2
Metaheurísticas para Problemas de Otimização Combinatória

Inicialmente, este capítulo faz algumas considerações sobre as principais técnicas usadas para tratar problemas de otimização combinatória. Em seguida, a Seção 2.2 aborda as principais definições da área de metaheurísticas, dando ênfase às metaheurísticas GRASP (do inglês *Greedy Randomized Adaptive Search Procedure*) e ILS (do inglês *Iterated Local Search*). Posteriormente, objetivando apresentar as aplicações usadas nesta tese para efeito de validação e como estudo de caso, a Seção 2.3 aborda o mTTP e a Seção 2.4 o problema da árvore geradora de custo mínimo com restrição de diâmetro.

2.1
Considerações Iniciais

Problemas de otimização têm como objetivo encontrar uma solução que maximize ou minimize uma função de custo definida sobre um certo domínio [29]. Grande parte dos problemas de otimização pertence à classe dos problemas $\mathcal{NP}$-difíceis [74]. Diante do fato de não se conhecerem algoritmos polinomiais que resolvam problemas dessa classe de forma exata, os algoritmos heurísticos fornecem soluções de boa qualidade em um tempo razoável. Embora as heurísticas não garantam nenhum nível de qualidade da solução desenvolvida, os custos das soluções encontradas, na prática, geralmente estão próximos da solução ótima [130].

Há várias técnicas diferentes usadas para se desenvolver os métodos heurísticos, entre elas as heurísticas construtivas, as heurísticas de busca local e as metaheurísticas [29, 34, 130, 131].

Heurísticas construtivas geram uma solução passo a passo de forma incremental. A cada iteração, um membro do conjunto de elementos é escolhido e inserido em uma solução, até completá-la [29].

Os algoritmos de busca local começam a partir de uma solução inicial viável e iterativamente substituem a solução corrente por uma solução vizinha melhor, onde solução vizinha é aquela que pode ser obtida através de pequenas mudanças estruturais da solução corrente [34].

As metaheurísticas são procedimentos de alto nível que objetivam explo-

rar eficientemente o espaço de busca para encontrar boas soluções. Como as metaheurísticas são objeto de estudo desta tese, a Seção 2.2 descreve detalhadamente suas características específicas.

2.2 Metaheurísticas

Metaheurísticas são modelos conceituais para combinar diferentes heurísticas [129, 130, 131]. As metaheurísticas diferenciam-se entre si basicamente pelo mecanismo usado para escapar de ótimos locais. Elas se dividem em duas categorias, de acordo com o princípio usado para explorar o espaço de soluções: métodos populacionais e métodos baseados em busca local.

Os métodos populacionais consistem em manter um conjunto de boas soluções e combiná-las de forma a tentar produzir soluções ainda melhores. Exemplo clássico de procedimento dessa categoria são os algoritmos genéticos [21, 78].

Nas metaheurísticas baseadas em busca local, a exploração do espaço de soluções é feita por meio de movimentos, que são aplicados a cada passo sobre a solução corrente, gerando outra solução promissora em sua vizinhança. Dentre as várias metaheurísticas pertencentes a esse grupo, destacam-se na literatura Busca Tabu [76, 77], *Simulated Annealing* [99, 135], GRASP (*Greedy Randomized Adaptive Search Procedure*) [60, 62], VNS (*Variable Neighborhood Search*) [89, 91] e ILS (*Iterated Local Search*) [102, 151]. As Seções 2.2.1 e 2.2.2 apresentam características específicas das metaheurísticas GRASP e ILS, respectivamente.

2.2.1 Metaheurística GRASP

A metaheurística GRASP (do inglês *Greedy Randomized Adaptive Search Procedure*) [60, 62, 132] é um processo iterativo, onde cada iteração é independente das demais e consiste de duas fases: construção e busca local. A primeira é uma heurística construtiva aleatorizada que gera uma solução passo-a-passo servindo-se dos elementos de uma lista restrita de candidatos (LRC). A segunda fase de cada iteração do algoritmo é um método de busca local, que investiga a vizinhança da solução criada na primeira fase, até que um ótimo local seja encontrado. A melhor solução obtida é guardada e retornada ao final do processo iterativo.

GRASP é adaptativo porque os benefícios associados com a escolha de cada elemento são atualizados em cada iteração da fase de construção para refletir as mudanças oriundas da seleção do elemento anterior. A componente

probabilística do procedimento reside no fato de que cada elemento é selecionado de forma aleatória a partir da LRC. Essa técnica de escolha permite que diferentes soluções sejam geradas em cada iteração GRASP [62].

O procedimento GRASP procura, portanto, conjugar bons aspectos dos algoritmos puramente gulosos, com aqueles dos procedimentos aleatórios de construção de soluções. Procedimentos híbridos mais sofisticados podem ser obtidos através de sua integração com outras técnicas [55, 128].

O Algoritmo 1 ilustra um procedimento GRASP padrão para um problema de minimização. O procedimento retorna a melhor solução S^* encontrada pelo algoritmo. Na linha 4, a variável auxiliar $custo^*$, usada para guardar o custo da melhor solução, é inicializada. O laço das linhas de 5 a 12 é repetido até que um certo critério de parada seja satisfeito. Na linha 6, uma solução S é construída através de um algoritmo guloso aleatorizado implementado pela função `ConstrucaoGulosaAleatoria()`. Na linha 7, uma busca local é aplicada à solução S pela função `BuscaLocal()`, obtendo-se a solução S'. Na linha 8, é averiguado se o custo da solução S' é menor do que $custo^*$. Se for, a solução S^* e o valor $custo^*$ são atualizados respectivamente nas linhas 9 e 10. Por último, a melhor solução encontrada entre todas as iterações é retornada na linha 13.

```
1  Algoritmo: GRASP_Padrao();
2  Entrada: CriterioParada, semente;
3  Saída: S*;
4  custo* ← ∞;
5  enquanto NOT.CriterioParada faça
6  |  S ← ConstrucaoGulosaAleatoria(LRC);
7  |  S' ← BuscaLocal(S);
8  |  se (Custo(S') < custo*) então
9  |  |  S* ← S';
10 |  |  custo* ← Custo(S');
11 |  fim
12 fim
13 retorna S*;
```

Algoritmo 1: Pseudo-código de um GRASP padrão.

2.2.2 Metaheurística ILS

A metaheurística ILS (do inglês *Iterated Local Search*) [102] começa de um ótimo local. Uma perturbação aleatória é aplicada à solução corrente, seguida de uma busca local. Se o ótimo local obtido após esses passos satisfizer um determinado critério de aceitação, ele é aceito como a nova solução corrente,

caso contrário a solução corrente não é alterada. A melhor solução encontrada é eventualmente atualizada e os passos acima são repetidos até que um determinado critério de parada seja atingido.

A qualidade das soluções depende da solução inicial, da perturbação, da busca local e do critério de aceitação usados. A perturbação precisa ser suficientemente forte para permitir que a busca local explore diferentes soluções, mas também não pode ser muito forte para evitar um reinício aleatório. O critério de aceitação determina quando uma solução substitui a solução corrente. Dois exemplos extremos são aceitar a nova solução somente se ela for melhor que o mínimo local anterior ou sempre aceitar a nova solução, independentemente de ser melhor ou não do que a anterior.

O Algoritmo 2 ilustra um procedimento ILS padrão. Na linha 4, uma solução inicial S é gerada através da função `CriaSolucao()`. Em seguida, na linha 5, uma busca local é aplicada à solução S pela função `BuscaLocal()`. A variável $custo^*$, inicializada na linha 6, guarda o valor da melhor solução. O laço das linhas de 7 a 15 é repetido até que algum critério de parada seja satisfeito. Uma perturbação é aplicada à solução S na linha 8 pela função `Perturbacao()`, obtendo-se S'.

```
 1 Algoritmo: ILS_Padrao();
 2 Entrada: CriterioParada, semente;
 3 Saída: S*;
 4 S ← CriaSolucao();
 5 S ← BuscaLocal(S);
 6 custo* ← Custo(S);
 7 enquanto NOT.CriterioParada faça
 8 |   S' ← Perturbacao(S);
 9 |   S' ← BuscaLocal(S');
10 |   S ← CriterioAceitacao(S, S');
11 |   se (Custo(S') < custo*) então
12 |   |   S* ← S';
13 |   |   custo* ← Custo(S*);
14 |   fim
15 fim
16 retorna S*;
```

Algoritmo 2: Pseudo-código de um ILS padrão.

Posteriormente, na linha 9, a função `BuscaLocal()` realiza uma busca local na solução S' a fim de aprimorá-la. Se o ótimo local S' satisfaz o critério de aceitação, então ocorre uma substituição da solução S na linha 10. Na linha 11, é averiguado se o custo de S' é melhor do que o $custo^*$. Se for, então a solução S^* e a variável $custo^*$ são atualizadas, respectivamente nas linhas 12 e 13. O procedimento retorna S^* na linha 16.

2.2.3 Metaheurística GRASP com ILS

Nas metaheurísticas híbridas que usam GRASP e ILS, a primeira é usada como geradora da solução inicial e, a segunda para o refinamento dessa solução. A idéia da hibridação GRASP com ILS consiste em aproveitar as características positivas de cada metaheurística.

A metaheurística GRASP tem a vantagem de geralmente apresentar boas soluções em tempo computacional relativamente baixo. A metaheurística ILS tem a capacidade de melhorar soluções iniciais. Dessa forma, a hibridação GRASP e ILS tem sido usada na resolução de vários problemas. As Seções 2.3.2 e 2.4.2 apresentam duas formas de explorar este método.

As próximas seções apresentam os dois problemas usados nesta tese como aplicações teste para validar a estratégia de paralelização proposta.

2.3 O Problema do Torneio com Viagens Espelhado

O escalonamento (ou programação) de jogos é uma classe importante de problemas de otimização combinatória. As ligas esportivas representam uma grande atividade econômica e muitas equipes investem em jogadores e infra-estrutura esperando obter um retorno econômico. Um maior retorno dos investimentos pode ser obtido por uma adequada programação dos jogos das competições.

O problema de programação de tabelas consiste em determinar quando e onde os jogos de um determinado torneio serão realizados. Problemas dessa natureza contêm em geral muitas restrições que devem ser satisfeitas e diferentes objetivos a cumprir, tais como a minimização dos deslocamentos das equipes durante o campeonato e o número máximo de partidas consecutivas realizadas na sede da equipe ou fora dela [147].

O problema do torneio com viagens (TTP, do inglês *Traveling Tournament Problem*) é um problema de geração de tabelas proposto inicialmente por Easton et al. [56]. O TTP consiste em gerar uma programação de jogos na qual cada equipe deve jogar duas vezes com cada uma das outras equipes, de forma que nenhuma delas jogue mais de três jogos consecutivos em casa nem mais de três jogos consecutivos fora, não aconteçam jogos consecutivos entre as mesmas duas equipes e a distância total percorrida pelas equipes durante o torneio seja minimizada. Esse tipo de torneio é conhecido na literatura por *Double Round Robin* (DRR) e são necessárias pelo menos $2(n-1)$ rodadas para a realização de todos os jogos de um torneio. Assume-se que cada equipe tem um estádio na sua cidade e as distâncias entre cada par de estádios são

conhecidas e simétricas, ou seja, a distância da sede de uma equipe A à sede de uma equipe B é igual a distância da sede de B à sede de A. Cada equipe inicia o torneio em sua cidade sede e viaja para cumprir seus jogos nas sedes escolhidas. Cada equipe retorna para sua sede após o último jogo fora.

O problema do torneio com viagens espelhado (mTTP, do inglês *mirrored Traveling Tournament Problem*) representa um caso particular do TTP. O mTTP pode ser visto como um torneio simples em suas $n-1$ primeiras rodadas, chamado de primeiro turno, seguido pelo mesmo torneio com os mandos de campo invertidos em relação às $n-1$ primeiras rodadas, chamado de returno. Isso quer dizer que a ordem dos jogos realizados no returno é exatamente a mesma dos jogos do turno, apenas com os estádios invertidos: isso é, quem jogou em casa jogará fora e vice-versa.

Assim como no TTP, deve-se assumir que cada equipe está na sua própria cidade no começo do torneio, para onde voltará quando o torneio acabar. Quando uma equipe joga dois jogos consecutivos fora de casa, ela viaja diretamente da cidade da primeira oponente para a cidade da segunda, sem retornar à sua cidade sede.

A grande variedade de combinações possíveis, acrescentada das diversas restrições impostas pelo mTTP, faz com que esse seja um problema de otimização bastante difícil. Como conseqüência, a maior instância teste para o qual a solução ótima é conhecida é composta de apenas oito equipes. Essa solução foi obtida após quatro dias de processamento em paralelo, usando 20 processadores Pentium II com 512 Mbytes de memória RAM [57].

A Seção 2.3.1 apresenta uma revisão bibliográfica dos principais trabalhos relacionados ao mTTP. Em seguida, a Seção 2.3.2 destaca os detalhes específicos da implementação seqüencial usada como referência para a paralelização do mTTP neste trabalho.

2.3.1 Trabalhos Relacionados ao Problema de Geração de Tabelas

A programação de tabelas é a principal área de aplicação dos métodos de pesquisa operacional em esportes. Esse problema tem sido tratado por vários autores em diferentes contextos, ligas e esportes como futebol, basquete, hóquei, críquete, rúgbi, beisebol e futebol americano [56, 57, 58, 134, 146].

Bean e Birge [25] trataram o problema da programação de tabela da *National Basketball League* (NBA) dos Estados Unidos, no qual as restrições mais limitantes estavam relacionadas aos dias de descanso e à disponibilidade de estádios. Costa [41] considerou o problema da programação da tabela da *National Hockey League* (NHL) dos Estados Unidos, no qual um dos objetivos

era minimizar a distância viajada pelas equipes durante o torneio.

Diversas técnicas têm sido usadas para a solução do TTP (e do mTTP) desde a sua formulação em [56]. Easton et al. [57] usaram programação inteira, obtendo os valores ótimos para as instâncias $nl4$, $nl6$ e $nl8$ (como já mencionado, o valor ótimo da instância $nl8$ só foi encontrado após quatro dias de processamento em paralelo, com 20 processadores [57]).

Grauwels e van Oudheusden em [44, 45] propuseram heurísticas baseadas em colônias de formigas para o TTP. Outro trabalho baseado em Colônia de Formigas para o TTP [38] não melhorou as melhores soluções já conhecidas, mas alcançou resultados superiores à implementação de [45].

Em [27, 92] propõe-se o uso de programação por restrições para resolver o TTP. Di Gaspero e Schaerf [51, 52] implementaram uma busca tabu eficiente, com a qual foi possível melhorar a melhor solução da instância $circ10$ e todas as soluções então conhecidas para as instâncias criadas com valores constantes entre as distâncias, chamadas de instâncias do grupo con$n$ [145].

Há dois trabalhos baseados em *simulated annealing* para o TTP. O primeiro foi proposto por Anagnostopoulos et al. [11, 12]. A outra implementação, proposta por Lim et al. [101], usa *simulated annealing* e um método de descida, tendo sido capaz de melhorar os melhores conhecidos para cinco instâncias do grupo circ$n$.

Em 2006, foram propostas três eficientes heurísticas construtivas [24, 71, 96] para o TTP. A heurística construtiva em [24] propõe a criação de um bloco de jogos para uma determinada equipe, que é gerado a partir de uma árvore geradora mínima. Os autores sugerem que essa heurística seja usada como ponto de partida por heurísticas de busca local, pois ela apresentou resultados promissores. A heurística construtiva apresentada em [71] considera apenas as instâncias do grupo con$n$. Para essas instâncias os autores apresentam resultados melhores do que os então conhecidos como os melhores na literatura. A heurística construtiva mostrada no artigo [96] inicialmente ignora as restrições do TTP. O problema é convertido em um problema do caixeiro viajante e soluções ótimas são construídas. Essas soluções ótimas para o problema do caixeiro viajante são usadas como base na construção de soluções viáveis para o TTP.

Rasmussen and Trick [124] propuseram um algoritmo híbrido, composto por programação inteira e programação por restrição. Esse algoritmo foi capaz de encontrar soluções aproximadas para grandes instâncias.

Outra eficiente hibridação foi proposta por Urrutia e Ribeiro em [134] para o mTTP. Esse algoritmo é uma hibridação baseada nas metaheurísticas GRASP e ILS, chamado `GRILS-mTTP`, e foi capaz de melhorar todas as

melhores soluções conhecidas na época para o mTTP, publicadas em [145]. Em conseqüência de sua eficiência, as implementações paralelas para o mTTP desenvolvidas nesta tese foram baseadas nessa hibridação. A Seção 2.3.2 mostra detalhes específicos dessa heurística híbrida.

2.3.2 Heurística Híbrida para o mTTP

A heurística híbrida `GRILS-mTTP` [134] basicamente substitui a fase de busca local da metaheurística GRASP pelo procedimento ILS. O Algoritmo 3 apresenta o pseudo-código da heurística híbrida `GRILS-mTTP` para encontrar boas soluções viáveis para o problema do torneio com viagens espelhado.

```
1  Algoritmo: GRILS-mTTP();
2  Entrada: CriterioParada, CriterioReinicializacao;
3  Saída: S*;
4  enquanto .NOT.CriterioParada faça
5  |  S ← ConstrucaoGulosaAleatoria();
6  |  S, S ← BuscaLocal(S);
7  |  repita
8  |  |  S' ← Perturbacao(S);
9  |  |  S' ← BuscaLocal(S');
10 |  |  S ← CriterioDeAceitacao(S, S');
11 |  |  S* ← AtualizaMelhorSolucao(S, S*);
12 |  |  S ← AtualizaMelhorSolucaoNaIteracao(S, S);
13 |  até CriterioReinicializacao ;
14 |  retorna S*;
15 fim
```

Algoritmo 3: Pseudo-código da heurística híbrida GRASP com ILS para o mTTP [147].

O ciclo externo no Algoritmo 3, formado pelas linhas de 4 até 15, executa uma fase de construção GRASP seguida por uma fase de busca local ILS, até que um critério de parada seja satisfeito. A fase de construção GRASP é realizada pelas linhas 5 e 6. Na linha 5 é gerada uma solução através da função `ConstrucaoGulosaAleatoria()`, e na linha 6 essa solução é melhorada pela função `BuscaLocal()`. O ciclo interior, que inclui as linhas de 7 até 13, é a fase de busca ILS que começa com a aplicação de uma perturbação à solução corrente S, obtendo-se a nova solução S', através da função `Perturbacao()`. Em seguida, na linha 9 é aplicado um algoritmo de busca local em S' pela função `BuscaLocal()`. Na linha 10, a nova solução S' é aceita ou não como a nova solução corrente, dependendo do resultado da aplicação do critério de aceitação implementado pelo algoritmo `CriterioDeAceitacao()`. A solução

S' é aceita como a nova solução corrente se seu custo é menor do que $(1 + \beta)$ vezes o custo da solução corrente S. O parâmetro β é inicializado com $0,0001$ no começo de cada iteração GRASP e reinicializado com o mesmo valor cada vez que a solução corrente muda, conforme sugerido em [147]. Se a solução corrente não muda após um determinado número de iterações (usou-se $12n$ na implementação, seguindo o proposto em [147]), o valor de β é dobrado, ou seja, o critério de aceitação é relaxado.

Nas linhas 11 e 12, a melhor solução S^* e a melhor solução na iteração GRASP corrente são eventualmente atualizadas e um novo ciclo começa com uma perturbação da solução corrente, até que algum critério de reinicialização seja atingido. Nesse caso, uma nova iteração GRASP começa se um determinado número de movimentos (usou-se 50 na implementação seqüencial) não aprimorantes foram aceitos desde a última vez em que a melhor solução na iteração GRASP corrente foi atualizada.

A reinicialização ocorre se demasiadas perturbações seguidas de busca local foram executadas sem melhorar a melhor solução na iteração GRASP. Dessa forma, procura-se fazer com que a iteração GRASP não seja interrompida se a solução corrente S ainda está melhorando.

O algoritmo `BuscaLocal()` aplicado em S', na linha 9 usa quatro estruturas de vizinhança. As três primeiras são simples trocas: Troca de Anfitriões (TA), Troca de Equipes (TE) e Troca Parcial de Rodadas (TPR). A vizinhança Rotação de Jogos (RJ) é baseada em cadeias de ejeção, isso significa que são realizados pequenos movimentos em alguns elementos da solução, os quais forçam que outros elementos sejam ejetados de seus estados atuais. Ela é explorada somente como um movimento de diversificação, devido ao alto custo computacional da geração e avaliação de seus movimentos. Conseqüentemente, essa última vizinhança é realizada com menos frequência pela heurística como uma pertubação [147].

Esse algoritmo de busca local usa uma estratégia do tipo primeiro aprimorante similar à do procedimento VND (do inglês, *Variable Neighborhood Descent*) [90]. TE é a primeira vizinhança explorada. Uma vez que um ótimo local com relação a essa vizinhança é atingido, uma busca local na vizinhança TA é executada na procura de uma melhor atribuição de mando de campos.

Em seguida, a vizinhança TPR é investigada. Como no caso anterior, uma vez que um ótimo local com relação a essa vizinhança é encontrado, o algoritmo executa uma busca local rápida na vizinhança TA à procura de uma melhor atribuição de campos. Esse esquema se repete até que um ótimo local com relação às três vizinhanças seja obtido. As soluções vizinhas são visitadas de forma aleatória dentro de cada vizinhança [147].

2.4 O Problema da Árvore Geradora de Custo Mínimo com Restrição de Diâmetro

O problema da árvore geradora desperta grande interesse por ter uma série de aplicações práticas, por exemplo, na instalação de linhas telefônicas (ou elétricas) e em projetos de redes de computadores e de televisão a cabo [1, 79, 80, 95].

Para uma definição formal desse problema, considera-se um grafo conexo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, com um custo $c_{ij} \geq 0$ associado a cada aresta $(i, j) \in E$. Um subgrafo $T = (V, E')$ conexo e sem ciclos de G é denominado de árvore geradora de G.

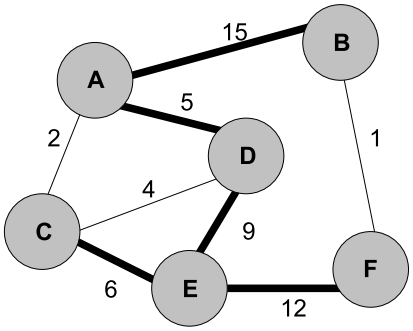

Figura 2.1: Árvore geradora.

Por exemplo, na Figura 2.1, o subgrafo cujas arestas estão indicadas em linhas grossas é uma árvore geradora do grafo representado. Uma árvore geradora com $|V|$ vértices possui $|E| = |V| - 1$ arestas. Uma árvore T de um grafo é denominada Árvore Geradora Mínima (AGM) de G se, e somente se, seu custo for mínimo dentre todas as árvores geradoras possíveis. A AGM do grafo da Figura 2.1 está representada na Figura 2.2.

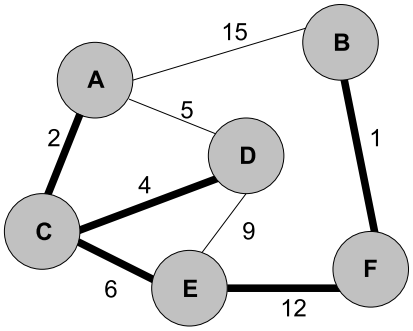

Figura 2.2: Árvore geradora mínima.

O diâmetro de uma árvore é a quantidade de arestas em seu maior caminho. Assim, o problema da Árvore Geradora de Custo Mínimo com Restrição de Diâmetro (AGMD) consiste em encontrar uma árvore geradora

de G com custo mínimo, tal que qualquer caminho nesta árvore contenha no máximo D arestas, onde $D \geq 2$. Esse problema é NP-difícil [74] para diâmetros $4 \leq D < |V| - 1$.

Para ilustrar esse problema, na Figura 2.3 é apresentado um exemplo. Na Figura 2.3-(a) é mostrado um grafo com custos. Em seguida, na Figura 2.3-(b) há uma AGM do grafo em (a) com custo igual a 21. Para finalizar, a Figura 2.3-(c) mostra uma solução ótima de custo 51 para a AGMD com diâmetro menor ou igual a 2.

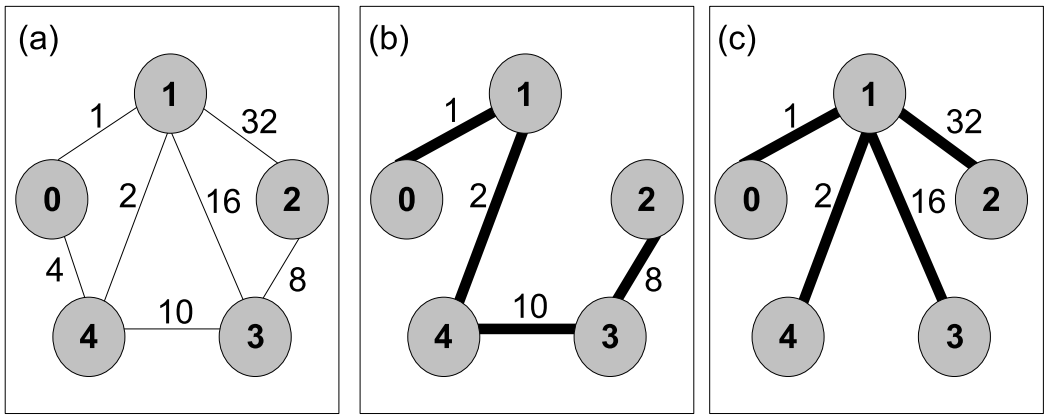

Figura 2.3: Árvore geradora de custo mínimo com restrição de diâmetro.

2.4.1 Trabalhos Relacionados ao Problema da AGMD

Na literatura há vários trabalhos que exploram o problema da AGMD. Os métodos usados variam desde algoritmos exatos [80, 86, 117] até heurísticos. As primeiras formulações matemáticas foram apresentadas em [2, 3]. Gruber e Raidl [86] propuseram formulações baseadas em desigualdades válidas, mas não conseguiram resolver instâncias maiores do que as instâncias já apresentadas na literatura. Outra formulação, baseada em programação de restrição, foi proposta em [117]. Com essa última formulação foi possível solucionar instâncias em grafos completos de até 40 vértices.

Abdalla [1] apresenta quatro heurísticas construtivas para o problema da AGMD. Dentre essas, a OTT (*One Time Tree*) é relatada como sendo a mais eficiente para resolver instâncias em grafos completos. Para as instâncias em grafos esparsos, o autor destaca uma heurística de refinamento iterativo da AGM, também descrita em [1]. Em [49] são apresentados alguns algoritmos paralelos para as heurísticas acima.

A heurística *Randomized Greedy Heuristic* (RGH) e um algoritmo genético foram desenvolvidos por Raidl e Julstrom [123], que realizaram testes com instâncias em grafos completos com até 1000 vértices. Os testes compa-

rativos realizados entre as heurísticas OTT e RGH apontam para um melhor desempenho da segunda. Outra heurística proposta por Gruber e Raidl [87], baseada na metaheurística VNS, apresentou resultados melhores do que as versões aprimoradas de um algoritmo genético [95].

Em [85], foram apresentadas uma implementação de colônia de formigas e um algoritmo evolucionário que usam as quatro vizinhanças apresentadas em [87]. Os resultados mostram que esses algoritmos foram capazes de melhorar as soluções encontradas pela heurística VNS implementada [87]. Além disso, nos casos em que há uma forte limitação de tempo de processamento, a implementação do algoritmo evolucionário obtém as melhores soluções se comparado à colônia de formigas. Por outro lado, nas execuções mais longas, os melhores resultados são obtidos pela colônia de formigas.

Em [136], são apresentadas uma formulação matemática, uma relaxação lagrangeana, adaptações nas heurísticas construtivas propostas em [49], quatro estratégias de busca local, uma heurística do tipo GRASP e outra híbrida GRASP e ILS. Nos testes realizados, constatou-se que a heurística híbrida apresentou um desempenho superior ao GRASP. Dessa forma, essa última foi escolhida para ser a segunda aplicação teste usada nesta tese para validar a estratégia de paralelização para metaheurísticas em ambientes *grid*.

2.4.2 A Heurística Híbrida para o Problema da AGMD

O algoritmo híbrido desenvolvido em [136] é baseado nas metaheurísticas GRASP e ILS. O pseudo-código do algoritmo híbrido desenvolvido é apresentado no Algoritmo 4. Inicialmente as variáveis *custo*, *tipo*, $filtro$, $filtro\_bl$ e $CriterioAtualizacaoFiltro$ são inicializadas. Em seguida, no laço mais externo formado pelas linhas de 9 até 28, as fases de construção GRASP e a fase de busca ILS são executadas até que o $CriterioParada$ seja satisfeito.

A fase de construção GRASP é usada para a geração de soluções iniciais. Essa fase é formada pelas linhas de 10 até 13, e é baseada na heurística OTT-M2[136], que é uma versão atualizada do algoritmo OTT [1]. A heurística OTT-M2 não garante a obtenção de soluções iniciais viáveis para grafos esparsos. Para resolver esse problema, são introduzidas arestas artificiais com custo muito alto, chamadas de falsas arestas. A solução inicial é então gerada como se o grafo fosse completo. Devido a isso, a função `OTT_M2()` é executada até que a solução gerada tenha um valor de custo menor do que a variável $filtro$ na linha 12. Em seguida, na linha 13 a solução S é melhorada através da função `BuscaLocal()`, que realiza uma busca local gerando as soluções S' e S^*.

A segunda fase dessa heurística híbrida é baseada na busca local ILS, desenvolvida pelo laço das linhas de 14 até 22. Uma perturbação é aplicada à solução S na linha 15 e a solução modificada é armazenada em S', através da função `Perturbacao()`. Em seguida, na linha 16 o desvio relativo, dado através da fórmula (`Custo`(S')-$custo^*$)/$custo^*$ é calculado a partir dos custos da melhor solução conhecida e da solução S'. Para que a função `BuscaLocal()` seja aplicada à solução proveniente da perturbação, *desvio* deve ser menor ou igual ao valor de $filtro\_bl$, analisado na linha 17. Se for, a busca local é aplicada na linha 18. Em seguida, na linha 19 é investigado através da função `CriterioAceitacao()` se a solução S' é melhor do que S. Se for, então a solução S' é aceita como solução corrente e S é atualizada. Na linha 20, a função `AtualizaMelhorSolucao()` verifica se a melhor solução S^* deve ser atualizada ou não.

Na linha 23 é averiguado se o algoritmo atingiu o critério de atualização do filtro. Se tiver atingido, o segundo filtro é relaxado na linha 24. O critério é atingido se for gerado mais de 100 soluções sem aprimorar a melhor solução conhecida. Nas linhas 26 e 27 a variável *tipo* que controla a perturbação a ser efetuada é atualizada. As perturbações são executadas alternadamente assim, se em uma dada iteração a variável *tipo* indicar que a perturbação a ser executada é a DCV (Deslocamento do Centro para um Vizinho), então na próxima iteração a perturbação a ser utilizada é a SAR (Substituição Aleatória de Raiz) e vice-versa.

Na perturbação DCV, um nó filho do nó central da árvore (caso par) é considerado o novo centro da árvore. No caso ímpar, um dos nós filhos de uma das extremidades da aresta central torna-se uma extremidade da aresta central.

No segundo tipo de perturbação, chamada de SAR, um nó é escolhido aleatoriamente para ser o novo centro da árvore no caso par (respectivamente, para substituir uma das extremidades da aresta central no caso ímpar). Todos os filhos do antigo nó central no caso par (respectivamente, de uma das extremidades da antiga aresta central no caso ímpar) são herdados pelo novo nó. O antigo nó central é conectado à árvore a custo mínimo, de modo que a solução permaneça viável.

Além disso, foi definido um limite para interromper a aplicação da perturbação e da busca local, e gerar nova solução inicial utilizando-se a heurística OTT-M2. Se a perturbação a ser aplicada é DCV, então o limite máximo de perturbações a ser aplicado é igual ao grau de saída do antigo nó central no caso par, ou da extremidade da aresta central a ser substituída

no caso ímpar. Quando a perturbação é do tipo SAR, o limite máximo de perturbação a ser aplicado é $|V|/3$.

Esses procedimentos são executados até que algum critério de parada estabelecido seja atingido, retornando uma solução S^* na linha 29. Nesse algoritmo, dois critérios de parada podem ser utilizados: o tempo de processamento ou a quantidade de iterações.

```
1  Algoritmo: Hibrido_AGMD();
2  Entrada: G = (V, E), CriterioParada, CriterioReinicio,
   semente;
3  Saída: S*;
4  custo ← ∞;
5  tipo ← DCV;
6  filtro ← [(|V| − 1)/3].M;
7  filtro_bl ← 0,15;
8  CriterioAtualizacaoFiltro ← 100;
9  enquanto NOT.CriterioParada faça
10 |  repita
11 |  |  S ← OTT_M2(semente);
12 |  até (Custo(S) ≤ filtro) ;
13 |  S', S* ← BuscaLocal(S);
14 |  enquanto .NOT.CriterioReinicio faça
15 |  |  S' ← Perturbacao(tipo, S);
16 |  |  desvio ← (Custo(S') − custo*)/custo*;
17 |  |  se (desvio ≤ filtro_bl) então
18 |  |  |  S' ← BuscaLocal(S');
19 |  |  |  S ← CriterioAceitacao(S, S');
20 |  |  |  S* ← AtualizaMelhorSolucao(S, S*);
21 |  |  fim
22 |  fim
23 |  se CriterioAtualizacaoFiltro então
24 |  |  filtro_bl ← filtro_bl + 0,02;
25 |  fim
26 |  se (tipo = DCV) então tipo ← SAR;
27 |  senão tipo ← DCV;
28 fim
29 retorna S*;
```

Algoritmo 4: Pseudo-código da heurística híbrida GRASP com ILS para AGMD.

A diferença entre esta heurística híbrida e a descrita na Seção 2.3.2 está relacionada aos filtros usados no Algoritmo 4, linhas 12 e 24. O primeiro controla a quantidade de falsas arestas presentes na solução. O segundo controla a qualidade das soluções provenientes de uma perturbação a serem exploradas pela busca local. O objetivo do segundo filtro é especificar em quais soluções pretende-se consumir tempo de busca. Uma solução gerada a partir de uma perturbação só é submetida à busca local se seu custo encontra-se no máximo 15% acima do valor da melhor solução conhecida [136]. Esse filtro é relaxado sempre que a busca evolui e torna-se mais difícil aprimorar a melhor solução conhecida. O filtro é incrementado em 2% sempre que o algoritmo permanece uma determinada quantidade de iterações sem aprimorar a melhor solução conhecida.

2.5 Considerações Finais

Metaheurísticas têm sido amplamente utilizadas para obter soluções de problemas de otimização combinatória. Nesse tipo de estratégia, boas soluções são geradas, em um tempo computacional inferior ao requerido para realizar uma enumeração completa.

Nesse contexto, este capítulo abordou algumas características específicas das metaheurísticas e dois problemas de otimização combinatória NP-difíceis, o mTTP e o problema da AGMD.

Como foi observado, a hibridação de metaheurísticas é uma técnica comumente usada para construir heurísticas mais eficientes e robustas. As heurísticas implementadas para o mTTP e para o problema da AGMD, tiveram para algumas das suas instâncias os melhores valores conhecidos na literatura através de uma heurística híbrida GRASP com ILS.

A hibridação GRASP com ILS tem apresentado bons resultados porque aproveita as características positivas de cada metaheurística. A metaheurística GRASP tem a vantagem de geralmente conseguir boas soluções em tempo computacional relativamente baixo, e a metaheurística ILS tem a capacidade de melhorar soluções inicias. Como as hibridações implementadas para o mTTP e para o problema da AGMD têm apresentado bons resultados, ambas foram usadas como aplicações testes desta tese.

Contudo, por mais eficientes que sejam as metaheurísticas, as mesmas ainda são aplicações que requerem um alto poder computacional. Dessa forma, a paralelização é uma estratégia cada vez mais comumente adotada nesse tipo de algoritmo. O próximo capítulo aborda as principais características das metaheurísticas paralelas.