Análise e Desenho de Algoritmos: Trabalho #2

To win, or not to win: that is the question

Prof. Margarida Mamede

Turno Prático 1

Luis Marques 34213

22 de Abril de 2013

Conteúdo

1 Apresentação do Problema

Uma empresa tem fantásticos prémios para entregar às pessoas que responderam corretamente (aces) a um difícil questionário. Estes prémios serão entregues às x pessoas que entregaram primeiro. Sendo x o número de prémios. As pessoas responsáveis por receber as respostas esqueceram-se de marcar a hora de cada entrega sendo assim muito difícil determinar quem tem direito aos prémios. Ainda assim estes funcionários têm a certeza absoluta de que algumas pessoas entregaram claramente antes que outras. O problema consiste em, com base nesta informação, descobrir quais as pessoas que garantidamente vão e não vão receber o prémio, independentemente do método escolhido para resolver os casos de dúvida.

2 Resolução do Problema

Para resolver este problema é preciso identificar quem ganha e quem não ganha. Para isso basta contar quantas pessoas de certeza receberam um prémio antes e depois da pessoa que estamos a questionar. Se a pessoa em questão tiver o mesmo número ou mais pessoas que têm que receber o prémio antes dela do que o número de prémios disponíveis, então garantidamente a pessoa não receberá o prémio. Por outro lado, e simetricamente, se o número de pessoas que têm que receber o prémio depois desta pessoa for superior ou igual ao número de pessoas que não vão receber prémios, então garantidamente esta pessoa receberá o prémio. Todos os casos que não se encaixem numa destas situações, à partida, não têm final decidido.

3 Implementação do Algoritmo

3.1 Técnica de desenho de algoritmos utilizada

Neste algoritmo utilizei grafos orientados e efetuei sobre eles uma versão, optimizada ao problema, do algoritmo de pesquisa em profundidade primeiro para calcular o número de sucessores totais. Utilizei profundidade em vez de pesquisa em largura porque pesquisando em profundidade torna-se mais simples optimizar a pesquisa, pois se durante a pesquisa se atingir qualquer um dos limites falados na secção "Resolução do Problema"essa informação pode ser guardada evitando futuras execuções desnecessárias.

3.2 Breve apresentação das interfaces e das classes criadas

Neste problema apenas criei uma classe de exceção. Necessitei desta classe para lançar a exceção quando o utilizador introduz um valor não válido. De resto não senti necessidade de criar novas interfaces/classes. Utilizei apenas as bibliotecas do Java.

3.3 Estruturas de dados usadas

Para guardar os antecessores(ant.) e os sucessores(suc.) decidi guardar em duas estruturas separadas para que seja mais eficiente iterá-las.

3.3.1 Grafo

Para implementar eficientemente o algoritmo identifiquei várias necessidades para as estruturas do grafo:

1. Adicionar sucessores ou antecessores.
2. Perguntar se dado vértice tem antecessores ou sucessores.
3. Obter os sucessores ou antecessores de um vértice dado o seu identificador, neste caso o inteiro.
4. Não ter elementos repetidos nas colecções de sucessores e antecessores.
5. Saber se dado vértice é sucessor ou antecessor de outro.

Optei por uma **tabela de dispersão** de chave o **inteiro** que identifica o vértice e de valor a **colecção de suc. ou ant**. Optei pela tabela de dispersão em vez de um vector pois há casos em que sabemos à partida que nem todos os vértices terão suc. ou ant. Por exemplo, no caso em que temos 1000 questionários e apenas 3 regras, no máximo, teremos apenas 3 vértices com suc. ou ant., assim sendo um vector de 1000 posições iria ter espaço desperdiçado. (Necessidades 2. e 3. resolvidas com complexidade constante no caso esperado).
Para a estrutura de dentro da tabela de dispersão optei por uma ***LinkedList*** por ser um estrutura dinâmica e por resolver também com complexidade constante a adição. Mas só por si, esta estrutura tem complexidade linear para verificar se um vértice está na lista, e não garante que não existem repetidos. Por isso, concebi que a melhor solução seria auxiliar esta estrutura com uma ***matriz de booleanos*** de dimensão aces*aces que tem a informação sobre se um vértice está na LinkedList do outro ou não.(Necessidades 1.,4. e 5. resolvidas com complexidade constante no caso esperado.).

3.3.2 Memória

Utilizei também um vector de inteiros para armazenar resultados de chamadas anteriores e assim impedir chamadas desnecessárias, otimizando o código. Este vector serve também para manter informação sobre se um nó está explorado ou não.

3.4 Pontos importantes do código

Procurei fazer o código "à prova de utilizador" negando inputs que não fazem sentido e dando a oportunidade ao utilizador de os corrigir.
Durante a execução do programa nunca executo o método para calcular o número de sucessores ou antecessores se já o tivesse feito anteriormente.

Outra otimização que fiz foi parar a pesquisa em profundidade quando chego a um estado em que tenho a certeza que independentemente do que falta explorar as condições explicitadas na secção "Resolução do Problema"têm um resultado certo. Ou seja, quando num determinado momento o número de antecessores é igual ou superior ao número de prémios ou quando o número de sucessores é igual ou superior ao número de pessoas que não vão receber o prémio.

4 Análise do Algoritmo

R - Número de Regras

A - Número de Aces

Di - Número de Sucessores Directos do nó i

Si - Número de Sucessores (directos e indirectos) do nó i

T - Número de Testes

4.1 Estudo da complexidade espacial

- Tabela de dispersão - θ(min(A,R)) todos os casos
- LinkedList - O(R) (soma de todas)
- Matriz de Booleanos - $\theta(A^2)$ todos os casos
- Array Memória - θ(A) todos os casos

$\theta(A^2)>=\theta$(A) em todos os casos.
O(min(A,R)+R) pode ser escrito como O(R) porque caso A seja maior fica O(2R) caso contrário fica O(R).
$\theta(A^2)>\theta$(R) porque como não há ciclos neste grafo o número de regras diferentes no máximo é A+(A-1)+...+(A-(A-1)) que é sempre inferior a A+A+A+...=(A^2)
Dado isto a complexidade espacial do algoritmo é: $\theta(A^2)$

4.2 Estudo da complexidade temporal

- Criação e inicialização de variáveis - $\theta(1)$ todos os casos
- Ciclo R vezes
  - Obter lista de sucessores - O(min(A,R)) no pior caso e O(1) no caso esperado.
  - Adicionar à lista de sucessores - O(1) todos os casos
- Ciclo T vezes

- Consulta vector e escrita em vector - O(1)
- Criação de cópia dos filhos directos - θ(Di)
- Criação de iterador - O(1)
- Ciclo Si vezes de operações constantes

Dado isto a complexidade temporal do algoritmo é: O(R*min(A,R)+T*Si) no pior caso e R+T*Si no caso esperado)

5 Conclusões

A complexidade temporal do algoritmo é muito elevada no pior caso mas penso que seja importante realçar que trata-se de um caso muito pouco frequente, caso em que as pesquisas na tabela de dispersão têm complexidade da ordem do número de elementos. No melhor caso o algoritmo tem complexidade muito inferior porque tem inteligência para perceber quando atinge os limites referidos na secção "Resolução do problema".

Alternativas estudadas ou que mereciam ser estudadas

Para a estrutura de dentro da tabela de dispersão comecei por optar por um ***HashSet*** porque permitia a adição de suc. e ant. com complexidade constante; porque consultar se vértice é suc. ou ant. de outro também tinha complexidade constante. Mas não funcionou pois claramente excede o limite de memória para o problema apresentado.

Assim sendo, percebi que precisava de uma estrutura de tamanho dinâmico. Por isso decidi utilizar ***TreeSet*** e já tinha uma complexidade espacial suficientemente boa para passar no teste do mooshak. Ainda assim, eu não estava satisfeito pois sentia que o algoritmo poderia ser muito mais eficiente se eu não tivesse complexidades para adicionar e consultar se dado vértice é suc. ou ant. de outro logarítmicas. Portanto criei a solução usada solução.

A Code Listing

A.1 Main.java

```
1  import java.util.*;
2
3  import exceptions.InvalidInputException;
4
5  /**
6   * @author lfmarques - 34213
7   * @problem - To win, or not to win: that is the question
8   */
9  public class Main {
10
11         private static final String YES = "Congratulations!";
12         private static final String MAYBE = "You have chances";
13         private static final String NO = "Sorry";
14
15         private static final String INVALID_RANGE =
16                         " is an invalid number, the value must be between 0 and aces-1";
17
18         public static void main(String[] args) {
19                 Scanner in = new Scanner(System.in);
20
21                 int prizes = in.nextInt();
22                 int aces = in.nextInt();
23                 int rules = in.nextInt();
24
25                 int noLimit = prizes;
26                 int yesLimit = aces - prizes;
27                 int first = yesLimit > noLimit ? 1 : 0;
28
29                 int maxSize = aces > rules ? rules : aces;
30
31                 /*
32                  * When "Bef" is used it means the graph of the ancestors and "Aft" the
33                  * graph of the successors.
34                  */
35
36                 HashMap<Integer, LinkedList<Integer>> graphBef =
37                                 new HashMap<Integer, LinkedList<Integer>>(maxSize);
38                 HashMap<Integer, LinkedList<Integer>> graphAft =
39                                 new HashMap<Integer, LinkedList<Integer>>(maxSize);
40
41                 boolean[][] belongsAft = new boolean[aces][aces];
42                 boolean[][] belongsBef = new boolean[aces][aces];
43
44                 int[] memoryAft = new int[aces];
45                 int[] memoryBef = new int[aces];
46
47                 // Read the rules
48                 for (; rules-- > 0;) {
49                         int bef = in.nextInt();
50                         int aft = in.nextInt();
51
52                         try {
53                                 validate(bef, aces);
54                                 validate(aft, aces);
55                         } catch (InvalidInputException e) {
```

```
56                              System.out.println(e.getMessage());
57                              rules++;
58                      }
59
60                      addEdge(graphAft, belongsAft, bef, aft);
61                      addEdge(graphBef, belongsBef, aft, bef);
62              }
63
64              // Do the tests
65              int tests = in.nextInt();
66              for (; tests-- > 0;) {
67                      int test = in.nextInt();
68                      try {
69                              validate(test, aces);
70                      } catch (InvalidInputException e) {
71                              System.out.println(e.getMessage());
72                              test++;
73                      }
74
75                      // Tests done here to prevent unnecessary depth searches
76                      if (memoryAft[test] >= yesLimit) {
77                              System.out.println(YES);
78                              continue;
79                      }
80                      if (memoryBef[test] >= noLimit) {
81                              System.out.println(NO);
82                              continue;
83                      }
84                      if (memoryBef[test] != 0 && memoryAft[test] != 0) {
85                              System.out.println(MAYBE);
86                              continue;
87                      }
88
89                      int testFirst = first;
90                      boolean returned = false;
91
92                      for (int i = 0; i < 2; i++) {
93                              switch (testFirst % 2) {
94                              case 0:
95                                      calculateDepthInList(test, test, memoryBef, graphBef,
96                                                      belongsBef, noLimit, 0, yesLimit,
97                                                      memoryAft);
98                                      if (memoryBef[test] >= noLimit) {
99                                              System.out.println(NO);
100                                             returned = true;
101                                     }
102                                     break;
103                             case 1:
104                                     calculateDepthInList(test, test, memoryAft, graphAft,
105                                                     belongsAft, yesLimit, 0, noLimit,
106                                                     memoryBef);
107                                     if (memoryAft[test] >= yesLimit) {
108                                             System.out.println(YES);
109                                             returned = true;
110                                     }
111                                     break;
112                             }
113                             if (returned)
114                                     break;
```

```
115                             testFirst++;
116                     }
117                     if (!returned)
118                             System.out.println(MAYBE);
119             }
120     }
121
122     /**
123      * @throws InvalidInputException
124      * if val<0 or val>=aces
125      */
126     private static void validate(int val, int aces)
127                     throws InvalidInputException {
128             if (val >= aces || val < 0)
129                     throw new InvalidInputException(val + INVALID_RANGE);
130     }
131
132     /**
133      * This method adds an edge to the graph and updates the auxiliary array
134      * <i>content</i>.
135      *
136      * @param graph
137      * - the graph to add the edge to.
138      * @param content
139      * - auxiliary information containing info about which vertexes
140      * are successors of the other vertexes in the graph.
141      * @param from
142      * - vertex ancestor on the edge
143      * @param to
144      * - vertex successor on the edge
145      */
146     private static void addEdge(HashMap<Integer, LinkedList<Integer>> graph,
147                     boolean[][] content, int from, int to) {
148             LinkedList<Integer> aux = graph.get(from);
149             if (aux == null) {
150                     graph.put(from, new LinkedList<Integer>());
151                     aux = graph.get(from);
152             }
153             if (!content[from][to]) {
154                     content[from][to] = true;
155                     aux.add(to);
156             }
157     }
158
159     /**
160      * Calculates the number of successors of the <i>vertex</i> on the
161      * <i>graph</i> and stores it on the <i>depthMemory</i>. If this number is
162      * less then <i>limit</i> then all the successors are put in the graph as
163      * direct successors
164      *
165      * Besides the big number of arguments, this function is easy to understand.
166      * This method does an optimized, to this problem, version of the depth
167      * first search algorithm. The "inv" arguments are essential to optimize
168      * future calls for the inverse graph, decreasing the total number of
169      * operations.
170      *
171      * @param vertex
172      * - vertex to be study
173      * @param initial
```

```
174         * - original vertex to be study (initial it is equal to
175         * <i>vertex</i>)
176         * @param depthMemory
177         * - array that serves both as a way to see if a vertex was
178         * already explored and to store the result of the search
179         * @param graph
180         * - the graph to search on
181         * @param contains
182         * - complement to the graph structure
183         * @param limit
184         * - the search stops when the number of successors is higher
185         * then this value
186         * @param depth
187         * - the depth in the search (initially 0)
188         * @param invLimit
189         * - the limit for the inverse search
190         * @param depthInvMemory
191         * - the memory array for the inverse search
192         */
193        private static void calculateDepthInList(int vertex, int initial,
194                        int[] depthMemory, HashMap<Integer, LinkedList<Integer>> graph,
195                        boolean[][] contains, int limit, int depth, int invLimit,
196                        int[] depthInvMemory) {
197                if (depthMemory[vertex] != 0)// Test if the vertex is already explored
198                        return;
199
200                if (depth >= invLimit)// This test is to try to optimize the call for
201                                                                // the inverse graph
202                        depthInvMemory[vertex] = Math.max(depth, depthInvMemory[vertex]);
203
204                if (depth >= limit) {
205                        depthMemory[initial] = depth;
206                        return;
207                }
208                LinkedList<Integer> aux = graph.get(vertex);
209                if (aux != null) {
210                        int auxSize = aux.size();
211                        if (auxSize >= limit) {
212                                depthMemory[vertex] = auxSize;
213                                return;
214                        }
215
216                        Iterator<Integer> it = new ArrayList<Integer>(aux).iterator();
217                        int thisDepth = Math.max(depth, 1);
218                        depth++;
219                        boolean reachedLimit = false;
220                        while (it.hasNext()) {
221                                int next = it.next();
222                                int max = Math.max(depthMemory[next] + thisDepth, aux.size());
223                                if (max >= limit) {
224                                        depthMemory[vertex] = max;
225                                        return;
226                                }
227                                calculateDepthInList(next, initial, depthMemory, graph,
228                                                contains, limit, depth, invLimit, depthInvMemory);
229                                if (depthMemory[next] != -1) {
230                                        if (depthMemory[initial] != 0) {
231                                                return;
232                                        }
```

```
233                                     if (depthMemory[next] + depth >= limit) {
234                                             depthMemory[vertex] = depthMemory[next] + thisDepth;
235                                             return;
236                                     }
237                                     LinkedList<Integer> nextSuc = graph.get(next);
238                                     for (int v : nextSuc) {
239                                             if (!contains[vertex][v]) {
240                                                     contains[vertex][v] = true;
241                                                     aux.add(v);
242                                                     if (++auxSize >= limit) {
243                                                             reachedLimit = true;
244                                                             break;
245                                                     }
246
247                                             }
248                                     }
249                             }
250                             if (reachedLimit)
251                                     break;
252                     }
253                     depthMemory[vertex] = auxSize;
254             } else
255                     depthMemory[vertex] = -1;
256     }
257 }
```

A.2 InvalidInputException.java

```
1  package exceptions;
2
3  /**
4   * @author lfmarques
5   *
6   */
7  public class InvalidInputException extends Exception {
8          private static final long serialVersionUID = 1451582905758517751L;
9
10         public InvalidInputException(){
11                 super();
12         }
13
14         public InvalidInputException(String message) {
15                 super(message);
16         }
17
18 }
```