Departamento de Informática

Análise e Desenvolvimento de Algoritmos 2012/2013

Relatório
Trabalho Prático 3
Book Delivery

Docente: Sofia Cavaco

Turno prático 3

Henrique Leandres N°34802

João Francisco Pinto N°35318

Apresentação do Problema

O problema Book Delivery consiste no cálculo de, dado um horário semanal de um número de pessoas pertencentes a um grupo de leitura, onde cada uma só efectua uma actividade por dia, calcula o número necessário de dias para transitar um livro entre duas pessoas nesse horário, via um sistema de passagem, assumindo que o livro é entregue a um elemento quando este se encontra numa actividade no mesmo dia que outro.

Para a resolução deste problema é necessário portanto, saber quantas pessoas estão incluídas no grupo de leitura, os seus horários semanais, e as pessoas entre quais queremos saber o número de dias necessário para o livro chegar de uma a outra.

Dados estes valores, o problema responde “*impossible*”, caso não haja uma ligação entre as duas pessoas dadas como argumentos, ou o número mínimo de dias para o livro chegar a um elemento do grupo, partindo de outro, a partir do primeiro dia quando o livro é entregue a outro elemento.

Resolução do Problema

Para a resolução deste problema representámos o horário em forma de grafo pesado, onde um vértice representa uma actividade num dado dia, um arco liga dois vértices quando uma pessoa transita de uma actividade para outra, e o peso de um arco é igual ao número inteiro de dias dessa transição.

A solução do problema é obtida através da aplicação do algoritmo de Dijkstra, sobre o grafo, onde as pessoas de quem parte e a quem se pretende que chegue o livro são representadas por dois nós virtuais introduzidos no grafo, com ligações de peso zero às suas respectivas actividades diárias.

O resultado do problema é impossível caso o caminho mínimo dado pelo algoritmo entre a origem e o destino do livro seja igual a $+\infty$; caso contrário, é o valor inteiro devolvido.

Implementação do Algoritmo

Implementámos este problema de grafos utilizando duas estruturas: um mapa que traduzia chaves inteiro/caracter, que representavam pares dia/actividade, em inteiros, que representam os vértices do grafo em si, este implementado através de um array de mapas, com base na implementação em listas de adjacências: cada mapa é efectivamente uma lista de adjacências, do vértice correspondente à sua posição num array, e cada entrada chave/valor corresponde a um arco para um vértice, onde a chave é o vértice de chegada, e o valor o peso do arco.

Escolhemos utilizar mapas em vez de listas para suprimir repetições de arcos, e um mapa permite fazer este controlo em tempo constante.

Quanto à implementação em concreto, a classe `Main` é responsável pela leitura dos dados: o horário introduzido é dado como argumento à criação da classe `bookDelivery`, que interpreta o horário e constrói o grafo. As posteriores perguntas são calculadas pelo método `solve()`, da classe `bookDelivery`, que introduz os vértices de destino e origem no grafo, aplica-lhe o algoritmo de Dijkstra, limpa os dados respectivos a essa pergunta em concreto, e devolve o resultado obtido.

O algoritmo de dijkstra foi implementado sobre uma fila de Fibonacci. Quanto a esta estrutura de dados, foram implementados quatro métodos, apresentados em anexo: `removeMinTree()`, `rebuildQueue()`, `linkTrees()` e `removeChild()`. Os restantes métodos da classe foram fornecidos.

Análise do Algoritmo

Complexidade Espacial

A complexidade espacial do nosso algoritmo corresponde à dimensão das estruturas de dados criadas para a representação do grafo atividade/dia, incluindo os nós de destino e origem, e o mapa que traduz pares dia/atividade em vértices.

Sabendo que a complexidade espacial da lista de adjacências é $\Theta(|V|+|A|)$, e que a complexidade espacial do HashMap é $\Theta(N)$ temos que a nossa implementação tem complexidade espacial $\Theta(|da+2|+|a|) + \Theta(|da+1|) = \Theta(|da+2|+|a|)$, sendo da o número de pares dia/atividade, e a este são somados os vértices de origem e destino, na construção do mapa de adjacências, e o vértice de destino no mapa que traduz dia/atividade em vértice. da é igual a 1 no melhor caso, e 182 no pior caso (26×7), e a o total de arcos do grafo dia/atividade que representa o horário.

Complexidade Temporal

A complexidade temporal do nosso algoritmo é definida pela complexidade temporal do algoritmo de Dijkstra com fila de Fibonacci, que é $\Theta(|A|+|V| \times \log |V|)$ em todos os casos. No nosso caso em particular, a complexidade temporal da nossa solução é $\Theta(|a|+|da+1| \times \log |da+1|)$, porque sabemos que o nó de origem nunca é visitado, pois ao criar o grafo, nenhum arco incide nele.

Conclusões

Embora a nossa solução não inclua uma interface para a classe Book Delivery, esta apenas possui um método público, que é a resolução do problema.

A utilização do mapa auxiliar para traduzir pares dia/actividade simplifica a resolução do problema, mas poderia ter sido definido localmente no construtor, pois só é utilizado fora deste quando queremos saber o número do vértice de destino, que é guardado numa variável global.

A utilização de um array de mapas para suprimir repetições alivia a execução do algoritmo de Dijkstra, porque se não se controlassem as repetições, sempre que duas pessoas tivessem actividades iguais em dias iguais em consecutivo (por exemplo, as mesmas actividades segunda, terça e quinta-feira, ou qualquer subsequência destas) existiria um arco repetido. O grafo apenas não controla quando existem vértices intermédios entre dois vértices.

Na resolução do problema, preferimos remover manualmente as ligações aos vértices origem e destino do array de mapas em oposição a criar um clone do grafo por cada problema, embora a remoção dos arcos para o destino seja linear.

Anexo

Classe Main

```
import java.util.*;

/**
 * @author Henrique Leandres Nº 34802
 * @author Joao Pinto Nº 35318
 */

public class Main {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        //Declaracao das variaveis do input
        int p = 0, q = 0, s = 0, d = 0;

        p = Integer.parseInt(in.nextLine());

        //Declaracao da matriz que representa o horario;
        char[][] schedule = new char[p][7];
        String[] temp = null;

        //Leitura dos horarios de cada pessoa
        for(int i = 0; i < p; i++){
            temp = in.nextLine().split(" ");
            for(int j = 0; j < 7; j++){
                schedule[i][j] = temp[j].charAt(0);
            }
        }

        q = in.nextInt();
        in.nextLine();

        int tmp = 0;
        bookDelivery problem = new bookDelivery(p,schedule);

        //Leitura dos casos de teste
        for(int i = 0; i < q; i++){
            s = in.nextInt();
            d = in.nextInt();
            in.nextLine();

            tmp = problem.solve(s,d);

            if(tmp == Integer.MAX_VALUE)
                System.out.println("impossible");
            else
                System.out.println(tmp);
        }

        in.close();
    }
}
```



```

//criar lista de adjacencias de sucessores para guardar ligacoes
//entre vertices
graph = new ArrayList<HashMap<Integer,Integer>>
        (destinationVertex+1);

for(int i = 0; i < destinationVertex+1; i++){
    graph.add(new HashMap<Integer,Integer>());
}

//preencher lista de sucessores

//tratamento linha a linha
for(int i = 0; i < p; i++){

    boolean first = true, loop = true;
    int j = 0, previous = -1, firstPos = -1;
    int emptyLine = 0;
    while(loop){

//        System.out.println("pessoa: "+i);
        if(schedule[i][j] != '-'){
            if(first){
                first = false;
                previous = firstPos = j;
            }
            else if(j == firstPos && firstPos != previous){
                //vai buscar ao dicionario o numero do
                //vertice anterior dando o dia e o caracter
                //nesse dia
                int prevVertex = map.get(new
                Pair<Integer,Character>
                (previous,schedule[i][previous]));

                //vai buscar ao dicionario o numero do
                //vertice do dia que esta a ser visitado na
                //iteracao e o caracter desse dia
                //System.out.println(map.get(
                //new Pair<Integer,Character>(j,schedule[i]
                [j]]));

                int vertex = map.get(
                    new
                Pair<Integer,Character>(firstPos,schedule[i][firstPos]));

                //calcula a diferenca entre os dias: como
                //nao deu a
                //volta, j > previous
                int length = j + 7 - previous;

                //adicionar current (j) ao previous;
                System.out.println(prevVertex+"->"+vertex);
                graph.get(prevVertex).put(vertex,length);
                break;
            }
            //Caso em que pessoa so tem uma actividade num so
            //dia
            else{
                //vai buscar ao dicionario o numero do
                //vertice anterior dando o dia e o caracter
                //nesse dia
                int prevVertex = map.get(new
                Pair<Integer,Character>

```



```

        min = null;
    }
    //Senao, liga os irmaos do min um ao outro, e poe o direito como min
    else{
        FibNode<K, V> rightSibling = min.getRightSibling();

        min.getLeftSibling().setRightSibling(min.getRightSibling());
        min.getRightSibling().setLeftSibling(min.getLeftSibling());

        min = rightSibling;
    }
}

// Rebuilds the queue with the trees in the specified array.
// Pre-condition: the queue is not empty.
protected void rebuildQueue( FibNode<K,V>[] trees )
{
    min = null;

    for(int i = 0; i < trees.length; i++){
        //Verifca se ha arvore com esse grau
        if(trees[i] != null){
            //Se min estiver a null, esta na primeira arvore e inicializa
            if(min == null){
                min = trees[i];
                min.makeSingleton();
            }
            //Senao ja ta pelo menos na segunda arvore, e atualiza irmaos
            //e talvez min
            else{

                FibNode<K, V> rightSibling = min.getRightSibling();

                trees[i].setLeftSibling(min);
                trees[i].setRightSibling(rightSibling);

                min.setRightSibling(trees[i]);
                rightSibling.setLeftSibling(trees[i]);

                //Verifica se nova arvore e min
                if(trees[i].getKey().compareTo(min.getKey()) < 0){
                    min = trees[i];
                }
            }
        }
    }
}

// Links two trees of degree k and
// returns the resulting tree of degree k + 1.
protected FibNode<K,V> linkTrees( FibNode<K,V> tree1, FibNode<K,V> tree2 )
{
    tree1.makeSingleton();
    tree2.makeSingleton();

    FibNode<K,V> root;
    FibNode<K,V> child;

    //Define a raiz da nova arvore e o filho a adiciona-la
    if(tree1.getKey().compareTo(tree2.getKey()) < 0){

```

```

        root = tree1;
        child = tree2;
    }
    else{
        root = tree2;
        child = tree1;
    }

    child.setParent(root);

    //Se raiz tiver grau um, aponta para a outra arvore e arvore
    //passa a ser filho unico
    if(root.getDegree() == 0){
        root.setChild(child);
    }
    else{
        FibNode<K, V> rootChild = root.getChild();
        FibNode<K, V> rootChildSibling = root.getChild().getRightSibling();

        child.setLeftSibling(rootChild);
        child.setRightSibling(rootChildSibling);

        rootChild.setRightSibling(child);
        rootChildSibling.setLeftSibling(child);
    }

    root.incrementDegree();
    return root;
}

// Removes the specified child from the list of children
// of the specified parent.
protected void removeChild( FibNode<K,V> parent, FibNode<K,V> child )
{
    //Verifica se child e filho de parent
    if(child.getParent() == parent){
        //Verifica se pai aponta para esse filho
        if(parent.getChild() == child){
            //Se irmao direito do filho for igual a si proprio e filho
            unico
                //e pai passa a ter child como null
                if(child.getRightSibling() == child){
                    parent.setChild(null);
                }
                //Senao, coloca pai a apontar para irmao direito, e actualiza
                //irmaos do no a remover
                else{
                    parent.setChild(child.getRightSibling());

                    child.getLeftSibling().setRightSibling(child.getRightSibling());
                    child.getRightSibling().setLeftSibling(child.getLeftSibling());
                }
                //Senao, tem pelo menos um irmao, e poe os irmaos a apontar um para
                //o outro
                else{
                    child.getLeftSibling().setRightSibling(child.getRightSibling());
                    child.getRightSibling().setLeftSibling(child.getLeftSibl
ing());

```

```
        }  
        child.setParent(null);  
        parent.decrementDegree();  
    }  
}
```