RELATÓRIO EP1 (13/04)

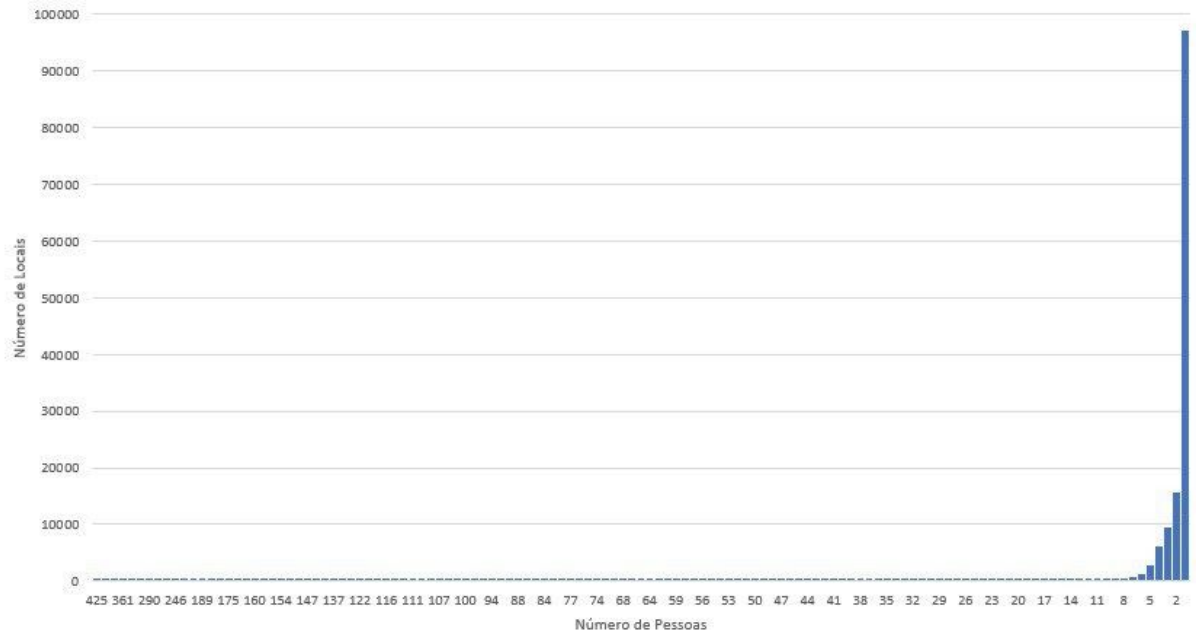
Processamento dos dados da pesquisa Origem/Destino de São Paulo, realizada em 2017

Alexandre Kenji Okamoto	11208371
Daniel Feitosa dos Santos	11270591
Fernanda Cavalcante Nascimento	11390827
Giovani Verginelli Haka	11295696
Larissa Yurie Maruyama	11295928
Matheus Antonio Cardoso Reyes	11270910
Otávio Nunes Rosa	11319037

REPOSITÓRIO NO GITHUB

https://github.com/matheus-reyes/AEDII-Grafos

HISTOGRAMA



TEMPO REAL DE PROCESSAMENTO TOTAL (EM SEGUNDOS)

Dell Inspiron 5584 - i7 (8º Geração) 8565U - 8GB RAM - SSD 128GB / HDD 1TB	4231.56
Lenovo Legion Y7000 - i7 (8ª Geração) 8750H - 8GB RAM - SSD 550GB	5917.10
Lenovo ThinkPad T430 - i5 (3º Geração) 3320M - 8GB RAM - HDD 500GB	6052.47
Acer Aspire 5 - i5 (8ª geração) 8265U - 16GB RAM - SSD 512GB	6993.99
Samsung Essentials E31 NP370E4K - i3 (5ª Geração) 5005U - 4GB RAM	9046.68
Lenovo Ideapad 330 - i7 (8ª Geração) 8550U - 8GB RAM - SSD 550GB	9469.82
Acer Aspire V3 - i7 (3ª Geração) 3632QM - 4GB RAM - HD 500GB	10625.73
Itautec InfoWay Note W7415 - Pentium(R) Dual-Core CPU T4500 - 3GB RAM - HD 500GB	10774.99

ESTIMATIVA DO TEMPO DE PROCESSAMENTO ASSINTÓTICO

$$Fp(\sum_{n-1}^{0}(n-1)! + x\sum_{n}^{1}n!) + C$$

Levando em conta uma pessoa y1, que seria a primeira a ser inserida na lista de lugares, o programa apenas salvaria todas as coordenadas que esse indivíduo frequentou.

Da forma em que o código foi estruturado, dentro do arquivo "ListaLigada.py", existe uma condição para verificar se as coordenadas inseridas já foram previamente colocadas por outros usuários, e como, neste caso, estamos tratando da primeira pessoa, o programa rodará o trecho de código correspondente a inserir a primeira coordenada sem a necessidade de verificação prévia.

Assim, à medida em que forem inseridas mais coordenadas dessa mesma pessoa, mais conjuntos de coordenadas terão de passar pela verificação se já são existentes ou não. Atribuindo aos locais a variável n, teremos $\Sigma\ (n - 1)!$, isso porque teríamos n lugares menos o primeiro caso que não precisa da verificação. E por ser um loop que aumenta

progressivamente a cada usuário inserido (consequentemente com seus conjuntos de coordenadas), temos o fatorial.

Conforme mais pessoas são acrescentadas, a lista ligada de lugares é verificada progressivamente por cada conjunto de coordenadas. A variável x sendo equivalente a y - 1, ou seja, o número total de pessoas menos o primeiro caso; temos o segundo trecho da equação em que multiplicamos x pela somatória de $n!$, já que a cada inserção de indivíduo haverá mais conjuntos de lugares acrescentados à lista para serem verificados.

$$Fp(\sum_{n-1}^{0}(n - 1)! + x\sum_{n}^{1}n!) + C$$

(I) $T(\Sigma(n - 1)! + x\Sigma n!) + C$
(II) $T(\Sigma(n - 2)! + x\Sigma(n + 1)!) + C + C$
(III) $T(\Sigma(n - 3)! + x\Sigma(n + 2)!) + C + C + C$
(IV) $T(\Sigma(n - 4)! + x\Sigma(n + 3)!) + C + C + C + C$

CASO BASE

$n - i = 1$
$i = n - 1$

$(i)T(\Sigma(n - i)! + x\Sigma(n + (i - 1))!) + iC$
$(n - 1)T(\Sigma(n - (n - 1))! + x\Sigma(n + (n - 1) - 1)!) + (n - 1)C$
$(n - 1)T(\Sigma 1! + x\Sigma(2n - 2!)) + (n - 1) + C$

Ou seja,

$O(n!)$

Melhor caso
No melhor caso, não haveria ninguém em lugar nenhum, pois o isolamento total seria respeitado hipoteticamente.

SUGESTÕES DE ALGORITMOS E ESTRUTURAS DE DADOS QUE FARIAM O PROGRAMA MAIS EFICIENTE E O PORQUÊ

A estrutura de dados escolhida pelo grupo foi uma lista ligada, conectando os elementos por ponteiros que conectam o próximo elemento. Analisando as estruturas conhecidas, dificilmente alguma conseguiria ser usada de forma muito eficiente, sem prejudicar os processos ou de inserção ou de busca.

O trecho de código que julgamos mais lento para o programa é o seguinte:

ListaLigada.py

```
18      if inicio:
19          ponteiro = inicio
20          # Enquanto o elemento tiver próximo na lista
21          while(ponteiro.prox):
22              # Caso já tenham locais iguais na lista, é adicionado idpessoa no
vetor frequentadores e a função é retornada
23              if ponteiro.coordenada_x == coordenada_x and ponteiro.coordenada_y ==
coordenada_y:
24                  ponteiro.adicionarFrequentador(id_pessoa)
25                  return
26              ponteiro = ponteiro.prox
27          # Caso não tenham Locais iguais e não tenham mais próximos na lista,
adiciona o novo local e ajusta o ponteiro prox anterior
28          Local(coordenada_x, coordenada_y)
29          ponteiro.prox = Local(coordenada_x, coordenada_y)
30          ponteiro.prox.adicionarFrequentador(id_pessoa)
31      # Caso ainda não tenham Locais na lista
32      else:
33          Local(coordenada_x, coordenada_y)
34          inicio = Local(coordenada_x, coordenada_y)
35          inicio.adicionarFrequentador(id_pessoa)
36      quantidade = quantidade + 1
```

Aqui, o programa necessita verificar as localizações individualmente duas vezes (uma para a coordenada X e outra para a coordenada Y) para checar se a coordenada já existe ou é uma nova, para depois conectar um ID a essa localização.

Assim, uma forma que poderia ser usada para otimizar esse trecho seria conseguir realizar uma busca binária. Para isso, seria necessário ordenar tanto a coordenada X quanto a coordenada Y. Visto isso, apesar de otimizar o processo de busca, essa modificação tornaria a inserção lenta, já que teríamos que ordenar a tabela duas vezes antes de finalmente inserir na lista ligada.

Algumas sugestões que surgiram no grupo foram sobre o uso de uma estrutura parecida com hashing, usando array para distribuir os IDs e uma lista ligada para associar suas coordenadas, mas o grupo não chegou a nenhuma conclusão eficiente sobre como melhorar a estrutura, apesar de saber que um tempo assintótico de $O\,(n!)$ é um dos piores casos que existe.