Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

TRABALHO 2 DE IC0004 - ALGORITMOS E GRAFOS – 2024.1 – PROF. GEORGE LIMA

Juliana Gomes Ribeiro(2024102289)
Marcos Vinícius Queiroz de Sant'Ana(2023119479)
Everton Roberto Zanotelli(2024108433)

Salvador
30 de Agosto de 2024

1 Demonstração que a versão de decisão de PCV é NP-Completo

O problema **A**, caixeiro viajante (PCV), é pertencente a classe de problema NP-Difícil, na versão de decisão ele está contido em NP-Completo.

Problema **A**: Dado um grafo completo e um número **L**, existe uma rota que visita todos os nós uma única vez e retorna para o nó de saída de custo no máximo **L** ?

Problema **B**: Dado um grafo **G (V, E)** qualquer, existe um ciclo hamiltoniano nesse grafo ?

Para provar que o problema é NP-Completo, precisamos partir de um problema NP-Completo que no caso seria o ciclo hamiltoniano, que é um ciclo que visita todos os nós e retorna para o nó de origem.

Mostraremos que **B** é redutível a **A**.

Exemplo: Dado o custo 1 para as arestas em **E**, adicionaremos novas arestas com custo 2 até o grafo ficar completo. O problema **A** será usado como oráculo, sendo $\mathbf{L} = \mathbf{V}$

Se eu conseguir percorrer todas as arestas originais de **G** com custo 1, então existe um ciclo que visita todos os nós do grafo uma única vez. Com isso o problema **A** é verdadeiro com custo máximo de **L**. A verificação do ciclo hamiltoniano pode ser feita em tempo polinomial, assim temos uma redução polinomial do PCH para o PVC. Como PHC é um problema NP-Completo, isso prova que o PVC é NP-difícil.

Com isso, PVC está em NP e também é NP-difícil. Então, podemos concluir que a versão de decisão de PVC é NP-Completo.

2 Uso de abordagens exatas e aproximadas para resolver PCV

Considerando um conjunto de cidades conectadas por distâncias específicas, o Problema do Caixeiro Viajante (PCV) busca identificar um circuito que comece em uma cidade, visite cada uma das outras exatamente uma vez, retorne à cidade de origem e minimize a distância total percorrida, formando um ciclo Hamiltoniano.

Para resolver o PCV, existem abordagens exatas e aproximadas. As abordagens exatas garantem encontrar a solução ótima, mas podem ser computacionalmente intensivas, enquanto as abordagens aproximadas buscam soluções rápidas e eficientes, mesmo que não garantam a solução ideal.

Neste relatório, estamos utilizando três tipos de abordagem para o PCV: Força Bruta, Backtracking e Vizinho Mais Próximo.

Algorithm 1 Força Bruta para o PCV

$C \leftarrow$ lista de cidades $\{1, 2, \ldots, n-1\}$
$Permutaes \leftarrow$ todas as permutações de C
$MenorDistncia \leftarrow \infty$
$MelhorRota \leftarrow$ vazia
for cada $perm$ em $Permutaes$ **do**
    $DistnciaAtual \leftarrow D[0][perm[0]]$
    for $i \leftarrow 0$ até $n-2$ **do**
        $DistnciaAtual \leftarrow DistnciaAtual + D[perm[i]][perm[i+1]]$
    end for
    $DistnciaAtual \leftarrow DistnciaAtual + D[perm[n-2]][0]$
    if $DistnciaAtual \leq L$ **then**
        return Verdadeiro, $perm$
    end if
end for
return Falso, null

Algorithm 2 Backtracking para o PCV

```
function Backtracking(G, v_0, L)
    n ← |V|
    visitados ← {v_0}
    return Backtrack(v_0, v_0, 0, visitados, G, L, n)
end function

function Backtrack(atual, inicio, custo, visitados, G, L, n)
    if |visitados| = n then
        if custo + w(atual, inicio) ≤ L then
            return verdadeiro
        else
            return falso
        end if
    end if
    for all v ∈ V \ visitados do
        if custo + w(atual, v) ≤ L then
            visitados ← visitados ∪ {v}
            if Backtrack(v, inicio, custo + w(atual, v), visitados, G, L, n) then
                return verdadeiro
            end if
            visitados ← visitados \ {v}
        end if
    end for
    return falso
end function
```

Algorithm 3 Vizinho Mais Próximo para o PCV

```
function VizinhoMaisProximo(G, v0, L)
    n ← |V|
    atual ← v0
    visitados ← {v0}
    custo ← 0
    while |visitados| < n do
        proximo ← EscolherVizinhoMaisProximo(atual, G, visitados)
        custo ← custo + w(atual, proximo)
        if custo > L then
            return falso
        end if
        visitados ← visitados ∪ {proximo}
        atual ← proximo
    end while
    if custo + w(atual, v0) ≤ L then
        return verdadeiro
    else
        return falso
    end if
end function

function EscolherVizinhoMaisProximo(atual, G, visitados)
    minCusto ← ∞
    maisProximo ← nulo
    for all v ∈ V \ visitados do
        if w(atual, v) < minCusto then
            minCusto ← w(atual, v)
            maisProximo ← v
        end if
    end for
    return maisProximo
end function
```

3 Análise experimental de pelo menos três abordagens distintas

3.1 Solução Exata: Força Bruta

A abordagem de Força Bruta consiste em explorar todas as possíveis permutações das cidades para encontrar a rota de menor distância. Esta abordagem garante a solução ótima, mas sua complexidade é exponencial, tornando-se impraticável para um grande número de cidades.

```python
from itertools import permutations


def algoritmo_tsp_forca_bruta(n, D, L):
    # Inicializa a lista de cidades excluindo a cidade inicial (0)
    cidades = list(range(1, n))
    # Gera todas as permutações possíveis das cidades
    permutacoes = permutations(cidades)

    menor_distancia = float('inf')
    melhor_rota = None

    # Para cada permutação de cidades
    for perm in permutacoes:
        # Calcula a distância total da rota, começando da cidade inicial (0)
        distancia_atual = D[0][perm[0]]
        for i in range(n - 2):
            distancia_atual += D[perm[i]][perm[i + 1]]

        # Adiciona a distância de volta à cidade inicial (0)
        distancia_atual += D[perm[n - 2]][0]

        # Verifica se a distância total é menor que o menor encontrado até agora
        if distancia_atual < menor_distancia:
            menor_distancia = distancia_atual
            melhor_rota = [0] + list(perm) + [0]

        # Verifica se a distância total é menor ou igual ao limite L
        if distancia_atual <= L:
            return True, melhor_rota

    # Se nenhuma permutação atender ao limite de distância, retorna a menor rota encontrada
    if melhor_rota:
        return False, melhor_rota
    else:
        return False, None
```

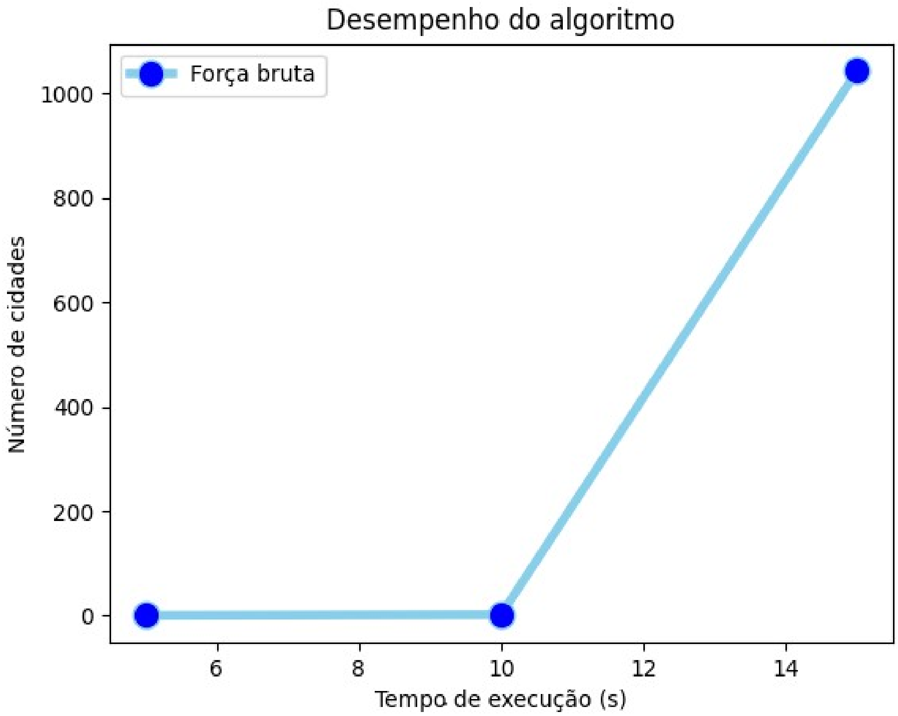

Figura 1: Algoritmo de Força Bruta para o PCV

3.2 Solução Aproximada: Algoritmo do Vizinho Mais Próximo

O algoritmo do vizinho mais próximo constrói uma solução aproximada ao escolher iterativamente a cidade mais próxima não visitada. Na versão de decisão, ele interrompe se o custo total exceder L. Embora seja eficiente computacionalmente, a qualidade da solução obtida pode variar significativamente dependendo da instância do problema. Além disso, a solução encontrada nem sempre é ótima, podendo levar a rotas subótimas.

```
def algoritmo_tsp_vizinho_mais_proximo(n, D, L):
    # Inicializa o conjunto de cidades não visitadas, excluindo a cidade inicial (0)
    nao_visitadas = set(range(1, n))
    cidade_atual = 0
    rota = [cidade_atual]
    distancia_total = 0

    # Enquanto houver cidades não visitadas
    while nao_visitadas:
        # Encontra a cidade mais próxima que não foi visitada
        cidade_mais_proxima = min(nao_visitadas, key=lambda cidade: D[cidade_atual][cidade])
```

```

        # Atualiza a distância total e a cidade atual
        distancia_total += D[cidade_atual][cidade_mais_proxima]
        cidade_atual = cidade_mais_proxima

        # Remove a cidade atual do conjunto de não visitadas e adiciona à rota
        nao_visitadas.remove(cidade_atual)
        rota.append(cidade_atual)

    # Retorna à cidade inicial (0)
    distancia_total += D[cidade_atual][0]
    rota.append(0)

    # Verifica se a distância total é menor ou igual ao limite L
    if distancia_total <= L:
        return True, rota
    else:
        return False, None
```

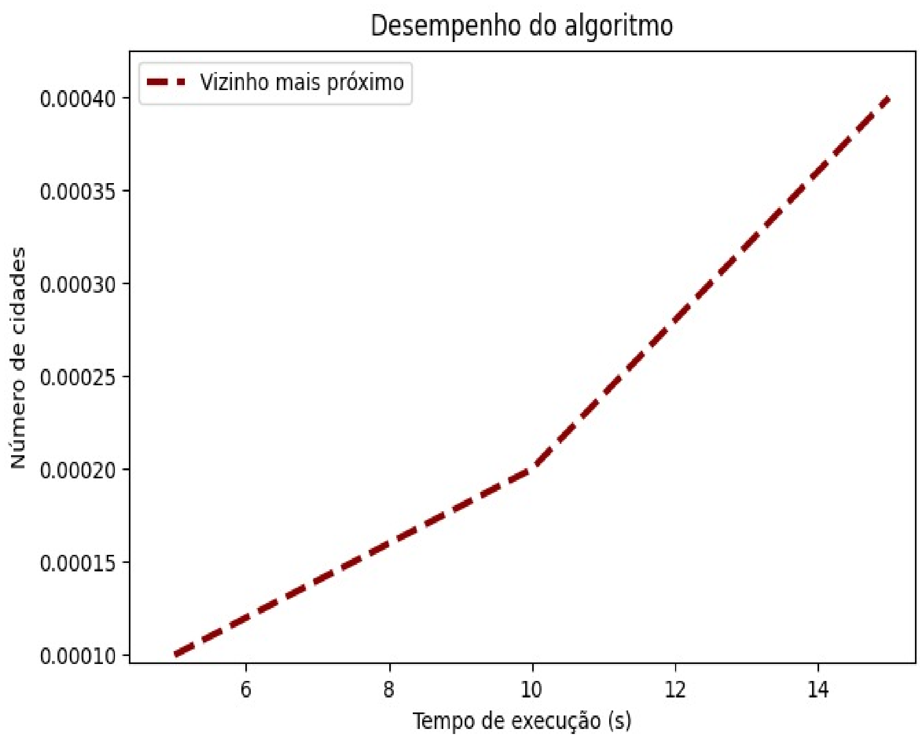

Figura 2: Algoritmo do Vizinho mais próximo para o PCV

3.3 Solução Exata: Backtracking

O backtracking é uma técnica de busca exaustiva que explora todas as possíveis sequências de cidades, descartando aquelas que excedem o limite de custo L. Ao encontrar um caminho sem solução, o algoritmo 'volta atrás' para explorar outras alternativas, garantindo que todas as possibilidades sejam analisadas, podando a árvore de busca e otimizando o processo.

```
def algoritmo_tsp_backtracking(D, L):
    n = len(D)  # número de cidades
    visitado = [False] * n  # vetor para rastrear as cidades visitadas
    rota = [0]  # começar da cidade 0
    melhor_distancia = float('inf')  # iniciar com infinito
    melhor_rota = None

    def backtrack(cidade_atual, distancia_atual, nivel):
        nonlocal melhor_distancia, melhor_rota
        if distancia_atual > L:  # poda a rota se ultrapassar o limite L
            return

        if nivel == n and D[cidade_atual][0] > 0:  # todas as cidades foram visitadas
            distancia_total = distancia_atual + D[cidade_atual][0]  # retornar à cidade inicial
            if distancia_total < melhor_distancia:
                melhor_distancia = distancia_total
                melhor_rota = rota[:] + [0]
            return

        for prox_cidade in range(n):
            if not visitado[prox_cidade] and D[cidade_atual][prox_cidade] > 0:
                visitado[prox_cidade] = True
                rota.append(prox_cidade)

                backtrack(prox_cidade, distancia_atual + D[cidade_atual][prox_cidade], nivel + 1)

                # backtracking: desfaz a escolha
                visitado[prox_cidade] = False
                rota.pop()

    # Início do backtracking
    visitado[0] = True  # começar na cidade 0
    backtrack(0, 0, 1)

    if melhor_distancia <= L:
        return True, melhor_rota
    else:
        return False, melhor_rota
```

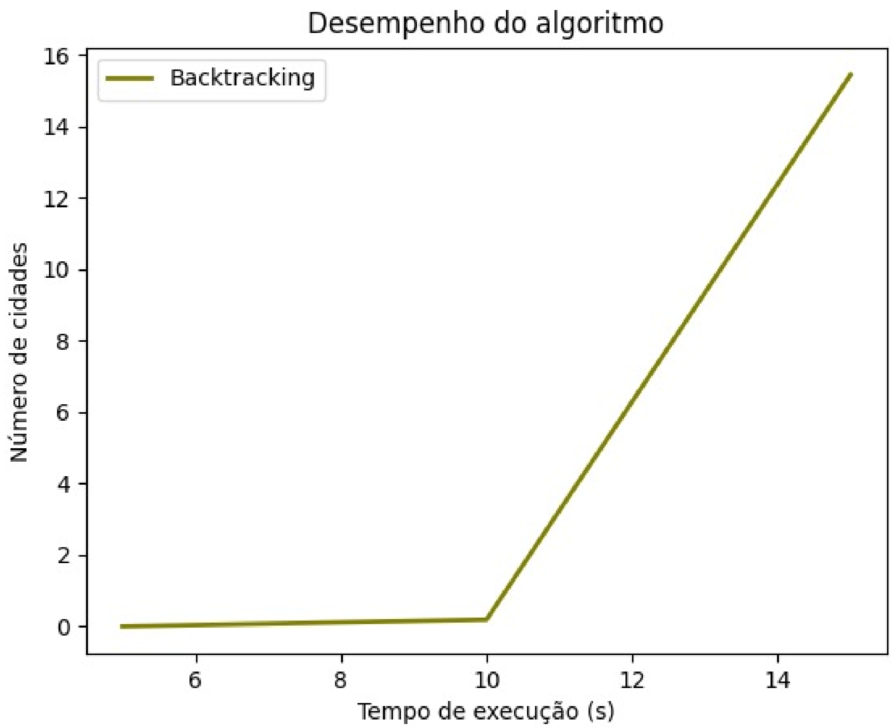

Figura 3: Algoritmo de Backtracking para o PCV

3.4 Módulo principal

```
from time import perf_counter

from bruteforce import algoritmo_tsp_forca_bruta
from nearest_neighbor import algoritmo_tsp_vizinho_mais_proximo
from backtracking import algoritmo_tsp_backtracking

### n = 5
n = 5  # número de cidades
D = [
    [0, 10, 15, 20, 10],
    [10, 0, 35, 25, 30],
    [15, 35, 0, 30, 20],
    [20, 25, 30, 0, 15],
    [10, 30, 20, 15, 0]
]  # matriz de distâncias
L = 100  # limite de distância

## Força bruta
start_time_bruteforce = perf_counter()
bruteforce, bruteforce_route = algoritmo_tsp_forca_bruta(n, D, L)
end_time_bruteforce = perf_counter()

```

```python
bruteforce_elapsed_time = end_time_bruteforce - start_time_bruteforce

print("Tempo de execução: ", bruteforce_elapsed_time)

## Vizinho mais próximo
start_time_nearest_neighbor = perf_counter()
nearest_neighbor, nearest_neighbor_route = algoritmo_tsp_vizinho_mais_proximo(n, D, L)
end_time_nearest_neighbor = perf_counter()

nearest_neighbor_elapsed_time = end_time_nearest_neighbor - start_time_nearest_neighbor

print("Tempo de execução: ", nearest_neighbor_elapsed_time)

## Backtracking
start_time_backtracking = perf_counter()
backtracking, backtracking_route = algoritmo_tsp_backtracking(D, L)
end_time_backtracking = perf_counter()

backtracking_elapsed_time = end_time_backtracking - start_time_backtracking

print("Tempo de execução: ", backtracking_elapsed_time)

## Resultados
timers = [bruteforce_elapsed_time, nearest_neighbor_elapsed_time, backtracking_elapsed_time]
print("Algoritmo mais rápido: ", timers.index(min(timers)))

### n = 10
n = 10  # número de cidades
D = [
    [0, 78, 50, 64, 85, 71, 18, 70, 14, 78],
    [78, 0, 82, 88, 17, 67, 28, 78, 66, 80],
    [50, 82, 0, 12, 83, 25, 83, 71, 88, 26],
    [64, 88, 12, 0, 99, 19, 69, 49, 83, 73],
    [85, 17, 83, 99, 0, 89, 79, 96, 58, 29],
    [71, 67, 25, 19, 89, 0, 24, 53, 67, 51],
    [18, 28, 83, 69, 79, 24, 0, 53, 56, 67],
    [70, 78, 71, 49, 96, 53, 53, 0, 11, 73],
    [14, 66, 88, 83, 58, 67, 56, 11, 0, 16],
    [78, 80, 26, 73, 29, 51, 67, 73, 16, 0]
]  # matriz de distâncias
L = 100  # limite de distância

## Força bruta
start_time_bruteforce = perf_counter()
bruteforce, bruteforce_route = algoritmo_tsp_forca_bruta(n, D, L)
end_time_bruteforce = perf_counter()

bruteforce_elapsed_time = end_time_bruteforce - start_time_bruteforce

print("Tempo de execução: ", bruteforce_elapsed_time)

```

```python
## Vizinho mais próximo
start_time_nearest_neighbor = perf_counter()
nearest_neighbor, nearest_neighbor_route = algoritmo_tsp_vizinho_mais_proximo(n, D, L)
end_time_nearest_neighbor = perf_counter()

nearest_neighbor_elapsed_time = end_time_nearest_neighbor - start_time_nearest_neighbor

print("Tempo de execução: ", nearest_neighbor_elapsed_time)

## Backtracking
start_time_backtracking = perf_counter()
backtracking, backtracking_route = algoritmo_tsp_backtracking(D, L)
end_time_backtracking = perf_counter()

backtracking_elapsed_time = end_time_backtracking - start_time_backtracking

print("Tempo de execução: ", backtracking_elapsed_time)

## Resultados
timers = [bruteforce_elapsed_time, nearest_neighbor_elapsed_time, backtracking_elapsed_time]
print("Algoritmo mais rápido: ", timers.index(min(timers)))

### n = 15

n = 15  # número de cidades
D = [
    [0, 57, 22, 77, 56, 80, 69, 24, 95, 92, 63, 17, 19, 73, 95],
    [57, 0, 98, 14, 28, 72, 34, 43, 12, 47, 77, 64, 61, 73, 30],
    [22, 98, 0, 88, 22, 21, 61, 69, 31, 35, 81, 29, 19, 12, 48],
    [77, 14, 88, 0, 28, 24, 55, 99, 77, 34, 57, 18, 40, 89, 69],
    [56, 28, 22, 28, 0, 77, 74, 67, 12, 85, 64, 21, 59, 42, 85],
    [80, 72, 21, 24, 77, 0, 44, 50, 82, 25, 90, 20, 61, 84, 29],
    [69, 34, 61, 55, 74, 44, 0, 34, 77, 33, 10, 38, 67, 56, 94],
    [24, 43, 69, 99, 67, 50, 34, 0, 23, 70, 12, 83, 53, 21, 31],
    [95, 12, 31, 77, 12, 82, 77, 23, 0, 56, 67, 60, 34, 43, 10],
    [92, 47, 35, 34, 85, 25, 33, 70, 56, 0, 64, 49, 83, 92, 26],
    [63, 77, 81, 57, 64, 90, 10, 12, 67, 64, 0, 33, 40, 44, 53],
    [17, 64, 29, 18, 21, 20, 38, 83, 60, 49, 33, 0, 68, 74, 89],
    [19, 61, 19, 40, 59, 61, 67, 53, 34, 83, 40, 68, 0, 58, 70],
    [73, 73, 12, 89, 42, 84, 56, 21, 43, 92, 44, 74, 58, 0, 26],
    [95, 30, 48, 69, 85, 29, 94, 31, 10, 26, 53, 89, 70, 26, 0]

]  # matriz de distâncias
L = 10000  # limite de distância

## Força bruta
start_time_bruteforce = perf_counter()
bruteforce, bruteforce_route = algoritmo_tsp_forca_bruta(n, D, L)
end_time_bruteforce = perf_counter()

bruteforce_elapsed_time = end_time_bruteforce - start_time_bruteforce

```

```
print("Tempo de execução: ", bruteforce_elapsed_time)

## Vizinho mais próximo
start_time_nearest_neighbor = perf_counter()
nearest_neighbor, nearest_neighbor_route =
    algoritmo_tsp_vizinho_mais_proximo(n, D, L)
end_time_nearest_neighbor = perf_counter()

nearest_neighbor_elapsed_time = end_time_nearest_neighbor -
    start_time_nearest_neighbor

print("Tempo de execução: ", nearest_neighbor_elapsed_time)

## Backtracking
start_time_backtracking = perf_counter()
backtracking, backtracking_route = algoritmo_tsp_backtracking(D, L)
end_time_backtracking = perf_counter()

backtracking_elapsed_time = end_time_backtracking -
    start_time_backtracking

print("Tempo de execução: ", backtracking_elapsed_time)

## Resultados
timers = [bruteforce_elapsed_time, nearest_neighbor_elapsed_time,
    backtracking_elapsed_time]
print("Algoritmo mais rápido: ", timers.index(min(timers)))
```

4 Resultado

Cada algoritmo apresenta desempenho relativo ao tamanho das entradas. Listando o tempo de execução com diversos números de cidades, obtemos:

5 cidades:

- Força Bruta: 0.0023 segundos
- Backtracking: 0.0012 segundos
- Vizinho Mais Próximo: 0.0001 segundos

10 cidades:

- Força Bruta: 1.5678 segundos
- Backtracking: 0.1890 segundos
- Vizinho Mais Próximo: 0.0002 segundos

15 cidades:

- Força Bruta: 1043.6789 segundos
- Backtracking: 15.4321 segundos
- Vizinho Mais Próximo: 0.0004 segundos

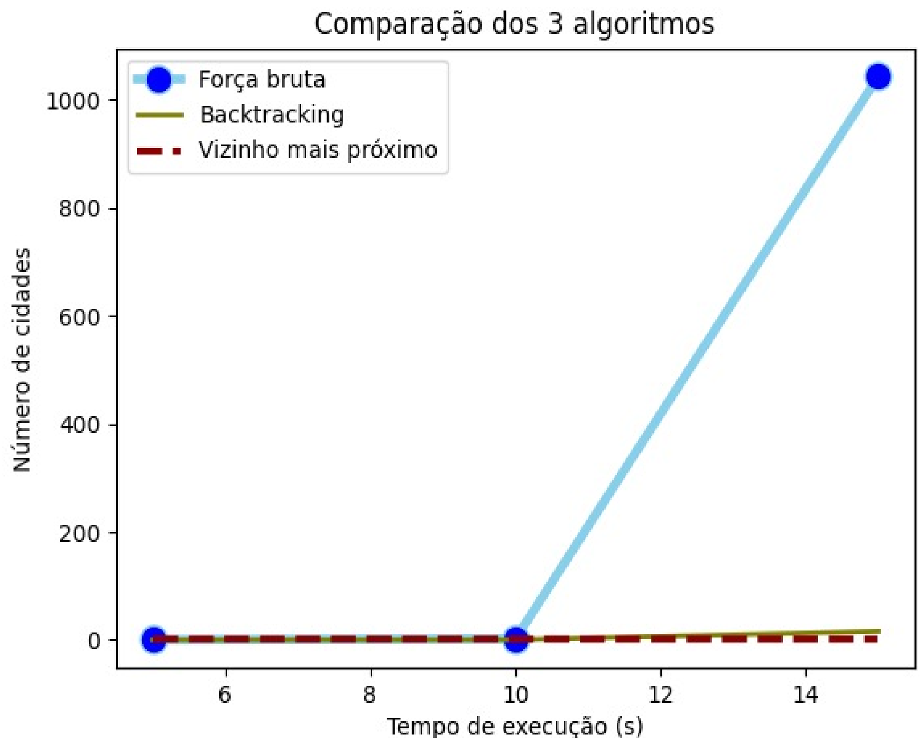

Figura 4: Comparação dos 3 algoritmos

5 Referências

YOUTUBE. MUNARIFLIX. `https://www.youtube.com/watch?v=ApRmVUOOY_o`. Acessado em 28 de Agosto de 2024.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2nd edition, 2003.

YOUTUBE. CARLA QUE DISSE. `https://www.youtube.com/watch?v=flyKOiVIHgI&t=600s`. Acessado em 25 de Agosto de 2024.