OTIMIZAÇÃO COMBINATÓRIA

Sumário

Prefácio

Compilado em 28 de junho de 2024.

Parte I

Heurísticas Gulosas e Busca Local

1
Introdução

A otimização combinatória é uma importante área relacionada à pesquisa operacional. Um problema de otimização combinatória consiste em encontrar uma solução ótima dentre um conjunto finito de soluções. Neste capítulo, o Problema do Caixeiro Viajante é apresentado, junto a uma breve discussão acerca de sua importância e dificuldade de resolução. Além disso, para que o problema possa ser expresso de forma simples e consistente, alguns conceitos básicos sobre grafos são abordados.

1.1 TSP

O Problema do Caixeiro Viajante (*Traveling Salesman Problem*, TSP) é um dos mais famosos problemas de otimização combinatória na literatura. Dado um conjunto de cidades, o TSP consiste em encontrar uma rota que visite todas elas de forma a minimizar a distância total percorrida[1] (ou o tempo total de viagem). A Figura 1.1 apresenta um exemplo de instância do TSP, na qual há seis cidades dispostas em um espaço bidimensional. No exemplo, a distância entre as cidades 1 e 4 é de 118 km, enquanto a das cidades 1 e 3 é de 174 km. De posse das distâncias entre cada par de cidades, obtém-se algo semelhante à Tabela 1.1.

Uma solução válida (o termo "viável" será utilizado daqui em diante) para um TSP é uma sequência que visita cada cidade exatamente uma vez (um *tour*). As figuras 1.2a e 1.2b mostram exemplos de soluções viáveis para a mesma instância. Ao verificar a Tabela 1.1, percebe-se que a distância total percorrida[2] na solução da Figura 1.2a é de $129 + 114 + 250 + 186 + 105 + 118 = 902$ km. A solução da Figura 1.2b, por sua vez, possui uma distância total percorrida de $174 + 114 +$

[1] Minimizar a distância total percorrida é a "função objetivo" do problema.

[2] Se a função objetivo do problema é minimizar a distância total percorrida, diz-se que a distância percorrida em uma solução é o seu "valor objetivo".

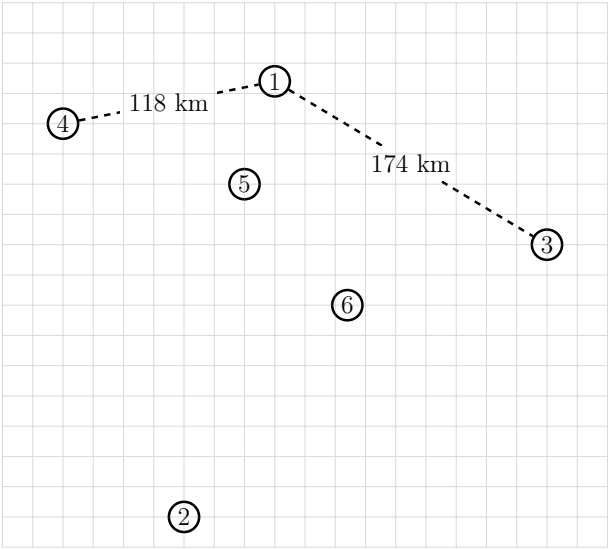


Figura 1.1: Exemplo de instância.

Tabela 1.1: Distâncias entre cada par de cidades (em km).

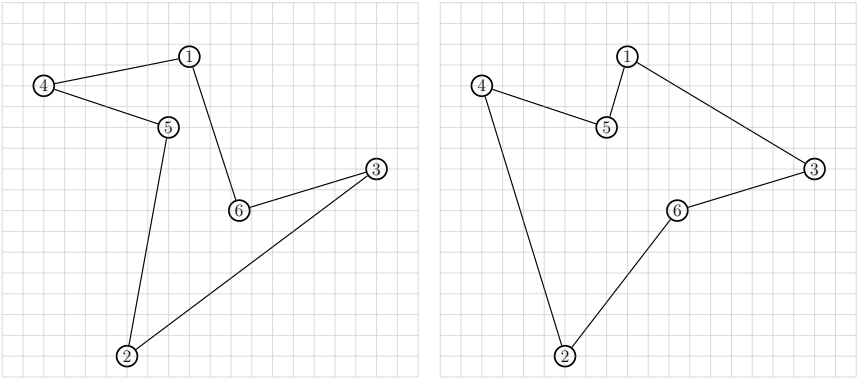
Cidade	1	2	3	4	5	6
1		245	174	118	59	129
2			250	226	186	147
3				274	169	114
4					105	185
5						87
6						

$147 + 226 + 105 + 59 = 825$ km, que é menor se comparada à solução da Figura 1.2a, tornando-a preferível.

1.1.1 Dificuldade

Uma maneira de determinar a solução com o menor custo de viagem para uma instância é examinar todas as soluções possíveis. Note que diferentes soluções podem ser vistas como permutações do conjunto de cidades. Além disso, (i) não importa de onde se inicia o *tour* (i.e., a sequência $(1, 3, 4, 5, 2, 1)$ é essencialmente igual à sequência $(2, 1, 3, 4, 5, 2)$); e (ii) percorrer um *tour* no sentido anti-horário é igual a percorrê-lo no sentido horário[3] (i.e., a sequência $(1, 2, 3, 4, 1)$ é igual à

[3] A versão do TSP abordada pressupõe que partir da cidade i para a cidade j é igual a partir da cidade j para a cidade i. A variante na qual isso não é necessariamente verdade chama-se ATSP (*Assymetrical Traveling Salesman Problem*).



(a) Exemplo de solução viável. (b) Exemplo de solução viável.

Figura 1.2: Exemplos de soluções viáveis.

sequência $(1, 4, 3, 2, 1)$).

Então, se n é o número de cidades de uma determinada instância, o número de soluções distintas é $(n-1)!/2$. Dessa forma, resolver a instância descrita anteriormente exigiria examinar $(5!)/2 = 60$ soluções. Contudo, o TSP possui aplicações[4] que envolvem instâncias com milhares (ou milhões) de cidades. Se $n \geq 60$, o número de soluções possíveis ultrapassa o número estimado de átomos no universo. Portanto, examinar cada uma das soluções para instâncias desse tipo é impraticável computacionalmente.

1.1.2 Abordagens

Há duas maneiras de resolver problemas de otimização combinatória como o TSP: os métodos exatos e os métodos heurísticos. O primeiro visa obter soluções comprovadamente ótimas, enquanto o segundo busca obter soluções boas em um tempo razoável. Normalmente, os métodos exatos costumam ser mais lentos, e seu desempenho depende da existência de bons limitantes superiores[5], que normalmente são providenciados pelos métodos heurísticos.

Embora o TSP seja um problema bem resolvido tanto de forma exata quanto heurística, ele serve como ponto de partida para o estudo de problemas mais avançados, como o Problema de Roteamento de Veículos e suas variantes, por compartilhar muitas de suas características com eles. O capítulo seguinte descreve

[4] O TSP é um subproblema de tarefas como a criação de trilhas que conectam pontos em placas de circuitos, sequenciamento de DNA, etc.

[5] A solução da Figura 1.2b, por exemplo, possui uma distância total de 825 km. Isso significa que uma solução ótima para essa instância não pode ter uma distância total maior que 825. Diz-se, portanto, que 825 é um "limite superior". Um método exato pode tirar proveito dessa informação. (*upper bound*).

uma abordagem heurística simples para resolver o TSP. A abordagem também é versátil, servindo como *framework* para a resolução de vários outros problemas de otimização combinatória.

1.2 Preliminares

O algoritmo heurístico descrito no próximo capítulo deve ser implementado na linguagem C++. Para facilitar na compreensão do problema e no processo de implementação, alguns conceitos e observações são apresentados a seguir.

1.2.1 Grafos

Sejam V e E conjuntos de vértices e arestas, respectivamente. O par $G = (V, E)$ é considerado um grafo. Mais precisamente, E é um conjunto de pares $\{i, j\}$ tais que $i, j \in V, i \neq j$. Assim, qualquer conjunto de vértices ligados por arestas pode ser considerado um grafo. Um exemplo de grafo é mostrado na Figura 1.3a. Nela, $V = \{1, 2, 3, 4, 5\}$ e $E = \{\{1, 4\}, \{1, 5\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{2, 4\}, \{3, 5\}\}$. Se todos os vértices estiverem conectados uns aos outros por arestas, G é um grafo completo. Um grafo completo com cinco vértices é ilustrado na Figura 1.3b.

Grafo com pesos

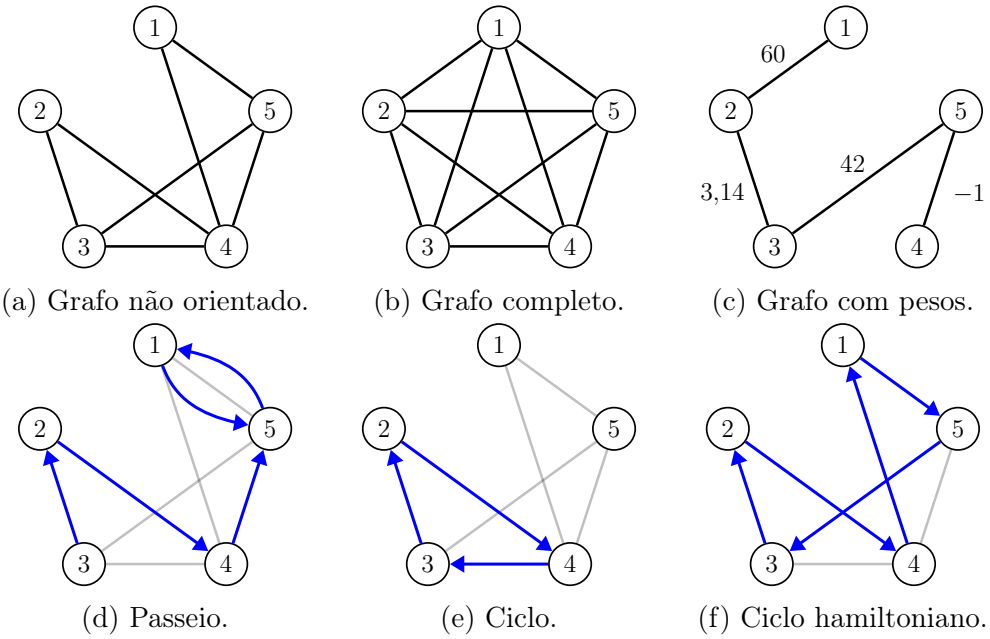
Se cada aresta $e = \{i, j\} \in E$ de um grafo estiver associada a um peso c_{ij} (um número real qualquer), diz-se que esse grafo é um grafo com pesos. A Figura 1.3c ilustra um grafo com pesos em que $c_{2,3} = 3{,}14$, $c_{1,2} = 60$, $c_{3,5} = 42$, e $c_{4,5} = -1$.

Passeio

Dado um grafo $G = (V, E)$, a sequência alternada de vértices e arestas $v_0 e_0 v_1 e_1 \ldots e_{k-1} v_k$ é considerada um passeio desde que toda aresta $e_i = \{v_i, v_{i+1}\}$ pertença a E. A Figura 1.3d contém um exemplo de um passeio percorrido sobre as arestas do grafo da Figura 1.3a. Observe que tanto vértices quanto arestas podem ser percorridos mais de uma vez.

Ciclo e ciclo Hamiltoniano

Um ciclo é um passeio que começa e termina no mesmo vértice. Se todos os vértices no grafo são visitados exatamente uma vez, o ciclo é dito hamiltoniano. As figuras 1.3e e 1.3f ilustram um ciclo e um ciclo hamiltoniano, respectivamente.



(a) Grafo não orientado. (b) Grafo completo. (c) Grafo com pesos.

(d) Passeio. (e) Ciclo. (f) Ciclo hamiltoniano.

Figura 1.3: Exemplos de grafos, caminhos e ciclos.

1.2.2 Representação do TSP através de grafos

Uma instância do TSP pode ser vista como um grafo completo com pesos, no qual cada nó corresponde a uma cidade, e cada aresta está associada a um custo que representa a distância entre as duas cidades correspondentes. Resolver o TSP equivale a encontrar o ciclo hamiltoniano nesse grafo cujo custo total (a soma dos pesos das arestas no ciclo) é o menor possível.

O grafo citado pode, ainda, ser representado por uma matriz M, na qual o elemento $M_{ij} = c_{ij}$ equivale ao peso da aresta $\{i, j\}$. Por conter informações sobre as arestas, que consistem em pares de vértices adjacentes, diz-se que M é uma matriz de adjacência.

1.2.3 Leitor de instâncias

Alguns arquivos de instâncias e o código de um leitor de instâncias se encontram neste link: https://github.com/cvneves/kit-opt/tree/master/GILS-RVND-TSP/leitor-instancias. O leitor computa a matriz de adjacência da instância, representando-a através do *array* bidimensional `matrizAdj`. Note que os custos são indexados a partir de 1. Logo, se uma instância contém 14 cidades, a distância entre a primeira e a última cidade é `matrizAdj[1][14]` (ou `matrizAdj[14][1]`).

1.2.4 *Standard Template Library*

O uso do paradigma de programação orientada a objetos não é obrigatório. Contudo, para facilitar a implementação das estruturas de dados e rotinas necessárias no algoritmo, recomenda-se o estudo dos componentes `vector` e `sort()` da *Standard Template Library* (STL). O trecho de código abaixo mostra como as soluções do TSP podem ser representadas.

```
1  typedef struct Solucao{
2      vector<int> sequencia;
3      double valorObj;
4  } Solucao;
5
6  // Exemplo de inicializacao
7  Solucao s1 = {{1,6,3,2,5,4,1}, 0};
8
9  // Iterando pelos vertices da solucao
10 void exibirSolucao(Solucao *s)
11 {
12     for(int i = 0; i < s->sequencia.size() - 1; i++)
13         std::cout << s->sequencia[i] << " -> ";
14     std::cout << s->sequencia.back() << std::endl;
15 }
16
17 exibirSolucao(&s1);
18 // RESULTADO: "1 -> 6 -> 3 -> 2 -> 5 -> 4 -> 1"
19
20 // Inicializando uma solucao vazia
21 Solucao s2 = {{}, 0.0};
22 // Adicionando vertices na solucao
23 s2.push_back(1);
24 s2.push_back(3);
25 s2.push_back(1);
26 exibirSolucao(&s2);
27 // RESULTADO: "1 -> 3 -> 1"
28 s2.insert(s2.begin() + 1, 4);
29 s2.insert(s2.end() - 1, 5);
30 s2.insert(s2.begin() + 2, 2);
31 exibirSolucao(&s2);
32 // RESULTADO: "1 -> 4 -> 2 -> 3 -> 5 -> 1"
33
34 void calcularValorObj(Solucao *s){
35     s->valorObj = 0;
36     for(int i = 0; i < s->sequencia.size() - 1; i++)
37         s->valorObj += matrizAdj[s->sequencia[i]][s->sequencia[i+1]];
38 }
```

1.2.5 Valores ótimos

O valor ótimo de custo das instâncias fornecidas já foi determinado por meio de métodos exatos. Uma lista com esses valores — que devem ser consultados para determinar a eficácia da implementação do algoritmo — se encontra neste link: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html`.

Depois de implementado, o algoritmo deve ser capaz de obter o valor ótimo para instâncias de até 300 cidades na maioria das vezes em que é executado[6].

[6] Além de não garantir a obtenção de uma solução ótima, o algoritmo não é determinístico. Por isso, determinar a robustez e velocidade de sua implementação exige testá-lo extensivamente.

2

Abordagem meta-heurística para o TSP

Uma meta-heurística é um método heurístico para resolver problemas de otimização combinatória de forma genérica. Uma única meta-heurística pode ser reutilizada para resolver vários problemas diferentes, exigindo, por vezes, pouco esforço de adaptação para cada um deles. Este capítulo descreve a meta-heurística *Iterated Local Search* (ILS), além das devidas adaptações necessárias para utilizá-la para resolver o TSP.

2.1 *Framework* ILS

Considere um problema de otimização combinatória qualquer. Assuma que o problema é de minimização. O código a seguir apresenta um algoritmo genérico para resolvê-lo:

```
1  Solution ILS(int maxIter, int maxIterIls)
2  {
3    Solution bestOfAll;
4    bestOfAll.cost = INFINITY;
5    for(int i = 0; i < maxIter; i++)
6    {
7      Solution s = Construcao();
8      Solution best = s;
9
10     int iterIls = 0;
11
12     while(iterIls <= maxIterIls)
13     {
14       BuscaLocal(&s);
15       if(s.cost < best.cost)
16       {
17         best = s;
18         iterIls = 0;
19       }
```

```
20        s = Perturbacao(best);
21        iterIls++;
22      }
23      if (best.cost < bestOfAll.cost)
24        bestOfAll = best;
25    }
26
27  return bestOfAll;
28  }
```

Primeiramente, algoritmo constrói uma solução baseando-se em palpites educados através do método Construção() (linha 7). Em seguida, tenta-se melhorar o custo dessa solução fazendo-se pequenas modificações através do método BuscaLocal() enquanto houver melhoras (linhas 14–18). Quando não for mais possível melhorar a solução, continua-se a partir de uma cópia levemente modificada da melhor solução da iteração corrente `best`, gerada pelo procedimento Perturbação() (linha 20). Continua-se até que um critério de parada seja atingido (linha 12). Se `best.cost` for menor que `bestOfAll.cost`, `best` passa a ser a melhor solução (linhas 23 e 24). Esse processo é repetido `maxIter` vezes (linha 5), e a melhor solução `bestOfAll` é retornada (linha 27).

Os procedimentos Construção(), BuscaLocal() e Perturbação() serão explicados em detalhes para o caso do TSP nas próximas seções. Recomenda-se que o leitor implemente-os na ordem em que são apresentados, para que então possa implementar o código acima.

2.2 Construção()

Uma heurística construtiva busca criar uma solução razoável para um problema de otimização. A solução criada pode então ser modificada e melhorada ao longo do tempo. Por esse motivo, é comum que o valor objetivo de soluções geradas por procedimentos esteja longe do valor ótimo. A heurística construtitva utilizada no método Construção() baseia-se no método *Greedy Randomized Adaptive Search Procedure* (GRASP) REF.

2.2.1 Inserção mais barata

Considere uma instância do TSP dada pelo grafo completo com pesos $G = (V, E)$. Considere, ainda, um subconjunto arbitrário de vértices $V' \subset V$, e que s' é um *tour* que visita todos os vértices de V'[1]. Seja $CL = V \setminus V'$ uma lista de candidatos a serem inseridos em s' de forma a obter um *tour* completo.

[1]Nesse caso, diz-se que s' é um *subtour*, já que visita apenas alguns vértices de V.

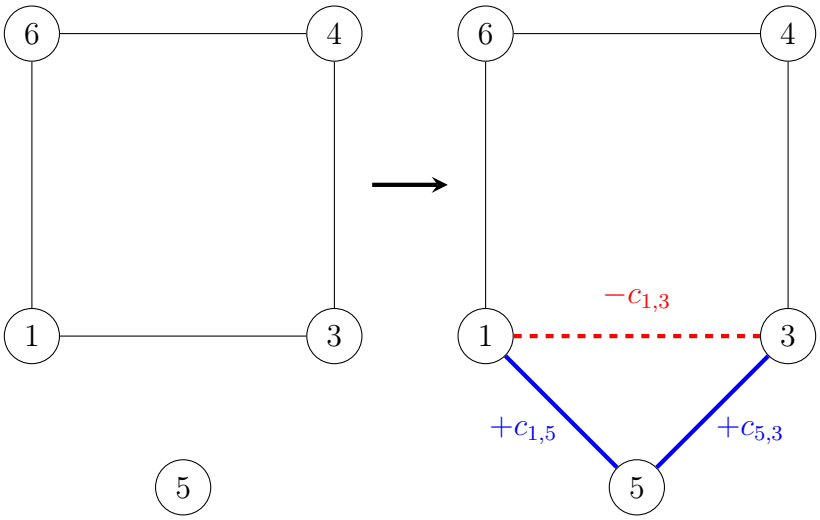


Figura 2.1: Inserção de um novo nó.

Analisemos o que ocorre quando um único vértice $k \in CL$ é inserido em s'. Note que para isso, é sempre necessário colocar k entre dois vértices adjacentes i e j. Essa operação implica na remoção da aresta $\{i, j\}$, e na adição de duas arestas $\{i, k\}$ e $\{k, j\}$. Suponha que s'' é um novo *subtour*, resultado da inserção de k em s'. A partir disso, deduz-se que o custo de inserção de k em s' é $\Delta = f(s'') - f(s') = c_{ik} + c_{kj} - c_{ij}$. Um exemplo dessa operação é mostrado na Figura 2.1, na qual o vértice 5 é inserido entre os vértices 1 e 3, resultando em um novo *subtour*.

É fácil perceber que escolher o vértice k e a aresta $\{i, j\}$ de forma a minimizar Δ significa diminuir o prejuízo no valor objetivo após a inserção. No entanto, escolher sempre o par $(k, \{i, j\})$ que minimiza Δ pode "viciar" o método, levando-o sempre a soluções parecidas, o que pode dificultar as buscas locais. Portanto, convém introduzir um pouco de aleatoriedade na escolha do par. Baseando-se nessas ideias, pode-se enumerar as etapas do método Construção() como segue.

1. Construir uma solução parcial s' de forma aleatória;
2. $V' \leftarrow$ vértices de s' e $CL \leftarrow V \setminus V'$;
3. Computar os pares $(k, \{i, j\})$ para todo $k \in CL$, $\{i, j\} \in s'$ e armazená-los em uma lista Ω;
4. Ordenar os pares em Ω em ordem crescente de Δ;
5. $\alpha \leftarrow$ número aleatório no intervalo $[0, 1]$;

6. Selecionar aleatoriamente um dos $\lfloor \alpha \times |\Omega| \rfloor$ primeiros pares em Ω;

7. Sendo $(k, \{i, j\})$ o par escolhido, retirar a aresta $\{i, j\}$ de s' e inserir o vértice k entre i e j;

8. Remover k da lista de candidatos CL;

9. Se CL estiver vazio, retornar s'. Senão, voltar para o passo 3.

É suficiente construir o *subtour* inicial do passo 1 com 3 vértices. Para isso, pode-se iniciar s' sempre como `s' = {1, 1}` e inserir 3 vértices de forma aleatória. Assim, o *subtour* inicial da Figura 2.1, por exemplo, seria `s' = {1,6,4,3,1}`.

Um exemplo de implementação do procedimento Construção() pode ser visto a seguir:

```
1  struct InsertionInfo
2  {
3    int noInserido; // no k a ser inserido
4    int arestaRemovida; // aresta {i,j} na qual o no k sera inserido
5    double custo; // delta ao inserir k na aresta {i,j}
6  };
7
8  std::vector<InsertionInfo> calcularCustoInsercao(Solution& s, std::vector<
       int>& CL)
9  {
10   std::vector<InsertionInfo> custoInsercao = std::vector<InsertionInfo>
         custoInsercao((s.size() - 1) * CL.size());
11   int l = 0;
12   for(int a = 0; a < s.sequence.size() - 1; a++) {
13     int i = s.sequence[a];
14     int j = s.sequence[a + 1];
15     for (auto k : CL) {
16       custoInsercao[l].custo = c[i][k] + c[j][k] - c[i][j];
17       custoInsercao[l].noInserido = k;
18       custoInsercao[l].arestaRemovida = a;
19       l++;
20     }
21   }
22   return custoInsercao;
23 }
24
25 Solution Construcao()
26 {
27   Solution s;
28   s.sequence = escolher3NosAleatorios();
29   std::vector<int> CL = nosRestantes();
30   /* Ex: V = {1,2,3,4,5,6,7,8,9,10}
31      s.sequence = {1,2,9,5,1}
```

```
32      CL = {3,4,6,7,8,10} */
33    while(!CL.empty()) {
34      std::vector<InsertionInfo> custoInsercao = calcularCustoInsercao(s, CL
            );
35      ordenarEmOrdemCrescente(custoInsercao);
36      double alpha = (double) rand() / RAND_MAX;
37      int selecionado = rand() % ((int) ceil(alpha * custoInsercao.size()));
38      inserirNaSolucao(s, custoInsercao[selecionado].k);
39    }
40
41      return s;
42  }
```

No exemplo, armazena-se as informações de cada par $(k, \{i, j\})$ em uma estrutura do tipo `InsertionInfo`. Os custos de inserção de cada par $(k, \{i, j\})$ são calculados pela função `calcularCustoInsercao()`, que é chamada na função `Construcao()` até que CL esteja vazio.

2.3 BuscaLocal()

A etapa de busca local possui como objetivo melhorar a solução corrente no decorrer da execução do algoritmo. O procedimento é feito modificando-se as soluções e avaliando o impacto das modificações na função objetivo. Antes de descrever o procedimento BuscaLocal(), alguns conceitos são apresentados nas subseções que seguem.

2.3.1 Movimentos

Seja s uma solução viável qualquer para um problema de otimização, e m uma operação que altera s de alguma maneira. Diz-se que m é um "movimento", e que $s' = s \oplus m$ é uma nova solução, obtida após executá-lo. Há varios tipos de movimentos possíveis, que dependem, por sua vez, do problema de otimização em questão. Um exemplo de movimento para o TSP é trocar as posições de dois vértices na sequência.

2.3.2 Estruturas de vizinhança

Sejam s uma solução qualquer, e M_k um conjunto de movimentos estruturalmente parecidos. O conjunto $\mathcal{N}_k(s)$ contém todas as posíveis soluções que se poderia obter executando em s os movimentos do conjunto M_k. Já que as soluções s e $s' \in \mathcal{N}_k(s)$ diferem em apenas um movimento, diz-se que elas são "vizinhas". Por esse motivo, $\mathcal{N}_k$ é considerado uma "estrutura de vizinhança".

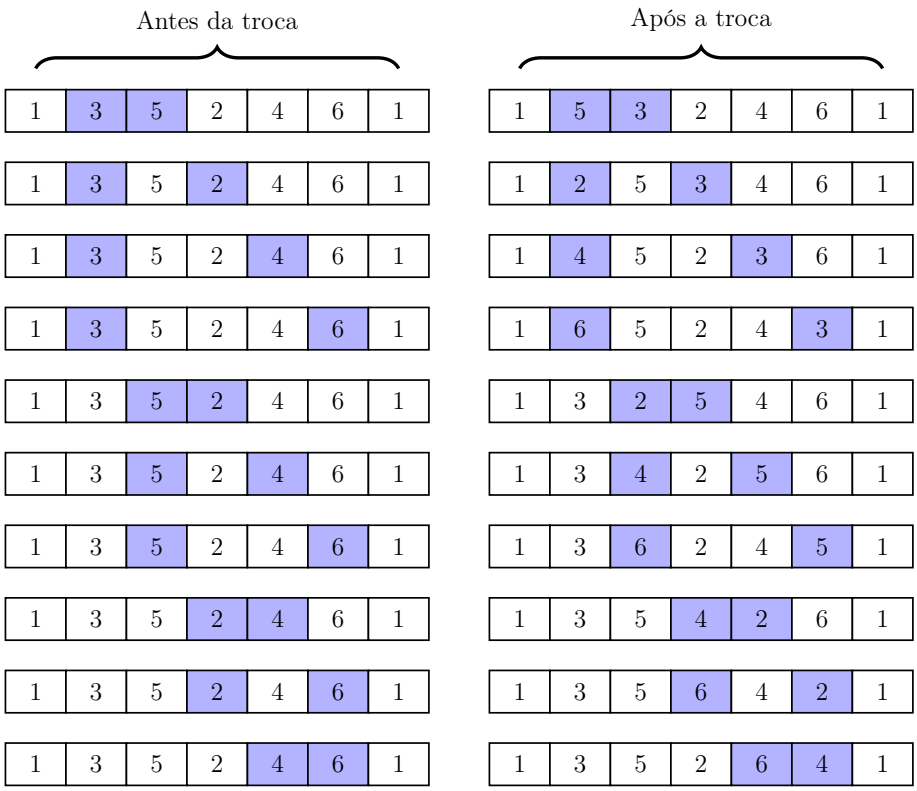


Figura 2.2: Exemplo da estrutura de vizinhança *swap*.

Para compreender melhor o conceito, consulte a Figura 2.2. Na figura, todas as combinações possíveis de movimentos da estrutura de vizinhança SWAP, que envolve trocar a posição de dois vértices quaisquer na sequência, foram listadas para a solução

$$s = (1, 3, 5, 2, 4, 6, 1).$$

2.3.3 *Best Improvement*

Dada uma solução s e uma estrutura de vizinhança $\mathcal{N}_k$, o método do melhor aprimoramento (*best improvement*) visa encontrar o vizinho de s com o menor custo possível. Em outras palavras, deseja-se encontrar o "melhor vizinho" de s considerando-se a estrutura de vizinhança $\mathcal{N}_k$. Se $f(s^*) < f(s)$, diz-se que houve uma melhora, e convém substituir s por s^*. O trecho de código a seguir mostra como utilizar o método *best improvement* para explorar toda a estrutura de vizinhança *swap*:

```
1 bool bestImprovementSwap(Solution *s)
2 {
```

```
3    double bestDelta = 0;
4    int best_i, best_j;
5    for(int i = 1; i < s->sequence.size() - 1; i++)
6    {
7        int vi = s->sequence[i];
8        int vi_next = s->sequence[i + 1];
9        int vi_prev = s->sequence[i - 1];
10     for(int j = i + 1; j < s.sequence.size() - 1; j++)
11     {
12           int vj = s->sequence[j];
13           int vj_next = s->sequence[j + 1];
14           int vj_prev = s->sequence[j - 1];
15           double delta = -c[vi_prev][vi] -c[vi][vi_next] + c[vi_prev][vj]
16                      + c[vj][vi_next] - c[vj_prev][vj] - c[vj][vj_next]
17                      + c[vj_prev][vi] + c[vi][vj_next];
18
19         if (delta < bestDelta)
20         {
21           bestDelta = delta;
22           best_i = i;
23           best_j = j;
24         }
25     }
26   }
27
28   if(bestDelta < 0)
29   {
30     std::swap(s->sequence[best_i], s->sequence[best_j]);
31     s->cost = s->cost + bestDelta;
32     return true;
33   }
34   return false;
35 }
```

No código , enumera-se todos os movimentos de troca entre dois nós possíveis, e o impacto de cada movimento na função objetivo é avaliado nas linhas 15–17. Note que não é necessário recalcular o custo da solução do zero, pois o custo de uma troca pode ser calculado com uma simples fórmula desde que as arestas a serem removidas e inseridas sejam conhecidas[2].

2.3.4 Estruturas de vizinhaça utilizadas

As estruturas de vizinhança que devem ser utilizadas são descritas como segue.

- SWAP — $\mathcal{N}_1$: Troca a posição de dois vértices na sequência;

[2]Levar isso em consideração é crucial para o bom desempenho do algoritmo.

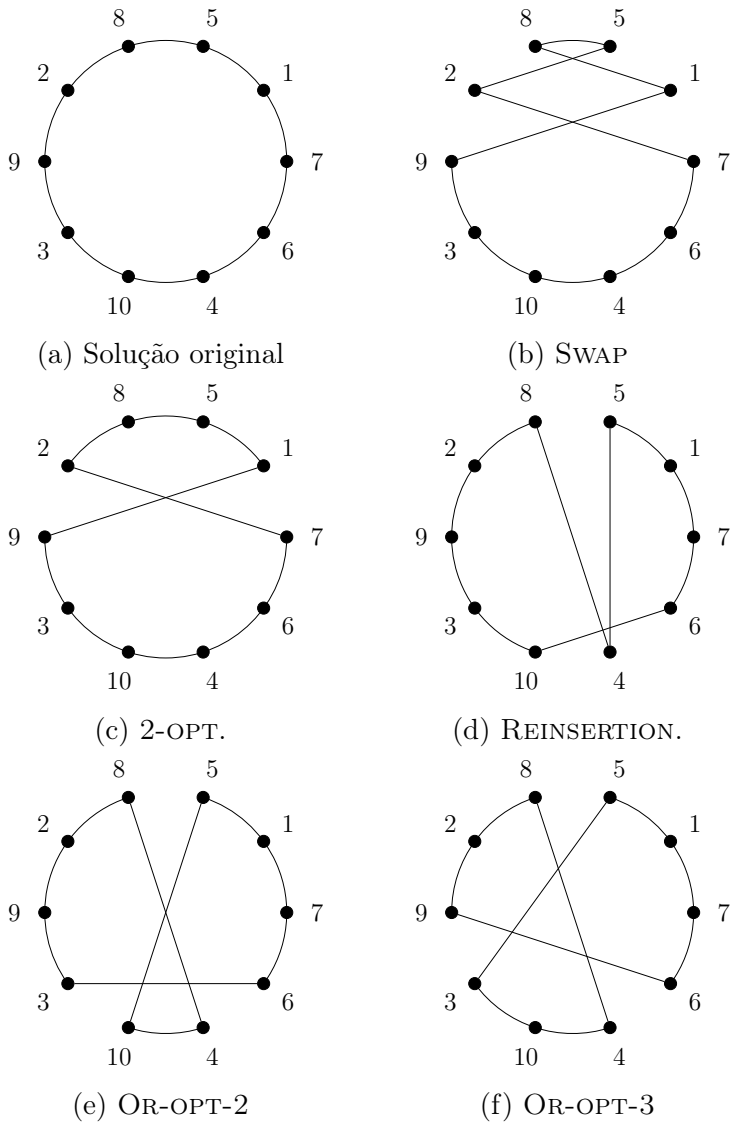
- 2-OPT — $\mathcal{N}_2$: Duas arestas não adjacentes da solução são removidas e o segmento entre elas é reinserido de maneira invertida, adicionando-se duas novas arestas para reconstruir a solução.
- REINSERTION — $\mathcal{N}_3$: Um único vértice é retirado de sua posição e inserido em outra;
- OR-OPT-2 — $\mathcal{N}_4$: Um bloco composto por dois vértices adjacentes é retirado de sua posição e inserido em outra;
- OR-OPT-3 — $\mathcal{N}_5$: Um bloco composto por três vértices adjacentes é retirado de sua posição e inserido em outra;

A Figura 2.3 apresenta exemplos de movimentos de cada uma das estruturas aplicados em uma solução com 10 vértices.

2.3.5 RVND

O procedimento BuscaLocal() consiste em uma implementação do método *Random Variable Neighborhood Descent* (RVND) utilizando-se as estruturas de vizinhança mencionadas. A ideia do procedimento é utilizar o método *best improvement* em diferentes estruturas de vizinhança (escolhidas de forma aleatória) enquanto houver melhora, eliminando estruturas de vizinhança que não provocaram melhoras. Isso pode ser visto no seguinte trecho de código:

```
1  void BuscaLocal(Solution *s)
2  {
3    std::vector<int> NL = {1, 2, 3, 4, 5};
4    bool improved = false;
5
6    while (NL.empty() == false)
7    {
8      int n = rand() % NL.size();
9      switch (NL[n])
10     {
11     case 1:
12       improved = bestImprovementSwap(s);
13       break;
14     case 2:
15       improved = bestImprovement2Opt(s);
16       break;
17     case 3:
18       improved = bestImprovementOrOpt(s, 1); // Reinsertion
19       break;
20     case 4:
21       improved = bestImprovementOrOpt(s, 2); // Or-opt2
```



(a) Solução original (b) Swap

(c) 2-opt. (d) Reinsertion.

(e) Or-opt-2 (f) Or-opt-3

Figura 2.3: Exemplos das estruturas de vizinhança.

```
22        break;
23      case 5:
24        improved = bestImprovementOrOpt(s, 3); // Or-opt3
25        break;
26      }
27
28      if (improved)
29        NL = {1, 2, 3, 4, 5};
30      else
31        NL.erase(NL.begin() + n);
32    }
33  }
```

No código, o procedimento BestImprovement() deve ser implementado para cada uma das 5 estruturas de vizinhança. A implementação de cada um dos procedimentos é semelhante ao exemplo mostrado anteriormente para a estrutura de vizinhança SWAP.

Já que as estruturas de vizinhança REINSERTION, OR-OPT-2 e OR-OPT-3 são muito parecidas, encoraja-se que sejam implementadas através de uma única função, que possui como um de seus parâmetros um número de 1 a 3, que representa o tamanho do bloco a ser movido.

2.4 Perturbação()

Uma solução é um "ótimo local" se ela não pode mais ser melhorada pelo procedimento BuscaLocal(). O objetivo do procedimento Perturbação() é modificar levemente soluções desse tipo. Embora uma solução obtida após uma perturbação aleatória seja quase sempre pior que a original, espera-se que ela possa ser melhorada através do procedimento BuscaLocal(), conforme ilustrado na Figura 2.4.

Para facilitar a visualização, a função objetivo $f(s)$ na Figura 2.4 foi representada por meio de uma curva contínua, enquanto o eixo horizontal representa o espaço de soluções possíveis. As linhas tracejadas delimitam as soluções vizinhas que as buscas locais conseguem "enxergar". Ao realizar uma busca local nas vizinhanças de s_0, obtém-se a solução s_1, que é um ótimo local. A solução s_1 é então perturbada, resultando em uma solução ligeiramente pior s_2. Porém, ao realizar uma busca local nas vizinhanças de s_2, obtém-se uma nova solução s_3, que é o mínimo global da função.

Neste caso, o procedimento Perturbação() consiste na execução de um movimento chamado *double bridge*. O movimento troca as posições de dois segmentos da sequência. As figuras 2.5a e 2.5b ilustram um exemplo de uma solução antes e após esse movimento.

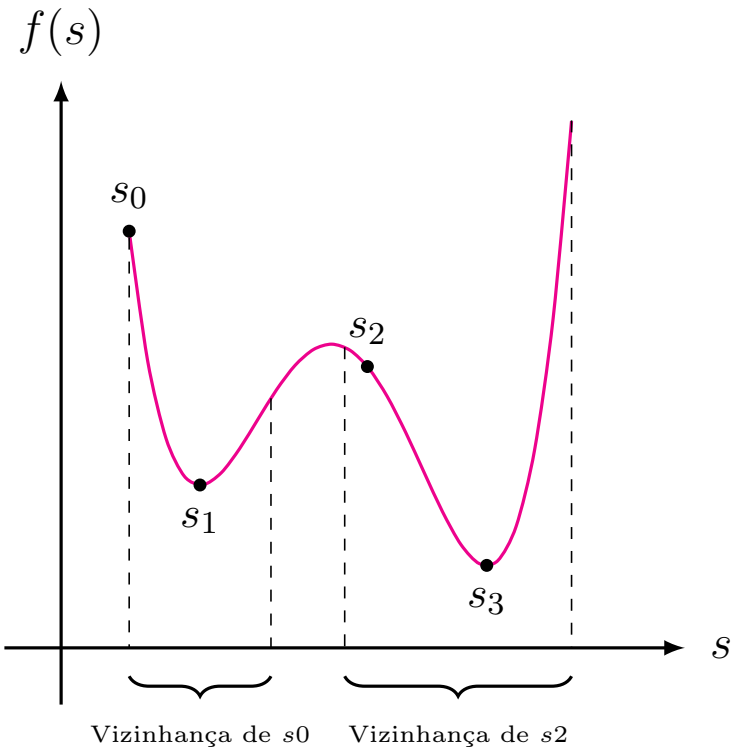


Figura 2.4: Visualização da busca local.

Uma chamada ao procedimento Perturbação(s) deve executar os seguintes passos:

1. $s' \leftarrow s$;
2. Escolher aleatoriamente dois segmentos não sobrepostos de s' de tamanhos entre 2 e $\lceil |V|/10 \rceil$;
3. Trocar a posição dos segmentos em s';
4. Retornar s'.

1	14	5	4	8	7	13	6	12	3	10	2	11	9	1

(a) Solução antes da perturbação.

1	14	3	10	2	11	8	7	13	6	12	5	4	9	1

(b) Solução após a perturbação.

Figura 2.5: Exemplo do movimento *double bridge* em uma sequência.

2.5 Parâmetros

Os parâmetros `maxIter` e `maxIterIls` influenciam no desempenho no algoritmo tanto em termos de tempo, quanto na qualidade das soluções. Neste caso, os valores dos parâmetros devem ser inicializados da seguinte forma:

1. `maxIter` $\leftarrow 50$;
2. `maxIterILS` $\leftarrow \begin{cases} |V|/2 \text{ se } |V| \geq 150 \\ |V| \text{ senão.} \end{cases}$

2.6 *Benchmark*

A Tabela 2.1 apresenta os valores médios de custo e tempo (em segundos) de resolução de cada instância após 10 execuções do algoritmo em um Intel® Core™ i7-3770 3.40GHz.

Tabela 2.1: Tempo e custo médios obtidos para cada instância.

Instância	**Resultados**		**Instância**	**Resultados**	
	Tempo	**Custo**		**Tempo**	**Custo**
a280	96,623	2579	kroA150	11,751	26524
ali535	1525	202384	kroA200	32,951	29368
att48	0,3	10628	kroB100	3,748	22141
att532	1778,96	27731	kroB150	10,634	26130
bayg29	0,043	1610	kroB200	35,53	29437,2
bays29	0,05	2020	kroC100	3,568	20749
berlin52	0,374	7542	kroD100	4,114	21294
bier127	10,209	118282	kroE100	3,745	22068
brazil58	0,479	25395	lin105	4,355	14379
brg180	12,824	1950	lin318	188,78	42045,7
burma14	0,004	3323	linhp318	187,536	42053,1
ch130	10,91	6110	pcb442	597,431	50876
ch150	10,43	6528	pr107	4,582	44303
d198	33,639	15780	pr124	7,021	59030
d493	1132,48	35042	pr136	13,632	96772
dantzig42	0,161	699	pr144	10,479	58537
eil101	4,436	629	pr152	8,708	73682
eil51	0,369	426	pr226	45,27	80369
eil76	1,549	538	pr264	64,758	49135
fl417	365,503	11861	pr299	130,098	48194,8
fri26	0,033	937	pr76	1,366	108159
gil262	82,271	2378,7	rat195	28,046	2326,1
gr120	9,065	6942	rat99	4,115	1211
gr137	11,348	69853	rd100	3,983	7910
gr17	0,008	2085	rd400	498,288	15296,1
gr202	37,105	40160,1	si175	17,333	21407
gr21	0,014	2707	si535	758,534	48466,8
gr229	61,498	134613	st70	1,03	675
gr24	0,028	1272	swiss42	0,155	1273
gr431	721,745	171530	ts225	28,869	126643
gr48	0,314	5046	tsp225	45,368	3916
gr96	3,475	55209	u159	10,828	42080
hk48	0,336	11461	ulysses16	0,008	6859
kroA100	3,468	21282	ulysses22	0,019	7013

3

Concatenação de subsequências

O Problema da Mínima Latência (*Minimum Latency Problem*, MLP) REF e o Problema de Roteamento de Veículos com Janelas de Tempo (*Vehicle Routing Problem with Time Windows*, VRPTW) REF são exemplos problemas de otimização combinatória mais difíceis que o TSP. Parte dessa dificuldade reside em características especiais desses problemas, seja em termos de função objetivo ou de restrições. Tais diferenças tornam dificil avaliar o impacto de modificações na solução. No MLP, por exemplo, a aplicação de um movimento como o *swap* pode alterar drasticamente o seu valor objetivo, que não pode mais ser calculado apenas somando e subtraindo custos de arestas, como no caso do TSP. No VRPTW, aplicar o mesmo movimento pode violar restrições do problema, exigindo uma avaliação prévia da sua viabilidade, que pode demandar muitas operações.

Reavaliar o custo de movimentos ou checar a viabilidade de novas soluções durante as buscas locais de forma ingênua pode comprometer e, por vezes, inviabilizar o método. Este capítulo aborda o método de concatenação de subsequências, que permite realizar essas avaliações de maneira eficiente para uma vasta classe de problemas. Para isso, a aplicação do método será apresentada para o MLP e o VRPTW, definidos nas seções seguintes. Ao fim do capítulo, espera-se que o leitor seja capaz de empregar as mesmas técnicas do capítulo anterior para esses problemas de forma eficiente.

3.1 MLP

Seja $G = (V, E)$ um grafo em que cada aresta $\{\sigma_i, \sigma_j\} \in E$ está associada a um custo $t_{\sigma_i\sigma_j}$. Suponha, ainda, que $|V| = n$ e que o *tour* $s = (\sigma_1, \ldots, \sigma_{n+1})$ é a sequência de vértices de um ciclo hamiltoniano em G, que começa e termina no nó $\sigma_1 = \sigma_{n+1}$. Considera-se $l(i) = \sum_{j=1}^{i-1} t_{\sigma_j\sigma_{j+1}}$ a "latência" i-ésimo nó da sequência s. Em outras palavras, se $t_{\sigma_i\sigma_j}$ é o tempo de viagem entre os nós σ_i e σ_j, então $l(i)$ pode ser pensado como o tempo total de viagem do início do *tour* σ_1 até o nó

σ_i. Assim, se $s = (1, 3, 6, 2, 4, 7, 5, 1)$, por exemplo, então

$$l(4) = t_{13} + t_{36} + t_{62}$$

é a latência do nó $\sigma_4 = 2$. O MLP visa encontrar um ciclo hamiltoniano s em G cuja latência total — ou custo acumulado — $f(s) = \sum_{i=1}^{n+1} l(i)$ é mínima.

Para compreender a motivação do problema, considere que um motorista deseja realizar uma sequência de entregas para n clientes distintos. Para encontrar uma rota satisfatória em termos de tempo de viagem, o entregador poderia tratar o seu problema como um TSP, visando minimizar o custo total da própria viagem. Para satisfazer os clientes (i.e., realizar as entregas em um tempo razoável para todos eles), por outro lado, o entregador poderia tratar o problema como um MLP, tentando minimizar o tempo de entrega de cada um. Nesse caso, a latência $l(i)$ pode ser vista como o tempo necessário para realizar a entrega do i-ésimo cliente. Portanto, numa abordagem mais "altruísta", o MLP busca minimizar a soma de todos os tempos de entrega (latências).

Por ter uma estrutura similar à do TSP, o problema pode ser abordado com o mesmo método baseado em busca local do capítulo anterior. Observe, porém, que a aplicação de qualquer um dos movimentos mostrados acarretaria drásticas mudanças no custo acumulado. Isso ocorre porque, diferente do caso do TSP, modificar um segmento no *tour* implica em alterar a latência de todos os nós seguintes, que dependem diretamente da disposição dos nós anteriores. Dito isso, as próximas seções descrevem uma maneira de recalcular o custo acumulado em um número constante de operações.

3.1.1 Subsequências

Seja $s = (\sigma_1, \ldots, \sigma_{n+1})$ uma solução viável para uma instância do MLP. Uma subsequência é uma sequência $\sigma = (\sigma_i, \ldots, \sigma_j)$, em que $i, j \in \{1, \ldots, n+1\}$. Além disso, considera-se $C(\sigma) = \sum_{k=i}^{j} l(\sigma_k)$ o custo acumulado da subsequência σ. Note que o número total de subsequências associadas a s, dentre as quais constam até mesmo sequências compostas por um único nó, é $(n+1)^2$.

3.1.2 Concatenação de subsequências

Sejam $\sigma' = (\sigma'_1, \ldots, \sigma'_a)$ e $\sigma'' = (\sigma''_1, \ldots, \sigma''_b)$ duas subsequências. A subsequência

$$\begin{aligned} \sigma &= \sigma' \oplus \sigma'' \\ &= (\sigma'_1, \ldots, \sigma'_a, \sigma''_1, \ldots, \sigma''_b) \end{aligned}$$

é resultado da concatenação de σ' e σ''. A utilidade da concatenação de subsequências pode ser vista na Figura 3.1, na qual o movimento *swap* é aplicado entre os

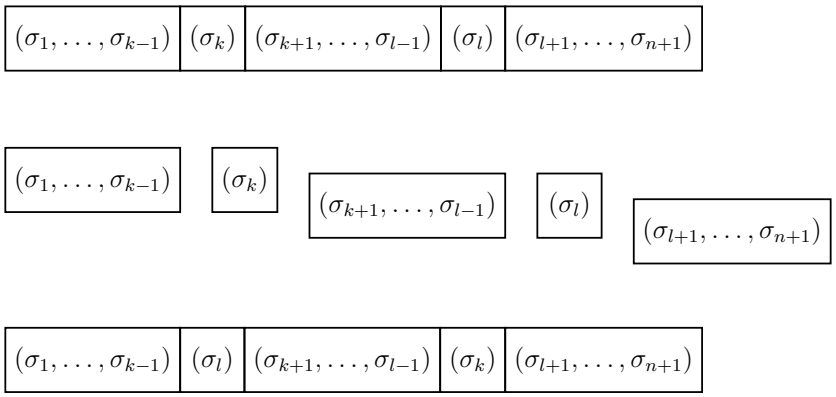


Figura 3.1: Quebra de uma solução em 5 subsequências e reconstrução durante o movimento *swap*.

nós σ_k e σ_l de uma solução. A aplicação do movimento pode ser compreendida como a "quebra" da sequência da solução em um número fixo de subsequências — 5 neste caso —, que são então rearranjadas e concatenadas para formar uma nova solução. Todos os outros movimentos mostrados no capítulo anterior também podem ser vistos da mesma forma[1]. Em outras palavras, se uma maneira eficiente de calcular o custo acumulado de uma subsequência obtida após a concatenação de outras duas subsequências for conhecida, o impacto dos movimentos de várias estruturas de vizinhança herdadas do TSP também poderá ser facilmente avaliado, permitindo que sejam reutilizadas.

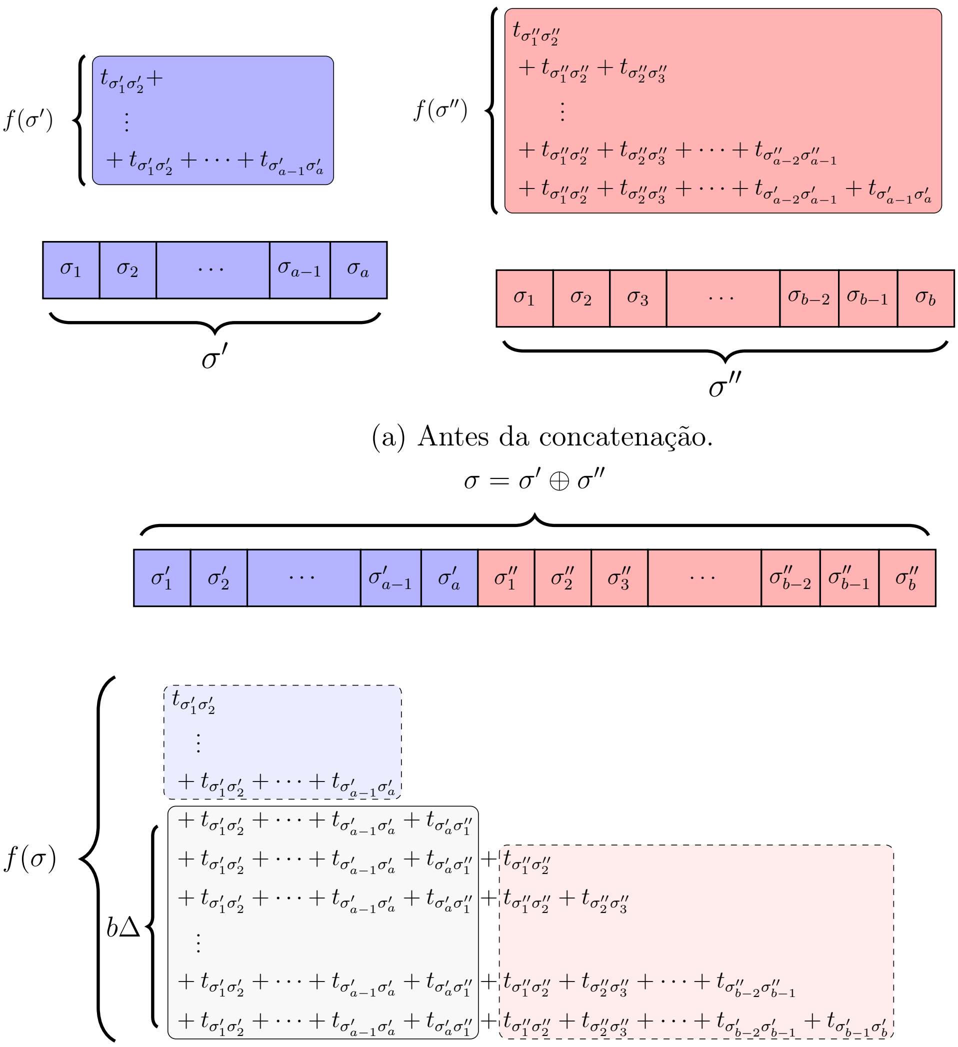
Para compreender como isso pode ser feito, observe a Figura 3.2, que ilustra a concatenação de duas subsequências σ' e σ'' de tamanhos arbitrários $a \geq 1$ e $b \geq 1$, respectivamente. Na Figura 3.2a, as duas subsequências, bem como seus custos acumulados $f(\sigma')$ e $f(\sigma'')$, escritos de forma explícita, são exibidos. Enquanto isso, o custo acumulado da subsequência $\sigma = \sigma' \oplus \sigma''$ é mostrado de forma similar na Figura 3.2b. Note que tanto $f(\sigma')$ quanto $f(\sigma'')$ reaparecem no cálculo de $f(\sigma)$, sendo acompanhados pelo termo

$$\Delta = t_{\sigma'_1\sigma'_2} + \cdots + t_{\sigma'_{a-1}\sigma'_a} + t_{\sigma'_a\sigma''_1},$$

que aparece b vezes na soma. Isso significa que se $f(\sigma')$, $f(\sigma'')$ e Δ forem conhecidos previamente, pode-se calcular o custo acumulado da subsequência resultante da concatenação como

$$f(\sigma) = f(\sigma') + b\Delta + f(\sigma'').$$

[1]Observe que as subsequências podem ter a sua ordem invertida. Isso ocore no caso do movimento *2-opt*, que divide a solução em três subsequências, inverte uma delas, e as concatena novamente para formar uma nova solução.



(a) Antes da concatenação.

(b) Após a concatenação.

Figura 3.2: Concatenação de duas subsequências σ' e σ''.

3.1.3 Estruturas auxiliares

Sejam $s = (\sigma_1, \ldots, \sigma_{n+1})$ uma solução viável qualquer para uma instância do MLP, e σ uma subsequência de s. Além de seu custo acumulado $C(\sigma)$, a subsequência σ também é caracterizada pelos valores auxiliares $W(\sigma)$ (custo de atraso) e $T(\sigma)$ (duração). Caso σ seja uma subsequência composta por um único nó,

$$T(\sigma) = 0, \tag{3.1}$$
$$C(\sigma) = 0, \tag{3.2}$$
$$W(\sigma) = \begin{cases} 0 \text{ se } \sigma = (\sigma_1), \\ 1 \text{ caso contrário.} \end{cases} \tag{3.3}$$

Se, por outro lado, σ for resultado da concatenação de outras duas subsequências σ' e σ'', então

$$T(\sigma' \oplus \sigma'') = T(\sigma') + t_{\sigma'_a \sigma''_1} + T(\sigma''), \tag{3.4}$$
$$C(\sigma' \oplus \sigma'') = C(\sigma') + W(\sigma'') \left(T(\sigma') + t_{\sigma'_a \sigma''_1}\right) + C(\sigma''), \tag{3.5}$$
$$W(\sigma' \oplus \sigma'') = W(\sigma') + W(\sigma''). \tag{3.6}$$

Conforme explicado na seção anterior, a equação (3.5) descreve o cálculo do custo acumulado da subsequência resultante a partir dos custos das duas subsequências envolvidas na operação de concatenação. Dessa forma, se $T(\sigma)$, $C(\sigma)$ e $W(\sigma)$ forem conhecidos previamente para todas as subsequências σ possíveis de uma solução s, qualquer outra solução vizinha s' obtida aplicando-se um dos movimentos do capítulo anterior pode ter seu custo facilmente avaliado através de algumas operações de soma.

Representação das estruturas auxiliares

O trecho de código a seguir mostra como as informações relevantes de uma subsequência podem ser armazenados. Além dos valores já mostrados, o primeiro e último nós (`first` e `last`, respectivamente) de cada subsequência também devem ser armazenados para que se possa aplicar as equações (3.4) e (3.5). A partir de duas subsequências `sigma_1` e `sigma_2`, a função `Concatenate()` aplica as equações (3.4)–(3.6) e retorna a subsequência resultante `sigma`.

```
1  struct Subsequence
2  {
3      double T, C;
4      int W;
5      int first, last; // primeiro e ultimo nos da subsequencia
6      inline static Subsequence Concatenate(Subsequence &sigma_1,
           Subsequence &sigma_2)
```

```
7       {
8           Subsequence sigma;
9           double temp = t[sigma_1.last][sigma_2.first];
10          sigma.W = sigma_1.W + sigma_2.W;
11          sigma.T = sigma_1.T + temp + sigma_2.T;
12          sigma.C = sigma_1.C + sigma_2.W * (sigma_1.T + temp) + sigma_2.C;
13          sigma.first = sigma_1.first;
14          sigma.last = sigma_2.last;
15
16          return sigma;
17      }
18  };
```

Inicialização das estruturas auxiliares

O trecho de código a seguir mostra como os valores de todas as subsequências de uma solução podem ser inicializados. Os valores auxiliares de cada subsequência de uma solução são armazenados em `subseq_matrix`. O elemento `subseq_matrix[i][j]` armazena as informações da subsequência que começa no i-ésimo nó e termina no j-ésimo nó de `s`. No código, primeiramente, aplica-se as equações (3.1)–(3.3) para que as subsequências formadas por apenas um nó sejam obtidas. Em seguida, as outras subsequências são obtidas por meio de operações de concatenação.

```
1   // n: numero de nos da instancia
2   // s: solucao corrente
3   // subseq_matrix = vector<vector<Subsequence>>(n, vector<Subsequence>(n));
4
5   void UpdateAllSubseq(Solution *s, vector<vector<Subsequence>> &
        subseq_matrix)
6   {
7       int n = s->sequence.size();
8
9       // subsequencias de um unico no
10      for (int i = 0; i < n; i++)
11      {
12          int v = s->sequence[i];
13          subseq_matrix[i][i].W = (i > 0);
14          subseq_matrix[i][i].C = 0;
15          subseq_matrix[i][i].T = 0;
16          subseq_matrix[i][i].first = s->sequence[i];
17          subseq_matrix[i][i].last = s->sequence[i];
18      }
19
20      for (int i = 0; i < n; i++)
21          for (int j = i + 1; j < n; j++)
```

```
22              subseq_matrix[i][j] = Subsequence::Concatenate(subseq_matrix[i
                    ][j-1], subseq_matrix[j][j]);
23
24      // subsequencias invertidas
25      // (necessarias para o 2-opt)
26      for (int i = n - 1; i >= 0; i--)
27          for (int j = i - 1; j >= 0; j--)
28              subseq_matrix[i][j] = Subsequence::Concatenate(subseq_matrix[i
                    ][j+1], subseq_matrix[j][j]);
29  }
```

Após construir a matriz de subsequências a partir de uma solução inicial obtida através de um procedimento construtivo, pode-se, enfim, calcular o impacto de movimentos de maneira eficiente. Isso pode ser visto no trecho de código a seguir, no qual o custo de uma solução obtida após a aplicação do movimento *2-opt* foi obtido utilizando-se duas operações de concatenação. O valor `sigma_2.C` equivale ao custo acumulado da nova solução, que pode então ser comparado ao custo da solução anterior.

```
1  Solution *s = Construction();
2  subseq_matrix = vector<vector<Subsequence>>(n, vector<Subsequence>(n));
3  UpdateAllSubseq(s, subseq_matrix);
4  // movimento 2-opt remove as arestas {i-1, i} e {j, j+1},
5  // dividindo a solucao em tres subsequencias.
6  // no 2-opt a subsequencia que comeca em i e termina em j e invertida
7  // por isso, ela foi acessada por meio de subseq_matrix[j][i], e nao
       subseq_matrix[i][j]
8  Subsequence sigma_1 = Subsequence::Concatenate(subseq_matrix[0][i-1],
       subseq_matrix[j][i]);
9  Subsequence sigma_2 = Subsequence::Concatenate(sigma_1, subseq_matrix[j
       +1][n]);
10 if (sigma_2.C < s->cost)
11 {
12     ApplyTwoOpt(s, i, j);
13     // A solucao foi modificada,
14     // logo, as subsequencias devem ser recalculadas
15     UpdateAllSubseq(s, subseq_matrix);
16 }
```

3.1.4 Atualização de subsequências

Sempre que uma nova solução é criada, a matriz de subsequências deve ser computada a partir do zero, exigindo $(n+1)^2 - n - 1$ operações de concatenação. Quando uma solução é modificada por um movimento, no entanto, algumas subsequências permanecem inalteradas. Isso pode ser visto mais facilmente na Figura 3.3, que mostra a matriz de subsequências de uma solução $s = (\sigma_1, \ldots, \sigma_{15})$ alterada por

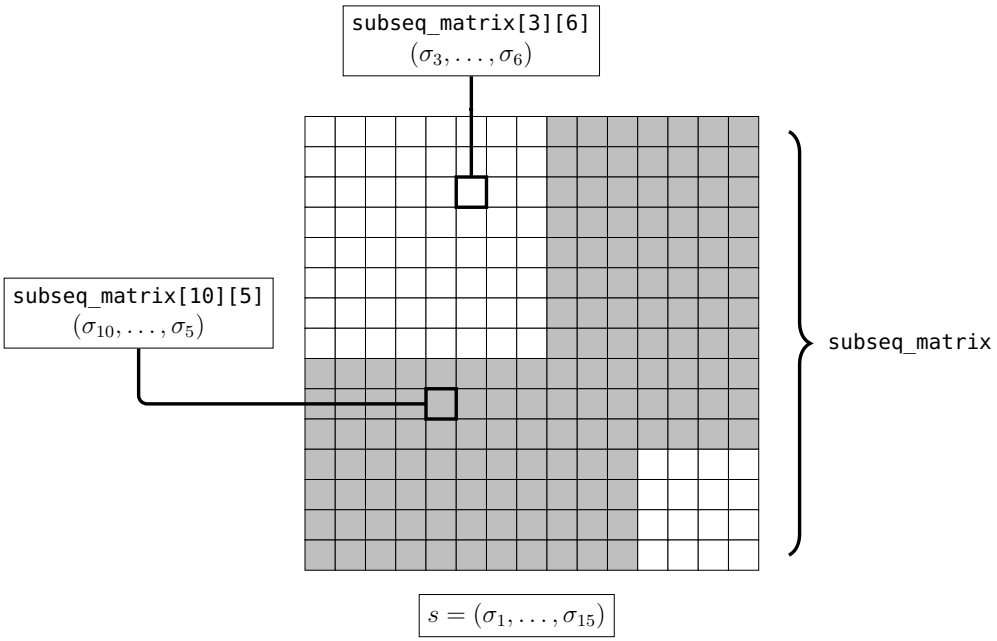


Figura 3.3: Matriz de subsequências de uma solução.

um movimento qualquer entre os nós σ_9 e σ_{11}. A região cinza na figura representa os elementos na matriz que precisam ser atualizados. Os elementos restantes, por sua vez, não precisam ser atualizados, já que as subsequências associadas não foram alteradas.

No geral, é sempre possível pensar em maneiras diferentes de atualizar a matriz de subsequências levando-se em conta as particularidades de cada movimento. Porém, isso exigiria a implementação de uma função de atualização de subsequências para cada movimento. Felizmente, a ideia mostrada na Figura 3.3 é aplicável a qualquer movimento, apresentando um bom equilíbrio entre facilidade de implementação e impacto no tempo de execução do algoritmo.

3.1.5 Aplicando o ILS ao MLP

Uma algoritmo simples para resolver o MLP é descrito em [11]. Para resolver o problema, os autores utilizaram a meta-heurística ILS. Com exceção do procedimento construtivo, mostrado no Algoritmo 1, todos os procedimentos, incluindo perturbação, busca local e estruturas de vizinhança, foram implementados exatamente como descrito no capítulo anterior. Para fazer as buscas locais de maneira eficiente, as estruturas auxiliares descritas nas seções anteriores foram utilizadas. O algoritmo é um dos métodos heurísticos mais efetivos para resolver o MLP. Como exercício, o leitor é encorajado a replicá-lo. Para isso, convém utilizar como base

o código desenvolvido no capítulo anterior, adicionando a estrutura `Subsequence` e a matriz de subsequências. Os resultados podem então ser comparados com os do artigo citado[2].

Algoritmo 1: Construction

```
1  Procedure Construction()
2  α ← random number in R
3  s ← {1}
4  Initialize Candidate List CL
5  CL ← CL − {1}
6  r ← 1
7  enquanto CL ≠ ∅ faça
8      Sort CL in ascending order according to their distance with respect to
         r
9      RCL ← α% best candidates of CL
10     Choose c ∈ RCL at random
11     s ← s ∪ {c}
12     r ← c
13     CL ← CL − {r}
14 retorna s
15 end Construction
```

A Tabela 3.1 contém os valores médios de custo acumulado e tempo (em segundos) de resolução das instâncias após 10 execuções do algoritmo em um Intel® Core™ i7-12700 2.10 GHz. Durante os experimentos, utilizou-se os parâmetros $I_{Max} = 10$, $I_{ILS} = \min\{100, n\}$ e $R = \{0.00, 0.01, 0.02, \ldots, 0.25\}$.

[2]Os autores do artigo utilizaram um subconjunto de instâncias do TSP. Portanto, o mesmo leitor leitor de instâncias pode ser usado.

Tabela 3.1: Resultados do método de [11].

Instância	**Resultados**	
	Tempo	**Custo**
dantzig42	0.03	12528.00
swiss42	0.02	22327.00
att48	0.06	209320.00
gr48	0.05	102378.00
hk48	0.06	247926.00
eil51	0.09	10178.00
berlin52	0.08	143721.00
brazil58	0.11	512361.00
st70	0.28	20557.00
eil76	0.40	17976.00
pr76	0.32	3455242.00
gr96	0.89	2097170.00
rat99	1.64	57986.00
kroA100	1.21	983128.00
kroB100	1.24	986008.00
kroC100	1.15	961324.00
kroD100	1.14	976965.00
kroE100	1.24	971266.00
rd100	1.19	340047.00
eil101	1.81	27518.70
lin105	1.22	603910.00
pr107	1.40	2026626.00

Parte II

Algoritmos Exatos

4

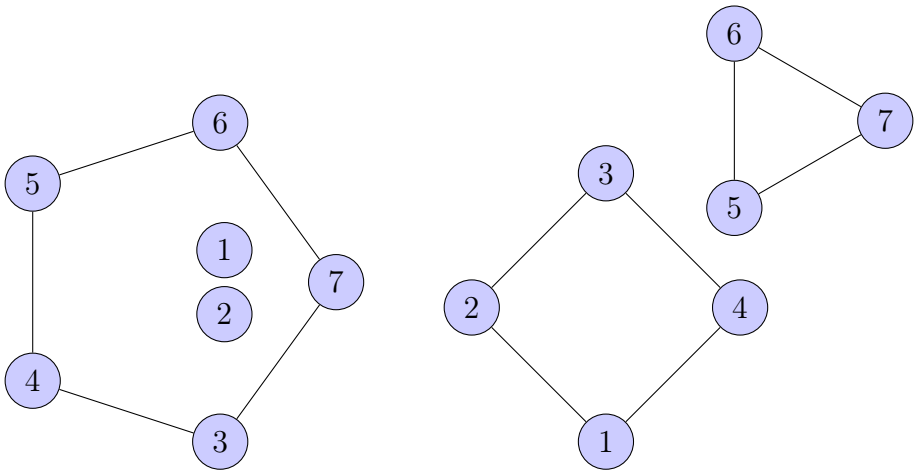
Branch-and-Bound combinatório

O *Branch-and-Bound* (BB) consiste em um algoritmo de enumeração implícita capaz de resolver problemas combinatórios como o TSP de maneira exata aproveitando-se da noção de *bounds*.

4.1 Relaxações

Todo problema de otimização combinatória possui um "espaço de soluções" associado. Aqui, entende-se por espaço de soluções o conjunto de todas as soluções válidas para uma certa instância do problema. O espaço de soluções para uma instância do TSP envolvendo o conjunto de vértices $\{1, 2, \ldots, n\}$, por exemplo, é o conjunto de todas as permutações da sequência $(1, 2, \ldots, n)$. Note que o espaço de soluções de um problema de otimização é diretamente moldado por suas restrições. Quanto mais restrições um problema de otimização combinatória possui, menor tende a ser o número de soluções que são válidas para ele. Em outras palavras, quanto mais restrito um problema de otimização combinatória é, menor é seu espaço de soluções associado. O TSP, por exemplo, possui duas restrições notórias: (i) cada uma das n cidades deve ser visitada exatamente uma vez; (ii) a sequência de visitas aos vértices deve formar um único *tour* (i.e., deve-se partir de uma cidade e retornar a ela). As figuras 4.1a e 4.1b apresentam soluções que não obedecem às restrições citadas, e que não pertencem ao espaço de soluções da versão original do TSP.

Ao retirar uma ou mais restrições de um problema de otimização, obtém-se um novo problema menos restrito. Diz-se que esse novo problema é uma "relaxação" do problema original. "Relaxar" uma restrição significa, portanto, removê-la do problema. Dito isso, há uma importante relação entre o espaço de soluções de um problema e o de suas relaxações. A Figura 4.2, por exemplo, ilustra o espaço de soluções de diferentes versões do TSP. Observe que o espaço de soluções da versão original do TSP está contido inteiramente no espaço de ambas as relaxações. Essa



(a) Solução que não obedece à restrição (i).

(b) Solução que não obedece à restrição (ii).

Figura 4.1: Exemplos de soluções inválidas.

propriedade é não apenas intuitiva — soluções para um problema mais restrito não deixam de ser válidas para um problema menos restrito —, como também é válida para qualquer problema de otimização, e será usada extensivamente nas próximas seções.

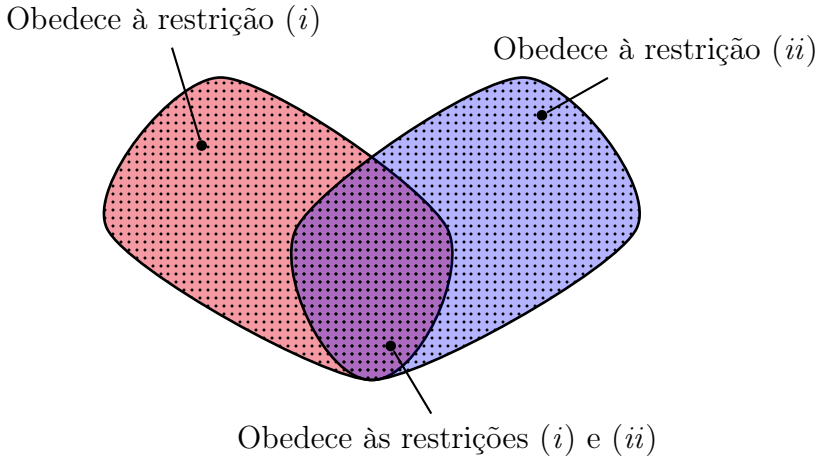


Figura 4.2: Espaços de soluções de diferentes versões do TSP.

4.2 Limitantes

Considere um problema de minimização qualquer e cuja solução ótima é $s^* \in \mathcal{S}$, em que $\mathcal{S}$ é seu espaço de soluções associado. Analogamente, considere que uma relaxação do mesmo problema está associada a um espaço de soluções $\mathcal{U}$ e uma solução ótima $u^* \in \mathcal{U}$. Já que o segundo problema consiste em uma relaxação do primeiro, então $\mathcal{S} \subseteq \mathcal{U}$, e portanto, a relação $f(u^*) \leq f(s^*)$ é sempre válida. Considere, agora, que $s' \in \mathcal{S}$ é uma solução qualquer — não necessariamente ótima — para o primeiro problema. Já que s^* é a solução ótima, então, por definição, $f(s^*) \leq f(s')$. Portanto, é garantido que o valor objetivo da solução ótima do primeiro problema se encontra dentro do intervalo $[f(u^*), f(s')]$. Por isso, diz-se que $f(u^*)$ é um limitante inferior (*lower bound*, LB) para o primeiro problema, enquanto $f(s')$ é um limitante superior (*upper bound*, UB).

O conceito de limitantes é útil, por exemplo, para determinar a qualidade de uma solução. Para ver isso, considere que s' é uma solução obtida de maneira heurística e que

$$100\% \times \frac{f(s') - f(u^*)}{f(u^*)}$$

é a diferença relativa entre $f(s')$ e $f(u^*)$, denominada *gap*. Note que se o *gap* for nulo, então $f(s') = f(u^*)$, o que significa que a solução s' é ótima, já que

$$f(u^*) \leq f(s^*) \leq f(s').$$

Por isso, quanto mais próximo de zero for o *gap*, maiores são as chances de que s' seja uma solução ótima para o problema.

O algoritmo *Branch-and-Bound* (BnB), apresentado neste capítulo, se baseia na ideia de diminuir o tamanho do intervalo $[f(u^*), f(s')]$ até que se possa garantir que a solução s' é ótima. O *gap* entre $f(s')$ e $f(u^*)$ funciona, portanto, como um critério de parada para o algoritmo, que finaliza apenas quando ele é nulo.

4.3 Algoritmo *Branch-and-Bound*

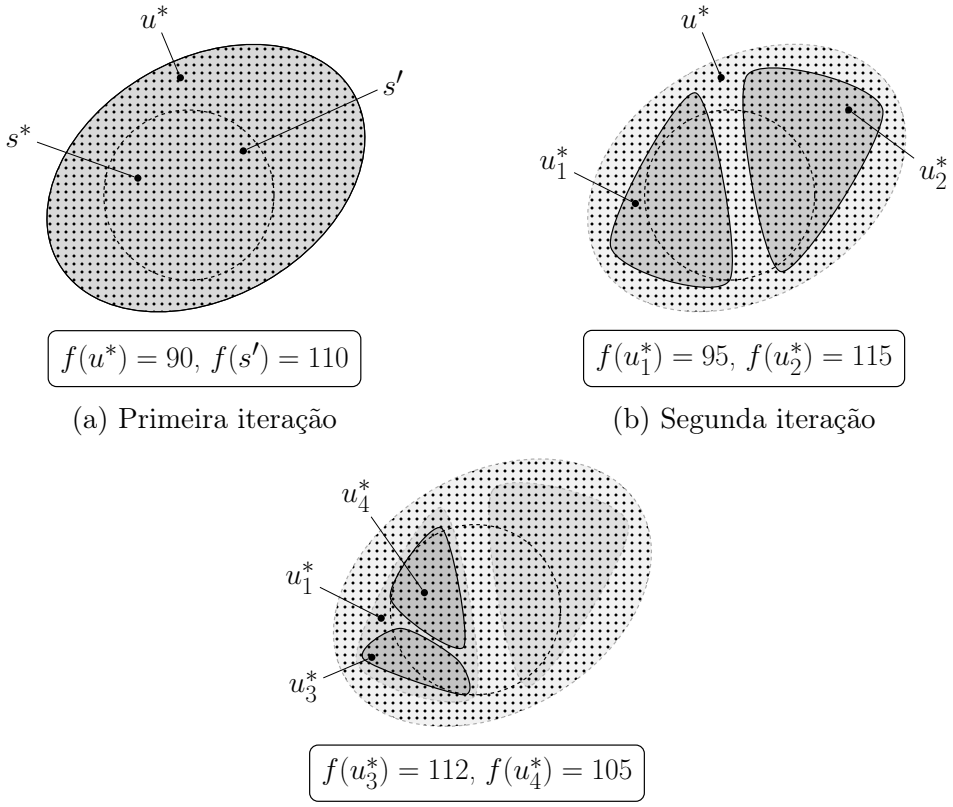
O Algoritmo 2 descreve o método *Branch-and-Bound* (BnB) para problemas de minimização. Primeiramente, o maior UB encontrado é inicializado com o valor ∞ (linha 2). Após isso, se uma heurística para o problema estiver disponível, ela é usada para gerar um UB válido (linhas 3–5). Em seguida, uma lista de subproblemas L é inicializada com o espaço $\mathcal{U}$, que corresponde ao espaço de soluções associado a uma relaxação qualquer do problema a ser resolvido. Após isso, um subproblema na lista L é escolhido e resolvido (linhas 8–10). Se a solução s' obtida for viável para o problema original, ela passa a ser a melhor solução,

e seu valor objetivo passa a ser o menor UB encontrado (linhas 11–14). Se, por outro lado, s' não for viável e seu valor objetivo não for maior que o melhor UB, o espaço associado a U é dividido em espaços mais restritos — e consequentemente menores —, que são então adicionados a L (linhas 15–18). O procedimento se repete enquanto a lista L não estiver vazia (linha 7).

Algoritmo 2: BRANCH-AND-BOUND

```
   Dados: ..
   Resultado: s*
1  início
2      UB ← ∞
3      se heurística disponível então
4          Obter solução s' usando heurística
5          UB ← f(s')
6      L ← {U}
7      enquanto L ≠ ∅ faça
8          S' ← U ∈ L
9          L ← L \ S'
10         s' ← Resolver(S')
11         se s' é viável então
12             se f(s') < UB então
13                 UB ← f(s')
14                 s* ← s'
15         senão
16             se f(s') ≤ UB então
17                 Dividir S' em subproblemas mais restritos S_1, S_2, ..., S_k
18                 L ← L ∪ {S_1, S_2, ..., S_k}
19     retorna s*
```

Para compreender a ideia geral por trás do Algoritmo 2, observe a Figura 2, que mostra o processo de subdivisão da região de soluções da relaxação de um problema de otimização qualquer. O espaço de soluções associados ao problema original e uma relaxação são representados por um círculo tracejado e uma região oval, respectivamente, na Figura 4.3a. Inicialmente, um LB foi obtido resolvendo-se a relaxação do problema na otimalidade, e apresenta o valor $f(u^*)$. Além disso, uma solução s' para o problema foi obtida de forma heurística, e possui um valor objetivo $f(s') = 110$. Nesse momento, a solução ótima do problema original (s^*) é desconhecida.



(a) Primeira iteração

(b) Segunda iteração

(c) Terceira iteração

Figura 4.3: Exemplo ilustrativo de algumas das iterações do BnB.

Suponha, agora, que fôssemos capazes de dividir a região associada à relaxação do problema em duas novas regiões mais restritas que excluem a solução u^*. O processo, chamado de *branching*, é mostrado na Figura 4.3b, e deve ser feito de forma que a solução s^* ainda esteja dentro de ao menos uma das regiões resultantes. Novamente, os problemas associados às regiões resultantes são resolvidos na otimalidade, gerando as soluções inviáveis u_1^* e u_2^*. Note, no entanto, que a solução u_2^* possui um valor objetivo maior que o da melhor solução viável encontrada (s'). Portanto, o espaço da direita pode ser ignorado, e precisa-se explorar apenas a região associada à solução u_1^*.

Na terceira iteração, mostrada na Figura 4.3c, a região esquerda é dividida em duas novas regiões que excluem a solução u_1^*. Novamente, a solução u_3^* possui um valor objetivo maior que o melhor UB conhecido, e sua região pode ser ignorada. A solução u_4^*, por sua vez, é viável para o problema original, e possui um valor objetivo menor que o de s', passando a ser o novo melhor UB encontrado. Observe, ainda, que não faz sentido dividir a região associada à solução u_4^* em regiões

menores. Fazer isso apenas geraria soluções cujo valor objetivo é maior ou igual ao atual melhor UB $f(u_4^*)$. Assim, já que não há mais nenhuma região a ser explorada, a solução u_4^* é a solução ótima para o problema.

Em suma, o BnB consiste em uma maneira inteligente de enumerar os elementos do espaço de soluções de um problema de otimização combinatória. O método é genérico, servindo como base para todos os algoritmos exatos apresentados no restante deste livro. Também é importante destacar o impacto positivo que o UB inicial $f(s')$ teve no exemplo mostrado. Se uma heurística não estivesse inicialmente disponível, e consequentemente o UB $f(s')$ não fosse conhecido, não teria sido possível ignorar a região associada à solução u_2^*. Consequentemente, determinar a solução ótima para o problema poderia exigir muito mais iterações. Por esse motivo, a elaboração de boas heurísticas é crucial para o bom desempenho dos algoritmos exatos.

4.4 Branch-and-bound para o TSP

Para aplicar o BnB a um problema de otimização combinatória, é necessário definir uma relaxação para o problema e um método de *branching*. Neste capítulo, será utilizado o *Assignment Problem* (AP), que é um problema de otimização combinatória que, se parametrizado de maneira apropriada, pode ser interpretado como uma relaxação para o TSP.

Tanto o AP quanto o método de *branching* necessários para aplicar o BnB ao TSP serão definidos nas próximas seções.

4.4.1 Transformação do TSP em um AP

Sejam U um conjunto de trabalhadores e W um conjunto de tarefas em que $|U| = |W|$. Designar um trabalhador $i \in U$ a uma tarefa $j \in W$ incorre um custo w_{ij}. O AP visa designar cada trabalhador a exatamente uma tarefa e cada tarefa a exatamente um trabalhador de forma que a soma total dos custos seja mínima. A Figura 4.4 mostra uma solução para uma instância do AP em que $U = \{1, 2, \ldots, 5\}$ e $W = \{6, 7, \ldots, 10\}$. O custo da solução é

$$w_{17} + w_{36} + w_{29} + w_{58} + w_{4,10}.$$

Para entender como o AP pode ser visto como uma relaxação do TSP, é necessário desenvolver um método que seja capaz de transformar uma instância arbitrária do TSP em um instância do AP. Considere, então, que o grafo completo $G = (V, E)$, em que $V = \{1, 2, \ldots, n\}$, é uma instância qualquer do TSP. A ins-

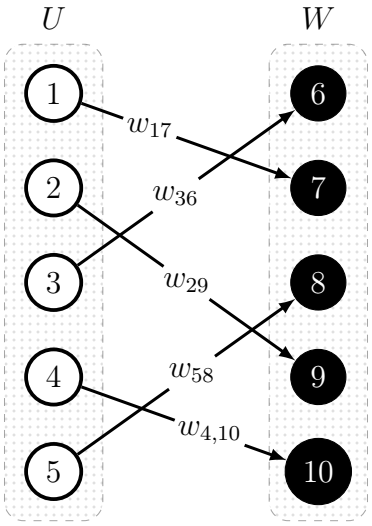


Figura 4.4: Exemplo de solução para uma instância do AP

tância do AP deve ser construída fazendo-se $U = V = W$ e

$$w_{ij} = \begin{cases} c_{ij} & \text{se } i \neq j \\ \infty & \text{se } i = j \end{cases}$$

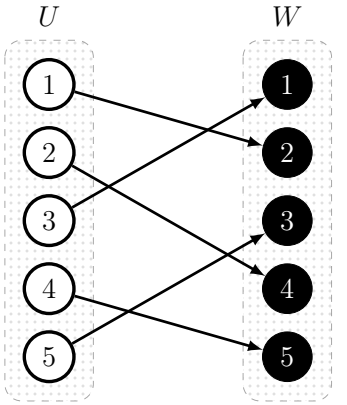
para todo $i \in U$, $j \in W$[1]. O resultado da transformação descrita é uma instância específica do AP que possui uma correspondência com a instância do TSP que a originou. Esse problema de otimização combinatória — que consiste em uma versão especial do AP — será chamado, daqui em diante, de AP_{TSP}.

A ligação entre o TSP e o AP_{TSP} pode ser compreendida observando-se as figuras 4.5a e 4.5b, que mostram duas soluções para uma instância do AP_{TSP}. As figuras 4.5c e 4.5d mostram uma representação alternativa das soluções das figuras 4.5a e 4.5b, respectivamente. É interessante notar que embora ambas as soluções sejam válidas para o AP_{TSP}, apenas a primeira delas é viável para o TSP. É fácil perceber, na verdade, que qualquer solução viável para o TSP também é viável para o AP_{TSP}. Por isso, o AP_{TSP} é uma relaxação do TSP.

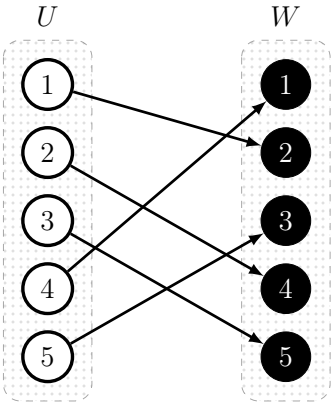
A vantagem de usar o AP_{TSP} como relaxação para o TSP é que existem algoritmos que resolvem o AP em tempo polinomial. Um deles é o Algoritmo Húngaro, que é capaz de resolver o AP em tempo $\mathcal{O}(n^3)$, em que $n = |U| = |W|$. Uma implementação do Algoritmo Húngaro e do método de transformação do TSP no AP_{TSP}, acompanhada de um leitor de instâncias do TSP, pode ser obtida

[1]Devido à natureza do AP, a transformação obriga cada nó a se conectar a algum outro nó. Fazer $w_{ij} = \infty$ para todo $i \in U$, $j \in W$, $i = j$ impede que um nó seja conectado a ele mesmo, já que o problema é de minimização.

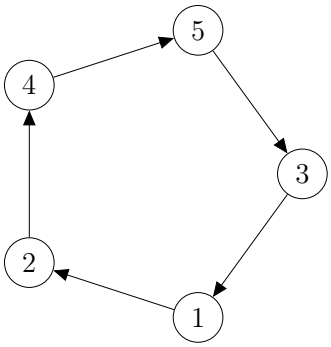
no seguinte *link*: https://github.com/carlosvinicius01/kit-opt/tree/master/BB-Combinat%C3%B3rio/algoritmo-hungaro.



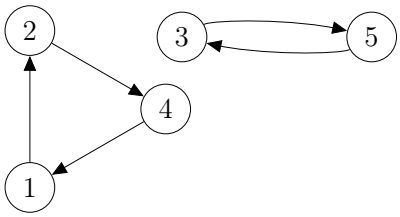
(a) Solução para o AP_{TSP}



(b) Solução para o AP_{TSP}



(c) Solução viável para o TSP



(d) Solução inviável para o TSP

Figura 4.5: Soluções do AP_{TSP}

4.4.2 Regra de *branching*

Suponha que uma instância do TSP foi convertida em uma instância do AP_{TSP}, que foi então resolvida por meio do Algoritmo Húngaro. Sabe-se que se a solução obtida for viável para o TSP, ela é a solução ótima. Caso contrário, é necessário dividir a região de soluções em regiões diferentes, de forma que a solução obtida não esteja contida em nenhuma das regiões resultantes. Uma forma simples de garantir isso é escolher o menor *subtour* da solução e proibir cada um de seus arcos separadamente, conforme mostrado na Figura 4.6. Na figura, inicia-se com uma solução para o AP_{TSP}, que é inviável para o TSP por possuir dois *subtours*. Nesse caso, o menor *subtour* é aquele que contém os nós 3 e 5. Então, duas novas

instâncias do AP_{TSP} são geradas: a primeira delas proíbe o arco $(5, 3)$, enquanto a segunda proíbe o arco $(3, 5)$. Um arco a pode ser proibido fazendo-se $c_a = \infty$.

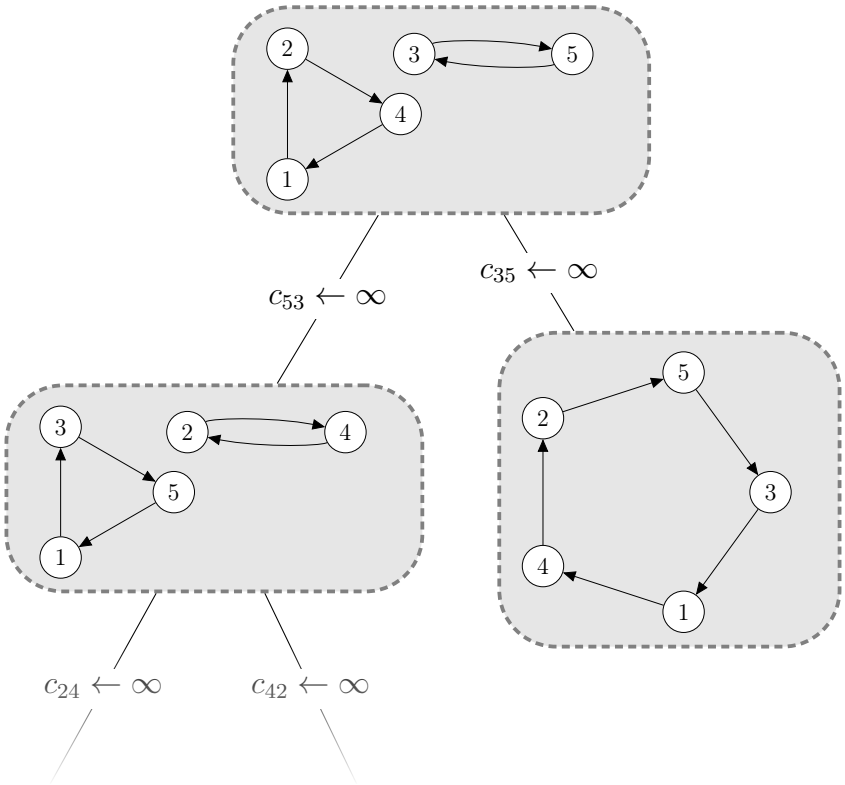


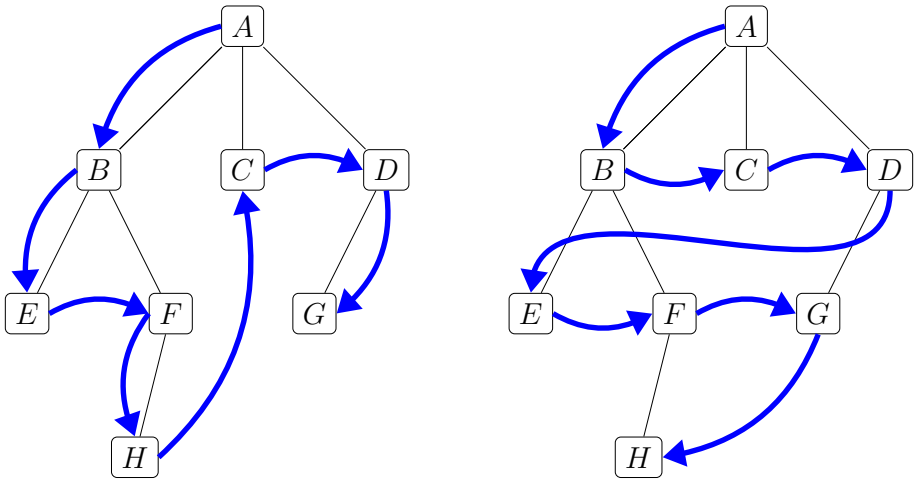
Figura 4.6: Iterações do BnB representadas por árvore

Na instância em que $c_{35} = \infty$, uma solução viável para o TSP foi obtida, e não é mais necessário subdividir o espaço de soluções nesse caso. A instância na qual $c_{53} = \infty$, por outro lado, gerou uma solução inviável. Por isso, o mesmo mecanismo é utilizado, dessa vez proibindo-se os arcos $(2, 4)$ e $(4, 2)$.

Assim, o BnB pode ser visto como um algoritmo que atua sobre uma árvore de decisões. Cada nó da árvore representa uma instância para um problema, que possui um espaço de soluções associado. Se, ao resolver o subproblema associado ao nó corrente na árvore, uma solução inviável for encontrada, deve-se gerar nós "filhos" restringindo-se ainda mais a instância associada. Além disso, os nós filhos sempre herdam as características dos "pais", que, neste caso, são os arcos proibidos.

4.4.3 Estratégias de *branching*

Em tese, o próximo nó a ser visitado pode ser escolhido de forma arbitrária, desde que a árvore seja explorada por completo. No entanto, a ordem na qual os nós



(a) DFS (b) BFS

Figura 4.7: DFS e BFS

são explorados possui um impacto significativo na prática. O método utilizado para escolher o próximo nó a ser explorado dentre os nós restantes na árvore é denominado "estratégia de *branching*".

As duas estratégias de *branching* mais simples são a busca por largura (*Breadth First Search, BFS*) e a busca por profundidade (*Depth First Search, DFS*), ilustradas na Figura 4.7, na qual uma árvore arbitrária é explorada de duas maneiras diferentes. A ordem na qual os nós são escolhidos é representada pelas setas azuis, e tem início no nó raiz A. Na DFS, a ordem na qual os nós são visitados se assemelha a um "mergulho". A partir do nó raiz A, mergulha-se em direção ao filho esquerdo (ou direito) até que isso não seja mais possível, como no caso do nó E, que não possui filhos. Em seguida, retorna-se ao ancestral mais próximo em busca de um nó filho não visitado. O processo se repete até que toda a árvore seja visitada. Na BFS, os nós são ordenados com base na sua distância em relação ao nó raiz (i.e., sua altura ou nível).

Note que a estratégia de *branching* deve ser executada no decorrer do algoritmo, ao mesmo tempo em que a árvore é gerada. Observe, ainda, que uma vez que um nó é visitado e seus possíveis filhos são adicionados à árvore, ele pode ser deletado. No caso da DFS, por exemplo, inicia-se a árvore apenas com o nó raiz, e os nós filhos são exaustivamente explorados e deletados um galho (*branch*) por vez. Na BFS, por outro lado, sempre que os nós de uma determinada altura são visitados, todos os seus filhos são adicionados à árvore, causando, muitas vezes, um crescimento exponencial no número de nós não visitados. Por isso, a DFS tende a ser mais eficaz que a BFS, principalmente em termos de uso de memória.

Por fim, há, ainda, a estratégia de busca pelo menor LB. Nela, o próximo nó

da árvore a ser escolhido é o que possui o menor LB associado. Para que cada nó tenha um LB, pode-se fazer com que os nós filhos herdem o LB do pai, obtido logo após a resolução do AP_{TSP}.

4.5 Implementação

A implementação do Algoritmo Húngaro fornecida na seção anterior retorna a solução na forma de uma matriz. A solução da Figura 4.5b, por exemplo, é representada pela matriz

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix},$$

cujo elemento A_{ij} assume o valor 1 se e somente se o trabalhador $i \in U$ foi designado à tarefa $j \in W$. Dito isso, o leitor deve implementar um algoritmo que é capaz de detectar se a matriz representa uma solução que contém *subtours*. Caso a solução possua *subtours*, o algoritmo deve, ainda, ser capaz de listar todos os arcos do menor deles. O trecho de código a seguir apresenta uma sugestão de estrutura de dados para representar a solução mostrada anteriormente em função de seus *subtours*:

```
1 std::vector<std::vector<int>> subtours = {{1,2,4,1}, {3,5,3}};
```

Em posse de um algoritmo que é capaz de resolver o AP_{TSP} e listar os *subtours*, resta apenas a tarefa de implementar o BnB, que pode ser facilitada utilizando-se a representação por meio da árvore mostrada na seção anterior. Cada nó da árvore, que está associado a um AP_{TSP}, pode ser representado conforme o seguinte trecho de código:

```
1 struct Node
2 {
3     std::vector<pair<int, int>> forbidden_arcs; // lista de arcos
          proibidos do no
4     std::vector<std::vector<int>> subtour; // conjunto de subtours da
          solucao
5     double lower_bound; // custo total da solucao do algoritmo hungaro
6     int chosen; // indice do menor subtour
7     bool feasible; // indica se a solucao do AP_TSP e viavel
8 };
```

Observe que não é preciso criar uma matriz de custos para cada nó. É possível modificar a matriz de custos original com base na lista de arcos proibidos em cada nó. Para proibir o arco $(2, 5)$, por exemplo, basta usar o seguinte trecho de código:

```
cost[2][5] = 99999999; /* usar std::limits<double>::infinity() pode causar
     erros de imprecisao numerica na implementacao do algoritmo hungaro
    usada */
```

O trecho de código a seguir contém sugestões de como implementar algumas rotinas do BnB utilizando-se uma lista encadeada para representar a árvore.

```
Node root; // no raiz
solve_hungarian(root); // resolver AP_TSP a partir da instancia original

/* criacao da arvore */
tree = std::list<Node>;
tree.push_back(root);

double upper_bound = std::numeric_limits::infinity<double>();

while (!tree.empty())
{
  auto node = branchingStrategy(); // escolher um dos nos da arvore
  vector<vector<int>> subtour = getSolutionHungarian(*node);

  if (node->lower_bound > upper_bound)
  {
    tree.erase(node);
    continue;
  }

  if (node->feasible)
    upper_bound = min(upper_bound, node->lower_bound);
  else
  {
    /* Adicionando os filhos */
    for (int i = 0; i < node.subtour[root.chosen].size() - 1; i++) //
        iterar por todos os arcos do subtour escolhido
    {
        Node n;
        n.arcos_proibidos = raiz.arcos_proibidos;

        std::pair<int,int> forbidden_arc = {
          node.subtour[root.chosen][i],
          node.subtour[root.chosen][i + 1]
        };

        n.forbidden_arcs.push_back(forbidden_arc);
        tree.push_back(n); //inserir novos nos na arvore
    }
  }

  tree.erase(node);
```

```
42 }
```

A vantagem de representar a árvore do BnB por meio de uma lista encadeada é que tanto a BFS quanto a DFS podem ser implementadas de forma simples. A BFS pode ser implementada escolhendo-se sempre o primeiro elemento (nó) da lista, enquanto na DFS deve-se escolher sempre o último elemento da lista. A corretude desse método é garantida desde que (*i*) todo nó seja deletado da lista assim que for visitado e (*ii*) os novos filhos sejam adicionados sempre ao fim (cauda) da lista. O leitor é encorajado a conferir essas propriedades por conta própria através de exemplos pequenos.

A estratégia de busca pelo menor LB também pode ser implementada usando uma lista encadeada para representar a árvore. Nesse caso, no entanto, o uso da estrutura de dados *binary heap*, disponível em C++ através do *container* `priority_queue`, é mais apropriado.

4.6 *Benchmark*

Todas as estratégias de *branching* mencionadas devem ser implementadas. Pode-se utilizar um parâmetro passado como argumento para o executável para escolher a estratégia a ser utilizada antes da execução do algoritmo, por exemplo. O algoritmo deve ser testado em todas as instâncias com menos que 30 vértices.

5
Relaxação Lagrangiana

A relaxação lagrangiana procura facilitar a obtenção de LBs para um problema de otimização combinatória através da remoção de restrições dificultantes. As restrições removidas são então tratadas por meio de penalizações na função objetivo. O objetivo é obter uma versão do problema que seja mais fácil de resolver, e que sirvam como uma boa relaxação, provendo LBs que sejam o mais próximos possíveis do valor ótimo do problema original. Este capítulo apresenta uma abordagem de resolução do TSP através do método da relaxação lagrangeana. Unido ao BnB através de uma regra de *branching* apropriada, o método supera o BnB combinatório do capítulo anterior em termos de performance, sendo capaz de resolver instâncias maiores.

5.1 Relaxação de um Problema Linear Inteiro

Considere o Problema Linear Inteiro (*Integer Linear Program*, ILP)

$$\min_{x \in X} \{\, cx \mid Ax \geq b, Bx \geq d \,\}, \tag{P}$$

em que $X \coloneqq \{\, y \in \mathbb{R}^m \mathbb{Z}^n \mid y \geq 0 \,\}$, i.e., a tupla x representa um conjunto composto por m variáveis reais e n variáveis inteiras. Suponha, agora, que a presença das restrições $Ax \geq b$ impõem um desafio significativo ao problema. É possível retirar essas restrições e penalizar a função objetivo caso sejam violadas, conforme o problema

$$\min_{x \in X} \{\, cx + \lambda(b - Ax) \mid Bx \geq d \,\}, \tag{PR$_\lambda$}$$

em que cada componente do vetor $\lambda \geq 0$ está associada a uma das restrições retiradas do problema original (P).

Embora não seja capaz de garantir que violações não ocorram, o termo adicional na função objetivo do problema (PR$_\lambda$) incentiva que as restrições $Ax \geq b$ sejam

respeitadas. Para perceber isso, seja a_i a i-ésima linha da matriz A. Note que o problema em questão é de minimização, e se $a_i x < b_i$ para algum i, um número não negativo (i.e., uma penalização) $\lambda_i(b_i - a_i x)$ será adicionado à função objetivo, aumentando-a.

Teorema 1. *Para qualquer* $\lambda \geq 0$, (PR_λ) *é uma relaxação de* (P).

Demonstração. Já que (P) possui todas as restrições de (PR_λ), então, o espaço de soluções de (P) está contido no espaço de soluções de (PR_λ). Para mostrar que o valor ótimo de (PR_λ) é um LB para o valor ótimo de (P), observe que, para qualquer solução viável x de (P),

$$(b - Ax) \leq 0.$$

Já que foi especificado que $\lambda \geq 0$, então

$$\lambda(b - Ax) \leq 0,$$

e, por consequência,

$$cx + \lambda(b - Ax) \leq cx,$$

completando a demonstração. □

Teorema 2. *Se* x^* *é uma solução ótima de* (PR_λ) *para algum* $\lambda \geq 0$ *e* $\lambda(b - Ax^*) = 0$ *e* $Ax^* \geq b$, *então* x^* *também é uma solução ótima de* (P).

Demonstração. Primeiramente, x^* é viável para (P), já que $Ax^* \geq b$. Além disso, o fato de que x^* é uma solução ótima para (PR_λ) significa que

$$cx^* + \lambda(b - Ax^*) \leq cx + \lambda(b - Ax)$$

para qualquer x, mas $\lambda(b - Ax^*) = 0$, então

$$cx^* \leq cx + \lambda(b - Ax),$$

e de acordo com o Teorema 1,

$$cx + \lambda(b - Ax) \leq cx,$$

logo,

$$cx^* \leq cx + \lambda(b - Ax) \leq cx,$$

e x^* é de fato uma solução ótima para (P). □

5.1.1 Exemplo com ILP simples

Considere o ILP

$$\min \left\{5x_1 + 4x_2 + 2x_3 \mid 2x_1 + 3x_2 + 4x_3 \geq 5, (x_1, x_2, x_3) \in \{0,1\}^3\right\}. \tag{P1}$$

Relaxando-se a restrição de (P1), obtém-se o novo ILP

$$\min \left\{5x_1 + 4x_2 + 2x_3 + \lambda(5 - (2x_1 + 3x_2 + 4x_3)) \mid (x_1, x_2, x_3) \in \{0,1\}^3\right\}. \tag{PR$_\lambda$1}$$

Já que o problema original (P1) possui uma única restrição, que foi removida, sua relaxação possui um único penalizador λ na função objetivo. A solução ótima de (PR$_\lambda$1) depende do valor do penalizador λ, que deve ser escolhido antes da resolução do problema. Se $\lambda = 0$, por exemplo, o valor ótimo é obtido fazendo-se $x_1 = x_2 = x_3 = 0$, tornando nulo o valor da função objetivo.

Para analisar o comportamento do problema em função de λ, convém reagrupar os termos da função objetivo de (PR$_\lambda$1), obtendo-se a expressão equivalente

$$(5 - 2\lambda)x_1 + (4 - 3\lambda)x_2 + (2 - 4\lambda)x_3 + 5\lambda. \tag{5.1}$$

Observando-se a Expressão (5.1), e sabendo-se que (PR$_\lambda$1) não possui restrições além do domínio das variáveis x_1, x_2 e x_3, é fácil perceber que

$$x_1 = \begin{cases} 1 & \text{se } \lambda > 5/2 \\ 0 & \text{caso contrário.} \end{cases}$$

Em outras palavras, fazer $x_1 = 1$ só irá diminuir a função objetivo se $5 - 2\lambda$ (termo que multiplica x_1 na Expressão (5.1)) for menor que zero.

Seguindo-se uma lógica similar,

$$x_2 = \begin{cases} 1 & \text{se } \lambda > 4/3 \\ 0 & \text{caso contrário,} \end{cases}$$

e

$$x_3 = \begin{cases} 1 & \text{se } \lambda > 1/2 \\ 0 & \text{caso contrário.} \end{cases}$$

Dessa forma, com o intuito de determinar o valor ótimo de (PR$_\lambda$1), é possível saber, para cada valor de λ, quais variáveis assumem o valor 1, e quais assumem o valor 0. A Figura 5.1 ilustra o comportamento do valor ótimo de (PR$_\lambda$1) em função do penalizador λ. Observe que $\lambda = 4/3$ foi o penalizador que proveu a melhor estimativa do valor ótimo do problema original (P1).

A próxima seção formaliza a tarefa que consiste em encontrar os melhores penalizadores para problemas como (PR$_\lambda$1).

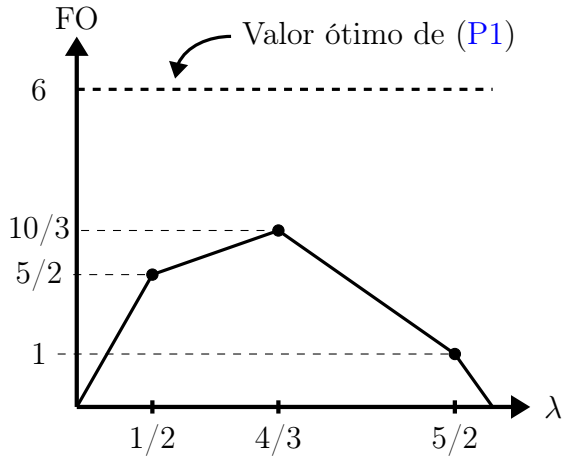


Figura 5.1: Valor da função objetivo de ($PR_\lambda 1$) em função de λ.

5.2 Dual lagrangiano

Um tema introduzido no capítulo anterior — e recorrente no restante deste livro — é a redução do *gap* entre o LB provido pela relaxação e o valor ótimo do problema original. No exemplo anterior, uma análise feita a partir de um ILP revelou que o valor ótimo de sua relaxação depende diretamente do valor escolhido para o penalizador. Essa noção pode ser facilmente generalizada, e conclui-se que, ironicamente, encontrar os melhores penalizadores para (PR_λ) pode ser formulado como o outro problema de otimização

$$\max_{\lambda \geq 0} \left\{ \min_{x \in X} \{ cx + \lambda(b - Ax) \mid Bx \geq d \} \right\}. \tag{D}$$

Por apresentar dois problemas de otimização interdependentes, o problema (D) pode parecer inicialmente confuso. Lembre-se, no entanto, de que como mostrado no exemplo anterior, o valor ótimo do problema interno depende do vetor de penalizadores λ. Em outras palavras, sendo $v(PR_\lambda)$ o valor objetivo da solução ótima de (PR_λ), o problema (D) pode ser formulado alternativamente como

$$\max_{\lambda \geq 0} \{ v(PR_\lambda) \}. \tag{D}$$

Diferente de (P) e (PR_λ), (D) possui como variáveis o vetor de penalizadores λ. Resolver (D) significa encontrar o melhor LB possível para o problema original (P) a partir de sua relaxação (PR_λ). No exemplo anterior, a solução ótima para o dual lagrangiano associado a ($PR_\lambda 1$) é $\lambda = 4/3$, e o valor objetivo referente à solução é $10/3$.

Resta, então, descobrir uma maneira de resolver o dual lagrangiano. Infelizmente, embora contínua, a função na Figura 5.1 claramente não é diferenciável, não permitindo o uso de técnicas do cálculo diferencial para determinar seus pontos máximos. No entanto, a função possui uma característica interessante: ela é côncava (i.e., a região abaixo da função forma um conjunto convexo) e, portanto, possui um ótimo global, ou seja, um único pico. Essas características, que são generalizadas para qualquer problema semelhante a (PR_λ) no seguinte teorema, serão exploradas mais adiante.

Teorema 3. *A função v(PR_λ) é sempre côncava.*

Demonstração. Qualquer ILP pode ser reformulado em função da envoltória convexa de sua região viável. Por isso, o problema (PR_λ) pode ser reescrito como

$$\min_{x \geq 0} \{cx + \lambda(b - Ax) \mid x \in \text{Co}\{x \in X \mid Bx \geq d\}\},$$

em que Co{.} denota a envoltória convexa do conjunto {.}. Sabe-se, ainda, que a solução ótima de um problema linear reside em um de seus pontos extremos. Assim, sendo $\{x^1, \ldots, x^K\}$ o conjunto de pontos extremos de $\text{Co}\{x \in X \mid Bx \geq d\}$, ($PR_\lambda$) pode ser novamente reformulado como

$$\min_{k=1,\ldots,K} \{cx^k + \lambda(b - Ax^k)\}.$$

Isso significa que a função v(PR_λ) equivale ao mínimo de uma ou mais funções lineares de λ, sendo, portanto, côncava, de acordo com um resultado conhecido da Análise Convexa. □

O Teorema 3 estabelece que resolver o dual lagrangiano se reduz a encontrar o máximo global de uma função contínua. As próximas seções descrevem os passos necessários para resolver esse problema de otimização usando uma teoria análoga às técnicas vistas em cursos de Cálculo.

5.3 Subgradiente

Dada uma função côncava $f : \mathbb{R}^m \mapsto \mathbb{R}$, o vetor $g \in \mathbb{R}^m$ é considerado um subgradiente no ponto $\lambda_0 \in \mathbb{R}^m$ se

$$f(\lambda) \leq g(\lambda - \lambda_0) + f(\lambda_0) \tag{5.2}$$

para qualquer $\lambda \in \mathbb{R}^m$.

De acordo com a Inequação (5.2), para que g seja um subgradiente válido da função $f(\lambda)$ no ponto λ_0, o gráfico da função linear

$$h(\lambda) = f(\lambda_0) + g(\lambda - \lambda_0)$$

precisa estar sempre acima do gráfico de $f(\lambda)$. Isso pode ser visto mais facilmente nas figuras 5.2a–5.2b, que apresentam exemplos de três vetores subgradientes para uma função de uma variável. Dois dos três vetores são subgradientes válidos para o ponto $\lambda = 1/2$, enquanto o outro é valido para o ponto $\lambda = 5$. Na Figura 5.2a, as retas tracejadas representam o gráfico da função $h(\lambda)$ para cada vetor subgradiente. Note que tais funções obedecem à Inequação (5.2), sendo sempre maiores ou iguais à função $f(\lambda)$. Os vetores ortogonais a tais retas não são exatamente os subgradientes em si, mas extensões deles: já que $f(\lambda)$ é uma função de uma única variável, os subgradientes devem pertencer a $\mathbb{R}$. Os verdadeiros subgradientes, que são vetores unidimensionais no caso do exemplo, se encontram na Figura 5.2b, espalhados verticalmente para melhor visualização.

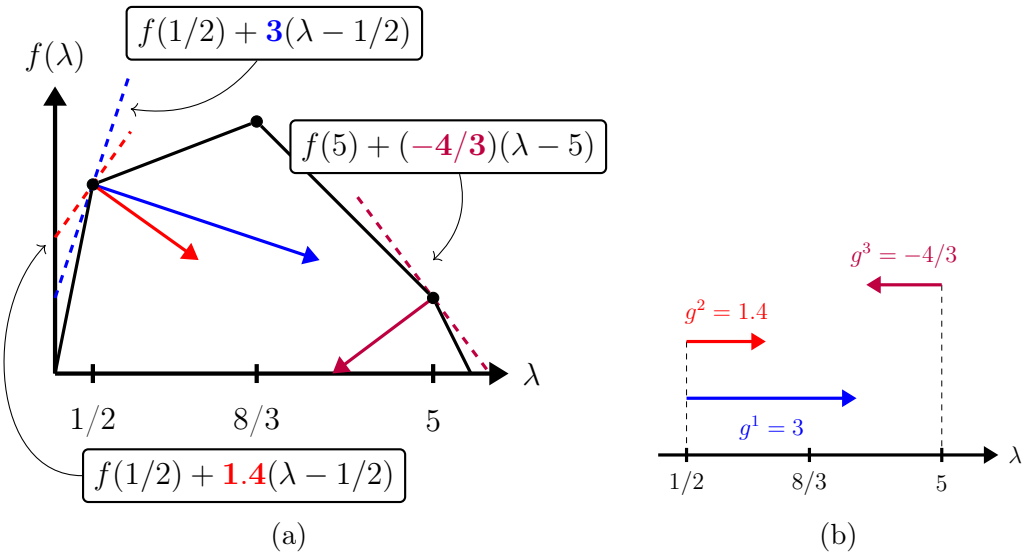


Figura 5.2: Exemplos de vetores subgradientes para uma função côncava de uma variável

A informação mais importante a ser extraída da Figura 5.2b é que, olhando-se a partir do ponto λ_0, o subgradiente associado "aponta" para o ótimo global de $f(\lambda)$ (8/3). Na verdade, se $f(\lambda)$ for uma função diferenciável de uma única variável, o subgradiente no ponto λ_0 é exatamente $df(\lambda_0)/d\lambda$. O vetor subgradiente nada mais é, portanto, que uma generalização do conceito de gradiente para funções côncavas não diferenciáveis. Dessa forma, uma maneira de resolver o dual lagrangiano é começar a partir de um ponto λ_0 e "dar um pequeno passo" na direção do vetor

subgradiente até o ponto $\lambda_0+\epsilon g$, em que ϵ é um número muito pequeno. Repetindo-se o processo até que o aumento em $f(\lambda)$ seja marginal, é possível chegar ao ótimo global.

Teorema 4. *Se x^* é a solução ótima de (PR_λ) para $\lambda = \lambda_0$, então $b - Ax^*$ é subgradiente de (D) no ponto λ_0.*

Demonstração. Sejam $f(\lambda)$ e $f(\lambda_0)$ os valores de $v(PR_\lambda)$ para os penalizadores λ e λ_0, respectivamente. Sabe-se que,

$$\begin{aligned} f(\lambda_0) &= \min_{x\in X}\{\, cx + \lambda_0(b - Ax) \mid Bx \geq d \,\} \\ &= cx^* + \lambda_0(b - Ax^*). \end{aligned}$$

Além disso embora a solução x^* seja ótima para (PR_λ) quando o penalizador é λ_0, isso não necessariamente é verdade para o penalizador arbitrário λ. Portanto,

$$\begin{aligned} f(\lambda) &\leq cx^* + \lambda(b - Ax^*) \\ &\leq cx^* + \lambda(b - Ax^*) + \lambda_0(b - Ax^*) - \lambda_0(b - Ax^*) \\ &\leq f(\lambda_0) + \lambda(b - Ax^*) - \lambda_0(b - Ax^*) \\ &\leq (b - Ax^*)(\lambda - \lambda_0) + f(\lambda_0), \end{aligned}$$

e o vetor $g = (b - Ax^*)$ é de fato um subgradiente válido no ponto λ_0, de acordo com a definição de subgradiente. □

O Teorema 4 nos dá uma maneira simples e direta de obter um vetor subgradiente para qualquer ponto λ. Em posse disso, o procedimento para resolver o dual lagrangiano, que baseia-se na ideia descrita anteriormente, é apresentado na próxima seção.

5.4 Método do subgradiente

Como discutido na seção anterior, a ideia por trás do método de resolução do dual lagrangiano é caminhar em direção ao ótimo global de (D) através do vetor subgradiente. O procedimento depende, dentre outros fatores, de um cálculo de um tamanho de passo e de um critério de parada adequado, para que possa convergir numa velocidade razoável. Uma das maneiras de implementar esse procedimento, proposta em REF, é descrita no Algoritmo 3.

No Algoritmo, 3, inicia-se com um UB obtido de maneira heurística na linha 2. Após isso, a solução ótima de (PR_λ) e seu LB asociado são obtidos nas linhas 7–8. Então, um pequeno passo na direção do subgradiente $b - Ax^*$ é dado na linha 10. O fator que determina o tamanho do passo é o número real μ, que é

Algoritmo 3: SUBGRADIENT METHOD

Dados: ..
Resultado: λ^*

```
1  início
2      UB ← Heuristic upper bound for original problem
3      λ ← 0
4      ϵ ← 1
5      k ← 0
6      enquanto stop criterion not met faça
7          x* ← optimal solution of (PR_λ)
8          w ← cx* + λ(b − Ax*)
9          μ ← ϵ(UB − w)/ ((b − Ax*)(b − Ax*))
10         λ ← λ + μ (b − Ax*)
11         se w > w* então
12             w* ← w
13             λ* ← λ
14             k ← 0
15         senão
16             k ← k + 1
17             se k ≥ k_max então
18                 k ← 0
19                 ϵ ← ϵ/2
20         stop criterion ← ϵ < ϵ_min or (Ax* ≥ b and λ(b − Ax*) = 0)
21     retorna λ*
```

atualizado na linha 9. Se o LB obtido na iteração atual for melhor que o melhor LB w^*, então w^* e λ^* são atualizados, e o número de iterações k é resetado (linhas 11–14). Caso contrário, o número de iterações é incrementado na linha 16. Além disso, se o valor do LB não melhorar (i.e., o valor objetivo do dual lagrangiano não aumentar) após $k_{\max}$ iterações, o número de iterações é resetado e o valor de ϵ é reduzido pela metade (linhas 17–19), resultando em tamanhos de passo cada vez menores.

O processo se repete até que o valor de ϵ seja menor que um valor muito pequeno $\epsilon_{\min}$. Isso significa que o algoritmo convergiu, e a solução λ não pode ser melhorada de forma significativa, sendo suficientemente próxima do ótimo global. Outra possibilidade (mais rara) é a de que condições semelhantes às estabelecidas no Teorema 2 sejam atingidas, caso no qual o Algoritmo também encerra.

5.5 Qualidade do Dual Lagrangiano

O dual lagrangiano neste capítulo foi introduzido a partir de um ILP. Sendo assim, é natural se perguntar o quão bom o LB obtido ao resolver o problema é se comparado a outras relaxações. A relaxação mais intuitiva de (P) é, definitivamente, sua relaxação linear

$$\min_{x \geq 0} \{\, cx \mid Ax \geq b, Bx \geq d \,\}, \tag{P_{lin}}$$

que difere de (P) apenas pelo fato de que todas as variáveis agora pertencem ao conjunto dos números reais.

Teorema 5. *O valor ótimo de* (D) *é maior ou igual ao valor ótimo de* (P_{lin}).

Demonstração. O problema (PR_λ) se torna menos restrito retirando-se suas restrições de integralidade. Portanto,

$$\max_{\lambda \geq 0} \left\{ \min_{x \in X} \{\, cx + \lambda(b - Ax) \mid Bx \geq d \,\} \right\} \geq \max_{\lambda \geq 0} \left\{ \min_{x \geq 0} \{\, cx + \lambda(b - Ax) \mid Bx \geq d \,\} \right\}.$$

Observe, ainda, que o lado direito da inequação acima pode ser reescrito como

$$\max_{\lambda \geq 0} \left\{ \lambda b + \min_{x \geq 0} \{\, (c - \lambda A)x) \mid Bx \geq d \,\} \right\}.$$

Substituindo-se o problema de minimização interno pelo seu dual, o problema anterior pode ser reescrito novamente como

$$\max_{\lambda \geq 0} \left\{ \lambda b + \max_{y \geq 0} \{\, yd \mid yB \leq (c - \lambda A) \,\} \right\} = \max_{\lambda \geq 0, y \geq 0} \{\lambda b + yd \mid yB + \lambda A \leq c \,\}.$$

Por fim, substituindo-se o problema resultante pelo seu dual, obtém-se

$$\min_{x \geq 0} \{cx \mid Ax \geq b, Bx \geq d \,\},$$

que é exatamente o problema (P_{lin}). □

O Teorema 5 é muito importante, pois estabelece que, no pior dos casos, o LB fornecido pelo dual lagrangiano de um ILP é tão bom quanto o LB fornecido pela sua relaxação linear, podendo, por vezes, ser ainda melhor. Uma consequência direta disso é que se (P) já for um problema linear (i.e., $X = \{\, x \in \mathbb{R}^{m+n} \mid x \geq 0 \,\}$), então o valor ótimo de (D) é igual ao de (P). O seguinte Teorema permite compreender mais profundamente a qualidade do LB associado ao dual lagrangiano.

Teorema 6. *O problema linear*

$$\min_{x \geq 0} \{\, cx \mid Ax \geq b, x \in Co\{\, Bx \geq d \mid x \in X \,\} \,\} \tag{CG}$$

possui o mesmo valor ótimo que (D).

Demonstração. O dual lagrangiano de (CG) é

$$\max_{\lambda \geq 0} \left\{ \min_{x \geq 0} \{\, cx + \lambda(b - Ax) \mid x \in \text{Co}\{\, Bx \geq d \mid x \in X \,\} \,\} \right\},$$

e pode ser reescrito como

$$\max_{\lambda \geq 0} \left\{ \min_{x \in X} \{\, cx + \lambda(b - Ax) \mid Bx \geq d \,\} \right\},$$

que é exatamente igual ao problema (D). Observe, ainda, que (CG) é um problema puramente linear. Portanto, segue do Teorema 5 que, como discutido anteriormente, a solução ótima de (D) possui o mesmo valor objetivo que a solução ótima de (CG). □

O resultado descrito no Teorema 6 estabelece uma equivalência entre (D) e (CG). Essencialmente, um problema é dual do outro. Observe que embora seja claramente uma relaxação de (P), (CG) não é meramente uma relaxação linear de (P). A diferença entre (P_{lin}) e (CG) é que o último captura os pontos inteiros do conjunto $\{\, Bx \geq d \mid x \in X \,\}$ por meio de uma envoltória convexa. Se, no entanto,

$$\{\, Bx \geq d \mid x \geq 0 \,\} = \text{Co}\{\, Bx \geq d \mid x \in X \,\},$$

diz-se que o problema original (P) possui a Propriedade da Integralidade. Nesse caso, tanto (CG) quanto (D) e (P_{lin}) fornecerão exatamente o mesmo LB. Para um melhor entendimento, consulte a Figura 5.3, que resume as informações discutidas por meio de uma interpretação geométrica. De forma similar, a Figura 5.4 ilustra o caso em que a região $\{Bx \geq d\}$ é igual à sua própria envoltória convexa (i.e., as restrições $Bx \geq d$ obedecem à propriedade da integralidade).

5.6 Relaxação e dual lagrangiano do TSP

Suponha que a variável binária x_{ij} assume o valor 1 se uma aresta de um grafo completo $G = (V, E)$ for utilizada. A partir dessa variável, o TSP pode ser formulado

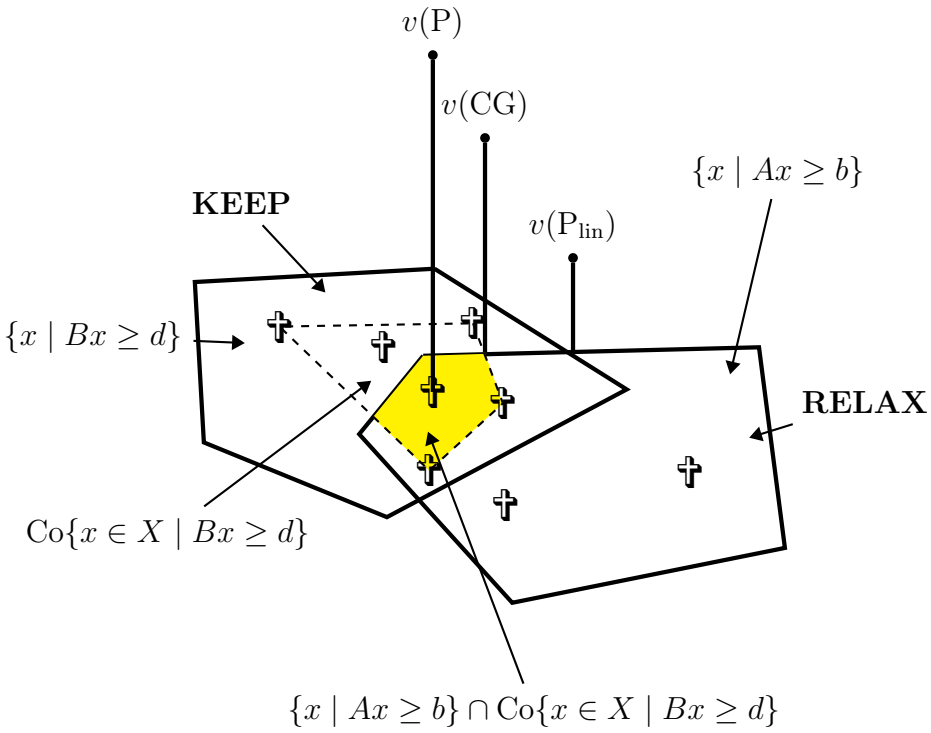


Figura 5.3: Representação geométrica das relaxações de um ILP

como o ILP

$$\min \sum_{i \in V, j > i} c_{ij} x_{ij} \tag{TSP}$$

$$\text{s.t.} \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2, i \in V, \tag{5.3}$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1, S \subset V, S \neq \emptyset, \tag{5.4}$$

$$x_{ij} \in \{0, 1\}, i \in V, j > i. \tag{5.5}$$

Na formulação (TSP), a função objetivo visa minimizar o custo total de viagem. As restrições (5.3) impõem que o grau de todos os vértices do grafo resultante seja igual a 2. As restrições (5.4), por sua vez, garantem que o grafo resultante será conexo, e portanto, formará um ciclo Hamiltoniano. Por fim, as restrições (5.5) descrevem a natureza das variáveis x_{ij}.

Neste caso, convém relaxar as restrições de grau para todos os vértices exceto o vértice $i = 0$. Isso significa que, na versão relaxada de (TSP) abordada neste capítulo, as restrições (5.3) serão removidas para todos os vértices $i \in V, i \neq$

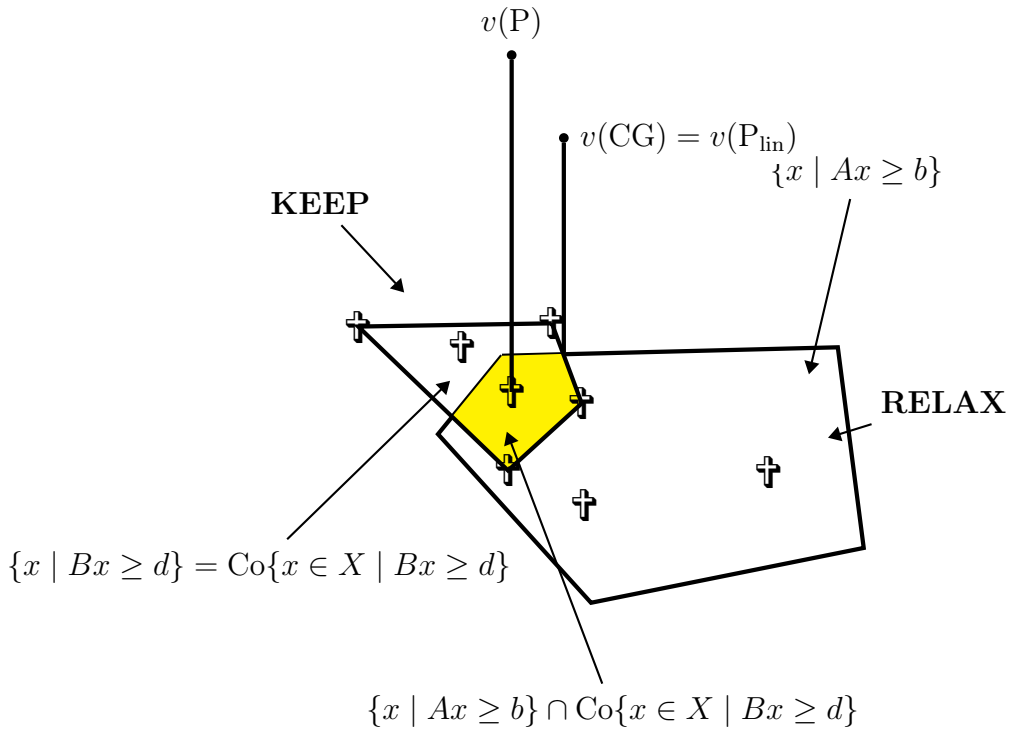


Figura 5.4: Representação geométrica do caso em que o problema obedece à propriedade da integralidade

0. Note, porém, que as restrições (TSP) são de igualdade. É possível, porém, reescrevê-las através de duas desigualdades como

$$\begin{aligned} \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} \geq 2, i \in V, \\ -\sum_{j>i} x_{ij} - \sum_{j<i} x_{ji} \geq -2, i \in V. \end{aligned} \tag{5.3}$$

A relaxação lagrangiana de (TSP) pode então ser escrita como

$$\begin{aligned}
\min \sum_{i\in V, j>i} c_{ij}x_{ij} &+ \sum_{i>0} \lambda_i^1 \left(2 - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}\right) \\
&+ \sum_{i>0} \lambda_i^2 \left(-2 + \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji}\right) && (\text{TSP}_\lambda) \\
\text{s.t.} \sum_{j>0} x_{0j} &= 2, && (5.6) \\
\sum_{i\in S} \sum_{j\notin S} x_{ij} &\geq 1, S \subset V, S \neq \emptyset, && (5.7) \\
x_{ij} &\in \{0,1\}, i \in V, j > i. && (5.8)
\end{aligned}$$

A formulação (TSP_λ) possui dois vetores de penalização λ_1 e λ_2, associados às versões com desigualdade das restrições (5.3). É possível simplificar a função objetivo significativamente reorganizando-se os termos e obtendo-se a expressão equivalente

$$\sum_{i\in V, j>i} c_{ij}x_{ij} + \sum_{i>0} \lambda_i^1 \left(2 - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}\right) - \sum_{i>0} \lambda_i^2 \left(2 - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}\right),$$

ou, ainda,

$$\sum_{i\in V, j>i} c_{ij}x_{ij} + \sum_{i>0} (\lambda_i^1 - \lambda_i^2) \left(2 - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}\right).$$

Note que se $\lambda_i^1 \geq 0$ e $\lambda_i^2 \geq 0$, então $\lambda_i^1 - \lambda_i^2$ pode assumir qualquer valor real. Portanto, é possível substituir o termo $(\lambda_i^1 - \lambda_i^2)$ por um único vetor λ cujas componentes são irrestritas (e não apenas maiores ou iguais a zero) e obter a expressão

$$\sum_{i\in V, j>i} c_{ij}x_{ij} + \sum_{i>0} \lambda_i \left(2 - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji}\right),$$

o que faz sentido, já que diferente das restrições do tipo $Ax \geq b$, que podem ser violadas somente se $Ax < b$, as restrições de igualdade como (5.3) podem ser violadas tanto para "mais" quanto para "menos", e é necessário penalizá-las conforme adequado. Reorganizando-se novamente, e supondo a existência de um penalizador "*dummy*" $\lambda_0 = 0$, obtém-se a expressão

$$\sum_{i\in V, j>i} c_{ij}x_{ij} - \sum_{i\in V} \lambda_i \left(\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji}\right) + 2\sum_{i>0} \lambda_i, \tag{5.9}$$

Analisando-se a Expressão (5.9), pode-se perceber que cada termo x_{ij} é multiplicado não só pelo respectivo fator c_{ij}, mas também pelos fatores $-\lambda_i$ e $-\lambda_j$. Finalmente, o problema (TSP_λ) pode ser formulado como

$$\min \sum_{i \in V, j > i} (c_{ij} - \lambda_i - \lambda_j) x_{ij} + 2 \sum_{i > 0} \lambda_i \tag{TSP_λ}$$

$$\text{s.t.} \sum_{j > 0} x_{0j} = 2, \tag{5.10}$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1, S \subset V, S \neq \emptyset, \tag{5.11}$$

$$x_{ij} \in \{0, 1\}, i \in V, j > i. \tag{5.12}$$

Em suma, (TSP_λ) se resume a encontrar um grafo conexo no conjunto de vértices $V' = \{1, 2, \ldots, n\}$ e conectá-lo ao nó $i = 0$ por meio de exatamente duas arestas. O custo de cada aresta $\{i, j\}$ depende de c_{ij} e dos penalizadores λ_i e λ_j, e a soma total dos custos do grafo obtido deve ser mínima.

Exemplos de soluções para uma instância do (TSP_λ) podem ser vistas nas Figuras 5.5a–5.5c. Como esperado, em todos os exemplos, o nó $i = 0$ está sempre conectado a exatamente outros dois nós. As figuras diferem, principalmente, no número de nós que viola as restrições (5.3). Na Figura 5.5a, por exemplo, os nós 8, 3 e 7 possuem grau 3, enquanto os nós 5, 4 e 6 possuem grau 1. A Figura 5.5b, por outro lado, possui menos violações, com a maioria dos nós tendo grau 2, com exceção dos nós 8, 4, 7 e 2. Por fim, a Figura 5.5c mostra uma solução viável tanto para (TSP_λ) quanto para (TSP), não violando nenhuma restrição de grau.

5.6.1 Resolução do MS1TP

Agora, resta encontrar uma maneira eficiente de resolver (TSP_λ). Para isso, primeiramente, considere um outro problema parecido, entitulado Problema da Árvore Geradora Mínima (*Minimum Spanning Tree*, MST). O MST visa, dado um grafo com pesos G, encontrar um subgrafo conexo e sem ciclos, e cuja soma dos pesos das arestas é a menor possível. Uma solução para o MST em um grafo $G = (V = \{1, \ldots, 13\}, E = V^2)$ é ilustrada na Figura 5.6a (desconsidere o nó 0 ilustrado).

Agora, considere que deseja-se resolver o (TSP_λ) para o grafo $G' = (V' = \{0, 1, \ldots, 13\}, E' = V'^2)$, e que T é a MST do grafo G. Conforme ilustrado na Figura 5.6b, a 1-árvore mínima de G' pode ser encontrada conectando-se o vértice 0 aos dois vértices mais próximos em T. Em outras palavras, para resolver (TSP_λ) no grafo $G = (V = \{0, 1, \ldots, n\}, E = V^2)$, é necessário apenas (i) encontrar a MST do grafo $G' = (V \setminus \{0\}, (V \setminus \{0\})^2)$ e (ii) conectar o nó 0 aos

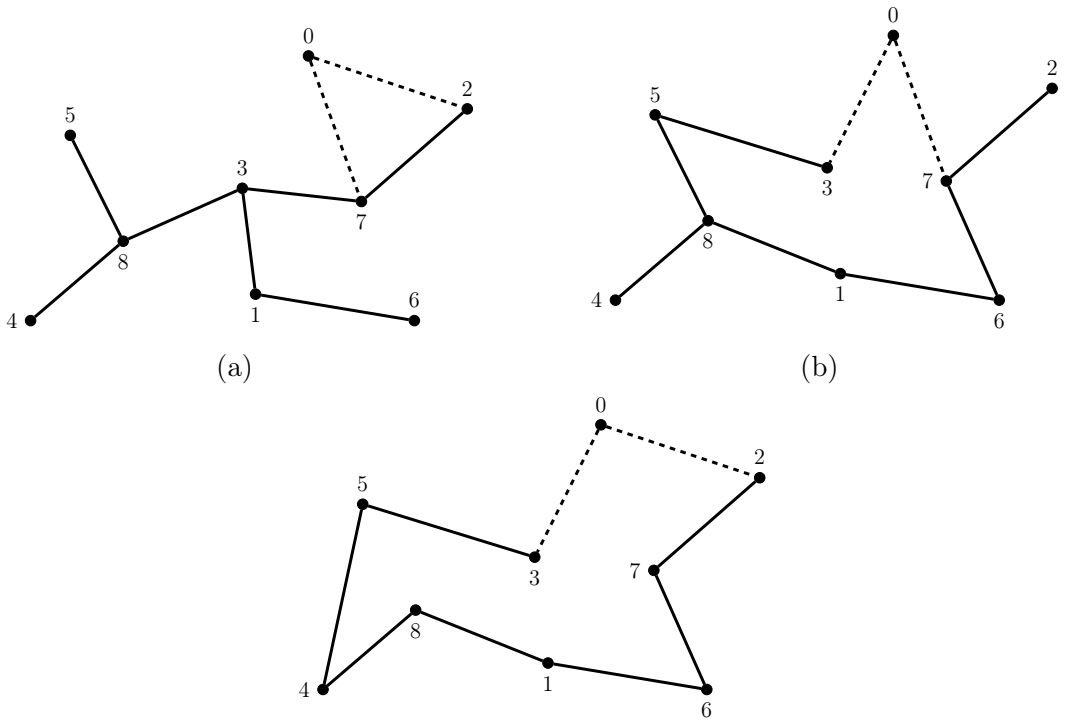


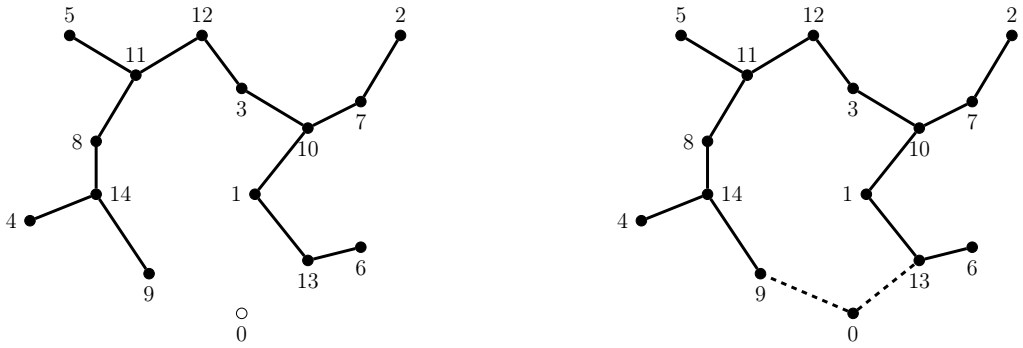
Figura 5.5: Exemplos de soluções para (TSP_λ)

dois nós mais próximos, formando uma 1-árvore. Uma implementação do algoritmo de Kruskal, que encontra a MST de um grafo de forma eficiente, está disponível em https://github.com/cvneves/kit-opt/tree/master/Relaxa%C3%A7%C3%A3o-Lagrangeana/min-spanning-tree. Lembre-se que é necessário desconsiderar o nó 0 antes de obter a MST, então deve-se modificar levemente o algoritmo ou a matriz de distâncias.

5.6.2 Exemplo numérico

Em posse do método descrito acima, é possível resolver o dual lagrangiano associado ao TSP usando o método do subgradiente conforme o Algoritmo 3. Para ilustrar o procedimento, considere um problema com 5 vértices, e cuja matriz de distâncias é

$$c = \begin{pmatrix} & 30 & 26 & 50 & 40 \\ & & 24 & 40 & 50 \\ & & & 24 & 26 \\ & & & & 30 \\ & & & & \end{pmatrix}.$$



(a) Árvore geradora de $V \setminus \{0\}$. (b) 1-árvore

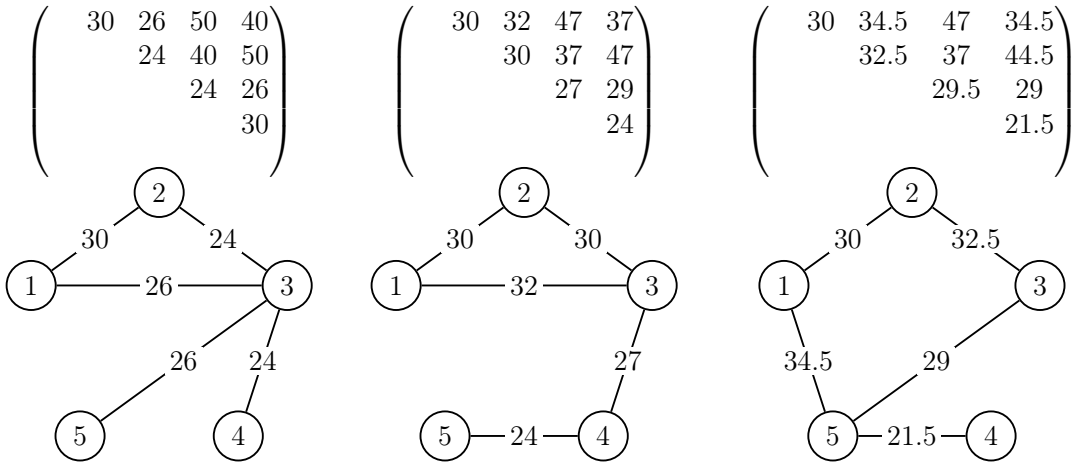
Figura 5.6: Obtendo a 1-árvore a partir da MST

Lembre-se que conforme definido na formulação de (TSP_λ), há um penalizador para cada vértice, e o penalizador do vértice 0 vale sempre 0. Iniciando o método com o vetor de penalizadores $\lambda = (0, 0, 0, 0, 0)$, a matriz de custos permanece intacta, e a MST encontrada com o método descrito acima é ilustrada na Figura 5.7a. Atualizando-se o vetor de penalizadores como na linha 10 do Algoritmo 3, obtém-se uma nova matriz de distâncias, e consequentemente uma 1-árvore diferente, ambos ilustrados na Figura 5.7b. Atualizando-se o vetor de penalizadores novamente, obtém-se a matriz de distâncias e a 1-árvore associada ilustradas na Figura 5.7c. Note que o número de nós de grau 2 é bem maior na última 1-árvore. Recomenda-se que o leitor verifique que os valores dos penalizadores mostrados estão corretos calculando-os manualmente (note que, para uma linha a_i da matriz A, tem-se que $b - a_i x^* = 2 - \text{grau}(i)$).

É necessário atentar-se ao fato de que a melhor 1-árvore encontrada pelo Algoritmo 3 pode não ser a última encontrada. Isso pode ser visualizado na Figura 5.8, que mostra o valor objetivo da 1-árvore encontrada a cada iteração do Algoritmo 3. Note que embora esteja convergindo, o valor objetivo oscila bastante. Por isso, é necessário manter armazenada a melhor 1-árvore obtida, bem como o vetor de penalizadores associados a essa 1-árvore.

5.7 Unindo a Relaxação Lagrangiana ao BnB

Embora a resolução do dual lagrangiano não garanta uma solução ótima para o TSP, pode-se resolver o problema de maneira exata unindo o método ao Branch-and-Bound. Para isso, basta utilizar o método descrito no capítulo anterior, substituindo-se o AP_{TSP} — resolvido a cada nó da árvore do BnB — pelo dual lagrangiano do TSP. Desta vez, no entanto, as soluções obtidas em cada nó da ár-



(a) $\lambda = (0, 0, 0, 0, 0)$ (b) $\lambda = (0, 0, -6, 3, 3)$ (c) $\lambda = (0, 0, -17/2, 3, 11/2)$

Figura 5.7: Soluções do (TSP_λ)

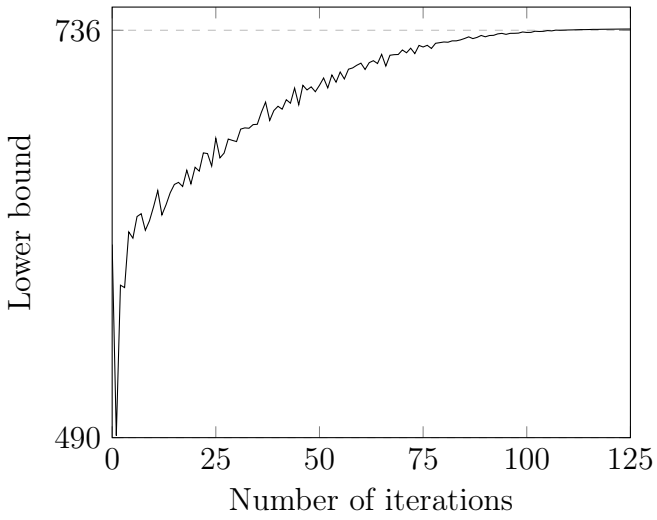


Figura 5.8: Valor objetivo do dual lagrangiano em função do número de iterações

vore do BnB não contém *subtours*, e portanto, a regra de branching também deve ser modificada como segue. Após a resolução do dual lagrangiano, caso a melhor 1-árvore obtida não seja um *tour*, deve-se encontrar o vértice com o maior grau e proibir os arcos associados a esse vértice nos nós filhos. Isso é ilustrado na Figura 5.9. Observe que na melhor 1-árvore obtida no primeiro nó, o vértice 5 possui o maior grau, estando conectado aos vértices 1, 3 e 4. Portanto, o nó deve gerar três filhos, cada um associado a um dos arcos proibidos. De forma semelhante ao capítulo anterior, o nó filho deve herdar os arcos proibidos do pai. Além disso,

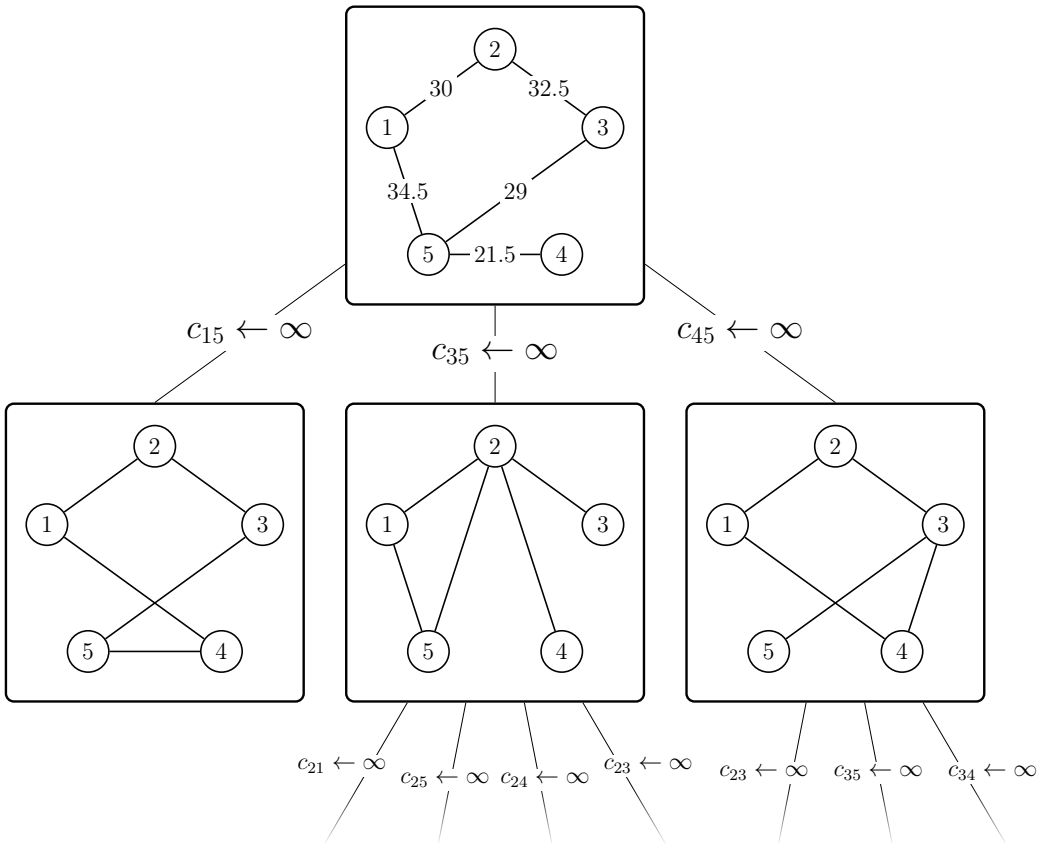


Figura 5.9: Representação em árvore do BnB com Relaxação Lagrangiana

os filhos também devem herdar os penalizadores associados à melhor 1-árvore do pai. Tais penalizadores devem ser utilizados como os penalizadores iniciais para resolver o dual lagrangiano. O leitor deve testar o algoritmo com as diferentes estratégias de busca descritas no capítulo anterior.

6

Branch-and-Cut

A formulação do TSP proposta por requer uma quantidade exponencial de espaço dada a natureza das restrições de *subtour*. Contudo, como será mostrado nesse capítulo, nem todas as restrições são necessárias para se obter a solução ótima. Em outras palavras, as restrições — ou cortes — de *subtour* podem ser geradas algoritmicamente "sob demanda".

6.1 *Lazy constraints*

Considere a seguinte formulação do TSP, sem as restrições de *subtour*:

$$\begin{aligned} \min & \sum_{i \in V} \sum_{j \in V, j>i} c_{ij} x_{ij} & \\ \text{s.a} & \sum_{j \in V, j<i} x_{ij} + \sum_{j \in V, j>i} x_{ij} = 2 & \forall i \in V \\ & x_{ij} \in \{0,1\} & \forall i,j \in V \end{aligned}$$

Utilizando-a como modelo, um *solver* produziria uma solução de estrutura semelhante às das soluções do algoritmo húngaro utilizado no Capítulo 2 — que possuem o somatório de pesos de arestas mínimo, mas que não garantem um único *tour*.

A partir da mesma formulação, uma instância com 6 nós, por exemplo, poderia gerar uma solução $s = \{\{1,3,5\},\{2,4,6\}\}$. Os dois *subtours* são indesejados, e após detectados, podem ser removidos adicionando-se as seguintes restrições

$$x_{13} + x_{35} + x_{51} \leq 3 - 1 \tag{6.1}$$

$$x_{24} + x_{46} + x_{62} \leq 3 - 1 \tag{6.2}$$

e resolvendo o problema novamente.

Se, ao resolver o problema novamente, uma solução ótima $s = \{1, 3, 5, 2, 4, 6\}$, fosse obtida, saberia-se que ela é a solução ótima do problema original, pois por conter apenas um *tour*, ela atende a todas as restrições de *subtour* implicitamente, mesmo que nem todas elas tenham sido usadas pelo *solver* durante a resolução.

As restrições 1 e 2 são chamadas de *lazy constraints*, pois ainda que sejam *necessárias* na procura de uma solução viável, são adicionadas apenas quando uma solução inteira que não satisfaz as restrições de *subtour* é encontrada.

Dessa forma, uma maneira de implementar um algoritmo exato que faz uso da noção de restrições sob demanda é utilizar a formulação matemática de sem as restrições de *subtour*. Após isso, sempre que uma solução ótima for obtida, pode-se utilizar um algoritmo simples de detecção de *subtours*[1], e proibi-los por meio da adição de restrições ao modelo e resolvê-lo novamente, até que uma solução inteira que contenha apenas um *tour* seja obtida.

6.2 *Min-Cut*

Dado um grafo $G = (V, E)$, se V é dividido em dois subconjuntos disjuntos V_1 e V_2 tais que $V_1 \cup V_2 = V$, diz-se que a partição $\{V_1, V_2\}$ é um corte de G. Suponha agora que G seja um grafo completo, e que K seja o conjunto de arestas compartilhadas por V_1 e V_2. Se $\sum_{e \in K} c_e$ é o mínimo possível, diz-se que a partição $\{V_1, V_2\}$ é um corte mínimo ou *min-cut* de G.

Dois métodos diferentes de obter o corte mínimo de um grafo serão apresentados nas próximas seções.

6.3 Cortes em um poliedro

Durante a resolução de um ILP como o TSP, *solvers* como o *CPLEX* fazem uso de algoritmos BB. Mais precisamente, relaxações lineares do problema original são tratadas como subproblemas, e escolhe-se as variáveis mais fracionárias para realizar o *branching*, que consiste em adicionar restrições de integralidade a essas variáveis. Dessa forma, o *CPLEX* itera sobre soluções fracionárias até que uma solução inteira seja obtida.

Como visto anteriormente, se a solução inteira violar as restrições de *subtour*, pode-se utilizar um algoritmo de separação e adicionar as restrições violadas até que uma solução inteira viável — e consequente ótima, nesse caso — seja ob-

[1] Tais algoritmos também podem ser chamados de algoritmos de separação, pois são capazes de separar uma solução inteira inviável do poliedro que descreve o problema ao adicionar restrições que a proiba.

tida. Acontece que, utilizando-se a mesma noção, é possível adicionar restrições no problema durante a resolução da relaxação linear (subproblema).

Na Figura 6.1, o maior poliedro representa a relaxação linear do problema original. Adicionando-se mais restrições, é possível "cortar"o poliedro original, aproximando-o mais da envoltória convexa de P. Por esse motivo, tais restrições adicionais também são chamadas de cortes[2].

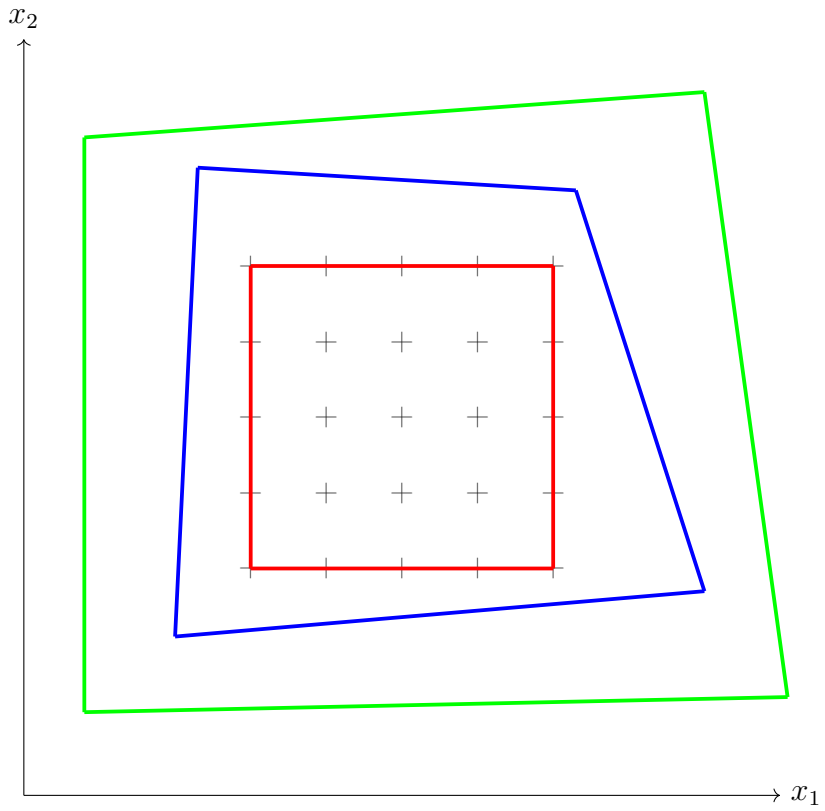


Figura 6.1: Poliedros.

6.4 Cortes no TSP

Uma solução fracionária para a relaxação linear do TSP sem restrição de *subtours* pode ser interpretada como um grafo completo. No grafo, todas as arestas terão um peso que varia de 0 a 1, e que obedecem às restrições de grau. Suponha que, ao resolver a versão relaxada do TSP, um *solver* chegue a uma solução fracionária cujo

[2]Observe que, embora estejam relacionados nesse contexto, cortes em um poliedro são totalmente diferentes de cortes de um grafo.

corte mínimo é ilustrado na Figura 6.2. Nela, o somatório dos pesos das arestas compartilhadas por S e $\overline{S}$ é maior que 2. Isso significa que a solução, mesmo que fracionária, obedece a todas as restrições de *subtour*, já que o corte mostrado é o menor possível.

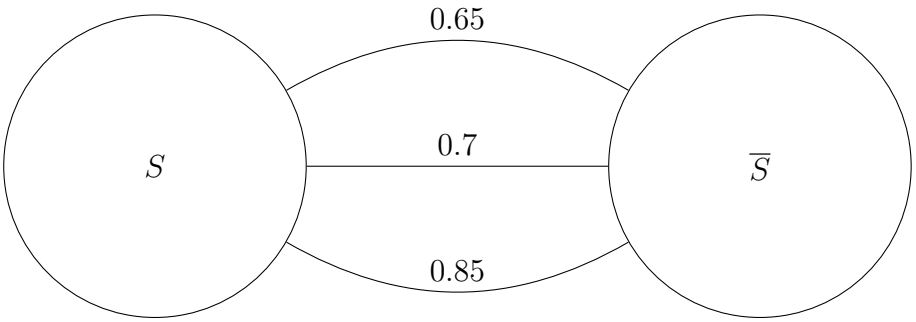


Figura 6.2: Solução fracionária para o TSP.

Se, no entanto, o somatório dos pesos das arestas fosse menor que 2, tanto S quanto $\overline{S}$ estariam violando as restrições de *subtour*, o que pode ser contornado adicionando-se a restrição $\sum_{i \in S} \sum_{j \in \overline{S}} x_{ij} \geq 2$. Dessa forma, múltiplas restrições de *subtour* podem ser adicionadas ao modelo antes mesmo de uma solução inteira ser obtida, além de aproximar o poliedro cada vez mais de sua envoltória convexa.

Conhecer o *min-cut* do grafo que representa as soluções fracionárias da relaxação linear do TSP torna-se, portanto, uma vantagem.

6.5 Algoritmos para a obtenção do *min-cut*

Nesta seção, serão descritos dois métodos para a obtenção do *min-cut*. Precisamente, na Seção 6.5.1, será apresentada a heurística *max-back* [10], um método bastante eficiente para a obtenção de cortes que, embora não garanta a obtenção do corte mínimo, resolve o problema da separação para soluções inteiras. Por outro lado, na Seção 6.5.2, é apresentado o algoritmo exato para obtenção do corte mínimo [13]. Recomenda-se, primeiramente, o uso do *max-back* na tentativa de detectar violações das restrições de *subtour*. Se elas não forem encontradas, utiliza-se o algoritmo exato. Detalhes sobre a implementação serão discutidos na Seção 6.5.3.

Algoritmo 4: MAX-BACK HEURISTIC

Dados: ..
Resultado:
1 **início**
2 **retorna**

Algoritmo 5: MINIMUM CUT

Dados: ..
Resultado:
1 **início**
2 **retorna**

6.5.1 Heurística *Max-back*

6.5.2 Algoritmo exato para o *Min-cut*

6.5.3 Implementação

Neste seção, estão ressaltados os principais trechos de código que compõem a implementação do algoritmo *Branch-and-Cut* aplicado ao TSP. O Código 6.1, por exemplo, apresenta o modelo para o TSP sem as restrições de proibição de *subtours*. Note que são definidas as variáveis associadas às arestas presentes no grafo, a função objetivo como a minimização da distância total percorrida e apenas as restrições de grau.

Código 6.1: Modelo TSP sem as restrições de proibição de subtours

```
1     IloEnv env;
2     IloModel model(env);
3
4     env.setName("Branch and Cut");
5     model.setName("Symmetrical Traveling Salesman Problem");
6
7     int dimension = data->getDimension();
8
9     /******** Creating one variable 'x' for each edge *******/
10    IloArray <IloBoolVarArray> x(env, dimension);
11
12    for (int i = 0; i < dimension; i++) {
13        IloBoolVarArray array(env, dimension);
14        x[i] = array;
15    }
```

```
16
17      /********** Adding variables 'x' to the model **********/
18      char var[100];
19      for (int i = 0; i < dimension; i++){
20          for (int j = i + 1; j < dimension; j++){
21              sprintf(var, "X(%d,%d)", i, j);
22              x[i][j].setName(var);
23              model.add(x[i][j]);
24          }
25      }
26
27      /**************** Objective Function *****************/
28      IloExpr obj(env);
29      for (int i = 0; i < dimension; i++) {
30          for (int j = i + 1; j < dimension; j++) {
31              obj += data->getDistance(i, j)*x[i][j];
32          }
33      }
34      model.add(IloMinimize(env, obj));
35
36      /******************** Constraints *********************/
37      IloRange r;
38      char name[100];
39
40      for (int i = 0; i < dimension; i++){
41          IloExpr sumX(env);
42          for (int j = 0; j < dimension; j++){
43              if (j < i){
44                  sumX += x[j][i];
45              }
46              if (i < j){
47                  sumX += x[i][j];
48              }
49          }
50          r = (sumX == 2);
51          sprintf(name, "c_%d", i);
52          r.setName(name);
53          model.add(r);
54      }
```

Definido o modelo, o próximo passo consiste na implementação dos cortes que proibirão soluções contendo *subtours*. Para tanto, é preciso utilizar funções denominadas *callbacks*. No escopo do resolvedor CPLEX, tais funções permitem que os usuários programem trechos de código que deverão ser executados durante a resolução do modelo [7]. Precisamente, são utilizadas três classes de *callbacks*, denominadas *MyBranchCallback*, *MyCutCallback* e *MyLazyCallback*, cujos comandos para suas inicializações estão descritos no Código 6.2.

Código 6.2: Comandos do CPLEX para a utilização das *callbacks*

```
    // [...]

    IloCplex STSP(model);

    /********** Creating Branch Callback Object **********/
    MyBranchCallback* branchCbk = new (env) MyBranchCallback(env);
    STSP.use(branchCbk);

    /************ Creating Cut Callback Object ************/
    MyCutCallback* cutCbk = new (env) MyCutCallback(env, x_ref, dimension)
        ;
    STSP.use(cutCbk);

    /************ Creating Lazy Callback Object ***********/
    MyLazyCallback* lazyCbk = new (env) MyLazyCallback(env, x_ref,
        dimension);
    STSP.use(lazyCbk);

    /**************** Cleaning the memory *****************/
    delete branchCbk;
    delete cutCbk;
    delete lazyCbk;
    env.end();

    // [...]
```

A classe *MyBranchCallback* é utilizada para [...]

Código 6.3: Código referente à classe *MyBranchCallback*

```
MyBranchCallback::MyBranchCallback(IloEnv env) : IloCplex::BranchCallbackI
    (env) {}

// returns a copy of the callback. Required by CPLEX
IloCplex::CallbackI* MyBranchCallback::duplicateCallback() const
{
   return new (getEnv()) MyBranchCallback(getEnv());
}

void MyBranchCallback::main()
{
  // How many branches would CPLEX create?
  IloInt const nbranch = getNbranches();

  if(nbranch > 0){
    // CPLEX would branch. Get the branches CPLEX would create
    // and create exactly those branches. With each branch store
```

```
17      // its NodeInfo.
18      // Note that getNodeData() returns NULL for the root node.
19      NodeInfo *data = dynamic_cast<NodeInfo *>(getNodeData());
20      unsigned int depth;
21
22      if(!data){
23          // se entrou aqui e porque o no atual e a raiz. Cplex nao guarda
                NodeInfo para a raiz, por isso o metodo acima retorna NULL.
24        // O NodeInfo da raiz e um objeto estatico da classe NodeInfo.
              Abaixo, verifica-se se objeto ja foi construido e aponta-se o
              ponteiro data para ele.
25        if(NodeInfo::rootData == NULL){
26          NodeInfo::initRootData();
27        }
28        data = NodeInfo::rootData;
29      }
30
31      depth = data->getDepth();
32
33      IloNumVarArray vars(getEnv());
34      IloNumArray bounds(getEnv());
35      IloCplex::BranchDirectionArray dirs(getEnv());
36
37      //cria as mesmas branches que o cplex criaria, mas coloca o NodeInfo.
38      for(IloInt i = 0; i < nbranch; ++i){
39        IloNum const est = getBranch(vars, bounds, dirs, i);
40        makeBranch(vars, bounds, dirs, est, new NodeInfo(depth + 1U));
41      }
42
43      dirs.end();
44      bounds.end();
45      vars.end();
46     }
47     else{
48      // CPLEX would not create any branch here. Prune the node.
49      prune();
50     }
51  }
```

Por sua vez, a classe *MyCutCallback* tem por finalidade [...]

Código 6.4: Código referente à classe *MyCutCallback*

```
1
2  /*********************** Class' Constructor ***********************/
3  MyCutCallback::MyCutCallback(IloEnv env, const IloArray<IloBoolVarArray>&
       x_ref, int nodes) : IloCplex::UserCutCallbackI(env), x(x_ref), x_vars(
       env), n(nodes)
4  {
```

```
5     /********** Filling x_vars **********/
6     for(int i = 0; i < n; i++) {
7       for(int j = i + 1; j < n; j++){
8         x_vars.add(x[i][j]);
9       }
10    }
11    /************************************/
12  }
13
14  /****** Return a callback copy. This method is a CPLEX requirement ******/
15  IloCplex::CallbackI* MyCutCallback::duplicateCallback() const
16  {
17     return new (getEnv()) MyCutCallback(getEnv(), x, n);
18  }
19
20  /**************** Callback's code that is runned by CPLEX ***************/
21  void MyCutCallback::main()
22  {
23    /**************** Getting the node's depth ****************/
24    //int depth = 0;
25    /*NodeInfo *data = dynamic_cast<NodeInfo*>(getNodeData());
26
27    if (!data){
28          if (NodeInfo::rootData == NULL)
29              NodeInfo::initRootData();
30
31      data = NodeInfo::rootData;
32      }
33    if (data) {
34      depth = data->getDepth();
35    }*/
36    int depth = getCurrentNodeDepth();
37    /**********************************************************/
38
39    /********** Getting the relaxed variables values **********/
40    IloNumArray x_vals(getEnv(), (0.5*(n)*(n-1)));
41    getValues(x_vals, x_vars);
42    /**********************************************************/
43
44    vector< vector<int> > cutSetPool;
45    vector<IloConstraint> cons;
46
47    double **x_edge = new double*[n];
48
49    for (int i = 0; i < n; i++) {
50      x_edge[i] = new double[n];
51    }
52
53    int l = 0;
```

```
54    for(int i = 0; i < n; i++) {
55      for(int j = i+1; j < n; j++) {
56        x_edge[i][j] = x_vals[l++];
57      }
58    }
59
60    cutSetPool = MaxBack(x_edge, n);
61
62    if (cutSetPool.empty() && depth <= 7) {
63      cutSetPool = MinCut(x_edge, n);
64      //cutSetPool = MultipleMinCut(x_edge, n);
65    }
66
67    /***************** Creating the constraints ***************/
68    if (!cutSetPool.empty()){
69      for (int c = 0; c < cutSetPool.size(); c++) {
70        IloExpr p(getEnv());
71        for(int i = 0; i < cutSetPool[c].size(); i++){
72          for(int j = 0; j < cutSetPool[c].size(); j++){
73            if(cutSetPool[c][i] < cutSetPool[c][j]){
74              p += x[cutSetPool[c][i]][cutSetPool[c][j]];
75            }
76          }
77        }
78        int RHS = cutSetPool[c].size();
79        cons.push_back(p <= RHS - 1);
80      }
81      /************ Adding the constraints to the model **********/
82      for(int i = 0; i < cons.size(); i++){
83        add(cons.at(i)).end();
84      }
85      /************************************************************/
86    }
87    /************************************************************/
88
89    /******************* Cleaning the memory *****************/
90    for (int i = 0; i < n; i++) {
91      delete[] x_edge[i];
92    }
93    delete[] x_edge;
94    /************************************************************/
95  }
```

Por fim, a classe *MyLazyCallback* é necessária para [...]

Código 6.5: Código referente à classe *MyLazyCallback*

Encerrar com um parágrafo de conclusão e indicando onde encontrar o código

referente a estrutura completa (que está sendo compartilhado como arquivo zip por e-maail atualmente).

Referências Bibliográficas

[1] Lagrangean relaxation. https://github.com/carlosvinicius01/kit-opt/blob/master/Relaxa%C3%A7%C3%A3o-Lagrangeana/LagrangianRelaxation.pdf.

[2] Tsp 1 tree. https://github.com/carlosvinicius01/kit-opt/blob/master/Relaxa%C3%A7%C3%A3o-Lagrangeana/TSP-1-tree.pdf.

[3] Tsp 1 tree. https://github.com/carlosvinicius01/kit-opt/blob/master/Relaxa%C3%A7%C3%A3o-Lagrangeana/relaxlag-2p.pdf.

[4] Teobaldo Bulhões. Exemplo de solução para o BPP, 2019. Notas de aula da disciplina Pesquisa Operacional, oferecida pelo Centro de Informática da Universidade Federal da Paraíba.

[5] Maxence Delorme, Manuel Iori, and Silvano Martello. Bpplib: a library for bin packing and cutting stock problems. *Optimization Letters*, 12:1–16, 03 2018.

[6] Monique Guignard. Lagrangean relaxation. *Top*, 11(2):151–200, December 2003.

[7] ILOG CPLEX Optimization Studio (IBM). What are callbacks? https://www.ibm.com/docs/en/icos/20.1.0?topic=callbacks-what-are, 2021.

[8] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.

[9] Nelson Maculan and Marcia H Costa Fampa. *Otimização Linear*. 2004.

[10] Denis Naddef. Polyhedral theory and branch- and-cut algorithms for the symmetric tsp. chapter 2, pages 84–85. Laboratoire Informatique et Distribution, Institut National Polytechnique de Grenoble, France.

[11] Marcos Melo Silva, Anand Subramanian, Thibaut Vidal, and Luiz Satoru Ochi. A simple and effective metaheuristic for the minimum latency problem. *European Journal of Operational Research*, 221(3):513 – 520, 2012.

[12] Marcone Jamilson Freitas Souza. Inteligência computacional para otimização. Instituto de Ciências Exatas e Biológicas, Universidade Federal de Ouro Preto, 35400-000 Ouro Preto, MG, jan 2011.

[13] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, July 1997.