UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

PROJETO 2

Algoritmos Para Streams de Dados

Gabriela Correia Holanda Freitas (gchf)
Rodrigo de Lima Oliveira (rlo)

RECIFE, 22 DE AGOSTO DE 2021
Professores: Nivan Roberto Ferreira Junior , Paulo Fonseca

Sumário

Seção 1. Apresentação:
O projeto se trata da implementação de dois algoritmos para dados ordenados. Os algoritmos escolhidos para a implementação foram o Q-digest e o GK. Gabriela ficou responsável pela implementação do Q-digest e Rodrigo do GK.

Seção 2. Q-digest

Seção 2.1 Implementação

A estrutura do código foi baseada no código feito em sala de aula. Foram implementadas duas classes: Node e Q_digest. A primeira é para a implementação dos nós da árvore e a segunda para a implementação dos métodos do algoritmo Q-digest. A implementação das duas teve muita influência do pseudo-código e do código python apresentados em aula. Acrescentado ao que foi implementado em aula, o método rank_element() também foi implementado e utilizado. Sua função é receber um valor de rank e retornar uma aproximação de um elemento que possui esse rank na árvore, e ele é utilizado para a pesquisa por quantile.

```
def rank_element(self, rank):
    v = self.root
    l, r = 0, self.U
    while v and (r-l) > 1:
        mid = (l+r)//2
        # se a soma dos pesos da esquerda for menor que o rank, vai pra a direita
        # se nao, vai pra a esquerda
        if v.child[0]:  # se existe filho à esquerda
            # peso da arvore do filho à esquerda
            u = v.child[0].tree_weight()
            if u < rank:
                if v.child[1]:
                    v = v.child[1]
                    rank -= u  # diminui do rank o q já foi visto
                    l = mid
                else:  # se u < rank mas n tiver filho na direita
                    return mid
            else:
                v = v.child[0]
                r = mid
        elif v.child[1]:  # se n existe nada à esquerda, só vai pra a direita e o resto continua igual
            v = v.child[1]
            l = mid
        else:
            return mid
```

Na função main há a validação da entrada e a atribuição dos valores iniciais. Em seguida há o update dos valores na árvore e então são feitas as queries.
Para aprender a fazer o parsing dos argumentos de linha de comando, foi estudado o tutorial [3].
O tempo de execução do algoritmo foi medido utilizando a biblioteca time e a sua função time().

Seção 2.2 Testes

Os testes foram aplicados aos dados enviados juntamente com a especificação do projeto, de nome network_flows.csv, e estes foram os resultados encontrados:

Para testar a funcionalidade da função update e a função rank, foi utilizado o mesmo método apresentado em sala de aula, com uma stream de tamanho 10, de valores aleatórios, dentro de um universo de tamanho 16. Os ranks verdadeiros foram calculados de forma objetiva, então puderam ser comparados aos dados obtidos pelo método rank() dentro da classe Q_digest. Tudo da mesma forma que foi feito em sala de aula, com resultados equivalentes, dentro de sua aleatoriedade. As árvores finais foram printadas usando os métodos print_tree() e _print(), também apresentados em sala de aula, e isso facilitou para checar que o método de update() também está dentro dos padrões apresentados na aula.

Serão apresentadas imagens adquiridas ao testar os métodos rank() e rank_element(), sendo o rank_element() usado para a implementação da condição quant e o rank() para a condição rank, estabelecidas nos comandos iniciais do processo.

Os testes do 1 ao 4 foram realizados sob a árvore apenas com updates normais, sem o uso do método compress() para compressão e otimização de memória.

Aqui foram introduzidos valores de entrada x para a busca de rank, e depois esses valores de rank foram utilizados no método rank_element() para retornarem uma aproximação para o valor de x.
Concluiu-se por esses dados e repetição de testes similares que as funções estavam corretas e com aproximações satisfatórias.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv rank --in entrada.txt
2066317
x original =  10
x =  9
2683413
x original =  20
x =  19
2856893
x original =  30
x =  29
3037886
x original =  40
x =  39
3094982
x original =  50
x =  46
3098470
x original =  60
x =  54
3207386
x original =  70
x =  65
3207386
x original =  80
x =  65
3207386
x original =  90
x =  65
3235432
x original =  100
x =  88
3235432
x original =  110
x =  88
3235432
x original =  120
x =  88
3321206
x original =  130
x =  124
```

(Teste 1)

Abaixo foram utilizados dados de entrada q para executar a condição quant, e estes foram multiplicados pelo peso total da árvore após todos os updates, e este valor resultante foi usado como argumento para o método rank_element() para retornar o elemento do rank da multiplicação q*total_weight.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv quant --in entrada.txt
q =  0.0
q*total_weight =  0.0
1
q =  0.1
q*total_weight =  353938.10000000003
1
q =  0.2
q*total_weight =  707876.2000000001
3
q =  0.25
q*total_weight =  884845.25
3
q =  0.3
q*total_weight =  1061814.3
3
q =  0.4
q*total_weight =  1415752.4000000001
5
q =  0.5
q*total_weight =  1769690.5
7
q =  0.6
q*total_weight =  2123628.6
11
q =  0.7
q*total_weight =  2477566.6999999997
15
q =  0.75
q*total_weight =  2654535.75
19
q =  0.8
q*total_weight =  2831504.8000000003
27
q =  0.9
q*total_weight =  3185442.9
62
q =  1.0
q*total_weight =  3539381.0
896
```

(Teste 2)

Abaixo foram utilizados os mesmos valores do exemplo anterior, porém estes foram introduzidos junto da entrada, e não no arquivo de entrada in.
Concluindo que as duas formas de entrada estão implementadas corretamente.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv quant 0 0.1 0.2 0.25 0.3 0.4 0.5 0.6 0.7 0.75 0.8 0.9 1
1
1
3
3
3
5
7
11
15
19
27
62
896
```

(Teste 3)

Os elementos encontrados nos dois testes anteriores foram utilizados como entrada para o método rank() e comparados com os valores iniciais dos dados q*total_weight, que foram utilizados como ranks para o rank_element() no **(Teste 2)**.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv rank 1 1 3 3 3 5 7 11 15 19 27 62 896
0
0
1039418
1039418
1039418
1401006
1672737
2119690
2380241
2609762
2808515
3098470
3474379
```

(Teste 4)

Os mesmos valores usados para a condição quant nos **(Testes 2 e 3)** foram repetidos, porém agora com uma árvore comprimida com o uso do método compress()

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv quant 0 0.1 0.2 0.25 0.3 0.4 0.5 0.6 0.7 0.75 0.8 0.9 1
1
1
3
3
3
5
7
11
15
19
27
60
768
```

(Teste 5)

Os valores resultantes dos **(Testes 2 e 3)** também foram utilizados na condição rank, com árvore comprimida, para serem comparados aos valores da árvore sem compressão.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv rank 1 1 3 3 3 5 7 11 15 19 27 62 896
0
0
1039418
1039418
1039418
1401006
1672737
2119690
2380240
2609758
2808508
3098443
3474361
```

(Teste 6)

Alguns resultados foram diferentes nos testes de árvore comprimida, porém todos similares e satisfatórios. A perda de informação se mostrou irrelevante diante do benefício de memória.

Para testes de tempo foi utilizada um arquivo de query de tamanho 9 e o tempo final para a execução da main foi de 23.37s.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv rank --in entrada.txt
3474361
3474361
3474361
3474361
3474361
3402351
3401779
3326928
3235414
tempo =  23.37303614616394
```

(Teste 7)

Foi constatado que o tempo de execução faz-se relevante apenas no update, pois as queries em si não fazem diferença significativa. Foi feito um teste com 1000 entradas de query e o tempo deu 24.64s, em comparação com o de nove entradas do **(Teste 7)** que foi de 23.37s. Aumentar o universo, no entanto, aumenta o tempo. Para um universo de tamanho 100000 o tempo deu 44.66s.
Essas conclusões estão dentro do esperado, pois os métodos da classe Q_digest agem sobre um universo de tamanho n dividindo-o pela metade a cada iteração, de forma logarítimica, e então sabe-se que o tempo depende do tamanho do universo.

```
C:\Users\gabri\OneDrive\Área de Trabalho\Algoritmos\Projeto>python qdig.py --val 4 --eps 0.1 --univ 1000 network_flows.csv quant --in entrada.txt
1
tempo =  24.131754636764526
```

(Teste 8)

Seção 3. GK

Seção 3.1 Implementação

A estrutura do código foi baseada na aula a respeito do GK, em [1] e no artigo [2].
O código é dividido basicamente em:

Tuples - Entidade que representa um entrada no GK (x, g, delta)

```
class Tuple():
   def __init__(self, val, g, delta):
       self.value  = val
       self.g = g
       self.delta = delta

   def __repr__(self):
       return 'Value: {} - ( g: {}, delta: {} )'.format(self.value, self.g,
self.delta)
```

Update - Parte responsável por fazer a inserção de uma nova entrada, avaliar a criação de uma nova posição ou não e quando necessário chamar o método compress.

```
  def update(self, value):
        self._count += 1
        if not self.entries:
            self.entries.append(Tuple(value,1,0))
        else:
            self.entries = sorted(self.entries, key = lambda x: x.value)
            idx = 0
            for i, entry in enumerate(self.entries):
                if value < entry.value:
                    idx = i
                    break
            delta = 0
            #for first and last index - delta is  0

            delta = self.entries[idx].g + self.entries[idx].delta - 1
            # sanity check, if condition is true something went wrong. this should
never happen
            if delta > int(math.floor(self.eps * float(self._count*2))):
                print("Delta is greater than allowable error. This never should
happen. Really.")
            xi = self.entries[idx]
            if (xi.g + xi.delta + 1) < (2 * self.eps * self._count):
                self.entries[idx].g+=1
            else:
                self.entries.append(Tuple(value,1,delta))
                self.entries = sorted(self.entries, key = lambda x: x.value)

            if value < self._min:
                self._min = value
            if value > self._max:
                self._max = value
            if self._count % self._compress_threshold == 0:
                    self.compress()
```

Um ponto que vale ressaltar, é que no pseudo código apresentado no livro texto [1], o compress parece ser chamado sempre que um elemento é adicionado. Contudo, baseado em [2], foi criado um compress_threshold para decidir se devemos ou não executar o compress.

Compress - Realiza a compressão de valores que atendem ao critério. Empilhando assim 2 posições imediatamente seguintes em um único valor.

```
def compress(self):
        remove_threshold = float(2.0 * self.eps * (self._count - 1))
        i = 0
        j = 1
        n_entries = len(self.entries) - 1
        while i < n_entries:
            xi = self.entries[i]
            xj = self.entries[j]
            if xi.g + xj.g + xj.delta <= remove_threshold:
                self.entries[j].g += self.entries[i].g
                self.entries.pop(i)
                break
            i+=1
            j+=1
        self.entries = sorted(self.entries, key = lambda x: x.value)
```

Quantile - Retorna o valor do elemento correspondente ao rank desejado

```
def quantile(self, q):
        if not (0 <= q <= 1):
            raise ValueError("q must be a value in [0, 1].")

        if self._count == 0:
            raise ValueError("GK sketch does not contain values.")

        rank = int(q * (self._count))
        spread = int(self.eps * (self._count - 1))
        g_sum = 0.0
        i = 0

        n_entries = len(self.entries)
        while i < n_entries:
            g_sum += self.entries[i].g
            if g_sum + self.entries[i].delta > rank + spread:
                break
            i += 1
        if i == 0:
            return self._min
        return self.entries[i - 1].value
```

Rank - Estima o rank para um dado elemento. Caso o elemento esteja no universo será retornado a estimativa do seu rank, e caso contrário será retornado o rank máximo no scketch.

```
def rank(self, value):
        g_sum = 0
        idx = 0
        for i, entry in enumerate(self.entries):
            if entry.value == value:
                idx = i
        if idx == 0:
            return self._max
        i = idx - 1
        j = 0
        while j <= i:
            g_sum+=self.entries[j].g
            j+=1
        rank = g_sum - 1 + (self.entries[idx].g + self.entries[idx].delta)/2
        return rank
```

Seção 3.2 Testes

Foram realizados testes utilizando como referência (Para comparação) os mesmo exemplos encontrados no livro texto [1]

(TESTE 1 - BASE LIVRO)

eps = 1/5

Stream utilizada - [1, 4, 2, 8, 5, 7, 6, 7, 6, 7, 2, 1]

GK Summary

Value: 2 - (g: 3, delta: 0)
Value: 5 - (g: 3, delta: 0)
Value: 6 - (g: 1, delta: 1)
Value: 6 - (g: 1, delta: 2)
Value: 7 - (g: 1, delta: 2)
Value: 8 - (g: 3, delta: 0)

Consulta rank - [1,2,4,5,6,7,8]

Query: 1 ,Rank: 8, Real_rank: 0, error: 8
Query: 4 ,Rank: 8, Real_rank: 4, error: 4
Query: 2 ,Rank: 8, Real_rank: 2, error: 6
Query: 8 ,Rank: 9.5, Real_rank: 11, error: 1.5
Query: 5 ,Rank: 3.5, Real_rank: 5, error: 1.5
Query: 7 ,Rank: 8.5, Real_rank: 8, error: 0.5
Query: 6 ,Rank: 7.5, Real_rank: 6, error: 1.5
Query: 7 ,Rank: 8.5, Real_rank: 8, error: 0.5
Query: 6 ,Rank: 7.5, Real_rank: 6, error: 1.5
Query: 7 ,Rank: 8.5, Real_rank: 8, error: 0.5
Query: 2 ,Rank: 8, Real_rank: 2, error: 6
Query: 1 , Rank: 8, Real_rank: 0, error: 8

Como esperado para os valores que estavam no summary o erro foi dentro do esperado. Para os que não estavam, o erro foi mais alto. Devido a abordagem de retornar o max_rank nesses casos.

Consulta quantile

Query quantile: 0.0,result: 1
Query quantile: 0.1,result: 2
Query quantile: 0.2,result: 2
Query quantile: 0.30000000000000004,result: 2
Query quantile: 0.4,result: 5
Query quantile: 0.5,result: 6
Query quantile: 0.6000000000000001,result: 6
Query quantile: 0.7000000000000001,result: 6
Query quantile: 0.8,result: 7
Query quantile: 0.9,result: 8

Seção 4. Referências

[1] - http://dimacs.rutgers.edu/~graham/ssbd/ssbd4.pdf
[2] - http://infolab.stanford.edu/~datar/courses/cs361a/papers/quantiles.pdf
[3] - https://www.datacamp.com/community/tutorials/argument-parsing-in-python