Trabalho avaliativo 2 Otimização / Tópicos em Algoritmos

17 de agosto de 2021

Resumo

Este relatório documenta a resolução do problema `Caminhada Máxima` proposto como tarefa avaliativa da disciplina de Otimização/Tópicos em Algoritmos. Esse problema consiste em obter um passeio fechado de peso máximo que não repete vértices para um grafo não orientado com pesos positivos. A estratégia de resolução *Branch-and-Bound* (*BnB*) é utilizada. O diferencial da abordagem presente nesse relatório é utilizar uma modelagem em programação linear como função limitante superior para realização dos cortes no espaço de busca. O algoritmo de *BnB* é apresentado algoritmicamente e uma implementação em *Python* é descrita. Uma avaliação comparativa entre o funcionamento de duas versões do algoritmo de *BnB* (com e sem ordenação segundo o maior limitante) é realizada. Os resultados indicam que a versão com ordenamento (chamada *ordered BnB*) é mais eficiente.

1 Estratégia de *Backtracking*

Iniciamos esse relatório com uma breve descrição da estratégia de enumeração conhecida como *backtracking*. A palavra "backtracking", conforme informa o dicionário *Merriam-Webster*, significa refazer um caminho anteriormente percorrido, ou voltar a um ponto anterior numa determinada sequência. Como veremos, essa definição nos diz algo a respeito da estratégia de enumeração *backtracking*. Um algoritmo de *backtracking* (recursivamente) constrói, incrementalmente, cada um dos objetos do universo que se quer enumerar. Dizemos que a construção é recursiva pois várias chamadas aninhadas são realizadas durante a construção parcial de cada solução, e dizemos que a enumeração ocorre incrementalmente pois partimos de candidados parcialmente formados que gradualmente crescem até serem rejeitados pelo algoritmo ou serem reconhecidos como um objeto que se quer enumerar.

Nota: Podemos utilizar uma estratégia de enumeração para fins de otimização, mantendo uma memória que registra o melhor candidato conhecido ao longo da enumeração.

Vejamos o seguinte exemplo de algoritmo de backtracking que enumera todas as listas de entradas binárias de tamanho n. O Algoritmo 1 implementa o *backtracking* e usa a função `RecursiveListasBinarias` do Algoritmo 2 como auxílio.

`ListasBinarias`(n);
Entrada: Tamanho n das listas binárias desejadas
Saída: Todas as listas binárias de tamanho n.
$L \leftarrow \emptyset$;
$l \leftarrow$ lista vazia;
`RecursiveListasBinarias`(l);
Retornar L;

Algoritmo 1: Algoritmo de *backtracking* que enumera todas as listas binárias de tamanho n

`RecursiveListasBinarias`(l);
Entrada: Lista binária de tamanho l
Dados: Tem acesso a L e a n, pois é chamada no escopo de `ListasBinarias`.
$m \leftarrow$ tamanho de l;
se $m = n$ **então**
  inserir l em L;
fim
senão se $m < n$ **então**
  `RecursiveListasBinarias`($l + [1,]$);
  `RecursiveListasBinarias`($l + [0,]$);
fim

Algoritmo 2: Parte recursiva do algoritmo de *backtracking* descrito no Algoritmo 1

2 Estratégia de *Branch-and-Bound (BnB)*

A estratégia de *Branch-and-Bound (BnB)* é uma adaptação da estratégia de *backtracking* para maximizar (ou minimizar) uma determinada função de custo em um universo de canditatos. O distintivo da estratégia *BnB* é utilizar uma função auxiliar descrita como função de limitante superior que avalia se determinada solução parcial tem o potencial de ser completada para uma solução ótima. Com efeito, rejeitar uma solução parcial por ela não ter o potencial de ser completada de modo o superar o melhor candidato conhecido "corta" um nó da árvore de *BnB*. Então o *BnB* opera reduzindo o espaço de buscas de um *backtracking* normal, com isso efetivamente reduzindo o tempo de computação.

Na Seção 4 apresentamos o algoritmo de *BnB* que resolve o problema `Caminhada Máxima` que será descrito na seção a seguir.

3 O problema

O problema `Caminhada Máxima` é descrito da seguinte forma:

Instância: Grafo G com vértices $V(G) = \{1, 2, 3, 4, \ldots, n\}$ e pesos $w : E(G) \to \mathbb{R}_{>0}$.

Solução: Passeio fechado de peso máximo (somando o peso das arestas do passeio) que começa e termina em 1 e não repete outros vértices além do vértice 1.

`Caminhada Máxima` é uma abstração do problema de, dados um conjunto de pontos em um mapa e dados diferentes caminhos que ligam esses pontos, determinar qual o passeio fechado que não repete pontos e que consome o maior tempo para ser realizado.

Considere por exemplo a instância de `Caminhada Máxima` dada pela Figura 1 e a solução fornecida na Figura 2. Observe que o problema não exige que a solução seja um ciclo, de modo que o pesseio fechado $P = (1, 3, 1)$ também é um candidato, mas não é ótimo devido a somatória dos seus pesos não ser máxima.

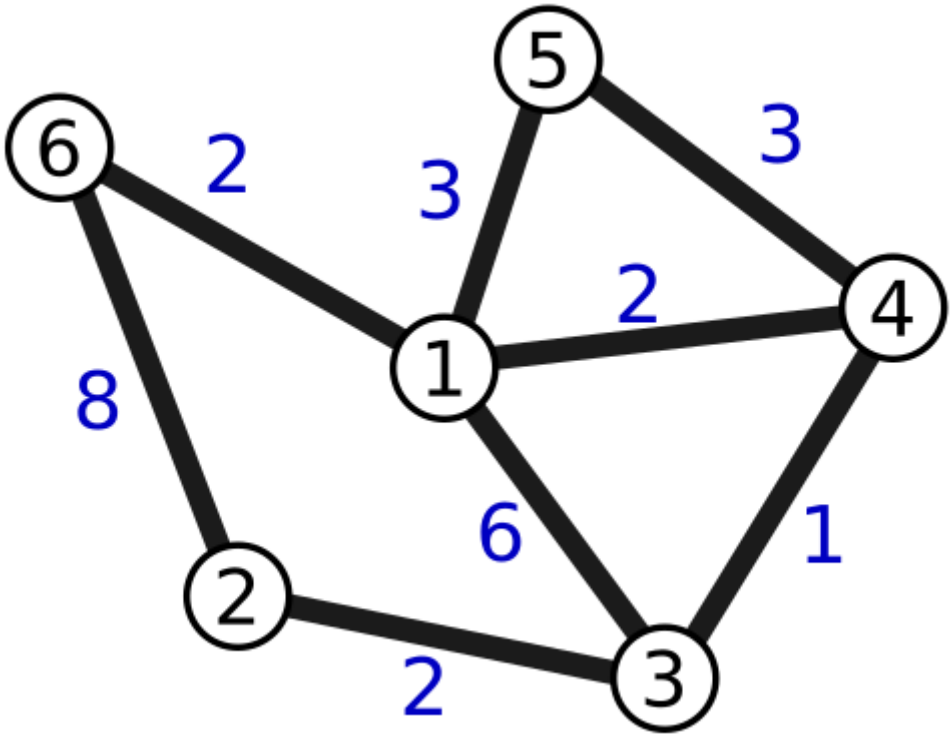

Figura 1: Exemplo de instância do problema de encontrar passeio fechado de peso máximo

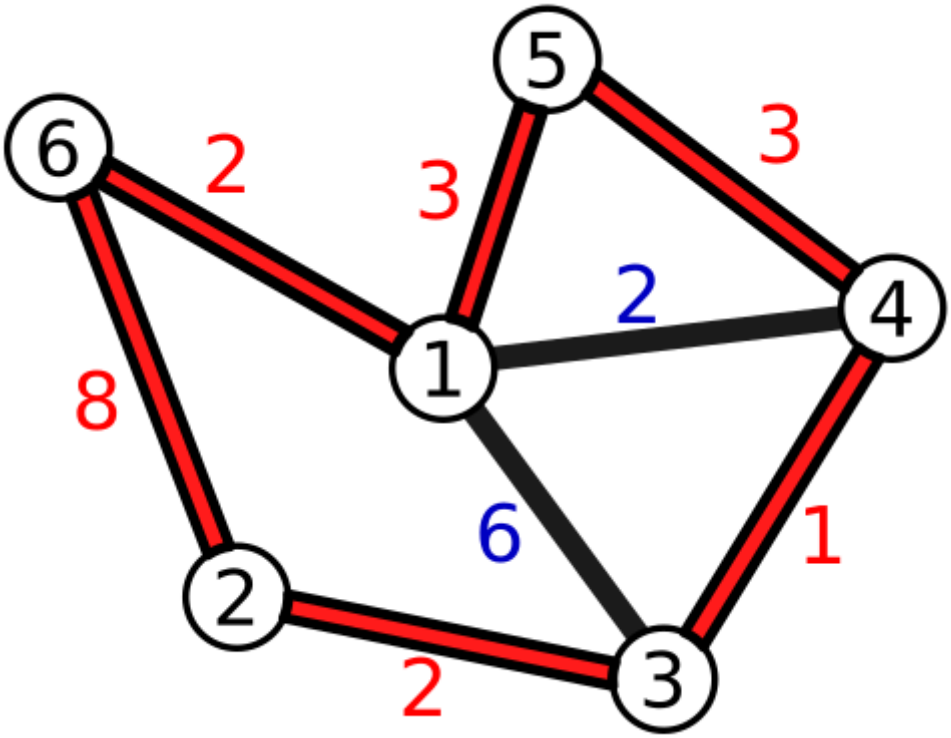

Figura 2: Solução da instância descrita na Figura 1

4 Algoritmo de *BnB*

O Algoritmo 3 apresenta o "envelope" do algoritmo proposto, chamado de `PasseioFechadoMax`. As chamadas recursivas são realizadas pela função `RecursivePasseioFechadoMax`. Observar que o algoritmo `RecursivePasseioFechadoMax` não retorna nenhum objeto, mas deve ser chamado dentro do escopo da função `PasseioFechadoMax`.

`PasseioFechadoMax`(G);
Entrada: $G = (\{1, 2, 3, \ldots, n\}, E(G)), w : E(G) \to \mathbb{R}_{>0}$.
Saída: Passeio fechado que começa em 1 e termina em 1, não repete outros vértices além de 1, e tem peso máximo.
$opt \leftarrow 0$;
$optP \leftarrow$ lista vazia;
`RecursivePasseioFechadoMax`((1,));
retornar opt e $optP$;

Algoritmo 3: Algoritmo de *BnB* para o problema da Seção 3

`RecursivePasseioFechadoMax`(P);
Entrada: P é um passeio não necessariamente fechado de G.
Dados: A função deve ser chamada no escopo de `PasseioFechadoMax`(G).
se P *é passeio fechado com extremidades* 1 **então**
    $c \leftarrow$ peso total do passeio P;
    se $c > opt$ **então**
        $opt \leftarrow c$;
        $optP \leftarrow P$;
    fim
fim
senão se P *não é passeio fechado* **então**
    $u \leftarrow$ último vértice do passeio P (extremidade oposta a 1);
    para $v \in N_G(u)$ **faça**
        se $v = 1$ *ou* $v \notin P$ **então**
            $B \leftarrow upper\_bound(P \cdot (u, v))$;
            se $B \geq opt$ **então**
                `RecursivePasseioFechado`($P \cdot (u, v)$);
            fim
        fim
    fim
fim

Algoritmo 4: Algoritmo recursivo `RecursivePasseioFechadoMax` usado no Algoritmo 3

5 Formulação da função de limitante superior

Como dito na Seção 4, a função limitante superior utiliza programação linear. Mais precisamente, modela-se um problema de programação linear inteira, e se considera o relaximento linear desse problema. Uma vez modelado o problema, a solução pode ser obtida eficientemente por meio de algoritmos consolidados da literatura de otimização linear, de modo que a função limitante superior proposta é fácil de calcular e provavelmente é "apertada" o bastante para cortar ramos que não tem o potencial para retornar boas soluções.

O problema de programação linear inteira considerado "simula" o problema de calcular um fluxo inteiro (discreto) unitário que sai do vértice 1 e deve retornar para o vértice 1; esse fluxo inteiro unitário deve maximizar a soma dos custos associados às arestas pelas quais o fluxo passa. Ocorre que modelar diretamente esse problema pode dar margem a *fluxos cíclicos indesejados* que *não* passam pelo vértice 1, por isso utilizamos o artifício de criar um vértice 0 com a mesma vizinhança que o vértice 1 e modelamos o problema de enviar um fluxo inteiro unitário do vértice 1 ao vértice 0. Observe que o vértice 1 e o vértice 0 são *virtualmente* o mesmo vértice na modelagem.

Nota: Estamos interessados no problema de determinar um fluxo **unitário**, pois consideramos que cada aresta tenha capacidade entre 0 a 1, e que uma aresta pela qual não passa fluxo algum não faz parte do passeio fechado, e que uma aresta pela qual passa fluxo 1 faz parte do passeio fechado.

Passamos a explicar o modelo.

Seja G um grafo com $V(G) = \{1, 2, 3, \ldots, n\}$ e com pesos $w : E(G) \to \mathbb{R}_{>0}$. Consideramos o grafo G' obtido a partir de G acrescentando o vértice 0 que seja adjacente a todo vizinho do vértice 1. Estenda w para

incluir os pesos das arestas incidentes em 0 da seguinte forma: o peso de uma aresta $\{0, j\}$ é igual ao peso da aresta $\{1, j\}$, para todo $j \in N_G(1)$. Denote por $w[i][j] = w[j][i]$ o peso $w(\{i, j\}), \forall \{i, j\} \in E(G')$. Para cada aresta $\{i, j\}$ considere as variáveis binárias $x[i][j] \in \{0, 1\}$ (que está para a aresta direcionada (i, j)) e $x[j][i] \in \{0, 1\}$ (que está para a aresta direcionada (j, i)). Consideramos que $x[i][j] = 1$ significa que existe um fluxo unitário de i para j e que a aresta (i, j) faz parte do caminho do fluxo solução.

A Equação (1) mostra a função objetivo do problema, que consiste em somar o peso de todas as arestas que fazem parte do fluxo. Observe que a restrição da Equação (7) faz com que ou $x[i][j] = 1$ ou $x[j][i] = 1$, mas não ambos. A restrição da Equação (2) faz com que o único sorvedouro do fluxo seja o vértice 0, e a restrição da Equação (3) faz com que 1 seja a única fonte do fluxo. Para todos os vértices de $V(G') - \{0, 1\}$, há no máximo uma unidade de fluxo saindo (Equação (5)) e no máximo uma unidade de fluxo entrando (Equação (4)), mas sempre com conservação de fluxo (Equação (6)). A Equação (8) descreve a restrição a variáveis binárias.

Como foi visto na disciplina de Otimização, o ótimo do relaxamento de um problema de programação linear inteira é um limitante superior do mesmo. Não há garantia porém que os valores das variáveis que atinjam o ótimo do problema relaxado sejam inteiros, mas isso não é uma dificuldade, visto que estamos interessados apenas em obter um limitante superior para o problema linear inteiro. Além disso, no problema relaxado pode ocorrer que a solução ótima consista de mais de um fluxo fracionário do vértice 1 para o vértice 0, mas isso também não deve comprometer a qualidade do limitante superior, visto que os fluxos fracionários também participam de modo fracionário na função objetivo.

$$\text{maximizar} \sum_{\{i,j\} \in E(G')} w[i][j](x[i][j] + x[j][i]) \tag{1}$$

$$\sum_{j \in N_G(0)} x[0][j] = 0 \quad \text{e} \quad \sum_{i \in N_G(0)} x[i][0] = 1 \tag{2}$$

$$\sum_{j \in N_G(1)} x[1][j] = 1 \quad \text{e} \quad \sum_{i \in N_G(1)} x[i][1] = 0 \tag{3}$$

$$\sum_{j \in N_G(i)} x[i][j] \leq 1, \forall i \in V(G') - \{0, 1\} \tag{4}$$

$$\sum_{i \in N_G(j)} x[i][j] \leq 1, \forall j \in V(G') - \{0, 1\} \tag{5}$$

$$\sum_{j \in N_G(u)} x[u][j] = \sum_{i \in N_G(u)} x[i][u], \forall u \in V(G') - \{0, 1\} \tag{6}$$

$$x[i][j] + x[j][i] \leq 1, \forall \{i, j\} \in E(G') \tag{7}$$

$$x[i][j], x[j][i] \in \{0, 1\}, \forall \{i, j\} \in E(G') \tag{8}$$

Agora considere que em um determinado ponto do funcionamento do algoritmo de *BnB* temos uma solução parcial $L = (1, v_2, v_3, \ldots, v_l)$ que está sendo avaliada. Obter um limitante superior para o ramo representado por L pode ser feito tomando o ótimo do relaxamento linear do problema de programação inteira descrito acima, porém acrescentado das restrições da Equação (9).

$$\begin{gathered} x[1][v_2] = 1 \\ x[v_2][v_3] = 1 \\ x[v_3][v_4] = 1 \\ \vdots \\ x[v_{l-1}][v_l] \end{gathered} \tag{9}$$

5.1 Benefício adicional do limitante superior

A função de limitante superior modelada na Seção 5 possui um benefício adicional. Ela serve ao propósito de cortar uma solução parcial (um nó da árvore de *BnB*) que não seja factível, isto é, do qual não se pode formar um passeio fechado que não repete vértices. De fato, no caso de uma determinada solução parcial $L = (1, v_2, v_3, \ldots, v_l)$ resultar em um problema de programação linear infactível (condição facilmente detectável com

uma implementação do "revised simplex"), então não existe possibilidade de existir um passeio fechado que estenda L e que não repita vértices.

Nota: a observação feita no parágrafo anterior é implementada da seguinte forma: caso a solução parcial L seja infactível, o limitante superior retornado para essa solução parcial é $-\text{inf}$, de modo que essa solução parcial sempre será cortada.

6 Implementação do algoritmo

A implementação do algoritmo de *BnB* proposto é realizada em linguagem *Python* (ver `https://www.python.org/`). Também são utilizadas bibliotecas específicas na linguagem *Python* para maior eficiência desta implementação, como a biblioteca `networkx` (ver `https://networkx.org/`; para lidar com grafos) e `scipy.optimize` (ver `https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html`; para lidar com otmização linear).

6.1 A implementação da função limitante superior

Nota: A implementação completa da função `upper_bound` (nome da função de limitante superior na implementação) não será apresentada, devido essa ter 290 linhas de código. São códigos simples que seguem um mesmo padrão repetidas vezes, como exemplificamos nesta subseção.

A implementação dos Algoritmos 3 e 4 foi realizada pela transcrição das instruções desses algoritmos em linguagem *Python*. Maior esforço foi necessário para a implementação da função `upper_bound` utilizada no Algoritmo 4. Como descrito na Seção 5, a função `upper_bound` se baseia na resolução de um problema de programação linear que foi feito na prática por meio da implementação `scipy.optimize.linprog`. Vejamos a seguir como realizar uma chamada para essa função.

A função `scipy.optimize.linprog` toma como argumentos:

- um vetor c de custos associados às variáveis do problema. O vetor c deve ter tamanho igual ao número de variáveis do problema de otimização. Observamos que a função `scipy.optimize.linprog` é implementada para *minimizar* a função objetivo, por isso passamos os pesos negativos nesse vetor;
- uma matriz retangular A_{le} e um vetor b_{le} tais que o vetor de variáveis x deva satisfazer $A_{le}x \leq b_{le}$, isto é, a matriz A_{le} e o vetor b_{le} deve codificar as restrições do tipo "menor ou igual" do problema de otimização; e
- uma matriz retangular A_{eq} e um vetor b_{eq} tais que o vetor de variáveis x deva satisfazer $A_{eq}x = b_{eq}$.

Pelo descrito, as variáveis na chamada à função `scipy.optimize.linprog` são passadas implicitamente. Essa informação está contida no tamanho dos vetores e matrizes $c, A_{le}, b_{le}, A_{eq}, b_{eq}$. Desta forma, a maior parte do código da implementação da função `upper_bound` consiste em formatar corretamente as linhas das matrizes A_{le}, A_{eq} e os valores dos vetores c, b_{le} e b_{eq}.

Vejamos como codificar corretamente uma restrição do modelo descrito na Seção 5 nas variáveis passadas à função `scipy.optimize.linprog`. Suponha que estejamos interessados em codificar a restrição da Equação (6) que expressa a conservação do fluxo que passa por um vértice $u \notin \{0, 1\}$. Como a conservação do fluxo é uma restrição de *igualdade*, devemos adicionar uma linha na matriz A_{eq} e adicionar uma entrada correspondente no vetor b_{eq}. Podemos escrever a equação da conservação do fluxo segundo a Equação (10). Formatamos a linha correspondente a essa restrição *do tipo igualdade* segundo o Código 1. Descrevemos a seguir cada uma das variáveis e instruções desse código:

- a linha 2 inicializa uma lista vazia a_{eq}. Essa lista será inserida, ao final no código, na matriz A_{eq}. Cada uma das entradas de a_{eq} representa o coeficiente que multiplica a variável correspondente;
- a linha 3 percorre a variável I. Essa variável é responsável por ordenar todos os arcos possíveis no grafo. Falaremos mais dessa variável adiante. Importa observar agora que quando percorremos linearmente os elementos de I, estamos considerando ordenadamente todas as variáveis do problema de otimização;

- a linha 4 indaga se estamos considerando um arco saindo de u. Caso a resposta seja afirmativa, a linha 5 adiciona o coeficiente 1 para esse arco na lista a_{eq};
- a linha 6 indaga se estamos considerando um arco entrando em u. Caso a resposta seja afirmativa, a linha 7 adiciona o coeficiente -1 para esse arco na lista a_{eq};
- a linha 8 indaga se estamos considerando um arco entrando ou saindo de u. Caso a resposta seja *negativa*, a linha 9 adiciona o coeficiente 0 para esse arco na lista a_{eq}; e
- por fim, as linhas 10 e 11 adicionam a_{eq} em A_{eq} e um coeficiente 0 correspondente em b_{eq}.

$$\sum_{j \in N_G(u)} x[u][j] - \sum_{i \in N_G(u)} x[i][u] = 0 \tag{10}$$

```
1  # Conservacao do fluxo
2  a_eq = []
3  for (i,j) in I:
4      if i == u:
5          a_eq.append(1)
6      elif j == u:
7          a_eq.append(-1)
8      else:
9          a_eq.append(0)
10 A_eq.append(a_eq)
11 b_eq.append(0)
```

Código 1: Codificação da restrição de conservação do fluxo

Como foi feito para a restrição de conservação de fluxo, assim é feito para todas as restrições do problema. Há no entanto mais um tipo de restrição que devemos implementar, que não faz parte do modelo de otimização linear, mas sim parte da implementação. Ocorre que para simplificar a implementação, convém considerar que as variáveis do problema são todos os pares do conjunto $\{(i,j) : i,j \in \{1,2,\ldots,n\}, i \neq j\}$, ainda que muitos pares (i,j) não sejam possíveis pela topologia do grafo G dado no problema. Por isso a definição da variável I é como dada pelo Código 2. Isso não cria nenhuma dificuldade, pois para os pares (i,j) que não são arcos factíveis, inserimos a restrição $x[i][j] = 0$ e para os pares (i,j) que são factíveis adicionamos a restrição $0 \leq x[i][j] \leq 1$ (ver Código 3, onde a variável `edges` representa os arcos factíveis).

```
1  I1 = [ (i,j) for i in range(0,n) for j in range(i+1,n+1) ]
2  I2 = [ (j,i) for i in range(0,n) for j in range(i+1,n+1) ]
3  I = I1 + I2
```

Código 2: Implementação das variáveis do problema de otimização

```
1  # edge representa os arcos factíveis
2
3  for (i,j) in I:
4      if ((i,j) in edges) or ((j,i) in edges):
5
6          # se o par (i,j) for arco factivel
7          a_le = []
8          for e in I:
9              if e == (i,j):
10                 a_le.append(1)
11             elif e != (i,j):
12                 a_le.append(0)
13         A_le.append(a_le)
14         b_le.append(1)
15
16     else:
17         # se o par (i,j) nao for arco factivel
18         a_eq = []
19         for e in I:
```

```
20              if e == (i,j):
21                  a_eq.append(1)
22              elif e != (i,j):
23                  a_eq.append(0)
24          A_eq.append(a_eq)
25          b_eq.append(0)
```

Código 3: Implementando a restrição adicional para o caso de um arco ser infactível

6.2 Implementação do *BnB*

A implementação do algoritmo de *BnB* (sem ordenamento) é descrita no Código 4. Observe que ele está implementado dentro de uma única função chamada `branch_and_bound_max_cycle` que toma de entrada um grafo G com $V(G) = \{1, 2, 3, \ldots, n\}$ e imprime na saída padrão o ótimo do problema `Caminhada Máxima` para a instância G e o passeio fechado que garante esse ótimo. Isso se mostrou conveniente do ponto de vista de implementação (implementar tudo dentro de uma função).

Vejamos algumas observações acerca do funcionamento da função `branch_and_bound_max_cycle`:

- as linhas $9, 10$ inicializam os contadores de nós visitados e de nós cortados;
- as linhas $12, 13$ inicializam as variáveis que guardam o ótimo e o passeio fechado que atinge esse ótimo;
- as linhas 15-21 calculam o grafo auxiliar `G0` que será utilizado pela função `upper_bound` (linha 23). O grafo `G0` corresponde ao grafo G com a adição do vértice 0;
- as linhas 23-32 mostram a definição da função de limitante superior `upper_bound` (a definição é omitida, ver Subseção 6.1);
- a linha 34 inicia a definição da função recursiva `recursive_branch_and_bound_max_cycle`;
- as linhas 36-41 declaram as variáveis *não locais*, isto é, faz referência às variáveis definidas fora da função recursiva, mas dentro da função `branch_and_bound_max_cycle`;
- a linha 43 pergunta se estamos diante de um passeio fechado;
- caso a resposta da pergunta da linha 43 seja verdadeira, as linhas 45-55 atualizam o ótimo e o passeio fechado ótimo;
- a linha 58 indaga se não estamos diante de um passeio fechado. Nesse caso devemos fazer outra chamada recursiva de `recursive_branch_and_bound_max_cycle`;
- a linha 61 define uma variável para o *último* vértice do passeio L;
- a linha 62 calcula todos os vizinhos do último vértice do passeio L;
- as linhas $64, 65$ iteram sobre todos os vértices para os quais faz sentido considerar um prolongamento de L;
- a linha 67 atualiza o número de nós visitados;
- a linha 69 calcula o limitante superior do passeio considerado (uma das extensões de L);
- as linhas $71, 73$ decidem se há condições de realizar um corte ou não; caso não seja possível, a linha 72 faz uma chamada recursiva.
- o restante das linhas se ocupam de gerar um relatório e imprimir a saída da função, ou seja, o ótimo e o passeio fechado ótimo.

```
def branch_and_bound_max_cycle(G):
    """
    Recebe um grafo da classe nx.Graph
    com pesos identificados pela palavra 'weight'
    e devolve o ciclo maximo, o otimo obtido,
    o numero de nos da arvore e o tempo total de execucao.
    """

    contador_de_nos = 1
    contador_nos_cortados = 0

    opt_cycle = []
    opt       = 0

    # Criando o grafo auxiliar G0
    G0 = nx.Graph()
    for (i,j) in G.edges():
        G0.add_edge(i,j,weight=G[i][j]["weight"])
    ## Adicionando vertice 0 em G0
    for j in [v for v in G[1]]:
        G0.add_edge(0,j,weight=G[1][j]["weight"])

    def upper_bound(L):
        """
        Retorna um limitante superior para o problema
        do ciclo de tamanho máximo.

        Usa a estrategia de duplicacao do no 1 para ser fonte,
        e insere o no 0 para ser sorvedouro.
        """

        <codigo omitido>

    def recursive_branch_and_bound_max_cycle(L):

        nonlocal G
        nonlocal opt_cycle
        nonlocal opt

        nonlocal contador_de_nos
        nonlocal contador_nos_cortados

        answer = is_cycle(L)
        #print(L,answer)
        if answer == True:
            total_weight = 0.0
            for i in range(len(L)-1):
                total_weight += G[L[i]][L[i+1]]["weight"]

            #print(total_weight)

            if total_weight > opt:

                opt = total_weight
                opt_cycle = L
                return

        elif answer == False:

            start = L[0]
            end = L[-1]
            neighbors = [v for v in G[end]]

            for v in neighbors:
```

```
65                 if v == start or v not in L:
66
67                     contador_de_nos += 1
68
69                     B = upper_bound(L + [v])
70
71                     if B >= opt:
72                         recursive_branch_and_bound_max_cycle(L + [v])
73                     elif B < opt:
74                         contador_nos_cortados += 1
75
76     start = time.time()
77
78     recursive_branch_and_bound_max_cycle([1])
79
80     end = time.time()
81
82     print("Relatorio da execucao do algoritmo de branch-and-bound",file=sys.stderr)
83     print("Otimo encontrado eh {}".format(opt),file=sys.stderr)
84     print("Ciclo otimo encontrado eh {}".format(opt_cycle),file=sys.stderr)
85     print("Numero total de nos visitados: {}".format(contador_de_nos),file=sys.stderr)
86     print("Numero total de nos cortados: {}".format(contador_nos_cortados),file=sys.stderr
       )
87     print("Tempo total de execucao: {}".format(end-start),file=sys.stderr)
88
89     print(opt)
90     cycle_str = '1'
91     for v in opt_cycle[1:]:
92         cycle_str += (' ' + str(v))
93     print(cycle_str)
```

Código 4: Implementação do *BnB*

6.3 Implementação do *ordered BnB*

A implementação do *ordered BnB* é em tudo semelhante à implementação do *BnB* apresentada na Subseção 6.2, exceto pelo ordenamento das chamadas recursivas de acordo com os maiores valores de limitante superiores da iteração seguinte. O Código 5 apresenta a implementação em *Python* do *ordered BnB*. Veja que as únicas linhas que distinguem essa implementação da anterior são as linhas 66-78. Vejamos as instruções dessas linhas:

- a linha 66 inicializa uma lista vazia;
- as linhas 67, 68 calculam todas as chamadas recursivas possíveis;
- a linha 69 conta os novos nós visitados na árvore de *BnB*;
- a linha 70 calcula o limitante superior para cada nó que será visitado;
- a linha 72 ordena os nós em ordem decrescente (do maior para o menor) de limitantes superiores obtidos;
- as linhas 74-78 percorrem os nós que serão visitados de acordo com a ordem calculada na linha 72.

```
1  def ordered_branch_and_bound_max_cycle(G):
2      """
3      Recebe um grafo da classe nx.Graph
4      com pesos identificados pela palavra 'weight'
5      e devolve o ciclo maximo, o otimo obtido,
6      o numero de nos da arvore e o tempo total de execucao.
7      """
8
9      contador_de_nos = 1
10     contador_nos_cortados = 0
```

```
    opt_cycle = []
    opt       = 0

    # Criando o grafo auxiliar G0
    G0 = nx.Graph()
    for (i,j) in G.edges():
        G0.add_edge(i,j,weight=G[i][j]["weight"])
    ## Adicionando vertice 0 em G0
    for j in [v for v in G[1]]:
        G0.add_edge(0,j,weight=G[1][j]["weight"])

    def upper_bound(L):
        """
        Retorna um limitante superior para o problema
        do ciclo de tamanho máximo.

        Usa a estrategia de duplicacao do no 1 para ser fonte,
        e insere o no 0 para ser sorvedouro.
        """

        <codigo omitido>

    def recursive_ordered_branch_and_bound_max_cycle(L):

        nonlocal G
        nonlocal opt_cycle
        nonlocal opt

        nonlocal contador_de_nos
        nonlocal contador_nos_cortados

        answer = is_cycle(L)
        #print(L,answer)
        if answer == True:
            total_weight = 0.0
            for i in range(len(L)-1):
                total_weight += G[L[i]][L[i+1]]["weight"]

            #print(total_weight)

            if total_weight > opt:
                opt = total_weight
                opt_cycle = L
                return

        elif answer == "Infeseable":
            return

        elif answer == False:

            start = L[0]
            end = L[-1]
            neighbors = [v for v in G[end]]

            bounds = []
            for v in neighbors:
                if v == start or v not in L:
                    contador_de_nos += 1
                    bounds.append((v,upper_bound(L+[v])))

            bounds = sorted(bounds, key = lambda x : x[1], reverse = True)

            for (v,B) in bounds:
```

```
                if B >= opt:
                    recursive_ordered_branch_and_bound_max_cycle(L + [v])
                elif B < opt:
                    contador_nos_cortados += 1
                    #print("Cut branch {}".format(L+[v]))

    start = time.time()

    recursive_ordered_branch_and_bound_max_cycle([1])

    end = time.time()

    print("Relatorio da execucao do algoritmo de ordered branch-and-bound",file=sys.stderr
    )
    print("Otimo encontrado eh {}".format(opt),file=sys.stderr)
    print("Ciclo otimo encontrado eh {}".format(opt_cycle),file=sys.stderr)
    print("Numero total de nos visitados: {}".format(contador_de_nos),file=sys.stderr)
    print("Numero total de nos cortados: {}".format(contador_nos_cortados),file=sys.stderr
    )
    print("Tempo total de execucao: {}".format(end-start),file=sys.stderr)

    print(opt)
    cycle_str = '1'
    for v in opt_cycle[1:]:
        cycle_str += (' ' + str(v))
    print(cycle_str)
```

Código 5: Implementação do *ordered BnB*

7 Comparação dos resultados

Essa seção apresenta uma comparação entre três algoritmos: um *backtracking* simples (Seção 1), *BnB* (Seção 6.2), e *ordered BnB* (Seção 6.3). Três métricas são utilizadas para fins de comparação: tempo total de execução, número de nós visitados na árvore de soluções, e número de nós cortados na árvore de soluções.

Para fins de comparação de resultados, utilizamos grafos aleatórios como instâncias. O algoritmo que gera o grafo aleatório está descrito no Algoritmo 5.

Nota: Os grafos aleatórios utilizados foram utilizados usando o Algoritmo 5 com parâmetros $p = 0{,}80$ e $w = 100$. Somente o parâmetro n foi modificado para obter grafos cada vez maiores.

`GrafoAleatorio`(n, p, w);
Entrada: Número n de vértices, probabilidade de selecionar uma aresta p, e um inteiro w que representa o peso máximo possível.
$G \leftarrow$ grafo vazio;
para $i = 1, 2, \ldots, n-1$ **faça**
  para $j = i+1, 2, \ldots, n$ **faça**
    $r \leftarrow$ número aleatório entre $0{,}0$ e $1{,}0$;
    se $r < p$ **então**
      Adicionar a aresta $\{i, j\}$ no grafo G;
      $q \leftarrow$ Sortear um inteiro entre $0{,}0$ e w;
      Adicionar peso q na aresta $\{i, j\}$;
    fim
  fim
fim
Retorna G;

Algoritmo 5: Algoritmo que gera o grafo aleatório

A Tabela 1 apresenta os resultados obtidos para os experimentos com grafos aleatórios com 8 vértices (3 instâncias de grafos aleatórios foram usadas). A Tabela 2 apresenta os resultados obtidos para os experimentos

com grafos aleatórios com 10 vértices (3 instâncias de grafos aleatórios foram usadas). Algumas observações podem ser feitas observando esses resultados:

- Os experimentos mostram que o tempo de execução do *backtracking* é proporcional ao número de nós visitado, como era de se esperar;
- Para 8 e 10 vértices, fica claro que a estratégia de *backtracking* normal tem menor tempo de execução do que as estragégias de *BnB*, ordenado ou não;
- O número de nós visitados na estratégia de *backtracking* é relativamente muito superior ao número de nós visitados nas estratégias de BnB, com ou sem ordenamento, mas em números absolutos não é grande o bastante para tornar a estratégia de *backtracking* inviável;
- O número de nós cortados na estratégia de *BnB* sem ordenamento é maior do que o número de nós cortados no *ordered BnB*, mas isso não é uma vantagem real, pois como há menor número de nós visitados com *ordered BnB*, a conclusão é que os cortes são feitos precocemente na estratégia *ordered BnB*, diminuindo assim o tempo de computação;
- Observar que o tempo de execução do *backtracking* aumenta significativamente quando passamos de 8 vértices para 10 vertices, resultado do aumento exponencial de passeios fechados possíveis;
- Quando consideramos grafos com 10 vértices, o número de nós visitados passa de 1.000.000 com o *backtracking*, mas o número de nós visitados nas estratégias *BnB* não passa de 1.000, o que é evidência do elevado potencial de corte da função limitante superior formulada;
- Com grafos aleatórios de 10 vértices já se torna percepível o menor tempo de computação do *ordered BnB* e o seu menor número de nós visitados;
- vemos que o tempo de computação do *backtracking* tradicional é muito menor que o dos algoritmos *BnB* para 8 vértices; para 10 vértices o tempo de execução do *backtracking* e do *BnB* sem ordenamento são da mesma ordem de grandeza; para 11 vértices (ver Tabela 3) o tempo de execução do *backtracking* é *superior* ao tempo de execução dos algoritmos de *BnB*; já para 12 vértices (ver Tabela 4) tanto o tempo de execução quando o número de nós visitados do *backtracking* é de maior ordem de grandeza do que as mesmas métricas para os algoritmos *BnB*.

	Backtracking	*BnB*	*ordered BnB*
Tempo (s)	0,42	14,45	8,75
Nós visitados	8.578	126	55
Nós cortados	Não se aplica	70	38
Tempo (s)	0,66	16.87	9,50
Nós visitados	11.745	129	79
Nós cortados	Não se aplica	80	54
Tempo (s)	0,65	12,90	7,55
Nós visitados	16.311	136	81
Nós cortados	Não se aplica	82	56

Tabela 1: Resultados para grafos aleatórios com 8 vértices

	Backtracking	*BnB*	*ordered BnB*
Tempo (s)	68,11	146,41	25,09
Nós visitados	1.054.811	570	93
Nós cortados	Não se aplica	392	72
Tempo (s)	76,15	78,79	36,18
Nós visitados	1.330.871	350	157
Nós cortados	Nâo se aplica	245	122
Tempo (s)	74,12	98,13	19,55
Nós visitados	1.252.583	479	96
Nós cortados	Não se aplica	335	75

Tabela 2: Resultados para grafos aleatórios com 10 vértices

	Backtracking	*BnB*	*ordered BnB*
Tempo (s)	393,12	266,42	63,02
Nós visitados	5.237.172	1.019	207
Nós cortados	Não se aplica	722	152
Tempo (s)	438,88	118,63	82,10
Nós visitados	7.572.817	460	273
Nós cortados	Não se aplica	328	210
Tempo (s)	148,39	177,61	47,02
Nós visitados	2.485.882	682	187
Nós cortados	Não se aplica	467	132

Tabela 3: Resultados para grafos aleatórios com 11 vértices

Nota: foram realizados experimentos com o algoritmo de *backtracking* somente para grafos com até 12 vértices, porque o tempo de processamento nesse caso é de aproximadamente 1 hora. Como o crescimento do tempo é exponencial em relação ao número de vértices, julguei desnecessário fazer novos experimentos com *backtracking* considerando que o tempo necessário seria impraticável para eu relatar a partir desse ponto.

	Backtracking	*BnB*	*ordered BnB*
Tempo (s)	4.552,95	452,80	88,69
Nós visitados	78.157.347	1.036	230
Nós cortados	Não se aplica	751	182
Tempo (s)	3.305,55	310,84	50,44
Nós visitados	38.748.040	700	125
Nós cortados	Não se aplica	492	100
Tempo (s)	10.448,51	432,83	56,92
Nós visitados	157.168.011	1.077	139
Nós cortados	Não se aplica	769	114

Tabela 4: Resultados para grafos aleatórios com 12 vértices

As observações feitas em relação aos resultados obtidos para os algoritmos de *BnB* e *ordered BnB* são feitas abaixo e dizem respeito aos resultados das Tabelas 5 (experimentos com $n = 14$) e 6 (experimentos com $n = 16$).

- a versão *ordered BnB* possui menor tempo de execução do que a versão *BnB* (sem ordenamento) em todos os casos testados;
- a versão *ordered BnB* visita menos nós da árvore que a versão *BnB* (sem ordenamento) em todos os casos testados;

- os dados sugerem que os cortes da versão *ordered BnB* são realizados precocemente em relação aos cortes da versão *BnB*, sugerindo que o ótimo do problema seja encontrado na versão *ordered BnB* mais rapidamente;
- o número de nós cortados é relativamente alto comparado ao número de nós visitados, o que sugere adequação da função de limitante superior formulado na Seção 5.

	BnB	*ordered BnB*
Tempo (s)	1.796,38	338,50
Nós visitados	1.834	319
Nós cortados	1.388	268
Tempo (s)	1.985,73	162,38
Nós visitados	2.038	182
Nós cortados	1.569	153
Tempo (s)	2.585,14	182,16
Nós visitados	2.337	181
Nós cortados	1.832	152

Tabela 5: Resultados para grafos aleatórios com 14 vértices

	BnB	*ordered BnB*
Tempo (s)	5.514,03	798,00
Nós visitados	3.268	487
Nós cortados	2.660	409
Tempo (s)	9.135,49	189,24
Nós visitados	5.188	124
Nós cortados	4.186	107
Tempo (s)	6.220,12	1.690,28
Nós visitados	3.676	1.101
Nós cortados	2.895	942

Tabela 6: Resultados para grafos aleatórios com 16 vértices

8 Como chamar o executável `caminhada`

ATENÇÃO Uma vez que os resultados indicam que o *ordered BnB* é o melhor das versões de *BnB*, é essa a versão utilizada de fato pelo executável `caminhada`.

ATENÇÃO Antes de realizar uma chamada ao executável `caminhada`, é preciso verificar que tipo de sistema operacional está sendo usado. Originalmente, o arquivo `caminhada` foi escrito em um sistema operacional windows.

ATENÇÃO Para que o programa `caminhada` funcione corretamente, é **necessário** utilizar uma versão de *Python* superior a 3, isso porque a diretiva `nonlocal` utilizada não tem suporte em *Python* 2.

8.1 Sistema operacional linux

Caso o sistema operacional seja linux, antes de tudo, execute o comando `make`. São duas as instruções do arquivo `Makefile`: 1) converte os caracteres de "nova linha" do arquivo `caminhada` para padrão unix; e 2) dá permissão de executável ao arquivo `caminhada`.

```
1 all: caminhada
2         dos2unix caminhada
3         chmod +x caminhada
```

Código 6: Instruções do arquivo `Makefile`

Uma vez que o comando `make` foi chamado, pode-se executar `caminhada` por meio de um pipe, redirecionando por meio da entrada padrão o arquivo que descreve a instância do problema formulado (ver Código 7).

```
# Redirecionando com um pipe para o executavel caminhada (como script)
cat <arquivo de instancia do problema> | ./caminhada

# Redirecionando com um pipe para caminhada (chamando explicitamente python)
cat <arquivo de instancia do problema> | python3 caminhada

# Entrando com a instancia manualmente (encerra com Ctrl+D)
python3 caminhada
# ou
./caminhada
```

Código 7: Modos de chamar o executável `caminhada` no linux

8.2 Sistema operacional Windows

Para o sistema operacional windows, supomos que o executável `python.exe` esteja instalado e que o sistema operacional saiba localizar esse executável no computador. **Não é preciso** executar make nesse caso. **ATENÇÃO:** os comandos foram testados utilizando o ambiente de programação `Spyder`, e não um simples terminal cmdos. Esse ambiente de programação sabe localizar corretamente as bibliotecas científicas utilizadas. Os comandos do Código 8 mostram como realizar uma chamada a `caminhada`.

```
# Redirecionando com um pipe para caminhada (chamando explicitamente python)
cat <arquivo de instancia do problema> | python caminhada

# Entrando com a instancia manualmente (encerra com Ctrl+Z)
python caminhada
```

Código 8: Modos de chamar o executável `caminhada` no Windows