Algoritmos e Estruturas de Dados

2ª Série

Problema: Pandemia Z (Segunda Parte)

N. 45827 Nome Daniel Azevedo

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2019/2020

10/06/2020

Índice

Conteúdo

Introdução

O programa simula o controlo da informação referente a pacientes testados durante uma pandemia. O centro em questão pretende obter resposta para as seguintes perguntas: quantas e quais as pessoas que podem ficar infetadas com este vírus; dada uma localidade, e se a mesma poderá ser considerada uma zona de risco. Cada amostra é anonimizada e registada com um identificador, localidade, os indivíduos que a pessoa contacta diariamente, a data em que foi produzida, e o estado clínico do paciente relativamente ao vírus.
O objetivo deste trabalho é armazenar e tratar as informações relativas a cada individuo, para que os seus valores possam estar em constante atualização, e possam ser extraídos de forma rápida.

Este trabalho encontra-se dividido em duas partes. Na primeira apenas é tratada a informação relativa aos distritos e seus respetivos números de diferentes tipos de situações possíveis (situação clínica do paciente).

A segunda parte têm como papel incidir na forma como o contágio entre as pessoas se desenvolve. Poderemos observar as diferentes consequências ao longo do tempo, consoante as diferentes situações clínicas dos indivíduos testados e grupos de pessoas associados.

Pandemia Z

Esta aplicação têm 4 funcionalidade disponíveis para o usuário: **add sourcefile.EZ** com o qual pode adicionar um novo ficheiro; **town_hall** que permite obter uma contagem por concelho de quantos indivíduos estão infetados, suscetíveis ou resistentes; **top k** que lista no standard output os k concelhos com mais infetados do programa; e por fim **exterminate** que encerra a aplicação avisando o utilizador antes.
Duas das 4, como podemos reparar requerem informação relativa ao Distrito, o que sugere uma hierarquia na forma como a informação deve estar disposta. Então para o mesmo efeito, foi criado duas classes individuas: Classe **Person** que representa a informação do individuo testado no centro, e a Classe **District** que representa apenas os números relativos aos resultados dos testes por distrito. Havia a hipótese de haver indivíduos já testados, a serem testados novamente no centro, o que me obrigava a atualizar a informação anterior desse paciente. Daí ter adotado pela estrutura de armazenamento HashMap, pois quando a key ou value é igual a nova a inserir, o valor no seu interior que correspondente a essa chave ou valor em particular é substituída pelo seu novo valor.

O método **insertInformation** é responsável por extraír a informação contida nos ficheiro .EZ, vai "ler" cada linha do ficheiro, e reparti-la de forma a que esta possa ser enviada para o Construtor da Classe Person, e assim possa ser criado o objeto associado à mesma. Contudo, não nos podemos esquecer de atualizar respetivos valores do Distrito desta pessoa. Para esse efeito crio o objeto auxiliar do tipo District para identificar se é necessário adicionar uma nova chave ao mapa, ou apenas atualizar os atributos relativos ao objeto contido nesta.

O método **userInterface** apenas têm o preceito de mostrar as diversas funcionalidades explicadas anteriormente. Encontra-se em loop, até decisão contrária do usuário terminar o programa ao introduzir a opção exterminate. O método auxiliar ask é usado para obter a resposta fornecida pela consola. Apenas existem 2 opções no switch mais pertinentes, top e town_hall. Town_hall percorre as diferentes chaves do HashMap district, e imprime os valores associados. Top necessitou de mais trabalho e atenção, foi-me sugerido pelo Docente a utilização de um algoritmo que não ordena se os Distritos em relação ao seu número de infetados, mas sim organiza-se. Isto porque só precisamos do k com mais infetados, e não nos é imposto a condição de estarem ordenados. Daí ter adotado o algoritmo *MaxHeap*, o qual têm um desempenho muito bom (n log (n)) em qualquer um dos seus casos (Ω, Θ, ou O). Iremos verificar esta propriedade do algoritmo nos resultados experimentais a seguir.

O código do programa em que foi feito alterações/acréscimos:

- **Código Java**

```
class Graph {

    private static Map<Integer, Person> adj;

    public static void setAdj(Map<Integer, Person> people) {
        adj = people;
    }

    public static LinkedList<Integer> breadthFirstSearch(LinkedList<Integer> infected) {
        LinkedList<Integer> infectedAndPossible = new LinkedList<>();
        if (infected.size() == 0) return infectedAndPossible;
        Iterator<Integer> it = infected.iterator();
        Queue<Integer> queue = new LinkedList<>();
        while (it.hasNext()) {
            Integer p = it.next();
            infectedAndPossible.add(adj.get(p).getId());
            adj.get(p).setPossibleClinicalState('I');
            queue.add(adj.get(p).getId());
            while (queue.size() != 0) {
                Integer first = queue.poll();
                HashSet<Integer> aux = adj.get(first).getContactedPersons();
                for (Integer n : aux) {
                    if (adj.get(n).getClinicalState() == 'R') break;
                    if (adj.get(n).getPossibleClinicalState() == 'N' && adj.get(n).getClinicalState()
!= 'R') {
                        infectedAndPossible.add(n);
                        adj.get(n).setPossibleClinicalState('I');
                        queue.add(n);
                    }
                }
            }
        }
        return infectedAndPossible;
    }

    public static boolean breadthFirstSearch(int id) {
        Queue<Integer> queue = new LinkedList<>();
        if(adj.get(id).getClinicalState() == 'I')
            return true;
        adj.get(id).setPossibleClinicalState('I');
        queue.add(id);
        while (queue.size() != 0) {
            id = queue.poll();
            HashSet<Integer> aux = adj.get(id).getContactedPersons();
            for(Integer a: aux) {
                if(adj.get(a).getClinicalState() == 'I')
                    return true;
                if(adj.get(a).getClinicalState() != 'R' && adj.get(a).getPossibleClinicalState() == 'N')
{
                    queue.add(a);
                    adj.get(a).setPossibleClinicalState('I');
                }
```

```
                if(adj.get(a).getClinicalState() == 'R')
                    break;
                }
            }return false;
        }
    }
```

Fiz *overloading* do método breadthFirstSearch para conseguir diminuir o tempo de execução da primeira funcionalidade (becomeInfected? x).

```
//array de adjacências
int n = values.length - 4;
HashSet<Integer> adjacency = new HashSet<Integer>();
for(int i = 4, j = 0; j < n; ++i, ++j){
   adjacency.add(Integer.parseInt(values[i]));
}
```

Foi acrescentado um HashSet como campo da class Person, de forma a permitir a elaboração das ligações entre os diferentes indivíduos testados.

```
case "id":
      Graph.setAdj(people);                                    //para restaurar campos da class
      int id = Integer.parseInt(aux[1]);
      if(people.get(id) == null) {
         System.out.println("Your id isn't a valid one");
         break;
      }
      boolean bool = Graph.breadthFirstSearch(id);
      System.out.println(bool);
      break;
case "becomeInfected":
      Graph.setAdj(people);                                    //para restaurar campos da class
      for (Map.Entry<Integer, Person> entry : people.entrySet()) {
         if(entry.getValue().getClinicalState()!='R' && entry.getValue().getClinicalState()!='S')
            infectedPeople.add(entry.getValue().getId());
      }
      LinkedList<Integer> possibleInfectedPersons = Graph.breadthFirstSearch(infectedPeople);
      Collections.sort(possibleInfectedPersons);               //substituir no futuro
      System.out.println("total:"+possibleInfectedPersons.size());
      for(Integer p: possibleInfectedPersons)
         System.out.println(p);
      break;
```

Este troço de código é responsável pela execução das duas primeiras funcionalidades.

Resultados experimentais

De forma a poder testar o programa corretamente, tive que criar o programa a seguir. Este permite-me criar ficheiros de grandes dimensões e com valores aleatórios enquadrados no problema. Basta modificar o campo *int howManyPersons* (número de pessoas pretendidas) e *String fileName* (nome do ficheiro pretendido), *districts (*um array com os diferentes nomes de distritos possíveis), *masPersonsContacted* (número máximo de pessoas que uma pessoa pode estar em contacto diariamente).

- **Código Java**

```
public class filesGenerator {

   final static int howManyPersons = 1200000;
   final static String fileName = "p8.EZ";
   final static int maxPersonsContacted = 20;
   final static String[] districts = {"Aveiro", "Porto", "Braga", "Viana_do_Caselo", "Vila_Real",
"Bragança", "Castelo_Branco",
         "Guarda", "Viseu", "Évora", "Santarém", "Lisboa", "Setúbal", "Beja", "Faro", "Portalegre",
"Leiria", "Coimbra", "Madeira", "Açores"};
   //final static int districtsAux = 1280;
   public static Integer[] aux = new Integer[howManyPersons];
   public static int auxCounter = 0;
   public static StringBuilder contactedPersonsID;

   final static Character[] clinicalState = {'S', 'I', 'R'};

   public static void main(String[] args) throws FileNotFoundException {
      File file = new File(fileName);
      PrintStream output = new PrintStream(file.getName());
      output.println(howManyPersons);
      /*
      for(int j = 0; j < districtsAux; j++){
         districts[j] = "Distric->"+j;
      }
       */
      for (int i = 0; i < howManyPersons; ++i) {
         int id = (int) (Math.random() * howManyPersons);
         while(aux[id]!=null) {
            id = searchAnotherId();
         }
         aux[id] = id;
         int whichDistrict = (int) (Math.random() * districts.length);
         int whichClinicalState = (int) (Math.random() * 3);
         if(auxCounter > 10) {
            int n = (int)(Math.random() * maxPersonsContacted);
            contactedPersonsID = new StringBuilder();
            for(int j = 0; j <= n; ++j) {
               int person = (int) (Math.random()*howManyPersons);
               if(aux[person] != null)
                  contactedPersonsID.append(aux[person]+" ");
            }
```

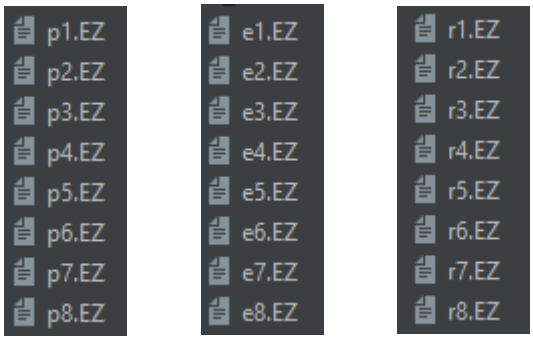
```
        }
        RandomDates rdn = new RandomDates();
        if(contactedPersonsID != null) {
            output.println("" + id + " " + districts[whichDistrict] + " " + clinicalState[whichClinicalState]
+ " " + rdn.date + " " + contactedPersonsID);
        }else{
            output.println("" + id + " " + districts[whichDistrict] + " " + clinicalState[whichClinicalState]
+ " " + rdn.date);
        }
        ++auxCounter;
      }
      System.out.println(districts.length);
    }

    private static int searchAnotherId() {
      return (int) (Math.random() * howManyPersons);
    }
}
```

Os ficheiros têm datas aleatórias compreendidas entre 2000 e 2020.

```
public class RandomDates {

    public String date;

    public RandomDates(){
      this.date =  ""+createRandomDate(2000, 2020);
    }

    public int createRandomIntBetween(int start, int end) {
      return start + (int) Math.round(Math.random() * (end - start));
    }

    public LocalDate createRandomDate(int startYear, int endYear) {
      int day = createRandomIntBetween(1, 28);
      int month = createRandomIntBetween(1, 12);
      int year = createRandomIntBetween(startYear, endYear);
      return LocalDate.of(year, month, day);
    }
}
```

FileGenerator garante que não existe id iguais no ficheiro, e que as pessoas contactas (pelo individuo testado) diariamente realmente existem. Com o intuito de não provocar conflitos com algumas das funcionalidades da segunda parte do problema.

Criei 3 conjuntos de ficheiros:



- Os ficheiros p.EZ têm um incremento de população (por ordem crescente dos ficheiros), de 150.000 pessoas de um ficheiro para o próximo. Contudo o número de distritos e número máximo de pessoas possíveis a serem contactadas (diariamente) pela pessoa testada é 20.

- Os ficheiros e.EZ têm um incremento de população semelhante aos p.EZ, porém o número de pessoas possíveis de contactar diariamente duplica a cada ficheiro. No ficheiro nº 8 esse limite é de 1280 pessoas.

- Por último os ficheiros r.EZ, semelhante a e.EZ, mas agora também o seu número possível de distritos também duplica a cada ficheiro.

Nem todos os ficheiros e.EZ e r.EZ foram usados devido ao erro - Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "main".
Foi o caso de e7, e8, r6, r7, r8.

Para medir os tempos em cada um dos testes, foi utilizado os métodos estáticos:

```
long initTime1 = System.nanoTime();
long initTime = System.currentTimeMillis();
long elapsedTime1 = System.nanoTime()-initTime1;
long elapsedTime = System.currentTimeMillis()-initTime;
```

Primeiro gostava de analisar os tempos de leitura do ficheiros com diferentes objetos. Neste caso vou comparar *Scanner* com *BufferedReader* em termos de custo de tempo na leitura dos ficheiros anteriores.

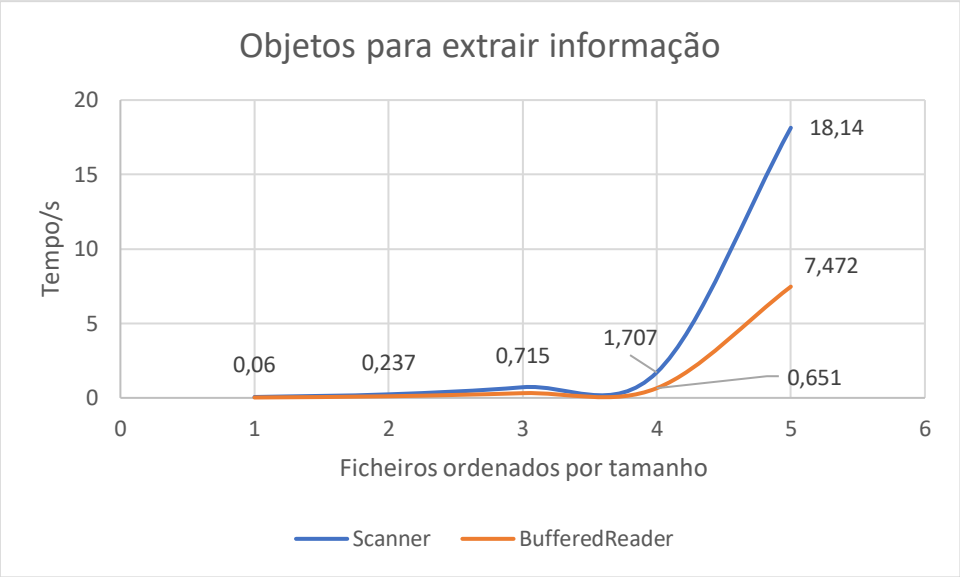


Figura 1 – Tempo de extração de informação dos ficheiros (máximo de 20 Distritos)

Como podemos ver, Scanner ao longo do tempo aproxima-se do dobro do custo de tempo em relação ao BufferedReader. Este fator levou me a substituir o Scanner anteriormente implementado no trabalho por BufferedReader.

Ficheiro	Ficheiros P	Ficheiros R	Ficheiros E
1	0.376	0.282	0.351
2	0.651	0.609	0.694
3	0.795	1.010	1.289
4	1.116	2.221	2.701
5	1.444	4.695	5.540
6	1.749		49.933
7	2.068		
8	2.086		

Figura 2 – Tempo de adicionar cada ficheiro à aplicação

Os quadriculados a amarelo, são aqueles ficheiros que não puderam ser usados, devido ao erro anteriormente especificado.

Top k têm custo O(k log n). Todavia no Heap, se aumentarmos gradualmente o seu número de elementos a sua complexidade no espaço continua a ser O(1). E independentemente dos casos do algoritmo (Ω, Θ, O) o seu custo temporal irá ser sempre k log n. Podemos averiguar esta situação nos resultados a seguir:

Figura 2 – Funcionalidade que retorna o TOP x distritos mais infetados (Ficheiros P)

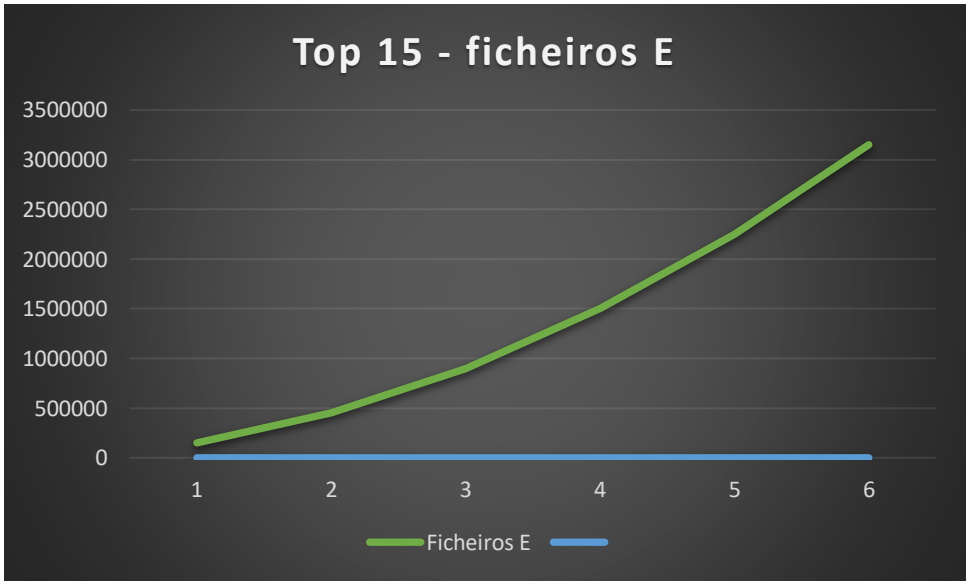


Figura 3 - Funcionalidade que retorna o TOP x distritos mais infetados (Ficheiros E)

Como os gráficos das figuras 2 e 3 demonstram, independentemente do número de pessoas albergadas no sistema, o tempo de execução da funcionalidade mantêm-se muito próximo de 0 segundos.

Ficheiros R	Coluna1
top 5	0.014215
top 10	0.0003215
top 20	0.0005269
top 40	0.0008512
top 80	0.0014933

Figura 4 - Funcionalidade que retorna o TOP x distritos mais infetados (Ficheiros R)

Como nos ficheiros tipo R o número de distritos não é constante ao longo ficheiros, fui adicionando ficheiro a ficheiro, e a medida que o fazia chamei a funcionalidade para retornar top distritos possíveis / 2, e tentei observar mudanças no tempo de execução. Os diferentes valores devem ter sido originados por variações do sistema em que aplicação foi executada (Ex: distintos processos a correr no sistema operativo).

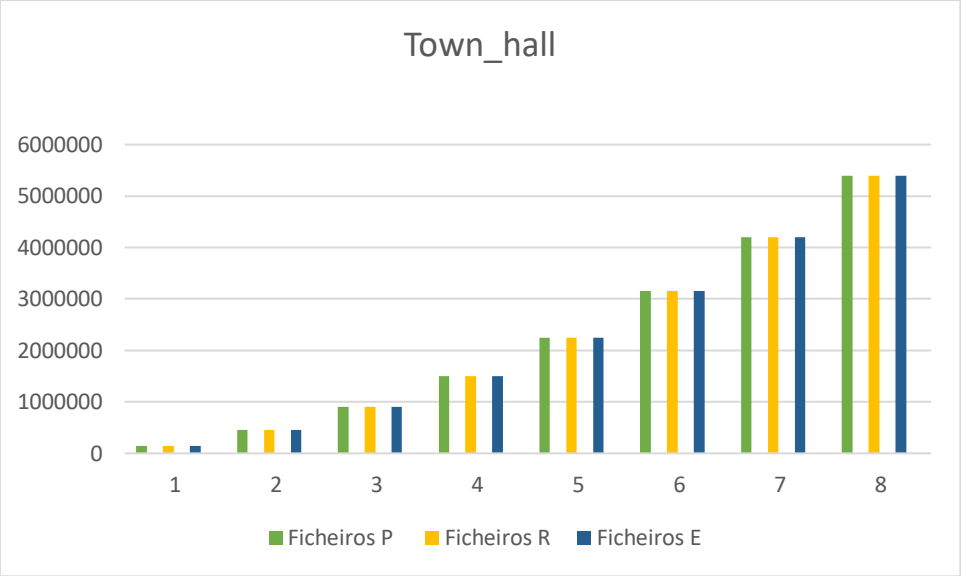


Figura 5 - Funcionalidade Town_hall x (tempo de execução dos 3 tipos de ficheiros)

No town_hall independentemente de eu aumentar a complexidade do espaço (chaves no *HashMap)* os tempos de execução mantêm se muito próximos. Mas à medida que o *HashMap* se dirige para valores muitos altos não tenho dúvidas que haverá alterações no tempo de execução da funcionalidade. A sua complexidade do tempo no pior caso é O(n), e como a sua complexidade do espaço também é O(n), sabemos que à medida que nos aproximamos de valores muitos elevados (chaves diferentes ou *hash code* iguais) na estrutura de dados, também nos aproximamos de O(n). Felizmente este pior caso não surge frequentemente.

Pessoas	Ficheiros P	Ficheiros R	Ficheiros E
150000	0.381	0.393	0.386
450000	0.779	0.710	0.807
900000	1.275	1.107	1.539
1500000	2.322	2.271	2.491
2250000	3.249	2.661	3.625
3150000	4.576		
4200000	5.859		
5400000	7.236		

Figura 6 – Tempo de execução da funcionalidade que indica o número de infetados, caso não seja encontrada uma vacina

Nº Testados	Ficheiros P	Ficheiros R	Ficheiros E
150000	0.0007579	0.0008035	0.0005993
450000	0.0001079	0.0000538	0.0002196
900000	0.0001249	0.0000559	0.0001416
1500000	0.0000404	0.0000424	0.0007013
2250000	0.0000544	0.0000589	0.0000575
3150000	0.0000484		0.0000654
4200000	0.0000492		
5400000	0.000042		

Figura 7 – Tempo de execução da funcionalidade que deteta se um determinado id (pessoa) é possível de ser infetado

A funcionalidade da figura 6 de forma a ser executada têm que percorrer o diferentes ids das pessoas em que o individuo(s) testado(s) contacta(m) diariamente. Com os resultados obtidos, podemos presumir que o crescimento desta é linear com o número de pessoas no sistema, o que é normal neste caso.

Contudo, na funcionalidade a seguir (Figura 7) é impossível dizer concretamente o tempo de execução, visto este depender do id fornecido pelo utilizador. Há diferentes fatores que estão diretamente relacionados com o percurso que algoritmo têm que fazer. Por exemplo, se a primeira pessoas que o individuo têm contacto é resistente ao vírus, automaticamente classifica a pessoa como não sendo capaz de ser infetada. Sim, não é realista, mas seria muito mais complexo inserir outras alternativas.

Conclusão

Achei muito adequado e motivador o tema do trabalho, pelo facto de estar relacionado com a pandemia presente no Mundo, o COVID-19. Inicialmente tive dificuldades em perceber como os HashMap funcionavam, ou como disponham a sua informação. Algumas análises feitas por mim podiam talvez ser mais aprofundadas, como é o caso de não ter conseguido comprovar o pior caso no HashMap, mas penso que seria necessário muitos mais dados para o efeito. Pude também observar que os algoritmos implementados não dispõem de memória muito elevada, pois 50 milhões de elementos já me impossibilitava de prosseguir o uso dos mesmos. E que nem todos os algoritmos da biblioteca com funções idênticas se comportam de maneira igual ao longo do seu tempo e espaço. Tive bastante dificuldade na implementação da segunda parte do trabalho, na implementação da lista de adjacências. O que me levou a atrasar imenso da implementação da segunda parte do problema.