Trabalho 4 - Programação Dinâmica

Letícia Americano Lucas - 691290

Problema:

O Problema da Mochila 1/0 é um desafio clássico de otimização em computação para otimização de algoritmos, nele, o objetivo do problema é maximizar o valor total dos itens colocados na mochila, sem ultrapassar a capacidade máxima com uma lista de itens fornecida, sendo que cada item tem um peso e um valor. Logo, uma forma de resolvê-lo é com a programação dinâmica

Algoritmo de Programação Dinâmica:

A programação dinâmica é uma técnica de otimização usada para resolver problemas computacionais, dividindo-os em subproblemas menores. Sendo a ideia principal, armazenar os resultados dos subproblemas presentes para evitar recálculos, economizando tempo de execução.

No problema clássico da mochila 0/1, a programação dinâmica pode ser aplicada usando uma matriz para armazenar os valores ótimos para diferentes capacidades e números de itens.

```python
def dynamic_programming(items):
    optimal_items = []
    capacity = items['Capacidade da Mochila']
    weights = [item['Peso'] for item in items['Itens'].values()]

    for i in range(len(weights) + 1):
        row = []
        for j in range(capacity + 1):
            if i == 0 or j == 0:
                row.append(0)
            elif weights[i - 1] <= j:
                row.append(max(optimal_items[i - 1][j - weights[i - 1]] + weights[i - 1], optimal_items[i - 1][j]))
            else:
                row.append(optimal_items[i - 1][j])
        optimal_items.append(row)

    i, j = len(weights), capacity
    selected_items = []
    while i > 0 and j > 0:
        if optimal_items[i][j] != optimal_items[i - 1][j]:
            selected_items.append(i - 1)
            j -= weights[i - 1]
        i -= 1

    selected_weights = [weights[item_idx] for item_idx in selected_items]
    selected_ids = [item_idx + 1 for item_idx in selected_items]

    return sum(selected_weights), selected_ids
```

Para resolver esse problema, a função começa inicializando uma lista chamada optimal_items, que servirá como uma matriz para armazenar os valores ótimos dos subproblemas. A capacidade da mochila e os pesos dos itens são extraídos do dicionário fornecido.

Em seguida, preenche a matriz optimal_items, sendo feito usando dois loops que percorrem todas as combinações possíveis de itens e capacidades. Sendo assim, cada célula optimal_items[i][j] na matriz representa o valor ótimo para uma configuração específica de escolha de itens até o índice i e uma capacidade j. Logo, calcula-se o valor

ótimo para cada subproblema com base nos pesos dos itens disponíveis e na capacidade da mochila.

Após o preenchimento da matriz optimal_items, a função determina quais itens foram selecionados para alcançar o valor ótimo. Ela utiliza a matriz para rastrear os itens que foram escolhidos e armazena os índices dos itens selecionados na lista selected_items.

Algoritmo de Greedy:

Sendo um algoritmo guloso, o Greedy é um método que usa de escolhas locais na esperança de encontrar a melhor solução global ideal. No contexto do Problema da Mochila 0/1, ele, sendo um algoritmo ganancioso, selecionaria itens com base em uma heurística específica, como valor por unidade de peso, na tentativa de alcançar a solução ótima.

```python
def greedy(items):
    total_weight = 0
    selected_items = []
    selected_ids = []

    capacity = items['Capacidade da Mochila']
    items_dict = items['Itens']
    sorted_items_list = sorted(items_dict.items(), key=lambda x: x[1]['Peso'])

    for item_id, item_details in sorted_items_list:
        if item_id not in selected_ids and total_weight + item_details['Peso'] <= capacity:
            total_weight += item_details['Peso']
            selected_items.append({'Item': item_id, 'Peso': item_details['Peso']})
            selected_ids.append(item_id)

    return selected_ids, total_weight
```

Como visto pelo algoritmo demostrado, o algoritmo Greedy recebe um dicionário que representa o conjunto de itens possíveis para inserir na mochila. Esse dicionário inclui informações sobre a capacidade máxima da mochila e uma lista de itens, cada um identificado por um número e associado a um peso.

Dentro da função, são inicializadas variáveis para controlar o peso total dos itens selecionados, a lista de itens escolhidos e os identificadores dos itens selecionados. A lógica é baseada na ordenação dos itens disponíveis por peso, permitindo iterar sobre eles em ordem crescente de peso.

Durante a iteração, cada item é verificado para garantir que não tenha sido selecionado anteriormente e que sua adição à mochila não ultrapasse a capacidade máxima. Se essas condições forem atendidas, o item é adicionado à lista de itens escolhidos, e seu peso é somado ao peso total.

Análise de Dados:

Utilizando a função generate_input, o código cria conjuntos aleatórios de dados para resolver o problema da Mochila. Assim, gerando um arquivo "inputs.txt" com diferentes capacidades de mochila e pesos variados para cada item. Assim, facilitando a a avaliação de algoritmos de resolução da Mochila ao alterar o número de conjuntos (num_inputs) e ajustar as capacidades e pesos máximos permitidos.

```
num_inputs = 10
max_capacity = 100
max_items = 10
max_weight = 50
```

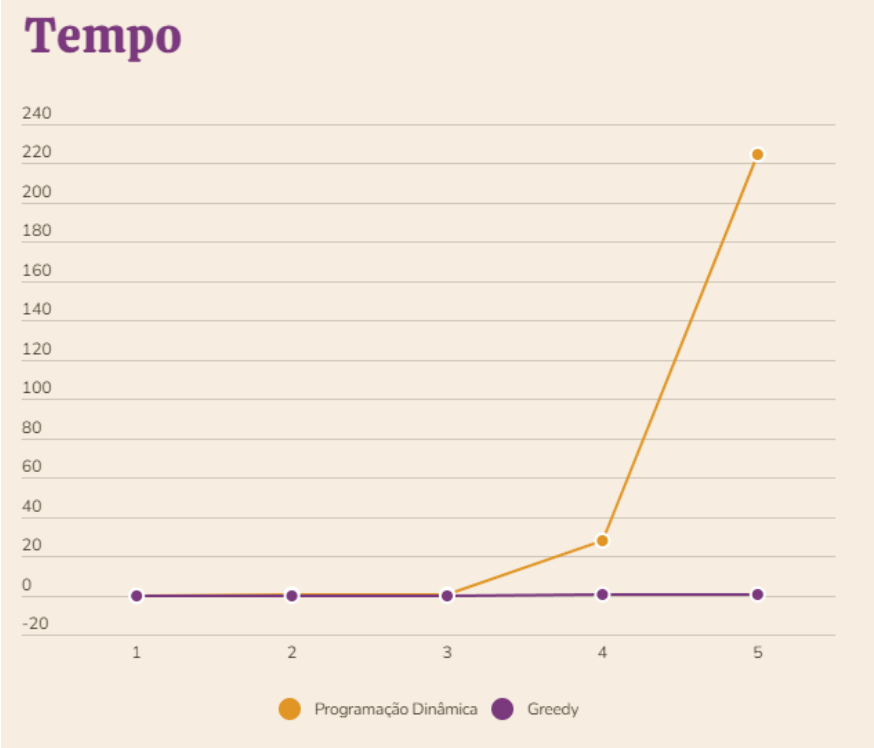
```
num_inputs = 50
max_capacity = 500
max_items = 50
max_weight = 250
```

```
num_inputs = 100
max_capacity = 1000
max_items = 100
max_weight = 500
```

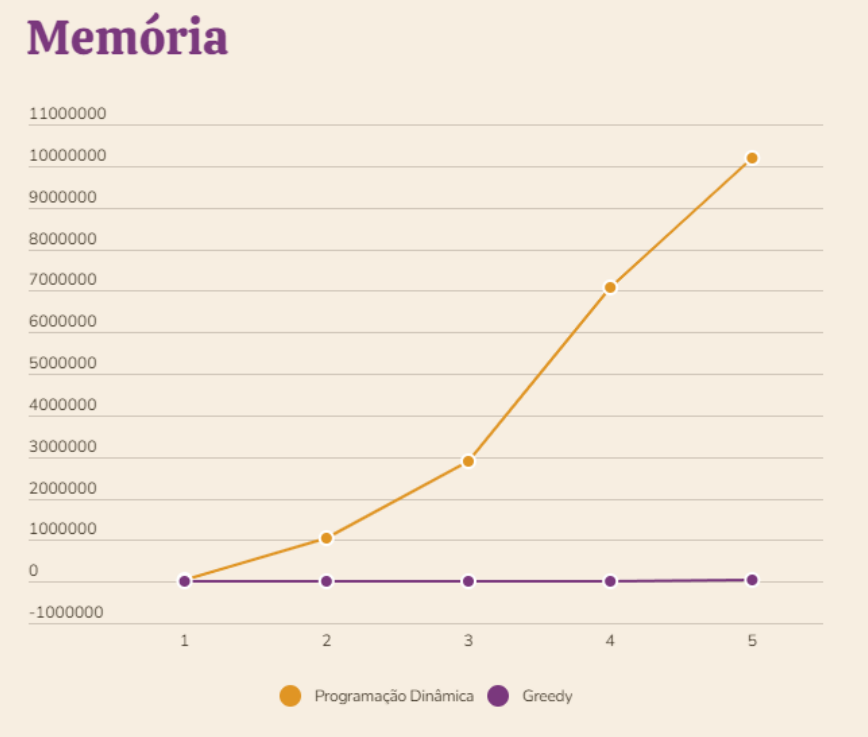
```
num_inputs = 500
max_capacity = 2500
max_items = 250
max_weight = 1250
```

```
num_inputs = 1000
max_capacity = 5000
max_items = 500
max_weight = 2500
```

Certamente, ao analisar as saídas, observa-se um aumento no tempo de processamento na abordagem de programação dinâmica à medida que a quantidade de entradas aumenta. Esse aumento é atribuído à necessidade do computador de intensificar o processamento para lidar com uma carga maior de dados, o que pode resultar nesse aumento no tempo de execução.



Além disso, ao analisar os gráficos resultantes das saídas dos diferentes inputs, é notável que o uso de memória na abordagem de programação dinâmica aumenta consideravelmente. Isso ocorre devido ao princípio fundamental dessa técnica, que exige o armazenamento de resultados intermediários para evitar recálculos, levando a uma maior utilização de memória em comparação com outras abordagens.



Conclusão:

Ao analisar os dados obtidos, verifica-se que o tempo de processamento na programação dinâmica aumenta significativamente com o acréscimo de entradas, devido à necessidade do computador em lidar com maior volume de dados. Além disso, os gráficos revelam um aumento expressivo no uso de memória nessa abordagem, decorrente do princípio de armazenamento de resultados intermediários. Embora a programação dinâmica demande mais recursos computacionais, sua complexidade pode resultar em soluções ótimas, principalmente em problemas nos quais os algoritmos gulosos podem não oferecer o melhor resultado. Assim, mesmo com um tempo de processamento mais extenso e maior consumo de memória, a programação dinâmica tende a fornecer soluções mais precisas e ótimas em comparação aos algoritmos gulosos em determinados contextos de problemas.