Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Tallys Hoover Yunes
e aprovada pela Banca Examinadora.
Campinas, 21 de julho de 2000
COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Problemas de Escalonamento no Transporte Coletivo: Programação por Restrições e Outras Técnicas

Tallys Hoover Yunes

Dissertação de Mestrado

UNICAMP
BIBLIOTECA CENTR
SEÇÃO CIRCULAN

Instituto de Computação
Universidade Estadual de Campinas

Problemas de Escalonamento no Transporte Coletivo: Programação por Restrições e Outras Técnicas

Tallys Hoover Yunes[1]

Março de 2000

Banca Examinadora:

- Prof. Dr. Arnaldo Vieira Moura (Orientador)
- Prof. Dr. Oscar Porto
  Departamento de Engenharia Elétrica – PUC/RJ
- Prof. Dr. Flávio Keidi Miyazawa
  Instituto de Computação – UNICAMP
- Prof. Dr. Jacques Wainer (Suplente)
  Instituto de Computação – UNICAMP

200012984

[1]Com apoio financeiro da CAPES e da FAPESP (processo 98/05999-4).

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

UNIDADE BC
N.º CHAMADA: T/UNICAMP Y92p
V. Ex.
TOMBO BC/ 42127
PROC. 16-278100
C [ ] D [X]
PREÇO R$ 11,00
DATA 19/09/00
N.º CPD

CM-00145885-8

IB ID 276976

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Yunes, Tallys Hoover

~~Yu92p~~ Y92p Problemas de escalonamento no transporte coletivo: programação por restrições e outras técnicas / Tallys Hoover Yunes -- Campinas, [S.P. :s.n.], 2000.

Orientador : Arnaldo Vieira Moura

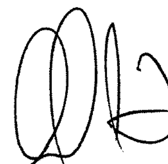
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Otimização combinatória. 2. Programação matemática. 3. Programação lógica I. Moura, Arnaldo Vieira. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

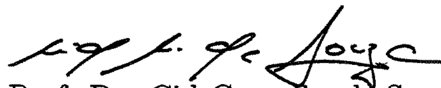
Problemas de Escalonamento no Transporte Coletivo: Programação por Restrições e Outras Técnicas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Tallys Hoover Yunes e aprovada pela Banca Examinadora.

Campinas, 26 de abril de 2000.



Prof. Dr. Arnaldo Vieira Moura
(Orientador)



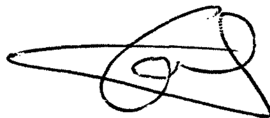
Prof. Dr. Cid Carvalho de Souza
(Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

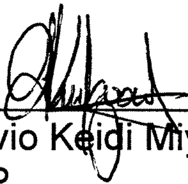
UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 26 de abril de 2000, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Oscar Porto
PUC - RJ



Prof. Dr. Flávio Keidi Miyazawa
IC-UNICAMP

Prof. Dr. Arnaldo Vieira Moura
IC-UNICAMP

UNICAMP
BIBLIOTECA CENT
SEÇÃO CIRCULAN

© Tallys Hoover Yunes, 2000.
Todos os direitos reservados.

UNICAMP
BIBLIOTECA CENTR
SEÇÃO CIRCULANT

Resumo

Este trabalho de mestrado procurou estudar e resolver um problema real de escalonamento de mão-de-obra oriundo da operação diária de uma empresa de ônibus urbanos da cidade de Belo Horizonte. Por questões de complexidade, este tipo de problema é normalmente dividido em dois subproblemas, a saber: *crew scheduling*, que trata a alocação diária de viagens a duplas de funcionários (motorista e cobrador), e *crew rostering*, que parte da solução do subproblema anterior e constrói uma escala de trabalho de mais longo prazo, e.g. um mês. Cada um desses subproblemas foi abordado utilizando-se técnicas de Programação Matemática e Programação por Restrições. Para o problema de *crew scheduling*, em particular, desenvolveu-se também um algoritmo híbrido de geração de colunas combinando as duas técnicas mencionadas e cujo desempenho foi significativamente melhor que o dos métodos isolados.

Em geral, os modelos matemáticos resultantes de problemas dessa natureza são de grande porte. No caso aqui tratado, a matriz de coeficientes do programa linear associado a algumas instâncias dos problemas chega a conter dezenas de milhões de colunas. Todos os algoritmos propostos para a solução do problema foram implementados e testados sobre dados reais obtidos junto à empresa em questão. A análise dos resultados computacionais mostra que foi possível obter soluções de excelente qualidade em um tempo de computação adequado para as necessidades da empresa. Em particular, para o subproblema de *scheduling*, foi possível comprovar que as soluções obtidas são ótimas.

Abstract

This dissertation aimed at studying and solving a real world crew management problem. The problem considered arises from the daily operation of an urban transit bus company that serves the metropolitan area of the city of Belo Horizonte, in Brazil. Due to its intrinsic complexity, the problem is usually divided in two distinct subproblems, namely: *crew scheduling*, that deals with the daily allocation of trips to crews, and *crew rostering*, which takes the solution of the first subproblem and extends the scheduling to a longer planning horizon, e.g. a month. We have tackled each one of these subproblems using Mathematical Programming (MP) and Constraint Logic Programming (CLP) approaches. Besides, we also developed a hybrid column generation algorithm for solving the crew scheduling problem, combining MP and CLP, which performed much better than the two previous approaches when taken in isolation.

Real world crew management problems typically give rise to large scale mathematical models. In our case, the coefficient matrix of the linear program associated with some instances of the problem contains tens of millions of columns. All the proposed algorithms have been implemented and tested over real world instances obtained from the aforementioned company. The analysis of our experiments indicates that it was possible to find high quality solutions within computational times that are suitable for the company's needs. In particular, we were able to find provably optimal solutions for the crew scheduling problem.

Agradecimentos

Agradeço...

Ao nosso Senhor, a quem recorri em tantas ocasiões e por inúmeras razões, agradeço pelas lições, pelas inspirações, pela força, pela saúde e, principalmente, por ter me ensinado que, em certos momentos, o coração é mais lúcido que a mente.

Aos meus pais, pelo amor, dedicação, carinho e, sobretudo, pelo apoio e incentivo que sempre me ofereceram, respeitando minhas decisões de vida mesmo quando resultaram em caminhos por demais tortuosos para todos nós.

À minha mãe, especialmente, pela *enorme* paciência e por pensar muito mais em mim do que em si própria.

Ao meu pai, especialmente, por estar sempre disposto e pronto a ajudar-me e por tantas vezes ter-me auxiliado sem que eu pedisse.

Aos meus orientadores, pela confiança que depositaram em mim e pelas mais de 200 horas de reuniões divertidas e proveitosas em que me apaixonei pela pesquisa. *Muito* mais do que orientadores, tive o privilégio de contar com mestres e amigos que jamais esquecerei.

À Renata, especialmente, por ter feito meus olhos brilharem.

Aos tantos amigos que tive o prazer de encontrar, pelo convívio, pelos momentos de descontração, pelos conselhos e pela troca de experiências.

À Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) pelo apoio financeiro que viabilizou o desenvolvimento deste trabalho.

Para ser grande, sê inteiro: nada
Teu exagera ou exclui.

Sê todo em cada coisa. Põe quanto és
No mínimo que fazes.

Assim em cada lago a lua toda
Brilha, porque alta vive.

Fernando Pessoa
Odes de Ricardo Reis, 1933

Conteúdo

Resumo xi

Abstract xiii

Agradecimentos xv

Lista de Tabelas xxiii

Lista de Figuras xxv

1 Introdução 1
1.1 O Problema de *Crew Scheduling* 2
1.2 O Problema de *Crew Rostering* 2
1.3 Organização do Texto 3

I Fundamentos Teóricos 5

2 Programação Matemática 7
2.1 Programação Linear 7
2.1.1 Introdução 7
2.1.2 Dualidade 8
2.1.3 O Algoritmo Simplex 8
2.2 Programação Linear Inteira 10
2.2.1 Introdução 10
2.2.2 *Branch-and-Bound* 11
2.3 Geração de Colunas 13
2.3.1 Conceitos Básicos 13
2.3.2 Um Exemplo: Corte Unidimensional 14
2.3.3 *Branch-and-Price* 15
2.4 Relaxação Lagrangeana 16

2.4.1 Conceitos Básicos 16
2.4.2 Um Exemplo Simplificado 17
2.4.3 O Método do Subgradiente 18
2.4.4 Contextos de Aplicação 19

3 Programação por Restrições **21**
3.1 Origens 21
3.2 Conceitos Básicos 23
3.3 Semântica 26
3.4 Implementações 28
3.5 Restrições Predefinidas 29

II O Trabalho Desenvolvido **31**

4 O Problema de *Crew Scheduling* **33**
1 Introduction 35
2 The Crew Scheduling Problem 37
2.1 Terminology 37
2.2 Input Data 38
2.3 Constraints 38
3 Mathematical Programming Approaches 39
3.1 Pure IP Approach 39
3.2 Column Generation with Dynamic Programming 40
3.3 A Heuristic Approach 42
4 Constraint Programming Approach 43
4.1 Improved Model 44
4.2 Refinements and Results 45
5 A Hybrid Approach 46
5.1 Implementation Issues 47
5.2 Computational Results 48
6 Conclusions and Future Work 49
References 50

5 O Algoritmo Híbrido para *Crew Scheduling* **59**
1 Introduction 60
2 The Crew Scheduling Problem 61
2.1 Terminology 61
2.2 Input Data 62

2.3 Constraints 62
3 Pure Approaches 63
4 A Hybrid Approach 65
4.1 The Column Generator 66
4.2 Computational Results 68
5 Conclusions and Future Work 70
References 71

6 O Problema de *Crew Rostering* 75
1 Introduction 76
2 The Crew Rostering Problem 77
2.1 Input Data 78
2.2 Problem Constraints 79
2.3 Objectives 79
3 The Input Data Sets 80
4 An Integer Programming Approach 80
4.1 The Model 81
4.2 Computational Results 82
5 A Constraint Logic Programming Approach 83
5.1 The Model 84
5.2 Computational Results 86
6 Proving Optimality 88
6.1 A Hybrid Model 88
6.2 Computational Results 89
7 Conclusions and Future Work 90
References 91

7 Detalhes de Implementação 97
7.1 Geradores de Código 97
7.2 A Heurística CFT 97
7.2.1 Descrição da Heurística 98
7.2.2 Diferenças na Implementação 99
7.2.3 Avaliação da Implementação 100
7.3 Geração de Colunas com Programação Dinâmica 100
7.3.1 Construção do GDA 100
7.3.2 Implementação do Algoritmo 103
7.3.3 Avaliação de Desempenho 105

III Considerações Finais **107**

8 Conclusões **109**

9 Trabalhos Futuros **111**

Bibliografia **113**

A Considerações sobre Programação por Restrições **119**
A.1 Introdução 119
A.2 Recomendações Gerais 120
A.2.1 Análise do Problema 120
A.2.2 O Modelo 121
A.2.3 Conselhos de Programação 122
A.2.4 Programação por Restrições × Programação Matemática 124
A.3 Aplicações 125
A.4 Conclusões 126

B Traduções Adotadas no Texto **127**

Lista de Tabelas

Capítulo 4

1 Number of feasible duties for OS 2222 and OS 3803 40
2 Computational results for OS 2222 (1 depot) 41
3 Heuristic over OS 3803 (2 depots) 43
4 Pure CP models, OS 2222 data set 45
5 Hybrid algorithm, OS 2222 data set (1 depot) 48
6 Hybrid algorithm, OS 3803 data set (2 depots) 50

Capítulo 5

1 (a) Sample from OS 3803 (b) Distribution of trips along the day 63
2 OS 2222 data set (1 depot) 69
3 OS 3803 data set (2 depots) 69

Capítulo 6

1 Description of the instances for the experiments 80
2 Computational experiments with the IP model 83
3 Computational experiments with the CLP model 87
4 Computational experiments with the hybrid model 90

Capítulo 7

7.1 Avaliação da qualidade da implementação da heurística CFT 101
7.2 Tempo de *pricing* com Geração de Colunas via GDA sobre a OS 2222 104

Lista de Figuras

Capítulo 2

2.1 Árvore de enumeração do algoritmo *Branch-and-Bound* 12

Capítulo 3

3.1 Estrutura básica de um programa em Programação por Restrições 23

Capítulo 4

1 Distribution of trips along the day 38
2 Simplified scheme of the hybrid column generation method 47
4.1 Comparação de desempenho entre as abordagens isoladas 53
4.2 Algoritmo híbrido *versus* *Branch-and-Bound* sobre a OS 2222 54
4.3 Algoritmo híbrido *versus* *Branch-and-Bound* sobre a OS 3803 55
4.4 Escalonamento final para a OS 2222 56
4.5 Carga de trabalho para um escalonamento ótimo da OS 2222 57

Capítulo 6

1 A column in the coefficient matrix of the set partitioning formulation 88
6.1 Comparação entre Programação Linear Inteira (PLI) e Programação por Restrições (PR) quanto à qualidade da solução 93
6.2 Comparação entre Programação Linear Inteira (PLI) e Programação por Restrições (PR) quanto ao desempenho 93
6.3 Comparação entre Programação por Restrições (PR) e o algoritmo híbrido quanto ao desempenho 94
6.4 Escalonamento mensal construído para a instância s20 95
6.5 Carga de trabalho para o escalonamento mensal da instância s20 96

Capítulo 7

7.1 Funcionamento da heurística CFT 98
7.2 Subrotina 3-PHASE() da heurística CFT 99
7.3 GDA com restrições de recursos nos arcos 102
7.4 Algoritmo de caminhos mais curtos no GDA 104

Capítulo 1

Introdução

Este trabalho de mestrado procurou estudar e resolver um problema real de escalonamento de mão-de-obra oriundo da operação diária de uma empresa de ônibus urbanos da cidade de Belo Horizonte. Por questões de complexidade, este tipo de problema é normalmente dividido em dois subproblemas, a saber: *crew scheduling*, que trata a alocação diária de viagens a duplas de funcionários (motorista e cobrador), e *crew rostering*, que parte da solução do subproblema anterior e constrói uma escala de trabalho de mais longo prazo, e.g. um mês. Cada um desses subproblemas foi abordado utilizando-se técnicas de Programação Matemática e de Programação por Restrições. Para o problema de *crew scheduling*, em particular, desenvolveu-se também um algoritmo híbrido de geração de colunas combinando as duas técnicas mencionadas e cujo desempenho foi significativamente melhor que o dos métodos isolados.

Em geral, os modelos matemáticos resultantes de problemas dessa natureza são de grande porte. No caso aqui tratado, a matriz de coeficientes do programa linear associado a algumas instâncias dos problemas chega a conter dezenas de milhões de colunas. Todos os algoritmos propostos para a solução do problema foram implementados e testados sobre dados reais obtidos junto à empresa em questão. A análise dos resultados computacionais mostra que foi possível obter soluções de excelente qualidade em um tempo de computação adequado para as necessidades da empresa. Em particular, para o subproblema de *scheduling*, foi possível comprovar que as soluções obtidas são ótimas.

Até o momento da impressão deste texto, este trabalho produziu dois relatórios técnicos e quatro publicações em congressos internacionais nas áreas de Pesquisa Operacional e Programação por Restrições, sendo duas delas apresentações convidadas. Um quinto artigo foi submetido para um outro congresso internacional da área de Programação por Restrições, mas a sua notificação de aceitação ainda não foi divulgada.

As próximas duas seções descrevem, respectivamente, os problemas de *crew scheduling* e *crew rostering* de forma mais completa.

1.1 O Problema de *Crew Scheduling*

Os dados de entrada para o problema de *crew scheduling* consistem no conjunto de viagens que devem ser executadas diariamente pela empresa de ônibus. A cada viagem está associado um horário de início, uma duração e *pontos de partida* e de *chegada*. Tais pontos recebem o nome de *pontos de controle*. É nesses pontos em que pode haver troca de motoristas. A saída esperada é uma atribuição de viagens a duplas de funcionários (motorista e cobrador) de modo que cada viagem seja alocada a exatamente a uma única dupla, ou seja, a atribuição induz uma partição no conjunto de viagens. Adicionalmente, essa atribuição de viagens deve procurar minimizar o número de duplas utilizadas, sem contudo violar restrições trabalhistas e regras operacionais da empresa. Por exemplo, há um limite máximo no número de horas que um funcionário pode trabalhar ao longo de um dia e todo funcionário tem direito a um tempo mínimo de descanso durante o seu período de trabalho.

Alguns termos específicos, normalmente utilizados no contexto desse problema, são definidos a seguir. Uma *jornada* é uma seqüência de viagens atribuídas à mesma dupla de funcionários. Caso a jornada respeite todas as restrições trabalhistas e as regras operacionais da empresa, ela recebe o nome de *jornada viável*. Dentro de uma jornada, dá-se o nome de *descanso* ao intervalo de tempo decorrido entre o término de uma viagem e o início da viagem seguinte. Quando uma jornada possui um descanso superior a duas horas, ela recebe o nome de *dupla-pegada*. Esse descanso prolongado, ao contrário dos demais descansos, não é contabilizado no tempo de trabalho da jornada. Um *escalonamento diário* é um conjunto de jornadas viáveis que formam uma partição do conjunto inicial de viagens.

1.2 O Problema de *Crew Rostering*

O problema de *crew rostering* recebe como entrada as jornadas viáveis que compõem o escalonamento diário construído pela solução do problema de *crew scheduling*. O objetivo, nesse caso, é construir escalonamentos de mais longo prazo, atribuindo jornadas viáveis, dia após dia, aos funcionários da empresa. Neste momento, outros tipos de restrições são relevantes, tais como, a necessidade de um período mínimo de descanso entre dois dias consecutivos de trabalho e o direito de pelo menos uma folga semanal para cada funcionário.

Este problema é influenciado por diversas características do horizonte de planejamento em questão, além de dados específicos para cada funcionário. Por exemplo, é preciso saber quais dias do mês são feriados e quais funcionários estão de férias ou impedidos de trabalhar. Ainda, a cada sete fins-de-semana, cada funcionário tem direito a uma folga semanal no domingo. Por outro lado, esta folga no domingo é mandatória para funcionários aos quais foi atribuída pelo menos uma dupla-pegada na semana anterior. Como a demanda por viagens varia de acordo com o dia da semana, o conjunto de jornadas a serem executadas também

pode diferir de um dia para outro. Para cada dia, é necessário que todas as suas jornadas sejam executadas. Uma *solução viável* para este problema é um escalonamento mensal que execute todas as jornadas para todos os dias do mês e respeite as restrições trabalhistas. Uma *solução ótima* é uma solução viável que emprega o menor número possível de duplas de funcionários.

Outros problemas de *crew rostering* normalmente encontrados na literatura diferem bastante do problema tratado nessa dissertação. Em muitos casos, o objetivo é simplesmente equilibrar a carga de trabalho entre os funcionários [2, 7, 25]. Em outras situações, procura-se um seqüência de todas as jornadas viáveis que obedeça às restrições trabalhistas e minimize o número de dias decorridos. Neste último caso, as escalas mensais de trabalho dos empregados são obtidas a partir de rotações sobre essa seqüência inicial [4, 6].

1.3 Organização do Texto

O texto dessa dissertação encontra-se dividido em três partes principais:

A Parte I apresenta resumidamente os fundamentos teóricos necessários para um entendimento adequado do trabalho descrito nesta dissertação. O Capítulo 2 expõe as idéias básicas a respeito de Programação Linear, Programação Linear Inteira, o método de Geração de Colunas e sua integração com o algoritmo de *Branch-and-Bound* (também conhecida por *Branch-and-Price*), e o método de Relaxação Lagrangeana. O Capítulo 3 contém uma introdução à metodologia de Programação por Restrições.

Na Parte II, estão incluídos três artigos resultantes deste trabalho de mestrado. Os artigos estão escritos em inglês. A cada um dos artigos estão anexadas uma nova introdução e uma nova conclusão estendidas, ambas em português. Os capítulos 4 e 5 discutem a solução do problema de *crew scheduling* e o Capítulo 6 trata o problema de *crew rostering*. Detalhes de implementação mais específicos, que foram omitidos dos artigos por restrições de espaço, estão descritos no Capítulo 7.

Por fim, a Parte III expõe as conclusões finais alcançadas com este trabalho e enumera algumas possíveis direções de pesquisa que poderiam ser tomadas no sentido de dar continuidade a este estudo.

O Apêndice A apresenta sugestões de modelagem e implementação em Programação por Restrições, compiladas a partir da experiência adquirida com este trabalho e de alguns artigos escritos por pesquisadores importantes da área.

Ao se traduzirem termos técnicos do inglês para o português, é possível que se criem dificuldades na compreensão do texto. Isto porque nem sempre há um consenso a respeito da melhor tradução para determinados termos ou expressões. Em algumas situações, ainda, não há como efetuar a tradução pois certas palavras da língua inglesa não possuem uma palavra correspondente em português. Desse modo, adotou-se a seguinte estratégia neste

texto. Sempre que um novo termo for introduzido, i.e. na sua primeira ocorrência, ele será escrito com uma fonte diferente da fonte principal, que é a fonte utilizada nesta frase. Se o termo estiver escrito em inglês ou constituir uma tradução de um termo em inglês relativamente bem conhecida, será utilizada a fonte *enfatizada*. No caso, de traduções não comumente adotadas, será utilizada a fonte *slanted* e o termo original, em inglês, aparecerá no Apêndice B.

Parte I

Fundamentos Teóricos

Capítulo 2

Programação Matemática

Este capítulo apresenta algumas metodologias conhecidas no campo da Programação Matemática. São fornecidos aqui os conceitos básicos necessários a uma melhor compreensão dos capítulos que se seguem.

Assume-se que o leitor possui conhecimentos elementares a respeito de Álgebra Linear. Neste sentido, recomenda-se a leitura da Seção 1.5 de [1].

2.1 Programação Linear

2.1.1 Introdução

O problema geral de Programação Linear é dado pelo *programa linear* abaixo:

$$z_{PL} = \min\{cx : Ax \leq b, x \in R^n_+\},$$

onde c é um vetor $1 \times n$ de *custos*, A é a *matriz $m \times n$ de coeficientes* e b é um vetor $m \times 1$. Todos os componentes de c, A e b são números racionais e R^n_+ denota todos os vetores $n \times 1$ cujas componentes assumem valores reais não negativos. Deseja-se, então, atribuir valores às componentes de x (*variáveis*) de modo a satisfazer às desigualdades $Ax \leq b$ e minimizar o valor da *função objetivo* cx. Dá-se o nome de *formulação* de um problema à sua expressão em termos de um conjunto de variáveis, restrições e uma função objetivo. Quando o conjunto $\{Ax \leq b, x \in R^n_+\}$ é vazio, o problema não tem solução e é dito *inviável*. Vale ressaltar, ainda, que z_{PL} não é necessariamente finito.

Todo programa linear pode ser escrito no formato acima, bastando para isso utilizarem-se algumas manipulações algébricas. Por exemplo, uma função objetivo de maximização $\max cx$ pode ser escrita na forma de minimização como $-\min -cx$. Da mesma forma, multiplicando-se por -1 ambos os lados de inequações do tipo $\geq$ obtêm-se inequações

do tipo $\leq$. Restrições de igualdade, por sua vez, podem ser reescritas na forma de duas restrições de desigualdade: uma do tipo $\leq$ e outra do tipo $\geq$.

2.1.2 Dualidade

É possível estabelecer relações entre certos pares de programas lineares cujas soluções compartilham propriedades importantes entre si. Utilizam-se as denominações *primal* e *dual* para referenciar tais programas. Seja (P) o seguinte programa linear primal:

$$z_{PL} = \max\{cx : Ax \leq b, x \in R^n_+\}\,. \tag{P}$$

O dual de (P) é dado por:

$$w_{PL} = \min\{ub : uA \geq c, u \in R^m_+\}\,, \tag{D}$$

onde u é o vetor de *variáveis duais* associadas a cada uma das restrições de (P).

Proposição 1 (Dualidade Fraca) [31] *Se x^* é uma solução viável para o problema primal (primal viável) e u^* é uma solução viável para o problema dual (dual viável), então*

$$cx^* \leq z_{PL} \leq w_{PL} \leq u^*b\,.$$

□

Teorema 1 (Dualidade Forte) [31] *Se o valor de z_{PL} ou de w_{PL} é finito, então (P) e (D) possuem valores ótimos finitos e $z_{PL} = w_{PL}$.* □

2.1.3 O Algoritmo Simplex

Seja (P) um programa linear, conforme apresentado a seguir:

$$z_{PL} = \min\{cx : Ax = b, x \in R^n_+\}\,, \tag{P}$$

Sejam a_j, $j = \{1, \ldots, n\}$, as colunas da matriz A, cujo posto é igual a m. Existe, portanto, uma submatriz quadrada $m \times m$ de A, denotada por B, cujo determinante é diferente de zero. Seja N a submatriz formada pelas colunas de A que não estão em B. Rearranjando as colunas de A e de x de modo que $A = (B, N)$ e $x = (x_B, x_N)$, pode-se escrever $Ax = b$ como

$$Bx_B + Nx_N = b\,,$$

cuja solução é dada por

$$x_B = B^{-1}b - B^{-1}Nx_N \quad \text{e} \quad x_N = 0\,. \tag{2.1}$$

A matriz B é chamada de *base* e a solução do sistema, dada por (2.1), recebe o nome de *solução básica*. As variáveis em x_B são *variáveis básicas* e as variáveis em x_N são denominadas *variáveis não básicas*. Se $B^{-1}b \geq 0$, diz-se que (x_B, x_N) é uma solução básica primal viável e B é uma base primal viável.

Teorema 2 [31] *Se z_{PL} é finito, então existe uma solução básica primal viável onde a função objetivo atinge o valor ótimo.* □

Com base nesse Teorema, a idéia do algoritmo Simplex é, portanto, caminhar de solução básica em solução básica procurando sempre melhorar o valor da função objetivo até que a solução ótima seja atingida. Precisa-se agora de um método para se passar de uma solução básica para outra.

Reescrevendo-se as componentes de c na forma (c_B, c_N), pode-se representar a função objetivo $z_{PL} = cx$ como $z_{PL} = c_B x_B + c_N x_N$. Substituindo-se o valor de x_B dado por (2.1) na equação anterior, tem-se

$$z_{PL} = c_B B^{-1} b - c_B B^{-1} N x_N + c_N x_N$$

o que equivale a

$$z_{PL} = z_0 + \sum_{j \in N} \overline{c}_j x_j \,,$$

onde $z_0 = c_B B^{-1} b$ e $\overline{c}_j = c_j - c_B B^{-1} a_j$. O valor $\overline{c}_j$ recebe o nome de *custo reduzido* da variável x_j. Este valor representa o acréscimo no valor da função objetivo para cada unidade de acréscimo no valor da variável x_j. Em se tratando de um problema de minimização, se $\overline{c}_j \geq 0$ para todo $j \in N$, significa que não há variável não básica que, ao entrar na base, contribua para melhorar o valor da função objetivo. Nesse caso, tem-se em mãos uma solução ótima para o problema. Caso $\overline{c}_j < 0$ para algum $j \in N$, pode-se construir uma nova base B' cuja solução básica associada é mais vantajosa em termos do valor da função objetivo.

Dado que existe um $\overline{c}_j < 0$, x_j pode entrar na base. Segundo o valor de x_B dado por (2.1), quando uma variável não básica sai do valor 0 para um valor positivo, os valores das variáveis básicas diminuem. Devido à restrição de não negatividade sobre as variáveis x, não é possível aumentar indefinidamente o valor de x_j pois, a partir de um certo ponto, alguma variável básica passaria a assumir um valor negativo. Portanto, a variável básica x_i que deve sair da base é aquela que mais contribui para limitar o crescimento do valor de x_j. Em termos algébricos, seja $\overline{b} = B^{-1} b$ e $\overline{a}_j = B^{-1} a_j$. Supondo que a variável não básica x_j entre na base, de (2.1) tem-se

$$x_B = \overline{b} - \overline{a}_j x_j \,, \quad x_B \geq 0, x_j \geq 0 \,. \tag{2.2}$$

Caso $\overline{a}_j \leq 0$, deduz-se por (2.2) que é possível aumentar indefinidamente o valor de x_j sem violar a restrição de não negatividade. Nesse caso, o valor ótimo da função objetivo é igual a $-\infty$. Caso algum componente $\overline{a}_{rj}$ de $\overline{a}_j$ seja positivo, seja a equação abaixo, onde $\overline{b}_i$ é o i-ésimo componente de $\overline{b}$:

$$\frac{\overline{b}_s}{\overline{a}_{sj}} = \min_{\overline{a}_{rj}>0} \left\{ \frac{\overline{b}_r}{\overline{a}_{rj}} \right\} .$$

O índice s indica a variável básica x_s que mais limita o crescimento de x_j e, portanto, x_s é a variável que deve sair da base.

2.2 Programação Linear Inteira

2.2.1 Introdução

Há momentos em que um programa linear, conforme apresentado na Seção 2.1, não é adequado para se resolver o problema em questão. Por exemplo, é possível que as variáveis x representem o número de trabalhadores a serem contratados para determinadas instalações de uma empresa. Nesse caso, não faz sentido dizer que 4.37 trabalhadores devem ser admitidos. Precisa-se de um número *inteiro* como resposta. Portanto, um modelo de Programação Linear para o problema não é apropriado pois não garante a integralidade da solução e arredondando-se os valores fracionários geralmente não se obtêm bons resultados.

Em casos como esse, tem-se em mãos um problema de *Programação Linear Inteira*, cuja formulação é dada por

$$z_{PI} = \min\{cx : Ax \leq b, x \in Z^n_+\}, \tag{PI}$$

onde Z^n_+ é o conjunto dos vetores de n componentes em que cada uma delas assume valores inteiros não negativos. Dá-se o nome de *relaxação linear* de (PI) ao problema obtido a partir de (PI) substituindo-se a restrição $x \in Z^n_+$ por $x \in R^n_+$. Note que o valor da solução da relaxação linear de (PI) é um limite inferior para o valor da solução de (PI). Isto porque $Z^n_+ \subset R^n_+$.

Diferentemente de Programação Linear, não há algoritmos polinomiais conhecidos para se resolver um problema genérico de Programação Linear Inteira [33]. É preciso, portanto, recorrer a algoritmos que, apesar de possuírem complexidade exponencial, percorrem o espaço de soluções de forma criteriosa. Um desses algoritmos é descrito a seguir, de acordo com [33].

2.2.2 *Branch-and-Bound*

A idéia básica de um algoritmo de *Branch-and-Bound* é enumerar, *implicitamente*, todas as soluções viáveis para o problema de Programação Linear Inteira, até que a solução ótima seja encontrada. Esse mecanismo evolui através de partições sucessivas do espaço de soluções. Seja o problema de Programação Linear Inteira abaixo

$$\begin{aligned} \min \quad & c(x) = cx \\ \text{sujeito a} \quad & Ax \leq b, \\ & x \geq 0 \text{ e inteiro}. \end{aligned} \tag{PI}$$

Resolvendo-se a relaxação linear de (PI), obtém-se uma solução x^0 que, caso seja inteira, corresponde a uma solução ótima de (PI). Todavia, em geral, x^0 não é inteira e seu custo $c(x^0)$ é apenas um limite inferior para o valor da solução ótima de (PI). O próximo passo do algoritmo consiste em dividir (PI) em dois subproblemas, (PI_1) e (PI_2), com o auxílio de duas restrições mutuamente exclusivas. Seja x_i^0 a i-ésima componente de x^0, e cujo valor não é inteiro. Tem-se, portanto

$$\begin{aligned} \min \quad & c(x) = cx \\ \text{s. a.} \quad & Ax \leq b, \\ & x \geq 0 \text{ e inteiro} \\ & x_i \leq \lfloor x_i^0 \rfloor , \end{aligned} \tag{PI$_1$}$$

e

$$\begin{aligned} \min \quad & c(x) = cx \\ \text{s. a.} \quad & Ax \leq b, \\ & x \geq 0 \text{ e inteiro} \\ & x_i \geq \lfloor x_i^0 \rfloor + 1 . \end{aligned} \tag{PI$_2$}$$

Note que uma solução ótima de (PI), denotada por x^*, tem de ser igual a uma solução ótima de um dos dois subproblemas, pois, necessariamente, ou $x_i^* \leq \lfloor x_i^0 \rfloor$ ou $x_i^* \geq \lfloor x_i^0 \rfloor + 1$. Nesse ponto, escolhe-se um dos subproblemas, e.g. PI_1, e resolve-se a sua relaxação linear. Tem-se agora uma outra solução, x^1, que pode novamente não ser inteira. De maneira semelhante, divide-se PI_1 em outros dois subproblemas e o processo todo se repete, criando a chamada *árvore de enumeração*, conforme apresentado na Figura 2.1. O conjunto de todos os subproblemas que ainda não foram divididos corresponde a uma partição das soluções viáveis do problema original. Num primeiro momento, é possível que, em um determinado nó da árvore, esse processo seja interrompido por uma de duas razões. Em primeiro lugar,

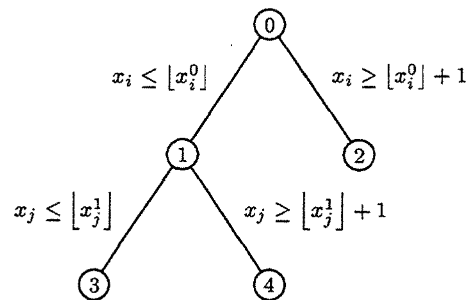


Figura 2.1: Árvore de enumeração do algoritmo *Branch-and-Bound*

a solução da relaxação linear do subproblema considerado pode ser inteira. Em segundo lugar, o programa linear correspondente àquele nó da árvore pode ser inviável.

Até este ponto, o algoritmo prosseguiu por meio de ramificações na árvore de enumeração. Essa operação recebe o nome de *branching*. Caso as operações de *branching* se repitam até que não haja mais como particionar um nó em subproblemas, aquela folha da árvore que eventualmente possua uma solução inteira viável com o menor custo corresponderá a uma solução ótima do problema original. Contudo, existe ainda uma outra ação a ser tomada que contribui para aumentar a eficiência do algoritmo como um todo. Trata-se da análise dos limitantes, ou *bounds*.

Suponha que, em algum ponto da execução do algoritmo de *Branch-and-Bound*, a *melhor* solução inteira conhecida tenha um valor igual a z_m. Suponha também que num determinado nó k da árvore, ainda não dividido em subproblemas, o limitante inferior fornecido pela solução da relaxação linear seja igual a z_k. Isso significa que qualquer solução inteira que se possa encontrar a partir de nós descendentes de k terá valor maior ou igual a z_k. Caso $z_k \geq z_m$, não é necessário dar continuidade ao processo de *branching* a partir do nó k, pois já se sabe que, nos nós descendentes de k, não haverá solução inteira melhor que z_m. Daí surge o nome *enumeração implícita*. Foram eliminados *todos* os nós descendentes de k sem a necessidade de se criá-los explicitamente. Esta, portanto, é uma terceira maneira de se interromper o avanço do algoritmo a partir de um nó da árvore de enumeração.

Por fim, dois tipos de decisões têm de ser tomadas a cada passo do algoritmo. É preciso escolher qual nó da árvore será o próximo a sofrer *branching* e, em seguida, qual variável de valor fracionário irá compor as restrições adicionadas aos subproblemas do nó selecionado. Essas decisões podem afetar sensivelmente o desempenho do algoritmo de *Branch-and-Bound* e, normalmente, a melhor estratégia depende do problema em questão.

2.3 Geração de Colunas

Esta seção descreve sucintamente o mecanismo de Geração de Colunas, comumente utilizado na resolução de problemas de Programação Linear com um número muito grande de colunas, ou de variáveis.

2.3.1 Conceitos Básicos

Seja o seguinte problema de Programação Linear

$$\begin{aligned} \min \quad & cx \\ \text{sujeito a} \quad & Ax = b, \\ & x \geq 0\,. \end{aligned}$$

Suponha que a matriz A possui tantas colunas que seria impossível armazenar todas elas de uma só vez na memória principal. Como foi visto na Seção 2.1, o algoritmo Simplex parte de uma base viável e prossegue caminhando de base em base até que não seja mais possível melhorar o valor da função objetivo. Isto equivale a dizer que não há variáveis não básicas com custo reduzido negativo (em caso de minimização). Se o número de colunas em A é relativamente pequeno, basta calcular o custo reduzido de todas as variáveis não básicas e verificar se alguma delas, ao entrar na base, poderia melhorar o valor da função objetivo. Em caso afirmativo, constrói-se a nova base e o processo se repete. Caso contrário, tem-se em mãos uma solução ótima para o programa linear em questão. Contudo, quando as colunas de A não estão disponíveis ou quando o seu número é muito grande, esta abordagem torna-se impraticável. Para alguns problemas, entretanto, é possível descobrir se alguma das variáveis não básicas possui custo reduzido negativo sem que haja a necessidade de se considerar todas elas explicitamente. E isso é suficiente para que se possa encontrar uma solução ótima para o problema.

O algoritmo de Geração de Colunas funciona em termos das chamadas iterações *mestre*. Resolve-se um problema restrito, i.e. com um subconjunto A^0 das colunas da matriz A. Obtém-se assim uma solução básica para este subproblema, juntamente com as colunas que compõem a base. De posse dos valores das variáveis duais associadas às restrições do problema original, dá-se início à resolução do subproblema *escravo*. Este subproblema consiste em encontrar um conjunto N contendo uma ou mais colunas de A com custo reduzido negativo. Se $N \neq \emptyset$, faz-se $A^{i+1} = A^i \cup N$ e o processo se repete. Caso contrário, a solução ótima do problema restrito corrente (A^i) também é uma solução ótima do problema original.

Não se sabe ao certo até que ponto a escolha do conjunto de colunas inicial A^0 pode influenciar o desempenho do algoritmo de Geração de Colunas. De acordo com Vanderbeck

[39], a escolha de uma base inicial trivial, tal como a matriz identidade, pode não ser uma boa idéia. Contudo, não há garantias de que uma outra base inicial, por exemplo obtida a partir de uma solução viável conhecida, não possa dar início à busca em uma direção equivocada. Também não há uma regra específica quanto à quantidade de colunas a adicionar à formulação em cada iteração mestre. No entanto, a experiência mostra que, para um bom desempenho do algoritmo, o tamanho ideal para o conjunto N depende do problema considerado, devendo ser ajustado experimentalmente.

Há diversas políticas diferentes para o gerenciamento das colunas que vão sendo acrescentadas na formulação ao longo da execução do algoritmo de Geração de Colunas. Pode-se manter na formulação todas as colunas geradas ou removê-las à medida em que vão saindo da base. Como estratégia intermediária, é possível ainda eliminar da formulação apenas aquelas colunas que não tenham permanecido na base por um número predeterminado de iterações consecutivas do algoritmo Simplex. Via de regra, a melhor estratégia a ser adotada depende em grande parte do problema em questão, e deve ser pesquisada empiricamente.

2.3.2 Um Exemplo: Corte Unidimensional

Considere uma empresa de papel que vende rolos pequenos de papel obtidos a partir do corte de rolos maiores[1], cuja largura é L. A demanda dos clientes da empresa pode ser descrita em termos de pares de números (b_i, ℓ_i), $i \in \{1, 2, \ldots, m\}$, indicando que b_i rolos de largura ℓ_i devem ser produzidos. Assume-se que $\ell_i \leq L$ para todo i e que todos os ℓ_i's são números inteiros. Dá-se o nome de *padrão* a uma maneira de se cortar um rolo grande em rolos menores. Por exemplo, um rolo grande de largura 70 pode ser cortado em três rolos de largura $\ell_1 = 17$ e um rolo de largura $\ell_2 = 15$, havendo uma sobra (ou desperdício) de largura 4.

É possível representar um padrão em termos de um vetor coluna a_j em que o i-ésimo elemento, a_{ij}, indica quantos rolos de largura ℓ_i são gerados por aquele padrão. No exemplo anterior, $a_j = (3, 1, 0, \ldots, 0)^T$. Para que um vetor $(a_{1j}, \ldots, a_{mj})^T$ represente um padrão viável basta que suas componentes sejam números inteiros e que a seguinte equação seja satisfeita

$$\sum_{i=1}^{m} a_{ij} \ell_i \leq L \,. \tag{2.3}$$

Seja n o número de padrões viáveis. Pode-se construir uma matriz $m \times n$, A, em que cada coluna a_j, $j \in \{1, \ldots, n\}$, é uma representação de um padrão viável. Note que, neste caso, n pode ser um número muito grande.

[1]Este exemplo foi retirado da Seção 6.2 de [1].

O objetivo da empresa é minimizar o número de rolos grandes utilizados para se atender à demanda de todos os clientes. Seja x_j o número de rolos grandes cortados segundo o padrão j. O programa linear abaixo descreve este problema de otimização.

$$\begin{aligned} \min \ & \sum_{j=1}^{n} x_j \\ \text{s. a.} \ & \sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i \in \{1, \dots, m\}, \\ & x_j \geq 0, \quad j \in \{1, \dots, n\}. \end{aligned}$$

A matriz identidade $m \times m$ constitui uma base inicial viável para este problema (conjunto A^0). Isto é, as colunas da base são os m padrões que produzem apenas um rolo de largura ℓ_i e nenhum rolo das demais larguras, para cada $i \in \{1, \dots, m\}$.

Tendo em mãos uma base viável B e sua solução básica associada, é preciso executar a próxima iteração do método Simplex. O vetor de variáveis duais do problema é dado por $p = c_B B^{-1}$. Por sua vez, o custo reduzido de uma coluna (padrão) a_j é dado por $\overline{c}_j = 1 - p\, a_j$. Em vez de se calcular o custo reduzido associado a cada um dos padrões a_j, resolve-se o problema de minimizar $1 - p\, a_j$ (ou maximizar $p\, a_j$) sobre todos os a_j. Se este máximo for menor ou igual a 1, não há colunas com custo reduzido negativo e a solução corrente é ótima. Caso contrário, a coluna a_j que maximiza o valor de $\overline{c}_j$ tem custo reduzido negativo e pode entrar na base.

Dada a descrição de um padrão viável (Equação 2.3), o problema de maximização acima é descrito como

$$\begin{aligned} \max \ & \sum_{i=1}^{m} p_i a_i \\ \text{s. a.} \ & \sum_{i=1}^{m} \ell_i a_i \leq L \\ & a_i \geq 0 \text{ e inteiro}, \quad i \in \{1, \dots, m\}. \end{aligned}$$

Note que esta formulação se identifica com um problema de *mochila*: p_i's são os valores dos itens, ℓ_i's são os seus pesos e L é a capacidade da mochila. Na prática, este problema pode ser resolvido de maneira eficiente por um algoritmo de Programação Dinâmica pseudo-polinomial [1].

2.3.3 *Branch-and-Price*

Quando se busca a solução de um problema de Programação Linear Inteira em que o número de colunas é muito grande, pode-se embutir o método de Geração de Colunas em

um algoritmo de *Branch-and-Bound*. O algoritmo resultante é normalmente conhecido como *Branch-and-Price*, pois a operação de busca por uma coluna com custo reduzido negativo também é chamada de *pricing*.

Num algoritmo de *Branch-and-Bound* puro, sempre que um nó da árvore de enumeração é selecionado para exploração, resolve-se a relaxação linear do subproblema associado a ele. Num algoritmo de *Branch-and-Price*, o processo é muito semelhante. A diferença está no fato de que a relaxação linear de cada subproblema é resolvida sem levar em consideração a totalidade das suas colunas, i.e. utiliza-se um processo de Geração de Colunas.

2.4 Relaxação Lagrangeana

Nesta seção discutem-se as principais idéias por trás do método de Relaxação Lagrangeana. Para uma introdução mais detalhada sobre o assunto, sugere-se a leitura de [18] e do Capítulo 6 de [34], os quais serviram de base para esta seção.

2.4.1 Conceitos Básicos

A teoria de Relaxação Lagrangeana conforme conhecida nos dias de hoje teve como seu marco inicial o trabalho de Held e Karp sobre o problema do Caixeiro Viajante, em 1970 [23].

A idéia principal reside no fato de que muitos problemas considerados difíceis podem ser encarados como problemas mais fáceis acrescidos de um pequeno conjunto de restrições complicadoras, denominadas *side constraints*. Dualizando-se as *side constraints*, produz-se um chamado *Problema Lagrangeano* que, em geral, deve ser fácil de resolver e pode fornecer um limite inferior de boa qualidade para o problema original.

Seja o seguinte problema de Programação Linear Inteira

$$\begin{aligned} z = \min \quad & cx \\ \text{s. a.} \quad & Ax \leq b \\ & Dx \leq e \\ & x \geq 0 \ \text{ e inteiro}, \end{aligned} \tag{P}$$

onde A é uma matriz $m \times n$, D é uma matriz $r \times n$, c é o vetor de custos $1 \times n$, x é o vetor de variáveis $n \times 1$ e os vetores b e e têm dimensões $m \times 1$ e $r \times 1$, respectivamente. As restrições de (P) foram divididas nos conjuntos $Ax \leq b$ e $Dx \leq e$. Suponha que a ausência das restrições $Ax \leq b$ permitiria a resolução do problema resultante por um algoritmo polinomial ou pseudo-polinomial, por exemplo. Seja agora a Relaxação Lagrangeana de

(P) dada abaixo

$$\begin{aligned} z_D(u) = \min \ & cx + u\,(Ax - b) \\ \text{s. a. } \ & Dx \leq e \\ & x \geq 0 \ \text{ e inteiro}, \end{aligned} \qquad (\text{LR}_u)$$

onde $u = (u_1, \ldots, u_m)$ é chamado de vetor de *multiplicadores de Lagrange*. Seja x^* uma solução ótima para (P). Como $z = cx^*$ e $Ax^* - b \leq 0$, tem-se que, para um vetor $u \geq 0$ dado,

$$z_D(u) \leq cx^* + u\,(Ax^* - b) \leq z\,.$$

Este resultado permite que se use a solução de (LR_u) como um limite inferior para (P) ao invés de sua relaxação linear tradicional. Em certos casos, essa troca é vantajosa pois a resolução da relaxação linear de (P) pode exigir um esforço computacional muito maior do que a resolução de (LR_u).

2.4.2 Um Exemplo Simplificado

Para ilustrar melhor as idéias da seção anterior, seja o problema de Cobertura de Conjuntos a seguir. A matriz de coeficientes A é dada por: $a_{ij} = 1$ se o conjunto (coluna) j cobre o elemento (linha) i. Caso contrário, a_{ij} é igual a 0. O problema de cobertura consiste em selecionar um subconjunto das colunas de A cujo custo seja mínimo e que cubra cada linha pelo menos uma vez:

$$\begin{aligned} \min \ & \sum_{j=1}^{n} c_j x_j \\ \text{s. a. } \ & \sum_{j=1}^{n} a_{ij} x_j \geq 1, \ \ i \in \{1, \ldots, m\} \\ & x_j \in \{0, 1\}, \ \ j \in \{1, \ldots, n\}\,. \end{aligned}$$

As variáveis booleanas x_j indicam se uma coluna pertence ou não à solução. A Relaxação Lagrangeana desse problema é dada por

$$\begin{aligned} \min \ & \sum_{j=1}^{n} c_j x_j + \sum_{i=1}^{m} u_i (1 - \sum_{j=1}^{n} a_{ij} x_j) \\ \text{s. a. } \ & x_j \in \{0, 1\}, \ \ j \in \{1, \ldots, n\} \end{aligned}$$

que equivale a

$$\min \sum_{j=1}^{n} C_j x_j + \sum_{i=1}^{m} u_i$$
$$\text{s. a. } x_j \in \{0,1\}, \; j \in \{1,\dots,n\},$$

onde $C_j = [c_j - \sum_{i=1}^{m} u_i a_{ij}]$ é denominado *custo Lagrangeano* de x_j. Dado um vetor de multiplicadores $u \geq 0$, a solução ótima desse problema Lagrangeano pode ser obtida por inspeção. Basta fazer $x_j = 1$ se $C_j \leq 0$ e $x_j = 0$ caso contrário.

Até este ponto, não se levou em consideração a escolha do vetor de multiplicadores de Lagrange, u. Sendo $z_D(u)$ um limite inferior para o valor da solução ótima de (P), seria interessante encontrar o vetor u que maximizasse o valor de $z_D(u)$

$$z_D = \max_u \; z_D(u)\,. \tag{D}$$

O problema (D) recebe o nome de problema *Dual Lagrangeano*. Existem muitas maneiras de se determinar o vetor u que fornece uma solução ótima, ou quase ótima, para (D). Um dos métodos de mais comumente utilizados na resolução de (D) é apresentado na seção seguinte.

2.4.3 O Método do Subgradiente

É possível mostrar que $z_D(u)$ é uma função linear por partes, convexa e não diferenciável (vide [18]). A propriedade de convexidade permitiria a utilização de um método do tipo *hill climbing*, tal como o método do gradiente, para se resolver (D). Entretanto, esta abordagem só é possível quando se trabalha com funções diferenciáveis. O método do subgradiente é uma adaptação do método do gradiente em que os gradientes são substituídos pelos chamados *subgradientes*. Desse modo, viabiliza-se a otimização de funções não diferenciáveis.

Um vetor y é um *subgradiente* de $z_D(u)$ em $\overline{u}$ se

$$z_D(u) \leq z_D(\overline{u}) + y(u - \overline{u}), \quad \text{para todo } u$$

Sempre que x' é solução de (LR$_u$), para qualquer u, pode-se verificar que o vetor $(Ax' - b)$ é um subgradiente de $z_D(u)$ [18].

O método do subgradiente parte de um vetor de multiplicadores de Lagrange u^i conhecido e vai calculando uma seqüência de novos vetores u^k, $(k > i)$, com o objetivo de aproximar $z_D(u)$ de z_D. Vale ressaltar que $z_D(u)$ não se aproxima monotonicamente de z_D.

Seja x^k a solução ótima de (LR$_{u^k}$) e u^0 um vetor de multiplicadores de Lagrange dado. A seqüência de vetores u^k é calculada da seguinte forma

$$u^{k+1} = u^k + t_k(Ax^k - b)\,.$$

Se o tamanho do passo t_k tende a zero à medida que k cresce e $\sum_{i=0}^{k} t_i$ converge para zero, é possível provar que o valor de $z_D(u^k)$ converge para z_D [18]. Isto é, o método do subgradiente converge para solução ótima num número finito de passos. Todavia, em certos casos práticos, os valores de t_i que geralmente produzem os melhores desempenhos do método não obedecem às propriedades de convergência. Na prática, interrompe-se o processo assim que um determinado limite no número de iterações é atingido.

O tamanho do passo t_k pode ser calculado de diferentes formas. Uma fórmula comumente utilizada é

$$t_k = \frac{\lambda_k \left(z^* - z_D(u^k)\right)}{||Ax^k - b||^2}, \quad \lambda_k \in (0..2]\,,$$

onde z^* é um limite superior para o valor de z_D [18]. O valor de λ_k é ajustado de acordo com o ritmo de aproximação de $z_D(u)$ com relação a z_D [18].

2.4.4 Contextos de Aplicação

O método de Relaxação Lagrangeana pode ser utilizado em diversos contextos diferentes. Como já mencionado no início da Seção 2.4.1, há problemas de Programação Linear Inteira para os quais o limite inferior obtido com o auxílio de Relaxação Lagrangeana tem um custo computacional baixo quando comparado ao custo de se resolver uma relaxação linear do mesmo problema. Uma aplicação direta deste limite inferior ocorre em algoritmos de *Branch-and-Bound*. Além disso, a solução de (LR_u) pode prover informação útil para guiar a regra de *branching* de um algoritmo de *Branch-and-Bound*. Por fim, ainda é possível obter soluções viáveis para (P) a partir de perturbações sobre soluções de (LR_u), que nem sempre são factíveis para (P). Esse tipo de algoritmo também recebe o nome de *heurística Lagrangeana*.

Capítulo 3

Programação por Restrições

Were you to ask me which programming paradigm is likely to gain most in commercial significance over the next 5 years I'd have to pick Constraint Logic Programming (CLP), even though it's perhaps currently one of the least known and understood.

Dick Pountain, Byte Magazine, fevereiro de 1995.

Neste capítulo, apresenta-se uma introdução à teoria de Programação por Restrições baseada em lógica[1]. Os conceitos principais necessários à compreensão do trabalho desenvolvido nessa dissertação são discutidos e exemplificados. Para uma introdução mais completa sobre este tema, sugere-se a leitura de [20, 27, 30].

Assume-se que o leitor possui conhecimentos elementares a respeito de Lógica de Primeira Ordem e programação na linguagem PROLOG. Neste sentido, recomenda-se a leitura dos capítulos 1 e 2 de [32] e dos capítulos 1 a 4 de [10], respectivamente.

3.1 Origens

Na década de 70, verificou-se o aparecimento do paradigma de programação conhecido como *Programação em Lógica*, tendo o PROLOG como uma de suas linguagens mais populares. O novo conceito introduzido por este paradigma de programação permite que o programador se preocupe com *o quê* (lógica) sem se ater ao *como* (controle). Isto representa uma grande mudança com relação ao paradigma de programação imperativa, característico de linguagens como FORTRAN, Pascal e C, onde prevalece o *como*. Nestas últimas linguagens, é preciso descrever, explicitamente, todos os passos a serem tomados para se chegar ao resultado final desejado. Já em PROLOG, por exemplo, é possível escrever programas de

[1] *Constraint Logic Programming*

maneira natural e declarativa, ganhando-se em expressividade e facilidades de compreensão e manutenção.

É claro que a Programação em Lógica também possui suas desvantagens, sendo duas delas as mais importantes. Em primeiro lugar, os objetos representados dentro de um programa em lógica são puramente sintáticos (elementos do universo de Herbrand associado ao programa em questão). Só é possível verificar a igualdade de objetos sintaticamente idênticos. Por sua vez, objetos semanticamente mais ricos têm de ser codificados explicitamente como termos da linguagem, i.e. não há representação implícita. Por exemplo, o conjunto de todos os pares (x, y) tais que $y^2 = x$ é um conjunto infinito e não poderia ser retornado como resposta de um programa numa linguagem como PROLOG. Contudo, Programação por Restrições pode retornar uma resposta simbólica do tipo $y^2 = x$.

A outra desvantagem advém do mecanismo de computação que nada mais é do que uma busca em profundidade na árvore de todas as possíveis derivações lógicas obtidas a partir do *goal* inicial e dos fatos e regras descritos no programa principal. A esse tipo de procedimento dá-se o nome de *generate-and-test* pois a viabilidade de uma solução só pode ser determinada no momento em que todas as incógnitas iniciais (variáveis) possuírem um valor definido ou quando não for mais possível dar continuidade ao processo de inferência. Isto pode implicar tempos de computação muito elevados caso a busca prossiga por ramos da árvore onde não existam soluções viáveis. No caso de aplicações de médio e grande porte, esse tipo de ineficiência pode tornar proibitivo o uso de Programação em Lógica.

O paradigma de Programação por Restrições procura superar esses problemas, sem contudo abrir mão das vantagens que uma linguagem declarativa oferece. Em 1987, o trabalho de Jaffar e Lassez [26] estabeleceu os fundamentos teóricos a partir dos quais diversas linguagens de Programação por Restrições puderam ser desenvolvidas[2]. A idéia básica é substituir o mecanismo de inferência lógica tradicional (unificação) por um mecanismo mais genérico e eficiente, conhecido como *manipulação de restrições*. Tais mecanismos de manipulação de restrições já são usados no campo da Inteligência Artificial desde a década de 80, na resolução dos chamados *Constraint Satisfaction Problems* (CSPs). Um CSP é composto por

- Um conjunto de *variáveis* $X = \{x_1, \ldots, x_n\}$;
- Para cada variável x_i, um conjunto finito de valores, D_i, denominado *domínio*;
- Um conjunto de *restrições*, C, que atuam sobre subconjuntos das variáveis de X, limitando os valores que lhes podem ser atribuídos.

Uma solução para um CSP é uma atribuição de valores a todas as variáveis de X que obedeça às restrições de C.

[2]Entretanto, vale ressaltar que desde a década de 60 já existem linguagens que lidam com restrições.

Muitos problemas em Pesquisa Operacional, tais como escalonamento de tarefas, alocação de horários e outros problemas combinatórios podem ser representados como CSPs. Programação por Restrições vem se afirmando como uma ferramenta alternativa bastante poderosa para abordar tais problemas.

3.2 Conceitos Básicos

A maneira mais simples e tradicional de se ilustrarem as principais idéias a respeito de Programação por Restrições é através da análise de programas exemplo. Os trechos de código em Programação por Restrições apresentados nesta seção seguem a sintaxe da linguagem ECL^{i}PS^e [3].

Um programa em Programação por Restrições geralmente obedece à estrutura apresentada na Figura 3.1.

```
<Declaração de Variáveis e Domínios>
<Imposição de Restrições>
<Busca por Soluções (labeling)>
```

Figura 3.1: Estrutura básica de um programa em Programação por Restrições

Note que essa estrutura se assemelha a um CSP, conforme descrito na Seção 3.1. As variáveis com seus domínios definem o espaço de soluções. As restrições estabelecem *relações* entre as variáveis, limitando os valores que elas podem assumir concomitantemente e reduzindo o espaço de busca. Essa redução se processa através de um mecanismo denominado *propagação de restrições*, cuja função é garantir a *consistência parcial* do sistema (CSP) como um todo. A entidade responsável por controlar este mecanismo recebe o nome de *constraint solver*.

O mecanismo de propagação de restrições funciona da seguinte forma. Toda vez que o domínio de uma variável é alterado, e.g. pela remoção de um de seus elementos, esta informação é transmitida (propagada) para as demais variáveis associadas a ela por uma ou mais restrições. Durante esse processo, procura-se sempre manter a consistência do sistema, i.e. a satisfação das restrições do programa.

Por exemplo: sejam X e Y variáveis que podem assumir valores inteiros no intervalo (domínio) $[1, 10]$. Isto significa que, inicialmente, os domínios de X e Y são os mesmos: $D_X = D_Y = [1, 10]$. Impondo-se a restrição de que $X < Y$, os domínios se alteram para $D_X = [1, 9]$ e $D_Y = [2, 10]$. Isto porque, se $X = 10$ e $X < Y$, conclui-se que $Y \geq 11$.

[3] `http://www.icparc.ic.ac.uk/eclipse`

Contudo 11 não pertence ao domínio de Y. Analogamente, mostra-se que Y não pode receber o valor 1. Suponha agora que a restrição $X \geq 5$ seja imposta. Isto faz com que $D_X = [5, 9]$. Como Y está relacionada a X pela restrição $X < Y$, o domínio de Y é automaticamente alterado para $D_Y = [6, 10]$.

A consistência do sistema é dita *parcial* porque não há algoritmos polinomiais conhecidos que garantam a consistência *total* (ou global) de um CSP genérico[4]. Por questões de eficiência, os algoritmos normalmente utilizados para detectar violações das restrições não são capazes de identificar certos tipos de estados inconsistentes. Um desses algoritmos, denominado *arc-consistency*, funciona assim: sejam duas variáveis, A e B, relacionadas por uma restrição r, e cujos domínios são, respectivamente, D_A e D_B. Diz-se que um valor $v_A \in D_A$ tem *suporte* no domínio D_B se, ao atribuir-se v_A a A, existir um valor $v_B \in D_B$ que pode ser atribuído a B sem violar r. O algoritmo, portanto, remove do domínio de cada variável todos aqueles valores que não têm suporte no domínio de uma outra variável qualquer, associada a ela por alguma restrição.

Por exemplo, no programa abaixo, os domínios iniciais das variáveis X, Y e Z são iguais a [0,1] e, na sintaxe do ECL^{i}PS^e, a restrição ## indica que as variáveis devem assumir valores diferentes:

```
prog([X,Y,Z]) :-
      [X,Y,Z]::[0,1],
       X ## Y, X ## Z, Y ## Z.
```

É fácil verificar que não há maneira de atribuir valores às variáveis X, Y e Z de modo a satisfazer a todas as restrições. Entretanto, essa inconsistência não é detectada pelo algoritmo de *arc-consistency.*

Diante disso, é preciso mais trabalho para se encontrar uma solução viável para um problema. Em resumo, o que acontece é o seguinte: os domínios originais das variáveis são inicialmente reduzidos pelas restrições do programa. Neste instante, é possível que haja alguma inconsistência ainda não detectada. A seguir, dá-se início ao processo de *labeling* (vide Figura 3.1) onde, a cada passo, seleciona-se uma variável a qual será atribuído um valor de seu domínio. A ordem de escolha das variáveis e dos valores é totalmente flexível, ficando a critério do programador. Ao se atribuírem valores às variáveis, o mecanismo de propagação se encarrega de restabelecer a consistência do sistema. Caso alguma inconsistência seja detectada, a última atribuição de valor tem de ser desfeita, dando lugar a uma outra alternativa (*backtracking*). O processo todo se repete até que uma solução seja encontrada ou até que se prove a inexistência de soluções viáveis.

O próximo exemplo, extraído de [13], ilustra melhor algumas das idéias discutidas até agora. Seja a seguinte operação de adição em aritmética simbólica

[4] CSPs pertencem à classe de problemas $\mathcal{NP}$-completos

```
 SEND
+ MORE
------
MONEY
```

A cada letra corresponde um número inteiro distinto no intervalo $[0, 9]$. O objetivo é descobrir quais números atribuir a quais letras de modo que a operação de adição esteja correta. O programa a seguir resolve o problema:

```
cripto(Vars) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars::[0..9],
        alldifferent(Vars),
        M ## 0,
        S ## 0,
                    1000*S + 100*E + 10*N + D
                  + 1000*M + 100*O + 10*R + E #=
        10000*M + 1000*O + 100*N + 10*E + Y,
        labeling(Vars).
```

Inicialmente, cria-se uma variável para cada letra da adição. Seja `Vars` a lista dessas variáveis. Como exposto inicialmente, cada variável pode assumir um valor inteiro entre 0 e 9, o que é representado pela restrição `Vars::[0..9]`. Para garantir que letras distintas assumam valores distintos, utilizou-se a restrição predefinida `alldifferent`, que recebe como parâmetro a lista de variáveis. Essa restrição funciona como se existissem 28 restrições de desigualdade entre cada par de variáveis distintas, tomadas dentre as 8 variáveis do programa. As letras `M` e `S` não podem assumir o valor 0 pois são dígitos iniciais dos números, daí o uso da restrição `##`, que indica desigualdade. A última restrição diz respeito à operação de adição propriamente dita, onde o símbolo `#=` denota a restrição de igualdade.

Antes mesmo que a busca por uma solução tenha início, o mecanismo de propagação de restrições já é capaz de reduzir os domínios das variáveis. Seja a operação intermediária de adição correspondente à coluna que contém as variáveis `S`, `M` e `O`. Sabe-se que

$$V_1 + \mathtt{S} + \mathtt{M} = \mathtt{O} + 10V_2\,,$$

onde V_1 é o "vem-um" da coluna anterior e V_2 é o "vai-um" dessa coluna. Com isso, deduz-se que o dígito inicial do número `MONEY` (letra `M`) é igual a V_2. Como V_1 e $V_2 \in \{0, 1\}$ e $\mathtt{M} \neq 0$, tem-se que $V_2 = \mathtt{M} = 1$ e o valor 1 é removido dos domínios das demais letras. Assim, a equação anterior reduz-se a

$$V_1 + \mathtt{S} = \mathtt{O} + 9\,.$$

Como $\mathtt{O} + 9 \geq 9$ e $V_1 \leq 1$, conclui-se que $\mathtt{S} \geq 8$. Logo, o domínio de S é reduzido para $[8, 9]$. Além disso, como $V_1 + \mathtt{S} \leq 10$, tem-se que $\mathtt{O} \leq 1$. Mas o valor 1 já foi eliminado do domínio de `O` e, portanto, `O` é fixada em 0. Para que V_1 seja igual a 1, a letra `N` tem que assumir o valor 0, pois V_1 é o "vai-um" da coluna com as letras `E`, `O` e `N`. Como `O` já é igual a 0, conclui-se que $V_1 = 0$ e, por conseguinte, $\mathtt{S} = 9$.

Resumindo, a princípio, `S`, `M` e `O` são fixadas em 9, 1 e 0, respectivamente, e as demais letras têm seus domínios reduzidos para o intervalo $[2, 8]$, por causa da restrição `alldifferent`. A partir deste ponto, a busca (*labeling*) tem início atribuindo-se o valor 2 à variável `E`. Como esse valor é inconsistente com as restrições, ocorre um *backtracking* e tenta-se utilizar o próximo valor do domínio. Este processo se repete até que a atribuição do valor 5 à letra `E` consegue ser bem sucedida. Como resultado disso, as demais variáveis têm seus domínios reduzidos para um único valor possível, produzindo a primeira solução viável: `S=9`, `E=5`, `N=6`, `D=7`, `M=1`, `O=0`, `R=8` e `Y=2`.

Em certos casos, procura-se não somente uma solução viável qualquer mas sim aquela que minimiza (ou maximiza) uma função objetivo dada. Uma forma de dar suporte ao conceito de otimização em Programação por Restrições é a seguinte. Sempre que uma solução é encontrada, calcula-se o seu custo c e impõe-se uma nova restrição indicando que o custo da solução procurada agora deve ser menor do que c (em caso de minimização). Repete-se este processo até que não haja mais soluções viáveis. Neste ponto, a última solução encontrada é uma solução de custo ótimo.

3.3 Semântica

Esta seção define mais formalmente as idéias básicas por trás de uma linguagem de Programação por Restrições. Seja $\mathcal{X}$ uma quádrupla $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$, onde:

Σ é uma *assinatura*, i.e. um conjunto de símbolos que denotam constantes, funções e/ou predicados, junto com seus respectivos cabeçalhos (números de parâmetros);

$\mathcal{D}$ é uma *estrutura* para Σ formada por um domínio não vazio $\mathcal{A}$ e uma atribuição semântica para cada símbolo de Σ (constantes, predicados e funções). Em outras palavras, cada símbolo de constante em Σ é associado a algum elemento do domínio, cada símbolo de predicado de Σ é associado a uma *relação* no domínio e cada símbolo de função de n parâmetros de Σ é associado a uma função de n parâmetros em $\mathcal{A}$.

$\mathcal{L}$ é a *classe de fórmulas*, ou restrições, que podem ser representadas usando-se símbolos de Σ. O conjunto $\mathcal{A}$ pode ser entendido como o domínio de computação sobre o qual são representadas as restrições da classe definida por $\mathcal{L}$.

$\mathcal{T}$ é uma *coleção de fórmulas* que estabelece as propriedades de $\mathcal{D}$ de maneira axiomática.

A linguagem de Programação por Restrições sobre o domínio de computação e de restrições $\mathcal{X}$ é indicada por CLP($\mathcal{X}$). Um *programa* em CLP($\mathcal{X}$) é formado por um conjunto finito de *regras de restrições* que obedecem à seguinte forma

$$\mathtt{h} : -\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n\,, \quad \mathtt{n} \geq 0\,,$$

onde h é um *átomo* e cada $\mathtt{a_i}$ pode ser um átomo ou uma restrição de $\mathcal{L}$. Um átomo segue a forma $\mathtt{p(b_1, \ldots, b_m)}$, onde p é um símbolo de predicado. Um *goal* em CLP($\mathcal{X}$) é escrito como

$$? - \mathtt{g}_1, \mathtt{g}_2, \ldots, \mathtt{g}_k\,, \quad \mathtt{k} \geq 1\,,$$

indicando que se deseja saber se a conjunção dos átomos $\mathtt{g_i}$ é ou não uma conseqüência lógica do programa em questão.

Uma maneira de se descrever a semântica operacional de uma linguagem CLP($\mathcal{X}$) é através de um sistema de transição de estados. Cada estado é formado por um par $\langle A, C \rangle$, onde A é um conjunto de átomos ou restrições e C é um conjunto de restrições denominado *repositório de restrições*. Em outras palavras, A contém os *goals* ainda pendentes e C contém o conjunto de restrições acumuladas desde o início da computação até o instante presente. Utiliza-se o símbolo $\perp$ para denotar um estado de *falha* (ou inconsistência). É preciso também que haja uma *regra de computação* que selecione, a cada iteração, o próximo elemento de A a ser considerado, bem como a transição a ser tomada a partir do estado atual. Todo este sistema de transição de estados depende ainda de dois elementos: o predicado consistent e a função infer. O predicado consistent(C) indica se o conjunto de restrições C é *consistente* no sentido de que existe uma atribuição de valores para as variáveis livres de C, denotadas por $\vec{x}$, e que satisfaça a todas as restrições de C:

$$\mathsf{consistent}(C) \Leftrightarrow \mathcal{D} \models \exists \vec{x} : C\,.$$

A função infer(C) retorna um conjunto de restrições C' que pode ser obtido a partir do conjunto C por meio de operações de simplificação, como por exemplo, através da eliminação de restrições redundantes:

$$\mathcal{D} \models \mathsf{infer}(C) \Leftrightarrow C\,.$$

Caso a regra de computação escolha uma restrição c de A, a transição a partir do estado atual $\langle A, C \rangle$ é dada por

$$\langle A, C \rangle \rightarrow \begin{cases} \langle A \setminus \{c\}, C' \rangle & \text{se} \quad \mathsf{consistent}(C \cup \{c\}) \text{ e } C' = \mathsf{infer}(C \cup \{c\}) \\ \perp & \text{se} \quad \neg\, \mathsf{consistent}(C \cup \{c\}) \end{cases}.$$

Caso a regra de computação escolha um átomo $\mathtt{b}$ de A e o programa contenha uma regra da forma $\mathtt{h} : -\mathtt{a}_1, \ldots, \mathtt{a}_n$ tal que $\mathtt{b}$ e $\mathtt{h}$ possuam o mesmo símbolo de predicado, a transição de estado se processa assim

$$\langle A, C \rangle \rightarrow \begin{cases} \langle (A \setminus \{\mathtt{b}\}) \cup \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}, C' \rangle & \text{se} \quad \begin{array}{l} \mathsf{consistent}(C \cup \{\mathtt{h} = \mathtt{b}\}) \text{ e} \\ C' = \mathsf{infer}(C \cup \{\mathtt{h} = \mathtt{b}\}) \end{array} \\ \\ \bot & \text{se} \quad \neg\,\mathsf{consistent}(C \cup \{\mathtt{h} = \mathtt{b}\}) \end{cases},$$

onde o conjunto das restrições correspondentes à unificação[5] dos argumentos de $\mathtt{h}$ e $\mathtt{b}$ é denotado resumidamente por $\{\mathtt{h} = \mathtt{b}\}$.

Dá-se o nome de *derivação* a uma seqüência, finita ou infinita, de transições de estados

$$\langle A_0, C_0 \rangle \rightarrow \langle A_1, C_1 \rangle \rightarrow \cdots \rightarrow \langle A_i, C_i \rangle \rightarrow \cdots .$$

Suponha que, dado um programa em CLP($\mathcal{X}$), deseje-se determinar a validade do *goal* $? - \mathtt{g}_1, \ldots, \mathtt{g}_k$. O estado inicial de computação é dado por $\langle \{\mathtt{g}_1, \ldots, \mathtt{g}_k\}, \emptyset \rangle$. Um *estado final* é um estado a partir do qual o sistema de transição de estados não é capaz de atingir outro estado. Quando o estado final de uma derivação é da forma $\langle \emptyset, C \rangle$, diz-se que a derivação foi *bem sucedida* e que C é a *answer constraint* da derivação. Caso o estado final seja $\bot$, considera-se que a derivação falhou.

Partindo-se de um programa em CLP($\mathcal{X}$) e um *goal* inicial, é possível gerar várias derivações distintas. Cada uma dessas derivações pode levar a uma solução (*answer constraint*) diferente ou a uma seqüência infinita de transições. Este processo é bastante semelhante ao processo de busca por soluções que ocorre dentro da árvore de busca de um programa em PROLOG.

Note que as implementações da regra de computação, do predicado consistent e da função infer estão intimamente relacionadas ao *constraint solver* específico para o domínio de computação considerado. Este domínio pode ser o dos números inteiros, o dos números reais, booleanos, racionais e outros, inclusive suas combinações.

3.4 Implementações

Na Seção 3.2, utilizou-se como exemplo o *solver* de restrições para domínios finitos (inteiros) presente na linguagem ECL^{i}PS^e. Entretanto, existem diversas outras linguagens que implementam *solvers* específicos para este e outros domínios de computação. Entre elas estão:

[5]Uma explicação sobre o mecanismo de unificação pode ser encontrada em [32].

Restrições em Domínios Finitos — Adequadas para problemas de escalonamento, planejamento, cortes e muitos outros. Exemplos de linguagens: CLP($\mathcal{FD}$), ECL^{i}PS^e, ILOG Solver, MOZART e GNU PROLOG.

Restrições Booleanas — Muito úteis para programas de teste sobre circuitos lógicos e verificação de propriedades de sistemas complexos. Exemplos de linguagens: CHIP, PROLOG III e SICSTUS.

Restrições sobre Conjuntos — Aplicam-se melhor a problemas combinatórios baseados em conjuntos, relações ou grafos. Por exemplo: localização de facilidades, alocação de recursos, empacotamento em reservatórios etc. Exemplo de linguagem: ECL^{i}PS^e.

Restrições em Aritmética Linear — Utilizadas em problemas oriundos da área de engenharia elétrica, bolsas de valores e raciocínio temporal. Também são capazes de lidar com restrições não lineares deixando-as temporariamente inativas até a instanciação de um número de variáveis suficiente para eliminar a não-linearidade. Exemplos de linguagens: CHIP, CLP($\mathcal{R}$) e ECL^{i}PS^e.

3.5 Restrições Predefinidas

As diversas linguagens de Programação por Restrições disponíveis no mercado costumam incluir predicados predefinidos que implementam certos tipos de restrições comumente encontradas em grande parte dos programas. Essas implementações normalmente se utilizam de recursos internos da linguagem ou, ainda, são feitas em outras linguagens, como C. Portanto, gozam de uma eficiência que dificilmente seria conseguida caso o usuário final tivesse de implementá-las por si só, a partir da combinação de restrições presentes na linguagem.

A seguir descrevem-se exemplos importantes de restrições predefinidas:

`alldifferent(L)` — `L` é uma lista de variáveis. Essa restrição obriga que todas as variáveis em `L` assumam valores distintos entre si. Equivale a uma restrição de desigualdade entre cada par de variáveis distintas de `L`. Quando uma variável recebe um valor, este é removido dos domínios das demais variáveis.

`atmost(N,L,V)` — `N` é um número inteiro e `V` é uma constante. `L` é uma lista de variáveis. Essa restrição obriga que no máximo `N` variáveis da lista `L` assumam o valor `V`. Quando isso ocorre, o valor `V` é removido do domínio das demais variáveis da lista `L`.

`element(I,L,X)` — `I` pode ser uma variável ou um número inteiro e `X` pode ser uma variável ou uma constante. `L` é uma lista de constantes. A semântica dessa restrição diz que `X` é o `I`-ésimo elemento da lista `L`, i.e. `L[I]=X`. Quando o domínio de `X` é alterado, o domínio de `I` é atualizado de forma coerente, e vice-versa.

Parte II

O Trabalho Desenvolvido

Capítulo 4

O Problema de *Crew Scheduling*

Prólogo

O artigo a seguir trata o problema de *crew scheduling* por meio de diversas abordagens. Após uma descrição detalhada do problema e das restrições envolvidas, são apresentados os itens a seguir:

- Um modelo de Programação Linear Inteira para o problema que foi resolvido utilizando-se três algoritmos diferentes:
  - Um algoritmo de *Branch-and-Bound*;
  - Um algoritmo de *Branch-and-Price* em que o subproblema gerador de colunas é resolvido por meio de um algoritmo de Programação Dinâmica;
  - Uma heurística baseada em Relaxação Lagrangeana;
- Um modelo em Programação por Restrições para o problema que, de forma semelhante aos algoritmos anteriores, não foi capaz de resolver instâncias de grande porte;
- Um algoritmo de *Branch-and-Price* híbrido para resolver o modelo anterior de Programação Linear Inteira, onde as colunas são geradas dinamicamente através de Programação por Restrições. Neste caso, foi possível obter soluções ótimas comprovadas para instâncias do problema com até 12 milhões de colunas presentes na matriz de coeficientes.

Este artigo é uma versão reduzida do relatório técnico [45] e foi submetido para a conferência *Second International Workshop on Practical Aspects of Declarative Languages*, que ocorreu na cidade de Boston, MA, EUA, nos dias 17 e 18 de janeiro de 2000. O

trabalho foi aceito e publicado no volume 1753 da série *Lecture Notes in Computer Science*, que contém os anais da conferência [46].

A divulgação deste artigo, quando de sua submissão, resultou em um convite para que seus resultados fossem apresentados na *INFORMS Fall 1999 Meeting*, que ocorreu na cidade da Filadélfia, PA, EUA, no período de 7 a 10 de novembro de 1999 [44]. O convite foi feito pelo Dr. Mark Wallace do Imperial College de Londres, responsável por uma das sessões do congresso.

A Hybrid Approach for Solving Large Scale Crew Scheduling Problems

Tallys H. Yunes*
tallys@acm.org

Arnaldo V. Moura
arnaldo@dcc.unicamp.br

Cid C. de Souza†
cid@dcc.unicamp.br

Abstract

We consider several strategies for computing optimal solutions to large scale crew scheduling problems. Provably optimal solutions for very large real instances of such problems were computed using a hybrid approach that integrates mathematical and constraint programming techniques. The declarative nature of the latter proved instrumental when modeling complex problem restrictions and, particularly, in efficiently searching the very large space of feasible solutions. The code was tested on real problem instances, containing an excess of 1.8×10^9 entries, which were solved to optimality in an acceptable running time when executing on a typical desktop PC.

1 Introduction

Urban transit crew scheduling problems have been receiving a great deal of attention for the past decades. In this article, we report on a hybrid strategy that is capable of efficiently obtaining provably optimal solutions for some large instances of specific crew scheduling problems. The hybrid approach we developed meshes some classical Integer Programming (IP) techniques and some Constraint Programming (CP) techniques. This is done in such a way as to extract the power of these two approaches where they contribute their best towards solving the large scheduling problem instances considered. The resulting code compiles under the Linux operating system, kernel 2.0. Running on a 350 MHz desktop PC with 320 MB of main memory, it computed optimal solutions for problem instances with an excess of 1.8×10^9 entries, in a reasonable amount of time.

The problem instances we used stem from the operational environment of a typical Brazilian transit company that serves a major urban area. In this scenario, employee wages may well rise to 50 percent or more of the company's total expenditures. Hence, in these situations, even small percentage savings can be quite significant.

*Supported by FAPESP grant 98/05999-4, and CAPES.
†Supported by FINEP (ProNEx 107/97), and CNPq (300883/94-3).

We now offer some general comments on the specific methods and techniques that were used. We started on a pure IP track applying a classical branch-and-bound technique to solve a set partitioning problem formulation. Since this method requires that all feasible duties are previously inserted into the problem formulation, all memory resources were rapidly consumed when we reached half a million feasible duties. To circumvent this difficulty, we implemented a column generation technique. As suggested in [5], the subproblem of generating feasible duties with negative reduced cost was transformed into a constrained shortest path problem over a directed acyclic graph and then solved using Dynamic Programming techniques. However, due to the size and idiosyncrasies of our problem instances, this technique did not make progress towards solving large instances.

In parallel, we also implemented a heuristic algorithm that produced very good results on large set covering problems [2]. With this implementation, problems with up to two million feasible duties could be solved to optimality. But this particular heuristic also requires that all feasible duties be present in memory during execution. Although some progress with respect to time efficiency was achieved, memory usage was still a formidable obstacle.

The difficulties we faced when using the previous approaches almost disappeared when we turned to a language that supports constraint specification over finite domain variables. We were able to implement our models in little time, producing code that was both concise and clear. When executed, it came as no surprise that the model showed two distinct behaviors, mainly due to the huge size of the search space involved. It was very fast when asked to compute new feasible duties, but lagged behind the IP methods when asked to obtain a provably optimal schedule. The search spaces of our problem instances are enormous and there are no strong local constraints available to help the resolution process. Also a good heuristic to improve the search strategy does not come easily [4].

To harness the capabilities of both the IP and CP techniques, we resorted to a hybrid approach to solve the larger, more realistic, problem instances. The main idea is to use the linear relaxation of a smaller core problem in order to efficiently compute good lower bounds on the optimal solution value. Using the values of dual variables in the solution of the linear relaxation, we can enter the generation phase that computes new feasible duties. This phase is modeled as a constraint satisfaction problem that searches for new feasible duties with a negative reduced cost. This problem is submitted to the constraint solver, which returns new feasible duties to be inserted in the IP problem formulation, and the initial phase can be taken again, restarting the cycle. The absence of new feasible duties with a negative reduced cost proves the optimality of the current formulation. This approach secures the strengths of both the pure IP and the pure CP formulations: only a small subset of all the feasible duties is efficiently dealt with at a time, and new feasible duties are quickly computed only when they will make a difference. The resulting code was

tested on some large instances, based on real data. As of this writing, we can solve, in a reasonable time and with proven optimality, instances with an excess of 150 trips and 12 million feasible duties.

This article is organized as follows. Section 2 describes the crew scheduling problem. In Sect. 3, we discuss the IP approach and report on the implementation of two alternative techniques: standard column generation and heuristics. In Sect. 4, we investigate the pure CP approach. In Sect. 5, we present the hybrid approach. Implementation details and computational results on real data are reported in sections 3, 4 and 5. Finally, in Sect. 6, we conclude and discuss further issues.

In the sequel, execution times inferior to one minute are reported as *ss.cc*, where *ss* denotes seconds and *cc* denotes hundredths of seconds. For execution times that exceed 60 seconds, we use the alternative notation *hh:mm:ss*, where *hh*, *mm* and *ss* represent hours, minutes and seconds, respectively.

2 The Crew Scheduling Problem

In a typical crew scheduling problem, a set of trips has to be assigned to some available crews. The goal is to assign a subset of the trips to each crew in such a way that no trip is left unassigned. As usual, not every possible assignment is allowed since a number of constraints must be observed. Additionally, a cost function has to be minimized.

2.1 Terminology

Among the following terms, some are of general use, while others reflect specifics of the transportation service for the urban area where the input data came from. A *depot* is a location where crews may change buses and rest. The act of driving a bus from one depot to another depot, passing by no intermediate depot, is named a *trip*. Associated with a trip we have its *start time*, its *duration*, its *departure depot*, and its *arrival depot*. The duration of a trip is statistically calculated from field collected data, and depends on many factors, such as the day of the week and the start time of the trip along the day. A *duty* is a sequence of trips that are assigned to the same crew. The *idle time* is the time interval between two consecutive trips in a duty. Whenever this idle time exceeds *Idle_Limit* minutes, it is called a *long rest*. A duty that contains a long rest is called a *two-shift duty*. The *rest time* of a duty is the sum of its idle times, not counting long rests. The parameter *Min_Rest* gives the minimum amount of rest time, in minutes, that each crew is entitled to. The sum of the durations of the trips in a duty is called its *working time*. The sum of the *working time* and the *rest time* gives the *total working time* of a duty. The parameter *Workday* is specified by union regulations and limits the daily total working time.

2.2 Input Data

The input data comes in the form of a two dimensional table where each row represents one trip. For each trip, the table lists: *start time*, measured in minutes after midnight, *duration*, measured in minutes, *initial depot* and *final depot*. We have used data that reflect the operational environment of two bus lines, Line 2222 and Line 3803, that serve the metropolitan area around the city of Belo Horizonte, in central Brazil. Line 2222 has 125 trips and one depot and Line 3803 has 246 trips and two depots. The input data tables for these lines are called OS 2222 and OS 3803, respectively. By considering initial segments taken from these two tables, we derived several other smaller problem instances. For example, taking the first 30 trips of OS 2222 gave us a new 30-trip problem instance. A measure of the number of active trips along a typical day, for both Line 2222 and Line 3803, is shown in Fig. 1. This figure was constructed as follows. For each (x, y) entry, we consider a time window $T = [x, x + \mathit{Workday}]$. The ordinate y indicates how many trips there are with start time s and duration d such that $s \in T$ or $s + d \in T$, i.e., how many trips are active in T.

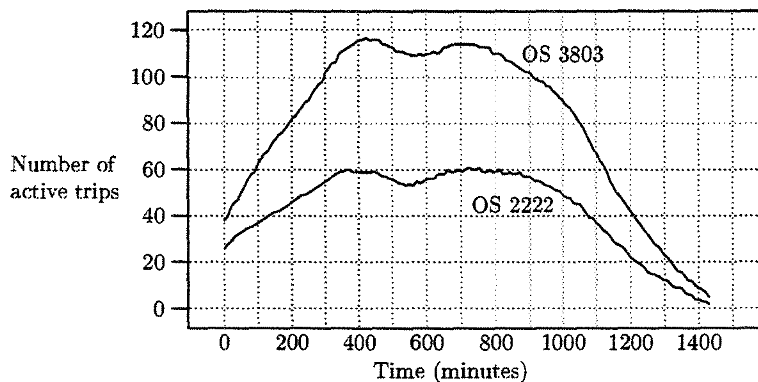


Figure 1: Distribution of trips along the day

2.3 Constraints

For a duty to be feasible, it has to satisfy constraints imposed by labor contracts and union regulations, among others. For each duty we must observe

$$\begin{aligned} \textit{total working time} &\leq \textit{Workday} \\ \textit{rest time} &\geq \textit{Min_Rest}. \end{aligned}$$

In each duty and for each pair (i, j) of consecutive trips, $i < j$, we must have

$$\begin{aligned} (\textit{start time})_i + (\textit{duration})_i &\leq (\textit{start time})_j \\ (\textit{final depot})_i &= (\textit{initial depot})_j . \end{aligned}$$

Also, at most one long rest interval is allowed, in each duty.

Restrictions from the operational environment impose $Idle_Limit = 120$, $Workday = 440$, and $Min_Rest = 30$, measured in minutes. A *feasible duty* is a duty that satisfies all problem constraints. A *schedule* is a set of feasible duties and an *acceptable schedule* is any schedule that partitions the set of all trips. Since the problem specification treats all duties as indistinguishable, every duty is assigned a unit cost. The cost of a schedule is the sum of the costs of all its duties. Hence, minimizing the cost of a schedule is the same as minimizing the number of crews involved in the solution or, equivalently, the number of duties it contains. A *minimal schedule* is any acceptable schedule whose cost is minimal.

3 Mathematical Programming Approaches

Let m be the number of trips and n be the total number of feasible duties. The pure IP formulation of the problem is:

$$\min \sum_{j=1}^{n} x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j = 1, \quad i = 1, 2, \ldots, m \tag{2}$$

$$x_j \in \{0, 1\}, \quad j = 1, 2, \ldots, n . \tag{3}$$

The x_j's are 0–1 decision variables that indicate which duties belong to the solution. The coefficient a_{ij} equals 1 if duty j contains trip i, otherwise, a_{ij} is 0. This is a classical set partitioning problem where the rows represent all trips and the columns represent all feasible duties.

We developed a constraint program to count all feasible duties both in OS 2222 and in OS 3803. Table 1 summarizes the results for increasing initial sections (column "# Trips") of the input data. The time (column "Time") needed to count the number of feasible duties (column "# FD") is also presented. For OS 2222, we get in excess of one million feasible duties, and for all trips in OS 3803 we get more than 122 million feasible duties.

3.1 Pure IP Approach

In the pure IP approach, we used the constraint program to generate an output file containing all feasible duties. A program was developed in C to make this file conform to the

UNICAMP

Table 1: Number of feasible duties for OS 2222 and OS 3803

OS 2222 (1 depot)			OS 3803 (2 depots)		
# Trips	#FD	Time	# Trips	#FD	Time
10	63	0.07	20	978	1.40
20	306	0.33	40	6,705	5.98
30	1,032	0.99	60	45,236	33.19
40	5,191	5.38	80	256,910	00:03:19
50	18,721	21.84	100	1,180,856	00:18:34
60	42,965	00:01:09	120	3,225,072	00:57:53
70	104,771	00:03:10	140	8,082,482	02:59:17
80	212,442	00:05:40	160	18,632,680	08:12:28
90	335,265	00:07:48	180	33,966,710	14:39:21
100	496,970	00:10:49	200	54,365,975	17:55:26
110	706,519	00:14:54	220	83,753,429	42:14:35
125	1,067,406	01:00:27	246	122,775,538	95:49:54

CPLEX[1] input format. The resulting file was fed into a CPLEX 3.0 LP solver. The node selection strategy used was *best-first* and branching was done upon the most fractional variable. Every other setting of the branch-and-bound algorithm used the standard default CPLEX configuration.

The main problem with the IP approach is clear: the number of feasible duties is enormous. Computational results for OS 2222 appear in Table 2, columns under "Pure IP". Columns "Opt" and "Sol" indicate, respectively, the optimal and computed values for the corresponding run. It soon became apparent that the pure IP approach using the CPLEX solver would not be capable of obtaining the optimal solution for the smaller OS 2222 problem instance. Besides, memory usage was also increasing at an alarming pace, and execution time was lagging behind when compared to other approaches that were being developed in parallel. As an alternative, we decided to implement a column generation approach.

3.2 Column Generation with Dynamic Programming

Column generation is a technique that is widely used to handle linear programs which have a very large number of columns in the coefficient matrix. The method works by repeatedly executing two phases. In a first phase, instead of solving a linear relaxation of

[1] CPLEX is a registered trademark of ILOG, Inc.

Table 2: Computational results for OS 2222 (1 depot)

# Trips	#FD	Opt	Pure IP Sol	Pure IP Time	CG+DP Sol	CG+DP Time	Heuristic Sol	Heuristic Time
10	63	7	7	0.02	7	0.01	7	0.05
20	306	11	11	0.03	11	0.07	11	0.30
30	1,032	14	14	0.06	14	0.52	14	10.37
40	5,191	14	14	3.04	14	9.10	14	13.02
50	18,721	14	14	14.29	14	00:01:29	14	00:30:00
60	42,965	14	14	00:01:37	14	00:07:54	14	00:30:22
70	104,771	14	14	00:04:12	14	00:44:19	14	00:03:28
80	212,442	16	16	00:33:52	16	03:53:58	16	00:16:24
90	335,265	18	18	00:50:28	18	08:18:53	18	00:22:42
100	496,970	20	20	02:06:32	20	15:08:55	20	00:50:01
110	706,519	22	-	-	-	-	22	01:06:17
125	1,067,406	25	-	-	-	-	25	01:55:12

the whole problem, in which all columns are required to be loaded in memory, we quickly solve a smaller problem, called the *master* problem, that deals only with a subset of the original columns. That smaller problem solved, we start phase two, looking for columns with a negative reduced cost. If there are no such columns, we have proved that the solution at hand indeed minimizes the objective function. Otherwise, we augment the master problem by bringing in a number of columns with negative reduced cost, and start over on phase one. The problem of computing columns with negative reduced costs is called the *slave* subproblem. When the original variables have integer values, this algorithm must be embedded in a branch-and-bound strategy. The resulting algorithm is also known as *branch-and-price*.

Generating Columns.

In general, the slave subproblem can also be formulated as another IP problem. In our case, constraints like the one on two-shift duties substantially complicate the formulation of a pure IP model. As another approach, Desrochers and Soumis [5] suggest reducing the slave subproblem to a constrained shortest path problem, formulated over a related directed acyclic graph. When this process terminates, it is easy to extract not only the shortest feasible path, but also a number of additional feasible paths, all with negative reduced costs. We used these ideas, complemented by other observations from Beasley and Christofides [1] and our own experience.

Implementation and Results.

To implement the branch-and-price strategy, the use of the ABACUS[2] branch-and-price framework (version 2.2) saved a lot of programming time. One of the important issues was the choice of the branching rule. When applying a branch-and-bound algorithm to set partitioning problems, a simple branching rule is to choose a binary variable and set it to 1 on one branch and set it to 0 on the other branch, although there are situations where this might not be the best choice [13]. This simple branching rule produced a very small number of nodes in the implicit enumeration tree (41 in the worst case). Hence, we judged that any possible marginal gains did not justify the extra programming effort required to implement a more elaborated branching rule (c.f. [12]). In Table 2, columns under "CG+DP", show the computational results for OS 2222. This approach did not reach a satisfactory time performance, mainly because the constrained shortest path subproblem is relatively loose. As a pseudo-polynomial algorithm, the state space at each node has the potential of growing exponentially with the input size. The number of feasible paths the algorithm has to maintain became so large that the time spent looking for columns with negative reduced cost is responsible for more than 97% of the total execution time, on the average, over all instances.

3.3 A Heuristic Approach

Heuristics offer another approach to solve scheduling problems and there are many possible variations. Initially, we set aside those heuristics that were unable to reach an optimal solution. As a promising alternative, we decided to implement the set covering heuristic developed by Caprara et al. [2]. This heuristic won the FASTER competition jointly organized by the Italian Railway Company and AIRO, solving, in reasonable time, large set covering problems arising from crew scheduling. Using our own experience and additional ideas from the chapter on Lagrangian Relaxation in [11], an implementation was written in C and went through a long period of testing and benchmarking. Tests executed on set covering instances coming from the OR-Library showed that our implementation is competitive with the original implementation in terms of solution quality. When this algorithm terminates, it also produces a lower bound for the optimal covering solution, which could be used as a bound for the partition problem, as well. We verified, however, that on the larger instances, the solution produced by the heuristic turned out to be a partition already.

Computational results for OS 2222 appear in Table 2, columns under "Heuristic". Comparing all three implementations, it is clear that the heuristic gave the best results. However, applying this heuristic to the larger OS 3803 data set was problematic. Since storage

[2] `http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html`.

Table 3: Heuristic over OS 3803 (2 depots)

# Trips	# FD	Opt	Sol	Time
20	978	6	6	0.35
40	6,705	13	13	3.60
60	45,236	15	15	52.01
80	256,910	15	15	00:08:11
100	1,180,856	15	15	00:13:51
110	2,015,334	15	15	00:23:24

space has to be allocated to accommodate all feasible columns, memory usage becomes prohibitive. It was possible to solve instances with up to 2 million feasible duties, as indicated in Table 3. Beyond that limit, 320 MB of main memory were not enough for the program to terminate.

4 Constraint Programming Approach

Modeling with finite domain constraints is rapidly gaining acceptance as a promising programming environment to solve large combinatorial problems. This led us to also model the crew scheduling problem using pure constraint programming (CP) techniques. All models described in this section were formulated using the ECL^{i}PS^e [3] syntax, version 4.0. Due to its large size, the ECL^{i}PS^e formulation for each run was obtained using a program generator that we developed in C.

A simple pure CP formulation was developed first. It used a list of items, each item being itself a list describing an actual duty. A number of recursive predicates guarantee that each item satisfies all labor and regulation constraints (see Sect. 2.3), and also enforce restrictions of time and depot compatibility between consecutive trips. These feasibility predicates iterate over all list items. The database contains one fact for each line of input data, as explained in Sect. 2.2. The resulting model is very simple to program in a declarative environment. The formulation, however, did not reach satisfactory results when submitted to the ECL^{i}PS^e solver, as shown in Table 4, columns under "First Model". A number of different labeling techniques, different clause orderings and several variants on constraint representation were explored, to no avail. When proving optimality, the situation was even worse. It was not possible to prove optimality for instances with only 10 trips in less than an hour of execution time. The main reason for this poor performance may reside on the

[3] `http://www.icparc.ic.ac.uk/eclipse`.

recursiveness of the list representation, and on the absence of reasonable lower and upper bounds on the value of the optimal solution which could aid the solver discard unpromising labelings.

4.1 Improved Model

The new model is based on a two dimensional matrix X of integers. The number of columns (rows) in X, $UBdutyLen$ ($UBnumDut$), is an upper bound on the size of any feasible duty (the total number of duties). Each X_{ij} element represents a single trip and is a finite domain variable with domain $[1..NT]$, where $NT = UBdutyLen \times UBnumDut$. Real trips are numbered from 1 to N, where $N \leq NT$. Trips numbered $N+1$ to NT are *dummy trips.* To simplify the writing of some constraints, the last trip in each line of X is always a dummy trip. A proper choice of the start time, duration and depots of the dummy trips avoids time and depots incompatibilities among them and, besides, prevents the occurrence of dummy trips between real trips. Moreover, the choice of start times for all dummy trips guarantees that they occupy consecutive cells at the end of every line in X. Using this representation, the set partitioning condition can be easily met with an `alldifferent` constraint applied to a list that contains X_{ij} elements.

Five other matrices were used: *Start*, *End*, *Dur*, *DepDepot* and *ArrDepot*. Cell (i,j) of these matrices represents, respectively, the start time, the end time, the duration, and the departure and arrival depots of trip X_{ij}. Next, we state constraints in the form `element`$(X_{ij}, S, Start_{ij})$, where S is a list containing the start times of the first NT trips. The semantics of this constraint assures that $Start_{ij}$ is the k-th element of list S where k is the value in X_{ij}. This maintains the desired relationship between matrices X and *Start*. Whenever X_{ij} is updated, $Start_{ij}$ is also modified, and vice-versa. Similar constraints are stated between X and each one of the four other matrices. Now, we can write:

$$End_{ij} \leq Start_{i(j+1)} \quad (4)$$

$$ArrDepot_{ij} + DepDepot_{i(j+1)} \neq 3 \quad (5)$$

$$Idle_{ij} = BD_{ij} \times \left(Start_{i(j+1)} - End_{ij}\right) \quad (6)$$

for all $i \in \{1, \dots, UBnumDut\}$ and all $j \in \{1, \dots, UBdutyLen-1\}$. Equation (4) guarantees that trips overlapping in time are not in the same duty. Since the maximum number of depots is two, an incompatibility of two consecutive trips is prevented by (5). In (6), the binary variables BD_{ij} are such that $BD_{ij} = 1$ if and only if $X_{i(j+1)}$ contains a real trip. Hence, the constraint on total working time, for each duty i, is given by

$$\sum_{j=1}^{UBdutyLen-1} (Dur_{ij} + BI_{ij} \times Idle_{ij}) \leq Workday\,, \quad (7)$$

Table 4: Pure CP models, OS 2222 data set

			First Model		Improved Model			
			Feasible		*Feasible*		*Optimal*	
# Trips	# FD	Opt	Sol	Time	Sol	Time	Sol	Time
10	63	7	7	0.35	7	0.19	7	0.63
20	306	11	11	12.21	11	0.47	11	9.22
30	1,032	14	15	00:02:32	15	0.87	14	00:29:17
40	5,191	14	15	00:14:27	15	0.88	-	> 40:00:00
50	18,721	14	15	00:53:59	15	0.97	-	-
60	42,965	14	-	-	15	2.92	-	-
70	104,771	14	-	-	16	3.77	-	-
80	212,442	16	-	-	19	8.66	-	-
90	335,265	18	-	-	24	17.97	-	-
100	496,970	20	-	-	27	29.94	-	-
110	706,519	22	-	-	27	39.80	-	-
125	1,067,406	25	-	-	32	00:01:21	-	-

where BI_{ij} is a binary variable such that $BI_{ij} = 1$ if and only if $Idle_{ij} \leq Idle_Limit$. The constraint on total rest time is

$$Workday + \sum_{j=1}^{UBdutyLen-1} (Idle_{ij} - Dur_{ij} - BI_{ij} \times Idle_{ij}) \geq Min_Rest \tag{8}$$

for each duty i. For two-shift duties, we impose further that at most one of the $Idle_{ij}$ variables can assume a value greater than *Idle_Limit*.

4.2 Refinements and Results

The execution time of this model was further improved by:

Elimination of Symmetries – Solutions that are permutations of lines of X are equivalent. To bar such equivalences, the first column of the X matrix was kept sorted. Since exchanging the position of dummy trips gives equivalent solutions, new constraints were used to prevent dummy trips from being swapped when backtracking.

Domain Reduction – For instance, the first real trip can only appear in $X_{1,1}$.

Use of Another Viewpoint – Different viewpoints [3] were also used. New Y_k variables were introduced representing "the cell that stores trip k", as opposed to the X_{ij} variables

that mean "the trip that is put in cell ij". The Y_k variables were connected to the X_{ij} variables through *channeling constraints*. The result is a redundant model with improved propagation properties.

Different Labeling Strategies – Various labeling strategies have been tried, including the one developed by Jourdan [9]. The strategy of choosing the next variable to label as the one with the smallest domain (*first-fail*) was the most effective one. After choosing a variable, it is necessary to select a value from its domain following a specific order, when backtracking occurs. We tested different labeling orders, like increasing, decreasing, and also middle-out and its reverse. Experimentation showed that labeling by increasing order achieved the best results. On the other hand, when using viewpoints, the heuristic developed by Jourdan rendered the model roughly 15% faster.

The improved purely declarative model produced feasible schedules in a very good time, as indicated in Table 4, under columns "Improved Model". Obtaining provably optimal solutions, however, was still out of reach for this model. Others have also reported difficulties when trying to solve crew scheduling problems with a pure CP approach [4, 8]. Finding the optimal schedule reduces to choosing, from an extremely large set of elements, a minimal subset that satisfies all the problem constraints. The huge search spaces involved can only be dealt with satisfactorily when pruning is enforced by strong local constraints. Besides, a simple search strategy, lacking good problem specific heuristics, is very unlikely to succeed. When solving scheduling problems of this nature and size to optimality, none of the these requirements can be met easily, rendering it intrinsically difficult for pure CP techniques to produce satisfactory results in these cases.

5 A Hybrid Approach

Recent research [6] has shown that, in some cases, neither the pure IP nor the pure CP approaches are capable of solving certain kinds of combinatorial problems satisfactorily. But a hybrid strategy might outperform them.

When contemplating a hybrid strategy, it is necessary to decide which part of the problem will be handled by a constraint solver, and which part will be dealt with in a classical way. Given the huge number of columns at hand, a column generation approach seemed to be almost mandatory. As reported in Sect. 3.2, we already knew that the dynamic programming column generator used in the pure IP approach did not perform well. On the other hand, a declarative language is particularly suited to express not only the constraints imposed by the original problem, but also the additional constraints that must be satisfied when looking for feasible duties with a negative reduced cost. Given that, it was a natural

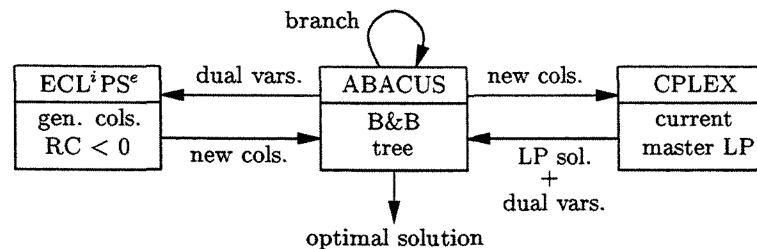


Figure 2: Simplified scheme of the hybrid column generation method

decision to implement a column generation approach where new columns were generated on demand by a constraint program. Additionally, the discussion in Sect. 4.2 indicated that the CP strategy implemented was very efficient when identifying feasible duties. It lagged behind only when computing a provably optimal solution to the original scheduling problem, due to the minimization constraint. Since it is not necessary to find a column with *the* most negative reduced cost, the behavior of the CP solver was deemed adequate. It remained to program the CP solver to find a set of new feasible duties with the extra requirement that their reduced cost should be negative.

5.1 Implementation Issues

The basis of this new algorithm is the same as the one developed for the column generation approach, described in Sect. 3.2. The dynamic programming routine is substituted for an ECL^{i}PS^e process that solves the slave subproblem and uses sockets to communicate the solution back to the ABACUS process. When the ABACUS process has solved the current master problem to optimality, it sends the values of the dual variables to the CP process. If there remain some columns with negative reduced costs, some of them are captured by the CP solver and are sent back to the ABACUS process, and the cycle starts over. If there are no such columns, the LP solver has found an optimal solution. Having found the optimal solution for this node of the enumeration tree, its dual bound has also been determined. The normal branch-and-bound algorithm can then proceed until it is time to solve another LP. This interaction is depicted in Fig. 2.

The code for the CP column generator is almost identical to the code for the improved CP model, presented in Sect. 4.1. There are three major differences. Firstly, the matrix X now has only one row, since we are interested in finding *one* feasible duty and not a complete solution. Secondly, there is an additional constraint stating that the sum of the values of the dual variables associated with the trips in the duty being constructed should represent a negative reduced cost. Finally, the minimization predicate was exchanged for a

Table 5: Hybrid algorithm, OS 2222 data set (1 depot)

# Trips	# FD	Opt	DBR	# CA	# LP	# Nodes	PrT	LPT	TT
10	63	7	7	53	2	1	0.08	0.02	0.12
20	306	11	11	159	4	1	0.30	0.04	0.42
30	1,032	14	14	504	11	1	1.48	0.11	2.07
40	5,191	14	14	1,000	26	13	8.03	0.98	9.37
50	18,721	14	14	1,773	52	31	40.97	3.54	45.28
60	42,965	14	14	4,356	107	41	00:04:24	14.45	00:04:40
70	104,771	14	14	2,615	58	7	00:01:36	4.96	00:01:42
80	212,442	16	16	4,081	92	13	00:01:53	18.84	00:02:13
90	335,265	18	18	6,455	141	11	00:02:47	31.88	00:03:22
100	496,970	20	20	8,104	177	13	00:06:38	51.16	00:07:34
110	706,519	22	22	11,864	262	21	00:16:53	00:02:28	00:19:31
125	1,067,406	25	25	11,264	250	17	00:19:09	00:01:41	00:21:00

predicate that keeps on looking for new feasible duties until the desired number of feasible duties with negative reduced costs have been computed, or until there are no more feasible assignments. By experimenting with the data sets at hand, we determined that the number of columns with negative reduced cost to request at each iteration of the CP solver was best set to 53. The redundant modeling, as well as the heuristic suggested by Jourdan, both used to improve the performance of the original CP formulation, now represented unnecessary overhead, and were removed.

5.2 Computational Results

The hybrid approach was able to construct an optimal solution to substantially larger instances of the problem, in a reasonable time. Computational results for OS 2222 and OS 3803 appear on Tables 5 and 6, respectively. Column headings # Trips, # FD, Opt, DBR, # CA, # LP and # Nodes stand for, respectively, number of trips, number of feasible duties, optimal solution value, dual bound at the root node, number of columns added, number of linear programming relaxations solved, and number nodes visited. The execution times are divided in three columns: PrT, LPT and TT, meaning, respectively, time spent generating columns, time spent solving linear programming relaxations, and total execution time. In every instance, the dual bound at the root node was equal to the value of the optimal integer solution. Hence, the LP relaxation of the problem already provided the best possible lower bound on the optimal solution value. Also note that the number of nodes visited by the algorithm was kept small. The same behavior can be observed with

respect to the number of columns added.

The sizable gain in performance is shown in the last three columns of each table. Note that the time to solve all linear relaxations of the problem was a small fraction of the total running time, for both data sets.

It is also clear, from Table 5, that the hybrid approach was capable of constructing a provably optimal solution for the smaller data set using 21 minutes of running time on a 350 MHz desktop PC. That problem involved in excess of one million feasible columns and was solved considerably faster when compared with the best performer (see Sect. 3.3) among all the previous approaches.

The structural difference between both data sets can be observed by looking at the 100 trip row, in Table 6. The number of feasible duties on this line is, approximately, the same number of one million feasible duties that are present in the totality of 125 trips of the first data set, OS 2222. Yet, the algorithm used roughly twice as much time to construct the optimal solution for the first 100 trips of the larger data set, as it did when taking the 125 trips of the smaller data set. Also, the algorithm lagged behind the heuristic for OS 3803, although the latter was unable to go beyond 110 trips, due to excessive memory usage.

Finally, when we fixed a maximum running time of 24 hours, the algorithm was capable of constructing a solution, and prove its optimality, for as many as 150 trips taken from the larger data set. This corresponds to an excess of 12 million feasible duties. It is noteworthy that less than 60 MB of main memory were needed for this run. A problem instance with as many as $150 \times (12.5 \times 10^6)$ entries would require over 1.8 GB when loaded into main memory. By efficiently dealing with a small subset of the feasible duties, our algorithm managed to surpass the memory bottleneck and solve instances that were very large. This observation supports our view that a CP formulation of column generation was the right approach to solve very large crew scheduling problems.

6 Conclusions and Future Work

Real world crew scheduling problems often give rise to large set covering or set partitioning formulations. We have shown a way to integrate pure Integer Programming and declarative Constraint Satisfaction Programming techniques in a hybrid column generation algorithm that solves, to optimality, huge instances of some real world crew scheduling problems. These problems appeared intractable for both approaches when taken in isolation. Our methodology combines the strengths of both sides, while getting over their main weaknesses.

Another crucial advantage of our hybrid approach over a number of previous attempts is that it considers *all* feasible duties. Therefore, the need does not arise to use specific rules to select, at the start, a subset of "good" feasible duties. This kind of preprocessing could prevent the optimal solution from being found. Instead, our algorithm implicitly looks at the

Table 6: Hybrid algorithm, OS 3803 data set (2 depots)

# Trips	# FD	Opt	DBR	# CA	# LP	# Nodes	PrT	LPT	TT
20	978	6	6	278	7	1	2.11	0.08	2.24
30	2,890	10	10	852	19	1	9.04	0.20	9.38
40	6,705	13	13	2,190	48	1	28.60	1.03	30.14
50	17,334	14	14	4,220	94	3	00:01:22	3.95	00:01:27
60	45,236	15	15	8,027	175	1	00:03:48	14.81	00:04:06
70	107,337	15	15	11,622	258	1	00:07:42	40.59	00:08:37
80	256,910	15	15	8,553	225	1	00:10:07	47.12	00:10:58
90	591,536	15	15	9,827	269	1	00:14:34	00:02:04	00:16:43
100	1,180,856	15	15	13,330	375	1	00:39:03	00:04:37	00:43:49
110	2,015,334	15	15	13,717	387	1	01:19:55	00:03:12	01:23:19
120	3,225,072	16	16	18,095	543	13	04:02:18	00:09:09	04:11:50
130	5,021,936	17	17	28,345	874	23	06:59:53	00:30:16	07:30:56
140	8,082,482	18	18	27,492	886	25	13:29:51	00:28:56	13:59:40
150	12,697,909	19	19	37,764	1,203	25	21:04:28	00:49:13	21:55:25

set of all feasible duties, when activating the column generation method. When declarative constraint satisfaction formulations are applied to generate new feasible duties on demand, they have shown to be a very efficient strategy, in contrast to Dynamic Programming.

We believe that our CP formulation can be further improved. In particular, the search strategy deserves more attention. Earlier identification of unpromising branches in the search tree can reduce the number of backtracks and lead to substantial savings in computational time. Techniques such as dynamic backtracking [7] and the use of *nogoods* [10] can be applied to traverse the search tree more efficiently, thereby avoiding useless work.

References

[1] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[2] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. Technical Report OR-95-8, DEIS, Università di Bologna, 1995.

[3] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 1998. Accepted for publication.

[4] K. Darby-Dowman and J. Little. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal on Computing*, 10(3), 1998.

[5] M. Desrochers and F. Soumis. A column generation approach to the urban transit crew scheduling problem. *Transportation Science*, 23(1), 1989.

[6] C. Gervet. Large Combinatorial Optimization Problems: a Methodology for Hybrid Models and Solutions. In *JFPLC*, 1998.

[7] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, (1):25–46, 1993.

[8] N. Guerinik and M. Van Caneghem. Solving crew scheduling problems by constraint programming. In *Lecture Notes in Computer Science*, pages 481–498, 1995. Proceedings of the First International Conference on the Principles and Practice of Constraint Programming, CP'95.

[9] J. Jourdan. *Concurrent Constraint Multiple Models in CLP and CC Languages: Toward a Programming Methodology by Modeling.* PhD thesis, Université Denis Diderot, Paris VII, February 1995.

[10] J. Lever, M. Wallace, and B. Richards. Constraint logic programming for scheduling and planning. *BT Technical Journal*, (13):73–81, 1995.

[11] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Wiley, 1993.

[12] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport.* North-Holland Publishing Company, 1981.

[13] F. Vanderbeck. *Decomposition and Column Generation for Integer Programming.* PhD thesis, Université Catholique de Louvain, CORE, September 1994.

Epílogo

A partir dos resultados expostos nesse artigo, é possível perceber que a grande dificuldade em se encontrar soluções ótimas para instâncias reais do problema de *scheduling* reside no tamanho dos modelos resultantes. Fica claro, a princípio, que é inviável lidar com todas as colunas (jornadas viáveis) ao mesmo tempo, sendo mister o uso de um método de geração de colunas. Todavia, uma das tentativas mais tradicionais de *pricing*, baseada num algoritmo de caminhos mais curtos em um grafo com restrições de recursos, transformou-se no gargalo do algoritmo em termos de tempo computacional.

Por outro lado, percebeu-se que o modelo de Programação por Restrições, apesar de ser ineficiente na busca de soluções ótimas para o problema, era capaz de encontrar jornadas viáveis com grande rapidez. Decidiu-se, portanto, combinar essas duas idéias, substituindo-se o algoritmo de programação dinâmica pelo modelo de Programação por Restrições, ligeiramente modificado. Os resultados obtidos com o algoritmo híbrido de *Branch-and-Price* foram altamente satisfatórios e superaram em muito as expectativas iniciais.

Comparações entre os algoritmos implementados para este problema, em termos de gráficos, aparecem nas figuras a seguir. A Figura 4.1 exibe os resultados obtidos com os algoritmos de *Branch-and-Bound* (PLI Puro), *Branch-and-Price* com Programação Dinâmica (GC+PD), Programação por Restrições (PR Puro) e com a heurística Lagrangeana (CFT). As Figuras 4.2 e 4.3 comparam o desempenho do algoritmo híbrido com o algoritmo de *Branch-and-Bound* para as linhas de ônibus com um e dois pontos, respectivamente.

Muitos dos detalhes a respeito do algoritmo híbrido foram omitidos neste capítulo. O Capítulo 5, a seguir, expõe mais a fundo algumas das idiossincrasias desse algoritmo.

Para que se tenha uma idéia melhor quanto à forma de uma solução do problema de *crew scheduling*, a Figura 4.4 exibe o escalonamento construído pelo algoritmo híbrido para a instância de 125 viagens da OS 2222. No eixo horizontal estão os horários de início das viagens, numa escala dividida em intervalos de duas horas. No eixo vertical estão os números que identificam as duplas de funcionários, de 1 a 25. Cada viagem aparece na forma de um retângulo numerado, cuja largura é proporcional à duração da viagem.

A Figura 4.5 mostra a distribuição de carga de trabalho para cada dupla de funcionários, obtida a partir do escalonamento apresentado na Figura 4.4. Apesar de o modelo não incluir explicitamente uma restrição exigindo que a carga de trabalho esteja balanceada entre os funcionários, é desejável que a solução do problema apresente essa característica. Isto porque todos os funcionários recebem o mesmo salário. A solução obtida com o algoritmo híbrido, além de possuir um custo mínimo, também se mostrou bastante satisfatória nesse aspecto.

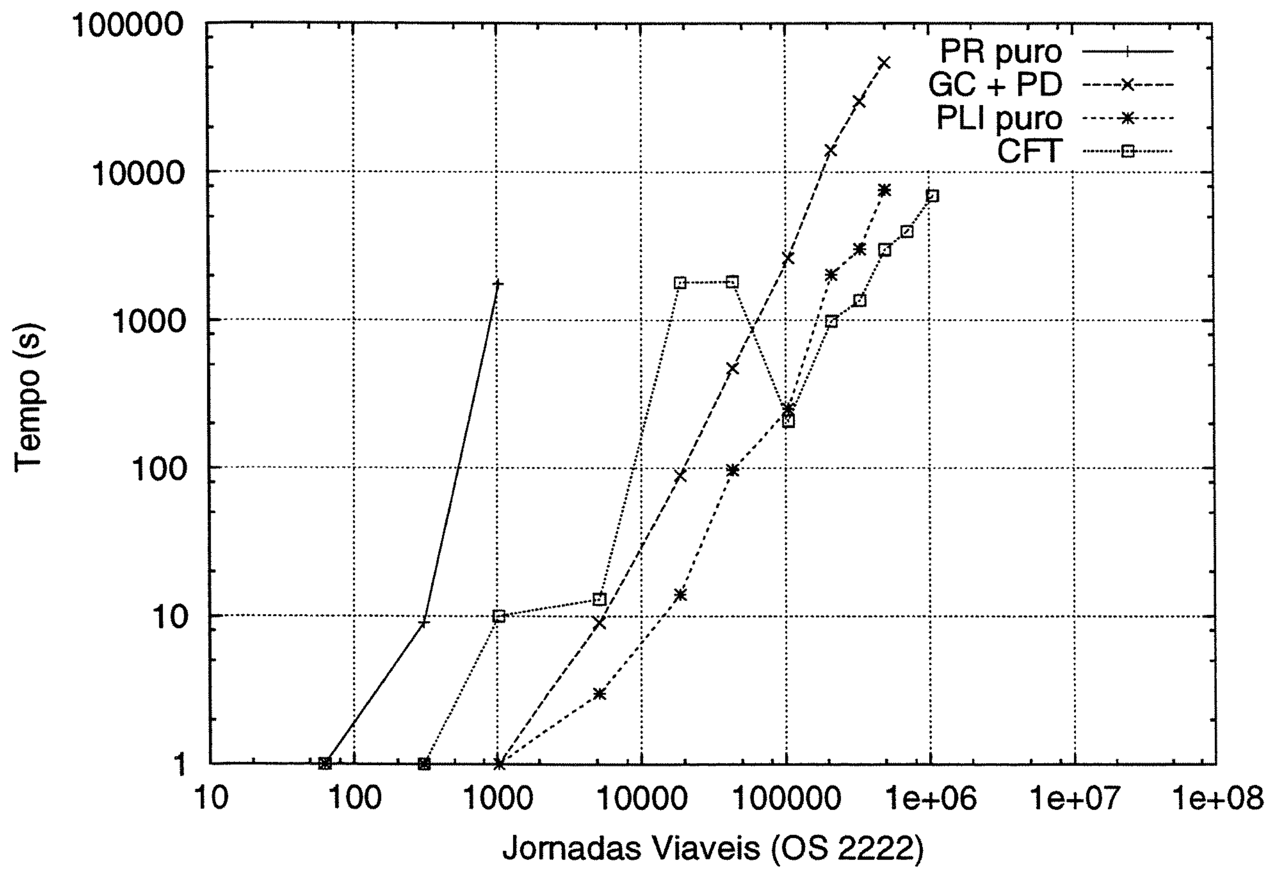


Figura 4.1: Comparação de desempenho entre as abordagens isoladas

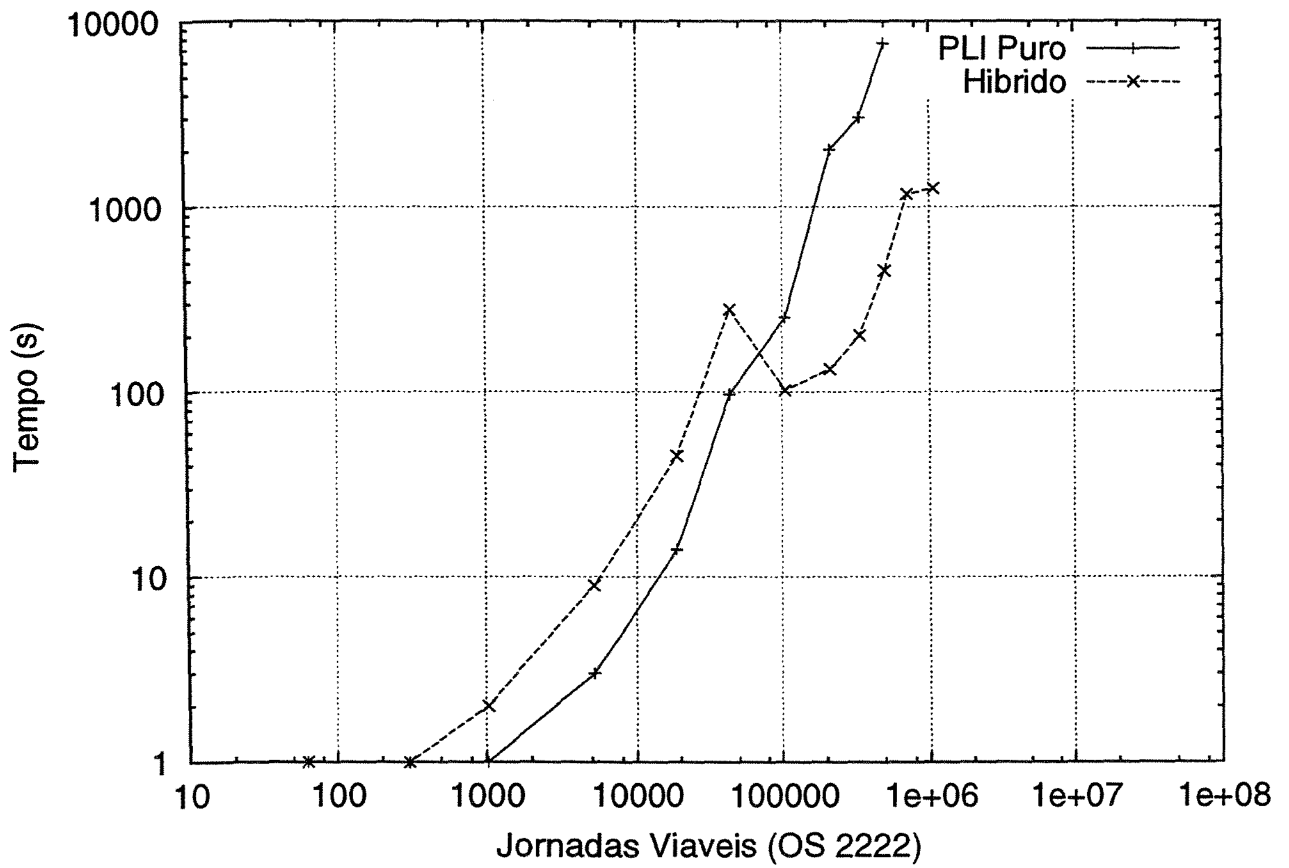


Figura 4.2: Algoritmo híbrido *versus* *Branch-and-Bound* sobre a OS 2222

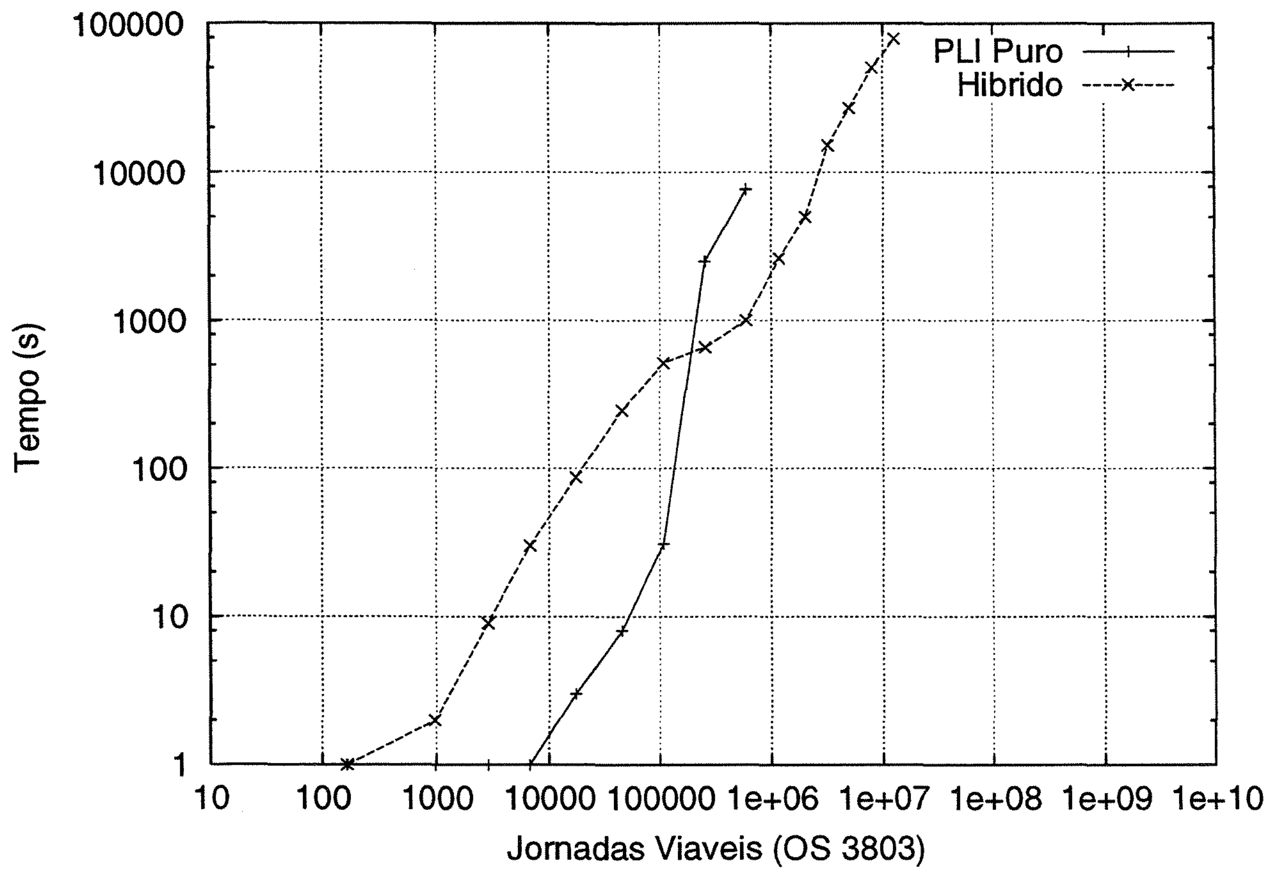


Figura 4.3: Algoritmo híbrido *versus* *Branch-and-Bound* sobre a OS 3803

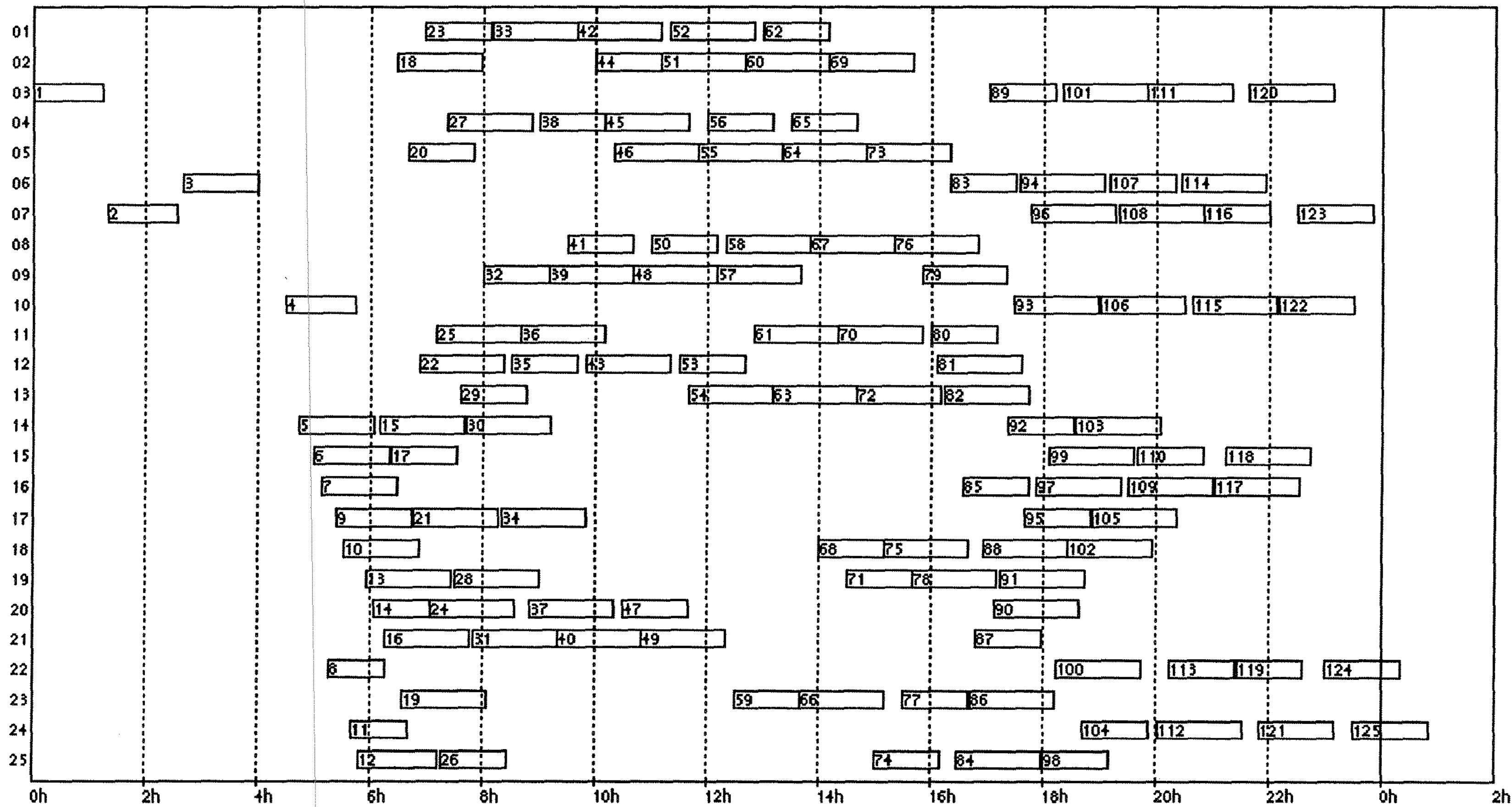


Figura 4.4: Escalonamento final para a OS 2222

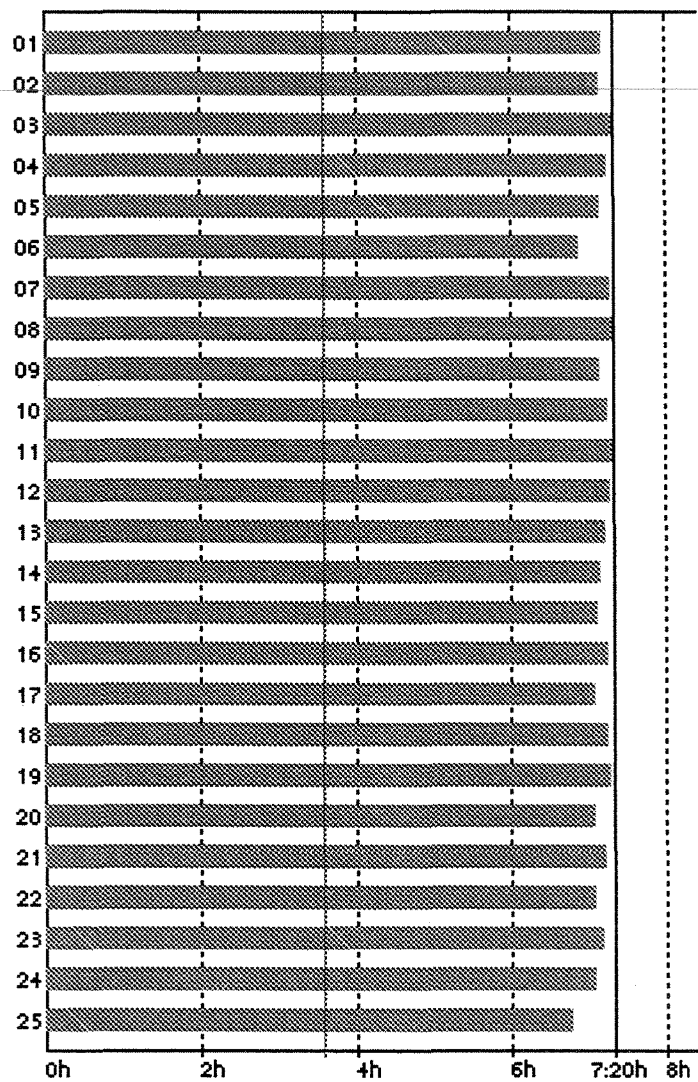


Figura 4.5: Carga de trabalho para um escalonamento ótimo da OS 2222

Capítulo 5

O Algoritmo Híbrido para *Crew Scheduling*

Prólogo

No capítulo anterior, discutiram-se várias possibilidades de solução para o problema de *crew scheduling*. As estratégias isoladas não obtiveram sucesso, sendo significativamente superadas por um algoritmo híbrido de *Branch-and-Price*. Este artigo focaliza este algoritmo, incluindo uma descrição mais detalhada sobre a sua implementação. Para o leitor já familiarizado com o conteúdo do artigo do Capítulo 4, é possível restringir a leitura deste artigo às Seções 3, 4 e 5.

Este artigo foi submetido para a conferência *14th ACM Symposium on Applied Computing*, que ocorreu na cidade de Como, Itália, no período de 19 a 21 de março de 2000. O trabalho foi aceito e publicado num volume dos *ACM Proceedings* contendo os anais da conferência [49].

Solving Very Large Crew Scheduling Problems to Optimality

Tallys H. Yunes* tallys@acm.org

Arnaldo V. Moura arnaldo@dcc.unicamp.br

Cid C. de Souza† cid@dcc.unicamp.br

Abstract

In this article, we present a hybrid methodology for the exact solution of large scale real world crew scheduling problems. Our approach integrates mathematical programming and constraint satisfaction techniques, taking advantage of their particular abilities in modeling and solving specific parts of the problem. An Integer Programming framework was responsible for guiding the overall search process and for obtaining lower bounds on the value of the optimal solution. Complex constraints were easily expressed, in a declarative way, using a Constraint Logic Programming language. Moreover, with an effective constraint-based model, the huge space of feasible solutions could be implicitly considered in a fairly efficient way. Our code was tested on real problem instances arising from the daily operation of an ordinary urban transit company that serves a major metropolitan area with an excess of two million inhabitants. Using a typical desktop PC, we were able find, in an acceptable running time, an optimal solution to instances with more than 1.5 billion entries.

1 Introduction

Crew scheduling problems have their great practical importance based on the fact that, in most companies, employee related expenses may rise to a very significant portion of the total expenditures. Therefore, these notoriously difficult combinatorial optimization problems deserve a great deal of attention. In this article, we present a hybrid strategy that is capable of efficiently obtaining provably optimal solutions for some large instances of specific crew scheduling problems. These instances stem from the operational environment of a mass transit company that serves the metropolitan area of the city of Belo Horizonte, in Brazil.

Previous attempts to solve similar crew scheduling problems to optimality, either with Integer Programming (IP) or with Constraint Programming (CP) techniques alone, faced difficulties when handling large scale instances due to a series of factors [2, 6, 8].

*Supported by FAPESP grant 98/05999-4, and CAPES.

†Supported by FINEP (ProNEx 107/97), and CNPq (300883/94-3).

In order to combine the capabilities of both the IP and CP techniques, we developed a hybrid approach to solve larger, more realistic, problem instances. The main idea is to use the linear relaxation of a smaller core problem in order to efficiently compute good lower bounds on the optimal solution value. Using the values of the dual variables in the solution of the linear relaxation, we can enter a column generation phase that computes new feasible duties. This phase is modeled as a constraint satisfaction problem which is then submitted to a constraint solver. The solver returns new feasible duties to be inserted in the IP problem formulation, and the initial phase can be taken again, restarting the cycle. This approach secures the strengths of both the pure IP and the pure CP formulations: only a small subset of all the feasible duties is efficiently dealt with at a time, and new feasible duties are quickly computed only when they will make a difference. The resulting code was tested on some large instances, based on real data. As of this writing, we can solve, in a reasonable time and with proven optimality, instances with an excess of 150 trips and 12 million feasible duties. The resulting code was compiled under the Linux operating system, kernel 2.0, and ran on a 350 MHz desktop PC with 320 MB of main memory.

This article is organized as follows. Section 2 describes the crew scheduling problem. In Section 3, we discuss some difficulties faced by the pure IP and CP approaches. In Section 4, we present the hybrid approach together with some implementation details, and also report on computational results based on real data. Finally, in Section 5, we conclude and discuss further issues.

2 The Crew Scheduling Problem

In a typical crew scheduling problem, a set of trips has to be assigned to some available crews. The goal is to assign a subset of the trips to each crew in such a way that no trip is left unassigned. As usual, not every possible assignment is allowed since a number of constraints must be observed. Additionally, a cost function has to be minimized.

2.1 Terminology

Among the following terms, some are of general use, while others reflect specifics of the transportation service for the urban area where the input data came from. A *depot* is a location where crews may change buses and rest. The act of driving a bus from one-depot to another depot, passing by no intermediate depots, is named a *trip*. Associated with a trip we have its *start time*, its *duration*, its *initial depot*, and its *final depot*. The duration of a trip is statistically calculated from field collected data, and depends on many factors, such as the day of the week and the start time of the trip along the day. A *duty* is a sequence of trips that are assigned to the same crew. Any time interval between two consecutive trips

in a duty is called an *idle interval.* Whenever this idle interval exceeds *Idle_Limit* minutes, it is called a *long rest.* During a long rest, crews can leave the premises and return later to resume their shift. A duty that contains a long rest is called a *split-shift duty.* The *rest time* of a duty is the sum of its idle intervals, not counting long rests. The parameter *Min_Rest* gives the minimum amount of rest time, in minutes, that each crew is entitled to. The sum of the durations of the trips in a duty is called its *working time.* The sum of the *working time* and the *rest time* gives the *total working time* of a duty. The time, in minutes, that a crew member works in excess of *Workday* minutes is called *overtime* and is given by $\max\{0, \textit{total working time} - \textit{Workday}\}$. The *Workday* is a given parameter, specified by union regulations, that bounds the maximum time that an employee can work without incurring in overtime. An upper bound on *overtime* is established by the parameter *Max_Overtime.* Finally, the *maximum working time* is given by *Workday* + *Max_Overtime.*

2.2 Input Data

The input data comes in the form of a two dimensional table where each row represents one trip. For each trip, the table lists four columns with information about this trip: *start time*, measured in minutes after midnight, *duration*, measured in minutes, *initial depot* and *final depot.* We have used data that reflect the operational environment of two bus lines, Line 2222 and Line 3803, that serve a major metropolitan area. Line 2222 has 125 trips and one depot and Line 3803 has 246 trips and two depots. The input data tables for these lines are called OS 2222 and OS 3803, respectively. By considering initial segments taken from these two tables, we derived several other smaller problem instances. For example, taking the first 30 trips of OS 2222 gave us a new 30-trip problem instance. Table 1(a) shows the first 10 rows of OS 3803. A measure of the number of active trips along a typical day, for both Line 2222 and Line 3803, is shown in Table 1(b). This graphic was constructed as follows. For each (x, y) entry, we consider a time window $T = [x, x + \textit{Workday}]$. The ordinate y indicates how many trips there are with start time s and duration d such that $s \in T$ or $s + d \in T$. The particular shapes of these two curves represent a typical daily workload in the operation of an urban bus company in Brazil.

2.3 Constraints

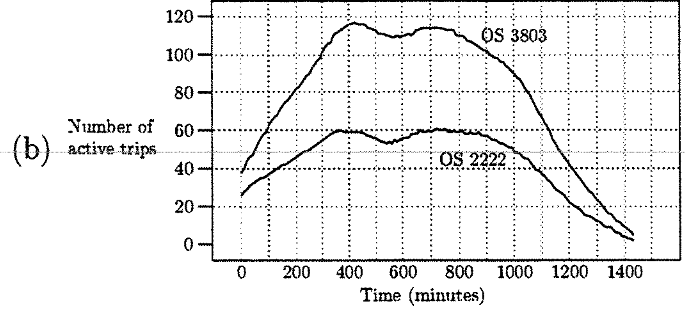
For a duty to be classified as feasible, it has to satisfy many constraints imposed by labor contracts and union regulations, among others. The most important constraints are, for every duty:

Table 1: (a) Sample from OS 3803 (b) Distribution of trips along the day

(a)

Start	Dur	I. dep.	F. dep.
1	38	1	2
50	40	2	1
90	38	1	2
130	38	2	1
170	38	1	2
210	38	2	1
250	39	1	2
290	38	2	1
285	45	1	2
335	45	2	1

(b)



i. For each pair of consecutive trips, i and j:
   (i) $(start\ time)_i + (duration)_i \leq (start\ time)_j$
   (ii) $(final\ depot)_i = (initial\ depot)_j$

ii. *total working time* $\leq$ *maximum working time*;

iii. *rest time* $+ \max\{0,$ *Workday* $-$ *total working time*$\} \geq$ *Min_Rest*; and

iv. At most one long rest interval is allowed;

Due to union regulations and operational constraints, the following values were used in our experiments: *Idle_Limit* $= 120$, *Workday* $= 440$, *Min_Rest* $= 30$ and *Max_Overtime* $= 0$, measured in minutes. A duty which satisfies all problem constraints is called a *feasible duty*. Any set of feasible duties constitutes a *schedule* and for a schedule to be *acceptable* it must partition the set of trips. The cost of a schedule is the sum of the costs of all its duties. As we are interested in minimizing the number of crews needed to operate the bus line, all duties are treated equally and their costs are set to one. With this assumption, minimizing the cost of a schedule reduces to minimizing the number duties (crews) in the solution. Finally, a *minimal schedule* is any acceptable schedule with minimum cost.

3 Pure Approaches

The crew scheduling problem, as described here, presents a classical difficulty: finding the optimal schedule reduces to choosing from an extremely large set F of feasible duties, a minimal subset that partitions the set of trips to be serviced. The very large size of F poses

serious problems, since a provably optimal solution can only be found if all elements of F are considered, either implicitly or explicitly.

When adopting IP approaches to solve this problem, one usually ends up with a set partitioning formulation where the elements of F constitute the columns of the coefficient matrix. The pure IP formulation of the problem is:

$$\min \sum_{j=1}^{n} x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j = 1, \quad i = 1, 2, \ldots, m \tag{2}$$

$$x_j \in \{0, 1\}, \quad j = 1, 2, \ldots, n \tag{3}$$

where m is the total number of trips and n is the size of F. The x_j's are 0-1 decision variables that indicate which duties belong to the solution. The coefficient a_{ij} equals 1 if duty j contains trip i, otherwise, a_{ij} is 0.

Due to the size of F, an implicit way to treat the set of feasible duties is needed. Column generation [1] is a technique that is widely used to handle linear programs which have a very large number of columns in the coefficient matrix. The method works by repeatedly executing two phases. In a first phase, instead of solving a linear relaxation of the whole problem, in which all columns are required to be loaded in memory, we quickly solve a smaller problem. This problem is called the *master* problem, and deals only with a subset of the original columns. That smaller problem solved, we start phase two, looking for columns with a negative reduced cost. If there are no such columns, we have proved that the solution at hand indeed minimizes the objective function. Otherwise, we augment the master problem by bringing in a number of columns with negative reduced costs, and start over on phase one. From the IP formulation above, the reduced cost of a feasible duty d is given by $1 - \sum_{j \in T} u_j$, where T is the set of trips serviced by d and u_j is the value of the dual variable associated with trip j. The problem of computing columns with negative reduced costs is called the *slave* subproblem. When the original variables are restricted to integer values, this algorithm must be embedded in a branch-and-bound strategy. The resulting algorithm is usually referred to as *branch-and-price*. The slave subproblem can be modeled as a constrained shortest path problem over a directed acyclic graph and it can be solved by a dynamic programming algorithm [3]. Nevertheless, the pseudo-polynomial algorithms that arise from this strategy turn out to be very time consuming. This is mainly because of the looseness of our problem constraints [8].

Difficulties also arise when trying to solve crew scheduling problems using pure constraint satisfaction techniques [2, 6, 8]. The huge search space can only be dealt with efficiently when pruning is enforced by strong local constraints. Besides, a simple search

strategy, lacking good problem specific heuristics, is very unlikely to succeed. When solving scheduling problems of this nature to optimality, none of the these requirements can be easily met, rendering it intrinsically difficult for pure CP techniques to produce satisfactory results in these cases.

4 A Hybrid Approach

Recent research [4] has shown that, in some cases, neither the pure IP nor the pure declarative CP approaches are capable of solving certain kinds of combinatorial problems satisfactorily. But a hybrid strategy may outperform these two methods. When contemplating a hybrid strategy, it is necessary to to decide which part of the problem will be handled by a constraint solver, and which part will be dealt with in a classical way. Given the huge number of columns at hand, the use of a column generation approach seemed to be almost mandatory.

A declarative constraint language is particularly suited to express the feasibility constraints imposed by the original problem in a clear and concise way. Furthermore, a constraint model of the original problem can also be easily turned into an efficient column generator by adding one extra constraint to the model: the generated duties must have a negative reduced cost so as to improve the objective function value [1]. Moreover, when looking for new feasible duties (columns) to enter the basis in the current set partitioning formulation, it is not necessary to find the one with *the* most negative reduced cost. This removes the minimization constraint from the formulation, rendering it much more efficient. Our hybrid strategy implemented a column generation approach where new columns were generated on demand by a declarative constraint program.

We decided to use the ABACUS[1] branch-and-price framework in order to save programming time. The Linear Programming relaxations of the set partitioning formulation of the problem are solved inside ABACUS with the help of a CPLEX[2] 3.0 LP solver. When the ABACUS process has solved the current master problem to optimality, it sends the values of the dual variables to the constraint solver, together with the number of columns with negative reduced costs it would like to get. If there remained some such columns, a subset of them is captured by the CP solver and sent back to the ABACUS process, and the cycle starts over. If there are no such columns, the LP solver has found an optimal solution. Having found the optimal solution for the current node of the enumeration tree, its dual bound has also been determined. The normal branch-and-bound algorithm can then proceed until it is time to solve another LP at a different node of the implicit enumeration tree. By experimenting with the data sets at hand, we determined that the number of columns

[1] `http://www.informatik.uni-koeln.de/ls_juenger/`

[2] CPLEX is a registered trademark of ILOG, Inc.

with negative reduced cost to request at each call of the CP solver was best set to 53.

4.1 The Column Generator

Modeling with finite domain constraints is rapidly gaining acceptance as a promising declarative programming environment to solve large combinatorial problems. All models described in this section were formulated using the ECL^{i}PS^e[3] syntax, version 4.0. Due to its large size, the ECL^{i}PS^e formulation for each run was produced by a program generator that we developed in C.

When creating the constraint-based column generator, our aim was to facilitate the direct representation of algebraic constraints. The model is based on a vector X of integers. The number of elements in X is an upper bound on the size of any feasible duty (*UBdutyLen*). To calculate *UBdutyLen*, we start by summing up the durations of the trips, taken in non-decreasing order. When we reach a value that is greater than *maximum working time* minutes, *UBdutyLen* is set to the number of trips used in the sum. Each X_i element, called a *cell*, represents a single trip and is a finite domain variable with domain $[1..NT]$, where $NT = N + UBdutyLen - 1$ and N is the number of real trips. Trips numbered $N + 1$ to NT are *dummy trips*. The start time of the first dummy trip equals the arrival time of the last real trip plus one minute and its duration is zero minutes. All the subsequent dummy trips also last zero minutes and their start times are such that there is a one minute idle interval between consecutive dummy trips, i.e., they start at each following minute. Their departure and arrival depots are equal to 0. These choices prevent incompatibilities arising from time intersection and mismatching of depots among the dummy trips. Besides, we avoid the occurrence of dummy trips between real trips in a feasible duty.

Note that, the way *UBdutyLen* is calculated assures that at least one dummy trip appears in X. Moreover, their start times guarantee that the dummy trips occupy consecutive cells at the end of X. This is on purpose, to facilitate the representation of some constraints.

Five additional vectors were used: *Start*, *End*, *Dur*, *DepDepot* and *ArrDepot*. The i-th cell of these vectors represents, respectively, the start time, the end time, the duration, and the departure and arrival depots of the trip assigned to X_i. Next, we state constraints of type `element`$(X_i, S, Start_i)$, where S is a list containing the start times of all NT trips. The semantics of this constraint assures that the value of $Start_i$ is the k-th element of list S where k is the value in X_i. This maintains the desired relationship between vectors X and *Start*. Whenever X_i is updated, e.g. due to constraint propagation, $Start_i$ is also modified, and vice-versa. Similar constraints are stated between X and each one of the four other

[3] `http://www.icparc.ic.ac.uk/eclipse`

vectors. Now, we can use these new vectors to easily state additional constraints, like:

$$End_i \leq Start_{i+1} \tag{4}$$
$$ArrDepot_i + DepDepot_{i+1} \neq 3 \tag{5}$$
$$Idle_i = BD_i \times (Start_{i+1} - End_i) \tag{6}$$

for all $i \in \{1, \ldots, UBdutyLen - 1\}$. Equation (4) guarantees that trips overlapping in time are not in the same duty. Since the maximum number of depots is two, an incompatibility between consecutive trips occurs only when the ending depot is 1 and the starting depot is 2, or vice-versa. Equation (5) forbids that situation. Additionally, the consecutiveness of two dummy trips is permitted (for the sum of their depots equals 0) and the appearance of the first dummy trip after the last real trip in a duty is not precluded by this constraint, because the sum of the depots in this case can only assume the values 1 or 2. With a one-depot instance, these constraints are not necessary and they are omitted, together with the *ArrDepot* and *DepDepot* vectors. Some other constraints are expressed using the $Idle_i$ variables of Equation (6). The binary variables BD_i, in (6), are such that $BD_i = 1$ if and only if X_{i+1} contains a real trip.

The constraint on total working time, for each generated duty, is given by:

$$TWT = \sum_{i=1}^{UBdutyLen-1} Dur_i + \sum_{i=1}^{UBdutyLen-1} BI_i \times Idle_i \tag{7}$$
$$TWT \leq \textit{maximum working time} \tag{8}$$

where BI_i is a binary variable such that $BI_i = 1$ if and only if $Idle_i \leq Idle_Limit$.

The constraint on total rest time is:

$$\sum_{i=1}^{UBdutyLen-1} Idle_i + \max\{0, Workday - TWT\} \geq Min_Rest\,. \tag{9}$$

To respect the constraint on split-shift duties, we impose that at most one of the $Idle_i$ variables can assume a value greater than *Idle_Limit*. This is done with an `atmost` constraint in the following manner: `atmost(1,` L`, 0)`. If list L contains all the BI_i variables of Equation (7), this means that at most one of them can assume the value zero.

Changing dummy trips in a feasible duty gives another duty that is equivalent to the original one. New constraints were imposed over the X cells in order to force dummy trips to have only one possible placement in X, given that the real trips had already been positioned. This is achieved with the following constraints, for all $i \in \{1, \ldots, UBdutyLen - 1\}$, where N is the number of real trips:

$$X_i \leq N \Rightarrow (X_{i+1} > N \Rightarrow X_{i+1} = N + 1) \tag{10}$$
$$X_i > N \Rightarrow X_{i+1} = X_i + 1\,. \tag{11}$$

There is one final constraint, which is responsible for assuring that generated duties have an associated negative reduced cost. Using the formula to calculate the reduced cost of a column (feasible duty) given in Section 3, this constraint reads:

$$\sum_{i=1}^{UBdutyLen} C_i > 1\,. \qquad (12)$$

For each i, C_i is determined by `element`(X_i, V, C_i), where V is a list whose elements are the values of the dual variables associated with each trip. The dual variables associated with dummy trips are assigned the value zero.

Various labeling strategies have been tried. The strategy of choosing the next variable to label as the one with the smallest domain (*first-fail* principle) proved to be the most effective one, after a number of experimental trials. Having chosen a variable, it is necessary to select a value from its domain following a specific order, when backtracking occurs. We tested different labeling orders, like increasing, decreasing, and also middle-out and its reverse. Experimentation showed that labeling by increasing order produced the best results. It would be possible to improve the overall performance of this search strategy by experimenting with more sophisticated techniques. Dynamic backtracking [5] and *nogood* assertions [7] could be used in this model so as to promote an earlier pruning of unpromising branches of the search tree.

A crucial advantage of this hybrid approach over a number of previous attempts is that it considers *all* feasible duties. Therefore, it is not necessary to use specific rules to select, at the beginning, a subset of "good" feasible duties. This kind of preprocessing could prevent the optimal solution from being encountered. Instead, with the column generation method, our algorithm implicitly looks at the set of all feasible duties.

4.2 Computational Results

In this section, execution times inferior to one minute are reported as *ss.cc*, where *ss* denotes seconds and *cc* denotes hundredths of seconds. When execution times exceed 60 seconds, we use the alternative notation *hh*:*mm*:*ss*, where *hh*, *mm* and *ss* represent, respectively, hours, minutes and seconds.

When compared to the pure approaches, the hybrid approach was able to construct an optimal solution to substantially large instances of the problem, in a reasonable time. Computational results for OS 2222 and OS 3803 appear on Tables 2 and 3, respectively. Column headings have the following meanings: #Trips is the number of trips; #FD stands for the number of feasible duties; Opt is the value of the optimal integer solution; DBR is the dual bound at the root node of the branch-and-bound enumeration tree; #CA is the number of columns added throughout each execution; #LP is the number of linear programming

Table 2: OS 2222 data set (1 depot)

#Trips	#FD	Opt	DBR	#CA	#LP	#Nodes	PrT	LPT	TT
10	63	7	7	53	2	1	0.08	0.02	0.12
20	306	11	11	159	4	1	0.30	0.04	0.42
30	1,032	14	14	504	11	1	1.48	0.11	2.07
40	5,191	14	14	1,000	26	13	8.03	0.98	9.37
50	18,721	14	14	1,773	52	31	40.97	3.54	45.28
60	42,965	14	14	4,356	107	41	00:04:24	14.45	00:04:40
70	104,771	14	14	2,615	58	7	00:01:36	4.96	00:01:42
80	212,442	16	16	4,081	92	13	00:01:53	18.84	00:02:13
90	335,265	18	18	6,455	141	11	00:02:47	31.88	00:03:22
100	496,970	20	20	8,104	177	13	00:06:38	51.16	00:07:34
110	706,519	22	22	11,864	262	21	00:16:53	00:02:28	00:19:31
125	1,067,406	25	25	11,264	250	17	00:19:09	00:01:41	00:21:00

Table 3: OS 3803 data set (2 depots)

#Trips	#FD	Opt	DBR	#CA	#LP	#Nodes	PrT	LPT	TT
20	978	6	6	278	7	1	2.11	0.08	2.24
30	2,890	10	10	852	19	1	9.04	0.20	9.38
40	6,705	13	13	2,190	48	1	28.60	1.03	30.14
50	17,334	14	14	4,220	94	3	00:01:22	3.95	00:01:27
60	45,236	15	15	8,027	175	1	00:03:48	14.81	00:04:06
70	107,337	15	15	11,622	258	1	00:07:42	40.59	00:08:37
80	256,910	15	15	8,553	225	1	00:10:07	47.12	00:10:58
90	591,536	15	15	9,827	269	1	00:14:34	00:02:04	00:16:43
100	1,180,856	15	15	13,330	375	1	00:39:03	00:04:37	00:43:49
110	2,015,334	15	15	13,717	387	1	01:19:55	00:03:12	01:23:19
120	3,225,072	16	16	18,095	543	13	04:02:18	00:09:09	04:11:50
130	5,021,936	17	17	28,345	874	23	06:59:53	00:30:16	07:30:56
140	8,082,482	18	18	27,492	886	25	13:29:51	00:28:56	13:59:40
150	12,697,909	19	19	37,764	1,203	25	21:04:28	00:49:13	21:55:25

relaxations solved; and #Nodes is the number of tree nodes visited. The execution times are divided in three columns: PrT is the time spent generating columns; LPT is the time spent solving linear programming relaxations and TT is the total execution time. Note that, in every instance tested, the dual bound at the root node was equal to the value of the optimal integer solution. Hence, the LP relaxation of the problem already provided the best possible lower bound on the optimal solution value. Also note that the number of nodes visited by the algorithm was kept small. The same behavior can be observed with respect to the number of added columns.

It is interesting to note, in the last three columns of each table, that the time taken to solve all linear relaxations of the problem was a small fraction of the total running time for both data sets.

From Table 2, we see that the hybrid approach was capable of constructing a provably optimal solution for the complete smaller data set using 21 minutes of running time on a 350 MHz desktop PC. That problem involves in excess of one million feasible duties.

The structural difference between both data sets can be appreciated by observing the entries on the line associated with 100 trips, in Table 3. The number of feasible duties on this line corresponds, approximately, to the same number of feasible duties that are present in the totality of 125 trips of the first data set, OS 2222. Yet, the algorithm used roughly twice as much time to construct the optimal solution when running over a test case that contained the first 100 trips of the larger data set, as it did when taking the 125 trips of the smaller data set. The existence of two depots in OS 3803, rather than a single one in OS 2222, seems to be at the origin of the increased problem complexity of this instance.

Finally, when we fixed a maximum running time of 24 hours, the algorithm was capable of constructing a solution, and prove its optimality, for as many as 150 trips taken from the larger data set. This corresponds to an excess of 12 million feasible duties. It is noteworthy that less than 60 MB of main memory were needed for this run to reach completion. A problem instance with as many as $150 \times (12.5 \times 10^6)$ entries would require over 1.8 GB of main memory, if needed to be loaded into main memory. By efficiently dealing with only a small subset of the feasible duties, our algorithm managed to surpass the resource consumption bottlenecks faced by the pure approaches and could solve instances that were very large. This observation supports our view that a declarative constraint formulation of column generation, embedded inside a branch-and-bound framework, was the right technique to solve these very large crew scheduling problems.

5 Conclusions and Future Work

We have integrated pure Constraint Programming and pure Integer Programming techniques in a hybrid column generation algorithm that is able to solve very large instances of

some real world crew scheduling problems to optimality. These problems appear intractable for both approaches when taken in isolation. Our methodology combines the strengths of both methods, getting over their main weaknesses.

Complex operational constraints could easily be expressed using declarative statements in a constraint programming language. The resulting model proved to be very efficient when looking for feasible duties with a negative reduced cost. A linear programming relaxation of the original problem formulation produced good lower bounds. Combining these two behaviors, it was possible to develop a very successful branch-and-price scheme. Using this hybrid method, a desktop PC was able to find optimal solutions for large one-depot instances in a reasonable time. The two-depot case, however, having a different structure and a larger search space, presented more difficulties. When we restricted the problem to the first 150 trips of the day, which already correspond to 12 million feasible duties, the hybrid algorithm produced an optimal solution. Nevertheless, even increasing the computation time limit to 24 hours, the hybrid algorithm was not able to compute an optimal schedule to the complete set of trips, which contained an excess of 122 million feasible duties. Further investigation is needed in order to devise more effective strategies to deal with such large two-depot instances.

References

[1] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. Technical Report COC-9403, Georgia Institute of Technology, Atlanta, USA, 1993. http://tli.isye.gatech.edu/research/papers/papers.htm.

[2] K. Darby-Dowman and J. Little. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal on Computing*, 10(3):276–286, 1998.

[3] M. Desrochers and F. Soumis. A column generation approach to the urban transit crew scheduling problem. *Transportation Science*, 23(1):1–13, 1989.

[4] C. Gervet. Large Combinatorial Optimization Problems: a Methodology for Hybrid Models and Solutions. In *Journées Francophones de Programmation en Logique et par Contraintes*, 1998. http://www.icparc.ic.ac.uk/papers_byauthor.html.

[5] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, (1):25–46, 1993.

[6] N. Guerinik and M. V. Caneghem. Solving crew scheduling problems by constraint programming. In *Lecture Notes in Computer Science*, pages 481–498, 1995. Proceedings of the First International Conference on the Principles and Practice of Constraint Programming, CP'95.

[7] J. Lever, M. Wallace, and B. Richards. Constraint logic programming for scheduling and planning. *British Telecom Technical Journal*, (13):73–81, 1995. http://www.icparc.ic.ac.uk/papers_byauthor.html.

[8] T. H. Yunes, A. V. Moura, and C. C. de Souza. Solving large scale crew scheduling problems with constraint programming and integer programming. Technical Report IC-99-19, Institute of Computing, University of Campinas, Brazil, 1999. http://goa.pos.dcc.unicamp.br/otimo/published.html.

Epílogo

Até este ponto, ao se procurar uma solução ótima para o subproblema de *crew scheduling* não houve preocupação com o fato de que esta solução servirá como dado de entrada para o subproblema seguinte, *crew rostering*. Contudo, conforme ressaltado por Caprara et al. [5], esta prática, em geral, não é muito recomendada. No caso específico do problema aqui tratado, a solução do problema de *scheduling* pode influenciar bastante a solução do problema de *rostering*. Apesar de não explicitado nos Capítulos 4 e 5, as soluções obtidas com o algoritmo híbrido apresentam uma característica importante e fundamental: suas jornadas são, em grande parte, duplas-pegadas. A conseqüência disso pode ser melhor entendida quando se observa mais atentamente as restrições associadas ao problema de *rostering*. O Capítulo 6 provê uma descrição completa do problema de *rostering* e aborda essas questões em maiores detalhes.

Capítulo 6

O Problema de *Crew Rostering*

Prólogo

O artigo a seguir considera o problema de *crew rostering*, comparando o desempenho de diversas técnicas utilizadas na sua resolução. Após uma descrição detalhada do problema e das restrições envolvidas, duas abordagens diferentes são apresentadas: um modelo de Programação Linear Inteira e um modelo de Programação por Restrições. O modelo de Programação Linear Inteira foi resolvido por um algoritmo de *Branch-and-Bound* e por um algoritmo híbrido de geração de colunas que segue as mesmas idéias básicas do algoritmo híbrido descrito nos Capítulos 4 e 5. Os modelos propostos são acompanhados de experimentos realizados sobre instâncias oriundas de dados reais, i.e. instâncias construídas a partir das soluções encontradas para o problema de *crew scheduling*.

Este artigo é uma versão do relatório técnico [47] e foi submetido para a *Sixth International Conference on Principles and Practice of Constraint Programming*, que ocorrerá em Cingapura, no período de 18 a 22 de setembro de 2000. Até o momento da impressão deste texto, a notificação de aceitação do artigo ainda não havia sido divulgada.

A divulgação deste artigo, quando de sua submissão, resultou em um convite para que seus resultados fossem apresentados no *17th International Symposium on Mathematical Programming*, que ocorrerá na cidade de Atlanta, GA, EUA, no período de 7 a 11 de agosto de 2000 [48]. O convite foi feito pelo Dr. Raymond Kwan da Universidade de Leeds, Inglaterra, responsável por uma das sessões do congresso.

Modeling and Solving an Urban Transit Crew Rostering Problem

Tallys H. Yunes[*] tallys@acm.org
Arnaldo V. Moura arnaldo@dcc.unicamp.br
Cid C. de Souza[†] cid@dcc.unicamp.br

Abstract

This article describes the crew rostering problem stemming from the operation of a Brazilian bus company that serves a major urban area in the city of Belo Horizonte. The problem is solved by means of Integer Programming (IP) and Constraint Logic Programming (CLP) approaches, whose models are discussed in detail. Lower bounds obtained with a Linear Programming relaxation of the problem are used in order to evaluate the quality of the solutions found. We also present a hybrid column generation approach for the problem, combining IP and CLP over a set partitioning formulation. Experiments are conducted upon real data sets and computational results are evaluated, comparing the performance of these three solution methods.

1 Introduction

The overall *crew management* problem concerns the allocation of trips to crews within a certain planning horizon. In addition, it is necessary to respect a specific set of operational constraints and minimize a certain objective function. Being a fairly complicated problem as a whole, it is usually divided in two smaller subproblems: *crew scheduling* and *crew rostering* [4]. In the crew scheduling subproblem, the aim is to partition the initial set of trips into a minimal set of *feasible duties*. Each such duty is an ordered sequence of trips which is to be performed by the same crew and that satisfies a subset of the original problem constraints: those related to the sequencing of trips during a workday. The crew rostering subproblem takes as input the duties output by the crew scheduling phase and builds a roster spanning a longer period, e.g. months or years.

This article describes the crew rostering problem stemming from the operation of a Brazilian bus company that serves a major urban area in the city of Belo Horizonte. The problem is solved by means of Integer Programming (IP) and Constraint Logic Programming (CLP) approaches, whose models are discussed in detail. Lower bounds obtained with

*Supported by FAPESP grant 98/05999-4, and CAPES.
†Supported by FINEP (ProNEx 107/97), and CNPq (300883/94-3).

a Linear Programming relaxation of the problem are used in order to evaluate the quality of the solutions found. We also present a hybrid column generation approach for the problem, combining IP and CLP. Experiments are conducted upon real data sets and computational results are evaluated, comparing the performance of these three solution methods.

Some quite specific union regulations and operational constraints make this problem fairly distinct from some other known crew rostering problems found in the literature [3, 5]. In general, it is sufficient to construct one initial roster consisting of a feasible sequencing of the duties that spans the least possible number of days. The complete roster is then built by just assigning shifted versions of that sequence of duties to each crew so as to have every duty performed in each day in the planning horizon. In other common cases, the main concern is to balance the workload among the crews involved [2, 6, 7]. Although we also look for a roster with relatively balanced workloads, these approaches will not in general find the best solution for our purposes. We are not interested in minimizing the number of days needed to execute the roster, since the length of the planning horizon is fixed in advance. Our objective is to use the minimum number of crews when constructing the roster for the given period. Another difficulty comes from the fact that some constraints behave differently for each crew, depending on the amount of work assigned to it in the previous month. Moreover, different crews have different needs for days off, imposed by personal requirements.

The text is organized as follows. Section 2 gives a detailed description of the crew rostering problem under consideration. Section 3 explains the format of the input data sets used in our experiments. In Sect. 4, we present an Integer Programming formulation of the problem, together with some computational results. A pure Constraint Logic Programming model for the problem is described in Sect. 5, where some experiments are also conducted to evaluate its performance. As one additional attempt to solve the problem, the results achieved with a hybrid column generation approach appear in Sect. 6. All computation times presented in Sects. 4 to 6 are given in CPU seconds of a Pentium II 350 MHz. Finally, we draw the main conclusions in Sect. 7.

2 The Crew Rostering Problem

The duties obtained as output from the solution of the crew scheduling phase[1] must be assigned to crews day after day, throughout an entire planning horizon. This sequencing has to obey a set of constraints that differs from the constraints which are relevant to the crew scheduling problem. This set includes, for example, the need for days off, with a certain periodicity, and a minimum rest time between consecutive workdays.

[1] For more specific information on the scheduling subproblem for this case, see [8].

2.1 Input Data

The set of duties to be performed on weekdays is different from the set of duties to be performed on weekends or holidays, due to fluctuations on customer demand. Therefore, the crew scheduling problem gives as input for the rostering problem a number of distinct sets of duties.

The planning horizon we are interested in spans one complete month. It is important to take into account as input data many features of the month under consideration, such as: the total number of days, which days are holidays and which day of the week is the first day of the month (the remaining weekdays can be easily figured out from this information). The differences in the number of working days from one month to the next one lead to variations on the number of crews actually working in each month. Consequently, some rules must be observed in order to select the crews that are going to be effectively used. If, say, in month m 40 crews were needed, and in month $m+1$ only 38 will be necessary, how to select the 2 crews that are going to be left out? Furthermore, suppose that, after eliminating those crews that cannot work on the current month for some reason, the company has 50 crews available. Even if the number of crews remains the same, e.g. 40, from one month to the next one, it is important to evenly distribute the work among them. This balance can be obtained considering the number of days each crew has worked since the beginning of the year, for example, or with the aid of another kind of ranking function for the crews. Finally, since some constraints refer to a time window that spans more than one month (see Sect. 2.2) some attributes, for each crew, have to be carried over between successive months.

The input data needed to build the roster for month m is the following:

- The sets of duties D_{wk}, D_{sa}, D_{su} and D_{ho} which have to be performed on weekdays, Saturdays, Sundays and on holidays, respectively;
- The number of days, d, in month m;
- The occurrence of holidays in month m;
- The day of the week corresponding to the first day in month m;
- The whole set of crews, C, employed by the company;
- For each crew $i \in C$:
  - The set of days, OFF_i, in which i is off duty (e.g. vacations, sickness), excluding its ordinary weekly rests;
  - The number of days between the last Sunday i was off duty and the first day of month m (ls_i);

  - A binary flag, wr_i, that is equal to 1 if and only if i had a weekly rest in the last week of month $m-1$;
  - A binary flag, sl_i, that is equal to 1 if and only if i performed a split-shift duty during the last week of month $m-1$;
  - The difference, in minutes, between the last minute i was working in month $m-1$ and the first minute of the first day of month m (lw_i);

- For each duty $k \in D_{\mathrm{wk}} \cup D_{\mathrm{sa}} \cup D_{\mathrm{su}} \cup D_{\mathrm{ho}}$:
  - The start and end times of k (ts_k and te_k, respectively), given in minutes after midnight;
  - A binary flag, ss_k, that equals 1 if and only if k is a split-shift duty;

- A list of all crews in C sorted according to a certain ranking function. This ordering will be used to assign priorities to the crews when identifying the subset of C that is going to work in month m.

2.2 Problem Constraints

The constraints associated to the sequencing of the duties are:

(a) The minimum rest time between consecutive workdays is 11 hours;

(b) Every employee must have at least one day off per week. Moreover, for every time window spanning 7 weeks, at least one of these days off must be on a Sunday;

(c) When an employee performs one or more split-shift duties during a week, his day off in that week must be on Sunday;

(d) In every 24-hour period starting at midnight, within the whole planning horizon, each crew can start to work on at most one duty.

2.3 Objectives

For each month, we are looking for the cheapest solution in terms of the number of crews needed to perform all the duties requested. Additionally, it is desirable to have balanced workloads among all the crews involved, but the models we present in this article are not concerned with this issue yet.

3 The Input Data Sets

Before describing the IP and CLP models for the rostering problem, it is important to understand the format of the instances used in the computational experiments. These instances correspond to actual schedules constructed by a crew scheduling algorithm executed over real world data from the same bus company mentioned in Sect. 1 [8]. Using the duties built during the crew scheduling phase, we have constructed a set of instances ranging from small ones up to large-sized ones, typically encountered by the management personnel in the bus company. The main features of these instances appear in Table 1.

Table 1: Description of the instances for the experiments

Name	#Crews	#Days	# Duties Week	Sat	Sun	Holy
string	c	$d\ (h)$	$ss_{\mathrm{wk}}/tt_{\mathrm{wk}}$	$ss_{\mathrm{sa}}/tt_{\mathrm{sa}}$	$ss_{\mathrm{su}}/tt_{\mathrm{su}}$	$ss_{\mathrm{ho}}/tt_{\mathrm{ho}}$

The *Name* is just a string identifying the instance. The number of crews available for the roster, c, appears under the heading *#Crews*. The column *#Days* shows the number of days in the planning horizon in the format $d\ (h)$, where d is the total number of days and h indicates how many of those d days are holidays. The next four columns show the number of duties that must be performed in each kind of the possible working days: weekdays, Saturdays, Sundays and holidays, respectively. The format used is ss/tt, where tt is the total number of duties and ss represents how many of the tt duties are split-shift duties. To begin with, we set the following parameters, for every crew i: $OFF_i = \emptyset$, $ls_i = 1$, $wr_i = 1$, $sl_i = 0$ and $lw_i = 660$. This is the same as ignoring any information from the previous month when constructing the roster for the current month.

4 An Integer Programming Approach

Let n be the total number of crews available and let d be the number of days in the current month m. Moreover, let p, q, r and s be the numbers of duties to be performed on weekdays, Saturdays, Sundays and holidays, respectively (i.e. $|D_{\mathrm{wk}}| = p$, $|D_{\mathrm{sa}}| = q$, $|D_{\mathrm{su}}| = r$ and $|D_{\mathrm{ho}}| = s$).

The IP formulation of the rostering problem is based on x_{ijk} binary variables which are equal to 1 if and only if crew i performs duty k on day j. If j is a weekday, k belongs to $\{0, 1, \ldots, p\}$. Analogously, if j is a Saturday, Sunday or holiday, k ranges over $\{0, p+1, \ldots, p+q\}$, $\{0, p+q+1, \ldots, p+q+r\}$ or $\{0, p+q+r+1, \ldots, p+q+r+s\}$,

respectively. The duty numbered 0 is a special duty indicating *idleness*. Thus, if $x_{ij0} = 1$ it means that crew i is not working on day j. For modeling purposes, we set $ts_0 = +\infty$, $te_0 = 0$ and $ss_0 = 0$.

Given a day j in m, K_j represents its set of duty indexes, except for the duty 0. For instance, if j is a Saturday then $K_j = \{p+1, \ldots, p+q\}$.

4.1 The Model

The main objective is to minimize the number of crews working during the present month. This is equivalent to maximizing the number of crews which are idle during the whole month. Let us define new variables $y_i \in R^+$, for all $i \in \{1, \ldots, n\}$, which are equal to 1 if $x_{ij0} = 1$, for all $j \in \{1, \ldots, d\}$, and are equal to 0 otherwise. To achieve this behavior for the y_i variables, it is necessary to relate them to the x_{ij0} variables through the following constraints

$$y_i \leq x_{ij0}, \quad \forall\, i, \forall\, j \ . \tag{1}$$

The objective function can then be written as $\max \sum_{i=1}^{n} y_i$. Equations (1) combined with the objective function force a y_i variable to be equal to 1 if and only if crew i is idle during the entire month.

The occurrence of days on which the crews are known to be off duty (e.g. previously assigned holiday periods) is satisfied by setting

$$x_{ij0} \quad = \quad 1, \quad \forall\, i, \forall\, j \in \mathit{OFF}_i \ . \tag{2}$$

The subsequent formulas take care of the feasibility of the roster (see Sect. 2.2).

Constraints (a) are dealt with in two steps. Equation (3) takes care of the assignment of duties for the first day in month m. For the other days, assume that a crew i does duty k on day $j-1$. The set $K'_j[k]$ of other duties that cannot be taken by the same crew i on day j because of the 660-minute minimum rest time is given by $\{k' \in K_j \mid ts_{k'} - (te_k - 1440) < 660\}$. Therefore, (4) guarantees the minimum rest time between successive days in month m.

$$x_{i1k} \quad = \quad 0, \quad \forall\, i, \forall\, k \in K_1 \mid ts_k + lw_i < 660 \ , \tag{3}$$

$$x_{i(j-1)k} + \sum_{k' \in K'_j[k]} x_{ijk'} \quad \leq \quad 1, \quad \forall\, i, \forall\, j \in \{2, \ldots, d\}, \forall\, k \in K_{j-1} \ . \tag{4}$$

Let us define a *complete week* as seven consecutive days, inside month m, ranging from Monday to Sunday. For every complete week, W, in m, we impose the mandatory day off as

$$\sum_{j \in W} x_{ij0} \geq 1, \quad \forall\, i \ . \tag{5}$$

If month m does not start with a complete week, let W' be the set of days in the first week of m up to Sunday. Each crew i with $wr_i = 0$ needs to rest in W' and this is achieved with

$$\sum_{j \in W'} x_{ij0} \geq 1, \quad \forall\, i \mid wr_i = 0 \ . \tag{6}$$

The constraint stating that for each period of time spanning 7 weeks each crew must have at least one day off on Sunday can be described as follows. For each crew i such that $ls_i + d \geq 49$, we construct the set T_i containing the Sundays in the first $(49 - ls_i)$ days of m. Then, we impose

$$\sum_{j \in T_i} x_{ij0} \geq 1, \quad \forall\, i \mid ls_i + d \geq 49 \ . \tag{7}$$

Together, (5) to (7) represent constraints (b).

Suppose that the first day of month m is not Monday and let j^* be the first Sunday in m. To satisfy constraint (c) for each crew i such that $sl_i = 1$, we must state that

$$x_{ij^*0} = 1 \ . \tag{8}$$

Let S_m be the set of Sundays in m after its 6th day and let P_j be the set of split-shift duties on day j. For these Sundays, we respect constraint (c) with

$$x_{ij0} \geq \sum_{k \in P_{j-r}} x_{i(j-r)k}, \quad \forall\, i,\ \forall\, j \in S_m,\ \forall\, r \in \{1, \ldots, 6\} \ . \tag{9}$$

Equation (10) guarantees that each crew is assigned exactly one duty in each day, thus satisfying constraints (d). Additionally, (11) represents the implicit constraint that every duty must be performed in each day, except for the special duty 0.

$$x_{ij0} + \sum_{k \in K_j} x_{ijk} = 1, \quad \forall\, i,\ \forall\, j \ , \tag{10}$$

$$\sum_{i=1}^{n} x_{ijk} = 1, \quad \forall\, j,\ \forall\, k \in K_j \ . \tag{11}$$

4.2 Computational Results

The computational results obtained with the IP model are shown in Table 2. The figures under the heading LB come from lower bounds on the value of the optimal solution returned by the linear programming relaxation of the IP model. Notice however that the objective function described in Sect. 4.1 asks for the maximization of the number of idle crews, which is equivalent to minimizing the number of crews needed to compose the roster. For the

Table 2: Computational experiments with the IP model

Name	#Crews	#Days	# Duties Week	Sat	Sun	Holy	LB	Sol	Time
s01	10	10 (1)	00/04	00/01	00/01	00/01	4	6	0.62
s02	10	15 (2)	00/04	00/01	00/01	00/01	4	7	1.50
s03	10	20 (2)	00/04	00/01	00/01	00/01	4	6	2.00
s04	10	25 (2)	00/04	00/01	00/01	00/01	4	6	4.33
s05	10	30 (2)	00/04	00/01	00/01	00/01	4	8	20.91
s06	10	30 (2)	01/04	00/01	00/01	00/01	4	6	9.06
s07	10	30 (2)	02/04	00/01	00/01	00/01	4	6	10.61
s08	10	30 (2)	03/04	00/01	00/01	00/01	4	7	6.81
s09	10	30 (2)	04/04	00/01	00/01	00/01	4	8	9.21
s10	10	30 (2)	04/04	01/01	00/01	00/01	4	7	5.05
s11	10	30 (2)	04/04	01/01	00/01	01/01	4	8	8.35
s12	15	30 (2)	00/04	00/01	00/01	00/01	4	5	8.90

purpose of comparison with the CLP model, the values in the *LB* and *Sol* columns of Table 2 represent the number of crews actually working, i.e. the total number of crews available minus the value of the objective function. Finding the optimal solution of the instances in Table 2 turned out to be a very difficult task, despite their relatively small size. Hence, the solution value in column *Sol* corresponds to the first integer solution found by the model, for each instance. The linear relaxations and the integer programs were solved with the CPLEX[2] Solver, version 6.5.

Although the computation times are quite small, the gap between the values of the lower bounds and the feasible solutions is noticeable. Further, these values are still not a good indication of the quality of the model, since we are dealing with very small instances. Yet, when trying to find integer solutions for instances with tens of duties in a workday, this model performed very poorly and no answer could be found within 30 minutes of computation time. Therefore, we decided to experiment with a pure Constraint Logic Programming formulation of the problem.

5 A Constraint Logic Programming Approach

Having found difficulties when solving the crew rostering problem with a pure IP model, as described in Sect. 4, we decided to try a constraint-based formulation. We used the

[2]CPLEX is a registered trademark of ILOG Inc.

ECL^{i}PS^e [3] finite domain constraint solver, version 4.2, to construct and solve the model.

5.1 The Model

Let n, d, p, q, r and s be defined as in Sect. 4. The main idea of the CLP model for the rostering problem is to represent the final roster as a bidimensional matrix, X, where each cell X_{ij} ($i \in \{1,\dots,n\}$, $j \in \{1,\dots,d\}$) contains the duty performed by crew i on day j.

The X_{ij}'s are finite domain variables whose domains depend on the value of j. As in Sect. 4, the duties obtained from the crew scheduling phase are numbered according to their classification as duties for weekdays, Saturdays, Sundays or holidays. In this model, we will not have the concept of a special duty for idleness, as the duty numbered 0 in the IP model. In fact, we will have, for each day, a set of *dummy duties* which tell that a certain crew is off duty.

It is easy to see that the number of crews needed to construct a roster must be at least the maximum number of duties that may be present in any given day of the current month. Thus, we can state that $n \geq \max\{p,q,r,s\}$. Consequently, as the number of X variables for each day j is equal to n, if the domains of these variables were restricted to be the set of duties for day j, some of them would have the same value in the final solution. As we will see later, modeling can be simplified if we avoid this situation and here comes the need for the dummy duties. Let K_j be defined as in Sect. 4. Moreover, let the total number of duties be calculated as $tnd = p + q + r + s$. The domains of the X_{ij} variables are then defined as

$$X_{ij} \ :: \ K_j \cup \{tnd+1, tnd+2, \dots, tnd+(n-|K_j|)\} \quad \forall\, i,\ \forall\, j\ . \tag{12}$$

If X_{ij} is assigned a duty whose number is greater than tnd, it means that crew i is idle on day j.

Three other sets of variables have to be defined in order to facilitate the representation of the constraints. Let TS, TE and SS be lists of integers defined as follows, $\forall\ k \in \{1,\dots,tnd\}$: $TS[k] = ts_k$, $TE[k] = te_k - 1440$, $SS[k] = ss_k$. The values of ts, te and ss for the dummy duties are $+\infty$, 0 and 0, respectively. The new variables are called $Start_{ij}$, End_{ij} and $Split_{ij}$ and relate to the X_{ij} variables through `element` constraints:

$$\begin{aligned}&\texttt{element}(X_{ij}, TS, Start_{ij})\ ,\\ &\texttt{element}(X_{ij}, TE, End_{ij})\ ,\\ &\texttt{element}(X_{ij}, SS, Split_{ij})\ .\end{aligned}$$

Now we can state the constraints (a) through (d) in the ECL^{i}PS^e notation.

[3] `http://www.icparc.ic.ac.uk/eclipse`.

Equations (13) and (14) assure that the minimum rest time between consecutive duties is 11 hours. Note the special case for the first day of month m.

$$Start_{i1} + lw_i \geq 660, \ \forall\, i \ , \tag{13}$$
$$Start_{ij} - End_{i(j-1)} \geq 660, \ \forall\, i, \ \forall\, j \in \{2, \ldots, d\} \ . \tag{14}$$

Similarly to what was defined in Sect. 4.1, we use the concept of a complete week, W_i, for each crew i, as a list of variables $[X_{it}, X_{i(t+1)}, \ldots, X_{i(t+6)}]$, where t is any Monday and $t+6$ is its subsequent Sunday, both in month m. The need for at least one day off during each week is represented by (15), for complete weeks. Notice that this constraint must be repeated for each complete week W_i associated with every crew i. If $wr_i = 0$ and the first day of m is not Monday, we also need to impose (16), for each crew i and initial week W'_i.

$$\texttt{atmost_less}(6, W_i, tnd + 1) \ , \tag{15}$$
$$\texttt{atmost_less}(|W'_i| - 1, W'_i, tnd + 1) \ . \tag{16}$$

In Equation (16), $|W'_i|$ denotes the number of elements in list W'_i. We use the predicate atmost_less(N, L, V) to state that at most N elements of list L can be smaller than V. This behavior is achieved with the definitions below

```
f_less([],_,[]) :- !.
f_less([X|Y],Val,[B|R]) :- #<(X,Val,B), f_less(Y,Val,R).
atmost_less(N,L,Val) :- f_less(L,Val,BF), atmost(N,BF,1).
```

To satisfy constraints (b), there is one condition missing, besides (15) and (16), which assumes at least one day off on Sunday, every seven weeks, for every crew. For each crew i, if $ls_i + d \geq 49$, then

$$\texttt{atmost_less}(|L_i| - 1, L_i, tnd + 1) \ , \tag{17}$$

where L_i is a list containing the X_{ij}'s associated with the Sundays present in the first $(49 - ls_i)$ days of m.

Constraints (c) also make use of the concept of complete weeks, but do not include Sundays. We denote this reduced complete week W^*_i as the list $[Split_{it}, Split_{i(t+1)}, \ldots, Split_{i(t+5)}]$. Notice that we now consider the *Split* variables instead of the X variables, as when representing constraints (b).

$$Split_{it} + \cdots + Split_{i(t+5)} \ \texttt{\#>}\ 0 \quad \texttt{\#=>} \quad X_{i(t+6)} \ \texttt{\#>}\ tnd, \ \forall\, i, \ \forall\, W^*_i \ , \tag{18}$$
$$X_{ik} \quad \texttt{\#>} \quad tnd, \ \forall\, i \ . \tag{19}$$

By (18), if one of the $Split_{it}, \ldots, Split_{i(t+5)}$ variables equals 1, then crew i must rest on the next Sunday, which corresponds to $X_{i(t+6)}$. The special case of the first week of m, when

the month does not start on Monday and $sl_i = 1$, is dealt with by (19). Here, k stands for the first Sunday of month m.

Our choice of variables already guarantees that each crew starts only one duty per day. But we must also make sure that every duty is assigned to one crew on each day. Because of the dummy duties, this condition can be met easily just by forcing the X_{ij} variables to be pairwise distinct, for each day j:

$$\texttt{alldifferent}([X_{1j}, \ldots, X_{nj}]), \ \forall j \ . \tag{20}$$

Finally, we need to preassign the rest days which are known in advance

$$X_{ij} \texttt{ #> } tnd, \ \forall i, \forall j \in OFF_i \ . \tag{21}$$

Labeling is done over the X_{ij} variables using the first-fail principle.

5.2 Computational Results

When compared to the IP model of Sect. 4, this model performed much better both in terms of solution quality and computation time. As can be seen in Table 3, it was possible to find feasible solutions for fairly large instances in a few seconds. Again, no minimization predicate was used and the solutions presented here are the first feasible rosters encountered by the model.

Some special cases deserve further consideration. When providing 15 crews to build the rosters for instances s16 and s17, the model could not find a feasible solution even after 10 hours of search. Then, after raising the number of available crews in these instances to 16 (s16a) and 18 (s17a), respectively, two solutions were easily found. Another interesting observation arises from instance s19. This instance comes from the solution of a complete real world crew scheduling problem. In this problem, the optimal solution for weekdays contains 25 duties, 22 of which are split shifts. As we did not have access to the input data sets for the other workdays, the sets of duties for Saturdays, Sundays and holidays are subsets of the solution given by the scheduling algorithm for a weekday. Instance s19a is made up of the same duties, except that all of them are artificially considered non-split shifts. Notice that the value of the first solution found is significantly smaller for instance s19a than it is for instance s19. This is an indication of how severe is the influence of the constraints (c) introduced in Sect. 2.2. Moreover, we can see from Table 3 that the values of the solutions grow quickly as the number of split-shift duties increases. With this point in mind, we generated two other solutions for the same crew scheduling problem where the total number of duties used was increased in favor of a smaller number of split shifts. These are s20 and s21. Despite the larger number of duties in the input, the final roster for these instances uses less crews than it did for instance s19. This strengthens the remark made by

Table 3: Computational experiments with the CLP model

Name	#Crews	#Days	# Duties Week	Sat	Sun	Holy	LB	Sol	Time
s01	10	10 (1)	00/04	00/01	00/01	00/01	4	5	0.08
s02	10	15 (2)	00/04	00/01	00/01	00/01	4	5	0.18
s03	10	20 (2)	00/04	00/01	00/01	00/01	4	5	0.23
s04	10	25 (2)	00/04	00/01	00/01	00/01	4	5	0.36
s05	10	30 (2)	00/04	00/01	00/01	00/01	4	5	0.48
s06	10	30 (2)	01/04	00/01	00/01	00/01	4	5	0.52
s07	10	30 (2)	02/04	00/01	00/01	00/01	4	5	0.50
s08	10	30 (2)	03/04	00/01	00/01	00/01	4	6	0.52
s09	10	30 (2)	04/04	00/01	00/01	00/01	4	7	0.52
s10	10	30 (2)	04/04	01/01	00/01	00/01	4	7	0.52
s11	10	30 (2)	04/04	01/01	00/01	01/01	4	7	0.53
s12	15	30 (2)	00/04	00/01	00/01	00/01	4	5	0.90
s13	15	30 (2)	00/10	00/06	00/05	00/05	10	13	1.22
s14	15	30 (2)	03/10	01/06	00/05	01/05	10	13	1.35
s15	15	30 (2)	03/10	03/06	00/05	03/05	10	15	1.36
s16	15	30 (2)	05/10	03/06	00/05	03/05	10	?	> 10 h
s16a	16	30 (2)	05/10	03/06	00/05	03/05	10	16	1.49
s17	15	30 (2)	07/10	03/06	00/05	03/05	10	?	> 10 h
s17a	18	30 (2)	07/10	03/06	00/05	03/05	10	18	1.78
s18	30	30 (2)	00/20	00/10	00/10	00/10	20	25	4.96
s19	50	30 (2)	22/25	12/15	12/15	12/15	25	47	14.46
s19a	40	30 (2)	00/25	00/15	00/15	00/15	25	33	9.36
s20	40	30 (2)	06/26	02/15	02/15	02/15	26	34	10.50
s21	40	30 (2)	00/31	00/20	00/20	00/20	31	36	8.30

Caprara et al. [4] that, ideally, the scheduling and rostering phases should work cyclicly, with some feedback between them.

6 Proving Optimality

In Sects. 4 and 5, we showed that finding provably optimal solutions for this rostering problem is a difficult task. Moreover, it is possible to see from Table 3 that the lower bound provided by the Linear Programming relaxation of the problem is always equal to the largest number of duties that must be performed on a workday. This is clearly a trivial lower bound and probably not a very good one. We decided then to try another formulation for the problem, so as to find better feasible solutions or, at least, better lower bounds.

6.1 A Hybrid Model

Another possible mathematical model for the rostering problem turns out to be a typical set partitioning formulation:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j = 1, \ \forall\, i \in \{1, \dots, e\} \\ & x_j \in \{0,1\}, \ \forall\, j \in \{1, \dots, n\} \ . \end{aligned}$$

All numbers a_{ij} in the coefficient matrix A are 0 or 1 and its columns are constructed as shown in Fig. 1. Each column is composed of d sequences of numbers, where d is the number of days in the planning horizon. For each $k \in \{1, \dots, d\}$, the k-th sequence, l_k, contains h_k numbers, where h_k is the number of duties that must be performed on day k. Also, at most one number inside each sequence is equal to 1. The number of lines e, in A, equals $\sum_{k=1}^{d} h_k$.

$$(\overbrace{0\cdots010\cdots0}^{h_1}\overbrace{0\cdots010\cdots0}^{h_2}\cdots\overbrace{0\cdots010\cdots0}^{h_d})^{\mathrm{T}}$$

Figure 1: A column in the coefficient matrix of the set partitioning formulation

Besides having the previous characteristics, a column in A must represent a feasible roster for one crew. More precisely, let $t = (u_1, u_2, \dots, u_d)$ be a feasible roster for a crew,

where u_k, $k \in \{1, \ldots, d\}$, is the number of the duty performed on day k. Remember from Sect. 4.1 that $u_k \in D_k \cup \{0\}$, where D_k may be equal to $\{1, \ldots, p\}$, $\{p+1, \ldots, p+q\}$, $\{p+q+1, \ldots, p+q+r\}$ or $\{p+q+r+1, \ldots, p+q+r+s\}$, depending on whether k is a weekday, a Saturday, a Sunday or a holiday, respectively. For every such feasible roster t, A will have a column where, in each sequence l_k, the i-th number will be equal to 1 ($i \in \{1, \ldots, h_k\}$) if and only if u_k is the i-th duty of D_k. In case $u_k = 0$, all numbers in sequence l_k are set to 0.

With this representation, the objective is to find a subset of the columns of A, with minimum size, such that each line is covered exactly once. This is equivalent to finding a number of feasible rosters which execute the all the duties in each day of the planning horizon.

It is not difficult to see that the number of columns in the coefficient matrix is enormous and it is hopeless to try to generate them all in advance. Hence, we decided to implement a *Branch-and-Price* algorithm [1] to solve this problem, generating columns as they are needed. This approach is considered hybrid because the column generation subproblem is solved by a Constraint Logic Programming model. In our case, this model is a variation of the CLP model of Sect. 5. Now, instead of looking for a complete solution for the rostering problem, we are only interested in finding, at each time, a feasible roster corresponding to a column in A with negative reduced cost. The whole algorithm follows the same basic ideas described in [8].

6.2 Computational Results

The best results for the hybrid model were achieved when setting the initial columns of matrix A as the columns corresponding to the first solution found by the CLP model of Sect. 5. Also, the ordinary labeling mechanism worked better than labeling according to the first-fail principle.

With this model, we could find provably optimal solutions for small instances of the rostering problem, as shown in Table 4, where column *Opt* gives the optimal value. This is a noticeable improvement over the pure IP model of Sect. 4, which was not able to find any optimal solution, even for the smallest instances. Besides, when comparing Tables 3 and 4, we can see that the first solutions found by the pure CLP model for instances s01 to s06 are indeed optimal.

This hybrid approach is still under development and there is a lot of work to be done. Nevertheless, we believe that the main reason for the behavior of this model resides on the fact that this formulation leads to a highly degenerate problem. When trying to solve larger instances, the pricing subroutine keeps generating columns indefinitely, with no improvements on the value of the objective function. This is because there are many basic variables

Table 4: Computational experiments with the hybrid model

Name	#Crews	#Days	# Duties Week	Sat	Sun	Holy	Opt	Time
s01	10	10 (1)	00/04	00/01	00/01	00/01	5	0.95
s02	10	15 (2)	00/04	00/01	00/01	00/01	5	2.19
s03	10	20 (2)	00/04	00/01	00/01	00/01	5	10.57
s04	10	25 (2)	00/04	00/01	00/01	00/01	5	639.75
s05	10	30 (2)	00/04	00/01	00/01	00/01	5	38.12
s06	10	30 (2)	01/04	00/01	00/01	00/01	5	30.60
s07	10	30 (2)	02/04	00/01	00/01	00/01	?	>1 h

with value zero which are replaced by other columns that enter the basis with value zero as well. As a consequence, the linear relaxation of the first node of the *Branch-and-Price* enumeration tree could not be completely solved in the medium and large-sized instances. Thus, in order to obtain better linear programming lower bounds, we need to address those degeneracy problems more closely.

7 Conclusions and Future Work

We have given a detailed description of an urban transit crew rostering problem that is part of the overall crew management process in a medium-sized Brazilian bus company. This problem is rather different from some other bus crew rostering problems found in the literature.

Three main approaches have been applied in order to solve this problem. Initially, a pure Integer Programming (IP) model was developed, enabling us to find feasible rosters for very small instances. We achieved better results with a pure Constraint Logic Programming (CLP) model, which managed to construct feasible solutions for typical real world instances in a few seconds.

Obtaining better lower bounds on the value of the optimal solution could be helpful in estimating more precisely the quality of the solutions obtained with the pure CLP model. Therefore, following our experience with good quality lower bounds provided by linear relaxations of set partitioning formulations [8], we devised a third approach. The rostering problem was then formulated as a set partitioning problem with a huge number of columns in the coefficient matrix. This integer program was fed into a hybrid column generation algorithm which followed the same ideas presented in [8]. With this attempt, we could find optimal solutions for small instances of the problem. Finding provably optimal solutions

for the largest instances is still a difficult task, apparently due to degeneracy problems. We believe that the performance of this third model can be significantly improved if these issues are investigated in more detail. Besides, it may also be possible to improve the labeling strategy with problem specific heuristics, and extract a better performance from the constraint-based column generator.

References

[1] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. Technical Report COC-9403, Georgia Institute of Technology, Atlanta, EUA, 1993. http://tli.isye.gatech.edu/research/papers/papers.htm.

[2] L. Bianco, M. Bielli, A. Mingozzi, S. Ricciardelli, and M. Spandoni. A heuristic procedure for the crew rostering problem. *European Journal of Operational Research*, 58(2):272–283, 1992.

[3] A. Caprara, M. Fischetti, P. Toth, and D. Vigo. Modeling and solving the crew rostering problem. Technical Report OR-95-6, DEIS, University of Bologna, Italy, 1995. Published in *Software Practice and Experience* number 28, 1998.

[4] A. Caprara, M. Fischetti, P. Toth, D. Vigo, and P. L. Guida. Algorithms for railway crew management. *Mathematical Programming*, 79:125–141, 1997.

[5] A. Caprara, F. Focacci, E. Lamma, P. Mello, M. Milano, P. Toth, and D. Vigo. Integrating constraint logic programming and operations research techniques for the crew rostering problem. Technical Report OR-96-12, DEIS, University of Bologna, Italy, 1996. Published in *Operations Research* number 46, 1999.

[6] P. Carraresi and G. Gallo. A multi-level bottleneck assignment approach to the bus drivers' rostering problem. *European Journal of Operational Research*, 16(2):163–173, 1984.

[7] J. K. Jachnik. Attendance and rostering system. In A. Wren, editor, *Computer Scheduling of Public Transport*, pages 337–343. North-Holland Publishing Co., 1981.

[8] T. H. Yunes, A. V. Moura, and C. C. de Souza. A hybrid approach for solving large scale crew scheduling problems. In *Lecture Notes in Computer Science,* vol. 1753, pages 293–307, Boston, MA, USA, January 2000. Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00).

Epílogo

Os resultados obtidos com o algoritmo de *Branch-and-Bound* para o modelo de Programação Linear Inteira mostraram-se bastante insatisfatórios. Foi possível apenas encontrar soluções viáveis, de baixa qualidade, para instâncias muito pequenas do problema. Da mesma maneira, os limites inferiores obtidos com a resolução da relaxação linear desse modelo não trouxeram benefícios significativos, igualando-se a outros limites inferiores triviais. O modelo de Programação por Restrições, por outro lado, foi bastante eficiente na busca de soluções viáveis para instâncias reais do problema. As soluções encontradas, apesar de não possuírem valores ótimos comprovados, custam menos que as soluções fornecidas pelo algoritmo de *Branch-and-Bound*. Ainda com relação ao modelo de Programação por Restrições, tentou-se utilizar um predicado de minimização com o objetivo de encontrar soluções ótimas para algumas instâncias do problema. No entanto, essa tentativa não foi bem sucedida, mesmo para instâncias muito pequenas. Finalmente, a abordagem híbrida não foi tão bem sucedida quanto no caso do problema de *crew scheduling*. Aqui, o número de colunas é consideravelmente maior e a formulação alternativa de Programação Linear Inteira resultou em um problema de partição de conjuntos altamente degenerado.

Comparações entre os algoritmos implementados para este problema, em termos de gráficos, aparecem nas figuras a seguir. Os nomes das instâncias são os mesmos nomes utilizados nas tabelas do artigo deste capítulo, com exceção da letra "s". As Figuras 6.1 e 6.2 comparam o modelo de Programação por Restrições com o primeiro modelo de Programação Linear Inteira, em termos de qualidade da primeira solução viável obtida e do tempo gasto para se encontrar essa solução, respectivamente. A Figura 6.3 compara o modelo de Programação por Restrições com o algoritmo híbrido implementado para o segundo modelo de Programação Linear Inteira quanto ao desempenho computacional. A coluna referente ao tempo gasto pelo algorimo híbrido para encontrar a solução ótima para a instância s04 foi desenhada com um padrão diferenciado. Isto porque o tempo gasto nesse caso foi de 639.75 segundos de CPU, distanciando-se muito dos demais.

Para que se tenha uma idéia melhor quanto à forma de uma solução do problema de *crew rostering*, a Figura 6.4 exibe o escalonamento construído pelo modelo de Programação por Restrições para a instância s20. As linhas verticais separam os dias do mês e os escalonamentos mensais de cada dupla de funcionários aparecem dispostos horizontalmente. Os retângulos numerados são as jornadas diárias obtidas a partir da solução do problema de *crew scheduling*.

A Figura 6.5 mostra a distribuição de carga de trabalho para cada dupla de funcionários, obtida a partir do escalonamento apresentado na Figura 6.4. A escala numerada horizontalmente indica a quantidade total de horas trabalhadas durante o mês em termos de número de dias, i.e. períodos 24 horas. Da mesma maneira que no caso do problema de *crew sche-*

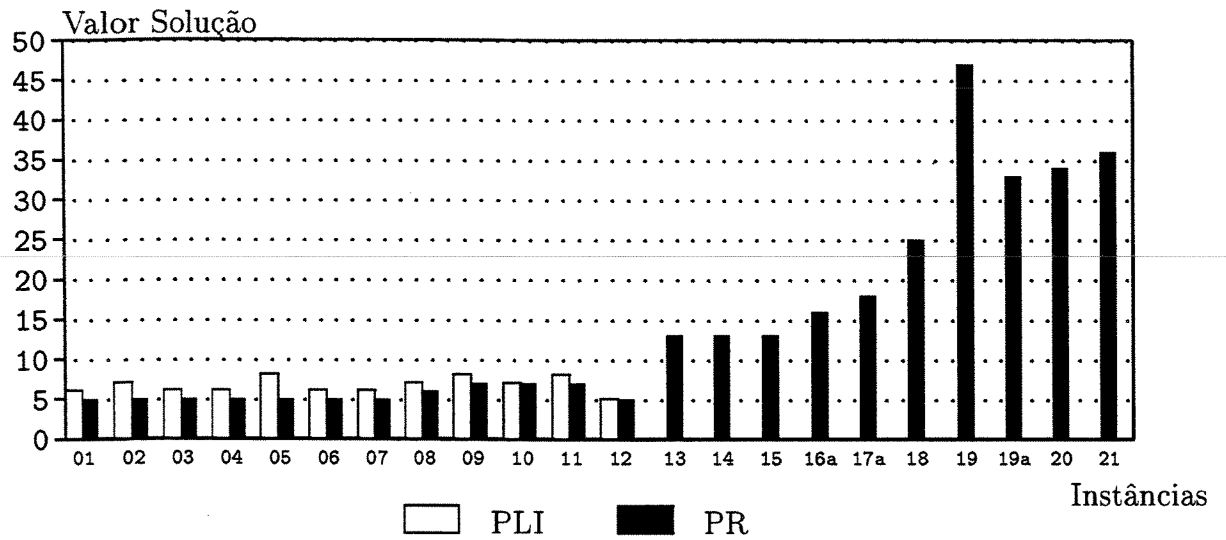


Figura 6.1: Comparação entre Programação Linear Inteira (PLI) e Programação por Restrições (PR) quanto à qualidade da solução

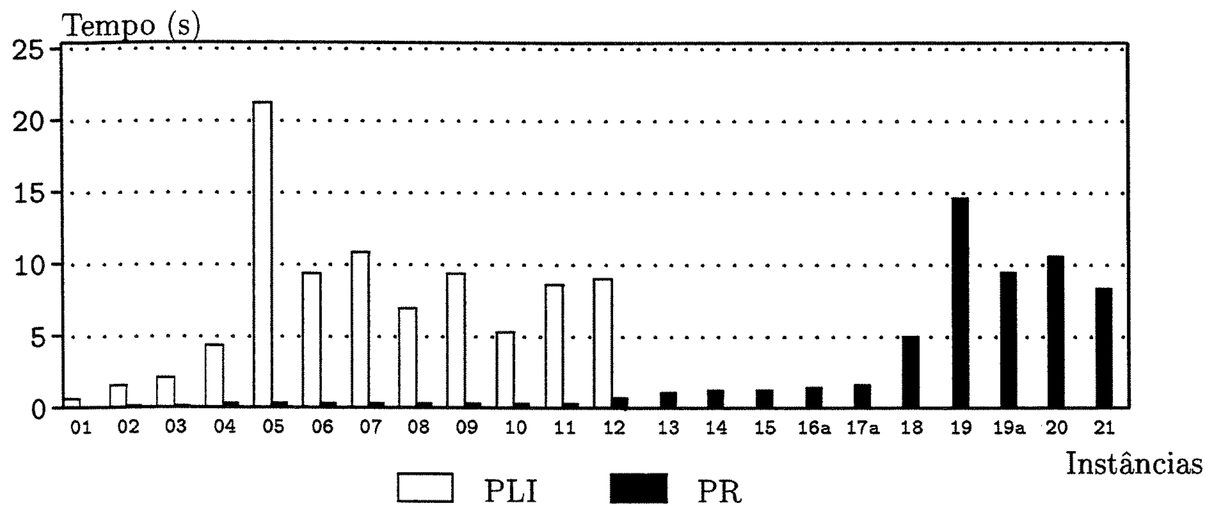


Figura 6.2: Comparação entre Programação Linear Inteira (PLI) e Programação por Restrições (PR) quanto ao desempenho

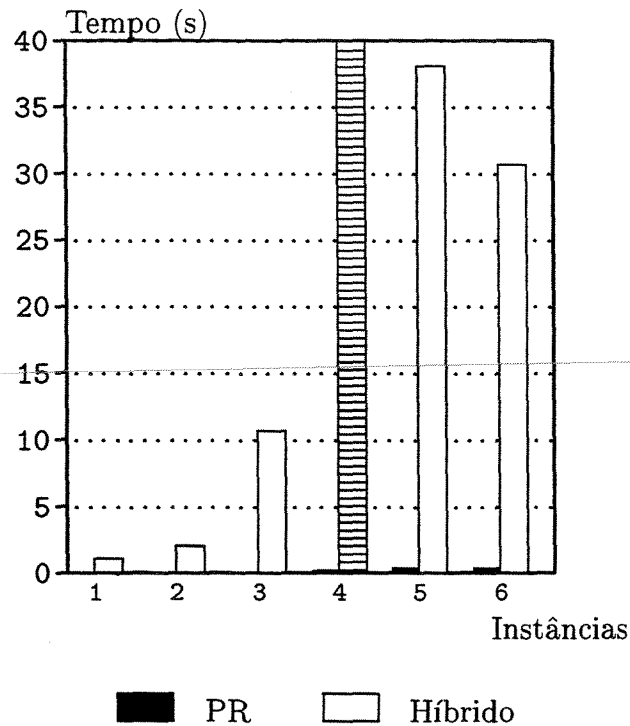


Figura 6.3: Comparação entre Programação por Restrições (PR) e o algoritmo híbrido quanto ao desempenho

duling, os modelos do problema de *crew rostering* não se preocupam explicitamente com o equilíbrio de trabalho entre os funcionários.

Não foi possível, dentro do escopo dessa dissertação, estudar em maiores detalhes as dificuldades inerentes à solução exata do problema de *crew rostering*. Entretanto, no Capítulo 9, discutem-se possíveis direções de investigação no sentido de superar essas dificuldades.

Figura 6.4: Escalonamento mensal construído para a instância s20

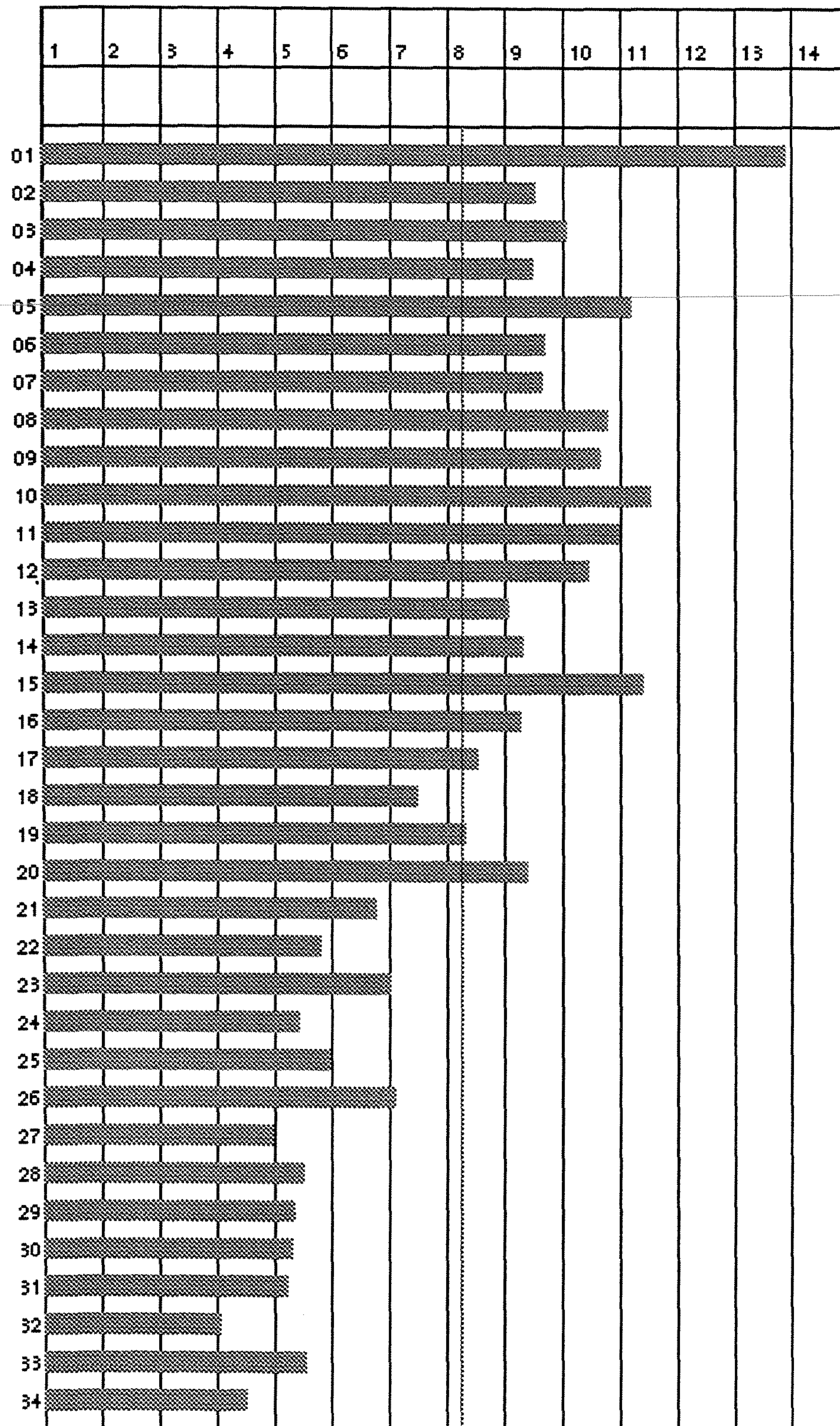


Figura 6.5: Carga de trabalho para o escalonamento mensal da instância s20

Capítulo 7

Detalhes de Implementação

Por motivos de conformidade com os padrões das conferências a que foram submetidos, os artigos incluídos como parte dessa dissertação sofreram restrições de espaço. Com isso, alguns detalhes importantes de implementação dos algoritmos tiveram de ser omitidos. Este capítulo descreve esses detalhes de maneira mais completa.

7.1 Geradores de Código

Como já mencionado anteriormente, os modelos em Programação por Restrições e Programação Linear Inteira necessários para se representarem os problemas abordados nessa dissertação apresentam um número muito grande de restrições e variáveis. Desse modo, não seria viável criá-los manualmente. Para se ter uma idéia, o modelo puro de Programação por Restrições para a resolução do subproblema de *crew scheduling* possui da ordem de 10^4 restrições. Além disso, tais modelos variam em função dos dados de entrada. Por essas razões, escreveram-se programas em C responsáveis por gerar os códigos na linguagem ECL^{i}PS^e e os arquivos de entrada para o CPLEX.

7.2 A Heurística CFT

Um dos algoritmos utilizados para se abordar o problema de *crew scheduling* foi uma heurística Lagrangeana, conforme apresentado na seção 3.3 do Capítulo 4. Essa heurística ficou conhecida por ter obtido sucesso na resolução de problemas de cobertura de conjuntos com um número muito grande de colunas na matriz de coeficientes [5]. A implementação da heurística feita neste trabalho baseou-se fundamentalmente na descrição contida no relatório técnico de Caprara et al. [3]. Apesar deste último relatório estar bastante completo, alguns detalhes de implementação não são totalmente esclarecidos em seu texto. Nesses

pontos, para a implementação descrita no presente trabalho, foram tomadas as decisões de projeto que se mostraram mais adequadas. Alguns ganhos de desempenho e qualidade foram conseguidos a partir de sugestões retiradas do Capítulo 6 de [34].

O modelo do problema de cobertura de conjuntos em que se baseia essa heurística é idêntico ao modelo apresentado na Seção 2.4.2 do Capítulo 2. A relaxação Lagrangeana utilizada também é igual àquela apresentada no mesmo exemplo.

7.2.1 Descrição da Heurística

O corpo principal da heurística CFT aparece na Figura 7.1. No início de cada iteração (passo 3), algumas variáveis são fixadas em 1, reduzindo-se a dimensão do problema. A seleção dessas variáveis é feita de acordo com o valor da solução heurística encontrada na iteração anterior (passo 7). O laço principal é interrompido quando os limites inferior (LB) e superior (z^*) estão suficientemente próximos ($z^* \leq \beta \cdot \text{LB}$) ou quando um limite de tempo é alcançado (*timeout*).

CFT()
1. $z^* \leftarrow +\infty$; $\overline{u} = 0$; $F = \emptyset$
2. **Repita**
3. FixaVariáveis(F)
4. 3-PHASE($\overline{x}$,$\overline{u}$)
5. **Se** custo($\overline{x}$) $< z^*$ **então**
6. $x^* \leftarrow \overline{x}$; $z^* \leftarrow$ custo($\overline{x}$);
7. $F \leftarrow$ AtualizaFixações($\overline{x}$)
8. **Até que** ($z^* \leq \beta \cdot \text{LB}$) **ou** *timeout*
9. **Retorne** x^*

Fim

Figura 7.1: Funcionamento da heurística CFT

O núcleo da heurística (passo 4) é uma seqüência de 3 fases que se repetem até que uma solução de boa qualidade seja encontrada. A subrotina 3-PHASE, apresentada na Figura 7.2, encapsula esses 3 passos.

No passo b, parte-se do vetor de multiplicadores de Lagrange corrente ($\overline{u}$) e itera-se o método do subgradiente até que se atinja uma condição de convergência. Nesse ponto, aplica-se uma heurística Lagrangeana gulosa que calcula uma solução viável para o problema (passo c). Novamente, algumas variáveis são fixadas em 1, reduzindo ainda mais a dimensão do problema, e o processo se repete até que a condição de parada seja satisfeita (passo e).

3-PHASE($\overline{x}$,$\overline{u}$)
a. **Repita**
b. FaseSubgradiente($\overline{u}$)
c. $\overline{x} \leftarrow$ SoluçãoHeurística($\overline{u}$)
d. Fixação de Variáveis
e. **Até que** CondParada($\overline{x}$)
Fim

Figura 7.2: Subrotina 3-PHASE() da heurística CFT

7.2.2 Diferenças na Implementação

A implementação da heurística CFT feita neste trabalho difere, em alguns pontos, da implementação original, descrita em [3].

Na página 9 de [3], afirma-se que nas iterações do método do subgradiente que fazem parte da fase heurística da subrotina 3-PHASE a norma do subgradiente não é alterada. Este cuidado não foi tomado na implementação feita neste trabalho. Na página 11 de [3], aparecem outras duas características do algoritmo original que também não foram implementadas. A primeira delas é a utilização de uma estrutura de dados adicional (conjunto B) que contribui para diminuir o tempo de computação da solução heurística. A segunda característica diz respeito ao passo final da construção da solução heurística, onde são removidas eventuais colunas redundantes. A remoção exata de colunas redundantes presentes na solução recai em um outro problema de cobertura de conjuntos. Na implementação original, quando o número de colunas redundantes é inferior a 10, utiliza-se um algoritmo exato de enumeração para efetuar a remoção. Caso contrário, utiliza-se um algoritmo heurístico até que o número de colunas redundantes diminua para 10 e, a partir desse ponto, o algoritmo exato é acionado. Neste trabalho, utiliza-se apenas o algoritmo heurístico até que não haja mais colunas redundantes, o que pode resultar em uma solução de custo superior. Por fim, o parâmetro β, apresentado no passo 8 da Figura 7.1, teve seu valor alterado de 1.0 para 1.002.

Além dessas diferenças, decidiu-se incrementar a heurística original com duas sugestões extraídas do Capítulo 6 de [34]. De acordo com [3], o vetor de multiplicadores de Lagrange u^{k+1} é calculado, componente a componente, segundo a fórmula abaixo:

$$u_i^{k+1} = \max\left\{u_i^k + Ts_i(u^k), 0\right\}, \text{ para todo } i, \text{ onde } T = \lambda\frac{\text{UB} - L(u^k)}{||s(u^k)||^2}.$$

No cálculo de T, à medida que $L(u^k)$ se aproxima de UB, o tamanho do passo T vai ficando cada vez mais próximo de 0. Isto pode fazer com que o algoritmo caminhe muito devagar.

Sendo assim, substituiu-se UB por 1.02 UB. Por último, nota-se ainda que se $u_i^k = 0$ e $s_i(u^k) < 0$ tem-se que $u_i^{k+1} = 0$. Contudo, um fator igual a $s_i(u^k)^2$ terá sido incluído no denominador da equação que calcula o tamanho do passo T, afetando o seu valor. Portanto, sempre que $u_i^k = 0$ e $s_i(u^k) < 0$, faz-se $s_i(u^k) = 0$, para todo i, antes de se calcular T.

Deve-se notar que os ajustes de parâmetros e as demais alterações na implementação, descritos nesta seção, objetivaram melhorar o desempenho do programa. As escolhas foram feitas empiricamente, segundo o comportamento da heurística durante sessões extensas de experimentação.

7.2.3 Avaliação da Implementação

Um dos parâmetros utilizados para se avaliar a qualidade da implementação dessa heurística foi a comparação com os resultados (limitantes superiores e inferiores) obtidos pela implementação original, feita na Universidade de Bologna [3]. Na Tabela 7.1, pode-se observar que os resultados obtidos com a implementação feita neste trabalho (colunas sob o título *CFT IC*) são de boa qualidade. As instâncias testadas pertencem à biblioteca de instâncias de teste para algoritmos de Cobertura de Conjuntos da OR-Library[1]. Os valores m e n representam, respectivamente, o número de linhas e o número de colunas da matriz de coeficientes. Os tempos de execução reportados em Bologna (colunas sob o título *CFT Bologna*) referem-se a uma máquina DECstation 5000/240. Os tempos de execução sob o título *CFT IC* referem-se a um Pentium II 350 MHz.

7.3 Geração de Colunas com Programação Dinâmica

O algoritmo de Programação Dinâmica mencionado na Seção 3.2 do Capítulo 4 baseou-se num trabalho de Desrochers e Soumis [16]. A idéia principal consiste em se representar o problema de busca por uma coluna com custo reduzido negativo na forma de um problema em grafos. Este último problema, normalmente conhecido como *Constrained Shortest Path Problem* (CSPP), envolve a procura de caminhos mais curtos entre um par de vértices num *grafo direcionado acíclico* (GDA) com restrições de recursos nos arcos.

7.3.1 Construção do GDA

A construção do GDA, denotado por G, é feita da seguinte forma. Para cada viagem i, incluem-se em G dois vértices, I_i e F_i, correspondentes aos instantes de início e término de i, respectivamente, e um arco orientado de I_i para F_i. A este arco, denominado *arco de viagem*, atribui-se um custo $-u_i$ igual ao valor da variável dual associada à viagem i,

[1] `http://mscmga.ms.ic.ac.uk/info.html`.

Dados da OR-Library			CFT Bologna		CFT IC		
Inst.	Tam. ($m \times n$)	Ótimo	Sol	Tempo	LB	Sol	Tempo
4.1	200 × 1000	429	429	2.3	429	429	1.3
4.2	200 × 1000	512	512	1.1	512	512	2.9
4.3	200 × 1000	516	516	2.1	516	516	1.8
4.4	200 × 1000	494	494	9.8	494	495	6.9
4.5	200 × 1000	512	512	2.1	512	512	1.8
4.6	200 × 1000	560	560	19.3	557	560	22.5
4.7	200 × 1000	430	430	2.7	430	430	2.2
4.8	200 × 1000	492	492	22.2	488	493	9.0
4.9	200 × 1000	641	641	1.8	638	641	8.5
4.10	200 × 1000	514	514	1.8	514	516	20.1
5.1	200 × 2000	253	253	3.3	251	253	29.9
5.2	200 × 2000	302	302	2.3	300	302	13.0
5.3	200 × 2000	226	226	2.1	226	226	3.1
5.4	200 × 2000	242	242	1.9	241	242	24.1
5.5	200 × 2000	211	211	1.2	211	211	9.4
5.6	200 × 2000	213	213	0.9	213	213	22.0
5.7	200 × 2000	293	293	15.0	292	293	152.7
5.8	200 × 2000	288	288	1.6	287	288	132.8
5.9	200 × 2000	279	279	2.6	279	279	11.4
5.10	200 × 2000	265	265	1.3	265	265	2.6
6.1	200 × 1000	138	138	22.6	126	138	34.1
6.2	200 × 1000	146	146	17.8	136	146	20.1
6.3	200 × 1000	145	145	2.3	138	145	52.7
6.4	200 × 1000	131	131	1.8	128	131	23.1
6.5	200 × 1000	161	161	2.2	150	161	266.8

Tabela 7.1: Avaliação da qualidade da implementação da heurística CFT

multiplicado por -1. O valor de u_i é determinado no momento em que a relaxação linear do problema mestre corrente tiver sido resolvida. Arcos com custo zero são criados entre o vértice final de um arco de viagem i e o vértice inicial de um arco de viagem j, para todo par de viagens (i, j) tal que o horário de término de i é menor ou igual ao horário de início de j. Adicionalmente, outros arcos com custo zero são criados de modo a conectar um vértice *origem*, denotado por s, ao vértice inicial de cada arco de viagem. Da mesma forma, arcos de custo zero conectam todos os vértices finais de arcos de viagem a um vértice *destino*, denotado por t. A Figura 7.3 ilustra o grafo resultante.

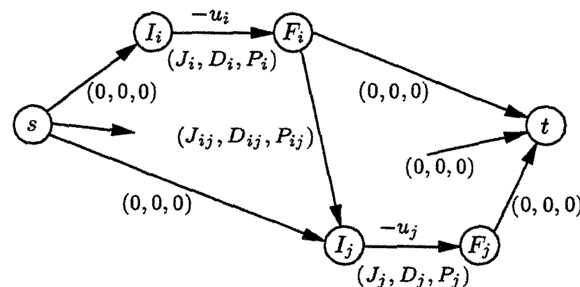


Figura 7.3: GDA com restrições de recursos nos arcos

Um caminho p de s para t, em G, representa uma jornada, denotada por D. O custo associado a p é igual a $\sum_{i \in D} -u_i$, já que somente arcos de viagem possuem custo diferente de zero. A partir da formulação em Programação Linear Inteira do problema de partição de conjuntos, sabe-se que o custo reduzido de uma jornada D é dado por $1 - \sum_{i \in D} u_i$. Sendo assim, para se obter uma jornada com custo reduzido negativo, procura-se por um caminho em G cujo custo seja inferior a -1. Entretanto, também é necessário garantir que este caminho represente uma jornada viável. Para isso, o algoritmo contabiliza o consumo de recursos de cada caminho durante a sua construção. Quando um novo arco é acrescentado a um caminho, este pode se tornar inviável. Caso o consumo de recursos acumulado até o momento, acrescido dos recursos consumidos pelo novo arco, ultrapasse os limites de algum recurso, tem-se uma inviabilidade. Se o caminho permanecer viável, o novo arco é adicionado ao caminho e os recursos consumidos por ele são contabilizados no consumo total de recursos do caminho. Além disso, o custo associado ao novo arco é incorporado ao custo do caminho. Este processo se repete, acrescentando novos arcos ao caminho, até que o nó destino t seja atingido. No caso aqui tratado, existem três recursos: o tempo de trabalho, o tempo de descanso e um valor binário indicando se o caminho equivale a uma dupla-pegada ou não. Portanto, a cada arco está associada uma tripla de números (J, D, P) indicando o consumo de cada um dos três recursos para aquele arco (vide Figura 7.3). Para os arcos que iniciam em s ou terminam em t, tem-se $J = D = P = 0$. Nos arcos de viagem,

J é a duração da viagem (em minutos) e D e P são iguais a zero. Por fim, arcos que conectam vértices de fim de viagem com vértices de início de viagem consomem recursos da seguinte forma. Sejam i e j as viagens envolvidas. Seja t o intervalo de tempo decorrido entre o término de i e o início de j. Se $t \leq 120$ min, tem-se $J = D = t$ e $P = 0$. Caso contrário, $J = 0$, $D = t$ e $P = 1$. Um caminho de s até um nó intermediário v é viável se, para o seu consumo acumulado de recursos (J, D, P), J não ultrapassa o tempo máximo de jornada e P vale no máximo 1 (no máximo uma dupla-pegada é tolerada). Um caminho de s a t é considerado viável se, além das condições anteriores, D é maior ou igual ao tempo mínimo de descanso.

7.3.2 Implementação do Algoritmo

Como caminhos distintos consomem recursos em quantidades diferentes, é necessário manter, em cada nó do grafo, uma lista com todos os caminhos viáveis que partem de s e chegam até ele. Um caminho só pode ser descartado dessa lista se for desvantajoso, em termos de consumo de recursos, quando comparado a outro caminho da mesma lista. Por exemplo, sejam dois caminhos p e q, cujos consumos de recursos são (J_p, D_p, P_p) e (J_q, D_q, P_q) e cujos custos são C_p e C_q, respectivamente. Diz-se que p é *desvantajoso* com relação a q se e somente se as quatro condições abaixo são verificadas simultaneamente:

$$\begin{aligned} C_q &\bowtie C_p\,, \\ J_q &\bowtie J_p\,, \\ P_q &\bowtie P_p \text{ e} \\ D_p &\bowtie D_q, \text{ com } D_p \text{ menor que o descanso mínimo}\,, \end{aligned}$$

onde $\bowtie \in \{\leq, <\}$ e no máximo três $\bowtie$'s são iguais a $\leq$. Em outras palavras, seja v um nó do GDA e seja p um caminho viável de s a v ainda não inserido na lista de caminhos viáveis do nó v. Para qualquer caminho viável q pertencente à lista de caminhos viáveis do nó v, observa-se o seguinte: se o caminho p possuir um custo inferior ao custo do caminho q ou se ele implicar uma jornada de trabalho menor ou com menos duplas-pegadas que o caminho q, então p deve ser inserido na lista de caminhos viáveis do nó v. Com relação ao descanso, dá-se preferência a um caminho que implique em um número maior de minutos de repouso. Vale ressaltar entretanto que, quando $D_p < D_q$ mas D_p já é maior ou igual ao tempo mínimo de descanso, não há mais como violar a restrição de descanso e, portanto, não faz sentido descartar p somente por causa dessa condição. Para garantir que o caminho inteiro possa ser recuperado ao se alcançar o nó t, mantém-se um apontador para o nó anterior em cada nó dos caminhos sendo construídos. Vale notar que, quando o algoritmo termina, a lista final de caminhos viáveis do nó t permite que se obtenha não só o caminho

de menor custo como também diversos outros caminhos que porventura também tenham custo reduzido negativo.

CSP(G)
    Construa(G)
    Para todo vértice v na ordem topológica **faça**
        Para todo caminho viável p de s a v **faça**
            Para todo arco a que sai de v **faça**
                Se factível(p,a) **então**
                    consome(p,a)
Fim

Figura 7.4: Algoritmo de caminhos mais curtos no GDA

O algoritmo implementado aparece na Figura 7.4. Trata-se de uma adaptação de um algoritmo de caminhos mais curtos em GDAs, retirado do Capítulo 25 de [12]. Este algoritmo tira vantagem do fato que, em um GDA, mesmo que haja arcos com pesos negativos, não há ciclos com pesos negativos. Além disso, o algoritmo pressupõe que os vértices serão percorridos de acordo com uma ordenação topológica, a qual pode ser facilmente extraída dos vértices do grafo G. A rotina factível(p,a) determina se o arco a pode ser acrescentado ao caminho p sem violar o consumo de algum recurso. A rotina consome(p,a) acrescenta efetivamente o arco a ao caminho p, criando outros caminhos viáveis.

# Viagens	Tempo de *pricing*	Tempo total	$\frac{\text{Tempo de } pricing}{\text{Tempo total}}$ %
20	0.04	0.07	57.1
30	0.43	0.52	82.7
40	8.82	9.10	96.9
50	00:01:26	00:01:29	96.9
60	00:07:45	00:07:54	98.2
70	00:43:58	00:44:19	99.2
80	03:53:06	03:53:58	99.6
90	08:18:11	08:18:53	99.9
100	15:07:22	15:08:55	99.8

Tabela 7.2: Tempo de *pricing* com Geração de Colunas via GDA sobre a OS 2222

7.3.3 Avaliação de Desempenho

A principal razão para o desempenho insatisfatório deste algoritmo está muito provavelmente associada ao fato de que as restrições de recursos são relativamente fracas. Em se tratando de um algoritmo pseudo-polinomial, o espaço de estados (número de caminhos viáveis) em cada nó do grafo tem o potencial de crescer exponencialmente com o tamanho da entrada. Para as instâncias reais consideradas nessa dissertação, o número de caminhos factíveis que o algoritmo precisa manter armazenados é grande. Com isso, o tempo gasto na resolução do problema de caminhos mais curtos sobre o grafo (que equivale ao tempo de *pricing*) é muito elevado, sendo responsável por mais de 90% do tempo total de execução do algoritmo de *Branch-and-Price*, na média. A Tabela 7.2 dá suporte a essa observação.

Parte III

Considerações Finais

Capítulo 8

Conclusões

Ao mesmo tempo em que procurou estudar o problema geral de escalonamento de mão-de-obra no âmbito de uma empresa de transporte coletivo urbano, essa dissertação teve como objetivo adicional comparar técnicas de Programação Matemática e Programação por Restrições.

Com relação ao problema de *crew scheduling*, ao se considerarem isoladamente as duas técnicas de resolução mencionadas, percebe-se uma clara vantagem a favor de Programação Matemática. Uma heurística Lagrangeana foi responsável por encontrar soluções para as instâncias de maior porte. Entretanto, nenhuma das abordagens isoladas foi capaz de tratar instâncias reais do problema. Todavia, adotando-se um modelo híbrido, foi possível encontrarem-se soluções ótimas para instâncias muito maiores.

No que diz respeito ao problema de *crew rostering*, os resultados parecem se inverter. Dentre as abordagens isoladas, Programação por Restrições apresentou os melhores resultados, superando em desempenho e qualidade de solução um modelo baseado em Programação Linear Inteira. Novamente, quando o interesse se voltou para a busca de soluções ótimas, as duas técnicas isoladas foram insuficientes. Um outro algoritmo híbrido, conceitualmente muito semelhante ao anterior, foi mais uma vez capaz de construir soluções ótimas. Nesse caso, contudo, alguns problemas com a formulação do problema impediram a solução de instâncias realmente grandes. Apesar disso, o aperfeiçoamento desse algoritmo aparenta ser o caminho mais promissor para a obtenção de resultados aplicáveis no dia-a-dia da empresa.

As abordagens de Programação por Restrições foram sempre superiores às técnicas mais tradicionais de otimização quanto à facilidade de desenvolvimento e manutenção dos modelos, como previsto inicialmente. Por outro lado, um dos pontos fracos de tais abordagens mostrou-se bastante claro em todos os casos considerados: a busca de soluções ótimas deteriora consideravelmente o desempenho dos programas baseados puramente em Programação por Restrições. Daí surge, portanto, a necessidade de se combinarem essas técnicas a outras metodologias existentes.

O campo de estudos associado ao desenvolvimento de algoritmos híbridos que combinam Programação por Restrições e outras técnicas de Programação Matemática ainda é muito incipiente. Já se tem conhecimento de alguns problemas para os quais as melhores soluções são obtidas com o auxílio de algoritmos híbridos [21, 35], e os resultados desse trabalho constituem mais uma contribuição nessa direção.

De uma maneira geral, acredita-se que os objetivos dessa dissertação foram alcançados satisfatoriamente. Instâncias reais do problema original foram resolvidas com tempos de computação aceitáveis. Além disso, foi possível conhecer mais a fundo todas as técnicas utilizadas e as dificuldades intrínsecas a cada tipo de problema tratado. Desse modo, contribuiu-se positivamente para o sucesso de futuros empreendimentos nessa área.

Capítulo 9

Trabalhos Futuros

O trabalho desenvolvido nesta dissertação pode ser estendido de diversas maneiras:

- Como o estudo procurou concentrar-se sobre instâncias reais do problema de escalonamento de mão-de-obra, o número de experimentos realizados foi relativamente pequeno. Seria interessante, contudo, gerar um conjunto maior de instâncias pseudo-aleatórias, seguindo aproximadamente o mesmo padrão de distribuição de viagens de uma instância real, e avaliar o desempenho e a robustez dos mesmos algoritmos sobre esses dados.

- Existem diversos estudos a respeito de formas mais inteligentes de *backtracking* [22] e sobre o uso de informações obtidas durante a procura de soluções de modo a permitir uma redução mais eficiente do espaço de busca [29]. É possível que a implementação dessas idéias traga melhorias no desempenho dos algoritmos baseados em Programação por Restrições utilizados neste trabalho.

- Os algoritmos aqui apresentados não dispõem de interfaces de operação adequadas para um usuário final não especializado. O desenvolvimento interfaces gráficas mais amigáveis certamente aproximaria este trabalho de sua utilização efetiva num ambiente comercial.

- Normalmente, as empresas dispõem de sistemas integrados para o controle de cadastro de funcionários, folha de pagamento, faltas dos empregados etc. Como complementação do item anterior, seria válido efetuar-se a integração dos algoritmos de escalonamento de mão-de-obra aqui desenvolvidos a sistemas maiores de modo a unificar essas funcionalidades, ganhando-se em produtividade.

- A causa principal para o desempenho insatisfatório do algoritmo híbrido de geração de colunas sobre o problema de *crew rostering* parece residir no fato de que o problema de partição de conjuntos mostrou-se altamente degenerado. Sendo assim, seria

interessante estudar a aplicabilidade, a este caso, de algumas técnicas existentes para se tratar esse tipo de dificuldade.

- Por fim, seria desejável estudar-se a adaptação do algoritmo de *Branch-and-Price* híbrido a outros problemas importantes onde o subproblema de geração de colunas não possa ser resolvido eficientemente através de abordagens convencionais. Além disso, a mesma idéia também poderia ser aplicada ao problema de geração de cortes, já que este último pode ser encarado como um dual do problema de geração de colunas.

Bibliografia

[1] D. Bertsimas e J. N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

[2] L. Bianco, M. Bielli, A. Mingozzi, S. Ricciardelli, e M. Spandoni. A heuristic procedure for the crew rostering problem. *European Journal of Operational Research*, 58(2):272–283, 1992.

[3] A. Caprara, M. Fischetti, e P. Toth. A heuristic method for the set covering problem. Relatório Técnico OR-95-8, DEIS, Universidade de Bolonha, Itália, 1995. Publicado em *Operations Research* número 47, 1999.

[4] A. Caprara, M. Fischetti, P. Toth, e D. Vigo. Modeling and solving the crew rostering problem. Relatório Técnico OR-95-6, DEIS, Universidade de Bolonha, Itália, 1995. Publicado em *Software Practice and Experience* número 28, 1998.

[5] A. Caprara, M. Fischetti, P. Toth, D. Vigo, e P. L. Guida. Algorithms for railway crew management. *Mathematical Programming*, 79:125–141, 1997.

[6] A. Caprara, F. Focacci, E. Lamma, P. Mello, M. Milano, P. Toth, e D. Vigo. Integrating constraint logic programming and operations research techniques for the crew rostering problem. Relatório Técnico OR-96-12, DEIS, Universidade de Bolonha, Itália, 1996. Publicado em *Operations Research* número 46, 1999.

[7] P. Carraresi e G. Gallo. A multi-level bottleneck assignment approach to the bus drivers' rostering problem. *European Journal of Operational Research*, 16(2):163–173, 1984.

[8] A. Chamard, A. Fischer, D.-B. Guinaudeau, e A. Guillaud. CHIC lessons on CLP methodology, 1995.
Disponível em http://www.icparc.ic.ac.uk/eclipse/reports/CHIC_Methodology.html.

[9] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, e J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2), maio de 1999.

[10] W. F. Clocksin e C. S. Mellish. *Programming in Prolog.* Springer, 1994.

[11] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, 1990.

[12] T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1998.

[13] K. Darby-Dowman e J. Little. The significance of constraint logic programming to operations research. Em M. Lawrence e C. Wilsdon, editores, *Operational Research Society Tutorial Papers*, pp. 20–45, 1995.

[14] K. Darby-Dowman e J. Little. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal on Computing*, 10(3), 1998.

[15] K. Darby-Dowman, J. Little, G. Mitra, e M. Zaffalon. Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem. *Constraints*, 1(3):245–264, 1997.

[16] M. Desrochers e F. Soumis. A column generation approach to the urban transit crew scheduling problem. *Transportation Science*, 23(1), 1989.

[17] A. Aggoun et al. *ECLiPSe User Manual, Release 4.0.* European Computer-Industry Research Center (ECRC GmbH), julho de 1998.
Disponível em http://www.ecrc.de/eclipse/eclipse.html.

[18] M. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[19] F. Focacci, A. Lodi, M. Milano, e D. Vigo. Solving TSP through the integration of OR and CP techniques. *Electronic Notes in Discrete Mathematics*, 1, 1998. Workshop sobre Large Scale Combinatorial Optimization, CP'98, Pisa, Itália. Disponível em http://www.elsevier.nl:80/cas/tree/store/disc/sub/endm/store/disc1/disc1003.pdf.

[20] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, e M. Wallace. Constraint logic programming - an informal introduction. Relatório Técnico ECRC-93-5, ECRC, Munique, Alemanha, 1993.
Em ftp://ftp.ecrc.de/pub/eclipse/ECRC_tech_reports/reports/ECRC-92-1.ps.Z.

[21] C. Gervet. Large combinatorial optimization problems: a methodology for hybrid models and solutions. Em *Journées Francophones de Programmation en Logique et par Contraintes*, 1998. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[22] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, (1):25–46, 1993.
Disponível em ftp://ftp.cirl.uoregon.edu/pub/users/ginsberg/papers/dynamic.ps.gz.

[23] M. Held e M. R. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[24] K. L. Hoffman e M. Padberg. Solving airline crew-scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[25] J. K. Jachnik. Attendance and rostering system. Em A. Wren, editor, *Computer Scheduling of Public Transport*, pp. 337–343. North-Holland Publishing Co., 1981.

[26] J. Jaffar e J. L. Lassez. Constraint logic programming. pp. 111–119, Munique, Alemanha, janeiro de 1987. Anais do *14th ACM Symposium on Principles of Programming Languages.*

[27] J. Jaffar e M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, pp. 503–581, 1994.

[28] C. Lassez. Constraint logic programming. *Byte Magazine*, 2(9):171–176, 1987.

[29] J. Lever, M. Wallace, e B. Richards. Constraint logic programming for scheduling and planning. *British Telecom Technical Journal*, (13):73–81, 1995. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[30] K. Marriott e P. J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[31] G. L. Nemhauser e L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, 1988.

[32] A. Nerode e R. A. Shore. *Logic for Applications.* Springer, 1997.

[33] C. H. Papadimitriou e K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, Inc., 1998.

[34] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Wiley, 1993.

[35] R. Rodosek, M. Wallace, e M. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. Em *Annals of Operations Research. Recent Advances in Combinatorial Optimization*, 1998. Aceito para publicação. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[36] B. M. Smith. A tutorial on constraint programming. Relatório Técnico 95.14, School of Computer Studies, Universidade de Leeds, Inglaterra, 1995. Disponível em ftp://agora.leeds.ac.uk/scs/doc/reports/1995/95_14.ps.Z.

[37] B. M. Smith, S. C. Brailsford, P. M. Hubbard, e H. P. Williams. The progressive party problem: Integer linear programming and constraint programming compared. Em *Lecture Notes in Computer Science*, vol. 976, pp. 36–52, 1995. Anais da *First International Conference on the Principles and Practice of Constraint Programming, CP'95*.

[38] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. The MIT Press, 1989.

[39] F. Vanderbeck. *Decomposition and Column Generation for Integer Programming.* Tese de Doutorado, Universidade Católica de Louvain, CORE, setembro de 1994.

[40] M. Wallace. Applying constraints for scheduling. Em B. Mayoh e J. Penjaam, editores, *NATO ASI Series.* Springer-Verlag, 1994. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[41] M. Wallace. Constraint programming. Em L. Jay, editor, *The Handbook of Applied Expert Systems.* CRC Press, 1998. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[42] M. Wallace, S. Novello, e J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1), 1997. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[43] M. Wallace e A. Veron. Two problems - two solutions: One system - ECLiPSe. Em *IEE Colloquium on Advanced Software Technologies for Scheduling*, 1993. Disponível em http://www.icparc.ic.ac.uk/papers_byauthor.html.

[44] T. H. Yunes, A. V. Moura, e C. C. de Souza. Exact solutions for real world crew scheduling problems. Em *INFORMS Fall 1999 Meeting*, Filadélfia, PA, EUA, novembro de 1999. Apresentação convidada.

[45] T. H. Yunes, A. V. Moura, e C. C. de Souza. Solving large scale crew scheduling problems with constraint programming and integer programming. Relatório Técnico IC-99-19, Instituto de Computação, Universidade Estadual de Campinas, 1999. Disponível em http://goa.pos.dcc.unicamp.br/otimo.

[46] T. H. Yunes, A. V. Moura, e C. C. de Souza. A hybrid approach for solving large scale crew scheduling problems. Em *Lecture Notes in Computer Science*, vol. 1753, pp. 293–307, Boston, MA, EUA, janeiro de 2000. Anais do *Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*.

[47] T. H. Yunes, A. V. Moura, e C. C. de Souza. Modeling and solving a crew rostering problem with constraint logic programming and integer programming. Relatório Técnico IC-00-04, Instituto de Computação, Universidade Estadual de Campinas, 2000. Disponível em http://goa.pos.dcc.unicamp.br/otimo.

[48] T. H. Yunes, A. V. Moura, e C. C. de Souza. Solving a real world crew rostering problem with integer programming and constraint logic programming models. Em *17th International Symposium on Mathematical Programming (ISMP'00)*, Atlanta, GA, EUA, agosto de 2000. Apresentação convidada.

[49] T. H. Yunes, A. V. Moura, e C. C. de Souza. Solving very large crew scheduling problems to optimality. Em *14th ACM Symposium on Applied Computing (SAC'00)*, Como, Itália, março de 2000.

Apêndice A

Considerações sobre Programação por Restrições

Este apêndice evoluiu como um exercício de estudo visando esclarecer algumas dúvidas surgidas durante o desenvolvimento dessa dissertação. Diversas idéias interessantes e sugestões relativamente genéricas foram coletadas a partir de textos introdutórios e avançados sobre Programação por Restrições, e a partir da experiência adquirida durante este trabalho. O objetivo principal foi condensar, em poucas páginas, informações que possam auxiliar os interessados em iniciar seus estudos nessa área.

A.1 Introdução

Este capítulo procurou acumular informações relevantes às seguintes questões: que tipos de problemas devem ser tratados com Programação por Restrições? O que se deve e o que não se deve fazer ao se trabalhar com Programação por Restrições? Terá a comunidade atingido um ponto em que se pode apontar seguramente que aspectos de um problema o tornam mais (ou menos) adequado para uma abordagem através de Programação por Restrições? Infelizmente, as respostas para essas perguntas não são simples. Ainda há muito trabalho a ser feito até que o conhecimento e a experiência no campo de Programação por Restrições fiquem maduros o bastante. Contudo, o número crescente de pesquisadores e aplicações que vêm surgindo nesse campo relativamente novo já contribuíram com resultados significativos. É claro que algumas conclusões são demasiado específicas e intimamente relacionadas a um problema ou outro. Não obstante, existe um certo consenso a respeito de alguns pontos importantes. As considerações aqui apresentadas foram agrupadas de acordo com sua aplicabilidade, incluindo-se apenas regras que possam ser úteis em quaisquer contextos.

Assume-se que o leitor conhece os conceitos básicos a respeito de Programação por Restrições baseada em lógica. A literatura introdutória sobre o assunto é bastante extensa.

Veja, por exemplo, [11, 13, 20, 27, 28, 30, 36, 41].

A.2 Recomendações Gerais

Acredita-se que quatro aspectos principais devem ser levados em consideração quando se deseja resolver um problema através de uma abordagem em Programação por Restrições. Desse modo, as recomendações a seguir foram divididas em quatro categorias.

A.2.1 Análise do Problema

Antes de se tomar decisões práticas a respeito do processo de desenvolvimento e dos detalhes de implementação (e.g. plataforma e linguagem de programação) é importante começar com uma visão mais abstrata do problema em questão. Algumas perguntas chave precisam ser respondidas. As respostas a essas perguntas podem ser bastante úteis, acabando por economizar doses consideráveis de esforço nas etapas subseqüentes.

Atente para simetrias: Sempre que possível, deve-se tentar reduzir o espaço de busca. Simetrias são um bom exemplo disso. Ao se procurar por soluções ótimas para um problema, é possível que diversas atribuições de valores às variáveis sejam equivalentes quanto ao valor da função objetivo. Além disso, a eliminação de soluções simétricas também contribui para a redução do número de pontos de falha (basta lembrar que em meio às soluções inviáveis também há simetrias) [8, 15, 29, 40].

Quanto mais restrito melhor: O desempenho de um *solver* de Programação por Restrições é função de três fatores principais: o número de variáveis e o tamanho de seus domínios; o número de restrições e sua *rigidez* (*hard* ou *soft*), e a estratégia de busca (escolha de variáveis e valores) [35]. Um espaço de busca pequeno é claramente melhor que um espaço de busca consideravelmente maior (e.g. 100 variáveis binárias correspondem a 2^{100} configurações possíveis). Ademais, restrições apertadas são, em geral, mais vantajosas pois diminuem o espaço de busca mais efetivamente [14].

Feitio de um problema adequado para Programação por Restrições:
De acordo com [8]:

- Número relativamente pequeno de variáveis;
- Muitas restrições *hard* (desde que resultem em boa propagação);
- Poucas restrições *soft* (pois elas normalmente introduzem *pontos de escolha* ou são tratadas na forma de funções objetivo que auxiliam pouco na propagação de informação);

- Grandes chances de evolução (requerendo manutenção freqüente);
- Interatividade (intervenção do usuário).

Decisões e combinatória: Aplicações de suporte à decisão que envolvem problemas combinatórios podem ser modeladas particularmente bem com Programação em Lógica. Nessas aplicações, é comum existirem relações entre entidades diferentes do problema, além da necessidade de se modelarem aspectos do problema com o auxílio de variáveis lógicas. Assim, uma linguagem de Programação por Restrições baseada em lógica é bastante adequada para esses casos [42].

Incompletude na busca: Se o espaço de busca é demasiadamente grande, deve-se considerar como opção uma busca que não seja exaustiva. Pode ser mais vantajoso explorar apenas algumas possibilidades significativas em cada nó da árvore de busca (até uma certa profundidade máxima) e eliminar o restante do que tentar explorar todas as possibilidades sob a imposição de um limite de tempo. Em outras palavras, pode ser melhor explorar uma árvore menos densa por inteiro do que proceder com uma busca exaustiva em uma região muito localizada da árvore completa [8]. Em certas situações, é melhor não se preocupar com o ótimo global que, em grande parte dos casos, não goza de robustez. Ou seja, o ótimo global é bastante sensível a pequenas alterações nos parâmetros de execução ou nos dados de entrada.

A.2.2 O Modelo

Não há dúvidas de que um bom modelo faz a diferença. O esforço dispensado na fase de modelagem é certamente mais um passo em direção a um programa eficiente [35]. Porém, o fato de se possuir apenas um modelo com baixo desempenho não é motivo para se desistir de Programação por Restrições. Talvez seja apenas uma questão de pequenos ajustes.

Existem (potencialmente) infinitas maneiras de se expressar um dado problema utilizando-se conjuntos alternativos de variáveis (com seus respectivos domínios) e restrições sobres essas variáveis. Claramente, a melhor escolha (ou pelo menos uma escolha razoável) nem sempre é facilmente reconhecida. Quanto maior a prática, melhores serão os modelos construídos.

Tente primeiro o modelo mais natural: Muitos programadores têm a tendência de enfatizar demais a importância de um código eficiente e negligenciam aspectos indispensáveis tais como legibilidade e facilidade de manutenção. Será que realmente vale a pena? Às vezes, um código complicado e obscuro é apenas ligeiramente mais rápido que uma versão consideravelmente mais simples. Um bom conselho é: ao se buscar melhorias de desempenho, deve-se guardar o código original. Pode ser que ele volte a ser útil [17].

Tente um modelo com redundâncias: Informação redundante pode ser útil na redução do espaço de busca [38]. Em [9], os autores conjecturam que problemas em que a fase de busca é responsável por uma parte significativa do tempo total de computação são os que mais podem se beneficiar de modelos redundantes. A idéia principal por trás de modelos redundantes é criar duas (ou mais) visões do mesmo problema e, em seguida, conectá-las através das chamadas *channeling constraints* [9]. O modelo combinado resultante pode apresentar um efeito maior de propagação. Isto porque, além de possuir a propagação usual entre as variáveis *dentro* de cada modelo (visão), ele também possui a propagação *entre* os modelos através das restrições que correlacionam as variáveis dos dois (ou mais) lados.

Utilize variáveis informativas: Poucas variáveis com domínios maiores são tipicamente melhores do que muitas variáveis com domínios pequenos. Variáveis binárias devem ser evitadas sempre que possível, pois elas aumentam artificialmente o tamanho do espaço de busca [8].

Não esqueça o princípio *first-fail*: Seguindo-se o princípio conhecido do *gargalo* ou *first-fail*, deve-se primeiro criar protótipos dos subproblemas mais restritos. Caso eles sejam factíveis, o problema como um todo poderá também ser factível [8].

Pense em modelos mais sofisticados: Quando nada mais parece funcionar, deve-se admitir que talvez o problema sendo considerado não seja tão simples como aparentava inicialmente.

A.2.3 Conselhos de Programação

O estilo de programação está profundamente relacionado ao gosto pessoal do programador, por um lado, e às primeiras lições que se aprende na sala de aula, por outro lado. A metodologia de ensino tem uma influência grande sobre o tipo de programador que se acaba formando. Ao invés de tentar livrar-se dos maus hábitos, uma boa idéia é tentar começar da maneira correta, e algumas sugestões podem ser úteis.

Utilize restrições predefinidas: Geralmente, as linguagens de Programação por Restrições disponíveis fornecem as chamadas *restrições predefinidas*. Essencialmente, são restrições mais complexas que, na forma de um único comando, equivalem a um conjunto de restrições mais primitivas. É recomendado utilizá-las sempre que possível. A representação do problema fica mais concisa e o processo de resolução pode ficar mais eficiente. É bom lembrar que, na implementação dessas restrições, requisitos de desempenho já foram levados em consideração pelos projetistas da linguagem [15, 29].

Adie escolhas ao máximo: Algoritmos de propagação de restrições têm complexidade polinomial. Por outro lado, a busca, na maioria das vezes, tem complexidade exponencial [8]. A imposição de restrições sobre as variáveis contribui para a redução dos domínios e, por conseguinte, diminui o espaço de busca. Dado que uma escolha errada foi tomada, para se identificar o erro é preciso explorar o espaço de busca completo correspondente à subárvore de possibilidades que seguem aquela escolha. É claro que quanto mais próximo da raiz essa escolha estiver na árvore de busca, mais tempo será necessário para que o mecanismo de *backtracking* consiga testar atribuições alternativas viáveis. As restrições com maior efeito limitador devem ser impostas antes das restrições mais relaxadas [30, 38].

Certamente, o poder de se expressar disjunções diretamente (em oposição a modelos de Programação Linear Inteira, que tratam disjunções indiretamente) é bem-vindo. Mas não se deve esquecer o fato de que disjunções introduzem pontos de escolha, potencializando sua ineficiência inerente [15, 35, 40].

Incorpore conhecimento específico ao programa: Em Programação por Restrições, a busca pode ser controlada não somente através da atribuição de valores às variáveis de uma determinada forma, mas também através da identificação de heurísticas específicas que possam guiar melhor essa atribuição. Desse modo, regiões mais promissoras do espaço de busca podem ser percorridas [14, 29, 35, 40]. Essas heurísticas normalmente advêm da experiência acumulada por uma pessoa durante anos de trabalho e podem ser acrescentadas ao modelo graças à flexibilidade proporcionada pela linguagem [29, 42].

Experimente usar assertivas do tipo *nogood*: Em certos casos é possível extrair informação útil de um ramo da árvore de busca que produziu uma falha. Algumas atribuições inviáveis de valores a subconjuntos de variáveis podem ser armazenadas e usadas mais adiante, ajudando a podar a árvore. Por exemplo, pode-se introduzir restrições adicionais ou eliminar padrões de valores inviáveis a priori [29, 40, 42, 43]. Felizmente, a inclusão dessas idéias como uma extensão de um programa existente é uma tarefa relativamente simples.

Não utilize recursos poderosos em excesso: Usualmente, no que diz respeito à implementação, deve-se fazer uso de restrições poderosas (predefinidas) apenas quando seu poder expressivo e capacidade de inferência são realmente necessários. Restrições mais simples devem ter sempre preferência sobre restrições mais complexas quando forem capazes de produzir as mesmas deduções [8]. Por exemplo, `X + Y + Z #<= 1` é melhor que `atmost(1,[X,Y,Z],1)`.

Monitore o número de pontos de escolha e *backtracks*: Em alguns *solvers* de Programação por Restrições existem subrotinas que retornam o número de pontos de escolha criados e o número de *backtracks* que ocorreram durante a execução de um programa ou uma parte dele. Ao se testarem instâncias pequenas com o intuito de validar e extrair erros do programa, é possível que algumas mudanças no código resultem num aumento do tempo de execução. Todavia, isso não significa que essas mudanças devem ser descartadas. O número de pontos de escolha e *backtracks* pode ser uma medida mais precisa de desempenho. Isto porque, para instâncias maiores, pode haver um decréscimo substancial no tempo de computação que não se mostrou aparente em face de instâncias menores [9].

A.2.4 Programação por Restrições × Programação Matemática

Quando Programação por Restrições começou a ganhar importância como uma ferramenta de resolução de problemas combinatórios, também foi vista como uma ameaça: idéias do campo da Inteligência Artificial competindo contra métodos tradicionais de Pesquisa Operacional [21]. Assim que a comunidade de Pesquisa Operacional passou a compreender melhor o potencial de Programação por Restrições, esta situação se modificou.

Experimentos feitos com uma variedade de problemas contribuíram para mostrar que, ao invés de serem técnicas adversárias, os métodos de Programação por Restrições e Pesquisa Operacional devem ser entendidos como estratégias colaborativas [15, 21, 35]. Seus algoritmos, quando tomados isoladamente, são certamente mais adequados para tipos específicos de problemas. Entretanto, em alguns casos, nenhuma dessas técnicas individualmente é capaz de superar uma abordagem híbrida bem elaborada.

Viável × ótimo: Às vezes, uma solução viável qualquer é tudo de que se precisa. Não importa se ela é ótima ou não. Nessas situações, Programação por Restrições tende a ser a melhor escolha [15, 37]. Além disso, funções objetivo complexas têm baixo poder de propagação quando se trabalha com domínios finitos [8, 19].

Expressividade natural pode ser útil: Programação por Restrições oferece a grande vantagem de uma formulação natural. Essa qualidade advém das suas origens em linguagens declarativas de Programação em Lógica. Estratégias eficientes de busca são geralmente baseadas em conhecimento sobre a estrutura do problema e esse tipo de visão é obtido mais facilmente quando se raciocina em termos dessa estrutura. Uma formulação natural e declarativa pode ser de grande valia [15, 19].

Hierarquia de restrições: Programação em Lógica permite que novas restrições sejam escritas em termos de restrições preexistentes. Com isso, tem-se um tipo de hierarquia que aumenta a organização, legibilidade e facilidade de reutilização de código. Por

outro lado, modelos de Programação Matemática não permitem a criação desse tipo de hierarquia [42].

Heterogeneidade de problemas: Programação por Restrições é um candidato natural para problemas heterogêneos mas *não* para problemas homogêneos onde um objeto global pode ser identificado (por exemplo, um grafo). Tipicamente, neste último caso, algoritmos existentes de Pesquisa Operacional tendem a resolver o problema com muito mais eficiência [8].

Dê preferência a Programação por Restrições quando: De acordo com [8]:

- Seu problema se encaixa num modelo existente mas é grande demais. Deve-se então lançar mão de uma busca heurística. Como Programação por Restrições pode explorar melhor o conhecimento específico sobre o problema, heurísticas funcionam bem com restrições;
- Seu problema não se encaixa em nenhum modelo existente (ou em nenhum modelo que você conhece). Nesse caso, Programação por Restrições é *a* ferramenta, pois:
  - É declarativa por natureza, o que ajuda quando o problema é mal definido e/ou as restrições são alteradas com freqüência;
  - Não é necessário implementar uma técnica de busca. Basta usar os algoritmos existentes;
  - Não se está restrito a restrições lineares;
  - O efeito real que novas restrições causam no tamanho do espaço de busca pode ser bem avaliado.

Considere um algoritmo híbrido: Como mencionado no início dessa seção, um problema pode possuir certas características intrínsecas que só podem ser atacadas eficientemente por uma abordagem híbrida. Se este é o caso, não se deve hesitar em procurar por casos semelhantes na literatura ou, ainda, criar um novo tipo de modelo cooperativo. Por questões de eficiência e escalabilidade, há uma conscientização crescente de que se precisa ir além do paradigma de Programação por Restrições, combinando, de forma híbrida, modelos e metodologias diferentes [15, 21, 35, 46].

A.3 Aplicações

É difícil afirmar qual a estratégia mais adequada para determinados tipos de problemas. Contudo, em alguns casos específicos, a prática mostra vantagens e desvantagens de uma

abordagem de Programação por Restrições com relação a uma abordagem de Programação Linear Inteira.

Exemplos a favor: Estes são alguns problemas em que Programação por Restrições superou Programação Linear Inteira: Cabinet Assignment Problem [13]; Progressive Party Problem [37]; escalonamento de partidas de golfe [14]; manufatura [14].

Exemplos contra: Aqui, as evidências são mais favoráveis à Programação Linear Inteira: Partição de Conjuntos [24, 35]; Agregação de Fluxo [14]; algumas instâncias de problemas de escalonamento de mão-de-obra [14].

A.4 Conclusões

O campo da Programação por Restrições vem evoluindo rapidamente. Foram reunidas aqui algumas recomendações que podem auxiliar analistas e programadores inexperientes, contribuindo para o desenvolvimento e melhoria de suas habilidades de modelagem e implementação. Como exposto anteriormente, não se espera que o leitor siga *rigorosamente* as idéias aqui apresentadas, mas sim que elas sirvam como guias no ataque a problemas específicos. Em caso de dúvida, a melhor abordagem continua sendo testar diferentes estratégias.

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

Apêndice B

Traduções Adotadas no Texto

A seguir são apresentadas as traduções para o português de alguns termos utilizados neste texto, seguidas das suas versões em inglês mais comumente encontradas nos textos técnicos sobre o assunto.

bem sucedida — *successful*

consistência parcial — *partial consistency*

descanso — *idle time*

dupla-pegada — *two-shift duty* ou *split-shift duty* ou *split shift*

falha — *failure*

jornada — *duty*

jornada viável — *feasible duty*

manipulação de restrições — *constraint handling*

mochila — *knapsack*

ponto de chegada — *arrival depot*

ponto de partida — *departure depot*

pontos de escolha — *choice points*

Programação em Lógica — *Logic Programming*

propagação de restrições — *constraint propagation*

regra de computação — *computation rule*

regras de restrições — *constraint rules*

repositório de restrições — *constraint store*

rigidez — *strength*

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE