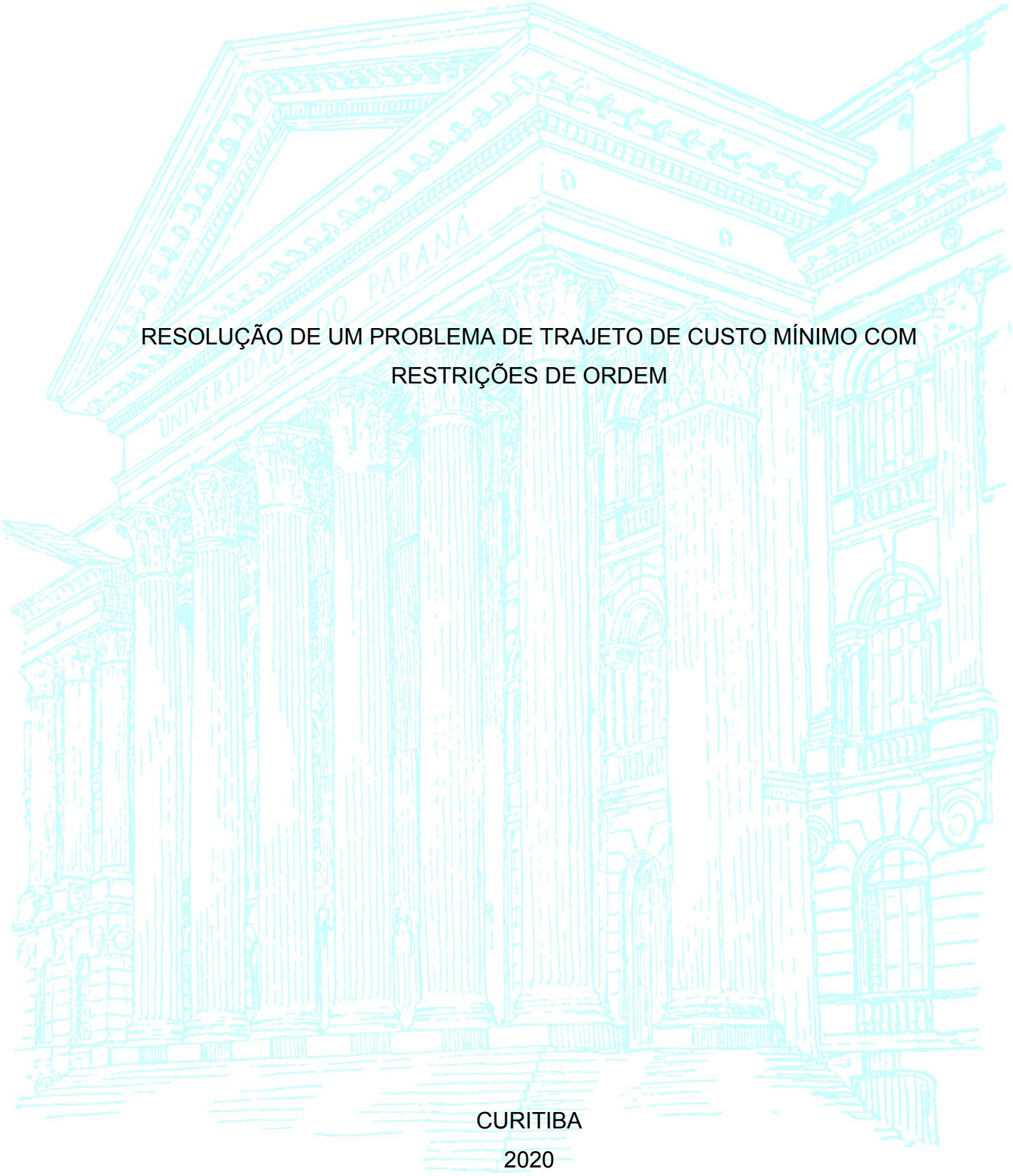
UNIVERSIDADE FEDERAL DO PARANÁ

FERNANDO ZANUTTO MADY BARBOSA



RESOLUÇÃO DE UM PROBLEMA DE TRAJETO DE CUSTO MÍNIMO COM RESTRIÇÕES DE ORDEM

CURITIBA

2020

FERNANDO ZANUTTO MADY BARBOSA

RESOLUÇÃO DE UM PROBLEMA DE TRAJETO DE CUSTO MÍNIMO COM RESTRIÇÕES DE ORDEM

Artigo apresentado como requisito parcial à conclusão da disciplina de Otimização/Tópicos em Algoritmos, Setor de Informática, Universidade Federal do Paraná.

Professor: Prof. Dr. André Luiz Pires Guedes

CURITIBA

2020

SUMÁRIO

1. **INTRODUÇÃO**

O problema abordado neste artigo consiste em encontrar um trajeto de menor custo, com restrições de ordem, utilizando a técnica de otimização *Branch and Bound*. Como no exemplo dado pelo orientador: “Leopoldina tem n lugares para visitar em sua cidade. Para evitar sair muito, ela vai fazer tudo em uma única saída, passando por um lugar de cada vez e voltando pra casa. Mas alguns lugares têm que ser visitados em uma ordem específica. Por exemplo, suponha que ela tem que passar no banco antes de ir à padaria. Leopoldina quer fazer isso gastando o mínimo tempo possível. Para saber o melhor caminho ela anotou todos os tempos de deslocamento entre cada par de lugares e listou as restrições de ordem aos pares. Qual a ordem que Leopoldina deve fazer suas visitas, garantindo as restrições de ordem em tempo mínimo?”

Um leitor atento provavelmente notou a semelhança do problema descrito com o conhecido problema Traveling Salesman (TSP). Se retirarmos as restrições de ordem os dois problemas levarão à mesma resposta, ou seja, é possível adaptar a solução de Branch and Bound do TSP para considerar as restrições e resolver o problema aqui proposto. Realizaremos tal adaptação ao longo dos próximos capítulos utilizando o capítulo 4.7 do livro *Combinatorial Algorithms: Generation, Enumeration, and Search*, dos autores Donald L. Kreher and Doug Stinson.

1.2 TRAVELING SALESMAN (TSP)

Nos moldes do TSP, Leopoldina deve partir de sua casa, visitar apenas uma vez todos os n locais de um conjunto V e retornar a casa. Representaremos um trajeto como uma lista $\overline{X}$ = [$x_0, x_1, \ldots x_{n-1}$¿, onde $x_i \in V$. O primeiro local é a casa, portanto $x_0 = 0$. Note que $[x_1, x_2, \ldots, x_{n-1}]$ é uma permutação de [1,...,n-1].

Dado:

- Número “n” de lugares (contando com a casa)
- matriz “A” de custos, onde uma posição (i,j) na matriz corresponde o custo de sair do local i para ir ao local j. O custo de sair e voltar ao mesmo local (k,k) é irrelevante para a solução.

Encontrar uma permutação $\overline{X}$ de [0,1,..., n-1] tal que:

- cost($\overline{X}$) = $\sum_{i=0}^{n-1} A[x_i][x_{i+1}]$

Observe que x_n = 0, já que Leopoldina retorna a casa.

Exemplo:

$$n = 3;\ A = \begin{matrix} 0 & 3 & 4 \\ 3 & 0 & 5 \\ 4 & 5 & 0 \end{matrix}$$

Trajetos possíveis:

[0,1,2] [0,2,1]

Trajeto com menor custo:

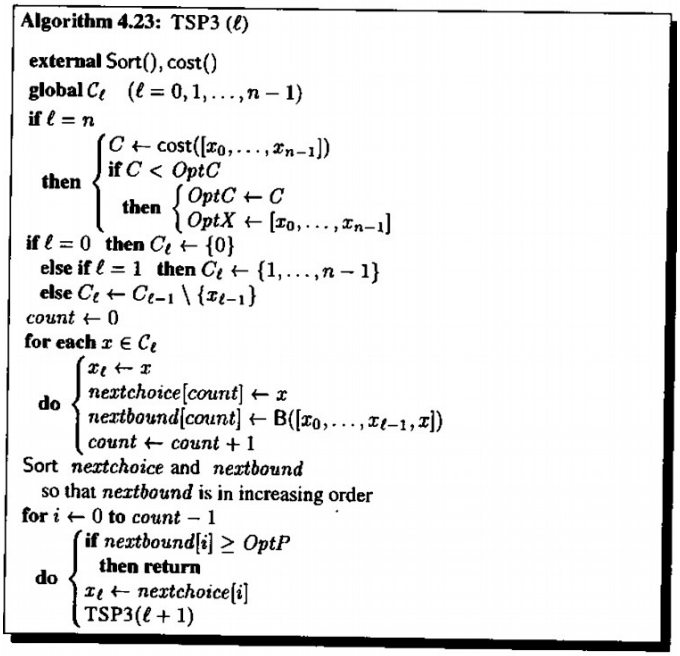
$\overline{X}$ = [0, 1, 2]

cost($\overline{X}$) = A[0][1] + A[1][2] + A[2,0] = 3 + 5 + 4 = 12

2. ADAPTANDO O CÓDIGO

Recomendamos fortemente a leitura do capítulo 4 do livro *Combinatorial Algorithms* antes de seguir aos próximos capítulos, pois o código original não será explicado a fundo.

FIGURA 1 – ALGORITMO ORIGINAL



Algorithm 4.23: TSP3 (ℓ)

```
external Sort(), cost()
global C_ℓ   (ℓ = 0, 1, ..., n − 1)
if ℓ = n
  then { C ← cost([x_0, ..., x_{n−1}])
         if C < OptC
           then { OptC ← C
                  OptX ← [x_0, ..., x_{n−1}] }
if ℓ = 0 then C_ℓ ← {0}
  else if ℓ = 1 then C_ℓ ← {1, ..., n − 1}
  else C_ℓ ← C_{ℓ−1} \ {x_{ℓ−1}}
count ← 0
for each x ∈ C_ℓ
  do { x_ℓ ← x
       nextchoice[count] ← x
       nextbound[count] ← B([x_0, ..., x_{ℓ−1}, x])
       count ← count + 1 }
Sort nextchoice and nextbound
   so that nextbound is in increasing order
for i ← 0 to count − 1
  do { if nextbound[i] ≥ OptP
         then return
       x_ℓ ← nextchoice[i]
       TSP3(ℓ + 1) }
```

FONTE: Kreher e Stinson (1999)

FIGURA 2 - FUNÇÃO TSP2

```
24  def TSP2(A, l, X, C, n, order, second):
25      global OptC
26      global OptX
27      global nodes
28      nodes += 1
29      if(l==n):
30          c = b1.cost(A, X)
31          if(c < OptC):
32              OptC = c
33              OptX = X[:]
34          return
35
36      if(l==0):
37          try:
38              C[l] = [0]
39          except IndexError:
40              C.append([0])
41
42      else:
43          if(l==1):
44              try:
45                  C[l] = range(1, n)
46              except IndexError:
47                  C.append(range(1, n))
48          else:
49              aux = C[l-1][:]
50              aux.remove(X[l - 1])
51              try:
52                  C[l] = aux[:]
53              except IndexError:
54                  C.append(aux[:])
55
56      choices = []
57      for i in C[l]:
58          X_aux = X + [i]
59          if(second):
60              aux = (i, b1.bound2(A, X_aux, n, order))
61          else:
62              aux = (i, b1.bound1(A, X_aux, n))
63          choices.append(aux)
64
65      choices.sort(key=lambda choice: choice[1])
66
67      for i in choices:
68          if b1.wrongOrder(X, i[0], order):
69              continue
70          if(i[1] >= OptC):
71              return
72          newX = X + [i[0]]
73          TSP2(A, l+1, newX, C, n, order, second)
```

FONTE: O autor (2020)

As linhas 29 a 54 fazem quase uma transcrição do livro, calculando os possíveis próximos vértices C_l, ou *choice set*. Os blocos try-except evitam o acesso de posições ainda não declaradas de C_l.

As linhas 56 a 65 calculam o custo mínimo para cada possível filho da solução parcial $\overline{X}$ e ordena de forma crescente (em relação à função bound) a dupla

$(x_i, bound(x_i)¿$. O bloco If-else da linha 59 apenas seleciona uma das duas funções disponíveis (posteriormente explicadas).

Na linhas 67 a 73 fazemos dois processos essenciais: inserção das restrições de ordem e a poda da árvore. A função *wrongOrder* exclui da busca de melhor trajeto as permutações que não atendem as restrições de ordem. O if da linha 70 faz a poda de filhos que já possuem um custo mínimo maior que o custo ótimo encontrado. Note que, como os possíveis filhos de $\overline{X}$ estão em ordem crescente de função custo, $i_k \geq Optc \rightarrow i_{k+1} \geq Optc$ portanto podemos retornar à chamada anterior de TSP2. Caso um possível filho passe pelos dois processos, chamamos a função TSP2 para o nodo do filho.

FIGURA 3 - FUNÇÃO “wrongOrder”

```
37  def wrongOrder(X, destiny, order):
38      for i in order:
39          if (destiny == i[1]) and (i[0] not in X):
40              return True
41      return False
42
```

FONTE: O autor (2020)

A função wrongOrder retorna *True* caso não seja possível ir ao vértice *destiny* devido às restrições de ordem. Caso contrário, retorna *False.*

3. PRIMEIRA FUNÇÃO LIMITANTE

Neste capítulo tentaremos utilizar a função limitante da imagem 4, criada para o problema TSP.

FIGURA 4: FUNÇÃO DE BOUND 1

For $x \in \mathcal{V}$ and $\mathcal{W} \subseteq \mathcal{V}$ (where $\mathcal{W} \neq \emptyset$), define

$$\mathrm{b}(x, \mathcal{W}) = \min\{\mathrm{cost}(x, y) : y \in \mathcal{W}\}.$$

We now prove an inequality that will lead to a bounding function.

THEOREM 4.2 *Let $X' = [x_0, \ldots, x_{n-1}]$ be the minimum cost Hamiltonian circuit that extends $[x_0, x_1, \ldots, x_{\ell-1}]$, where $\ell \leq n-1$. Then it holds that*

$$\mathrm{cost}(X') \geq \sum_{i=0}^{\ell-1} \mathrm{cost}(x_i, x_{i+1}) + \mathrm{b}(x_{\ell-1}, \mathcal{Y}) + \sum_{y \in \mathcal{Y}} \mathrm{b}(y, \mathcal{Y} \cup \{x_0\}).$$

FONTE: Kreher e Stinson (1999)

Como primeiro passo é necessário verificar que um limite inferior do problema TSP também limita inferiormente o custo mínimo do trajeto da Leopoldina. Considere:

- T: conjunto de possíveis permutações (ou trajetos) do problema TSP que estendem $[x_0, x_1, \ldots, x_{l-1}]$
- L: conjunto de possíveis trajetos da Leopoldina que estendem $[x_0, x_1, \ldots, x_{l-1}]$
- O: conjunto de impossíveis trajetos (devido às restrições de ordem) da Leopoldina que estendem $[x_0, x_1, \ldots, x_{l-1}]$
- X^k: trajeto de custo mínimo do conjunto K

Como L = T – O, se L é não vazio segue:

$$L \subseteq T \rightarrow X^L \in T \therefore cost(X^L) \geq cost(X^T)$$

Portanto a função da figura 4 também limita o problema da Leopoldina.

3.1 IMPLEMENTAÇÃO

A implementação da função limitante da figura 4 é feita na função bound1 da figura 5. Na linha 24 calculamos a lista y de vértices ainda não selecionados. O loop da linha 26 computa o custo já presente na solução parcial X = $[x_0, x_1, \ldots, x_{l-1}]$, note que i varia de 0 a L-2. Na linha 29 encontramos a aresta de menor custo $[x_{l-1}, k]$, onde $k \in y$. As linhas 31 a 33 somam as arestas de menor custo tendo como partida cada vértice de y.

FIGURA 5 – FUNÇÃO BOUND1 CÓDIGO

```
18  def bound1(A, X, n):
19      l = len(X)
20      if(l<1): return -1
21      if(l==n): return cost(A, X)
22      total = 0
23
24      y = [i for i in range(n) if i not in X]
25
26      for i in range(l-1):
27          total += A[X[i]][X[i+1]]
28
29      total += boundAux(A, X[-1], y)
30
31      y_aux = y + [0]
32      for i in y:
33          total += boundAux(A, i, y_aux)
34
35      return total
```

FONTE: O autor (2020)

A função boundAux da figura 6 realiza a operação descrita na figura 4 como $b(x, W)$. Dada um ponto de partida x e um conjunto W de possíveis destinos, a função retorna o menor custo do trajeto $[x, ?]$.

FIGURA 6 – FUNÇÃO “boundAux”

```
2   def boundAux(A, x, W):
3       min = float("inf")
4       for y in W:
5           if ((x!=y) and (A[x][y]) <= min):
6               min = A[x][y]
7       return min
8
```

FONTE: O autor (2020)

3.2 EXEMPLO DE EXECUÇÃO

Considere:

- o trajeto parcial $X = [0,1]$
- uma matriz de custos A =

0	1	2	3	4
5	0	6	7	8
9	10	0	11	12
13	14	15	0	16
17	18	19	20	0

O custo já presente no trajeto é a aresta [0,1] = 1. Ao sair do vértice 1, temos as opções y = [2,3,4], a de menor custo é [1, 2] = 6. Como precisamos sair dos vértices do conjunto y de alguma forma, listamos as arestas que têm como partida cada elemento de y e escolhemos as de menor custo.

- [2, 4] [2, 3] [2,0] → Menor custo: [2,0] = 9
- [3,2] [3,4] [3,0] → Menor custo: [3,0] = 13
- [4,2] [4,3] [4,0] → Menor custo: [4,0] = 17

Portanto, bound1(X) = 1 + 6 + 9 + 13 + 17 = 46

4. SEGUNDA FUNÇÃO LIMITANTE

Como segunda função que limita inferiormente o custo de um trajeto parcial $\overline{X}$, utilizaremos a função do capítulo 3 aprimorada com as restrições de ordem. Três aspectos foram considerados em tal aprimoramento. Dada uma restrição $[o_1, o_2]$:

Aspecto 1: Se $o_1 \notin \overline{X}$, a aresta $[x_{l-1}, o_2]$ é impossível de ocorrer, já que o_1 não seria incluído no trajeto final.

Aspecto 2: A aresta $[o_1, 0]$ é impossível de ocorrer, já que o_2 não seria incluído no trajeto final.

Aspecto 3: A aresta $[o_2, o_1]$ é impossível ocorrer.

FIGURA 7: FUNÇÃO BOUND2

```
48  def bound2(A, X, n, order):
49      l = len(X)
50      if (l < 1): return -1
51      if (l == n): return cost(A, X)
52      total = 0
53
54      y = [i for i in range(n) if i not in X]
55      column1 = [i[0] for i in order]
56      column2 = [i[1] for i in order]
57
58      for i in range(l - 1):
59          total += A[X[i]][X[i + 1]]
60
61
62      y_aux = y[:]
63      for element in order:
64          if element[0] not in X and element[1] in y_aux:
65              y_aux.remove(element[1])
66
67
68      total += boundAux(A, X[-1], y_aux)
69
70
71      for i in y:
72          if i in column1:
73              y_aux = y[:]
74          else:
75              y_aux = y + [0]
76
77          if i in column2:
78              try:
79                  y_aux.remove(column1[column2.index(i)])
80              except ValueError:
81                  pass
82
83          total += boundAux(A, i, y_aux)
84
85      return total
```

FONTE: O autor (2020)

No loop da linha 63 eliminamos dos possíveis custos as arestas que afetam o aspecto 1. O bloco if-else da linha 72 verifica se a aresta [i, 0] é viável, de acordo com o aspecto 2. O bloco try-except tenta remover qualquer aresta que afeta diretamente a ordem, como no aspecto 3.

Como a função boundAux(A, i, W) escolhe a menor aresta de um dado conjunto W, é evidente que:

$$Q \subseteq W \rightarrow boundAux(A, i, Q) \geq boundAux(A, i, W)$$

Portanto a segunda função *bound2* deve fornecer um limite inferior mais próximo ao real custo mínimo de um trajeto que estende $[x_0, x_1, \ldots, x_{l-1}]$.

4.1 EXEMPLO DE EXECUÇÃO.

Para a matriz A e trajeto parcial $\overline{X}$ dados no capítulo 3.2, vamos recalcular o custo mínimo utilizando a função *bound2*. Ao sair do vértice 1, temos as opções y = [2,3], a de menor custo é [1, 2] = 6.

- [2, 4] [2, 3] [2,0] → Menor custo: [2,0] = 9
- [3,2] [3,4] → Menor custo: [3,2] = 15
- [4,2] [4,0] → Menor custo: [4,0] = 17

Portanto, bound2(X) = 1 + 6 + 9 + 15 + 17 = 48.

5 TESTES

5.1 COMPARAÇÃO DE NÓS

Para verificar se a segunda função limitante realmente fornece um limite inferior mais aprimorado, comparam-se os nós percorridos ao utilizar as duas funções. Esperamos que número de nós da função 2 seja menor ou igual ao número de nós da função 1, já que cortamos os nós com custo mínimo maior ou igual ao custo ótimo encontrado (linha 70 da figura 2).

Antes de realizar este teste precisamos fornecer iguais condições às funções. Note que, ao utilizar a função 2 alteramos o valor de custo mínimo dos possíveis filhos da solução parcial X e posteriormente ordenamos de forma crescente. Como a técnica *Branch and Bound* utiliza esta ordem para buscar a melhor solução, a função 2 deve realizar a busca na mesma ordem que a função 1. Fazendo isso, evitamos o caso em que a função 1 ganha da função 2 apenas com a "sorte" de receber uma ordem melhor da variável *choices.*

Na figura 8, chamamos a segunda função de bound com a ordem de ordenação da primeira. A função *callingBound2* da figura 10 refaz o custo mínimo de

cada elemento de *choice.* Na figura 9 trocamos “return” por “continue” já que *choices* não estará mais em ordem crescente.

FIGURA 8: MESMA ORDEM

```
58      choices = []
59      for i in C[l]:
60          X_aux = X + [i]
61          aux = [i, b1.bound1(A, X_aux, n)]
62          choices.append(aux)
63
64      choices.sort(key=lambda choice: choice[1])
65      if(second):
66          b1.callingBound2(A, choices, n, X, order)
```

FONTE: O autor (2020)

FIGURA 9: PEQUENA ALTERAÇÃO

```
72              if(i[1] >= OptC):
73                  continue#need to change
```

FONTE: O autor (2020)

FIGURA 10: FUNÇÃO “callingBound2”

```
37  def callingBound2(A, choices, n, X, order):
38      for choice in choices:
39          X_aux = X + [choice[0]]
40          choice[1] = bound2(A, X_aux, n, order)
41
```

FONTE: O autor (2020)

A tabela 1 Apresenta os resultados do teste realizado. O número de lugares (contando com a casa) e o número de restrições foram escolhidos pelo autor. A matriz de custos e os elementos das restrições são aleatórios. Os tempos de execução não foram considerados, já que *callingBound2* realiza o trabalho adicional de calcular cada bound1.

Como esperado, a quantidade de nós ao executar a função de bound2 foi menor ou igual aos nós da execução com bound1. Apesar da pequena modificação realizada, a função 2 na instância 6 diminuiu 67% da quantidade de nós, um resultado muito promissor.

TABELA 1 – COMPARAÇÃO DE NÓS

instância	Número n de lugares	Restrições de ordem	Nós função 1	Nós função 2
1	5	1	7	7
2	10	5	2362	1522
3	10	1	64	64
4	15	10	4695	3030

5	15	12	5775	2621
6	17	10	6454	2139
7	17	13	303278	191751
8	17	13	2727	1168
9	17	13	23449	11373
10	17	5	12692	12092
11	20	13	373936	196205
12	20	30	43675	21516

FONTE: O autor (2020)

5.2 COMPARAÇÃO DE TEMPO DE EXECUÇÃO

Neste segundo teste permitimos que a função de bound2 faça uma ordenação diferente da ordenação da função de bound1, como na figura 2. Esperamos que a função 2 corte mais nós na tabela 2, já que é provável encontrar a solução ótima em nós onde a função limitante é menor (KREHER; STINSON, 1999). Observe que, embora improvável, é possível que o tempo de execução ao utilizar a função limitante 1 seja menor, devido à diferença na ordem de *choices* e o trabalho adicional da função limitante 2. Nas instâncias 1-2-3-10 bound2 “perdeu” por pouco. Na instância 6, diminuímos em 57% o tempo de execução.

TABELA 2 – COMPARAÇÃO TEMPO DE EXECUÇÃO

Instância	Número n de lugares	Restrições de ordem	Nós função 1	Nós função 2	Tempo 1 (ms)	Tem 2 (ms)
1	5	1	7	7	0.56	0.61
2	10	5	2362	1522	312	326
3	10	1	64	64	20	23
4	15	10	4695	2722	3502	2695
5	15	12	5775	2525	3920	2394
6	17	10	6454	1912	7239	3079
7	17	13	303278	178216	212826	181485
8	17	13	2727	839	2764	1328
9	17	13	23449	10478	23604	15415
10	17	5	12692	11370	11366	123508
11	20	13	373963	197055	533007	391104
12	20	30	43675	20598	49943	37087

FONTE: O autor (2020)

6 CONCLUSÕES E SUGESTÕES

6.1 A MELHOR FUNÇÃO DE BOUND

Como mostrado nos capítulos anteriores, a função *bound2* corta mais nós em comparação à função *bound1.* Porém, uma execução de bound1 poderá ser mais rápida em três casos ruins:

1. Bound1 tem mais “sorte” na ordenação de *choices.*
2. O custo adicional de bound2 compromete o tempo de execução
3. Os dois acima

O primeiro caso não é possível controlar, já que não sabemos em qual nodo a solução ótima estará. Quando ao segundo caso, é muito provável que o número de nodos cortados pela função bound2 compensará o custo adicional. No capítulo seguinte sugerimos algumas alterações para diminuir o custo da segunda função limitante.

6.2 SUGESTÕES PARA FUTURAS PESQUISAS

Neste capítulo serão apresentadas ideias que podem melhorar o desempenho do programa anteriormente discutido gerado para o problema de menor trajeto com restrições de ordem. É importante ressaltar que soluções melhores podem ser alcançadas adaptando qualquer função que limita inferiormente o custo de um trajeto do problema original TSP, como provado no capítulo 3. Outras funções de *bound* para TSP podem ser encontradas em [1] e [3].

6.2.1 Sugestões para a função “TSP2”

A técnica utilizada para percorrer as permutações viáveis (quanto à ordem) foi percorrer as (n-1)! permutações de [1,...,n-1] retirando aquelas que infligem a ordem. O custo do programa poderá será reduzido caso as permutações viáveis sejam calculadas previamente (não passo a passo).

Também é possível diminuir o tempo de execução de TSP2 verificando se as restrições de ordem tornam o problema impossível. Por exemplo, caso haja duas restrições opostas, [i, j] e [j, i], não há trajeto possível.

Como a linguagem utilizada, Python, é interpretada e dinâmica, sugere-se a implementação do problema em linguagens de programação compiladas nativamente como C e C++ para alcançar maior velocidade de execução.

6.2.2 Sugestões para a função “bound2”

Observe que os aspectos 2 e 3 do capítulo 4 podem ser calculados previamente, ao invés de calculá-las a cada execução do bound2 (linha 71 figura 7). O aspecto 1 pode ser previamente calculado caso refeito na forma:

Novo Aspecto 1: Se a aresta $[0, o_2]$ é impossível de ocorrer, já que o_1 não seria incluído no trajeto final.

O aspecto 1 também pode ser calculado na forma antiga com menos comparações a cada passo, basta ver que se $\overline{X}=[x_0, x_1, \ldots, x_{l-1}]$ $\overline{X'}=[x_0, x_1, \ldots, x_{l-1}, x_l]$:

$$se\ x_l \neq o_1, o_1 \notin \overline{X} \rightarrow o_1 \notin \overline{X'}$$

$$o_1 \in \overline{X} \rightarrow o_1 \in \overline{X'}$$

No cálculo do aspecto 1 podem ser usadas diversas técnicas de programação dinâmica (ver [2]).

REFERÊNCIAS

[1].KREHER, Donald L.; STINSON, Doug. **Combinatorial Algorithms**: Generation, Enumeration, and Search. [*S. l.*: *s. n.*], 1999.

[2].CORMEN, Thomas H. *et al*. **Introduction to Algorithms**: THIRD EDITION. [*S. l.*: *s. n.*], 1989.

[3].PAPADIMITRIOU, Christos H.; STEIGLITZ, Kenneth. **Combinatorial Optimization**: Algorithms and Complexity. [*S. l.*: *s. n.*], 1982.