3
Um algoritmo GRASP com VND

Problemas de otimização combinatória aparecem frequentemente em vários setores da economia. Como exemplos desses problemas pode-se citar o projeto de redes de telecomunicação, a construção de agendas para tripulações e a produção de rotas eficientes para coleta de lixo, entre outros. Grande parte dos problemas de otimização são intratáveis (não possuem um algoritmo polinomial para sua solução) por natureza ou são grandes o suficiente para tornar inviável o uso de algoritmos exatos. Nesses casos, métodos heurísticos são usualmente empregados. Esses métodos frequentemente obtêm a solução ótima para um problema, porém nenhuma garantia de convergência é fornecida na maior parte dos casos.

A metaheurística GRASP tem sido usada para obter soluções de qualidade para muitos problemas de otimização combinatória [22]. Possui várias vantagens sobre outras metaheurísticas. Sua implementação é simples, usando-se algoritmos para construção de soluções e de busca local desenvolvidos para serem usados em outras abordagens. Além disso, o ajuste de parâmetros é mais simples em um procedimento GRASP do que em outras metaheurísticas.

As heurísticas GRASP e VND serão discutidas com maiores detalhes na seções 3.1 e 3.2, respectivamente. Nas seções seguintes serão apresentadas as etapas que compõem o algoritmo GRASP híbrido proposto. Por fim, serão mostrados os resultados computacionais.

3.1
Heurística GRASP

Um problema de otimização combinatória pode ser definido por um conjunto base $E = \{1, ..., n\}$, um conjunto de soluções viáveis $X \subseteq 2^{|E|}$ e uma função objetivo $f : 2^{|E|} \rightarrow I\!R$ tal que $f(x) = \Sigma_{e \in x} c(e)$, $\forall x \in 2^{|E|}$, onde $c(e)$ é o custo associado à inclusão do elemento $e \in E$ na solução x. A solução ótima $x^* \in X$ de um problema de minimização é tal que

$f(x^*) \leq f(x)$, $\forall x \in X$. O conjunto E, os custos c e o conjunto de soluções viáveis X são específicos de cada problema.

GRASP (*Greedy Randomized Adaptive Search Procedure*) [20, 21, 22, 60] é uma metaheurística de partidas múltiplas usada para obter soluções para problemas de otimização combinatória. Cada iteração é formada por duas etapas: uma fase construtiva, encarregada de produzir uma solução, e uma fase de busca local, que explora a vizinhança da solução construída, retornando um mínimo local. A melhor solução produzida durante as iterações será o resultado obtido pelo algoritmo.

Na fase construtiva, uma solução viável é construída iterativamente, inserindo-se na solução parcial um elemento de cada vez. A cada iteração da fase construtiva, são avaliados apenas elementos que podem ser adicionados à solução sem violar as restrições de viabilidade. Esses elementos são chamados de elementos candidatos. A escolha do próximo elemento a ser adicionado à solução é determinada ordenando-se todos os elementos candidatos em uma lista de candidatos C, de acordo com uma função gulosa. Essa função mede o benefício associado à seleção de cada elemento. A heurística é adaptativa porque os benefícios associados a cada elemento são atualizados a cada iteração da fase construtiva, para incorporar as mudanças causadas pela escolha do último elemento. A componente probabilística é caracterizada pela escolha aleatória de um dos melhores candidatos da lista C, que não é necessariamente o melhor. A lista de melhores candidatos é denominada de lista restrita de candidatos (LRC). A fase construtiva é semelhante a heurística proposta independentemente por Hart e Shogan [42], que consiste em uma estratégia de partidas múltiplas baseada em construções gulosas randomizadas, porém sem o uso de busca local. A Figura 3.1 apresenta o pseudo-código do procedimento de construção padrão. Na linha 1 a solução x a ser construída é inicializada. O conjunto de candidatos C a serem inseridos na solução é inicializado na linha 2. O laço nas linhas 3-10 garante que a solução x estará em construção enquanto houver candidatos a serem inseridos. Na linha 5 a lista restrita de candidatos é definida. Nas linhas 6 e 7 um elemento de LRC é selecionado e inserido em x na linha 8. Na linha 9 o conjunto de candidatos C é atualizado. A suloção x é retornada na linha 11.

Na maior parte dos casos, é possível melhorar a solução construída, aplicando-se a ela uma busca local. Um procedimento de busca local parte sempre de uma solução inicial $x^0 \in X$ e gera uma seqüência de soluções $x^1, x^2, ..., x^k$. Dada uma solução $x \in X$, os elementos da sua vizinhança $N(x)$ são as soluções que podem ser obtidas através da aplicação de uma

Procedimento `Construtiva`(*semente*)
Entrada
semente - semente para geração de números aleatórios.
Saída
x - solução construída.
Início
01. $x \leftarrow \emptyset$;
02. Inicialize o conjunto de candidatos C;
03. **Enquanto** $C \neq \emptyset$ **faça**
04. **Início**
05. Construa LRC;
06. $i \leftarrow$ `rand`(*semente*);
07. Selecione e_i de LRC;
08. $x \leftarrow x \cup e_i$;
09. Atualize o conjunto de candidatos C;
10. **Fim_enquanto**
11. **Retorne** x;
Fim-`Construtiva`

Figura 3.1: Algoritmo de construção padrão.

modificação elementar em x, chamada de *movimento*.

Sem perda de generalidade, em um problema de minimização, a vizinhança $N(x^k)$ de x^k é analisada durante a k-ésima iteração e procura-se encontrar uma solução $x^{k+1} \in N(x^k)$, tal que $f(x^{k+1}) < f(x^k)$. Quando uma solução que melhora a solução corrente é encontrada, ela passa a ser a nova solução corrente e a sua vizinhança é analisada. Em caso contrário, a busca termina e a solução corrente é um ótimo local. A eficiência de um procedimento de busca local depende de vários fatores, tais como a estrutura da vizinhança, a função que deve ser otimizada e a solução inicial. Na Figura 3.2 um procedimento de busca local é mostrado em pseudo-código. Na linha 1 a variável x que armazena a melhor solução encontrada até o momento é inicializada com a solução inicial x^0. O laço nas linhas 2-6 faz com que a busca prossiga enquanto houver algum vizinho de x melhor avaliado do que x. Se um vizinho x' melhor do que x é encontrado, então a solução x' é atribuída a x, como mostrado na linha 5.

A idéia básica da metaheurística GRASP consiste em usar diferentes soluções iniciais como pontos de partida para a busca local. Uma solução x é dita como pertencente à bacia de atração de um ótimo local quando, a partir de uma busca local iniciada em x, é possível atingir este ótimo local. Caso uma das soluções iniciais esteja na bacia de atração de um ótimo global, a busca local irá encontrar este ótimo global. Caso contrário, a solução do algoritmo será um ótimo local.

Uma solução na bacia de atração de um ótimo global será eventualmente produzida, caso um número grande de soluções geradas aleatoria-

```
Procedimento BuscaLocal(x^0)
Entrada
    x^0 - solução inicial;
Saída
    x - o mínimo local encontrado;
Início
01.    x ← x^0;
02.    Enquanto existir x' ∈ N(x) tal que f(x') < f(x) faça
03.    Início
04.        Selecione x' ∈ N(x) tal que f(x') < f(x);
05.        x ← x';
06.    Fim_enquanto
07.    Retorne x;
Fim-BuscaLocal
```

Figura 3.2: Algoritmo de busca local padrão.

mente seja usado para iniciar a busca local. Porém, soluções produzidas aleatoriamente são, em geral, de baixa qualidade. Além disso, o número de movimentos necessários para que soluções geradas aleatoriamente (e que estejam na bacia de atração de um ótimo global) atinjam o ótimo possivelmente será muito elevado [45]. Por outro lado, algoritmos gulosos geralmente produzem boas soluções, mas depois da fase de busca local as soluções encontradas são geralmente apenas ótimos locais e não globais. Isso acontece porque a diversidade das soluções produzidas por um algoritmo guloso é muito pequena. Caso não existam elementos com valores idênticos produzidos pela função gulosa, ou caso uma regra determinística seja usada para selecionar entre esses elementos, o algoritmo guloso produzirá sempre a mesma solução. Para garantir diversidade de soluções e, ao mesmo tempo, um controle na qualidade das soluções produzidas, a metaheurística GRASP usa um algoritmo semi-guloso [20, 42] para produzir as soluções iniciais usadas em cada iteração. Na Figura 3.3 é mostrado um algoritmo GRASP básico para problemas de minimização. O laço nas linhas 2-11 garante que o processo será repetido por *maxIterações* iterações. Na linha 4 uma solução x é construída. A solução x é submetida a uma busca local na linha 5 gerando a solução x'. Verifica-se se a solução x' encontrada é melhor do que a melhor solução encontrada até o momento, x^*. Em caso afirmativo, na linha 8 a solução x' é atribuída a x^*.

3.2 Variable Neighborhood Descent (VND)

Variable Neighborhood Descent (VND) é uma estratégia de refina-

```
Procedimento GRASP(maxIterações, semente)
Entrada
    maxIterações - número de iterações a serem realizadas;
    semente - semente utilizada na geração de números aleatórios;
Saída
    x* - a melhor solução encontrada;
Início
01.   f* ← ∞;
02.   Para iter = 0, ..., maxIterações faça
03.   Início
04.       x ← Construtiva(semente);
05.       x' ← BuscaLocal(x);
06.       Se f(x') < f* então
07.       Início
08.           x* ← x';
09.           f* ← f(x*);
10.       Fim_se;
11.   Fim_para
12.   Retorne x*;
Fim-GRASP
```

Figura 3.3: Algoritmo GRASP.

mento de soluções proposta por Hansen e Mladenović [39, 40, 41, 53]. O que difere esta estratégia do método de busca local tradicional é que ao invés de utilizar uma única estrutura de vizinhança, várias estruturas de vizinhança são utilizadas. Estas vizinhanças estendidas procuram por soluções aprimorantes que estão "mais distantes"da solução atual, assim permitindo ao método escapar de ótimos locais com respeito a uma vizinhança menor.

Sejam $N_1, N_2, ..., N_p$ p estruturas de vizinhança tal que $N_k(x)$ é o conjunto de soluções na k-ésima vizinhança da solução x. Normalmente é assumido que a vizinhança N_{k+1} possui mais elementos do que a vizinhança N_k. A Figura 3.4 apresenta o pseudo-código do algoritmo VND padrão. O laço nas linhas 3-13 é repetido até que todas as vizinhanças sejam analisadas sem sucesso. Na linha 5 uma busca local na vizinhança N_k é aplicada a x, gerando a solução x'. Se x' for melhor do que x então a solução x' é atribuída a x e o processo é reiniciado com a vizinhança N_1, como é mostrado nas linhas 8 e 9. Caso contrário, o processo continua com a vizinhança N_{k+1}.

3.3 Método construtivo guloso aleatorizado

Seja E o conjunto de todos os possíveis taxons e $W = \{t^{(1)}, ..., t^{(n)}\} \subseteq E$, o conjunto dos taxons operacionais sob análise. Uma árvore filogenética s para o conjunto de taxons operacionais W pertence ao conjunto S de todas

Procedimento VND($x_0, N_1, ..., N_p$)
Entrada
x_0 - uma solução viável para o problema;
$N_1, N_2, ..., N_p$ - as vizinhanças;
Saída
x - Uma solução refinada;
Início
01. $x \leftarrow x_0$;
02. $k \leftarrow 1$;
03. **Enquanto** $k \leq p$ **faça**
04. **Início**
05. Aplique uma busca local em x usando a vizinhança N_k. Seja x' o ótimo local;
06. **Se** $f(x') < f(x)$ **então**
07. **Início**
08. $x \leftarrow x'$;
09. $k \leftarrow 1$;
10. **Senão**
11. $k \leftarrow k + 1$;
12. **Fim-se**
13. **Fim-enquanto**
14. **Retorne** x;
Fim-VND

Figura 3.4: Algoritmo VND.

as árvores não enraizadas com n folhas (cada folha tem um relacionamento um-para-um com um taxon operacional $t^{(i)} \in W$, i=1,...,n) e todos os nós internos com grau três. Seja f:$S \rightarrow I\!R$ uma função que associa cada filogenia $s \in S$ com seu valor de parcimônia.

Uma árvore filogenética $s \in S$ relacionando os taxons descritos em W pode ser construída em n =|W| iterações, como é mostrado no pseudo-código da Figura 3.5. Na linha 1 a filogenia s é inicializada. O conjunto U que representa o conjunto dos taxons já inseridos em s é inicializado na linha 2. O laço nas linhas 4-10 garante que todos os taxons serão inseridos na filogenia s. Na linha 6-9 um novo taxon é escolhido aleatoriamente para a inserção e as variáveis U e s são atualizadas. Na linha 11 a filogenia s construída é retornada.

As variantes deste algoritmo se diferem pelo critério que usam para definir o próximo taxon a ser inserido e pela maneira pela qual $s^{(k)}$ é modificada para se obter $s^{(k+1)}$. O conjunto de alterações aplicadas a $s^{(k)}$ para se alcançar a configuração $s^{(k+1)}$ é denominado *incremento*. O acréscimo do valor de parcimônia devido ao incremento realizado em $s^{(k)}$ para transformá-la em $s^{(k+1)}$ pode ser computado em tempo $O(mk)$, onde m é o número de características binárias e k o número de taxons já inseridos na filogenia atual. Os incrementos são definidos pela inserção de um novo taxon em uma aresta da solução parcial atual, como ilustrado na Figura 3.6.

```
Procedimento ConstruirFilogenia(semente)
Entrada
    semente - semente para geração de números aleatórios.
Saída
    s - filogenia construída.
Início
01.   s ← 0;
02.   U ← ∅;
03.   k ← 1;
04.   Enquanto k ≤ n faça
05.   Início
06.       Selecione, aleatoriamente, um taxon t ∈ W\U;
07.       U ← U ∪ t;
08.       Insira o taxon t na filogenia s;
09.       k ← k + 1;
10.   Fim-enquanto
11.   Retorne s;
Fim-ConstruirFilogenia
```

Figura 3.5: Algoritmo básico para construção de uma filogenia.

Neste caso, existem três alternativas possíveis para a inserção do taxon D em uma filogenia formada pelos taxons A, B e C.

Andreatta e Ribeiro [3, 4] realizaram um estudo detalhado de algoritmos de construção para o problema da filogenia, no qual foram testadas e comparadas diversas variantes do algoritmo construtivo básico descrito na Figura 3.5. Usou-se neste trabalho o algoritmo `GStep_wR` (*greedy step with randomness*) na fase de construção da heurística GRASP implementada, uma vez que é citado em [3, 4] que geralmente este encontra as melhores soluções (embora exija um gasto de tempo superior ao dos outros algoritmos). Testa-se a inserção de cada taxon t em cada aresta da filogenia parcial s, onde $t \notin s$. O par taxon-aresta é aleatoriamente selecionado entre um subgrupo que obteve os menores custos de incremento. Na itereção k existem ainda n - (k - 1) taxons não pertencentes a s e $2k$ - 5 arestas para possível inserção. Assim, a complexidade de cada construção usando o algoritmo `GStep_wR` é dado por

$$\sum_{k=3}^{n} O(mk)(2k-5)[n-(k-1)] \;=\; O(mn^4). \tag{3-1}$$

3.4 Estruturas de vizinhança

Os métodos de busca local são baseados na investigação do espaço de soluções, explorando sucessivamente a vizinhança da solução atual e,

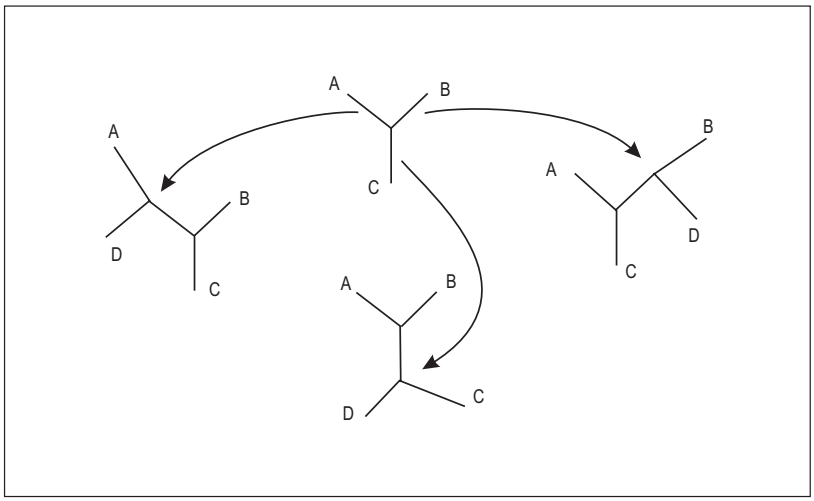

Figura 3.6: Alternativas para a inserção de um novo taxon em uma filogenia com três taxons.

criteriosamente, movendo-se para uma de suas vizinhas. O algoritmo de busca local básico é apresentado na Figura 3.2.

Andreatta e Ribeiro [3, 4] relataram e descreveram três relações de vizinhança para o problema de filgenia: *Nearest Neighborhood Interchanges* (NNI), *Single Step* (STEP), e *Subtree Pruning and Regrafting* (SPR ou 1-SPR). Nesta última, uma subárvore da árvore filogenética atual é desconectada e reconectada em uma posição diferente.

A Figura 3.7 ilustra um exemplo de um movimento realizado na vizinhança SPR. As soluções vizinhas são obtidas da solução atual da seguinte maneira:

- Passo 1: Uma aresta $q = (c, f)$ da árvore filogenética atual é selecionada e eliminada. A subárvore contendo o nó c é chamada de subárvore base, enquanto a que contém o nó f é a subárvore pendente.
- Passo 2: O nó c é destruído (uma vez que este tem grau dois na subárvore base) e uma aresta r da subárvore base é selecionada para a reconexão da subárvore pendente.
- Passo 3: A aresta r é eliminada e um novo nó h é criado e conectado às duas extremidades de r, originalmente na subárvore base.
- Passo 4: A subárvore pendente é conectada na subárvore base usando o nó h, recentemente criado.

Como uma determinada filogenia possui $O(n)$ subárvores e cada uma delas pode ser desconectada e depois reinserida em $O(n)$ possíveis arestas, então qualquer solução tem $O(n^2)$ vizinhas dentro desta vizinhança.

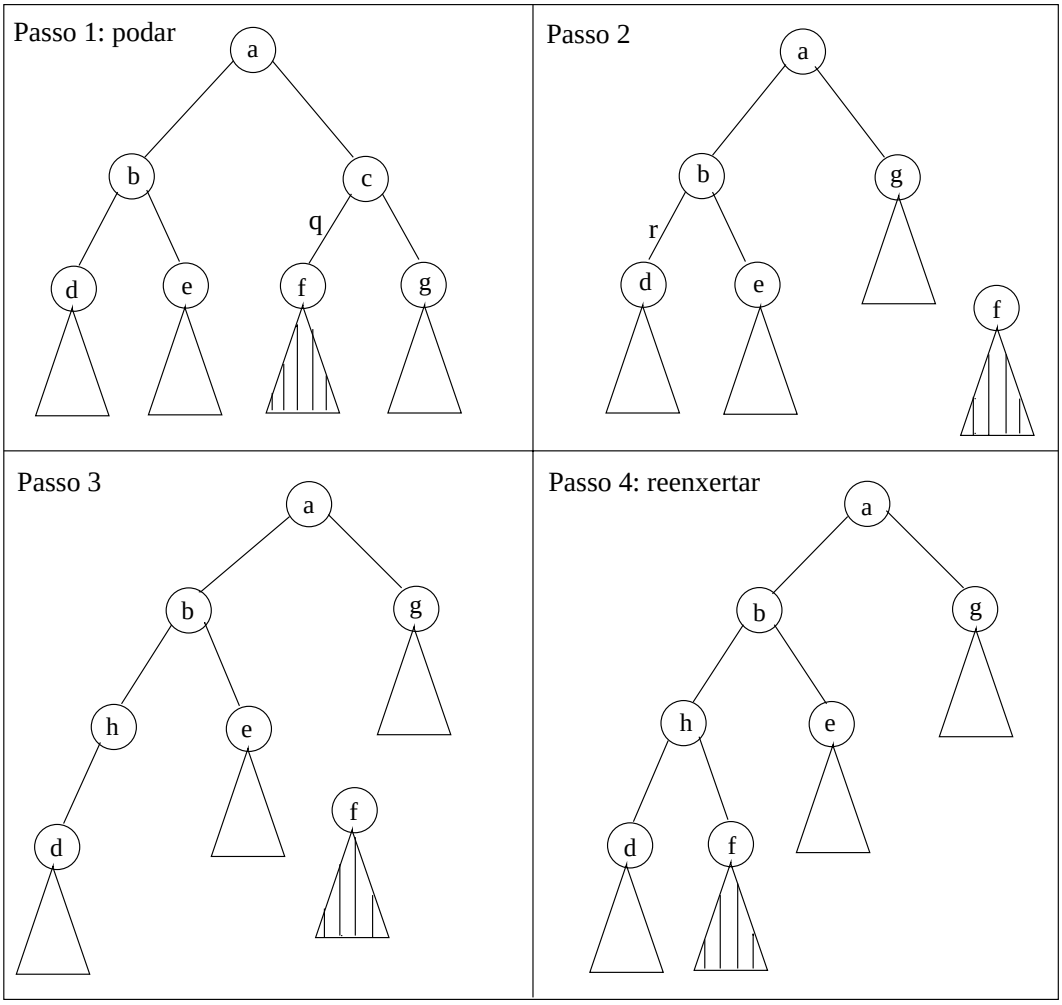

Figura 3.7: Ilustração de um movimento dentro da vizinhança SPR.

A Figura 3.8 apresenta o pseudo-código do procedimento `BL`[1] aplicado na busca por um melhor vizinho da solução atual s dentro da vizinhança SPR. O laço nas linhas 1-20 realiza a busca investigando a eliminação de cada aresta da solução atual. Cada aresta q é temporariamente eliminada de s na linha 3, criando as subárvores s_1 (base) e s_2 (pendente). Os custos parciais das subárvores s_1 e s_2 são calculados na linha 4 em tempo $O(mn)$ pelo algoritmo `CalcularCCBin` mostrado na Subseção 2.4.2. A reconexão da árvore pendente usando cada aresta da subárvore base é investigada nas linhas 5-14. A variável f' inicializada na linha 5 armazena o custo mínimo incremental dentre todos os vizinhos de s obtidos pela eliminação da aresta q. O laço nas linhas 6-14 realiza a busca sobre todas as arestas r da subárvore base. O custo incremental $\bar{f}$ de reconectar a subárvore pendente s_2 na subárvore base s_1 usando a aresta r é calculado na linha 8 em tempo $O(m)$ pelo algoritmo `CalcValorInsercaoBin` mostrado na Subseção 2.4.2. Se o custo incremental $\bar{f}$ melhora o melhor custo incremental corrente,

```
Procedimento BL¹(s)
Entrada
    s - a filogenia de entrada;
Saída
    s - a filogenia de entrada após o refinamento;
Início
01.   Para cada aresta q da solução atual s faça
02.   Início
03.       Remova q de s obtendo as subárvores s1 e s2;
04.       Calcule f(s1) e f(s2);
05.       f' ← ∞;
06.       Para cada aresta r da subárvore s1 faça
07.       Início
08.           Calcule o custo incremental f̄ de reconectar s2 em s1 usando r;
09.           Se f̄ < f' então
10.           Início
11.               f' ← f̄;
12.               melhor_r ← r;
13.           Fim-se
14.       Fim-para
15.       Obtenha s' reconectando s2 em s1 usando a aresta melhor_r;
16.       Calcule o custo f(s') = f(s1) + f(s2) + f' da nova solução s';
17.       Se f(s') < f(s) então
18.           retorne s';
19.       Restaure a solução s;
20.   Fim-para
21.   Retorne s;
Fim-BL¹
```

Figura 3.8: Pseudo-código do procedimento de busca $\texttt{BL}^1$ usando a vizinhança SPR.

então este último é atualizado nas linhas 9-13 e a melhor aresta a ser usada na reconexão é armazenada em $melhor\_r$. O melhor vizinho s' da solução atual s é contruído na linha 15 e seu custo $f(s')$ é calculado na linha 16. Se $f(s')$ é menor do que o custo da solução atual s, então s' é retornada na linha 18. Caso contrário, a aresta q é reinserida e a solução atual s é restaurada na linha 19. Se nenhuma melhora for encontrada, a própria solução atual s é retornada na linha 21 no final da busca em vizinhança. Como existem $O(n)$ possíveis arestas onde ocorrer a desconexão da subárvore pendente, para cada desconexão existem $O(n)$ possíveis arestas para reinserção e cada inserção é verificada em tempo $O(m)$, então a investigação de cada vizinhança SPR pode ser implementada em tempo $O(mn^2)$. Esta nova implementação é uma ordem de magnitude mais rápida do que a originalmente proposta em [3, 4].

A vizinhança *Multiple Subtree Prunning and Regrafting* (k-SPR), proposta neste trabalho, é definida pela composição de k sucessivos movimentos SPR aplicados à solução atual. O caso $k = 1$ corresponde à vizinhança SPR

```
Procedimento BL²(s)
Entrada
    s - a filogenia de entrada;
Saída
    s - a filogenia de entrada após o refinamento;
Início
01.   Para cada aresta q da solução atual s faça
02.   Início
03.     Remova q de s obtendo as subárvores s1 e s2;
04.     Calcule f(s1) e f(s2);
05.     f' ← ∞;
06.     Para cada aresta r da subárvore s1 faça
07.     Início
08.       Calcule o custo incremental f̄ de reconectar s2 em s1 usando r;
09.       Se f̄ < f' então
10.       Início
11.         f' ← f̄;
12.         melhor_r ← r;
13.       Fim-se
14.     Fim-para
15.     Obtenha s' reconectando s2 em s1 usando a aresta melhor_r;
16.     Calcule o custo f(s') = f(s1) + f(s2) + f' da nova solução s';
17.     Se f(s') < f(s) então
18.       retorne s';
19.     Para cada aresta q' da solução intemediária s' faça
20.     Início
21.       Remova q' de s' obtendo as subárvores s'1 e s'2;
22.       Calcule f(s'1) e f(s'2);
23.       f' ← ∞;
24.       Para cada aresta r' da subárvore s'1 faça
25.       Início
26.         Calcule o custo incremental f̄ de reconectar s'2 em s'1 usando r';
27.         Se f̄ < f' então
28.         Início
29.           f' ← f̄;
30.           melhor_r ← r';
31.         Fim-se
32.       Fim-para
33.       Obtenha s'' reconectando s'2 em s'1 usando a aresta melhor_r;
34.       Calcule o custo f(s'') = f(s'1) + f(s'2) + f' da nova solução s'';
35.       Se f(s'') < f(s) então
36.         retorne s'';
37.       Restaure a solução s';
38.     Fim-para
39.     Restaure a solução s;
40.   Fim-para
41.   Retorne s;
Fim-BL²
```

Figura 3.9: Pseudo-código do procedimento de busca $\texttt{BL}^2$ usando a vizinhança 2-SPR.

```
Procedimento VNDFilogenia(s)
Entrada
    s - a filogenia de entrada;
Saída
    s - a filogenia de entrada após o refinamento;
Início
01.   k ← 1;
02.   Enquanto k ≤ 2 faça
03.   Início
04.       s_vizinha ← BL^k(s);
05.       Se f(s_vizinha) < f(s) então
06.       Início
07.           s ← s_vizinha;
08.           k ← 1;
09.       Senão
10.           k ← k + 1;
11.       Fim-se
12.   Fim-enquanto
13.   Retorne s;
Fim-VNDFilogenia
```

Figura 3.10: Procedimento VND para busca local.

descrita anteriormente. A Figura 3.9 apresenta o pseudo-código do procedimento $\texttt{BL}^2$ usado na busca por um vizinho aprimorante da solução atual s na vizinhança 2-SPR, na qual cada solução vizinha é obtida através de um movimento envolvendo dois passos. As linhas 1-18 correspondem ao primeiro passo, no qual soluções intermediárias dentro da vizinhança SPR são investigadas. As linhas 19-36 correspondem ao segundo passo, no qual um movimento SPR adicional é aplicado à solução intermediária. Similarmente ao procedimento $\texttt{BL}^1$, a investigação de cada vizinhança 2-SPR pode ser implementada em tempo $O(mn^3)$.

3.5 Busca local usando VND

Seja k-SPR, $k = 1, ..., k_{max}$, uma estrutura de vizinhanças para o problema da filogenia. Usou-se neste trabalho um procedimento de busca local baseado em vizinhanças variáveis com $k_{max} = 2$, o qual essencialmente é uma variante da estratégia *Variable Neighborhood Descent* (VND) proposta por Hansen e Mladenović [39, 40, 41, 53]. Algumas aplicações que utilizaram VND dentro de GRASP com sucesso estão relatadas em [22, 49, 63, 64].

A Figura 3.10 apresenta a descrição algorítmica do procedimento `VNDFilogenia`, o qual implementa a busca local VND iniciando pela solução s construída na fase de construção. A vizinhança inicial 1-SPR é definida

na linha 1. O laço nas linhas 2-12 investiga uma vizinhança por vez, até um ótimo local com respeito as vizinhanças 1-SPR e 2-SPR ser encontrado. A primeira solução aprimorante $s_{vizinha}$ dentro da vizinhança k-SPR é obtida pela aplicação do procedimento $\texttt{BL}^k$ à solução atual s na linha 4. No caso de um movimento aprimorante ser encontrado, a solução atual é atualizada na linha 7 e a busca recomeça pela vizinhança 1-SPR. Caso contrário, a ordem da vizinhança é aumentada em um na linha 10. Com isto, a busca é reiniciada pela vizinhança (k+1)-SPR. A solução s que representa o ótimo local atingido com respeito a todas as vizinhanças é retornada na linha 13.

3.6 Heurística GRASP com VND

```
Procedimento GRASP+VND(maxIterações, semente)
Entrada
    maxIterações - número de iterações a serem executadas;
    semente - semente para geração dos números aleatórios;
Saída
    s* - melhor solução encontrada;
Início
01.     f* ← ∞;
02.     Para k=1, ..., maxIterações faça
03.     Início
04.         s ← GStep_wR(semente);
05.         s' ← VNDFilogenia(s);
06.         Se f(s') < f* então
07.         Início
08.             f* ← f(s');
09.             s* ← s';
10.         Fim-se
11.     Fim-Para
12.     Retorne s*;
Fim-GRASP+VND
```

Figura 3.11: Pseudo-código da heurística GRASP+VND.

O pseudo-código na Figura 3.11 baseado no modelo descrito por Resende e Ribeiro [60] ilustra o bloco principal de uma heurística GRASP para o problema da filogenia. O algoritmo recebe como parâmentros o número máximo de iterações, *maxIterações*, e o valor *semente* usado como a semente inicial para um gerador de números pseudo-aleatório. O laço nas linhas 2-11 realiza *maxIterações* iterações. O algoritmo `GStep_wR` [3, 4] descrito na Seção 3.3 é usado na fase de construção na linha 4. A estratégia de busca local VND usando $k_{max} = 2$ e vizinhanças 1-SPR e 2-SPR é

implementada na linha 5, como descrito na Seção 3.5. A melhor solução encontrada é atualizada na linha 9 a cada iteração e retornada na linha 12.

3.7 Resultados computacionais

Todos os experimentos computacionais foram realizados em um computador com processador Pentium IV com 2GHz de freqüência e com memória principal de 512MBytes. A heurística GRASP+VND foi implementada em C usando a versão 6.0 do compilador Microsoft Visual C++.

Usou-se uma implementação em C do gerador de números aleatórios descrito em [67].

Para a realização dos testes foi necessária a construção de 20 instâncias aleatóriamente. Isso ocorreu devido à pouca quantidade de instâncias testes existentes na literatura que apresentavam características binárias.

Na primeira parte dos experimentos computacionais, foram usadas as 20 instâncias geradas aleatóriamente. O gerador proposto neste trabalho recebe como parâmetros o número de taxons, o número de características e a razão de indeterminação, a qual corresponde à fração de características indefinidas em cada taxon. As instâncias com as maiores razões de indeterminação tendem a ser mais difíceis. O número de taxons nestas instâncias varia de 45 a 75, o número de características de 61 a 159 e a razão de indefinição de 20 a 50%. Na Tabela 3.1, para cada instância são relatados sua identificação, o número de taxons (n), o número de características (m) e a razão de indeterminação nas características de cada taxon. Tanto para o algoritmo GRASP descrito em [3, 4] quanto para a heurística GRASP+VND proposta neste trabalho (com o número de iterações GRASP fixado em $maxIterações = 50$ para ambos algoritmos) foram relatados na Tabela 3.2 os tempos computacionais e as melhores soluções obtidas. A nova heurística encontrou a melhor solução para todas as instâncias, exceto para a instância TST06. A média de melhoramento no valor da solução é de aproximadamente 1%. Além disso, os tempos computacionais observados com a nova heurística são significativamente menores para todas as instâncias testes.

Na segunda parte dos experimentos computacionais, foram usados as mesmas oito instâncias [37, 48, 57, 58] testadas em [3, 4] e já apresentadas na Tabela 2.1. Cada instância foi executada dez vezes com diferentes sementes. Na Tabela 3.3 são apresentadas, para cada instância, as melhores soluções encontradas pela heurística `GRASP+VND` após as dez execuções. O número de iterações GRASP foi $maxIterações = 500$. Estes resultados mostraram

Instância	n	m	indefinição (%)
TST01	45	61	20
TST02	47	151	30
TST03	49	111	40
TST04	50	97	50
TST05	52	75	20
TST06	54	65	30
TST07	56	143	40
TST08	57	119	50
TST09	59	93	20
TST10	60	71	30
TST11	62	63	40
TST12	64	147	50
TST13	65	113	20
TST14	67	99	30
TST15	69	77	40
TST16	70	69	50
TST17	71	159	20
TST18	73	117	30
TST19	74	95	40
TST20	75	79	50

Tabela 3.1: Características dos problemas gerados aleatoriamente.

que a heurística `GRASP+VND` é bastante robusta. Ela não só melhorou as melhores soluções conhecidas para três das oito instâncias testes (ROPA, GOLO, SCHU), mas também alcançou as melhores soluções conhecidas para todas as outras instâncias. Deve-se destacar que as melhores soluções previamente relatadas na literatura para cada problema teste não foram sempre encontradas pelo mesmo algoritmo [3, 4].

Em outro experimento realizado, o mesmo tempo computacional foi dado para cada algoritmo. Dez execuções com 1000 segundos cada foram realizadas para cada instância. Os resultados computacionais com as oito instâncias testes da literatura estão descritos na Tabela 3.4, enquanto os realizados com as vinte instâncias testes geradas aleatoriamente são descritos na Tabela 3.5. Para cada instância, são dados o valor médio e o melhor valor dentre as soluções obtidas durante as dez execuções de cada algoritmo. A nova heurística obteve soluções médias estritamente melhores para sete das oito instâncias da literatura e para todas as instâncias geradas aleatoriamente. O algoritmo `GRASP+VND` também encontrou soluções estritamente melhores para sete das oito instâncias da literatura e para todas as instâncias geradas aleatoriamente.

Outra comparação realizada entre a nova heurística `GRASP+VND` com o

	GRASP[3, 4]		GRASP+VND	
Instância	tempo (s)	valor	tempo (s)	valor
TST01	530.30	558	526.86	551
TST02	1560.00	1377	663.47	1364
TST03	1731.44	851	687.69	845
TST04	1614.34	605	779.39	598
TST05	1129.05	807	736.72	797
TST06	1357.53	608	960.48	609
TST07	3746.81	1304	1040.93	1291
TST08	3082.61	881	1164.59	870
TST09	2700.21	1167	1344.66	1152
TST10	2550.61	734	1454.67	733
TST11	2649.65	557	1520.33	553
TST12	6715.86	1250	2055.82	1243
TST13	4975.89	1545	2499.95	1532
TST14	5528.49	1186	2820.38	1177
TST15	4946.88	782	2974.00	774
TST16	4885.68	556	3309.26	551
TST17	8002.02	2481	3886.32	2468
TST18	8125.33	1568	3774.29	1554
TST19	7331.50	1042	3558.12	1036
TST20	6884.55	694	3884.26	682

Tabela 3.2: Resultados comparativos sobre problemas gerados aleatoriamente.

Instância	melhor	GRASP+VND
ANGI	216	216
GRIS	172	172
TENU	682	682
ETHE	372	372
ROPA	326	(*) 325
GOLO	497	(*) 496
SCHU	760	(*) 759
CARP	548	548

Tabela 3.3: Resultados obtidos pela heurística GRASP+VND.

	Média das soluções obtidas (após dez execuções)		Melhor solução obtida (após dez execuções)	
Instância	GRASP [3, 4]	GRASP+VND	GRASP [3, 4]	GRASP+VND
GRIS	172.0	172.0	172	172
ANGI	217.7	216.0	217	216
TENU	683.2	682.0	683	682
ETHE	374.3	372.8	374	372
ROPA	329.0	326.0	329	325
GOLO	508.1	498.2	504	497
SCHU	827.0	761.0	814	759
CARP	568.2	552.4	559	550

Tabela 3.4: Resultados comparativos para dez execuções do algoritmo GRASP em [3, 4] e a heurística GRASP+VND para as oito instâncias testes da literatura.

	Média das soluções obtidas (após dez execuções)		Melhor solução obtida (após dez execuções)	
Instância	GRASP [3, 4]	GRASP+VND	GRASP [3, 4]	GRASP+VND
TST01	554.3	550.8	553	549
TST02	1376.0	1367.2	1375	1361
TST03	852.3	843.8	849	843
TST04	606.7	599.2	606	598
TST05	805.0	796.8	803	793
TST06	614.4	606.6	613	605
TST07	1306.1	1288.6	1303	1283
TST08	884.8	873.0	875	868
TST09	1166.3	1159.2	1165	1156
TST10	739.4	732.0	738	730
TST11	558.6	554.8	555	554
TST12	1253.5	1242.0	1244	1237
TST13	1548.3	1532.4	1544	1529
TST14	1194.0	1182.0	1191	1180
TST15	785.0	774.8	784	769
TST16	561.7	550.6	560	547
TST17	2505.3	2479.0	2501	2475
TST18	1572.1	1554.6	1571	1548
TST19	1052.2	1041.0	1048	1035
TST20	697.9	685.4	695	682

Tabela 3.5: Resultados comparativos para dez execuções do algoritmo GRASP em [3, 4] e a heurística GRASP+VND para as instâncias testes geradas aleatóriamente.

algoritmo GRASP original em [3, 4] foi verificar o comportamento de ambos algoritmos nas instâncias ROPA e GOLO usando a metodologia proposta por Aiex et al. [2] e recentemente revisado por Resende e Ribeiro [60]. Cem independentes execuções de cada heurística para cada instância foram feitas. Cada execução terminava quando uma solução de valor menor ou igual a um certo valor alvo era encontrada. Para os valores alvos foram atribuídos os valores das melhores soluções previamente conhecidas para cada uma das duas instâncias, ou seja, 326 para a instância ROPA e 497 para a instância GOLO. Embora cada um destes valores tenha sido escolhido de tal forma que a heurística mais rápida pudesse terminar depois de um tempo computacional considerável, o comportamento relativo das duas heurísticas não é afetado por esta escolha. Distribuições empíricas de probabilidades para a variável tempo-para-valor-alvo são traçadas nas Figuras 3.12 e 3.13. Para traçar a distribuição empírica para cada algoritmo, seguiu-se o procedimento descrito em [2]. Associou-se com o i-ésimo menor tempo de execução t_i uma probabilidade $p_i = (i - \frac{1}{2})/100$ e marcou-se o ponto $z_i = (t_i, p_i)$, para $i = 1, ..., 100$.

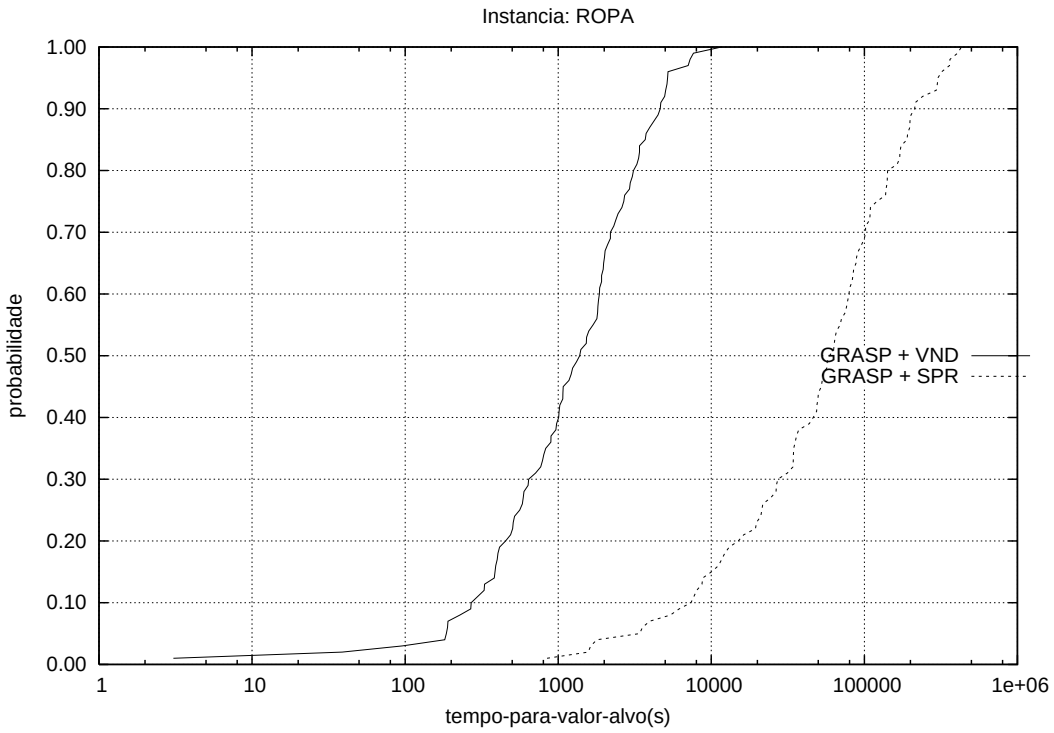

Figura 3.12: Distribuições empíricas de probabilidade do tempo-para-valor-alvo para a instância ROPA para o algoritmo GRASP em [3, 4] e a heurística `GRASP+VND`.

Os gráficos das Figuras 3.12 e 3.13 mostram que a heurística `GRASP+VND` é aproximadamente duas ordens de magnitude mais rápida do

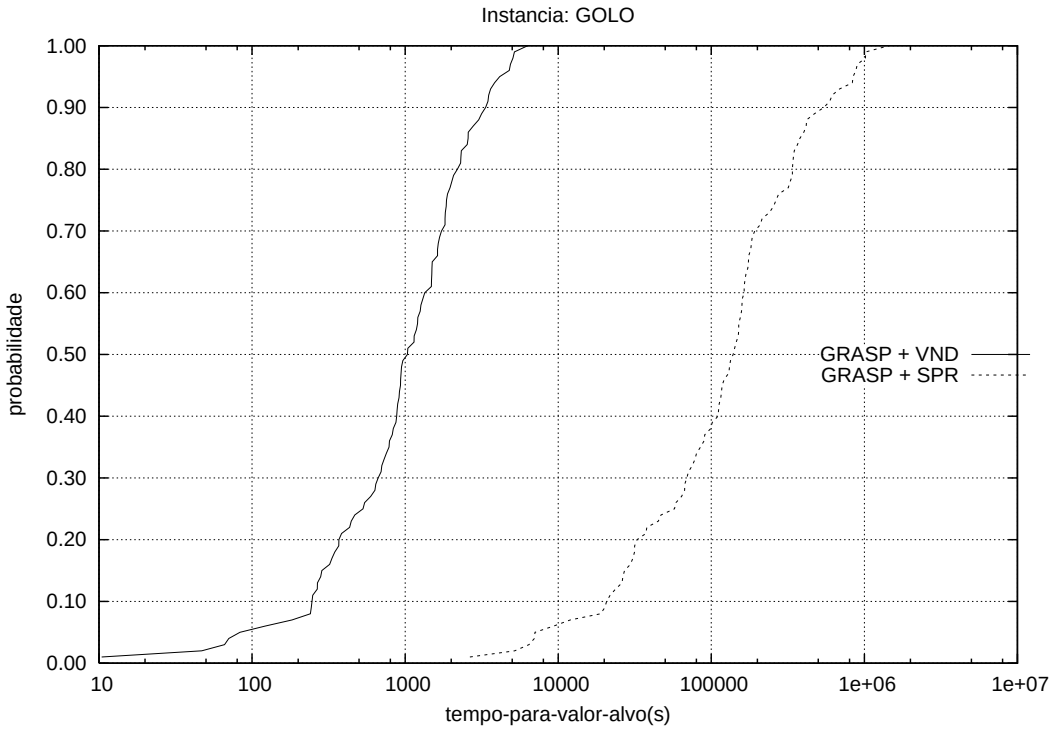

Figura 3.13: Distribuições empíricas de probabilidade do tempo-para-valor-alvo para a instância GOLO para o algoritmo GRASP em [3, 4] e a heurística `GRASP+VND`.

que a implementação GRASP em [3, 4], claramente mostrando a melhoria ocasionada pelo uso de uma estratégia de busca local VND. A nova heurística claramente supera a citada em [3, 4]: para um dado tempo de computação, a probabilidade de encontrar uma solução pelo menos tão boa quanto o valor alvo é muito maior para a heurística `GRASP+VND`.

3.8 Conclusão

Neste capítulo foi implementado o algoritmo híbrido `GRASP+VND`, o qual é um algoritmo GRASP que utiliza VND como método de busca local. A estrutura de vizinhança utilizada é a k-SPR, que foi proposta neste trabalho. Este algoritmo foi comparado com o algoritmo GRASP em [3, 4], que utilizava a vizinhança SPR em seu método de busca local. Os dois algoritmos foram submetidos a experimentos utilizando 8 instâncias da literatura e 20 instâncias geradas aleatoriamente. O testes mostraram que o algoritmo proposto é superior tanto em qualidade das soluções obtidas quanto em tempo computacional. Os testes também mostraram que se ambos algoritmos forem executados durante um determinado tempo, o

algoritmo GRASP+VND possui uma probabilidade maior de encontrar uma solução pelo menos tão boa quanto um determinado alvo.