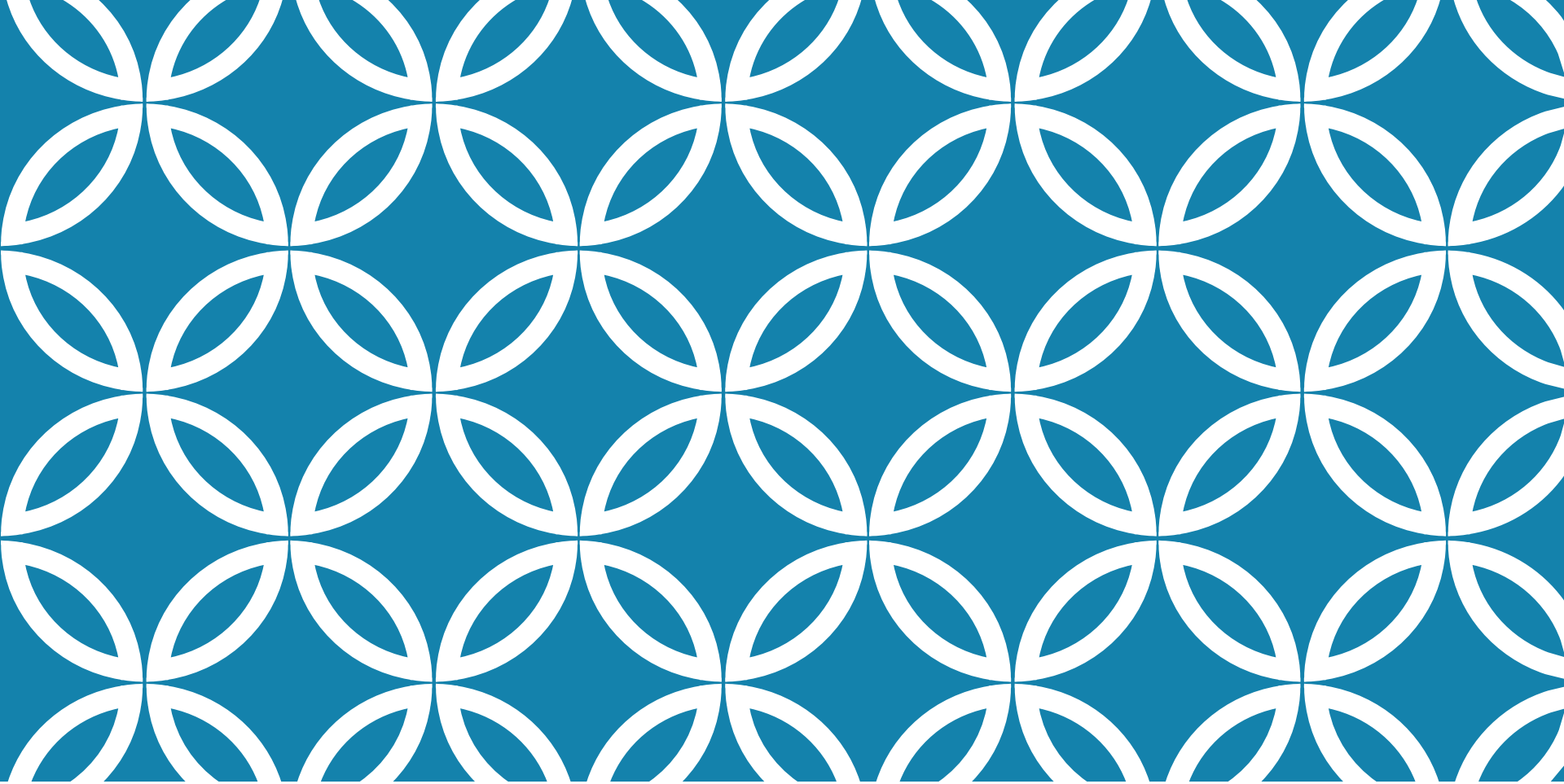


Victor Leonardo Mascarenhas Soares Horta

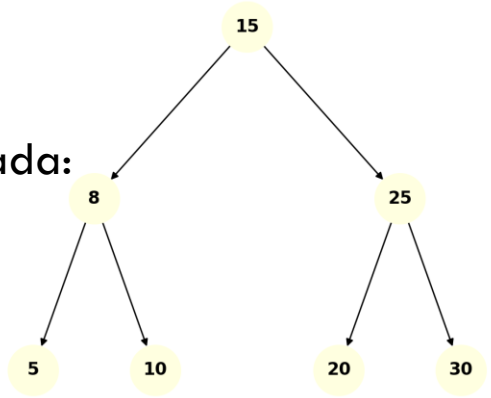
ÁRVORE BINÁRIA DE BUSCA ÓTIMA (OBST)

INTRODUÇÃO

O que é uma árvore binária de busca(BST)?

Uma **árvore binária de busca** é um grafo $G = (V, E)$tal que:

1. Gé **conexo**.
2. Gé **acíclico**.
3. Existe uma raiz r que define uma estrutura **hierárquica**.
4. Cada nó tem no máximo 2 filhos (grau ≤ 3 se considerar o pai).
5. Existe uma **ordem total** $<$sobre as chaves em V, e ela é respeitada:
    1. Subárvore esquerda = elementos menores.
    2. Subárvore direita = elementos maiores.



Do ponto de vista de **uso**, ela permite buscas rápidas: cada comparação descarta metade dos nós possíveis.

PROBLEMA

O **Problema da Árvore Binária de Busca Ótima (OBST)** consiste em determinar a estrutura de uma árvore binária de busca que **minimiza o custo esperado das operações de busca,** dado que não apenas conhecemos o conjunto de chaves, mas também as **probabilidades associadas a cada chave.**

Formalmente, seja $K = \{k_1, k_2, \ldots, k_n\}$ um conjunto ordenado de chaves e p_i a probabilidade de busca da chave k_i .O objetivo é construir uma BST de modo a reduzir:

$$E[\text{custo}] = \sum_{i=1}^{n} p_i \cdot \text{profundidade}(k_i)$$

onde profundidade(k_i) representa a posição do nó k_i na árvore (raiz com profundidade 1).

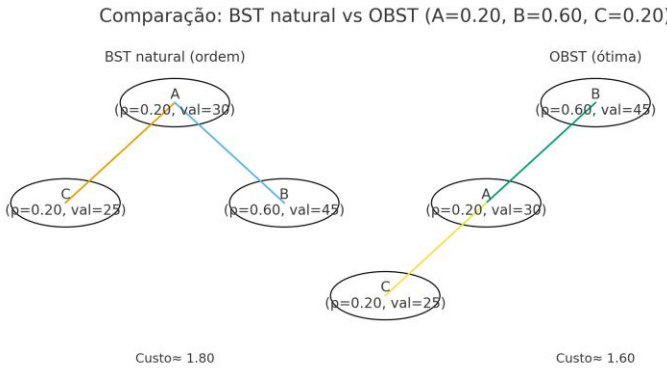
🔎 **Exemplo intuitivo**

Se temos 3 chaves:

A: 20% das buscas e valor 30

B: 60% das buscas e valor 45

C: 20% das buscas e valor 25



Em uma árvore **sem considerar probabilidades,** A ficaria na raiz.

Mas como B é a mais provável (60%), a **OBST coloca B como raiz,** para reduzir o custo médio.

APLICAÇÕES

Esse problema não é só teórico. Ele aparece em várias aplicações:

Em **compiladores,** para organizar tabelas de símbolos.
Em **bancos de dados,** quando muitas consultas são feitas em alguns registros específicos.
Em **compressão e recuperação de informação,** onde acessamos dados com frequências diferentes.

E, no dia a dia?

Organizar prateleiras de supermercado
Menus de aplicativos
Agendas

O que é mais usado fica em destaque.

COMO O ALGORITMO RESOLVE

A solução do problema usa **programação dinâmica**. Não vou entrar nos detalhes, mas a ideia geral é esta:

1. Divide em subproblemas

Analisa intervalos menores de chaves (subárvores).

Ex.: $\{A, B, C, D\}$ →primeiro A, B, C, Dsozinhos → depois pares (A, B) →depois trios (A, B, C) ,etc.

2. Usa tabelas auxiliares (Programação Dinâmica)

$e[i, j]$:custo mínimo do intervalo.

$w[i, j]$:soma das probabilidades.

$root[i, j]$:raiz ótima daquele intervalo.

Evita recalcular problemas já resolvidos.

3. Testa raízes possíveis

$$e[i, j] = \min_{r=i}^{j} \quad (e[i, r-1] + e[r, +1j] + w[i, j])$$

4. Reconstrói a árvore ótima

Após preencher as tabelas, identifica a raiz global.

Usa $root[i, j]$para reconstruir recursivamente a árvore final.

COMO O ALGORITMO RESOLVE

“O custo de um intervalo de chaves é o menor valor que consigo ao testar cada possível raiz, somando o custo da esquerda, o custo da direita e o peso do intervalo.”

EXEMPLO

Suponha que temos um sistema de busca de nomes em uma lista ordenada de quatro pessoas: **Ana, Bruno, Carla e Diego.**
As probabilidades de cada nome ser pesquisado são:
Ana: **35%**
Bruno: **25%**
Carla: **25%**
Diego: **15%**
Além disso, existe uma pequena chance de buscas por nomes que não estão na lista, distribuída igualmente entre os espaços entre as chaves.

Como organizar esses nomes em uma Árvore Binária de Busca de forma que o número médio de comparações para encontrar um nome seja o menor possível?

Em outras palavras, queremos construir a **Árvore Binária de Busca Ótima (OBST)** para esse conjunto de chaves e probabilidades.

CÓDIGO EM PYTHON

```python
from typing import List, Any
import sys

# -------------------------
# Validação de p e q
# -------------------------
def validate_probabilities(p: List[float], q: List[float]) -> None:
    if len(q) != len(p) + 1:
        raise ValueError("q deve ter tamanho len(p)+1 (há n+1 gaps para n chaves).")
    if any(x < 0 for x in p + q):
        raise ValueError("Probabilidades negativas não são permitidas.")
    s = sum(p) + sum(q)
    if abs(s - 1.0) > 1e-6:
        raise ValueError(f"As probabilidades devem somar 1.0 (atual: {s:.6f}).")
```

CÓDIGO EM PYTHON

```python
# Núcleo: OBST (DP clássica)
# --------------------------

#Implementa a OBST por programação dinâmica e devolve três tabelas:
def optimal_bst(p: List[float], q: List[float], keys: List[Any]):
    n = len(p)
    if len(keys) != n:
        raise ValueError("keys e p devem ter o mesmo tamanho.")
    #menor custo esperado para as chaves do intervalo i..j;
    e = [[0.0]*(n+2) for _ in range(n+2)]
    #peso (soma das probabilidades p e q) no intervalo i..j;
    w = [[0.0]*(n+2) for _ in range(n+2)]
    #índice da raiz ótima escolhida para i..j.
    root = [[0]*(n+2) for _ in range(n+2)]

    #Inicializa os casos-base (intervalo vazio i..i-1):
    #custo = q[i-1], peso = q[i-1].
    #Isto modela a chance de "não encontrar" entre as chaves.
    for i in range(1, n+2):
        e[i][i-1] = q[i-1]
        w[i][i-1] = q[i-1]
    for L in range(1, n+1):
        for i in range(1, n-L+2):
            j = i + L - 1
            #Inicializa o custo do intervalo com infinito
            e[i][j] = float("inf")
            #Atualiza o peso do intervalo i..j a partir do peso i..j-1:
            #adiciona p[j-1] (prob. da chave j) e q[j] (gap após j).

            w[i][j] = w[i][j-1] + p[j-1] + q[j]
            melhor_r, melhor_custo = None, float("inf")
            #Testa cada chave r em i..j como candidata a raiz do intervalo.
            for r in range(i, j+1):
```

CÓDIGO EM PYTHON

```
66          for r in range(i, j+1):
67
68              #Calcula o custo se r for raiz:
69              #custo da subárvore esquerda e[i][r-1] +
70              # custo da subárvore direita e[r+1][j] +
71              #w[i][j] (todo mundo do intervalo fica um nível mais fundo quando escolhemos uma raiz).
72
73              custo = e[i][r-1] + e[r+1][j] + w[i][j]
74              if custo < melhor_custo:
75                  melhor_custo, melhor_r = custo, r
76          e[i][j] = melhor_custo
77          root[i][j] = melhor_r
78      return e, w, root
```

CÓDIGO EM PYTHON

```python
# --------------------------
# Ordens (percursos) a partir da tabela root
#Essas três funções percorrem recursivamente os intervalos indicados pela tabela root para gerar, sem construir a árvore, as listas de chaves em pré-ordem, em-ordem e
# pós-ordem.
# --------------------------
def preorder_from_root(root, keys, i, j):
    if j < i:
        return []
    r = root[i][j]
    return [keys[r-1]] + preorder_from_root(root, keys, i, r-1) + preorder_from_root(root, keys, r+1, j)

def inorder_from_root(root, keys, i, j):
    if j < i:
        return []
    r = root[i][j]
    return inorder_from_root(root, keys, i, r-1) + [keys[r-1]] + inorder_from_root(root, keys, r+1, j)

def postorder_from_root(root, keys, i, j):
    if j < i:
        return []
    r = root[i][j]
    return postorder_from_root(root, keys, i, r-1) + postorder_from_root(root, keys, r+1, j) + [keys[r-1]]
```

CÓDIGO EM PYTHON

```python
# --------------------------
# Programa principal
# --------------------------
def main():
    # Exemplo visível de entrada:
    keys = ["Ana", "Bruno", "Carla", "Diego"]
    p    = [0.35, 0.25, 0.25, 0.15]
    q    = [0.02, 0.02, 0.02, 0.02, 0.02]

    print(">>> EXEMPLO UTILIZADO:")
    print(f"    keys = {keys}")
    print(f"    p    = {p}")
    print(f"    q    = {q}\n")

    # Normaliza p+q para 1.0 (mantém proporções) – evita erro de soma
    total = sum(p) + sum(q)
    if abs(total - 1.0) > 1e-9:
        esc = 1.0 / total
        p = [round(x*esc, 10) for x in p]
        q = [round(x*esc, 10) for x in q]
```

CÓDIGO EM PYTHON

```python
    # Validação final
    validate_probabilities(p, q)

    print(">>> Executando OBST (programação dinâmica)...")
    e, w, root = optimal_bst(p, q, keys)
    n = len(keys)

    # Custo
    cost_dp = e[1][n]
    print(f">>> Custo mínimo esperado (DP): {cost_dp:.6f}\n")

    # Raízes por intervalo (decisões do algoritmo)
    print(">>> Raízes escolhidas por intervalo:")
    for L in range(1, n+1):
        for i in range(1, n-L+2):
            j = i + L - 1
            r = root[i][j]
            print(f"    Intervalo [{i},{j}] → chaves {keys[i-1:j]} | raiz = '{keys[r-1]}'")

    # Ordem escolhida (percursos)
    pre  = preorder_from_root(root, keys, 1, n)
    ino  = inorder_from_root(root, keys, 1, n)
    post = postorder_from_root(root, keys, 1, n)

    print("\n>>> Ordem escolhida (percursos da OBST):")
    print("    Pré-ordem (raiz → esquerda → direita):", pre)
    print("    Em-ordem  (esquerda → raiz → direita):", ino, "  (deve sair ordenada)")
    print("    Pós-ordem (esquerda → direita → raiz):", post)

    print("\n>>> FIM – A OBST minimiza o número médio de comparações considerando as probabilidades.")

```

SAÍDA

```
PS C:\Users\masca> & C:/Users/masca/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/masca/OneDrive/Documentos/Mestrado 2025,1/obst_completo.py"
>>> Executando OBST (programação dinâmica)...
>>> Custo mínimo esperado (DP): 2.036364

>>> Raízes escolhidas por intervalo:
    Intervalo [1,1] → chaves ['Ana'] | raiz = 'Ana'
    Intervalo [2,2] → chaves ['Bruno'] | raiz = 'Bruno'
    Intervalo [3,3] → chaves ['Carla'] | raiz = 'Carla'
    Intervalo [4,4] → chaves ['Diego'] | raiz = 'Diego'
    Intervalo [1,2] → chaves ['Ana', 'Bruno'] | raiz = 'Ana'
    Intervalo [2,3] → chaves ['Bruno', 'Carla'] | raiz = 'Bruno'
    Intervalo [3,4] → chaves ['Carla', 'Diego'] | raiz = 'Carla'
    Intervalo [1,3] → chaves ['Ana', 'Bruno', 'Carla'] | raiz = 'Bruno'
    Intervalo [2,4] → chaves ['Bruno', 'Carla', 'Diego'] | raiz = 'Carla'
    Intervalo [1,4] → chaves ['Ana', 'Bruno', 'Carla', 'Diego'] | raiz = 'Bruno'

>>> Ordem escolhida (percursos da OBST):
    Pré-ordem (raiz → esquerda → direita): ['Bruno', 'Ana', 'Carla', 'Diego']
    Em-ordem  (esquerda → raiz → direita): ['Ana', 'Bruno', 'Carla', 'Diego']   (deve sair ordenada)
    Pós-ordem (esquerda → direita → raiz): ['Ana', 'Diego', 'Carla', 'Bruno']

>>> FIM - A OBST minimiza o número médio de comparações considerando as probabilidades.
```

CONCLUSÃO

•**O OBST é** forma de organizar uma BST para **minimizar o custo médio** de busca, usando **probabilidades** de acerto (p) e de falha (q).

•**Como decide:** avalia subproblemas com tabelas e[i,j] (custo), w[i,j] (peso) e root[i,j] (raiz ótima), escolhendo a estrutura de **menor profundidade esperada**.

•**Por que importa:** quando o padrão de buscas é **conhecido/estável**, reduz comparações em média → **desempenho previsível**.

•**No nosso exemplo (Ana 0.35, Bruno 0.25, Carla 0.25, Diego 0.15):** a OBST escolhe **Bruno como raiz**, pois **equilibra** os acessos e diminui o custo total (mesmo Ana sendo a mais provável isoladamente).

•**Quando usar:** tabelas de símbolos, índices/consulta em BD, dicionários, autocompletar; quando **não usar**: cenários muito dinâmicos (prefira árvores balanceadas/splay).

•**Essência: probabilidade guiando estrutura** → menos comparações em média, respeitando a ordem da BST.

OBRIGADO!

O repositório no GitHub contém:

- Slides em PDF
- Código em Python
- Dados de exemplo
- README.md com link para vídeo.