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.

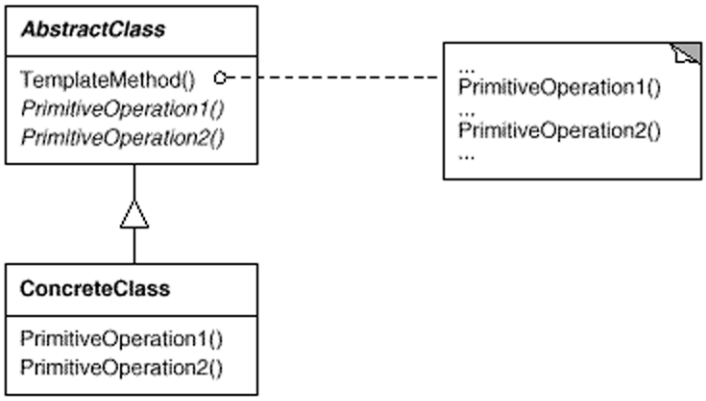

Figura 3.1: Estrutura do padrão *template method*

– **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.

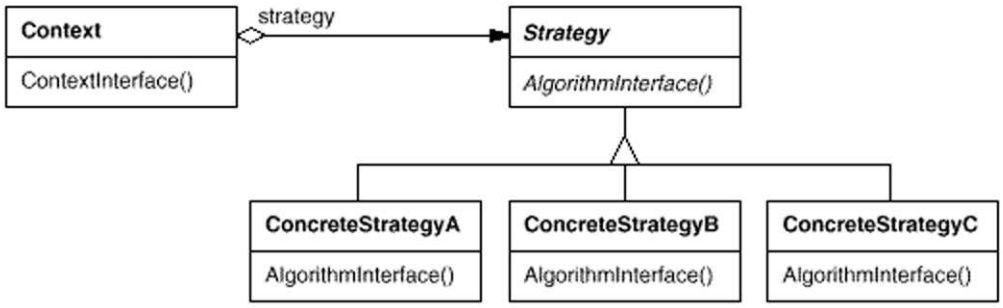

Figura 3.2: Estrutura do padrão *strategy*

- **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.

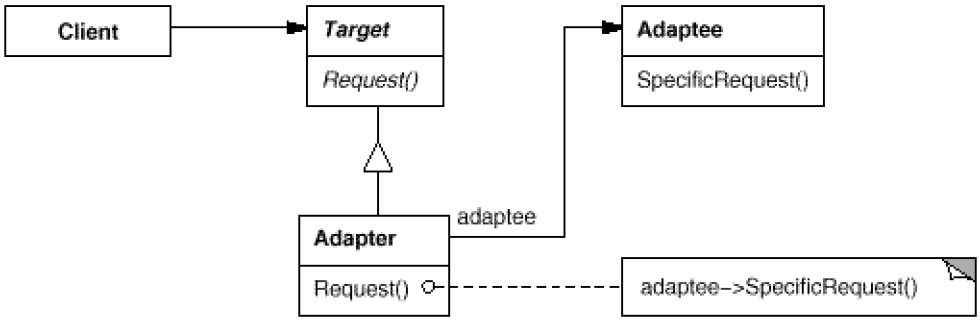

Figura 3.3: Estrutura do padrão *adapter*

- **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

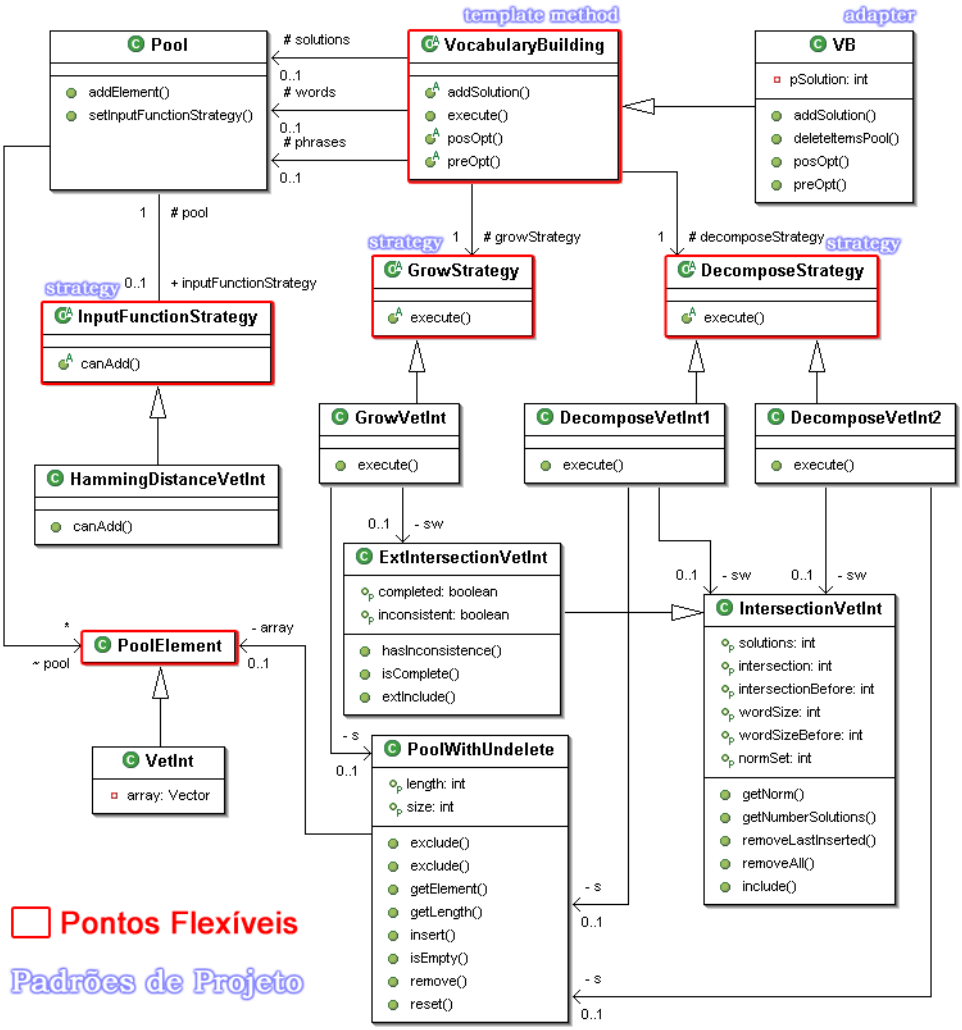

Figura 3.4: Diagrama de classes

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.