INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE
CAMPUS PASSO FUNDO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO TÉCNICO - ESTRUTURA DE DADOS III:

QUEM É O IMPOSTO?

DARLAN NOETZOLD
HURIEL FERREIRA LOPES
JAKELYNY SOUSA DE ARAÚJO

PASSO FUNDO - RS
2022

RESUMO

O presente trabalho tem como objetivo revelar as táticas utilizadas por um possível impostor que ameaça a continuidade da exploração espacial do povo Sborniano no planeta X9. Toda a tripulação de dez astronautas foi dizimada e os destroços encontrados no planeta hostil serviram como base para informações importantes acima do caso de sabotagem. Todo o povo Sborniano segue aguardando com anseio pelos resultados que aqui serão revelados.

Os bravos agentes infiltrados no planeta X9 descobriram ações suspeitas acerca dos logs de tarefas da nave. Após diversas tentativas e falhas para organizar os eventos dos usuários em ordem de logs e ordenando seus meses, foi definido que o usuário da tarefa 1000001º deveria ser investigado de forma mais detalhada. Acompanhe o restante do relatório para juntos podermos ajudar o povo Sborniano a continuar sua jornada de exploração.

Palavras-chave: Sbornia; impostor; logs.

SUMÁRIO

1. INTRODUÇÃO DO ASSUNTO

Inicialmente, é preciso ver o que foi feito anteriormente, onde o problema existente era a ordenação de dados massivos para encontrar o impostor que sabotou a nave Sborniana. O impostor foi encontrado usando o eficiente algoritmo de Counting Sort estável, sem o uso de operadores de comparação (por uma limitação dos computadores da nave). No script criado, os logs da nave foram divididos em meses, reduzindo a quantidade de logs para ordenar e consequentemente o tempo de execução do algoritmo.

Porém a paz na República da Sbórnia ainda não é uma realidade alcançada e mais um problema computacional surge. O problema que precisa de uma solução imediata está relacionado a novos algoritmos de busca e melhores segregações de logs para catalogação de registros suspeitos. O próximo capítulo terá uma apresentação mais detalhada da problemática desta nova solicitação da República da Sbórnia.

2. DELIMITAÇÃO DO TEMA, PROBLEMA, COM OBJETIVOS

O problema encontrado é acerca dos algoritmos de busca utilizados, onde atualmente são feitos de forma sequencial e, portanto, pouco eficiente. A partir desta limitação dos sistemas governamentais, existe uma solicitação de melhora para que seja possível encontrar registros de logs de operações que foram armazenadas como suspeitas ou que devam ser recuperadas.
Assim, o presente trabalho tem como objetivo a elaboração de um algoritmo que não tenha um tempo O(n). Desta forma, as técnicas selecionadas para o melhoramento dos scripts utilizados atualmente são: espalhamento de dados com tabelas hash e um algoritmos de busca melhor que o sequencial utilizado no momento atual.

12Os algoritmos de busca escolhidos são a Busca Binária e a Busca Interpolada. Será realizado uma análise entre os dois algoritmos para a escolha do mais eficiente.

3. CONTEXTUALIZAÇÃO BIBLIOGRÁFICA (NO CONTEXTO DA DISCIPLINA)

Este capítulo tem por objetivo apontar os algoritmos estudados e apontados para a resolução do problema proposto. Serão tratados assim três principais tópicos: Busca Binária, Busca Interpolada e Tabela Hash.

3.1. BUSCA BINÁRIA

É um algoritmo de busca que segue o paradigma de divisão e conquista. Ela parte do pressuposto de que o vetor está ordenado e realiza sucessivas divisões do espaço de busca, comparando o elemento buscado com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E, finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor. A complexidade desse algoritmo é da ordem de O(log2n), em que n é o tamanho do vetor de busca (FEOFILOFF, 1998).

3.2. BUSCA INTERPOLADA

A pesquisa por interpolação é um algoritmo para pesquisar uma chave em uma matriz que foi ordenada por valores numéricos atribuídos às chaves (valores de chave). Em cada etapa, o algoritmo calcula onde no espaço de busca restante o item procurado pode estar, com base nos valores da chave, nos limites do espaço de busca e no valor da chave procurada, geralmente por meio de uma interpolação linear. O valor-chave realmente encontrado nessa posição estimada é então comparado ao valor-chave que está sendo procurado. Se não for igual, dependendo da comparação, o espaço de pesquisa restante será reduzido à parte anterior ou posterior à posição

estimada. Este método só funcionará se os cálculos sobre o tamanho das diferenças entre os valores de chave forem sensatos. A complexidade deste algoritmo é log(log(n)), em que n é o tamanho do vetor (Graham, 1879).

3.3. TABELA DE HASH

As tabelas de hash são um método eficiente de armazenar um pequeno número de inteiros de um grande intervalo. Uma tabela de hash usa uma função de hash para calcular um índice, também chamado de código de hash, em uma matriz de buckets ou slots, a partir dos quais o valor desejado pode ser encontrado. Durante a pesquisa, a chave é criptografada e o hash resultante indica onde o valor correspondente está armazenado (MORIN, Pat).

Em uma tabela de hash bem dimensionada, a complexidade de tempo médio para cada pesquisa é independente do número de elementos armazenados na tabela. Muitos designs de tabela de hash também permitem inserções e exclusões arbitrárias de pares chave-valor, a um custo médio constante amortizado por operação (MORIN, s.d.).

O hashing é um exemplo de troca de espaço-tempo. Se a memória for infinita, a chave inteira pode ser usada diretamente como um índice para localizar seu valor com um único acesso à memória. Por outro lado, se o tempo infinito estiver disponível, os valores podem ser armazenados independentemente de suas chaves, e uma pesquisa binária ou linear pode ser usada para recuperar o elemento (MORIN, s.d.).

4. TECNOLOGIAS

As tecnologias utilizadas para a resolução do problema apresentado estão envolvidas estritamente com o Python. A escolha da linguagem se deu pela predisposição da mesma de ser usada para scripts dedutíveis e curtos, além de ser muito utilizada para o processamento de dados massivos e custosos. Desta forma, por mais que o Python não alcance a eficiência do C ele consegue chegar muito próximo e seu código é mais facilmente mantido e manutenível.

As bibliotecas utilizadas no script foram a time, para calcular a complexidade e o tempo levado de busca, e a biblioteca json para a importação dos dados coletados.

5. SOLUÇÕES PROPOSTAS

A solução desenvolvida para toda a problemática apresentada foi a criação de um script analisando as duas buscas propostas, assim como um espalhamento de dados com tabela de hash por divisão. Neste código, primeiramente seriam consumidos os dados para que o usuário digite o número dos logs a analisar, qual algoritmo de busca ele quer usar e se ele quer ver os logs completos ou apenas do mês do impostor. Após a atuação do usuário, será feito o espalhamento dos dados com o hash table e então, com os dados espalhados, será feita a busca com o algoritmo escolhido.

Existem alguns pontos encontrados durante o desenvolvimento que precisam ser analisados. Primeiramente, o algoritmo de Busca Binária precisa ser normalizado antes de ser executado, pois se no meio da lista existir um conjunto de dados vazios ele vai retornar um erro. Portanto, para os logs do mês do impostor, foi feita uma normalização ao enviar os dados para o algoritmo de Busca Binária. Outro ponto a ser analisado é a forma que os dados serão trabalhados em ambos algoritmos de busca, em que será usado uma matriz de dicionário. Este formato é necessário para conseguir armazenar a saída do hash table e manter as informações dos logs.

Ambas buscas foram testadas com os dados completos e do mês do impostor, e ambos obtiveram resultados aceitáveis. Tanto os resultados da

Busca Binária quanto da Busca Interpolada serão apresentados no próximo capítulo.

Segue um print screen do algoritmo de Busca Binária desenvolvido:

```
282  def busca_binaria(valor, vetor):
283      esquerda, direita = 0, len(vetor) - 1
284      while esquerda <= direita:
285          meio = (esquerda + direita) // 2
286          if vetor[meio][0].get("log") == valor:
287              return vetor[meio]
288          elif vetor[meio][0].get("log") > valor:
289              direita = meio - 1
290          else:  # A[meio] < item
291              esquerda = meio + 1
292      return -1
293
```

Neste trecho é possível visualizar a função que recebe como parâmetro o valor (log) a ser buscado e o vetor de busca e retorna os dados do valor buscado. Esta lógica é feita através da divisão de responsabilidades, onde a busca é dividida em duas frentes para ser mais ágil e consumir menos tempo.

Agora, uma análise mais precisa do algoritmos de Busca Interpolada usado no trabalho:

```
253  def interpolation_search(array, x):
254      low = 0
255      array_low = array[low][0].get("log")
256      aux = 1
257      while True:
258          try:
259              high = len(array) - aux
260              array_high = array[high][0].get("log")
261              break
262          except Exception:
263              aux += 1
264
265      while (low <= high) and (x >= array_low) and (x <= array_high):
266          if len(array[low]) == 0 or len(array[high]) == 0: continue
267          array_low = array[low][0].get("log")
268          array_high = array[high][0].get("log")
269          pos = int(low + ((high - low) / (array_high - array_low)) * (x - array_low))
270          if len(array[pos]) == 0: continue
271          if array[pos][0].get("log") < x:
272              low = pos+1
273
274          elif array[pos][0].get("log") > x:
275              high = pos-1
276
277          else:
278              return array[pos]
279
280      return -1
281
```

Neste algoritmo é possível notar como que a busca interpolada consegue ser mais eficiente que a binária, isto, através do uso de um cálculo

de onde no espaço de busca restante o item procurado pode estar, com base nos valores da chave, nos limites do espaço de busca e no valor da chave procurada. E, após isso, ele retorna os dados encontrados do log, se ele estiver no vetor.

Além disso, existem algumas normalizações dos dados que foram implementadas em ambos os algoritmos de busca. No script de interpolação foi feito uma validação do tamanho do vetor, da seguinte forma:

```
257     while True:
258         try:
259             high = len(array) - aux
260             array_high = array[high][0].get("log")
261             break
262         except Exception:
263             aux += 1
264
```

Já na binária é feito uma validação para impedir logs vazios no vetor, isto é feito desta maneira:

```
369   def normalize_data(hash_table):
370       for i in hash_table:
371           if len(i) == 0:
372               hash_table.remove(i)
373
374       return hash_table
375
```

O HashTable é feito com o algoritmo de divisão:

```
294   def hash_by_division(input_array):
295       print("Criando o hash table...")
296       hash_table = [[] for _ in range(100000)]
297
298       def _hashing(keyvalue):
299           return keyvalue % len(hash_table)
300
301       def _insert(hash_table, keyvalue, value):
302           i = _hashing(keyvalue)
303
304           hash_table[i].append(value)
305           return i
306
307       for i in input_array:
308           _insert(hash_table, i.get("log"), i)
309
310       return hash_table
```

Nesse tipo de função, a chave é interpretada como um valor numérico que é dividido por um valor N. O resto dessa divisão inteira, um valor entre 0 e N-1, é considerado o endereço em uma tabela de N posições.

E por último, existem alguns pontos a serem adicionados. Então, é preciso acrescentar que os dados que entram nos algoritmos de busca são previamente espalhados pela função de hash table por divisão e normalizados. E, por último, vão ser visualizados através da seguinte função:

```
358  def display_hash(hash_table):
359      if hash_table == -1:
360          print("Valor não existe!")
361          return
362      for i in hash_table:
363          print(i.get('user'), end=" ")
364          print("-->", end=" ")
365          print(i, end=" ")
366
367          print()
```

6. RESULTADOS

Para analisar os resultados encontrados nos teste feitos foi usado o critério de tempo calculado para a execução de cada algoritmo. Para a execução do script foi usado um Notebook Acer, com um processador Intel i5 de sétima geração, uma placa gráfica integrada NVIDIA 940mx, 20 GB de memória RAM e um SSD de 500 GB. O script foi rodado no python 3.8 na IDE Pycharm da JetBrains.

Com essas especificações os resultados dos testes com todos os dados foram os seguintes:

Todos os Dados	Busca Binária	Busca Interpolada
Tempo de execução da Busca(s)	0.0041182041168	0.0009875297
Tempo de criação do Hash Table(s)	5.43	

Já os testes com os dados do mês do impostor tiveram os seguintes resultados:

Dados dos Meses	Busca Binária	Busca Interpolada

Tempo de execução da Busca(s)	0.0	0.0
Tempo de criação do Hash Table(s)	0.1585764	

7. REFERÊNCIAS

FEOFILOFF, Paulo. **Busca em vetor ordenado**. Internet Archive, 1998. Disponível em:

<https://web.archive.org/web/20050911173556/http://www.ime.usp.br/~pf/algoritmos/aulas/bubi2.html>. Acesso em: 15 jul. 2022.

Graham, S. L. **interpolation SearchA Log LogN Search**, ACM, v 21, n 7, julho 1879. Disponível em: <https://www.cs.technion.ac.il/~itai/publications/Algorithms/p550-perl.pdf>. Acesso em 16 jul. 2022.

MORIN, Pat. **Hash Tables**. Open Data Structures. Disponível em: http://opendatastructures.org/versions/edition-0.1e/ods-java/5_Hash_Tables.html. Acesso em: 16 jul. 2022.