**Programação por restrições e
escalonamento baseado em restrições:
Um estudo de caso na programação de
recursos para o desenvolvimento de
poços de petróleo**

Thiago Serra Azevedo Silva

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

Programa: Ciência da Computação
Orientador: Profa. Dra. Yoshiko Wakabayashi

São Paulo, julho de 2012

**Programação por restrições e escalonamento baseado em restrições:
Um estudo de caso na programação de recursos para o
desenvolvimento de poços de petróleo**

Esta dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa realizada por Thiago Serra Azevedo Silva em 23/05/2012.

O original encontra-se disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Profa. Dra. Yoshiko Wakabayashi (orientadora) – IME-USP
- Prof. Dr. Walter Figueiredo Mascarenhas – IME-USP
- Prof. Dr. Arnaldo Vieira Moura – IC-UNICAMP

Resumo

SERRA, T. **Programação por restrições e escalonamento baseado em restrições: Um estudo de caso na programação de recursos para o desenvolvimento de poços de petróleo.** 2012. 151 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

O objetivo dessa dissertação é apresentar um problema de otimização do uso de recursos críticos no desenvolvimento de poços de petróleo marítimos e a técnica empregada para a abordagem proposta ao problema. A revisão da técnica de Programação por Restrições é feita analisando aspectos relevantes de modelagem, propagação, busca e paradigmas de programação. A especialização da técnica para problemas de escalonamento, o Escalonamento Baseado em Restrições, é descrita com ênfase nos paradigmas descritivos e nos mecanismos de propagação de restrições. Como subsídio ao uso da técnica em outros problemas, a linguagem comercial de modelagem OPL é apresentada no Apêndice. O objetivo da abordagem ao problema é obter um escalonador para maximizar a produção de óleo obtida no curto prazo. O escalonador proposto baseia-se na declaração de um modelo empregando variáveis de intervalo. Um algoritmo e um modelo de Programação Linear Inteira abordando relaxações do problema são apresentados para que se obtenha um limitante superior ao valor de produção ótimo. Para o cenário real no qual a análise experimental foi feita, foram obtidas soluções a menos de 16% do ótimo após uma hora de execução; e os testes em instâncias de tamanhos variados evidenciaram a robustez do escalonador. Direções para trabalhos futuros são apresentadas ponderando os resultados obtidos.

Palavras-chave: Programação por Restrições, Escalonamento Baseado em Restrições, Desenvolvimento de Poços de Petróleo Marítimos.

Abstract

SERRA, T. **Constraint programming and constraint-based scheduling: A case study in the scheduling of resources for developing offshore oil wells.** 2012. 151 f. Dissertation (M.Sc.) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

The aim of this work is to present a problem of optimizing the use of critical resources to develop offshore oil wells and the technique used to approach the problem. The review of the Constraint Programming technique is made by analyzing relevant aspects of modeling, propagation, search and programming paradigms. The specialization of the technique to scheduling problems, known as Constraint-Based Scheduling, is described with emphasis on descriptive paradigms and constraint propagation mechanisms. In order to support the use of the technique to tackle other problems, the commercial modeling language OPL is presented in the appendix. The aim of the approach to the problem is to obtain a scheduler that maximizes the short-term production of oil. The scheduler presented relies on the description of a model using interval variables. An algorithm and an Integer Linear Programming model approaching relaxations of the problem are presented in order to obtain an upper bound for the optimal production value. For the real scenario upon which the experimental analysis was done, there were found solutions within 16% of the optimal after one hour of execution; and the tests on instances of varied sizes gave evidence of the robustness of the scheduler. Directions for future work are presented based on the results achieved.

Keywords: Constraint Programming, Constraint-Based Scheduling, Offshore Oil Well Developments.

Prefácio

Essa dissertação se divide entre a abordagem a um problema prático oriundo da Petrobras e uma introdução teórica à técnica empregada, sendo dado bastante destaque a essa última. E por que tanta ênfase à técnica? Comparativamente, a programação por restrições não goza de muito reconhecimento no Brasil. Sua importância se deve à facilidade com que possibilita abordar problemas combinatórios e de escalonamento tratados frequentemente no meio acadêmico e na indústria. Dessa forma, ainda que o problema estudado seja bastante interessante e pouco trivial, elaborar material atualizado e em português sobre a técnica pode ser muito mais importante do que simplesmente reproduzir o conteúdo dos artigos elaborados acerca do problema. Por tal razão, decidi dar igual importância à técnica e ao problema, além de repetir algumas definições em mais de uma parte do trabalho para possibilitar ao leitor a uma leitura sequencial.

Pelo hábito e confiança à técnica, sei que me arrisco a ser omissos ou mesmo tendenciosos. Dada a multidisciplinaridade do campo da pesquisa operacional, é muito comum que cada praticante se concentre em algumas técnicas e acabe por ver qualquer problema apenas sob a ótica das mesmas. No meu caso, a marcha foi inversa e acabei por me especializar no assunto em razão dos resultados que obtive no projeto aqui descrito. Entretanto, nem sempre fui bem-sucedido em passar essa mensagem adiante e utilizar os argumentos corretos para justificar seu emprego. O material que apresento na primeira parte dessa dissertação reflete o desenvolvimento do meu discurso a esse respeito, procurando encorajar ao máximo a aplicação da técnica em situações com maior potencial para sucesso. Além disso, a terminologia reconhecida em inglês para os principais conceitos é informada para que o leitor possa procurar por outras referências com maior facilidade. Como adendo a essa parte teórica, a linguagem de modelagem comercial OPL é introduzida no apêndice como forma de subsídio à utilização efetiva da técnica.

Não obstante o foco dado à programação por restrições, o problema abordado tem uma aplicação importante e é tratado em um grau de detalhe maior se comparado à literatura existente. A programação de recursos ditos críticos ao desenvolvimento de novos poços de petróleo é crucial devido à escassez e ao alto custo desses recursos, que são o gargalo do processo. Levando-se em conta o cenário recente de descobertas relativamente abundantes na camada do pré-sal e as limitações econômicas envolvidas, qualquer melhoria no desempenho operacional tem uma relação direta com a antecipação da produção nesses novos reservatórios e com a consequente redução dos investimentos necessários, ou com o aumento do escopo de tais investimentos. Além disso, o detalhamento dado ao problema serve tanto como registro documental sobre o processo sob o ponto de vista do problema de otimização decorrente como também constitui um estudo de caso interessante pela variedade de interdependências entre as atividades a serem programadas. A abordagem proposta atém-se ao paradigma declarativo, de forma que sua relevância se dá pela forma com que captura a estrutura do problema para que seja resolvido por ferramentas que se encontram no estado da

arte. Dessa forma, além de validar a técnica com uma aplicação significativa, o desenvolvimento do estudo de caso também figura como uma contribuição paralela àquela da apresentação teórica.

Minha participação nesse projeto da Petrobras concentrou-se no desenvolvimento de um novo módulo resolvidor para o sistema ORCA (Otimização dos Recursos Críticos nas Atividades), que consiste no estudo de caso apresentado. A existência do sistema na Petrobras precede em muito minha colaboração, sendo importante ressaltar o empenho pioneiro de Ricardo Accioly e Fernando Marcellino, de Romulo Albuquerque no desenvolvimento de um resolvidor para versões intermediárias, de Gilberto Nishioka na retomada do projeto e nos detalhamentos posteriores, além de muitos outros colaboradores da força de trabalho da Petrobras, e de alunos e professores da Unicamp que se envolveram de alguma forma ao longo desses anos. Espero manter esse elo pré-existente, fazendo com que a publicação dessa dissertação auxilie em desenvolvimentos posteriores ou em projetos similares, além de encorajar o emprego da técnica em outros nichos com potencial de sucesso.

Agradecimentos

Em primeiro lugar, à minha família pelo papel fundamental para que esse trabalho se concretizasse. À minha companheira Sabrina pela presença e carinho ao longo desses anos juntos, além do apoio quando precisei me dedicar aos estudos e do capricho dos modelos que fez para minhas apresentações. À minha mãe Ana Maria pelo sacrifício, cuidado e amor com que criou seus filhos e possibilitou que pudéssemos buscar nossos sonhos. À minha irmã Ana Caroline pela companhia nos bons e maus momentos desde muito cedo na vida. À minha sogra Maria Inês pelos conselhos da experiência e toda a co-orientação informal ajudando nos meus escritos e apresentações. Por fim, à memória de meu pai Joaquim Benedito e de meus quatro avós, que a seu tempo contribuíram direta e indiretamente para essa realização.

Não poderia também esquecer o apoio de vários professores e profissionais diversos no âmbito acadêmico. À minha orientadora Yoshiko Wakabayashi por ter aceitado o desafio de me orientar em um tema diferente de sua linha de pesquisa, bem como pelo nível de suas aulas e atenção ao desempenho dos alunos. Ao professor Arnaldo Vieira Moura, meu orientador de iniciação científica na Unicamp, por ter possibilitado meu primeiro contato com a Petrobras. Ao professor Carlos Eduardo Ferreira que, em conjunto com a Yoshiko e o Arnaldo, contribuiu com sugestões valiosas em minha banca de qualificação. Aos professores Arnaldo, Cid Carvalho de Souza, Cláudio Leonardo Lucchesi, Rodolfo Jardim Azevedo e Miguel Mira da Silva pelas cartas de recomendação para minha admissão no mestrado. Decerto que a lista de professores que fizeram diferença é muito maior que a apresentada e gostaria de representá-los simbolicamente por meio da professora Edileuza, da minha quarta série do primário, por remar contra a maré e se preocupar com seus alunos no contexto de um ensino público de níveis básico e médio bastante deficitário. Por fim, ao corpo de funcionários do IME e da USP por possibilitar a execução de meu trabalho.

Cabem também agradecimentos para que o projeto ocorresse com sucesso na Petrobras. Ao Fernando Marcellino por ter confiado a mim a responsabilidade de desenvolver um novo módulo resolvidor. A toda a equipe do ORCA e à estrutura subjacente, com especial atenção a Gilberto Nishioka, Antônio de Oliveira e à gerência da TIC em São Paulo. A meus colegas e ex-colegas de trabalho pelos eventuais apoios, revisões de rascunhos e sugestões, incluindo Edson Nishida, Tony Lopes, Daniel Ferber, Felipe Sodré, Henrique Goulart e Gustavo Pacianotto. Aos colegas e professores da Universidade Petrobras, que acompanharam o momento de escrita desse trabalho.

Por último, mas não menos importante, gostaria de lembrar algumas pessoas de contextos diversos. A meus colegas de Unicamp Alan Godoy, Tiago Tavares e André Ciré, que acompanharam bem de perto o desenrolar desse mestrado e colaboraram como puderam para seu sucesso por meio da revisão de rascunhos e trocas de idéias. A outros amigos de longa data como a Amanda, o Leandro, o Lucas e o Gustavo em Santo André; a Aline, o Pedro e o Hugo em Lisboa; e meus colegas de EC03, CACo e Maratona na Unicamp. À Ivonete pelo zelo com que cuidou de nossa casa

e possibilitou que eu e a Sabrina pudéssemos coordenar estudo e trabalho. E a todos os demais que, entendendo minha fraqueza de memória e o curto espaço disponível, saibam o quanto representam para essa realização por terem cruzado comigo pela vida.

“Sabemos bem que toda a obra tem que ser imperfeita, e que a menos segura das nossas contemplações estéticas será a de aquilo que escrevemos. Mas imperfeito é tudo, nem há poente tão belo que o não pudesse ser mais, ou brisa leve que nos dê sono que não pudesse dar-nos um sono mais calmo ainda. E assim, contempladores iguais das montanhas e das estátuas, gozando os dias como os livros, sonhando tudo, sobretudo, para o converter na nossa íntima substância, faremos também descrições e análises, que, uma vez feitas, passarão a ser coisas alheias, que podemos gozar como se viessem na tarde.”

Bernardo Soares (heterônimo de Fernando Pessoa), Livro do Desassossego

Sumário

Lista de Abreviaturas	xv
Lista de Figuras	xvii
Lista de Tabelas	xix
I INTRODUÇÃO	1
1 Introdução	3
1.1 Referências e Notas de Apoio à Leitura	3
1.2 A Indústria de Petróleo e a Programação de Recursos Críticos	5
1.3 Problemas de Escalonamento e a Programação por Restrições	8
1.4 Organização do Restante do Trabalho	10
II DESCRIÇÃO DA TÉCNICA	11
2 Programação por Restrições	13
2.1 Modelagem	14
2.1.1 Aplicabilidade em Função da Modelagem	15
2.1.2 Boas Práticas em Modelagem	18
2.2 Propagação de Restrições	19
2.2.1 Técnicas de Consistência Binária	20
2.2.2 Técnicas de Consistência Generalizada	27
2.3 Busca	29
2.3.1 Busca por Recursão e Retrocesso	30
2.3.2 Busca Local e Estocástica	36
2.4 Paradigmas	37
2.4.1 Ferramentas Comerciais	38
2.4.2 Modelagem Abstrata	39
3 Escalonamento Baseado em Restrições	41
3.1 Modelagem e Paradigmas Descritivos	42
3.1.1 Caracterização por Atividades e Recursos	42
3.1.2 Caracterização por Variáveis de Intervalo	46
3.2 Elementos Complementares de Resolução	49

3.2.1	Propagação	49
3.2.2	Busca	55
III	ESTUDO DE CASO	57
4	O Problema da Programação de Recursos Críticos e Sua Abordagem	59
4.1	Decisões e Entidades Relacionadas	60
4.2	CrITÉrios de Otimização	60
4.3	Restrições do Problema	61
4.3.1	Restrições Gerais dos Recursos	61
4.3.2	Restrições Gerais de Execução das Atividades	62
4.3.3	Restrições de Estoque dos Recursos	62
4.3.4	Restrições de Estoque das Atividades	62
4.4	Modelagem por Variáveis de Intervalo	63
4.4.1	Associação entre Atividades e Recursos	63
4.4.2	Associação entre Atividades e Localidades	67
4.4.3	Restrições de Estoque e Transporte de Linhas	68
4.4.4	Função Objetivo	70
4.5	Limitante Superior ao Ótimo	70
4.5.1	Algoritmo Guloso sobre Relaxação	71
4.5.2	Modelo de PLI sobre Relaxação	71
5	Análise Experimental do Escalonador	77
5.1	Dados Disponíveis e Ambiente de Testes	77
5.2	Experimentos Realizados e Resultados	78
5.3	Discussão	81
IV	CONCLUSÃO	83
6	Conclusão	85
V	APÊNDICE	87
A	Programação por Restrições em OPL	89
A.1	A Linguagem OPL e o Resolvedor CP	89
A.2	Representação do Modelo e da Entrada de Dados	90
A.3	Elementos de Uso Geral da Linguagem	91
A.3.1	Escolha do Resolvedor do Modelo	91
A.3.2	Comentários em Arquivos de Modelo e de Dados	92
A.3.3	Variáveis de Tipos Primitivos e Atribuições	92
A.3.4	Estruturas de Dados Básicas: Vetores e Domínios	93
A.3.5	Atribuições Básicas a Elementos de Vetores	94
A.3.6	Expressões Numéricas e Predicados	95

A.3.7	Operadores de Agregação e Funções Sobre Vetores	96
A.3.8	Estruturas de Dados Avançadas: Tuplas e Conjuntos	98
A.4	Modelagem de Problemas Combinatórios	99
A.4.1	Variáveis de Decisão de Tipos Primitivos	100
A.4.2	Expressões de Decisão	100
A.4.3	Sobre a Impossibilidade de Usar Variáveis <i>float</i> em CP	102
A.4.4	Função Objetivo	102
A.4.5	Restrições Básicas Sobre Tipos Primitivos	103
A.4.6	Restrições Especializadas Sobre Tipos Primitivos	104
A.4.7	Execução do Resolvedor e Configurações Básicas do CP	105
A.4.8	Manipulação da Solução e Impressão dos Resultados no Pós-Processamento	106
A.4.9	Um Exemplo de Problema Combinatório Modelado em OPL	106
A.5	Modelagem de Problemas de Escalonamento	108
A.5.1	Problemas de Escalonamento e a Motivação ao Emprego de CP	108
A.5.2	Variáveis de Intervalo e Suas Propriedades	108
A.5.3	Restrições de Precedência	110
A.5.4	Predicados Sobre Conjuntos de Intervalos	111
A.5.5	Sequências e Transições	112
A.5.6	Modelagem de Recursos Limitados	113
A.5.7	Funções Cumulativas	114
A.5.8	Funções de Estado	117
A.5.9	Um Exemplo de Problema de Escalonamento Modelado em OPL	118
A.6	Considerações Finais e Agradecimentos	120
Referências Bibliográficas		121
Índice Remissivo		127

Lista de Abreviaturas

CBS	Escalonamento Baseado em Restrições (de <i>Constraint-Based Scheduling</i>)
CLP	Programação Lógica por Restrições (de <i>Constraint Logic Programming</i>)
CP	Programação por Restrições (de <i>Constraint Programming</i>)
CSOP	Problema de Otimização da Satisfação de Restrições (de <i>Constraint Satisfaction Optimization Problem</i>)
CSP	Problema de Satisfação de Restrições (de <i>Constraint Satisfaction Problem</i>)
EBR	Escalonamento Baseado em Restrições
GRASP	Procedimentos de Busca Gulosa, Aleatória e Adaptativa (de <i>Greedy Randomized Adaptive Search Procedures</i>)
JSP	Problema do Escalonamento de Chão-de-Fábrica (de <i>Job-Shop Scheduling Problem</i>)
MA	Meta-Aprendizagem
ML	Meta-Aprendizagem (de <i>Meta-Learning</i>)
OPL	Linguagem de Programação de Otimização (de <i>Optimization Programming Language</i>)
ORCA	Otimização de Recursos Críticos nas Atividades (sistema desenvolvido na Petrobras)
PEC	Problema do Escalonamento de Chão-de-Fábrica
PI	Programação Inteira
PLI	Programação Linear Inteira
PM	Programação Matemática
PLR	Programação Lógica por Restrições
PR	Programação por Restrições
POSR	Problema de Otimização da Satisfação de Restrições
PSR	Problema de Satisfação de Restrições
PTS	Problema Temporal Simples
SAT	Problema de Satisfabilidade
STP	Problema Temporal Simples (de <i>Simple Temporal Problem</i>)

Lista de Figuras

1.1	Sonda marítima Transocean Legend.	6
1.2	Barco lançador de linhas Polar Queen.	6
4.1	Relação elementar entre intervalos associados a atividades e recursos.	65
4.2	Restrição de domínios temporais e de alocação no desenvolvimento de poços.	66
4.3	Relação entre intervalos associados a atividades e suas respectivas localidades.	68
4.4	Representação das restrições de estoque do recurso por porto e na totalidade.	69
5.1	Melhoria da solução obtida em cada rodada do escalonador para a instância $Q1$	79
5.2	Melhoria da solução obtida em cada rodada do escalonador para a instância $Q2$	79
5.3	Melhoria da solução obtida em cada rodada do escalonador para a instância $Q3$	79
5.4	Melhoria da solução obtida em cada rodada do escalonador para a instância $Q4$	80
5.5	Melhoria da solução obtida em cada rodada do escalonador para a instância $H1$	80
5.6	Melhoria da solução obtida em cada rodada do escalonador para a instância $H2$	80
5.7	Melhoria da solução obtida em cada rodada do escalonador para a instância O	81

Lista de Tabelas

4.1	Variáveis de decisão do modelo PLI do problema relaxado.	72
5.1	Principais características das instâncias testadas.	77
5.2	Principais resultados dos experimentos do escalonador.	78

Parte I

INTRODUÇÃO

Capítulo 1

Introdução

Problemas combinatórios ocorrem em diversas indústrias, mas nem sempre são percebidos como oportunidades para automação, otimização e consequente redução de custos. Um tratamento formal e analítico desses problemas tem como benefício um melhor aproveitamento de mão-de-obra especializada e dos bens de capital envolvidos. Tal aproveitamento pode ser observado em termos quantitativos de produção, nas métricas de qualidade de serviço, na observância a normas de segurança, etc. Entretanto, é importante observar que o sucesso de uma abordagem prática depende da habilidade com que o problema é analisado e do ferramental à disposição dos profissionais envolvidos. Dessa forma, é possível entender que a contribuição de um trabalho acadêmico com foco na aplicação se dê pelo legado para que projetos posteriores tenham maior chance de sucesso. Tendo essa observação em mente, o caminho escolhido nessa dissertação foi o de integrar a abordagem de um problema real à descrição da técnica empregada.

O problema estudado nessa dissertação relaciona-se ao desenvolvimento de novos poços marítimos de produção de petróleo e à manutenção de outros pré-existentes, sendo abordado pela técnica de programação por restrições. Cabe ressaltar que a técnica adotada para a abordagem não representa a única opção possível, mas é uma modalidade que goza de pouco reconhecimento a despeito da relativa facilidade com que poderia ser aplicada a muitos problemas. De forma a tornar o assunto abordado mais acessível, alguns conceitos fundamentais de áreas correlatas e referências que entrem em maiores detalhes são apresentados na seção 1.1. Na sequência, as seções 1.2 e 1.3 contextualizam o problema em foco na indústria em que se insere e em função do ferramental empregado, respectivamente. O propósito do restante da dissertação é apresentar a técnica adotada, introduzir uma ferramenta que viabilize seu uso e tratar o problema mencionado em maiores detalhes. A organização adotada para os demais capítulos é apresentada na seção 1.4.

1.1 Referências e Notas de Apoio à Leitura

A apresentação da indústria petrolífera descrita neste capítulo reflete o conhecimento adquirido pelo autor com sua experiência na Petrobras, tanto por meio de treinamentos quanto pelo levantamentos de requisitos. Como sugestão de referência para aprofundamento, veja [Thomas \(2004\)](#).

Como parte da discussão sobre abordagens para a resolução de problemas de otimização, são mencionados ou presumidos alguns conhecimentos de otimização numérica, programação linear e de programação linear inteira. Para uma introdução ao primeiro tópico e aos conceitos de convexidade de funções que possibilitam o uso de algoritmos de gradiente para obter um ótimo global,

veja [Nocedal e Wright \(2000\)](#). Para uma introdução ao segundo, veja [Bazaraa *et al.* \(1990\)](#). Para uma introdução ao último e aos tipos de problemas que permitem uma resolução eficiente com seu uso, veja [Wolsey \(1998\)](#).

Por fim, diversos conceitos de teoria da computação são mencionados no desenvolvimento teórico acerca da técnica. Para maiores detalhes sobre linguagens formais, veja [Sipser \(2007\)](#). Para uma introdução às classes de complexidade de algoritmos, veja [Cormen *et al.* \(2002\)](#). Para maiores informações sobre teoria dos grafos, veja [Bondy e Murty \(2008\)](#). De forma a facilitar a consulta a conceitos de maior relevância, um pequeno resumo de alguns deles é apresentado a seguir.

Classes de problemas.

- **Problemas NP.** A classe dos problemas NP contempla problemas decisórios cujas soluções podem ser verificadas em tempo polinomial com relação ao tamanho da entrada. (Mais formalmente, diz-se que tais soluções podem ser verificadas em tempo polinomial por uma máquina de Turing não-determinística.)
- **Problemas NP-Difíceis.** Dizemos que um problema Q pertence à classe dos problemas NP-Difíceis se todo problema em NP pode ser reduzido a Q em tempo polinomial.
- **Problemas NP-Completo.** Dizemos que um problema Q pertence à classe dos problemas NP-Completo se Q pertence à classe NP, e além disso é NP-Difícil. Assim, se tivéssemos um algoritmo polinomial para um problema NP-Completo, então poderíamos resolver em tempo polinomial todos os problemas em NP.

Grafos. Um grafo G consiste em um par (V, E) , onde V é um conjunto de elementos chamados vértices e E é um conjunto elementos chamados arestas, sendo que cada aresta corresponde a um par de vértices de V . Uma aresta correspondente a um par de vértices $u, v \in V$ é denotada por uv , e é dita incidente ao tal par. Quando uma aresta é duplamente incidente a um mesmo vértice, ela é denominada laço; e quando duas arestas distintas correspondem a um mesmo par de vértices elas são chamadas paralelas. Um grafo sem laços e sem arestas múltiplas é chamado simples.

- **Orientação.** Uma orientação de um grafo $G = (V, E)$ é um grafo direcionado – ou digrafo – $D = (V, E')$ tendo o mesmo conjunto de vértices mas arestas dirigidas, de forma que a cada aresta $uv \in E$ corresponde uma aresta $\vec{uv} \in E'$ ou $\vec{vu} \in E'$, mas não ambas.
- **Ordenação topológica.** Uma ordenação topológica de um digrafo $D = (V, E')$ consiste em uma ordenação linear de seus vértices tal que para cada aresta $\vec{uv} \in E'$, o vértice u precede v nesta ordem (linear).
- **Árvore.** Uma árvore é um grafo conexo e sem circuitos (ou equivalentemente, um grafo onde há um único caminho entre cada par de vértices).
- **Subgrafo.** Diz-se que um grafo $H = (V', E')$ é subgrafo de $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$.
- **Subgrafo induzido.** Dizemos que $H = (V', E')$ é um subgrafo induzido de G se E' corresponde ao conjunto de todas as arestas de E incidentes exclusivamente a vértices de V' . Neste caso, H é induzido por um conjunto de vértices. (O conceito de subgrafos induzidos por um conjunto de arestas não será usado aqui.)

- **Isomorfismo.** Dois grafos $G = (V, E)$ e $H = (V', E')$ são isomorfos se existe uma bijeção $f : V \rightarrow V'$ tal que $uv \in E \leftrightarrow f(u)f(v) \in E'$.
- **Grafo bipartido.** Um grafo $G = (V, E)$ é bipartido se existem conjuntos disjuntos V_1 e V_2 tais que $V = V_1 \cup V_2$, e toda aresta em E é incidente a um vértice de V_1 e a um vértice de V_2 .
- **Emparelhamento máximo de um grafo bipartido.** Um emparelhamento consiste em um conjunto $M \subseteq E$ tal que nenhum par de arestas de M incide sobre um mesmo vértice. Um emparelhamento M de G é máximo se não existe outro emparelhamento M' de G tal que $|M'| > |M|$.
- **Corte.** Um corte de um grafo $G = (V, E)$ é o conjunto de todas as arestas incidentes a um algum conjunto não-vazio $W \subset V$ e a seu complemento $V \setminus W$.

1.2 A Indústria de Petróleo e a Programação de Recursos Críticos

A indústria petrolífera representa um cenário interessante a abordagens quantitativas para a tomada de decisão devido aos altos custos envolvidos em várias etapas de sua cadeia produtiva. Em primeira instância, uma parcela considerável dos processos desenvolvidos nessa indústria podem ser divididos entre o que se convencionou chamar de *upstream* e *downstream*, respectivamente responsáveis pela obtenção de óleo cru – composto por gás natural, óleo, água e sedimentos, sendo separados fisicamente após a obtenção – e pelo seu processamento para a obtenção de derivados – produtos resultantes de processos físicos e químicos realizados em uma refinaria. No caso específico do *upstream*, os processos podem ser subdivididos entre exploração¹ e produção², o primeiro relacionado à prospecção de potenciais reservatórios e o segundo ao desenvolvimento e consequente extração de matéria-prima dos reservatórios descobertos. Para a produção, destacam-se a etapa de desenvolvimento dos poços de um reservatório, a produção de óleo e gás propriamente dita, intervenções periódicas de manutenção nos poços e o fechamento dos poços quando a produção cessa. O foco desse trabalho se dará nas etapas de desenvolvimento e manutenção dos poços para produção, com atenção às peculiaridades da extração marítima em águas profundas. Por águas profundas, consideramos profundidades a partir de 300 metros por representarem o limite para a intervenção humana direta e para a utilização de armaduras nos poços, o que possibilitaria aplicar técnicas similares àquelas da extração terrestre.

A execução das atividades de desenvolvimento e manutenção da infraestrutura de produção depende da alocação de recursos adequados às restrições operacionais. No caso de águas profundas, os recursos de maior criticidade são sondas marítimas e barcos lançadores de linhas, exemplificados pela figura 1.1 de Yue (2008) e pela figura 1.2 de Acergy-Group (2010). Esses recursos têm um custo significativamente maior do que aquele verificado para equipamentos empregados em outras modalidades de produção comercialmente viáveis atualmente. Apesar dos custos envolvidos em desenvolvimentos marítimos serem muito maiores do que aqueles observados nas terrestres, eles ainda são viáveis em razão do preço pago pelo barril atualmente. No entanto, existe uma exigência mais elevada de produção para que exista uma compensação financeira ao desenvolvimento de poços nessas condições. Os recursos necessários a essas empreitas são tidos como críticos por serem pouco

¹Denominada *exploration* em inglês.

²Denominada *exploitation* em inglês.

numerosos e certas vezes alugados por períodos limitados de tempo, sendo necessário que transitem entre diversas localidades de uma região de grandes dimensões. Além disso, devem obedecer a rotinas constantes de manutenção que interrompem a capacidade de execução de alguns tipos de atividade ou mesmo de qualquer atividade. Dessa forma, o uso adequado desses recursos pode representar um ganho operacional significativo.

Figura 1.1: *Sonda marítima Transocean Legend.*

Figura 1.2: *Barco lançador de linhas Polar Queen.*

A programação das atividades nos recursos contempla tanto a alocação a um recurso, i.e., *como*

realizar cada atividade, quanto o escalonamento do recurso, i.e., *quando* realizá-las. Cada poço começa a produzir ou tem uma retomada de sua produção tão logo suas atividades previstas sejam concluídas. Por conseguinte, a programação deve ser feita de forma a antecipar o ganho resultante no curto prazo para compensar os custos associados. Para o número de atividades e de restrições de um cenário real desse problema, a programação manual das atividades passou a ser inviável por tomar vários dias de trabalho de mão-de-obra especializada. Nessa situação, torna-se impraticável reprogramar, em curto prazo, a escala completa de atividades devido à falta de um recurso ou ao atraso na data de conclusão prevista para uma atividade. Por consequência, a otimização da escala é muitas vezes desconsiderada numa abordagem manual. Portanto, a formalização e o desenvolvimento de ferramentas capazes de resolver o problema de alocação e escalonamento de recursos é de grande importância para um melhor desempenho da organização.

A primeira abordagem a este problema foi feita por Hasle *et al.* (1996) e motivou desenvolvimentos posteriores para escopos e realidades distintas, como é o caso da brasileira. A abordagem de Hasle *et al.* (1996) consistia em um protótipo para aplicação no desenvolvimento de poços no Mar do Norte. Um sistema similar denominado ORCA foi desenvolvido pela Petrobras e descrito em Accioly *et al.* (2002). A exemplo do precursor, esse sistema empregava a técnica de programação por restrições. Apesar de contemplar requisitos necessários ao desenvolvimento de poços, o escopo desse sistema restringiu-se às atividades de manutenção. A demanda por esse escopo restrito surgiu devido à menor oferta relativa de recursos disponíveis para tal fim, o que tornava imperativo o melhor uso possível desses recursos. Uma abordagem alternativa empregando programação matemática e busca *tabu*³ sobre uma simplificação do problema tratado pelo sistema da Petrobras foi desenvolvida por do Nascimento (2002) a partir de um contato inicial entre a empresa e a Unicamp. Essa autora provou que mesmo tal versão simplificada é NP-Difícil, sendo tão difícil quanto o Problema do Escalonamento de Chão-de-Fábrica (PEC)⁴, o que significa que é pouco provável que existam algoritmos eficientes para o caso geral do problema tratado. Um detalhamento progressivo dessa especificação original foi apresentado em trabalhos posteriores, cuja abordagem foi incorporada ao sistema original da empresa. Pereira *et al.* (2005) abordou a versão completa do problema utilizando a meta-heurística GRASP⁵, tendo um resultado superior ao original para determinados cenários. Um refinamento dessa abordagem é apresentada por Moura *et al.* (2008b), contemplando a incorporação de restrições para o deslocamento de recursos entre localidades. O trabalho no sistema ORCA foi retomado recentemente e teve seu foco principal alterado para o desenvolvimento de poços. A abordagem descrita nessa dissertação contempla o que foi publicado recentemente em Serra *et al.* (2010), Serra *et al.* (2011) e Serra *et al.* (2012), além de refinamentos adicionais que constarão em publicação posterior sobre o trabalho desenvolvido. A retomada frequente no desenvolvimento do ORCA se deve à constante identificação de novos gargalos que impossibilitem a aplicação dos resultados sugeridos pelas soluções obtidas, o que é natural em situações desse tipo.

Convém observar que existem abordagens sobre problemas relacionados ao ciclo de vida dos

³ Meta-heurística denominada *tabu search* em inglês, proposta por Glover (1989), na qual as operações empregadas mais recentemente para obter uma solução a partir de outra pré-existente são memorizadas para não serem repetidas por um determinado número de passos ao longo da busca por uma solução de melhor qualidade.

⁴ Tradução livre adotada para o *Job-Shop Scheduling Problem* (JSP), que consiste no problema de minimizar o tempo de conclusão do escalonamento de um conjunto de seqüências de tarefas, cada tarefa compatível com um determinado recurso.

⁵ Acrônimo em inglês para *Greedy Randomized Adaptive Search Procedures*, ou Procedimentos de Busca Gulosa, Aleatória e Adaptativa, proposta por Feo e Resende (1989), na qual soluções são geradas por meio de uma construção semi-gulosa que dá margem à aleatoriedade e de um aperfeiçoamento simples ao final da construção

reservatórios de petróleo, com relação às quais esse trabalho pode ser contextualizado. A programação de atividades de manutenção dos poços em produção representa um caso especial do problema em estudo, abordada por [Paiva \(1997\)](#) e [Aloise *et al.* \(2006\)](#) no caso de poços terrestres. O planejamento das atividades exploratórias é abordado por [Glinz e Berumen \(2009\)](#), sendo caracterizado por um nível menor de detalhamento por representar um estágio preliminar com menor grau de informação e maior incerteza acerca das características e do potencial dos poços a serem perfurados. As decisões tomadas no planejamento da exploração possuem um caráter mais estratégico do que operacional, estando associadas à aquisição de recursos para aumentar o retorno do investimento. Por outro lado, as entidades na programação do desenvolvimento de poços possuem restrições relativas às características de cada poço, recurso ou atividade. Não obstante tal nível de detalhamento, a programação do desenvolvimento aqui abordada tem ainda um caráter tático por possibilitar a avaliação do ganho associado à aquisição de um novo recurso pela comparação das soluções com e sem o seu emprego. Dessa forma, o problema em estudo tem um papel mais crítico devido à sua proximidade ao nível operacional em um dos processos mais custosos na cadeia de valor da companhia.

1.3 Problemas de Escalonamento e a Programação por Restrições

Problemas de escalonamento tratam da alocação de atividades a recursos e de sua programação no tempo, sendo sujeitos a restrições técnicas que devem ser satisfeitas para que essas decisões sejam válidas em cada situação possível. Conforme discutido anteriormente, é comum que esse tipo de problema envolva um detalhamento maior do que aquele encontrado em problemas de planejamento. Em muitos casos, isso acaba levando a modelos matemáticos de representação que descrevem problemas computacionalmente difíceis. Por esse motivo, problemas de escalonamento nem sempre são abordados com a ambição de que uma solução ótima seja encontrada. A obtenção de uma solução factível já representa um grande avanço de acordo com o caso, como no problema de programação de transportes por dutos tratado por [Moura *et al.* \(2008a\)](#). Em outras situações, é importante ter uma sistemática de busca que facilite a obtenção de soluções cada vez melhores a um problema cuja otimização seja a parte difícil, com a finalidade de tornar os resultados mais confiáveis. Representando um pouco de cada situação, o problema tratado nesse trabalho encerra uma oportunidade interessante para a análise do ferramental disponível para a resolução de problemas de otimização de ordem prática.

Na ausência de algoritmos específicos que abordem um problema de modo eficiente, técnicas generalistas como a Programação Matemática (PM) são utilizadas para uma abordagem sucinta. De maneira bem simplificada, em PM um problema é descrito com a minimização ou maximização de uma função sujeita a restrições impostas por outras funções sobre as mesmas variáveis de decisão. Em alguns casos, entretanto, problemas de escalonamento representam um desafio à aplicação de PM devido a limitações de tempo e memória dos recursos computacionais. Tais empecilhos decorrem das limitações sobre o tipo de variáveis e o tipo de restrições permitidos na formulação para o emprego de certos algoritmos. O emprego de formulações com restrição de linearidade acaba exigindo a integralidade das variáveis de decisão em certos casos e faz com que seu tamanho também cresça consideravelmente com as dimensões do problema. Por tal razão, o tempo necessário para resolver um problema numérico que seja tão somente convexo acaba por crescer muito rapidamente com

tais dimensões. Disso resultam formulações maiores e, em certos casos, grandes demais para serem resolvidas com os recursos computacionais disponíveis. Dessa forma, a despeito do desenvolvimento observado e do consenso sobre a eficácia da PM, nem todos os problemas podem ser abordados de forma eficiente com a mesma⁶.

A Programação por Restrições (PR)⁷ representa uma alternativa à abordagem de problemas combinatórios, inserindo-se no nicho não atendido completamente pela Programação Linear Inteira (PLI) devido ao tamanho das formulações resultantes de tais abordagens. Em PR, a modelagem é baseada em variáveis sobre domínios discretos e finitos com restrições arbitrárias, porém especializadas. Dessa forma, a modelagem não está restrita a funções lineares ou mesmo convexas como no caso das técnicas de PM; o que leva a algoritmos de busca diferentes daqueles de PM, tanto mais generalistas quanto mais focados na factibilidade do que na otimalidade. Apesar disso, a modelagem possui uma ligação mais estreita com as técnicas de resolução devido à existência de restrições dedicadas à descrição de certas estruturas de problemas combinatórios. Isso permite que o algoritmo de resolução possa trabalhar com maior facilidade no domínio do problema e inferir reduções do espaço de busca mais eficazes. No caso de problemas de escalonamento, não apenas as restrições como as próprias variáveis de decisão empregadas são específicas ao domínio de aplicação, constituindo o que se conhece por Escalonamento Baseado em Restrições (EBR)⁸. Em razão dessas peculiaridades de modelagem e da forma com que tais elementos são utilizados no processo de resolução dos problemas, atribui-se à técnica uma natureza declarativa.

Convém notar que existem técnicas alternativas com um enfoque no paradigma imperativo de programação, sendo tanto concorrentes quanto complementares à PR, dado que podem ser empregadas como mecanismo de busca subjacente no segundo caso. Uma opção consiste no emprego de algoritmos de aproximação, que geram soluções de modo eficiente e com uma garantia teórica de proximidade ao ótimo (Vazirani (2001)). No entanto, muitos desses algoritmos são propostos para interesses mais teóricos do que práticos, o que faz com que nem sempre sejam considerados em aplicações. Ainda pesa na opção por uma abordagem enumerativa como a PR em contraposição ao emprego de um algoritmo determinístico a possibilidade de geração de múltiplas soluções. Outra opção é a do uso de heurísticas que se valem da experiência prática de tomadores de decisão no domínio de aplicação para que soluções com certa confiabilidade sejam obtidas de maneira eficiente. Entretanto, essa confiabilidade não é traduzida em uma medida de proximidade às soluções ótimas e sim na evidência de que cada heurística se baseia em uma sucessão de decisões aparentemente razoáveis. Uma coletânea de resultados satisfatórios em casos diversos geralmente leva à generalização desses procedimentos sob a forma de meta-heurísticas aplicáveis a uma gama de problemas combinatórios. Tais meta-heurísticas podem servir tanto à geração de soluções quanto à obtenção de novas soluções a partir de outras pré-existentes, sendo respectivamente denominadas construtivas e de busca local⁹. A argumentação em favor de alguns desses procedimentos generalistas remete, em muitos casos, a paralelos nos campos da física e da biologia, como pode ser observado

⁶ Um leitor mais atento e experiente poderia objetar devido à existência de técnicas para a abordagem de problemas de formulação explosiva, como a geração de colunas ou de planos de corte em programação linear inteira. Entretanto, convém lembrar que o enfoque dado no presente trabalho é pela busca de uma abordagem mais acessível e que exija menos experiência para ser replicada em outros problemas.

⁷ Denominada *Constraint Programming* (CP) em inglês.

⁸ Denominado *Constraint-Based Scheduling* (CBS) em inglês.

⁹ Conforme pode ser constatado pela descrição em notas anteriores, a GRASP é construtiva e a Busca Tabu é de busca local.

em Aarts e Lenstra (1997). Battiti *et al.* (2008) faz uma ressalva ao foco excessivo dado a esse tipo de argumento em trabalhos recentes em detrimento à compreensão do motivo pelo qual resultados satisfatórios são obtidos e, por consequência, à adoção do mais adequado em cada caso. Por fim, é importante observar a compilação de estudos recentes por Hooker (2006) sobre a integração de técnicas de forma a obter o melhor de cada uma delas, partindo de quais tipos de formulações são resolvidos com maior facilidade por quais técnicas de otimização. Ainda que boa parte do que foi mencionado neste parágrafo não seja retomada na dissertação, é importante que tais considerações constem como referência aos leitores interessados em abordagens alternativas.

1.4 Organização do Restante do Trabalho

Além da introdução, esse trabalho possui outras três partes e um apêndice. Na segunda parte, a técnica de programação por restrições é descrita no Capítulo 2 e sua extensão à abordagem de problemas de escalonamento é apresentada no Capítulo 3. Na terceira parte, o estudo de caso é dividido em duas etapas: a descrição do problema e a abordagem no Capítulo 4, e a análise dos resultados obtidos no Capítulo 5. A quarta parte contempla as considerações finais sobre o trabalho desenvolvido. No Apêndice A, uma possibilidade de ligação entre a técnica descrita na segunda parte e a abordagem prática tratada na terceira parte é dada pela descrição de uma linguagem algébrica comercial para o uso de PR.

Apesar da limitação imposta pela apresentação de uma única ferramenta para o uso prático da técnica, é importante frisar a importância pedagógica e introdutória; à semelhança do que ocorre em disciplinas de introdução à programação, em que apenas os elementos básicos de uma linguagem são apresentados. Espera-se que a bagagem teórica e a noção de como é possível colocá-la em prática dêem subsídios suficientes para que leitores iniciantes no assunto possam tornar-se praticantes independentes se assim o desejarem.

Parte II

DESCRIÇÃO DA TÉCNICA

Capítulo 2

Programação por Restrições

A Programação por Restrições (PR)¹ é uma técnica declarativa para formular e resolver problemas combinatórios por meio da identificação de estruturas que permitam uma representação sucinta e facilmente explorável de cada problema. Devido à essa generalidade de propósito e à flexibilidade com que os problemas podem ser modelados, os mecanismos de resolução geralmente se baseiam em técnicas do tipo “tentativa-e-erro”. Não obstante, seu diferencial – e foco – se deve à exploração das representações adotadas para os problemas, de forma a reduzir o esforço necessário para a resolução dos mesmos. Dessa forma, a teoria subjacente à essa técnica possui tanto um foco especialista para compreender como ela pode ser adequada a contextos de aplicação específicos com maior eficácia quanto outro generalista para inferir as limitações ao estabelecimento de uma abordagem eficiente de amplo espectro.

O propósito deste capítulo é analisar os aspectos teóricos gerais da PR para elucidar como ela funciona e como tirar proveito dela, sendo foco do próximo capítulo a especialização desses conceitos para o caso de problemas de escalonamento. Para tanto, convém compreender de antemão o significado do nome da técnica e os elementos fundamentais de sua constituição.

A PR é definida por Lustig e Puget (2001) como uma técnica de programação de computadores dedicada à resolução de problemas combinatórios. Por meio dela, tais problemas são modelados com restrições projetadas para captar a estrutura do problema mais facilmente sobre as variáveis de decisão, sendo esta a razão para o nome e a vantagem subjacente à técnica. Dessa forma, o termo “programação” não assume o significado de um problema matemático², mas sim de uma técnica de codificação para gerar sistemas computacionais especialistas na resolução de um determinado tipo de problema combinatório. Lustig e Puget (2001) enfatizam a existência de uma camada de modelagem e outra de definição da estratégia para a realização da busca, o que é sintetizado simbolicamente por Hentenryck (2010) como “ $PR = Modelo + Busca$ ”³. Tal decomposição caracteriza a natureza declarativa da técnica, que é devida à independência do modelo quanto à forma de busca. Devido às sucessivas transformações do paradigma à aplicação da PR por usuários finais, Barták (1999) observa uma ênfase crescente à modelagem em detrimento à busca, culminando no paralelo atribuído

¹ Denominada *Constraint Programming* (CP) em inglês.

² Como exemplo oposto, é destacada por Lustig e Puget (2001) a semântica desse termo para o caso da Programação Linear (PL): a alocação de recursos limitados visando otimizar um objetivo, ou seja, estabelecer a programação mais adequada desses recursos. Para maiores detalhes, as preleções do Prêmio Nobel por Kantorovich (1975) e Koopmans (1975) apresentam a interpretação econômica dos modelos resultantes, que conferem a semântica diferenciada ao uso do termo “programação”.

³No original em inglês, “ $CP = Model + Search$ ”.

a Eugene Freuder pela máxima “*A Programação por Restrições representa uma das abordagens mais próximas que a ciência da computação tem feito por ora ao Santo Graal da programação: o usuário define o problema, o computador resolve-o.*”⁴. Entretanto, o binômio modelo-busca sobre sucessivos paradigmas não descreve com precisão o funcionamento da PR por marginalizar a propagação de restrições⁵, tida por referências como Barták (2001) como peça fundamental à viabilidade do emprego dessa técnica. Como elemento do tripé, a propagação opera na redução ativa dos domínios das variáveis de decisão por meio de inferências especializadas sobre as restrições em que tais variáveis estão presentes. Em outras palavras: define-se um nível desejado de consistência entre as variáveis do modelo para que atribuições de valores não conflitem, sendo a manutenção de tal nível garantida por algoritmos de filtragem. Conforme observado por Bockmayr e Kasper (1998), tal conceito guarda bastante similaridade com a geração de planos de corte para a resolução de problemas de Programação Linear Inteira (PLI), uma vez que ambos os casos representam o emprego de inferência para a geração de novas restrições sobre uma formulação. Em suma, a apresentação da teoria geral sobre PR, que se segue, analisa separadamente modelagem, propagação, busca e paradigmas de programação nas seções 2.1 a 2.4.

2.1 Modelagem

Formalmente, o objeto da Programação por Restrições é o Problema de Satisfação de Restrições (PSR)⁶. O PSR consiste no problema de encontrar atribuições para um conjunto de n variáveis de decisão levando em conta m restrições de relacionamento entre tais variáveis. Ele é caracterizado na literatura pela tripla (X, D, R) , sendo X o conjunto de variáveis de decisão x_i para $i = 1 \dots n$, D o conjunto de domínios D_i para $i = 1 \dots n$, e R o conjunto de restrições r_j para $j = 1 \dots m$. Cada variável possui seu respectivo domínio discreto e finito de valores para atribuição, de forma que $(x_i \leftarrow \bar{x}_i) \leftrightarrow (\bar{x}_i \in D_i)$ para $i = 1 \dots n$. Cada restrição $r_j(x_1, x_2, \dots, x_n)$ é uma relação matemática representando um subconjunto S_j do conjunto $D_1 \times D_2 \times \dots \times D_n$. Tais restrições podem ser representadas tanto de modo enumerativo por meio da listagem de todas as tuplas que as satisfazem quanto de modo declarativo por meio de uma expressão que descreva a linguagem formada pelas tuplas que satisfazem a restrição⁷. Uma restrição r_j é dita satisfeita por $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ se, e somente se, $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n) \in S_j$. A solução de um PSR consiste em uma atribuição a suas variáveis, digamos $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$, tal que $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n) \in S_j$, $j = 1 \dots m$, i.e., $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ satisfaz todo o conjunto de restrições R . No caso de um problema de otimização, a definição do PSR é estendida por Lustig e Puget (2001) e alguns outros autores como o Problema de Otimização da Satisfação de Restrições (POSR)⁸ para contemplar a existência de uma função objetivo $f : D_1 \times D_2 \times \dots \times D_n \rightarrow \Re$ que mapeie cada solução possível ou tentativa do PSR a um valor real, a ser maximizado ou minimizado. Apesar do caráter meramente formal das definições apresentadas, elas acabam por corresponder à estrutura com que os problemas são modelados em PR em razão

⁴ No original em inglês, citado por Barták (1999), “*Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.*”.

⁵ Denominada *constraint propagation* em inglês.

⁶ Denominado *Constraint Satisfaction Problem* (CSP) em inglês.

⁷ Na literatura, restrições declarativas são descritas como expressões numéricas. Entretanto, uma expressão regular também é conveniente – e talvez mais apropriada em alguns casos –, visto que representa qualquer combinação de tuplas de um alfabeto. No caso da PR, o alfabeto seriam os domínios das variáveis de decisão.

⁸ Denominado *Constraint Satisfaction Optimization Problem* (CSOP) em inglês.

da generalidade com relação às restrições possíveis.

No entanto, convém ressaltar que a forma com que a otimização é considerada na PR não é de todo consensual. Notadamente, o PSOR corresponde a uma concepção de otimização muito mais arraigada à pesquisa operacional do que à inteligência artificial, campos esses com visões bastante distintas sobre a técnica. Brailsford *et al.* (1999) compreende a otimização na PR como a forma pela qual restrições fracas⁹, cuja satisfação é desejável mas não necessária à viabilidade das soluções do problema, procuram ser satisfeitas. Nessa linha, a otimização é interpretada como o processo de busca de soluções que reduzam a violação das restrições fracas. Essa visão é partilhada por Dechter (2003), sendo esse o ponto de maior crítica à sua obra na resenha feita por Barták (2008). Sem embargo, dado o enfoque à abordagem de um problema de pesquisa operacional na próxima parte da dissertação, optamos por enfatizar o PSOR como a forma com que consideramos a otimização na PR.

2.1.1 Aplicabilidade em Função da Modelagem

A utilidade da técnica de PR decorre da abrangência de problemas que sua modelagem pode contemplar e que, por conseguinte, sua busca pode – em muitos casos – solucionar. Uma vez que é possível representar problemas pertinentes à classe NP-Completo por meio de PSRs, Dechter (2003) observa que PSRs também são problemas NP-Completo. Por motivo similar, o PSOR pertence à classe dos problemas NP-Difíceis, de forma que a expectativa sobre a PR nesse caso não é de que soluções ótimas possam ser sempre encontradas em tempo polinomial para quaisquer instâncias. Dessa feita, o esforço empreendido pela PR objetiva facilitar a resolução de nichos específicos empregando o ferramental à sua disposição, i.e., modelos que facilitem a propagação e a busca. Ainda que não seja uma percepção comum na literatura sobre PR, a prática da área corresponde à pedra angular da Meta-Aprendizagem (MA)¹⁰ para o problema da seleção de algoritmos. Smith-Miles (2008) apresenta o objeto de estudo da MA para a seleção de algoritmos como sendo a identificação de parâmetros de problemas que justifiquem a diferença de desempenho de diversos algoritmos, de forma a escolher o algoritmo mais apropriado a cada problema. Observações preliminares à elaboração dessa dissertação foram tecidas a esse respeito em Serra e Wakabayashi (2010), ressaltando que não é possível obter o melhor benefício possível da PR sem a compreensão do tipo de problemas para os quais componentes específicos foram concebidos. Destarte, é importante avaliar se a forma com que a propagação e a busca operam sobre o modelo repercutem em uma geração eficaz de soluções. No caso de problemas de otimização, cabe ainda a avaliação da qualidade das soluções obtidas com relação tanto ao que poderia ser obtido por outras técnicas cobrindo o mesmo nicho quanto pelo ganho potencial representado por pequenas modificações sobre as soluções obtidas – o que evidenciaria falta de qualidade nas soluções geradas. Portanto, a adequação da sintaxe de modelagem a um problema representa uma condição necessária mas não suficiente para o sucesso da abordagem de tal problema pela técnica.

No que diz respeito aos tipos de problema com modelagem facilitada em PR, consideramos aqueles contendo apenas decisões discretas e para os quais, geralmente, a formulação utilizando Programação Inteira (PI) não é uma opção devido à explosão no número de variáveis e/ou restrições e à dificuldade para satisfazer as restrições de integralidade. Tal é o caso, por exemplo, de

⁹Denominadas *soft constraints* em inglês.

¹⁰Denominada *Meta-Learning (ML)* em inglês.

muitos problemas de escalonamento: muitos deles são formulados com variáveis 0/1 em que cada variável, digamos y_{ijt} , expressa um predicado como “atividade i está sendo realizada com configuração j no instante de tempo t ”. No caso da PI, esse tipo de variável permite que o problema seja modelado empregando apenas relações lineares no conjunto de restrições, de forma a se beneficiar dos algoritmos mais eficientes de PI que poderiam possivelmente resolver o problema. Entretanto, tal decomposição em variáveis binárias – pelo produto cartesiano das dimensões do problema – cresce em tamanho muito rapidamente, especialmente se a dimensão temporal é aumentada. Portanto, apesar do fato de cada variável de decisão em PR requerer muito mais memória de acordo com o manual da [IBM \(2010\)](#), ela possui um valor semântico muito maior e pode representar tanta informação quanto diversas variáveis binárias combinadas. Esse mesmo argumento pode ser aplicado com relação ao conjunto de restrições. Dessa forma, o papel da PR acaba por ser complementar ao de outras técnicas e, no que diz respeito à modelagem, sua aplicação é favorecida em domínios de problemas nos quais representações mais compactas são possíveis – não se olvidando a questão de desempenho do modelo mencionada anteriormente.

Exemplo de Aplicação: Rotulação Graciosa de Grafos

Pelo perfil de problemas que a PR se propõe a resolver, a opção mais direta para exemplificar sua aplicação seria por meio de modelos de problemas combinatórios clássicos para os quais foi observado um desempenho razoável. Entretanto, isso não agregaria tanto quanto apontar um caso em que foi percebida uma vantagem considerável em um nicho com menos abordagens bem-sucedidas. E esse é o caso do problema da rotulação graciosa¹¹ de grafos.

Diz-se que um grafo $G = (V, E)$ com $n = |V|$ vértices e $m = |E|$ arestas possui uma rotulação graciosa se é possível atribuir a cada um de seus vértices um rótulo único $f : V \rightarrow \{0, 1, 2 \dots m\}$, e a cada uma de suas arestas um rótulo único $g : E \rightarrow \{1, 2 \dots m\}$, de modo que $g(ij) = |f(i) - f(j)|$ para cada aresta $ij \in E$, onde $i, j \in V$. O problema consiste em encontrar para um dado grafo uma tal rotulação graciosa, se existente.

De acordo com [Lustig e Puget \(2001\)](#), mesmo para pequenas instâncias desse problema, não se sabia se estas possuíam ou não uma rotulação graciosa até que a PR fosse aplicada. Como forma de explicar o que significa uma modelagem de PR na prática, a seguir é apresentado um modelo adaptado de [Lustig e Puget \(2001\)](#) para esse problema na linguagem OPL. A adaptação foi necessária devido às mudanças a que a linguagem foi submetida ao longo da última década. Para facilitar a compreensão, as variáveis e domínios do modelo tiveram sua nomenclatura traduzida. Ainda que não seja o escopo dessa parte do trabalho a descrição de uma linguagem de modelagem para PR, o exemplo apresentado tem um papel importante por materializar a modelagem em PR e por permitir uma explicação no nível necessário aos objetivos presentes. Para maiores informações sobre a linguagem OPL, o leitor pode se referir ao Apêndice [A](#).

¹¹ Tradução livre da expressão *graceful labeling* em inglês.

```

1  int noVertices = ...;
2  range dVertices = 1..noVertices;
3  tuple Aresta {
4    Vertices i;
5    Vertices j;
6  };
7  {Aresta} arestas = ...;
8  int noArestas = card(arestas);
9  range dRotulos = 0..noArestas;
10 dvar int rotuloVertice[dVertices] in dRotulos;
11 dvar int rotuloAresta[arestas] in dRotulos;
12 subject to {
13     allDifferent(rotuloVertice);
14     allDifferent(rotuloAresta);
15     forall (a in arestas) {
16         rotuloAresta[a] = abs(rotuloVertice[a.i] - rotuloVertice[a.j]);
17         rotuloAresta[a] > 0;
18     }
19 };

```

As primeiras nove linhas do modelo descrevem estruturas e entradas de dados. A declaração explícita de domínio das rotulações na linha 9 serve como subsídio à declaração das variáveis de decisão. As variáveis de decisão referem-se aos valores de rotulação dos vértices e arestas, conforme declarado nas linhas 10 e 11. Como pode ser observado pela ausência de um critério de otimização no código, trata-se de um problema estritamente de satisfabilidade. A restrição *allDifferent*, representada nas linhas 13 e 14 pelo nome *allDifferent* em OPL, define que os valores atribuídos a cada entrada do vetor de variáveis de decisão dado como argumento sejam únicos, de forma que vértices distintos ou arestas distintas são tenham a mesma rotulação. Essa restrição está presente em muitos modelos de problemas combinatórios e é descrita em maior detalhe na seção 2.2.2. Na linha 15, é declarado um bloco de restrições indexado sobre cada aresta do grafo, garantindo na linha 16 a diferença absoluta de rotulação dos vértices conforme exigido pelo problema. Na linha 17, define-se que a rotulação nula não seja dada a nenhuma aresta. Uma última observação importante do modelo refere-se ao domínio das variáveis de decisão: em uma formulação de PI, por exemplo, não seria possível expressar uma restrição como *allDifferent* empregando o mesmo conjunto de variáveis de decisão adotado.

Outros Exemplos de Aplicação: Problema das n Rainhas e *Sudoku*

Outros dois exemplos de problemas mais conhecidos e que ilustram a expressividade da técnica são o problema das n rainhas e o *sudoku*. O primeiro deles, que consiste em posicionar n rainhas em um tabuleiro de xadrez sem nenhum par possa se atacar, é apresentado na seção A.4.9 do apêndice. O segundo, que consiste em preencher um quadro de 9x9 com os números 1 a 9 de tal forma que os números não se repitam em nenhuma linha, coluna ou quadrante 3x3, possui uma formulação bastante concisa por meio de restrições *allDifferent*. Dessa forma, o modelo apresentado a seguir pode ser utilizado para resolver problemas de *sudoku*, sendo que a inclusão de valores fixos a algumas posições é facilmente tratada pela inclusão de restrições adicionais ou pela declaração de um domínio mais restrito na primeira linha do modelo:

```

1  dvar int rotuloPosicao [1..9][1..9] in 1..9;
2  subject to {
3      forall(i in 1..9) {
4          allDifferent(rotuloPosicao[i]);
5          allDifferent(all(j in 1..9) rotuloPosicao[j][i]);
6      }
7      forall(i in 0..2, j in 0..2) {
8          allDifferent(
9              all(ii in i*3 + 1..i*3+3, jj in j*3 + 1..j*3 +3)
10                 rotuloPosicao[ii][jj] );
11      }
12  };

```

A restrição *alldifferent* da linha 4 do modelo acima garante que o mesmo valor não seja usado mais de uma vez na mesma linha do quadro. A linha 5 garante o mesmo para cada coluna do quadro. Nas linha 8 a 10, isso é garantido para cada um dos 9 quadros 3x3 resultantes do particionamento do quadro 9x9 em três partes horizontes e três partes verticais.

2.1.2 Boas Práticas em Modelagem

A despeito dessa natureza declarativa e da generalidade com relação às restrições possíveis, já foi observado anteriormente que não se pode ignorar o impacto da modelagem sobre o desempenho do sistema. Dito de outra forma, é importante notar que a diferentes modelos com a mesma finalidade corresponderão desempenhos diversos.

Um caso bem comum desse fator diz respeito à existência de simetrias no modelo, tendo como consequência um espaço de busca maior do que o necessário. Consideremos, por exemplo, a situação mencionada por [Hentenryck et al. \(2000\)](#) em que diferentes configurações de um tabuleiro de xadrez diferem apenas por rotação: tais soluções seriam equivalentes para a resolução do problema sendo abordado? Em caso positivo, pode-se partir para a identificação de simetrias a serem quebradas por meio do estabelecimento de restrições adicionais que reduzam o espaço de busca. Tal redução deve ser feita sem prejuízo à corretude, por meio da preservação de ao menos uma solução equivalente a cada solução do modelo original. Cabe a ressalva de que a quebra de simetria pode trazer mais sobrecarga do que benefício. Isso pode ocorrer se o foco não for uma busca exaustiva e a quebra representada pela novas restrições incidir sobre alguma parte do espaço de soluções que, de outra forma, seria explorado antes e com maior proveito. Dessa forma, compete que a identificação de simetrias seja observada na modelagem, mas a implementação de uma quebra deve atentar à possibilidade de ganho em desempenho para a resolução do sistema.

Além da questão sobre as simetrias, certos problemas podem admitir diversos modelos que representem o mesmo espaço de busca. A esse respeito, o manual da [IBM \(2010\)](#) enfatiza que o estabelecimento de boas práticas para a modelagem usando PR ainda é uma área bastante cinzenta, para a qual não existe uma consolidação similar à descrita para a PM em [Williams \(2005\)](#). De certa forma, essa discussão não pode ser feita apenas no nível da modelagem, uma vez que a qualificação do desempenho do modelo depende de conceitos relativos à propagação e à busca. Como exemplo dessa correlação, [Brailsford et al. \(1999\)](#) destaca a importância do emprego de restrições redundantes no modelo para que a precisão do modelo não se deva às mesmas restrições que mantém o nível de propagação, podendo estas serem mais simples do que aquelas. Assim sendo, cabe que esse assunto

seja sempre discutido em caráter transversal aos elementos de modelo, busca e propagação da PR.

2.2 Propagação de Restrições

A propagação de restrições consiste na geração de inferências sobre as restrições empregadas no modelo como forma de reduzir o esforço necessário à realização da busca. Tais inferências consistem na inspeção de quais valores das variáveis são incompatíveis entre si em função das restrições, levando à remoção dos valores dos domínios das variáveis antes ou durante a busca. Essas reduções são fundamentadas no princípio de que nenhuma solução ao problema possuirá os valores eliminados atribuídos às suas respectivas variáveis. Se uma remoção é realizada antes da busca, isso significa que nunca será desfeita porque não é circunstancial. No caso de ser feita durante a busca, ela pode ser reflexo de decisões tomadas ao longo da resolução. Nesse caso, ela pode ou não ser desfeita até o término da busca, de acordo com a necessidade de reverter tais decisões para que não haja prejuízo à completude do processo. O nível em que essa inspeção de compatibilidade entre as variáveis é realizada impacta tanto na redução que pode ser obtida para os domínios quanto nos recursos computacionais necessários para que tal redução seja atingida. Portanto, compete avaliar onde reside a melhor relação de custo-benefício entre o esforço necessário e a redução de domínios obtida para cada caso. De modo geral, a propagação permite que as soluções ao problema se tornem mais evidentes e que as inconsistências remanescentes não prejudiquem tanto o desempenho da busca para encontrá-las.

A propagação é realizada por técnicas de consistência dos mais variados tipos e níveis. Com relação ao tipo, existem técnicas dedicadas a uma restrição específica ou a uma classe delas, como a classe de restrições que determinam a relação entre um determinado número de variáveis. Para cada classe de restrições, são estabelecidos níveis de consistência em termos de certificados sobre quais configurações inválidas são evitadas por meio das inferências de redução dos domínios, como quantos valores podem ser atribuídos sem que ocorra um conflito. Tais inferências podem ser produzidas por diversos algoritmos possíveis, ditos algoritmos de filtragem. Em razão da existência de mais de um nível para a propagação, torna-se evidente o fato que as reduções obtidas variam e, por conseguinte, a eliminação de atribuições inconsistentes não é sempre completa. Na prática, [Barták \(2001\)](#) observa que a consistência completa do modelo – situação na qual a busca sobre o modelo é feita sem que nenhuma decisão da busca tenha que ser revertida ou que o modelo é tido como insatisfatório por algum domínio ter se tornado vazio – não costuma ser uma opção viável devido ao esforço necessário para atingi-la. Afinal, reduzir os domínios a tal ponto que a busca seja feita em tempo linear no número de variáveis – ou seja, em tempo fixo por atribuição – corresponderia a delegar à propagação a resolução do problema NP-Completo em que consiste o PSR no caso geral. Além disso, os níveis mais elementares costumam garantir a remoção da maior parte dos valores inconsistentes, de forma que compensa mais explorar a combinação de outras técnicas com algoritmos de consistência que não consumam muito tempo. Técnicas mais generalistas costumam ser aplicadas a tipos mais simples e em níveis menores, ao passo que outras mais especializadas são mais eficazes na tarefa de redução de domínios restritas a alguns tipos de restrições. No que se segue, as técnicas mais comuns de consistência são apresentadas das mais generalistas e simples para as mais especializadas e elaboradas. Para maiores referências sobre a síntese feita da literatura a respeito de propagação, veja [Barták \(2001, 1999\)](#); [Dechter \(2003\)](#).

2.2.1 Técnicas de Consistência Binária

O tipo mais simples e comum de técnica de consistência estudado envolve pares de variáveis, sendo conhecido como consistência binária. Ela é aplicada a restrições com uma ou duas variáveis, respectivamente denominadas de restrições unárias e binárias. Nesse tipo de consistência, o modelo pode ser analisado por meio de um grafo $G = (V, E)$, sendo cada variável de decisão x_i representada por um vértice $v_i \in V$, cada restrição binária $r_j(x_{i_1}, x_{i_2})$ por uma aresta $v_{i_1}v_{i_2} \in E$ incidente aos vértices $v_{i_1}, v_{i_2} \in V$ e cada restrição unária $r_j(x_i)$ por um laço $v_i v_i \in E$ duplamente incidente ao vértice $v_i \in V$. Um vértice é dito consistente se nenhum valor do domínio de sua variável correspondente está em desacordo com qualquer restrição unária. Por analogia, uma aresta é dita consistente se para cada valor do domínio de uma de suas variáveis associadas existir um valor correspondente no domínio da outra, de forma que a restrição possa ser satisfeita se tal valor for atribuído à variável. Se todas as arestas estiverem consistentes, então o modelo é dito consistente nesse nível. Cabe ressaltar que, historicamente, o nome dado a esse nível de consistência é o de consistência de arco e ao anterior o de consistência de nó, sendo muito comum encontrar na literatura uma nomenclatura sobre consistência binária trocando vértices por nós e arestas por arcos¹². Por fim, conforme veremos a seguir, níveis mais elevados ou incompletos e intermediários podem ser estabelecidos para a consistência binária.

Antes de proceder ao detalhamento dos níveis e algoritmos de consistência binária, vejamos no modelo apresentado a seguir como os conceitos descritos se aplicam a um caso simples.

```

1  using CP;
2  dvar int x in 1..10;
3  dvar int y in 1..10;
4  subject to {
5      x > 5;
6      x < y;
7  };

```

No modelo ilustrado, temos duas variáveis x e y no domínio dos inteiros entre 1 e 10, conforme descrito nas linhas 2 e 3. Na linha 5, a primeira restrição força o valor de x a ser maior que 5. Dessa forma, a propagação já poderia inferir que o domínio de x está reduzido, na prática, ao intervalo entre 6 e 10. Na linha 6, a relação entre x e y permite reduzir ainda mais o domínio de x , visto que ele não pode assumir o valor 10, ao passo que o domínio de y nas soluções se confina ao intervalo entre 7 e 10. Assim sendo, os domínios das variáveis x e y podem ser reduzidos de 10 possíveis valores para 4 cada.

A consistência ao nível dos vértices é atingida com a primeira inferência sugerida sobre a restrição da linha 5. No caso geral, ela pode ser atingida com uma única inspeção a cada restrição unária do modelo. Por sua vez, a consistência ao nível das arestas é atingida gradativamente por uma sucessão de aplicações que garantem a consistência individual de arestas. Conforme observado no exemplo, uma aresta pode ser inspecionada mais de uma vez para que o nível de consistência global seja atingido em razão dos efeitos colaterais das inspeções de arestas anteriores. Convém enfatizar que esse processo é sempre finito, uma vez que novas inspeções a uma mesma aresta só ocorrem em

¹² Apesar de ser uma prática aceita, essa nomenclatura não é reforçada nesse trabalho por não ser aquela empregada em teoria dos grafos, sob o risco de prejudicar a explicação dos conceitos ao público de interesse, que não corresponde àquele que já usa a técnica.

função de reduções prévias, sendo o número total de reduções limitado à soma dos tamanhos dos domínios das variáveis do modelo.

Níveis de Consistência Binária

Conforme mencionado anteriormente, é possível arbitrar diversas séries de níveis de consistência. Na consistência binária, o caso padrão é dado sequencialmente pelo número de variáveis para as quais é possível realizar uma atribuição sem que um conflito ocorra. Abaixo segue uma listagem desde os casos mais simples, passando pelas generalizações até atingir a consistência completa do modelo:

- **Consistência em nível de vértice ou 1-consistência**¹³ Diz-se que o vértice v_i do grafo do modelo associado à variável de decisão x_i é consistente se qualquer valor $\bar{x}_i$ do domínio D_i puder ser atribuído a x_i sem que ocorra um conflito com as restrições unárias do modelo, i.e., restrições envolvendo apenas x_i . Sendo todos os vértices do grafo do modelo consistentes, o modelo é dito consistente em nível de vértice.
- **Consistência em nível de aresta ou 2-consistência**¹⁴ Uma aresta $v_{i_1}v_{i_2}$ do grafo do modelo, associada a uma restrição r_j envolvendo as variáveis de decisão x_{i_1} e x_{i_2} do modelo, é dita consistente se, para qualquer valor $\bar{x}_{i_1} \in D_{i_1}$, existir um valor $\bar{x}_{i_2} \in D_{i_2}$ tal que $r_j(\bar{x}_{i_1}, \bar{x}_{i_2})$ seja satisfeita e vice-versa, i.e., invertendo as posições de $\bar{x}_{i_1}$ e $\bar{x}_{i_2}$ na condição. Quando todas as arestas são consistentes, o modelo é dito consistente em nível de aresta.
- **Consistência em nível de caminho ou 3-consistência**¹⁵ Um caminho $v_{i_1}v_{i_2} \dots v_{i_k}$ é consistente se, para qualquer atribuição $x_{i_1} \leftarrow \bar{x}_{i_1} : \bar{x}_{i_1} \in D_{i_1}$ e $x_{i_k} \leftarrow \bar{x}_{i_k} : \bar{x}_{i_k} \in D_{i_k}$, existirem valores $\bar{x}_{i_2} \in D_{i_2}, \dots, \bar{x}_{i_{k-1}} \in D_{i_{k-1}}$ de forma que as arestas associadas às restrições de cada par consecutivo de vértices do caminho esteja satisfeita. Conforme [Barták \(1999\)](#) e [Dechter \(2003\)](#) ressaltam, se a consistência dos caminhos de tamanho 2 (duas arestas e $k = 3$) for satisfeita para o grafo do modelo, então todos os caminhos são consistentes e o modelo é consistente em nível de caminho.
- **Consistência em k -ésimo nível ou k -consistência**¹⁶ Dado um conjunto de valores para atribuição a quaisquer $k - 1$ variáveis de um modelo dentro de seus respectivos domínios – $x_{i_1} \leftarrow \bar{x}_{i_1} : \bar{x}_{i_1} \in D_{i_1}, x_{i_2} \leftarrow \bar{x}_{i_2} : \bar{x}_{i_2} \in D_{i_2}, \dots, x_{i_{k-1}} \leftarrow \bar{x}_{i_{k-1}} : \bar{x}_{i_{k-1}} \in D_{i_{k-1}}$ – o modelo é dito k -consistente se sempre houver um valor para atribuição no domínio de qualquer outra k -ésima variável – $x_{i_k} \leftarrow \bar{x}_{i_k} : \bar{x}_{i_k} \in D_{i_k}$ – tal que nenhuma restrição do modelo seja violada por tais atribuições. A k -consistência representa uma generalização dos níveis apresentados acima.
- **Consistência forte em k -ésimo nível ou k -consistência forte**¹⁷ Um modelo é fortemente k -consistente se ele for k' -consistente para qualquer $k' \leq k$. Conforme enfatizado por [Dechter \(2003\)](#), o fato de um modelo ser k' -consistente não implica que ele seja $(k' - 1)$ -consistente.

¹³ Denominada *node consistency* ou *1-consistency* em inglês.

¹⁴ Denominada *arc consistency* ou *2-consistency* em inglês.

¹⁵ Denominada *path consistency* ou *3-consistency* em inglês.

¹⁶ Denominada *k-consistency* em inglês.

¹⁷ Denominada *strong k-consistency* em inglês.

- **Consistência completa do modelo ou n -consistência**¹⁸ Um modelo é completamente consistente se qualquer atribuição de valores dos domínios às variáveis de decisão – $x_{i_1} \leftarrow \bar{x}_{i_1} : \bar{x}_{i_1} \in D_{i_1}, x_{i_2} \leftarrow \bar{x}_{i_2} : \bar{x}_{i_2} \in D_{i_2}, \dots, x_{i_n} \leftarrow \bar{x}_{i_n} : \bar{x}_{i_n} \in D_{i_n}$ – satisfizer às restrições do modelo, de forma que – conforme observado anteriormente – passa a ser possível obter uma solução ao problema em tempo $O(n)$. De acordo com Focacci *et al.* (2000), esse nível corresponde naturalmente à n -ésima consistência, mas pode ser atingido se a $k + 1$ -ésima consistência for verificada em um grafo de modelo cujo vértice de maior grau¹⁹ tenha valor k . Sem embargo, convém ressaltar que uma consistência completa não depende apenas da redução do domínio das variáveis de decisão, mas também da redução dos domínios das restrições. Tais domínios são representados como subconjuntos das tuplas das variáveis de decisão associadas a cada restrição. Em última análise, a redução dos mesmos significa mudar a própria semântica da restrição nos modelos com consistência mais elevada. Isso se opera para $n \geq 3$ no caso binário, pois já na inspeção de caminhos entre pares de vértices é possível, por exemplo, que as respectivas variáveis não tivessem nenhuma restrição mútua e suas combinações de valores concorrentes tivessem que ser limitadas para garantir a consistência em nível de caminho.

Outras séries foram estudadas – ainda que em menor grau – a partir da extensão da série acima, representando cada caso uma alternativa de generalização, diversidade ou compromisso com relação ao paradigma estabelecido. Pela inclusão de parâmetros adicionais à definição dos níveis de consistência, pode-se generalizar os níveis pré-estabelecidos. Alternativamente, a ponderação de parâmetros diversos possibilita uma perspectiva mais ampla sobre os certificados de consistência possíveis. Por fim, o estabelecimento de níveis intermediários sobre a parametrização de maior aceitação representa um maior nível de detalhamento acerca dos efeitos da propagação. Dentre essas possibilidades, esta despertou maior atenção na literatura em razão de sua proximidade e potencial de catalisação de resultados com relação à abordagem tradicional.

O estabelecimento de níveis intermediários parte de uma lógica muito similar àquela aplicada na definição dos níveis originais. Uma vez que a verificação das restrições para garantir um nível de consistência pode ter efeitos colaterais no que diz respeito à necessidade de uma nova iteração para filtragem, passou-se a apurar o impacto dessas novas iterações para a redução dos domínios e o esforço computacional que exigiam. A observação empírica de muitos autores a esse respeito é de que a quantidade de valores inconsistentes que são removidos dos domínios é muito maior nas primeiras verificações de propagação das restrições do modelo do que nas iterações subsequentes. Dessa forma, o esforço na remoção de valores nas primeiras verificações tem maior compensação do que aquele observado nas remoções seguintes. Em outros casos, foi observado que a forma com que o modelo em estado consistente é utilizado na busca pode fazer com que certo esforço adicional tenha sido em vão. Por conseguinte, a revisão exaustiva para a manutenção do nível de consistência em um determinado nível passa a ser tão questionável quanto a manutenção de níveis de consistência mais elevados.

A seguir estão listadas algumas das definições alternativas de níveis de propagação:

- **Consistência direcional em nível de aresta**²⁰ A consistência direcional em nível de aresta é definida exclusivamente sobre uma orientação do grafo do modelo, i.e., cada aresta tem sua

¹⁸ Denominada *full consistency* ou *n-consistency* em inglês.

¹⁹ O grau de um vértice é dado pelo número de arestas incidentes ao mesmo no grafo.

²⁰ Denominada *directed arc consistency* em inglês.

consistência verificada em apenas um sentido. Esse sentido é definido de acordo com uma ordenação prévia dos vértices do grafo do modelo: uma aresta $v_{i_1}v_{i_2}$ é verificada de v_{i_1} para v_{i_2} se a variável x_{i_1} preceder x_{i_2} na ordenação definida. Se as arestas forem percorridas de acordo com a ordenação topológica dos vértices correspondente à ordenação das variáveis de decisão, i.e., $v_{i_1}v_{i_2}$ sendo verificada antes de $v_{i_3}v_{i_4}$ se x_{i_2} preceder x_{i_4} na ordem, então nenhuma aresta precisa ser inspecionada pela segunda vez. Esse nível de consistência é mais fraco que a consistência ao nível da aresta, mas pode ser equivalente em alguns modelos, como aqueles representados por árvores. Além disso, de acordo com [Dechter \(2003\)](#), o benefício da propagação nesse nível é sempre equivalente ao da consistência ao nível da aresta em uma busca que atribua valores às variáveis de acordo com a mesma ordenação das variáveis de decisão usada para a manutenção de consistência.

- **Consistência restrita em nível de caminho**²¹ A consistência restrita em nível de caminho encerra a consistência em nível de aresta acrescida de um caso específico da consistência em nível de caminho. Esse caso é representado pela verificação da consistência de caminho a partir de duas atribuições $x_{i_1} \leftarrow \bar{x}_{i_1} : \bar{x}_{i_1} \in D_{i_1}$ e $x_{i_k} \leftarrow \bar{x}_{i_k} : \bar{x}_{i_k} \in D_{i_k}$ que devam obrigatoriamente ocorrer em conjunto, i.e., $(x_{i_1} \leftarrow \bar{x}_{i_1}) \leftrightarrow (x_{i_k} \leftarrow \bar{x}_{i_k})$. O acréscimo da verificação desse caso à consistência de aresta se justifica pelo seu potencial na redução dos domínios envolvidos sem que todos os casos possíveis tenham que ser tratados.
- **Consistência por limites**²² A consistência por limites representa uma simplificação de outros níveis de consistência pela verificação restrita aos valores mínimo e máximo de cada domínio. O conceito de mínimo e máximo não convém a domínios simbólicos – i.e., em que não existe uma ordem definida entre os elementos do conjunto –, mas facilita a análise em domínios numéricos, como uma faixa sequencial de números inteiros. Em alguns casos, a verificação dos extremos do domínio basta para atingir um determinado nível de consistência. Quando isso não é possível, sua aplicação pode decorrer da constatação de que a diferença de redução resultante é pontual e é compensada pelo menor tempo de processamento resultante da escolha.
- **Consistência por conjunto unitário**²³ A consistência por conjunto unitário representa uma restrição adicional que pode ser aplicada a outros níveis de consistência. Ela define que um determinado nível de consistência seja atingido para qualquer atribuição $x_i \leftarrow \bar{x}_i : \bar{x}_i \in D_i$ possível do modelo, i.e., o modelo possui tal nível de consistência para qualquer atribuição adicional que seja feita.
- **Consistência (i, j)** ²⁴ A consistência (i, j) representa uma generalização da k -consistência: enquanto nesta garante-se que a atribuição de valores a $k - 1$ variáveis de decisão não impeça a atribuição de valor a uma outra variável de decisão, naquela a garantia é de que a atribuição de valores a i variáveis de decisão não impeça a atribuição de valores a outras j variáveis de decisão. Dessa forma, a k -consistência é equivalente à consistência $(k - 1, 1)$.

²¹ Denominada *path-restricted consistency* em inglês.

²² Denominada *bounds consistency* em inglês.

²³ Denominada *singleton consistency* em inglês.

²⁴ Denominada (i, j) -consistency em inglês.

- **Consistência inversa**²⁵ A consistência inversa representa uma especialização da consistência (i, j) no sentido oposto ao da k -consistência: um modelo é inversamente k -consistente se tiver consistência $(1, k-1)$, i.e., a cada atribuição $x_{i_1} \leftarrow \bar{x}_{i_1} : \bar{x}_{i_1} \in D_{i_1}$ deve corresponder ao menos uma atribuição $x_{i_2} \leftarrow \bar{x}_{i_2} : \bar{x}_{i_2} \in D_{i_2}, \dots, x_{i_n} \leftarrow \bar{x}_{i_n} : \bar{x}_{i_n} \in D_{i_n}$ que não viole as restrições do modelo, para quaisquer k variáveis de decisão.

Algoritmos de Filtragem para Consistência Binária

A garantia de um nível de consistência pode ser feita por diversos algoritmos, sendo possível conceber muitos deles diretamente da definição. A presente seção ilustra os algoritmos dos níveis gerais para os quais maior atenção foi dada. De forma a não tomar tanto espaço em um texto introdutório, apenas a lógica por trás de tais algoritmos é descrita. Longe de se prestar como uma apresentação exaustiva, a sucessão de tais algoritmos de filtragem ilustra como os métodos de manutenção de consistência podem ser aperfeiçoados a partir de sua concepção original.

Para garantir a consistência em nível de vértice, um algoritmo que realize o mínimo esforço necessário pode ser trivialmente concebido: uma inspeção sobre cada restrição unária basta para atingir o nível. Assim sendo, o desenvolvimento descrito a seguir cobre os algoritmos de filtragem para a consistência em nível de aresta e de caminho. Os algoritmos para níveis intermediários de consistência representam modificações sobre aqueles para consistência precisa. Em última análise, o problema de garantir a consistência completa de um modelo que descreve um problema combinatório NP-Completo – como o PSR no caso geral – também será NP-Completo.

Algoritmos para consistência em nível de aresta Existe um padrão na literatura baseado na nomenclatura dos algoritmos para consistência ao nível da aresta pelo prefixo AC²⁶ seguido de um índice sequencial. Nessa linha, o algoritmo **AC-1** representa o método mais intuitivo, consistindo na repetição da revisão dos domínios das variáveis associadas aos vértices até que nenhuma alteração ocorra. No algoritmo **AC-2**, apenas as revisões de domínio necessárias são repetidas: o domínio da variável de um vértice só é revisado se ele for adjacente a qualquer vértice cuja variável tenha tido o domínio reduzido desde sua última revisão. No **AC-3**, o processo de revisão é aperfeiçoado pela operação por iteração em uma fila de arestas: inserções ocorrem quando o domínio da variável de um vértice incidente é alterado e remoções ocorrem quando essas arestas são revisadas. O ganho obtido pelo uso da fila se deve à prevenção de que arestas sejam revisadas mais de uma vez. O algoritmo **AC-4** itera sobre estruturas um pouco mais elaboradas, que evitam a revisão de domínios pela iteração sobre as arestas: para cada aresta $v_{i_1} v_{i_2}$ do grafo do modelo, cada valor $\bar{x}_{i_1} \in D_{i_1}$ da variável x_{i_1} associada ao vértice v_{i_1} possui um conjunto de valores de suporte $VS_{\bar{x}_{i_1} i_2} \subseteq D_{i_2}$, correspondendo àqueles que não são incompatíveis com a atribuição $x_{i_1} \leftarrow \bar{x}_{i_1}$. Essa idéia foi simplificada no **AC-6**, que armazena apenas um valor de suporte $vs_{\bar{x}_{i_1} i_2} \in VS_{\bar{x}_{i_1} i_2}$ para cada valor do domínio de D_{i_1} e procura um novo valor de suporte se o anterior passar a ser inválido, evitando a utilização proibitiva de memória e tempo de seu antecessor. O **AC-7** representa uma redução de esforço pela simples constatação de que o suporte de valores é simétrico: se o valor $\bar{x}_{i_1} \in D_{i_1}$ suporta $\bar{x}_{i_2} \in D_{i_2}$, $\bar{x}_{i_2} \in D_{i_2}$ suporta $\bar{x}_{i_1} \in D_{i_1}$, i.e., $(vs_{\bar{x}_{i_1} i_2} \leftarrow \bar{x}_{i_2}) \leftrightarrow (vs_{\bar{x}_{i_2} i_1} \leftarrow \bar{x}_{i_1})$. Com relação à análise de complexidade assintótica do tempo de execução no pior caso, sendo m o número de restrições, n o número de

²⁵ Denominada *inverse consistency* em inglês.

²⁶ Acrônimo em inglês para *Arc Consistency*.

vértices e k o tamanho do maior domínio considerado, AC-1 e AC-2 são $O(mnk^3)$, AC-3 é $O(mk^3)$ e o limite inferior provado ao caso geral de $O(mk^2)$ é constatado para o AC-4 em diante.

Com relação ao algoritmo **AC-5** descrito em [Hentenryck et al. \(1992\)](#), convém uma análise mais apurada devido à forma com que esse algoritmo representa o conceito da especialização dos mecanismos de resolução em função das restrições. O AC-5 não é completamente definido, sendo possível parametrizá-lo por meio de implementações diferenciadas para dois métodos: um para a verificação inicial de quais valores do domínio D_{i_1} não são suportados por nenhum valor do domínio D_{i_2} ; outro para a verificação iterativa em fila de quais valores do domínio D_{i_1} não são mais suportados se o valor $\bar{x}_{i_2}$ for removido do domínio D_{i_2} . Uma implementação possível desses métodos coincide com o AC-3, ao passo que outra coincide com o AC-4. No entanto, a vantagem desse algoritmo se deve ao fato que ele possibilita que a consistência de classes específicas de restrições sejam garantidas a uma complexidade de tempo inferior àquela do caso geral: restrições binárias funcionais – como $x_{i_1} + x_{i_2} = k$, sendo k uma constante –, anti-funcionais – como $x_{i_1} + x_{i_2} \neq k$ – e monotônicas – como $x_{i_1} + x_{i_2} < k$ – em todo o domínio ou por partes podem ser propagadas em tempo $O(mk)$.

Algoritmos para consistência em nível de caminho Os algoritmos para consistência de caminho foram desenvolvidos em paralelo aos de consistência de aresta, sendo costumeiramente prefixados como PC²⁷. Dessa forma, podemos resumi-los revisando as idéias apresentadas anteriormente. O algoritmo **PC-1** consiste na revisão de todos os caminhos até que não ocorra nenhuma redução de domínio. No **PC-2**, a verificação é repetida apenas para os caminhos que possam ter sido afetados por alguma revisão prévia. O algoritmo **PC-3** representou a primeira proposta de extensão das idéias do algoritmo AC-4 para a consistência em nível de caminho. Entretanto, ele foi substituído pelo **PC-4** por remover valores que poderiam pertencer a uma atribuição válida, caso evitado pelas correções que diferem este algoritmo daquele. Por fim, o algoritmo **PC-5** representou para o PC-4 a mesma simplificação de implementação que o AC-6 representou para o AC-4, reduzindo a quantidade de memória utilizada em estruturas de dados auxiliares.

Algoritmos para níveis intermediários de consistência Por ser efetuada em uma única passagem, não existem nuances de aperfeiçoamento para a execução do nível de consistência direcional em nível de aresta, à semelhança do que observou-se para a consistência em nível de vértice. Por sua vez, a consistência restrita ao nível do caminho foi concebida como uma especialização do algoritmo AC-4, havendo verificação de consistência de caminhos entre as atribuições $x_{i_1} \leftarrow \bar{x}_{i_1}$ e $x_{i_2} \leftarrow \bar{x}_{i_2}$ apenas quando $|VS_{\bar{x}_{i_1}i_2}| = 1$, i.e., o conjunto de suporte de uma atribuição se restringir a uma única atribuição à outra variável.

Considerações sobre os demais níveis de consistência De acordo com [Barták \(1999\)](#), apesar de existirem algoritmos para k -consistência forte com $k > 2$, seu custo computacional raramente compensa pelo fato de que ainda não são capazes de isentar a busca de falhas potenciais. Com relação aos níveis alternativos de consistência, [Barták \(2001\)](#) observa que a manutenção de i -uplas para a manutenção da consistência (i, j) torna-se proibitiva para $i \geq 2$. No caso da consistência inversa, só há interesse para $k = 3$, uma vez que ela coincide com a consistência convencional em

²⁷ Acrônimo em inglês para *Path Consistency*.

níveis menores e torna-se proibitiva para níveis maiores. Por fim, a implementação de algoritmos de filtragem para consistência por conjunto unitário geralmente consiste no acréscimo de um laço superior que fixa cada atribuição possível.

Binarização de Restrições

A hipótese de representar um PSR por meio de predicados com aridade máxima 2 – i.e., via restrições envolvendo até duas variáveis de decisão cada – para que a propagação binária possa ser empregada é bastante restritiva. Ainda que possível, ela pode levar a modelos demasiado extensos, de forma que muitas simplificações não são possíveis senão por representação imprecisas. Dessa limitação surge o impasse entre empregar uma representação simplificada, que possa se beneficiar da propagação binária; ou recorrer a uma gama mais ampla de representações, com potencial a uma propagação mais eficaz no espaço de busca ao risco de um custo computacional de execução mais elevado. Entretanto, a opção a ser feita não é exclusiva, pois o uso ativo de restrições redundantes na modelagem – mencionado na seção 2.1.2 – condiz com o emprego de ambas as possibilidades: restrições precisas que são complementadas por aproximações que possibilitem a propagação em um nível mais elementar. Nessa seção, é explorada a possibilidade de simplificar automaticamente a representação por meio de decomposições binárias. Adiante, será discutida a hipótese de trabalhar com representações mais elaboradas.

A decomposição em um modelo binário pode se dar por diversas formas. Ainda que algumas delas levem a modelos imprecisos, é importante observar que, à semelhança do caso das restrições redundantes, o objetivo de tais modelos não é substituir o modelo original na verificação da satisfação das restrições, mas tão somente na operação da propagação. A seguir, constam três decomposições identificadas por [Barták \(2001\)](#):

- **Projeção** A projeção consiste na aproximação de uma restrição por um conjunto de relações binárias. Para cada par de variáveis envolvido na restrição original, delimita-se o domínio possível para uma variável em função da outra no contexto da restrição. Formalmente, para cada par de variáveis x_{i_1} e x_{i_2} de uma restrição $r_j(\dots, x_{i_1}, \dots, x_{i_2}, \dots)$, define-se uma restrição $r'_{ji_1i_2}(x_{i_1}, x_{i_2})$ satisfeita apenas por duplas $(\bar{x}_{i_1}, \bar{x}_{i_2})$ que possam concorrer em uma solução – i.e., $S_{ji_1i_2} = \{(\bar{x}_{i_1}, \bar{x}_{i_2}) \in D_{i_1} \times D_{i_2} : \exists (\bar{x}_1, \dots, \bar{x}_{i_1}, \dots, \bar{x}_{i_2}, \dots, \bar{x}_n) \in S_j, \bar{x}_1 \in D_1, \dots, \bar{x}_{i_1-1} \in D_{i_1-1}, \bar{x}_{i_1+1} \in D_{i_1+1}, \dots, \bar{x}_{i_2-1} \in D_{i_2-1}, \bar{x}_{i_2+1} \in D_{i_2+1}, \dots, \bar{x}_n \in D_n\}$. Em alguns casos, a projeção obtida é tão restritiva quanto a restrição original, mas em outros ela não atinge o mesmo nível e representa uma relaxação do modelo.
- **Codificação dual** A codificação dual refere-se à dualização do modelo para representação por meio de um grafo, feita pela inversão entre variáveis e restrições. Um vértice v_j do grafo representa a restrição r_j do modelo original, tendo como domínio S_j – i.e., todas as tuplas de atribuições válidas para satisfazer a restrição. Por sua vez, cada aresta representa a intersecção de variáveis de decisão entre restrições distintas – i.e., os vértices relativos a duas restrições distintas são incidentes a uma aresta em comum se tais restrições compartilharem uma determinada variável de decisão. Dessa forma, a operação da propagação nesse conjunto de arestas do grafo da dualização do modelo garante a unicidade do valor atribuído a uma variável de decisão do modelo.

- **Codificação por variáveis ocultas** Na codificação por variáveis ocultas, tanto os domínios das variáveis de decisão quanto os domínios das restrições são considerados. Assim sendo, o grafo resultante possui $n + m$ vértices, sendo o subgrafo induzido pelos n vértices associados às variáveis de decisão isomorfo ao grafo do modelo e o subgrafo induzido pelos m vértices restantes, associados às restrições, isomorfo ao grafo do modelo dual. Dessa forma, a inspeção de consistência opera sobre ambos os tipos de domínio.

2.2.2 Técnicas de Consistência Generalizada

A generalização dos conceitos de consistência previamente explorados para o caso binário pode decorrer de uma necessidade incondicional para que uma determinada abordagem seja viável ou emergir como uma oportunidade de maior benefício em outros. Na primeira situação, tratam-se de PSRs cujo modelo binário obtido não apresenta um nível de propagação adequado para reduzir o esforço necessário à busca. Na segunda, o modelo corresponde ao que se esperaria de um caso geral, mas oportunidades a uma propagação mais eficaz são identificadas. Em ambas as situações, o problema se resume ao fato de a representação binária não ser capaz de atingir o nível de propagação desejado.

Entretanto, cabe ressaltar que a generalização do conceito de consistência para a operação sobre restrições arbitrárias por si só é proibitiva. Em termos de implementação, o caso geral implica na operação sobre hipergrafos: hiper-arestas generalizando o conceito de arestas para um número arbitrário de vértices. Há um certo consenso na literatura de que essa via de abordagem não compensa o esforço devido à dificuldade de implementação e à exigência de recursos computacionais decorrentes. E é nesse ponto que entra em cena o conceito central à PR: empregar restrições adequadas para propagar de modo eficiente sobre a estrutura do problema e se beneficiar de suas especificidades, conforme tratado a seguir.

Restrições Globais

O conceito de restrição global²⁸ representa tanto uma restrição especializada a um domínio de aplicação quanto o algoritmo de filtragem subjacente que garante a manutenção do nível de consistência. O termo “global” atribuído a uma propagação especializada contrasta com as limitações de extensão para que uma propagação generalizada não tome demasiado tempo, razão pela qual esta envolve menos variáveis e é vista como “local”. Dessa forma, a generalização de consistência provida pela restrição global vem ao custo de especializar o algoritmo de filtragem àquele caso. Essa abordagem especializada é compensada pela recorrência com que algumas dessas restrições acabam por ser encontradas na formulação de problemas. Disso resulta o esforço de catalogar sistematicamente tais restrições por meio das propriedades estruturais que elas representam, conforme tratado por Beldiceanu *et al.* (2007). Portanto, a propagação por meio de restrições globais representa um esforço de especialização dos mecanismos de propagação, tendo em vista não apenas o benefício da abordagem imediata como a de outros problemas com a mesma estrutura.

Como pode ser visto pela própria definição apresentada, o conceito de consistência e o algoritmo de filtragem se misturam. Bessière e Hentenryck (2003) noticiam que um tutorial apresentado na

²⁸Denominada *global constraint* em inglês.

conferência CP²⁹ de 2002 levantou um debate acalorado em decorrência de como pesquisadores de áreas distintas concebem restrições globais sob óticas diferentes. Por esse motivo, esses autores sugerem uma definição de níveis progressivos à globalidade de uma restrição:

- **Algorítmico:** a globalidade algorítmica corresponde à obtenção de um algoritmo de filtragem mais eficiente do que aquele correspondente ao caso geral para o nível de consistência atingido.
- **Operacional:** a globalidade operacional corresponde à capacidade de que mais valores sejam removidos do que no caso em que a mesma restrição seja representada por um conjunto de restrições locais.
- **Semântico:** a globalidade semântica corresponde à possibilidade de representar uma restrição que um conjunto de restrições locais não é capaz de representar.

Exemplo de restrição global operacional: *alldifferent*

A restrição *alldifferent* representa uma estrutura comum a diversos problemas combinatórios, como o da rotulação graciosa de grafos descrito na seção 2.1.1. Semanticamente, $\text{alldifferent}(x_1, \dots, x_n)$ corresponde ao conjunto de restrições binárias $x_i \neq x_j \ \forall \ i \in [1, n], j \in [i + 1, n]$ – i.e., define que a atribuição de cada variável de decisão envolvida deve ter um valor diferente das demais. O que confere o caráter de globalidade operacional a essa restrição é que, ao contrário do conjunto de restrições binárias, ela possibilita que se detecte facilmente situações como a do modelo a seguir:

```

1  using CP;
2  dvar int x[1..3] in 1..2;
3  subject to {
4      alldifferent(x);
5  };

```

No modelo, três variáveis de decisão compartilham um domínio contendo dois valores. Pelo princípio da casa dos pombos³⁰, é evidente que qualquer atribuição de valores às variáveis incorrerá em que ao menos duas variáveis compartilhem a atribuição de algum valor, de forma que o modelo é insatisfatível – i.e., não há como satisfazê-lo e ele não tem solução. No entanto, o modelo é consistente em nível de aresta e o tratamento de casos como o apresentado exigiria o emprego de um nível generalizado de consistência.

Régin (1994) observa que garantir a consistência generalizada da restrição *alldifferent* com um algoritmo de filtragem genérico demandaria a inspeção de todas as tuplas de atribuição possíveis – i.e., $A_{k,n} = \frac{k!}{(n-k)!}$, para n variáveis de decisão e k valores distintos nos domínios das variáveis. Dessa feita, Régin (1994) propõe um algoritmo de filtragem baseado no conceito de emparelhamento máximo de grafos bipartidos. Nesse algoritmo, utiliza-se um grafo bipartido no qual os vértices de uma classe da partição estão associados às variáveis de decisão da restrição, os vértices da outra

²⁹ A série de conferências anuais CP, acrônimo reduzido para International Conference on Principles and Practice of Constraint Programming, ou Conferência Internacional sobre os Princípios e a Prática da Programação por Restrições, representa a maior conferência mundial de PR. A 17ª edição do evento ocorreu em Perúgia, na Itália, entre 12 e 16 de setembro de 2011.

³⁰ O “princípio da casa dos pombos” consiste em um jargão empregado para a constatação de que não é possível que n elementos sejam particionados em m conjuntos, para $n > m$, sem que ao menos um dos conjuntos tenha mais do que um elemento.

classe aos valores distintos dos domínios dessas variáveis e as arestas representam as atribuições possíveis entre a variável associada a um vértice da primeira classe e o valor associado a um vértice da segunda. Portanto, a restrição só pode ser satisfeita por alguma atribuição de valores às variáveis se o emparelhamento máximo tiver tantas arestas quantas forem as variáveis envolvidas, sendo que uma atribuição só é permitida se pertencer a algum emparelhamento máximo do grafo. O algoritmo proposto a partir das constatações anteriores vale-se de resultados da teoria dos grafos para atingir a consistência desejada em tempo $O(n^2k^2)$. Por essa razão, o estudo de Régim (1994) representa um caso de particular interesse sobre como o tratamento de especificidades leva à obtenção de um melhor desempenho na resolução de problemas usando PR.

2.3 Busca

Devido ao caráter declarativo da PR, a busca pode operar por meios bastante diversos. Conforme observado anteriormente, os métodos mais empregados para a busca em PR são denominados, de um modo geral, como “tentativa-e-erro”. Isso se deve ao fato de que a propagação – apesar de ser um componente fundamental à técnica – não é viável operando isoladamente, de forma que cabe à busca lidar com certo grau de inconsistência do modelo. Como reflexo, o foco da busca em PR está muito mais ligado à factibilidade do que à otimalidade, i.e., costuma-se priorizar que uma solução seja encontrada em tempo hábil e não tanto que uma solução ótima seja obtida. No entanto, seria errôneo reduzir o ferramental de busca aos algoritmos de recursão e retrocesso – i.e., que atribuem valores a variáveis de maneira ordenada e recuem em caso de conflito -, ainda que boa parte das abordagens para a busca em PR envolvam esse tipo de algoritmo. Dessa forma, compete elencar as alternativas possíveis para então compará-las de acordo com os resultados descritos na literatura.

A forma com que a busca é feita em PR depende do objetivo de sua realização. A primeira dimensão de análise diz respeito à completude: é necessário realizar uma busca exaustiva por todas as soluções possíveis ou basta encontrar uma solução para encerrar a busca? A busca exaustiva, também dita completa, pode ser realizada por algoritmos de recursão e retrocesso ou de programação dinâmica ³¹. Entretanto, a análise de algoritmos de programação dinâmica possui uma importância mais teórica do que prática devido à exigência de memória decorrente da aplicação no caso geral, conforme pode ser visto em Dechter (2003). Para a busca por factibilidade, geralmente incompleta por finalizar sem necessariamente exaurir o espaço de busca, tanto os algoritmos de recursão e retrocesso quanto os de busca local ou estocástica podem ser empregados. A maior diferença entre esses procedimentos consiste na sistemática de exploração e garantia de finalização daqueles, que não necessariamente encontram equivalência nestes devido à forma intermitente e aleatória com que exploram o espaço de busca. Em ambos os casos, artifícios podem ser empregados para orientar a busca a uma conclusão antecipada, tanto para a factibilidade quanto para obter uma solução razoável de acordo com a função objetivo que se quer otimizar. A ambivalência de emprego dos algoritmos de recursão e retrocesso tanto na busca exaustiva quanto incompleta é creditada por van Beek (2006) ao fato de operarem sobre uma única solução por vez, de forma que a busca não precisa ser finalizada para que alguma solução seja obtida – ao contrário do que ocorre no caso da programação dinâmica. É por essa razão que o estudo de algoritmos de busca por recursão e

³¹A programação dinâmica é um método de resolução de problemas pela decomposição em problemas menores, valendo-se do princípio da sub-estrutura ótima para que uma solução ótima ao problema seja obtida a partir da solução ótima dos sub-problemas associados. Para uma introdução ao tema, veja Cormen *et al.* (2002).

retrocesso para PSRs acaba por se confundir um pouco com o estudo da busca em PR. Entretanto, é importante evitar uma associação indissolúvel entre ambos os conceitos para que o emprego da técnica de PR não se torne demasiado restrito, conforme adverte [Hooker \(2007\)](#).

O restante da seção é dedicado à introdução dos principais elementos dos algoritmos de recursão e retrocesso e a uma breve apresentação dos métodos de busca local e estocástica.

2.3.1 Busca por Recursão e Retrocesso

A busca por recursão e retrocesso³² consiste no processo de atribuir progressivamente valores às variáveis que ainda não foram instanciadas. O algoritmo termina quando todas as possibilidades de atribuição foram tentadas ou quando alguma solução é encontrada durante o processo e isso bastar ao propósito da busca. O retrocesso se dá quando uma falha é detectada devido ao conjunto de atribuições acumulado ter violado alguma restrição do modelo, quando então ao menos uma das atribuições anteriores terá de ser desfeita e alterada. É possível que o retrocesso se transforme em uma reação em cadeia caso todos os valores possíveis já tenham sido atribuídos à variável anterior, levando a um novo retrocesso e assim por diante. Apesar de não ser necessário que seja implementada como uma recursão, essa forma de busca se assemelha à recursão em dois aspectos:

- as atribuições anteriores são mantidas em uma estrutura de pilha para reversão das mesmas no caso de retrocesso;
- o algoritmo consiste em uma recorrência sobre o passo da atribuição, de forma que a recursão emerge como uma forma evidente de implementação.

Nessas linhas gerais, vários algoritmos foram propostos visando aperfeiçoar aspectos específicos da estrutura de funcionamento descrita.

A despeito do algoritmo empregado, a extensão da busca por recursão e retrocesso para tratar de PSORs em vez de PSRs pode ser feita de diversas formas que independem das especificidades dos aperfeiçoamentos. Entretanto, a garantia de otimalidade em todos os casos depende de uma busca exaustiva. A hipótese mais simples consiste em registrar e manter atualizada a indicação de qual foi a solução com melhor valor de acordo com a função objetivo. Outra possibilidade consiste em reiniciar a busca após cada solução encontrada, restringindo o espaço de busca às soluções que sejam melhores do que a obtida. Nesse caso, o espaço de busca pode ser explorado pela busca no intervalo entre o valor da solução obtida e algum limite evidente ao ótimo, o que pode operar de três formas distintas segundo [van Beek \(2006\)](#):

- pela restrição de que a solução seja tão somente melhor do que a atual;
- pela restrição de que a solução não apenas seja melhor do que atual mas que sua diferença com relação ao limite ao ótimo esteja em uma determinada faixa;
- pela bissecção do intervalo de busca, explorando primeiramente a metade mais próxima ao limite ao ótimo e explorando a outra metade somente se a primeira definir um PSR inviável.

Dessas formas, [Lustig e Puget \(2001\)](#) observam que a última representa o de maior aceitação, sendo denominada busca dicotômica.

³²Denominada *backtracking search* em inglês.

A forma mais elementar de recursão e retrocesso consiste naquela com retorno à variável de atribuição imediatamente anterior no caso de conflito, sendo conhecido como retrocesso cronológico³³ ou BT³⁴. A respeito desse algoritmo mais elementar, [Barták \(1999\)](#) elenca três problemas comuns de desempenho decorrentes da ausência de ações preventivas ou reativas às falhas durante a busca:

- **Degeneração da busca**³⁵: o processo iterativo de instanciação das variáveis falha repetitivamente por um mesmo motivo, carecendo de uma estratégia de contenção da causa da falha.
- **Trabalho redundante**: algumas variáveis possuem valores conflitantes entre si, de forma que a memorização poderia evitar que fossem novamente atribuídos em conjunto durante o restante da busca.
- **Deteção tardia de conflito**: nenhum conflito de atribuição de variáveis é previsto com antecedência, sendo todos tratados à medida que ocorrem.

Além desses fatores, constatou-se que a adoção de uma estratégia de atribuição de valores mais adequada a cada caso tem grande impacto no desempenho do processo de busca. Disso resultou a proposição de um conjunto de aperfeiçoamentos na literatura, sendo que muitos deles podem ser utilizados para a composição de algoritmos de recursão e retrocesso mais elaborados. A seguir, serão mencionados alguns dos mais relevantes.

Estratégias de Ramificação

O progresso da busca recursiva costuma ser descrito como uma árvore enraizada³⁶ sobre a qual estratégias podem ser estabelecidas para antecipar a conclusão da busca. Nessa representação, cada decisão tomada durante a busca representa uma ramificação da árvore. Da perspectiva do modelo sendo resolvido, tais ramificações correspondem a partições do espaço de busca. Em última análise, o processo de resolução consiste na redução progressiva do espaço de busca pelo aumento da restritividade sobre o modelo ao ponto de restar apenas uma ou nenhuma solução. Tais possibilidades representam a distinção entre a satisfação do PSR e o retrocesso na busca, respectivamente. A respeito da forma com que esse processo de ramificação pode ser promovido, existem tipos diferentes de ramificação e heurísticas para decidir como ramificar, conforme apresentado a seguir. Na sequência, uma análise da importância dessas heurísticas e uma abordagem alternativa à busca por recursão e retrocesso baseada nas mesmas é indicada.

Tipos de ramificação. Três formas possíveis de ramificação para a resolução de PSRs são elencadas por [van Beek \(2006\)](#):

³³Denominado *chronological backtracking* em inglês.

³⁴Acrônimo em inglês surgido a partir do termo *backtracking*.

³⁵Denominada *thrashing* em inglês.

³⁶ Uma árvore enraizada consiste em uma árvore orientada em que apenas um vértice – a raiz – possui apenas arestas de saída, de forma que existe um caminho direcionado entre a raiz e cada vértice da árvore por meio das arestas orientadas, sendo os vértices finais em cada direção de ramificação denominados de folhas. Um vértice no final de um caminho é dito sucessor ou descendente dos outros vértices do mesmo caminho, que lhe são antecessores ou antecedentes.

- **Pontos de decisão binária ou divisão em 2**³⁷: representa a opção de atribuição de um determinado valor a uma das variáveis de decisão, de forma que o nível seguinte possui dois vértices: um em que esse valor é fixado à variável e outro em que a atribuição desse valor não é mais permitida à variável.
- **Divisão de domínios**³⁸: não representa necessariamente a atribuição a uma variável, mas antes a partição de seu domínio. Dessa forma, o conjunto de valores que podem ser atribuídos à variável nos dois vértices do nível seguinte não possui intersecção entre si.
- **Enumeração ou divisão em d** ³⁹: representa a atribuição de cada valor possível a uma das variáveis de decisão, de forma que cada vértice do nível seguinte representa o mesmo problema com um valor distinto fixado para tal variável.

Cabe ressaltar que as três modalidades representam uma sequência de generalizações que coincidem para o caso de domínios binários, visto que a divisão em d coincide com a divisão em 2 e que a divisão de um domínio binário invariavelmente induz a uma atribuição.

Heurísticas para ramificação. A escolha da sequência de opções que serão realizadas ao longo da busca representa a estratégia de ramificação. Essa escolha pode ser baseada em decisões locais a cada ramificação ou ser estabelecida de antemão, sendo esse o ponto de distinção entre as ordenações dinâmicas e estáticas das atribuições. Tais decisões são tomadas por meio de heurísticas sobre as propriedades do modelo e partem de premissas sobre a consequência de cada escolha possível mediante as condições observadas durante a busca. O objetivo mais comum a essas escolhas é de que façam com que a busca seja feita com o menor esforço possível. A esse respeito, [Refalo \(2004\)](#) destaca três princípios básicos que costumam orientar as ramificações:

1. Sempre escolher para atribuição as variáveis com maior potencial à redução do espaço de busca. Tal critério pode direcionar a escolha, por exemplo, para aquelas com menor tamanho de domínio ou maior participação nas restrições do modelo. Na literatura, esse princípio costuma ser identificado pela expressão “falhar primeiro”⁴⁰ devido ao fato de induzir à escolha de variáveis com maior potencial à ocorrência de falhas prematuras durante a busca. Essa opção objetiva evitar que a mesma falha ocorra com maior frequência, caso essa falha viesse a ocorrer em níveis posteriores da busca.
2. Sempre escolher para atribuição os valores que maximizem as possibilidades de ocorrência de outras atribuições, de forma que possuam uma maior probabilidade de presença nas soluções ao problema. Como pode ser observado, o princípio de escolha de valores é antagônico ao de escolha de variáveis.
3. Sempre privilegiar decisões mais próximas ao topo da árvore de busca, pois têm uma importância maior devido às consequências mais desastrosas que uma má decisão a esse ponto traria. A razão desse princípio é consonante ao raciocínio apresentado para endossar o princípio de “falhar primeiro”, mencionado anteriormente.

³⁷ Denominada *binary choice points* ou *2-way* em inglês.

³⁸ Denominada *domain splitting* em inglês.

³⁹ Denominada *enumeration* ou *d-way* em inglês.

⁴⁰ Denominado *fail-first principle* em inglês.

Refalo (2004) observa que um desenvolvimento maior foi conferido inicialmente aos dois primeiros princípios enunciados por meio de heurísticas para ordenar a preferência de escolha de variáveis e de valores para atribuição, como pode ser observado na revisão de Brailsford *et al.* (1999). No entanto, o terceiro princípio ganhou um reconhecimento maior ao longo da última década. Como exemplo disso, Refalo (2004) explora o emprego de uma heurística que demanda maior processamento na raiz da busca, Kautz *et al.* (2002) emprega estratégias dinâmicas de reinício da busca para alterar a ramificação próxima da raiz e em Arbelaez *et al.* (2009) é proposto um modelo preditivo de impacto das heurísticas correlacionando diversas características do modelo.

É importante não esquecer o caráter tentativo de qualquer estratégia de ramificação devido ao fato de que o problema de formulação da estratégia ótima de ramificação acaba por também ser NP-Difícil. Isso pode ser constatado de forma simples no caso da abordagem a um PSR satisfatível: uma estratégia perfeita de escolha de valores basta para que uma solução seja obtida sem que nenhum retrocesso ocorra. No caso de um PSR insatisfatível, a incumbência para uma estratégia ótima recai sobre a ordem de instanciação das variáveis, uma vez que tal escolha induz a uma exploração exaustiva menos trabalhosa se os conflitos ocorrerem mais próximos à raiz da busca. É interessante observar que o foco na literatura se concentra muito mais na ordenação das variáveis em razão da importância atribuída à antecipação dos conflitos na árvore de busca. Como um contraponto a essa ênfase, a seguir é apresentado um algoritmo alternativo de busca que se concentra exclusivamente na ordenação dos valores. Apesar de não ser exatamente um algoritmo baseado em retrocesso, optou-se por sua apresentação dentro da presente seção devido à similaridade entre ambos e à forma alternativa com que explora estratégias de ramificação.

Busca por discrepância limitada. A busca por discrepância limitada⁴¹ opera sobre instanciações completas das variáveis de decisão até que seja obtida uma solução ao problema, i.e., até que não haja nenhuma violação às restrições do PSR por parte das atribuições. Nesse algoritmo, parte-se de uma instanciação das variáveis com o valor de maior prioridade a cada variável, correspondente à ramificação mais à esquerda⁴² na árvore de busca – seja ela factível ou não. A cada iteração, um número progressivo de valores é alterado a partir da instanciação original como forma de obter uma solução – i.e., o nível 0 consiste no conjunto de atribuições inicial, o nível 1 aos conjuntos de atribuições que diferem em apenas uma variável do conjunto inicial e assim por diante. O intuito no projeto desse algoritmo é de que o espaço de busca seja explorado apenas sob a orientação de um conjunto referencial de valores de atribuição, assim evitando que uma má ordenação das variáveis degenera a busca. De acordo com Baptiste *et al.* (2001), a premissa ao sucesso dessa abordagem é de que as heurísticas empregadas falham em poucos pontos, a despeito de quais sejam esses pontos. Dessa forma, o sucesso de uma busca desse tipo acaba por depender muito mais da forma com que os valores são ordenados.

Técnicas Preventivas de Melhoria da Busca

O objetivo das técnicas preventivas⁴³ é de manter um nível de consistência do modelo ao longo da busca como forma de evitar a detecção tardia de conflito em determinados casos. Essa manutenção

⁴¹Denominada *Limited Discrepancy Search (LDS)* em inglês.

⁴²Por ramificação mais à esquerda em uma árvore de busca, ilustra-se graficamente o conceito do caminho obtido pela escolha da primeira opção em cada nível da busca.

⁴³Denominadas *look ahead schemas* em inglês.

costuma ser feita por técnicas de consistência em nível de aresta, sendo as três modalidades mais comuns as seguintes:

- **Verificação adiantada ou FC⁴⁴**. Na verificação adiantada, a manutenção da consistência em nível de aresta é feita sobre restrições binárias envolvendo uma variável instanciada e outra não instanciada. A propagação resultante dessa manutenção ocorre a cada nova atribuição e deve ser revertida se a atribuição for desfeita. Nessa modalidade, uma nova atribuição de valor a uma variável será sempre válida devido aos domínios estarem consistentes no corte entre os vértices das variáveis instanciadas e os vértices das variáveis não instanciadas. Dessa forma, o retrocesso não opera devido ao conflito imediato de uma atribuição à satisfação de uma restrição, mas sim pelo esvaziamento do domínio de uma variável que ainda não foi instanciada durante a propagação – razão pela qual haveria um conflito iminente para qualquer atribuição à variável de domínio esvaziado. Segundo [Barták \(1999\)](#), foi constatado que o desempenho decorrente da aplicação dessa técnica é superior ao do retrocesso cronológico.
- **Antecipação parcial⁴⁵**. A antecipação parcial consiste em um desdobramento à manutenção da consistência em nível de aresta não apenas para as arestas no corte tratado pela verificação adiantada, mas também em uma orientação do restante das arestas entre vértices de variáveis não instanciadas. Ela é dita parcial por operar apenas sobre uma orientação e não sobre todas as arestas do grafo do modelo, de forma que representa a consistência direcional em nível de aresta. Convém ressaltar que essa modalidade é elencada por [Barták \(1999\)](#) mas é ignorada por [van Beek \(2006\)](#), que considera apenas os outros dois casos. Entretanto, conforme mencionado anteriormente, [Dechter \(2003\)](#) aponta que a manutenção de consistência direcional ao nível da aresta surte o mesmo efeito que a não-direcional no caso em que a orientação condiz com a estratégia de ramificação da busca. Dessa forma, a antecipação parcial pode substituir com vantagem de desempenho a antecipação total em algumas situações, como no caso de uma ordenação estática para ramificação do restante do espaço de busca.
- **Antecipação total, manutenção da consistência em nível de aresta ou MAC⁴⁶**. Como o próprio nome alternativo anseia, a antecipação total consiste na manutenção da consistência ao nível da aresta a todo o grafo do modelo, sem restrição de orientação ou extensão. [Barták \(1999\)](#) observa que o esforço necessário à MAC pode tornar seu desempenho pior que o da verificação antecipada ou do retrocesso cronológico em alguns casos. Por outro lado, [van Beek \(2006\)](#) considera que a controvérsia a respeito da questão de desempenho nesse caso foi superada por novos estudos empíricos e que a antecipação total representa uma opção superior às demais no caso geral.

Técnicas Reativas de Melhoria da Busca

O objetivo das técnicas reativas⁴⁷ é utilizar informações sobre conflitos que ocorreram durante a busca para evitar sua repetição ou prover uma contingência mais adequada à sua ocorrência. Sua necessidade se deve à impossibilidade de se atingir níveis completos de consistência, de forma

⁴⁴ Denominada *Forward Checking (FC)* em inglês.

⁴⁵ Denominada *partial look ahead* em inglês.

⁴⁶ Denominada *(full) look ahead* ou *Maintaining Arc Consistency (MAC)* em inglês.

⁴⁷ Denominadas *look back schemas* em inglês.

que a detecção tardia de conflito não pode ser completamente erradicada. Assim sendo, o foco dessas técnicas é evitar o trabalho redundante e, por consequência, a degeneração da busca. Entretanto, convém sempre avaliar se o esforço exigido para a execução de procedimentos de verificação adicionais não será maior do que o perda em desempenho cuja ocorrência se procura evitar.

A seguir, listamos algumas das técnicas reativas mais estudadas na literatura, conforme [Dechter \(2003\)](#) e [van Beek \(2006\)](#):

- **Aprendizado de restrições**⁴⁸. O aprendizado de restrições consiste na acumulação de novas restrições ao modelo original em decorrência dos conflitos detectados durante a busca. Cada nova restrição é obtida por meio da identificação de um conjunto de conflito, que representa o conjunto de variáveis cujas atribuições justificam a violação de alguma restrição. Por sua vez, as atribuições que conferem a violação de uma restrição a esse conjunto de conflito representam uma rejeição⁴⁹, a partir da qual define-se uma nova restrição que proíba exatamente esse conjunto de atribuições. Apesar de redundante por estar implícita no conjunto de restrições original do modelo, essa nova restrição possibilita uma propagação mais eficiente por evidenciar um conflito que já ocorreu anteriormente, de forma que interessa determinar o menor conjunto de conflito possível para que uma quantidade maior de conflitos futuros seja evitada. O maior inconveniente a essa técnica – e que faz com que não seja utilizada na prática – é que ela pode levar à inclusão de uma quantidade exagerada de restrições, de forma que exige a definição de um critério à inclusão de novas restrições para privilegiar as mais generalistas e outro critério à remoção de restrições para não continuar propagando aquelas cujos conflitos evitados já não possam mais ocorrer.
- **Retrocesso inteligente**⁵⁰. O retrocesso inteligente promove um retrocesso não-cronológico – i.e., não necessariamente à última variável atribuída – pela análise do conjunto de conflito. Dito de outra forma, ele salta variáveis da pilha cuja atribuição não tenha relação com a impossibilidade de progresso da busca a partir daquele ponto. Sendo a alteração da atribuição a alguma variável do conjunto de conflito condição necessária – mas não suficiente – para que alguma solução seja obtida, um salto que ignore apenas variáveis ausentes do conjunto não impede a completude da busca. Portanto, evita-se o trabalho redundante na busca pela exaustão de possibilidades de atribuição que seriam exploradas sem serem capazes de evitar a reincidência da falha observada. Uma sequência de algoritmos contemplando uma gama cada vez mais ampla de saltos é descrita na literatura, dos quais dois têm maior destaque: o algoritmo de retrocesso por salto⁵¹ ou de Gaschnig, que retrocede não cronologicamente apenas de vértices da árvore de busca cujos descendentes sejam folhas – i.e., nenhuma atribuição ocorreu sem conflito a partir daquele ponto; e o algoritmo de retrocesso baseado em conflito⁵², que pode retroceder a partir de outros vértices da árvore de busca devido ao acúmulo do conjunto de conflito dos sucessores dos quais retornou.
- **Retrocesso dinâmico**⁵³. O retrocesso dinâmico consiste na permuta da ordem das variáveis

⁴⁸Denominado *backmarking* em inglês.

⁴⁹Denominada *nogood* em inglês.

⁵⁰Denominado *intelligent backtracking* ou *backjumping* em inglês.

⁵¹Denominado *BackJumping (BJ)* em inglês.

⁵²Denominado *Conflict-Based BackJumping (CBJ)* em inglês.

⁵³Denominado *dynamic backtracking* em inglês.

da pilha de atribuições, de forma a aproximar a posição das atribuições de variáveis conflitantes. Dessa forma, o retrocesso dinâmico representa uma alternativa ao retrocesso inteligente por reduzir o trabalho redundante ao diminuir o número de atribuições a variáveis em ordem intermediária com relação às conflitantes. No entanto, a livre permuta das variáveis na ordem de atribuição pode comprometer a sistemática da busca – i.e., torná-la cíclica ou incompleta. Portanto, esse subterfúgio tem maior utilidade no início da busca ou se incorporada às estratégias de ramificação.

2.3.2 Busca Local e Estocástica

A busca local e a busca estocástica procuram explorar o espaço de busca de forma não sistemática, podendo valer-se dos resultados intermediários para guiar o restante do processo de resolução. No caso da busca estocástica, a justificativa a um processo não sistemático se dá pelo foco à exploração de regiões com maior potencial em detrimento à completude. No entanto, é possível conferir essa última característica a algumas das abordagens desse tipo. Como exemplo disso, temos as estratégias de reinício como as descritas em [Gomes *et al.* \(1998\)](#) e [Kautz *et al.* \(2002\)](#), que interrompem a busca para reiniciá-la com alterações às estratégias de ramificação. Observa-se de [Gomes *et al.* \(1998\)](#) que é possível manter a completude da busca empregando estratégias de reinício se o número mínimo de retrocessos à interrupção da busca aumentar ao longo da resolução, de forma que em algum momento a quantidade mínima ultrapasse o número de retrocessos necessários à realização de uma busca completa. Entretanto, tal peculiaridade interessa menos como uma característica desse tipo de abordagem à busca e mais como uma garantia de que o fracasso da estratégia de reinício não prejudique a busca por sempre limitá-la apenas a algumas regiões. No caso da busca local, a exploração do espaço de busca é limitada a aperfeiçoar resultados obtidos por outros meios sob o argumento de que a completude não pode ser cogitada devido ao tamanho do espaço de busca. Ela opera por meio de transformações de soluções obtidas anteriormente na resolução, seja para minimizar a penalização decorrente de uma solução que não atenda a todas as restrições ou para obter uma melhor aproximação à solução ótima a partir de soluções que já sejam factíveis. Dessa forma, existe uma certa similaridade e intersecção entre ambas as modalidades, mas convém analisá-las separadamente devido ao escopo diferenciado de ambas: na busca estocástica ainda é possível operar sobre a mesma estrutura que a de recursão e retrocesso, enquanto que a busca local opera sobre estruturas alternativas.

A estruturação da busca local costuma priorizar a obtenção de soluções a partir de outras pré-existentes. Essa geração de soluções pode priorizar a qualidade – seja por um melhor atendimento às restrições ou uma melhor valoração da função objetivo – ou a variabilidade – como no caso em que os critérios quantitativos de comparação das soluções não são tidos como suficientes e certo grau de liberdade é conferido à intervenção humana pela escolha dentre um conjunto de soluções distintas. Por questão de foco ao tipo de abordagem realizado no estudo de caso do presente trabalho, serão exemplificadas a seguir apenas estruturas de priorização à qualidade. [Dechter \(2003\)](#) ilustra o caso de melhor atendimento às restrições pelo emprego de métodos gulosos de busca local, de forma que a resolução é decomposta entre uma atribuição completa de valores despreocupada com o atendimento às restrições e a operação de reparos locais a tais atribuições como forma de reduzir a violação decorrente. É possível observar certa similaridade entre essa estrutura e a busca por discrepância limitada, com ressalva à ausência de sistematização para completude da busca nesse caso. Focando a

otimalidade em vez da satisfabilidade, é estudada uma decomposição distinta em Laborie e Godard (2007), dividida entre operadores que geram novas soluções por meio de modificações em soluções factíveis e operadores que ajustam as soluções geradas para que satisfaçam às restrições no caso de alguma violação ter sido causada. Duas referências são recomendadas para o aprofundamento na aplicação de busca local em PR, tendo enfoques distintos: Dechter (2003) parte da PR para discutir o emprego de busca local, ao passo que Hoos e Stützle (2005) partem da busca local estocástica para discutir sua aplicação em PR.

2.4 Paradigmas

O desenvolvimento da PR ao longo dos anos ocorreu sob uma sucessão de paradigmas de sintaxe dos resolvedores cuja contextualização faz-se importante a uma melhor compreensão de como a técnica foi sendo aperfeiçoada. O principal deles consiste na Programação Lógica por Restrições (PLR)⁵⁴. De acordo com Barták (1999); Jaffar e Lassez (1987), a PR partiu da extensão de linguagens lógicas⁵⁵ como Prolog para uma resolução mais eficiente de PSRs. A extensão dessas linguagens para a resolução de PSRs se deu principalmente pela inclusão de mecanismos de inferência sobre as restrições, de forma a permitir cortes⁵⁶ mais eficazes durante a busca. No entanto, elas conferiram aos programas resultantes um caráter distinto devido à manipulação explícita dos predicados, remetendo ao paradigma de programação funcional⁵⁷. Nesse contexto de impasse, Jaffar e Lassez (1987) propuseram a definição da classe de linguagens PLR como uma unificação dos paradigmas declarativos funcional e lógico capaz de descrever as extensões sendo então promovidas na literatura. A classe PLR representa linguagens direcionadas à resolução por inferência no domínio de discurso dos problemas, em contraposição à tradução dos mesmos a uma estrutura unificada de inferência lógica, dessa forma conservando o caráter de programação declarativa. Por essa razão, a PLR delineou uma estrutura prática ao desenvolvimento de mecanismos de propagação de resolvedores de problemas combinatórios, provendo suporte ao emprego das técnicas pré-existentes e facilitando a emergência de aperfeiçoamentos às mesmas.

No entanto, a PLR não bastou à PR devido à limitação do papel assumido pelo usuário final do resolvidor nessas linguagens. Isso porque a dinâmica estabelecida impedia que manipulações especializadas a domínios mais restritos se viabilizassem. Essas manipulações foram facilitadas a partir dos anos 1990 pelo emprego de linguagens imperativas como C++, nas quais a estrutura de resolução é oferecida por meio de bibliotecas de código que podem ser especializadas na definição de como se operam tanto a propagação quanto a busca. Entretanto, Lustig e Puget (2001) notam que a PR ainda permanecia como uma técnica bastante restrita a essa época devido à exigência de conhecimentos de programação de computadores para seu emprego. Nesse contexto, linguagens de modelagem algébrica como as empregadas para programação matemática surgiram ao final da

⁵⁴ Denominada *Constraint Logic Programming (CLP)* em inglês.

⁵⁵ No paradigma de programação lógica, programas são definidos como um conjunto de expressões sobre variáveis ou termos constantes, designados predicados devido a seu resultado se resumir às condições de verdadeiro ou falso. O objetivo da execução de um programa lógico consiste na verificação de que um predicado possa ser satisfeito mediante uma atribuição de valores constantes a suas variáveis, sendo essa atribuição denominada unificação. Para maiores detalhes, veja Scott (1999).

⁵⁶ No contexto da programação lógica, um corte representa um ponto em que se assume a falha à satisfação de um determinado predicado na hipótese de retrocesso durante a busca.

⁵⁷ No paradigma de programação funcional, a declaração dos programas é essencialmente recursiva e desprovida de atribuições explícitas, evidenciando a forma com que a resolução deve operar.

década de 1990 e foram se transformando ao longo da década seguinte. Ainda que essas linguagens já não correspondam à estrutura dos mecanismos de resolução subjacentes, existe uma relação clara entre a expectativa de como o resolvidor será utilizado e a disposição da sintaxe de modelagem. Dessa forma, a compreensão da forma com que a PR tem se desenvolvido desde então passa obrigatoriamente pela análise tanto das contribuições à PR na literatura que foram adotadas nos resolvidores quanto da sintaxe com que os problemas passaram a ser modelados.

Como subsídio a uma introdução às tendências mais recentes observadas, a seção 2.4.1 serve à análise de duas das ferramentas comerciais baseadas em PR mais empregadas, e a seção 2.4.2 aponta os esforços recentes voltados ao desenvolvimento de linguagens abstratas – i.e., voltadas à PR mas independentes de resolvidores.

2.4.1 Ferramentas Comerciais

Duas linguagens – e seus respectivos resolvidores – bastante empregadas atualmente por praticantes da PR são a OPL⁵⁸ e a Comet. A análise que se segue de cada uma delas é baseada nos manuais da IBM (2010) e Dynadec (2009), respectivas proprietárias das mesmas, além de outras menções observadas sobre seus desenvolvimentos na literatura.

OPL

De acordo com Lustig e Puget (2001), a OPL foi uma das primeiras linguagens algébricas com suporte à PR. Ela oferece uma sintaxe declarativa para expressões matemáticas e algumas restrições especializadas a certos domínios de aplicação, como no caso de problemas de escalonamento. Como pode ser observado pelos exemplos descritos em Hentenryck *et al.* (2000) e em Lustig e Puget (2001), as versões iniciais permitiam a definição separada do modelo e da busca – a última em termos de estratégias não necessariamente completas de ramificação. No entanto, versões mais recentes sofreram uma redução do escopo em que a busca pode ser definida. Em parte, é possível que isso se deva ao argumento de Puget (2004) de que a simplicidade de uso era necessária à ampliação do público-alvo da PR, defendida por ele como sendo “qualquer engenheiro”. No mesmo evento, Refalo (2004) apresentou uma técnica de ramificação mais elaborada do que as heurísticas em uso como alternativa à delegação de definição da busca ao usuário, alegando inspiração na programação inteira. Um esforço similar foi observado com relação à aplicação em problemas de escalonamento em todas as dimensões possíveis:

- novas abstrações de modelagem em Laborie e Rogerie (2008) e Laborie *et al.* (2009);
- exploração de níveis de propagação mais efetivos e de algoritmos de filtragem mais eficientes em Laborie (2003a), entre outros;
- algoritmos de busca dedicados a problemas de escalonamento em Godard e Laborie (2005) e Laborie e Godard (2007).

Dessa forma, a sintaxe da OPL foi se transformando em direção à automatização dos mecanismos de resolução em detrimento à manipulação da definição da busca. Isso fez com que a definição da busca ficasse restrita à configuração de parâmetros do conjunto de técnicas empregadas.

⁵⁸ Acrônimo em inglês para *Optimization Programming Language*, ou Linguagem de Programação para Otimização.

Comet

De acordo com [Hentenryck \(2010\)](#), que foi um dos autores da OPL, a Comet teve sua primeira versão apresentada três anos após a introdução da OPL. A intenção da linguagem era permitir uma manipulação explícita de estratégias de busca local em PR. A princípio, sua estrutura era muito parecida com aquela descrita anteriormente para a OPL em [Hentenryck et al. \(2000\)](#). No entanto, a linguagem oferece muitos mecanismos de diferentes paradigmas de programação, tais como fechos⁵⁹ e métodos acionados por eventos⁶⁰ em forma propícia à definição de algoritmos de busca local. No que diz respeito ao domínio de problemas de escalonamento, a estrutura de modelagem mantém o emprego de abstrações tradicionais como atividades e recursos por meio de uma gama de possibilidades de restrições entre tais abstrações. No entanto, os algoritmos de resolução subjacentes à modelagem não diferem tanto entre a Comet e a OPL. Assim sendo, a maior diferença nos emprego dessas linguagens e de seus respectivos mecanismos de resolução é que a definição da busca é explícita na primeira e implícita na segunda. Entretanto, algumas publicações recentes tiveram seu foco voltado à abstração da busca na Comet, como a proposta de um sintetizador de algoritmos de busca baseado na estrutura do problema para a resolução de PSRs em [Hentenryck e Michel \(2007\)](#) e de problemas de escalonamento especificamente em [Monette et al. \(2009\)](#). A justificativa a ambos os casos é atribuída a [Gomes e Selman \(2007\)](#), sendo similar àquela apresentada por [Puget \(2004\)](#): a necessidade de ampliar a audiência sendo atendida pela abstração da busca. No caso de [Gomes e Selman \(2007\)](#), alega-se a ausência do que se convencionou chamar de resolvedores “caixa-preta”, nos quais a modelagem independe de conhecimentos sobre como a busca opera – algo que deve ser tratado com cautela para obter o melhor desempenho possível, conforme defendido em [Serra e Wakabayashi \(2010\)](#).

2.4.2 Modelagem Abstrata

Uma área de trabalho recente em PR diz respeito às linguagens de modelagem abstratas, por meio das quais um determinado tipo de problema possa ser modelado a despeito do resolvidor que será empregado. Esse tipo de linguagem surge com maior facilidade na programação matemática devido à relativa uniformidade com que os problemas são modelados. No caso da PR, os obstáculos são maiores devido à falta de padronização sobre como a modelagem deve ser feita. A MiniZinc, originalmente proposta por [Nethercote et al. \(2007\)](#), representou a primeira empreita desse tipo. Nela, os projetistas dos resolvedores são incumbidos de prover uma tradução de baixo nível entre as estruturas de modelagem disponíveis de um lado e a sintaxe de seu resolvidor do outro. No entanto, os autores reconhecem que há uma diferença considerável de desempenho contra a modelagem na linguagem nativa do resolvidor, de forma que é necessário que seja dada uma atenção maior às restrições globais e à definição da busca. Em meados de 2011, ocorreram testes controlados de uma versão experimental da linguagem comercial de modelagem abstrata AIMMS capaz de utilizar resolvedores de PR, conforme divulgado em [van Hoeve et al. \(2011\)](#), sendo possível que as versões oficiais da linguagem contemplem essa possibilidade nas próximas versões.

⁵⁹Mecanismo de linguagens funcionais representando um corpo de código que pode ser manipulado e executado.

⁶⁰Métodos de programação imperativa que podem ser acionados caso uma determinada condição monitorada ocorra.

Capítulo 3

Escalonamento Baseado em Restrições

O Escalonamento Baseado em Restrições (EBR)¹ representa uma extensão da PR à abordagem de problemas de escalonamento. Essa extensão consiste na especialização dos elementos de modelagem, propagação e busca à resolução de problemas envolvendo a alocação temporal de atividades mediante as limitações dos recursos à disposição. Notadamente, o desenvolvimento observado nessa linha de aplicação destacou-se das demais em PR e foi alavancado por pesquisa tanto na academia quanto na indústria. O interesse acadêmico se deu pela integração e incentivo a frentes bastante diversas de pesquisa em um ferramental de resolução unificado. Na indústria, identificou-se a oportunidade de aplicação a problemas reais graças à forma compacta com que modelos de EBR descrevem certos problemas e aos resultados observados. Dessa forma, o EBR representa um caso de particular interesse da PR devido ao incentivo que representa à aplicação da técnica e ao benefício advindo dessa aplicação no caso de problemas de escalonamento.

O propósito deste capítulo é analisar a especialização dos elementos constituintes da PR à abordagem de problemas de escalonamento de diversos tipos. Para tanto, convém apresentar de antemão as razões pelas quais a articulação entre esses elementos encontrou um campo mais fértil nessa área do que em outras.

De acordo com Baptiste *et al.* (2006), o sucesso na abordagem de problemas de escalonamento por meio da PR se deu pela aderência de técnicas pré-existentes à componentização de acordo com o tripé modelo-propagação-busca, descrito no Capítulo 2. A modelagem por meio de variáveis e restrições especializadas possibilitou uma descrição natural e flexível de problemas de escalonamento. A propagação por meio de restrições globais aproveitou-se de uma sinergia entre a generalidade de constituição dos problemas abordados na inteligência artificial e a eficiência de algoritmos de filtragem especializados da pesquisa operacional. A descrição das heurísticas mais comuns de escalonamento adequou-se à estrutura de árvore de busca empregada na PR. Além disso, a natureza declarativa da modelagem possibilitou o emprego de métodos alternativos de busca, conforme mencionado na introdução do Capítulo 2 e exemplificado na seção 2.3.2. Dessa forma, a EBR representou antes uma compilação de ferramental disperso em uma organização propícia do que uma constituição completamente inédita e independente das abordagens alternativas à resolução de problemas desse tipo.

A organização do restante do capítulo objetiva uma descrição do EBR por meio de suas especificidades com relação à PR. Portanto, a estrutura do capítulo é mais compacta mas bastante similar

¹ Denominado *Constraint-Based Scheduling* (CBS) em inglês.

àquela empregada no capítulo anterior. A seção 3.1 trata dos paradigmas e da modelagem para a definição dos problemas no EBR e a seção 3.2 da propagação e da busca para a resolução desses problemas.

3.1 Modelagem e Paradigmas Descritivos

Apesar da modelagem dos problemas de escalonamento variar em função da tipologia de cada problema, existem alguns elementos comuns a todos eles. Os PSRs específicos ao EBR se baseiam nos conjuntos $A = \{a_1, \dots, a_n\}$ de atividades e $R = \{r_1, \dots, r_m\}$ de recursos necessários às atividades. Sobre ambos os conjuntos, propriedades podem ser definidas como variáveis ou restrições do problema. Essas propriedades podem dizer respeito à alocação temporal das atividades, à sua demanda pela disponibilidade dos recursos, às restrições temporais entre atividades ou à limitação da disponibilidade dos recursos ao longo do tempo. No caso de POSRs, a constituição da função objetivo também é feita de variáveis de decisão advindas dessas propriedades. O que varia com as diferentes tipologias de problema é a natureza, o valor ou a manipulação realizada das propriedades que caracterizam os elementos dos conjuntos A e R , bem como sua inter-relação.

Convém, contudo, observar que a especialização da modelagem no EBR acaba por depender do paradigma subjacente em um grau que impede uma análise separada desses elementos. Dessa forma, o detalhamento da modelagem se dará pela descrição da forma clássica de representação de EBRs em função da manipulação explícita de entidades representando atividades e recursos na seção 3.1.1 e por uma caracterização alternativa através de variáveis de intervalo que representam indiretamente os conceitos de atividade e recurso na seção 3.1.2. Além disso, é importante observar que a notação adotada a seguir representa uma síntese dos conceitos por parte do autor e não guarda correspondência exata com aquela encontrada na literatura. Entretanto, no caso da seção 3.1.2, há uma grande correspondência – ainda que não de todo precisa – com a notação da linguagem OPL devido ao fato da origem dessa modalidade estar associada ao próprio desenvolvimento da linguagem.

3.1.1 Caracterização por Atividades e Recursos

A caracterização de PSRs do EBR por meio de relações explícitas entre as propriedades de atividades e recursos permitiu que os mecanismos de resolução se especializassem com um benefício tangível de desempenho. Ela é adotada por referências centrais na literatura como Baptiste *et al.* (2001) e Baptiste *et al.* (2006). Além disso, é empregada de modo direto na abordagem a problemas de escalonamento utilizando a linguagem Comet, de acordo com o manual da Dynadec (2009). E ainda que não seja explicitamente adotada na linguagem OPL conforme o manual da IBM (2010), as representações empregadas encerram correspondência suficiente para que os mesmos mecanismos de propagação sejam utilizados nesse caso. Dessa forma, a caracterização por atividades e recursos persiste como dominante na literatura e a caracterização por variáveis de intervalo – tratada na seção 3.1.2 – presta-se como uma representação mais generalista mas ainda capaz de se beneficiar dos desenvolvimentos voltados à estrutura descrita na presente seção.

Caracterização do Conjunto de Atividades

Visto que a modelagem dos recursos interessa somente à medida em que limitam a execução das atividades, estas constituem o principal objeto de interesse no EBR. Cada atividade $a_i \in A$ pode ter como propriedades a data de liberação² dl_i a partir da qual pode ser executada, a data de limite de conclusão³ dc_i em que deve ter sido concluída e o tempo de processamento a plena capacidade tp_i que determina o esforço necessário à sua realização. Quanto à tipologia das atividades, é possível caracterizar um problema de escalonamento em níveis crescentes de generalização como não-preemptivo, preemptivo ou elástico em função da natureza de suas atividades e de sua relação com as propriedades descritas:

- **Atividade não-preemptiva** Uma atividade não-preemptiva a_i deve ser realizada de maneira ininterrupta a partir de seu início. Dessa forma, ela pode ser caracterizada pelo par de variáveis discretas $inicio(a_i)$ e $fim(a_i)$ representando, respectivamente, os instantes de início e fim de a_i tais que:

$$inicio(a_i) \geq dl_i \quad (3.1)$$

$$fim(a_i) \leq dc_i \quad (3.2)$$

$$fim(a_i) - inicio(a_i) = tp_i \quad (3.3)$$

- **Atividade preemptiva** Uma atividade preemptiva a_i pode ser realizada de maneira intermitente. Dessa forma, é possível caracterizá-la pelo conjunto $IF(a_i) = \{\{inicio_j(a_i); fim_j(a_i)\} : j \in [1, rodadas(a_i)]\}$ representando os pares de início e fim das rodadas necessárias à execução de a_i tal que:

$$inicio_j(a_i) < fim_j(a_i), j \in [1, rodadas(a_i)] \quad (3.4)$$

$$fim_{j-1}(a_i) \leq inicio_j(a_i), j \in [1, rodadas(a_i) - 1] \quad (3.5)$$

$$inicio_1(a_i) \geq dl_i \quad (3.6)$$

$$fim_{rodadas(a_i)}(a_i) \leq dc_i \quad (3.7)$$

$$\sum_{j=1}^{rodadas(a_i)} (fim_j(a_i) - inicio_j(a_i)) = tp_i \quad (3.8)$$

- **Atividade elástica** Uma atividade elástica a_i pode ser realizada a níveis variáveis de intensidade, de forma que o tempo de execução difere do tempo de processamento em razão dessa variabilidade. É possível caracterizá-la pelo conjunto $IFP(a_i) = \{\{inicio_j(a_i); fim_j(a_i); produtividade_j(a_i)\} : j \in [1, rodadas(a_i)]\}$ representando os trios de início, fim e produtividade das rodadas necessárias à execução de a_i . Ele difere da caracterização de $IF(a_i)$ apenas

²Denominada *release date* em inglês.

³Denominada *deadline* em inglês.

na última relação:

$$\sum_{j=1}^{rodadas(a_i)} (fim_j(a_i) - inicio_j(a_i)) * produtividade_j(a_i) = tp_i \quad (3.9)$$

Convém observar que cada variável $produtividade_j(a_i)$ representa uma fração no intervalo real $(0, 1]$, algo que pode parecer contraditório com a manipulação de domínios discretos e finitos em PSRs. Contudo, os valores atribuídos a cada uma dessas variáveis depende da forma com que cada rodada expropria níveis discretos da capacidade dos recursos empregados em sua execução – conforme descrito adiante. Dessa forma, os níveis de produtividade de cada rodada não variam arbitrariamente, mas em partições discretas da capacidade dos recursos dos quais dependem. Em última análise, essas variáveis não representam decisões em si, mas sim propriedades advindas das decisões discretas que determinam cada solução possível.

Além dessa tipologia de atividades, em alguns casos é necessário ponderar pela ocorrência ou não de cada atividade a_i tida como opcional por meio de uma variável booleana adicional $ocorrencia(a_i)$. E tanto esta quanto as outras propriedades das atividades podem ser empregadas para a definição de restrições entre entidades distintas. Portanto, o nível de generalização necessário a duas características das atividades do conjunto A leva a seis possibilidades de classificação para cada problema de escalonamento – interessando-nos classificar cada um no nível mais específico possível para o benefício de uma resolução mais eficiente.

Caracterização do Conjunto de Recursos

Conforme observado anteriormente, a alocação de uma atividade a um recurso interessa à modelagem na medida em que reduz a possibilidade de execução de outras atividades. Essa alocação tanto pode ser definida *a priori* quanto decidida a partir de um conjunto de recursos disponíveis em que cada um possa satisfazer à necessidade da atividade, sendo possível que uma atividade dependa de diferentes recursos por diferentes critérios. As limitações dos recursos dizem respeito aos seus estados e capacidades.

Estado dos recursos A cada recurso r_j podem ser atribuídos diversos tipos de estado. Sendo e_k um desses tipos e $\bar{E}_k$ o conjunto de estados possíveis a esse tipo, é possível expressar a necessidade de que uma atividade a_i dependa do estado $\bar{e}_k \in \bar{E}_k$ atribuído ao tipo de estado e_k do recurso r_j durante sua execução por meio da restrição $e_k(r_j, a_i) = \bar{e}_k$. Por meio do estabelecimento dessa restrição, evita-se que um mesmo recurso seja empregado simultaneamente por atividades que exijam estados diferentes de um mesmo tipo – o que torna seus períodos de execução mutuamente excludentes. No caso em que mais de um recurso possa satisfazer a exigência de uma determinada atividade, a restrição pode ser generalizada como sendo $e_k(R_i, a_i) = \bar{e}_k$ para um conjunto $R_i \subseteq R$ de recursos que possam satisfazer à demanda da atividade a_i . Nesse caso, ao menos um recurso $r_j \in R_i$ deve satisfazer a restrição $e_k(r_j, a_i) = \bar{e}_k$ para que a restrição $e_k(R_i, a_i) = \bar{e}_k$ seja satisfeita. Além da exclusividade temporal de diferentes estados limitarem o uso do um recurso, a transição entre esses estados pode tomar tempo e incorrer em custos à solução. Para representar essa situação, à declaração de um tipo de estado e_k pode estar associada a uma função de tempo de transição $t : \bar{E}_k \times \bar{E}_k \rightarrow \mathbb{N}$ que mapeia cada transição possível entre estados do tipo e_k . De forma similar e para

efeito na função objetivo, é possível definir uma função de custo de transição $c : \overline{E}_k \times \overline{E}_k \rightarrow \mathbb{R}$. De certa forma, no entanto, o mapeamento e a consequente restrição de estados possibilita um controle no modelo que acaba por ser mais qualitativo do que quantitativo por não impedir a concorrência de uma quantidade arbitrária de atividades com mesma configuração de estados no recurso.

Capacidade dos recursos Para um controle quantitativo mais apurado da concorrência nos recursos, a modelagem deve focar a capacidade em vez do estado dos mesmos. A respeito da capacidade, os recursos podem ser definidos gradativamente como disjuntivos, cumulativos ou reservatórios em razão do limite de concorrência de atividades:

- **Recurso disjuntivo** Um recurso disjuntivo pode ser usado por apenas uma atividade por vez. Para um recurso r_j e uma atividade a_i , a restrição $capacidade(r_j) = 1$ define essa limitação unitária de concorrência no recurso; e a restrição $demanda(r_j, a_i) = 1$ ou $demanda(R_i, a_i) = 1$ que a atividade a_i exige o emprego em caráter temporal exclusivo do recurso r_j ou algum recurso de R_i , respectivamente.
- **Recurso cumulativo** Um recurso cumulativo pode ter mais de uma atividade sendo realizada ao mesmo tempo, sendo que diferentes atividades podem exigir diferentes níveis de capacidade do recurso. Nesse caso, o controle de concorrência em um recurso r_j é generalizado pela restrição $capacidade(r_j) = k_j$ para um valor de capacidade $k_j \geq 1$. O mesmo se opera no caso da atividade: a restrição $demanda(r_j, a_i) = k_{ij}$ define um valor de capacidade $k_{ij} \geq 1$ requerida à execução da atividade a_i no recurso r_j . Dessa forma, a concorrência de atividades no recurso é limitada pelo nível de capacidade requerido pelas mesmas contra o nível máximo possível ao recurso.
- **Recurso reservatório** Um recurso reservatório representa uma generalização do conceito de capacidade pela manipulação de um montante que varia em função de eventos das atividades associadas ao recurso. Em vez de uma capacidade fixa do recurso r_j e uma demanda simultânea à execução de uma atividade a_i , define-se a capacidade como uma função $capacidade(r_j) : T \rightarrow \mathbb{N}$ sobre o domínio temporal T . Essa função varia a partir de um valor de capacidade inicial $capacidade_0(r_j)$ em decorrência de acréscimos ou decréscimos de unidades de capacidade pelas restrições $producao(r_j, t) = k_i$ e $consumo(r_j, t) = k_i$, respectivamente, para uma quantia de k_i unidades de variação e $t = inicio(a_i)$ ou $t = fim(a_i)$ para alguma atividade a_i . A partir dessa formalização, o valor de $capacidade(r_j)$ pode ser restrito a um determinado valor em um dado instante ou em todo o seu domínio. Por exemplo, podemos definir a restrição $capacidade(r_j) \leq M_j$ para limitar superiormente a capacidade do recurso r_j em M_j unidades a todo instante.

Com relação à forma de execução, a contemplação de atividades elásticas pode levar ao uso de quantidades fracionárias da capacidade dos recursos. Entretanto, essa possibilidade não será explorada explicitamente devido ao propósito meramente introdutório do texto.

Limites à Caracterização por Atividades e Recursos

Uma das dificuldades do emprego dessa notação explícita envolvendo atividade e recursos decorre da combinação de possibilidades descritas anteriormente. Por exemplo, um recurso pode ter

uma relação de estado e capacidade com uma atividade, que por sua vez pode ou não ter sua execução dividida em rodadas. Isso se torna bem claro no caso do manual da [Dynadec \(2009\)](#): contemplar todas as combinações possíveis leva a uma listagem bastante extensa de possibilidades. Em decorrência desse tipo de limitação que dificulta um emprego elegante dessas relações, a seguir discutimos a representação por variáveis de intervalo.

3.1.2 Caracterização por Variáveis de Intervalo

A representação de modelos de EBR por variáveis de intervalo foi introduzida por [Laborie e Rogerie \(2008\)](#) como uma alternativa de modelagem que privilegia a representação de atividades opcionais, mesmo em problemas em que elas não ocorreriam explicitamente. Essa representação parte do trabalho iniciado por [Laborie \(2003b\)](#), que estendeu problemas envolvendo apenas a relação entre variáveis temporais tratada por [Vilain *et al.* \(1990\)](#) para a inclusão de peculiaridades de recursos nos modelos. Em muitos aspectos, o intervalo corresponde em suas propriedades à noção previamente definida de atividade. Entretanto, ele difere na forma de manipulação ao ponto de também poder caracterizar recursos em alguns casos. Isso se explica pelo estímulo ao emprego corriqueiro de intervalos opcionais por ser possível representar explicitamente cada combinação de alocação de uma atividade a um recurso por uma variável distinta, de forma que as especificidades de cada combinação de alocação possam ser tratadas separadamente. E naqueles pontos em que tal correspondência entre intervalos e atividades ou recursos não seja possível, elementos adicionais foram propostos por [Laborie *et al.* \(2009\)](#) para complementação. O benefício alegado para essa abordagem consiste na facilidade de modelar uma gama maior de problemas de escalonamento e na consequente possibilidade de realizar uma propagação mais eficiente sobre as restrições estabelecidas. Dessa forma, a representação por variáveis intervalares se apresenta como uma alternativa de modelagem tão desvencilhada quanto possível da caracterização convencional por atividades e recursos sem, contudo, deixar de se beneficiar dos mecanismos de resolução pré-existentes.

Propriedades das Variáveis de Intervalo

Para uma introdução aos pontos mais relevantes da formulação por variáveis de intervalo, consideremos o conjunto $V = \{v_1, \dots, v_n\}$ de variáveis intervalares. Cada variável intervalar v_i representa uma organização estruturada de decisões interdependentes. Algumas delas são definidas a partir de propriedades análogas àquelas que caracterizam atividades, como $inicio(v_i)$, $fim(v_i)$ e $ocorrencia(v_i)$; além da propriedade $duracao(v_i) = fim(v_i) - inicio(v_i)$, explicitada como forma de controlar diretamente o tempo de duração e intensificar a propagação entre as variáveis constituintes do intervalo. Dessa forma, define-se que uma variável de intervalo v_i representa um evento programado se o predicado $ocorrencia(v_i)$ for satisfeito, e sua extensão temporal é representada pelo intervalo $[inicio(v_i), fim(v_i))$.

Restrições entre Variáveis de Intervalo

De acordo com [Laborie e Rogerie \(2008\)](#), as propriedades previamente elencadas já permitem estabelecer uma estrutura de propagação local. Essa estrutura se dá tanto em termos de restrições lógicas relativas à ocorrência quanto de restrições temporais envolvendo as propriedades de início ou fim, ambas estabelecidas entre pares de variáveis intervalares distintas. No caso de restrições lógicas,

temos restrições como $ocorrencia(v_{i_1}) \rightarrow \neg ocorrencia(v_{i_2})$, que estabelecem relações diretas entre os domínios booleanos de ocorrência das variáveis. No caso de restrições temporais, temos restrições como $fim(v_{i_1}) + k_{i_1 i_2} \leq inicio(v_{i_2})$, que estabelecem relações entre pares de propriedades de variáveis distintas.

Para o caso de interdependências envolvendo uma quantidade arbitrária de variáveis intervalares, algumas restrições globais podem ser empregadas para estabelecer uma hierarquia entre as decisões do modelo. Para uma variável de intervalo v_i e um conjunto de variáveis intervalares $V' \in V \setminus \{v_i\}$, temos os seguintes exemplos:

- **span**⁴ A restrição $span(v_i, V')$ define que a ocorrência de v_i está condicionada à ocorrência de ao menos uma variável $v' \in V'$; e que o início e o fim de v_i coincidem, respectivamente, com o menor e o maior valor de início e fim de alguma variável do conjunto V' . Dito de outra forma, temos as seguintes relações:

$$ocorrencia(v_i) \rightarrow (\exists v' \in V' : ocorrencia(v')) \quad (3.10)$$

$$ocorrencia(v_i) \rightarrow [(inicio(v_i) \leq inicio(v') \ \forall v' \in V') \wedge (\exists v' \in V' : inicio(v') = inicio(v_i))] \quad (3.11)$$

$$ocorrencia(v_i) \rightarrow [(fim(v_i) \geq fim(v') \ \forall v' \in V') \wedge (\exists v' \in V' : fim(v') = fim(v_i))] \quad (3.12)$$

- **alternative** A restrição $alternative(v_i, V')$ define que apenas uma variável $v' \in V'$ pode ocorrer e que seus valores coincidem com os de v_i . Isso pode ser expresso compactamente pela seguinte relação:

$$ocorrencia(v_i) \rightarrow [\exists v' \in V' : ocorrencia(v') \wedge (\neg ocorrencia(v'') \ \forall v'' \in V' \setminus \{v'\}) \wedge span(v_i, \{v'\})] \quad (3.13)$$

Estruturas Complementares às Variáveis de Intervalo

O mapeamento de outras características dos recursos é feito por meio de elementos complementares definidos a partir de variáveis de intervalo. Entre esses elementos, encontram-se sequências, funções cumulativas e funções de estado.

Sequência Uma sequência representa uma permutação de um conjunto de variáveis de intervalo. De modo mais específico, seja s uma variável de sequência sobre um conjunto V_s de variáveis intervalares, então seu domínio é definido como sendo todas as possíveis funções bijetoras $\pi : V_s \rightarrow [1, |V_s|]$, cada uma estabelecendo um sequenciamento distinto das variáveis pertencentes a V_s . Em primeira instância, sequências interessam à modelagem pela possibilidade de estabelecer a restrição *no-overlap*(s), que impede a sobreposição de intervalos e força sua ocorrência na ordem de sequência

⁴Ao contrário de outras terminologias, a nomenclatura de restrições globais difundidas na literatura como *alldifferent*, *span*, *alternative*, *always-in*, *always-nostate* e *always-constant* foi preservada em inglês ao longo do texto para evitar a criação de termos desnecessários e que possam dificultar a consulta à literatura.

das variáveis. Opcionalmente, a restrição pode ser estabelecida pela sintaxe *no-overlap*(s, t) para uma função de tempo de transição $t : V_s \times V_s \rightarrow \mathbb{N}$ que define o tempo mínimo entre o fim e o início de intervalos consecutivos na sequência determinada. Formalmente, essa última definição da restrição corresponde ao seguinte conjunto de relações:

$$fim(v_i) + t(v_i, v_j) \leq inicio(v_j) \quad \forall v_i, v_j \in V_s : \pi(v_i) < \pi(v_j) \quad (3.14)$$

Dessa forma, sequências correspondem à representação de recursos disjuntivos na caracterização por variáveis de intervalo.

Função cumulativa Uma função cumulativa representa uma quantia discreta que varia ao longo do tempo em função de eventos relacionados às variáveis de intervalo. Dessa forma, com base em um domínio temporal T , ela pode ser representada por uma função $f : T \rightarrow \mathbb{N}$. A variação no valor de f se dá em níveis discretos, i.e., por “saltos” instantâneos decorrentes de acréscimos ou decréscimos da quantia representada por meio de coeficientes associados aos eventos de início e/ou fim de determinadas variáveis intervalares. A natureza dessa variação pode ser de duas formas elementares – por degrau ou pulso:

- **Função degrau** O termo *degrau*(t, k) representa uma função com valor 0 no domínio $(-\infty, t)$ e valor k no domínio $[t, +\infty)$, sendo $t = inicio(v_i)$ ou $t = fim(v_i)$ para alguma variável de intervalo v_i .
- **Função pulso** O termo *pulso*(t_1, t_2, k) representa uma função com valor 0 no domínio $(-\infty, t_1)$, valor k no domínio $[t_1, t_2)$ e valor 0 no domínio $[t_2, +\infty)$, sendo $t_1 = inicio(v_i)$ e $t_2 = fim(v_i)$ para alguma variável de intervalo v_i .

A partir desses termos, é possível definir funções cumulativas compostas como $f = degrau(inicio(v_1), 1) - degrau(fim(v_2), 1)$. E sobre essas funções, é possível estabelecer restrições gerais de valor, como $f \geq 0$; ou restrições específicas ao domínio sobre um intervalo, como *always-in*($f, v_1, 0, 1$), que restringe o valor da função f durante a ocorrência do intervalo v_1 aos valores no domínio $[0, 1]$. Ainda é possível que a variação dos termos elementares se dê por uma variável inteira em vez de uma constante. Dessa forma, por meio de funções cumulativas é possível representar desde recursos disjuntivos até recursos reservatórios.

Função de estado De maneira análoga à caracterização de estado para recursos na seção 3.1.1, a função de estado representa um equivalente qualitativo das funções cumulativas. Dessa forma, a definição da função de estado difere da função cumulativa por não explicitar uma composição. No máximo, ela admite uma função de transição para estabelecer o tempo mínimo entre estados consecutivos. Assim sendo, a variação dos estados é completamente determinada pela imposição de restrições tanto da forma *always-in* como de outras similares: *always-nostate*(f, v_i) para estabelecer que a função f não está definida durante a ocorrência do intervalo de v_i , *always-constant*(f, v_i) para estabelecer que a função f não varie de estado a despeito de qual ele seja durante o intervalo de v_i etc. A despeito do estado acabar por ser representado por uma quantidade, o que importa na modelagem por funções de estado é que o domínio seja discreto e a representação por valores inteiros atende a esse critério. Em suma, a função de estado acaba por exercer papel similar às propriedades de estado do recurso.

3.2 Elementos Complementares de Resolução

A resolução de problemas no EBR costuma apresentar algumas peculiaridades que o diferem do que foi apresentado anteriormente para o caso geral da PR. Enquanto que a PR genérica tem uma aplicação mais eficaz no caso de problemas com domínios reduzidos, o mesmo não pode ser dito dos problemas do EBR em razão do tamanho dos domínios temporais geralmente envolvidos. Naquele caso, a situação é propícia para que a propagação possa operar com impacto significativo na redução do espaço de busca e que a ramificação por partição dos domínios seja eficaz em explorar as possibilidades remanescentes. No presente caso, não se espera uma redução significativa de domínios temporais em razão de seu tamanho e da baixa restritividade. Por consequência, o esforço necessário à garantia de que todas as reduções possíveis foram obtidas passa a ser questionável. Isso reflete em duas decisões de projeto importantes no caso do EBR:

- a ramificação não se dá por partição de domínios temporais, mas pela ordenação relativa entre as variáveis temporais, de forma que a posição absoluta seja determinada de acordo com a função objetivo;
- os algoritmos de filtragem são projetados visando eficiência para reduzir ao máximo o espaço de busca na primeira aplicação, nem sempre sendo aplicados repetidamente para convergir a um ponto fixo de consistência.

Essas decisões de projeto estabelecem uma via dupla entre propagação e busca que torna fundamental analisar esses elementos de modo integrado. A forma diversa de ramificação da busca acaba por impactar na forma com que a propagação opera para prover suporte, pois é mais benéfico que exista um alinhamento entre a forma que o espaço de busca é explorado e que a propagação opera para reduzi-lo. E o mesmo se dá no sentido inverso: se limitada, a forma com que a propagação opera deve atentar às ramificações que a busca provavelmente fará na sequência para torná-la mais eficaz. Dessa forma, considerar a propagação e a busca de forma integrada no caso do EBR é fundamental em razão das especificidades que foram observadas e das escolhas pertinentes ao projeto dos resolvedores que elas causaram.

A apresentação que se segue descreve mecanismos de propagação relacionando-os à busca. Em razão da variedade de tipos de algoritmos de propagação e de busca, o detalhamento considera apenas alguns dos mais elementares ou mais amplamente aceitos de acordo com referências como Baptiste *et al.* (2001), Laborie (2003a), Baptiste *et al.* (2006) e Vilím (2007). O tratamento terá um foco muito maior na propagação do que na busca em decorrência da relativa abundância de abordagens e da consolidação da teoria relativa ao primeiro desses elementos, o que evidencia seu grau de importância na resolução. Dessa forma, as considerações finais sobre a busca contemplam apenas o que é alheio à propagação.

3.2.1 Propagação

A propagação de restrições no escalonamento – sobretudo aquela envolvendo restrições globais – representa uma das áreas de maior estudo no desenvolvimento de procedimentos de resolução para o EBR. Por um lado, existem estruturas entre pares de variáveis que podem ser aproveitadas em mecanismos de propagação localizada por meio de algoritmos de filtragem bastante eficientes. Por outro lado, restrições globais emergem com facilidade nos modelos e a propagação sobre estruturas

com maior interdependência é sempre preferível devido ao maior poder de redução do espaço de busca. De modo semelhante aos níveis de consistência para restrições genéricas, restrições globais de escalonamento possuem tipos variados de consistência – mas não necessariamente com alguma relação de nível estabelecida entre si. Dessa forma, incorre um questionamento ainda mais amplo que no caso geral da PR sobre a definição dos melhores tipos de consistência e algoritmos de filtragem associados.

Propagação Local

A propagação localizada em problemas de escalonamento opera sobre redes booleanas e temporais – as primeiras relativas às variáveis de ocorrência de determinados eventos e as segundas relativas às restrições entre o início e o fim de eventos distintos. Essas redes correspondem a grafos orientados restringindo de modo conjugado o domínio das variáveis associadas. A respeito da consistência completa nessas redes, é observado por [Laborie e Rogerie \(2008\)](#) que ela pode ser obtida eficientemente em modelos binários se isolado cada caso, mas certas formas de integração dos dois tipos é suficiente para tornar o tipo de problema resultante NP-Completo.

Redes booleanas O caso mais simples de PSR consiste naquele em que o domínio das variáveis é binário, sendo denominado de problema de satisfabilidade ou simplesmente SAT. Quando as relações se limitam apenas a pares de variáveis, temos um 2-SAT. O 2-SAT pode ser resolvido eficientemente pela construção de um grafo em que cada vértice representa uma atribuição de um valor a uma variável e cada aresta uma restrição pela combinação de atribuições que não podem ocorrer simultaneamente; de forma que o problema não possui solução se, e somente se, houver um caminho entre os vértices das duas atribuições possíveis a uma variável – o que acarreta uma contradição lógica. Quando as relações incluem um número maior de variáveis, [Cormen *et al.* \(2002\)](#) observa que é sempre possível reduzi-lo a um 3-SAT – i.e., a um problema de domínios binários envolvendo restrições entre até 3 variáveis. De qualquer forma, ele já se torna um problema NP-Completo no caso geral. No entanto, algumas referências apontam que a decomposição de restrições em relações booleanas ainda sim pode ser proveitosa. Por exemplo, [Brafman \(2001\)](#) aponta que a decomposição de alguns problemas de planejamento em relações booleanas leva a modelos nos quais a maior parte das restrições envolve até duas variáveis. Por essa razão, [Brafman \(2001\)](#) explora o desenvolvimento de mecanismos de propagação que simplifiquem as restrições não-binárias restantes para torná-las menores em tamanho e quantidade, de forma que seja viável que a busca opere por “tentativa-e-erro” na ramificação das variáveis envolvidas nas restrições remanescentes desse tipo.

Redes temporais A propagação isolada em redes temporais consegue obter frequentemente a consistência completa da rede de modo eficiente devido a premissas observadas na manipulação de marcas de tempo no EBR. Conforme evidencia [Baptiste *et al.* \(2006\)](#), uma das características mais peculiares da propagação em redes temporais é que ela opera sobre os limites do domínio e não sobre cada valor do mesmo. Dois fatores pesam em favor dessa decisão de projeto:

- domínios temporais costumam ser muito maiores do que aqueles em que a consistência sobre cada valor é aplicada, dificultando a aplicação desse tipo de algoritmo de filtragem;
- boa parte das restrições empregadas entre variáveis temporais invalida as fronteiras do domínio e não pontos específicos em seu interior, de forma que o esforço necessário para contemplar a

remoção de valores individuais internos em domínios que são notadamente amplos acaba por não compensar.

Além disso, outro aspecto que caracteriza os algoritmos de filtragem em redes temporais é o fato de que a relação entre variáveis temporais costuma ser binária por expressar o posicionamento relativo entre determinados eventos. Dessa forma, é possível representar, com relativa precisão, uma rede de relações temporais através do Problema Temporal Simples (PTS)⁵.

O PTS consiste em um conjunto de variáveis temporais $X = \{x_1, \dots, x_n\}$ associadas, respectivamente, ao conjunto de domínios contínuos $D = \{[l_1, ls_1], \dots, [l_n, ls_n]\}$ delimitando inferior e superiormente cada variável e um conjunto de restrições entre pares de variáveis $R = \{r_{i_1 i_2}(x_{i_1}, x_{i_2}) \leftrightarrow (a_{i_1 i_2} \leq x_{i_1} - x_{i_2} \leq b_{i_1 i_2}), x_{i_1} \in X, x_{i_2} \in X\}$ que estabelece limites superiores e inferiores para a diferença entre o valor de alguns pares de variáveis temporais em X . A vantagem dessa representação se deve à proposição de filtragem do PTS por meio de algoritmos para caminhos de custo mínimo por [Cesta e Oddi \(1996\)](#), visto que as restrições em R possibilitam estabelecer o menor e o maior tempo de alocação de qualquer variável $x_i \in X$ através de duas aplicações daquele algoritmo. Na primeira aplicação, obtém-se o tempo mínimo de alocação de cada variável x_i como sendo a distância mínima da origem ao vértice que a representa a partir de um grafo com uma origem fictícia no tempo 0, custo da aresta entre essa origem fictícia e o vértice de cada variável x_i dado pelos limite inferior de domínio l_i , e custo da aresta entre vértices cujas variáveis possuem uma restrição $r_{i_1 i_2}$ dado em um sentido como $a_{i_1 i_2}$ e no outro sentido como $b_{i_1 i_2}$. Na segunda aplicação, o tempo máximo de alocação de cada variável x_i é obtido como a negação da distância mínima entre o vértice representante e um destino fictício no tempo 0, sendo o custo da aresta entre o destino e o vértice de cada variável v_i dado pela negação do limite superior, $-ls_i$, e o custo da aresta entre vértices cujas variáveis possuem uma restrição $r_{i_1 i_2}$ dado em um sentido como $-a_{i_1 i_2}$ e no outro sentido como $-b_{i_1 i_2}$. Por fim, sendo mantida a representação que gerou os novos limitantes, é possível atualizar esses limites com base em decisões de ramificação ao longo do processo de busca.

Propagação Global

A propagação global no EBR costuma depender, direta ou indiretamente, da ordenação relativa das variáveis temporais e nem sempre se opera de forma completa. O primeiro desses fatores decorre em parte do motivo já discutido pelo qual a propagação temporal se dá apenas nos limites do domínio: a dimensão dos domínios é relativamente maior e a determinação dos limites é mais significativa do que a eliminação de valores específicos. Além disso, [Laborie \(2003a\)](#) destaca que a frequência com que restrições de precedência temporal ocorrem em modelos de EBR torna sua manipulação fundamental à obtenção de um melhor desempenho na resolução. No entanto, essa tendência foi se estabelecendo de maneira gradativa: primeiro pela definição de tipos de consistência a partir de hipóteses de ordenação das variáveis e depois pela manipulação explícita dessa ordenação. Com relação ao segundo fator, a amplitude dos domínios também representa um argumento contra a convergência da consistência a ponto fixo, i.e., a aplicação repetitiva dos algoritmos de filtragem para garantir a consistência definida, que pode ser necessária se uma modificação de domínio induzir outras modificação para aquele nível de consistência. Conforme elencado no princípio da seção 3.2, uma vez que essas reduções vão se tornando cada vez menos significativas e que o espaço de busca

⁵Denominado *Simple Temporal Problem* (STP) em inglês.

é muito amplo, vale mais à pena direcionar a propagação em função do processo de busca e limitar sua execução em razão do benefício esperado. Dessa forma, a apresentação de alguns dos tipos de consistência mais empregados e conhecidos feita a seguir segue o avanço dessa tendência ao longo dos anos, desde a verificação completa de posições absolutas no tempo até o emprego de verificações incompletas e das relações de precedência como subsídio à propagação. A apresentação é organizada com base em Laborie (2003a) e Baptiste *et al.* (2006), mas é sintetizada pelo agrupamento de tipos similares de propagação.

Propagação por tabela de tempos A tabela de tempos⁶ consiste na propagação pela manipulação de dados absolutos sobre a utilização e disponibilidade dos recursos ao longo do tempo. Dessa forma, ela serve a uma propagação da disponibilidade por duas vias: tanto de como a disponibilidade do recurso impacta no domínio de programação da atividade quanto de como a programação da atividade impacta na disponibilidade do recurso. A relação a ser observada entre a alocação dos recursos e a programação das atividades ao longo do tempo é a seguinte:

$$\sum_{i: inicio(a_i) \leq t < fim(a_i), i \in [1, n]} demanda(r_j, a_i) \leq capacidade(r_j) \quad \forall t \in T, j \in [1, m] \quad (3.15)$$

Propagação disjuntiva e energética As modalidades de propagação disjuntiva e energética inferem reduções dos domínios temporais a partir da constatação de impossibilidade de concorrência entre um par de atividades por um determinado recurso.

- **Disjuntiva** A propagação disjuntiva representa o caso mais simples das duas, no qual a redução de domínios é obtida pela inferência de uma posição relativa entre as atividades. Dito de outra forma, uma atividade deve ser concluída antes do início da outra para qualquer par de atividades (a_{i_1}, a_{i_2}) associadas a um mesmo recurso disjuntivo, conforme a seguinte relação:

$$Dis(a_{i_1}, a_{i_2}) \leftrightarrow [fim(a_{i_1}) \leq inicio(a_{i_2})] \vee [fim(a_{i_2}) \leq inicio(a_{i_1})] \quad (3.16)$$

- **Energética** A propagação energética representa uma generalização desse conceito a partir de demandas arbitrárias para o caso do recurso ser cumulativo em vez de disjuntivo:

$$[demanda(r_j, a_{i_1}) + demanda(r_j, a_{i_2}) \leq capacidade(r_j)] \vee Dis(a_{i_1}, a_{i_2}) \quad \forall j \in [1, m], i_1, i_2 \in [1, n], i_1 \neq i_2 \quad (3.17)$$

Dessa forma, a propagação disjuntiva corresponde ao caso no qual as demandas e a capacidade são todas unitárias.

Propagação por busca e ausência de arestas Devido à tendência em manipular a ordenação relativa das atividades na resolução, uma das modalidades mais exploradas de propagação consiste na análise dessa ordenação entre todas as atividades. Ela representa uma generalização do caso anterior pela manipulação de uma quantidade arbitrária de atividades por meio da definição de propriedades análogas àquelas definidas para cada atividade: para um conjunto de atividades Ω , seja $dl_\Omega = \min_{i: a_i \in \Omega}(dl_i)$ a data de liberação, $dc_\Omega = \max_{i: a_i \in \Omega}(dc_i)$ a data de limite de conclusão e

⁶Denominada *timetable* em inglês.

$tp_\Omega = \sum_{i:a_i \in \Omega} (tp_i)$ o tempo de processamento a plena capacidade do conjunto de atividades $\Omega \subseteq A$. Tal ordenação conjunta pode ser vista como um grafo direcionado denominado grafo de precedências, sendo cada atividade associada a um vértice e cada aresta orientada representando uma relação de precedência entre atividades cuja ordem relativa é determinada. Para fins de notação, as expressões $a_{i_1} \ll a_{i_2}$ e $a_{i_2} \gg a_{i_1}$ denotam ambas que a atividade a_{i_1} deve ser executada antes da atividade a_{i_2} . Dessa representação surgem duas modalidades comuns de propagação:

- **Busca de arestas**⁷ Na busca de arestas, o objetivo é inferir novas arestas no grafo de precedências a partir das janelas de execução das atividades. Tal inferência opera entre um conjunto de atividades Ω e uma outra atividade $a_i \notin \Omega$, de forma a estabelecer se a_i pode ou não ser a primeira ou última atividade a ser executada no conjunto $\Omega \cup \{a_i\}$. Nesse contexto, temos as seguintes relações de inferência possíveis entre as propriedades e a ordem relativa:

$$\forall \Omega \subset A \ \forall a_i \in A \setminus \Omega \ [dc_{\Omega \cup \{a_i\}} - dl_\Omega < tp_\Omega + tp_i] \rightarrow [a_i \ll \Omega] \quad (3.18)$$

$$\forall \Omega \subset A \ \forall a_i \in A \setminus \Omega \ [dc_\Omega - dl_{\Omega \cup \{a_i\}} < tp_\Omega + tp_i] \rightarrow [a_i \gg \Omega] \quad (3.19)$$

Na primeira relação, infere-se que a_i precede as demais atividades em Ω pela impossibilidade de principiar a execução de $\Omega \cup \{a_i\}$ com alguma atividade de Ω e concluir antes da data limite do conjunto $\Omega \cup \{a_i\}$. Na segunda relação, infere-se que a_i sucede as demais atividades em Ω pela impossibilidade de concluir a execução de $\Omega \cup \{a_i\}$ com alguma atividade de Ω devido à data limite do conjunto $\Omega \cup \{a_i\}$. No sentido inverso, a fixação de uma relação de ordem entre uma atividade e um conjunto reduz o domínio de alocação temporal da atividade de acordo com as seguintes relações:

$$\forall \Omega \subset A \ \forall a_i \in A \setminus \Omega \ [a_i \ll \Omega] \rightarrow [fim(a_i) \leq \min_{\emptyset \neq \Omega' \subseteq \Omega} (dc_{\Omega'} - tp_{\Omega'})] \quad (3.20)$$

$$\forall \Omega \subset A \ \forall a_i \in A \setminus \Omega \ [a_i \gg \Omega] \rightarrow [inicio(a_i) \geq \max_{\emptyset \neq \Omega' \subseteq \Omega} (dl_{\Omega'} + tp_{\Omega'})] \quad (3.21)$$

A despeito da existência de $O(n2^n)$ pares do tipo (a_i, Ω) para $|A| = n$, Baptiste *et al.* (2006) observa que existe uma relação de dominância de propagação entre subconjuntos de A , de forma que não mais do que $O(n^2)$ dessas tuplas são necessárias para garantir esse tipo de propagação. O primeiro algoritmo de filtragem adotado para propagação por busca de arestas em recursos disjuntivos foi extraído dos cortes propostos por Carlier e Pinson (1994) para reduzir o tempo de resolução do Problema de Escalonamento de Chão-de-Fábrica (PEC) e tinha complexidade de tempo $O(n \log(n))$. Outros algoritmos foram propostos em sequência, notadamente um de implementação mais simples mas complexidade $O(n^2)$ por Baptiste *et al.* (2001) e outro de complexidade $O(n \log(n))$ – porém sem convergência da consistência a ponto fixo – por Vilím (2007).

- **Ausência de arestas**⁸ A propagação por ausência de arestas representa uma generalização da busca de arestas pela inferência reversa de quais atividades não podem ser a primeira ou a

⁷Denominada *edge-finding* em inglês.

⁸Denominada *not-first / not-last* em inglês.

última na execução de um conjunto. Dessa forma, a inferência de uma aresta posicionando uma atividade a_i com relação a um conjunto de atividades Ω na busca de arestas é equivalente à inferência da ausência de $|\Omega|$ possíveis arestas posicionando cada atividade $a'_i \in \Omega$ com relação ao conjunto $\Omega \cup \{a_i\}$. Apresentando um tempo maior de execução, a redução dos domínios resultante desse tipo de propagação domina a propagação por busca de arestas. As reduções inferidas por esse tipo de propagação são as seguintes:

$$\forall \Omega \subset A \ \forall a_i \in A \setminus \Omega \ [dc_i - dl_\Omega < tp_\Omega + tp_i] \rightarrow [fim(a_i) \leq \max_{i' \in \Omega} (dc_{i'} - tp_{i'})] \quad (3.22)$$

$$\forall \Omega \subset A \ \forall a_i \in A \setminus \Omega \ [dc_\Omega - dl_i < tp_\Omega + tp_i] \rightarrow [inicio(a_i) \geq \min_{i' \in \Omega} (dl_{i'} + tp_{i'})] \quad (3.23)$$

No primeiro caso, infere-se que a_i não pode ser a última atividade a ser executada por não ser possível executar o conjunto $\Omega \cup \{a_i\}$ entre o tempo de liberação dl_Ω e o tempo de conclusão dc_i , de forma que a conclusão de a_i deve preceder o início de execução de ao menos uma atividade de Ω . No segundo caso, infere-se que a_i não pode ser a primeira atividade a ser executada por não ser possível executar o conjunto $\Omega \cup \{a_i\}$ entre o tempo de liberação dl_i e o tempo de conclusão dc_Ω , de forma que o início de a_i deve suceder a conclusão de ao menos uma atividade de Ω . Baptiste *et al.* (2006) e Vilím (2007) observam que os algoritmos de filtragem existentes relevantes a esse tipo de propagação possuem complexidade de tempo $O(n^2)$ no pior caso.

Precedência energética e balanceamento As modalidades de propagação por precedência energética e por balanceamento foram propostas por Laborie (2003a) como uma forma de valer-se explicitamente das precedências entre atividades para obter uma redução mais expressiva dos domínios, em especial no caso em que os domínios temporais das atividades são bastante folgados.

- **Precedência energética** A propagação por precedência energética infere a postergação do início de uma atividade a_i em razão da demanda pela capacidade de recursos de todas as atividades que a precedam. Sendo esse conjunto de atividades precedentes a a_i representado como Ω , temos a seguinte relação:

$$[a_i \gg \Omega] \rightarrow [inicio(a_i) \geq dl_\Omega + \frac{\sum_{i': a_{i'} \in \Omega} (demanda(r_j, a_{i'}) \times tp_{i'})}{capacidade(r_j)}] \ \forall j \in [1, m] \quad (3.24)$$

Ela pode ser obtida por um algoritmo de filtragem apresentado em Laborie (2003a) em tempo $O(n(p + \log(n)))$ para um conjunto de n eventos de variação da capacidade do recurso e até p precedências para cada atividade.

- **Balanceamento** A propagação por balanceamento infere reduções do espaço de busca pela estimativa de limites à capacidade de um recurso reservatório ao longo do tempo em decorrência das relações da ordenação dos eventos associados. Esses limites são definidos sobre cada evento $t = inicio(a_i)$ ou $t = fim(a_i)$ de cada atividade $a_i \in A$, e são de quatro tipos possíveis: $L_{min}^<(t)$ para o menor valor de capacidade do recurso imediatamente antes da ocorrência de um evento t , $L_{max}^<(t)$ para o maior valor nas mesmas circunstâncias; e $L_{min}^>(t)$ e $L_{max}^>(t)$ para os respectivos valores equivalentes imediatamente após a ocorrência do evento t . Cada um

desses limites é estipulado pelas combinações de maior e menor variação de capacidade em decorrência dos eventos que alteram a capacidade do recurso e que podem ocorrer antes ou depois de cada evento. Por meio dessas estimativas, torna-se possível promover quatro formas de propagação:

- descobrir antecipadamente que uma determinada configuração sendo construída na resolução não é viável pela carência de capacidade no recurso;
- deduzir limites mais restritos de capacidade dos recursos face às possibilidades de produção e às necessidades de consumo do reservatório;
- reduzir o domínio temporal dos eventos pela inferência de que alguns eventos devem se posicionar antes ou após determinado evento;
- inferir novas relações de precedência entre os eventos quando o caso anterior se restringir a uma relação entre pares de eventos.

O algoritmo de filtragem proposto por Laborie (2003a) para essa modalidade de propagação possui complexidade de tempo $O(n^2)$ sem a inferência de novas relações de precedência e $O(n^3)$ caso contrário. Por fim, Laborie (2003a) sugere que as informações sobre o balanço energético sejam utilizadas como subsídio às estratégias de ramificação do EBR.

3.2.2 Busca

Conforme mencionado anteriormente, a estruturação da busca no EBR – sobretudo por recursão e retrocesso – se dá em torno dos mecanismos de propagação empregados. De todo modo, os passos gerais da busca por recursão e retrocesso no EBR consistem nos seguintes segundo Baptiste *et al.* (2001):

1. Selecionar um recurso no qual as atividades não estão completamente ordenadas.
2. Selecionar uma atividade para ser executada antes das demais no recurso; fixar a relação de precedência entre ela e as demais; manter uma relação das atividades restantes como substitutas no caso de retrocesso à escolha da primeira atividade no recurso.
3. Iterar sobre o passo anterior até que todas as atividades associadas estejam completamente ordenadas no recurso.
4. Iterar sobre os passos anteriores até que as atividades estejam completamente ordenadas em todos os recursos.

Dessa forma, as estratégias de ramificação no EBR acabam por depender de heurísticas peculiares a problemas de escalonamento para a ordenação de escolha dos recursos e das atividades durante a resolução. Decerto que essa escolha não é completamente independente do tipo de propagação sendo empregado, de forma que a variadas combinações de propagação e busca competem estratégias diversas. Portanto, as especificidades de cada modelo a ser resolvido podem determinar o emprego de combinações distintas.

Por fim, convém observar que o caráter declarativo da PR corrobora a elaboração de algoritmos de busca estruturados de maneira diversa. Notadamente, um dos campos mais profícuos foi o de busca local. No resolvidor subjacente à linguagem OPL, operadores de busca local para modificar

soluções de problemas de escalonamento cumulativo foram propostas em [Godard e Laborie \(2005\)](#) e mecanismos adaptativos para a escolha dos operadores mais propícios em cada rodada de resolução foram estudados em [Laborie e Godard \(2007\)](#). No resolvidor subjacente à linguagem Comet, o estudo descrito em [Monette *et al.* \(2009\)](#) revela um distanciamento de heurísticas especializadas em problemas de escalonamento para a aplicação de meta-heurísticas mais propícias a cada modelo em razão de sua estrutura. Em ambos os casos, evidencia-se a constatação de [Dechter \(2003\)](#) sobre o propósito de abordagens de busca local: resolver problemas com tamanho em ordens de grandeza maiores do que seria possível de modo sistemático. Convém observar que essa linha de mecanismos adaptativos para a resolução de EBRs seguida em ambos os casos já havia sido explorada por outros autores no âmbito estrito da academia, como [Carchrae e Beck \(2005\)](#). A abordagem de escolha automatizada condiz com uma linha em evidência no ambiente acadêmico em geral, para a qual [Smith-Miles \(2008\)](#) provê uma revisão bastante ampla e [Battiti *et al.* \(2008\)](#) uma discussão sobre as principais decisões de projeto envolvidas. Portanto, há um forte indicativo que essa linha de desenvolvimento persista por algum tempo para a resolução de problemas por EBR.

Parte III

ESTUDO DE CASO

Capítulo 4

O Problema da Programação de Recursos Críticos e Sua Abordagem

Recursos críticos como sondas e barcos lançadores de linhas são utilizados nas etapas sequenciais de exploração, produção e manutenção de poços de petróleo marítimos. Informações de uma etapa são necessárias às etapas seguintes, de forma que o desenvolvimento e a manutenção dos poços de um reservatório são projetados a partir dos dados obtidos na etapa de exploração. A exploração é realizada em um conjunto de localizações nas quais estudos geológicos preliminares identificaram a possibilidade de que existam reservas de petróleo e gás em quantidade suficiente para que a produção possa ser comercialmente viável. Uma vez confirmada a existência da reserva por perfurações iniciais, são feitos planos para a perfuração de um conjunto de poços cuja sequência de perfuração e entrada em produção é estipulada de forma a maximizar a extração possível naquele reservatório. Alguns poços são produtores e servem à extração de óleo e gás do reservatório, enquanto outros são injetores e servem à inserção de água ou gás para manter a pressão de produção do poço. O estabelecimento de cada novo poço envolve atividades de perfuração, completação e interligação. Ao final da perfuração, o poço possui um revestimento externo de proteção e um duto interno para o escoamento de fluidos. Com a completação, instala-se um componente de controle de vazão na cabeça do poço conhecido como Árvore de Natal Molhada (ANM). Linhas são conectadas a cada ANM, interligando-as a uma Unidade Estacionária de Produção (UEP) na superfície ou a um *manifold* intermediário que concentra o escoamento de vários poços para reduzir o número de linhas ligadas diretamente à UEP. Por fim, atividades periódicas de manutenção são programadas em uma janela temporal flexível, de forma a manter a operação dos poços dentro dos patamares desejáveis. Dessa forma, o emprego desses recursos críticos atende a necessidades bastante distintas ao longo do ciclo de vida dos reservatórios, sendo que a limitação em sua oferta impacta diretamente as operações de produção da companhia.

As atividades a serem realizadas pelos recursos críticos possuem prazos, relações de precedência e características técnicas que devem ser obedecidas. Por exemplo, a profundidade na qual é empreendida uma atividade deve ser compatível com a faixa de profundidades permitida à operação de um dado recurso, enquanto que este deve possuir a bordo todos os equipamentos necessários à execução daquela. Conforme observado no Capítulo 1 e no parágrafo anterior, o nível de detalhamento dessas atividades aumenta ao longo do ciclo de vida em razão das informações previamente adquiridas. Assim sendo, a programação dos recursos críticos para o desenvolvimento e a manutenção de

poços marítimos possui diversas peculiaridades ignoradas quando se considera apenas a etapa de exploração.

4.1 Decisões e Entidades Relacionadas

A programação de recursos críticos na produção pode ser vista como uma questão de decidir *se*, *quando* e *como* realizar cada uma dentre várias atividades valendo-se de um conjunto de sondas e barcos. O tempo de programação das atividades nos recursos é discretizado em dias, a partir de uma data inicial definida como 0. Cada atividade de desenvolvimento está associada a uma localidade consistindo em um poço marítimo. Algumas atividades podem exigir a instalação de linhas que, por sua vez, devem ser carregadas previamente em um porto. Os carregamentos de linhas em portos representam um dos tipos de atividade auxiliar necessárias à completa programação do recurso. Outros eventos poderiam ser considerados como tipos de atividade auxiliar, como os períodos de deslocamento e as manutenções programadas dos recursos. No entanto, a falta de liberdade sobre boa parte desses eventos em relação aos demais faz com que eles não sejam considerados explicitamente como decisões do problema mas apenas restrições à ocorrência dos restantes. Portanto, as principais decisões esperadas em uma solução a este problema são as seguintes:

- a alocação de recursos a atividades de desenvolvimento;
- a definição do número de cargas a serem associadas a cada recurso;
- a associação de linhas a cada carga;
- a programação temporal das atividades alocadas em cada recurso.

Por convenção de notação no detalhamento do problema, cada tipo de entidade principal emprega um índice diferente a seguir. *Recursos* são representados pelo índice i , que varia entre 1 e n_r . *Atividades* são representadas pelo índice j , que varia entre 1 e n_a , sendo as primeiras n_p atividades associadas ao desenvolvimento de poços e o restante a cargas em portos. *Localidades* são representadas pelo índice k , que varia entre 1 e n_{lo} , sejam elas poços ou portos. *Linhas* são representadas pelo índice l , que varia entre 1 e n_l . Qualquer outro índice, quando necessário, será representado por x .

Sendo o número de atividades auxiliares de carga desconhecido, o valor de n_a é indeterminado a princípio. Como forma de estabelecer um limite superior fixo, emprega-se a relação $n_a = n_p + n_l$. Essa relação contempla o caso extremo em que cada linha é carregada separadamente das demais, de forma que cada atividade $j \in [n_p + 1, n_p + n_l]$ atende às restrições necessárias ao carregamento da linha $l = j - n_p$. Dessa feita, uma das decisões do problema passar a ser pela ocorrência ou não de cada uma das $n_a - n_p$ atividades de carga resultantes.

4.2 Critérios de Otimização

O objetivo do problema da programação de recursos críticos é antecipar ganhos e reduzir custos. No presente caso, o objetivo primário é maximizar a produção de curto prazo, definida como a quantidade que cada poço irá produzir desde o dia em que é conectado até uma data limite H . A cada atividade j é associada uma taxa de produção diária p_j desde o dia em que é concluída,

sendo não-nula somente quando for a atividade de conclusão do desenvolvimento de um poço. Assim sendo, atividades associadas a poços com maior produção acabam por ter prioridade na escala. No entanto, nem todas as atividades são concluídas antes dessa data H e outras não possuem impacto na produção independentemente do dia de conclusão. Por essa razão, um critério secundário é empregado: a minimização da soma dos tempos de conclusão das atividades da solução. Por meio desse critério de desempate, a conclusão das atividades nos recursos é antecipada e estes são liberados o quanto antes. Apesar de fugir ao escopo do problema, essa antecipação representa uma redução de gastos no caso de recursos alugados ou uma maior disponibilidade futura para atividades ainda não previstas. Dessa forma, maximiza-se o retorno imediato associado ao emprego dos recursos disponíveis por meio de uma configuração que procura reduzir custos futuros.

4.3 Restrições do Problema

4.3.1 Restrições Gerais dos Recursos

- Cada recurso pode realizar apenas uma atividade por vez.
- Cada recurso i possui um período de contrato do dia ic_i até o dia fc_i , e não está disponível fora desse período.
- Um recurso pode realizar apenas atividades que sejam compatíveis com suas características. Temos $c_{ij} = 1$ se, e somente se, o recurso i e a atividade j forem compatíveis.
- Durante o período de contrato, existem indisponibilidades previstas do recurso. Tais indisponibilidades representam períodos durante os quais nenhuma atividade pode ser realizada pelo recurso. Para um recurso i , seja nir_i o número de indisponibilidades associadas e o par (iir_{ix}, fir_{ix}) as respectivas datas de início e fim da x -ésima indisponibilidade do recurso i .
- Algumas indisponibilidades podem ser flexíveis dentro de um intervalo em vez de completamente pré-determinadas, sendo dir_{ix} o tempo de duração da x -ésima indisponibilidade do recurso i , que deve ocorrer dentro do intervalo temporal $[iir_{ix}, fir_{ix}]$.
- Certas indisponibilidades podem ser parciais, significando uma restrição da capacidade do recurso que impede a execução de algumas, mas não de todas as atividades. Seja $pir_{ix} = 1$ se, e somente se, a x -ésima indisponibilidade do recurso i for parcial e $cir_{ijx} = 1$ se, e somente se, a possibilidade de realização da atividade j no recurso i for afetada em razão da indisponibilidade x .
- Apenas um recurso por vez pode operar sobre cada poço.
- Para que um recurso realize atividades consecutivas em localidades distintas, é necessário considerar o tempo para mover o recurso de um lugar para o outro.
- Alguns recursos não podem aproximar-se mais do que uma certa distância um do outro devido ao risco de colisão. No entanto, essa restrição não é detalhada por falta de dados à disposição para teste.

4.3.2 Restrições Gerais de Execução das Atividades

- Cada atividade é realizada ininterruptamente desde seu início até a conclusão.
- Cada atividade j possui datas pré-definidas de início mais cedo e início mais tarde, representadas por imc_j e imt_j .
- Cada atividade j está relacionada a uma localidade loc_j .
- Para cada atividade j que não de carregamento, a duração é fixa e dada por d_j .
- Uma atividade pode ser precedida por outras, cada uma exigindo um tempo mínimo de intervalo entre a conclusão de uma e o início da outra. Sem perda de generalidade, podemos definir $prc_{j_1 j_2}$ como o tempo mínimo entre a conclusão da atividade j_1 e o início da atividade j_2 , sendo $prc_{j_1 j_2} = -\infty$ caso não exista uma relação de precedência entre j_1 e j_2 .
- Algumas atividades podem pertencer a um cluster, e todas as atividades de um cluster devem ser realizadas por um mesmo recurso. Sem perda de generalidade, podemos supor que cada atividade j está associada a um cluster de índice $cluster_j$, de forma que atividades pertencentes a um mesmo cluster tenham valores iguais para esse índice e atividades que não pertencem a nenhum cluster tenham um valor único.

4.3.3 Restrições de Estoque dos Recursos

- Todo recurso inicia suas atividades com estoque vazio.
- Cada recurso i possui uma capacidade máxima de estoque ce_i , que é não-nula apenas para barcos de carregamento.
- Apenas recursos com capacidade de estoque positiva podem realizar atividades de carregamento nos portos.
- Cada porto k pode receber até s_k atividades de carregamento ao mesmo tempo.
- Uma atividade de carregamento realizada por um recurso i dura de mic_i a mac_i dias.
- O aumento do estoque de um recurso i devido a uma atividade de carregamento é linearmente proporcional à sua duração, atingindo ce_i se durar mac_i dias.

4.3.4 Restrições de Estoque das Atividades

- Cada linha l está associada a uma atividade de conexão cl_l , na qual a linha é descarregada do recurso.
- Para realizar a atividade de conexão de uma linha, o recurso precisa primeiramente realizar a atividade de carregamento envolvendo a mesma linha.
- Cada linha l possui peso pl_l para fins de impacto no estoque do recurso e pode ser carregada na base bl_l a partir de sua data de disponibilização dl_l .
- Atividades de carregamento não estão sujeitas a restrições de cluster ou precedência.

- Uma atividade j de manutenção de linhas já instaladas pode exigir alguma capacidade disponível cd_j no recurso que a realizar.

4.4 Modelagem por Variáveis de Intervalo

A abordagem de resolução do problema apresentado como estudo de caso consiste em um modelo de Escalonamento Baseado em Restrições (EBR) que emprega variáveis de intervalo. Conforme discutido no Capítulo 3, variáveis intervalares representam uma abstração para o agregado de decisões interdependentes relacionado à modelagem de problemas de escalonamento por meio de uma sintaxe que facilita a manipulação de eventos opcionais. Tal manipulação se dá através de estruturas hierárquicas, como é o caso da relação entre uma determinada tarefa e todas as possibilidades de realização dessa tarefa por configurações distintas de uso dos recursos disponíveis. Sendo nosso foco dirigido à concepção do modelo mais adequado à resolução do problema, enfatizamos as seguintes diretrizes como orientação às diversas decisões de projeto relativas ao desenvolvimento do modelo que se segue:

- obter uma representação do problema correta e compacta a partir da sintaxe de modelagem à disposição;
- reduzir o espaço de busca e intensificar a propagação decorrente da instanciação de cada variável de decisão;
- prover diretrizes a uma exploração mais eficiente do espaço de busca resultante do modelo.

Essas questões são debatidas ao longo da descrição do modelo na presente seção. Na sequência, a seção 4.5 apresenta duas abordagens para estimar um limite superior ao valor ótimo de produção, de forma a analisar a qualidade das soluções obtidas.

Boa parte das variáveis de decisão e restrições do problema de programação de recursos críticos pode ser descrita através da sintaxe de variáveis intervalares, para a qual Laborie e Rogerie (2008) e Laborie *et al.* (2009) propuseram tanto uma notação algébrica quanto gráfica. Na descrição que se segue, as duas modalidades de notação são adaptadas com a finalidade de facilitar a compreensão do modelo sendo descrito. Convém observar que a seção A.5 do apêndice apresenta a implementação desses conceitos por meio da linguagem OPL, e um exemplo menos detalhado de abordagem ao problema é apresentado na seção A.5.9. Pontos específicos dessa parte do apêndice serão referenciados conforme necessário ao longo do texto. Além disso, ressaltamos que as representações gráficas que complementam a descrição do modelo algébrico não ilustram explicitamente todas as relações possíveis entre as variáveis do modelo ou os pormenores de cada um deles. A disposição de ilustrações buscou elucidar ao menos um exemplo de cada tipo de relação do modelo e destacar os fatores mais relevantes a cada tipo de relação como forma de transmitir visualmente a noção das hierarquias presentes no modelo.

4.4.1 Associação entre Atividades e Recursos

Cada atividade e cada recurso do problema é representado por meio de um conjunto de variáveis de decisão do modelo dispostos em vetores ou matrizes. A representação mais elementar do

conjunto de atividades se dá por meio do vetor de intervalos $\mathbf{a}$, que contém uma variável de intervalo associada a cada atividade do modelo, sendo as propriedades de cada variável correspondentes àquelas da atividade em si. A implementação desse conceito em OPL é descrita na seção A.5.2 do apêndice. Como as primeiras n_p atividades do problema referem-se ao desenvolvimento de poços, sua ocorrência é compulsória e isso repercute em uma restrição de obrigatoriedade dos respectivos intervalos:

$$\text{ocorrencia}(a_j) = 1 \quad \forall j \in [1, n_p] . \quad (4.1)$$

A associação entre atividades e recursos se dá pela matriz de intervalos M , que representa o produto cartesiano dos conjuntos de recursos e atividades, de forma que cada variável $m_{ij} \in M$ representa o intervalo associado ao recurso i realizando a atividade j . O vetor $\mathbf{a}$ e a matrix M são relacionados por restrições *alternative*, cuja sintaxe em OPL é descrita na seção A.5.4. Essa restrição associa cada entrada de um vetor a uma coluna da outra matriz¹, de forma que no máximo um intervalo de cada coluna j de M – representada por M_j – ocorra e seus valores correspondem aos do intervalo a_j :

$$\text{alternative}(a_j, M_j) \quad \forall j \in [1, n_a] . \quad (4.2)$$

A representação dos eventos associados a cada recurso se dá pelo vetor de sequências $\mathbf{r}$, sendo que cada sequência $r_i \in \mathbf{r}$ agrupa as entradas de M associadas ao i -ésimo recurso, ou seja, r_i agrupa as entradas do vetor $(M^T)_i$. A representação de sequências em OPL é descrita na seção A.5.5. Considerando cada sequência como um conjunto dos intervalos, temos o seguinte:

$$r_i = \{m_{ij} : j \in [1, n_r]\} . \quad (4.3)$$

De forma a prevenir que um recurso seja associado a mais do que uma atividade ao mesmo tempo, a restrição *no-overlap* é associada a cada sequência $r_i \in \mathbf{r}$. Como argumento adicional, é definida uma função de tempo de transição $t_i : [1, n_a] \times [1, n_a] \rightarrow \mathbb{N}$ entre a conclusão de um intervalo e o início de outro em função da localização das atividades associadas e do recurso em consideração:

$$\text{no-overlap}(r_i, t_i) \quad \forall i \in [1, n_r] . \quad (4.4)$$

A figura 4.1 apresenta os conceitos anteriormente descritos para modelar a associação entre atividades e recursos, além de um dos tipos de indisponibilidade descritos adiante.

Tanto nessa figura quanto nas demais que se seguem são adotadas algumas convenções de notação para facilitar a compreensão do modelo. Variáveis intervalares são apresentadas como retângulos, sendo acinzentadas se sua presença é obrigatória. Sequências são apresentadas como quadrados com cantos arredondados e funções cumulativas como círculos, tendo ambas uma linha diagonal no preenchimento interno. Restrições e composições de sequências ou funções são apresentadas por meio de setas, sendo rotuladas pela restrição ou regra de agrupamento que definem. O agrupamento de elementos é feito por retângulos mais espessos. Possíveis restrições entre intervalos de um determinado agrupamento são delimitadas por retângulos pontilhados. Por fim, conforme mencionado anteriormente, as restrições apresentadas graficamente representam um tipo de cada e não todas aquelas que existem no modelo.

¹ Por convenção de notação, quando necessário, um vetor formado pela j -ésima coluna de M será representado como M_j e um vetor formado pela i -ésima linha de M será representado como $(M^T)_i$.

Figura 4.1: Relação elementar entre intervalos associados a atividades e recursos.

A definição dos domínios temporais de programação dos intervalos e dos domínios booleanos de alocação dos recursos é feita sobre os elementos da matriz M . Essa escolha se deve à maior eficácia de propagar da matriz M para o vetor $\mathbf{a}$ e do fato de tornar-se desnecessário o emprego de intervalos para representar a extensão do contrato de cada recurso. A ocorrência de cada intervalo m_{ij} depende da compatibilidade entre o recurso i e a atividade j , condicionado por $c_{ij} = 1$:

$$ocorrencia(m_{ij}) \rightarrow (c_{ij} = 1) \quad \forall i \in [1, n_r], j \in [1, n_a]. \quad (4.5)$$

O início de cada intervalo m_{ij} de desenvolvimento está sujeito tanto ao início do contrato do recurso ic_i quanto à data de início mais cedo da atividade imc_j :

$$ocorrencia(m_{ij}) \rightarrow [inicio(m_{ij}) \geq MAX(ic_i, imc_j)] \quad \forall i \in [1, n_r], j \in [1, n_p]. \quad (4.6)$$

No caso de atividades de carga, a correspondência entre cada atividade j e a linha $l = j - n_p$ leva à substituição da data de início mais cedo pela data de disponibilização dl_l , de forma que temos a seguinte restrição para o início de cada carregamento:

$$ocorrencia(m_{ij}) \rightarrow [inicio(m_{ij}) \geq MAX(ic_i, dl_{j-n_p})] \quad \forall i \in [1, n_r], j \in [n_p + 1, n_a]. \quad (4.7)$$

De forma similar, o fim de cada intervalo m_{ij} de desenvolvimento está sujeito tanto ao fim do contrato do recurso fc_i quanto à soma da data de início mais tarde imt_j da atividade com a duração d_j da mesma:

$$ocorrencia(m_{ij}) \rightarrow [fim(m_{ij}) \leq MIN(fc_i, imt_j + d_j)] \quad \forall i \in [1, n_r], j \in [1, n_p]. \quad (4.8)$$

No caso de atividades de carga, apenas o fim do contrato do recurso é levado em conta:

$$ocorrencia(m_{ij}) \rightarrow [fim(m_{ij}) \leq fc_i] \quad \forall i \in [1, n_r], j \in [n_p + 1, n_a]. \quad (4.9)$$

Por sua vez, a duração de cada intervalo m_{ij} associado a uma atividade de desenvolvimento é fixa

da seguinte forma:

$$ocorrencia(mi_{ix}) \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.15)$$

$$inicio(mi_{ix}) \geq iir_{ix} \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.16)$$

$$fim(mi_{ix}) \leq fir_{ix} \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.17)$$

$$duracao(mi_{ix}) = dir_{ix} \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.18)$$

A partir das entradas de MI , é possível estender a restrição anterior ao caso geral de indisponibilidades totais:

$$[pir_{ix} \neq 1] \rightarrow always-in(u_i, mi_{ix}, 0, 0) \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.19)$$

A mesma estrutura poderia ser estendida para o caso de indisponibilidades parciais com o emprego de uma função cumulativa para cada conjunto distinto de atividades limitado por alguma indisponibilidade. No entanto, a modelagem adotada nesse caso considerou o emprego de sequências que agrupam cada intervalo de programação de indisponibilidade com intervalos das atividades que são respectivamente incompatíveis com o mesmo. A composição dessas sequências constitui a matriz MP , na qual cada elemento mp_{ix} consiste em uma sequência associada à indisponibilidade x do recurso i . Cada sequência mp_{ix} agrupa o intervalo de indisponibilidade mi_{ix} e os intervalos de M associados ao recurso i cuja compatibilidade é afetada pela indisponibilidade x :

$$mp_{ix} = \{mi_{ix}\} \cup \{m_{ij} : j \in [1, n_p] \wedge c_{ij} = 1 \wedge cir_{ijx} = 1\} \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.20)$$

Dessa forma, é possível estabelecer a restrição de *no-overlap* sobre cada sequência de MP :

$$[pir_{ix} = 1] \rightarrow no-overlap(m_{ix}) \quad \forall i \in [1, n_r], x \in [1, nir_i] . \quad (4.21)$$

Essa escolha alternativa de modelagem para as indisponibilidades parciais se deu pela necessidade de uma propagação dedicada à interdependência entre o conjunto de intervalos envolvidos, algo que não se fazia tão necessário no caso anterior devido à redundância decorrente da restrição de *no-overlap* sobre as sequências do vetor $\mathbf{r}$ e as funções cumulativas do vetor $\mathbf{u}$.

4.4.2 Associação entre Atividades e Localidades

De acordo com o limite de concorrência, cada localidade do problema é associada a uma sequência ou a uma função cumulativa, de forma a atender às restrições do problema. Localidades representando poços são associadas a sequências do vetor $\mathbf{p}$, enquanto que portos são associados a funções cumulativas do vetor $\mathbf{b}$.

Cada sequência $p_k \in \mathbf{p}$ representa o conjunto de atividades associadas ao desenvolvimento de um poço identificado pela localidade k :

$$p_k = \{a_j : j \in [1, n_p] \wedge loc_j = k\} \quad \forall k \in [1, n_{lo}] . \quad (4.22)$$

De forma a prevenir que duas atividades sejam programadas de modo concorrente em um poço, a restrição *no-overlap* é estabelecida sobre cada sequência $p_k \in \mathbf{p}$:

$$no-overlap(p_k) \quad \forall k \in [1, n_{lo}] . \quad (4.23)$$

Para modelar a restrição de distância de segurança entre recursos, a mesma lógica poderia ser usada para estabelecer restrições sobre sequências representando atividades cuja execução concorrente por determinados recursos não é permitida.

Cada função cumulativa $b_k \in \mathbf{b}$ representa o número de cargas ocorrendo ao longo do tempo no porto identificado pela localidade k :

$$b_k = \sum_{i=n_p+1}^{n_a} pulso(a_j, 1) \quad \forall k \in [1, n_{lo}] . \quad (4.24)$$

Dessa forma, o controle de concorrência em cada porto pode ser garantido pela seguinte restrição:

$$b_k \leq s_k \quad \forall k \in [1, n_{lo}] . \quad (4.25)$$

Por fim, é comum que existam relações de precedência entre atividades relacionadas ao desenvolvimento de um mesmo poço, de forma que elas são descritas na presente seção em vez da anterior. Elas são estabelecidas diretamente sobre os elementos do vetor $\mathbf{a}$:

$$fim(a_{j_1}) + prc_{j_1 j_2} \leq inicio(a_{j_2}) \quad \forall j_1 \in [1, n_p], j_2 \in [1, n_p] . \quad (4.26)$$

Os conceitos de associação entre atividades e localidades no modelo são ilustrados na figura 4.3.

Figura 4.3: Relação entre intervalos associados a atividades e suas respectivas localidades.

4.4.3 Restrições de Estoque e Transporte de Linhas

O controle do estoque a bordo de barcos é feito por meio de funções cumulativas do vetor $\mathbf{l}$ e das matrizes LB , CLB e DLB . Cada elemento $lb_{ik} \in LB$ representa o estoque associado às linhas carregadas por um recurso i e um porto k , sendo definido como a diferença entre a função cumulativa $clb_{ik} \in CLB$ relativa aos carregamentos realizados pelo recurso i no porto k e a função cumulativa $dlb_{ik} \in DLB$ relativa aos descarregamentos equivalentes:

$$clb_{ik} = \sum_{j=n_p+1:loc_j=k}^{n_a} grau(fim(m_{ij}, ce_i * \frac{duracao(m_{ij})}{mac_i}) \quad \forall i \in [1, n_r], k \in [1, n_{lo}] . \quad (4.27)$$

$$dlb_{ik} = \sum_{l=1:bl_l=k}^{n_l} \text{degrau}(\text{fim}(m_i, cl_l, pl_l)) \quad \forall i \in [1, n_r], k \in [1, n_{lo}] . \quad (4.28)$$

$$lb_{ik} = clb_{ik} - dlb_{ik} \quad \forall i \in [1, n_r], k \in [1, n_{lo}] . \quad (4.29)$$

Como pode ser observado pelas relações acima, a cada atividade de carga é associado um acréscimo de estoque proporcional à sua duração e capacidade de estoque mas inversamente proporcional ao tempo necessário ao carregamento completo; e a cada atividade de descarga é associado um decréscimo fixo proporcional à soma do peso das linhas que descarrega. O estoque no recurso i é representado ao longo do tempo pela função cumulativa $l_i \in \mathbf{I}$, que é definida sobre cargas e descargas associadas a cada porto k por meio da função cumulativa $lb_{ik} \in LB$ e sobre estocagens temporárias associadas aos intervalos de $(M^T)_i$ na forma de pulsos:

$$l_i = \sum_{k=1}^{n_{lo}} lb_{ik} + \sum_{j=1}^{n_p} \text{pulso}(m_{ij}, cd_j) \quad \forall i \in [1, n_r] . \quad (4.30)$$

A definição separada da função de estoque associada à carga em cada porto é necessária para restringir que o volume de origem em cada porto não seja descarregado em maior quantidade do que foi carregado:

$$lb_{ik} \geq 0 \quad \forall i \in [1, n_r], k \in [1, n_{lo}] . \quad (4.31)$$

Para completar o controle dos limites de estoque, resta a classe de restrições de estoque dos recursos em si ao longo do tempo, que atenta apenas ao limite a bordo devido à garantia de não-negatividade advinda do conjunto de restrições anterior:

$$l_i \leq ce_i \quad \forall i \in [1, n_r] . \quad (4.32)$$

A figura 4.4 ilustra as restrições de estoque estabelecidas sobre cada recurso apresentando a composição de LB diretamente das entradas de M .

Figura 4.4: Representação das restrições de estoque do recurso por porto e na totalidade.

De forma a garantir que as linhas com restrição de data sejam carregadas apenas após o prazo determinado, um conjunto de restrições é estabelecido a partir da estrutura de cargas opcionais proporcionais ao número de linhas existentes descrita previamente. Nesse contexto, a restrição associada ao carregamento de cada linha l por cada recurso i é descrita da seguinte forma:

$$\begin{aligned} & \text{ocorrencia}(m_{icl_l}) \rightarrow \\ & MIN_{j \in [n_p+1, n_a]: \text{ocorrencia}(m_{ij}) \wedge dl_{j-n_p} \geq dl_l} fim(m_{ij}) \leq inicio(m_{icl_l}) \\ & \forall i \in [1, n_r], l \in [1, n_l] . \end{aligned} \quad (4.33)$$

Dessa forma, a ocorrência de ao menos uma carga cuja data mínima de início é pelo menos tão tarde quanto a data de disponibilidade da linha em questão e que o tempo de conclusão da mesma seja menor do que o tempo de início da descarga da linha no mesmo recurso garantem que cada linha pode ser carregada no tempo certo. A partir dessa garantia, a associação das linhas às cargas pode ser feita de modo trivial.

4.4.4 Função Objetivo

A composição da função objetivo de maximização fo é baseada em uma combinação balanceada das funções de produção de curto prazo fop e de somatório dos tempos de conclusão das atividades foc :

$$fop = \sum_{j=1}^{n_p} MAX(H - fim(a_j), 0) * p_j ; \quad (4.34)$$

$$foc = \sum_{j=1}^{n_a} (fim(a_j)) ; \quad (4.35)$$

$$fo = M_p * fop - M_c * foc . \quad (4.36)$$

O balanceamento dos coeficientes M_p e M_c deve ser feito de forma a garantir que os valores assumidos por foc não afetem as decisões a ponto de alterar o valor assumido por fop , o que teoricamente denota-se por $M_p \gg M_c$. Sem perda de generalidade, essa definição pode ser feita sobre um dos coeficientes para um valor fixo do outro. No entanto, convém observar que uma diferença de grandeza entre os coeficiente além do necessário pode afetar a precisão do resultado. Dessa forma, optou-se pelo valor fixo $M_p = 1$ e, sob a hipótese de valores inteiros para a taxa de produção, M_c tal que $\frac{1}{M_c} > foc_{MAX}$. O valor máximo assumível pela função foc nas soluções, foc_{MAX} , pode ser estimado por pré-inspeção dos dados do problema.

4.5 Limitante Superior ao Ótimo

De forma a avaliar a qualidade das soluções obtidas, duas abordagens distintas foram desenvolvidas para estabelecer limitantes superiores à produção de curto prazo de instâncias do problema. Com base nos limitantes obtidos e em sua diferença com relação às soluções obtidas pelo modelo, é possível estimar a diferença, no pior caso, entre o valor de produção dessas soluções e o valor ótimo de produção: ela corresponde ao ótimo ser igual ao menor desses limites. As duas abordagens consistem na relaxação de um conjunto de restrições do problema, de forma que uma solução

ótima do problema relaxado possa ser obtida. No entanto, é pouco provável que a solução de tal problema relaxado seja também uma solução viável do problema original. No primeiro caso, o problema resultante é resolvido com um algoritmo guloso com baixo custo computacional. No segundo caso, um modelo de PLI é formulado para ser rodado em um resolvidor de PM. O desenvolvimento da primeira abordagem é descrito em [Serra *et al.* \(2011\)](#) e visou estabelecer um limitante que não dependesse de outras ferramentas de resolução. A segunda abordagem é descrita em [Serra *et al.* \(2012\)](#) e visou obter uma estimativa mais precisa por relaxar menos o problema original. Apesar dos resultados obtidos com a segunda abordagem serem melhores, ambas são descritas a seguir com o intuito de ilustrar a diferença dos resultados a partir do nível de relaxação adotado em cada caso.

4.5.1 Algoritmo Guloso sobre Relaxação

O algoritmo de inferência de limitante superior opera gerando uma instância relaxada do problema cuja resolução é trivial e, resolvendo-a, gera um valor de produção superestimado devido às relaxações. As relaxações presentes na instância gerada são de quatro tipos, cada qual afetando um aspecto distinto da instância:

1. **Número e duração das atividades de desenvolvimento:** Reduz o desenvolvimento de cada poço a uma única atividade cuja duração é dada pelo tempo mínimo entre o início da primeira atividade e o término da última da sequência da instância original do problema.
2. **Produção associada às atividades:** Associa o maior valor de produção de um poço da instância original à atividade da instância relaxada com menor duração e assim por diante.
3. **Disponibilidade e compatibilidade dos recursos:** Considera todos os recursos disponíveis indefinidamente a partir do início de seu contrato e compatíveis com todas as atividades.
4. **Linhas a serem transportadas:** Ignora todas os transportes e cargas temporárias da instância original na instância relaxada.

Assim sendo, a solução ótima dessa relaxação pode ser obtida pela escala das atividades em ordem não-crescente de produção – e, por consequência, não-decrescente de duração – nos recursos disponíveis. Dessa forma, a complexidade de pior caso do algoritmo é $O(n_a^2 + n_{lo} * \log(n_{lo}) + n_{lo} * n_r)$, sendo o primeiro termo relativo à determinação da duração mínima do desenvolvimento de cada poço, o segundo relativo à ordenação das atividades associadas a cada poço tanto por duração quanto por produção e o terceiro relativo à alocação dessas atividades nos recursos. A duração mínima do desenvolvimento de cada poço é obtida pelo emprego de um algoritmo de caminhos mais curtos a partir de uma única origem através da conversão dos tempos mínimos à conclusão de cada atividade em vértices para os quais é medida a distância com relação ao vértice associado ao instante inicial da escala. Portanto, o algoritmo de limitante superior gera uma relaxação cuja produção ótima é ao menos tão grande quanto a do problema original, resolve-a de modo eficiente e gerando um limitante válido à produção de curto prazo.

4.5.2 Modelo de PLI sobre Relaxação

Apesar de válida e de fácil implementação, a estimativa que pode ser obtida pelo método descrito na seção anterior acaba por ser muito fraca devido à grande quantidade de premissas adotadas para

relaxar o problema. De forma a possibilitar que uma solução ótima seja obtida por um algoritmo guloso, muitas restrições fundamentais foram ignoradas. Por esse motivo, uma segunda proposta de estimativa foi desenvolvida com base em um modelo de PLI. Nesse caso, apenas a obrigatoriedade das atividades e os detalhes relacionados a linhas como existência, necessidade de carga em portos e manutenção dos limites de estoque nos recursos foram abstraídos. Como resultado da manutenção de restrições de compatibilidade, do conjunto de atividades de desenvolvimento em cada poço e da disponibilidade de cada recurso, um limite menor foi obtido em [Serra *et al.* \(2012\)](#) para o mesmo conjunto de instâncias e escala adotados em [Serra *et al.* \(2011\)](#) com o uso da abordagem anterior. Convém observar que o modelo de PLI não é sempre resolvido rapidamente, de forma que um tempo limite é estabelecido para que o limitante superior do modelo relaxado seja utilizado no lugar da solução ótima do mesmo. O modelo descrito a seguir representa a configuração com melhor desempenho na análise empírica desenvolvida em [Serra *et al.* \(2012\)](#).

O conjunto de variáveis de decisão do modelo MIP é descrito na tabela 4.1. Como pode ser observado, variáveis contínuas foram utilizadas para representar a programação das atividades ao longo do tempo, sendo acompanhadas por variáveis inteiras para representar a ocorrência da atividade, sua atribuição a um recurso e seu sequenciamento com relação a outras atividades.

Var.	Domain	Description
S_j	$\mathbb{R}^+$	Tempo de início da atividade j .
E_j	$\mathbb{R}^+$	Tempo de finalização da atividade j .
Y_j	$\{0, 1\}$	Se a atividade j termina antes de H .
X_j	$\mathbb{R}^+$	Dias antes da atividade j se concluída se $Y_j = 1$.
F_j	$\mathbb{R}^+$	Variável auxiliar que é positiva se $Y_j = 0$.
W_{ij}	$\{0, 1\}$	Se o recurso i é atribuído à atividade j .
$Z_{j_1 j_2}$	$\{0, 1\}$	Se a atividade j_1 precede a atividade j_2 .

Tabela 4.1: Variáveis de decisão do modelo PLI do problema relaxado.

A função objetivo (4.37) descreve a maximização da produção até o dia H :

$$\max. \sum_{j=1}^{n_p} (H \times Y_j - X_j) \times p_j \quad (4.37)$$

Ainda que seja possível representar uma produção negativa com essa função se $X_j > 0$ e $Y_j = 0$ para uma atividade j , tal situação não ocorre na resolução do problema porque uma produção maior poderia ser atingida com $X_j = 0$ e $F_j = E_j$. Dessa forma, a solução parcial obtida em cada etapa da busca sobre esse modelo sempre terá $X_j = 0$ em uma situação desse tipo.

Para as restrições (4.42), (4.50) e (4.53) a seguir, existe um coeficiente M_j associado a cada atividade j . Esse coeficiente deve ter sempre um valor tão grande quanto qualquer atribuição válida a E_j . Na literatura, esse tipo de coeficiente geralmente é chamado de “grande M^2 ”. Apesar de ser possível definir $M_j = imc_j$, o uso de um valor menor para cada um desses coeficientes facilitaria o processo de resolução do problema. Dessa forma, em caráter preliminar, um modelo contendo apenas as restrições (4.38) a (4.55) é rodado para atualizar o valor de M_j como $\overline{E_j} + H - 1$, onde $\overline{E_j}$ é o valor atribuído a E_j como resultado da rodada. Explicações mais detalhadas a respeito do comportamento do processo de resolução em função do valor desses coeficientes podem ser

²Denominado “big M ” em inglês.

encontradas em Camm *et al.* (1990), onde uma avaliação empírica dos problemas de escalabilidade decorrentes da seleção de valores muito grandes para esse tipo de coeficiente é discutida.

As restrições (4.38) a (4.40) limitam o valor das variáveis temporais dos conjuntos S e E em função da data de início mais cedo, da data de início mais tarde e da duração da atividade.

$$S_j + d_j = E_j \quad \forall j \in [1, n_p] \quad (4.38)$$

$$S_j \geq imc_j \quad \forall j \in [1, n_p] \quad (4.39)$$

$$E_j \leq imt_j + d_j \quad \forall j \in [1, n_p] \quad (4.40)$$

A semântica dos conjuntos de variáveis X e Y conforme descrita na tabela 4.1 é garantida pelas restrições (4.41) a (4.43). Dessa forma, $F_j > 0$ se, e somente se, $Y_j = 0$ para cada atividade j . De outra forma, $X_j = E_j$. Assim sendo, os conjuntos de variáveis X e Y podem ser usados para compor a função objetivo e representar a produção obtida.

$$E_j = X_j + F_j \quad \forall j \in [1, n_p] \quad (4.41)$$

$$F_j \leq M_j \times (1 - Y_j) \quad \forall j \in [1, n_p] \quad (4.42)$$

$$E_j \geq H \times (1 - Y_j) \quad \forall j \in [1, n_p] \quad (4.43)$$

A restrição (4.44) é um corte incluído para facilitar a resolução. Esse corte provê uma ligação adicional entre os conjuntos Y e X , de forma a evidenciar quando uma atividade não pode ser realizada antes do dia H .

$$X_j + Y_j \leq H \quad \forall j \in [1, n_p] \quad (4.44)$$

Cortes são restrições inferidas a partir do modelo se levadas em conta as restrições de integralidade. Essas restrições adicionais não são necessárias para garantir a corretude do modelo, mas elas colaboram com a redução do espaço de busca pela eliminação de regiões que não possuam soluções inteiras, o que facilita a resolução do problema.

As relações de precedência são garantidas pela equação (4.45) sobre os conjuntos de variáveis S e E , bem como pelo corte adicional (4.46) em Y . Esse corte limita a ocorrência de uma atividade antes de H ao caso no qual as suas predecessoras também ocorram antes de H .

$$E_{j_1} + prc_{j_1 j_2} \leq S_{j_2} \quad \forall j_1, j_2 \in [1, n_p] \quad (4.45)$$

$$Y_{j_1} \geq Y_{j_2} \quad \forall j_1, j_2 \in [1, n_p], prc_{j_1 j_2} \neq -\infty \quad (4.46)$$

A equação (4.47) garante que cada atividade concluída antes de H é atribuída a um recurso, bem como que esse recurso seja compatível com a atividade.

$$\sum_{i \in [1, n_p]: c_{ij}=1} W_{ij} = Y_j \quad \forall j \in [1, n_p] \quad (4.47)$$

A equação (4.48) limite a duração total das atividades atribuídas a cada recurso à disponibilidade

de curto prazo do mesmo.

$$\begin{aligned} & \min\{H - 1, fc_i\} - \min\{H - 1, ic_i\} \\ & \geq \sum_{j \in [1, n_p]: c_{ij}=1} d_j \times W_{ij} \quad \forall i \in [1, n_r] \end{aligned} \quad (4.48)$$

As equações (4.49) e (4.50) limitam a programação de cada atividade ao contrato do recurso ao qual ela é atribuída.

$$S_j \geq ic_i \times W_{ij} \quad \forall i \in [1, n_r], j \in [1, n_p], c_{ij} = 1 \quad (4.49)$$

$$\begin{aligned} E_j & \leq fc_i + M_j \times (1 - W_{ij}) \quad \forall i \in [1, n_r], j \in [1, n_p], \\ & c_{ij} = 1 \end{aligned} \quad (4.50)$$

As restrições (4.51) a (4.53) limitam a ocorrência simultânea de atividades. Esse tipo de restrição deve-se à ocorrência de atividades em um mesmo poço e à utilização de um mesmo recurso por atividades distintas. No caso de atividades em um mesmo poço, o tempo de transição em (4.53) é nulo. No caso de atividades em localidades distintas, ele é definido como o maior tempo de transição de um recurso compatível com ambas as atividades.

$$Z_{j_1 j_2} + Z_{j_2 j_1} \leq 1 \quad \forall j_1, j_2 \in [1, n_p], j_1 < j_2 \quad (4.51)$$

$$\begin{aligned} Z_{j_1 j_2} + Z_{j_2 j_1} & \geq Y_{j_1} \quad \forall j_1, j_2 \in [1, n_p], \\ & j_1 \neq j_2, loc_{j_1} = loc_{j_2} \end{aligned} \quad (4.52)$$

$$\begin{aligned} E_{j_1} + dt_{j_1 j_2} - (M_{j_1} + dt_{j_1 j_2}) \times (1 - Z_{j_1 j_2}) \\ \leq S_{j_2} \quad \forall j_1, j_2 \in [1, n_p], j_1 \neq j_2 \end{aligned} \quad (4.53)$$

A atribuição das atividades pertencentes a um mesmo cluster é restrita a recursos capazes de realizar aquelas atividades programadas no curto prazo pela restrição (4.54) e a um único recurso pela restrição (4.55).

$$\begin{aligned} W_{ij_1} & \leq 1 - Y_{j_2} \quad \forall i \in [1, n_r], j_1, j_2 \in [1, n_p], \\ & c_{ij_1} = 1, c_{ij_2} \neq 1, \\ & cluster_{j_1} = cluster_{j_2} \end{aligned} \quad (4.54)$$

$$\begin{aligned} W_{ij_1} & \leq 1 - Y_{j_2} + W_{ij_2} \quad \forall i \in [1, n_r], j_1, j_2 \in [1, n_p], \\ & c_{ij_1} = 1, c_{ij_2} = 1, \\ & cluster_{j_1} = cluster_{j_2} \end{aligned} \quad (4.55)$$

Por fim, o sequenciamento temporal das atividades atribuídas a um mesmo recurso é garantido pela restrição (4.56). Nesse caso, se duas atividades j_1 e j_2 são realizadas por um único recurso,

$Z_{j_1 j_2} = 1$ ou $Z_{j_2 j_1} = 1$ para evitar sua ocorrência simultânea.

$$\begin{aligned}
 Z_{j_1 j_2} + Z_{j_2 j_1} &\geq W_{i j_1} + W_{i j_2} - 1 \\
 \forall i &\in [1, n_r], j_1, j_2 \in [1, n_p], \\
 c_{i j_1} &= 1, c_{i j_2} = 1 \\
 j_1 &< j_2
 \end{aligned} \tag{4.56}$$

Capítulo 5

Análise Experimental do Escalonador

Alguns experimentos foram conduzidos para analisar o desempenho do escalonador obtido a partir da implementação do modelo proposto em um resolvidor comercial. O objetivo primário desses experimentos foi avaliar a qualidade das soluções obtidas ao longo do tempo e o resultado ao final do tempo de resolução estipulado pelo usuário final. Dessa forma, a abordagem descrita no Capítulo 4 foi implementada e testada sobre um conjunto de dados reais conforme a especificação descrita na seção 5.1. Os experimentos realizados e os resultados obtidos para os mesmos são apresentados na seção 5.2 e discutidos na seção 5.3.

5.1 Dados Disponíveis e Ambiente de Testes

A avaliação experimental do escalonador foi baseada em um conjunto de dados representando um cenário real ocorrido no passado. Devido ao fato desse cenário representar uma única instância do problema, outras instâncias foram geradas a partir desta como forma de avaliar a estabilidade de desempenho do resolvidor. A instância *O* representa o cenário completo com 465 atividades em 171 poços. Essa instância foi particionada em conjuntos de atividades de tamanhos similares para gerar instâncias de porte menor a serem testadas individualmente, todas empregando o conjunto completo de recursos disponível. O primeiro particionamento das atividades foi em duas metades, *H1* e *H2*, e a segunda em quatro quartos, *Q1* a *Q4*. A tabela 5.1 descreve as principais características de cada uma dessas instâncias.

O modelo de variáveis intervalares foi implementado usando a linguagem OPL, para a qual o Apêndice A provê uma pequena introdução. O escalonador foi executado em uma única *thread* por meio do resolvidor IBM Cplex Studio 12.2. O computador usado nos testes possui 4 processadores

Instância	<i>Q1</i>	<i>Q2</i>	<i>Q3</i>	<i>Q4</i>	<i>H1</i>	<i>H2</i>	<i>O</i>
Atividades	116	118	116	115	231	234	465
Poços	46	37	45	43	82	89	171
Linhas	17	17	13	19	32	34	66
Sondas	64						
Barcos	9						
Indisponibilidades	12						

Tabela 5.1: Principais características das instâncias testadas.

Instância		Q1	Q2	Q3	Q4	H1	H2	O
Tempo Solução Inicial (s)	Média	3,14	2,99	3,86	2,82	10,19	12,71	13,62
	Desvio	0,08	0,16	0,22	0,12	0,64	2,63	6,64
Valor Solução Inicial	Média	97,23	86,03	99,08	86,13	95,98	91,86	88,31
	Desvio	0,13	22,17	0,52	8,58	1,19	0,41	0,98
Valor Solução Final	Média	99,98	100,00	99,98	99,88	99,66	99,60	98,98
	Desvio	0,03	0,00	0,02	0,20	0,28	0,35	1,18
Limitante Superior	Algoritmo	122,20	138,56	117,82	118,30	122,83	125,02	135,61
	Modelo	104,14	114,25	100,65	103,52	105,44	105,98	115,81

Tabela 5.2: Principais resultados dos experimentos do escalonador.

Dual-Core AMD Opteron 8220, 16 Gb de memória RAM e sistema operacional Linux. Por fim, o algoritmo descrito na seção 4.5.1 para inferir um limitante superior para o ótimo foi implementado em Java em razão da baixa demanda por recursos do mesmo e à necessidade de compatibilidade e facilidade de manutenção com relação ao sistema existente. O modelo em PLI descrito na seção 4.5.2 foi implementado em OPL e utilizou o mesmo resolvidor para sua resolução.

5.2 Experimentos Realizados e Resultados

Os testes realizados procuraram avaliar a estabilidade do resolvidor, seu comportamento de acordo com o tamanho do conjunto de atividades e a qualidade das soluções obtidas com base nos limitantes obtidos. Eles são similares aos publicados em [Serra *et al.* \(2011\)](#) e [Serra *et al.* \(2012\)](#), que foram ambos realizados em uma versão simplificada do problema descrito neste trabalho. Essa versão simplificada diferia por contemplar apenas indisponibilidades totais fixas e não possuir restrição de disponibilidade ou porto de embarque das linhas. Tais requisitos foram ignorados na época por ainda estarem em discussão. Cada instância foi resolvida quatro vezes pelo escalonador pelo tempo limite de uma hora, sendo registradas as soluções obtidas ao longo do processo de resolução. Como forma de reduzir o efeito de más escolhas em questões de desempate arbitrário no processo de ramificação ao longo da busca, a repetição do processo de resolução foi feita com sementes de aleatorização distintas. O uso de sementes de aleatorização e o fato de resultados diferentes serem esperados com seu uso se deve à arbitração de critérios de desempate ao longo da busca, conforme descrito na seção A.4.7 do apêndice. O tempo limite estabelecido é baseado na expectativa de tolerância do usuário final para obter uma solução, sendo empregado tanto no modelo de PR quanto no de PLI. Por motivo de sigilo, a medida dos valores de produção apresentados para cada solução é normalizada com base no valor 100 para a melhor solução obtida em cada instância. Essa normalização também visou facilitar a comparação de progresso da resolução em instâncias distintas. Os principais resultados desses experimentos estão resumidos na tabela 5.2.

Como forma de demonstrar o progresso do escalonador enquanto resolve o modelo, os gráficos das figuras que se seguem representam a melhor solução obtida a cada instante de resolução. As figuras 5.1 a 5.4 representam, respectivamente, o progresso nas 4 rodadas realizadas sobre as instâncias Q1 a Q4. Por sua vez, as figuras 5.5 e 5.6 ilustram o progresso das 4 rodadas sobre as instâncias H1 e H2. Por fim, a figura 5.7 apresenta o progresso de resolução das 4 rodadas sobre a instância O.

Figura 5.1: Melhoria da solução obtida em cada rodada do escalonador para a instância Q1.

Figura 5.2: Melhoria da solução obtida em cada rodada do escalonador para a instância Q2.

Figura 5.3: Melhoria da solução obtida em cada rodada do escalonador para a instância Q3.

Figura 5.4: Melhoria da solução obtida em cada rodada do escalonador para a instância Q4.

Figura 5.5: Melhoria da solução obtida em cada rodada do escalonador para a instância H1.

Figura 5.6: Melhoria da solução obtida em cada rodada do escalonador para a instância H2.

Figura 5.7: Melhoria da solução obtida em cada rodada do escalonador para a instância *O*.

5.3 Discussão

Os resultados indicam que o tempo limite estabelecido pelo usuário do escalonador é satisfatório para evitar flutuações muito grandes nos resultados finais. Isso pode ser observado pelo fato de que a média do valor das soluções finais obtidas nas rodadas para cada instância não ficou muito abaixo de 1% da melhor solução obtida. Além disso, os gráficos de melhor solução obtida ao longo do tempo evidenciam uma convergência a determinados patamares de valores. No entanto, a tendência aparente de aumento do valor do limitante com relação à melhor solução obtida e a maior dispersão das curvas relativas às instâncias com um maior número de atividades evidenciam que os resultados poderiam ser melhores na hipótese de um tempo limite de execução maior para a resolução de instâncias maiores.

De qualquer forma, a influência da quantidade de atividades da instância para a variação dos resultados obtidos foi mais relevante com relação ao tempo para obtenção e ao valor da solução inicial obtida do que com relação ao valor da solução final. Para as instâncias *Q1* a *Q4*, apenas uma rodada de *Q2* e três rodadas de *Q4* tiveram uma solução inicial com diferença maior do que 3% com relação à melhor solução obtida em uma hora, sendo que as soluções seguintes passam a figurar dentro desse patamar. Outro fator relevante para a convergência das rodadas é a quantidade de linhas da instância, uma vez que a flutuação observada para os resultados da instância *Q4* – que possui uma quantidade acima da média para o número de atividades – foi maior. Nas instâncias *H1* e *H2*, as soluções iniciais de todas as rodadas tinham uma diferença inferior a 10% com relação à melhor solução obtida em uma hora. No caso da instância *O*, no máximo duas soluções por rodada estavam abaixo da diferença de 10%. Com relação à qualidade das soluções, todas as rodadas obtiveram uma solução final a menos de 16% da solução ótima. Com exceção de *Q2*, esse valor foi menor do que 6% nas menores instâncias. Além disso, convém observar que essa distância pode ser ainda menor, uma vez que a diferença entre o valor das soluções obtidas e o ótimo frequentemente não equivale à diferença com relação ao limitante. Dessa forma, há uma grande evidência da robustez do escalonador devido à pequena quantidade de tempo para a obtenção de uma solução inicial, à estabilidade dos resultados finais entre rodadas independentes e sua diferença máxima com relação ao ótimo.

A despeito desses resultados, ainda há muitas possibilidades a serem consideradas para melhorar

a qualidade da solução inicial e reduzir o valor do limitante inferido. O desenvolvimento de um gerador de soluções iniciais para servir de entrada ao escalonador proposto serviria para levar em conta o conhecimento de quem realiza essa programação manualmente e, por consequência, evitar a degeneração no início da busca. Se bem-sucedido, esse gerador levaria a uma diminuição da variação observada entre instâncias e aumentaria a confiança do usuário final no sistema. A suspeita sobre a possibilidade de ser possível reduzir o limitante estimado deriva do fato que o carregamento de linhas sequer foi considerado na relaxação PLI, que obteve os limitantes mais precisos. No mais, aquele valor é um pouco elevado se levarmos em conta que a melhoria das soluções obtidas em instâncias menores cessa precocemente em algumas rodadas e melhorias significativas não são normalmente esperadas nesses casos. Portanto, é possível que uma relaxação ainda menos simplificada do problema possa ser obtida em tempo hábil e levar a uma avaliação mais precisa da qualidade dos resultados do escalonador.

Parte IV

CONCLUSÃO

Capítulo 6

Conclusão

Este trabalho procurou atingir dois objetivos inter-relacionados. O primeiro era de introduzir em português a técnica de Programação por Restrições (PR) e seu desdobramento com foco na abordagem de problemas de escalonamento – o Escalonamento Baseado em Restrições (EBR) – como ferramental válido à abordagem de problemas de otimização. A apresentação se deu por meio de uma revisão bibliográfica sobre o tema, incluindo uma definição dos principais conceitos, a contextualização de algumas das ferramentas comerciais disponíveis e um tutorial para facilitar o uso de uma das linguagens de modelagem mais utilizadas atualmente – a OPL. O segundo era de abordar o Problema da Programação de Recursos Críticos, que consiste na programação de sondas e barcos para o desenvolvimento de poços de petróleo. Tendo em vista as recentes descobertas de reservas abundantes na camada do pré-sal da costa brasileira, a obtenção eficiente de soluções de alta qualidade em tempo razoável é de grande importância para que o país se posicione como um grande exportador de petróleo nos próximos anos. Dessa forma, a abordagem do problema por meio do uso da técnica permitiu ilustrar como empregá-la em um problema real, com o intuito de que sirva de exemplo e encorajamento para que outros profissionais também tentem utilizá-la.

Convém observar que a abordagem proposta representa apenas uma das alternativas possíveis. Não obstante, os resultados obtidos para o cenário real estudado foram bastante otimistas, uma vez que uma solução a menos de 16% do ótimo foi obtida para um problema com cerca de 70 recursos e mais de 400 atividades. Dessa forma, observamos como possibilidades de trabalho futuro tanto abordar em um maior nível de detalhe o problema tratado quanto recorrer a outras técnicas de resolução para a promoção de uma análise comparativa. Essa segunda possibilidade diz respeito tanto à resolução do problema quanto à obtenção de um limitante superior mais preciso para avaliar a qualidade das soluções. Portanto, não se espera que este trabalho tenha caráter conclusivo e asserções definitivas em nenhum aspecto, mas antes que sirva como base para futuros desenvolvimentos.

Parte V

APÊNDICE

Apêndice A

Programação por Restrições em OPL

O propósito deste apêndice é apresentar os principais elementos da linguagem de modelagem OPL, com ênfase naqueles relacionados à técnica de programação por restrições. Dessa forma, a apresentação da linguagem não será completa e alguns conceitos serão introduzidos à medida que necessários ao escopo definido. Ainda sim, essa apresentação cobrirá uma boa parte dos elementos de uso geral à modelagem de problemas de decisão e de otimização. No entanto, convém a ressalva de que parte dos elementos são exclusivos à modelagem pela técnica de programação por restrições.

Esse material foi adaptado de uma apostila elaborada pelo autor para ministrar um curso sobre o emprego da linguagem OPL na técnica de programação por restrições. Boa parte dessa apostila é baseada no manual da [IBM \(2010\)](#), ainda que em um escopo bastante reduzido e numa estrutura diferente. Para a adequada compreensão do material, é desejável algum conhecimento de otimização e de linguagens de programação. O tratamento teórico sobre programação por restrições apresentado na segunda parte da dissertação pode ser de alguma valia, mas não será exigido de maneira explícita. No decorrer do texto, será introduzida a linguagem OPL e elementos importantes à modelagem de problemas combinatórios e de escalonamento, em ordem crescente de complexidade. Dessa forma, a leitura do apêndice pode ser independente do restante da dissertação ao passo que a forma como o texto está estruturado difere um pouco devido ao propósito com que foi elaborado originalmente.

A organização do restante do apêndice segue a estrutura descrita a seguir. Na seção [A.1](#), é dada uma visão geral sobre o que é a linguagem OPL, com especial ênfase ao resolvidor de programação por restrições *CP Optimizer*. A separação entre modelo e dados é tratada na seção [A.2](#). Nesse ponto, começa a descrição da linguagem por meio de diversos exemplos auto-contidos em cada seção, de forma a facilitar a compreensão sequencial e consulta posterior do material. Essa apresentação se prolonga por 3 seções: a seção [A.3](#) cobre elementos básicos da linguagem, desde a definição do resolvidor, passando pela manipulação das estruturas de dados mais comuns até a formulação de expressões; a seção [A.4](#) cobre variáveis de decisão, função objetivo, restrições e pós-processamento, além de um exemplo no final; e a seção [A.5](#) trata das estruturas específicas para problemas de escalonamento, com um exemplo de sua utilização. Por fim, a seção [A.6](#) apresenta algumas conclusões e agradecimentos aos revisores das versões preliminares desse texto.

A.1 A Linguagem OPL e o Resolvidor CP

OPL é uma linguagem de modelagem algébrica para a descrição de problemas decisórios ou de otimização. Ela era de propriedade da ILOG, que foi adquirida em 2009 pela IBM. A descrição

de um problema de otimização nessa linguagem consiste em um modelo. Uma vez especificado o modelo, um resolvidor compatível com a linguagem pode ser utilizado para resolver o problema. No caso da linguagem OPL, existem dois resolvidores compatíveis: o CPLEX e o CP.

A OPL pode ser entendida como uma linguagem de programação como outra qualquer, de acordo com a visão de Knuth destacada por Lustig e Puget (2001), pois ela se propõe a instruir uma máquina sobre como resolver um determinado problema. Porém, ela não é feita para uso geral mas para expressar problemas decisórios ou de otimização de maneira clara. Além disso, sua estrutura é essencialmente declarativa, de forma que boa parte do modelo descrito expressa *qual* é o problema e não *como* resolvê-lo. Porém, tanto no pré quanto no pós-processamento, é possível incluir alguns trechos de programação imperativa para tratamento dos dados, impressão na tela e configuração do resolvidor, conforme veremos adiante.

O grande diferencial da linguagem OPL foi que, ao contrário de suas predecessoras, ela dá suporte à descrição de problemas para a utilização de resolvidores de programação por restrições como o resolvidor CP. Até então, linguagens de modelagem só serviam à descrição de problemas de programação matemática, desde programação linear a casos especiais de programação não-linear, que em OPL são cobertos pelo resolvidor CPLEX. Esse diferencial amplia a possibilidade de aplicação da linguagem devido às vantagens em se utilizar programação por restrições em alguns tipos de problemas para os quais as técnicas de resolução utilizando programação matemática não são viáveis devido ao excessivo consumo de tempo e memória para uma representação completa dos modelos associados. Nesses casos, mesmo que uma abordagem utilizando programação por restrições resulte em uma solução sem garantias de otimalidade, ao menos é possível obter uma solução factível em tempo hábil e com menor exigência computacional. E sendo uma ferramenta de modelagem algébrica, a possibilidade de uso dessas técnicas torna-se acessível a um público mais amplo do que apenas programadores de computadores.

A programação por restrições consiste na modelagem e resolução de problemas combinatórios empregando variáveis de decisão com domínios discretos e finitos, e restrições capazes de expressar concisamente certas combinações de valores e domínios. A mesma modelagem, em programação matemática, seria muito mais complexa ou mesmo inviável em alguns casos. Apesar da falta de garantias das técnicas empregadas na programação por restrições, mecanismos de redução dos domínios baseados na propagação das restrições e algoritmos específicos para a resolução de algumas classes especiais de problemas foram responsáveis por um considerável ganho de credibilidade, consolidando a modelagem em programação por restrições como uma boa prática para diversos problemas combinatórios e, em especial, de problemas de escalonamento.

A.2 Representação do Modelo e da Entrada de Dados

A descrição do modelo em OPL consiste em um arquivo geralmente com extensão *.mod* que é interpretado linha a linha. Como os dados do problema não são sempre os mesmos por variar de instância para instância, é importante considerar a entrada de dados separadamente. Isso é feito por meio de um segundo arquivo com extensão *.dat*. Ao separar modelagem e a entrada de dados, evita-se que o projetista do modelo se preocupe em tratar detalhes da entrada no mesmo arquivo utilizado para representar a formulação do problema.

No arquivo de descrição do modelo, interessa-nos manipular o conjunto de dados, descrever o

problema de otimização e dar saída a esse resultado de alguma maneira. Dessa forma, não convém o tratamento explícito da entrada de dados, que só pode ser feito por meio de constantes dentro do arquivo do modelo. Através do arquivo *.dat*, a entrada de dados pode ser feita por meio de acesso a banco de dados, planilhas eletrônicas ou arquivos de texto que obedeçam a algumas regras de formatação. Por simplicidade, nos ateremos apenas a esse último caso no restante do texto, especialmente porque o acesso a banco e planilhas é especificado por meio de um arquivo de texto nessa sintaxe, e o leitor pode encontrar facilmente como fazê-lo para uma variedade de plataformas na documentação oficial da linguagem. Portanto, o modelo consiste fisicamente em um arquivo de texto interpretado em linha de comando em conjunto com um, nenhum ou vários arquivos de entrada de dados, geralmente com extensão *.dat*. Vale lembrar que, para que uma execução não resulte em erro imediato, é necessário que todos os dados necessários à execução do modelo estejam definidos exatamente uma vez – seja no código do modelo ou nos demais arquivos de entrada de dados – e que não exista nenhum dado sem correspondência ao modelo nesses arquivos.

Num exemplo simples de execução em linha de comando com um arquivo de modelo *modelo.mod* e um arquivo de dados *dados.dat*, temos a seguinte chamada:

```
oplrun modelo.mod dados.dat
```

Apesar de interpretado em linha de comando, a execução pode ser realizada por meio de ambientes fornecidos pela ILOG como a OPL IDE. Além do desenvolvimento de modelos e execução, a OPL IDE permite que parâmetros do resolvidor possam ser configurados em um terceiro arquivo de configurações com extensão *.ops*. Entretanto, muitos desses parâmetros podem ser definidos no próprio modelo, motivo pelo qual será abordada apenas essa última opção quanto o assunto for tratado.

A.3 Elementos de Uso Geral da Linguagem

Nesta seção, são apresentados muitos dos elementos de uso geral da linguagem OPL e alguns peculiares ao resolvidor CP. É importante frisar ao leitor que não se trata de uma cobertura exaustiva, mas de uma que garanta ao leitor domínio sobre as estruturas comuns de maior uso. Essa cobertura procura dar embasamento ao acompanhamento do restante do texto.

Muitos trechos de código serão apresentados, e por isso convém explicitar desde já a convenção adotada. Todos eles serão delimitados por barras formando um quadro. Trechos de código do modelo serão apresentados com barras duplas e trechos do arquivo de entrada de dados serão apresentados com barras simples. A numeração de linhas só ocorrerá se o modelo for completo, não se aplicando a trechos isolados. No texto subsequente a cada trecho de código, comentários sobre o exemplo dado podem conter referências a variáveis em *itálico* ou trechos entre aspas e *itálico*, representando parte do código ilustrado ou alguma pequena variação a partir desse código. Esse trecho pode ou não ser válido na linguagem. Em especial, trechos incorretos apresentados como contra-exemplo não foram destacados por meio de quadros para evitar que pudessem ser confundidos com os trechos corretos.

A.3.1 Escolha do Resolvidor do Modelo

Devido ao foco restrito à programação por restrição e sua aplicação à modelagem de problemas combinatórios ou de escalonamento, o texto não se aterá apenas a conceitos gerais da linguagem

OPL, mas também a elementos especialmente desenvolvidos para melhor expressar problemas desse tipo. Assim, fica claro que certos modelos desenvolvidos com base nesses elementos não são compatíveis com um resolvidor de programação matemática, restando a questão de como poderia o interpretador do modelo determinar qual resolvidor utilizar para um modelo descrito se ambos servirem. Por padrão, um modelo em OPL é interpretado como sendo de programação matemática, a menos que na sua primeira linha exista uma referência em contrário. Atualmente, dois resolvidores são possíveis: o CP para programação por restrições e o CPLEX para programação matemática. Para utilizar o CP, faz-se necessário constar o seguinte na primeira linha do modelo:

```
using CP;
```

A.3.2 Comentários em Arquivos de Modelo e de Dados

Como em muitas outras linguagens, é possível inserir algum texto explicativo que seja ignorado pelo resolvidor tanto no arquivo de modelo quanto nos arquivos de dados, mas que seja de grande valia para outros seres humanos compreenderem com maior facilidade um código alheio. Os comentários em OPL possuem a mesma sintaxe que em C++ ou Java: qualquer texto após a ocorrência de um `//` numa linha é ignorado, bem como qualquer texto entre uma ocorrência de `/*` e a próxima ocorrência de `*/`, conforme o exemplo abaixo:

```
// Linha comentada.
/*
 * Bloco de texto comentado.
 */
```

A.3.3 Variáveis de Tipos Primitivos e Atribuições

Para modelar qualquer problema relevante, é necessários conhecer primeiramente os tipos primitivos e estruturas de dados mais básicas da linguagem. Na OPL, existem três tipos primitivos: *int*, *float* e *string*. Ao longo desse texto, serão enfatizados os dois primeiros tipos. Na declaração de uma variável, é facultativo realizar uma atribuição inicial no arquivo de modelo. No exemplo abaixo, são declaradas uma variável inteira sem atribuir um valor, bem como uma variável inteira e outra de ponto flutuante com atribuição de valor. Repare que cada instrução é finalizada por um ponto-e-vírgula em OPL:

```
int noFabricas;
int noDepositos = 10;
float taxaProducao = 5.0;
```

Para que uma variável tenha seu valor inicial atribuído por alguma fonte de dados externa, é realizada uma atribuição simbólica utilizando reticências (três pontos) em vez de um valor, conforme o exemplo abaixo:

```
int noFabricas = ...;
```

Em correspondência a essa declaração, em algum dos arquivos de dados deverá haver uma atribuição relacionada, como a apresentada a seguir:

```
noFabricas = 7;
```

Quando não é atribuído um valor explicitamente, a inicialização da variável terá de ocorrer por meio de blocos de programação imperativa, representados pela palavra reservada *execute*, como no trecho de modelo abaixo:

```
int noFabricas;
execute {
    noFabricas = 7;
}
```

Dentro desses blocos, é utilizada uma linguagem de script chamada ILOG Script, cuja estrutura é baseada em Javascript. Conforme abordado adiante, é possível declarar variáveis nos trechos de ILOG Script, mas tais variáveis não servem à modelagem do problema e sim a controles internos desses blocos. Afinal, tais variáveis são visíveis somente dentro dos blocos imperativos e sequer possuem tipo explícito.

Quando um modelo é interpretado, ele é processado linha a linha. Dessa forma, quando o bloco é executado, ele só tem visibilidade e poder de ação sobre as variáveis com declaração nas linhas acima dele. É importante frisar que com os blocos de script é possível alterar o valor atribuído a uma variável ao longo do processamento do modelo, algo que não se pode fazer de maneira declarativa na linguagem. O ILOG Script será tratado novamente em outros pontos ao longo do texto.

A.3.4 Estruturas de Dados Básicas: Vetores e Domínios

Por meio de vetores e domínios já se torna possível representar de maneira razoável os dados de muitos problemas, de forma que boa parte dos exemplos dados ao longo do texto empregarão essas estruturas de dados.

O vetor definido em OPL é muito similar ao encontrado em outras linguagens de programação. Ele consiste em uma representação compacta de um conjunto de dados por meio de indexação de uma ou mais dimensões. Um vetor com uma única dimensão consiste em uma sequência de valores, enquanto que em duas dimensões ele pode ser visto como uma matriz, e assim por diante. Segue abaixo um exemplo de declaração de vetores de uma dimensão e de duas dimensões em OPL:

```
int noProcessos[1..3];
float teorEnxofre[1..3][1..2];
```

Uma diferença a ser notada na declaração de vetores em OPL é que, além do índice final, declara-se explicitamente o índice inicial em vez de adotar algum padrão como 0 na linguagem C ou 1 em Pascal. Entre os índices inicial e final, constam dois pontos consecutivos, utilizados em conjunção com os dois índices para formar o domínio de cada dimensão. Isso remete à definição da estrutura de dados utilizada para representar explicitamente um domínio em OPL, o *range*, com o qual é possível tornar a declaração acima mais clara conforme apresentado abaixo:

```
range dRefinarias = 1..noRefinarias;
range dDerivados = 1..noDerivados;
int noProcessos[dRefinarias];
float teorEnxofre[dDerivados][dRefinarias];
```

Como pode ser observado, os exemplos sendo apresentados possuem uma convenção sintática peculiar – por exemplo, o prefixo *no* para quantidades inteiras e o prefixo *d* para o domínio variando

entre 1 e a quantidade determinada. Apesar de não representar nenhuma regra estrita, essa convenção partiu da observação do autor de algumas nomenclaturas comuns de variáveis nos modelos em OPL e foi complementada com outras convenções que facilitaram o desenvolvimento de modelos mais elaborados ao longo do tempo.

A.3.5 Atribuições Básicas a Elementos de Vetores

Da mesma forma que se faz para variáveis de tipos primitivos, é possível atribuir os valores de um vetor de variáveis pelas formas mencionadas anteriormente: estaticamente, delegando a alguma fonte de entrada de dados ou por meio de blocos de programação imperativa. Além disso, tanto em um caso como no outro também é possível usar expressões para uma atribuição declarativa mais concisa, conforme será visto mais adiante.

No trecho abaixo, os mesmos vetores dos exemplos anteriores são declarados em conjunto com uma atribuição estática:

```
int noProcessos[dRefinarias] = [ 10, 15, 20];
float teorEnxofre[dDerivados][dRefinarias] = [
    [ 1/1000, 1/1000000],
    [ 1/1200, 1/1200000],
    [ 1/1500, 1/1500000]
];
```

Abaixo, o primeiro desses vetores é representado com delegação de atribuição para alguma fonte de dados externa:

```
int peso[dCarros] = ...;
```

E a seguir o trecho na fonte de dados:

```
peso = [ 10000, 15000, 20000];
```

Para realizar atribuições a valores de vetores utilizando ILOG Script, é possível empregar estruturas de controle de fluxo idênticas às encontradas em Javascript. Temos, por exemplo, as primitivas *if*, *for*, *while* etc, bem como outros operadores da sintaxe de Javascript. Segue um exemplo usando *for*:

```
int peso[dCarros];
execute {
    for (var ic in dCarros) {
        peso[ic] = 5000*(ic+1);
    }
}
```

Como pode ser visto no exemplo acima, é possível declarar uma variável sem tipologia utilizando a palavra reservada *var* e iterar sobre um domínio com a palavra reservada *in* no ILOG Script.

Uma variável declarada em um bloco é acessível por outros blocos de script, de forma que declarar duas vezes o mesmo identificador como variável em blocos distintos resultará em erro de interpretação quando a segunda declaração for atingida pelo interpretador. Dessa forma, outra boa prática observada consiste em sempre utilizar identificadores diferentes para evitar que um pequeno descuido leve a um erro de interpretação nas linhas finais de um modelo, desperdiçando muito

tempo de processamento. Por esse motivo, foi utilizado “*var ic*” no exemplo em vez de “*var i*”, como seria de praxe.

Devido à falta de tipologia na declaração e à documentação inconclusiva, o autor desconhece formas de declarar um vetor nos blocos de script mas não pode afirmar que elas não existam. Ainda sim, é possível enunciar um vetor na parte declarativa do modelo para utilização na parte imperativa, como no exemplo dado.

A.3.6 Expressões Numéricas e Predicados

Para realizar algumas atribuições comuns baseadas em relações conhecidas entre os índices no vetor e o valor a ser atribuído, as formas descritas anteriormente obrigariam o projetista do modelo a inserir vários trechos de programação imperativa no mesmo. No entanto, é possível evitar isso por meio de expressões, com as quais representam-se muitas dessas atribuições de maneira declarativa e muito mais simples. Por exemplo, é possível atribuir um único valor para todas as entradas de uma matriz em sua declaração:

```
int n = ...;
int matrizNula[1..n][1..n] = 0;
```

Torna-se uma atribuição declarativa mais dinâmica através da utilização de variáveis identificando os índices do vetor para atribuir valores individualmente para cada elemento do vetor. Por exemplo, vejamos a operação de transposição da matriz M no trecho de modelo abaixo:

```
int n = ...;
int M[1..n][1..n] = ...;
int transpostaM[i in 1..n][j in 1..n] = M[j][i];
```

É importante ressaltar que as atribuições em expressões sobre valores são feitas com base nos próprios valores e não por referência. Dessa forma, se a matriz M for alterada após a criação da matriz *transpostaM*, tais alterações não refletirão nessa segunda matriz.

Em OPL, é possível construir expressões numéricas e booleanas, sendo as últimas normalmente conhecidas como predicados. Nessas expressões, podemos nos valer de variáveis representando índices do vetor recebendo uma atribuição. Por exemplo, é possível reescrever a atribuição apresentada na seção anterior com bloco imperativo em uma única linha declarativa por meio de uma expressão:

```
int peso[c in dCarros] = 5000*(c+1);
```

Nesse exemplo, uma expressão numérica atribui um valor para cada elemento do vetor. Ao contrário do que ocorre para os blocos de programação imperativa, o escopo dessas variáveis sobre os índices de um vetor é restrito à expressão de atribuição e o identificador do índice pode ser utilizado em outros contextos sem nenhum problema.

Para expressões numéricas com variáveis e constantes inteiras, é possível utilizar operadores conhecidos de outras linguagens como $+$, $-$, $*$, *div*, *mod* (ou $\%$). Os dois últimos operadores podem parecer estranhos a alguns leitores em razão da nomenclatura adotada. Eles representam, respectivamente, as operações para obtenção de quociente e de resto em uma divisão inteira. Além desses operadores, a função *abs* pode ser utilizada para obter o valor absoluto do inteiro dado como argumento e o termo *maxint* é utilizado para representar a constante correspondendo ao maior inteiro possível em uma variável do tipo *int*.

Em expressões numéricas envolvendo alguma variável ou constante de ponto flutuante, conta-se com os operadores $+$, $-$, $*$, $/$ e a palavra reservada *infinity* para representar simbolicamente uma quantidade infinita. Em OPL, o operador $/$ é utilizado exclusivamente para divisão de ponto flutuante, de forma que o resultado de uma divisão de inteiros com $/$ representará sempre o valor da divisão em ponto flutuante desses inteiros e não a respectiva divisão inteira.

Para introduzir uma forma de expressão numérica válida e extremamente útil para valores inteiros e de ponto flutuante, apresentaremos outro exemplo de manipulação de matrizes. No trecho de modelagem abaixo, temos a instanciação de uma matriz identidade:

```
int identidade[i in 1..n][j in 1..n] = ((i==j)?(1):(0));
```

No exemplo anterior, observamos a utilização do operador ternário com a mesma sintaxe e semântica encontrada em C e outras linguagens: a expressão numérica $((X)?(Y):(Z))$. Ela é composta por uma expressão booleana X e duas expressões numéricas Y e Z , sendo seu resultado igual ao de Y se X for verdadeira e igual a Z caso contrário. Dessa forma, no exemplo, conseguimos que toda entrada da diagonal principal tenha valor 1 e as demais entradas tenham valor 0.

Expressões de valor booleano também são chamadas de predicados. É possível construir predicados utilizando expressões numéricas, variáveis e constantes numéricas relacionadas pelos operadores de comparação $==$, $!=$, $>$, $<$, $>=$ e $<=$. Além disso, predicados podem ser compostos por outros predicados com base nos operadores booleanos de conjunção ($\&\&$), disjunção ($\|\|$) e negação ($!$) à semelhança do que existe em C e outras linguagens similares.

Expressões booleanas também podem ser vistas como expressões numéricas, sendo cada predicado verdadeiro equivalente à constante *true* e ao valor 1, e cada predicado falso equivalente à constante *false* e ao valor 0. Assim, poderíamos reescrever o exemplo anterior de maneira ainda mais compacta – ainda que de modo um tanto quanto obscuro – como abaixo:

```
int identidade[i in 1..n][j in 1..n] = ( i == j );
```

Uma última ressalva a ser feita com relação à equivalência entre expressões numéricas e predicados é que, ao contrário de outras linguagens nas quais 0 é falso e qualquer outro valor é verdadeiro, isso não é sempre verdade em OPL. Dessa forma, no projeto de um modelo, o projetista deve estar ciente de que não pode esperar mais do que 0 ser equivalente a falso e 1 a verdadeiro. De uma maneira geral, o que podemos concluir dessa observação é que não é boa prática utilizar valores que não 0 ou 1 no lugar de expressões booleanas.

Como em outras linguagens, operadores numéricos e lógicos inserem-se em uma relação de precedência de forma a resolver boa parte das ambiguidades possíveis na interpretação das expressões válidas na linguagem. Na OPL, essa relação é idêntica à de muitas outras linguagens. Por exemplo, a multiplicação é precedente à soma, que é precedente a qualquer operador lógico. Na dúvida, ou para garantir a semântica da expressão sendo construída a despeito de grandes mudanças na linguagem, parênteses podem ser utilizados à vontade.

A.3.7 Operadores de Agregação e Funções Sobre Vetores

Para concluir a apresentação sobre expressões e predicados, restam os operadores de agregação e as funções sobre vetores. Os operadores de agregação representam operações muito comuns no tratamento dos dados de problemas combinatórios. Alguns deles operam sobre os termos de um

conjunto indexado numérico, sendo eles *min*, *max*, *sum* e *prod*. Eles são usados, respectivamente, para obter o valor mínimo, o valor máximo, a somatória e o produto de um conjunto de termos. Vide, por exemplo, a atribuição a um vetor usando a expressão *max* sobre as linhas de uma matriz *M* e de uma variável usando a expressão *sum* sobre os elementos do vetor resultante:

```
int n = ...;
int M[1..n][1..n] = ...;
int maxLinha[i in 1..n] = max(j in 1..n) M[i][j];
int somaMaximos = sum(i in 1..n) maxLinha[i];
```

Além de operadores de agregação para expressões numéricas, também existem operadores de agregação para predicados, como *or* e *and*. O operador *or* representa um predicado com valor verdadeiro se algum termo do conjunto indexado for verdadeiro, enquanto que o *and* é verdadeiro se todos os termos do conjunto indexado forem verdadeiros.

Um item importante a ser observado é o comportamento desses operadores quando o conjunto for vazio: *min* resulta em *infinity*, *max* em *-infinity*, *prod* é 1, *sum* é 0, *or* é falso e *and* verdadeiro.

Os termos dos conjuntos utilizados nos operadores de agregação podem ser outros operadores ou expressões da mesma natureza que o operador, seja ele numérico ou booleano. No exemplo abaixo sobre como calcular o quadrado de uma matriz *M*, é ilustrado um caso no qual cada termo de um somatório consiste em uma expressão numérica sobre o índice *k*:

```
int quadradoM[i in 1..n][j in 1..n] = sum(k in 1..n) M[i][k]*M[k][j];
```

Em alguns casos, é possível substituir a composição de operações de agregação por uma única operação com mais índices. No exemplo abaixo, é definido um somatório sobre termos da *M* empregando dois índices distintos para somar apenas o valor dos elementos abaixo de sua diagonal principal:

```
int somaElementosInferioresM = sum(i in 1..n, j in 1..n : i>j) M[i][j];
```

Para se somar apenas os elementos abaixo da diagonal principal, uma condição foi imposta além da especificação do domínio dos índices. Essa condição delimita quais duplas de valores devem ser consideradas utilizando o símbolo *:* acrescido de um predicado que define um critério de filtragem.

Quando especificamos um subconjunto do produto cartesiano dos domínios dos índices para utilizar um operador de agregação, estamos definindo um parâmetro formal. Mais adiante, serão vistas outras situações na linguagem em que parâmetros formais podem ser empregados para representar conjuntos. Vale ressaltar que, até a presente versão do OPL, parâmetros formais não suportam a utilização interna de operadores de agregação, isto é, um operador de agregação não pode ser utilizado dentro do índice de outro operador de agregação.

Em alguns casos especiais, é possível substituir a utilização de operadores de agregação sobre conjuntos indexados por uma função que opere sobre vetores. Por exemplo, para contar o número de ocorrências de um valor em um vetor, pode-se utilizar a função *count*. Essa função recebe um vetor de uma dimensão e um valor como parâmetros para retornar o número de vezes que o valor aparece no vetor. Para calcular o desvio padrão de um vetor inteiro, existe a função *standardDeviation*. Ambas estão apresentadas no exemplo a seguir:

```

int x = ...;
int n = ...;
int vetor[1..n] = ...;
int M[1..n][1..n] = ...;
int elementosVetorIguaisX = count(vetor, x);
int ocorrenciasXporLinhaM[i in 1..n] = count(M[i], x);
float desvioPadraoPorLinhaM[i in 1..n] = standardDeviation(M[i]);

```

Na penúltima linha do exemplo acima, foi utilizada a função *count* sobre um vetor extraído da matriz *M*. Dessa forma, foi contado o número de ocorrências de um valor para cada linha dessa matriz. Tal vetor foi obtido a partir da matriz *M* pela não-indexação de sua última dimensão. Fazer o mesmo de forma tão simples para obter uma coluna de *M* não é possível, e em princípio isso poderia limitar a aplicabilidade de funções sobre vetores, como é o caso de *count*.

Para esses casos, utiliza-se o operador *all* para criar um vetor unidimensional a partir de um parâmetro formal sobre um vetor qualquer de uma ou mais dimensões. Esse operador é necessário porque tais funções não permitem a utilização de parâmetros formais diretamente, como é o caso dos operadores de agregação. Seguem alguns exemplos:

```

int ocorrenciasXporColuna[j in 1..n] = count(all(i in 1..n) M[i][j], x);
int ocorrenciasXnaDiagonal = count(all(i in 1..n, j in 1..n : i==j) M[i][j], x);

```

Na primeira linha do exemplo acima, foi apresentada uma forma de contar as ocorrências de um valor na coluna de uma matriz por meio da criação de um vetor contendo apenas os elementos daquela coluna da matriz. Na segunda linha, foi apresentada uma forma de calcular a ocorrência de um valor na diagonal principal dessa matriz.

O operador *all* também pode ser utilizado para atribuição em um vetor, isto é, podemos extrair a coluna de uma matriz para um vetor e até mesmo unir o resultado de algumas dessas operações para obter um vetor maior por meio da função *append*, como nos exemplos abaixo:

```

int coluna2deM[1..n] = all(i in 1..n) M[i][2];
int colunas2e5deM[1..2*n] =
    append( all(i in 1..n) M[i][2], all(i in 1..n) M[i][5] );

```

Na primeira linha do exemplo, foi criado um vetor com base na segunda coluna de uma matriz *M*. Na segunda linha, foi criado um vetor com o dobro do tamanho do anterior por meio da junção de dois vetores contendo, cada um, uma coluna de *M*.

As funções *count* e *standardDeviation* não são os únicos exemplos de expressões que utilizem vetores de uma dimensão como argumento. Quando forem tratadas as restrições especializadas e predicados para escalonamento, observaremos outras funções com essa característica, expandindo o leque de possibilidades de aplicação do operador *all*.

A.3.8 Estruturas de Dados Avançadas: Tuplas e Conjuntos

Devido ao propósito introdutório desse texto, não se entrará em muitos detalhes sobre as estruturas de dados mais avançadas disponíveis em OPL: as tuplas e os conjuntos. Entretanto, não é possível ignorar as tuplas e conjuntos por completo sem algum prejuízo quando o texto atingir a parte sobre escalonamento.

Uma tupla consiste em uma composição de variáveis a partir da qual se define um tipo através da palavra reservada *tuple*, possibilitando a declaração de variáveis prefixando-as com o tipo dado à tupla, como no exemplo a seguir:

```
tuple ProcessoConversao {
    float entrada;
    float saida;
}
ProcessoConversao conversoes[1..3] = [
    <3.5, 2.9>,
    <4.0, 3.0>,
    <4.5, 3.1>,
];
```

Com base no exemplo acima, o acesso ao valor de entrada do primeiro processo é dado pela expressão “*conversoes[1].entrada*”.

Os conceitos apresentados para a atribuição por expressão também são válidos para tuplas, de forma que é possível criar expressões indexadas para especificar uma tupla para cada índice diferente de um vetor, como na adaptação do exemplo anterior feita abaixo:

```
ProcessoConversao conversoes[i in 1..3] =
    < 3 + 0.5*i, 2.8 + 0.1*i>;
```

Um conjunto consiste em um agrupamento de valores de algum tipo que, ao contrário dos vetores, não possui dimensões de indexação. No exemplo abaixo, é declarado um conjunto de inteiros sem qualquer indexação explícita:

```
{ int } pesos = {10000, 15000, 20000};
```

Claramente, existe uma vantagem expressiva pela utilização de tuplas e conjuntos pelo fato que as propriedades de uma entidade do modelo possam ser representadas compactamente tanto na declaração quanto na atribuição dos dados. Dessa forma, é recomendado que o leitor interessado em um uso prolongado da ferramenta se aprofunde no estudo dessas estruturas para projetar modelos mais estruturados.

A.4 Modelagem de Problemas Combinatórios

Até esse ponto do texto foram tratados apenas elementos relevantes ao pré-processamento do problema: escolha do resolvidor, definição das estruturas de dados, atribuição dos dados e manipulação inicial dos dados. Com esses elementos em mãos e apropriadamente descritos no modelo, podemos passar à formulação do problema para ser resolvido pelo resolvidor. Um arquivo de modelo em OPL geralmente consiste em um único problema a ser resolvido. Após a descrição do problema ser interpretada e a etapa de resolução concluída, passa-se à etapa de pós-processamento, na qual novas manipulações de dados tomam lugar e na qual, geralmente, encontram-se rotinas de impressão para apresentar os resultados obtidos com o modelo da maneira que melhor interessar ao modelador.

Dessa forma, passaremos agora à definição do problema e quaisquer outras informações relevantes ao pós-processamento.

A.4.1 Variáveis de Decisão de Tipos Primitivos

O primeiro passo para expressar um problema decisório ou de otimização consiste na declaração de suas variáveis de decisão. A solução de um problema consistirá em uma atribuição válida de valores a tais variáveis. A escolha dessa atribuição dependerá de outros fatores que serão vistos adiante, como o conjunto de restrições e a função objetivo.

A palavra reservada *dvar* é empregada antes do tipo da variável para identificar que a variável sendo declarada é de decisão. Dos tipos primitivos, é possível declarar variáveis de decisão usando *int* e *float*, além da opção de acrescentar *+* ao nome do tipo para que a variável seja não-negativa, isto é, maior ou igual a 0. Além disso, podemos substituir *int* por *boolean* se os valores válidos para a variável forem apenas 0 e 1. Convém observar que as variáveis do tipo *float* não serão detalhadas devido ao fato de não serem aceitas em modelos empregando o resolvidor CP. Isso se deve à premissa de que as variáveis tenham domínio discreto e finito em problemas abordados com programação por restrições. É possível declarar esse domínio explicitamente utilizando a palavra reservada *in* após o nome da variável seguida do domínio da mesma. Se um domínio não for especificado, são utilizados os limites de uma variável inteira na arquitetura adotada pelo resolvidor.

Abaixo são apresentados alguns exemplos de declarações de variáveis de decisão inteiras na linguagem OPL:

```
dvar int escolha in 1..5;
dvar int+ quantidade;
```

Na primeira declaração do exemplo, temos uma variável inteira restrita a um domínio através da utilização do operador *in*. Na segunda declaração, temos uma variável inteira não-negativa sem restrição de domínio, ficando subentendido ao resolvidor que o domínio da variável é $0..maxint$.

Os conceitos apresentados acima são naturalmente aplicáveis à declaração de vetores de variáveis de decisão. Nesse caso, expressões sobre os índices podem ser empregadas para definir um domínio diferente a cada variável do vetor, como pode ser visto no exemplo a seguir:

```
int noProcessos = ...;
range dProcessos = 1..noProcessos;
int nivelMaximoProcesso[dProcessos] = ...;
dvar int nivel[p in dProcessos] in 0..nivelMaximoProcesso[p];
```

Ao contrário das outras variáveis, as variáveis de decisão de um modelo normalmente não possuem atribuição em sua declaração, apesar disso ser possível em casos mais elaborados fora do escopo desse texto. No entanto, em algumas situações o modelo sendo desenvolvido pode deixar clara a necessidade de manipulação de algumas variáveis dependentes de valores de variáveis de decisão. Tais variáveis poderiam ser definidas como variáveis de decisão e essa relação poderia ser expressa por meio de restrições. No entanto, existe uma alternativa em OPL para evitar uma quantidade excessiva de variáveis de decisão no modelo: as expressões de decisão, descritas a seguir.

A.4.2 Expressões de Decisão

Uma expressão de decisão pode ser entendida como uma variável que possui o identificador *dexpr* antes de seu tipo na declaração. Ela sempre recebe uma atribuição na forma de expressão, que pode envolver constantes, variáveis, variáveis de decisão e outras expressões de decisão. Ao contrário do que ocorre com as variáveis de decisão, não existe restrição sobre expressões de decisão envolvendo o

tipo *float* em modelos utilizando o resolvidor CP. Além disso, as expressões de decisão possibilitam reduzir o espaço de busca ao estabelecer uma relação entre diferentes parâmetros do sistema sendo modelado sem necessariamente defini-los todos como variáveis de decisão. Dessa forma, é possível definir uma hierarquia entre decisões e facilitar o processo de resolução do problema.

Abaixo temos um exemplo da utilização de expressões de decisão sobre o vetor de variáveis de decisão previamente definido:

```
int maxNivelProcesso = max (p in dProcessos) nivelMaximoProcesso[p];
range dNiveis = 0..maxNivelProcesso;
float custoExecucaoProcessoNivel[p in dProcessos][dNiveis] = ...;
dexpr float custoExecucaoTotal =
    sum(p in dProcessos, n in dNiveis)
        ( nivel[p]==n ? ( custoExecucaoProcessoNivel[p][n] ) : (0) );
```

À primeira vista, pode parecer que a expressão de decisão do exemplo acima poderia ser reescrita de maneiras mais compactas. Dessa forma, vamos analisar como se suporia que esse trecho poderia ter sido progressivamente compactado para ilustrar as limitações inerentes.

Primeiro, poderíamos supor que o predicado do operador condicional pudesse ser colocado dentro do somatório, criando um parâmetro formal mais restrito. Entretanto, isso não é possível e esse é um problema bastante comum nos primeiros modelos que se faz em OPL: variáveis e expressões de decisão não podem constar em parâmetros formais antes do pós-processamento, isto é, enquanto essas variáveis não possuem um valor fixo definido. É mencionado em muitos pontos do manual oficial da linguagem que certos operadores e predicados não suportam aplicação como meta-restrições, o que significa que não se pode empregar nessas funções nenhum argumento baseado em variáveis de decisão ou expressões de decisão. Nesse caso, não é possível escrever “*sum(p in dProcessos, n in dNiveis : nivel[p]==n)*” porque “*nivel[p]*” representa uma variável de decisão e a expressão não pode ser interpretada antes dessa variável ter seu valor decidido pela resolução.

Segundo, poderíamos supor uma compactação ainda maior do modelo usando as variáveis de decisão para indexação, reescrevendo o somatório como “*sum(p in dProcessos) (custoExecucaoProcessoNivel[p][nivel[p]])*”. Entretanto, a limitação mencionada anteriormente também se estende à utilização de variáveis prefixadas por *dvar* e *dexpr* para o acesso a índices de vetores em expressões de decisão ou mesmo em restrições. Afinal, isso significaria usar uma variável de decisão como índice de acesso a uma matriz para gerar o valor de uma expressão de decisão durante a execução, algo que o interpretador não é capaz de fazer por não ter um valor definido às variáveis de decisão e expressões.

Além dessas questões, outra precaução a ser tomada com relação a variáveis de decisão e expressões de decisão se relaciona ao fato de que elas são variáveis sem valor definido para manipulação antes da resolução do problema, que ocorre após o bloco de restrições – não tratado até esse ponto do texto. Dessa forma, instruções imperativas que tentem acessar o valor dessas variáveis antes do pós-processamento podem ter comportamento inesperado. No mais, essas variáveis podem ser utilizadas indistintamente de outros tipos de variáveis na definição da função objetivo e das restrições do modelo, como veremos adiante.

A.4.3 Sobre a Impossibilidade de Usar Variáveis *float* em CP

Visto que os valores que podem ser atribuídos a uma variável de ponto flutuante também representam um conjunto discreto devido à limitação de precisão em máquina, poder-se-ia argumentar que também seria possível utilizar uma variável de decisão em ponto flutuante em programação por restrições, a despeito da dificuldade de implementação de algoritmos de resolução baseados nesse fato.

Um bom argumento para justificar a inexistência dessa possibilidade tanto em OPL como em outras linguagens é o de que o objetivo da programação por restrições consiste na resolução de problemas combinatórios e não de programação matemática, para os quais outros resolvidores possuem melhor desempenho. Essa especialização dos algoritmos desenvolvidos para a resolução de problemas combinatórios em modelos de programação por restrições torna a técnica bastante competitiva para tais problemas, mas implica em trabalhar sobre domínios com características bastante peculiares. No caso de um intervalo de valores representáveis por uma variável de ponto flutuante, isso exigiria o projeto de todo um novo conjunto de algoritmos mais complexos e que se prestariam, em muitos casos, a resolver problemas que podem ser tratados de maneira mais elegante por outras técnicas. Ainda sim, é possível que algumas propriedades da solução de um problema combinatório tenham valores em ponto flutuante sem que seja necessário representar tais propriedades como variáveis de decisão do modelo, como vimos para as variáveis de expressão de decisão.

A.4.4 Função Objetivo

Quando trabalhamos com um problema de otimização, desejamos não apenas obter uma solução que satisfaça às restrições do modelo – como será visto adiante. Além disso, podemos querer que a solução encontrada seja comparada com outras soluções possíveis de acordo com algum critério quantitativo para que a solução final seja a melhor dentre as encontradas durante a resolução. A essa medida quantitativa damos o nome de função objetivo, que deve ser maximizada ou minimizada, conforme o problema e a modelagem dados.

Em OPL, uma função objetivo é declarada após as variáveis de decisão e antes do bloco de restrições, sendo possível declarar apenas uma função objetivo por modelo. Ela é caracterizada por alguma das palavras reservadas *minimize* ou *maximize*, dependendo do problema ser de minimização ou de maximização. Após a palavra reservada, a função objetivo declarada segue as mesmas regras de composição que as expressões de decisão, sendo boa prática em grandes projetos definir a função objetivo apenas sobre uma função linear de expressões de decisão, isto é, a soma do produto de expressões de decisão por constantes. Embora seja comum que problemas sejam de otimização, isso nem sempre é verdade e ela pode não ser encontrada no modelo – caso em que se trata de um problema decisório. Nesse caso, o resolvidor interrompe a busca quando encontra a primeira solução, visto que não há razão para continuar a procura se não houver qualquer métrica que justifique que a solução encontrada não seja a mais adequada.

Segue abaixo um exemplo de função objetivo de minimização sobre a expressão de decisão declarada anteriormente:

```
minimize custoExecucaoTotal;
```

A.4.5 Restrições Básicas Sobre Tipos Primitivos

Uma vez definidas as variáveis de decisão e a função objetivo, resta a definição do conjunto de restrições para completar a modelagem do problema. Essa definição é feita dentro de um bloco prefixado por *subject to*, como a seguir:

```
subject to {
    // Restrições do modelo.
}
```

Dentro desse bloco, serão especificados apenas predicados. Em termos de programação por restrições, esses predicados definirão um problema de satisfabilidade, isto é, um problema para o qual a solução consistirá em uma atribuição de valores às variáveis de decisão de tal forma que todos os predicados do bloco de restrições estejam satisfeitos. Se não houver configuração com a qual isso seja possível, diz-se que o problema é insatisfatível, não havendo solução.

Conforme mencionado anteriormente, esses predicados podem ser definidos em termos de expressões numéricas envolvendo operações aritméticas inteiras e de ponto flutuante, operadores de agregação ou outras funções como *abs* e *count* em associação a comparadores, como no exemplo abaixo:

```
dvar int+ criancas;
dvar int+ menores;
dvar int+ adolescentes;
subject to {
    criancas == menores - adolescentes;
}
```

No exemplo acima, foi definida uma relação entre as variáveis de decisão *criancas*, *menores* e *adolescentes*.

A partir dos próximos exemplos, as restrições serão geralmente apresentadas sem o bloco externo demarcado com *subject to*, por simplicidade.

Em muitos problemas de otimização, encontram-se conjuntos de restrições definidas sobre índices. Para fazer o mesmo em OPL, utiliza-se a palavra reservada *forall* seguida de um par de parênteses dentro do qual um parâmetro formal é definido e um bloco para a definição de restrições que podem contar com os índices do parâmetro formal. No exemplo a seguir, ilustra-se o conceito com o vetor de variáveis de decisão definido na seção anterior:

```
forall(p in dProcessos : p<noProcessos) {
    nivel[p] >= nivel[p+1];
}
```

Com a restrição acima, forçamos que o vetor esteja em ordem não-crescente de valores, i.e., o valor de *nivel[p]* nunca é menor que o de *nivel[p+1]*. Observe que os limites de indexação são observados graças a uma condição no parâmetro formal que impede o estouro do vetor.

Convém frisar novamente que, na definição de parâmetros formais dentro do bloco de restrições, não se pode utilizar variáveis de decisão. Apesar disso, existe uma construção de implicação lógica que aceita variáveis de decisão. Ela consiste de dois predicados separados por $\Rightarrow$, significando que a veracidade do primeiro implica na veracidade do segundo para que a restrição seja satisfeita. Segue um exemplo baseado nas mesmas variáveis e em declarações da seção anterior:

```
forall(p in dProcessos : p<) {
    ( nivel[p]==nivelMaximoProcesso[p]) =>
        ( nivel[p+1] < nivel[p] );
}
```

A.4.6 Restrições Especializadas Sobre Tipos Primitivos

Além das restrições possíveis pelo rol de elementos descritos até agora, em CP ainda é possível utilizar algumas restrições especializadas. Essas restrições representam ocorrências comuns em problemas combinatórios, permitindo modelar vários problemas com uma quantidade menor de variáveis e que o resolvidor trabalhe de maneira mais eficiente em tais problemas por existirem algoritmos de redução de domínios específicos para essas restrições. São elas: *allDifferent*, *allMinDistance*, *inverse*, *lex* e *pack*.

O predicado *allDifferent* é verdadeiro se todos os elementos do vetor inteiro unidimensional dado como argumento forem diferentes entre si. É possível, por exemplo, expressar o seguinte:

```
allDifferent(niveis);
```

Nesse exemplo, expressa-se que cada nível deverá ser diferente dos demais. Com *allDifferent*, evita-se ter que especificar uma restrição de desigualdade para cada par de elementos distintos do vetor, reduzindo bastante o tamanho da modelagem e facilitando a resolução do problema.

O predicado *allMinDistance* representa uma generalização desse conceito, recebendo como segundo argumento um valor inteiro que especifica a diferença absoluta mínima entre quaisquer dois elementos do vetor. Dessa forma, o valor 0 torna o predicado trivialmente verdadeiro e o valor 1 torna o predicado equivalente ao anterior, isto é, é possível expressar a mesma regra que foi modelada anteriormente da seguinte forma:

```
allMinDistance(niveis, 1);
```

O predicado *inverse* cria uma relação de bijeção entre dois vetores. Por exemplo, se forem dados como argumentos os vetores x e y , para qualquer índice i vale que, se $x[i] = j$, então $y[j] = i$. O predicado *lex* também recebe dois vetores inteiros como argumento, sendo ambos do mesmo tamanho, e é verdadeiro se o primeiro vetor for lexicograficamente menor que o segundo vetor, isto é, se o elemento de cada posição do primeiro for menor ou igual ao do segundo.

Por fim, o predicado *pack* representa boa parte das relações existentes em um problema da mochila com m mochilas e n itens – i.e., particionar n elementos de acordo com o limite de peso admitido em cada uma das m partições. Em sua versão mais simples, ele envolve três vetores inteiros unidimensionais, o primeiro com dimensão m e os outros com dimensão n . Um quarto argumento opcional consiste em um valor inteiro. O primeiro vetor consiste no peso total de cada mochila, o segundo vetor na determinação da mochila a qual cada item foi atribuído e o terceiro vetor ao peso de cada item. O argumento opcional representa a quantidade de mochilas com algum item atribuído. O terceiro vetor é constante, mas os dois primeiros podem ser ambos de variáveis de decisão – mas não apenas um deles e o outro não. Se os dois primeiros vetores forem variáveis de decisão, o quarto argumento pode ser uma variável de decisão também.

A seguir, é apresentado um exemplo envolvendo a atribuição de passageiros a carros e a relação do peso dos passageiros com o peso total em cada carro para ilustrar como o predicado *pack* opera:

```

int noCarros = ...;
range dCarros = 1..noCarros;
int noPassageiros = ...;
range dPassageiros = 1..noPassageiros;
int pesoPassageiro[p in dPassageiros] = ...;
dvar int pesoNoCarro[c in dCarros];
dvar int carroComPassageiro[p in dPassageiros] in dCarros;
dvar int carros in dCarros;
subject to {
    pack(pesoNoCarro, carroComPassageiro, pesoPassageiro, carros);
}

```

No exemplo, temos dois vetores de variáveis de decisão e uma variável de decisão inteira, todos completamente interdependentes: o modelo procura determinar em qual carro cada passageiro vai, o que determina o peso total de passageiros nesse carro de acordo com o peso de cada um deles e também quantos carros têm pelo menos um passageiro atribuído.

A.4.7 Execução do Resolvedor e Configurações Básicas do CP

Uma vez realizado o pré-processamento e processada a definição do problema de otimização, o interpretador realiza o passo geralmente mais demorado, que consiste na resolução do problema descrito pelo modelo. Na saída visualizada na tela, é possível observar informações gerais sobre o modelo, o consumo de memória e tempo do processo de busca. Durante a busca, a períodos regulares ocorre uma saída sobre novas soluções encontradas ou sobre o número de variáveis sem atribuição. Acompanhando a variação desse valor, com o tempo podemos inferir a dificuldade de resolução do modelo para uma dada instância do problema e compreender como modelar o problema de forma que sua resolução seja facilitada.

O processo de busca ocorre apenas se o resolvedor não inferir que o problema é infactível e pode durar até que uma solução seja encontrada no caso de não haver função objetivo, até a melhor solução ser encontrada se houver uma função objetivo, até todo o espaço de busca ser examinado sem que nenhuma solução seja encontrada ou – caso essas situações não ocorram facilmente – até um determinado limite de tempo ou número de iterações ser atingido. Tais limites representam parâmetros de configuração do resolvedor. Para surtir efeito no processo de busca, tais parâmetros devem ser alterados antes do bloco de restrições do modelo, isto é, no pré-processamento. A parametrização do resolvedor pode ser realizada por meio de instruções específicas nos blocos de programação imperativa antes do bloco de restrições, como no exemplo abaixo:

```

execute {
    cp.param.TimeLimit = 60;
    cp.param.LogPeriod = 1000;
    cp.param.randomSeed = 42;
}

```

Nesse exemplo, foram alterados três parâmetros do resolvedor: estabelecemos que a busca deva terminar em, no máximo, 60 segundos; que uma linha de estado deve ser impressa na tela a cada 1000 iterações realizadas; e atribuímos uma nova semente aleatória ao algoritmo. A importância da definição e variação das sementes empregadas está relacionada ao emprego de critérios de desempate na escolha dos próximos passos da busca: quando o critério de avaliação tem um empate técnico

entre algumas opções – como na escolha da próxima variável ou valor a atribuir –, é possível arbitrar a escolha com alguma aleatoriedade, de forma que pode ser interessante rodar o resolvidor várias vezes com sementes distintas.

A fixação e posterior mudança no valor de uma semente para o modelo pode ser útil quando, por qualquer motivo, o resolvidor não gera uma solução para uma instância apesar de gerar para outras instâncias muito parecidas. Se desejarmos avaliar a robustez no desempenho de um modelo, nosso interesse será em utilizar sementes variadas em execuções sucessivas do modelo sobre uma mesma instância para avaliar como isso altera os resultados obtidos. Nesse tipo de avaliação, a variação na qualidade da solução obtida ou no tempo de execução para atingir soluções em um determinado patamar dá um bom indício da robustez do modelo para o conjunto de instâncias esperado.

A.4.8 Manipulação da Solução e Impressão dos Resultados no Pós-Processamento

Após a resolução do problema do modelo, é possível inserir novas declarações de vetores e blocos de programação imperativa; de forma a manipular os resultados obtidos para as variáveis de decisão e expressões de decisão, obtendo assim informações de interesse ao projetista do modelo. No pós-processamento, as variáveis de decisão são vistas como variáveis comuns nas expressões e podem ser utilizadas para atribuições a novos vetores. É nesse ponto que geralmente encontram-se blocos de programação imperativa com rotinas de impressão, apesar de que é possível empregar rotinas de impressão no pré-processamento. Essas rotinas de impressão consistem nas funções imperativas *write* e *writeln*, com as quais podemos manipular texto da mesma forma como fazemos em Java, sendo a diferença entre ambas as funções que a segunda pula de linha automaticamente após a impressão de seu conteúdo.

No exemplo a seguir, ilustra-se um exemplo pela geração de um vetor e impressão da informação de uma variável de decisão empregada em exemplos anteriores:

```
int processosPorNivel[n in dNiveis] = count(nivel, n);
execute {
    writeln("Niveis definidos a algum processo :");
    for (var ini in dNiveis) {
        if (processosPorNivel[ini]>0) {
            writeln("Nivel "+ini);
        }
    }
}
```

Convém observar que o exemplo poderia ainda ser reescrito com menos linhas se, em vez de declarar um vetor, fosse utilizada diretamente a expressão “*count(nivel,ini)*” na condição do *if*.

A.4.9 Um Exemplo de Problema Combinatório Modelado em OPL

Para não concluir a apresentação dos conceitos elementares da linguagem OPL para a modelagem de problemas combinatórios sem um exemplo completo para resolver um problema, segue um exemplo de Lustig e Puget (2001) baseado no problema das n rainhas. Nesse problema, interessa-nos posicionar n rainhas em um tabuleiro $n \times n$ sem que nenhuma delas possa atacar outra:


```

1  using CP;
2
3  int n = ...;
4  range dN = 1..n;
5  dvar int rainhaLinha[dN] in dN; // Define onde está a rainha de cada linha.
6
7  subject to {
8      // Duas rainhas não podem ficar na mesma linha:
9      allDifferent(rainhaLinha);
10
11     // Duas rainhas não podem ficar na mesma diagonal:
12     //   Verifica se quaisquer duas rainhas pertencem à mesma diagonal.
13     //   Índice das rainhas é ordenado para evitar verificações desnecessárias.
14     forall (i in dN, j in dN : i < j) {
15
16         // Relação da diagonal principal: coluna + linha = constante
17         (rainhaLinha[i] + i) !=(rainhaLinha[j] + j);
18
19         // Relação da diagonal secundária: coluna - linha = constante
20         (rainhaLinha[i] - i) !=(rainhaLinha[j] - j);
21     }
22 }
23
24 execute {
25     writeln("Resultado:");
26     for (var i in dN) {
27         writeln("Linha " + i + " - Coluna " + rainhaLinha[i]);
28     }
29 }

```

Para esse exemplo e $n = 10$, obtemos a seguinte resposta por meio de impressões na tela:

Resultado:

```

Linha 1 - Coluna 5
Linha 2 - Coluna 9
Linha 3 - Coluna 4
Linha 4 - Coluna 7
Linha 5 - Coluna 8
Linha 6 - Coluna 6
Linha 7 - Coluna 10
Linha 8 - Coluna 1
Linha 9 - Coluna 3
Linha 10 - Coluna 2

```

Ainda com relação ao exemplo apresentado, uma alternativa ao parâmetro formal “ i in dN , j in $dN : i < j$ ” poderia ser “ i in dN , j in $1..i$ ”. Entretanto, tais ganhos de processamento devem ser avaliados com cautela, pois podem tornar o código menos legível. Uma vez processado o modelo para que o resolvidor proceda com a busca por meio de uma representação mais simples, não haverá diferença entre as duas representações.

Pode ser de interesse do leitor saber como obter soluções alternativas do problema. Para tanto, é necessário um controle externo que solicitasse novas soluções do resolvidor. Isso é feito por meio de um bloco imperativo declarado com a palavra reservada *main* em vez de *execute* no arquivo de modelo. Apesar de não tratarmos o assunto nesse texto introdutório, é importante que o leitor saiba da existência desse recurso.

A.5 Modelagem de Problemas de Escalonamento

Um problema de escalonamento consiste na programação da execução de um conjunto de atividades ao longo de um período de tempo. Nessa seção, apresentaremos estruturas e especializações interessantes para a modelagem de problemas de escalonamento em OPL, de forma similar ao que foi feito anteriormente para problemas combinatórios. Esses elementos adicionais não descartam tudo o que foi visto até o momento para problemas combinatórios, mas servem de complemento para a obtenção de um grau maior de expressividade.

A.5.1 Problemas de Escalonamento e a Motivação ao Emprego de CP

Em um problema de escalonamento, estamos interessados em decidir, entre outras coisas, *quando* certos eventos ocorrem e *como* cada evento ocorre. Dessa forma, a solução representa uma escala – ou programação – desses eventos. O conjunto de restrições quanto às possibilidades de construção de uma escala refletem características dos eventos, regras de ordenação entre eles, a utilização de algum tipo de recurso compartilhado, o tempo de separação entre eventos consecutivos de acordo com suas características etc.

Para modelos de programação matemática, as variáveis de decisão associadas a tais eventos geralmente são vetores booleanos, representando se cada atividade ocorre ou é iniciada em cada instante de tempo possível, e podem ter muitas dimensões de acordo com os graus de liberdade e as restrições do problema. Quando a janela de tempo é muito grande e a discretização de intervalos muito fina, a resolução de tais modelos pode se tornar inviável. Em muitos casos, não são conhecidas formas de expressar o problema de uma maneira mais compacta dentro dos limites do que é possível com programação matemática.

Em programação por restrições, entretanto, os limites de flexibilização são maiores. Dessa forma, variáveis de decisão com domínios diversos e algoritmos de resolução especializados dão suporte a formulações que usam menos memória e tempo de processamento em algumas situações. Isso possibilita a obtenção de soluções para instâncias de problemas de escalonamento de grande porte que, de outra forma, não poderiam ser tratados com os recursos computacionais disponíveis.

Os elementos apresentados a seguir servem ao propósito de expressar a modelagem de problemas combinatórios de escalonamento com semântica mais apurada, o que torna sua representação mais compacta e facilita a resolução desses modelos pela utilização de algoritmos especializados nessas estruturas.

A.5.2 Variáveis de Intervalo e Suas Propriedades

Muitas variáveis de um problema de escalonamento podem ser expressas por intervalos, delimitando propriedades como o início e o fim de um evento. Em OPL, é possível expressar variáveis de decisão intervalares por meio do tipo *interval*, como a seguir:

```
dvar interval atividade;
```

O início e o fim de um intervalo representam variáveis de decisão inteiras e interdependentes. Quando declara-se que um intervalo possui um domínio, é definido que o início e o fim do intervalo devem pertencer àquele domínio, como no exemplo abaixo:

```
int pDia = ...;
int uDia = ...;
range dDias = pDia..uDia;
dvar interval escalaAtividade in dDias;
```

Além do início e do fim, uma variável do tipo *interval* também possui outras propriedades interdependentes como tamanho e duração. Tais propriedades também podem ter algum grau de liberdade, situação na qual assumem o papel de variáveis de decisão nas relações entre início e fim. Para acessar explicitamente os valores de início e fim de um intervalo em expressões, são utilizadas as funções *startOf* e *endOf* dando como argumento o intervalo. A duração, que consiste na diferença entre o fim e o início de um intervalo, pode ser obtida com a função *lengthOf*. O tamanho de um intervalo, obtido pela função *sizeOf*, pode ser menor do que sua duração. Isso ocorre quando uma função de intensidade é definida para o intervalo. Apesar de não se tratarem funções de intensidade nesse texto introdutório, o conceito pode ser entendido de maneira fácil pelo seguinte exemplo. Podemos dizer que uma atividade durou uma semana corrida, 7 dias. Entretanto, seu tamanho pode ser de 5 dias, visto que a função de intensidade possua valor nulo para os dias do fim-de-semana.

O tamanho de uma variável também pode ser especificado na declaração da variável. Dessa forma, a definição do domínio de um intervalo em sua declaração pode se tornar mais elaborada do que aquela encontrada em tipos primitivos. No exemplo a seguir, é declarado um intervalo com tamanho fixo:

```
int tAtividade = ...;
dvar interval escalaAtividade
    in dDias
    size tAtividade;
```

Abaixo, tem-se um exemplo da declaração de um intervalo com tamanho variável, utilizando um domínio para delimitar o menor e o maior valor possível para o tamanho do intervalo:

```
int tMinAtividade = ...;
int tMaxAtividade = ...;
range dTamAtividade = tMinAtividade..tMaxAtividade;
dvar interval escalaAtividade
    in dDias
    size dTamAtividade;
```

Um intervalo também pode ser opcional. Nessa condição, é facultada a opção de que ele não ocorra, i.e., representa um evento que não necessariamente precisa ser escalonado numa solução do problema. O primeiro modificador da declaração de um intervalo opcional consiste na palavra reservada *optional* logo após o identificador da variável declarada, como no exemplo abaixo:

```
dvar interval escalaAtividade optional;
```

Quando se declara um vetor de variáveis intervalares, é possível definir que alguns intervalos do vetor são opcionais e outros obrigatórios. Nesse caso, ao modificador *optional* é acrescentado um par de parênteses com uma expressão booleana definindo se o intervalo é ou não opcional, como abaixo:

```
int noAtividades = ...;
range dAtividades = 1..noAtividades;
int atividadeOpcional[dAtividades] = ...;
dvar interval escalaAtividades[a in dAtividades] optional(atividadeOpcional[a]);
```

Sobre a utilização dessa estrutura, é importante frisar que, salvo se o valor da expressão entre parênteses for igual a 1, o intervalo será obrigatório. Isso já foi mencionado quando expressões numéricas e predicados foram discutidos, mas não custa repetir: apenas o valor 1 é garantidamente interpretado como verdadeiro e apenas o valor 0 como falso no contexto de uma expressão booleana em OPL.

Para acessar a informação sobre uma variável intervalar opcional estar presente ou não em uma solução, é utilizado o predicado *presenceOf*. Por exemplo, define-se que um intervalo de índice *b* par em um vetor de intervalos deve estar presente se o intervalo de índice *a* que o precede estiver presente pelo seguinte conjunto de restrições:

```
forall(a in dAtividades, b in dAtividades : b==a+1 && b%2==0) {
    presenceOf(escalaAtividade[a]) =>
        presenceOf(escalaAtividade[b]);
}
```

Ao contrário do que ocorre com os parâmetros formais nos operadores de agregação, a restrição de implicação lógica permite que condições sobre variáveis de decisão sejam empregadas. Assim, pode-se usar não apenas o predicado *presenceOf*, como é possível compor predicados com outras funções que operam sobre predicados.

Para as funções *startOf*, *endOf*, *sizeOf* e *lengthOf* vistas anteriormente, existe um segundo argumento opcional: o valor retornado pela função se o intervalo não estiver presente. Por exemplo, é possível definir que um vetor de atividades opcionais tenha todas as atividades presentes nas primeiras posições do vetor e que elas estejam dispostas em ordem crescente de início:

```
forall(a in dAtividades, b in dAtividades : b==a+1) {
    startOf(escalaAtividade[a], maxint) <= startOf(escalaAtividade[b], maxint);
}
```

Restrições desse tipo representam uma prática frequente de modelagem em programação por restrições: a quebra de simetria. Uma restrição de quebra de simetria reduz o conjunto possível de soluções para um problema ao descartar soluções que são equivalentes. Nesse caso, por exemplo, poderíamos ter soluções cuja diferença fosse apenas a ordem dos intervalos no vetor, sendo desnecessário considerar tais soluções.

A.5.3 Restrições de Precedência

Em problemas de escalonamento, é bastante comum estabelecer restrições de precedência entre eventos. Um projetista pode, por exemplo, definir que um intervalo só começa após o fim de outro e assim por diante. Apesar das funções que acessam as propriedades de início, fim e duração de um intervalo já serem suficientes para estabelecer muitas dessas relações no conjunto de restrições

do modelo, sua frequência levou à definição de uma família de predicados de precedência, como o seguinte:

```
endBeforeStart(escalaAtividade[1], escalaAtividade[2], 5);
```

Na restrição acima, expressa-se que o fim do primeiro intervalo somado ao valor 5 é menor ou igual ao início do segundo intervalo. Um exemplo um pouco diferente é o predicado abaixo, que define que o início dos dois intervalos deve ser igual:

```
startAtStart(escalaAtividade[1], escalaAtividade[2]);
```

De uma maneira geral, temos predicados começando pela propriedade do primeiro intervalo (*start* ou *end*), seguidos de um comparador (*Before* para “<=”, *At* para “==” e *After* para “>=”) e por fim a propriedade do segundo intervalo (*Start* ou *End* – esse último, ao contrário do primeiro, começando por letra maiúscula). Esses predicados possuem dois intervalos como argumento e um terceiro argumento opcional representando um valor a ser somado à propriedade do primeiro intervalo antes de compará-lo à propriedade do segundo intervalo.

A.5.4 Predicados Sobre Conjuntos de Intervalos

Em algumas situações, pode ser necessário representar um evento por vários intervalos distintos ou definir um intervalo delimitando o início e o fim de um conjunto de outros intervalos. Em OPL, existem três predicados que descrevem a relação entre um intervalo *a* e um conjunto de intervalos *B*: *span*, *alternative* e *synchronize*, todos tendo como argumentos um intervalo *a* seguido de um vetor unidimensional *B*.

Com o *span*, é possível definir que um intervalo *a* inicia junto com o primeiro intervalo presente de *B* e termina junto com o último intervalo presente de *B*, e que o intervalo *a* é ausente se todos os intervalos de *B* forem ausentes, como no exemplo abaixo:

```
dvar interval escalaAtividade;  
dvar interval escalaAtividades[dAtividades];  
span(escalaAtividade, escalaAtividades);
```

Se utilizado como uma restrição do modelo, o predicado *span* condiciona a presença do intervalo *a* à presença de alguma atividade do vetor de intervalos *B* e vice-versa.

Usando *alternative*, estabelece-se que apenas um intervalo de um vetor *B* está presente e seu início e fim coincidem com aqueles do intervalo *a*. Por um condicionamento similar ao de *span*, se *a* não estiver presente, nenhum intervalo em *B* está presente. Por exemplo, é possível ter um conjunto de atividades que devem ser realizadas e um conjunto de máquinas capazes de realizá-las, e expressar que uma, e apenas uma, máquina deve realizar cada atividade, como no exemplo a seguir:

```

int noMaquinas = ...;
range dMaquinas = 1..noMaquinas;
dvar interval escalaMaquinaAtividade[dMaquinas][dAtividades] optional;
subject to {
    forall (a in dAtividades) {
        alternative(
            escalaAtividades[a],
            all (m in dMaquinas) escalaMaquinaAtividade[m][a] );
    }
}

```

Como em outros casos vistos anteriormente, pelo exemplo dado observa-se que é possível utilizar o operador *all* para criar vetores unidimensionais de intervalos para aplicar predicados envolvendo vetores unidimensionais a qualquer conjunto possível de intervalos.

Por fim, com *synchronize* todos intervalos do conjunto B iniciam e terminam junto com o intervalo a – se a estiver presente. Ao contrário de *span* e *alternative*, no predicado *synchronize* a presença de algum intervalo de B não implica na presença do intervalo a e não existe nenhuma restrição entre os intervalos de B se a for ausente.

A.5.5 Sequências e Transições

Variáveis intervalares podem ser agrupadas em uma sequência, que é um tipo de variável de decisão declarado com a palavra reservada *sequence*. Ao criar um escalonamento, o resolvidor procura ordenar os intervalos de cada sequência para só então definir o instante de alocação para cada intervalo do conjunto. Uma sequência é declarada por um vetor unidimensional. Esse vetor é especificado após a palavra reservada *in*, como no exemplo abaixo:

```

dvar sequence todasAtividades in escalaAtividades;

```

Para interesse imediato, com sequências consegue-se definir que não podem haver sobreposições em um grupo de intervalos e ainda definir o tempo de transição entre intervalos consecutivos na ordenação. Mas isso só ocorre se uma restrição for imposta sobre a sequência. No exemplo abaixo, define-se uma sequência sobre os intervalos de índice par de um vetor de intervalos, e é estabelecido que apenas um deles pode ocorrer por vez através do predicado *noOverlap*:

```

dvar sequence atividadesPares
    in all(a in dAtividades : a % 2 == 0) escalaAtividades[a];
subject to {
    noOverlap(atividadesPares);
}

```

Além de impedir que dois intervalos se sobreponham numa sequência, com *noOverlap* é possível definir uma separação entre a conclusão de um intervalo e o início do intervalo seguinte na sequência por meio de um argumento adicional especificando o tempo de transição entre tipos de intervalos. O primeiro passo para que isso seja possível consiste em associar a cada intervalo de uma sequência um inteiro representando seu tipo na própria declaração da sequência por meio da palavra reservada *types* seguida de um vetor unidimensional de inteiros com o mesmo tamanho que o vetor de intervalos especificado antes, de forma que cada entrada de um vetor seja associada à

entrada correspondente do outro vetor. Além disso, é necessário passar como argumento da restrição *noOverlap* o tempo de separação de intervalos consecutivos de acordo com o inteiro associado a cada intervalo. Isso é feito por meio do argumento opcional de *noOverlap*. Ele consiste em um conjunto de tuplas, sendo cada tupla formada por três inteiros. Assim, a quantidade de unidades de tempo entre o fim de um intervalo e o início do seguinte é consultado nesse conjunto, sendo o primeiro inteiro relativo ao tipo do intervalo anterior, o segundo ao tipo do posterior e o terceiro estabelece o tempo de separação.

Um exemplo de declaração de sequência com tipos para os intervalos e tempo de separação entre intervalos consecutivos é apresentado a seguir:

```
dvar sequence todasAtividades
  in all(a in dAtividades) escalaAtividades[a]
  types all(a in dAtividades) a;
tuple TempoTransicao { int de; int para; int tempo; };
{ TempoTransicao } fTransicao = ...;
subject to {
  noOverlap(todasAtividades , fTransicao);
}
```

No exemplo apresentado, cada intervalo da sequência possui um identificador inteiro para seu tipo. Temos uma tupla representando uma estrutura com três inteiros: o tipo inicial, o tipo final e o tempo entre o fim de uma atividade do primeiro tipo e o início de uma atividade do segundo tipo na sequência, usado na declaração de um conjunto que serve de função de transição, dado como segundo argumento do predicado *noOverlap*.

A atribuição a tuplas foi apresentada de maneira bem superficial em um ponto anterior do texto. Apenas para deixar claro como declarar no arquivo de entrada de dados o tempo de transição para o exemplo acima, é apresentado abaixo um exemplo envolvendo dois tipos e os quatro respectivos tempos de transição:

```
fTransicao= {<1, 1, 0>, <1, 2, 10>, <2, 1, 10>, <2, 2, 0> };
```

No exemplo apresentado acima, o tempo de transição entre atividades do mesmo tipo é 0 e o tempo de transição entre atividades de tipos distintos é 10.

Por fim, vale observar que a declaração dos nomes dos campos da tupla é irrelevante para sua utilização na definição dos tempos de transição, de forma que o projetista tem toda a liberdade de definir a nomenclatura que mais lhe agradar, desde que mantida a ordem de que a primeira variável representa o tipo da atividade anterior, a segunda do tipo seguinte e a terceira o tempo de separação.

A.5.6 Modelagem de Recursos Limitados

Em muitos problemas de escalonamento, além de eventos que podem ou não ocorrer, é possível que existam recursos com propriedades que restrinjam a possibilidade de execução conjunta desses eventos. Muitas restrições envolvendo recursos associados a determinados conjuntos de eventos podem ser modelados em OPL por meio de funções cumulativas e de estado, quando não por predicados sobre sequências, conforme já mencionado. Por exemplo, uma máquina pode exigir certas configurações para realizar uma determinada atividade e outras para realizar outra atividade, de forma que os intervalos representando cada uma delas não podem se sobrepor, ao passo que

atividades com configurações idênticas podem ser realizadas simultaneamente. Entretanto, pode também ser o caso de haver um limite para o número de atividades realizadas ao mesmo tempo. Enquanto que no primeiro caso temos uma situação que pode ser modelada por uma função de estado, no segundo é possível realizar a modelagem por meio de uma função cumulativa – ambos os casos retratados nas próximas seções.

A.5.7 Funções Cumulativas

Com as funções cumulativas, é possível modelar propriedades do sistema que variam ao longo do tempo em função da contribuição numérica de expressões relacionadas ao início e/ou fim de um conjunto de intervalos, abrindo a possibilidade de expressar restrições sobre essas propriedades. Uma função cumulativa é identificada em sua declaração pela palavra reservada *cumulFunction* e consiste em uma função temporal constante por partes e composta por degraus, cujos termos são expressões que aumentam ou diminuem o valor da função ao início e/ou fim de um intervalo. Tais expressões elementares podem ser pulsos ou degraus.

Um pulso, definido pela função *pulse*, em sua forma mais simples recebe como argumento um intervalo *a* e um valor constante positivo *b*. Ele representa uma função com valor 0 antes do início de *a*, *b* entre o início e o fim de *a*, e 0 após o fim de *a*.

Um degrau pode ser definido por ao menos duas funções distintas: *stepAtStart* ou *stepAtEnd*. Em sua forma mais simples, também recebe como argumento um intervalo *a* e um valor constante positivo *b*. No caso de *stepAtStart*, representa uma função com valor 0 antes do início de *a* e *b* do início de *a* em diante. No caso de *stepAtEnd*, uma função com valor 0 antes do fim de *a* e *b* do fim de *a* em diante.

A essa constante positiva *b* das funções mencionadas, será dado o nome de coeficiente de variação.

As restrições mais simples que podem ser formuladas com funções cumulativas consistem na comparação da função com um inteiro, indicando uma relação válida para a função em todo o seu domínio. Por exemplo, para definir que no máximo 5 atividades podem ocorrer simultaneamente, é possível expressar o seguinte:

```
cumulFunction atividadesSimultaneas =
    sum(a in dAtividades) pulse(escalaAtividade[a],1);
subject to {
    atividadesSimultaneas <= 5;
}
```

Como pode ser visto no exemplo acima, foi criada uma função cumulativa composta por diversos pulsos. Nunca é demais lembrar que o coeficiente de variação das funções de pulso e degrau deve ser uma constante inteira positiva, mas ainda assim é possível somar e subtrair pulsos e degraus na composição de uma expressão de função cumulativa. Por exemplo, para expressar que o número de atividades pares ocorrendo em qualquer instante de tempo é maior que o número de atividades ímpares, é possível definir o seguinte no modelo:


```

cumulFunction saldoAtividadesPares =
  sum(a in dAtividades : a % 2 == 0) pulse(escalaAtividade[a], 1)
  - sum(a in dAtividades : a % 2 == 1) pulse(escalaAtividade[a], 1);
subject to {
  saldoAtividadesPares >= 0;
}

```

Também é possível expressar restrições sobre partes do domínio da função por meio do predicado *alwaysIn* que, em sua forma mais simples, leva como argumentos uma função cumulativa F , o intervalo a no qual o valor da função é restrito e os valores constantes de mínimo x e máximo y permitidos à função dentro desse intervalo. Por exemplo, é possível expressar que sempre haverá ao menos uma e no máximo cinco atividades ocorrendo em cada instante desde o início da primeira atividade de um vetor até o término da última, conforme o exemplo a seguir:

```

dvar interval escalaAtividade;
dvar interval escalaAtividades[dAtividades];
cumulFunction atividadesSimultaneas =
  sum(a in dAtividades) pulse(escalaAtividade[a], 1);
subject to {
  span(escalaAtividade, escalaAtividades);
  alwaysIn(atividadesSimultaneas, escalaAtividade, 1, 5);
}

```

Nas funções *pulse*, *stepAtStart*, *stepAtEnd* e *alwaysIn*, o intervalo dado como argumento pode ser substituído por duas constantes representando o início e o fim de um intervalo fixo. Por exemplo, pode-se expressar que exatamente duas atividades devem ocorrer simultaneamente no intervalo entre 31 e 40 do exemplo acima da seguinte forma:

```

alwaysIn(atividadesSimultaneas, 31, 40, 2, 2);

```

Em modelos mais elaborados, a variação da função cumulativa em função de alguns intervalos pode ter graus de liberdade. Isso é obtido pela substituição da constante b das funções de pulso e degrau por duas constantes x e y , representando que a variação do valor da função cumulativa em decorrência dos eventos de início e/ou fim de um dado intervalo pode assumir algum valor inteiro entre x e y . Por exemplo, para um determinado recurso podemos ter um conjunto de atividades de descarga que devem ser realizadas e de atividades de carga opcionais, estas últimas realizadas para manter o estoque do recurso, como no exemplo abaixo:

```

int noCargas = ...;
int noDescargas = ...;
range dCargas = 1..noCargas;
range dDescargas = 1..noDescargas;
int limEstoque = ...;
int estoquePorDescarga[dDescargas] = ...;
dvar interval atividadesCarga[dCargas] optional;
dvar interval atividadesDescarga[dDescargas];
cumulFunction estoque =
    sum(a in dCargas)
        stepAtEnd(atividadeCarga[a], 1, limEstoque)
    -
    sum(a in dDescargas)
        stepAtEnd(atividadeDescarga[a], estoquePorDescarga[a]);
subject to {
    estoque >= 0;
    estoque <= limEstoque;
}

```

No modelo apresentado, ao término de cada atividade de descarga ocorre uma variação negativa específica do estoque. Devido à existência da restrição de que o estoque deva ser sempre não-negativo mas não maior que um limite, atividades de carga também são alocadas para que ocorram variações positivas no estoque modelado pela função. A variação decorrente de cada atividade de carga é variável, assumindo qualquer valor inteiro positivo até o limite máximo do estoque. Por não assumirmos um valor sempre fixo para o impacto de um intervalo na função cumulativa, evitamos problemas com combinações de cargas e descargas que gerem estoques residuais, por exemplo. Para explicar um pouco melhor, imagine que permitíssemos apenas uma carga completa, isto é, no valor do limite máximo do estoque. Se não houver um conjunto de descargas que esvazie completamente o estoque do recurso, isso implica que não há como realizar uma nova carga com o recurso e ele não poderá ser recarregado para realizar outras atividades de descarga. A variação na função cumulativa em função do início ou fim de um intervalo pode ser medida como uma expressão de decisão por meio das funções *heightAtStart* e *heightAtEnd*, que recebem como argumento um intervalo *a* e uma função cumulativa *F* e devolvem a variação da função cumulativa ocorrida em *F*, respectivamente, no início ou fim de *a* por conta de expressões envolvendo *a*. Essa variação soma ou subtrai as contribuições de cada termo de *F* de acordo com o sinal que precede cada termo. De posse dessa função, podemos estabelecer uma relação entre o tempo da atividade de carga e sua variação de estoque, como no exemplo a seguir:

```

forall (a in dCargas) {
    lengthOf(atividadeCarga[a]) == heightAtStart(atividadeCarga[a], estoque);
}

```

As funções *heightAtStart* e *heightAtEnd* aceitam um terceiro argumento opcional com o valor a ser devolvido caso o intervalo seja opcional e não esteja presente.

Um último ponto a se observar sobre funções cumulativas refere-se à questão do limite de precisão do tipo inteiro: se a soma dos valores dos pulsos e degraus de todos os termos exceder o limite representável, o resolvidor pode considerar o problema infactível mesmo que exista solução para o problema e que as restrições do modelo impossibilitem que a função assumira um valor que

não possa ser representado por um inteiro.

A.5.8 Funções de Estado

Além de funções baseadas na contribuição numérica dos intervalos, uma outra forma de analisar propriedades e estabelecer relações entre intervalos consiste nas funções de estado. Uma função de estado é declarada pela palavra reservada *stateFunction*, mas não possui uma fórmula explícita em função dos intervalos. A relação entre uma função de estado e intervalos é feita por predicados no conjunto de restrições.

Começemos pelo predicado *alwaysIn*, que é idêntico ao apresentado anteriormente para funções cumulativas. Quando se define que uma função de estado deve ter sempre um valor entre um mínimo e um máximo no domínio de um intervalo, limita-se as possibilidades de concorrência de intervalos. Por exemplo, é possível definir que atividades pares e ímpares não ocorram simultaneamente da seguinte forma:

```
stateFunction tipoAtividade;
subject to {
    forall(a in dAtividades : a % 2 == 0) {
        alwaysIn(tipoAtividade, escalaAtividades[a], 0, 0);
    }
    forall(a in dAtividades : a % 2 == 1) {
        alwaysIn(tipoAtividade, escalaAtividades[a], 1, 1);
    }
}
```

Pelo exemplo acima, força-se que o valor da função de estado *tipoAtividade* sempre seja 0 durante a execução de atividades de índice par e sempre seja 1 durante a execução de atividades de índice ímpar. Dessa forma, uma atividade par e uma atividade ímpar nunca concorrerão pelo recurso – apenas uma delas pode ocorrer a cada instante.

Além de *alwaysIn*, também existem os predicados *alwaysEqual*, *alwaysConstant* e *alwaysNoState*. O *alwaysEqual* consiste em uma especialização do *alwaysIn*, recebendo apenas um valor *v* em vez de um valor mínimo e outro máximo como argumento, para expressar que o valor da função de estado *F* deve ser igual a *v* durante o intervalo *a*. No exemplo anterior, isso significaria não ter que colocar duas vezes o mesmo valor para forçar a igualdade. O *alwaysConstant* é uma forma de generalização do *alwaysEqual*, que determina que o valor da função de estado seja constante no domínio do intervalo, sem especificar que valor seria esse. De forma complementar, o predicado *alwaysNoState* estabelece que a função de estado não deve ter estado definido no intervalo dado.

Além das restrições sobre os predicados mencionados, é possível estabelecer o tempo de transição entre estados da função. Essa transição é modelada com conjuntos de tuplas, da mesma forma como é feito para as sequências. Porém, ela é associada diretamente à função de estado em sua declaração com a palavra reservada *with*, como abaixo:

```
tuple tempoTransição { int de; int para; int tempo; };
{ tempoTransição } fTransição = { <i, j, ((i==j)?(0):(1))> : i in 0..1, j in
    0..1 };
stateFunction tipoAtividade with fTransição;
```

Nesse exemplo, foi estabelecido um tempo unitário de transição entre os estados numericamente identificados por 0 e 1.

A.5.9 Um Exemplo de Problema de Escalonamento Modelado em OPL

Para concluir a apresentação dos elementos para modelagem de problemas de escalonamento, será dado como exemplo um modelo completo para uma versão bem simples do problema de programar recursos móveis para a execução de um conjunto de atividades. Ele é tratado em maiores detalhes na terceira parte dessa dissertação.

Nessa versão simples, temos um conjunto de atividades e um conjunto de recursos. Cada atividade possui uma localização associada e deve ser realizada por um recurso. Um recurso só pode fazer uma atividade por vez e dois recursos não podem ocupar o mesmo lugar ao mesmo tempo. Para realizar atividades consecutivas em locais diferentes, a alocação de cada recurso deve considerar o tempo de deslocamento necessário entre terminar a atividade anterior em um local e começar a atividade seguinte em outro local. Segue um exemplo de modelo para esse problema:

```

1  using CP;
2
3  int noAtividades = ...;
4  range dAtividades = 1..noAtividades;
5
6  int noRecursos = ...;
7  range dRecursos = 1..noRecursos;
8  int noLocais = ...;
9  range dLocais = 1..noLocais;
10
11 // Vetor que, numericamente, associa cada atividade a um local.
12 int localizacao[a in dAtividades] = ...;
13 // Definição da tupla representando o tempo de transição entre locais.
14 tuple Transicao { int deLocal; int paraLocal; int tempoDeslocamento; };
15 // Vetor de conjuntos
16 {Transicao} tempoTransicao[r in dRecursos] = ...;
17
18 dvar interval escalaAtividade[a in dAtividades];
19 dvar interval escalaAtividadeRecurso[a in dAtividades][r in dRecursos] optional;
20
21 // Agrupamento das atividades realizadas por cada recurso,
22 // associando tipo relativo à localização.
23 dvar sequence sqAtvRecurso[r in Resources]
24     in all(a in Atividades) escalaAtividadeRecurso[a][r]
25     types all(a in Atividades) localizacao[a];
26 // Agrupamento das atividades realizadas por cada local.
27 dvar sequence sqAtvLocal[l in Locais]
28     in all(a in dAtividades : location[a]==l) escalaAtividade[a];
29
30 subject to {
31     // Cada atividade é realizada por exatamente um recurso.
32     forall(a in dAtividades) {
33         // Força com que cada atividade (com intervalo obrigatório) esteja
34         // presente em apenas um intervalo de recurso (os intervalos da
35         // matriz Atividade x Recurso são opcionais).
36         alternative(escalaAtividade[a],
37             all(r in dRecursos) escalaAtividadeRecurso[a][r]);
38     }
39
40 // Cada recurso realiza uma atividade por vez, e separa-as pelo tempo
41 // de deslocamento entre os locais de atividades consecutivas.
42 forall(r in dRecursos) {
43     noOverlap(sqAtvRecurso[r], tempoTransicao[r]);
44 }
45
46 // Apenas uma atividade pode ser realizada por vez em cada local.
47 forall(l in dLocais) {
48     noOverlap(sqAtvLocal[l]);
49 }
50 }

```

A.6 Considerações Finais e Agradecimentos

A expectativa do autor com relação a este material é que ele sirva à apresentação dos conceitos necessários para familiarizar o leitor com a modelagem algébrica de modelos utilizando a técnica de programação por restrições. Como subsídio, foi utilizada a linguagem OPL em razão de sua maturidade e popularidade. Para um aprofundamento na prática atual usando OPL, recomenda-se tanto o manual da [IBM \(2010\)](#) quanto uma leitura de [Laborie \(2009\)](#). É importante observar que alguns artigos mais antigos como [Hentenryck *et al.* \(2000\)](#) e [Lustig e Puget \(2001\)](#) apresentam uma sintaxe e certas estruturas que foram descontinuadas em versões posteriores. Entretanto, sua leitura permanece útil à compreensão dos princípios que nortearam o desenvolvimento da linguagem e o funcionamento do resolvidor.

Esse texto contou com a colaboração de vários revisores, que contribuíram para torná-lo mais acessível e direto por meio de uma leitura minuciosa. Até a presente versão, foram eles: Gilberto Koji Nishioka, Edson Mikio Nishida, Daniel Felix Ferber, Gustavo Pacianotto Gouveia e Yoshiko Wakabayashi.

Referências Bibliográficas

- Aarts e Lenstra(1997)** E. H. L. Aarts e J. K. Lenstra. *Local Search in Combinatorial Optimization*. Wiley. Citado na pág. 10
- Accioly et al.(2002)** R. Accioly, F. J. M. Marcellino e H. Kobayashi. Uma aplicação da programação por restrições no escalonamento de atividades em poços de petróleo. Em *Simpósio Brasileiro de Pesquisa Operacional*. Sobrapo. Citado na pág. 7
- Acergy-Group(2010)** Acergy-Group. Polar queen apr 10.pdf. [http://www.acergy-group.com/publicroot/webresources/6P6D55RDUN/\\$file/Polar%20Queen%20Apr%2010.pdf](http://www.acergy-group.com/publicroot/webresources/6P6D55RDUN/$file/Polar%20Queen%20Apr%2010.pdf), 2010. Formato PDF. Acesso em: 1 jun. 2010. Citado na pág. 5
- Aloise et al.(2006)** D. J. Aloise, D. Aloise, C. T. M. Rocha, C. C. Ribeiro, J. C. R. Filho e L. S. S. Moura. Scheduling workover rigs for onshore oil production. *Discrete Applied Mathematics*, 154: 695–702(5). Citado na pág. 8
- Arbelaez et al.(2009)** A. Arbelaez, Y. Hamadi e M. Sebag. Online heuristic selection in constraint programming. Em *Proceedings of the International Symposium on Combinatorial Search (SoCS'09)*, páginas 674–681. Citado na pág. 33
- Baptiste et al.(2001)** P. Baptiste, C. Le Pape e W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer. Citado na pág. 33, 42, 49, 53, 55
- Baptiste et al.(2006)** P. Baptiste, P. Laborie, C. Le Pape e W. Nuijten. Constraint-based scheduling and planning. Em F. Rossi, P. van Beek e T. Walsh, editors, *Handbook of Constraint Programming*, chapter 22. Elsevier. Citado na pág. 41, 42, 49, 50, 52, 53, 54
- Barták(2001)** R. Barták. Theory and practice of constraint propagation. Em *Proceedings of the 3rd Workshop on Constraint Programming for Decision and Control (CPDC2001)*, páginas 7–14. Wydawnictwo Pracowni Komputerowej. Citado na pág. 14, 19, 25, 26
- Barták(1999)** R. Barták. Constraint programming: In pursuit of the holy grail. Em *Proceedings of the Week of Doctoral Students (WDS99)*, páginas 555–564. MatFyzPress. Citado na pág. 13, 14, 19, 21, 25, 31, 34, 37
- Barták(2008)** R. Barták. Book review: Constraint Processing, Handbook of Constraint Programming. *Computer Science Review*, 2(2):123 – 130. Citado na pág. 15
- Battiti et al.(2008)** R. Battiti, M. Brunato e F. Mascia. *Reactive Search and Intelligent Optimization*. Springer. Citado na pág. 10, 56
- Bazaraa et al.(1990)** M. S. Bazaraa, J. J. Jarvis e H. D. Sherali. *Linear Programming and Network Flows*. Wiley. Citado na pág. 4
- Beldiceanu et al.(2007)** Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey e Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12:21–62. Citado na pág. 27

- Bessière e Hentenryck(2003)** C. Bessière e P. V. Hentenryck. To be or not to be ... a global constraint. Em Francesca Rossi, editor, *Principles and Practice of Constraint Programming*, volume 2823 of *Lecture Notes in Computer Science*, páginas 789–794. Springer Berlin Heidelberg. Citado na pág. 27
- Bockmayr e Kasper(1998)** A. Bockmayr e T. Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10: 287–300(3). Citado na pág. 14
- Bondy e Murty(2008)** A. Bondy e U. S. R. Murty. *Graph Theory*. Springer. Citado na pág. 4
- Brafman(2001)** R. I. Brafman. A simplifier for propositional formulas with many binary clauses. Em *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*. Citado na pág. 50
- Brailsford et al.(1999)** S. C. Brailsford, C. N. Potts e B. M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research (EJOR)*, 119: 557–581. Citado na pág. 15, 18, 33
- Camm et al.(1990)** J. D. Camm, A. S. Raturi e S. Tsubakitani. Cutting big M down to size. *Interfaces*, 20:61–66. Citado na pág. 73
- Carchrae e Beck(2005)** T. Carchrae e J.C. Beck. Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence*, 21:372–387. Citado na pág. 56
- Carlier e Pinson(1994)** J. Carlier e E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research (EJOR)*, 78:146–161. Citado na pág. 53
- Cesta e Oddi(1996)** A. Cesta e A. Oddi. Gaining efficiency and flexibility in the simple temporal problem. Em *Proceedings of the 3rd Workshop on Temporal Representation and Reasoning (TIME'96)*. Citado na pág. 51
- Cormen et al.(2002)** T. H. Cormen, C. E. Leiserson, R. L. Rivest e S. Clifford. *Algoritmos – Teoria e Prática*. Editora Campus. Citado na pág. 4, 29, 50
- Dechter(2003)** R. Dechter. *Constraint Processing*. Morgan Kaufmann. Citado na pág. 15, 19, 21, 23, 29, 34, 35, 36, 37, 56
- do Nascimento(2002)** J. M. do Nascimento. Ferramentas computacionais híbridas para a otimização da produção de petróleo em Águas profundas. Dissertação de Mestrado, Instituto de Computação, Universidade Estadual de Campinas, Brasil. Citado na pág. 7
- Dynadec(2009)** Dynadec. *Comet 2.0 Tutorial*, 2009. Citado na pág. 38, 42, 46
- Feo e Resende(1989)** T. A. Feo e M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71. Citado na pág. 7
- Focacci et al.(2000)** F. Focacci, A. Lodi, M. Milano e D. Vigo. An introduction to constraint programming. *Ricerca Operativa*, 91. Citado na pág. 22
- Glinz e Berumen(2009)** I. Glinz e L. Berumen. Optimization model for an oil well drilling program: Mexico case. *Oil and Gas Business*, 1. Citado na pág. 8
- Glover(1989)** F. Glover. Tabu search – part I. *ORSA Journal on Computing*, 1:190–206(3). Citado na pág. 7
- Godard e Laborie(2005)** D. Godard e P. Laborie. Randomized large neighborhood search for cumulative scheduling. Em *Proceedings of the 2005 International Conference on Automated Planning and Scheduling (ICAPS 2005)*, páginas 81–89. Citado na pág. 38, 56

- Gomes e Selman(2007)** C. Gomes e B. Selman. The science of constraints. *Constraint Programming Letters*, 1:15–20. Citado na pág. 39
- Gomes et al.(1998)** C. P. Gomes, B. Selman e H. Kautz. Boosting combinatorial search through randomization. Em *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, páginas 431–437. Citado na pág. 36
- Hasle et al.(1996)** G. Hasle, R. Haut, B. Johansen e T. Ølberg. Well activity scheduling – an application of constraint reasoning. Em B. Braunschweig e B. A. Bremdal, editors, *Artificial Intelligence in the Petroleum Industry: Symbolic and Computational Applications II*, páginas 209–228. Technip, Paris. Citado na pág. 7
- Hentenryck(2010)** P. V. Hentenryck. Constraint programming as declarative algorithmics. Em F. Benhamou, N. Jussien e B. O’Sullivan, editors, *Trends in Constraint Programming*. Wiley. Citado na pág. 13, 39
- Hentenryck e Michel(2007)** P. V. Hentenryck e L. Michel. Synthesis of constraint-based local search algorithms from high-level models. Em *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI-07)*, páginas 273–278. Citado na pág. 39
- Hentenryck et al.(1992)** P. V. Hentenryck, Y. Deville e C. M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57. Citado na pág. 25
- Hentenryck et al.(2000)** P. V. Hentenryck, L. Perron e J. F. Puget. Search and strategies in opl. *ACM Transactions on Computational Logic (TOCL)*, 1:285–320(2). Citado na pág. 18, 38, 39, 120
- Hooker(2007)** J. N. Hooker. Good and bad futures for constraint programming (and operations research). *Constraint Programming Letters*, 1:21–32. Citado na pág. 30
- Hooker(2006)** J. N. Hooker. *Integrated Methods for Optimization (International Series in Operations Research & Management Science)*. Springer-Verlag New York, Inc. Citado na pág. 10
- Hoos e Stützle(2005)** H. H. Hoos e T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann. Citado na pág. 37
- IBM(2010)** IBM. *ILOG CPLEX Optimization Studio 12.2 documentation for ODM Enterprise*, 2010. Citado na pág. 16, 18, 38, 42, 89, 120
- Jaffar e Lassez(1987)** J. Jaffar e J. L. Lassez. Constraint logic programming. Em *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL ’87)*, páginas 111–119. Citado na pág. 37
- Kantorovich(1975)** L. V. Kantorovich. Mathematics in economics: Achievements, difficulties, perspectives. http://www.nobelprize.org/nobel_prizes/economics/laureates/1975/kantorovich-lecture.html, 1975. Último acesso em 6/9/2011. Citado na pág. 13
- Kautz et al.(2002)** H. Kautz, E. Horvitz, Y. Ruan, C. Gomes e B. Selman. Dynamic restart policies. Em *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, páginas 674–681. Citado na pág. 33, 36
- Koopmans(1975)** T. C. Koopmans. Concepts of optimality and their uses. http://www.nobelprize.org/nobel_prizes/economics/laureates/1975/koopmans-lecture.html, 1975. Último acesso em 6/9/2011. Citado na pág. 13
- Laborie(2009)** P. Laborie. IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. Em W. J. van Hoes e J. N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *Lecture Notes in Computer Science*, páginas 148–162. Springer Berlin Heidelberg. Citado na pág. 120

- Laborie(2003a)** P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143:151–188. Citado na pág. 38, 49, 51, 52, 54, 55
- Laborie(2003b)** P. Laborie. Resource temporal networks: Definition and complexity. Em *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*. Citado na pág. 46
- Laborie e Godard(2007)** P. Laborie e D. Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. Em *Proceedings of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2007)*, páginas 276–284. Citado na pág. 37, 38, 56
- Laborie e Rogerie(2008)** P. Laborie e J. Rogerie. Reasoning with conditional time-intervals. Em *Proceedings of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS 2008)*. Citado na pág. 38, 46, 50, 63
- Laborie et al.(2009)** P. Laborie, J. Rogerie, P. Shaw e P. Vilím. Reasoning with conditional time-intervals, part ii: An algebraical model for resources. Em *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference (FLAIRS 2009)*. Citado na pág. 38, 46, 63
- Lustig e Puget(2001)** I. J. Lustig e J. F. Puget. Program does not equal program: Constraint programming and its relationship to mathematical programming. *Interfaces*, 31:29–53(6). Citado na pág. 13, 14, 16, 30, 37, 38, 90, 106, 120
- Monette et al.(2009)** J. N. Monette, Y. Deville e P. V. Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. Em *Proceedings of 11th INFORMS Computing Society Conference*. Citado na pág. 39, 56
- Moura et al.(2008a)** A. V. Moura, C. C. de Souza, A. A. Cire e T. M. T. Lopes. Planning and scheduling the operation of a very large oil pipeline network. Em Peter Stuckey, editor, *Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, páginas 36–51. Springer Berlin Heidelberg. Citado na pág. 8
- Moura et al.(2008b)** A. V. Moura, R. A. Pereira e C. C. de Souza. Scheduling activities at oil wells with resource displacement. *International Transactions in Operational Research*, 15:659–683(25). Citado na pág. 7
- Nethercote et al.(2007)** N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck e G. Tack. MiniZinc: Towards a standard CP modelling language. Em C. Bessière, editor, *Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, páginas 529–543. Springer Berlin Heidelberg. Citado na pág. 39
- Nocedal e Wright(2000)** J. Nocedal e S. J. Wright. *Numerical Optimization*. Springer. Citado na pág. 4
- Paiva(1997)** R. O. Paiva. Itinerário de sondas com quantificação de perdas através de simulador de reservatórios. Dissertação de Mestrado, Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Brasil. Citado na pág. 8
- Pereira et al.(2005)** R. A. Pereira, A. V. Moura e C. C. de Souza. Comparative experiments with GRASP and constraint programming for the oil well drilling problem. Em S. E. Nikolettseas, editor, *WEA*, volume 3503 of *Lecture Notes in Computer Science*, páginas 328–340. Springer. Citado na pág. 7

- Puget(2004)** J. F. Puget. Constraint programming next challenge: Simplicity of use. Em Mark Wallace, editor, *Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, páginas 5–8. Springer Berlin Heidelberg. Citado na pág. [38](#), [39](#)
- Refalo(2004)** P. Refalo. Impact-based search strategies for constraint programming. Em Mark Wallace, editor, *Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, páginas 557–571. Springer Berlin Heidelberg. Citado na pág. [32](#), [33](#), [38](#)
- Régin(1994)** J. C. Régin. A filtering algorithm for constraints of difference in CSPs. Em *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, páginas 362–367. Citado na pág. [28](#), [29](#)
- Scott(1999)** M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann. Citado na pág. [37](#)
- Serra e Wakabayashi(2010)** T. Serra e Y. Wakabayashi. The full employment theorem for solver designers and related issues in CP modeling. Em *Anais do V Workshop de Teses e Dissertações em Andamento (WTDIA)*. SBC. Citado na pág. [15](#), [39](#)
- Serra et al.(2010)** T. Serra, G. Nishioka e F. Marcellino. A constraint programming approach to the problem of oil well drilling scheduling with resource displacement and inventory management. Em *ALIO-INFORMS Joint International Meeting*. ALIO-INFORMS. Citado na pág. [7](#)
- Serra et al.(2011)** T. Serra, G. Nishioka e F. J. M. Marcellino. A constraint-based scheduling of offshore well development activities with inventory management. Em *Simpósio Brasileiro de Pesquisa Operacional*. Sobrapo. Citado na pág. [7](#), [71](#), [72](#), [78](#)
- Serra et al.(2012)** T. Serra, G. Nishioka e F. J. M. Marcellino. On estimating the return of resource acquisitions through scheduling: An evaluation of continuous-time MILP models to approach the development of offshore oil wells. Em *Scheduling and Planning Applications workshop (SPARK)*. AAAI. Citado na pág. [7](#), [71](#), [72](#), [78](#)
- Sipser(2007)** M. Sipser. *Introdução à Teoria da Computação*. Thomson. Citado na pág. [4](#)
- Smith-Miles(2008)** K. A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41. Citado na pág. [15](#), [56](#)
- Thomas(2004)** J. E. Thomas. *Fundamentos de Engenharia de Petróleo*. Interciência. Citado na pág. [3](#)
- van Beek(2006)** P. van Beek. Backtracking search algorithms. Em F. Rossi, P. van Beek e T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier. Citado na pág. [29](#), [30](#), [31](#), [34](#), [35](#)
- van Hoeve et al.(2011)** W. J. van Hoeve, M. Hunting e C. Kuip. The aimms interface to constraint programming. Em T. Achterberg e J. C. Beck, editors, *Proceedings of CPAIOR Late Breaking Abstracts*, páginas 41–43. Konrad-Zuse-Zentrum für Informationstechnik Berlin. Citado na pág. [39](#)
- Vazirani(2001)** V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag. Citado na pág. [9](#)
- Vilain et al.(1990)** M. Vilain, H. Kautz e P. van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. Em *Readings in qualitative reasoning about physical systems*. Morgan Kaufmann. Citado na pág. [46](#)
- Vilím(2007)** P. Vilím. *Global Constraints in Scheduling*. Tese de Doutorado, Faculty of Mathematics and Physics, Charles University, República Tcheca. Citado na pág. [49](#), [53](#), [54](#)
- Williams(2005)** H. P. Williams. *Model Building in Mathematical Programming*. Wiley. Citado na pág. [18](#)

Wolsey(1998) L. A. Wolsey. *Integer Programming*. Wiley. Citado na pág. 4

Yue(2008) Z. Q. Yue. 9320168.jpg. <http://www.panoramio.com/photo/9320168>, 2008. Largura: 2272 pixels. Altura: 1704 pixels. Tamanho: 1017,17KB. Formato JPEG. Acesso em: 1 jun. 2010. Citado na pág. 5

Índice Remissivo

águas profundas, 5

árvore, 4

algoritmo

AC, 24

AC-1, 24

AC-2, 24

AC-3, 24

AC-4, 24

AC-5, 25

AC-6, 24

AC-7, 24

BJ, 35

BT, 31

CBJ, 35

de aproximação, 9

de busca, 29, 55

de caminhos mais curtos, 71

de filtragem, 14, 19, 24, 25, 28

de retrocesso baseado em conflito, 35

de retrocesso por salto, 35

FC, 34

MAC, 34

PC, 25

PC-1, 25

PC-2, 25

PC-3, 25

PC-4, 25

PC-5, 25

alocação, 6, 8

antecipação

parcial, 34

total, 34

arco, 20

atividade

elástica, 43

não-preemptiva, 43

opcional, 44, 46

preemptiva, 43

barcos lançadores de linhas, 5

busca, 13, 29, 39, 41, 55

algoritmo de, 55

completa, 29

corte na, 37

degeneração da, 31, 35

dicotômica, 30

espaço de, 63

estocástica, 29, 36

exaustiva, 29

incompleta, 29

local, 9, 29, 36

local estocástica, 37

por discrepância limitada, 33, 36

por factibilidade, 29

tabu, 7, 9

classe de problemas

NP, 4

NP-Completo, 4, 15, 19, 24, 50

NP-Díficeis, 4, 7, 15

codificação

dual, 26

por variáveis ocultas, 27

conjunto

de conflito, 35

de valores de suporte, 24

consistência, 14, 19, 33, 49

k -consistência, 21

k -consistência forte, 21

n -consistência, 22

(i,j), 23

1-consistência, 21

2-consistência, 21

3-consistência, 21

binária, 20, 21, 24

níveis de, 21

completa, 19, 22

de arco, 20

de nó, 20

direcional em nível de aresta, 22

em k -ésimo nível, 21

em nível de aresta, 21, 24, 34

em nível de caminho, 21, 25

em nível de vértice, 21, 24

forte, 21

- generalizada, 27, 28
- inversa, 24
- manutenção da, 34
- níveis de, 14, 19, 21, 22
- níveis intermediários de, 25
- ponto fixo de, 49, 51, 53
- por conjunto unitário, 23
- por limites, 23, 50
- restrita em nível de caminho, 23
- técnicas de, 19
- tipos de, 19
- CP
 - conferência, 28
- data
 - de liberação, 43, 52
 - limite de conclusão, 43, 52
- detecção tardia de conflito, 31, 33, 35
- domínio
 - de restrição, 22
 - de variável, 14
- downstream, 5
- emparelhamento máximo, 5, 28
- escalonamento, 7, 8, 41
 - baseado em restrições, 9, 41, 63
- exploitation, 5
- exploração, 5
- factibilidade, 29
- ferramentas comerciais, 38
- função
 - cumulativa, 48
 - degrau, 48
 - pulso, 48
 - de custo de transição, 45
 - de estado, 48
 - de tempo de transição, 44, 48, 64
- grafo, 4, 16
 - bipartido, 5, 28
 - corte de um, 5
 - de precedências, 53
 - do modelo, 21, 27
 - do modelo dual, 26
 - hipergrafo, 27
 - subgrafo, 4
 - subgrafo induzido, 4, 27
- grafos
 - isomorfismo de, 5, 27
 - ordenação topológica de, 4
 - orientação de, 4
 - rotulação graciosa de, 16
- GRASP, 7, 9
- heurística, 9
 - construtiva, 9
 - de busca local, 9
 - meta-heurística, 9
- inferência, 19
 - geração de, 19
- inviável, 30
- linguagem
 - AIMMS, 39
 - algébrica, 38
 - Comet, 38, 39, 56
 - de modelagem abstrata, 39
 - MiniZinc, 39
 - OPL, 16, 17, 38, 42, 55, 63, 77
 - Prolog, 37
- meta-aprendizagem, 15
- modelagem, 14, 18, 39, 41, 42
- modelo, 13, 14, 18, 42, 63
 - binário, 26
 - dual, 26
 - insatisfatório, 28
- nó, 20
- nível
 - de planejamento, 8
 - operacional, 8
 - tático, 8
- ORCA
 - sistema, 7
- ordenação
 - das atribuições, 32
 - dinâmica, 32
 - estática, 32
- otimalidade, 29
- otimização numérica, 3
- paradigmas, 37
- princípio
 - da casa dos pombos, 28
 - da sub-estrutura ótima, 29
 - de falhar primeiro, 32
- problema
 - combinatório, 3, 9, 16
 - da seleção de algoritmos, 15
 - das n rainhas, 17
 - de escalonamento de chão-de-fábrica, 53
 - de formulação da estratégia ótima de ramificação, 33
 - de otimização da satisfação de restrições, 14
 - de satisfação de restrições, 14

- de satisfabilidade, 50
- de satisfabilidade 2-SAT, 50
- do escalonamento de chão-de-fábrica, 7
- insatisfável, 19, 33
- satisfável, 33
- temporal simples, 51
- produção, 5
- programação, 8
 - dinâmica, 29
 - funcional, 37
 - inteira, 15
 - lógica, 37
 - lógica por restrições, 37
 - linear, 3, 13
 - linear inteira, 3, 9, 14
 - matemática, 8
 - por restrições, 7, 9, 13
- projeção, 26
- propagação, 41, 63
 - de restrições, 14, 19, 49
 - disjuntiva, 52
 - energética, 52
 - global, 27, 51
 - local, 27, 50
 - por ausência de arestas, 53
 - por balanceamento, 54
 - por busca de arestas, 53
 - por precedência energética, 54
 - por tabela de tempos, 52
- quebra de simetrias, 18
- ramificação, 31, 78
 - estratégias de, 31
 - heurísticas para, 32
 - tipos de, 31
- recursão, 29, 30
- recurso
 - capacidade do, 45
 - cumulativo, 45
 - disjuntivo, 45
 - estado do, 44
 - reservatório, 45
 - tipo de estado do, 44
- redes booleanas, 50
- rejeição, 35
- relaxação, 71
- resolvedor caixa-preta, 39
- restrição, 14
 - alldifferent, 17, 18, 28
 - alternative, 47, 64
 - always-constant, 48
 - always-in, 48, 66
 - always-nostate, 48
 - anti-funcional, 25
 - aprendizado de, 35
 - aridade de, 26
 - binária, 20
 - binarização de, 26
 - declarativa, 14
 - enumerativa, 14
 - fraca, 15
 - funcional, 25
 - global, 27, 39, 47
 - algorítmica, 28
 - operacional, 28
 - semântica, 28
 - implícita, 35
 - monotônica, 25
 - no-overlap, 47, 64, 67, 68
 - redundante, 18, 26, 35
 - span, 47
 - unária, 20, 24
- retrocesso, 29, 30
 - cronológico, 31
 - dinâmico, 35
 - inteligente, 35
- SAT, *veja* problema de satisfabilidade
- sequência de intervalos, 47
- simetria, 18
- sintaxe, 63
- sistemática de exploração, 29
- solução equivalente, 18
- sondas marítimas, 5
- sudoku, 17
- técnicas
 - preventivas, 33
 - reativas, 34
- trabalho redundante, 31, 35, 36
- upstream, 5
- vértice
 - grau do, 22
- valor de suporte, 24
- variável
 - de decisão, 14
 - de intervalo, 46, 63
- variável intervalar, *veja* variável de intervalo
- verificação adiantada, 34