4
Heurísticas

O modelo apresentado no Capítulo 3 possibilita uma abordagem exata para a resolução do problema de correspondência inexata de grafos. Esta modelagem permite que um algoritmo de enumeração, como por exemplo um algoritmo de *branch–and–bound*, seja utilizado na obtenção de uma solução exata para o problema. A grande limitação desses algoritmos de enumeração encontra-se no fato de apresentarem complexidade exponencial no tamanho da entrada, ou seja, exigem um tempo de processamento que cresce exponencialmente com o aumento do tamanho do problema.

Uma técnica bastante utilizada na abordagem de problemas NP-difíceis é a utilização de procedimentos heurísticos, como por exemplo algoritmos para construção de soluções, técnicas de busca local e meta-heurísticas. Basicamente, pode-se dizer que uma metaheurística consiste na combinação de heurísticas mais simples com o objetivo de encontrar soluções melhores, ou seja, procura-se reduzir as chances de que o algoritmo fique preso a ótimos locais tal qual em uma busca local.

Este capítulo propõe alguns algoritmos aproximados para o problema de correspondência inexata de grafos. Inicialmente, é apresentado um novo algoritmo construtivo guloso aleatorizado. Em seguida, são descritos os algoritmos de busca local desenvolvidos em [14]. Por fim, é proposto um algoritmo GRASP que combina o algoritmo construtivo guloso com uma busca local e com um procedimento original de reconexão por caminhos.

4.1
Métodos construtivos

Os métodos construtivos são procedimentos heurísticos, normalmente baseados nas propriedades estruturais dos problemas abordados, cuja função é fornecer como resposta uma solução viável para o problema. Seu objetivo é oferecer boas soluções, sem se preocupar com uma garantia da qualidade das soluções encontradas.

Uma solução viável para o problema de correspondência de grafos apresentado no Capítulo 2 consiste em um conjunto de associações, cada uma delas formada por um vértice de G_M e um vértice de G_I, respeitando-se as restrições do modelo.

A seguir são descritas três variantes de métodos para construção de soluções viáveis. Primeiramente, descreve-se sucintamente o algoritmo construtivo apresentado em [14], aqui denominado `Construtivo_1`. Em seguida, é proposto um método guloso aleatorizado, chamado `Construtivo_2` e uma variante desse algoritmo denominada `Construtivo_2a`.

4.1.1
`Construtivo_1`

O Pseudo-código 1 foi proposto por Boeres em [14]. Os vértices de V_I são percorridos em uma ordem aleatória e, um por um, são associados a algum vértice de V_M também escolhido ao acaso. Uma solução é concluída quando todos os vértices de G_I tiverem sido tratados. Esta solução pode ser viável ou não, dependendo se atende às restrições impostas pelo modelo. Esse processo é repetido $MaxTentativas$ vezes ou até que $MaxSolucoes$ soluções viáveis tenham sido construídas. Sua complexidade para construção de uma solução é $O(|E_M|.|V_I|^2)$.

Os Pseudo–códigos 2 e 3 representam os refinamentos dos passos 4 e 17 respectivamente.

algoritmo `Construtivo_1`($Semente, MaxTentativas, MaxSolucoes, G_M, G_I$)

1 $n\_tentativas \leftarrow 1$; $n\_solucoes \leftarrow 1$;
2 $S^* \leftarrow \emptyset$; $S \leftarrow \emptyset$;
3 **Enquanto** ($n\_tentativas \leq MaxTentativas$) **e** ($n\_solucoes \leq MaxSolucoes$) **faça**
4 `Inicializações`();
5 $T_I \leftarrow V_I$;
6 **Enquanto** $T_I \neq \emptyset$ **faça**
7 $j \leftarrow$ `obtemVerticeAleatoriamente`(T_I);
8 $T_I \leftarrow T_I - \{j\}$;
9 $T_M \leftarrow V_M$;
10 **Enquanto** $T_M \neq \emptyset$ **e** $A_S^{-1}(j) = \emptyset$ **faça**
11 $i \leftarrow$ `obtemVerticeAleatoriamente`(T_M);
12 $T_M \leftarrow T_M - \{i\}$;
13 **Se** ($s^v(i,j) > 0$) **e** ($G_I^{Ind}(A_S(i) \cup \{j\})$ é conexo) **então**
14 $x_{ij} \leftarrow 1$;
15 $A_S(i) \leftarrow A_S(i) \cup \{j\}$;
16 $A_S^{-1}(j) \leftarrow \{i\}$;
17 `Calcula_Contribuições`();
18 **fim_se**
19 **fim_enquanto**
20 **fim_enquanto**
21 $c_v \leftarrow c_v \cdot \alpha / (|V_M| \cdot |V_I|)$;
22 $c_a \leftarrow c_a \cdot (1-\alpha) / (|E_M| \cdot |E_I|)$;
23 $f(S) \leftarrow c_v + c_a$;
24 **Se** $A_S(i) \neq \emptyset, \forall i \in V_M$ **e** $A_S^{-1}(j) \neq \emptyset, \forall j \in V_I$ **então**
25 $n\_solucoes \leftarrow n\_solucoes + 1$;
26 **Se** $f(S) > f(S^*)$ **então**
27 $S^* \leftarrow S$;
28 **fim_se**
29 **fim_se**
30 $n\_tentativas \leftarrow n\_tentativas + 1$;
31 **fim_enquanto**
32 **retorna** (S^*);
fim

Pseudo–código 1: Algoritmo construtivo puramente aleatorizado.

algoritmo `Inicializações`()

1 $A_S(i) \leftarrow \emptyset, \forall i \in V_M$;
2 $A_S^{-1}(j) \leftarrow \emptyset, \forall j \in V_I$;
3 $c_v \leftarrow \sum_{(i,j) \in V_M \times V_I} (1 - s^v(i,j))$;
4 $c_a \leftarrow \sum_{((i,i'),(j,j')) \in E_M \times E_I} (1 - s^a((i,i'),(j,j')))$;
5 $x_{ij} \leftarrow 0, \forall i \in V_M \; e \; \forall j \in V_I$;
fim

Pseudo–código 2: Inicializações para cada tentativa de construção.

```
algoritmo Calcula_Contribuições()
   1 c_v ← c_v + (2 · s^v(i, j) − 1);
   2 c_a ← 0;
   3 Para_todo i' ∈ Γ_i faça
   4    Para_todo j' ∈ Γ_j faça
   5       Se x_{i'j'} = 1 então
   6          c_a ← c_a + (2 · s^a((i, i'), (j, j')) − 1);
   7       fim_se
   8    fim_para_todo
   9 fim_para_todo
fim
```

Pseudo–código 3: Cálculo das contribuições de vértices e arestas.

4.1.2

`Construtivo_2`

O algoritmo `Construtivo_2` foi projetado com base no algoritmo anterior. A diferença entre eles se resume à escolha do vértice de G_M que será associado a cada elemento de G_I. Ao contrário do algoritmo `Contrutivo_1`, que faz uma escolha aleatória que satisfaça às restrições, este novo algoritmo utiliza-se de uma função gulosa. Dado um vértice j de G_I, são computados os valores de contribuição para todos os vértices de G_M cuja associação a j seja viável. O vértice de G_M com maior contribuição é então associado a j na solução construída. O Pseudo-código 4 apresenta o algoritmo `Construtivo_2`.

Assim como o algoritmo anterior, o algoritmo `Construtivo_2` não precisa recalcular a função objetivo a cada iteração. Uma vez computado inicialmente o termo constante, a cada associação efetuada apenas a variação causada na função deve ser adicionada. Os cálculos da inicialização e posteriores atualizações foram mostrados no Capítulo 2.

O algoritmo recebe como parâmetros a semente inicial para geração de números aleatórios, o número máximo de tentativas ($MaxTentativas$) a serem realizadas e o número máximo de soluções viáveis a serem construídas ($MaxSolucoes$). As linhas 1 e 2 efetuam as inicializações globais para o algoritmo e a linha 4 realiza as inicializações para cada iteração (refinada no Pseudo-código 2). O laço das linhas 3 a 40 é o responsável pelas $MaxTentativas$ vezes em que se tenta construir no máximo $MaxSolucoes$ soluções viáveis. O laço das linhas 6 a 29 constrói uma solução para o PCIG. Nas linhas 7 e 8, um vértice é aleatoriamente retirado de T_I e guardado em

algoritmo `Construtivo_2`($Semente, MaxTentativas, MaxSolucoes, G_M, G_I$)

1 $n\_tentativas \leftarrow 1$; $n\_solucoes \leftarrow 1$;
2 $S^* \leftarrow \emptyset$;$S \leftarrow \emptyset$;
3 **Enquanto** ($n\_tentativas \leq MaxTentativas$) **e** ($n\_solucoes \leq MaxSolucoes$) **faça**
4 `Inicializações`();
5 $T_I \leftarrow V_I$;
6 **Enquanto** $T_I \neq \emptyset$ **faça**
7 $j \leftarrow$ `obtemVerticeAleatoriamente`(T_I);
8 $T_I \leftarrow T_I - \{j\}$;
9 $T_M \leftarrow V_M$;
10 **Para_todo** $i \in T_M$ **faça**
11 $\Delta[i] \leftarrow \Delta_a[i] \leftarrow \Delta_v[i] \leftarrow -\infty$
12 **Se** ($s^v(i,j) > 0$) **e** ($G_I^{Ind}(A_S(i) \cup \{j\})$ é conexo) **então**
13 $\Delta_v[i] \leftarrow 2 \cdot s^v(i,j) - 1$;
14 $\Delta_a[i] \leftarrow 0$;
15 **Para_todo** $j^{'} \in \Gamma_j$ **faça**
16 $i^{'} \leftarrow A_S^{-1}(j^{'})$
17 **Se** $x_{i^{'}j^{'}} = 1$ **e** $i^{'} \in \Gamma_i$ **então**
18 $\Delta_a[i] \leftarrow \Delta_a[i] + 2 \cdot s^a((i,i^{'}),(j,j^{'})) - 1$;
19 **fim_se**
20 **fim_para_todo**
21 $\Delta[i] \leftarrow \alpha \cdot \Delta_v[i] + (1-\alpha) \cdot \Delta_a[i]$;
22 **fim_se**
23 **fim_para_todo**
24 $best \leftarrow$ `argmax`$\{\Delta[i], i \in T_M\}$
25 $x_{best,j} \leftarrow 1$;
26 $A_S(best) \leftarrow A_S(best) \cup \{j\}$;
27 $A_S^{-1}(j) \leftarrow \{best\}$;
28 $c_v \leftarrow c_v + \Delta_v[best]$; $c_a \leftarrow c_a + \Delta_a[best]$;
29 **fim_enquanto**
30 $c_v \leftarrow c_v \cdot \alpha/(|V_M| \cdot |V_I|)$;
31 $c_a \leftarrow c_a \cdot (1-\alpha)/(|E_M| \cdot |E_I|)$;
32 $f(S) \leftarrow c_v + c_a$;
33 **Se** $A_S(i) \neq \emptyset, \forall i \in V_M$ **e** $A_S^{-1}(j) \neq \emptyset, \forall j \in V_I$ **então**
34 $n\_solucoes \leftarrow n\_solucoes + 1$;
35 **Se** $f(S) > f(S^*)$ **então**
36 $S^* \leftarrow S$;
37 **fim_se**
38 **fim_se**
39 $n\_tentativas \leftarrow n\_tentativas + 1$;
40 **fim_enquanto**
41 **retorna** (S^*);

fim

Pseudo–código 4: Algoritmo construtivo guloso aleatorizado.

j. Na linha 9 é criada uma cópia de V_M chamada T_M. O laço das linhas 10 a 23 realiza os cálculos da variação na função objetivo relativa à associação de cada um dos vértices de T_M ao vértice j. Nas linhas 24 a 28 é escolhido o vértice de maior contribuição e são atualizados os valores das contribuições de vértices e arestas para a solução atual. As linhas 30 a 32 atualizam o valor da função objetivo para a solução atual construída e o teste da linha 33 verifica se esta solução é viável. Em caso afirmativo, verifica-se a necessidade de se atualizar a melhor solução construída até o momento na linha 36.

No pior caso, para um determinado elemento de G_I selecionado, devem ser feitos os cálculos de contribuições para todos os vértices de G_M. Cada um destes cálculos tem complexidade $O(|V_I|)$. Assim, para cada vértice de G_I, os cálculos de contribuições têm complexidade $O(|V_M| \cdot |V_I|)$, o que resulta em uma complexidade total $O(|V_M| \cdot |V_I|^2)$. Como a inicialização da contribuição de arestas é $O(|E_M| \cdot |E_I|)$, a complexidade para construir uma solução é $O(|E_M| \cdot |E_I| + |V_M| \cdot |V_I|^2) = O(|E_M| \cdot |V_I|^2)$. Este procedimento é executado no máximo $MaxTentativas$ vezes.

4.1.3

`Construtivo_2a`

Este algoritmo é uma variação do algoritmo `Construtivo_2` e foi projetado com o intuito de se inserir um certo grau de aleatorização na escolha gulosa das associações. Algoritmos construtivos não determinísticos são apropriados para utilização em procedimentos GRASP, como será visto com mais detalhes na Seção 4.3.

Basicamente, este algoritmo consiste em, para cada vértice j de G_I, selecionar em G_M, de maneira equiprovável, um vértice dentre os cinco com melhores contribuições para se associar a j, ao contrário do algoritmo anterior que escolhe necessariamente a associação de maior contribuição.

De modo geral, a alteração descrita acima é a única diferença entre os algoritmos `Construtivo_2` e `Construtivo_2a`. Desta forma, sua complexidade é a mesma apresentada pelo anterior.

4.1.4

`Construtivo_3`

O algoritmo `Construtivo_3` foi criado com base no esquema descrito por Resende e Ribeiro [55] para a fase de construção de uma heurística GRASP.

A idéia original do esquema é a de se construir uma solução a partir de uma lista restrita de candidatos (LRC). É criado um conjunto de elementos candidatos a entrar na solução em construção, para os quais são calculados os custos incrementais. A lista deve ser formada apenas por elementos que mantenham a viabilidade da solução. A cada iteração, escolhe-se um elemento da LRC para ser incorporado à nova solução e atualiza-se a lista de candidatos. Este processo é então repetido até que a solução esteja construída.

Para o caso do PCIG, os elementos que compõem a LRC são todas as associações viáveis entre pares de vértices de V_M e V_I, ou seja, as associações que não violam as restrições de conexidade e de similaridade não nula. A cada iteração uma associação é selecionada aleatoriamente da LRC. O número de elementos (n) da LRC depende da cardinalidade de V_I. Na implementação para o PCIG desenvolvida neste trabalho, utiliza-se $n = 5$, se $|V_I| < 50$; $n = 20$, se $|V_I| > 200$ e $n = 0,1 \cdot |V_I|$, caso contrário. A probabilidade de escolha de cada associação é proporcional ao seu custo incremental, ou seja, aquelas com maiores custos incrementais são as mais prováveis de serem escolhidas.

No esquema proposto em [55], a cada iteração a lista restrita de candidatos é atualizada. No entanto, para o PCIG, a complexidade dessa atualização é alta. Por outro lado, a realização de uma associação não necessariamente interfere nos cálculos de contribuições de outras associações da lista que ainda não foram realizadas. Com o objetivo de aproveitar os cálculos de iterações anteriores, o algoritmo `Construtivo_3` realiza várias iterações com a mesma LRC. A cada vez que a lista é reaproveitada para selecionar a nova associação a entrar na solução, deve-se recalcular o custo incremental desta solução e verificar sua viabilidade. Uma associação somente é aceita se for viável e a variação do custo incremental calculado anteriormente e o custo atual for menor que 1%. Caso a associação escolhida não satisfaça as restrições, uma nova deve ser selecionada. A LRC deve ser reconstruída quando nenhuma associação satisfizer estes critérios.

O Pseudo-código 5 apresenta o algoritmo `Construtivo_3`. Nas linhas 3 a 5, é atribuído o valor de n. O laço iniciado na linha 6 constrói as $MaxSolucoes$ soluções em no máximo $MaxTentativas$ tentativas. Após as devidas inicializações, o laço das linhas 8 a 17 constrói uma solução para o PCIG. Na linha 9, a chamada `ConstruirLRC()` constrói a lista restrita de candidatos (LRC) com todas as associações viáveis (i, j) tais que $i \in V_M$ e $j \in V_I$. No processo de construção da LRC, também é efetuado o cálculo das contribuições relativas a cada associação de maneira idêntica aos algoritmos anteriores (Pseudo-código 3). O laço das linhas 10 a 16 realiza associações selecionadas da LRC, enquanto existirem associações viáveis e cujos custos incrementais não tenham se alterado mais de 1%. Estes testes são realizados pela chamada `Valida?()`. Na linha 11, a chamada `SelecioneAssociacaoValida()` seleciona aleatoriamente da LRC uma associação válida. As linhas 12 a 15 fazem as atualizações na solução em construção e removem a associação realizada da LRC.Quando nenhuma associação satisfizer as restrições, a LRC deverá ser reconstruída. Ao final da construção de cada solução, é feito o teste de viabilidade da solução construída e atualizada a melhor solução encontrada, se for o caso (linhas 18 a 22).

A diferença deste algoritmo para o algoritmo `Construtivo_2a` é que este último faz as escolhas aleatórias dentre as melhores associações para um vértice $j \in V_I$ fixo e repete este procedimento para cada elemento de V_I. O algoritmo `Construtivo_3` escolhe uma dentre todas as associações viáveis entre quaisquer pares de vértices e repete este procedimento até que todos os vértices de V_I tenham sido associados a algum vértice de V_M.

O custo para se construir a LRC é $O(|V_M| \cdot |V_I|^2)$, pois são feitos os cálculos de contribuições ($O(|V_I|)$) para cada par de vértices ($O(|V_M| \cdot |V_I|)$). No pior caso, a lista restrita de candidatos deverá ser recalculada em todas as $|V_I|$ iterações. Assim, a complexidade para se construir uma solução utilizando o algoritmo `Construtivo_3` é $O(|V_M| \cdot |V_I|^3)$.

4.2 Métodos de busca local

Os métodos construtivos discutidos na seção anterior são utilizados na construção de soluções viáveis para o problema estudado, mas não garantem

```
algoritmo Construtivo_3(Semente, MaxTentativas, MaxSolucoes,
G_M, G_I)
 1  n_tentativas ← 1; n_solucoes ← 1;
 2  S* ← ∅;S ← ∅;
 3  n ← 0.1 · |V_I|;
 4  Se n < 5 então n ← 5;
 5  Senão se n > 20 então n ← 20;
 6  Enquanto (n_tentativas ≤ MaxTentativas)
    e (n_solucoes ≤ MaxSolucoes) faça
 7     Inicializações();
 8     Enquanto ∃j ∈ V_I : A_S^{-1}(j) = ∅ faça
 9        ConstruirLRC(LRC);
10        Enquanto ∃(i, j) ∈ LRC : Valida?(i, j) =verdadeiro
    faça
11           (i, j) ← SelecioneAssociacaoValida(LRC, n);
12           x_{i,j} ← 1;
13           A_S(i) ← A_S(i) ∪ {j};
14           A_S^{-1}(j) ← {i};
15           LRC ← LRC − {(i, j)};
16        fim_enquanto
17     fim_enquanto
18     Se A_S(i) ≠ ∅, ∀i ∈ V_M e A_S^{-1}(j) ≠ ∅, ∀j ∈ V_I então
19        n_solucoes ← n_solucoes + 1;
20        Se f(S) > f(S*) então
21           S* ← S;
22        fim_se
23     fim_se
24     n_tentativas ← n_tentativas + 1;
25  fim_enquanto
26  retorna (S*);
fim
```

Pseudo–código 5: Algoritmo construtivo guloso aleatorizado com lista de candidatos.

que tais soluções sejam, nem mesmo, ótimos locais. Heurísticas de busca local são métodos que procuram modificar uma solução na tentativa de gerar uma nova solução com melhor qualidade. Pode-se compreender esses métodos como procedimentos de busca que exploram o espaço de soluções do problema, procurando a solução ótima.

Os métodos de busca local necessitam da definição da vizinhança de uma solução, ou seja, o conjunto de soluções que, segundo algum critério, encontram-se próximas a uma dada solução no espaço de busca. A seguir serão definidas as vizinhanças utilizadas neste trabalho e que foram propostas em [14].

Uma vez definida a vizinhança, os métodos de busca local em geral consideram a cada iteração uma determinada solução viável e avaliam suas soluções vizinhas, tentando encontrar uma de melhor qualidade. Uma vez encontrada uma solução vizinha aprimorante, ela se torna a solução corrente e o processo é repetido. Tipicamente, os métodos de busca local terminam quando encontram um ótimo local, ou seja, uma solução sem vizinhos aprimorantes.

Neste trabalho, o método de busca local utilizado é denominado método de melhoria iterativa. A mais típica especialização deste método seleciona, entre os vizinhos da solução corrente, o primeiro vizinho aprimorante.

Outro critério para a seleção de vizinhos corresponde ao método de descida mais rápida. Seu objetivo é selecionar o vizinho aprimorante que represente a melhor solução entre todos os vizinhos da solução atual. Para isso, é necessário calcular o valor da solução de todos os vizinhos, para somente então determinar o melhor deles.

Na prática, observa-se que freqüentemente procedimentos de busca local com ambas estratégias levam à mesma solução final, porém com menores tempos de processamento para a melhoria iterativa. Além disso, a convergência prematura para ótimos locais não globais é mais comum para o método de descida mais rápida [55].

4.2.1 Vizinhanças

A qualidade dos algoritmos de busca local depende fundamentalmente da definição de uma boa vizinhança. Uma característica desejável de uma vizinhança é a conexidade: deve haver um caminho entre duas soluções quaisquer no espaço de busca. É conveniente que o diâmetro da vizinhança, isto é, a maior distância entre duas soluções, seja o menor possível [58].

São descritas a seguir as duas vizinhanças propostas por Boeres [14] para o PCIG, denotadas por V_a e V_b.

A vizinhança $V_a(S)$ é composta pelas soluções viáveis geradas pela modificação de uma associação de S, ou seja, para algum j pertencente a V_I, muda-se o valor de $A_S^{-1}(j)$, mantendo-se a viabilidade da nova solução

gerada.

Define-se o conjunto $C(j)$, para cada vértice j pertencente a V_I da solução atual S, como sendo o conjunto dos vértices de V_M candidatos à substituição do vértice associado com j na solução corrente. Assim,

$C(j) = \{k \in V_M | x^{'}$ *é uma solucão viável, onde* $x^{'}_{i\ell} = 1$ *se* $i = k$ *e* $\ell = j$, $x^{'}_{i\ell} = 0$ *se* $i = A_S^{-1}(j)$ *e* $\ell = j, x^{'}_{i\ell} = x_{i\ell}$ *caso contrário*$\}$.

A Figura 4.1 mostra duas soluções vizinhas de acordo com a vizinhança V_a. O vértice $2 \in G_I$ está associado ao vértice $d \in G_M$ na solução atual. Na solução vizinha, 2 aparece associado ao vértice $a \in G_M$.

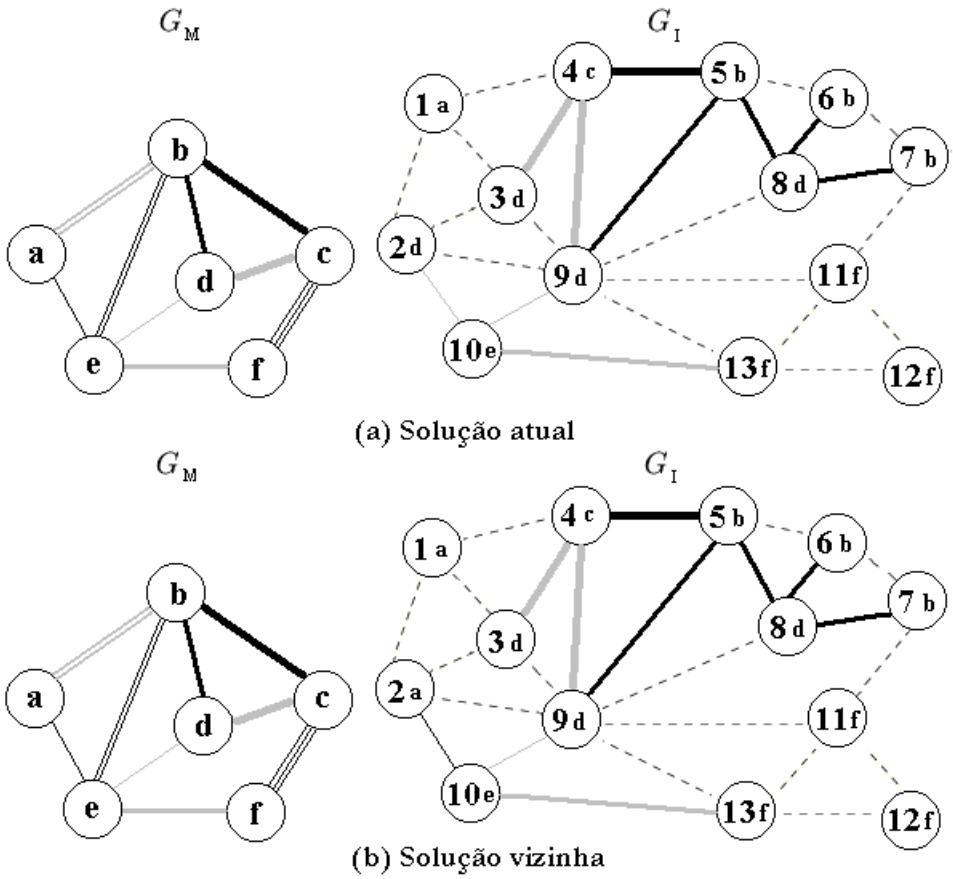

Figura 4.1: Duas soluções vizinhas para a vizinhança V_a.

A vizinhança $V_b(S)$ é formada por soluções viáveis obtidas pela troca dos valores de $A_S^{-1}(j^{'})$ por $A_S^{-1}(j^{''})$, e vice-versa, em S, para dois vértices distintos $j^{'}$ e $j^{''}$ de V_I.

Um exemplo com duas soluções vizinhas de acordo com a vizinhança V_b pode ser encontrado na Figura 4.2. Na solução atual, tem-se os vértices 2 e 10 pertencentes a G_I associados respectivamente aos vértices e e d de

G_M. A solução vizinha pode ser obtida pela troca de associações entre esses vértices. Neste caso, o vértice 2 passa a se associar ao d e o 10 ao e.

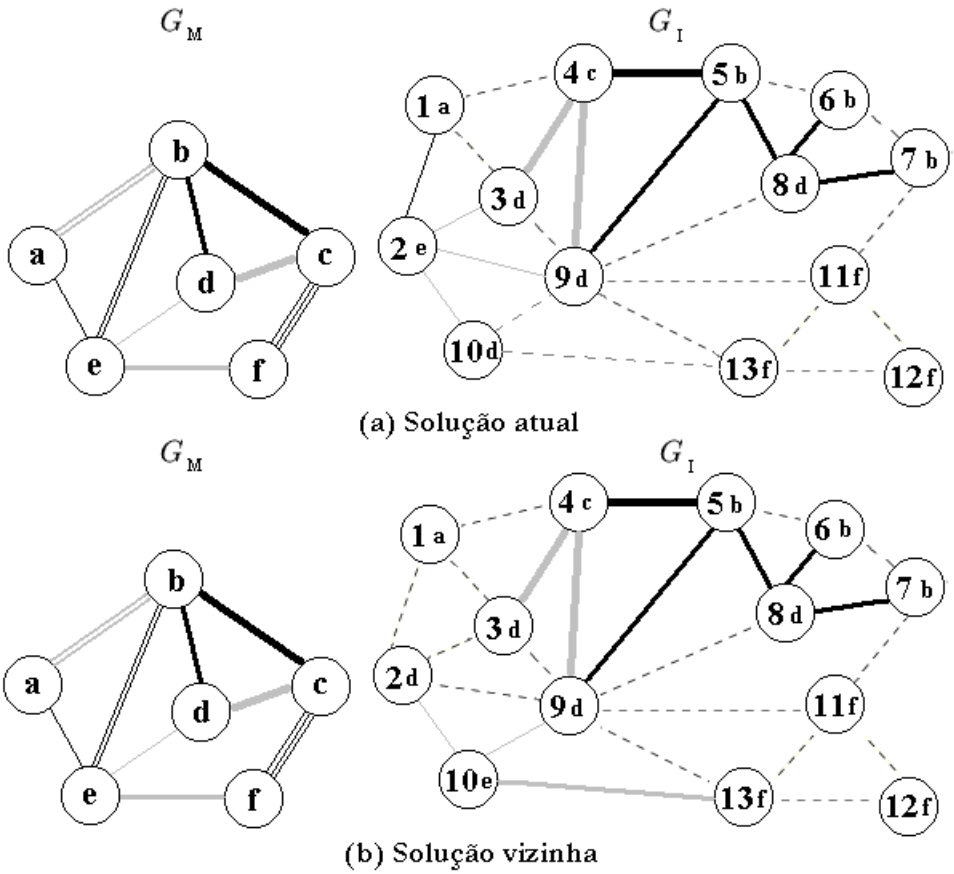

Figura 4.2: Duas soluções vizinhas para a vizinhança V_b.

Na vizinhança $V_a(S)$ de uma solução S, o número máximo de soluções vizinhas de uma solução S é igual a $\sum_{j \in V_I} |C(j)| \leq |V_M| \cdot |V_I|$. Na vizinhança $V_b(S)$, o número máximo de vizinhos é igual a $|V_M| \cdot (|V_M| - 1)/2$, pois o número de pares de vértices de V_I associados a vértices de V_M distintos que podem ser trocados é no máximo igual ao número de combinações dois a dois dos vértices de V_M.

4.2.2

Busca local na Vizinhança V_a

O Pseudo-código 6 mostra o algoritmo de busca local $\texttt{BL}_a$ com base na vizinhança V_a, que recebe como entradas os grafos G_M e G_I, além de uma solução inicial S obtida pelo algoritmo construtivo. Seu objetivo é buscar, na vizinhança da solução corrente, uma solução que melhore o valor da função

algoritmo $\texttt{BL}_a(Semente, G_M, G_I, S)$
1 $improv \leftarrow$ **verdadeiro**;
2 Constrói os conjuntos $C(j)$, $\forall j \in V_I$;
3 **Enquanto** $improv =$ **verdadeiro faça**
4 $improv \leftarrow$ **falso**;
5 **Para_todo** $j \in V_I$ **enquanto** $improv =$ **falso faça**
6 $i \leftarrow A_S^{-1}(j)$;
7 **Para_todo** $k \in C(j)$ **enquanto** $improv =$ **falso faça**
8 $\Delta^v \leftarrow 2 \cdot s^v(k,j) - 2 \cdot s^v(i,j)$;
9 $\Delta^a \leftarrow 0$;
10 **Para_todo** $j^{'} \in \Gamma_j$ **faça**
11 **Se** $A_S^{-1}(j^{'}) \neq \emptyset$ **então**
12 $k^{'} \leftarrow A_S^{-1}(j^{'})$;
13 **Se** $k^{'} \in \Gamma_i$ **então**
14 $\Delta^a \leftarrow \Delta^a + 1 - 2 \cdot s^a((i,k^{'}),(j,j^{'}))$;
15 **fim_se**
16 **Se** $k^{'} \in \Gamma_k$ **então**
17 $\Delta^a \leftarrow \Delta^a - 1 + 2 \cdot s^a((k,k^{'}),(j,j^{'}))$;
18 **fim_se**
19 **fim_se**
20 **fim_para_todo**
21 $\Delta \leftarrow \alpha \cdot \Delta^v/(|V_M| \cdot |V_I|) + (1-\alpha) \cdot \Delta^a/(|E_M| \cdot |E_I|)$;
22 **Se** $\Delta > 0$ **então**
23 $improv \leftarrow$ **verdadeiro**;
24 $A_S^{-1}(j) \leftarrow \{k\}$;
25 $A_S(i) \leftarrow A_S(i) - \{j\}$;
26 $A_S(k) \leftarrow A_S(k) \cup \{j\}$;
27 $f(S) \leftarrow f(S) + \Delta$;
28 Atualizar os conjuntos $C(j)$, $\forall j \in V_I$;
29 **fim_se**
30 **fim_para_todo**
31 **fim_para_todo**
32 **fim_enquanto**
33 **Retorna** (S);
fim

Pseudo–código 6: Busca local na vizinhança V_a.

objetivo, movendo-se para a primeira solução aprimorante que encontrar. Este processo é repetido enquanto existir alguma solução aprimorante.

As linhas 1 e 2 fazem as devidas inicializações. O laço das linhas 3 a 32, enquanto existir algum vizinho aprimorante, busca pelo primeiro deles. Quando o encontra, move-se para este vizinho, ou seja, faz com que a nova solução corrente passe a ser o vizinho encontrado.

O laço das linhas 5 a 31, a cada passo, se ainda não encontrou um vizinho aprimorante, seleciona aleatoriamente um elemento $j \in V_I$ que ainda não foi analisado. Em seguida, avalia o conjunto de candidatos de $C(j)$, verificando se a troca da associação atual de j por algum de seus candidatos resulta em melhora da qualidade da solução (aumento da função objetivo f). Caso encontre alguma troca aprimorante, pára a procura e efetua a troca.

O laço das linhas 10 a 20 faz os cálculos das variações na função objetivo das contribuições de vértices e arestas. A instrução da linha 21 realiza o cálculo final da variação na função objetivo. A linha 22 testa se a troca é aprimorante e, em caso afirmativo, as linhas 23 a 28 fazem as atualizações necessárias para que o vizinho aprimorante encontrado torne-se a nova solução corrente.

A complexidade da construção dos conjuntos $C(j)$ é $O(|E_I| \cdot |V_I|)$, pois para cada vértice de V_I deve-se testar a viabilidade de troca para todos os vértices de V_M e o custo de cada teste de viabilidade é $O(|V_I| + |E_I|)$. A atualização das linhas 23 a 28 tem custo $O(|E_I| \cdot |V_I|)$, uma vez que os conjuntos $C(j)$ devem ser atualizados.

No pior caso, o laço das linhas 5 a 31 procura por um vizinho para todos os vértices de V_I e, para cada um deles, analisa todos os seus candidatos. Isso totaliza um número de execuções proporcional a $|V_M| \cdot |V_I|$ para os cálculos das linhas 8 a 21. A atualizações das linhas 23 a 28 são executadas apenas uma vez para cada solução atual, pois quando o vizinho aprimorante é encontrado, a busca começa novamente do inicio.

Os cálculos das linhas 8 a 21 têm complexidade $O(|V_I|)$. Portanto o laço das linhas 5 a 31 tem complexidade final $O(|V_M| \cdot |V_I|^2)$ e é executado até que nenhum vizinho aprimorante seja encontrado. A complexidade de cada iteração da busca local na vizinhança V_a é então $O(|V_M| \cdot |V_I|^2 + |E_I| \cdot |V_I|)$.

4.2.3

Busca local nas vizinhanças V_a e V_b

O procedimento de busca descrito na seção anterior procura por soluções aprimorantes na vizinhança V_a, até que um ótimo local seja encontrado. Vale ressaltar que o espaço de busca considerado é o de soluções viáveis e que cada vizinho de uma solução difere da mesma por apenas uma associação. O número total de vizinhos de uma solução é igual a $|V_I|\cdot(|V_M|-1)$, mas na prática muitos desses vizinhos são inviáveis e portanto não participam da busca, o que caracteriza uma limitação da vizinhança V_a. Este fato pode ser exemplificado pela solução apresentada na Figura 4.3. Apesar dessa solução possuir 12 vizinhos, apenas sete são viáveis, pois o vértice 6 não pode ter sua associação trocada (é o único associado ao vértice c) e os vértices 1, 2 e 4 não podem se associar ao vértice c, o que violaria a restrição de conexidade.

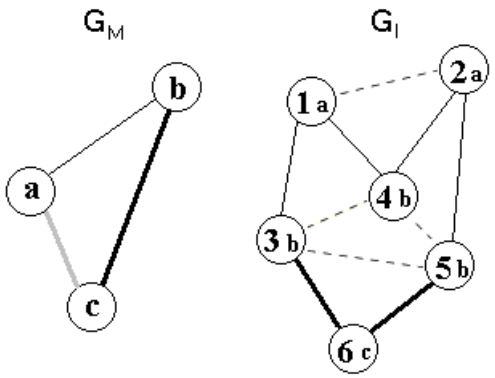

Figura 4.3: Exemplo de solução para uma instância pequena do PCIG.

A busca local $\texttt{BL}_{a+b}$ tem como objetivo transpor essa limitação, desprendendo-se dos ótimos locais para a vizinhança V_a e buscando por um vizinho aprimorante na vizinhança V_b. A seguir é descrita com detalhes a busca local $\texttt{BL}_{a+b}$.

O algoritmo de busca local $\texttt{BL}_{a+b}$ procede inicialmente a uma busca local na vizinhança V_a, exatamente como o faz o algoritmo $\texttt{BL}_a$, até que nenhum vizinho aprimorante seja encontrado. Em seguida, efetua uma tentativa de saída do ótimo local em que se encontra, realizando uma busca na vizinhança V_b. Quando uma solução aprimorante é encontrada, o algoritmo recomeça a busca usando a vizinhança V_a. Este procedimento continua até que a busca na vizinhança V_a chegue a um ótimo local do qual a busca na vizinhança V_b também não consiga sair, ou seja, um ótimo local relativo às duas vizinhanças.

No Pseudo-código 7, a linha 4 executa uma busca local na vizinhança V_a, tal qual foi descrita no tópico anterior. O laço das linhas 5 a 18 corresponde à busca na vizinhança V_b e é executado a partir do ótimo local encontrado na vizinhança V_a. As duas buscas são realizadas dentro do laço que começa na linha 2, o qual é executado enquanto a busca na vizinhança V_b conseguir aprimorar a solução retornada pela busca $\texttt{BL}_a$.

```
algoritmo BL_{a+b}(Semente, G_M, G_I, S)
 1  improv2 ←verdadeiro;
 2  Enquanto improv2 = verdadeiro faça
 3     improv2 ←falso;
 4     S ← BL_a(Semente, S);
 5     Para_todo (j ∈ V_I) enquanto improv2 = falso faça
 6        i ← A_S^{-1}(j);
 7        Para_todo (j' ∈ V_I e j' > j)
          enquanto improv2 = falso faça
 8           i' ← A_S^{-1}(j');
 9           Se i ≠ i' então
10              Se s^v(i, j') > 0 e s^v(i', j) > 0 e
                   (G_I^{ind}(A_S(i)) − {j} ∪ {j'} é conexo) e
                   (G_I^{ind}(A_S(i')) − {j'} ∪ {j} é conexo) então
11                 CalculaValorDaTroca();
12                 Δ ← α · Δ^v/(|V_M| · |V_I|) + (1 − α) · Δ^a/(|E_M| ·
   |E_I|);
13                 Se Δ > 0 então
14                    improv2 ←verdadeiro;
15                    A_S^{-1}(j) ← i';
16                    A_S^{-1}(j') ← i;
17                    A_S(i) ← A_S(i) − {j} ∪ {j'};
18                    A_S(i') ← A_S(i') − {j'} ∪ {j};
19                    f(S) ← f(S) + Δ;
20                 fim_se
21              fim_se
22           fim_se
23        fim_para_todo
24     fim_para_todo
25  fim_enquanto
26  Retorna (S);
fim
```

Pseudo–código 7: Busca local nas vizinhanças V_a e V_b.

Os laços iniciados nas linhas 5 e 7 têm como objetivo analisar todos os possíveis pares de vértices de G_I, até que a troca de associações entre um par de vértices seja aprimorante. Neste caso, a troca é efetuada e a busca na vizinhança V_b termina.

```
algoritmo CalculaValorDaTroca()
 1  Δ^v ← 2 · s^v(i, j') + 2 · s^v(i', j) − 2 · s^v(i, j) − 2 · s^v(i', j');
 2  Δ^a ← 0;
 3  Para_todo j'' ∈ Γ_j e j'' ≠ j' faça
 4      i'' ← A_S^{-1}(j'');
 5      Se i'' ∈ Γ_i então
 6          Δ^a ← Δ^a + 1 − 2 · s^a((i, i''), (j, j''));
 7      fim_se
 8      Se i'' ∈ Γ_{i'} então
 9          Δ^a ← Δ^a − 1 + 2 · s^a((i', i''), (j, j''));
10      fim_se
11  fim_para_todo
12  Para_todo j'' ∈ Γ_{j'} e j'' ≠ j faça
13      i'' ← A_S^{-1}(j'');
14      Se i'' ∈ Γ_{i'} então
15          Δ^a ← Δ^a + 1 − 2 · s^a((i', i''), (j', j''));
16      fim_se
17      Se i'' ∈ Γ_i então
18          Δ^a ← Δ^a − 1 + 2 · s^a((i, i''), (j', j''));
19      fim_se
20  fim_para_todo
fim
```

Pseudo–código 8: Cálculo do valor da troca de associações.

O teste da linha 10 verifica se a troca em questão mantém a viabilidade da solução gerada. O refinamento da chamada da linha 11 é apresentado no Pseudo-código 8 e efetua os cálculos da variação na função objetivo para cada troca. Caso esta variação seja positiva, as linhas 14 a 19 realizam a atualização da solução atual.

A complexidade do procedimento de busca de uma solução na vizinhança V_b é $O(|V_I|^2 \cdot |E_I|)$. Existem no máximo $O(|V_I|^2)$ pares de elementos de V_I associados a diferentes elementos de V_M. Para cada par, são executados os cálculos da variação na função objetivo que são da ordem de $|V_I|$. Como o teste de viabilidade da troca de cada associação é $O(|E_I| + |V_I|)$, os cálculos de contribuição e testes de viabilidade para todas as associações custam $O(|V_I|^2 \cdot |E_I|)$.

A busca local na vizinhança V_b apresenta uma complexidade maior que a busca na vizinhança V_a, que é $O(|V_M| \cdot |V_I|^2 + |E_I| \cdot |V_I|)$. Esta diferença entre as complexidades justifica a utilização da busca na vizinhança V_b apenas como uma forma de se desprender dos ótimos locais para a vizinhança V_a.

4.3 GRASP

A metaheurística GRASP (*Greedy Randomized Adaptive Search Procedure*) foi originalmente proposta por Feo e Resende [24]. Seu princípio básico é a combinação de uma fase de construção com uma busca local em um processo iterativo [55]. Resende e Ribeiro [56] sugerem que o método utilizado na fase de construção baseie-se em uma função gulosa para selecionar cada elemento a ser incorporado à solução em construção, utilizando uma escolha aleatória. A cada passo um novo elemento é inserido na solução até que ela esteja completa. A função gulosa privilegia elementos cuja inserção na solução acarretem um maior ganho no valor da função objetivo (problema de maximização). A escolha probabilística apresenta um certo grau de aleatoriedade que possibilita uma maior diversidade das soluções construídas.

O algoritmo GRASP fornece como resultado o melhor dentre todos os ótimos locais encontrados pelo procedimento de busca local, cada um obtido a partir de uma diferente solução inicial construída pelo algoritmo construtivo guloso aleatorizado. Estudos sobre a construção, o desempenho e aplicações da metaheurística GRASP podem ser encontrados em [27, 30, 56], entre outras referências.

Uma forma de se implementar um algoritmo construtivo guloso aleatorizado é criar-se uma lista restrita de candidatos (LRC), selecionados com base na função gulosa, e a cada passo escolher-se aleatoriamente um deles para se incluir na solução. A função gulosa deve ser adaptada com base na decisão do elemento que foi incluído. A LRC pode ser definida por um número fixo de elementos ou ser especificada por uma medida de qualidade que determina se um elemento pertence a ela ou não. Experimentos computacionais mostram que a qualidade média da solução gerada pelo algoritmo depende da qualidade dos elementos da lista de candidatos e da diversidade das soluções construídas, que está diretamente relacionada à cardinalidade da lista [55].

Em geral, algoritmos GRASP possuem poucos parâmetros a serem ajustados. Os mais comuns são a restritividade da lista de candidatos (se utilizada) e o número de iterações. Este pequeno número de parâmetros ajustáveis facilita a avaliação e eventual correção de insucessos. Em contrapartida, uma desvantagem é o fato de tais algoritmos não utilizarem algum

tipo de memória das informações coletadas durante a busca.

O Pseudo-código 9 apresenta uma implementação básica dessa metaheurística para o PCIG. A chamada da linha 3 representa a fase de construção, na qual um dos algoritmos apresentados na Seção 5.3 deve ser utilizado. O procedimento de busca local referenciado na linha 4 é $\texttt{BL}_{a+b}$, descrito na Seção 4.2.3.

```
algoritmo GRASP_Básico(Semente, MaxIter)
   1 f(S*) ← −∞
   2 Para iter de 1 até MaxIter faça
   3     S ← Alg_Construtivo(MaxTentativas, MaxSolucoes,
     Semente);
   4     S ← BL_{a+b}(S);
   5     Se f(S) > f(S*) então
   6         S* ← S;
   7     fim_se
   8 fim_para
   9 Retorna (S*);
fim
```

Pseudo–código 9: GRASP básico.

Diversas aplicações de procedimentos GRASP são encontradas na literatura em diversas áreas, como por exemplo em problemas de: roteamento [7, 9, 10, 17], recobrimento e particionamento [5, 6, 24, 34, 38], localização [1, 22], escalonamento [12, 25, 26, 28], transporte [7, 25, 27] e telecomunicações [2, 8, 57]. Outras aplicações podem ser encontradas em [29].

4.3.1 GRASP com reconexão por caminhos

Uma alternativa para a inclusão de memória em procedimentos GRASP é a técnica de reconexão por caminhos. Esta técnica foi inicialmente proposta por Glover [35] (como é citado em [56]), como uma forma de intensificação num processo de busca tabu. Ela consiste em explorar trajetórias no espaço de busca que conectam a solução corrente a soluções de elite. Começando-se de uma ou mais soluções, explora-se os caminhos que levam a outras soluções de elite em busca de melhorias. Para a geração dos caminhos, movimentos são selecionados para introduzir, na solução atual, atributos que estão presentes na solução de elite. O procedimento de reconexão por caminhos pode ser visto como uma busca por atributos presentes em boas

soluções para serem incorporados a uma outra solução. A solução de elite, cujos atributos serão inseridos na solução corrente, é também denominada solução guia.

A técnica de reconexão por caminhos já foi utilizada com sucesso em conjunto com GRASP para uma variedade de aplicações, tais como problemas de atribuição quadrática [47], atribuição com três índices [4], roteamento de circuitos privativos em redes de comunicações [57], problema de projeto de redes de 2-caminhos [59], problema das p-medianas [54] e problema de Steiner em grafos [60]. Diversas alternativas já foram consideradas e combinadas em recentes implementações de reconexão por caminhos [56]. Neste trabalho, ela é utilizada como uma forma de melhoria para cada solução encontrada pela busca local.

Na combinação do procedimento de reconexão por caminhos com um algoritmo GRASP, é necessária a manutenção de uma lista de soluções de elite, que guarda as melhores soluções encontradas. Ao final de cada iteração, é realizada uma busca no caminho entre a solução encontrada pela busca local e uma solução escolhida aleatoriamente da lista de elite. Uma das soluções deve ser escolhida como guia, sendo a outra portanto a solução de partida para a reconexão. Esta é feita analisando-se cada movimento possível para a solução corrente em direção à guia e escolhendo-se a cada passo o melhor dentre eles. Uma lista com os possíveis movimentos é obtida através do cálculo da diferença simétrica entre as duas soluções.

Diversas são as abordagens de reconexão por caminhos encontradas na literatura. Resende e Ribeiro [56] descrevem algumas dessas abordagens:

- reconexão periódica: a reconexão por caminhos não é aplicada sistematicamente, mas apenas periodicamente;
- reconexão *forward*: a solução escolhida como guia é a melhor entre a duas selecionadas para a reconexão;
- reconexão *backward*: a solução escolhida como guia é a pior entre a duas selecionadas para a reconexão;
- reconexão *back and forward*: são exploradas as duas trajetórias separadamente, primeiro no sentido *forward* e depois no *backward*;
- reconexão mista: são exploradas simultaneamente as duas trajetórias (*forward* e *backward*), até que se encontrem em uma solução intermediária equidistante das duas soluções iniciais;

– reconexão aleatorizada: ao invés de se selecionar o melhor dentre os movimentos disponíveis, escolhe-se um aleatoriamente de uma lista com os mais promissores;
– reconexão truncada: é explorada apenas parte da trajetória entre as duas soluções selecionadas.

O Pseudo-código 10 ilustra uma heurística GRASP para o PCIG com utilização de reconexão por caminhos ao final de cada iteração. Os passos das linhas 1 a 4 são idênticos aos do algoritmo GRASP básico. Na linha 5, verifica-se se a solução encontrada pela busca local é melhor que a melhor solução até o momento. Em caso afirmativo, na linha 6 é feita a atualização da melhor solução e ela é inserida na lista de soluções de elite (linha 7). Caso contrário, o teste da linha 8 verifica se a solução S_1 deve entrar na lista de elite.

A função `insereSolucaoElite`() tem como objetivo inserir uma solução na lista de elite, desde que esta solução ainda não esteja em tal lista. Ou seja, a solução passada somente é inserida caso ela ainda não pertença à lista. A chamada `f_PiorSolElite()` retorna o valor da função objetivo para a pior solução da lista de elite, se a lista estiver cheia. Se a lista ainda não foi preenchida completamente a chamada retorna $-\infty$, de forma que, enquanto a lista tiver capacidade, todas as diferentes soluções encontradas pela busca local são inseridas nesta lista.

O teste da linha 11 evita que se execute a reconexão por caminhos na primeira iteração, uma vez que apenas uma solução foi obtida até o momento. Na linha 12, é selecionada aleatoriamente uma solução da lista de elite para a reconexão por caminhos.

Na linha 13 são calculadas as diferenças entre as duas soluções selecionadas (dif), onde S_1 é a solução obtida pela busca local e S_2 a solução de elite. Para o PCIG, estas diferenças podem ser representadas por um conjunto com as associações presentes na solução guia e ausentes na solução corrente.

Na linha 14 é então executado o procedimento de reconexão por caminhos no sentido da solução da busca local para a solução de elite (em geral, sentido *forward*). Nas linhas 15 a 20 são feitas as atualizações necessárias, inclusive a inserção da nova solução na lista de elite, se for o caso.

Um procedimento análogo é realizado nas linhas 22 a 28, agora no sentido inverso, ou seja, a partir da solução de elite para a solução encontrada pela busca local (em geral, sentido *backward*). Para isso, na linha 21 as diferenças entre as soluções devem ser recalculadas, pois ocorre a inversão de papéis entre as soluções guia e inicial.

```
algoritmo GRASP_RC(Semente, MaxIter)
 1 f(S*) ← −∞
 2 Para iter de 1 até MaxIter faça
 3    S1 ← Alg_Construtivo(MaxTentativas, MaxSolucoes,
   Semente);
 4    S1 ← BL_{a+b}(S1);
 5    Se f(S1) > f(S*) então
 6       S* ← S1;
 7       insereSolucaoElite(S1);
 8    Senão se f(S1) > f_PiorSolElite() então
 9       insereSolucaoElite(S1);
10    fim_se
11    Se iter > 1 então
12       S2 ← obtemSolucaoElite();
13       CalculaDiferencas(S1, S2, dif);
14       S1 ← RC(S1, dif);
15       Se f(S1) > f(S*) então
16          S* ← S1;
17          insereSolucaoElite(S1);
18       Senão se f(S1) > f_PiorSolElite() então
19          insereSolucaoElite(S1);
20       fim_se
21       CalculaDiferencas(S2, S1, dif);
22       S1 ← RC(S2, dif);
23       Se f(S1) > f(S*) então
24          S* ← S1;
25          insereSolucaoElite(S1);
26       Senão se f(S1) > f_PiorSolElite() então
27          insereSolucaoElite(S1);
28       fim_se
29    fim_se
30 fim_para
31 Retorna (S*);
fim
```

Pseudo–código 10: GRASP com reconexão por caminhos.

O procedimento de reconexão por caminhos, referenciado nas linhas 14 e 22 do algoritmo `GRASP_RC`, aparece refinado no Pseudo-código 11. Este algoritmo, aplicado ao PCIG, se assemelha muito à busca local na vizinhança V_a, apresentada anteriormente. A diferença consiste no fato

de que, ao contrário da busca local, ele não considera todas as trocas de associações possíveis para um dado vértice. Seja o vértice j de G_I associado a um elemento i de G_M na solução resultante da busca local e associado ao elemento $i^{'}$ na solução de elite considerada. Se $i \neq i^{'}$ esta troca de associação é a única considerada para o elemento j. Uma vez realizada, esta troca de associação deixa a solução da busca local mais parecida com a solução de elite. Além disso, ao contrário da busca local na vizinhança V_a, o procedimento de reconexão por caminhos não se prende aos possíveis ótimos locais no caminho entre as duas soluções consideradas.

O laço iniciado na linha 3 investiga todas as associações presentes na solução guia e ausentes na solução corrente. Nas linhas 4 a 10, para cada vértice j de V_I cuja associação na solução corrente seja diferente daquela na solução guia, calcula-se o custo da troca de associação e verifica-se se tal troca mantém a viabilidade da solução. Nas linhas 11 a 15, escolhe-se a melhor troca, caso alguma seja possível; caso contrário termina-se o procedimento pela inexistência de um movimento que mantenha a viabilidade da solução. As linhas 16 a 24 efetuam a troca escolhida e atualizam, se necessário, a melhor solução encontrada na busca.

O laço com inicio na linha 4 realiza no pior caso $|V_I|$ iterações, assim como o laço iniciado na linha 3. Logo, no total, o laço da linha 4 pode ser executado até $|V_I|^2$ vezes. Os cálculos do procedimento chamado na linha 6 (refinados no Pseudo-código 12) são análogos aos da busca local V_a e têm complexidade $O(|V_I|)$. Como a complexidade do teste de viabilidade é $O(|E_I| + |V_I|)$, a complexidade final do algoritmo é $O(|V_I|^2 \cdot |E_I|)$.

4.3.2

GRASP com reconexão por caminhos modificada

O procedimento de reconexão por caminhos proposto na Seção 4.3.1 apresenta uma limitação, de forma semelhante à busca local na vizinhança V_a. Pelos mesmos motivos apresentados anteriormente, boa parte dos movimentos considerados na diferença simétrica entre as soluções são inviáveis, o que reduz consideravelmente as chances de sucesso da reconexão por caminhos em encontrar soluções aprimorantes. Uma alternativa para contornar essa limitação é a aceitação de movimentos que causem uma inviabilidade temporária da solução. A lógica desta medida é o fato de que movimentos que sozinhos levem à inviabilidade, podem, em conjunto, recuperar a viabi-

```
algoritmo RC(SolucaoPR, dif)
 1  S ← SolucaoPR;
 2  S* ← S;
 3  Enquanto dif ≠ ∅ faça
 4     Para_todo (k, j) ∈ dif faça
 5        Se trocaViavel?(A_S^{-1}(j) = k) então
 6           CalculaDeltas(k, j)
 7        Senão
 8           Δ[j] ← −∞;
 9        fim_se
10     fim_para_todo
11     Se max{Δ[j], j ∈ V_I} > −∞ então
12        best ← argmax{Δ[j], j ∈ V_I}
13     Senão
14        Retorna S*
15     fim_se
16     i ← A_S^{-1}(best);
17     k ← A_dif^{-1}(best) ;
18     dif ← dif − {(k, best)};
19     A_S^{-1}(best) ← k;
20     A_S(i) ← A_S(i) − {best};
21     A_S(k) ← A_S(k) ∪ {best};
22     Se f(S) > f(S*) então
23        S* ← S;
24     fim_se
25  fim_enquanto
26  Retorna S*
fim
```

Pseudo–código 11: Procedimento de reconexão por caminhos.

```
algoritmo CalculaDeltas(k, j)
 1  i ← A_S^{-1}(j)
 2  Δ^v ← 2 · s^v(k, j) − 2 · s^v(i, j);
 3  Δ^a ← 0;
 4  Para_todo j' ∈ Γ_j faça
 5     i' ← A_S^{-1}(j');
 6     Se i' ∈ Γ_i então
 7        Δ^a ← Δ^a + 1 − 2 · s^a((i, i'), (j, j'));
 8     fim_se
 9     Se i' ∈ Γ_k então
10        Δ^a ← Δ^a − 1 + 2 · s^a((k, i'), (j, j'));
11     fim_se
12  fim_para_todo
13  Δ[j] ← α · Δ^v/(|V_M| · |V_I|) + (1 − α) · Δ^a/(|E_M| · |E_I|);
fim
```

Pseudo–código 12: Procedimento de cálculo dos valores de Δ.

lidade da solução, aumentando as possibilidades de sucesso do algoritmo.

A implementação alternativa do procedimento de reconexão por caminhos é apresentada no Pseudo-código 13 e pode ser utilizado, em substituição ao procedimento mostrado no Pseudo-código 11, em conjunto com o GRASP. O trecho de código das linhas 9 a 20 representa a modificação do algoritmo de reconexão por caminhos. Seu objetivo é buscar inicialmente o melhor movimento viável, se existir algum (linhas 10 a 14). Caso nenhum movimento seja viável, aceita-se o melhor dos inviáveis (linhas 15 a 20).

4.4 Discussão

Este capítulo propôs algoritmos construtivos gulosos aleatorizados para o PCIG, baseados no algoritmo construtivo desenvolvido por Boeres [14]. Na implementação desses algoritmos, foram projetadas novas estruturas de dados e modificados detalhes em relação à implementação do método anterior. O algoritmo `Construtivo_2` utiliza uma função gulosa na escolha da associação para cada nó do grafo G_I. O algoritmo `Construtivo_2a` incorporou um grau de aleatoriedade na escolha da associação para cada vértice de V_I, sem deixar de considerar a função gulosa. O algoritmo `Construtivo_3` introduziu a utilização de uma lista de candidatos com as melhores associações possíveis. A cada passo, é efetuada uma associação, selecionada aleatoriamente da lista. Os algoritmos `Construtivo_2a` e `Construtivo_3` foram implementados com o objetivo de se verificar se uma aleatorização da escolha puramente gulosa pode levar a alguma melhora em termos de qualidade das soluções construídas.

O métodos de busca local utilizados neste trabalho são os mesmos daqueles apresentados em [14], salvo alguns detalhes de implementação. Novas estruturas de dados e formas mais eficientes para realização de alguns cálculos são implementados, por exemplo a atualização dos conjuntos $C(j)$ quando uma nova associação é feita na busca local $\texttt{BL}_a$. Neste caso, são considerados apenas os conjuntos passíveis de alteração.

O algoritmo GRASP proposto neste trabalho combina um dos métodos construtivos apresentados com uma busca local, além de um procedimento de reconexão por caminhos. Foram projetadas duas variações de reconexão por caminhos, sendo a primeira baseada na versão proposta em

```
algoritmo RC+(SolucaoPR, dif)
 1  S ← SolucaoPR; S* ← S;
 2  Enquanto dif ≠ ∅ faça
 3     Δ[j] ← −∞, ∀j ∈ V_I;
 4     feasible[j] ← falso, ∀j ∈ V_I;
 5     Para_todo (j, k) ∈ dif faça
 6        CalculaDeltas(j, k)
 7        feasible[j] ← trocaViavel?(A_S^{-1}(j) = k);
 8     fim_para_todo
 9     best ← −1; Δ_best ← −∞;
10     Para j de 0 a |V_I| faça
11        Se Δ[j] > Δ_best e feasible[j] =verdadeiro então
12           Δ_best ← Δ[j]; best ← j;
13        fim_se
14     fim_para
15     Se best = −1 então
16        i ← argmax{Δ[k], k ∈ V_I};
17        Se Δ[i] > −∞ então
18           best ← i;
19        fim_se
20     fim_se
21     Se best ≠ −1 então
22        i ← A_S^{-1}(best);
23        k ← A_dif^{-1}(best) ;
24        dif ← dif − {(k, best)};
25        A_S^{-1}(best) ← k;
26        A_S(i) ← A_S(i) − {best};
27        A_S(k) ← A_S(k) ∪ {best};
28        Se f(S) > f(S*) e Viavel?(S) então
29           S* ← S;
30        fim_se
31     fim_se
32  fim_enquanto
33  Retorna S*
fim
```

Pseudo–código 13: Procedimento de reconexão por caminhos modificado.

[55] e a segunda apresentando uma modificação com o objetivo de aceitar soluções inviáveis no caminho entre as soluções de elite. Esta modificação foi proposta visando diminuir as possibilidades de término prematuro do algoritmo por não encontrar um movimento viável.

Estes algoritmos heurísticos podem constituir uma ferramenta para a resolução aproximada do PCIG, bem como serem utilizados para o cálculo de um limite inferior para o problema, atuando em conjunto com a solução do modelo exato de programação inteira apresentado no Capítulo 3.