Análise e Desenho de Algoritmos: Trabalho #3

Book Hand Delivery

Prof. Margarida Mamede

Turno Prático 1

Luis Marques 34213

19 de Maio de 2013

Conteúdo

1 Apresentação do Problema

Existe um grupo de pessoas que fazem parte de um grupo de leitura. Estas pessoas precisam de trocar os livros entre si mas acontece que fazer esta troca por correio acaba por sair caro, então precisam de uma alternativa. Um dos membros reparou que existem pessoas com outras actividades em comum, o que o levou a questionar-se se não era possível passar o livro de mão em mão até chegar à pessoa desejada. Isto seria uma solução possível pois as pessoas de semana para semana não mudavam as suas actividades. Assim sendo, o problema está em determinar qual o número menor de dias, ou se é possível, que um livro demora a chegar à outra pessoa através do método de passagem de mão em mão.

2 Resolução do Problema

Cada pessoa tem no máximo uma actividade por dia e portanto se abstraírmos o problema podemos imaginar estas transições como arcos e o menor número de dias o menor caminho pesado entre os dois vértices. Com base nesta ideia, decidi construir um grafo que tem como vértices uma actividade num determinado dia. Os arcos representam a diferença de dias entre os vértices mas só estão presentes no grafo se pelo menos uma pessoa participa nos pares actividade/dia de ambos os vértices. Por outro lado, fora do grafo, guardo quais as actividades em que cada pessoa está presente, para quando for pesquisar qual o menor caminho entre as duas pessoas possa saber quais os vértices onde começar e onde acabar.

3 Implementação do Algoritmo

3.1 Técnica de desenho de algoritmos utilizada

Neste algoritmo utilizei um grafo para representar a informação dada e uma pesquisa de custo uniforme para descobrir qual o menor caminho pesado entre os vértices iniciais e finais.
Utilizei o algoritmo de custo uniforme em vez do algoritmo de Dijkstra pois só preciso saber qual o menor caminho pesado entre um dos nós iniciais e um dos nós finais e não de um para todos os outros.

3.2 Breve apresentação das interfaces e das classes criadas

Para representar um arco utilizei a interface Edge que é implementada pela class EdgeClass e mantém a informação de qual a direcção e do arco e qual o seu peso. Criei também um objecto Pair que representa um par actividade/dia. Por fim, criei ainda uma classe que implementa a pesquisa de custo uniforme chamada UniformCost. Esta classe recebe um grafo no construtor e tem apenas um método search que recebe quais os vértices iniciais e quais os vértices finais da pesquisa. Para efectuar esta pesquisa criei uma classe privada que representa os vértices que vão sendo gerados.

3.3 Estruturas de dados usadas

Para guardar as actividades de uma pessoa utilizei um vector de listas ligadas. O índice do vector é o identificador da pessoa.
Para guardar o grafo utilizei uma lista de adjacências porque é muito mais eficiente descobrir quais os sucessores directos de um vértice desta maneira. Esta lista de adjacências é novamente um vector de listas ligadas e que têm como índice o identificador do nó.
Para construir a fila de espera utilizada no algoritmo de pesquisa utilizei uma fila com prioridade, pois é uma estrutura que me garante que ao remover obtenho o elemento com a menor chave. Utilizei esta estrutura em vez da fila de Fibonacci pois o número máximo de nós deste grafo é muito pequeno (182) e a fila de Fibonacci apenas tem complexidades amortizadas interessantes. Ou seja, para se notar real diferença na perfomance do algoritmo seria necessário um número muito maior de nós.

4 Análise do Algoritmo

P - Número de Pessoas
D - Número de dias (nesta instância, constante, igual a 7)
A - Número de actividades diferentes (nesta instância, constante, igual a 26)
C - Número de combinações diferentes de actividades/dia (nesta instância, constante, igual a 182)
V - Número de vértices (V<=C)
E - Número de Arcos (Edges)
T - Número de testes

4.1 Estudo da complexidade espacial

- Vector de pessoas - θ(P)
  - LinkedList (actividades de uma pessoa) - O(D)
- Vértices do grafo - θ(C)
  - LinkedList (lista de sucessores) - O(V-1) -> O(V)

  Leitura de input
  - Mapa <(Actividade,Dia),Vértice> - θ(C)
  - Matriz bool (verifica presença de arco) - $\theta(C^2)$

  Pesquisa
  - Vector Estados Finais - θ(V+1) -> θ(V)

  - Fronteira de espera - θ(V+1) -> θ(V)
  - Vector de custos encontrados - θ(V+1) -> θ(V)

$V \leq C \leq C * V \leq C^2$ **Complexidade Espacial - O($P * D + C^2$)**

4.2 Estudo da complexidade temporal

Construção Grafo

- Para cada pessoa (Ciclo θ(P))
  - Criar LinkedList (actividades da pessoa) - θ(1)
  - Para cada dia (Ciclo θ(D))
    - Leitura input - θ(1)
    - Consulta&Escrita vector - θ(1)
    - Criação LinkedList - θ(1)
    - Para cada actividade - (Ciclo O(D))
      - Consulta&Escrita vector - θ(1)
    - Adicionar actividade - θ(1)

Complexidade Temporal para construção do Grafo - O($P * D^2$)

Pesquisa

- Criação & Inicialização PriorityQueue - θ(V)
- Criação & Inicialização Vector Custos - θ(V)
- $\leq$ |V| inserir origens na fila - O(V*log(V))
- $\leq$ |V| remover mínimo da fila - O(V*log(V))
- $\leq$ |V| percorrer sucessores directos - O(E)
- $\leq$ |E| remover e/ou adicionar na fila - O(E*log(V))

Complexidade Temporal para pesquisa no Grafo - O($(V + E) * \log(V)$)

Complexidade Temporal total - O($P * D^2 + T * (V + E) * \log(V)$)

5 Conclusões

Um dos pontos fortes da minha solução é nunca modificar o grafo inicial, não adicionando arcos de peso zero que podem não ser necessários para uma pesquisa posterior.

Uma alternativa estudada foi a utilização da fila de Fibonacci no lugar da fila com prioridade. Mas como expliquei anteriormente, apesar da complexidade amortizada da fila de Fibonacci ficar melhor no papel, para um número de nós tão reduzido não consegui ver um ganho real na performance do algoritmo. Sem dúvida nenhuma que para um grafo de dimensões muito maiores a fila de Fibonacci seria a escolha acertada. Em resumo, ganhei muito ao conhecer e aprender o funcionamento da fila de Fibonacci mas para esta instância deste problema penso que a sua utilização não se justifica.

A Code Listing

A.1 Main.java

```
1  import java.util.*;
2
3  import algorithm.*;
4
5  public class Main {
6
7          private static final int DAYS = 7;
8          private static final int FIRST_CHAR = 'A';
9          private static final int LAST_CHAR = 'Z';
10         private static final int DIF_ACT = LAST_CHAR - FIRST_CHAR + 1;
11         private static final int DIF_COMB = (DAYS * DIF_ACT) + 1;
12         private static final int WITHOUT_ACT = '-';
13         private static final String IMPOSSIBLE = "impossible";
14
15         /**
16          * lfmarques2@gmail.com -> Luis Marques 34213 (FCT/UNL)
17          *
18          * @param args
19          */
20         public static void main(String[] args) {
21
22                 Scanner in = new Scanner(System.in);
23
24                 int people = in.nextInt();
25                 in.nextLine();
26
27                 @SuppressWarnings("unchecked")
28                 List<Pair>[] activities = new List[people];
29
30                 @SuppressWarnings("unchecked")
31                 List<Edge>[] graph = new List[DIF_COMB];
32
33                 short num_vertices = readData(in, people, activities, graph);
34
35                 UniformCost uc = new UniformCost(graph);
36
37                 int questions = in.nextInt();
38
39                 for (; questions-- > 0;) {
40                         int from = in.nextInt();
41                         int to = in.nextInt();
42
43                         boolean[] goalState = new boolean[num_vertices + 1];
44                         for (Pair p : activities[to])
45                                 goalState[p.getVertex()] = true;
46
47                         int result = uc.search(activities[from], goalState);
48
49                         if (result < 0)
50                                 System.out.println(IMPOSSIBLE);
51                         else
52                                 System.out.println(result);
53                 }
54
55         }
```

```
56
57          /**
58           * Reads the input from the Scanner in and populates the list of peoples'
59           * activities and builds the graph.
60           *
61           * @param in
62           * - Scanner of the input file
63           * @param people
64           * - Number of people
65           * @param activities
66           * - Lists of the peoples' activities
67           * @param graph
68           * - Graph
69           * @return the number of nodes of the graph;
70           */
71          private static short readData(Scanner in, int people,
72                          List<Pair>[] activities, List<Edge>[] graph) {
73                  short[][] map = new short[DAYS][DIF_ACT]; // map<(Day,Activitie),
74                                                                                  // Vertex)>
75                  boolean[][] added = new boolean[DIF_COMB][DIF_COMB]; // Edge is in the
76
77                  short num_vertices = 0;
78                  for (int person = 0; person < people; person++) {
79                          activities[person] = new LinkedList<Pair>();
80                          for (int day = 0; day < DAYS; day++) {
81                                  int s = in.next().charAt(0);
82                                  if (s == WITHOUT_ACT)
83                                          continue;
84                                  s -= FIRST_CHAR;
85                                  if (map[day][s] == 0) {
86                                          graph[++num_vertices] = new LinkedList<Edge>();
87                                          map[day][s] = num_vertices;
88                                  }
89                                  short vertex = map[day][s];
90                                  for (Pair p : activities[person]) {
91                                          int otherVertex = p.getVertex();
92                                          if (added[otherVertex][vertex])
93                                                  continue;
94                                          int distance = day - p.getDay();
95                                          graph[otherVertex].add(new EdgeClass(vertex, distance));
96                                          added[otherVertex][vertex] = true;
97                                          graph[vertex].add(new EdgeClass(otherVertex, 7 - distance));
98                                          added[vertex][otherVertex] = true;
99                                  }
100                                 activities[person].add(new PairClass(vertex, day));
101                         }
102                 }
103                 return num_vertices;
104         }
105
106 }
```

A.2 UniformCost.java

```
1  package algorithm;
2
3  import java.util.List;
4  import java.util.PriorityQueue;
5  import java.util.Queue;
```

```
6
7  public class UniformCost {
8
9          private static final int INFINITY = Integer.MAX_VALUE;
10
11         private List<Edge>[] graph;
12
13         public UniformCost(List<Edge>[] graph) {
14                 this.graph = graph;
15         }
16
17         /**
18          * This method discovers and returns what's the minimum weight between any
19          * initial vertex and any end vertex.
20          *
21          * @param list
22          * - List of initial vertex
23          * @param goalState
24          * - Array that has the positions of the end vertexes true and
25          * the others false.
26          * @return the minimum weight between any initial vertex and any end vertex
27          */
28         public int search(List<Pair> list, boolean[] goalState) {
29                 Queue<Vertex> frontier = new PriorityQueue<Vertex>(goalState.length);
30                 int[] cost = new int[goalState.length];
31                 for (int i = 0; i < cost.length; i++)
32                         cost[i] = INFINITY;
33                 for (Pair p : list) {
34                         if (goalState[p.getVertex()])
35                                 return 0;
36                         frontier.add(new Vertex(p.getDay(), p.getDay(), p.getVertex()));
37                         cost[p.getVertex()] = 0;
38                 }
39                 while (!frontier.isEmpty()) {
40                         Vertex v = frontier.remove();
41                         int vertex = v.vertex;
42                         if (goalState[vertex])
43                                 return v.cost;
44                         for (Edge e : graph[v.vertex]) {
45                                 Vertex auxVert = new Vertex(v.initialDay, v.currentDay
46                                                 + e.getWeight(), e.getTo());
47                                 if (cost[auxVert.vertex] > auxVert.cost) {
48                                         if (cost[auxVert.vertex] != INFINITY)
49                                                 frontier.remove(auxVert);
50                                         frontier.add(auxVert);
51                                         cost[auxVert.vertex] = auxVert.cost;
52                                 }
53                         }
54                 }
55                 return -1;
56         }
57
58         private class Vertex implements Comparable<Vertex> {
59                 int initialDay;
60                 int currentDay;
61                 int vertex;
62                 int cost;
63
64                 Vertex(int initialDay, int currentDay, int vertex) {
```

```
65                      this.initialDay = initialDay;
66                      this.currentDay = currentDay;
67                      this.vertex = vertex;
68                      cost = currentDay - initialDay;
69              }
70
71              @Override
72              public int hashCode() {
73                      final int prime = 31;
74                      int result = 1;
75                      result = prime * result + getOuterType().hashCode();
76                      result = prime * result + vertex;
77                      return result;
78              }
79
80              @Override
81              public boolean equals(Object obj) {
82                      if (this == obj)
83                              return true;
84                      if (obj == null)
85                              return false;
86                      if (getClass() != obj.getClass())
87                              return false;
88                      Vertex other = (Vertex) obj;
89                      if (!getOuterType().equals(other.getOuterType()))
90                              return false;
91                      if (vertex != other.vertex)
92                              return false;
93                      return true;
94              }
95
96              @Override
97              public int compareTo(Vertex o) {
98                      if (cost > o.cost)
99                              return 1;
100                     else if (cost == o.cost)
101                             return 0;
102                     return -1;
103             }
104
105             private UniformCost getOuterType() {
106                     return UniformCost.this;
107             }
108
109     }
110
111 }
```

A.3 Interface Edge.java

```
1  package algorithm;
2
3  public interface Edge {
4
5          public int getTo();
6
7          public int getWeight();
8
9  }
```

A.4 Implementação EdgeClass.java

```
1  package algorithm;
2
3  public class EdgeClass implements Edge {
4
5          protected int to;
6          protected int weight;
7
8          public EdgeClass(int to, int weight) {
9                  this.to = to;
10                 this.weight = weight;
11         }
12
13         public int getTo() {
14                 return to;
15         }
16
17         public int getWeight() {
18                 return weight;
19         }
20
21 }
```

A.5 Interface Pair.java

```
1  package algorithm;
2
3  public interface Pair {
4
5          public int getVertex();
6
7          public int getDay();
8
9  }
```

A.6 Implementação PairClass.java

```
1  package algorithm;
2
3  public class PairClass implements Pair {
4          protected int vertex;
5          protected int day;
6
7          public PairClass(int vertex, int day) {
8                  this.vertex = vertex;
9                  this.day = day;
10         }
11
12         public int getVertex() {
13                 return vertex;
14         }
15
16         public int getDay() {
17                 return day;
18         }
19 }
```