Pontifícia Universidade Católica
do Rio de Janeiro

Dárlinton Barbosa Feres Carvalho

Um Framework para Construção de Vocabulário e sua Aplicação ao Problema de Seqüenciamento de Carros

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio

Orientadores: Prof. Carlos José Pereira de Lucena
Prof. Celso da Cruz Carneiro Ribeiro

Rio de Janeiro
Março de 2007

Pontifícia Universidade Católica
do Rio de Janeiro

Dárlinton Barbosa Feres Carvalho

Um Framework para Construção de Vocabulário e sua Aplicação ao Problema de Seqüenciamento de Carros

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos José Pereira de Lucena
Orientador
Departamento de Informática – PUC–Rio

Prof. Celso da Cruz Carneiro Ribeiro
Orientador
Departamento de Informática – PUC–Rio

Prof. Edward Hermann Haeusler
Departamento de Informática – PUC–Rio

Prof. Noemi Rodriguez
Departamento de Informática – PUC–Rio

Prof. Simone Lima Martins
Departamento de Ciência da Computação – UFF

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico – PUC–Rio

Rio de Janeiro, 19 de Março de 2007

Dárlinton Barbosa Feres Carvalho

Graduou–se na Universidade Federal de Ouro Preto (Ouro Preto, Brasil), cursando Ciência da Computação. Atualmente é Analista de Sistemas Sênior no Departamento de Modernização e Programas da Educação Superior da Secretaria de Educação Superior do Ministério da Educação.

Ficha Catalográfica

Carvalho, Dárlinton Barbosa Feres

Um Framework para Construção de Vocabulário e sua Aplicação ao Problema de Seqüenciamento de Carros / Dárlinton Barbosa Feres Carvalho; orientadores: Carlos José Pereira de Lucena, Celso da Cruz Carneiro Ribeiro. – 2007.

100 f.: il. ; 30 cm

Dissertação (Mestrado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

Inclui bibliografia.

1. Informática – Teses. 2. Construção de Vocabulário. 3. Heurísticas. 4. Frameworks. 5. Seqüenciamento de Veículos. 6. Otimização Combinatória. I. Lucena, Carlos José Pereira de. II. Ribeiro, Celso da Cruz Carneiro. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Agradecimentos

À minha família toda: papai, mamãe, irmãos, avós. Especialmente, vovó Odete que sentia minha ausência, mas compreendia com seu carinho.

Aos meus orientadores pelo exemplo, apoio e incentivo para a realização deste trabalho.

Aos meus colegas da PUC–Rio, pelo excelente ambiente para pesquisa e produção. Em especial para os amigos Thiago Noronha, Bruno, Leonardo, Rodrigos, Pedro, Vinícius, Malcher, Sebastian, Eraldo, Daniel, Machado, Sérgio, entre outros não menos importantes que ficam nas lembranças do coração.

Aos mestres e amigos Marcone, Elton e Gromato pelos fundamentos para realização deste trabalho.

Ao pessoal do departamento de Informática pela ajuda de todos os dias, em particular à Professora Clarisse, Débora e Solange.

À CAPES e à PUC–Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Resumo

Carvalho, Dárlinton Barbosa Feres; Lucena, Carlos José Pereira de; Ribeiro, Celso da Cruz Carneiro. **Um Framework para Construção de Vocabulário e sua Aplicação ao Problema de Seqüenciamento de Carros**. Rio de Janeiro, 2007. 100p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Construção de vocabulário é uma heurística para problemas de otimização combinatória que propõe identificar porções de boas soluções e recombiná-las de modo a intensificar a busca em regiões do espaço de soluções identificadas como promissoras. A técnica de construção de vocabulário pode ser aplicada de diversas maneiras na resolução de problemas. Para facilitar a implementação e comparação de algoritmos de um mesmo domínio, a tecnologia de *frameworks* é uma solução que já demonstrou ser muito eficaz. O objetivo deste trabalho é desenvolver um *framework* para a implementação de heurísticas baseadas em construção de vocabulário. O desenvolvimento foi fundamentado em extensa revisão bibliográfica sobre a técnica e em boas práticas de engenharia de software, como *frameworks* orientados a objetos e padrões de projeto. Como um estudo de caso, foram geradas aplicações a partir do *framework* para a resolução do problema de seqüenciamento da produção de carros, que é um problema combinatório proposto a partir de necessidades reais da indústria.

Palavras–chave

Construção de Vocabulário. Heurísticas. Frameworks. Seqüenciamento de Veículos. Otimização Combinatória.

Abstract

Carvalho, Dárlinton Barbosa Feres; Lucena, Carlos José Pereira de; Ribeiro, Celso da Cruz Carneiro. **A Framework for Vocabulary Building Heuristic and yours Application to the Car Sequencing Problem**. Rio de Janeiro, 2007. 100p. MSc Thesis – Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Vocabulary building is a heuristic for solving combinatorial optimization problems, based on the identification of solution fragments which are common to good solutions and on their combination to intensify the search on promising regions of the solution space. This technique can be vastly applied on problem solving. The technology of frameworks is an efficient strategy to facilitate the implementation and comparison of same domain algorithms. The objective of this work is to develop a framework for the implementation of heuristics based on vocabulary building. Its development was based on a wide bibliographic revision about the technique and good software engineering practices, like oriented objects frameworks and design patters. We generated applications of the framework to solve the car sequencing problem, which is a combinatorial problem proposed by real requirements of the industry.

Keywords

Vocabulary Building. Heuristics. Frameworks. Car Sequencing Problem. Combinatorial Optimization.

Sumário

Lista de figuras

Lista de tabelas

Malleable forms of memory entail certain dangers — potential pitfalls that go hand in hand with the ability to provide valuable strategic opportunities. These dangers are the price to be paid for the evolution of "intelligent" mechanisms, including biological mechanisms embodied in a brain. This perspective invites a reconsideration of popular themes: e.g., if an increase in mental capacities creates such attendant risks, the act of acquiring a "knowledge of good and evil," as in the Garden of Eden story, is an understandable basis for expulsion from a simpler (and safer) existence.

Fred Glover [12].

1
Introdução

Construção de vocabulário é uma técnica de memória adaptativa para métodos heurísticos de busca. Sua proposta é identificar porções de boas soluções e recombiná-las de modo a intensificar a busca em regiões identificadas como promissoras. Problemas de natureza combinatória inviabilizam uma ampla busca pelo espaço de soluções, pelo enorme número de possibilidades existentes, sendo necessário restringir o foco da pesquisa. Construção de vocabulário permite aprender sobre regiões promissoras do espaço de busca e ao mesmo tempo combinar novas formas de explorar a vizinhança.

A técnica foi originalmente proposta por Glover [11] a partir da observação dos benefícios obtidos pela aplicação de estruturas de vizinhança mais complexas, em particular, as que consideram grandes fragmentos de solução, como na vizinhança cadeia de ejeção [10]. Seu nome vem da analogia com métodos da inteligência artificial para extração de texto. A primeira proposta de utilização dessa técnica foi como uma memória de longo prazo para a metaheurística busca tabu [14].

Na literatura, encontram-se trabalhos aplicados a diversos problemas, com resultados que corroboram os benefícios da utilização de construção de vocabulário. No problema de roteamento de veículos, os trabalhos pioneiros de Rochat e Taillard [28] e Kelly e Xu [19] alcançaram resultados expressivos. O problema de escalonamento da produção de uma indústria de aço foi bem resolvido por Lopez, Carter e Gendreau [21], com uma técnica similar, denominada "canibalismo". Scholl et al. [30] desenvolveram uma heurística baseada em padrões para resolver o problema de seqüenciamento de produtos em uma linha de montagem. Seguindo a proposta de Kelly e Xu [19], Combs [6] desenvolveu uma busca tabu adaptativa para o problema de escalonamento de tripulação. Ribeiro et al. [25] propuseram um GRASP híbrido combinando algoritmos de mineração de dados. Fernandes [8] aplicou construção de vocabulário ao problema de seqüenciamento de DNA e Aloise [1] ao projeto de redes com funções de custo discretas, mostrando que agregando essa técnica a outras heurísticas aumenta a eficiência das heurísticas de busca. Sua importância como uma eficiente implementação de memória a longo prazo também é en-

contrada em outros artigos [12, 13, 32].

Há diversas maneiras de aplicar construção de vocabulário na resolução de problemas. Proposta para ter uma memória de longo prazo, ela pode ser utilizada diversas vezes durante a execução da busca [1, 21, 30] ou, simplesmente, como uma heurística de pós-otimização [8, 28]. O conjunto de soluções elite pode ser construído segundo diversos critérios [16], consistindo, por exemplo, da diversidade, qualidade e quantidade dessas soluções. Para identificar fragmentos nas soluções, podem ser utilizadas técnicas baseadas em mineração de dados [25], representações específicas da solução [6, 19, 21, 28, 30] ou, como a proposta original [11], através da interseção [1, 8] de soluções. Para gerar novas soluções, na proposta original, é realizado um procedimento inverso chamado de interseção estendida [12]. Outro modo de formar soluções é fixar boa parte da solução e completá-la por outro método, exato [1] ou heurístico [21, 28, 30]. Também, pode-se utilizar a resolução exata de um problema auxiliar, como o problema de partição de conjuntos, para formar novas soluções [19]. Nem sempre é possível aplicar todos esses procedimentos a um determinado problema, pois cada problema possui características particulares que indicam qual técnica deve ser mais apropriada e produz melhores resultados.

Para facilitar a implementação e comparação de algoritmos, a tecnologia de *frameworks* [23] é uma solução que já demonstrou ser muito eficaz. Na literatura existem muitos trabalhos sobre o desenvolvimento de *frameworks* e bibliotecas para software de otimização [34], porém, nenhum se propõe a desenvolver construção de vocabulário. Atualmente, as boas práticas de engenharia de *software* como sistemas orientados a objetos e padrões de projeto [9] pautam o desenvolvimento de *frameworks*. Visto que há diversas maneiras de se implementar uma construção de vocabulário, uma metodologia para implementar e comparar essas diferentes abordagens é bastante desejável.

O objetivo deste trabalho é desenvolver um *framework* para a implementação de heurísticas baseadas na técnica de construção de vocabulário. Como estudo de caso são desenvolvidas aplicações para a avaliação da utilização de operadores clássicos de construção de vocabulário na resolução do problema de seqüenciamento da produção de carros.

O problema de seqüenciamento da produção de carros é um problema combinatório proposto a partir de necessidades reais da indústria. Sua formulação, definida pela fábrica de carros Renault, foi divulgada no concurso Challenge ROADEF'2005 [24], onde competiram equipes do mundo inteiro. Foram utilizadas diversos métodos de otimização na resolução das instâncias escolhidas para o concurso. Os resultados obtidos no concurso estão disponíveis

on-line [24] e podem ser utilizados para avaliar novas técnicas. Nenhuma equipe utilizou construção de vocabulário, cuja aplicação já se mostrou eficiente na resolução de problemas de seqüenciamento. Uma vez que essa técnica necessita de um conjunto de soluções geradas previamente, serão utilizadas as heurísticas propostas por Ribeiro et al. [26], descritas com mais detalhes em Rocha [29].

A presente dissertação está organizada conforme apresentado a seguir. No Capítulo 2, apresenta-se a técnica de construção de vocabulário acompanhada de uma revisão bibliográfica. No Capítulo 3, é proposto o *framework* para heurísticas baseadas em construção de vocabulário, definindo-se os pontos fixos e os flexíveis. No Capítulo 4, o problema de seqüenciamento de carros (PSC) é definido e é apresentada uma estratégia algorítmica para sua resolução. No Capítulo 5, são apresentados experimentos computacionais com as implementações das heurísticas propostas para resolver o PSC. Finalmente, no Capítulo 6, são feitas conclusões sobre o trabalho desenvolvido e são propostos trabalhos futuros.

2
Construção de vocabulário

A partir de pesquisas sobre a heurística busca tabu, Glover [11] propôs a técnica de construção de vocabulário. Essa técnica está relacionada com heurísticas que visam a criação de novas soluções em função das características de outras, como em análise alvo [11] e reconexão por caminhos [14]. Segundo Glover [11], em análise alvo pode-se implicitamente combinar ou subdividir atributos para melhorar a decisão de um movimento na busca tabu. Já em reconexão por caminhos, buscam-se novas soluções explorando trajetórias no espaço de soluções que conectam soluções de elite.

A idéia básica de construção de vocabulário consiste em gerar um conjunto de fragmentos de soluções e recombiná-los até obterem-se soluções completas [12]. Identificam-se fragmentos significativos das soluções, ao invés de só focar em soluções completas. O conjunto desses fragmentos é progressivamente enriquecido e combinado para criar fragmentos maiores, até finalmente produzirem soluções completas.

Este processo recebe seu nome por analogia com o processo de construção de vocabulário, em que há o enriquecimento do conjunto de palavras conhecidas sempre que for necessário representar um novo conhecimento. Construções significativas em cada nível podem ser visualizadas como representações de "palavras de ordem superior", assim como em linguagem natural geram-se novas palavras para substituir coleções de palavras que personificam conceitos úteis [15]. As palavras contêm características das soluções, que podem representar uma solução completa ou apenas fragmentos delas. Um conjunto de palavras, através de alguma combinação, forma uma frase. Uma sentença é uma frase que representa uma solução completa. Na analogia com a técnica de construção de vocabulário, combinar fragmentos de soluções até obter soluções completas é como combinar palavras em frases até formar-se uma sentença completa.

O motivo subjacente à construção de vocabulário é obter vantagem dos contextos onde certas configurações parciais de soluções ocorrem freqüentemente como componentes de boas soluções completas. Além disso, uma estratégia de procurar "boas configurações parciais" (bons elementos de voca-

bulário) e tratá-las como uma unidade pode ajudar a contornar a explosão combinatorial, que potencialmente resulta da simples manipulação dos elementos mais primitivos por si só.

Em uma classificação mais recente [12], a técnica de construção de vocabulário pode ser vista como um procedimento de programação com memória adaptativa, relacionada com métodos baseados em reconexão por caminhos e algoritmos evolucionários. Ainda segundo Glover [14], a construção de vocabulário pode ser qualificada como uma instância de reconexão por caminhos. Há dois objetivos principais: identificar uma boa coleção de pontos de referência e identificar caminhos em um ou mais espaços de vizinhança que unirão componentes destas soluções parciais, com modificações adequadas, para produzir soluções completas.

O diferencial desta técnica é como as soluções são consideradas. Na visão de construção de vocabulário, analisar um conjunto de soluções é tratá-lo como um texto, de modo a descobrir combinações de características compartilhadas pelas soluções. As combinações de características, que são significantes o bastante, são qualificadas como unidade de vocabulário e são tratadas como novos atributos. As unidades de vocabulários podem ser combinadas em unidades maiores, como uma base para a construção de novas soluções.

2.1 Descrição geral

Uma visão geral do processo de construção de vocabulário está ilustrada na Figura 2.1 [15].

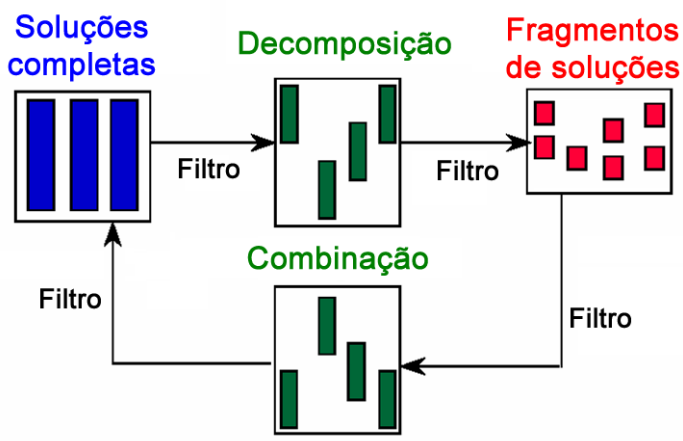

Figura 2.1: Processo de construção de vocabulário

Para exemplificar o processo de construção de vocabulário é apresentada a estratégia definida por Glover [11, 14]. As operações para encontrar palavras e formar frases, apresentadas a seguir, são simples e aplicáveis a diversos problemas combinatórios.

Um atributo é representado em uma palavra através de uma atribuição $y_j = p$, que deve ser lida como o atributo j da palavra y possuindo valor p. Uma palavra é composta por diversos atributos. Pelo fato de sua dimensão e codificação poderem ser diferentes das soluções do problema, justifica-se diferenciar uma solução de uma palavra. Uma solução pode ser mapeada para uma palavra e vice-versa.

Sejam $\mathcal{U}$ o conjunto de atributos e y' e y'' duas palavras. Define-se o operador de *interseção* pela regra:

$$INT(y', y'') = z$$

$$z_u = \begin{cases} y'_u, & \text{se} \quad y'_u = y''_u \\ *, & \text{se} \quad y'_u \neq y''_u, \end{cases}$$

aplicada a cada atributo $u \in \mathcal{U}$.

Para avaliar a qualidade das palavras são definidas algumas medidas. A função $Size(y)$ indica a quantidade de atributos de y que são diferentes de "$*$", ou seja, o tamanho de uma palavra. Dadas duas palavras y e x, o operador $\subseteq$ é definido de forma que $y \subseteq x$ implica que $y_u = x_u$ ou $y_u = *$, para todo $u \in \mathcal{U}$. Com este operador, define-se o conjunto:

$$Enclosure(y, \mathcal{Y}) = \{x \in \mathcal{Y} \,|\, y \subseteq x\}\,.$$

Deste modo, $EncValue(y, \mathcal{Y}) = |Enclosure(y, \mathcal{Y})|$ indica a quantidade de palavras do conjunto $\mathcal{Y}$ que contêm a palavra y.

A partir dessas definições, Glover sugere duas abordagens para a construção de um método para buscar palavras em um conjunto de soluções. Vale lembrar que as soluções são mapeadas em palavras e a diferença entre palavra e solução está no fato de uma palavra poder representar uma solução incompleta. Uma palavra é obtida pela sucessiva aplicação do operador de interseção as soluções obtidas de um conjunto de soluções elite $\mathcal{X}$. A primeira abordagem prima por obter palavras que possuam o maior valor possível para $EncValue(y, \mathcal{X})$, respeitando $Size(y) \geq s_{min}$, onde s_{min} é um parâmetro que estabelece o menor tamanho de uma palavra. Ou seja, uma palavra obtida a partir do máximo possível de interseções de soluções cujo tamanho seja maior do que um dado valor mínimo. A outra abordagem busca palavras, por exemplo a palavra y, que possuam o maior valor possível para $Size(y)$, respeitando

a restrição $EncValue(y, \mathcal{X}) \geq v_{min}$, onde v_{min} é um parâmetro que estabelece a menor quantidade de soluções utilizadas para se obter a palavra y.

A solução de um problema de otimização combinatória pode ser freqüentemente representada por uma combinação ou uma permutação de seus elementos. Por exemplo, no problema da mochila um elemento x_i pode pertencer à solução ($x_i = 1$) ou não ($x_i = 0$), podendo esta relação de pertinência ser um atributo da solução. No problema do caixeiro viajante, um atributo interessante é a precedência dos elementos na solução, onde $x_{ij} = 1$ se na solução do problema a cidade i for visitada antes de j; $x_{ij} = 0$ caso contrário. Os problemas de seqüenciamento podem ser representados dessa última forma.

A seguir é exemplificado o funcionamento do operador INT sobre um conjunto de soluções que foram mapeadas em palavras. Determina-se que uma palavra é válida se seu tamanho for maior que um dado limite. Dado o conjunto de palavras apresentado na Tabela 2.1, uma combinação da aplicação do operador INT é apresentado na Tabela 2.2.

y_1	0	0	1	1	0	0	0	0	1	0
y_2	1	0	1	1	0	0	0	0	0	0
y_3	0	0	1	1	1	0	0	0	0	0
y_4	1	0	0	1	1	0	0	0	0	0
y_5	0	0	1	0	0	0	0	1	1	0

Tabela 2.1: Conjunto de soluções

$INT(y_1, y_4)$	$*$	0	$*$	1	$*$	0	0	0	$*$	0
$INT(INT(y_1, y_4), y_3)$	$*$	0	$*$	1	$*$	0	0	0	$*$	0
$INT(INT(INT(y_1, y_4), y_3), y_5)$	$*$	0	$*$	$*$	$*$	0	0	$*$	$*$	0

Tabela 2.2: Utilização do operador INT

Para exemplificar o que este resultado pode significar, suponha que este conjunto represente soluções do problema da mochila. Cada posição no vetor representa a pertinência dos objetos, sendo "1" para presente, "0" para ausente e "$*$" para não se sabe. A palavra obtida $[*, 0, *, *, *, 0, 0, *, *, 0]$ identifica elementos que não pertencem a boas soluções. A partir deste novo conhecimento, pode-se definir um novo problema onde há um conjunto reduzido de elementos a serem escolhidos.

Um exemplo onde o tamanho mínimo de uma palavra válida é seis e a seqüência utilizada para compor as palavras é y_2, y_3, y_4, y_5, y_1 está apresentado na Tabela 2.3.

A palavra w_3 não é válida, pois possui tamanho quatro, que é menor do que o valor definido para uma palavra válida. Esta restrição é útil para evitar que se trabalhe com fragmentos de solução muito pequenos.

$w_1 = INT(y_2, y_3)$	$*$	0	1	1	$*$	0	0	0	0	0
$w_2 = INT(w_1, y_4)$	$*$	0	$*$	1	$*$	0	0	0	0	0
$w_3 = INT(w_2, y_5)$	$*$	0	$*$	$*$	$*$	0	0	$*$	$*$	0
$w_4 = INT(w_2, y_1)$	$*$	0	$*$	1	$*$	0	0	0	1	0

Tabela 2.3: Exemplo da aplicação do operador INT

Dado um conjunto de palavras, essas podem ser combinadas até formarem frases completas através do operador de *interseção estendido*. As frases completas são chamadas de sentenças. Sejam $\mathcal{U}$ o conjunto de atributos e y' e y'' duas palavras. Define-se o operador de *interseção estendido* pela regra:

$$EINT(y', y'') = z$$

$$z_u = \begin{cases} y'_u, & \text{se} \quad y''_u = * \\ y''_u, & \text{se} \quad y'_u = * \\ y'_u, & \text{se} \quad y'_u = y''_u \\ \#, & \text{se} \quad y'_u \neq y''_u, \end{cases}$$

aplicada a cada atribudo $u \in \mathcal{U}$.

Para exemplificar o uso do operador $EINT$, considere o seguinte conjunto de palavras representado na Tabela 2.4. Um exemplo da combinação das palavras y_1, y_5 e y_2 está apresentado na Tabela 2.5.

y_1	0	$*$	1	1	0	$*$	0	$*$	$*$	0
y_2	$*$	0	1	1	$*$	0	$*$	0	$*$	0
y_3	0	$*$	1	$*$	$*$	0	$*$	$*$	0	$*$
y_4	1	0	$*$	1	$*$	$*$	0	0	0	0
y_5	$*$	$*$	1	$*$	0	$*$	$*$	1	1	$*$

Tabela 2.4: Conjunto de palavras

$EINT(y_1, y_5)$	0	$*$	1	1	0	$*$	0	1	1	0
$EINT(EINT(y_1, y_5), y_2)$	0	0	1	1	0	0	0	0	1	0

Tabela 2.5: Utilização do operador $EINT$

Um agregado de palavras formado a partir de sucessivas aplicações do operador $EINT$ recebe o nome de frase. A frase $EINT(y_1, y_5)$ é chamada de frase incompleta enquanto $EINT(EINT(y_1, y_5), y_2)$ de sentença (ou frase completa).

Frases podem ter valores inconsistentes ('#') que podem ser considerados como inconsistências sintáticas. Um exemplo de frase com valores inconsistentes é apresentado na Tabela 2.6.

$EINT(y_1, y_3)$	0	$*$	1	1	0	0	0	$*$	0	0
$EINT(EINT(y_1, y_3), y_4)$	#	0	1	1	0	0	0	0	0	0

Tabela 2.6: Utilização do operado $EINT$ gerando frase inconsistente

Além dessas inconsistências sintáticas, ainda podem ocorrer inconsistências semânticas. Por exemplo, no problema da mochila pode-se obter uma sentença completa que represente uma solução inviável em relação às restrições do problema. Apesar disso, assim como frases incompletas, as sentenças inconsistentes também contêm informações significativas sobre soluções para o problema.

Uma sentença que não possui qualquer tipo de inconsistência é dita significativa. As sentenças significativas são soluções viáveis para o problemas. Eventualmente, elas ainda podem ser aperfeiçoadas por outros métodos.

2.2 Revisão da literatura

Apesar da técnica de construção de vocabulário ainda ser pouco estudada, é possível encontrar na literatura algumas de suas aplicações.

Um dos trabalhos pioneiros em construção de vocabulário foi o de Rochat e Taillard [28]. A técnica foi aplicada na resolução do problema de roteamento de veículos (VRP, do inglês *Vehicle Routing Problem*), em sua versão elementar (CVRP) e na variante com janelas de tempo (CVRPTW). O problema elementar de roteamento de veículos idênticos consiste em determinar rotas para veículos que partem de um único depósito e devem atender uma seqüência de consumidores de um conjunto $C = \{1, \ldots, n\}$, onde a distância entre os consumidores i e j é c_{ij}, entregando quantidades de mercadoria q_i a cada consumidor $i = 1, \ldots, n$, de modo que:

- seja minimizada a distância percorrida pelos veículos;
- somente um veículo visite cada consumidor; e
- seja respeitado o limite Q de carga de mercadorias de cada veículo.

No problema de roteamento de veículos com janelas de tempo, as distâncias entre os consumidores c_{ij} são vistas como o tempo gasto de ir de i até j e cada consumidor i requer um tempo de serviço t_i. A visita ao consumidor i deve se realizar durante uma janela de tempo $[b_i, e_i]$ pré-estabelecida.

Uma das perspectivas que foram utilizadas no desenvolvimento do método foi determinar quais são as variáveis consistentes nas boas soluções encontradas durante a busca. Segundo os autores, uma variável consistente é aquela que se determina fortemente e freqüentemente em um valor específico

(ou em um intervalo curto). De um modo mais específico, quanto mais freqüentemente a variável receber um determinado valor nas melhores soluções, mais altamente se qualifica como consistente. Essa é a idéia de se identificar fragmentos significativos nas boas soluções.

O método proposto pelos autores possui duas etapas na sua execução. A geração de boas soluções e a criação do conjunto de fragmentos são os objetivos da primeira fase. Fragmentos de uma solução são definidos como as rotas de cada veículo. À medida que são geradas novas soluções, o conjunto de rotas é ampliado com as novas rotas encontradas. Espera-se que toda a informação necessária para criar novas soluções de boa qualidade exista nessas soluções iniciais, mas de um modo não aparente. No caso do VRP, essa informação está incluída nas rotas. Então, espera-se que a primeira fase crie um conjunto de rotas que inclua membros não muito diferentes dos que são encontradas em boas soluções. Não é irreal pensar que seja fácil criar boas rotas. O desafio é encontrar um conjunto de rotas cujos elementos sejam simultaneamente bons para todos os consumidores.

Na segunda etapa, a meta é extrair as rotas boas e melhorá-las. O autor parte do princípio que, se uma nova solução possui rotas que pertençam a uma boa solução conhecida, então o valor da função objetivo da nova solução é, provavelmente, melhor do que o de outra solução que não contenha essas rotas. Portanto, a segunda fase está relacionada com a extração de rotas de modo a intensificar a busca em novas soluções que contenham essas rotas.

Foi proposta uma técnica de pós-otimização, em que se tenta obter a melhor combinação das rotas, de modo a formar uma solução completa válida. Isso é feito através de um modelo de programação inteira. Segundo o autor, essa abordagem torna o problema mais difícil de ser resolvido, além de não acrescentar novas informações para o conjunto de rotas. O propósito dessa proposta foi obter soluções relaxadas, de tal forma que a solução pudesse ser viabilizada e melhorada por uma busca local. Nem sempre é possível reconstruir soluções completas a partir dos fragmentos conhecidos. Contudo, ainda é desejável soluções incompletas que possuam variáveis consistentes, pois estas podem ser convertidas em soluções completas e viáveis.

Esta é uma clara aplicação da idéia de construção de vocabulário. Geram-se boas soluções, a partir das quais encontram-se palavras. Essas palavras são combinadas em frases e, por fim, conseguem-se montar novas soluções.

Os resultados desse trabalho são expressivos, encontrando as melhores soluções conhecidas para diversas instâncias teste do VRP.

Na mesma época, Kelly e Xu [19] propuseram uma heurística baseada em particionamento de conjuntos para a resolução do VRP. Os autores

exploraram modos de usar uma geração simples com refinamento heurístico para construção de soluções, que sejam competitivas ou melhores do que as obtidas pelos algoritmos mais complicados.

Considera-se que uma solução pobre, produzida por uma heurística simples, pode conter um subconjunto de "boas" rotas. Baseado nisto, foi proposto um procedimento de duas fases como solução do VRP. Na primeira fase, heurísticas simples são utilizadas para gerar um número suficiente de rotas diferentes. Na segunda fase, identificam-se as "boas" rotas que são combinadas para formar uma solução através de uma heurística baseada em particionamento de conjuntos. O modelo de particionamento de conjuntos é utilizado para realizar a tarefa de consolidação na segunda fase. A heurística proposta requer o uso de técnicas sofisticadas de busca, pois o problema de particionamento de conjuntos (SPP, do inglês *set-partitioning problem*) é um problema NP–difícil. Para tal, foi desenvolvida uma heurística de busca tabu para solucioná-lo. Como heurísticas simples podem não produzir todas as rotas das melhores soluções e o modelo de particionamento de conjunto é resolvido de modo aproximado, aprimorou-se o processo criando novas rotas durante o processo de busca.

A fase de geração de rotas utiliza-se de heurísticas clássicas para o VRP. Após N rotas serem produzidas, emprega-se o modelo de particionamento de conjuntos para selecionar m rotas e montá-las em uma solução. A formulação matemática utilizada para a solução do problema é apresentada a seguir. Seja c_j o custo (nesse caso, o comprimento da rota) de uma rota j, e a_{ij} um valor binário igual a 1 se, e somente se, a rota j contém o consumidor i; e $a_{ij} = 0$ caso contrário. Define-se x_j como uma variável binária tal que $x_j = 1$ se, e somente se, a rota j for selecionada para a solução; $x_j = 0$ caso contrário. O modelo de programação inteira é formulado como:

$$\texttt{minimizar} \quad \sum_{j=1}^{N} c_j x_j \tag{2-1}$$

$$\texttt{sujeito a:} \sum_{j=1}^{N} a_{ij} x_j = 1, \quad i = 1, \ldots, n, \tag{2-2}$$

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

Nesta formulação, a função objetivo consiste em minimizar a soma dos comprimentos das rotas. A restrição (2-2) especifica que cada consumidor deve ser coberto pela solução somente uma única vez (essa restrição é relaxada na heurística utilizada, para permitir mais combinações).

Segundo os autores, a abordagem defendida por eles tem muitas similaridades com o método proposto por Rochat e Taillard [28]. Ambos os

métodos procuram combinar inteligentemente rotas geradas previamente por outro método. No trabalho apresentado pelos autores, é utilizado um conjunto enorme de rotas iniciais e as soluções são produzidas por um procedimento elaborado de atualização e recombinação de rotas, enquanto Rochat e Taillard usam um conjunto reduzido de rotas iniciais e criam novas soluções usando seleção probabilística combinada com um procedimento de melhoria local. Não obstante, esse método de busca local é um dos algoritmos mais eficientes conhecidos da literatura e também um resolvedor comercial de programação inteira (CPLEX) é utilizado para resolver o modelo de particionamento como pós-otimização.

Um método baseado em busca tabu para resolver o problema de escalonamento da produção de tiras quentes de aço foi proposto por Lopez, Carter e Gendreau [21] Este problema é baseado em um estudo de caso real para uma indústria canadense, com propósito de aplicação prática.

Segundo a análise desse problema pelos autores como um problema de programação matemática, há duas atividades importantes associadas. A primeira é relacionada com os pedidos que devem ser escolhidos para produção (seleção) e a segunda com o seqüenciamento dos produtos listados nos pedidos selecionados (seqüenciamento). A seleção dos pedidos pode ser representada como um problema da mochila. Determinar a seqüência de produção dos produtos selecionados corresponde ao bem conhecido problema do caixeiro viajante assimétrico. Para uma modelagem completa do problema, é apresentada uma formulação matemática através de uma generalização do caixeiro viajante com coleta de prêmios. Como são problemas de difícil solução exata, optou-se por uma heurística baseada em busca tabu.

O algoritmo é composto de duas fases. A primeira gera um conjunto de soluções iniciais. A segunda fase, chamada de canibalismo, utiliza esse conjunto para melhorar outras soluções. O canibalismo consiste em identificar blocos ruins na solução corrente e trocá-los por blocos bons de outras soluções, atendendo melhor aos requisitos do problema. Vale ressaltar a idéia básica utilizada: gerar diversas soluções de modo a aproveitar fragmentos significativos para se produzir novas soluções.

O problema de escalonar a produção de um produto que tem vários modelos em uma linha de montagem flexível foi tratado por Scholl, Klein e Domschke [30]. Para resolver esse problema foi desenvolvido um procedimento de busca tabu com uma estratégia de construção de vocabulário baseada em padrões. Unidades de vocabulário são definidas através de seqüências parciais de modelos dos produtos. Uma seqüência parcial é chamada padrão se a linha de produção retorna para seu estado inicial após processar os produtos da

seqüência. Um padrão é dito elementar se não existe uma decomposição em padrões menores. Esses padrões são gerados e escolhidos por um procedimento de geração de colunas e regras heurísticas de transformação, baseados numa reformulação do problema original.

Foi então definido o problema de combinação de padrões (PCP), que consiste em combinar padrões elementares para atender as demandas do problema original. Ele pode ser modelado como um problema de programação linear inteira. Se todos os padrões elementares necessários são conhecidos, a solução desse problema é solução para o problema original. Entretanto, essa abordagem se depara com a dificuldade principal do número de padrões elementares ser usualmente muito grande. Além do mais, é custoso identificar e excluir padrões não elementares a serem considerados na resolução do problema.

Desenvolveu-se uma abordagem mais apropriada baseada em construção de vocabulário, que permitiu produzir e combinar somente um subconjunto significativo de padrões. Primeiramente, é construído um vocabulário "difuso" que necessita ser refinado em fases subseqüentes. Um subconjunto de padrões é determinado pela solução da relaxação linear do PCP com um procedimento de geração de colunas. Embora as quantidades com que cada padrão é escolhido possam não ser inteiras, a solução do problema relaxado é transformada em uma solução do PCP através de um procedimento heurístico de arredondamento. Entretanto, esse procedimento não garante que todos os padrões elementares sejam produzidos. Para contornar esse inconveniente, a aplicação de uma busca tabu, que utiliza informações codificadas nos padrões encontrados anteriormente, permite aprimorar a seqüência de padrões que representa a solução para o problema original. Em comparação com os outros métodos conhecidos da literatura, a versão da busca tabu que utiliza a informação codificada no vocabulário domina as outras abordagens com respeito à qualidade da solução e velocidade computacional.

O procedimento de construção de vocabulário baseado em padrões apresentado é altamente influenciado pelo comprimento máximo do padrão. Enquanto o tempo computacional requerido para gerar o vocabulário cresce exponencialmente com um aumento do valor deste parâmetro, a complexidade do problema de seqüenciamento dos padrões diminui, e vice versa. Se o comprimento do padrão é grande, as decisões de arredondamento na viabilização de uma solução relaxada do PCP tendem a ser críticas com relação ao valor da função objetivo. Em contrapartida, os efeitos compensatórios relacionados com a utilização dos padrões podem não ser obtidos se o comprimento máximo do padrão for muito pequeno. Por exemplo, perde-se tempo considerando um

grande volume de unidades de vocabulário pouco significativas ao invés de aproveitar o conhecimento extraído de blocos maiores e mais significativos.

Um novo paradigma para otimização combinatória foi proposto por Toulouse, Thulasiraman e Glover [33], chamado de *Multi-Level Cooperative Search*. A proposta desse paradigma é uma estratégia para organizar, de um modo configurável, a rede de dependências entre programas cooperativos. Algoritmos cooperativos necessitam de mecanismos para compartilhar informação e melhorar o comportamento da busca em programas paralelos. A estratégia proposta usa princípios advindos de *scatter search* e construção de vocabulário para guiar a "combinação" da informação compartilhada e originada nas diferentes fontes. Assume-se que as soluções de problemas de otimização combinatória possam ser representadas como seqüências ou partições. Variáveis de decisão, que definem o valor da função objetivo, servem como uma escolha lógica para constituir os componentes de mais baixo nível, de uma estrutura de vários níveis. Progressivamente, combinam-se estas variáveis para prover maiores agregados de variáveis. Forma-se uma estrutura em diversos níveis, onde níveis superiores trabalham com agregados maiores que compõem as soluções.

Combs [6] propôs uma busca tabu adaptativa para solucionar o problema de escalonamento da tripulação de reabastecedores aéreos. O problema consiste em se determinar a tripulação que fará um vôo de reabastecimento de uma série de aeronaves, sujeito a diversas restrições. Por exemplo, cada aeronave que será reabastecida é atendida por apenas uma tripulação e após uma série de vôos as tripulações devem terminar seu turno na base de onde partiram. Existem diversas bases de onde partem as tripulações. Uma das abordagens desenvolvidas para solucionar o problema combina busca tabu adaptativa com o problema de particionamento de conjuntos. O sistema desenvolvido foi implementado em OpenTS, um *framework* Java para busca tabu [18]. Define-se como um segmento de vôo qualquer vôo sem parada entre pares de bases operacionais. Existem diversos pontos para reabastecimento entre a partida e a chegada nas bases operacionais. A partir dessa definição, o autor formulou um problema baseado no SPP seguindo a abordagem de Kelly e Xu [19]. A resolução do SSP foi feita pelo CPLEX, mas, muitas vezes, após várias horas de computação, não obteve-se o valor ótimo ou, pior, se esgotou a memória do resolvedor. Por isso, também foi desenvolvida uma heurística que a partir de soluções incompletas do SPP permite reconstruir e melhorar as soluções geradas na construção de vocabulário. Essa abordagem, baseada em construção de vocabulário, foi o último melhoramento realizado na busca tabu e possibilitou um aumento expressivo na robustez do método proposto.

Uma aplicação de técnicas de mineração de dados no contexto de oti-

mização foi apresentada por Ribeiro, Plastino e Martins [25]. Mineração de dados faz referência à extração de conhecimento novo e potencialmente útil a partir de um conjunto de dados em termos de padrões e regras. Os autores acreditam que técnicas de mineração de dados podem ser utilizadas para extrair padrões que representam características de soluções sub-ótimas de um problema de otimização combinatorial. Além disso, esses padrões podem ser utilizados para guiar a busca para soluções melhores em procedimentos metaheurísticos.

Foi proposta uma hibridização da metaheurística GRASP [7] que incorpora o processo de mineração de dados. Basicamente, após a heurística GRASP executar um número significativo de iterações, o processo de mineração de dados extrai padrões de um conjunto de soluções de elite que guia as iterações GRASP seguintes. Soluções são representadas por conjuntos de itens. Padrões são definidos como subconjuntos de itens que ocorrem num número significativo de soluções. O processo de obter estes padrões é o bem conhecido problema em mineração de dados chamado Mineração de Conjunto Freqüente de Itens (FIM, do inglês *Freqüente Itemset Mining*).

Foram apresentadas estratégias distintas pela variação de como se gera e utiliza o conjunto de padrões. Essas estratégias diferentes não foram propostas de uma vez, mas como uma evolução da pesquisa. Por exemplo, a primeira é chamada maior padrão híbrido. Ela utiliza apenas o maior padrão encontrado num conjunto de padrões. Esse conjunto é formado pelos padrões presentes em pelo menos um percentual de elementos do conjunto de soluções elite. A segunda estratégia, chamada maior suporte híbrido, procura pelo padrão mais freqüente no conjunto elite dado um tamanho mínimo. Assim, foram variando o modo de utilização dos algoritmos na mineração de dados para obter padrões diferentes. Os padrões identificados são utilizados como ponto de partida para a fase de construção GRASP e, posteriormente, refinados na busca local.

Seguindo a proposta de Glover [14], Fernandes [8] utilizou a técnica de construção de vocabulário em suas heurísticas para resolução do problema de seqüenciamento de DNA por hibridação, cuja solução é uma seqüência de DNA formada a partir de um encadeamento de subseqüências. Essa seqüência pode ser vista como um conjunto de arestas que ligam vértices representando subseqüências de DNA. O conjunto de arestas é representado por um vetor de adjacência. A partir de um conjunto de soluções de elite, as soluções são mapeadas em vetores de adjacência onde se utiliza os operadores INT e $EINT$ para encontrar e combinar palavras. Entretanto, as soluções produzidas pelo método podem conter algumas inconsistências lógicas como, por exemplo, várias arestas direcionadas para uma mesma subseqüência de DNA. Para

contornar essa dificuldade, cada solução produzida pelo método é analisada e, se necessário, é aplicado um algoritmo de viabilização das soluções. Os resultados obtidos demonstram os benefícios da utilização da técnica como complementar às outras heurísticas utilizadas.

Na mesma linha, Aloise [1] agregou construção de vocabulário às suas heurísticas para o projeto de redes com funções de custos discretas. Cada solução desse problema é uma atribuição de valores para as arestas de uma rede, de modo a respeitar as restrições de largura de banda e demanda dos nós, entre outras. Uma representação natural é um vetor em que cada posição representa o valor alocado para uma aresta. A partir de um conjunto de soluções de elite, encontram-se palavras que por sua vez são combinadas em frases, conforme a estratégia proposta por Glover. Observou-se que o método produz muitas frases incompletas, embora haja informação relevante sobre a solução ótima. Para aproveitar estas frases, utiliza-se um modelo de programação inteira onde os valores encontrados são fixados e apenas as lacunas são determinadas pelo resolvedor. A técnica foi utilizada de modo complementar para compor a estratégia, juntamente com outras heurísticas propostas.

3
Framework

Para facilitar a implementação e comparação de algoritmos, a tecnologia de *frameworks* [23] é uma solução que já demonstrou ser muito eficaz [34]. Na literatura existem diversos trabalhos sobre o desenvolvimento de *frameworks* e bibliotecas para *software* de otimização [2, 3, 4, 5, 34], porém, nenhum para construção de vocabulário. Andreatta [2] desenvolveu um *framework* para a implementação de heurísticas de busca local para problemas de otimização combinatória [3, 4, 5]. Atualmente, as boas práticas de engenharia de software, tais como sistemas orientados a objetos e padrões de projeto [9], pautam o desenvolvimento de *frameworks*.

Construção de vocabulário pode ser aplicada de diversas maneiras na resolução dos problemas. Como uma técnica proposta para ser uma memória de longo prazo, ela pode ser utilizada diversas vezes durante a execução da busca [1, 21, 30] ou, simplesmente, como pós-otimização [8, 28]. O conjunto de soluções armazenadas pode ser construído segundo diversos critérios [16]. Para identificar as porções de solução, podem ser utilizados métodos baseados em mineração de dados [25], representações específicas da solução [6, 19, 21, 28, 30] ou, como a proposta original [11], através da interseção [1, 8] entre soluções. Para gerar novas soluções, na proposta original, é realizado um procedimento inverso chamado de interseção estendida [12]. Um outro modo de formar soluções é fixar boa parte da solução e completá-la por outro método, exato [1] ou heurístico [21, 28, 30]. Também, pode-se aplicar a resolução exata de um problema auxiliar, como o de partição de conjuntos, para formar novas soluções [19]. Nem sempre é possível usar todas esses procedimentos a um determinado problema, pois cada um possui características particulares que indicam quais devem ser mais apropriadas e produzem melhores resultados.

Visto que há diversas maneiras de programar heurísticas baseadas em construção de vocabulário, uma metodologia para implementar e comparar essas diferentes abordagens é bastante desejável. Com o intuito de facilitar a geração dessas heurísticas, este trabalho propõe um *framework*.

Além de considerar a diversidade de aplicação da técnica, o *framework* foi projetado para facilitar sua utilização. A descrição da construção de

vocabulário, apresentada na Seção 2.1, especifica operadores clássicos para encontrar e combinar palavras, os operadores INT e $EINT$ respectivamente. Esses operadores estão disponíveis no *framework*, possibilitando uma rápida experimentação da técnica.

Uma questão relevante para as heurísticas baseadas em construção de vocabulário são os repositórios de dados, por exemplo, o repositório de boas soluções. O gerenciamento do repositório deve considerar o número de soluções armazenadas, sua diversidade, sua qualidade, quais soluções entram e, eventualmente, quais saem para dar lugar a novas soluções [16]. Por ser intrínseco a vários métodos heurísticos, o repositório é um objeto estudado em diversas heurísticas, em especial, naquelas baseadas em memória adaptativa. Greistorfer e Voß [16] apresentam uma extensa revisão bibliográfica relacionando diversos métodos baseados em memória adaptativa, incluindo construção de vocabulário, seguida por questões sobre utilização e diretrizes para implementação desses repositórios.

Arquitetura do *framework* foi projetada para permitir a troca das funções para gerar palavras e montar frases, além de um gerenciamento configurável dos repositórios. Desse modo, facilita-se a experimentação de diversas implementações e ajustes a fim de melhorar seu desempenho. Como também permite a troca em tempo de execução das funções para gerar palavras e montar frases, possibilita composição de diferentes implementações permitindo adaptabilidade a instâncias especiais de problemas.

Neste capítulo é apresentado o *framework* proposto com sua parte imutável, apresentada como pontos fixos, e a parte configurável, chamada de pontos flexíveis. Por utilizar o paradigma de programação orientado a objetos, é apresentado um diagrama de classes relativo à implementação. Nesse diagrama, visualizam-se os pontos fixos e os pontos flexíveis além do relacionamento entre os diversos componentes da arquitetura. A seção algoritmos apresenta a implementação dos operadores INT e $EINT$. Por fim, são apresentados alguns padrões de projeto que foram utilizados no desenvolvimento do *framework*.

3.1 Pontos fixos

– Repositório de soluções

O gerenciamento das soluções armazenadas é definido por diretrizes. São consideradas informações como quantidade, diversidade e qualidade das soluções, além de critérios para a escolha de quais soluções entram e quais devem, eventualmente, sair para dar lugar às novas.

– Operador *interseção* INT

São fornecidas duas implementações deste operador para extração de palavras a partir de um repositório de vetores de números inteiros. A primeira obtém palavras a partir do máximo possível de interseções de soluções cujo tamanho da palavra gerada seja maior do que um dado valor mínimo. A outra abordagem busca as palavras mais extensas, e por isso com menor quantidade de soluções utilizadas respeitando um valor mínimo de soluções. A primeira combina mais soluções para obter palavras mais consistentes e a segunda combina menos soluções para obter palavras mais extensas.

– Operador *interseção estendido* $EINT$

É uma implementação deste operador para combinar palavras a partir de um repositório de palavras geradas pelo operador INT.

– Representação de soluções

É fornecida uma representação padrão através de um vetor de números inteiros, pois essa é uma forma genérica de representar soluções de problemas de otimização combinatória. Essa representação é adequada para a implementação dos métodos de extração (INT) e composição de palavras ($EINT$).

3.2 Pontos flexíveis

– Representação de soluções

A variação na representação de soluções permite a utilização do *framework* na geração de heurísticas para a resolução de problemas que necessitem de representação especial. Suporta-se qualquer representação de solução, desde que os outros pontos flexíveis sejam implementados para suportar a estrutura desejada.

– Método para encontrar palavras

É possível variar o procedimento para encontrar palavras. Por exemplo, pode-se utilizar técnicas de mineração de dados com esse objetivo [25].

– Método para formar frases

As frases são formadas a partir das palavras, que podem ser combinadas por métodos heurísticos ou exatos. Um exemplo de metodologia para combinar palavras de modo exaustivo é utilizar um resolvedor para o problema de partição de conjuntos, cuja solução é um conjunto de palavras que produz a melhor frase [19].

– Gerenciamento do repositório

Como deve-se armazenar uma grande quantidade de soluções, é importante que o gerenciamento seja personalizado ao contexto do problema. A partir das definições do repositório de soluções, deve-se especificar funções de gerenciamento como, por exemplo, a função que determina se uma solução deverá entrar no repositório ou não.

3.3 Padrões de projeto utilizados

No projeto do *framework* foram utilizados três padrões de projeto: *template method*, *strategy* e *adapter*. A seguir, é apresentado um resumo baseado na definição desses padrões [9].

3.3.1 Template Method

Define o esqueleto de um algoritmo em uma operação, delegando alguns passos para subclasses. *Template method* possibilita subclasses redefinirem alguns passos de um algoritmo sem mudar a estrutura desse algoritmo. A estrutura deste padrão é apresentada na Figura 3.1.

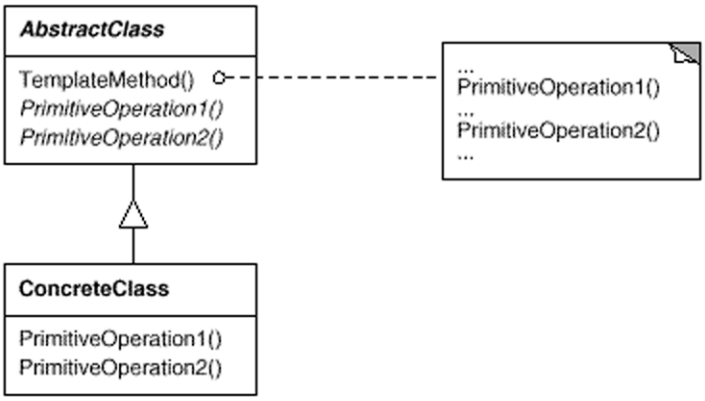

Figura 3.1: Estrutura do padrão *template method*

– **Participantes**

  – `AbstractClass`

    * Define operações primitivas abstratas onde subclasses concretas implementam os passos de um algoritmo.
    * Implementa um método *template* definindo o esqueleto de um algoritmo. O método *template* chama operações primitivas definidas em `AbstractClass` ou em outros objetos.

  - `ConcreteClass`
    * Implementa as operações primitivas para executar passos específicos de subclasse do algoritmo.

- **Colaboração**
  - `ConcreteClass` delega para `AbstractClass` a implementação dos passos invariantes do algoritmo.

Template method é uma técnica fundamental para reuso de código, que leva a uma estrutura de controle invertida que se refere a como uma classe pai chama as operações das subclasses e não do modo inverso. Essa técnica é particularmente importante em bibliotecas de classes, pois é como se realiza a refatoração de comportamentos comuns em classes.

3.3.2 Strategy

Define uma família de algoritmos, encapsulando cada um deles e tornando-os intercambiáveis. *Strategy* possibilita ao algoritmo variar de modo independente dos clientes que o usa. A estrutura deste padrão é apresentada na Figura 3.2.

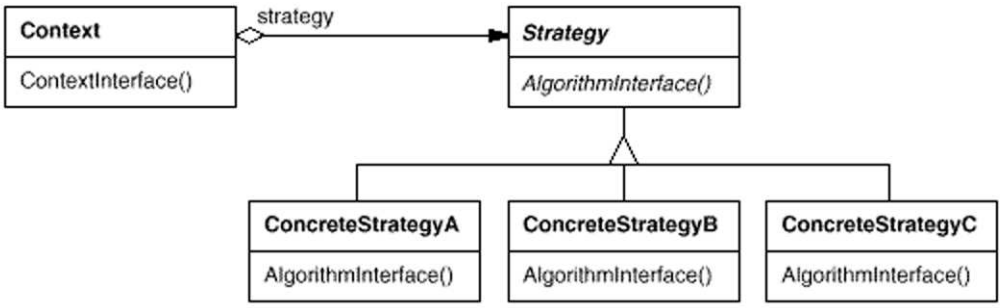

Figura 3.2: Estrutura do padrão *strategy*

- **Participantes**
  - `Strategy`
    * Declara uma interface comum a todos algoritmos suportados. `Context` usa esta interface para chamar o algoritmo definido por uma `ConcreteStrategy`.
  - `ConcreteStrategy`
    * Implementa o algoritmo usando uma interface `Strategy`.

  - `Context`
    * É configurado com um objeto `ConcreteStrategy`.
    * Mantém uma referência para um objeto `Strategy`.
    * Deve definir uma interface que possibilite `Strategy` acessar seus dados.

- **Colaboração**
  - `Strategy` e `Context` interagem para implementar o algoritmo escolhido. Um contexto deve fornecer para a estratégia todos os dados requeridos pelo algoritmo. Alternativamente, o contexto pode se passar como argumento para as implementações de `Strategy`. Isso possibilita à estratégia chamar o contexto quando for necessário.
  - Um contexto encaminha requisições de seus clientes para sua estratégia. Os clientes geralmente criam e passam um objeto `ConcreteStrategy` para o contexto. Desse modo, clientes interagem somente com o contexto. Há sempre uma família de classes `ConcreteStrategy` para um cliente escolher alguma específica.

Como estratégias podem prover diferentes implementações do mesmo comportamento, o cliente deve escolher aquelas que melhor lhe atendem em relação às diferentes relações na utilização de tempo e espaço. O padrão tem uma inconveniência potencial, já que o cliente deve entender como as estratégias diferem antes de selecionar a mais apropriada. Clientes podem ficar expostos a questões de implementação. Entretanto, deve-se usar o padrão *Strategy* quando a variação no comportamento é relevante para os clientes.

3.3.3 Adapter

Converte a interface de uma classe em outra interface esperada pelos clientes. Este padrão permite que classes trabalhem juntas apesar de terem interfaces incompatíveis.

Deve-se aplicar o padrão *adapter* quando for preciso usar uma classe existente e sua interface não for compatível com sua necessidade. Outra aplicação é na criação de uma classe reutilizável que coopera com classes não relacionadas ou não previstas, ou seja, classes que não tenham necessariamente interfaces compatíveis.

Adaptadores variam na quantidade de trabalho necessária para adaptar interfaces de `Adaptee` para `Target`. Há uma gama de possibilidades que vai desde uma simples conversão de interface (por exemplo, trocar o nome de operações) até converter representações e um conjunto inteiro de operações

diferentes. A quantidade de trabalho que `Adapter` realiza depende de quanto a interface `Target` é semelhante à de `Adaptee`.

Uma classe é mais reutilizável quando se minimiza as suposições que outras classes devem fazer para usá-la. Realizar adaptação de interface através de uma classe elimina a suposição de que outras classes tenham a mesma interface. Em outras palavras, adaptação de interface permite incorporar classes em sistemas existentes que podem ter interfaces diferentes.

A estrutura deste padrão é apresentada na Figura 3.3.

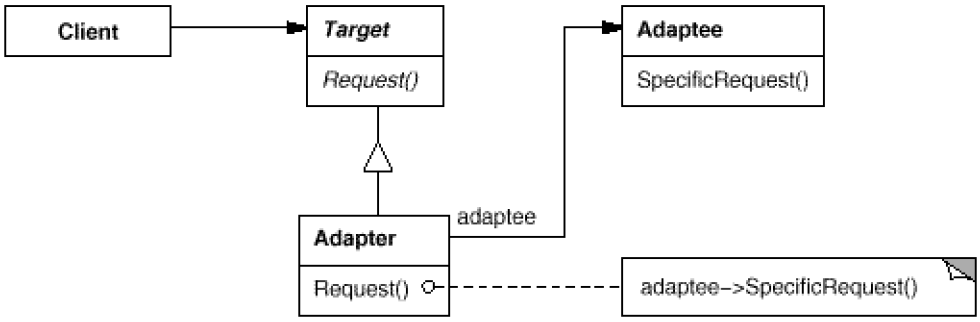

Figura 3.3: Estrutura do padrão *adapter*

- **Participantes**
  - `Target`
    - Define a interface específica do domínio de `Client`.
  - `Client`
    - Colabora com objetos conforme a interface de `Target`.
  - `Adaptee`
    - Define uma interface existente que necessita de adaptação.
  - `Adapter`
    - Adapta a interface de `Adaptee` para a interface de `Target`.
- **Colaboração**
  - Clientes chamam operações de uma instância de `Adapter`. Por sua vez, o adaptador chama as operações de `Adaptee` que realizam a requisição.

3.4 Diagrama de classes

O desenvolvimento do *framework* foi pautado pelas premissas de baixo acoplamento das heurísticas geradas com outros métodos, facilidade na variação das implementações e flexibilidade na representação de dados. Uma heurística gerada pelo *framework* é acessada através de uma classe principal, que coordena a interação entre os diversos componentes (repositórios e métodos para extração e combinação de palavras). O diagrama de classes apresentado na Figura 3.4 mostra essa classe principal, denominada `VocabularyBuilding`. As funções de extração e combinação de palavras são definidas por estratégias representadas, respectivamente, nas classes `DecomposeStrategy` e `GrowStrategy`, que podem ser facilmente alteradas inclusive em tempo de execução. Os repositórios são obtidos a partir de uma classe genérica, chamada `Pool`, que armazena objetos do tipo `PoolElement`. Através de classes que herdam `PoolElement`, pode-se armazenar diferentes tipos de dados.

A classe `VocabularyBuilding` possui referências para os repositórios (`Pool`) de soluções, palavras e frases além das estratégias de extração (`DecomposeStrategy`) e combinação (`GrowStrategy`) de palavras. Um procedimento de construção de vocabulário consiste em extrair palavras do repositório de boas soluções, armazenando-as no repositório de palavras, e em seguida combiná-las, armazenando-as no repositório de frases. A execução da heurística é disparada através da chamada ao método `execute`. Para permitir a execução de procedimentos específicos antes e após a heurística de construção de vocabulário, são definidos os métodos virtuais `preOpt` e `posOpt`, conforme o padrão de projeto *template method*.

Por exemplo, pode-se utilizar uma heurística construtiva para gerar soluções que são adicionadas no repositório de boas soluções, definida em `preOpt`, e uma busca local para refinar as soluções presentes no repositório de frases, definida em `posOpt`. A chamada ao método `execute` inicia a execução pelo procedimento definido em `preOpt`, gerando o repositório de soluções, seguido pelas estratégias para extração e combinação de palavras, e termina com a busca local `posOpt`.

Para gerar soluções também pode ser utilizado algum método externo à heurística, utilizando o método `addSolution` para adicioná-las no repositório de boas soluções. As soluções de problemas, assim como as palavras e as frases, devem ser objetos de uma classe que herda `PoolElement`, pois é um requisito por definição dos repositórios `Pool`.

A inserção de um elemento no repositório está condicionada a uma estratégia de controle, definida em `InputFunctionStrategy`. Essa estratégia

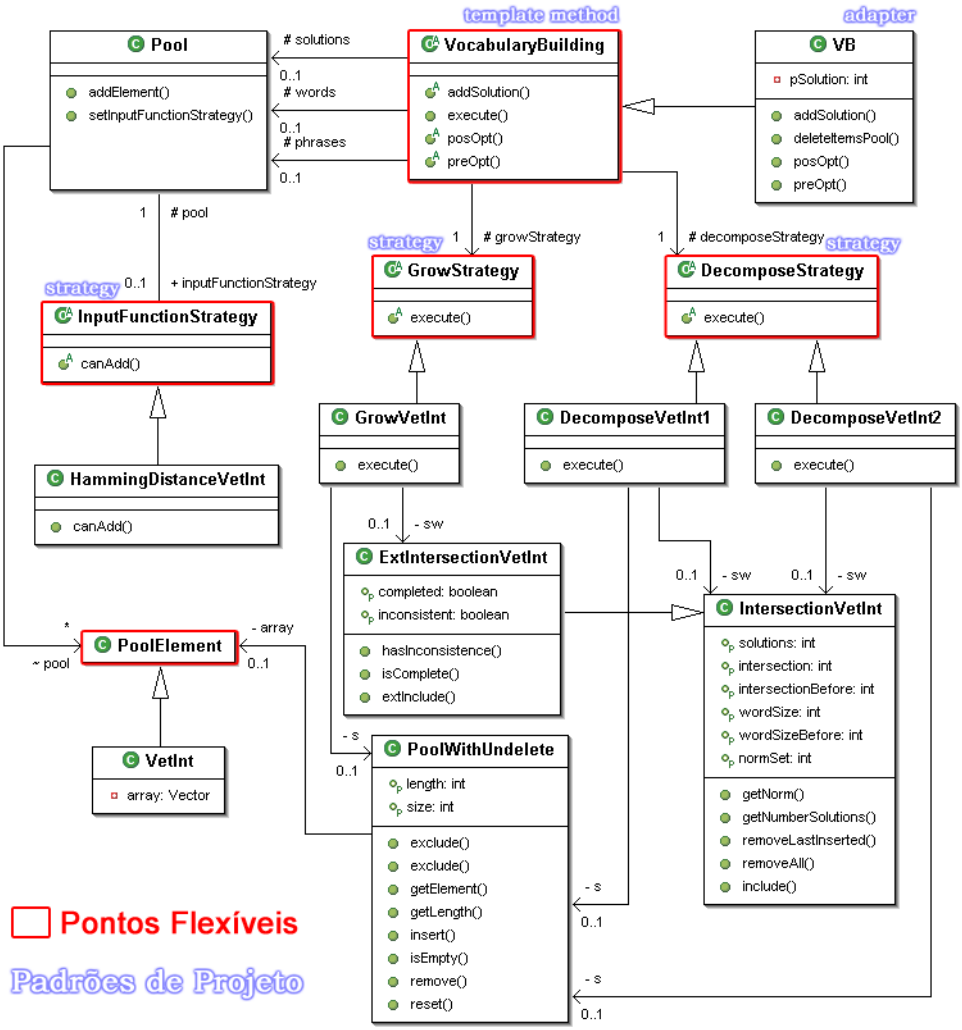

Figura 3.4: Diagrama de classes

deve avaliar, considerando diretrizes de gerenciamento, se um elemento pode ser aceito no repositório e, se necessário, realizar a substituição de outro já existente. As diretrizes de gerenciamento são determinadas pela quantidade, diversidade e qualidade que se espera dos elementos no repositório. A classe `HammingDistanceVetInt` implementa a aprovação de uma inserção através da análise da distância de Hamming do elemento a ser inserido em relação aos demais presentes no repositório, tal que a menor distância seja maior que um dado valor. A distância de Hamming entre dois vetores é o número de posições nas quais os valores correspondentes são diferentes. Desse modo, garante-se a diversidade dos elementos no repositório.

Na classe `VocabularyBuiliding` emprega-se o padrão de projeto *adapter* para facilitar o acoplamento das heurísticas de construção de vocabulário a outros procedimentos. Através do método `addSolution`, deve-se realizar uma conversão da representação da solução utilizada nas outras heurísticas para

uma representação mais adequada à técnica de construção de vocabulário. Para utilizar os operadores fornecidos com o *framework*, deve-se converter as soluções do problema em objetos da classe `VetInt`.

A extração e combinação de palavras são definidos como estratégias da heurística de construção de vocabulário conforme o padrão de projeto *strategy*. Estratégias para identificar palavras são implementadas através de classes que herdam `DecomposeStrategy`. Essas classes recebem um repositório de soluções (objeto da classe `Pool`) e geram um repositório de palavras. De modo análogo, estratégias para combinar palavras são implementadas através de classes que herdam `GrowStrategy`, onde recebem um repositório de palavras e geram um *pool* de frases.

A classe `VetInt` é uma implementação para representar vetores de números inteiros, que são utilizados pelas implementações baseadas nos operadores clássicos para extração (INT) e composição($EINT$) de palavras, definidos na Seção 2.1. As classes `DecomposeVetInt1` e `DecomposeVetInt2` são implementações baseadas no operador INT que realizam a extração de palavras. A composição de palavras é realizada pela classe `GrowVetInt` que implementa um método baseado no operador $EINT$. A descrição dos algoritmos presentes nessas classes é apresentada na Seção 3.5.

São definidas três estruturas de dados de suporte à implementação dos métodos de extração e combinação de palavras. Essas estruturas foram especialmente desenvolvidas para obter um desempenho melhor na execução dos algoritmos implementados. A classe `PoolWithUndelete` é uma implementação diferenciada de um repositório de dados que permite desfazer rapidamente a exclusão de seus elementos. A classe `IntersectionVetInt` representa uma interseção de soluções do tipo `VetInt`. É possível obter rapidamente o tamanho da palavra gerada, o número de soluções envolvidas, a enumeração das soluções envolvidas e a palavra obtida da interseção das soluções. Também suporta a rápida reversão da última inserção de uma solução no conjunto que forma a interseção. A classe `EIntersectionVetInt` complementa a herança da classe `IntersectionVetInt` adicionando suporte para a verificação de inconsistências e detecção da formação de uma solução completa.

3.5 Algoritmos

São fornecidas duas implementações baseadas no operador INT para extração de palavras a partir de um repositório de soluções representadas como vetores de inteiros. A primeira implementação obtém palavras a partir do máximo possível de interseções de soluções cujo tamanho da palavra gerada

seja maior do que um dado valor mínimo. A outra abordagem busca palavras mais extensas, e por isso com uma menor quantidade de soluções utilizadas respeitando um valor mínimo de soluções utilizadas. A primeira combina mais soluções para obter palavras mais consistentes e a segunda combina menos soluções para obter palavras mais extensas.

O algoritmo de extração de palavras da classe `DecomposeVetInt1` está representado na Figura 3.5 pelo procedimento `IntOpt1`. Procura-se formar palavras através da interseção de uma maior quantidade de soluções, respeitando um tamanho mínimo para as palavras. A execução inicia-se com o repositório de soluções S e o tamanho mínimo aceitável para uma palavra w_{min}. O repositório de palavras W é definido na linha 1. O conjunto S', cujo valor inicial é atribuído na linha 2, possui todas as soluções de S que ainda não foram utilizadas na construção de palavras. O laço nas linhas 3–18 procura por palavras enquanto houver soluções em S', ou seja, ainda houver soluções que não participaram da formação de palavras. Como critério de parada do procedimento, cada solução participa da formação de apenas uma palavra. O processo de encontrar uma palavra começa na linha 4 com uma seleção aleatória da solução s de S' para compor a nova palavra. A solução s é o primeiro elemento de $\hat{S}$, que representa o conjunto de soluções que compõem uma palavra e é definido na linha 5. O conjunto S'', criado na linha 6, possui todas as soluções que ainda não participaram na formação de palavras. O laço nas linhas 7–12 procura adicionar novas soluções ao conjunto $\hat{S}$. A condição de parada para a formação de uma palavra ocorre quando não houver mais soluções disponíveis em S''. Escolhe-se aleatoriamente outra solução s para participar na formação da nova palavra (linha 8). Ao adicionar uma solução ao conjunto $\hat{S}$, deve-se verificar se o tamanho da palavra obtida pela interseção das soluções presentes nesse conjunto é maior que o valor mínimo w_{min} (linha 9). A função Int retorna uma palavra através da interseção das soluções de um conjunto. A função $Size$ retorna o tamanho de uma palavra. As definições dessas funções estão apresentadas na Seção 2.1. Se o tamanho da palavra for válido, então a solução s é adicionada ao conjunto $\hat{S}$ (linha 10). A solução s, que foi avaliada na formação da nova palavra, é removida do conjunto S'' na linha 11. Após analisar todas as soluções que não participavam na formação de palavras, é avaliado se o conjunto $\hat{S}$ possui mais de uma solução (linha 13). São consideradas como palavras válidas os resultados da interseção de pelo menos duas soluções. A palavra encontrada é adicionada ao conjunto W na linha 15. O conjunto S' é atualizado com a remoção das soluções utilizadas na formação da palavra (linha 17). O algoritmo retorna o conjunto de palavras encontradas na linha 19.

```
procedure IntOpt1(S, w_min);
1  W ← ∅
2  S' ← S
3  while S' ≠ ∅ do
4      s ← seleciona aleatoriamente um elemento de S'
5      Ŝ ← {s}
6      S'' ← S'\{s}
7      while S'' ≠ ∅ do
8          s ← seleciona aleatoriamente um elemento de S''
9          if Size(Int(Ŝ ∪ {s})) ≥ w_min then
10             Ŝ ← Ŝ ∪ {s}
11         S'' ← S''\{s}
12     end-while
13     if |Ŝ| > 1 then
14         w ← Int(Ŝ)
15         W ← W ∪ {w}
16     end-if
17     S' ← S'\Ŝ
18 end-while
19 return W
end IntOpt1;
```

Figura 3.5: Pseudo-código da heurística para extração de palavras IntOpt1.

O algoritmo de extração de palavras da classe `DecomposeVetInt2` está representado na Figura 3.6 pelo procedimento `IntOpt2`. Procura-se formar palavras maiores através da interseção de uma quantidade mínima de soluções. A execução inicia-se com o repositório de soluções S e o número mínimo de soluções s_{min} que devem compor uma interseção para formar uma palavra. O repositório de palavras W é definido na linha 1. O conjunto S', cujo valor inicial é atribuído na linha 2, possui todas as soluções de S que ainda não foram utilizadas na formação de palavras. O laço nas linhas 3–12 busca por palavras, enquanto o número de soluções em S' for maior que o número mínimo de soluções s_{min} que devem compor uma interseção. O conjunto $\hat{S}$, que representa o conjunto de soluções que compõem uma palavra, é criado na linha 4. O laço nas linhas 5–9 adiciona s_{min} soluções ao conjunto $\hat{S}$. Uma solução de S' é escolhida aleatoriamente (linha 6) e é adicionada a $\hat{S}$ na linha 7. Essa solução é removida de S' (linha 8), pois ela está participando na formação da nova palavra. Visto que a única restrição para uma palavra válida é que ela seja composta por pelo menos s_{min} soluções, todas as palavras geradas pelo laço de 5 a 9 são válidas. Por isso, a palavra w obtida da interseção das soluções de

$\hat{S}$ (linha 10) é adicionada ao conjunto de palavras W (linha 11). O algoritmo retorna o conjunto de palavras encontradas W na linha 13.

```
procedure IntOpt2(S, s_min);
1  W ← ∅
2  S' ← S
3  while |S'| ≥ s_min do
4      Ŝ ← ∅
5      for i = 1 ... s_min do
6          s ← seleciona aleatoriamente um elemento de S'
7          Ŝ ← Ŝ ∪ {s}
8          S' ← S'\{s}
9      end-for
10     w ← Int(Ŝ)
11     W ← W ∪ {w}
12 end-while
13 return W
end IntOpt2;
```

Figura 3.6: Pseudo-código da heurística para extração de palavras IntOpt2.

A combinação de palavras, a partir de um repositório de palavras representadas como vetores de inteiros, é baseada no operador $EINT$. O algoritmo de extração de palavras da classe `GrowVetInt` está representado na Figura 3.7 pelo procedimento `EIntOpt`. Procura-se combinar palavras até obter uma solução completa ou não ser mais possível combinar palavras. A execução inicia-se com o repositório de palavras W. O repositório de frases P é criado na linha 1. O conjunto W', inicializado na linha 2, possui todas as palavras que ainda não foram utilizadas na formação de frases. O laço nas linhas 3–18 busca por frases enquanto houver palavras disponíveis em W'. Como critério de parada, cada palavra participa da formação de apenas uma frase. Para compor uma frase é escolhida aleatoriamente uma palavra w de W' (linha 4). O conjunto de palavras que formam uma frase é representado por $\check{S}$ e tem seu valor inicial definido com a palavra w (linha 5). O conjunto W'', definido na linha 6, possui todas as palavras disponíveis para formar frases, ou seja, as palavras que ainda não participaram na formação de frases. O laço nas linhas 7–12 tenta montar uma frase a partir das palavras disponíveis. Para começar a composição de uma frase, uma palavra w é escolhida aleatoriamente no conjunto W'' (linha 8). É verificado se adicionando-se a palavra w a $\check{S}$ ainda obtém-se uma frase consistente (linha 9). Uma frase consistente, conforme definição da Seção 3.5, não pode ter valores diferentes para uma mesma posição. Se for consistente, a palavra é adicionada a $\check{S}$ na linha 10. O conjunto

W'' é atualizado na linha 11. Frases incompletas podem ser obtidas tanto pela exaustão das palavras disponíveis como pela inviabilidade na formação de frases consistentes. Por isso, verifica-se se $\check{S}$ forma uma frase incompleta (linha 13), ou seja, se representa uma solução incompleta, e se for o caso pode-se completar a frase através do método *complete* (linha 14). A frase p é gerada através da aplicação do operador $EInt$ às palavras presentes no conjunto $\check{S}$ (linha 15). A frase p é adicionada ao conjunto P (linha 16) e as palavras utilizadas em sua geração são removidas do conjunto W' (linha 17). O algoritmo retorna o conjunto de frases encontradas P na linha 19.

```
procedure EIntOpt(W);
1   P ← ∅
2   W' ← W
3   while W' ≠ ∅ do
4       w ← seleciona aleatoriamente um elemento de W'
5       Š ← {w}
6       W'' ← W\{w}
7       while W'' ≠ ∅ and not isComplete(Š) do
8           w ← seleciona aleatoriamente um elemento de W''
9           if consistent(Š ∪ {w}) then
10              Š ← Š ∪ {w}
11          W'' ← W''\{w}
12      end-while
13      if not isComplete(Š) then
14          complete(Š)
15      p ← EInt(Š)
16      P ← P ∪ {p}
17      W' ← W'\Š
18  end-while
19  return P
end EIntOpt;
```

Figura 3.7: Pseudo-código da heurística para combinação de palavras EIntOpt.

4
Estudo de caso: Problema de seqüenciamento de carros

O problema de seqüenciamento de carros em linhas de produção das indústrias automobilísticas é um tipo particular de problema de escalonamento que consiste em determinar a ordem dos carros a serem produzidos na linha de montagem de cada dia de produção que melhor satisfaz os requisitos das oficinas de pintura e de montagem. Estes requisitos procuram reduzir os custos da linha de produção e atender às demandas dos clientes nos prazos estabelecidos. Uma revisão bibliográfica sobre o problema encontra-se em [29]. Neste trabalho considera-se o problema tema do concurso ROADEF'2005 [24] proposto pela Renault.

4.1
Descrição do problema

O problema de seqüenciamento de carros (PSC) é um subproblema de planejamento e escalonamento da produção de veículos. O processo de planejamento da produção se inicia com os pedidos dos clientes, que são enviados em tempo real para as fábricas. Diariamente as fábricas determinam um dia para produção de cada veículo solicitado, respeitando as capacidades da linha de produção e a data de entrega que foi prometida aos consumidores. A linha de produção pode ser considerada como um processo de manufatura linear composta por três oficinas como ilustrado na Figura 4.1. Em seguida deve-se escalonar a ordem para construção dos carros na linha de montagem, satisfazendo o máximo de requisitos das oficinas de carroceria, pintura e montagem. A seqüência de carros na linha de montagem é a mesma para todas as oficinas.

Figura 4.1: Oficinas da linha de produção de uma fábrica de carros

Para o escalonamento diário, a oficina de carroceria não possui requisitos e o conjunto de veículos pré-determinado na fase anterior não pode ser alterado. Os requisitos das oficinas de pintura e de montagem são apresentados nas próximas seções.

Nas fábricas da Renault, uma aplicação industrial gerencia o processo de planejamento e escalonamento, usando programação linear para planejamento e *simulated annealing* para o escalonamento.

4.1.1 Requisitos de pintura

A oficina de pintura busca minimizar o consumo de solvente de tinta. O solvente é utilizado para limpar pistolas de tinta toda vez que a cor de pintura é alterada entre dois veículos consecutivos na seqüência de produção. Portanto, para minimizar o número de trocas da cor de pintura procura-se agrupar em lotes veículos com mesma cor de pintura.

Seqüências de veículos de mesma cor de pintura têm uma limitação no tamanho máximo, já que pistolas de tinta precisam ser lavadas regularmente mesmo se não houver troca na cor da tinta. Esta limitação é uma restrição forte e deve ser respeitada para se obter uma solução viável do problema.

4.1.2 Requisitos de montagem

Com o intuito de suavizar a carga de trabalho na linha de montagem, veículos que requerem operações especiais de montagem têm que ser igualmente distribuídos através do total de carros processados. Esses veículos são considerados como sendo "difíceis de montar", exigindo, por exemplo, teto solar ou ar condicionado, e não podem exceder uma dada quota sobre qualquer seqüência de veículos. Estes requisitos são modelados através de uma restrição de capacidade definida por uma razão N/P.

Uma restrição de capacidade N/P significa que no máximo N carros em cada seqüência consecutiva de P devem estar associados a essa restrição. Por exemplo, se $N/P = 3/5$, então não deve haver mais do que três carros restritos em qualquer seqüência consecutiva de cinco veículos. De um modo mais geral, um carro que possui uma restrição $1/P$ significa que na solução deve haver pelo menos uma seqüência de $P - 1$ carros adjacentes que não possuam essa restrição para que não haja violações dessa restrição.

Há dois tipos de restrições de capacidade: restrições de alta prioridade e restrições de baixa prioridade. As restrições de alta prioridade são relativas às características de carros que demandam um grande adicional na carga de

trabalho na linha de montagem. As restrições de baixa prioridade resultam de características dos carros que causam pequena inconveniência para a produção. Restrições de alta prioridade devem ser satisfeitas preferencialmente, em relação às restrições de baixa prioridade.

Restrições de capacidade são consideradas restrições leves,ou seja, atender a todas as restrições de capacidade não pode ser garantido de antemão quando um dia de produção é planejado. Por isso, um dos objetivos da otimização é minimizar o número de violações das restrições de capacidade.

4.1.3
Regra do dia de produção $D - 1$

Quando o dia de produção D é escalonado, os últimos veículos escalonados do dia de produção $D - 1$ devem ser considerados. Os veículos do dia de produção $D - 1$ já estão escalonados e suas posições não podem ser alteradas. A computação do número de violações das restrições de razão no dia de produção D deve levar em conta os últimos veículos do dia de produção $D - 1$. O dia de produção $D + 1$ é ignorado enquanto o dia de produção D é escalonado.

4.1.4
Definição do Problema

Deve-se escalonar uma seqüência de veículos que satisfaça melhor aos requisitos da oficina de pintura e da linha de montagem. Visto que é um problema multi-objetivo, há diversas possibilidades para a ordem de prioridade entre os requisitos. Na prática existe um algoritmo para tratar cada problema. Neste trabalho considera-se a seguinte ordem de objetivos:

1. minimizar violações das restrições de capacidade de alta prioridade;
2. minimizar violações das restrições de capacidade de baixa prioridade e
3. minimizar número de trocas de cores.

A qualidade das soluções é avaliada através de uma função de custo que atribui pesos aos objetivos de acordo com os níveis de prioridade. A idéia é que a diferença entre esses pesos seja suficientemente grande para reduzir a possibilidade de compensação de um objetivo em detrimento de outro. Dessa forma, pretende-se que a melhor solução seja aquela com o menor valor do primeiro objetivo, em seguida, com o menor valor do segundo e, por fim, com o menor valor do terceiro objetivo.

4.1.5 Formalização

Segundo Rocha [29], o problema de seqüenciamento de carros pode ser definido por uma tupla (C, O, p, q, r), onde $C = \{c_1, \ldots, c_n\}$ é o conjunto de carros cuja produção deve ser escalonada e $O = \{o_1, \ldots, o_m\}$ é o conjunto das diferentes opções que os carros podem requerer. As restrições de capacidades associadas com cada opção $o_j \in O$ são definidas pelas funções $p : O \rightarrow \mathbb{N}$ e $q : O \rightarrow \mathbb{N}$, que determinam que, para cada seqüência de carros consecutivos de tamanho $q(o_j)$, no máximo $p(o_j)$ destes carros podem requerer a opção o_j. As demandas de opções são definidas pela função $r : C \times O \rightarrow \{0, 1\}$, ou seja, $r(c_j, o_j)$ retorna 1 se a opção o_j deve ser instalada no carro c_i, e retorna 0 caso contrário. A notação a seguir será utilizada para manipular as seqüências de carros:

- uma **seqüência** de k carros é denotada por $\pi =< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$;
- o número de carros que uma seqüência π contém é chamado **tamanho** de π e é denotado por $|\pi|$;
- a **concatenação** de duas seqüências π_1 e π_2 é a seqüência formada pelos carros de π_1 seguidos pelos carros de π_2 e é denotada por $\pi_1|\pi_2$;
- o **número de carros que requerem uma opção** o_j em uma seqüência $\pi =< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$ é dado por

$$r(\pi, o_j) = \sum_{l=1}^{k} r(c_{il}, o_j);$$

- o **custo** de uma seqüência $\pi =< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$ é o número de restrições de capacidade violadas, dado por

$$custo(\pi) = \sum_{j=1}^{m} \sum_{\substack{\pi' : |\pi'| = q(o_j) \\ \pi' \subseteq \pi}} violacao(\pi', o_j),$$

onde

$$violacao(\pi', o_j) = \begin{cases} 0 \text{ , se } r(\pi', o_j) \leq p(o_j); \\ 1 \text{ , caso contrário.} \end{cases}$$

Para efeito de simplificação, o número de carros que requerem uma opção o_j no conjunto formado por todos os carros C também é dado por

$$r(C, o_j) = \sum_{i=1}^{n} r(c_i, o_j).$$

Além da notação acima, existe um conceito importante utilizado pela heurística construtiva, chamado de **taxa de utilização** das opções [31]. Este conceito determina, juntamente com o número de carros a serem produzidos e o número de configurações diferentes de opções, a dificuldade de uma instância do problema. A taxa de utilização de uma opção o_j em um conjunto ou uma seqüência de carros é a razão entre o número de carros que demandam o_j na seqüência e o número máximo de carros que podem demandar o_j nesta seqüência de modo a satisfazer a restrição de capacidade associada. Formalmente, a taxa de utilização de uma opção o_j no conjunto de carros C é definida por

$$taxaUtil(o_j) = \frac{r(C, o_j) \cdot q(o_j)}{n \cdot p(o_j)}.$$

Escrevendo de outra forma, a taxa de utilização é definida por

$$taxaUtil(o_j) = \left(\frac{r(C, o_j)}{n}\right) / \left(\frac{p(o_j)}{q(o_j)}\right).$$

Ou seja, a taxa de utilização de uma opção é dada pela razão entre o número de carros demandando essa opção e o número de carros da seqüência dividida pela razão da restrição correspondente. Se a taxa de utilização é maior que 1, a razão do numerador é maior que a razão do denominador e, conseqüentemente, a demanda é maior que a capacidade. Logo, a restrição será violada. Se, ao contrário, a taxa de utilização é muito pequena e próxima de 0, a demanda é muito baixa em relação à capacidade da linha de produção.

4.2 Estratégia algorítmica

Neste trabalho utiliza-se a estratégia para solução proposta por Ribeiro et al. [26, 27, 29]. O projeto da estratégia baseia-se nas características e peculiaridades do problema. O PSC é um problema que visa a otimização de três objetivos tratados com níveis de prioridade diferentes, através de pesos a eles associados na função objetivo. Denomina-se de subproblema o tratamento da otimização separada de cada objetivo. Como a diferença entre os pesos deve ser relativamente grande para enfatizar seus diferentes níveis de prioridade, a idéia de resolver os subproblemas do PSC separadamente por ordem decrescente de prioridade torna-se uma estratégia viável e adequada.

No entanto, quando resolvem-se os subproblemas separadamente é muito comum existirem muitas soluções de mesmo custo, o que se denomina de um empate. Para o subproblema de minimizar as trocas de cores, que é um problema mais simples, essa característica é facilmente notada, por exemplo,

através da permutação de grupos de carros de mesma cor. Para o subproblema de minimizar violações de restrições de capacidade de alta ou baixa prioridade, os empates podem ser observados no exemplo da Figura 4.2, onde se tem várias seqüências com o mesmo número de violações para uma restrição de capacidade de razão 1/4. Mesmo quando o número de restrições aumenta, os empates ainda acontecem.

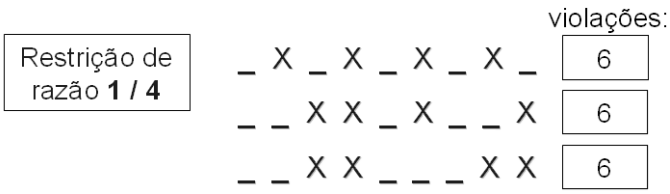

Figura 4.2: Soluções com mesmo número de violações

A estratégia utilizada para resolver o PSC foi, então, desenvolver algoritmos específicos para cada um de seus subproblemas, respeitando o nível de prioridade dos objetivos. Os algoritmos desenvolvidos são técnicas heurísticas e são aplicados de forma seqüencial, conforme citado abaixo:

`Passo 1`: *Heurística construtiva*;
`Passo 2`: *Algoritmo para otimizar o primeiro objetivo*;
`Passo 3`: *Algoritmo para otimizar o segundo objetivo*;
`Passo 4`: *Algoritmo para otimizar o terceiro objetivo.*

4.2.1 Heurística construtiva

Uma solução inicial é construída escolhendo-se iterativamente o próximo carro a ser escalonado na seqüência de acordo com um critério guloso. O critério guloso adotado consiste em escolher o carro que causa o menor número de novas violações de restrições em cada iteração. De uma maneira mais formal, dada uma seqüência parcial π, o próximo carro c_i a ser escolhido é aquele que minimiza

$$novasViolacoes(\pi, c_i) = \sum_{j=1}^{m} \left(r(c_i, o_j) \times violacao(ultCars(\pi| < c_i >, q(o_j)), o_j) \right),$$

onde

$$ultCars(\pi', k) = \begin{cases} \text{seqüência com os últimos } k \text{ carros de } \pi', \text{ se } |\pi'| \geq k \\ \pi', \text{caso contrário.} \end{cases}$$

No entanto, freqüentemente existem vários carros candidatos que minimizam o número de novas violações. Diante disto, um segundo critério guloso é adotado para resolver os empates. Este critério é um pouco mais elaborado, pois favorece uma distribuição eqüitativa das opções. A distribuição é baseada na seguinte idéia: se o número médio de carros que demandam uma opção é menor na seqüência em construção π do que no conjunto total de carros C, então estes carros são favorecidos para entrar na seqüência e vice-versa. Mais formalmente, escolhe-se o carro c_i que minimiza a função:

$$\eta_{SD}(c_i, \pi) = \sum_{j=1}^{m} \left\{ (r(c_i, o_j) = 0) \text{ XOR } \left(\frac{r(C, o_j)}{n} > \frac{r(\pi, o_j)}{|\pi|} \right) \right\}.$$

O primeiro carro a ser escalonado é escolhido aleatoriamente entre os carros com o número máximo de opções, uma vez que a fórmula acima não pode ser aplicada quando $|\pi| = 0$.

Ainda que o critério acima seja utilizado para quebrar os empates, não é raro que eles persistam. Neste caso, um terceiro critério guloso é utilizado, favorecendo carros que requerem opções com alta taxa de utilização. Isso é feito através da soma das taxas de utilização das opções requeridas por cada carro. A taxa de utilização ora considerada é dinâmica, pois é atualizada sempre que um carro é adicionado à seqüência parcial π e é definida pela seguinte função:

$$taxaUtilDin(o_j, \pi) = \frac{[r(C, o_j) - r(\pi, o_j)] \cdot q(o_j)}{[n - |\pi|] \cdot p(o_j)}$$

O último critério de desempate é chamado de soma de taxas de utilização dinâmicas e é dado por:

$$\eta_{SD}(c_i, \pi) = \sum_{j=1}^{m} r(c_i, o_j) \cdot taxaUtilDin(o_j, \pi).$$

Escolhe-se, então, o carro ainda não escalonado que maximiza η_{SD}. Se os empates persistirem após esta segunda tentativa de desempate, um carro é escolhido aleatoriamente dentre os carros candidatos.

4.2.2
Vizinhanças e busca local

A otimização dos diversos objetivos é realizada por heurísticas que utilizam estruturas de vizinhança em comum. Os movimentos aplicados pelas heurísticas de busca local são baseados nas estruturas de vizinhança de troca e deslocamento.

Na vizinhança de troca, um movimento corresponde a inverter as posições de dois carros e é definido por um par de posições (i, j). Em uma seqüência de carros $\pi =< c_{k_1}, \ldots, c_{k_n} >$, a aplicação do movimento $swap(\pi, i, j)$ resulta na seqüência $\pi' =< c_{k_1}, \ldots, c_{k_{i-1}}, c_{k_j}, c_{k_{i+1}}, \ldots, c_{k_{j-1}}, c_{k_i}, c_{k_{j+1}}, \ldots, c_{k_n} >$.

Na vizinhança de deslocamento, um movimento também é representado por um par de posições (i, j) e corresponde a remover um carro de sua posição atual i e inseri-lo em uma nova posição, j. A aplicação do movimento $shift(\pi, i, j)$ na seqüência de carros $\pi =< c_{k_1}, \ldots, c_{k_n} >$ resulta na seqüência $\pi' =< c_{k_1}, \ldots, c_{k_{i-1}}, c_{k_{i+1}}, \ldots, c_{k_j}, c_{k_i}, c_{k_{j+1}}, \ldots, c_{k_n} >$, se $i < j$.

É utilizado um procedimento de busca local diferenciado que não é do tipo primeiro aprimorante, nem maior aprimorante [27]. A vizinhança é dividida em diversas sub-vizinhanças, onde cada uma contém todos vizinhos gerados pela troca ou deslocamento de um carro. Portanto, há n sub-vizinhanças, uma para cada carro. As sub-vizinhanças são utilizadas progressivamente, do primeiro ao último carro, a cada iteração da busca. A partir de uma solução, a busca avança para a melhor solução que não é pior do que a solução corrente na sub-vizinhança correspondente, mesmo se tiver custo igual.

```
procedure LocalSearch(π);
1  repeat
2      φ ← cost(π)
3      for i = 1, ..., |π| do
4          Δ* ← 0
5          L ← ∅
6          for j = 1, ..., |π| do
7              Δ ← cost(move(π, i, j)) − cost(π)
8              if Δ < Δ* then
9                  L ← {j}
10                 Δ* ← Δ
11             else if Δ = Δ* then
12                 L ← L ∪ {j}
13             end-if
14         end-for
15         if L ≠ ∅ then
16             Escolha k ∈ L aleatoriamente
17             π ← move(π, i, j)
18         end-if
19     end-for
20 until φ = cost(π)
21 return π
end LocalSearch;
```

Figura 4.3: Pseudo-código da heurística de busca local.

A Figura 4.3 mostra o pseudo-código do procedimento de busca local. O custo da solução π é calculado por $cost(\pi)$ e $move(\pi, i, j)$ significa o movimento de inverter os carros nas posições i e j ou o movimento de deslocar o carro da posição i para a posição j, dependendo da vizinhança que estiver sendo usada. O procedimento de busca é definido pelo laço nas linhas 1–20, cuja execução é realizada até que nenhuma solução aprimorante possa ser encontrada. O custo da solução corrente ϕ é inicializado na linha 2. O laço nas linhas 3–19 trata os movimento relativos a cada carro $i = 1, \ldots, n$. A variável Δ^* contém o valor da redução no custo da solução corrente referente ao melhor movimento aceitável e é inicializada com zero na linha 4, visto que somente movimentos que não piorem a solução corrente podem ser aceitos. O conjunto L dos melhores movimentos envolvendo o carro i é inicializado na linha. Os movimentos associados ao carro i são investigados no laço 6–14. A redução Δ no custo da solução corrente, relativa ao movimento $move(\pi, i, j)$, é computada na linha 7. Se Δ é menor do que a menor redução de custo do melhor movimento identificado até então, o último é atualizado nas linhas 9 e 10. Se tem custo igual ao mehor movimento identificado até então, o novo movimento é adicionado na lista de melhores movimento na linha 12. No caso de serem encontrados vários movimentos de mesmo custo envolvendo o carro i que não piorem a solução corrente, o movimento a ser aplicado na solução corrente é selecionado aleatoriamente a partir do conjunto de melhores movimentos na linha 16. O movimento escolhido é aplicado à solução corrente na linha 17. A solução obtida pela busca local é retornada na linha 19.

4.2.3
Otimização do primeiro objetivo

O primeiro objetivo está relacionado com as restrições de capacidade de alta prioridade. O objetivo é minimizar o número de violações dessas restrições e, de preferência, eliminá-las.

A heurística construtiva obtém uma solução inicial considerando somente as restrições de capacidade. Por isso, pode-se iniciar esta fase a partir de uma solução inviável através da violação no tamanho limite para as seqüências de carros de mesma cor. O algoritmo para otimização do primeiro objetivo também permite explorar regiões com soluções inviáveis, deixando a restauração da viabilidade para um momento mais tarde.

O algoritmo desenvolvido é baseado na metaheurística busca local iterativa (ILS) [22]. A estratégica básica dessa metaheurística consiste em sucessivamente aplicar perturbações e buscas locais na solução corrente.

A função de perturbação utilizada no algoritmo proposto consiste em

remover um dado número de carros da solução corrente e, em seguida, inseri-los na solução de acordo com o critério guloso descrito anteriormente. Os carros removidos sempre são carros envolvidos em violações das restrições de capacidade de alta prioridade.

A busca local aplicada é baseada na vizinhança definida pelo movimento de troca, com um melhoramento adicional. Uma busca local rápida foi desenvolvida considerando uma propriedade básica. Só consegue-se melhorar uma solução para o primeiro objetivo quando pelo menos um dos carros trocados viola alguma restrição de razão de alta prioridade. Empiricamente é verificado que a fração de carros envolvidos nas violações é menor do que 40%, este melhoramento agiliza a exploração de soluções aprimorantes nesta vizinhança. Portanto, são apenas realizadas trocas de pares de carros que envolvam pelo menos um carro que viole alguma restrição relativa ao primeiro objetivo.

Uma fase de intensificação foi adicionada à heurística desenvolvida. Essa fase consiste da sucessiva aplicação de dois procedimentos de busca local sempre que, depois de um dado número de iterações, nenhum melhoramento for obtido na solução corrente. Ela começa por uma busca local na vizinhança de deslocamento de carros, seguido de uma busca local na vizinhança de troca. Essa fase também pode ser vista como um procedimento de diversificação, pois além de otimizar na vizinhança da solução corrente, é permitido aceitar movimentos com o mesmo custo, escolhidos aleatoriamente. Desse modo, é possível visitar muitas soluções distintas com o mesmo custo durante uma mesma aplicação da busca local, estendendo o conjunto de soluções visitadas.

São realizados reinícios na busca sempre que a solução corrente não é melhorada após um dado número de iterações. As soluções iniciais da busca são obtidas por uma perturbação mais forte da solução corrente, através da remoção de um maior número de carros ou pela aplicação da heurística construtiva. A primeira opção é preferida, visto que nenhuma informação será guardada das iterações anteriores quando se reinicia com uma solução obtida pela heurística construtiva.

O número de reinícios é um dos critérios de parada utilizados pelo algoritmo. O segundo critério de parada é baseado no tempo de computação, visto que há um limite total de tempo para a otimização de todos objetivos. O terceiro e último critério de parada é quando não há mais violação das restrições de capacidade de alta prioridade na solução corrente.

Na Figura 4.4 está o pseudocódigo do algoritmo `ILS_HPRC` para otimização do primeiro objetivo. Na linha 1, a melhor solução π^* e a solução corrente π iniciais são definidas como a solução obtida pela heurística construtiva. O laço nas linhas 2–25 é realizado até que um dos critérios de parada

```
procedure ILS_HPRC(π0);
1   π ← π0, π* ← π0
2   while critérios de parada não satisfeitos do
3       π' ← Perturbation(π)
4       π' ← LocalSearch(π)
5       if custo(π) ≥ custo(π') then
6           π ← π'
7       end-if
8       if custo(π) ≥ custo(π*) then
9           π* ← π
10      end-if
11      if o número de iterações desde o último
            melhoramento em π for igual a α then
12          π' ← Intensification(π)
13          if custo(π) ≥ custo(π') then
14              π ← π'
15          end-if
16          if custo(π) ≥ custo(π*) then
17              π* ← π
18          end-if
19      end-if
20      if o número de iterações desde o último
            melhoramento em π for igual a β then
21          if cost(π) = cost(π*) then
22              π ← Restart(π)
23          else π ← π*
24      end-if
25  end-while
26  return π*
end ILS_HPRC;
```

Figura 4.4: Pseudocódigo do algoritmo de minimização do primeiro objetivo.

seja satisfeito. Cada iteração do algoritmo ILS inicia na linha 3 com uma perturbação aplicada à solução corrente π, levando a uma solução intermediária π'. Uma busca local na vizinhança de troca de carros é aplicada a π' na linha 4. Se a nova solução π' for melhor ou tão boa quanto a π, ela substitui a solução corrente na linha 6. A melhor solução conhecida π^* é atualizada na linha 9 se π' for melhor ou tão boa quanto ela. Se nenhum melhoramento foi obtido na solução atual π depois de α iterações, a fase de intensificação é realizada na linha 12 e a solução corrente tem a possibilidade de ser atualizada na linha 14. A melhor solução conhecida π^* pode ser atualizada na linha 17 se a solução obtida na intensificação for melhor. Se nenhum melhoramento na solução corrente π for obtido depois de β iterações (com $\alpha < \beta$), um reinício é

realizado se o custo dessa solução for igual ao custo da melhor solução conhecida π^*. Caso contrário, faz-se a solução corrente π ser igual a melhor solução conhecida (linha 23).

4.2.4
Otimização do segundo objetivo

O algoritmo proposto para a otimização do segundo objetivo inicia com uma solução obtida pelo algoritmo para a otimização do primeiro objetivo. O primeiro (número de violações das restrições de razão de alta prioridade) e o segundo (número de violações das restrições de razão de baixa prioridade) objetivos são levados em consideração nesta fase. O segundo objetivo é otimizado sem piora do primeiro, que também pode ser melhorado durante este processo.

O algoritmo desenvolvido é baseado na metaheurística busca em vizinhança variável (VNS) [17]. A metaheurística VNS propõe uma troca sistemática de vizinhanças $N_1, \ldots, N_{k_{max}}$ onde são aplicadas perturbações dentro dessas vizinhanças. Basicamente, cada iteração consiste em uma fase de perturbação, seguida de um procedimento de busca local. A busca inicia pela exploração de vizinhanças menores. A região de busca é aumentada sempre que uma solução de melhora não é encontrada dentro da vizinhança atual.

A fase de perturbação faz uso de duas estruturas de vizinhança de um modo oscilatório, trocando o tipo de perturbação sempre que a vizinhança de mais alta ordem da perturbação corrente é atingida. Na primeira estrutura de vizinhança são realizados movimentos de troca aplicados em pares de carros do mesmo tipo escolhidos aleatoriamente, preservando-se assim o valor do custo do primeiro objetivo. Aqui, carros do mesmo tipo são aqueles associados ao mesmo conjunto de restrições de razão de alta prioridade. A segunda estrutura de vizinhança está relacionada com movimentos de deslocamento, que consiste em remover alguns carros envolvidos nas violações da solução corrente e inseri-los em novas posições.

Denota-se por $N_k^1, k = k_1, \ldots, k_{max}$, a seqüência de vizinhanças de deslocamento, onde k é o número de carros que são removidos e subseqüentemente inseridos em novas posições. Similarmente, denota-se por $N_q^2, q = q_1, \ldots, q_{max}$, a seqüência de vizinhança de troca, onde q é o número de trocas de carros.

A estratégia para uma busca local rápida apresentada anteriormente é utilizada. Soluções que sofrem perturbações por movimentos de deslocamento podem ter seu primeiro objetivo deteriorado. Por isso, o procedimento de busca local tenta otimizar a soma ponderada do primeiro com o segundo objetivos. No caso de soluções que sofrem perturbações por movimentos de troca, a busca local considera somente pares de carros do mesmo tipo, conseqüentemente

preservando o custo associado ao primeiro objetivo.

O procedimento de intensificação é similar ao descrito para a otimização do primeiro objetivo. Ele é aplicado sempre que o tipo de perturbação é alternado. O primeiro e o segundo objetivos podem ser melhorados, uma vez que ambos são considerados nesse procedimento.

A execução é finalizada quando qualquer um dos três critérios de parada for satisfeito. O primeiro é um número máximo de fases de intensificação desde o último melhoramento da solução corrente. O segundo critério é definido por um limite de tempo gasto na otimização do segundo objetivo, para garantir que o terceiro objetivo também seja otimizado. Por último, o terceiro critério de parada é satisfeito se não houver violações de restrições de razão de baixa prioridade.

O pseudocódigo apresentado na Figura 4.5 provê uma visão geral do algoritmo `VNS_LPRC` para a otimização do segundo objetivo. Algumas variáveis e parâmetros são descritos a seguir. A variável π^* representa a solução corrente, que é também a melhor solução conhecida, e π' a solução obtida por alguma perturbação. Representa-se por ϕ^* o custo da solução corrente e por ϕ o custo da solução obtida na última fase de intensificação. O contador i acumula o número de fases de intensificação realizadas desde o último melhoramento na solução corrente. O limite superior para este contador é o parâmetro i_{max}. Contadores k e q representam a ordem da vizinhança corrente usada na geração das perturbações de deslocamento e troca, respectivamente. Os parâmetros k_1 e q_1 definem as mais baixas ordens de vizinhança para as perturbações de deslocamento e troca, respectivamente. Analogamente, k_{max} e q_{max} definem as mais altas ordens de vizinhança para as perturbações de deslocamento e troca, respectivamente. Finalmente, P é uma variável binária cujo valor é 1 se a perturbação corrente consiste de movimentos deslocamento e 0 caso contrário.

O algoritmo possui como solução inicial a solução π_0 obtida pela heurística `ILS_HPRC`, usada para a otimização do primeiro objetivo. Configurações iniciais são realizadas nas linhas 1–2. O laço nas linhas 3–27 é realizado até um dos critérios de parada ser satisfeito. As ordens de vizinhança e o custo da solução corrente tem seus valores iniciais definidos nas linhas 4 e 5, respectivamente. Uma iteração do laço nas linhas 6–18 é composto pela aplicação da perturbação na solução corrente π^* resultando na solução π' na linha 7, seguida por uma busca local aplicada nessa última solução obtida na linha 8. A nova solução π' é aceita na linha 9 para substituir a solução corrente π^* se ela for pelo menos tão boa quanto a melhor solução conhecida. O condicional na linha 12–17 atualiza a ordem de vizinhança que será usada na próxima iteração. Se qualquer melhoramento for realizado na solução corrente, a busca

```
procedure VNS_LPRC(π0);
1  π* ← π0, φ* ← cost(π0)
2  i ← 0, P ← 1
3  while critérios de parada não satisfeitos do
4      k ← k1, q ← q1
5      φ ← φ*
6      while (P = 1 and k ≤ kmax) or (P = 0 and q ≤ qmax) do
7          π' ← Perturbation(π*, P, N_k^1, N_q^2)
8          π' ← LocalSearch(π')
9          if cost(π') ≤ cost(π*) then
10             π* ← π'
11         end-if
12         if cost(π*) < φ* then
13             k ← k1, q ← q1
14             φ* ← cost(π*)
15         else
16             k ← k + 1, q ← q + 1
17         end-if
18     end-while
19     P ← 1 − P
20     π* ← Intensification(π*)
21     φ* ← cost(π*)
22     if φ* < φ then
23         i ← 0
24     else
25         i ← i + 1
26     end-if
27 end-while
28 return π*
end VNS_LPRC;
```

Figura 4.5: Pseudocódigo do algoritmo de minimização do segundo objetivo.

reiniciará a partir da mais baixa ordem de vizinhança (linha 13) e o custo ϕ^* da melhor solução conhecida é atualizado (linha 14). Caso contrário, a busca reiniciará a partir de uma ordem de vizinhança maior (linha 16). Quando a perturbação corrente alcançar sua mais alta ordem (k_{max} ou q_{max}, dependendo do seu tipo), o laço nas linhas 6–18 termina e o tipo da perturbação é trocado na linha 19. O procedimento de intensificação é aplicado à solução corrente na linha 20 e o custo da melhor solução conhecida é atualizado na linha 21. A variável i, usada para o critério de parada, é atualizado na linha 23 ou na linha 25, dependendo do resultado do procedimento de intensificação. O algoritmo pára sempre que algum dos critérios de parada for satisfeito.

4.2.5
Otimização do terceiro objetivo

O algoritmo para a otimização do terceiro objetivo também é baseado na metaheurística VNS [17]. Todos os objetivos são considerados simultaneamente, de modo que o primeiro objetivo seja preservado e o segundo e terceiro objetivos são melhorados. A solução inicial obtida pelo algoritmo `VNS_LPRC`, usado para otimizar o segundo objetivo, pode ser inviável. Por exemplo, pode haver lotes de carros de mesma cor cujo tamanho exceda o tamanho máximo permitido. Por isso, a primeira fase do algoritmo consiste em tornar viável a solução inicial, sempre que necessário.

No processo de tornar viável a solução inicial alguns cuidados devem ser tomados para não piorar os objetivos previamente otimizados. Nesse sentido, dois procedimentos separados foram desenvolvidos. O primeiro utiliza movimentos de troca para eliminar as violações. Os movimentos são aplicados somente em pares de carros que compartilhem as mesmas restrições de capacidade, preservando o custo do primeiro e segundo objetivos. Se o primeiro procedimento não for capaz de conseguir a viabilidade, o segundo é aplicado. Somente o último garante a viabilidade da solução, embora com um possível deterioramento dos objetivos previamente otimizados. O segundo procedimento, através de movimentos de deslocamento, remove carros de lotes que violem o tamanho máximo para carros de mesma cor, inserindo-os em novas posições viáveis. As novas posições são aquelas que levam às menores variações do custo da solução.

No algoritmo de otimização do terceiro objetivo, as perturbações são produzidas por movimentos de deslocamento e troca, do mesmo modo descrito para o algoritmo de otimização do segundo objetivo. Como anteriormente, o tipo de perturbação é alternado sempre que a perturbação de mais alta ordem for alcançada. Soluções produzidas por movimento de deslocamento que deteriorem o primeiro objetivo são descartadas e a busca local é aplicada na solução corrente. Isso não se aplica no caso de soluções produzidas por movimentos de troca, pois o custo associado ao primeiro objetivo é preservado através da garantia de que as trocas envolvam apenas carros do mesmo tipo (em relação às restrições de razão de alta prioridade).

Os conjuntos de vizinhança $N_s^1, s = s_1, \ldots, s_{max}$ e $N_t^2, t = t_1, \ldots, t_{max}$ correspondem, respectivamente, aos movimentos de deslocamento e troca. Os valores de s_{max} e t_{max} devem ser menores do que aqueles usados no algoritmo para otimização do segundo objetivo, senão muitas soluções perturbadas serão descartadas.

Soluções obtidas por perturbações têm o valor do primeiro objetivo

preservado, mas podem ter o valor do segundo objetivo deteriorado. Assim sendo, a busca local otimiza a soma ponderada do segundo e terceiro objetivos. A busca local completa é adotada, mas somente são considerados movimentos de trocas de carros envolvendo carros de mesmo tipo (com relação às restrições de razão de alta prioridade). A solução obtida pela busca local é descartada se ela deteriorar o segundo objetivo.

A fase de intensificação é similar àquela previamente apresentada. A única diferença é que todos os objetivos são simultaneamente considerados. Uma fase de intensificação é realizada sempre que o tipo de perturbação é alterado.

A Figura 4.6 mostra o pseudocódigo do algoritmo `VNS_PCC` proposto para a otimização do terceiro objetivo. Sua estrutura é similar à do algoritmo `VNS_LPRC` para a minimização do segundo objetivo, descrito anteriormente. A única diferença consiste na adição do procedimento de reparo para tornar uma solução viável (linha 1) e dos filtros realizados nas soluções obtidas pela perturbação de inserção de carros (linhas 8 até 10). O algoritmo encerra quando esgotar o tempo para execução (linha 4).

```
procedure VNS_PCC(π_0);
1  if π_0 é inviável then π_0 ← MakeFeasible(π_0)
2  π* ← π_0, φ* ← cost(π*)
3  P ← 1
4  while tempo limite não excedido do
5      s ← s_1, t ← t_1
6      while (P = 1 and s ≤ s_max) or (P = 0 and t ≤ t_max) do
7          π' ← Perturbation(π*, P, N_s^1, N_t^2)
8          if first_objective_cost(π') > first_objective_cost(π*) then
9              π' ← π*
10         end-if
11         π' ← LocalSearch(π')
12         if cost(π') ≤ cost(π*) then
13             π* ← π'
14         end-if
15         if cost(π*) < φ* then
16             s ← s_1, t ← t_1
17             φ* ← cost(π*)
18         else
19             s ← s + 1, t ← t + 1
20         end-if
21     end-while
22     P ← 1 − P
23     π* ← Intensification(π*)
24     φ* ← cost(π*)
25 end-while
26 return π*
end VNS_PCC;
```

Figura 4.6: Pseudocódigo do algoritmo de minimização do terceiro objetivo.

5
Experimentos computacionais

A partir do *framework* desenvolvido, são propostas três aplicações para exemplificar a utilização de construção de vocabulário. Neste estudo de caso, são geradas três heurísticas para a resolução do problema de seqüenciamento de carros (PSC). O PSC, considerado nesta dissertação, foi proposto pela indústria automobilística Renault para o desafio Challenge ROADEF'2005, promovido pela Sociedade Francesa de Pesquisa Operacional. No concurso foram disponibilizadas 19 instâncias para o tipo de problema estudado, agrupadas em dois conjuntos de teste. Essas instâncias são descritas a seguir e podem ser obtidas no endereço eletrônico da competição [24].

A Tabela 5.1 apresenta as principais informações a respeito das instâncias utilizadas. Para cada instância, apresenta-se de qual conjunto de teste T ela faz parte, o seu nome ou identificação id, o número de carros a serem escalonados n, o número de restrições de capacidade de alta prioridade m_h e de baixa prioridade m_l, as taxas de utilização média t_h^{med} e máxima t_h^{max} das restrições de capacidade de alta prioridade e as taxas de utilização média t_l^{med} e máxima t_l^{max} das restrições de capacidade de baixa prioridade. Além disso, são apresentados o número de cores diferentes na instância, denotado por c, e o tamanho máximo permitido de grupos de carros de mesma cor g.

As heurísticas implementadas são obtidas com variações na utilização dos componentes do *framework*, explicadas nas seções a seguir. O repositório de soluções, necessário para a construção de vocabulário, é gerado através das soluções encontradas durante a busca pelo método proposto por Ribeiro et al. [26, 27, 29], chamada de heurística HPSC. Essa heurística foi desenvolvida para o Challenge ROADEF'2005 e projetada, conforme estabelecido no desafio, para que seu tempo de execução máximo seja de 10 minutos.

A representação das soluções do PSC utilizada é um vetor com a seqüência numerada dos carros que serão produzidos. Por exemplo, o vetor $[1, 3, 5, 4, 6, 2]$ representa uma solução de seis carros onde o primeiro veículo a entrar na linha de produção é o de identificador 1, seguido pelo 3, continuando assim até o último carro, o de identificador 2. As palavras são vetores incompletos de uma solução, ou seja, com lacunas na seqüência de carros.

T	id	n	m_h	m_l	t_h^{max}	t_h^{med}	t_l^{max}	t_l^{med}	c	g
A	024_38_3	1274	5	8	0.96	0.85	0.95	0.54	13	10
	024_38_5	1329	5	8	0.96	0.85	0.96	0.39	13	10
	025_38_1	1232	4	18	0.76	0.65	1.12	0.70	24	10
	048_39_1	618	5	12	1.00	0.89	0.96	0.67	12	10
B	022_S22_J1	540	2	7	0.85	0.45	1.00	0.23	13	500
	023_S23_J3	1130	9	8	1.04	0.57	0.55	0.26	13	25
	024_V2_S22_J1	1319	6	7	1.36	1.12	1.21	0.94	13	10
	025_S22_J3	1257	4	12	0.81	0.67	4.17	0.93	19	10
	028_ch1_S22_J2	385	9	11	1.14	0.60	0.95	0.68	20	15
	028_ch2_S23_J3	92	1	8	0.72	0.72	2.00	0.58	3	20
	029_S21_J6	773	4	3	1.05	0.58	1.09	0.42	12	15
	035_ch1_S22_J3	133	2	2	1.28	1.25	1.33	1.21	7	70
	035_ch2_S22_J3	239	3	2	2.10	1.45	1.39	1.72	7	150
	039_ch1_S22_J4	1263	2	9	0.58	0.29	0.85	0.77	15	25
	039_ch3_S22_J4	1119	2	9	0.68	0.57	0.88	0.68	17	20
	048_ch1_S22_J3	902	6	19	0.90	0.56	0.75	0.39	14	10
	048_ch2_S22_J3	595	7	16	0.89	0.54	0.58	0.30	12	10
	064_ch1_S22_J3	854	11	3	0.90	0.42	0.96	0.90	14	15
	064_ch2_S22_J4	451	4	1	0.85	0.34	1.16	1.16	10	15

Tabela 5.1: Dados das instâncias dos conjuntos de teste A e B

Nas palavras, os elementos '$*$' representam ausência de carro naquela posição. Por exemplo, o vetor $[1, *, 5, 4, *, *]$ representa uma palavra de tamanho 3.

Na formação de frases pelo método `EIntOpt`, descrito na Seção 3.5, pode-se obter frases incompletas, ou seja, um vetor com elementos '$*$'. Nas frases incompletas aplica-se a heurística construtiva, apresentada na Seção 4.2.1, para inserir os carros ausentes nas soluções parciais.

Todos os algoritmos foram implementados na linguagem de programação C++, utilizando-se a versão 3.2.2 do compilador gcc. Os experimentos foram executados em um computador equipado com processador Pentium IV 1.7GH, memória RAM de 249 MB e sistema operacional Red Hat Linux versão 9 kernel 2.4.20-31.9. Os algoritmos foram executados compartilhando-se o processamento da máquina com aplicações do sistema ou de outros usuários.

Com estes experimentos, pretende-se exemplificar a utilização do *framework* e avaliar se acrescentar heurísticas de construção de vocabulário com os operadores clássicos à estratégia definida por HPSC produz um método melhor para a resolução do problema de seqüenciamento de carros. As definições dos pontos flexíveis do *framework* são apresentadas para cada heurística desenvolvida. No Apêndice A é apresentado o código utilizado na implementação das heurísticas e as modificações realizadas em HPSC.

5.1 Heurística HCV1

A heurística HCV1 é uma construção de vocabulário que busca palavras mais consistentes. Essa heurística é aplicada como uma técnica de pós-otimização e possui como entrada de dados um repositório de boas soluções. A execução consiste em extrair fragmentos de boas soluções do repositório e combiná-los em fragmentos maiores até obter soluções completas ou não ser mais possível combiná-los. A execução termina quando todos os fragmentos identificados forem avaliados em combinações, conforme a estratégia escolhida. O método para encontrar palavras é o algoritmo `IntOpt1`, descrito na Seção 3.5, que está disponível na classe `DecomposeVetInt1` do *framework*. Para combinar as palavras é utilizado o algoritmo `EIntOpt`, descrito na mesma seção do algoritmo anterior, implementado na classe `GrowVetInt`.

Nos testes realizados foram consideradas duas execuções da heurística HPSC como o método gerador do repositório de soluções, uma com 5 minutos e a outra com 10 minutos. As soluções candidatas ao repositório são as melhores encontradas durante a busca. Ou seja, sempre que for encontrada uma solução melhor ou, pelo menos, de mesmo custo que a melhor solução conhecida, ela é enviada ao repositório. Uma solução é adicionada no repositório se satisfizer os critérios de aceitação. No caso desta heurística, é avaliada a similaridade da nova solução em relação às demais presentes no repositório. A similaridade tolerada para as soluções no repositório de soluções (*pool*) é de pelo menos 10 posições diferentes, valor que foi arbitrado por experimentação para evitar que o repositório de soluções ficasse cheio de soluções similares. A fim de analisar os parâmetros de entrada do método, foram testados três tamanhos mínimos para as palavras (w_{min}).

Foram realizados testes com as instâncias de teste A e B. A Tabela 5.2 mostra os resultados dos testes realizados com o conjunto A e a Tabela 5.3 mostra os resultados do conjunto B. Na coluna $T_1(s)$ está o tempo de execução em segundos dado para a heurística HPSC, em $|pool|$ está o número de soluções que estão no repositório utilizado na construção de vocabulário e em $T_{w_{min}}$ estão as taxas em porcentagens do tamanho de uma solução utilizadas para determinar os tamanhos mínimos para as palavras (w_{min}). Tal que, $w_{min} = n \times T_{w_{min}}$, onde n é o número de carros a serem escalonados apresentado na Tabela 5.1. Nas colunas $|p|$ e $|f|$ estão as quantidades de palavras e frases, respectivamente, encontradas pelo método de construção de vocabulário e em $T_2(s)$ o tempo em segundos gasto na execução do método. As colunas HPRC, LPRC, PCC possuem o número de violações das restrições de alta prioridade e de baixa prioridade e o número de troca de cores, respectivamente, das

melhores soluções obtidas pelos métodos. Na linha onde há valor para $T_1(s)$ os valores de HPRC, LPRC, PCC são relativos à melhor solução obtida pela heurística HPSC, enquanto as demais são relativas às melhores soluções obtidas pela construção de vocabulário.

Os resultados mostram o impacto do parâmetro $T_{w_{min}}$ sobre a construção de vocabulário. Quanto menos restritivo, aceitando palavras menores, o método realiza interseções com mais soluções para a formação das palavras. Quanto mais restritivo, menos soluções são utilizadas para se obter palavras de maior comprimento. Nos testes da instância 025_38_1 com $T_{w_{min}} = 90\%$, a restrição é tão forte que até inviabiliza a formação de palavras. Um bom valor para $T_{w_{min}}$ é aquele que busca a formação de palavras com equilíbrio entre mais soluções na composição, tornando-as mais consistentes, e palavras mais extensas, mais próximas de uma solução completa. O valor $T_{w_{min}} = 50\%$ obteve os melhores resultados na média considerando todas as instâncias. Por isso, só são exibidos os resultados dos testes com as instâncias do tipo B para as execuções com $T_{w_{min}} = 50\%$.

Pelo fato de não aceitar inconsistências (carros diferentes para uma mesma posição), em nenhum teste foi possível formar uma frase com mais de uma palavra. A geração de palavras mais extensas se beneficia disso, e possibilitou até o caso com a instância 024_38_3, $T_1(s) = 300$ e $T_{w_{min}} = 90\%$, em que a construção de vocabulário produziu uma solução melhor que a melhor encontrada até então na busca. Quanto mais próxima uma palavra está de uma solução completa há menos carros para serem inseridos. Por isso, as soluções geradas pelo método com $T_{w_{min}} = 20\%$ são piores que as demais, já que é necessário inserir muitos carros considerando apenas os critérios gulosos. Os testes com $T_{w_{min}} = 50\%$ na instância 024_38_5 mostram que determinar palavras mais consistentes pode ser uma boa estratégia, visto que possibilitou obter resultados melhores que os obtidos com palavras mais extensas.

A variação do tempo de execução da heurística HPSC mostra que o método HCV1 é sensível à qualidade e ao tamanho do repositório de soluções. Embora, com a instância 024_38_3 foi observado o contrário, na qual um repositório menor produziu soluções melhores. Nas demais instâncias, foi observado o que era esperado.

O tempo de execução de HCV1 está relacionado com o tamanho e a quantidade de palavras. Quanto menor o tamanho das palavras e maior o número de palavras encontradas, verifica-se que é maior o tempo gasto pelo método. Nas instâncias 024_38_3 e 024_38_5 com $T_{w_{min}} = 50\%$ o tempo de execução de HCV1 é aproximadamente 1/3 do tempo da heurística HPSC e menor nas demais instâncias. Essa relação sinaliza ser possível utilizar o

Instância	$T_1(s)$	$\|pool\|$	$T_{w_{min}}$	$\|p\|$	$\|f\|$	$T_2(s)$	HPRC	LPRC	PCC
024_38_3	300						4	1248	1115
		141	20%	28	28	124	6	1237	1094
			50%	36	36	107	5	1161	1081
			90%	30	30	41	4	1184	1091
	600						4	15	1080
		179	20%	39	39	188	7	1079	1103
			50%	45	45	140	6	1175	1094
			90%	36	36	55	6	114	1095
024_38_5	300						4	650	1159
		144	20%	30	30	169	6	644	1161
			50%	32	32	105	5	683	1151
			90%	33	33	41	6	631	1162
	600						4	80	538
		239	20%	49	49	267	6	619	1164
			50%	59	59	197	4	688	1160
			90%	55	55	81	5	634	1160
025_38_1	300						0	240	867
		20	20%	2	2	14	0	673	766
			50%	7	7	35	0	464	803
			90%	0	0	< 0	-	-	-
	600						0	215	795
		33	20%	5	5	35	0	647	761
			50%	11	11	53	0	409	843
			90%	0	0	1	-	-	-
048_39_1	300						0	79	522
		212	20%	33	33	45	3	778	506
			50%	40	40	44	6	771	505
			90%	50	50	27	2	825	519
	600						0	77	322
		220	20%	28	28	44	4	776	509
			50%	42	42	47	4	893	516
			90%	51	51	29	2	819	521

Tabela 5.2: Instâncias do teste A resolvidas pela heurística HCV1

Instância	$T_1(s)$	$\|pool\|$	$T_{w_{min}}$	$\|p\|$	$\|f\|$	$T_2(s)$	HPRC	LPRC	PCC
022_S22_J1	600						0	3	144
		35	50%	7	7	4	0	16	211
023_S23_J3	600						48	0	431
		563	50%	51	51	108	48	92	947
024_V2_S22_J1	600						1082	1327	1104
		153	50%	6	6	49	1195	2038	1035
025_S22_J3	600						0	3912	867
		24	50%	5	5	25	0	4011	868
028_ch1_S22_J2	600						54	7	110
		617	50%	1	1	90	54	165	154
028_ch2_S23_J3	600						0	70	6
		6	50%	1	1	1	0	72	10
029_S21_J6	600						35	2150	183
		620	50%	109	109	111	35	2150	281
035_ch1_S22_J3	600						67	52	49
		603	50%	161	161	25	67	61	81
035_ch2_S22_J3	600						386	342	204
		600	50%	0	0	65	-	-	-
039_ch1_S22_J4	600						0	29	262
		19	50%	4	4	9	0	42	312
039_ch3_S22_J4	600						0	0	231
		14	50%	3	3	6	0	0	275
048_ch1_S22_J3	600						0	0	216
		32	50%	7	7	17	0	23	384
048_ch2_S22_J3	600						3	0	344
		50	50%	3	3	3	5	10	456
064_ch1_S22_J3	600						0	0	215
		48	50%	10	10	15	0	14	231
064_ch2_S22_J4	600						0	69	130
		15	50%	3	3	1	0	69	169

Tabela 5.3: Instâncias do teste B resolvidas pela heurística HCV1

método durante uma busca para diversificar as soluções. A heurística HCV3, descrita na Seção 5.3, explora essa possibilidade de composição das heurísticas.

5.2 Heurística HCV2

A heurística HCV2 é uma construção de vocabulário que busca palavras mais extensas. Para isso, ela considera um número reduzido de soluções na formação das palavras. O método para encontrar palavras é o algoritmo `IntOpt2`, descrito na Seção 3.5, que está disponível na classe `DecomposeVetInt2` do *framework*. Para combinar as palavras é utilizado o algoritmo `EIntOpt`, descrito na mesma seção do algoritmo anterior, implementado na classe `GrowVetInt`. A heurística HCV2 é aplicada como uma técnica

de pós-otimização.

Nos testes realizados são consideradas duas execuções da heurística HPSC como o método gerador do repositório de soluções, uma com 5 minutos e outra com 10 minutos. As soluções candidatas ao repositório são as melhores encontradas durante a busca. Ou seja, sempre que for encontrada uma solução melhor ou, pelo menos, de mesmo custo que a melhor solução conhecida ela é enviada ao repositório. Uma solução é adicionada no repositório se satisfizer aos critérios de aceitação. No caso desta heurística, é avaliada a similaridade da nova solução em relação às demais presentes no repositório. A similaridade tolerada para as soluções no repositório de soluções (*pool*) é de pelo menos 10 posições diferentes. A fim de analisar os parâmetros de entrada do método, foram testados três quantidades mínimas de palavras que compõe uma palavra (s_{min}).

Foram realizados testes com as instâncias de teste A e B. A Tabela 5.4 mostra os resultados dos testes realizados com o conjunto A e a Tabela 5.5 mostra os resultados do conjunto B. Na coluna $T_1(s)$ está o tempo de execução em segundos dado para a heurística HPSC, em $|pool|$ está o número de soluções que estão no repositório utilizado na construção de vocabulário e em s_{min} estão os números de soluções que participam na formação das palavras. Nas colunas $|p|$ e $|f|$ estão as quantidades de palavras e frases, respectivamente, encontradas pelo método de construção de vocabulário e em $T_2(s)$ o tempo em segundos gasto na execução do método. As colunas HPRC, LPRC, PCC possuem o número de violações das restrições de alta prioridade e de baixa prioridade e o número de troca de cores, respectivamente, das melhores soluções obtidas pelos métodos. Na linha onde há valor para $T_1(s)$ os valores de HPRC, LPRC, PCC são relativos à melhor solução obtida pela heurística HPSC, as demais são relativas às melhores soluções obtidas pela construção de vocabulário.

Pelo fato da heurística HCV2 buscar por palavras maiores pela combinação de um número reduzido de soluções, o maior número de palavras encontradas eleva o tempo de execução do método. O tempo de execução de HCV2 chega a ser superior ao da heurística HPSC em casos de testes com as instâncias 024_38_3 e 024_38_5. O parâmetro s_{min} varia de 2 a 4, implicando diretamente no número de palavras geradas ($|p|$) e por conseqüência no tempo de execução ($T_2(s)$) do método .

Em alguns casos, como os testes com a instância 024_29_3 e $T_1(s) = 600$, foi possível combinar duas palavras. Isso fica evidenciado pelo diferente número de palavras e frases encontradas. No teste com a instância 024_29_5, $T_1(s) = 300$ e $s_{min} = 3$, três palavras foram combinadas para formar uma frase, ou foram geradas 2 frases com a combinação de 2 palavras cada. A diversidade

Instância	$T_1(s)$	$\|pool\|$	s_{min}	$\|p\|$	$\|f\|$	$T_2(s)$	HPRC	LPRC	PCC
024_38_3	300						4	1248	1115
		141	2	70	70	398	9	1195	1112
			3	47	46	281	65	426	999
			4	36	35	216	75	470	961
	600						4	15	1080
		180	2	90	89	531	46	785	1050
			3	60	59	361	73	440	990
			4	45	44	266	75	455	983
024_38_5	300						4	650	1159
		144	2	72	72	463	70	385	1102
			3	48	46	302	90	265	975
			4	36	35	231	98	237	966
	600						4	78	470
		243	2	122	122	792	9	657	1188
			3	82	81	534	91	278	1009
			4	61	60	393	94	273	960
025_38_1	300						0	240	867
		20	2	10	10	66	0	485	779
			3	7	7	51	0	675	779
			4	6	5	35	0	524	777
	600						0	216	841
		32	2	17	17	114	0	397	828
			3	11	11	79	0	568	769
			4	9	8	59	0	657	778
048_39_1	300						0	79	522
		212	2	106	106	180	7	781	495
			3	71	70	122	20	516	476
			4	54	53	95	25	495	497
	600						0	77	320
		221	2	111	111	188	9	884	519
			3	74	74	128	20	451	496
			4	56	55	96	21	444	507

Tabela 5.4: Instâncias do teste A resolvidas pela heurística HCV2

das soluções inviabiliza a composição de mais palavras em frases, por gerarem inconsistências.

O tempo de execução é muito variável, e no caso de instâncias grandes gasta-se muito mais tempo do que o disponível para a busca. O exemplo com a instância 023_S23_J3 ilustra bem o tempo necessário para executar completamente HCV2 em instâncias grandes, precisando de $1952s$ para concluir a execução. Na instância 028_ch2_S23_J3 foi necessário apenas $1s$ para concluir a execução da heurística.

Os testes com $s_{min} = 2$ geraram palavras mais extensas e, como já observado nos testes de HCV1, essas palavras produziram as melhores soluções. Entretanto, a qualidade das soluções é pior do que as geradas pela heurística HCV1. Ao analisar os experimentos chegamos à conclusão que combinar soluções aleatoriamente para gerar palavras extensas não é uma boa estratégia para este problema.

5.3 Heurística HCV3

A heurística HCV3 é uma construção de vocabulário com o mesmo propósito da HCV1, mas utilizada durante a busca e não como pós-otimização. O método para encontrar palavras é o algoritmo `IntOpt1`, descrito na Seção 3.5, que está disponível na classe `DecomposeVetInt1` do *framework*. Para combinar as palavras é utilizado o algoritmo `EIntOpt`, descrito na mesma seção do algoritmo anterior, implementado na classe `GrowVetInt`.

Nos testes realizados, a heurística HPSC foi executada como método gerador do repositório de soluções durante 5 minutos. Em seguida, executou-se a heurística de construção de vocabulário para gerar uma nova solução, em que foi considerada a melhor solução encontrada. A solução encontrada anteriormente foi melhorada pela heurística HPSC por mais 5 minutos, descontando-se o tempo gasto na execução da construção de vocabulário. No total, o método tem 10 minutos para execução.

As soluções candidatas ao repositório são as melhores encontradas durante a busca. Ou seja, sempre que for encontrada uma solução melhor ou, pelo menos, de mesmo custo que a melhor solução conhecida, ela é enviada ao repositório. Uma solução é adicionada no repositório se satisfizer aos critérios de aceitação. No caso desta heurística, é avaliada a similaridade da nova solução em relação às demais presentes no repositório. A similaridade tolerada para as soluções no repositório de soluções (*pool*) é de pelo menos 10 posições diferentes.

Foi utilizado $T_{w_{min}} = 75\%$, que representa a fração do tamanho de

Instância	$T_1(s)$	$\|pool\|$	s_{min}	$\|p\|$	$\|f\|$	$T_2(s)$	HPRC	LPRC	PCC
022_S22_J1	600						0	3	144
		34	2	17	17	11	0	6	309
023_S23_J3	600						48	0	431
		246	2	123	123	1952	48	90	957
024_V2_S22_J1	600						1082	1327	1104
		152	2	76	76	544	1205	2110	1046
025_S22_J3	600						0	3912	867
		221	2	111	111	188	9	884	519
028_ch1_S22_J2	600						54	7	110
		614	2	307	307	233	54	309	152
028_ch2_S23_J3	600						0	70	6
		5	2	2	2	1	0	72	9
029_S21_J6	600						35	2150	183
		621	2	310	309	358	35	2150	379
035_ch1_S22_J3	600						67	52	49
		604	2	302	302	8	67	58	74
035_ch2_S22_J3	600						386	342	204
		601	2	300	299	40	388	341	213
039_ch1_S22_J4	600						0	29	262
		18	2	9	9	39	0	44	771
039_ch3_S22_J4	600						0	0	231
		13	2	6	6	23	0	0	341
048_ch1_S22_J3	600						0	0	216
		31	2	15	15	62	0	13	463
048_ch2_S22_J3	600						3	0	344
		50	2	25	25	50	3	53	423
064_ch1_S22_J3	600						0	0	215
		47	2	23	23	63	0	14	270
064_ch2_S22_J4	600						0	69	130
		14	2	7	7	2	0	69	162

Tabela 5.5: Instâncias do teste B resolvidas pela heurística HCV2

uma solução utilizada para determinar os tamanhos mínimos para as palavras (w_{min}), de modo que $w_{min} = n \times T_{w_{min}}$, onde n é o número de carros a serem escalonados apresentado na Tabela 5.1.

Foram realizados testes com as instâncias de teste A e B e os resultados estão, respectivamente, nas tabelas Tabela 5.6 e Tabela 5.7. O método foi executado cinco vezes e o resultado relativo à melhor solução encontrada para cada instância é apresentado. Na coluna $|pool|$ é informado o número de soluções que estão no repositório utilizado na construção de vocabulário, em $|p|$ e $|f|$ são informados as quantidades de palavras e frases, respectivamente, encontradas pelo método de construção de vocabulário e $T_{VB}(s)$ é o tempo em segundos gasto na execução do método. As colunas HPRC, LPRC e PCC informam o número de violações das restrições de alta prioridade e de baixa

prioridade e o número de troca de cores, respectivamente, das melhores soluções obtidas pelos métodos.

Instância	$\|pool\|$	$\|p\|$	$\|f\|$	$T_{VB}(s)$	HPRC	LPRC	PCC
024_38_3	147	38	38	80	4	16	519
024_38_5	105	21	21	39	4	63	478
025_38_1	15	0	0	0	0	199	848
048_39_1	213	44	44	35	0	72	416

Tabela 5.6: Instâncias do teste A resolvidas pela heurística HCV3

Instância	$\|pool\|$	$\|p\|$	$\|f\|$	$T_{VB}(s)$	HPRC	LPRC	PCC
022_S22_J1	31	6	6	2	0	3	117
023_S23_J3	426	48	48	78	48	0	349
024_V2_S22_J1	73	0	0	4	1081	896	540
025_S22_J3	21	5	5	15	0	3912	869
028_ch1_S22_J2	603	92	92	105	54	9	156
028_ch2_S23_J3	3	0	0	39	0	70	6
029_S21_J6	596	146	146	124	35	2150	321
035_ch1_S22_J3	602	1	1	29	67	52	50
035_ch2_S22_J3	594	0	0	187	385	341	205
039_ch1_S22_J4	9	2	2	14	0	29	362
039_ch3_S22_J4	13	2	2	1	0	0	247
048_ch1_S22_J3	36	6	6	10	0	0	219
048_ch2_S22_J3	48	3	3	3	3	0	346
064_ch1_S22_J3	38	10	10	12	0	0	220
064_ch2_S22_J4	10	2	2	1	0	69	130

Tabela 5.7: Instâncias do teste B resolvidas pela heurística HCV3

A heurística HCV3 obteve resultados superiores às heurísticas anteriores. Entretanto, não foi possível melhorar a resolução do problema de seqüenciamento de carros pela heurística HPSC. A heurística HPSC foi utilizada como um componente estanque, e não era possível otimizar seu funcionamento no contexto da aplicação proposta. Cabe ressaltar que essa heurística foi configurada para o desafio Challenge ROADEF'2005, para ser executada em 10 minutos contínuos, considerando esse tempo para avançar a execução dos algoritmos na otimização de cada objetivo.

6
Conclusões e trabalhos futuros

Nesta dissertação foi proposto um *framework* para facilitar a implementação de heurísticas baseadas em construção de vocabulário. O desenvolvimento foi fundamentado em extensa revisão bibliográfica sobre a técnica e em boas práticas de engenharia de software, como *frameworks* orientados a objetos e padrões de projeto. O projeto do *framework* foi pautado pelas premissas de baixo acoplamento com outros métodos, facilidade na variação das implementações e flexibilidade na representação de dados.

São fornecidas implementações dos operadores clássicos de extração e combinação de palavras definidos por Glover, explicados na Seção 2.1 e formalizados na Seção 3.5. Esses operadores já demonstraram sua eficácia na resolução em outros problemas [1, 8].

O operador clássico para extração de palavras INT foi disponibilizado nas duas implementações. Uma obtém palavras a partir do máximo possível de interseções de soluções cujo tamanho da palavra gerada seja maior do que um dado valor mínimo. A outra abordagem busca palavras mais extensas, e por isso com uma menor quantidade de soluções utilizadas respeitando um valor mínimo de soluções utilizadas. A primeira combina mais soluções para obter palavras mais consistentes e a segunda combina menos soluções para obter palavras mais extensas.

As implementações dos operadores clássicos utilizam vetores de inteiros como tipo genérico para representação de soluções. Para facilitar o acoplamento da heurística gerada pelo *framework* a métodos já existentes, foi utilizado o padrão de projeto *adapter*. Sendo assim, possibilita-se utilizar os operadores clássicos em problemas com representação específica. Para isso, basta que a solução na sua representação original seja traduzida para sua representação como vetor de inteiros.

Durante o desenvolvimento do *framework*, foi verificado que é necessário avaliar as soluções candidatas ao repositório de boas soluções para evitar o armazenamento de soluções muito parecidas, piorando o desempenho do método. Optamos pela diversidade das soluções no repositório e, por isso, foi implementada a avaliação pela distância de Hamming para comparar duas

soluções. Assim, permite-se garantir que as soluções no repositório sejam diferentes em pelo menos um número mínimo de posições do vetor de inteiros.

Como um estudo de caso, foram geradas três aplicações a partir do *framework* para a resolução do problema de seqüenciamento da produção de carros. Em cada aplicação foi obtida uma heurística diferente pela utilização de diferentes configurações dos pontos flexíveis: representação de solução, métodos para extrair e combinar palavras e gerenciamento do repositório de soluções.

O desenvolvimento dessas aplicações foi realizado de modo incremental, avaliando se acrescentar construção de vocabulário com os operadores clássicos à estratégia proposta por Ribeiro et al. [26, 27, 29], apresentada na Seção 4.2, melhoraria na resolução do problema de seqüenciamento de carros. Para representar uma solução do problema foi adotada a representação por vetor de inteiros (`VetInt`). Como a solução do problema nessa estratégia é representada por uma estrutura complexa, foi necessário converter essas soluções em objetos `VetInt`, tarefa que foi bem implementada no *framework*.

Os operadores clássicos, por serem operadores genéricos, podem ser utilizados na resolução de outros problemas. Entretanto, os resultados computacionais demonstraram que com a utilização desses operadores não foi possível melhorar a resolução do problema de seqüenciamento de carros.

Para esse problema, o operador clássico de combinação de palavras não conseguiu formar frases. Para contornar essa limitação, foi considerada a possibilidade de utilizar-se outras formas de representação da solução. Algumas formas alternativas são a representação por vetor de adjacência, como definido em [8], ou a representação por vetor com tipo de carros, onde considera-se carros de mesma cor e restrições como iguais. Entretanto, a formação de ciclos na recuperação da solução inviabilizou na prática a utilização dessas abordagens para esse problema.

Pelo fato da composição dos operadores no *framework* ter sido definida utilizando o padrão de projeto *strategy*, é possível variá-los em tempo de execução, permitindo assim, a comparação entre implementações numa mesma execução. Essa característica foi explorada nos experimentos computacionais quando diversos parâmetros de métodos foram avaliados para uma mesma instância.

Dentre os trabalhos futuros ficam a implementação de outros operadores de extração e combinação de palavras como os que são os definidos em [25]. Esses operadores são baseados em técnicas de mineração de dados. Soluções são representadas por conjuntos de itens. Padrões são definidos como subconjuntos de itens que ocorrem num número significativo de soluções. O processo de obter esses padrões é o bem conhecido problema em mineração de dados chamado

Mineração de Conjunto Freqüente de Itens. São utilizadas estratégias distintas pela variação de como se gera e utiliza o conjunto de padrões. Por exemplo, uma é chamada maior padrão híbrido. Ela utiliza apenas o maior padrão encontrado num conjunto de padrões. Esse conjunto é formado pelos padrões presentes em pelo menos um percentual de elementos do conjunto de soluções elite. Outra estratégia, chamada maior suporte híbrido, procura pelo padrão mais freqüente no conjunto suporte dado um tamanho mínimo. É parecido com a forma de utilizar o operador clássico INT, mas feito de uma forma mais sofisticada.

Outro trabalho futuro é utilizar a distância de Levenshtein [20], também conhecida como distância de edição, para avaliar soluções candidatas ao repositório de soluções. A distância de Hamming é definida somente para vetores de mesmo comprimento. Dados dois vetores, a distância de Hamming é o número total de valores que são diferentes em posições respectivas dos vetores. Na distância de Levenshtein, a distância entre dois vetores é dada pelo número mínimo de operações necessárias para transformar um vetor no outro. São consideradas as operações de inserção, remoção ou substituição de um valor. Assim, pode-se determinar quão semelhante são dois vetores, mesmo de tamanhos diferentes. Essa métrica é muito utilizada em corretores ortográficos. Como a distância de Levenshtein é mais sofisticada e necessita de mais processamento, sua utilização pode degradar o desempenho da heurística que gastará mais tempo para avaliar as soluções candidatas ao repositório.

Dentre as vantagens em se utilizar o *framework* proposto neste trabalho, considera-se a reutilização dos componentes desenvolvidos como a melhor. Pois, o desenvolvimento de heurísticas para outros problemas se torna mais rápido e fácil aproveitando-se esses componentes. Esse benefício é ampliado com a utilização do *framework* em outras situações. Pode-se aproveitar os componentes previamente implementados, bem como aproveitar futuramente os componentes desenvolvidos para outras aplicações. A desvantagem em se utilizar um *framework* é a necessidade de aprendê-lo, no sentido de conhecer seus componentes e a interação entre eles. Como o *framework* proposto neste trabalho possui poucas classes, é fácil aprender sua arquitetura e desenvolver novos componentes para utilizá-lo na resolução de outros problemas. Enfim, o *framework* desenvolvido está apto a ser utilizado na geração de heurísticas para resolução de outros problemas.

Referências Bibliográficas

[1] ALOISE, D.. **Heurísticas para o projeto de redes com funções de custo discretas**. Dissertação de Mestrado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2005.

[2] ANDREATTA, A. A.. **Uma Arquitetura Abstrata de Domínio para o Desenvolvimento de Heurísticas de Busca Local com uma Aplicação ao Problema de Construção de árvores Filogenéticas**. Tese de doutorado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1998.

[3] ANDREATTA, A. A.; CARVALHO, S. E. ; RIBEIRO, C. C.. **An object-oriented framework for local search heuristics**. Em: 26TH CONFERENCE ON TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, p. 33–45, 1998.

[4] FONTOURA, M.; LUCENA, C. J.; ANDREATTA, A. A.; CARVALHO, S. E. ; RIBEIRO, C. C.. **Using UML-FW to enhance framework development: A case study in the local search heuristics domain**. Em: JOURNAL OF SYSTEMS AND SOFTWARE 57, p. 201–206, 2001.

[5] ANDREATTA, A. A.; CARVALHO, S. E. ; RIBEIRO, C. C.. **A framework for local search heuristics for combinatorial optimization problems**. Em: Voss, S.; Woodruff, D., editores, OPTIMIZATION SOFTWARE CLASS LIBRARIES, p. 59–79. Kluwer, 2002.

[6] COMBS, T. E.. **A combined adaptive tabu search and set partitioning approach for the crew scheduling problem with an air tanker crew application**. Tese de doutorado, AFIT/DS/ENS/02, Air Force Institute of Technology, Wright-Patterson AFB, OH, 2002.

[7] FEO, T.; RESENDE, M.. **Greedy randomized adaptive search procedures**. Journal of Global Optimization, 6:109–133, 1995.

[8] FERNANDES, E. L. R.. **Heurísticas para o problema de seqüenciamento de DNA por hibridação**. Dissertação de Mestrado, Departa-

mento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2005.

[9] GAMMA, E.; HELM, R. ; JOHNSON, R.. **Design Patterns. Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[10] GLOVER, F.. **Ejection chains, reference structures and alternating path methods for traveling salesman problem**. Discrete Applied Mathematics, 49:231–255, 1992.

[11] GLOVER, F.; LAGUNA, M.. **Tabu Search**. Em: Reeves, C. R., editor, MODERN HEURISTIC TECHNIQUES FOR COMBINATORIAL PROBLEMS, p. 70–141. Blackwell Scientific Publishing, 1993.

[12] GLOVER, F.. **Tabu search and adaptive memory programing – Advances, applications and challenges**. Em: Barr, R. S.; Helgason, R. V. ; Kennington, J. L., editores, INTERFACES IN COMPUTER SCIENCE AND OPERATIONS RESEARCH, p. 1–75. Kluwer, 1996.

[13] GLOVER, F.; KELLY, J. P. ; LAGUNA, M.. **New advances and applications of combining simulation and optimization**. Em: PROCEEDINGS OF THE 28TH CONFERENCE ON WINTER SIMULATION, p. 144–152. ACM Press, 1996.

[14] GLOVER, F.; LAGUNA, M.. **Tabu Search**. Kluwer, 1997.

[15] GLOVER, F.. **Scatter search and path relinking**. Em: Corne, D.; Dorigo, M. ; Glover, F., editores, NEW IDEAS IN OPTIMIZATION, p. 297–316. McGraw-Hill Education, 1999.

[16] GREISTORFER, P.; VOSS, S.. **Controlled pool maintenance for metaheuristics**. Em: Rego, C.; Alidaee, B., editores, METAHEURISTIC OPTIMIZATION VIA MEMORY AND EVOLUTION, p. 387–424. Kluwer, 2005.

[17] HANSEN, P.; MLADENOVIC, N.. **Variable Neighborhood Search**. Em: Glover, F.; Kochenberger, G., editores, HANDBOOK OF METAHEURISTICS, p. 145–184. Kluwer, 2002.

[18] HARDER, R.. **OpenTS — Java tabu search**. Referência on-line disponível em http://www.coin-or.org/OpenTS/, último acesso em 3 de julho de 2006.

[19] KELLY, J. P.; XU, J.. **A set-partitioning-based heuristic for the vehicle routing problem**. INFORMS J. on Computing, 11:161–172, 1999.

[20] LEVENSHTEIN, V. I.. **Binary codes capable of correcting deletions, insertions, and reversals**. Soviet Physics Doklady, 10:707–710, 1966.

[21] LOPEZ, L.; CARTER, M. W. ; GENDREAU, M.. **The hot strip mill production scheduling problem: A tabu search approach**. European Journal of Operational Research, 106:317–335, 1998.

[22] LOURENÇO, H. P.; MARTIN, O. ; STÜETZLE, T.. **Iterated Local Search**. Em: Glover, F.; Kochenberger, G., editores, HANDBOOK OF METAHEURISTICS, p. 321–353. Kluwer, 2002.

[23] MARKIEWICZ, M. E.; LUCENA, C. J.. **Issues on object-oriented framework development**. ACM Crossroads, 7:3–9, 2001.

[24] NGUYEN, A.. **Challenge ROADEF'2005: Car sequencing problem**. Referência on-line disponível em http://www.prism.uvsq.fr/∼vdc/ROADEF/CHALLENGES/2005/, último acesso em 3 de julho de 2006.

[25] RIBEIRO, M. H.; TRINDADE, V.; PLASTINO, A. ; MARTINS, S. L.. **Hybridization of GRASP metaheuristics with data mining techniques**. Em: Blum, C.; Roli, A. ; Sampels, M., editores, PROCEEDINGS OF THE FIRST INTERNATIONAL WORKSHOP ON HYBRID METAHEURISTICS, p. 69–78. Valência, Espanha, 2004.

[26] RIBEIRO, C. C.; ALOISE, D.; NORONHA, T. F.; ROCHA, C. T. ; URRUTIA, S.. **A hybrid heuristic for a real-life car sequencing problem with multiple requirements**. Em: 18TH MINI-EURO CONFERENCE ON VNS, 2005.

[27] RIBEIRO, C. C.; ALOISE, D.; NORONHA, T. F.; ROCHA, C. T. ; URRUTIA, S.. **An efficient implementation of a VNS/ILS heuristic for a real-life car sequencing problem**. European Journal of Operational Research, a ser publicado.

[28] ROCHAT, Y.; TAILLARD, E.. **Probalistic diversification and intensification in local search for vehicle routing**. Journal of Heuristics, 1:147–167, 1995.

[29] ROCHA, C. T. M.. **Heurísticas para o problema de seqüenciamento da produção de carros**. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal Fluminense, Niterói, 2005.

[30] SCHOLL, A.; KLEIN, R. ; DOMSCHKE, W.. **Pattern based vocabulary building for effectively sequencing mixed-model assembly lines**. Journal of Heuristics, 4:359–381, 1998.

[31] SMITH, B. M.. **Succeed-first or fail-first: A case study in variable and value ordering**. Em: PROCEEDINGS OF THE PARALLEL COMPUTING TECHNOLOGIES 4TH INTERNATIONAL CONFERENCE, p. 321–330. Yaroslavl, 1997.

[32] TAILLARD, E.; BADEAY, P.; GENDREAU, M.; GUERTIN, F. ; POTVIN, J.-Y.. **A new neighborhood structure for the vehicle routing problem with time windows**. Relatório Técnico CRT-95-66, Centre de Recherche sur les Transports, Université de Montréal, Montréal, 1995.

[33] TOULOUSE, M.; THULASIRAMAN, K. ; GLOVER, F.. **Multi-level cooperative search: A new paradigm for combinatorial optimization and an application to graph partitioning**. Em: PROCEEDINGS OF THE EUROPEAN CONFERENCE ON PARALLEL PROCESSING, p. 533–542. Toulouse, França, 1999.

[34] VOSS, S.; WOODRUFF, D. L.. **Optimization Software Class Libraries**. Kluwer, 2002.

A
Códigos desenvolvidos na implementação do framework

A seguir são apresentados os códigos das classes que compõem o *framework*, os pontos flexíveis, a implementação das heurísticas utilizadas no estudo de caso e as alterações que foram necessárias em outros programas para sua utilização.

No estudo de caso, as heurísticas utilizadas foram acopladas a um método previamente existente. Foi necessário definir um novo objeto para representar a heurística de construção de vocabulário na classe que representa uma solução para o problema, de acordo com a arquitetura da heurística já existente, bem como modificar o método que salva os dados de uma solução quando for encontrada uma solução melhor do que a já conhecida. Também foi criado um método para inicializar o método de construção com o repositório de boas soluções, chamado *InitPool*.

```
/*******************************************/
/* Classe que armazena os dados da solucao */
/*******************************************/
class Solution{
  ...
  /* atributos */
  VB *vb;
  ...
  /* métodos */
  Solution(Instance*);
  ~Solution();
  void InitPool(GNA *gna);
  ...
  // Grava dados da melhor solução encontrada
  void SaveSolution();
  ...
}
```

```
/*****************************************/
/* Definição de métodos                  */
/*****************************************/
Solution::Solution(Instance *pInst)
{
  vb=0;
  ...
}

Solution::~Solution(){
  ...
  if(vb != 0)
    delete(vb);
}

void Solution::InitPool(GNA *gna){
  vb = new VB();
}

void Solution::SaveSolution()
{
  vb->addSolution(*this);
  ...
}
```

Na classe principal do programa, deve-se considerar na fase de incialização de variáveis uma chamada ao método *InitPool*. Após executar diversos algoritmos para resolver o problema, e nessa execução encontrar soluções que são adicionadas ao repositório de boas soluções pelo método *SaveSolution*, a heurística de construção de vocabulário é executada por uma chamada ao método estático *VBController::startTests*, que será explicado a seguir.

```
int main(int argc, char ** argv)
{
  ...
  Instance inst(nomeInstancia);
  GNA gna(semente);  // Gerador de Números Aleatório
  Solution sol(&inst);
  sol.InitPool(&gna);
  ...
```

```
  // Heurísticas para solucionar o problema em questão
  ...
  VBController::startTests(sol.vb);
  ...
}
```

A classe *VBController* auxilia na configuração e execução das diversas heurísticas de construção de vocabulário geradas a partir do *framework*. Abaixo é apresentado o código dessa classe bem como das classes associadas com a instanciação das heurísticas para o estudo de caso realizado nesta dissertação. Vale lembrar que para utilizar o *framework* é necessário definir os pontos flexíveis, que são a representação de solução, método para encontrar palavras, método para formar frases e gerenciamento do repositório, de acordo com a necessidade do problema.

```
/************************************************************
 *
 * Definições das estruturas utilizadas na instanciação de
 * heurísticas de construção de vocabulário para o PSC
 *
 ************************************************************/
#ifndef VBSTRUCTURES_H
#define VBSTRUCTURES_H

/* Bibliotecas C++
*******************/
#include <vector>

/* Bibliotecas locais
**********************/
#include "pool.h"
#include "vb.h"
#include "solution.h"
class Solution;

class VB : public VocabularyBuilding {
  friend class VBController;

  private:
    Solution *pSolution;
```

```
// Function executed before the decompose strategy
  void preOpt();
// Function executed after the grow strategy
  void posOpt();

protected:
  void deletePhrases() {
    deleteItemsPool(pPhrases);
  }
  void deleteWords() {
    deleteItemsPool(pWords);
  }
  Pool &getWords() {
    return *pWords;
  }
  Pool &getPhrases() {
    return *pPhrases;
  }
  Pool &getPool() {
    return *pPool;
  }
  void setPool(Pool *newPool) {
    delete pPool;
    pPool = newPool;
  }

public:
  VB()
  : VocabularyBuilding() {
    pPool = new Pool();
    pPool->setInputFunctionStrategy(new
      HammingDistanceVetInt(pPool, 10));
    pSolution = NULL;
  }
  ~VB() {
    deleteItemsPool(pPool);
    deleteItemsPool(pWords);
    deleteItemsPool(pPhrases);
  }
```

```
    void addSolution(Solution&);
    void deleteItemsPool(Pool *pool);
};

class VBController {
  public :
    static void configVB1(VB &vb, int wmin);
    static void configVB2(VB &vb, int smin);
    static void cleanVB(VB &vb);
    static void startTests(VB &vb);
};
#endif

/*************************************************************
 *
 * Implementações das estruturas utilizadas na instanciação
 * de heurísticas de construção de vocabulário para o PSC
 *
 *************************************************************/
/* Bibliotecas C++
*******************/
#include <iostream>
#include <vector>
#include <time.h>

using namespace std;

/* Bibliotecas Locais
**********************/
#include "bibrand.h"
#include "vbstructures.h"
#include "pool.h"
#include "perturbation.h"
#include "ils.h"

long long makeSentence(VetInt *pVetInt, Solution *pSolution) {
  vector<int> &array = pVetInt->array;
  vector<int> carsOut;
  int vCars[pSolution->pInstance->iNumCars];
```

```
int i, j;
bool isInPhrase;

/* verifica quais carros nao estao na frase */
for(i = 0;  i < pSolution->pInstance->iNumCars; i++) {
  isInPhrase = false;
  for(j = 0;  j < pSolution->pInstance->iNumCars
    && !isInPhrase; j++)
    if( array[j] == i )
      isInPhrase = true;
  if( !isInPhrase ) {
    carsOut.push_back(i);
  }
}

/* copia a frase jutando os carros e acrescenta carsOut */
for(i = 0, j = 0; i < pSolution->pInstance->iNumCars; i++) {
  if( array[i] != DecomposeVetInt1::STAR ) {
    vCars[j] = array[i];
    j++;
  }
}

i = pSolution->pInstance->iNumCars - carsOut.size();
for(j = 0 ; i < pSolution->pInstance->iNumCars; i++, j++) {
  vCars[i] = carsOut[j];
}

//cout << "Montando uma solution...";
long long fx = pSolution->GetSolution(vCars);

/* Completa solucoes que estiverem incompletas */
H5_Perturbation_HPRC_LPRC_PCC p(pSolution);
p.reinsertConstraintNO(carsOut.size(),
  GNA::getInstance(), NULL);
pSolution->UpdateCosts();
pSolution->UpdateHashGraph();
fx = pSolution->Function();
return fx;
```

```
}

void vbPosOpt(Pool &vPhrases, Solution *pSolution) {
  long long fxBest;
  long fxBestColors;
  long fxBestHigh;
  long fxBestLow;
  int bestSolution[pSolution->pInstance->iNumCars];

  int best = -1;
  cout << "\n posOpt..." << endl;

  for(int i = 0; i < vPhrases.pool.size(); i++) {
    long long fx = makeSentence(
      static_cast<VetInt*>(vPhrases.pool[i]), pSolution);
    if( best == -1 || fx < fxBest ) {
      best = i;
      fxBest = fx;
      fxBestColors = pSolution->iNumWashes;
      fxBestHigh = pSolution->iNumHighViolations;
      fxBestLow = pSolution->iNumLowViolations;
      for(int j = 0; j < pSolution->pInstance->iNumCars; j++)
      bestSolution[j] = pSolution->vSolution[j];
    }
  }

  if(vPhrases.pool.size() == 0)
    return;
  long long fx = pSolution->GetSolution(bestSolution);
  pSolution->SaveSolution();
  cout << "A melhor solucao e " << best << " " << fx << endl;
}

void VB::preOpt() {
}

void VB::posOpt() {
    vbPosOpt(getPhrases(), pSolution);
}
```

```
void VB::addSolution(Solution& solution) {
  VetInt *vetInt = new VetInt(solution.pInstance->iNumCars);
  for(register int i=0;i<solution.pInstance->iNumCars;i++)
  {
    vetInt->array[i]=solution.vSolution[i];
  }
  pPool->addElement(vetInt);
  if( pSolution == NULL ) {
    pSolution = &solution;
  }
}

void VB::deleteItemsPool(Pool *pool) {
  std::vector<PoolElement*> &p = pool->pool;
  for(int i = 0; i < p.size(); i++) {
    delete static_cast<VetInt*>(p[i]);
  }
}

void VBController::configVB1(VB &vb, int wmin) {
  cout << " Setting up VB1...\n";
  DecomposeVetInt1 *decomposeVetInt = new DecomposeVetInt1();
  GrowStrategy *growVetInt = new GrowVetInt();
  decomposeVetInt->wmin = wmin;
  vb.setDecomposeStrategy(static_cast<DecomposeStrategy*>
    (decomposeVetInt));
  vb.setGrowStrategy(static_cast<GrowStrategy*>(growVetInt));
  cout << " Settings done!\n";
}

void VBController::configVB2(VB &vb, int smin) {
  cout << " Setting up VB2...\n";
  DecomposeVetInt2 *decomposeVetInt = new DecomposeVetInt2();
  GrowStrategy *growVetInt = new GrowVetInt();
  decomposeVetInt->smin = ( smin > 0 ? smin : 2);
  vb.setDecomposeStrategy(static_cast<DecomposeStrategy*>
    (decomposeVetInt));
  vb.setGrowStrategy(static_cast<GrowStrategy*>(growVetInt));
  cout << " Settings done!\n";
```

```
}

void VBController::cleanVB(VB &vb) {
  cout << " Cleaning up VB...";
  vb.deleteWords();
  vb.deletePhrases();
  cout << " done!\n";
}


void VBController::startTests(VB &vb) {
  time_t timeIni, timeFim;

  if(!vb.pSolution){
    cout << " Pool of solutions are empty!\n";
    return;
  }

  cout << " Starting tests...\n";
  int iNumToday = vb.pSolution->pInstance->iNumToday;
  int iNumYesterday = vb.pSolution->pInstance->iNumYesterday;

  cout << " HCV1 - Cenario 1= 20%...\n";
  configVB1(vb, static_cast<int>(iNumToday*0.2)+iNumYesterday);
  time(&timeIni);
  vb.execute();
  time(&timeFim);
  cleanVB(vb);
  printf("Tempo gasto: %f s \n", difftime(timeFim,timeIni));

  cout << "\n HCV1 - Cenario 2= 50%...\n";
  configVB1(vb, static_cast<int>(iNumToday*0.5)+iNumYesterday);
  time(&timeIni);
  vb.execute();
  time(&timeFim);
  cleanVB(vb);
  printf("Tempo gasto: %f s \n", difftime(timeFim,timeIni));

  cout << "\n HCV1 - Cenario 3= 90%...\n";
```

```
configVB1(vb, static_cast<int>(iNumToday*0.9)+iNumYesterday);
time(&timeIni);
vb.execute();
time(&timeFim);
cleanVB(vb);
printf("Tempo gasto: %f s \n", difftime(timeFim, timeIni));

cout << " HCV2 - Cenario 1= 2...\n";
configVB2(vb, 2);
time(&timeIni);
vb.execute();
time(&timeFim);
cleanVB(vb);
printf("Tempo gasto: %f s \n", difftime(timeFim, timeIni));

cout << " HCV2 - Cenario 2= 3...\n";
configVB2(vb, 3);
time(&timeIni);
vb.execute();
time(&timeFim);
cleanVB(vb);
printf("Tempo gasto: %f s \n", difftime(timeFim, timeIni));

cout << " HCV2 - Cenario 3= 4...\n";
configVB2(vb, 4);
time(&timeIni);
vb.execute();
time(&timeFim);
cleanVB(vb);
printf("Tempo gasto: %f s \n", difftime(timeFim, timeIni));

cout << "\n HCV3 - Cenario 1= 75%...\n";
configVB1(vb, static_cast<int>(iNumToday*0.75)+iNumYesterday);
time(&timeIni);
vb.execute();
time(&timeFim);
vb.getPool().setInputFunctionStrategy(new
  IF_NoAddStrategy(&(vb.getPool())));
vb.pSolution->tempoVB = static_cast<int>
```

```
    (difftime(timeFim, timeIni));
  cleanVB(vb);
  printf("Tempo gasto: %f s \n", difftime(timeFim, timeIni));
}
```

As próximas classes apresentadas são as definições básicas utilizadas nas heurísticas apresentadas anteriormente. Essas classes correspondem à implementação do framework, e são reutilizáveis na geração de heurísticas para outros problemas.

```
/*************************************************************
 *
 * Framework para geração de heurísticas baseadas em
 * construção de vocabulário.
 *
 * Definições do respositório de soluções, palavras e frases.
 *
 *************************************************************/

#ifndef POOL_H
#define POOL_H

/* Bibliotecas C
******************/
#include <unistd.h>
#include <vector>

class PoolElement
{
  public:
    virtual bool operator != ( const PoolElement & rhs );
};

class Pool;
class InputFunctionStrategy {
  public:
    Pool *pool;
    InputFunctionStrategy(Pool *p) {
      pool = p;
    }
```

```
    ~InputFunctionStrategy() {
    }
    virtual bool canAdd(PoolElement* poolElement) = 0;
};

class Pool {
  private:
    InputFunctionStrategy *inputFunctionStrategy;

  public:
    std::vector<PoolElement*> pool;
    Pool( ) {
      inputFunctionStrategy = NULL;
    }
    ~Pool( ) { }
    void addElement(PoolElement *element) {
      if (inputFunctionStrategy  &&
          inputFunctionStrategy->canAdd(element)) {
        pool.push_back(element);
      } else if ( !inputFunctionStrategy ) {
        pool.push_back(element);
      }
    }
    void setInputFunctionStrategy(InputFunctionStrategy
      *inputFunc) {
      inputFunctionStrategy = inputFunc;
    }
};
#endif

/*************************************************************
 *
 * Framework para geração de heurísticas baseadas em
 * construção de vocabulário.
 *
 * Implementações do respositório de soluções, palavras e
 * frases.
 *
 *************************************************************/
```

```
#include "pool.h"

bool PoolElement::operator != ( const PoolElement & rhs )
{
  return this != &rhs;
}

/*************************************************************
 *
 * Framework para geração de heurísticas baseadas em
 * construção de vocabulário.
 *
 * Definições das classes do framework.
 *
 *************************************************************/
#ifndef VB_H
#define VB_H

/* Bibliotecas Globais
**********************/
#include <iostream>

/* Bibliotecas Locais
**********************/
#include "pool.h"

class DecomposeStrategy {
  public:
    virtual Pool* execute(Pool* solutions) = 0;
};

class GrowStrategy {
  public:
    virtual Pool* execute(Pool* words) = 0;
};

class VocabularyBuilding {
  protected:
    Pool *pPool;
```

```
    Pool *pWords;
    Pool *pPhrases;
    DecomposeStrategy *pDecomposeStrategy;
    GrowStrategy *pGrowStrategy;
    // Function executed before the decompose strategy
    virtual void preOpt() = 0;
    // Function executed after the grow strategy
    virtual void posOpt() = 0;

  public:
    VocabularyBuilding() {
      std::cout << "    -  Creating a VB -     \n";
      pDecomposeStrategy = NULL;
      pGrowStrategy = NULL;
      pPool = NULL;
      pWords = NULL;
      pPhrases = NULL;
    }
    ~VocabularyBuilding() {
      if( pPool != NULL )
        delete pPool;
      if( pWords != NULL )
        delete pWords;
      if( pPhrases != NULL )
        delete pPhrases;
    }
    void setDecomposeStrategy(DecomposeStrategy *dStrategy) {
      if( pDecomposeStrategy != NULL )
        delete pDecomposeStrategy;
      pDecomposeStrategy = dStrategy;
    }
    void setGrowStrategy(GrowStrategy *gStrategy) {
      if( pGrowStrategy != NULL )
        delete pGrowStrategy;
      pGrowStrategy = gStrategy;
    }
    // Executa o procedimento de construcao de vocabulario
    void execute() {
      std::cout << "Executing Vocabulary Building";
```

```
      std::cout << "There are " << (pPool?pPool->pool.size():0)
        << " solutions in the Pool\n";
      preOpt();
      if( pDecomposeStrategy != NULL )
        pWords =  pDecomposeStrategy->execute(pPool);
      std::cout << (pWords?pWords->pool.size():0) <<
        " words were found." << std::endl;
      if( pGrowStrategy != NULL )
        pPhrases = pGrowStrategy->execute(pWords);
      std::cout << (pPhrases?pPhrases->pool.size():0) <<
        " phrases were built " << std::endl;
      posOpt();
    }
    virtual void deleteItemsPool(Pool *pool) = 0;
};

class VetInt : public PoolElement {
  public:
    std::vector<int> array;
    int num;
    VetInt() {
      num = 0;
    }
    VetInt(const std::vector<int>& rhs) {
      array = rhs;
    }
    VetInt(int length) {
      array.resize(length);
    }
    bool operator != (const PoolElement& rhs) {
      return array != ((VetInt&) rhs).array;
    }
};

class HammingDistanceVetInt: public InputFunctionStrategy {
  public:
    // Minimal distance that must a solution have from all
    // others in the pool to be accept
    // Distance = 1 means that allows only different solutions
```

```
    int distance;

    HammingDistanceVetInt(Pool *pool, int dist)
      : InputFunctionStrategy(pool) {
      distance = dist;
    }
    ~HammingDistanceVetInt() {
    }
    bool canAdd(PoolElement* poolElement);
};

class IF_NoAddStrategy: public InputFunctionStrategy {
  public:
    IF_NoAddStrategy(Pool *pool)
      : InputFunctionStrategy(pool) {
    }
    ~IF_NoAddStrategy() {
    }
    bool canAdd(PoolElement* poolElement){
      return false;
    }
};

class PoolWithUndelete {
  private:
    int size;
    int length;
    std::vector<PoolElement*> array;

  public:
    PoolWithUndelete() : size(0), length(0) {
    }
    PoolWithUndelete(const Pool *rhs) {
      array = rhs->pool;
      length = size = array.size();
    }
    ~PoolWithUndelete() {
    }
    const PoolWithUndelete& operator =
```

```
      (const PoolWithUndelete& rhs) {
      array = rhs.array;
      length = rhs.length;
      size = rhs.size;
    }
    // return true if the pool is empty
    bool isEmpty( ) const {
      return 0 == length;
    }
    // Get the element given the position
    PoolElement* getElement(int pos) const {
      return array[pos];
    }
    // Return how many elements are in the pool
    int getLength() const {
      return length;
    }
    // undelete the excluded items
    void reset() {
      length = size;
    }
    void insert(const PoolElement* x);
    void remove(const PoolElement* x);
    // Remove an element but supports undo.
    void exclude(int solution);
    // Remove many elements but supports undo.
    void exclude(std::vector<int> solutions);
};

class IntersectionVetInt {
  protected:
    std::vector<int> solutions;
    std::vector<int> intersection;
    std::vector<int> intersectionBefore;
    int wordSize;
    int wordSizeBefore;
    int normSet;

  public:
```

```
    IntersectionVetInt()
    : wordSize(0), wordSizeBefore(0), normSet(0) { }
    int getWordSize() const {
      return wordSize;
    }
    int getNumberSolutions() const {
      return solutions.size();
    }
    std::vector<int> getSolutions() const {
      return solutions;
    }
    VetInt* getIntersection() const {
      VetInt* vetInt = new VetInt(intersection);
      vetInt->num = normSet;
      return vetInt;
    }
    // Remove the last solution inserted in the intersection
    // WARNING: Must be used only once, at least until
    //          another solution be inserted
    void removeLastInserted() {
      solutions.pop_back();
      intersection = intersectionBefore;
      wordSize = wordSizeBefore;
      normSet--;
    }
    // Remove all solutions wich are in the intersection
    void removeAll() {
      solutions.clear();
      wordSize = 0;
      normSet = 0;
    }
    void include(int pos, const VetInt* pool);
    int getNorm() const {
      return normSet;
    }
};

class ExtIntersectionVetInt : public IntersectionVetInt {
  protected:
```

```
    int completed;
    int inconsistent;
  public:
    ExtIntersectionVetInt()
      : completed(0), inconsistent(0) {
      IntersectionVetInt::IntersectionVetInt() ;
    }
    int hasInconsistence(){
      return inconsistent;
    }
    int isComplete() {
      return completed;
    }
    void complete() {
    }
    void extInclude(int pos, const VetInt* words);
};


class DecomposeVetInt1 : public DecomposeStrategy {
  public:
    static const int STAR;
    int wmin;
    Pool* execute(Pool* solutions);
    DecomposeVetInt1(int wSize=0)
      : wmin(wSize) {
    }
};

class DecomposeVetInt2 : public DecomposeStrategy {
  public:
    int smin;
    Pool* execute(Pool* solutions);
    DecomposeVetInt2(int numSolutions=0) : smin(numSolutions) {
    }
};

class GrowVetInt : public GrowStrategy {
  public:
```

```
    Pool* execute(Pool* words);
    GrowVetInt() {
    }
};
#endif

/*************************************************************
 *
 * Framework para geração de heurísticas baseadas em
 * construção de vocabulário.
 *
 * Implementações das classes do framework.
 *
 *************************************************************/
/* Bibliotecas C++
*******************/
#include <iostream>
#include <vector>

using namespace std;

/* Bibliotecas Locais
**********************/
#include "vb.h"
#include "bibrand.h"

const int DecomposeVetInt1::STAR = -1;

bool HammingDistanceVetInt::canAdd(PoolElement* poolElement) {
  VetInt *solution = static_cast<VetInt*>(poolElement);
  for(int i = 0; i < pool->pool.size(); i++) {
    VetInt *poolSolution = static_cast<VetInt*>(pool->pool[i]);
    int cont = 0;
    for(int j = 0; j < solution->array.size(); j++)
      if( poolSolution->array[j] == solution->array[j] )
        cont++;
      if( (solution->array.size() - cont) < distance ){
        return false;
    }
```

```
  }
  return true;
}

void IntersectionVetInt::include(int pos, const VetInt* pool) {
  if( solutions.size() == 0 ) {
    solutions.push_back(pos);
    intersection = pool->array;
    wordSize = pool->array.size();
    return;
  }
  solutions.push_back(pos);
  intersectionBefore = intersection;
  wordSizeBefore = wordSize;
  for( int i = 0; i < pool->array.size(); i++ )
    if( intersection[i] != DecomposeVetInt1::STAR &&
      pool->array[i] != intersection[i] ) {
      wordSize--;
      intersection[i] = DecomposeVetInt1::STAR;
    }
 normSet++;
}

void ExtIntersectionVetInt::extInclude(int pos,
  const VetInt* word) {
  if( solutions.size() == 0 ) {
    solutions.push_back(pos);
    intersection = word->array;
    wordSize = 0;
    for( int i = 0; i < word->array.size(); i++ )
      if( word->array[1] != DecomposeVetInt1::STAR )
        wordSize++;
    return;
  }
  solutions.push_back(pos);
  intersectionBefore = intersection;
  wordSizeBefore = wordSize;
  for( int i = 0; i < word->array.size(); i++ )
    if( intersection[i] == DecomposeVetInt1::STAR &&
```

```
      word->array[i] != DecomposeVetInt1::STAR ) {
      wordSize++;
      intersection[i] = word->array[i];
    } else if( intersection[i] != DecomposeVetInt1::STAR &&
             word->array[i] != intersection[i] ) {
      inconsistent = true;
    }
  if( wordSize == intersection.size() ) {
    completed = true;
  }
  normSet++;
}

void PoolWithUndelete::insert(const PoolElement* x) {
  array.push_back( const_cast< PoolElement* > (x) );
  if( length < size ) {
    PoolElement* tmp;
    tmp = array[length+1];
    array[length+1] = array[size+1];
    array[size+1] = tmp;
  }
  size++;
  length++;
}

void PoolWithUndelete::remove(const PoolElement* x) {
  int i;
  for( i = 0; i < length; i++ )
    if( !(array[i] != x) )
      break;
  if( i < length )
    exclude(i);
}

void PoolWithUndelete::exclude(int solution) {
  PoolElement* aux;
  if( length < 1 ) {
    length--;
    return;
```

```
  }
  aux = array[length-1];
  array[length-1] = array[solution];
  array[solution] = aux;
  length--;
}

void PoolWithUndelete::exclude(vector<int> solutions) {
  for( int i = 0; i < solutions.size(); i++ ) {
    exclude(solutions[i]);
  }
}

void findWords(const int &wmin, PoolWithUndelete &solutions,
  Pool* words) {
  PoolWithUndelete solutionsAux;
  IntersectionVetInt intersection;

  while( !solutions.isEmpty() ) {
    int length = solutions.getLength();
    int solution = GNA::getInstance()->rand(length);
    VetInt *aSolution;
    aSolution = static_cast<VetInt*>
      (solutions.getElement(solution));

    intersection.removeAll();
    intersection.include(solution, aSolution);

    solutionsAux = solutions;
    solutionsAux.exclude(solution);
    while( !solutionsAux.isEmpty() ) {
        length = solutionsAux.getLength();
        solution = GNA::getInstance()->rand(length);
        aSolution = static_cast<VetInt*>
          (solutionsAux.getElement(solution));
        intersection.include(solution, aSolution);
         if( intersection.getWordSize() < wmin ) {
           intersection.removeLastInserted();
         }
```

```
        solutionsAux.exclude(solution);
    }
    if ( intersection.getNumberSolutions() > 1 ) {
      words->pool.push_back(intersection.getIntersection());
    }
    solutions.exclude(intersection.getSolutions());
  }
}

Pool* DecomposeVetInt1::execute(Pool* pool) {
  cout << " Finding words... \n";
  Pool* words = new Pool();
  PoolWithUndelete solutions(pool);
  findWords(wmin, solutions, words);
  solutions.reset();
  return words;
}

Pool* DecomposeVetInt2::execute(Pool* pool) {
  cout << " Finding words... \n";
  Pool* words = new Pool();
  PoolWithUndelete solutions(pool);
  IntersectionVetInt intersection;

  while( solutions.getLength() >= smin  ) {
    intersection.removeAll();
    for (int i = 0; i < smin; i++) {
      int length = solutions.getLength();
      int solution = GNA::getInstance()->rand(length);
      VetInt *aSolution;
      aSolution = static_cast<VetInt*>
        (solutions.getElement(solution));
      intersection.include(solution, aSolution);
      solutions.exclude(solution);
    }
    words->pool.push_back(intersection.getIntersection());
  }
  return words;
}
```

```
void makePhrases(PoolWithUndelete &wordsPool, Pool *phrases) {
  PoolWithUndelete wordsAux;
  ExtIntersectionVetInt eintersection;
  while( ! wordsPool.isEmpty() ) {
    int length = wordsPool.getLength();
    int word = GNA::getInstance()->rand(length);
    VetInt *aWord;
    aWord = static_cast<VetInt*>( wordsPool.getElement(word));
    eintersection.removeAll();
    eintersection.extInclude(word, aWord);
    wordsAux = wordsPool;
    wordsAux.exclude(word);
    while(!wordsAux.isEmpty() && !eintersection.isComplete()) {
      length = wordsAux.getLength();
      word = GNA::getInstance()->rand(length);
      aWord = static_cast<VetInt*>( wordsAux.getElement(word));
      eintersection.extInclude(word, aWord);
      if( eintersection.hasInconsistence() )
        eintersection.removeLastInserted();
      wordsAux.exclude(word);
    }
    if( ! eintersection.isComplete() )
      eintersection.complete();
    phrases->pool.push_back(eintersection.getIntersection());
    wordsPool.exclude(eintersection.getSolutions());
  }
}

Pool* GrowVetInt::execute(Pool* words) {
  cout << " Making phrases... \n";
  Pool* phrases = new Pool();
  PoolWithUndelete wordsPool(words);
  makePhrases(wordsPool, phrases);
  return phrases;
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas