3
Algoritmos

Neste capítulo são apresentadas as heurísticas desenvolvidas neste trabalho e discutidos outros aspectos relativos aos algoritmos envolvidos na solução do PSBH. Na Seção 3.1 é apresentada uma discussão sobre os principais algoritmos propostos na literatura para solucionar o problema de montagem do seqüenciamento por hibridação. Antes de solucionar o PSBH é necessário realizar o cálculo das sobreposições entre cada par de provas do espectro, para determinar os valores da tabela o e, conseqüentemente, da tabela w. Uma solução para este problema é discutida na Seção 3.2. Na Seção 3.3 é apresentada uma heurística multi-partida para solucionar o PSBH. Esta heurística é bem simples e ineficiente, mas é a base para o desenvolvimento das outras três heurísticas mais competitivas, propostas neste trabalho. Na Seção 3.4 é apresentada uma melhoria para a heurística multi-partida baseada em uma estrutura de memória adaptativa e que resulta em uma nova heurística para o PSBH. Um procedimento de construção de vocabulário é apresentado na Seção 3.5 e resulta em mais duas heurísticas para o PSBH.

3.1
Revisão bibliográfica

Diversos algoritmos, exatos e aproximados, tratam do problema de montagem considerando somente falsos negativos ou somente falsos positivos [11, 23, 34]. Para o caso mais geral, existe apenas um algoritmo exato, baseado em "branch-and-bound" [6]. Entretanto, este método é incapaz de resolver em tempo razoável algumas instâncias de porte médio e grande. Por isto, surgiram heurísticas para tratar este problema. Em [34] é apresentado um novo algoritmo exato considerando apenas falsos positivos e uma idéia para tratar os falsos negativos heuristicamente. Um método distribuído baseado na metaheurística colônia de formigas [9] foi proposto em

[3]. Entretanto, estes dois métodos consideram informações adicionais sobre a freqüência de erros no espectro.

Com o objetivo de avaliar, na prática, o desempenho dos algoritmos propostos neste trabalho, são realizados testes para compará-los com os três algoritmos que obtiveram os melhores resultados até o momento. O primeiro é um algoritmo baseado na metaheurística busca tabu [15] que foi proposto em [5]. Neste algoritmo, uma solução inicial é criada concatenando-se provas do espectro de forma aleatória, até obter-se uma seqüência com o tamanho máximo. O procedimento de busca tabu utiliza movimentos de inserção, remoção e deslocamento. Uma inserção consiste em inserir uma prova do espectro que não esteja na solução corrente em alguma posição desta solução. A remoção consiste em retirar alguma prova da solução corrente. Já o deslocamento consiste em trocar a posição de uma prova dentro da seqüência de provas da solução atual. A escolha dos movimentos é realizada utilizando-se uma função objetivo diferenciada. Esta função é denominada *condensação* e definida como a divisão entre a quantidade de provas e o tamanho da seqüência de DNA resultante da solução. Isto é necessário para propiciar a comparação de movimentos de inserção com movimentos de remoção e deslocamento, que não são capazes de melhorar a função objetivo original (maximização da quantidade de provas na solução). Também é utilizada uma idéia de agrupamento de provas consecutivas que possuem sobreposição ótima, criando um elemento único com estas provas. Os movimentos de remoção e deslocamento são considerados para estes elementos também.

O segundo algoritmo é uma heurística que utiliza idéias simples mas obteve bons resultados [4]. Esta heurística utiliza um algoritmo construtivo para gerar um conjunto de seqüências de provas. O algoritmo construtivo é baseado em uma idéia denominada pelo autor como *janela de sobreposição*. As seqüências de provas são construídas de uma forma que cada prova do espectro é utilizada por no máximo uma seqüência do conjunto. Então, o algoritmo realiza algumas operações na tentativa de estender algumas seqüências do conjunto. Uma extensão é realizada através da concatenação de duas seqüências do conjunto. A intenção é chegar a uma única grande seqüência que seja a seqüência original.

O terceiro algoritmo utilizado aqui como base de comparação é aquele que alcançou os melhores resultados até agora. Este método é um algoritmo genético [16] e foi proposto em [13]. Ele utiliza uma regra baseada no alinhamento inexato entre cada par de provas para filtrar o espectro retirando os prováveis falsos positivos. A partir do espectro filtrado, o autor

propõe um modelo baseado no *problema do caixeiro viajante* (PCV). Os vértices do grafo são as provas do espectro filtrado e a distância entre dois vértices é determinada através do mesmo alinhamento inexato utilizado no processo de filtragem. O objetivo neste modelo é encontrar um caminho com peso mínimo que visite todos os vértices do grafo, pois o espectro está virtualmente livre de falsos positivos. Qualquer algoritmo sugerido para o PCV pode ser utilizado para resolver este modelo. O autor sugere a utilização de um algoritmo genético e apresenta os resultados para este algoritmo. Também é realizado um processo de *agrupamento* de alguns vértices para diminuir a quantidade de vértices do problema.

Apesar destas três heurísticas apresentarem os melhores resultados conhecidos até o momento, elas não foram capazes de solucionar várias instâncias não muito grandes. As taxas de acerto para instâncias de tamanho bem menor do que as seqüenciadas hoje em dia, através do método de corrida em gel, são muito baixas. Estes fatores motivam o desenvolvimento de novos algoritmos para o problema de montagem do SBH.

3.2 Cálculo das sobreposições entre provas

Neste trabalho assume-se que o espectro é fornecido como um conjunto de palavras sobre o alfabeto de nucleotídeos. Os valores da função o, que define a sobreposição máxima entre cada par de provas, são calculados com base na seqüência de nucleotídeos de cada prova. Existem várias formas de calcular estes valores e aqui será apresentada uma alternativa que utiliza *árvores de sufixo* [33] e possui complexidade $O(l \cdot |S|^2)$. Vale salientar que o problema de calcular as sobreposições possui complexidade $\Omega(|S|^2)$, pois a saída do procedimento é constituída por $|S|^2$ valores.

Uma árvore de sufixo permite a recuperação de diversas informações sobre um texto. É possivel, por exemplo, determinar se um padrão ocorre no texto em tempo linear no tamanho do padrão e encontrar todas as ocorrências do padrão em tempo linear no tamanho do texto. Uma árvore de sufixo é um caso especial da árvore *trie* (de "retrieval"). Uma *trie* é uma estrutura que armazena um conjunto de textos T. Se os textos de T estão definidos sobre o alfabeto Σ, então cada aresta da árvore é rotulada com um símbolo de Σ. Cada texto de T é representado por algum nó da árvore e pode ser recuperado pela concatenação dos símbolos no caminho da raiz até o nó que representa o texto. Nem todo nó representa um texto de T, alguns representam apenas prefixos de textos de T. Textos que possuem um prefixo

em comum compartilham o caminho da raiz até o nó que representa o prefixo comum. Um exemplo da *trie* que armazena os textos `ATAG`, `TAG`, `AG` e `G` é ilustrado na Figura 3.1. O primeiro texto está codificado no caminho que parte da raiz, segue sempre pelo filho mais a esquerda e vai até uma folha. Concatenando os símbolos das arestas utilizadas neste caminho, obtem-se o texto `ATAG`. Um nó é marcado como terminal se ele representa um texto do conjunto T (e não um prefixo de algum texto de T). Os nós terminais do exemplo estão marcados em preto. Os textos do exemplo são justamente todos os sufixos do texto `ATAG`. Dado um texto t, a árvore de sufixo de t é a *trie* de todos os seus sufixos. Existem algoritmos para construir a árvore de sufixo de um texto que possuem complexidade linear no tamanho do texto. Estes algoritmos não serão apresentados aqui pois são discutidos com detalhes em outros trabalhos [22, 33].

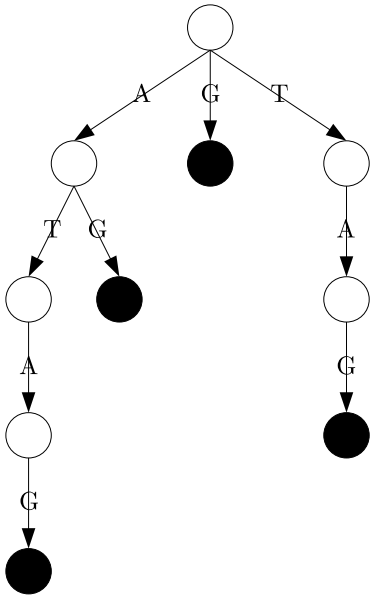


Figura 3.1: Árvore de sufixo do texto `ATAG`.

O algoritmo implementado neste trabalho para calcular os valores da função o utiliza as árvores de sufixo das provas do espectro. A complexidade para calcular as árvores de sufixo de $|S|$ textos com tamanho l é $O(l \cdot |S|)$. Utilizando as árvores de sufixo, o cálculo dos valores de o é feito independentemente para cada par ordenado (s_i, s_j) de provas. O cálculo do valor de $o(s_i, s_j)$, que é o tamanho da sobreposição máxima entre s_i e s_j, é realizado caminhando-se na árvore de sufixo de s_i guiado pelo texto de s_j. O procedimento mantém um *nó atual* e um *símbolo atual*. No início, o nó atual é a raiz da árvore de sufixo de s_i e o primeiro símbolo de s_j é o símbolo atual. A cada passo, o próximo nó é alcançado seguindo pela aresta do nó atual rotulada com o símbolo atual e este nó será o próximo nó

atual. O símbolo atual passa a ser o próximo símbolo de s_j. O procedimento também armazena a profundidade do último nó terminal visitado (distância da raiz até o nó), que será o valor atribuído a $o(s_i, s_j)$. O procedimento pára quando o nó atual não possui uma aresta rotulada com o símbolo atual. Considerando o exemplo da árvore de sufixo do texto `ATAG`, o cálculo da sobreposição máxima entre `ATAG` e `TAGC` é realizado como descrito a seguir. O percurso seguido na árvore será determinado pelo texto guia `TAGC`. Na raiz, a aresta escolhida é aquela rotulada com o símbolo `T` (pois é o primeiro do texto guia) e, então, move-se para o nó adjacente a ela. Em seguida, a escolha será pela única aresta do nó corrente, rotulada com o segundo símbolo do texto guia, `A`. No terceiro passo, a aresta seguida será aquela rotulada com o símbolo `G`, que é o terceiro símbolo do texto guia. Como o nó adjacente a esta aresta é terminal, a sua profundidade (3) é armazenada. No quarto passo, o algoritmo pára pois o nó atual não possui uma aresta rotulada com o símbolo `C`, que é o quarto do texto guia, e o valor de $o(s_i, s_j)$ é determinado como sendo igual a 3.

3.3 Heurística multi-partida

Heurísticas *multi-partida* são vastamente utilizadas para solucionar problemas de otimização combinatória, inclusive como base de algumas meta-heurísticas. A meta-heurística GRASP [27] é um exemplo. Uma heurística multi-partida utiliza um *algoritmo construtivo aleatorizado* para gerar várias soluções e retorna a melhor delas. Um algoritmo construtivo aleatorizado possui uma componente aleatória que lhe permite construir uma solução diferente a cada vez que é chamado.

O algoritmo construtivo aleatorizado (ou, simplesmente, algoritmo construtivo) constitui a principal componente de uma heurística multi-partida. O algoritmo construtivo proposto aqui utiliza um critério guloso e probabilístico na construção das soluções. Um *algoritmo guloso* parte de uma solução vazia e, a cada iteração, estende a solução parcial atual inserindo um elemento. Uma vez que um elemento é inserido na solução, nunca mais ele é removido. O elemento inserido é aquele que otimiza a *função gulosa*, definida no conjunto de elementos possíveis de serem incluídos na solução. O algoritmo construtivo apresentado aqui, além de guloso, é probabilístico porque o elemento a ser inserido na solução é escolhido probabilisticamente dentre uma *lista restrita de candidatos* (LRC). A LRC é constituída dos elementos que possuem os melhores valores para a função gulosa.

O algoritmo construtivo proposto para o PSBH parte de uma solução composta apenas da prova inicial $a = (s_0)$ e, a cada passo, estende o caminho associado a esta solução inserindo uma prova no seu final. A prova a ser inserida é escolhida probabilisticamente dentre aquelas do conjunto R, que representa a LRC e é definido como:

$$R = \{s \in S \backslash S(a) \mid o(u,s) \geq (1-\alpha) \cdot \max_{t \in S \backslash S(a)} o(u,t) \text{ e } w(a)+w(u,s) \leq n-l\}, \tag{3-1}$$

onde u é a última prova inserida em a e $\alpha \in [0;1]$ é um parâmetro do algoritmo. O algoritmo pára quando R torna-se vazio. A LRC contém provas com uma sobreposição mínima (definida pelo parâmetro α) com a última prova inserida na solução corrente, restringindo as construções do algoritmo a uma região mais promissora do espaço de soluções. A probabilidade de uma prova $s \in R$ ser inserida após a prova u é dada por

$$p(u,s) = \frac{v(u,s)}{\sum_{t \in R} v(u,t)}, \tag{3-2}$$

onde o valor $v(u,s)$ é definido como:

$$v(u,s) = \frac{\min_{t \in R} w(u,t)}{w(u,s)}. \tag{3-3}$$

Quanto maior o valor de $v(u,s)$, maior é a probabilidade de se inserir a prova s após a prova u. O valor de $v(u,s)$ é inversamente proporcional ao valor de $w(u,s)$ e, conseqüentemente, proporcional ao tamanho da sobreposição entre as provas u e s. Por isto, quanto maior a sobreposição entre a prova u e a prova s, maior será a probabilidade de inserir s após u na solução sendo construída. É importante salientar que neste momento o valor do numerador da Equação 3-3 é irrelevante. Se este numerador fosse igual a um, as probabilidades de cada prova não seriam alteradas. Para um dado R, o valor deste numerador é constante e é cancelado na Equação 3-2. Entretanto, utilizar o numerador $\min_{t \in R} w(u,t)$ será importante no momento em que a memória adaptativa for considerada (Seção 3.4). Este numerador resulta em uma normalização dos valores de $v(u,s)$, de modo que $v(u,s) = 1$ para $s = \text{argmax}_{t \in R} v(u,t)$.

O algoritmo construtivo proposto neste trabalho é denominado `ConstrutivoAleatorizado` e uma ilustração do seu pseudo-código é apresentada na Figura 3.2. Em relação à complexidade de pior caso do algoritmo, é fácil notar que o custo para preencher a LRC é $O(|S|)$, assim como o custo para calcular as probabilidades dos seus elementos e selecionar um deles.

Como, no pior caso, serão inseridas $|S|$ provas na solução, a complexidade deste algoritmo é $O(|S|^2)$.

A heurística multi-partida utilizando o algoritmo construtivo aleatorizado descrito acima é a primeira heurística proposta aqui para solucionar o PSBH e é denominada `MP`. Uma ilustração do seu pseudo-código é apresentada na Figura 3.3. O parâmetro n_{iter} determina quantas construções serão realizadas, ou seja, quantas vezes o algoritmo `ConstrutivoAleatorizado` será executado.

algoritmo `ConstrutivoAleatorizado`($S,\ l,\ n,\ s_0,\ o,\ w,\ \alpha$)
1. $a \leftarrow (s_0)$;
2. $R \leftarrow$ conjunto de provas que satisfazem o critério da Equação 3-1;
3. **enquanto** $R \neq \emptyset$ **faça**
4. Seleciona uma prova $s \in R$ de acordo com as probabilidades definidas pela Equação 3-2;
5. Insere s no final de a;
6. $R \leftarrow$ conjunto de provas que satisfazem o critério da Equação 3-1;
7. **fim**;
8. **retorna** a;

fim;

Figura 3.2: Algoritmo construtivo aleatorizado que compõe a heurística multi-partida proposta neste trabalho.

algoritmo `MP`($S,\ l,\ n,\ s_0$)
1. Inicia o, w, n_{iter}, α;
2. $a^* \leftarrow$ **nulo**;
3. **para** $i = 1 \ldots n_{iter}$ **faça**
4. $a \leftarrow$ `ConstrutivoAleatorizado`($S,\ l,\ n,\ s_0,\ o,\ w,\ \alpha$);
5. **se** $a^* =$ **nulo ou** $|a| > |a^*|$ **então** $a^* \leftarrow a$;
6. **fim**;
7. **retorna** a^*;

fim;

Figura 3.3: Heurística multi-partida proposta neste trabalho.

3.4 Memória adaptativa

Nas heurísticas multi-partida, geralmente, cada iteração é independente das demais. Ou seja, nenhuma informação gerada nas construções anteriores é utilizada na construção de uma nova solução. Entretanto, em [14]

foi sugerido uma heurística multi-partida que utiliza uma memória adaptativa para auxiliar a construção de novas soluções em cada iteração do algoritmo. Esta memória é baseada em um conjunto que mantém algumas das melhores soluções construídas nas iterações anteriores. Apresenta-se aqui uma heurística baseada nesta idéia.

A estrutura de memória é um *conjunto de soluções de elite* Q. Este conjunto armazena q soluções de boa qualidade e suficientemente diferentes umas das outras. No início, o conjunto contém q soluções *nulas* que possuem, por definição, zero provas cada uma. Uma solução a pode ser considerada *candidata* a entrar em Q se $|a| > \min_{a' \in Q} |a'|$ (i.e., se a for melhor do que alguma solução de Q). Para duas soluções a e a', seja $dist(a, a')$ uma medida de distância entre elas, de modo que, quanto maior o valor de $dist(a, a')$ maior será a diferença entre a e a'. Uma solução candidata a substituirá a pior solução de Q se $\min_{a' \in Q} dist(a, a') \geq d$, onde d é um parâmetro (i.e., se a for suficientemente diferente de todas as soluções de Q). Uma solução candidata a também substituirá a pior solução de Q se $|a| > \max_{a' \in Q} |a'|$ (i.e., se a for melhor do que todas as soluções de Q).

A idéia proposta em [14] consiste em utilizar um conjunto de soluções de elite em uma heurística multi-partida. Cada solução construída é considerada a entrar no conjunto de soluções de elite. No algoritmo construtivo, a freqüência com que a aresta (u, s) aparece nas soluções de elite do conjunto Q é considerada no cálculo da probabilidade de inserir a prova s após a prova u. Para isto, define-se

$$e(u, s) = \lambda \cdot v(u, s) + i(u, s), \tag{3-4}$$

onde $v(u, s)$ é o valor definido na Equação 3-3 que é baseado na sobreposição das provas u e s, $i(u, s)$ depende da freqüência de utilização da aresta (u, s) em Q e λ é um parâmetro utilizado para balancear os dois valores. O valor de $i(u, s)$ é definido como:

$$i(u, s) = \frac{\sum_{a'' \in Q | (u,s) \in E(a'')} |a''|}{\max_{a' \in Q} |a'|}. \tag{3-5}$$

Desta forma, a probabilidade de inserir a prova $s \in R$ após a prova u em uma iteração do algoritmo construtivo é redefinida como:

$$p(u, s) = \frac{e(u, s)}{\sum_{t \in R} e(u, t)}. \tag{3-6}$$

Utilizando este novo valor no lugar daquele dado pela Equação 3-2, no

critério da linha 4 do algoritmo `ConstrutivoAleatorizado`, é possível realizar construções que privilegiam os elementos que aparecem com maior freqüência nas melhores soluções construídas nas iterações anteriores da heurística multi-partida. O algoritmo construtivo aleatorizado modificado desta maneira é denominado `ConstrutivoAleatorizadoMem` e recebe dois argumentos a mais, o conjunto de soluções de elite Q e o parâmetro λ. Observa-se que a Equação 3-5 leva em consideração, além da freqüência de utilização da aresta, a qualidade das soluções que a utilizam. A abordagem apresentada acima é baseada na freqüência de utilização das arestas nas soluções de elite. Outra abordagem seria considerar apenas a freqüência de utilização de cada prova. Entretanto, a abordagem por arestas é mais atraente pois permite que a memória indique sub-seqüências de provas muito boas. Isto é possível porque uma aresta indica a posição relativa entre duas provas.

A heurística multi-partida utilizando memória adaptativa proposta neste trabalho é denominada `MP+Mem` e uma ilustração do seu pseudo-código é apresentada na Figura 3.4. O valor do parâmetro λ deve ser grande durante as primeiras iterações do algoritmo, quando a qualidade das soluções do conjunto Q ainda é incerta. A medida que mais construções são realizadas, a importância da parcela $i(u, s)$ deve ser aumentada através da diminuição do valor do parâmetro λ. O parâmetro α também deve ser ajustado no descorrer da heurística multi-partida para permitir que o algoritmo guloso probabilístico construa soluções diferentes quando está sendo guiado pela memória. Estes aspectos são discutidos com mais detalhes no Capítulo 4.

algoritmo `MP+Mem`($S,\ l,\ n,\ s_0$)
1. Inicia o, w, n_{iter}, α, λ, q, d e Q;
2. $a^* \leftarrow$ **nulo**;
3. **para** $i = 1 \ldots n_{iter}$ **faça**
4. $\quad a \leftarrow$ `ConstrutivoAleatorizadoMem`($S,\ l,\ n,\ s_0,\ o,\ w,\ \alpha,\ Q,\ \lambda$);
5. $\quad$ **se** $a^* =$ **nulo ou** $|a| > |a^*|$ **então** $a^* \leftarrow a$;
6. $\quad$ Atualiza Q com a;
7. $\quad$ Se necessário, atualiza os parâmetros α e λ;
8. **fim**;
9. **retorna** a^*;

fim;

Figura 3.4: Heurística multi-partida com memória adaptativa.

A complexidade do algoritmo construtivo não é alterada com a utilização da memória adaptativa desde que alguns cuidados sejam tomados na implementação do conjunto de soluções de elite. Em primeiro lugar, é

importante salientar que na implementação de algoritmos para o PSBH é conveniente representar o espectro como um vetor de provas (cada prova pode ser, simplesmente, uma seqüência de caracteres). Desta forma, as provas do espectro podem ser referenciadas apenas pelos índices do vetor do espectro, ou seja, por inteiros entre 1 e $|S|$. Portanto, daqui em diante as provas do espectro serão tratadas como inteiros do conjunto $\{1, 2, \ldots, |S|\}$. Voltando à implementação do conjunto de soluções de elite, considera-se uma matriz M de inteiros com $|S|$ linhas e $|S|$ colunas, de forma que $M[u][s]$ é uma referência para o valor armazenado na linha u e coluna s de M, onde $u, s \in S$ (i.e., $u, s \in \{1, 2, \ldots, |S|\}$). Os valores de M são iniciados com zero e o valor $M[u][s]$ irá armazenar a soma dos custos das soluções de Q que utilizam a aresta (u, s). Quando uma solução a é inserida em Q, o valor $M[u][s]$ é aumentado de $|a|$ para cada aresta $(u, s) \in E(a)$. Quando uma solução é removida, os valores correspondentes são diminuídos de maneira similar. Se o custo da melhor solução do conjunto de elite também é armazenado, então o cálculo da Equação 3-5 pode ser realizado em $O(1)$, para cada aresta, nas iterações do algoritmo construtivo (o numerador para cada aresta está acumulado na matriz M e o denominador é o custo da melhor solução em Q).

A complexidade de realizar a inserção de uma solução no conjunto de elite depende diretamente do custo para calcular o valor $dist(a, a')$ pois, no pior caso, este valor será calculado entre a solução candidata e cada solução do conjunto. Utilizando-se uma representação adequada para as soluções, este cálculo pode ser realizado em $O(|S|)$. Uma solução consiste em um caminho no grafo G e pode ser representada por um conjunto de arestas de G. Apesar de haver $|S|^2$ arestas em G, uma solução não pode conter mais do que $|S|$ arestas pois representa um caminho acíclico e portanto, disjunto por vértices. Isto implica que no máximo uma aresta sai de cada vértice. Por isto, uma solução a pode ser representada por um *vetor de adjacência* $x^a = \langle x_1^a, x_2^a, \ldots, x_{|S|}^a \rangle$, tal que x_u^a é uma variável cujo valor determina a prova que vem a seguir da prova $u \in S$ no caminho da solução a. Uma variável x_u^a pode ser nula se a prova u não é utilizada na solução a ou é a última prova da solução (que não possui sucessor). Desta forma, o valor $dist(a, a')$ é igual à $\sum_{u \in S \mid x_u^a \neq x_u^{a'}} 1$, ou seja, é igual à quantidade de variáveis com valor diferente nos vetores de adjacência x^a e $x^{a'}$. Então, a complexidade de inserir uma solução no conjunto é igual a $O(q \cdot |S|)$ mais a complexidade de atualizar a matriz M, que é $O(|S|)$. Portanto, a complexidade total é $O(q \cdot |S|)$.

3.5 Construção de vocabulário

Uma característica freqüente nas soluções ótimas para o PSBH são os sub-caminhos com sobreposição ótima. Isto significa que uma solução ótima para o problema, geralmente, é composta por alguns sub-caminhos nos quais as provas consecutivas possuem sobreposição ótima, ou seja, todas as arestas destes sub-caminhos possuem peso unitário. Estes sub-caminhos são conectados uns aos outros por arestas com peso maior do que um, formando a solução completa. Esta característica motiva a implementação de alguma técnica capaz de encontrar sub-caminhos com sobreposição ótima e tratá-los como elementos básicos para construir soluções combinando-os. O método de *construção de vocabulário* foi proposto por Glover e Laguna em [15] e trata de aspectos desta natureza, apesar de ser baseado em idéias mais gerais e robustas do que esta. A denominação do método vem do processo de construção de vocabulário de linguagens naturais, onde frases e sentenças gramaticalmente corretas são formadas por elementos básicos da linguagem, as palavras. Na construção de vocabulário proposta por Glover e Laguna, o objetivo é encontrar partes comuns a boas soluções e então combiná-las na tentativa de construir novas soluções melhores.

Um pré-requisito para a aplicação da construção de vocabulário é codificar uma solução do problema como um vetor de variáveis, cujos valores determinam as propriedades da solução. Esta representação deve ser definida de forma que qualquer solução seja representada por um vetor de mesmo tamanho e uma variável sempre diz respeito a uma mesma propriedade da solução. No caso do PSBH, uma solução pode ser representada desta forma como um vetor de adjacência (discutido na Seção 3.4). Cada variável não nula do vetor define uma propriedade da solução. No caso do PSBH, uma propriedade corresponde a uma aresta do grafo G.

Buscando palavras

O primeiro objetivo da construção de vocabulário é encontrar partes comuns a boas soluções. No caso do PSBH, uma parte de uma solução consiste de um conjunto de arestas e é representada por um vetor de adjacência, onde as variáveis com valor não-nulo definem as arestas do conjunto. Por isto, um vetor de adjacência pode representar tanto uma solução completa (viável) como uma solução incompleta, ou seja, um conjunto de arestas que não necessariamente definem uma solução viável. Um vetor que representa arestas comuns a várias soluções boas é denominado *palavra*. Como

ferramenta para o procedimento de busca por palavras, Glover e Laguna definiram um operador de interseção entre vetores, que extrai as propriedades comuns a eles. Dados dois vetores x, x', a operação $\text{Int}(x, x')$ produz um novo vetor y tal que:

$$y_u = \begin{cases} x_u, & \text{se } x_u = x'_u \\ \text{nulo}, & \text{caso contrario} \end{cases}, \ \forall \ u \in S. \tag{3-7}$$

Este operador produz um vetor de adjacência que possui somente as arestas comuns aos dois operandos. Um exemplo da aplicação deste operador é ilustrado na Figura 3.5. Neste exemplo, são apresentados três grafos: os dois operandos na parte superior e o grafo resultante na parte inferior da figura. Os vetores de adjacência correspondentes a cada grafo aparecem logo abaixo deles. Claramente, este operador é associativo e pode ser aplicado a um conjunto de vetores. Para um conjunto de vetores X, a operação $\text{Int}(X)$ produz um novo vetor y tal que:

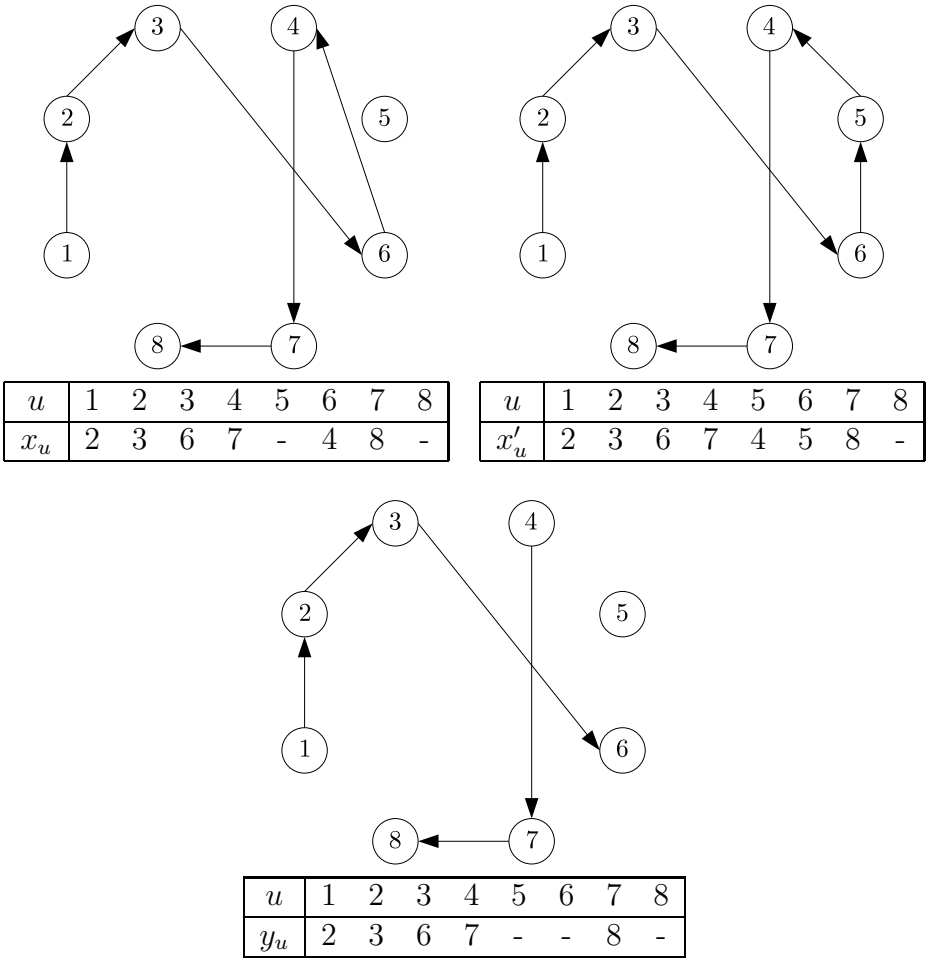
$$y_u = \begin{cases} v, & \text{se } x_u = v, \ \forall \ x \in X \\ \text{nulo}, & \text{caso contrario} \end{cases}, \ \forall \ u \in S. \tag{3-8}$$

Uma exemplo da aplicação deste operador sobre um conjunto é ilustrado na Figura 3.6.

Também torna-se nececessário a definição de alguns valores com relação à qualidade de um vetor de adjacência. Estes valores possibilitam a definição de regras para classificar um vetor de adjacência, indicando se ele é uma palavra ou não. O valor $\text{Size}(y)$ indica a quantidade de variáveis do vetor de adjacência y que possuem valor não-nulo, ou seja, é a quantidade de arestas definidas por y. No exemplo da Figura 3.6, tem-se $\text{Size}(x) = 6$, $\text{Size}(x') = 7$, $\text{Size}(x'') = 7$ e $\text{Size}(y) = 3$. Dados dois vetores de adjacência x e y, seja o operador $\subseteq$ definido de modo que $y \subseteq x$ implica que $y_u = x_u$ ou $y_u = \text{nulo}$ $\forall u \in S$. Ou seja, o vetor de adjacência x contém todas as arestas definidas em y. Em outras palavras, as arestas do vetor de adjacência y estão contidas em x. Desta forma, dados um conjunto de vetores de adjacência X e um vetor de adjacência y define-se o conjunto

$$\text{Enclosure}(y, X) = \{x \in X \mid y \subseteq x\}, \tag{3-9}$$

composto dos vetores de adjacência do conjunto X que contêm y. Assim, outro valor é definido como $\text{EncValue}(y, X) = |\text{Enclosure}(y, X)|$ e indica a quantidade de vetores de adjacência do conjunto X que contêm o vetor de adjacência y.



u	1	2	3	4	5	6	7	8
x_u	2	3	6	7	-	4	8	-

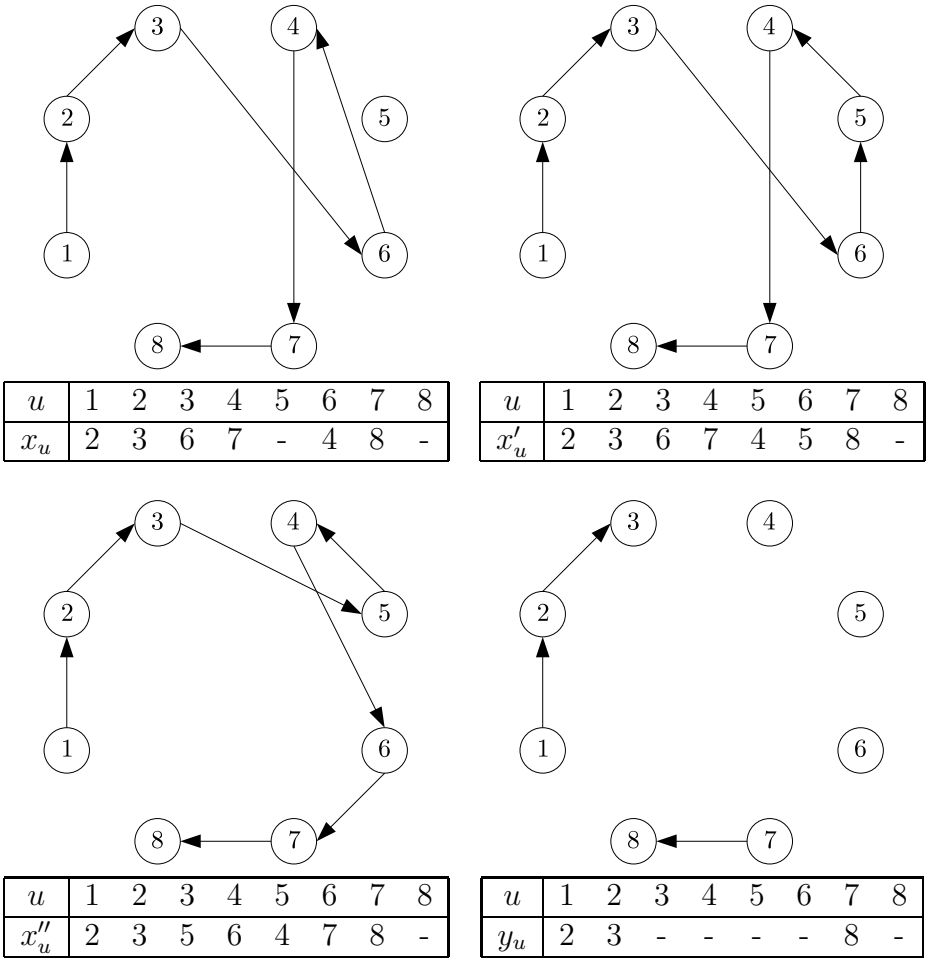
u	1	2	3	4	5	6	7	8
x'_u	2	3	6	7	4	5	8	-

u	1	2	3	4	5	6	7	8
y_u	2	3	6	7	-	-	8	-

Figura 3.5: Operação $y = \text{Int}(x, x')$.

Utilizando o operador Int e as definições de Size e EncValue, Glover e Laguna sugeriram duas abordagens para o projeto de um algoritmo que busca palavras em um dado conjunto de vetores de adjacência X. Em ambas as abordagens, uma palavra será definida pelo resultado da operação de interseção sobre um subconjunto de X. A primeira abordagem consiste em construir vetores y que possuam o maior valor possível para $\text{EncValue}(y, X)$, respeitando a restrição $\text{Size}(y) \geq s_{min}$, para um dado valor s_{min}. A outra abordagem consiste em construir vetores y que possuam o maior valor possível para $\text{Size}(y)$, respeitando a restrição $\text{EncValue}(y, X) \geq v_{min}$, para um dado v_{min}.

Apresenta-se um algoritmo denominado `BuscaPalavras` para encontrar palavras a partir de um conjunto de vetores de adjacência baseado na primeira abordagem citada acima. Este algoritmo é descrito a seguir e seu pseudo-código é ilustrado na Figura 3.7. Dados um conjunto de vetores de adjacência X e um inteiro s_{min}, o algoritmo `BuscaPalavras` gera um conjunto Y de palavras tal que $\forall\ y \in Y,\ \text{Size}(y) \geq s_{min}$ e $y = \text{Int}(X^y)$, onde $X^y \subseteq X$. Em cada iteração do algoritmo, uma palavra y é gerada e adici-



u	1	2	3	4	5	6	7	8
x_u	2	3	6	7	-	4	8	-

u	1	2	3	4	5	6	7	8
x'_u	2	3	6	7	4	5	8	-

u	1	2	3	4	5	6	7	8
x''_u	2	3	5	6	4	7	8	-

u	1	2	3	4	5	6	7	8
y_u	2	3	-	-	-	-	8	-

Figura 3.6: Operação $y = \text{Int}(X)$, onde $X = \{x, x', x''\}$.

onada ao conjunto Y. Na realidade, em cada iteração, um conjunto X^y é gerado e no final da iteração faz-se $y \leftarrow \text{Int}(X^y)$ e $Y \leftarrow Y \cup \{y\}$. No decorrer do algoritmo, o conjunto $X' \subseteq X$ armazena os vetores de adjacência que ainda não foram utilizados para criar alguma palavra, ou seja, os elementos de X que em nenhuma iteração foram incluídos em um conjunto X^y.

No início de cada iteração, seleciona-se aleatoriamente um vetor x do conjunto X', ou seja, seleciona-se um vetor que ainda não foi utilizado na geração de alguma palavra. Então, faz-se $X^y \leftarrow \{x\}$ e $X'' \leftarrow X \backslash \{x\}$. No decorrer da iteração, o conjunto X'' armazena os vetores de X que ainda não foram considerados para entrar no conjunto X^y que está sendo construído. A cada passo desta iteração, um outro vetor x é selecionado aleatoriamente dentre os elementos de X'' e faz-se $X'' \leftarrow X'' \backslash \{x\}$. Caso $\text{Size}(\text{Int}(X^y \cup \{x\})) \geq s_{min}$, então faz-se $X^y \leftarrow X^y \cup \{x\}$. A iteração termina quando o conjunto X'' torna-se vazio, ou seja, quando todos os elementos de X já tenham sido considerados para entrar no conjunto X^y. Ao final de uma iteração, faz-se $X' \leftarrow X' \backslash X^y$ e, se $|X^y| > 1$, o vetor $y = \text{Int}(X^y)$ é adicionado ao conjunto Y. Em seguida, o algoritmo passa para a próxima

iteração. O algoritmo pára quando o conjunto X' torna-se vazio, ou seja, quando todos os elementos de X foram, em alguma iteração, incluídos num conjunto X^y.

```
algoritmo BuscaPalavras(X, s_min)
 1.  Y ← ∅;
 2.  X' ← X;
 3.  enquanto X' ≠ ∅ faça
 4.      x ← seleciona aleatoriamente um elemento de X';
 5.      X^y ← {x};
 6.      X'' ← X\{x};
 7.      enquanto X'' ≠ ∅ faça
 8.          x ← seleciona aleatoriamente um elemento de X'';
 9.          se Size(Int(X^y ∪ {x})) ≥ s_min então X^y ← X^y ∪ {x};
10.          X'' ← X''\{x};
11.      fim;
12.      se |X^y| > 1 então
13.          y ← Int(X^y);
14.          Y ← Y ∪ {y};
15.      fim;
16.      X' ← X'\{X^y};
17.  fim;
18.  retorna Y;
fim;
```

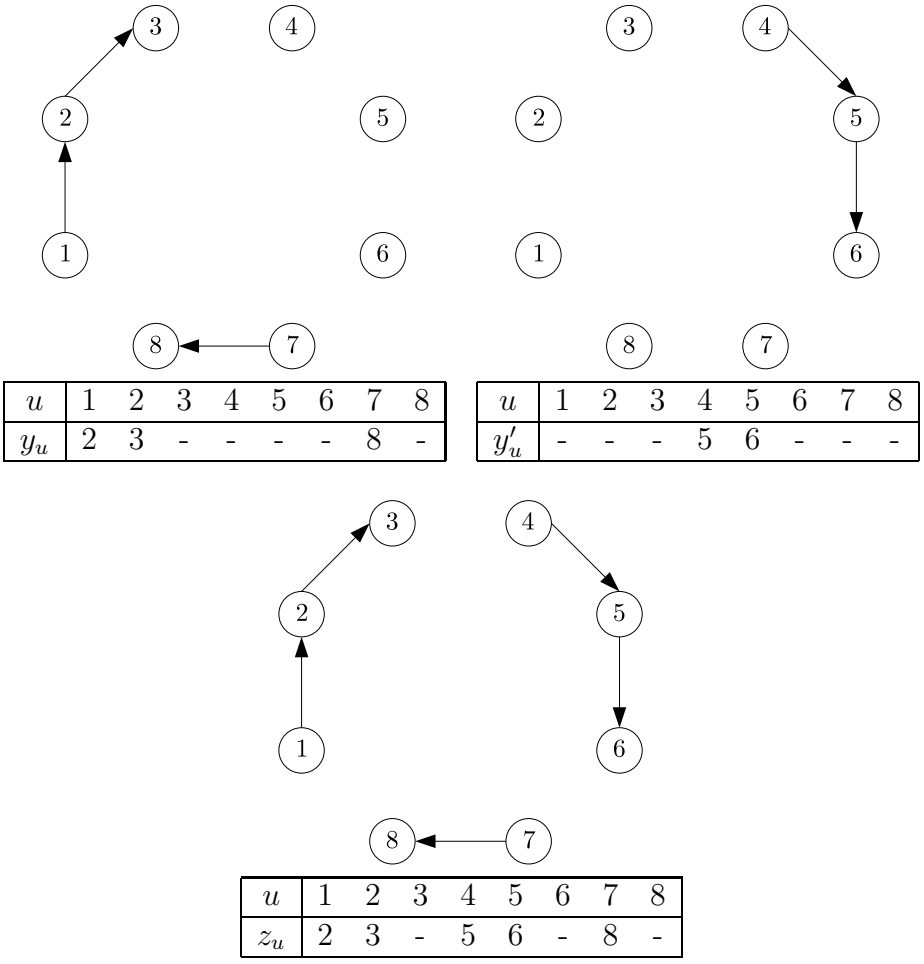
Figura 3.7: Algoritmo para encontrar palavras.

Combinando palavras

A partir do momento que tem-se um conjunto de palavras, o objetivo da construção de vocabulário é combiná-las no intuito de formar novas soluções ainda melhores do que aquelas utilizadas para encontrar as palavras. Para isto, Glover e Laguna definiram outro operador, estendendo a definição do operador de interseção. Este operador é definido para combinar vetores de adjacência de uma forma que os valores das variáveis não-nulas de um, sejam atribuídos às variáveis nulas do outro. Dados dois vetores y, y', a operação $\mathrm{EInt}(y, y')$ produz um novo vetor z tal que:

$$z_u = \begin{cases} y_u, & \text{se } y_u = y'_u \text{ ou } y'_u = \text{nulo} \\ y'_u, & \text{se } y_u = \text{nulo} \\ \text{nulo}, & \text{caso contrario} \end{cases}, \ \forall\ u \in S. \qquad (3\text{-}10)$$

Um exemplo da aplicação deste operador é ilustrado na Figura 3.8. Assim como o operador Int, este operador também é associativo e pode ser definido



u	1	2	3	4	5	6	7	8
y_u	2	3	-	-	-	-	8	-

u	1	2	3	4	5	6	7	8
y'_u	-	-	-	5	6	-	-	-

u	1	2	3	4	5	6	7	8
z_u	2	3	-	5	6	-	8	-

Figura 3.8: Operação $z = \text{EInt}(y, y')$.

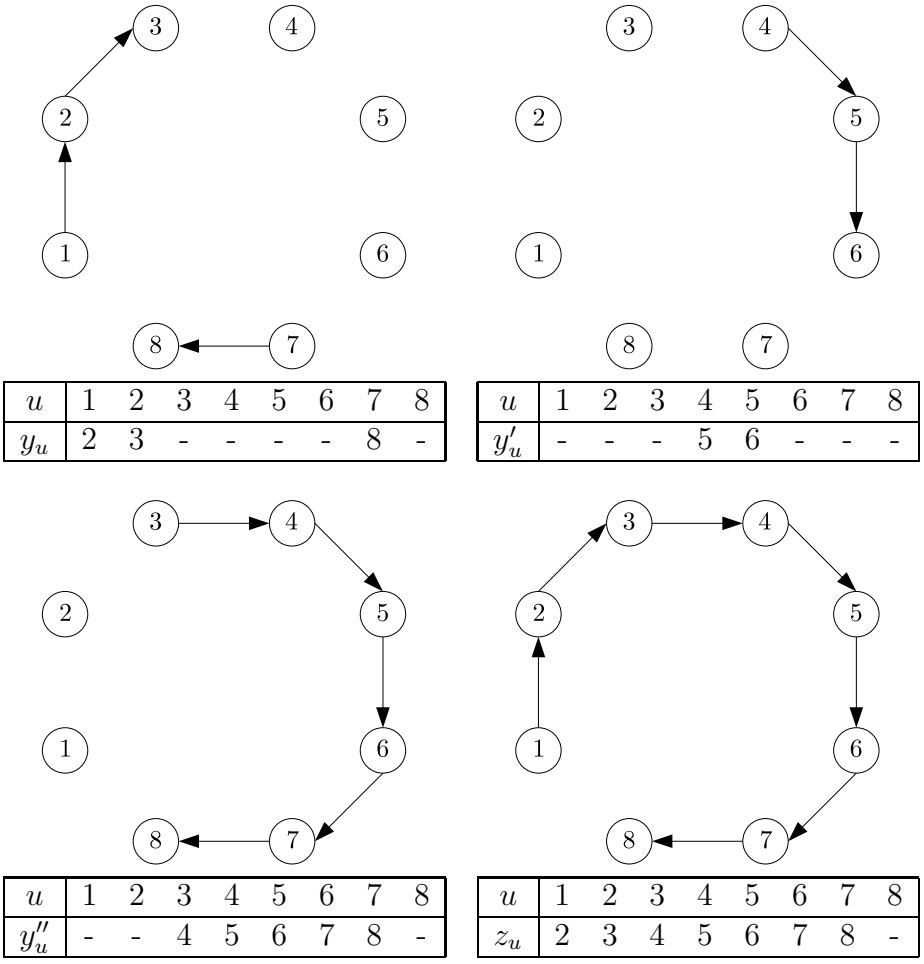
para um conjunto de vetores. Dado um conjunto de vetores Y, a operação EInt(Y) produz um novo vetor z tal que:

$$z_u = \begin{cases} v, & \text{se } y_u = \text{nulo ou } y_u = v, \ \forall\ y \in Y \\ \text{nulo}, & \text{caso contrario} \end{cases}, \ \forall\ u \in S. \qquad (3\text{-}11)$$

Um exemplo da aplicação deste operador sobre um conjunto de vetores é ilustrado na Figura 3.9.

Vetores gerados através do operador EInt são chamados de *frases*. O algoritmo proposto aqui para gerar frases é muito parecido ao algoritmo `BuscaPalavras`. A diferença está apenas no operador utilizado e na regra de aceitação de uma operação (linha 9 do algoritmo `BuscaPalavras`). Em cada iteração, o algoritmo gera uma frase (z) combinando um conjunto de palavras (Y^z) dentre aquelas fornecidas (Y), utilizando o operador EInt. Este algoritmo é denominado `CombinaPalavras` e seu pseudo-código é ilustrado na Figura 3.10.

O objetivo da fase de construção de frases é gerar frases que repre-



u	1	2	3	4	5	6	7	8
y_u	2	3	-	-	-	-	8	-

u	1	2	3	4	5	6	7	8
y'_u	-	-	-	5	6	-	-	-

u	1	2	3	4	5	6	7	8
y''_u	-	-	4	5	6	7	8	-

u	1	2	3	4	5	6	7	8
z_u	2	3	4	5	6	7	8	-

Figura 3.9: Operação $z = \text{EInt}(Y)$, onde $Y = \{y, y', y''\}$.

sentem soluções viáveis para o PSBH. Uma solução viável consiste de um caminho sem ciclos, com início no vértice s_0 e cujo peso total não excede $n - l$. A aplicação do operador EInt a um conjunto de palavras não gera, necessariamente, uma frase que representa uma solução viável. Entretanto, após a execução do algoritmo `CombinaPalavras` alguns procedimentos podem ser realizados na tentativa de gerar uma solução viável a partir de cada frase construída. Frases geradas com o operador EInt podem violar diferentes restrições do problema. Algumas destas restrições merecem atenção especial pois a sua violação significa fortes conflitos entre as palavras envolvidas. Nestes casos, a frase gerada contém informações conflitantes que torna difícil a geração de uma solução viável a partir dela. Existem duas restrições deste tipo. Uma delas é denominada *grau máximo de entrada* e garante que no máximo uma aresta pode chegar em cada vértice. Um exemplo da violação desta restrição após a aplicação do operador EInt sobre dois vetores que respeitam esta restrição é descrito a seguir e ilustrado na Figura 3.11. Sejam dois vetores de adjacência y, y' e três vértices $u, v, t \in S$. Se $y_u = t$, $y_v =$ nulo, $y'_u =$ nulo e $y'_v = t$, o vetor $z = \text{EInt}(y, y')$ será tal

```
algoritmo CombinaPalavras(Y)
 1.  Z ← ∅;
 2.  Y' ← Y;
 3.  enquanto Y' ≠ ∅ faça
 4.      y ← seleciona aleatoriamente um elemento de Y';
 5.      Y^z ← {y};
 6.      Y'' ← Y\{y};
 7.      enquanto Y'' ≠ ∅ faça
 8.          y ← seleciona aleatoriamente um elemento de Y'';
 9.          se Verifica(EInt(Y^z), y) = verdadeiro então Y^z ← Y^z∪{y};
10.          Y'' ← Y''\{y};
11.      fim;
12.      se |Y^z| > 1 então
13.          z ← EInt(Y^z);
14.          Z ← Z ∪ {z};
15.      fim;
16.      Y' ← Y'\{Y^z};
17.  fim;
18.  retorna Z;
fim;
```

Figura 3.10: Algoritmo que combina as palavras fornecidas para gerar frases.

que $z_u = z_v = t$. Isto significa que o operador EInt pode produzir vetores que possuem mais de uma aresta chegando no mesmo vértice, o que viola a restrição de grau máximo de entrada.

Outra restrição do PSBH é denominada *grau máximo de saída*. Esta restrição garante que no máximo uma aresta sai de cada vértice. Esta restrição é garantida automaticamente devido à representação das frases como vetores de adjacência, pois somente uma variável define a aresta que sai de cada vértice. Entretanto, a operação EInt pode ser aplicada a vetores que possuem diferentes arestas saindo do mesmo vértice. Estas informações são conflitantes porque não há como combiná-las em uma solução viável. Um exemplo desta situação é ilustrado na Figura 3.12 e descrito a seguir. Novamente, sejam dois vetores de adjacência y, y' e três vértices $u, v, t \in S$. Se $y_u = v$ e $y'_u = t$, o vetor $z = \text{EInt}(y, y')$ será tal que $z_u =$ nulo, pois $y_u \neq y'_u$. O resultado não viola a restrição de grau máximo de saída para o vértice u. Entretanto, estas informações são conflitantes em relação ao significado dos vetores y e y'. O primeiro indica que a melhor opção após o vértice u é seguir para v e o outro indica que é melhor seguir para t. Por isto, operações deste tipo são tratadas como se violassem a restrição de grau máximo de saída.

Operações que violam as restrições de grau máximo são evitadas pelo

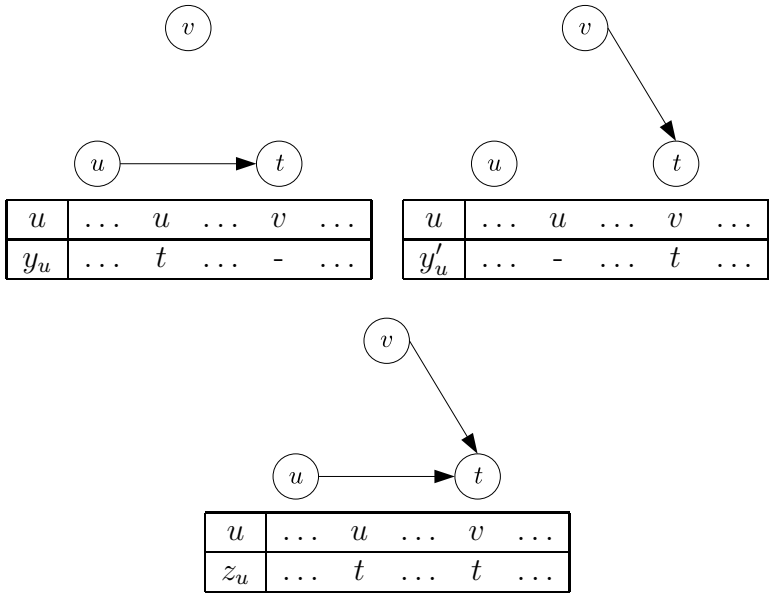


Figura 3.11: Operação $z = \text{EInt}(y, y')$. O resultado viola a restrição de grau máximo de entrada (no máximo uma aresta pode chegar a um vértice).

algoritmo `CombinaPalavras`. A verificação destas restrições é realizada pelo procedimento `Verifica` (linha 9 do algoritmo `CombinaPalavras`). Este procedimento recebe como parâmetro dois vetores de adjacência y, y' e retorna verdadeiro se o vetor $z = \text{EInt}(y, y')$ não viola as restrições de grau máximo. Caso contrário, este procedimento retorna falso e o algoritmo `CombinaPalavras` não insere a palavra selecionada (y) no conjunto que representa a frase em construção (Y^z).

Viabilizando frases

As restrições verificadas durante o procedimento de geração de frases garantem que elas serão compostas apenas por caminhos disjuntos por vértices. Entretanto, poderá haver mais de um caminho com peso total maior do que $n - l$. Os caminhos poderão ser cíclicos e não há garantia de que o vértice inicial s_0 seja utilizado por algum caminho. Alguns exemplos de frases geradas pelo algoritmo `CombinaPalavras` são ilustrados na Figura 3.13. Alguns procedimentos podem ser realizados para tentar gerar uma solução viável a partir de uma frase construída pelo algoritmo `CombinaPalavras`. Um algoritmo para este fim é proposto neste trabalho e denominado `Viabiliza`. Este algoritmo é descrito a seguir e seu pseudo-código é ilustrado na Figura 3.14.

Seja z um vetor de adjacência que representa uma frase gerada pelo algoritmo `CombinaPalavras` e que é passado ao algoritmo `Viabiliza`. Na

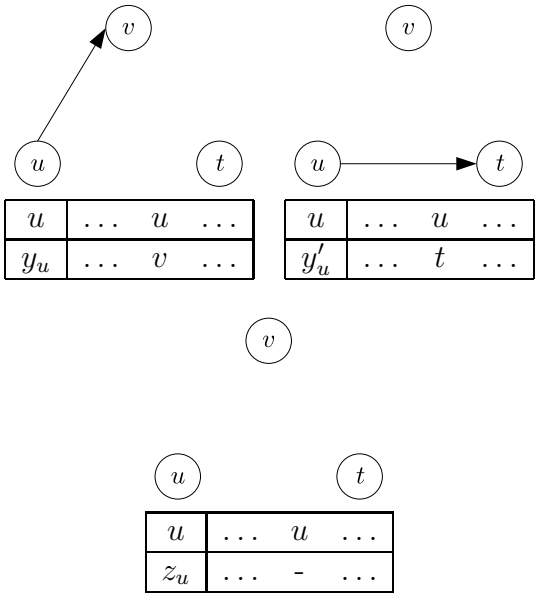
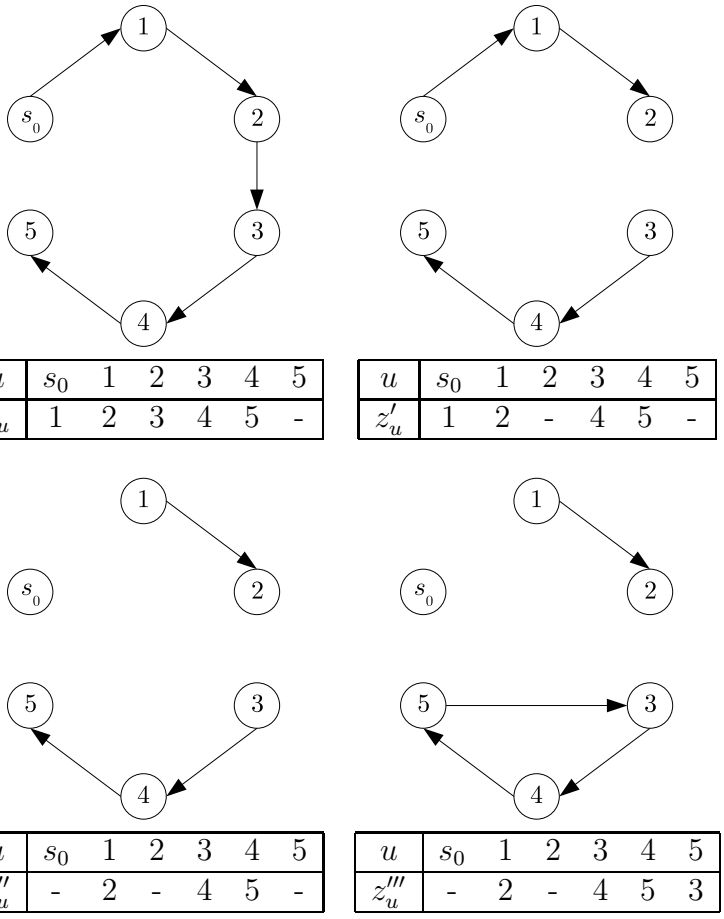


Figura 3.12: Operação $z = \text{EInt}(y, y')$. O resultado não viola a restrição de grau máximo de saída. Entretanto, o vértice u possui diferentes sucessores nos vetores y e y', o que é conflitante em relação ao significado destas informações.

tentativa de construir uma solução viável a partir de z, é necessário, em primeiro lugar, verificar se a prova inicial s_0 é utilizada em algum caminho de z. Se esta prova é utilizada em algum caminho ($z_{s_0} \neq$ nulo), então ela está no início deste caminho. Isto porque nenhuma solução viável utiliza uma aresta que chega na prova s_0. Logo, uma aresta deste tipo não estará presente em nenhuma palavra e, conseqüentemente, em nenhuma frase ($z_v \neq s_0,\ \forall v \in S$). Caso a prova s_0 não seja utilizada por nenhum caminho ($z_{s_0} =$ nulo), pode-se tentar inserí-la em algum caminho de z. Neste caso, o algoritmo `InserePrimeiraProva` é chamado. Este algoritmo é descrito a seguir e seu pseudo-código é ilustrado na Figura 3.15. No algoritmo `InserePrimeiraProva`, percorre-se as arestas de G adjacentes a s_0, em ordem crescente de peso. Escolhe-se a primeira aresta que leve a um vértice u que esteja na frase z ($u = \text{argmin}_{v \in S | z_v \neq \text{nulo}} w(s_0, v)$). Se a sobreposição entre s_0 e u é nula, então a frase z é descartada e o algoritmo retorna uma solução nula. Caso contrário, a prova s_0 é inserida na frase z, imediatamente antes de u, fazendo-se $z_{s_0} \leftarrow u$. Antes disto, é necessário verificar se já existe alguma prova v antes de u na frase z ($\exists\ v \in S$ tal que $z_v = u$, veja Figura 3.16). Caso exista, faz-se $z_v \leftarrow$ nulo. Se o caminho de z que contém a prova u for cíclico, as operações $z_v \leftarrow$ nulo e $z_{s_0} \leftarrow u$ o tornam acíclio e com início na prova s_0 (Figura 3.16). Se este caminho já for acíclico mas sua primeira prova não for u, estas operações criam um caminho a partir de s_0 e outro caminho com a parte anterior a u (Figura 3.17). Por outro lado,



u	s_0	1	2	3	4	5
z_u	1	2	3	4	5	-

u	s_0	1	2	3	4	5
z'_u	1	2	-	4	5	-

u	s_0	1	2	3	4	5
z''_u	-	2	-	4	5	-

u	s_0	1	2	3	4	5
z'''_u	-	2	-	4	5	3

Figura 3.13: Exemplos de frases geradas pelo algoritmo `CombinaPalavras`. A frase z é ideal, pois representa um único caminho que já contém a prova inicial s_0. A frase z' representa dois caminhos e um deles contém a prova inicial. A frase z'' também representa dois caminhos mas a prova inicial não está em nenhum deles. Já a frase z''', representa dois caminhos, não contém a prova inicial e um dos caminhos é cíclico.

se o caminho que contém u for acíclico e sua primeira prova for u, estas operações somente inserem s_0 no início deste caminho (Figura 3.18).

A partir do momento que a prova s_0 está na frase z, o algoritmo `Viabiliza` inicia a construção da solução a utilizando os caminhos definidos pela frase z. Uma ilustração do conteúdo da frase z e da solução a no decorrer do algoritmo `Viabiliza` é apresentada na Figura 3.19. Inicialmente, a solução a é vazia. A cada iteração, o algoritmo estende a solução a inserindo no seu final as provas de um dado caminho de z. Na primeira iteração, o caminho escolhido é aquele que contém a prova s_0. Dado um caminho, a extensão da solução a é realizada inserindo-se as provas deste caminho, uma a uma, na ordem determinada pelo caminho. À medida que as provas são inseridas em a, elas são retiradas de z. O processo de extensão pára quando a inserção da próxima prova implica em $w(a) > n - l$ ou quando todas as provas do caminho foram inseridas. Quando isto ocorre, outro caminho de

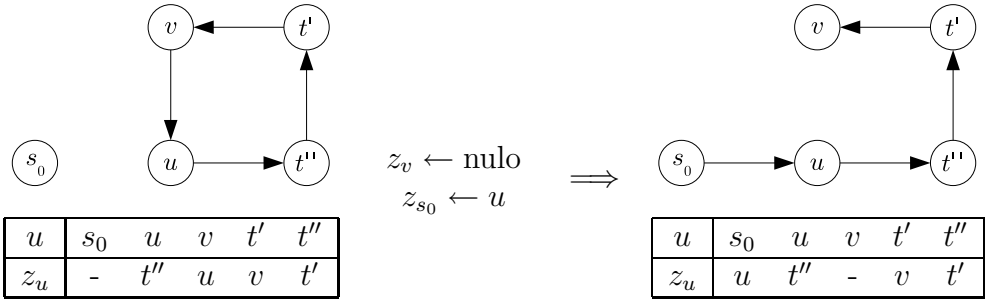
```
algoritmo Viabiliza(S, l, n, s_0, o, w, z)
 1.  se z_{s_0} = nulo então z ← InserePrimeiraProva(S, s_0, o, w, z);
 2.  se z = nulo então retorna nulo;
 3.  a ← ();
 4.  R ← {s_0};
 5.  enquanto R ≠ ∅ faça
 6.      u ← nulo;
 7.      Seleciona uma prova s ∈ R de acordo com as probabilidades
     definidas pela Equação 3-2;
 8.      enquanto s ≠ nulo e (u = nulo ou w(a) + w(u, s) ≤ n − l) faça
 9.          Insere s no final de a;
10.          u ← s; s ← z_s; z_s ← nulo;
11.      fim;
12.      Atualiza a lista restrita de candidatos R;
13.  fim;
14.  retorna a;
fim;
```

Figura 3.14: Algoritmo para gerar uma solução viável a partir de uma frase.

```
algoritmo InserePrimeiraProva(S, s_0, o, w, z)
 1.  u ← argmin_{v∈S|z_v≠nulo} w(s_0, v);
 2.  se o(s_0, u) = 0 então
 3.      retorna nulo;
 4.  senão
 5.      se ∃ v ∈ S tal que z_v = u então z_v ← nulo;
 6.      z_{s_0} = u;
 7.      retorna z;
 8.  fim;
fim;
```

Figura 3.15: Algoritmo que tenta inserir a primeira prova em uma frase.

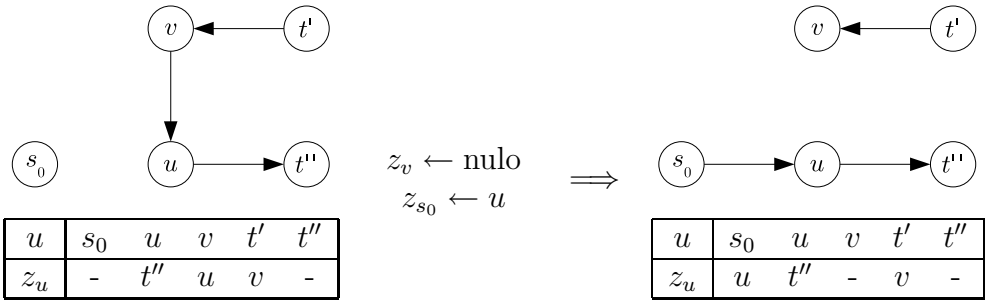
z é escolhido para a próxima iteração do algoritmo. A forma de escolher o próximo caminho é similar àquela utilizada no algoritmo construtivo aleatorizado apresentado na Seção 3.3. Um conjunto R, representando uma lista restrita de candidatos, é construído com as provas de z que estão no início de algum caminho acíclico ou em um caminho cíclico de z e que possuem sobreposição suficiente com a última prova u inserida em a, de modo que a sua inserção em a resulta em uma seqüência com comprimento menor ou igual a n. Uma prova s é selecionada probabilisticamente dentre as provas de R de acordo com a Equação 3-2. Então, o caminho da prova s é utilizado para estender a solução a na próxima iteração do algoritmo. O algoritmo termina quando o conjunto R torna-se vazio. Neste momento, a solução a construída é retornada.



u	s_0	u	v	t'	t''
z_u	-	t''	u	v	t'

u	s_0	u	v	t'	t''
z_u	u	t''	-	v	t'

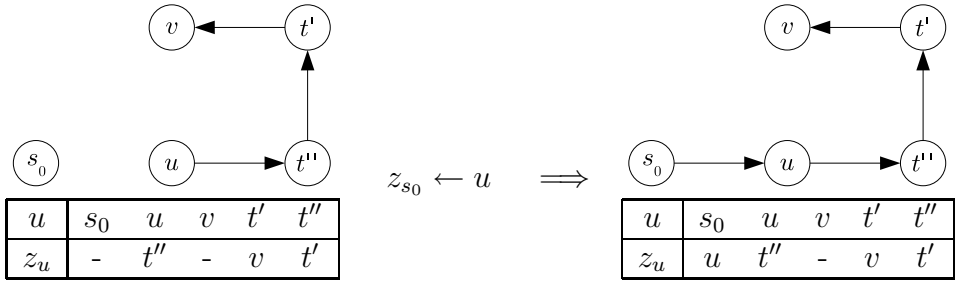
Figura 3.16: Exemplo das operações necessárias para inserir a primeira prova s_0 na frase z, imediatamente antes da prova u. A prova u está em um caminho cíclico da frase z. O resultado das operações $z_{s_0} \leftarrow u$ e $z_v \leftarrow$ nulo é um caminho acíclico a partir da prova s_0.



u	s_0	u	v	t'	t''
z_u	-	t''	u	v	-

u	s_0	u	v	t'	t''
z_u	u	t''	-	v	-

Figura 3.17: Exemplo das operações necessárias para inserir a primeira prova s_0 na frase z, imediatamente antes da prova u. A prova u está em um caminho acíclico da frase z, mas não está no início deste caminho (a prova v possui uma aresta para u). O resultado das operações $z_{s_0} \leftarrow u$ e $z_v \leftarrow$ nulo é um caminho acíclico a partir da prova s_0 e outro caminho formado pela parte anterior à prova u no antigo caminho que o continha.



u	s_0	u	v	t'	t''
z_u	-	t''	-	v	t'

u	s_0	u	v	t'	t''
z_u	u	t''	-	v	t'

Figura 3.18: Exemplo das operações necessárias para inserir a primeira prova s_0 na frase z, imediatamente antes da prova u. A prova u está no início de um caminho acíclico da frase z. O resultado da operação $z_{s_0} \leftarrow u$ é um caminho acíclico a partir da prova s_0.

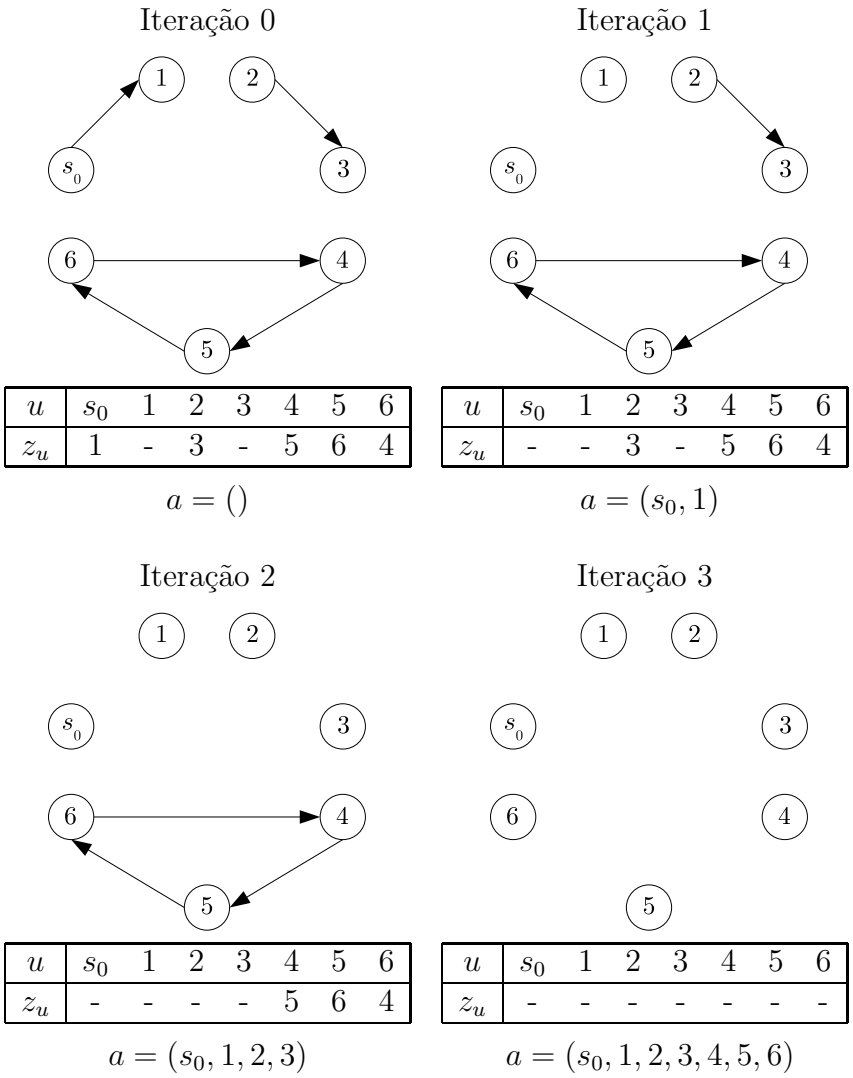


Figura 3.19: Ilustrações do conteúdo da solução a e da frase z a cada iteração do algoritmo `Viabiliza`. A frase z contém três caminhos que são inseridos na solução a, um a um.

Algoritmo completo de construção de vocabulário

O algoritmo `Viabiliza` descrito acima completa a lista dos três algoritmos utilizados na implementação do método de construção de vocabulário. O primeiro passo deste método é encontrar um conjunto de palavras a partir de um dado conjunto de vetores (algoritmo `BuscaPalavras`). Em seguida, as palavras encontradas são combinadas para gerar frases (algoritmo `CombinaPalavras`). Então, para cada frase, tenta-se gerar uma solução viável (algoritmo `Viabiliza`). O algoritmo que implementa o método de construção de vocabulário proposto aqui é denominado `ConstrucaoDeVocabulario` e seu pseudo-código é ilustrado na Figura 3.20.

O método de construção de vocabulário pode ser incorporado à heurística multi-partida para criar uma nova heurística denominada `MP+CV`.

```
algoritmo ConstrucaoDeVocabulario(S, l, n, s_0, o, w, X, s_min)
 1.  Y ← BuscaPalavras(X, s_min);
 2.  Z ← CombinaPalavras(Y);
 3.  A ← ∅;
 4.  para cada z ∈ Z faça
 5.      a ← Viabiliza(S, l, n, s_0, o, w, z);
 6.      se a ≠ nulo então A ← A ∪ {a};
 7.  fim;
 8.  retorna A;
fim;
```

Figura 3.20: Procedimento de construção de vocabulário

Nesta heurística o algoritmo `ConstrucaoDeVocabulario` é chamado algumas vezes entre suas iterações. A entrada do algoritmo é obtida a partir do conjunto de soluções construídas nas iterações anteriores da heurística. Mais especificamente, é mantido um conjunto de soluções de elite Q', com tamanho q' e distância mínima d', que será passado ao algoritmo `ConstrucaoDeVocabulario`. Toda solução construída pelo algoritmo construtivo é considerada a entrar no conjunto Q'. O pseudo-código da heurística `MP+CV` é ilustrado na Figura 3.21. O operador MOD retorna o resto da divisão inteira entre os dois operandos. O parâmetro n_{int} especifica o intervalo de iterações no qual o algoritmo `ConstrucaoDeVocabulario` é executado, ou seja, a cada n_{int} construções o método de construção de vocabulário proposto aqui é aplicado às soluções do conjunto de elite Q'.

```
algoritmo MP+CV(S, l, n, s_0)
 1.  Inicia o, w, n_iter, n_int, α, q', d', Q' e s_min;
 2.  a* ← nulo;
 3.  para i = 1 ... n_iter faça
 4.      a ← ConstrutivoAleatorizado(S, l, n, s_0, o, w, α);
 5.      se a* = nulo ou |a| > |a*| então a* ← a;
 6.      Atualiza Q' com a;
 7.      se i MOD n_int = 0 então
 8.          A ← ConstrucaoDeVocabulario(S, l, n, s_0, o, w, Q', s_min);
 9.          Atualiza Q' com cada solução de A;
10.          Atualiza a* caso exista alguma solução melhor em A;
11.      fim;
12.  fim;
13.  retorna a*;
fim;
```

Figura 3.21: Heurística multi-partida com construção de vocabulário.

O método de construção de vocabulário pode ainda ser combinado

com a heurística multi-partida e a memória adaptativa. Neste trabalho apresenta-se uma heurística para o PSBH baseada nesta idéia. Esta nova heurística é denominada `MP+Mem+CV` e seu pseudo-código é ilustrado na Figura 3.22. Um conjunto de soluções de elite é utilizado tanto pela memória adaptativa quanto pelo método de construção de vocabulário. Entretanto, quando estas duas técnicas são combinadas, é importante utilizar dois conjuntos de elite distintos, pois as características de cada um devem ser diferentes. Por exemplo, o tamanho do conjunto de elite utilizado pela memória adaptativa deve ser bem menor do que o tamanho do conjunto utilizado pelo método de construção de vocabulário. Na construção de vocabulário é interessante obter várias soluções boas para possibilitar a geração de um número razoável de palavras. Já na memória adaptativa, o conjunto deve ser mais restrito para garantir um certo grau de intensificação na região das soluções de elite.

algoritmo `MP+Mem+CV`($S,\ l,\ n,\ s_0$)
1. Inicia o, w, n_{iter}, n_{int}, α, λ, q, d, Q, q', d', Q' e s_{min};
2. $a^* \leftarrow$ **nulo**;
3. **para** $i = 1 \ldots n_{iter}$ **faça**
4. $a \leftarrow$ `ConstrutivoAleatorizadoMem`($S,\ l,\ n,\ s_0,\ o,\ w,\ \alpha,\ Q,\ \lambda$);
5. **se** $a^* =$ **nulo ou** $|a| > |a^*|$ **então** $a^* \leftarrow a$;
6. Atualiza Q e Q' com a;
7. Se necessário, atualiza α e λ;
8. **se** i MOD $n_{int} = 0$ **então**
9. $A \leftarrow$ `ConstrucaoDeVocabulario`($S,\ l,\ n,\ s_0,\ o,\ w,\ Q',\ s_{min}$);
10. Atualiza Q e Q' com cada solução de A;
11. Atualiza a^* caso exista alguma solução melhor em A;
12. **fim**;
13. **fim**;
14. **retorna** a^*;

fim;

Figura 3.22: Heurística multi-partida com memória adaptativa e construção de vocabulário.