FGV EMAp

Escrita: Thalis Ambrosim Falqueto

Projeto e Análise de Algoritmos

Exercícios de Slides

Rio de Janeiro

2025

Conteúdo

Técnicas de Projeto

1.1 Maior subsequência de Strings

Dadas duas strings, encontre o comprimento da maior subsequência comum entre elas.

Essa solução usa o paradigma da Programação Dinâmica, onde usaremos uma matriz para guardar os valores de cada sub string. Por exemplo, a matriz no índice ixj será o tamanho da maior subsequênciade string dado que nossas substrings são $string1$ `[:i]` e $string2$`[:j]`.

Após construir a matriz, passamos completando cada elemento. Caso as letras sejam iguais, nós atualizamos a tabela somando o valor de um e o valor das substrings passadas, quando não tínhamos nenhuma das duas letras comparadas. Caso contrário, pegamos o maior entre $string1$ `[:i-1]` e $string2$`[:j]`

1.1.1 explicar melhor os casos e pq funciona

```py
def string_problem(string1, string2):
    str1 = list(string1)
    str2 = list(string2)
    len_str1 = len(str1)
    len_str2 = len(str2)
    M = [[0] * (len_str1 + 1) for _ in range(len_str2 + 1)]

    for j in range (1, len_str2 + 1):
        for i in range (1, len_str1 + 1):
            if str1[i - 1] == str2[j - 1]:
                M[j][i] = 1 + M[j-1][i-1]
            else:
                M[j][i] = max(M[j][i - 1], M[j - 1][i])

    return M[len_str2][len_str1]
```

1.2 Menor quantidade de moedas

Dado um valor v e uma lista de denominações de moedas (de um sistema canônico), encontre o número de moedas para formar v.

Nesse problema, vamos usar o paradigma Guloso. Considerando a lista de moedas **ordenada**, declaramos duas váriaveis, e para cada moeda da lista, se for `0` (significa que nossa soma de moedas chegou no valor que queríamos), acabamos o código. Caso contrário, pegamos a divisão inteira, que no caso significa quantas vezes cada moeda consegue fazer parte do valor, subtraímos do que falta em e incrementamos a contagem.

```py
def coin_problem(v, coins):
    counting_coins = 0
    v_fake = v

    for coin in coins:
        if v_fake == 0:
            break
```

```
        qtd = v_fake // coin
        if qtd > 0:
            v_fake -= qtd * coin
            counting_coins += qtd

    return counting_coins
```

1.3 Menor quantidade de comparações

Dado um array A com n elementos, encontre simultaneamente o maior e o menor elemento usando o menor número possível de comparações.

Uma solução genérica, mas que não atende a menor quantidade de comparações possível é se tivermos duas variáveis, uma para o mínimo e outra para o máximo, e passar pela lista em apenas um for. Infelizmente, a cada elemento o algoritmo deve verificar se o número é o menor ou o maior dentre o valor das variáveis anteriores. Desde que isso traz duas verificações por elementos, ao fim temos $2n$ comparações. Mas essa não é a solução ótima.

1.3.1 Solução (recursiva):

Essa solução, assim como a outra, depende de uma sacada interessante, em que queremos reduzir a quantidade de comparações $(2n)$ do algoritmo ingênuo. Para isso, usaremos uma estratégia de comparação entre **pares**, utilizando o paradigma Dividir e Conquistar. Tratamos os dois casos base na função e dividimos a lista ao meio para fazer isso recursivamente, até que tenhamos apenas 1 ou 2 elementos a serem analisad os, e isso nos traz 3 verificações por elemento (uma para o par `[left] < arr[right]`), duas para comparar com os mínimos globais(uma pro máximo, outra pro mínimo), trazendo $\approx 3\frac{n}{2}$ comparações.

```py
def max_min(arr, left, right):
    if left == right:
        return arr[left], arr[right]
    if left == right - 1:
        if arr[left] < arr[right]:
            return arr[left], arr[right]
        else:
            return arr[right], arr[left]

    mid = (left + right) // 2
    min1, max1 = max_min(arr, left, mid)
    min2, max2 = max_min(arr, mid+1, right)

    min_global = min(min1, min2)
    max_global = max(max1, max2)

    return (min_global, max_global)
```

1.3.2 Solução (pairwise):

Essa solução é interativa, e usa uma ideia parecida com o algoritmo anterior. Foi separada em duas funções para evitar duplicação de código. Declaramos algumas variáveis para tracking dos mínimos e máximos, e vemos se o tamanho da lista é par ou ímpar. Se for par, então teremos uma

quantidade de pares sem nenhuma sobra, e assim executamos `pairwise`, que passa na lista de elementos de par em par e verifica o maior (e menor) entre a dupla (uma verificação) e atualiza o máximo e mínimo global (2 verificações).

Para o caso de uma lista de tamanho ímpar, apenas é feita a mesma verificação para o último elemento ainda não verificado.

```py
def pairwise(arr, max_local, min_local, max_global, min_global):
    for i in range(0, len(arr) - 1, 2):
        if arr[i] >= arr[i+1]:
            max_local = arr[i]
            min_local = arr[i + 1]
        else:
            max_local = arr[i + 1]
            min_local = arr[i]

        if max_local > max_global:
            max_global = max_local
        if min_local < min_global:
            min_global = min_local

    return min_global, max_global

def comparation_problem(arr):
    max_global = -float('inf')
    min_global = float('inf')
    max_local = 0
    min_local = 0

    if len(arr) % 2 == 0:
        min_global, max_global = pairwise(arr, max_local, min_local, max_global, min_global)
    else:
        min_global, max_global = pairwise(arr, max_local, min_local, max_global, min_global)

        k = len(arr) - 1
        max_local = arr[k]
        min_local = arr[k]

        if max_local > max_global:
            max_global = max_local
        if min_local < min_global:
            min_global = min_local

    return (min_global, max_global)
```

Grafos

2.1 Matriz de adjacência

Implemente uma classe para representar um grafo utilizando matriz de adjacência.

2.1.1 C++

Em breve

2.1.2 Python

Nada de difícil entendimento aqui, portanto não precisa ser explicado. É apenas a implementação em Python do código dado nos slides de PAA do mesmo exercício em C++.

```py
class GraphMatrix:

    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.matrix = [[0 for _ in range(num_vertices)] for i in
        range(num_vertices)]

    def has_edge(self, v1, v2):
        if 0 <= v1 < self.num_vertices and 0 <= v2 < self.num_vertices:
            return self.matrix[v1][v2]
        return False

    def add_edge(self, v1, v2):
        if 0 <= v1 < self.num_vertices and 0 <= v2 < self.num_vertices:
            self.matrix[v1][v2] = 1
            self.matrix[v2][v1] = 1
        else:
            print("Erro")

    def remove_edge(self, v1, v2):
        if 0 <= v1 < self.num_vertices and 0 <= v2 < self.num_vertices:
            self.matrix[v1][v2] = 0
            self.matrix[v2][v1] = 0
        else:
            print("Erro")

    def print(self):
        for v1 in range(self.num_vertices):
            list_adj = []
            for v2 in range(self.num_vertices):
                if self.has_edge(v1, v2):
                    list_adj.append(f"({v1},{v2})")
            print(list_adj)

    def print_matrix(self):
```

```
        for v1 in range(self.num_vertices):
            row = []
            for v2 in range(self.num_vertices):
                row.append(self.matrix[v1][v2])
            print(row)
```

2.2 Lista de adjacências

Implemente uma classe para representar um grafo utilizando lista de adjacências.

2.2.1 C++

Em breve

2.2.2 Python

Considerando que os vértices são sempre sequências de inteiros de 0 a $n-1$, podemos fazer apenas uma lista de listas em vez de usar ponteiros em Python. Caso não fosse, poderíamos usar uma hashtable de listas, ou algo semelhante.

```py
class GraphAdjList:

    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.listadj = [[] for _ in range(num_vertices)]

    def has_edge(self, v1, v2):
        for i in range(len(self.listadj[v1])):
            if v2 in self.listadj[v1]:
                return True
        return False

    def add_edge(self, v1, v2):
        self.listadj[v1].append(v2)
        self.listadj[v2].append(v1)

    def remove_edge(self, v1, v2):
        self.listadj[v1].remove(v2)
        self.listadj[v2].remove(v1)

    def print_listadj(self):
        for vertex in range(self.num_vertices):
            print(f'{vertex}: {self.listadj[vertex]}')

    def print_matrix(self):
        matrix = [[0 for _ in range(self.num_vertices)] for i in
        range(self.num_vertices)]
        for vertex in range(self.num_vertices):
            for edge in self.listadj[vertex]:
                matrix[vertex][edge] = 1
```

```
30              print(matrix[vertex])
```

2.3 Verificação de subgrafo

Dados dois grafos $G = (V, E)$ e $H = (V', E')$ com $V = V'$, crie um algoritmo que verifica se H é subgrafo de G.

Nesse problema, $V = V'$, então é possível usarmos matriz de adjacência tranquilamente (lista de adjacência também). Sabendo disso, vamos fazer para os dois casos:

2.3.1 Matriz de adjacência

Para a matriz, basta apenas passar por cada elemento das matrizes e verificarmos se, quando em H é 1, G é 0, pois isso significaria que existe alguma aresta fora do grafo original.

```py
def is_subgraph_matrix(gmatrix, hmatrix):
    num_vertices = len(gmatrix)
    for row in range(num_vertices):
        for column in range(num_vertices):
            if hmatrix[row][column] == 1 and gmatrix[row][column] == 0:
                return False
    return True
```

2.3.2 Lista de adjacência

Para a lista, basta apenas passarmos por cada elemento de cada lista de vértice, e ver so vértice está na lista do grafo original.

```py
def is_subgraph_list(glist,hlist):
    num_vertices = len(glist)
    for vi in range(num_vertices):
        for vj in hlist[vi]:
            if vj not in glist[vi]:
                return False
    return True
```

Busca em Grafos

3.1 Verificação de caminho e caminho simples

Dado um grafo $G = (V, E)$ e um caminho P composto por uma sequência de vértices, verifique se P é um caminho de G, e se o caminho é simples.

3.1.1 Matriz de adjacência

Basta passar a matriz e a cada v_i e v_{i+1} verificar se é 1 na matriz. Eu decidi verificar se é um caminho simples em uma função separada, mas o leitor pode fazer junto se quiser. (a função serve tanto para lista quanto para matriz, por isso não irei repeti-la). É só olhar o tamanho da lista de caminhos se for igual quanto a transformamos em conjunto.

```py
def is_path_matrix(matrix, path):
    for order in range(len(path) - 1):
        if matrix[path[order]][path[order + 1]] != 1:
            return False
    return True

def is_simple_path(path):
    if len(set(path)) != len(path):
        return False
    return True
```

3.1.2 Lista de adjacência

Basta passar a lista e a cada v_i e v_{i+1} verificar se existe o vértice na lista de arestas de v_i

```py

def is_path_list(list, path):
    for order in range(len(path) - 1):
        if path[order + 1] not in list[path[order]]:
            return False
    return True
```

3.2 Verificação de númeração topológica

Crie um algoritmo que verifica se a numeração dos vértices de um grafo $G = (V, E)$ é topológica.

3.2.1 Matriz de adjacência

Trivialmente, basta verificar se cada $i \geq j$(evitando laços). Como a matriz não é simétrica, não podemos ignorar metade da matriz.

```py
def is_topological_matrix(matrix):
    num_vertices = len(matrix)
    for i in range (num_vertices):
        for j in range (num_vertices):
            if matrix[i][j] == 1 and i >= j:
                return False
    return True
```

3.2.2 Lista de adjacência

Análogo, só que para lista :D

```py
def is_topological_list(list):
    num_vertices = len(list)
    for i in range (num_vertices):
        for j in range(len(list[i])):
            if i >= list[i][j]:
                return False
    return True
```

3.3 Verificação de ordenação topológica (e determinação)

Crie um algoritmo para determinar se um grafo possui ordenação topológica e determiná-la.

3.3.1 Versão Slow (lista de adjacência)

Nessa versão, usamos a lista de ordem, counter e o número de vértices novamente. Então enquanto não preenchermos a lista de ordem corretamente (ou seja, `counter < V`), tentamos achar algum vértice com característica que nos ajudará a identificar a topologia do grafo, ou seja, se o grau de entrada do vértice é 0 (indício de fonte) e o vértice ainda não foi colocado na ordem.

Se nessa procura não acharmos esse vértice, então não temos essa ordenação topológica, e retornamos False. Se isso não ocorreu, então significa que o for parou exatamente no índice do vértice que satisfaz essas condições. Portanto marcamos ele na lista de ordem. Incrementamos o counter, e, por fim, decrementamos dos graus de saída dos vértices ligados ao vértice de fonte selecionado i, simulando a remoção do vértice.

```py
def in_degree(list_adj):
    V = len(list_adj)
    in_d = [0] * V
    for v1 in list_adj:
        for v2 in v1:
            in_d[v2] += 1
    return in_d

def has_topologic_order(list_adj):
    num_vertices = len(list_adj)
    order = [-1] * num_vertices
    counter = 0
    in_degre = in_degree(list_adj)
    while counter < num_vertices:
        i = 0
        while i < num_vertices:
            if in_degre[i] == 0  and order[i] == -1:
                break
            i += 1
        if i >= num_vertices:
            return False
```

```
22          order [i] = counter
23          counter += 1
24          for v in list_adj[i]:
25              in_degre[v] -= 1
26      return True
```

A complexidade do `in_degree()` tem complexidade $O(V + E)$, já que passa em cada vértice pelas suas arestas que estão ligadas a ela.

Isso tem complexidade de bastante. Como melhorar isso?

3.3.2 Versão rápida (lista de adjacência)

Nessa nova ideia, usamos uma fila. Chamamos a função de contagem de graus de entrada, e declaramos a queue. Adicionamos todas as fontes iniciais na queue, e criamos o counter e a lista da ordem topológica.

Enquanto a queue não estiver vazia, guardamos o primeiro elemento da fila e o retiramos. Para cada vértice ligado na fonte, decrementamos sua saída, e se ela for zero, é uma nova fonte que adicionamos na queue.

```py
from collections import deque       #uma fila com ponteiro de entrada e saída

def has_topologic_order(list_adj):
    num_vertices = len(list_adj)
    in_degre = in_degree(list_adj)
    queue = deque()
    for i in range(num_vertices):
        if in_degre[i] == 0:
            queue.append(i)
    topological_order = []
    counter = 0

    while queue:
        u = queue.popleft()
        topological_order.append(u)
        counter += 1
        for v in list_adj[u]:
            in_degre[v] -= 1
            if in_degre[v] == 0:
                queue.append(v)

    if counter == num_vertices:
        return topological_order
    else:
        return None
```

`in_degree()` é $O(V + E)$, o primeiro for é $O(V)$, e o while passa ou deveria passar, se existir a ordem topológica, em todos os vértices, e dentro dele ainda passamos por todos as suas ligações, trazendo $O(V + E)$, ou seja, $O(V + E) + O(V + E) + O(V) = O(V + E)$.

Menor caminho em grafos

4.1 Caminho mais curto em um DAG

Dado um grafo $G = (V, E)$, como criar um algoritmo capaz de gerar a SPT de um DAG iniciando na sua única fonte?

Lembre-se: nesse código, estamos considerando uma ordenação topológica já pré-determinada, por isso nosso for é simples e não precisamos olhar vértices novamente.

```py
def dag_spt(list_adj):
    inf = len(list_adj)
    distance = [inf] * inf
    parent = [-1] * inf
    distance[0] = 0
    parent[0] = 0

    for i in range(inf):
        for vizinho in list_adj[i]:
            if distance[i] + 1 < distance[vizinho]:
                distance[vizinho] = distance[i] + 1
                parent[vizinho] = i

    return distance, parent
```

O código cria um vetor de distâncias e um vetor de pais de cada vértice, e preenche o inicial, considerando ordenação topológica. Graças a característica da ordenação topológica existente, o for que fazemos passa por cada vértice da lista, e depois por cada vizinho, verificando se suas arestas estão relaxadas ou não (considerando o peso de cada aresta sempre 1), se ela tiver tensa, então atualizamos com a distância do vetor pai $+1$.

Criamos dois vetores $O(V)$, e o for de fora passa por todos os vértices ($O(V)$) e o for de dentro passa por todos os vértices (no total, não a cada iteração), trazendo $O(E)$ ao final dos dois fors. Portanto, a complexidade é $\Theta(V + E)$.

4.2 Caminho mais curto em grafo não-dirigido/com ciclos

Dado um grafo $G = (V, E)$, implemente a adaptação do algoritmo BFS para encontrar o caminho mais curto entre um vértice e todos os acessíveis por ele.

A diferença agora é que não fazemos o for na ordem dos vértices, ou seja, na ordem topológica, e agora, partimos de um v_0, e usamos um deque para administrar a ordem com que colocamos na fila, para fazermos uma busca em profundidade.

```py
from collections import deque

def spt(v0, list_adj):
    inf = len(list_adj)
    distance = [inf] * inf
    parent = [-1] * inf
    distance[v0] = 0
```

```
8       parent[v0] = 0
9
10      fila = deque()
11      fila.append(v0)
12      while fila:
13          v1 = fila.popleft()
14          for vizinho in list_adj[v1]:
15              if distance[vizinho] == inf:
16                  distance[vizinho] = distance[v1] + 1
17                  parent[vizinho] = v1
18                  fila.append(vizinho)
19
20      return distance, parent
```

Agora, o nosso if também verifica apenas se a distância não foi alterada, trazendo assim apenas uma alteração por valor `distance[v]`. Ainda, como estamos fazendo uma verificação por nível, fica claro que a distância é sempre a do pai $+1$, e que agora também funciona em ciclos, já que ele só processa o vizinho se nunca tiver visto ele antes.

A complexidade é a mesma, já que o deque é $O(1)$ para tirar à esquerda e para appendar. É a mesma complexidade $O(V + E)$, pelos mesmos motivos.

Podemos avaliar a corretude desse algoritmo através das suas invariantes: primeiro, toda aresta (v_i, v_j) de T (a árvore definida por parent) está relaxada com relação à distance; segundo, para cada aresta (v_i, v_j), se v_i está em T e v_j está fora de T, então v_i está na fila. Ao término da execução, a fila está vazia e, a partir da invariante (2), conclui-se que toda aresta com v_i em T também possui v_j em T. O vetor distance é um potencial relaxado, portanto, T é uma SPT e distance fornece o comprimento do caminho entre a raiz e os demais vértices acessíveis a partir dela.

4.3 Djikstra fast