2
GRASP

2.1
Introdução

Um problema de otimização combinatória pode ser definido por um conjunto base $E = \{1, \ldots, n\}$, um conjunto de soluções viáveis $X \subseteq 2^{|E|}$ e uma função objetivo $f : 2^{|E|} \rightarrow \mathbb{R}$ tal que $f(x) = \sum_{e \in x} c(e)$, $\forall x \in 2^{|E|}$, onde $c(e)$ é o custo associado à inclusão do elemento $e \in E$ na solução x. A solução ótima de um problema de minimização $x^* \in X$, 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) [43, 45, 48] é um processo iterativo caracterizado por partidas múltiplas, onde cada iteração consiste de duas fases. Na fase construtiva, uma solução viável é produzida, enquanto que, na fase de busca local, um ótimo local na vizinhança da solução construída é procurado. A melhor solução obtida é armazenada como resultado.

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 $g : C \rightarrow \mathbb{R}$. Essa função mede o benefício (guloso) 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*

```
procedimento CONSTRUTIVA(semente)
1     x = ∅;
2     Inicialize o conjunto de candidatos C;
3     enquanto C ≠ ∅ faça
4           Construa LRC;
5           i=rand(semente);
6           Selecione s_i de LRC;
7           x = x ∪ {s_i};
8           Atualize o conjunto de candidatos C;
9     fim_enquanto;
10    retorne (x);
fim CONSTRUTIVA
```

Figura 2.1: Algoritmo semi-guloso usado na fase construtiva.

(LRC). A fase construtiva é semelhante à heurística semi-gulosa proposta independentemente por Hart e Shogan [68], que consiste em uma estratégia de partidas múltiplas baseada em construções gulosas randomizadas, porém sem o uso de busca local. A Figura 2.1 mostra o pseudo-código do procedimento semi-guloso usado na fase construtiva.

Na maior parte dos casos, é possivel 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 $x^1, x^2, \ldots, x^k$. Dada uma solução $x \in X$, os elementos da sua vizinhança $N(x)$ são soluções que podem ser obtidas através da aplicação de uma modificação elementar em x, chamada de *movimento*. A notação de *movimentos de trocas-(p,q)* será usada nesse capítulo e nos seguintes, indicando que dada uma solução x, a sua vizinhança será gerada removendo-se p elementos de x e inserindo-se q novos elementos em x. Por exemplo, seja $x = (1, 2, 3)$ e seja a vizinhança $N(x)$ gerada por movimentos de trocas-(2,2) aplicados a um vetor de permutação. Soluções na vizinhança são geradas pela remoção de dois elementos da solução corrente, onde cada elemento é dado pelo par valor/posição no vetor de permutações e pela inserção de dois novos elementos na solução. Nesse caso, os movimentos são gerados pela troca na posição do vetor de permutações de valores escolhidos dois a dois. Tem-se que a vizinhança de x é dada por $N(x) = \{(3, 2, 1), (2, 1, 3), (1, 3, 2)\}$. 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

```
procedimento BUSCA_LOCAL(x^0)
1     Inicialize x = x^0;
2     enquanto existe x' ∈ N(x) tal que f(x') < f(x) faça
3           Selecione x' ∈ N(x) tal que f(x') < f(x);
4           x = x';
5     fim_enquanto;
6     retorne (x);
fim BUSCA_LOCAL
```

Figura 2.2: Procedimento básico de busca local (para um problema de minimização).

busca local depende de vários fatores, tais como a estrutura da vizinhança, a função que deve ser otimizada e a solução inicial. Na Figura 2.2 um procedimento de busca local é mostrado em pseudo-código.

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 possivel 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 aleatoriamente 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 ou até mesmo exponencial no tamanho do problema [71]. Por outro lado, algoritmos gulosos geralmente produzem soluções de melhor qualidade do que as geradas aleatoriamente. O uso de soluções gulosas como ponto de partida para uma busca local, em geral, levará a boas soluções, porém, sub-ótimas, isto é, soluções de qualidade inferior à qualidade dos ótimos 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 semiguloso [43, 68] para produzir as soluções iniciais usadas em cada iteração.

```
procedimento GRASP(max_iter, semente)
1     f* = ∞;
2     para iter = 0, ..., max_iter faça
3           x = CONSTRUTIVA(semente);
4           x' = BUSCA_LOCAL(x);
5           se f(x') < f* então
6                 x* = x';
7                 f* = f(x*);
8           fim-se;
9     fim-para;
10    retorne (x*);
fim GRASP
```

Figura 2.3: Algoritmo GRASP básico.

Na Figura 2.3 é mostrado um algoritmo GRASP básico para problemas de minimização.

A seguir, serão discutidos alguns aspectos importantes no projeto de uma heurística GRASP, tais como: a forma como a lista restrita de candidatos (LRC) é construída, a introdução de memória de longo prazo, a hibridização com outras metaheurísticas e a paralelização.

2.2 A lista restrita de candidatos

Existem duas estratégias básicas usadas para construir a LRC. A primeira delas é um esquema baseado em *cardinalidade*, onde para um valor inteiro k pré-estabelecido, coloca-se na LRC os k melhores elementos da lista de candidatos. A outra abordagem é um esquema baseado no *valor* associado a cada elemento candidato. Sejam $\bar{g} = \max\{g(c)|c \in C\}$, $\underline{g} = \min\{g(c)|c \in C\}$ e um parâmetro $\alpha \in [0,1]$. Em um problema de minimização, uma LRC baseada em valores será determinada por LRC $= \{c \in C \mid g(c) \leq \underline{g} + \alpha(\bar{g} - \underline{g})\}$. No caso $\alpha = 0$, o algoritmo semi-guloso corresponde ao algoritmo guloso puro, enquanto que para $\alpha = 1$ são construídas soluções aleatórias. Analogamente, em um esquema baseado em cardinalidade, no caso k=1 o algoritmo semi-guloso é transformado em um algoritmo guloso, enquanto que para $k = |C|$ são construídas soluções aleatórias. Dessa forma, os parâmetros α e k determinam se a fase construtiva da metaheurística GRASP será mais semelhante a um algoritmo de construção aleatória ou a um algoritmo guloso.

A maior parte das implementações de GRASP encontradas na literatura usam algum tipo de LRC baseada no valor da função gulosa dos

elementos candidatos [99]. As primeiras implementações do método usavam valores fixos para o α, que eram determinados através de experimentos. Para cada problema tratado ou até mesmo para cada classe de problemas testes, um valor diferente de α era estabelecido. Porém, de acordo com [85], a fase construtiva de um algoritmo GRASP com $\alpha < 1$ fixo pode nunca gerar uma solução na bacia de atração de um ótimo global. Caso isso ocorra, o algoritmo não convergirá assintoticamente. Para evitar esse problema, Resende et al. [112] propuseram o uso de um α gerado aleatoriamente no intervalo $[0, 1]$ em cada iteração do algoritmo. No restante dessa tese, serão considerados apenas algoritmos GRASP onde a LRC é baseada no valor da função gulosa.

De acordo com [105], a metaheurística GRASP pode ser vista como uma técnica de amostragem repetitiva. Uma solução produzida em uma iteração do método é uma amostra de alguma distribuição desconhecida, onde a média e a variância são funções da natureza restritiva da LRC. Por exemplo, caso a LRC seja formada por apenas um elemento, apenas uma solução será produzida e a variância e a média da distribuição serão, respectivamente, 0 e o valor da solução gulosa. Caso a LRC seja formada por um número maior de elementos, então várias soluções serão produzidas e a variância da distribuição será maior do que no caso anterior. Nesse caso, a média dos custos das soluções obtidas deverá ser de qualidade inferior à solução gulosa. Porém, a melhor solução obtida deverá ser de qualidade superior à solução obtida pelo algoritmo guloso.

Resende e Ribeiro [105] estudaram o comportamento de um algoritmo GRASP em relação à diversidade de soluções, à qualidade de soluções e ao tempo computacional, em função do parâmetro α, que controla a restritividade da LRC. Concluíram que, quanto maior a variância dos valores das soluções obtidas durante a fase construtiva, maior será a variância das soluções obtidas após a busca local. A probabilidade de um algoritmo GRASP encontrar uma solução ótima é maior quando a variância das soluções construídas é grande. Porém, nesse caso, como a diferença entre o valor médio das soluções construídas e o valor da melhor solução é grande, a busca local precisará, em média, de um maior tempo computacional. Conseqüentemente, o tempo computacional do algoritmo será maior quando a variância dos custos das soluções produzidas na fase construtiva for alta. Verificou-se que, muitas vezes, é possível gerar várias soluções do método GRASP no mesmo espaço de tempo necessário para que o procedimento de busca local convirja de uma única solução gerada aleatoriamente. Conclui-se, então, que a escolha correta do parâmetro α é crítica para se obter um

bom equilíbrio entre tempo computacional e qualidade de solução.

2.3 Mecanismos alternativos usados na fase construtiva

Nessa seção serão discutidos melhoramentos e técnicas alternativas aplicadas à fase construtiva da metaheurística GRASP. Essas técnicas incluem: o procedimento de GRASP Reativo, perturbações de custo usadas ao invés de seleção aleatória, funções usadas para guiar a escolha de elementos, memória e aprendizado, e busca local aplicada a soluções parcialmente construídas.

2.3.1 GRASP Reativo

Em um procedimento do tipo *GRASP Reativo* [101, 102], o parâmetro α é ajustado em tempo de execução de acordo com informações de soluções visitadas anteriormente. Em cada iteração do algoritmo, o valor do parâmetro α é escolhido aleatoriamente. Porém, ao invés de escolher esse parâmetro de uma distribuição uniforme, ele é escolhido a partir de um conjunto de valores discretos $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. A cada α_k associa-se uma probabilidade de escolha $p(\alpha_k)$. Essas probabilidades são inicialmente idênticas, isto é, $p(\alpha_i) = 1/m, i = 1, \ldots, m$. O procedimento de GRASP Reativo atualiza em tempo de execução as probabilidades $\{p(\alpha_1), p(\alpha_2), \ldots, p(\alpha_m)\}$ para favorecer valores de α que tenham produzido soluções de qualidade. Seja f^* o valor da melhor solução encontrada até o momento e a_i a média dos valores das soluções encontradas usando α_i na fase construtiva. Valores $q_i = f^*/a_i$ e $p(\alpha_i) = q_i / \sum_{j=1}^{m} q_j$ são computados periodicamente pelo algoritmo. Isso permite que valores de α_i que tenham produzido soluções boas nas iterações anteriores, tenham uma maior probabilidade de serem escolhidos.

Um procedimento de GRASP Reativo pode ser visto como uma estratégia que usa uma memória de longo prazo para escolher valores do parâmetro α da LRC. Em [18, 40, 100, 101, 102] observou-se que o procedimento de GRASP Reativo permitiu melhorar o algoritmo GRASP básico em termos de robustez e de qualidade de solução.

2.3.2 Funções usadas para guiar a escolha de elementos

Durante a fase construtiva de um algoritmo GRASP básico, o próximo elemento a ser inserido na solução é selecionado aleatoriamente entre os elementos candidatos na LRC. Cada elemento da LRC possui a mesma probabilidade de ser selecionado. Bresina propõe em [20] uma abordagem para a construção da LRC, onde a cada elemento da lista de candidatos é atribuída uma probabilidade de seleção. Primeiramente, o valor da função gulosa $g(c)$ de cada elemento $c \in C$ é calculado e os elementos são ordenados de acordo com esse valor. Seja $rank(c)$ a posição do elemento c, após a ordenação dos elementos de C. A cada elemento c da lista ordenada é associado um valor dado por uma função $\texttt{bias}(rank(c))$. A probabilidade de seleção de um elemento c da lista de candidatos é então determinada por:

$$p(c) = \texttt{bias}(rank(c)) / \sum_{j=1}^{|C|} \texttt{bias}(rank(j)).$$

Várias funções podem ser usadas para guiar a escolha de elementos:

1. logarítmica: $\texttt{bias}(rank(c)) = 1/\log(rank(c)+1)$, para $c \in$ LRC;
2. linear: $\texttt{bias}(rank(c)) = 1/rank(c)$, para $c \in$ LRC;
3. polinomial em n: $\texttt{bias}(rank(c)) = 1/rank(c)^n$, para $c \in$ LRC;
4. exponencial: $\texttt{bias}(rank(c)) = 1/e^{rank(c)}$, para $c \in$ LRC;
5. aleatória: $\texttt{bias}(rank(c)) = 1$, para $c \in$ LRC.

Em [16] é usada uma abordagem semelhante, onde a atribuição de probabilidades é limitada aos elementos da LRC. Deve-se notar que o algoritmo GRASP básico usa uma função aleatória para guiar a escolha de elementos na fase construtiva.

2.3.3 Princípio da Proximidade da Otimalidade

O Princípio da Proximidade da Otimalidade, ou POP (*Proximate Optimality Principle*) foi proposto em [58] para a busca tabu. Segundo esse princípio, "soluções boas em um nível possivelmente estarão 'próximas' a soluções boas em um nível adjacente". Em [49], Fleurent e Glover adaptaram esse conceito ao contexto da metaheurística GRASP. A abordagem proposta por Fleurent e Glover consiste em, durante a fase construtiva, aplicar um

procedimento de busca local à solução parcial, com o objetivo de reduzir imperfeições que possivelmente tenham sido incorporadas à solução. O POP não deve ser aplicado à solução parcial a cada passo da fase construtiva para preservar a variabilidade de soluções iniciais e também devido ao seu alto custo computacional. O POP é usado em [16] em uma abordagem GRASP para o problema de escalonamento de tarefas JSP.

2.3.4 Memória e aprendizado

Uma das possíveis deficiências do algoritmo GRASP básico é o fato de que suas iterações não usam informações colhidas nas iterações anteriores. Em cada iteração do algoritmo, são armazenadas apenas a solução corrente e a melhor solução encontrada até o momento.

Uma abordagem para introdução de uma memória de longo prazo na fase construtiva de métodos de partidas múltiplas foi proposta por Fleurent e Glover [49]. A memória de longo prazo é usada para modificar as probabilidades de escolha de cada elemento da LRC. Para isso, um conjunto P de soluções de elite é mantido para ser usado na fase construtiva. Um solução candidata S entrará para o conjunto de elites P, caso ela seja melhor do que a melhor solução em P, ou caso ela seja melhor do que a pior solução em P e seja suficientemente diferente de todas as outras soluções em P.

Seja $I(c)$ uma *função de intensificação* que mede o número de vezes em que um atributo c aparece nas soluções do conjunto de elite. Dessa forma, a inserção de um atributo c em uma solução S, resultará em um valor alto de $I(c)$, caso essa inserção torne S mais semelhante às soluções no conjunto de elite P. A função de intensificação $I(c)$ é usada na fase construtiva da seguinte forma. Seja $E(c) = F(g(c), I(c))$ uma função das funções gulosa e de intensificação. Por exemplo, $E(c) = \lambda g(c) + I(c)$, onde λ é um parâmetro do algoritmo. O esquema de intensificação proposto incentiva a seleção de elementos da LRC que possuam valores de $E(c)$ altos. Para isso, a cada elemento c da LRC são atribuídas probabilidades de escolha dadas por $p(c) = E(c)/\sum_{j \in \mathrm{LRC}} E(j)$. A função $E(c)$ pode variar com o tempo, alterando-se o valor de λ para favorecer mais a parcela gulosa ou a parcela de intensificação dessa função. Procedimentos para variar o valor de λ são dados por Fleurent e Glover [49] e por Binato et al. [16]. Essa estratégia é aplicada ao problema de escalonamento de tarefas JSP em [16].

2.3.5 Perturbações de custo

A estratégia de introdução de ruído nos custos originais de um problema é semelhante à abordagem chamada de *método do ruído*, introduzida por Charon e Hudry [31, 32]. Essa abordagem é útil em casos em que o algoritmo de construção não é muito sensível à randomização, e em casos em que não existe um algoritmo guloso para ser randomizado.

Em Canuto et al. [27] foi proposto um algoritmo GRASP híbrido para o problema da árvore de Steiner com prêmios, que usa o algoritmo primal-dual de Goemans e Williamson [61] para construir soluções iniciais. Uma nova solução é construída a cada iteração perturbando-se os prêmios dos nós de acordo com a estrutura da solução corrente. Dois esquemas de perturbação de custos são usados:

1. *Perturbação por eliminação*: para permitir uma diversificação na busca, nós persistentes, isto é, nós que estão presentes na solução construída na iteração anterior e que permanecem na solução após a busca local, são proibidos de fazer parte da solução corrente. Isso é obtido alterando-se para zero os prêmios desses nós. Um parâmetro θ controla a fração de nós persistentes que terão seus prêmios temporariamente alterados.

2. *Perturbação por mudança de prêmios*: ruídos são introduzidos nos prêmios dos nós de forma semelhante ao que é proposto em [31, 32] para alterar o valor da função objetivo. Para cada nó i, é gerado um fator de perturbação $\beta(i)$ no intervalo [1-γ, 1+γ], onde γ é um parâmetro da implementação. O prêmio associado ao nó i é temporariamente alterado para $\pi'(i) = \pi(i) \cdot \beta(i)$, onde $\pi(i)$ é o prêmio original.

Em Ribeiro et al. [117] são introduzidos em uma abordagem GRASP para o problema de Steiner, métodos de perturbação de custo que incorporam os mecanismos de intensificação e diversificação, originalmente propostos no contexto da busca tabu. São usadas três estratégias de perturbação de custos, chamadas de *D, I* e *U*. Os pesos originais, isto é, sem perturbação, são usados nas três primeiras iterações do algoritmo, combinados com três heurísticas de construção. As três estratégias de perturbação de custo são usadas de forma cíclica nas iterações seguintes. Seja w_e o peso de uma aresta e. A cada iteração i, o peso modificado w_e^i de cada aresta e é selecionado aleatoriamente de uma distribuição uniforme, entre w_e e $r_i(e) \cdot w_e$, onde o

coeficiente $r_i(e)$ depende da estratégia de perturbação de custo aplicada na iteração i. Seja $t_{i-1}(e)$ o número de ótimos locais onde a aresta e aparece após $i-1$ iterações do algoritmo GRASP. As estratégias de perturbação de custo *D, I* e *U* são descritas a seguir:

1. *D* é uma estratégia de diversificação, com $r_i(e) = 1.25 + 0.75 \cdot t_{i-1}(e)/(i-1)$. Portanto, valores elevados de $r_i(e)$ são atribuídos a arestas que aparecem mais freqüentemente nos ótimos locais encontrados anteriormente.

2. *I* é uma estratégia de intensificação, com $r_i(e) = 2 - 0.75 \cdot t_{i-1}(e)/(i-1)$. Portanto, arestas que aparecem poucas vezes nos ótimos locais encontrados anteriormente são fortemente penalizadas.

3. *U* é uma estratégia de penalização uniforme, independente de informações de freqüência, onde $r_i(e) = 2$.

A alternância entre métodos de intensificação (*I*) e diversificação (*D*) caracteriza uma abordagem de oscilação estratégica [59]. O algoritmo GRASP com perturbações de custo e religamento de caminhos está entre as melhores heurísticas disponíveis para o problema de Steiner em grafos.

2.4 Religamento de caminhos

O religamento de caminhos incorporado a um algoritmo GRASP tem como objetivo intensificar a busca em regiões onde soluções de qualidade tenham sido encontradas. O religamento de caminhos foi inicialmente introduzido no contexto da metaheurística busca tabu [58], como uma abordagem para integrar estratégias de intensificação e diversificação ao processo de busca. Em [60] é apresentado um *survey* sobre religamento de caminhos. A técnica consiste em explorar trajetórias que conectam soluções de alta qualidade, começando de uma *solução inicial* e gerando um caminho na vizinhança dessa solução na direção de outra solução, chamada de *solução guia*. Esse caminho é gerado selecionando-se movimentos que introduzam atributos da solução guia na solução inicial. A cada passo, todos os movimentos que incorporam atributos da solução guia são analisados e o melhor movimento é escolhido.

Essa abordagem é denominada de religamento de caminhos porque quaisquer duas soluções visitadas por um algoritmo de busca tabu estão ligadas por uma seqüência de movimentos. Na Figura 2.4 são mostrados

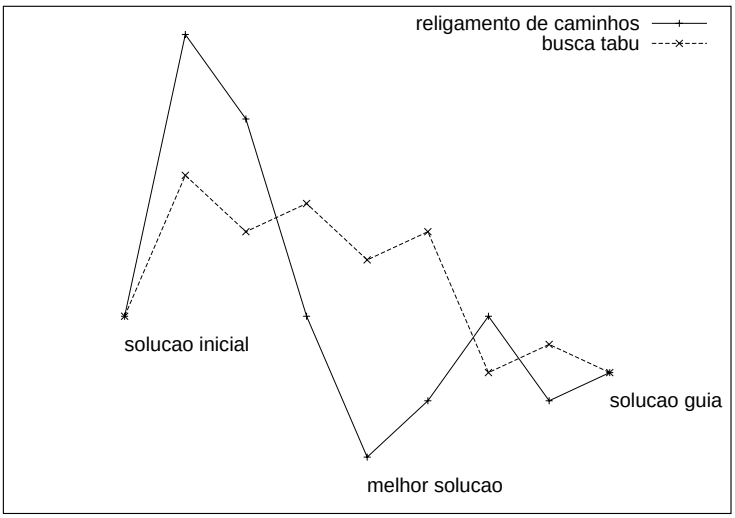

Figura 2.4: Religamento de caminhos no contexto da busca tabu.

dois caminhos hipotéticos, que ligam uma solução inicial a uma solução guia através de uma seqüência de movimentos (para um problema de minimização, sem perda de generalidade). A linha pontilhada mostra soluções visitadas por um algoritmo de busca tabu e a linha contínua mostra as soluções visitadas pelo religamento de caminhos. Observa-se que as trajetórias seguidas pelas duas estratégias são diferentes. Isso ocorre principalmente porque em cada iteração do algoritmo de busca tabu, os movimentos são escolhidos de forma "gulosa", isto é, escolhe-se o movimento não proibido que minimize localmente o valor da função objetivo. Durante o religamento de caminhos, o objetivo principal é incorporar atributos da solução guia à solução inicial, melhorando assim o valor da função objetivo. Como apresentado na Figura 2.4, o objetivo de realizar o religamento de caminhos no contexto da metaheurística busca tabu é obter soluções não visitadas pela trajetória original, que melhorem a função objetivo.

O uso do religamento de caminhos em uma abordagem GRASP foi originalmente introduzido por Laguna e Martí [73]. Na implementação proposta por eles, as três melhores soluções obtidas pela estratégia são armazenadas em um conjunto de elite P para serem usadas como soluções guia pelo religamento de caminhos. Após cada iteração do algoritmo GRASP, a solução obtida é comparada às soluções armazenadas em P. Caso a nova solução seja melhor do que alguma das soluções armazenadas, o conjunto de elite P é atualizado. O religamento de caminhos é feito usando-se a solução produzida pela busca local como solução inicial e uma solução escolhida aleatoriamente de P como solução guia. Outras abordagens que incorpo-

ram o religamento de caminhos a um algoritmo GRASP foram propostas em [7, 5, 27, 113, 117]. Nessas abordagens, nota-se que o religamento de caminhos permite obter soluções de qualidade superior àquelas obtidas pelo método puro.

O religamento de caminhos pode ser incorporado a uma abordagem GRASP de duas formas básicas. A primeira delas consiste em aplicar o religamento de caminhos após cada iteração do algoritmo, entre a solução produzida pela busca local e uma solução de elite. A segunda consiste em realizar o religamento de caminhos entre soluções de elite, obtidas durante as iterações do algoritmo GRASP. Portanto, o conceito de religamento deve ser interpretado de forma diferente no contexto da metaheurística GRASP, visto que uma solução produzida em uma iteração não está ligada às soluções produzidas em outras iterações (como é o caso em um algoritmo de busca tabu). No caso do método GRASP, o religamento de caminhos é usado para "ligar" soluções obtidas em diferentes iterações, com o objetivo de encontrar melhores soluções ao longo da trajetória.

2.5 GRASP em heurísticas híbridas

Em geral, verifica-se que abordagens híbridas de GRASP possibilitam a obtenção de soluções de qualidade superior àquelas obtidas pelo método puro, tendo em alguns casos [47, 117], permitido reduzir os custos de problemas em aberto na literatura. Além da hibridização do método GRASP com o religamento de caminhos, discutido na Seção 2.4, várias abordagens híbridas do método têm sido desenvolvidas. Essas estratégias serão brevemente discutidas a seguir.

Grande parte das estratégias híbridas desenvolvidas a partir de um algoritmo GRASP baseiam-se na substituição da fase de busca local por outro algoritmo de busca. A hibridização do método GRASP com um algoritmo de busca tabu foi inicialmente estudada por Laguna e González-Velarde [72]. Em Delmaire et al. [40], o método GRASP é combinado com um algoritmo de busca tabu de duas formas diferentes. Na primeira delas, o algoritmo GRASP é usado em um esquema de diversificação. A segunda abordagem consiste em substituir a fase de busca local de um procedimento de GRASP Reativo por um algoritmo de busca tabu. Outros exemplos de abordagens híbridas de GRASP, onde um algoritmo de busca tabu ou de *simulated annealing* são usados na fase de busca local, podem ser encontrados em [69, 34, 77].

As estratégias de VNS (*Variable Neighborhood Search*) e VND (*Variable Neighborhood Descent*), propostas por Hansen e Mladenović [67, 84], foram inicialmente introduzidas no método GRASP por Martins et al. [83]. Nesse algoritmo GRASP, desenvolvido para o problema de Steiner em grafos, soluções são construídas de acordo com a estratégia gulosa randomizada do método. A hibridização é feita substituindo-se a busca local por uma estratégia de VND, que explora duas estruturas de vizinhança. Essa abordagem foi melhorada em Ribeiro et al. [117], onde foram desenvolvidas extensões, tais como uma nova estratégia para a exploração de vizinhanças diferentes. Em Ribeiro e Souza [114] o método GRASP foi combinado a uma estratégia de VND para o problema de árvore geradora mínima com restrição de grau. Em Festa et al. [47] foram estudadas variantes da estratégia híbrida de GRASP e VNS para o problema de corte máximo, que permitiram reduzir o custo da melhor solução conhecida para alguns problemas testes. Em Canuto et al. [27] uma estratégia de VNS é usada como um procedimento de pós-otimização em um algoritmo GRASP para o problema de árvore de Steiner com prêmios.

O método GRASP também foi usado juntamente com algoritmos genéticos. Ahuja et al. [3] propõem um algoritmo genético para o problema de atribuição quadrática, onde a população inicial é obtida através do algoritmo guloso randomizado, usado na fase construtiva da heurística GRASP proposta por Li et al. [76]. Uma estratégia semelhante é proposta por Armony [12], onde a população inicial é formada por soluções construídas aleatoriamente e por soluções construídas por uma heurística GRASP. Em Lourenço et al. [80] o método GRASP é usado para fazer cruzamentos.

2.6 Mecanismos usados para acelerar o tempo computacional do método GRASP

Algumas técnicas de implementação podem ser aplicadas ao método GRASP para reduzir o seu tempo computacional. A primeira delas, consiste em usar uma tabela *hash* [2, 35] para acelerar a execução do algoritmo. Uma tabela *hash* é uma estrutura de dados usada para implementar dicionários (conjuntos dinâmicos com operações de inserção, remoção e consulta). Essa estrutura de dados é computacionalmente eficiente, pois o tempo esperado para cada consulta é $O(1)$. Em [83] uma tabela *hash* é usada pela heurística GRASP para armazenar todas as soluções usadas como soluções iniciais pela busca local. Após cada fase construtiva, a tabela *hash* é consultada para

verificar se a solução gerada é nova. Em caso positivo, a solução é inserida na tabela *hash* e o procedimento de busca local é iniciado a partir dessa solução. Caso contrário, o procedimento de busca local não é executado e uma nova iteração é iniciada.

Estratégias de filtragem também podem ser usadas para acelerar a execução do método GRASP. Essas estratégias consistem em aplicar a busca local apenas a soluções obtidas durante a fase construtiva, que sejam consideradas promissoras. Para isso, a cada iteração, submete-se a solução construída a um critério de qualidade definido em função do valor da melhor solução encontrada até o momento. Exemplos de tais estratégias podem ser vistos em [44, 83, 102].

2.7 Estratégias de paralelização

A maior parte das estratégias paralelas de GRASP encontradas na literatura são exemplos de *múltiplas trajetórias independentes*, de acordo com as taxonomias discutidas em [38, 129]. Essa abordagem consiste em distribuir um número máximo k de iterações do algoritmo GRASP, entre ρ processos. O programa é interrompido após a execução de k/ρ iterações por cada processo. Cada processo executa uma cópia do algoritmo seqüencial sobre os dados do problema. Um aspecto importante na implementação dessa abordagem é que deve-se garantir que nenhum par de iterações (seja em um mesmo processo ou entre processos diferentes), seja iniciado com a mesma semente do gerador de números aleatórios [91]. Isso levaria à produção desnecessária de soluções repetidas. Ao final de cada execução, a melhor solução encontrada globalmente pela estratégia é recuperada em uma variável global. Exemplos de estratégias paralelas de GRASP que seguem esse esquema podem ser encontrados em [9, 10, 44, 76, 82, 83, 81, 86, 90, 91].

Em [44, 110] uma heurística GRASP para o problema de conjunto máximo independente foi paralelizada particionando-se o espaço de busca em várias regiões. Em cada região, o espaço de busca é decomposto fixando-se dois vértices para pertencer ao conjunto independente. O algoritmo GRASP é aplicado a cada uma dessas regiões em paralelo. Essa estratégia também pode ser classificada como *múltiplas trajetórias independentes*, de acordo com as taxonomias discutidas em [38, 129].

Em [9, 10] foi proposta uma abordagem para paralelização do método GRASP segundo o esquema de *múltiplas trajetórias independentes*, onde procura-se obter um melhor balanceamento de carga entre os processadores.

Um processo mestre controla as sementes usadas pelos processos escravos. Esse processo envia aos processos escravos blocos de sementes que serão usadas para iniciar iterações do algoritmo. Quando um processo escravo finaliza suas iterações referentes ao bloco de sementes recebido, ele requisita ao processo mestre um novo bloco de sementes. Processos escravos enviam ao processo mestre a melhor solução obtida referente a cada bloco de sementes processado. Observou-se que, em um ambiente onde a carga externa era crítica, uma aplicação com um processo mestre e $\rho - 1$ processos escravos executando o algoritmo GRASP era mais eficiente do que uma estratégia que executava ρ processos GRASP. Essa abordagem foi também usada para paralelizar o problema de atribuição de tráfego em [102].

Estratégias de paralelização classificadas como *múltiplas trajetórias cooperativas*, de acordo com as taxonomias discutidas em [38, 129] também foram desenvolvidas para o método GRASP. Essas estratégias são semelhantes à estratégias de *múltiplas trajetórias independentes*, porém os processos trocam informações colhidas durante a busca. Espera-se que com a troca de informações, a abordagem paralela encontre mais rapidamente uma solução de uma dada qualidade. Estratégias cooperativas podem ser implementadas usando as informações armazenadas para realizar o religamento de caminhos. Em uma abordagem centralizada, as soluções de elite são armazenadas em um *pool* central que é compartilhado pelos processos GRASP. As soluções armazenadas no *pool* central poderão ser usadas para participar do religamento de caminhos com soluções produzidas pela busca local de cada processo. Em uma abordagem distribuída, em cada processo GRASP será mantido um *pool* local e soluções promissoras obtidas por cada processo devem ser enviadas aos demais processos. Em Canuto et al. [27] foi proposta uma estratégia paralela com religamento de caminhos, onde pares de soluções de elite armazenadas em um *pool* central são distribuídas entre os processadores, que realizam o religamento de caminhos em paralelo.

2.8 Conclusão

Desde o final dos anos 80, a metaheurística GRASP tem sido aplicada a uma grande variedade de problemas de pesquisa operacional e de otimização. Esses problemas incluem: escalonamento, roteamento, problemas em lógica, particionamento, localização e layout, otimização em grafos, atribuição, manufatura, transportes, telecomunicação, desenho automático, sistemas

elétricos de força e projeto de circuitos VLSI. Em [48] é apresentada uma bibliografia anotada sobre a metaheurística GRASP.

Um algoritmo básico de GRASP pode ser melhorado usando-se estratégias como a hibridização com outras heurísticas, a paralelização e a introdução de mecanismos alternativos na fase construtiva. Essas extensões foram abordadas nesse capítulo e, em geral, têm tornado o método mais robusto.

Um algoritmo GRASP pode ser implementado em paralelo de forma eficiente. Isso será analisado no próximo capítulo através do estudo do tempo que o método necessita para alcancar uma solucao melhor ou igual a determinado valor alvo.