Licenciatura em Ciência de Dados - 1º ano

Relatório: "Carregamento e ordenação de Big Data"

Unidade Curricular de Estruturas de Dados e Algoritmos

17 de abril de 2023

Discentes: João Dias nº 110305 / David Franco nº 110733

2022/2023

iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

a) Leitura do CSV

Para ler o CSV desenvolveu-se a função **'partitions'** que leva como argumentos o **'file_path'** - caminho para o ficheiro com os dados - e o **'n'** - número de observações por partição.

```
def partitions(file_path, n):
    with open(file_path, "r") as file:
        file_reader = csv.reader(file)
        header = next(file_reader)
        for chunk in iter(lambda: list(islice(file_reader, n)), []):
            yield chunk
```

Fig. 1 - Implementação da função partitions

A função foi pensada para dividir o CSV em partições de tamanho "n" e devolver um gerador que permitisse iterar sobre as partições sem que fosse necessário carregá-las todas na memória ao mesmo tempo, por se tratar de um ficheiro de grande dimensão com 100 000 observações.

Para isso, começou-se por abrir o ficheiro CSV, que se encontrava no caminho **'file_path'** no modo leitura ("r"). O **'with'** teve como objetivo garantir que o ficheiro fosse corretamente fechado após a leitura.

```
with open(file_path, "r") as file:
```

Fig. 2 - Parte da função partitions (abertura do ficheiro)

De seguida, criou-se o objeto **'file_reader'** que permitiu ler o ficheiro como uma lista de listas, através da função **'csv.reader'** da biblioteca csv, recebendo como argumento o ficheiro aberto anteriormente.

```
file_reader = csv.reader(file)
```

Fig. 3 - Parte da função partitions (leitura do ficheiro)

A primeira linha do CSV, que corresponde ao cabeçalho “x”, foi lida e guardada na variável **‘header’**.

```
    header = next(file_reader)
```

Fig. 4 - Parte da função partitions (guardar o cabeçalho)

O **‘for’** loop foi o responsável por percorrer o ficheiro para a criação das várias partições do CSV e a função **‘iter’** foi utilizada para criar um iterador que chama a função **‘lambda’** a cada iteração, que por sua vez permite aplicar a função **‘islice’**, do módulo itertools, para criar as partições de tamanho “n”, até que seja devolvida uma lista vazia pela função **‘lambda’**. A cada iteração do **‘for’** loop, uma nova partição é criada e devolvida pelo gerador.

```
    for chunk in iter(lambda: list(islice(file_reader, n)), []):
```

Fig. 5 - Parte da função partitions (criar as partições)

Por fim, o **‘yield’** é utilizado para devolver um gerador que permite iterar sobre as partições do arquivo sem que tenham de ser todas carregadas na memória ao mesmo tempo.

```
    yield chunk
```

Fig. 6 - Parte da função partitions (devolver o gerador)

b) Sorting

Para a escolha do algoritmo de ordenação optou-se pelo Merge Sort, uma vez que o seu desempenho é consistente e é independente da dimensão ou distribuição dos dados a ordenar. Isto acontece pelo facto do

Merge Sort dividir os dados em listas menores e, em seguida, combiná-las, em ordem crescente ou decrescente. Este processo recursivo garante um desempenho estável e previsível, mesmo perante um elevado número de observações, como é o caso do CSV em análise.

Além disso, tendo em conta que o objetivo do projeto é encontrar o valor máximo e mínimo de uma lista de 100 000 valores, tendo em vista a complexidade temporal dos algoritmos, importa ver que o Merge Sort apresenta uma complexidade temporal de O(n log n), o que vem reforçar o que já foi dito anteriormente relativamente à sua eficiência para grandes quantidades de dados. A título de exemplo, tanto o Bubble Sort como o Insertion Sort apresentam uma complexidade temporal de O(n^2), sendo por isso piores opções. Pelos motivos acima descritos, optou-se pela implementação do Merge Sort na função **'sorting'**, conforme mostra a figura 7.

```
def sorting(lst, ascending):
    if len(lst) <= 1:
        return lst, 0, lst[0], lst[0]

    mid = len(lst) // 2
    left, left_iterations, left_min, left_max = sorting(lst[:mid], ascending)
    right, right_iterations, right_min, right_max = sorting(lst[mid:], ascending)

    merged = []
    i = j = 0
    iterations = 0

    while i < len(left) and j < len(right):
        if ascending:
            if left[i] <= right[j]:
                merged.append(left[i])
                i = i+1
            else:
                merged.append(right[j])
                j = j+1
        else:
            if left[i] >= right[j]:
                merged.append(left[i])
```

```
                i = i+1
            else:
                merged.append(right[j])
                j = j+1
        iterations = iterations +1

    merged = merged + left[i:]
    merged = merged + right[j:]

    return merged, left_iterations + right_iterations + iterations, min(left_min,
right_min), max(left_max, right_max)
```

Fig. 7 - Implementação da função sorting

O objetivo da função é ordenar uma lista **'lst'** em ordem crescente ou decrescente, mediante o valor do parâmetro **'ascending'** seja True ou False, respetivamente.

Como se trata de um algoritmo recursivo, o primeiro passo é estabelecer o caso base, que neste caso é verificar se a lista é de tamanho 1 ou menor, significando que já está ordenada e será devolvida com o número de iterações igual a 0 e os valores máximos e mínimos iguais ao único elemento da lista.

```
    if len(lst) <= 1:
        return lst, 0, lst[0], lst[0]
```

Fig. 8 - Parte da função sorting (caso base)

Caso a lista não seja de tamanho 1 ou inferior, a função é chamada de forma recursiva em ambas as metades da lista, obtendo-se a lista ordenada da metade esquerda e direita, tal como o número de iterações necessárias para ordenar cada metade e os seus valores mínimos e máximos. A variável **'mid'** indica a posição do meio da lista e é o ponto de referência para o fim da primeira metade e início da segunda metade, conforme se pode ver na figura 9.

```
    mid = len(lst) // 2
    left, left_iterations, left_min, left_max = sorting(lst[:mid], ascending)
    right, right_iterations, right_min, right_max = sorting(lst[mid:], ascending)
```

Fig. 9 - Parte da função sorting (recursividade)

A lista ordenada **‘merged’** é obtida ao juntar as duas metades ordenadas. De seguida, é utilizada uma estrutura de repetição **‘while’** para iterar sobre as posições **‘left’** e **‘right’**, estabelecendo a comparação entre o menor valor de cada uma delas, com base na variável **‘ascending’**, que indica se a lista deve ser ordenada em ordem crescente ou decrescente.

Para a ordem crescente, o algoritmo verifica se o elemento na posição **‘i’** da lista **‘left’** é menor ou igual ao elemento na posição **‘j’** da lista **‘right’** e, caso isso se verifique, o elemento é adicionado à lista **‘merged’** e o índice **‘i’** recebe um incremento de 1. Caso contrário, é o elemento na posição **‘j’** da lista **‘right’** que é adicionado à lista **‘merged’** e o índice **‘j’** que recebe o incremento de 1. Para a ordem decrescente, a lógica é a mesma de forma invertida. A variável **‘iterations’** recebe sempre um incremento de 1, para contabilizar cada iteração realizada pelo algoritmo.

```
    merged = []
    i = j = 0
    iterations = 0

    while i < len(left) and j < len(right):
        if ascending:
            if left[i] <= right[j]:
                merged.append(left[i])
                i = i+1
            else:
                merged.append(right[j])
                j = j+1
        else:
            if left[i] >= right[j]:
                merged.append(left[i])
                i = i+1
            else:
```

```
                merged.append(right[j])
                j = j+1
        iterations = iterations +1
```

Fig. 10 - Parte da função sorting (ordenação)

Por fim, caso a lista **‘left’** ou **‘right’** tenha sido totalmente percorrida, os restantes elementos da outra lista são adicionados à lista **‘merged’**. No final, é devolvida a lista ordenada, o número de iterações, o valor mínimo e máximo.

```
    merged = merged + left[i:]
    merged = merged + right[j:]

    return merged, left_iterations + right_iterations + iterations, min(left_min,
right_min), max(left_max, right_max)
```

Fig. 11 - Parte da função sorting (final)

c) Função execução

A função **‘execute’** executa o algoritmo de ordenação merge sort em partições de um arquivo CSV com o caminho em **‘data’**, onde cada partição apresenta **‘n’** observações. O argumento **‘ascending’** indica se a ordenação deve ser crescente ou decrescente. Por outras palavras, a função **‘execute’** permite a aplicação das funções desenvolvidas nas alíneas a) e b).

```
    def execute(data, n, ascending):
        results = []

        for i in partitions(data, n):
            start_time = time.time()

            partition_data = i[:]

            partition_numbers = [[float(x) for x in row][0] for row in
                                 partition_data]
```

```
            sorted_partition, iterations, min_val, max_val = sorting(
                                        partition_numbers, ascending)
            end_time = time.time()
            execution_time = end_time - start_time

            partition_info = {
                'Tempo de execução': execution_time,
                'Número de iterações': iterations,
                'Valor mínimo': min_val,
                'Valor máximo': max_val
            }

            results.append(partition_info)

        results_df = pd.DataFrame(results)

        return results_df
```

Fig. 12 - Implementação da função execute

Para começar, a variável **'results'** irá armazenar os resultados de cada partição. De seguida, a função **'partitions'** é chamada para iterar sobre as várias partições do ficheiro e o seu conteúdo é armazenado na variável **'i'**.

```
        results = []

        for i in partitions(data, n):
```

Fig. 13 - Parte da função execute (loop inicial)

Para cada partição iterada e para uma melhor noção do desempenho geral, começa-se por medir o tempo, utilizando a função **'time.time()'**, sendo este valor armazenado na variável **'start_time'**; e a partição atual guardada na variável **'partition_data'**. De seguida, transformou-se cada partição em listas de valores numéricos, em vez de listas de listas de strings, com recurso a uma list comprehension, para se poder trabalhar com os dados numéricos e

aplicar o algoritmo de ordenação desenvolvido. Estes valores foram armazenados na variável **'partition_numbers'**.

```
            start_time = time.time()

            partition_data = i[:]

            partition_numbers = [[float(x) for x in row][0] for row in
                                 partition_data]
```

Fig. 14 - Parte da função execute (primeiras medições e transformações)

Após os passos acima descritos, estávamos em condições para aplicar o algoritmo de merge sort desenvolvido na alínea b), com a função de **'sorting'**, e guardar a lista ordenada, o número de iterações, o valor mínimo e máximo dessa lista nas variáveis **'sorted_partition'**, **'iterations'**, **'min_val'** e **'max_val'**, respetivamente. O tempo de execução foi medido novamente com recurso à função **'time.time()'** e guardado em **'end_time'** e, para obter o tempo de execução total, fez-se a diferença entre o **'end_time'** e o **'start_time'**, armazenando-se o resultado na variável **'execution_time'**.

```
            sorted_partition, iterations, min_val, max_val = sorting(
                                        partition_numbers, ascending)
            end_time = time.time()
            execution_time = end_time - start_time
```

Fig. 15 - Parte da função execute (ordenação)

De seguida, criou-se um dicionário com a informação para o **'Tempo de execução'**, **'Número de iterações'**, **'Valor mínimo'** e **'Valor máximo'**, utilizando as variáveis descritas no parágrafo anterior, que é adicionado à lista de resultados **'results'** para cada partição iterada. Para finalizar, é criado e devolvido um DataFrame com os resultados obtidos utilizando a função **'pd.DataFrame'** da biblioteca pandas.

```
            partition_info = {
                'Tempo de execução': execution_time,
                'Número de iterações': iterations,
                'Valor mínimo': min_val,
                'Valor máximo': max_val
            }

            results.append(partition_info)

        results_df = pd.DataFrame(results)

        return results_df
```

Fig. 15 - Parte da função execute (dicionário e DataFrame)

d) Experiências

Para a elaboração das experiências, começou-se por criar uma lista **'test_values'** com os diferentes valores considerados para o **'n'** (*esta lista foi atualizada sempre foi feito um novo gráfico, conforme se pode ver nos gráficos que se seguem, para procurar o valor que minimiza o tempo de execução da função*) e uma lista vazia **'times'** onde seriam armazenados os respectivos tempos de execução para cada valor a testar.

```
    test_values = [500, 1000, 2500, 4000, 5000]
    times = []
```

Fig. 16 - Variáveis criadas para a realização das experiências

O **'for'** loop irá percorrer a lista **'test_values'**, para testar as diferentes opções para o **'n'**, que aqui aparece representado como **'i'**. De seguida, como o output da função **'execute'** é um DataFrame, calculou-se o tempo de execução somando os valores do **'Tempo de execução'** por partição e guardou-se o valor na variável criada para o efeito **'times'**.

```
for i in test_values:
    result = execute("data.csv", i, True)
    total_time = result['Tempo de execução'].sum()
    times.append(total_time)
```

Fig. 17 - Loop para testar os diferentes valores para 'n' e guardar os tempos de execução

Por fim, com recurso à biblioteca **'matplotlib'** desenvolveu-se um gráfico que mostra os tempos de execução para cada valor de **'n'** em **'test_values'**.

```
plt.plot(test_values, times)
plt.title('Tempo de execução para diferentes valores de n')
plt.xlabel('n')
plt.ylabel('Tempo (s)')
plt.show()
```

Fig. 18 - Implementação dos gráficos com recurso ao matplotlib

Para testar o valor de **'n'** que minimiza o tempo de execução da função, como nos encontramos perante um CSV com 100 000 observações, começámos por testar valores de maior dimensão. A primeira experiência variou entre 5000 e 25000 nos valores que o **'n'** pode assumir e aqui pôde ver-se que os valores mais próximos de 5000 são os que apresentam menores tempos de execução.

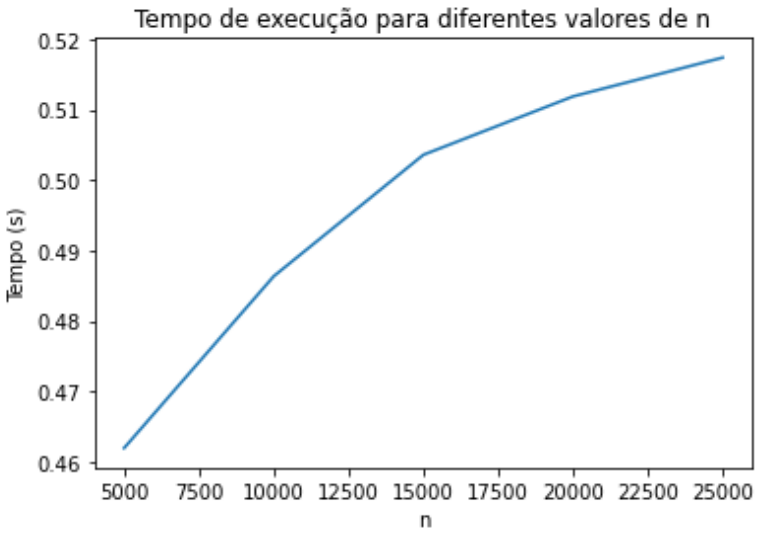

Fig. 19 - Tempo de execução para diferentes valores de n (entre 5000 e 25000)

Como os valores que minimizam o tempo de execução são os menores possíveis que foram colocados anteriormente, optou-se por reduzir os limites do intervalo que agora varia entre 500 e 5000. Ainda que tenha havido o ajuste no intervalo, o padrão acima registado manteve-se e pode ver-se que as opções que minimizam o tempo de execução da função são, tal como anteriormente, as que estão próximas do menor valor colocado - 500 - conforme se pode ver na figura abaixo.

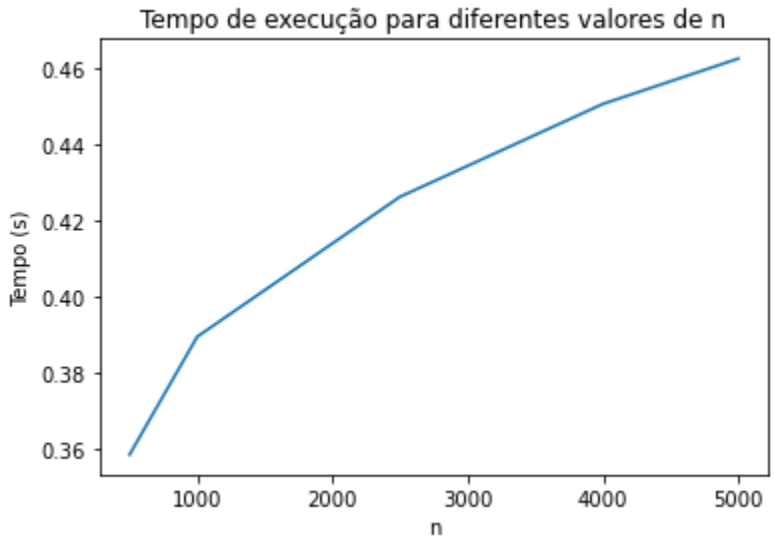

Fig. 20 - Tempo de execução para diferentes valores de n (entre 500 e 5000)

Seguindo a dinâmica anterior, optou-se por reduzir novamente o intervalo de valores possíveis que o 'n' pode tomar para os valores entre 1 e 500. E, tal como esperado, o menor valor possível que o 'n' pode assumir (1) foi o que apresentou o melhor tempo de execução.

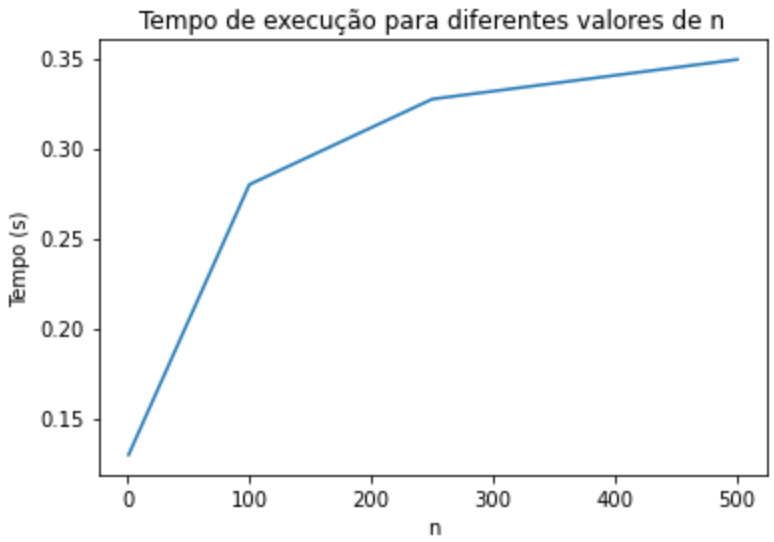

Fig. 21 - Tempo de execução para diferentes valores de n (entre 1 e 500)

Conforme foi possível observar, de uma forma geral, as curvas apresentam um crescimento próximo ao de uma função logarítmica, ou seja, quanto maior o tamanho da entrada, menor é o aumento no tempo de execução em relação ao tamanho da entrada.

Isto pode ser explicado pelo facto de o algoritmo de ordenação utilizado ser o Merge Sort e este ter uma complexidade temporal de O(n log n), o que significa que o tempo de execução aumenta proporcionalmente ao tamanho da entrada multiplicado pelo logaritmo desse tamanho. Isso faz com que, para entradas muito grandes, o tempo de execução não cresça exponencialmente, tornando este algoritmo, conforme já descrito acima, como uma opção bastante eficiente em relação a outros.

Sabendo que o 1 é o número de **'n'** que minimiza o tempo de execução da função pode encontrar-se o mínimo e o máximo do conjunto de valores do CSV executando a função da alínea c) conforme mostra a figura abaixo, obtendo o valor mínimo da coluna 'Valor mínimo' do DataFrame e o valor máximo da coluna 'Valor máximo' do DataFrame.

```
a = execute("data.csv", 1, True)
print("Mínimo:", a['Valor mínimo'].min())
print("Máximo:", a['Valor máximo'].max())
```

Fig. 22 - Valores mínimos e máximos do conjunto

Desta forma pode ser concluído que:

- Mínimo: **-9999841.0**
- Máximo: **9999991.0**