, portanto 1 ∈ V_O)
2  0    (indica que o nó 2 é um ponto opcional)
3  0    (indica que o nó 3 é um ponto opcional)
4  2    (indica que o nó 4 é um ponto crítico, portanto 4 ∈ V_C)
...
```

Exemplo 5.2: Exemplo de arquivo descritivo de restrições aplicado sobre o grafo de entrada

A aplicação de teste lê os dois arquivos que descrevem o problema e aplica o algoritmo de *Branch & Cut* para resolvê-lo conforme representado na figura 5.1:

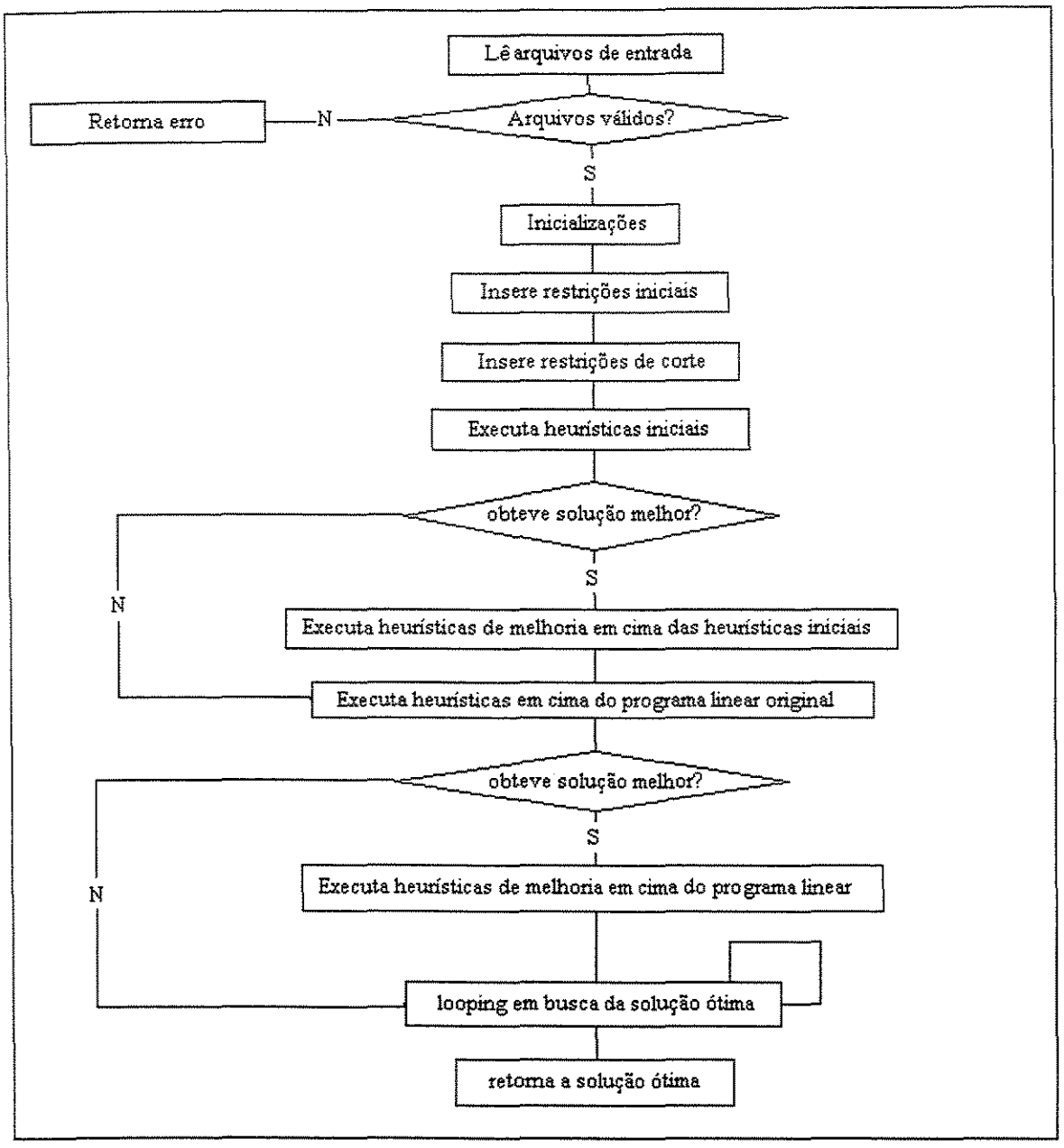


Figura 5.1: representação do fluxo principal da aplicação de teste

A figura 5.1 representa o fluxo principal de execução da aplicação de teste. Como forma de simplificar o entendimento da mesma, dividimos o fluxo da aplicação em duas partes:

- busca por uma solução heurística inicial
- busca pela solução ótima do programa linear original

A rotina nomeada como "*looping em busca da solução ótima*" é responsável pela segunda parte onde buscamos uma solução ótima para o programa linear original.

O fluxo básico para busca da solução ótima está representado na figura 5.2:

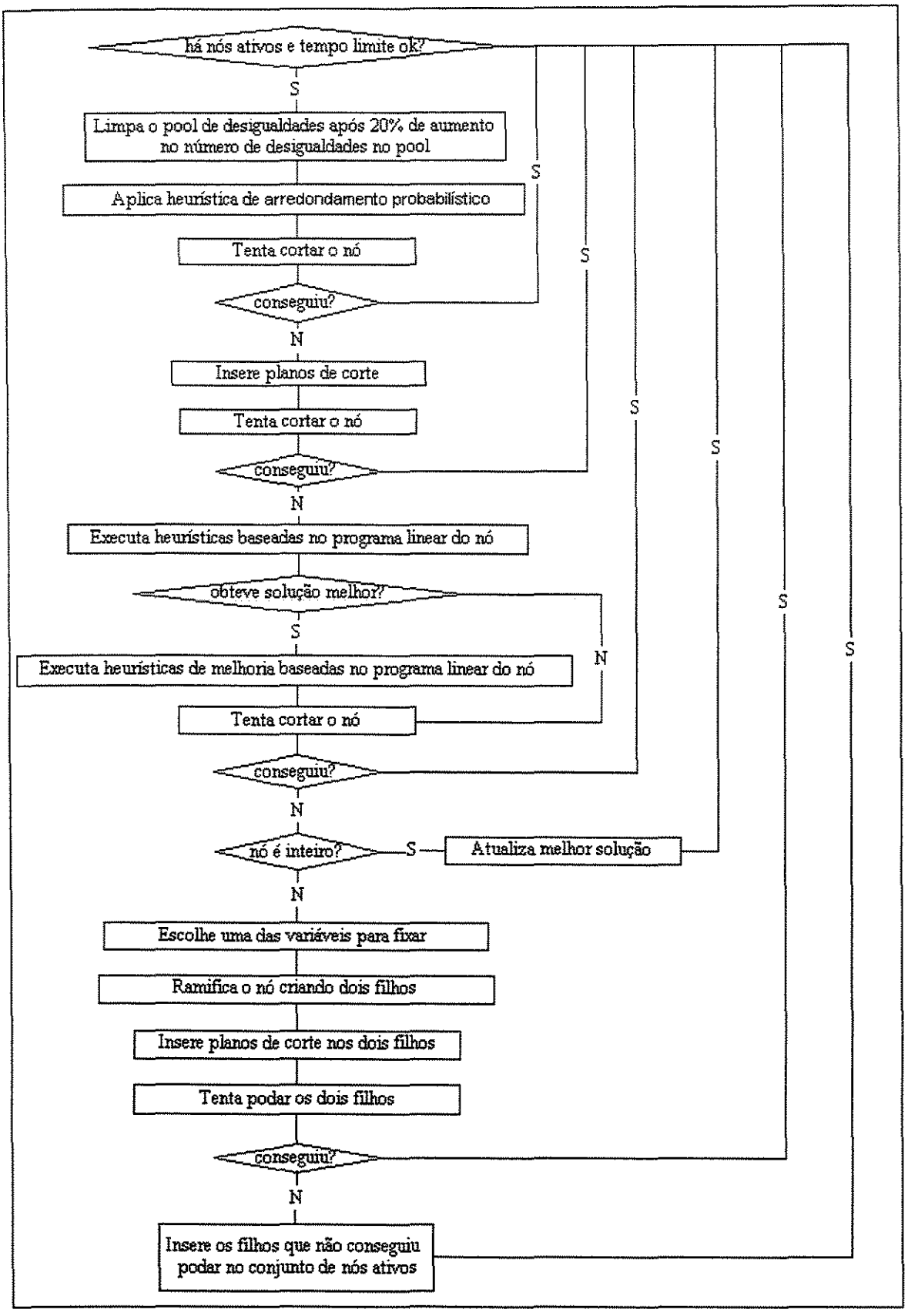


Figura 5.2: rotina de procura por uma solução ótima da aplicação de teste

A figura 5.2 representa o fluxo básico utilizado pela aplicação de teste para procurar por uma solução ótima para o programa linear original.

A rotina de inserção de restrições iniciais utilizada na aplicação de teste pode ser representada pelo algoritmo 5.1:

```
var G; // estrutura que armazena grafos
G ← grafo recebido como parâmetro;
Se o grafo é nulo retorna (falso);
Para todos os nós do grafo
        Se a restrição do nó for maior que zero        // nó não é opcional
                Identifica todas as arestas adjacentes ao nó;
                Insere a desigualdade dependendo da restrição associada ao nó;
Retorna (verdadeiro);
```

Algoritmo 5.1: inserção de restrições iniciais utilizado na aplicação de teste

Para obter as desigualdades de cortes, utilizamos as estruturas de dados *union-find* com *path compression*. O leitor que não conhece esses conceitos e estiver interessado em obter mais informações, aconselhamos a leitura do livro [15].

Para encontrar os cortes mínimos, tentamos utilizar duas estratégias distintas: uma através da inclusão de cortes mínimos testando todos os pares de vértices relevantes e outra, a árvore de Gomory-Hu. No primeiro caso a implementação ficou muito lenta o que nos levou a mudar de estratégia passando a utilizar apenas a árvore de Gomory-Hu.

O algoritmo 5.2 representa a rotina que foi utilizada pela aplicação de teste para geração de planos de cortes em um determinado nó:

```
var n;                  // estrutura que armazena nós
var P;                  // estrutura que armazena programa linear
var A;                  // estrutura que armazena a árvore de Gomory-Hu
var ncortes;            // contador de cortes
var max_u, max_v        // armazena requisitos de conectividade
var U, V;               // armazena conjunto de nós
var valorno;            // armazena o valor do nó gerado pela otimização do programa linear
                        //        associado
n ← vértice recebido por parâmetro;
P ← programa linear associado a n;
ncortes ← 0;
Faça enquanto é possível inserir cortes
        Se não é possível otimizar P
                ncortes ← 0; sai do loop;
        valorno ← valor do nó após otimização de P;
        Se o nó não é promissor          // se leva a uma solução inviável ou de valor
                ncortes ← 0; sai do loop; //        maior ou igual à solução já existente
        A ← árvore de Gomory-Hu;

        //essa técnica reduz a árvore de Gomory-Hu em duas partes U e V
        Para cada aresta e ∈ A
                //utilizando union-find com path compression
                Remove a aresta e∈ A e contrai todas as arestas;

        max_u ← máximo requisito de conectividade da parte U;
        max_v ← máximo requisito de conectividade da parte V;
        //se existem pontos em partes distintas que precisam ser conectados, temos que
        //        inserir uma desigualdade de corte
        Se max_u ≥ 1 e max_v ≥ 1
                ncortes ← ncortes + 1;
                Insere a desigualdade dependendo das restrições max_u e max_v;
Devolva (ncortes);
```

Algoritmo 5.2: gerador de planos de corte utilizado na aplicação de teste

No algoritmo 5.2, a árvore de Gomory-Hu foi gerada através da adaptação de um algoritmo para geração de árvores de Gomory-Hu disponível em [16].

No algoritmo, a estrutura da árvore é representada por ponteiros, que apontam para o nó pai, que são armazenados na própria estrutura do nó. O peso de uma aresta da árvore é guardada no nó filho. A raiz da árvore de cortes é o primeiro nó na lista de nós

do grafo. Esta implementação está descrita em [17],[18] e foi utilizada neste trabalho como caixa preta sendo que não nos preocupamos com os detalhes desta implementação.

A rotina de execução de heurísticas iniciais consiste em executar as heurísticas k-redução, arredondamento probabilístico e o algoritmo de Jain modificado.

Neste caso, estamos executando quatro heurísticas iniciais:

- Algoritmo de Jain.
- Arredondamento probabilístico.
- 1-redução, ou seja, heurística k-redução com k = 1.
- 2-redução, ou seja, heurística k-redução com k = 2.

Inicialmente aplicamos a heurística de Jain, então aplicamos a heurística de arredondamento probabilístico e, finalmente, as heurísticas de k-redução com uma e duas iterações respectivamente.

Após a execução de cada uma dessas heurísticas, caso seja encontrada uma solução melhor, executamos a heurística de melhoria 2-OPT para tentar melhorar ainda mais a solução encontrada.

Os algoritmos das heurísticas de Jain, arredondamento probabilístico e k-redução já foram apresentados pelos algoritmos 4.1, 4.3 e 4.2 respectivamente.

No caso da nossa aplicação de teste, o algoritmo de arredondamento probabilístico é executado repetidas vezes durante 10 minutos e a escolha da variável que será arredondada para 1 é feita da seguinte forma: consideramos todas as variáveis fracionárias e sorteamos uma delas com probabilidade proporcional ao valor da variável fracionária. Ou seja, uma variável com valor 0,75 tem mais chances de ser sorteada que uma variável de valor 0,25.

Todas as heurísticas iniciais são executadas com tempo máximo de 10 minutos, porém a aplicação de teste não interrompe o processamento de uma resolução de programa linear. Assim, se o tempo limite for atingido neste momento, a aplicação espera o término da resolução para interromper a heurística em questão.

Como heurística de melhoria, implementamos a heurística 2-OPT que está representada no algoritmo 5.3:

```
var H;                    //estrutura que armazena soluções
var V_H;                  //estrutura que armazena nós
var E_H;                  //estrutura que armazena conjunto de arestas
var bloqueador;           //estrutura que armazena arestas bloqueadas
var C;                    //estrutura que armazena circuitos
var e, e1, e2;            //estrutura que armazena arestas
H ← (V_H,E_H);            // a solução a ser otimizada recebida como parâmetro
bloqueador ← ϕ;
Enquanto houver circuitos C tal que C ⊆ E_H \ bloqueador
    Para todo par de arestas não adjacentes e1 e e2 de C faça
        Se existe f1 e f2 tal que C - e1 - e2 + f1 + f2 é circuito e
                peso (e1) + peso(e2) > peso (f1) + peso(f2) então
            C ← C - e1 – e2 + f1 + f2;
            Atualize H;                  //o conjunto de arestas bloqueador
            Seja e ∈ C;                  // é usado para que após a otimização
            bloqueador ← bloqueador + e; //de um circuito, o mesmo não seja
Devolva H;                               //considerado para próximas otimizações
```

Algoritmo 5.3: heurística 2-OPT implementada na aplicação de teste

Estipulamos em 2 horas o tempo máximo para execução do algoritmo de *branch & cut* principal e consideramos que todas as soluções passaram pela heurística 2-OPT.

Para fins deste estudo, em cada nó da árvore executamos uma heurística para tentar obter uma solução viável a partir das restrições desse nó. A heurística utilizada foi a heurística de arredondamento probabilístico para 20% dos nós.

Assim, após executar as heurísticas iniciais, a aplicação de teste fica em *looping* na tentativa de encontrar uma solução ótima para o programa linear original. O *looping* é executado enquanto houverem nós ativos ou enquanto o tempo limite de execução ainda não tiver sido atingido.

A limpeza do pool de desigualdades é feita quando este aumenta ou diminui de 20% em relação à última limpeza.

Utilizamos este método porque a limpeza do pool de desigualdades é uma operação cara e, durante este trabalho, não notamos uma melhora significativa de performance quando o pool é limpo.

O processo de limpeza consiste em percorrer todo o pool e marcar as restrições que não são mais utilizadas. Depois, a estrutura que armazena o pool é limpa e o programa linear é atualizado com as restrições ativas.

O processo descrito é repetido até que uma solução ótima seja encontrada ou até que o tempo limite de execução do programa expire.

Durante o processo, algumas informações foram gravadas em arquivos de forma que viabilizasse a captura dos resultados computacionais que são apresentados no próximo capítulo.

Uma das informações que é gravada é a taxa percentual que indica o quanto estamos próximos ou não da solução ótima. Essa taxa é representada por:

$$\text{taxa} = \frac{UB}{LB}$$

Onde:

UB representa a melhor solução que temos no momento (*upper bound*)

LB representa a menor solução ótima relaxada (*lower bound*) entre os nós ativos

A seguir, apresentamos os resultados obtidos pela aplicação de teste descrita acima.

Capítulo 6

6. Resultados Computacionais

Neste capítulo apresentamos os resultados obtidos pelas implementações construídas na aplicação de teste que desenvolvemos no decorrer deste trabalho.

Os resultados aqui apresentados foram obtidos pela execução da aplicação de teste em problemas hipotéticos, disponíveis publicamente.

Todos os testes foram realizados em um computador com 770MB de memória RAM e processador de1.2GHz.

As tabelas 6.1 e 6.2 descrevem as características dos problemas que foram submetidos à aplicação de teste, onde cada coluna representa respectivamente:

- **NP**: nome dado ao problema.
- **QTN**: quantidade total de nós da rede.
- **QNOp**: quantidade de nós opcionais.
- **QNOr**: quantidade de nós ordinários.
- **QNC**: quantidade de nós críticos.

Nas tabelas que seguem, os grafos densos foram obtidos nos repositórios TSPLIB e STEINLIB e os grafos esparsos foram gerados a partir dos grafos densos. Para geração dos grafos, utilizamos o seguinte critério: um vértice liga-se aos 10 vértices mais próximos considerando-se as arestas originais.

Grafos densos				
NP	**QTN**	**QNOp**	**QNOr**	**QNC**
att48fst_a	139	91	48	0
eil76fst_a	237	161	76	0
Bier127fst_a	258	131	127	0
eil101fst_a	330	229	101	0
kroA150fst_a	389	239	150	0
Usa48	48	10	24	14
Usa48_a	48	0	34	14
Usa48_b	48	0	24	24
Usa48_c	48	0	15	33
gr_eil51_a	51	0	36	15
gr_eil51_b	51	0	25	26
gr_eil51_c	51	0	15	36
st70_a	70	0	51	19
st70_c	70	0	21	49
pr76_a	76	0	56	20
pr76_b	76	0	34	42
pr76_c	76	0	21	55

Tabela 6.1: grafos densos que foram submetidos à aplicação de teste

Grafos esparsos				
NP	**QTN**	**QNOp**	**QNOr**	**QNC**
ulysses16_spp_a	16	7	6	3
ulysses16_spp_b	16	4	6	6
usa48_spp_a	48	18	21	9
usa48_spp_b	48	18	15	15
usa48_spp_c	48	18	11	19
eil51_spp_a	51	20	22	9
eil51_spp_b	51	20	15	16
eil51_spp_c	51	20	11	20
berlin52_spp_a	52	23	20	9
berlin52_spp_b	52	20	16	16
berlin52_spp_c	52	20	12	20
st70_spp_a	70	27	29	14
st70_spp_b	70	27	21	22
st70_spp_c	70	27	13	30
eil76_spp_a	76	30	30	16
eil76_spp_b	76	30	21	25
eil76_spp_c	76	40	13	23
pr76_spp_a	76	30	30	16
pr76_spp_b	76	30	21	25
pr76_spp_c	76	30	13	33

Tabela 6.2: grafos esparsos que foram submetidos à aplicação de teste

Como forma de exemplificar o processo, as figuras a seguir representam os grafos obtidos a partir de cada uma das heurísticas iniciais e o grafo obtido como melhor solução do problema usa48 obtida pelo algoritmo de *Branch & Cut*. Para isso, as figuras utilizam-se da seguinte nomenclatura:

- quadrados representam pontos críticos.
- círculos representam pontos ordinários.
- triângulos representam pontos opcionais.

A figura 6.1 representa a solução obtida pela heurística de Jain no problema usa48:

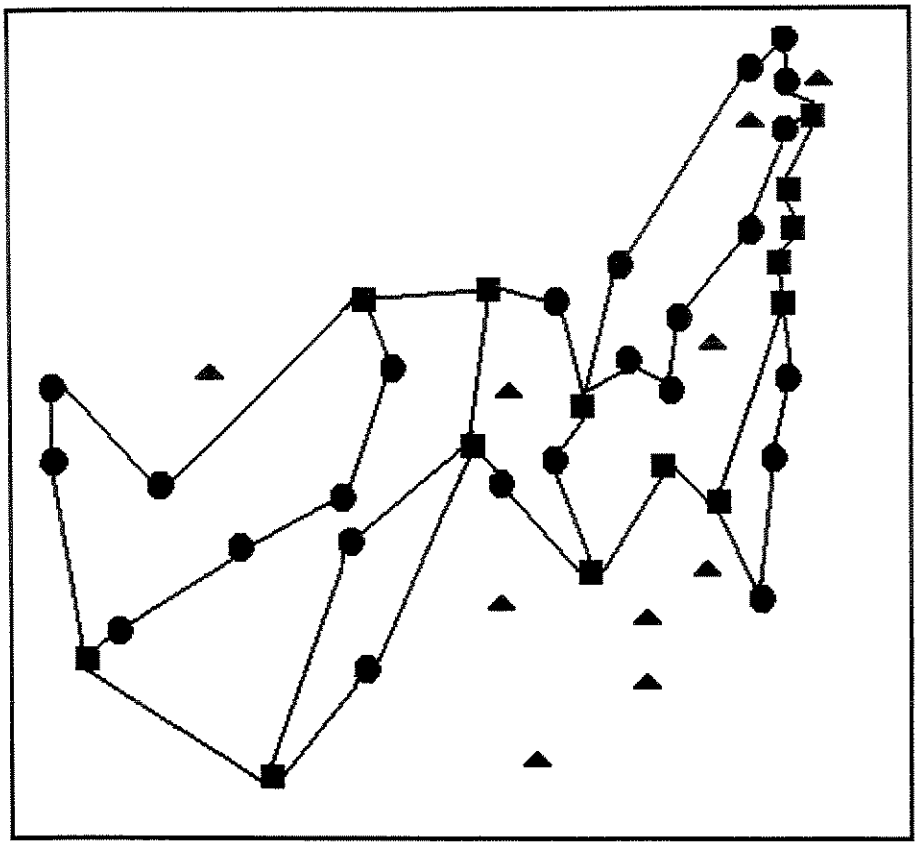


Figura 6.1: grafo que representa a solução obtida pela heurística de Jain no problema usa48

A figura 6.2 representa a solução obtida pela heurística de arredondamento probabilístico no problema usa48:

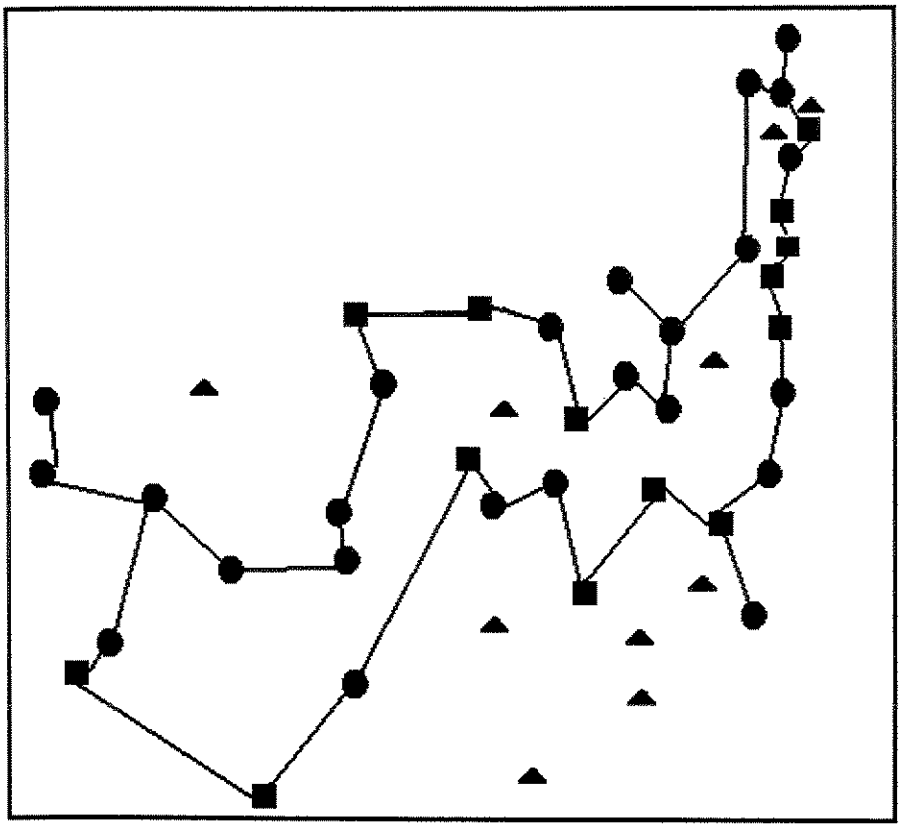


Figura 6.2: grafo que representa a solução obtida pela heurística de arredondamento probabilístico no problema usa48

A figura 6.3 representa a solução obtida pela heurística 1-redução no problema usa48:

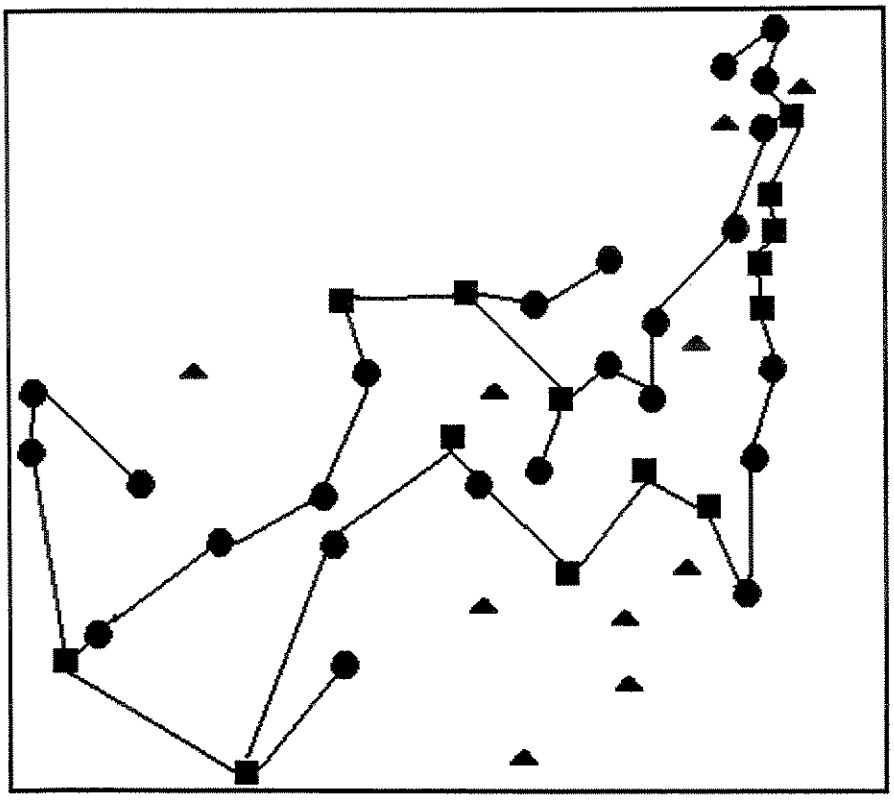


Figura 6.3: grafo que representa a solução obtida pela heurística 1-redução no problema usa48

A figura 6.4 representa a solução obtida pela heurística 2-redução no problema usa48:

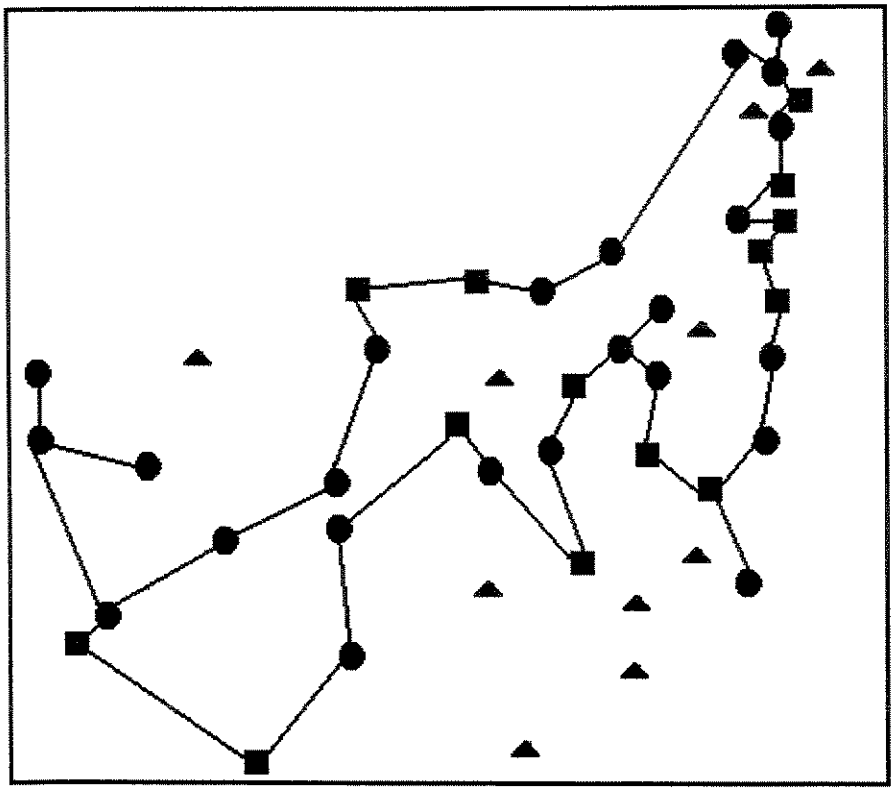


Figura 6.4: grafo que representa a solução obtida pela heurística 2-redução no problema usa48

A figura 6.5 representa a melhor solução ótima obtida ao término do processamento no problema usa48:

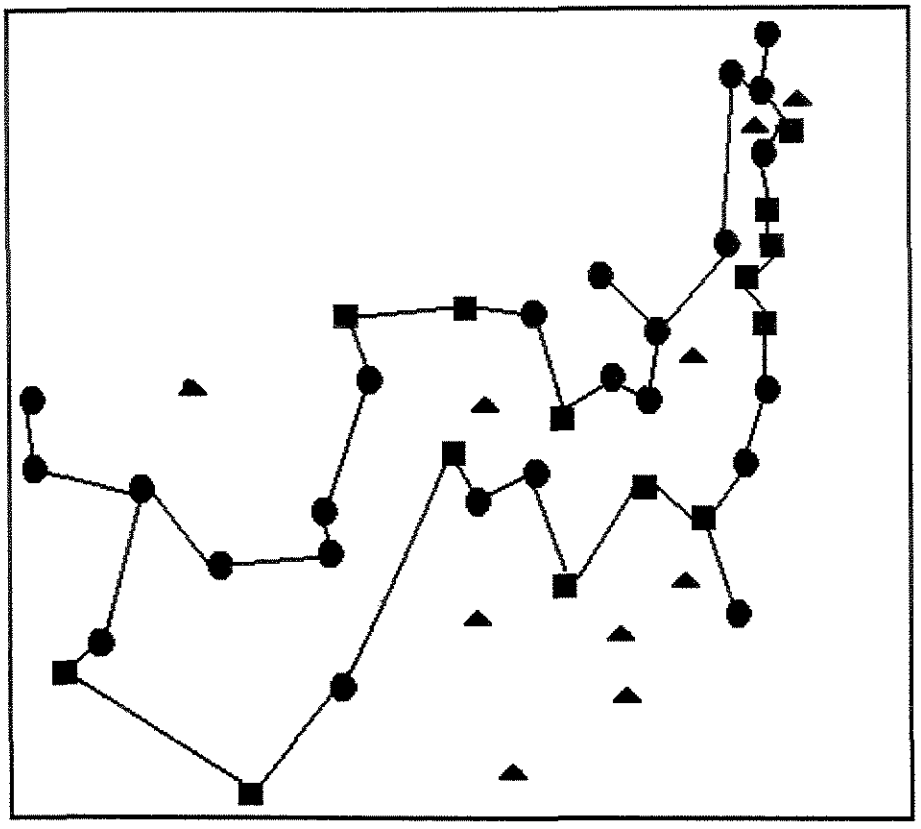


Figura 6.5: grafo que representa a melhor solução ótima obtida ao término do processamento no problema usa48

Em alguns testes realizados, o processo não foi concluído devido a consumo excessivo de memória. Nestes casos, os dados apresentados estão identificados com a marca $^{\intercal}$.

O símbolo * que aparece à esquerda de alguns dados significa que a solução ótima para o algoritmo em questão foi obtida em tempo inferior a 1 segundo.

Os dados apresentados em negrito representam as melhores soluções obtidas para o problema.

As tabelas 6.3 e 6.4 apresentam alguns dados sobre a performance obtida pela aplicação de teste em todos os problemas testados no que diz respeito a cada uma das heurísticas iniciais. Nessa tabela, cada coluna representa respectivamente:

- **NP**: nome dado ao problema.
- **J**: heurística de Jain.
- **TJ**: tempo, em segundos, obtido pela heurística de Jain.
- **TxJ**: taxa percentual que indica o quanto a solução obtida pela heurística de Jain está próxima da solução ótima.
- **AP**: heurística de arredondamento probabilístico.

- **TAP**: tempo, em segundos, obtido pela heurística de arredondamento probabilístico.
- **TxAP**: taxa percentual que indica o quanto a solução obtida pela heurística de arredondamento probabilístico está próxima da solução ótima.
- **R1**: heurística 1-redução.
- **TR1**: tempo, em segundos, obtido pela heurística 1-redução.
- **TxR1**: taxa percentual que indica o quanto a solução obtida pela heurística 1-redução está próxima da solução ótima.
- **R2**: heurística 2-redução.
- **TR2**: tempo, em segundos, obtido pela heurística 2-redução.
- **TxR2**: taxa percentual que indica o quanto a solução obtida pela heurística 2-redução está próxima da solução ótima.

Vale lembrar que as heurísticas de construção são executadas no início da aplicação de teste e, depois de cada uma delas, é executada a heurística de melhoria 2-OPT.

A taxa percentual de cada heurística é a razão entre o valor obtido pelo respectivo algoritmo e a menor solução ótima relaxada (*lower bound*) conhecida para o problema.

Grafos densos												
NP	**J**	**TJ**	**TxJ**	**AP**	**TAP**	**TxAP**	**R1**	**TR1**	**TxR1**	**R2**	**TR2**	**TxR2**
usa48	39886,03	*1	1,616	**27379,06**	616	1,109	30446,72	19	1,233	28100,47	610	1,138
att48fst_a	36121	*1	1,436	**31652**	602	1,259	34933	11	1,389	34933	8	1,389
eil76fst_a	**546**	1	1,560	558	614	1,594	577	621	1,649	577	602	1,649
bier127fst_a	115072	1	1,317	**108692**	712	1,244	114028	606	1,306	114028	601	1,306
eil101fst_a	723	1	1,730	692	1448	1,666	**683**	638	1,634	**683**	604	1,634
kroA150fst_a	29474	3	1,673	**27426**	80	1,556	28843	615	1,637	28843	604	1,637
usa48_a	48331,29	1	1,812	**33250,42**	622	1,246	34412,94	114	1,290	33514,57	614	1,256
usa48_b	41484,39	*1	1,429	**33347,78**	607	1,148	33687,15	604	1,160	33475,81	608	1,153
usa48_c	51193,44	*1	1,589	**33339,55**	603	1,035	33605,15	6	1,043	33343,49	605	1,035
gr_eil51_a	517,83	1	1,616	425,90	623	1,329	437,17	608	1,364	**424,55**	616	1,325
gr_eil51_b	559,28	*1	1,519	**431,37**	608	1,171	437,12	210	1,187	438,95	611	1,192
gr_eil51_c	567,17	*1	1,435	432,42	603	1,094	436,63	86	1,105	**429,83**	607	1,088
st70_a	895,46	1	1,741	**653,03**	673	1,270	707,25	618	1,375	670,72	621	1,304
st70_c	891,16	1	1,468	664,62	638	1,095	**655,18**	92	1,079	660,66	615	1,089
pr76_a	141059,08	2	1,746	**100354,46**	669	1,242	106137,52	625	1,314	100548,25	643	1,245
†pr76_b	136766,07	1	1,505	104342,50	621	1,148	**101750,42**	620	1,120	102694,57	623	1,130
pr76_c	146000,71	1	1,477	104640,88	626	1,059	105449,90	166	1,067	**103151,38**	611	1,044

Tabela 6.3: comparação dos resultados obtidos pelas heurísticas iniciais em grafos densos

Grafos esparsos												
NP	J	TJ	TxJ	AP	TAP	TxAP	R1	TR1	TxR1	R2	TR2	TxR2
berlin52_spp_a	7701,23	*1	1,577	**5864,31**	610	1,201	6048,75	67	1,239	5957,09	606	1,220
berlin52_spp_b	8902,37	*1	1,545	6192,02	601	1,075	6288,59	3	1,091	**6164,54**	603	1,070
berlin52_spp_c	7759,05	*1	1,316	**6082,17**	600	1,031	6159,73	7	1,045	**6082,17**	601	1,031
eil51_spp_a	405,76	*1	1,621	**298,39**	607	1,192	315,20	25	1,259	300,01	605	1,199
eil51_spp_b	431,43	*1	1,515	**309,15**	600	1,085	320,85	9	1,126	309,33	602	1,086
eil51_spp_c	430,03	*1	1,426	**306**	600	1,015	321,84	5	1,067	309,46	98	1,026
ulysses16_spp_a	103	*1	1,816	**56,73**	600	1	66,83	*1	1,178	57,17	17	1,008
ulysses16_spp_b	97,24	*1	1,391	**69,91**	600	1	74,83	*1	1,070	70,13	1	1,003
usa48_spp_a	38739,71	*1	1,607	**27711,90**	610	1,150	30591,10	21	1,269	28441,65	603	1,180
usa48_spp_b	39544,72	*1	1,538	27402,88	601	1,066	29674,22	6	1,154	**27310,97**	602	1,062
usa48_spp_c	40271,16	*1	1,503	**27368,03**	600	1,021	28002,49	4	1,045	**27368,03**	602	1,021
st70_spp_a	762,97	1	1,712	**521,72**	604	1,171	535,91	110	1,203	525,23	606	1,179
st70_spp_b	782,44	*1	1,611	525,94	601	1,083	545,65	40	1,124	**516,09**	604	1,063
st70_spp_c	667,42	*1	1,315	526,03	602	1,036	542,79	28	1,069	**520,88**	602	1,026
eil76_spp_a	452,53	*1	1,440	**396,87**	623	1,263	393,12	485	1,251	398,12	612	1,267
eil76_spp_b	596,68	1	1,643	405,72	613	1,117	417,87	51	1,151	**404,57**	607	1,114
eil76_spp_c	513,59	*1	1,323	407,53	608	1,050	417,46	17	1,075	**403,86**	605	1,040
pr76_spp.a	119010,95	*1	1,711	**78994,03**	621	1,136	81455,86	63	1,171	79860,12	614	1,148
pr76_spp_b	123972,07	*1	1,688	80768,40	610	1,100	83482,40	35	1,137	**80124,02**	607	1,091
pr76_spp.c	108561,76	*1	1,362	81529,32	600	1,023	83533,14	14	1,048	**81491,78**	602	1,022

Tabela 6.4: comparação dos resultados obtidos pelas heurísticas iniciais em grafos esparsos

Analisando os resultados apresentados nas tabelas 6.3 e 6.4, podemos notar que, em geral, o algoritmo de Jain apresenta os piores resultados em relação à solução ótima. Isto deve-se ao fato de existir um grande número de variáveis fracionárias de valor pelo menos ½ na solução. Neste caso, todas essas variáveis são arredondadas para 1, dobrando o custo associado a estas arestas.

Por outro lado, as heurísticas de arredondamento probabilístico e 2-redução apresentam os melhores resultados. Este resultado pode ser explicado pelos seguintes fatos:

- Na heurística de arredondamento probabilístico, os valores das variáveis associadas à solução fracionária são interpretados como probabilidades. Assim, o conceito básico é que quanto maior o valor fracionário das variáveis, maior a chance destas estarem na solução ótima.
- Na heurística de k-Redução procuramos obter um grafo reduzido contendo arestas candidatas com maior chance de estarem na solução ótima. Aqui, o conceito básico consiste no fato de que as arestas com valor não nulo da solução ótima fracionária possuem maiores chances de aparecerem na solução ótima inteira. Dessa forma, a cada passo, definimos um programa linear para as arestas não escolhidas e escolhemos as próximas arestas que aparecem na solução ótima fracionária deste programa linear.

As figuras 6.6 e 6.7, apresentadas a seguir, comparam graficamente os resultados obtidos entre as heurísticas iniciais em grafos densos e esparsos respectivamente:

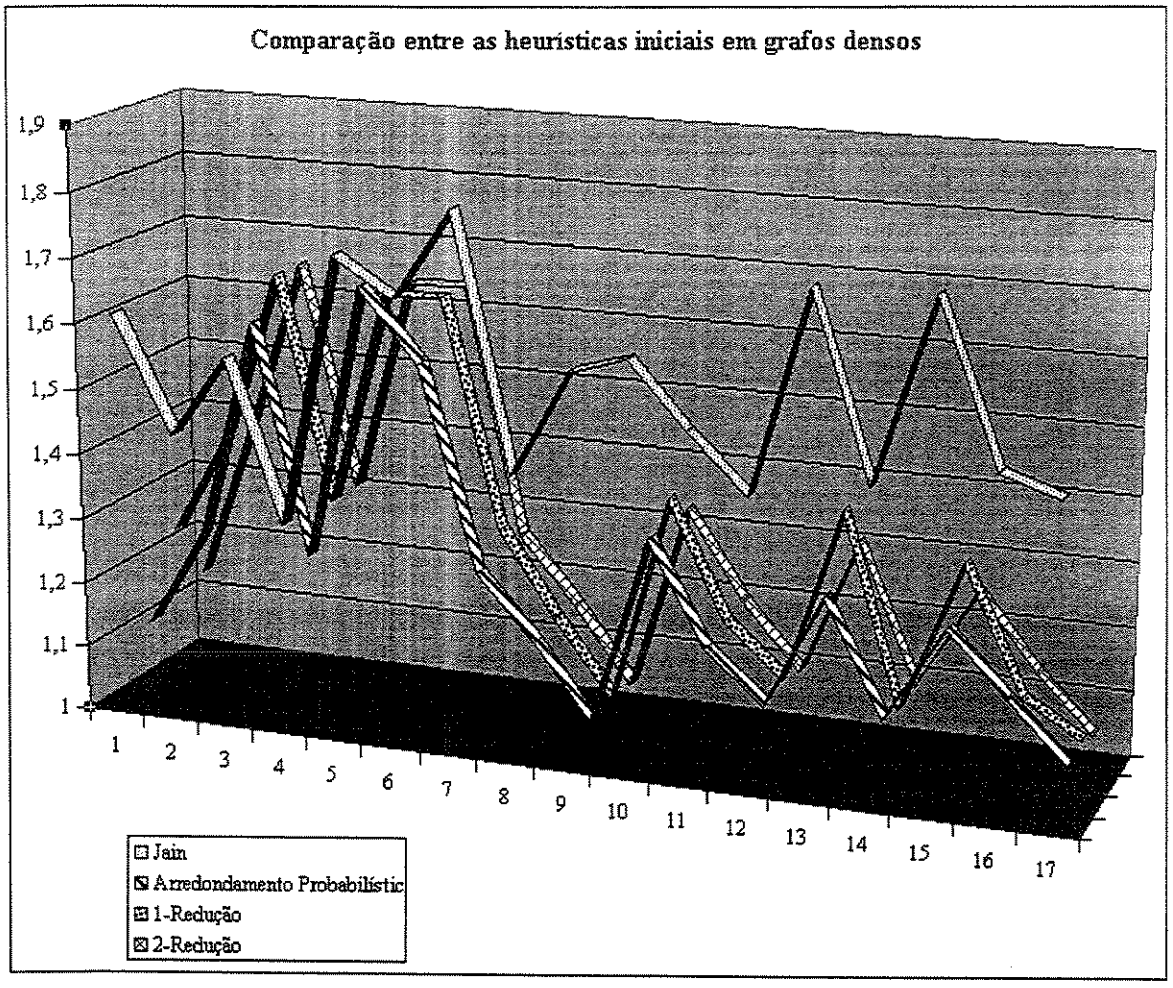


Figura 6.6: comparação entre as taxas de proximidade da solução ótima das heurísticas iniciais em grafos densos

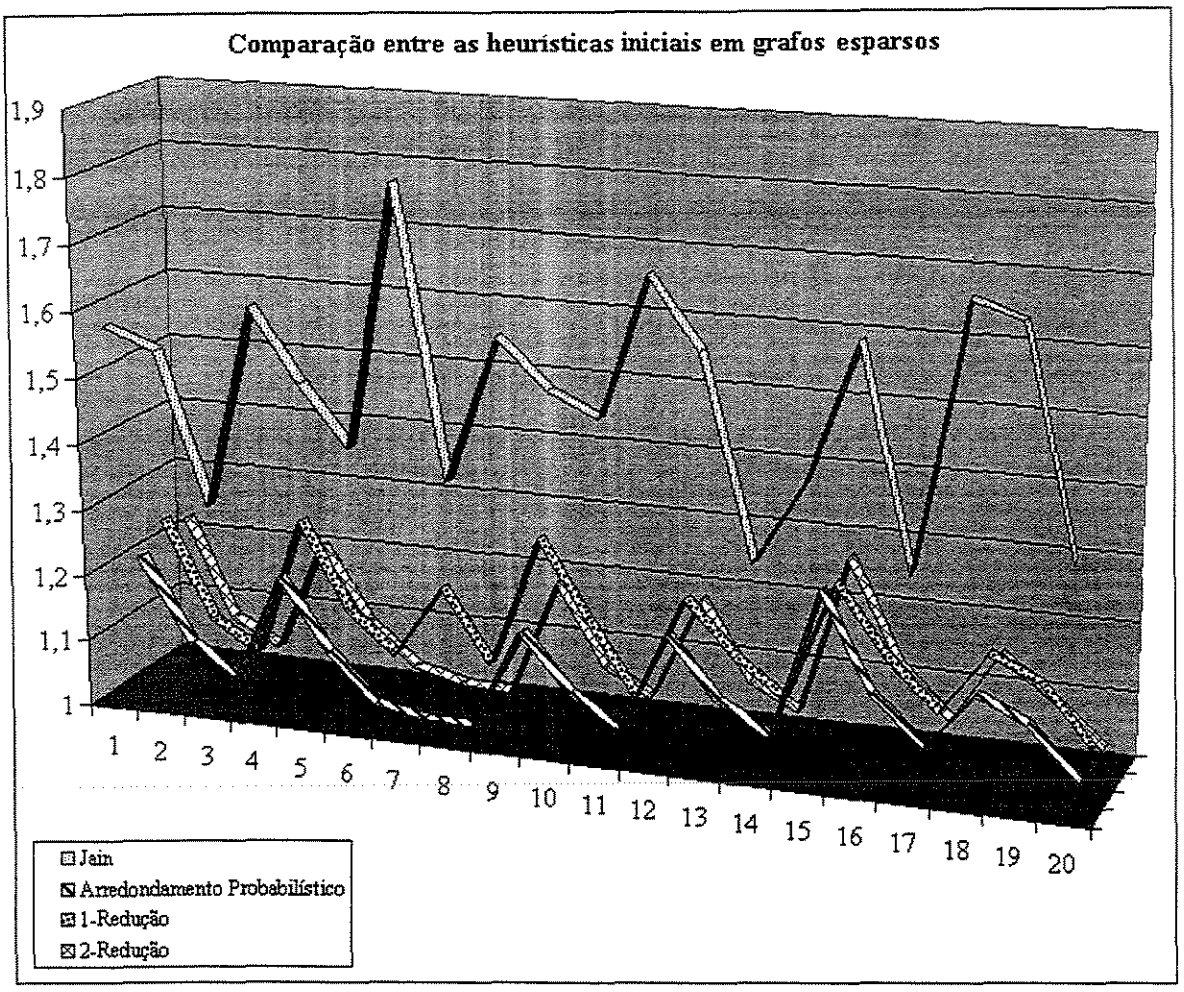


Figura 6.7: comparação entre as taxas de proximidade da solução ótima das heurísticas iniciais em grafos esparsos

Graficamente é fácil observar que a heurística de Jain, em geral, apresenta as piores taxas de proximidade da solução ótima. Já as heurísticas de arredondamento probabilístico e 2-Redução apresentam as melhores aproximações.

Após execução das heurísticas iniciais, a aplicação de teste utiliza a melhor solução obtida e submete ao algoritmo de *branch & cut*. As tabelas 6.5 e 6.6 apresentam o melhor resultado obtido em cada um dos problemas e o respectivo tempo de execução.

Nas tabelas 6.5 a 6.9, cada coluna representa respectivamente:

- **NP**: nome dado ao problema.
- **%0**: porcentagem de pontos opcionais no grafo original.
- **%1**: porcentagem de pontos ordinários no grafo original.
- **%2**: porcentagem de pontos críticos no grafo original.
- **MS**: melhor solução obtida até o limite máximo de execução de aproximadamente 2 horas.
- **MLB**: melhor lower bound conhecido para o problema.
- **NN**: número de nós na arvore de *branch & cut* no momento que a solução ótima foi obtida.
- **T**: tempo em que a solução ótima foi obtida.
- **Tx**: taxa percentual que indica o quanto a solução ótima obtida, até o limite máximo de 2 horas, está próxima da solução ótima.

Grafos densos								
NP	**%0**	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
usa48	21	50	29	27379,06	24683,96	4385	02:21:06	1,109
att48fst_a	65	35	0	31242	25150	2001	02:10:56	1,422
eil76fst_a	68	32	0	546	350	181	02:42:30	1,560
bier127fst_a	51	49	0	108692	87348	591	02:34:59	1,444
eil101fst_a	69	31	0	683	418	41	03:36:22	1,340
kroA150fst_a	61	39	0	27408	17623	191	02:43:07	1,552
usa48_a	0	71	29	33250,42	26676,42	4849	02:32:11	1,246
usa48_b	0	50	50	33347,78	29036,29	6735	02:30:45	1,149
usa48_c	0	31	69	33197,92	32218,36	9213	02:20:23	1,030
gr_eil51_a	0	71	29	424,55	320,48	3287	02:32:24	1,325
gr_eil51_b	0	49	51	431,37	368,31	4647	02:24:29	1,171
gr_eil51_c	0	29	71	429,83	395,13	5441	02:22:01	1,088
st70_a	0	73	27	653,03	514,21	2083	02:33:50	1,270
st70_c	0	30	70	655,18	606,94	3933	02:23:24	1,079
pr76_a	0	74	26	100354,45	80784,79	2189	02:35:31	1,242
[Y] pr76_b	0	45	55		90853,52			
pr76_c	0	28	72	103151,38	98836,88	4101	02:23:59	1,044

Tabela 6.5: resultados e tempos obtidos em grafos densos

Grafos esparsos								
NP	**%0**	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
berlin52_spp_a	38	44	17	5864,31	4883,21	8757	02:21:59	1,201
berlin52_spp_b	38	31	31	6136,64	5761,93	12459	02:20:16	1,065
berlin52_spp_c	38	23	38	6082,17	5897,09	22593	02:21:36	1,031
eil51_spp_a	39	43	18	298,39	250,32	10247	02:20:50	1,192
eil51_spp_b	39	29	31	306,35	284,87	16677	02:20:43	1,075
eil51_spp_c	39	22	39	306	301,55	19521	02:12:00	1,015
ulysses16_spp_a	44	38	19	56,73	56,73	18731	00:17:15	1
ulysses16_spp_b	25	38	38	69,91	69,91	231	00:10:06	1
usa48_spp_a	38	44	19	27711,90	24101,55	15925	02:20:48	1,150
usa48_spp_b	38	31	31	27229,98	25718,87	22291	02:21:59	1,059
usa48_spp_c	38	23	40	27368,03	26794,21	26265	02:21:19	1,021
st70_spp_a	39	41	20	519,81	445,55	8137	02:22:22	1,167
st70_spp_b	39	30	31	516,09	485,56	9639	02:21:05	1,063
st70_spp_c	39	19	43	519,4	507,7	11773	02:21:14	1,023
eil76_spp_a	39	39	21	393,12	314,22	5315	02:29:20	1,251
eil76_spp_b	39	28	33	404,57	363,13	6533	02:21:33	1,114
eil76_spp_c	56	17	30	403,86	388,19	8003	02:20:54	1,040
pr76_spp_a	39	39	21	78994,03	69544,95	5285	02:27:07	1,136
pr76_spp_b	39	28	33	78845,57	73441,26	8741	02:21:13	1,074
pr76_spp_c	39	17	43	81236,67	79736,77	16009	02:20:56	1,019

Tabela 6.6: resultados e tempos obtidos em grafos esparsos

Analisando os resultados apresentados nas tabelas 6.5 e 6.6, notamos que em grafos densos que não possuem nenhum ponto crítico, a taxa percentual de proximidade da solução ótima é significativamente ruim.

Por outro lado, em grafos densos ou esparsos contendo maior quantidade de pontos críticos, a taxa percentual de proximidade da solução ótima melhora significativamente.

Este comportamento, deve-se, provavelmente, ao fato de que em casos onde há maior incidência de prontos críticos, as possibilidades de ligação dos pontos da rede diminuem devido às restrições de conectividade associadas ao processo.

Como forma de analisar o comportamento dos algoritmos em grafos com características proporcionais entre vértices ordinários e críticos, executamos a aplicação de teste comparando problemas onde variamos a porcentagem de ocorrência desses pontos. As tabelas 6.7 a 6.9 apresentam os resultados obtidos:

Grafos densos							
NP	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
usa48_b	50	50	33347,78	29036,29	6735	02:30:45	1,149
gr_eil51_b	49	51	431,37	368,31	4647	02:24:29	1,171
[Y] pr76_b	45	55		90853,52			
Grafos esparsos							
NP	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
berlin52_spp_b	50	50	6136,64	5761,93	12459	02:20:16	1,065
eil51_spp_b	48	52	306,35	284,87	16677	02:20:43	1,075
ulysses16_spp_b	50	50	69,91	69,91	231	00:10:06	1
usa48_spp_b	50	50	27229,98	25718,87	22291	02:21:59	1,059
St70_spp_b	49	52	516,09	485,56	9639	02:21:05	1,063
eil76_spp_b	46	54	404,57	363,13	6533	02:21:33	1,114
pr76_spp_b	46	54	78845,57	73441,26	8741	02:21:13	1,074

Tabela 6.7: resultados obtidos em grafos com 50% de nós ordinários e 50% de nós críticos

Grafos densos							
NP	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
usa48_a	71	29	33250,42	26676,42	4849	02:23:11	1,246
gr_eil51_a	71	29	424,55	320,48	3287	02:32:34	1,325
st70_a	73	27	653,03	514,21	2083	02:33:50	1,270
pr76_a	74	26	100354,45	80784,79	2189	02:35:31	1,242
Grafos esparsos							
NP	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
berlin52_spp_a	69	31	5864,31	4883,21	8757	02:21:59	1,201
eil51_spp_a	71	29	298,39	250,32	10247	02:20:50	1,192
ulysses16_spp_a	67	33	56,73	56,73	18731	00:17:15	1
usa48_spp_a	70	30	27711,90	24101,55	15925	02:20:48	1,150
St70_spp_a	67	33	519,81	445,55	8137	02:22:22	1,167
eil76_spp_a	65	35	393,12	314,22	5315	02:29:20	1,251
pr76_spp_a	65	35	78994,03	69544,95	5285	02:27:07	1,136

Tabela 6.8: resultados obtidos em grafos com 70% de nós ordinários e 30% de nós críticos

Grafos densos							
NP	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
usa48	37	65	27379,06	24683,96	4385	02:21:06	1,109
usa48_c	31	69	33197,92	32218,36	9213	02:20:23	1,030
gr_eil51_c	29	71	429,83	395,13	5441	02:22:01	1,088
st70_c	30	70	655,18	606,94	3933	02:23:24	1,079
pr76_c	28	72	103151,38	98836,88	4101	02:23:59	1,044
Grafos esparsos							
NP	**%1**	**%2**	**MS**	**MLB**	**NN**	**T**	**Tx**
Berlin52_spp_c	38	63	6082,17	5897,09	22593	02:21:36	1,031
eil51_spp_c	35	65	306	301,55	19521	02:12:00	1,015
usa48_spp_c	37	63	27368,03	26794,21	26265	02:21:19	1,021
St70_spp_c	30	70	519,4	507,7	11773	02:21:14	1,023
eil76_spp_c	28	72	403,86	388,19	8003	02:20:54	1,040
pr76_spp_c	28	72	81236,67	79736,77	16009	02:20:56	1,019

Tabela 6.9: resultados obtidos em grafos com 30% de nós ordinários e 70% de nós críticos

Comparando os resultados apresentados nas tabelas 6.7, 6.8 e 6.9, notamos que a melhor aproximação da solução ótima foi obtida em grafos contendo maior número de nós críticos. Isto deve-se, provavelmente, ao fato de que as possibilidades de links de conexão diminuem conforme aumentamos o número de restrições.

A figura 6.8, apresentada a seguir, compara graficamente os resultados obtidos em grafos contendo quantidades proporcionais de pontos críticos e ordinários:

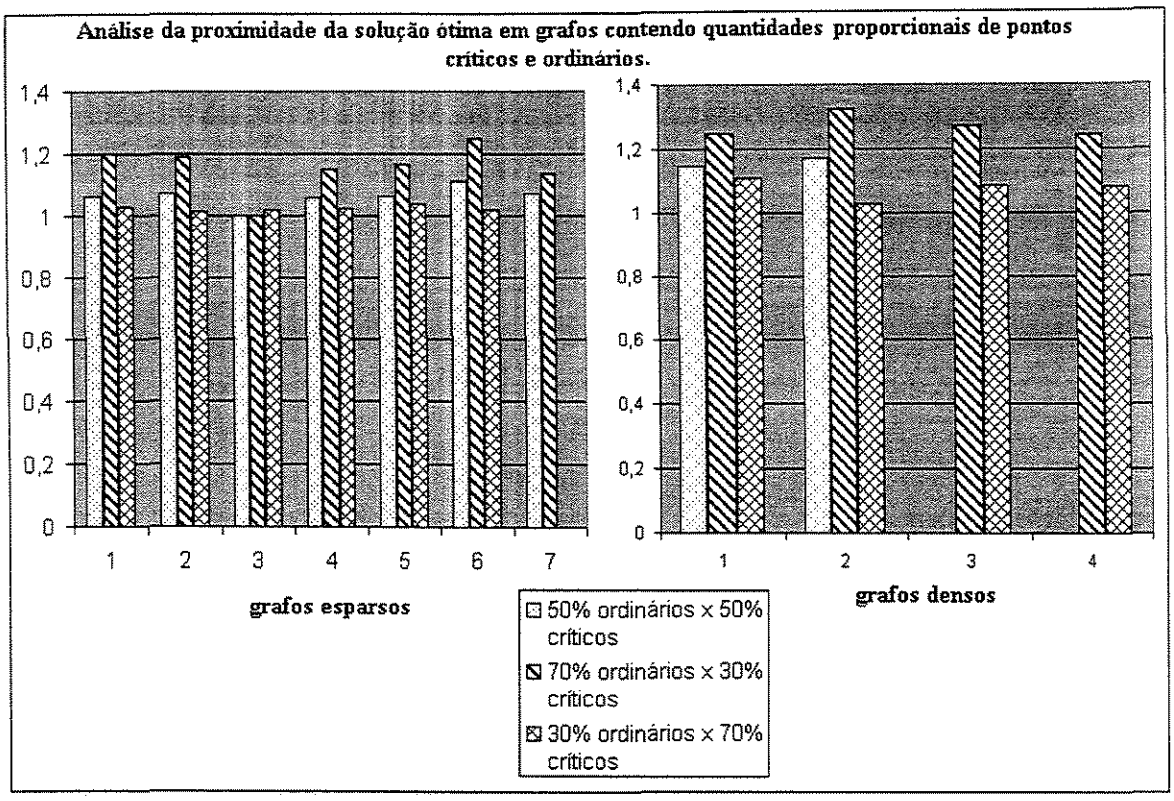


Figura 6.8: análise da proximidade da solução ótima em grafos contendo quantidades proporcionais de pontos críticos e ordinários.

Graficamente é fácil notar que conforme diminui a quantidade de pontos críticos, a taxa de proximidade da solução ótima piora. Já quando a quantidade de pontos críticos aumenta, a taxa de proximidade da solução ótima melhora.

As tabelas 6.10 e 6.11 apresentam uma comparação entre as soluções obtidas pelas heurísticas iniciais contra a melhor solução obtida pelo algoritmo de *Branch & Cut* em problemas envolvendo grafos densos e esparsos. Nessa tabela, cada coluna representa respectivamente:

- **NP**: nome dado ao problema.
- **SJ**: solução obtida pela heurística de Jain.
- **SAP**: solução obtida pela heurística de arredondamento probabilístico.
- **SR1**: solução obtida pela heurística 1-redução.
- **SR2**: solução obtida pela heurística 2-redução.
- **SM**: melhor solução encontrada pelo algoritmo de *Branch & Cut*.

Os dados em negrito representam os resultados que apresentam a solução com melhor aproximação da melhor solução obtida pelo algoritmo de *Branch & Cut* para o problema.

Grafos densos							
100% nós ordinários e 0% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SM**
att48fst_a	100	0	36121	**31652**	34933	34933	31242
eil76fst_a	100	0	**546**	558	577	577	546
bier127fst_a	100	0	115072	**108692**	114028	114028	108692
eil101fst_a	100	0	723	692	**683**	**683**	683
kroA150fst_a	100	0	29474	**27426**	28843	28843	27408
70% nós ordinários e 30% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SM**
usa48_a	71	29	48331,29	**33250,42**	34412,94	33514,57	33250,42
gr_eil51_a	71	29	517,83	425,9	437,17	**424,55**	424,55
st70_a	73	27	895,46	**653,03**	707,25	670,72	653,03
pr76_a	74	26	141059,08	**100354,46**	106137,52	100548,25	100354,45
50% nós ordinários e 50% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SM**
usa48_b	50	50	41484,39	**33347,78**	33687,15	33475,81	33347,78
gr_eil51_b	49	51	559,28	**431,37**	437,12	438,95	431,37
[T] pr76_b	45	55	136766,07	104342,5	101750,42	102694,57	
30% nós ordinários e 70% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SM**
usa48	37	65	39886,03	**27379,06**	30446,72	28100,47	27379,06
usa48_c	31	69	51193,44	**33339,55**	33605,15	33343,49	33197,92
gr_eil51_c	29	71	567,17	432,42	436,63	**429,83**	429,83
st70_c	30	70	891,16	664,62	**655,18**	660,66	655,18
pr76_c	28	72	146000,71	104640,88	105449,9	**103151,38**	103151,38

Tabela 6.10: comparação entre valores obtidos em cada heurística inicial com a melhor solução obtida em grafos densos

Grafos esparsos							
70% nós ordinários e 30% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SO**
berlin52_spp_a	69	31	7701,23	**5864,31**	6048,75	5957,09	5864,31
eil51_spp_a	35	65	405,76	**298,39**	315,2	300,01	298,39
ulysses16_spp_a	67	33	103	**56,73**	66,83	57,17	56,73
usa48_spp_a	70	30	38739,71	**27711,9**	30591,1	28441,65	27711,9
st70_spp.a	67	33	762,97	**521,72**	535,91	525,23	519,81
eil76_spp_a	65	35	452,53	396,87	**393,12**	398,12	393,12
pr76_spp_a	65	35	119010,95	**78994,03**	1455,86	9860,12	78994,03
50% nós ordinários e 50% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SO**
berlin52_spp_b	50	50	8902,37	6192,02	6288,59	**6164,54**	6136,64
eil51_spp_b	48	52	431,43	**309,15**	320,85	309,33	306,35
ulysses16_spp_b	50	50	97,24	**69,91**	74,83	70,13	69,91
usa48_spp_b	50	50	39544,72	27402,88	29674,22	**27310,97**	27229,98
st70_spp_b	49	51	782,44	525,94	545,65	**516,09**	516,09
eil76_spp_b	46	54	596,68	405,72	417,87	**404,57**	404,57
pr76_spp_b	46	54	123972,07	80768,4	83482,4	**80124,02**	78845,57
30% nós ordinários e 70% nós críticos							
NP	**%1**	**%2**	**SJ**	**SAP**	**SR1**	**SR2**	**SO**
berlin52_spp_c	38	63	7759,05	**6082,17**	6159,73	**6082,17**	6082,17
eil51_spp_c	35	65	430,03	**306**	321,84	309,46	306
usa48_spp_c	37	63	40271,16	**27368,03**	28002,49	**27368,03**	27368,03
st70_spp_c	30	70	667,42	526,03	542,79	**520,88**	519,4
eil76_spp_c	28	72	513,59	407,53	417,46	**403,86**	403,86
pr76_spp_c	28	72	108561,76	81529,32	83533,14	**81491,78**	81236,67

Tabela 6.11: comparação entre valores obtidos em cada heurística inicial com a melhor solução obtida em grafos esparsos

Analisando os dados apresentados nas tabelas 6.10 e 6.11, observamos que em grafos esparsos com 70% dos nós ordinários e 30% dos nós críticos, na maioria dos casos, as soluções que mais aproximam-se da solução ótima obtida para o problema, são dadas pela heurística de arredondamento probabilístico. Já nos demais casos, os melhores resultados foram obtidos pela heurística 2-redução, porém seguidas de perto pela heurística de arredondamento probabilístico. Por outro lado, a heurística de Jain, apresenta as piores aproximações.

Em grafos densos, em geral, os melhores resultados também são obtidos pelas heurísticas de arredondamento probabilístico e 2-redução.

As figuras 6.9 e 6.10 apresentam graficamente uma visão comparativa entre as soluções obtidas pelas heurísticas iniciais e a melhor solução obtida pelo algoritmo de *Branch & Cut* em grafos densos e esparsos respectivamente. Vale observar que para facilitar a visualização, os gráficos foram construídos com uma amostragem dos dados obtidos e não com todos os dados apresentados nas tabelas.

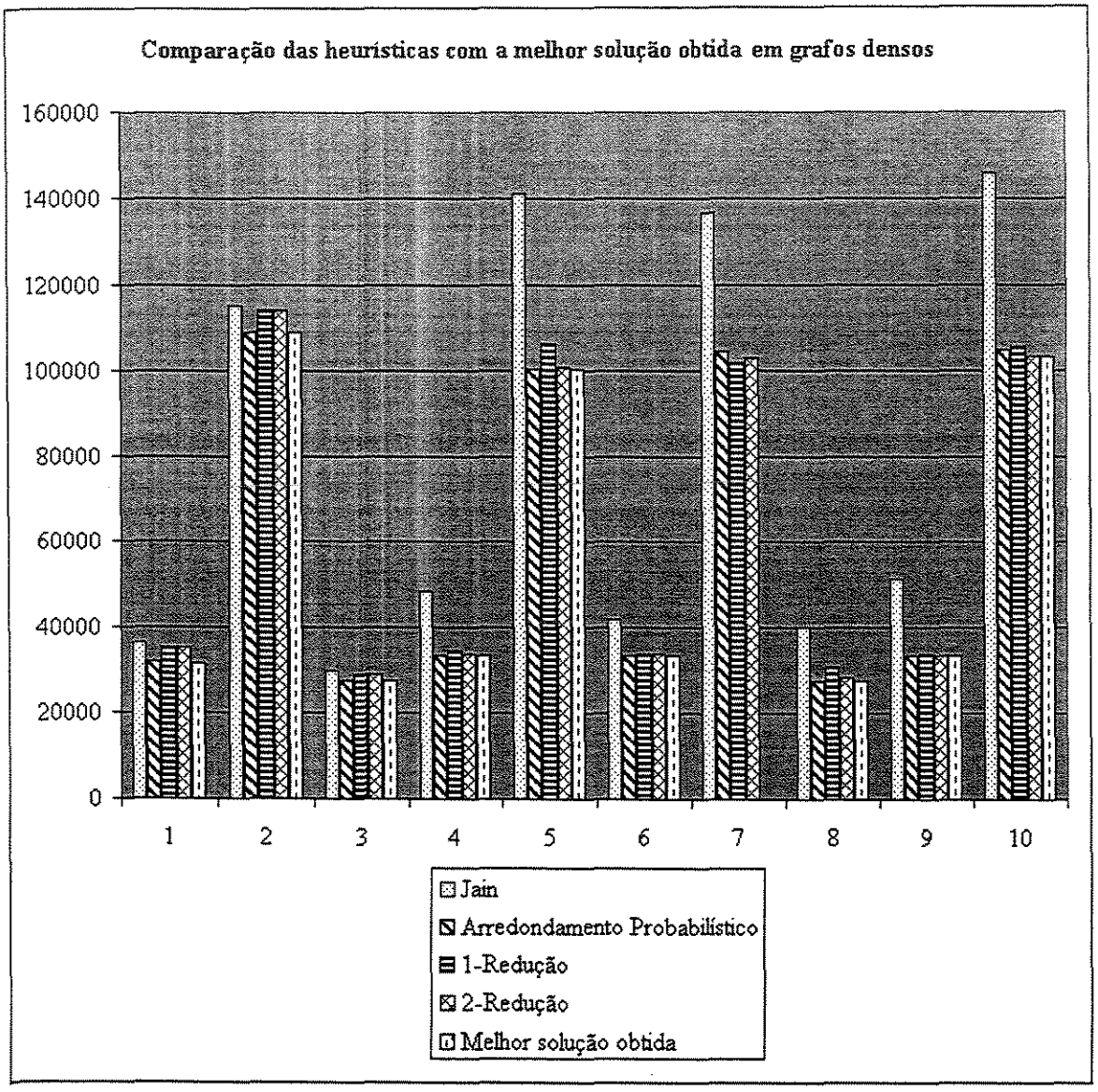


Figura 6.9: comparação entre as soluções heurísticas e a melhor solução obtida pelo algoritmo de *Branch & Cut* em grafos densos

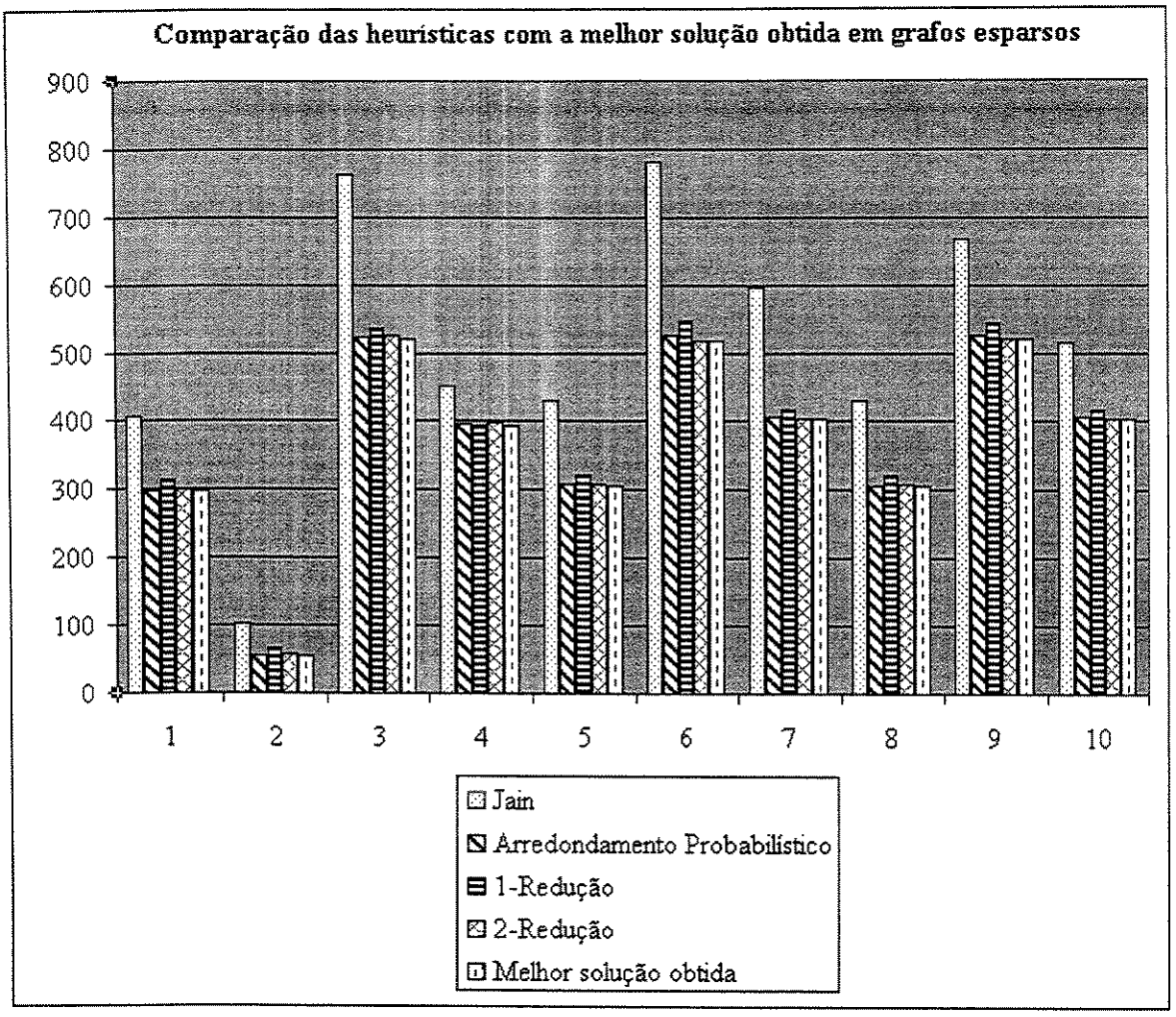


Figura 6.10: comparação entre as soluções heurísticas e a melhor solução obtida pelo algoritmo de *Branch & Cut* em grafos esparsos

Graficamente é fácil observar que a heurística de Jain, em geral, apresenta os piores resultados quando comparadas com a melhor solução obtida pelo algoritmo de *Branch & Cut*. Já as heurísticas de arredondamento probabilístico e 2-Redução, apresentam os melhores resultados.

A seguir, apresentamos as conclusões que foram obtidas como resultado dos estudos realizados durante este trabalho.

Capítulo 7

7. Conclusão

No decorrer deste trabalho, estudamos o problema de projeto de redes considerando condições de conectividade. Nosso enfoque foi direcionado para a topologia da rede considerando falhas que indisponibilizam conexões entre os pontos que ligam a rede.

Estudamos diversas técnicas e desenvolvemos uma aplicação de teste para testar alguns dos conceitos envolvidos e sugerir uma linha mestra que pode ser utilizada como base para pessoas interessadas no assunto. Nesta aplicação de teste, implementamos um algoritmo *Branch & Cut* exato, um algoritmo aproximado e heurísticas para o problema.

Dada a dificuldade em obter instâncias reais, os testes foram executados em instâncias geradas computacionalmente e em instâncias adaptadas de outros problemas disponíveis em repositórios públicos. Assim, estes testes não representam uma amostragem ideal para problemas reais.

Observando o desempenho do algoritmo exato, verificamos que o uso de cortes de conectividade ainda nos dá limitantes inferiores de baixa qualidade. Como forma de melhorar esse limitante inferior, é possível implementar os cortes de partição, porém não é claro que seu uso irá melhorar o limitante inferior, uma vez que eles são bastante relacionados com os cortes de conectividade.

Verificamos também que o algoritmo de aproximação, com melhor fator de aproximação teórico para este problema, nos leva a soluções muito distantes do ótimo. Por outro lado, este algoritmo nos levou a idéias simples e eficientes como o do arredondamento probabilístico, que obteve soluções muito próximas das melhores soluções obtidas pelo algoritmo *Branch & Cut*.

Muitos pesquisadores estão atuando na área para obter resultados melhores, seja a partir de heurísticas ou através de outras abordagens exatas. Diante dos estudos e conceitos apresentados aqui e dos testes realizados, concluímos que este problema ainda representa um tópico de pesquisa que precisa ser mais investigado. Neste ponto, acreditamos que este trabalho apresenta diretrizes importantes, que podem ser seguidas, para quem enfrenta problemas desse tipo.

Referências Bibliográficas

1. Mechthild Stoer. *Design of Survivable Networks*. Springer-Verlag Heidelberg, 1992.
2. Carlos E. Ferreira e Yoshiko Wakabayashi. *Combinatória Poliédrica e Planos-de-Corte Faciais*. 10ª. Escola de Computação. Editora da Unicamp, 1996.
3. Darci Prado. *Programação Linear*. Editora DG, 1999.
4. Matthew Saltzman. *The COIN-OR Open Solver Interface: Design Issues for a Portable Solver API*. Mathematical Sciences – Clemson University – INFORMS San Jose – November 17, 2002.
5. W. Grover. *Network Survivability: A Crucial Issue for the Information Society*. IEEE Canadian Review, summer 1997, pp. 16-21.
6. K. I. Aardal & S. P. M. van Hoesel. *Polyhedral Techniques in Combinatorial Optimization*. Statistica Neerlandica, 50, 3-26, 1996.
7. M. H. Carvalho, M.R. Cerioli, R. Dahab, P. Feofiloff, C. G. Fernandes, C. E. Ferreira, K. S. Guimarães, F. K. Miyazawa, J.C. Pina Jr., J. Soares, Y. Wakabayashi. *Uma Introdução Sucinta a Algoritmos de Aproximação*. 23o Colóquio Brasileiro de Matemática, 2001.
8. Flávio Keidi Miyazawa. *Notas de aula do Curso de Otimização*. Unicamp, 2002.
9. Clyde L. Monma, David F. Shallcross. *Methods for designing Communications Networks with certain two-connected survivability constraints*. Operations Research, Volume 37, No. 4, julho a agosto 1989.
10. Thomas L. Magnanti, S. Raghavan. *Technical Research Report – Strong Formulations for Network Design Problems with Connectivity Requirements*. Institute For System Research, abril 1999.
11. Kamal Jain. *Enhancing Techniques in LP Based Approximation Algoritms*. PhD thesis. Georgia Institute of Technology, Georgia, USA, agosto 2000.
12. M. X. Goemans, D. Bertsimas. *Survivable Networks, Linear Programming Relaxations and the Parsimonious Property*. Mathematical Programming, 60, 145--166, 1993.
13. M. R. Garey, D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, CA, 1979.
14. Kátia S. Guimarães. *Introdução à NP-completude*. Maio/2000.
15. Thomas H. Cormen, Charler E. Leiserson. Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms – second edition*. Editora MIT Press, 2001.
16. Site ZIB. *MATHPROG: A Collection of Codes for Solving Various Mathematical Programming Problems*.
17. D. Gusfield. *Very Simple Algorithms and Programs for All Pairs Network Flow Analysis*. Computer Science Division, University of California, Davis, 1987.
18. R.E Gomory and T.C. Hu. SIAM J. *Multi-Terminal Network Flows*. Applied Math, 9(1961), 55-570
19. M.X. Goemans, D.P. Williamson. *A General Approximation Technique for Constrained Forest Problems*. SIAM Journal on Computing, abril 1995 volume 24, número 2.

20. H. N. Gabow, M.X. Goemans, D. P. Williamson. *An Efficient Approximation Algorithm for the Survivable Network Design Problem.* Mathematical Programming, 82, 13--40, 1998.