Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa
Ano Letivo 2017-2018
Análise e Desenho de Algoritmos
Relatório do Trabalho Prático 2 - The Suspicious Backpack

Inês Ribeiro nº 47226
Sofia Martins nº 47508
Turno Prático 8

Apresentação do Problema

O problema consiste em descobrir, a partir de entrevistas a suspeitos de um crime que ocorreu numa galeria, quem está a passar uma mensagem inconsistente (mentir), partindo de informações que se obtêm nestas.

As informações podem ser de dois tipos: conjecturas de precedência ou de concorrência. As conjecturas de precedência indicam que um suspeito X estava na galeria antes de um suspeito Y. Por outro lado as conjecturas de concorrência indicam, que num dado momento no tempo, 2 suspeitos X e Y estavam ambos na galeria. Por outras palavras, o objetivo será encontrar, a partir um conjunto de conjecturas, as inconsistências.

É dada a informação de quantos suspeitos existem (S), de quantas conjecturas de precedência existem (P) e de quais são e de quantas conjecturas de concorrência existem (C) e quais são. Com esta informação é pretendido chegar à conclusão, após a análise de um grafo, se as conjecturas são consistentes ou não.

Ao observar a informação que é dada, entende-se que este problema de grafos tem como inspiração o Teste à Aciclicidade em grafos orientados. Neste caso, depois de construir corretamente um grafo todas as conjecturas, a solução será ver a aciclidade deste para observar se as conjecturas são consistentes (veracidade das entrevistas).

A chave para esta resolução está em uma boa construção do grafo, que será explicada na secção de Resolução do Problema.

Resolução do Problema

Nota Importante: Para facilitar a compreensão deste capitulo, definem-se os vértices recebidos como letras do alfabeto. No entanto, na realidade do problema, estes são definidos como números inteiros, começando pelo 0(zero) e terminando em suspeitos-1.

Supondo que o problema apresentando tem um número de suspeitos S = 5; cada um deles representado por uma letra do alfabeto de A a E. É também dada uma lista de conjecturas de precedências, neste caso, assumir-se-á que estas serão: A B, C D e C E. As conjecturas de concorrência dadas serão algo como: C B e A D.

Devido a este problema dar importância, não só aos suspeitos, mas também ao tempo que estes se encontram dentro da galeria, por cada suspeitos vão existir 2 eventos representados por 2 vertentes num grafo. Estes 2 eventos serão um evento de entrada do suspeito na galeria e um de saída do suspeito da galeria. Estes 2 eventos estarão ligados da seguinte forma, sendo X cada um dos suspeitos: “X entrou” antes de “X saiu”.

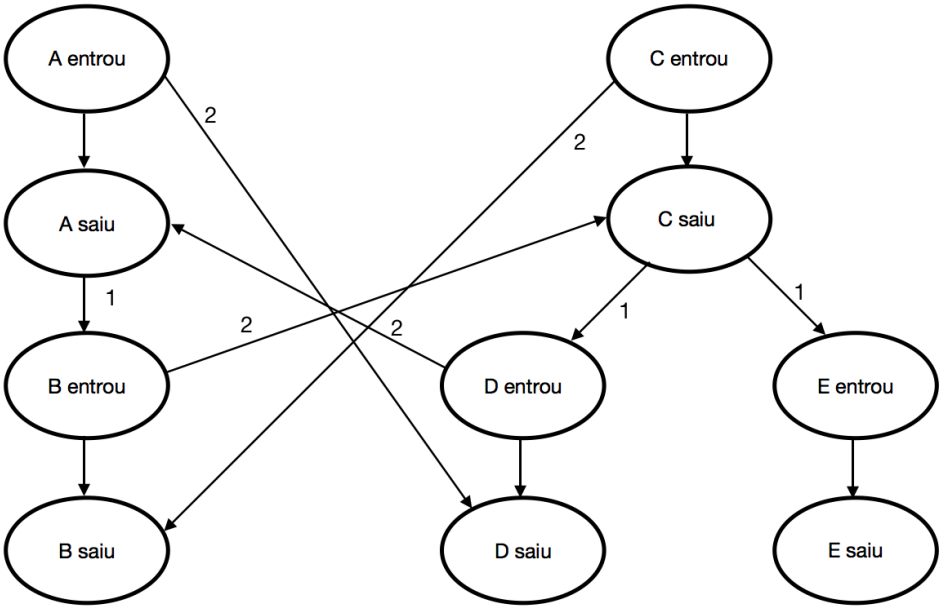
Para representar as conjecturas de precedência no grafo, sendo uma conjectura de precedência do género “X Y”, a ligação feita será entre o evento de saída de X e de entrada do Y.

Para representar as conjecturas de concorrência no grafo, sendo uma conjectura de concorrência do género “X Y”, envolvererá mais do que uma ligação. Será necessário representar que X e Y estiveram ao mesmo tempo na galeria e esse evento poderá ocorrer de duas formas, que serão as duas ligações no grafo. Então, a conjetura de concorrência “X Y” seria representada com a ligação entre “X entrou” e “Y saiu” (X entrou antes de Y sair) e com a ligação entre “Y entrou” e “Y saiu” (X entrou antes de Y sair).

O grafo para a resolução do problema será orientado, podendo ser acíclico ou não.

Sendo o grafo definido como G=(V,A) e tendo em conta o problema apresentado anteriormente nesta secção, vértices V = {A_entrou, A_saiu, B_entrou, B_saiu, C_entrou, C_saiu, D_entrou, D_saiu, E_entrou, E_saiu} e arcos A = < (A_entrou, A_saiu), (B_entrou, B_saiu), (C_entrou, C_saiu), (D_entrou, D_saiu), (E_entrou, E_saiu), (A_saiu, B_entrou)[1], (C_saiu, D_entrou)[1], (C_saiu, E_entrou)[1], (C_entrou, B_saiu)[2], (B_entrou, C_saiu)[2], (A_entrou, D_saiu)[2], (D_entrou, A_saiu)[2]>.

Para facilitar a compreensão, encontra-se em seguida, a representação gráfica deste grafo.



O problema clássico de grafos para resolver este problema será o Teste à Aciclicidade, utilizando o algoritmo isAcyclic - que no código de resolução do problema se chama isConsistent. Existe uma alteração face ao pseudo-código apresentado de resolução deste problema : em vez de um BagIn.. para guardar os nós do grafo, temos uma Queue implementada em LinkedList, o que faz com que para se remover o elemento da Queue seja feito queue.poll().

Implementação do Algoritmo

Estrutura do Programa

Para a resolução deste problema usam-se apenas três classes, uma classe Main, na qual se tratam os inputs dados pelo utilizador e também no output que este irá receber e uma classe SuspBackpack que trata do problema, fazendo tudo o necessário para descobrir a solução do nosso problema e retorná-lo para a classe Main de modo a que esta pudesse tratar do mesmo. E uma classe Graph que trata da construção do grafo e dá as informações necessárias sobre este e sobre os seus nós.

No que diz respeito à classe SuspBackpack, usa-se o método solve, o qual é chamado pela Main e é o encarregue de dar a resposta que se quer. Chamando apenas um método auxiliar, o método isConsistent, que dá o resultado que se pretende obter, através da verificação se o grafo que criamos, tendo em conta as informações que são dadas, é ou não acíclico.

Sobre a classe Main, esta trata dos inputs, chamando um método da classe SuspBackpack, à medida que lê os mesmos, o método chamado da classe SuspBackpack chamado tanto poderá ser o addToGraphPrecedence ou o addToGraphConcorrence, dependendo do input, se o input for uma precedência, irá chamar o primeiro, se for uma concorrência, chamará o segundo. Ambos os métodos limitam-se a chamar um método da classe Graph, onde se irá construir o grafo.

Na classe Graph, para além de criarmos o grafo, esta também contêm informações importantes e úteis sobre os nós do mesmo.

---

[1] Representação num arco de uma conjectura de precedência.

[2] Representação em dois arcos de uma conjectura de concorrência.

Estruturas de Dados

Para a resolução deste problema, na classe Graph, usa-se um Vetor de Listas de inteiros (graph), que será o nosso grafo, um vetor de inteiros (inDegree), guardando este o número de precedências que cada vértice tem, ou seja, o número de arcos que cada vértice tem a chegar ao mesmo, e por fim, temos um inteiro (numSuspects), sendo este o número de suspeitos. Este inteiro equivale ao inteiro (suspects), na classe SuspBackpack. Também nesta classe, tem-se ainda um objeto de Graph. Este, serve para ter todo o acesso necessário para resolver este problema ao nosso grafo.

O vetor de inteiros, inDegree, serve para se saber quantas precedências é que cada vértice tem. Assim dabe-se quando é que esse vértice, que representa um suspeito, pode ou não estar ao mesmo tempo que um outro vértice/suspeito.

O graph/vetor de listas serve para guardar todas as conjecturas de precedências, com a ajuda do vetor de inteiros, inDegree, a conseguir-se saber se um suspeito pode ou não estar ao mesmo tempo que outro. Serve também para guardar quando é que supostamente dois sujeitos estão juntos ao mesmo tempo, para ser possivel saber se alguém mentiu, ou seja, se o grafo é ou não consistente.

Pontos Importantes no Código

Para a resolução deste problema usam-se apenas duas classes: a classe Main e a classe SuspBackPack. A classe Main é a classe onde se tratam os inputs dados pelo utilizador e também o output que este irá receber. A classe SuspBackpack trata do problema, fazendo tudo o necessário para descobrir a solução do problema e retorná-la para a classe Main de modo a que esta possa tratar da mesma.

No que diz respeito à classe SuspBackpack, usa-se o método solve, o qual é chamado pela Main. Este é o método encarregue de dar a resposta que queremos, consultando apenas um boolean gerado por um método auxiliar, isConsistent, o qual se baseia no algoritmo para verificar se um grafo é ou não acíclico.

Uma das partes mais relevantes do código são os métodos addToGraphPrecedence e addToGraphConcorrence, na classe Graph, uma vez que são eles que vão construindo o nosso grafo. Na Main, é chamado cada um deles quando é suposto, ou seja, enquanto os inputs são conjecturas de precedência, chama-se o primeiro, quando essas acabam e passam a ser conjecturas de concorrência, passa-se a chamar o segundo.

Outro ponto importante a referir é como é possivel identificar se um vértice do grafo é um nó de entrada ou de saída. Visto que cada nó que se recebe é um inteiro e sabe-se qual o número total de nós à partida, a codificação dos nós de entrada é a que é dada pelo problema e a codificação do nó de saida desse mesmo vértice é o número do nó dado somando o número de vértices totais.

```
private int codeOut(int node) {
                return node + numSuspects;
        }
```

Legenda: codificação dos nós de saída.

Existem também mais partes relevantes na inserção das conjecturas de precedência vs. a inserção das conjecturas de concorrência no grafo. Na inserção das conjeturas de precedências, como explicado na secção de Resolução do Problema, o objetivo é ligar o nó de saída do pai ao nó de entrada do filho. No entanto, ao receber um vértice, quer seja pai ou filho, não se tem a certeza se os nós de entrada e de saída deste já estão ligados ou não, nunca podendo assumir apenas um dos casos. Por isso, antes de se adicionar esta ligação, é necessário verificar se esta já existe.

Na inserção das conjeturas de concorrência, como explicado na secção de Resolução do Problema, tem como objetivo, recebendo 2 vértices X e Y, ligar o nó de entrada do vértice X ao nó de saída do vértice Y e o nó de entrada do vértice Y ao nó de saída do vértice X. No entanto, tal como anteriormente, é necessário verificar, tanto para o nó X como para o Y, se o vértice de entrada deste já está ligado ao vértice de saída dele mesmo.

Só desta forma é que se pode assegurar que existe um número certo de ligações no grafo, nem a mais, nem a menos.

```
public void addToGraphPrecedence(int parent, int child) {
		int outParent = codeOut(parent);
		int outChild = codeOut(child);

		if( graph[parent].isEmpty() ) {
			graph[parent].add(outParent);
			inDegree[outParent]++;
		}
		graph[outParent].add(child);
		inDegree[child]++;

		if( graph[child].isEmpty() ) {
			graph[child].add(outChild);
			inDegree[outChild]++;
		}
}
```

```
public void addToGraphConcorrence(int firstConcurrent, int
secondConcurrent) {
		int outFirst = codeOut(firstConcurrent);
		int outSecond = codeOut(secondConcurrent);

		if( graph[firstConcurrent].isEmpty() ) {
			graph[firstConcurrent].add(outFirst);
			inDegree[outFirst]++;
		}

		if( graph[secondConcurrent].isEmpty() ) {
		graph[secondConcurrent].add(outSecond);
		inDegree[outSecond]++;
		}

		graph[firstConcurrent].add(outSecond);
		inDegree[outSecond]++;

		graph[secondConcurrent].add(outFirst);
		inDegree[outFirst]++;
}
```

Legenda: código de ambos os métodos para facilitar a compreensão da explicação

Análise do Algoritmo

Complexidade Espacial

Sendo o M o número de suspeitos e N o número conjecturas de precedências e/ou concorrências do nó a quem a mesma pertence:

int suspects (em SuspBagpack - linha 21)	O(1)
Graph graph (em SuspBagpack - linha 17)	O(1)
int numSuspects (em Graph - linha 25)	O(1)
int[] inDegree (em Graph - linha 20)	O(M)
List<Integer>[] graph (em Graph - linha 15)	O(2 * M * N) = O(M * N)

Tendo em conta as Complexidades Espaciais a cima representadas, chega-se à conclusão que a nossa complexidade espacial será de O(2 * M * N), ou seja, **O(M * N) = O(MN)**.

Esta tem este valor uma vez que é a maior complexidade espacial que se tem - M * N > M.

O seu valor é devido ao vetor de listas, uma vez que o nosso vetor terá 2 * M (número de suspeitos) e o tamanho das listas varia tendo em conta o número de conjecturas precedência e/ou concorrência, sendo este no máximo N.

Complexidade temporal

Sendo o M o número de suspeitos e N o número de precedências e/ou concorrências do nó a quem a mesma pertence.

Na classe *Graph*:

Construtor	O(2M) = O(M)
private int codeOut	O(1)
private void addToGraphPrecedence	O(1)
private void addToGraphConcorrence	O(1)
private int getInDegree	O(1)
private int numNodes	O(1)
public List<Integer> outAdjacentNodes	O(1)

Ficando a complexidade temporal O(M), para já.

Na classe *SuspBackPack*, sendo A o número de arcos:

Construtor	O(1)
private boolean isConsistent	O(2M+A)=O(M+A)
private void addToGraphPrecedence	O(1)
private void addToGraphConcorrence	O(1)
private String solve	O(1)

Então, a complexidade temporal total fica O(M + A), para já, visto que M+A > M.

Esta complexidade deve-se ao facto de serem percorridos uma vez todos os nós do grafo, observando-se quais não têm dependências, metendo-os numa lista (ready), vindo daqui 2M=M (tendo em conta que as constantes caiem). De seguida, percorrem-se todos os arcos aos quais os nós pertencentes pertencem a essa mesma lista ready, não percorrendo mais que o número de arcos totais do nosso grafo - A.

Concluiu-se então que a complexidade temporal é **O(M+A)**.

Conclusões

Ao analisar o problema pela primeira vez, consideramos que a solução deste seria inspirada no problema da ordenação topológica, fazendo a cada passo a verificação se não existia inconsistências nos nós. No entanto, apercebemo-nos que a solução não poderia ser utilizando a ordenação topológica, pois esta verificação não resultava. Para além disso, resolvendo o problema desta forma, as complexidades temporal e espacial seriam demasiado elevadas para a solução.

Para além desta, outra solução alternativa que consideramos passava por não ter o grafo guardado numa classe à parte, que era aceite no mooshak, mas não era tão organizada como a solução a que chegámos.

Acreditamos que um ponto forte da nossa solução são as complexidades temporal e espacial desta. São complexidades que, comparadas com o problema original do Teste à Aciclicidade, são muito próximas, ou até mesmo iguais (complexidade temporal), visto que usamos um vetor de listas para implementar o grafo.

```
1. import java.io.BufferedReader;
2. import java.io.IOException;
3. import java.io.InputStreamReader;
4.
5. import suspBackpack.SuspBackpack;
6. /**
7.  * Main class
8.  * @author Ines Ribeiro 47226 Sofia Martins 47508
9.  *
10. */
11.public class Main {
12.
13.  public static void main(String[] args) throws IOException {
14.        BufferedReader sc =
15.                  new BufferedReader( new
   InputStreamReader(System.in) );
16.        String line[] = sc.readLine().split(" ");
17.        int details[] = new int[3];
18.        for(int i = 0; i < line.length; i ++) {
19.             details[i] = Integer.parseInt(line[i]);
20.        }
21.
22.        SuspBackpack sb = new SuspBackpack(details[0]);
23.        String[] lineProblem;
24.
25.        for(int i = 0; i < details[1] +  details[2]; i++) {
26.             lineProblem = sc.readLine().split(" ");
27.             if(i < details[1]) {
28.
   sb.addToGraphPrecedence(Integer.parseInt(lineProblem[0]),
   Integer.parseInt(lineProblem[1]));
29.             }else {
30.
   sb.addToGraphConcorrence(Integer.parseInt(lineProblem[0]),
   Integer.parseInt(lineProblem[1]));
31.             }
32.        }
33.        sc.close();
34.        String result = sb.solve();
35.        System.out.println(result);
36.  }
37.}
38.
```

```
1. package suspBackpack;
2.
3. import java.util.LinkedList;
4. import java.util.Queue;
5.
6. import graph.Graph;
7.
8. /**
9.  * Class to solve the Suspicious Backpack problem.
10. * @author Ines Ribeiro 47226 Sofia Martins 47508
11. */
12.public class SuspBackpack {
13.
14.   /**
15.    * Graph that represents the problem.
16.    */
17.   private Graph graph;
18.   /**
19.    * Number of suspects
20.    */
21.   private int suspects;
22.
23.   public SuspBackpack(int s) {
24.         suspects = s;
25.         graph = new Graph(s);
26.   }
27.
28.   /**
29.    * Tests if the graph is acyclic.
30.    * @return true if it is acyclic, therefore is consistent
31.    */
32.   private boolean isConsistent() {
33.         int nodesFound = 0;
34.         int[] inDegree = new int[2* suspects];
35.         Queue<Integer> ready = new LinkedList<Integer>();
36.         for ( int i = 0 ; i < 2 * suspects ; i++ ) {
37.               inDegree[i] = graph.getInDegree(i);
38.               if ( graph.getInDegree(i) == 0 )
39.                     ready.add(i);
40.         }
41.         do {
42.               int node = ready.poll();
43.               nodesFound++;
44.               for ( int v : graph.outAdjacentNodes(node)){
45.                     inDegree[v] --;
46.                     if ( inDegree[v] == 0 )
47.                           ready.add(v);
48.               }
49.         } while ( !ready.isEmpty() );
50.         return nodesFound == graph.numNodes();
51.   }
52.
53.   /**
54.    * Adds a precedence to the graph.
55.    * @param parent is the parent node
56.    * @param child is the child node
57.    */
58.   public void addToGraphPrecedence(int parent, int child) {
59.         graph.addToGraphPrecedence(parent, child);
60.   }
```

```
61.
62.   /**
63.    * Adds a concurrence to the graph, ie., two nodes that need to be in the
   graph at the same time.
64.    * @param firstConcurrent is one of the nodes.
65.    * @param secondConcorrent is the other node.
66.    */
67.   public void addToGraphConcorrence(int firstConcurrent, int
   secondConcurrent) {
68.         graph.addToGraphConcorrence(firstConcurrent, secondConcurrent);
69.   }
70.
71.
72.   /**
73.    * If the graph is acyclic then the conjunctures are consistent. Else,
   they ate not.
74.    * @return String of response.
75.    */
76.   public String solve() {
77.         boolean isAcyclic = isConsistent();
78.         if(isAcyclic)
79.               return "Consistent conjectures";
80.         else
81.               return "Inconsistent conjectures";
82.   }
83.}
84.
```

```
1. package graph;
2.
3. import java.util.LinkedList;
4. import java.util.List;
5.
6. /**
7.  * Class of the graph
8.  * @author Ines Ribeiro 47226 Sofia Martins 47508
9.  */
10.public class Graph {
11.
12.   /**
13.    * Graph that represents the problem.
14.    */
15.   private List<Integer>[] graph;
16.
17.   /**
18.    * Used to register how many inward edges the node has.
19.    */
20.   private int[] inDegree;
21.
22.   /**
23.    * number of nodes the graph
24.    */
25.   private int numSuspects;
26.
27.   @SuppressWarnings("unchecked")
28.   public Graph(int size) {
29.         numSuspects = size;
30.         inDegree = new int[2* numSuspects];
31.         graph = new List[2 * numSuspects];
32.
33.         for( int i = 0; i < (2 * numSuspects); i++ ) {
34.               graph[i] = new LinkedList<Integer>();
35.               inDegree[i] = 0;
36.         }
37.   }
38.
39.   /**
40.    * Generates the index of an outNode
41.    * @param node index of the nodeIn
42.    * @return the index
43.    */
44.   private int codeOut(int node) {
45.         return node + numSuspects;
46.   }
47.
48.   /**
49.    * Adds a precedence to the graph.
50.    * @param parent is the parent node
51.    * @param child is the child node
52.    */
53.   public void addToGraphPrecedence(int parent, int child) {
54.         int outParent = codeOut(parent);
55.         int outChild = codeOut(child);
56.
57.         if( graph[parent].isEmpty() ) {
58.               graph[parent].add(outParent);
59.               inDegree[outParent]++;
60.         }
```

```
61.         graph[outParent].add(child);
62.         inDegree[child]++;
63.
64.         if( graph[child].isEmpty() ) {
65.               graph[child].add(outChild);
66.               inDegree[outChild]++;
67.         }
68.   }
69.
70.   /**
71.    * Adds a concurrence to the graph, ie., two nodes that need to be in the
   graph at the same time.
72.    * @param firstConcurrent is one of the nodes.
73.    * @param secondConcorrent is the other node.
74.    */
75.   public void addToGraphConcorrence(int firstConcurrent, int
   secondConcurrent) {
76.         int outFirst = codeOut(firstConcurrent);
77.         int outSecond = codeOut(secondConcurrent);
78.
79.         if( graph[firstConcurrent].isEmpty() ) {
80.               graph[firstConcurrent].add(outFirst);
81.               inDegree[outFirst]++;
82.         }
83.
84.         if( graph[secondConcurrent].isEmpty() ) {
85.               graph[secondConcurrent].add(outSecond);
86.               inDegree[outSecond]++;
87.         }
88.
89.         graph[firstConcurrent].add(outSecond);
90.         inDegree[outSecond]++;
91.
92.         graph[secondConcurrent].add(outFirst);
93.         inDegree[outFirst]++;
94.   }
95.
96.   /**
97.    *
98.    * @param node node that we want to know how many inward edges the node
   has
99.    * @return the in-degree of the specified node
100.   */
101.  public int getInDegree(int node) {
102.        int inDegreeNode = inDegree[node];
103.        return inDegreeNode;
104.  }
105.
106.  /**
107.   *
108.   * @return the number of nodes
109.   */
110.  public int numNodes() {
111.        return graph.length;
112.  }
113.
114.  /**
115.   *
116.   * @param node the node that we want know all adjacent nodes
```

```
117.    * @return the nodes adjacent to the specified node along outgoing edges
   from it
118.    */
119.   public List<Integer> outAdjacentNodes(int node){
120.         List<Integer> adjacentNodes = graph[node];
121.         return adjacentNodes;
122.   }
123.
124.}
125.
```