

Relatório de ADA

Trabalho 3

Miguel Ferrão Ferreira, nº 41983

Tiago dos Reis Morais da Silva, nº 42056

Análise do Problema

O problema Polygon Phobia

Bob é um jovem artista cujos trabalhos actuais são pinturas com segmentos de recta coloridos. No entanto, o Bob tem fobia a polígonos, ou seja, simplesmente não consegue desenhar qualquer cadeia de segmentos de recta fechada.

De modo a contornar este problema, no seu próximo workshop de pintura Bob experimentou uma nova aproximação ao problema, “a sorte do desenho”. Ele começará por escolher com cuidado os possíveis segmentos de recta e escolhendo dois pontos extremos para cada um deles num pedaço de papel, como se de um sistema de coordenadas cartesiano se tratasse.

De seguida, todos esses pedaços de papel serão colocados num grande vaso, que será bem agitado. Neste ponto, o processo de pintura começará, usando a sua aproximação “a sorte do desenho”. Enquanto o vaso não estiver vazio, Bob retirará um pedaço de papel do mesmo, um de cada vez, e pintará o correspondente segmento de recta a não ser que este feche uma cadeia existente (devido à sua fobia). Todos os papéis retirados são colocados na reciclagem. Um segmento p1,pn fecha uma cadeia se já existir um desenho com os segmentos na forma:

**p1,p2, p2,p3, p3,p4, ..., pn−1,pn**

onde p,q denota um segmento de recta cujos pontos extremos são p e q. De notar que p,q = q,p. Também é de relevar que cadeias são definidas por pontos extremos e não por intersecções.

Resolução do Problema

A resolução deste problema passa pela correta implementação de um algoritmo que evite a existência de cadeias fechadas. Para tal foi utilizado o algoritmo de representante com compressão do caminho, pois este permite Se estas regras forem bem aplicadas o resultado obtido estará dependente dos vértices e dos correspondentes antecessores e sucessores.

A resolução do problema separou-se assim em diversos casos, que estavam condicionados pelas regras e pela ordem dos vértices (quanto menor o seu número, maior a prioridade em relação aos demais):

Regra a): Nesta regra tivemos que ter em conta a existência dos vértices sem antecessores, não tratados e com um número inferior (exemplo vértice 3), e a existência de sucessores (de um outro vértice) não tratados mas com um número superior (exemplo vértice 6). Neste caso os sucessores têm prioridade, mesmo sobre vértices com um menor número.

Um problema que surgiria da aplicação desta regra era o tratamento de um sucessor de um vértice, que tivesse uma dependência de sucessor com outros vértices não tratados. Este caso levava a resultados incorretos. A solução passou por só se poder tratar o vértice sucessor, no caso em que todos os seus antecessores já tinham sido vistos e tratados.

Tratado este problema, a regra a) estava resolvida e podia ser bem aplicada.

Regra b): Esta regra aborda o caso em que um vértice tem mais do que um sucessor. Neste caso os sucessores devem ser tratados tal como referido pela regra a) (do menor para o maior valor), e para isso é preciso ordenar a ordem dos sucessores de maneira a que o vértice com menor valor seja o primeiro a ser tratado. Feita a ordenação será necessário verificar as condições impostas pela regra a).

Foi nesta linha de pensamento que desenvolvemos o algoritmo que está implementado no método friendlyManuals.

Implementação do Algoritmo

Para implementar o algoritmo foi necessário usar uma classe Main e uma classe UnionFindInArray fornecida pela professora, com a respectiva interface e três classes onde estão implementadas três excepções. Na classe Main estão implementados dois métodos: o método main onde são inseridas as coordenadas do vértices dos segmentos de recta através do input e retornado o output, ou seja, o número de segmentos de recta formados, sem a existência de cadeias fechadas, e o método PolygonPhobia onde é implementado o algoritmo em si. Na classe UnionFindInArray estavam implementados todos os métodos de união: união por altura, união por tamanho e de representante com compressão do caminho, tendo nós de escolher o que achássemos adequado.

Classe Main

Método main:

O input é colocado num buffer, sendo que na primeira linha são colocados o número de segmentos de recta a ser desenhados. Nas seguintes linhas são colocadas as coordenadas dos vértices do segmento de recta: x1 e y1 são as coordenadas de um dos vértices e x2 e y2 do outro vértice. É usado um HashMap para representar os pontos, sendo a string as coordenadas do ponto e o Integer o valor do ponto em si. É também usada uma fila implementada em LinkedList para guardar os pontos em si. Caso o ponto de coordenadas x1 e y1 em questão ainda não exista no HashMap, este é adicionado ao mesmo, é adicionado o valor do contador à lista e o contador é incrementado em uma unidade. Caso contrário, esse ponto é adicionado à lista. O mesmo método e usado para o ponto de coordenadas x2 e y2. Isto é feito até se terem lidos todas as linhas do input.

Método PolygonPhobia:

É neste método que vai ser implementado o algoritmo pretendido para este problema. É criado um objecto do tipo UnionFindInArray, chamado union, que referencia a classe fornecida pela docente onde estão implementados os métodos da união. É também criada uma variável result que guarda o resultado, ou seja, a quantidade de segmentos de recta já desenhados, que não fecham cadeias.

Análise do Algoritmo

Complexidade temporal

A complexidade temporal do método Main, ou seja da leitura do input, está dependente do número de vértices ou do número de arcos do grafo em questão. Isto é caso o número de vértices seja maior que o número de arcos, então a complexidade temporal depende do número total de vértices, pois entramos num ciclo em que se inicializam vetores com tantas posições como o número de vértices, caso contrário depende do número de arcos devido ao mesmo caso.

Assim o melhor caso para este método é quando temos E = 2, D =1 com E = vértices e D = arcos:

Θ(E)

No pior caso tem-se:

Se E >D Θ(E)

Se E<D Θ(D)

A complexidade temporal do método FriendlyManuals, ou seja da execução do algoritmo, está dependente do número de vértices, pois todos os vértices terão de ser tratados, o que resulta na entrada de um ciclo, em que o seu número de iterações e igual ao número de vértices.

Apesar de dentro deste ciclo referido em cima, existirem outros ciclos a serem executados, bem como a possível ordenação de uma fila estes atingem sempre complexidades temporais inferiores.

Assim, melhor/pior caso para este método têm complexidade temporal:

Θ(E), com E = número de vértices do grafo.

Concluindo o programa tem complexidade temporal da ordem Θ(n).

Complexidade espacial

A complexidade espacial, está dependente das variáveis e estruturas de dados (listas e vetores) utilizadas para guardar as informações relativas ao grafo. Assim, visto que as estruturas de dados guardam informações referentes aos vértices (seus sucessores, antecessores, se foi visto e/ou se foi tratado), o seu tamanho e consequente gasto de memória, está diretamente relacionado com o tamanho do grafo, mais precisamente com o número de vértices.

Com isto pode-se dizer-se que cada estrutura tem a capacidade, para o numero de vértices existentes no grafo, logo:

(para cada estrutura) Θ(E), com E = número de vértices do grafo.

Assim, o espaço ocupado em memória estará dependente do número de estruturas utilizadas no programa e a complexidade espacial de cada uma, e assim obtemos:

Se para cada estrutura a complexidade é Θ(E), com E = número de vértices do grafo.

Então temos, nrTADS \* Θ(E)

De onde podemos concluir que a complexidade espacial do programa é Θ(n).

Conclusões

Em relação aos pontos fracos, não podemos dizer grande coisa, pois não fizemos qualquer medição do tempo de cálculo da execução do algoritmo e também não houve outras alternativas a serem pensadas. As estruturas de dados que se utilizaram também pareceram ser as mais corretas.

Este foi o mais simples dos três trabalhos, sendo que o algoritmo em si já nos era dado pela professora. Tivemos apenas que implementar a parte do input e o método polygonPhobia, usando a classe dada pela professora de forma correcta.

A implementação do problema em questão não nos causou grandes problemas em termos de submissão no Mooshak.

Anexo

**Classe Main**

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**import** java.util.Queue;

**import** java.util.StringTokenizer;

**import** dataStructures.UnionFindInArray;

/\*\*

\* **@author** Miguel Ferrão Ferreira nº 41983 e Tiago dos Reis Morais da Silva nº 42056

\*

\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) **throws** NumberFormatException, IOException {

BufferedReader buffer = **new** BufferedReader(**new** InputStreamReader(System.in));

String str = buffer.readLine();

StringTokenizer token = **new** StringTokenizer(str);

**int** lineSegments = Integer.parseInt(token.nextToken()); //Le o numero de linhas

//Cria um HashMap em que a chave e uma String(coordenadas) e o valor e um Integer(0, 1, 2,...)

HashMap<String, Integer> points = **new** HashMap<String, Integer>();

Queue<Integer> p = **new** LinkedList<Integer>();

**int** counter = 0; //Variavel que atribuira os inteiros a cada ponto

**for**(**int** i = 0; i < lineSegments; i++){ // Enquanto existirem linhas

String str1 = buffer.readLine();

StringTokenizer token1 = **new** StringTokenizer(str1); // Leitura das coordenadas dos extremos do segmento

**int** x1 = Integer.parseInt(token1.nextToken());

**int** y1 = Integer.parseInt(token1.nextToken());

**int** x2 = Integer.parseInt(token1.nextToken());

**int** y2 = Integer.parseInt(token1.nextToken());

String coord1 = String.valueOf(x1) + " " + String.valueOf(y1);//Constroi a string que contem as coordenadas do primeiro ponto

String coord2 = String.valueOf(x2) + " " + String.valueOf(y2);//Constroi a string que contem as coordenadas do segundo ponto

**if**(!points.containsKey(coord1)){ //Se o primeiro ponto nao existe no HashMap...

points.put(coord1, counter); //...insere o ponto no HashMap e atribui lhe um valor que ainda nao existe (depois incrementa)

p.add(counter);

counter++;

}

**else**{

p.add(points.get(coord1));

}

**if**(!points.containsKey(coord2)){ //Se o segundo ponto nao existe no HashMap...

points.put(coord2, counter);//...insere o ponto no HashMap e atribui lhe um valor que ainda nao existe (depois incrementa)

p.add(counter);

counter++;

}

**else**{

p.add(points.get(coord2));

}

}

**int** out = PolygonPhobia(p, lineSegments);

System.out.println(out); //Imprime o resultado

}

**private** **static** **int** PolygonPhobia(Queue<Integer> p, **int** lineSegments) {

UnionFindInArray union = **new** UnionFindInArray(lineSegments \* 2); //Referencia a classe onde ocorre a uniao

**int** result = 0; //Variavel que guarda o resultado

**while**(!p.isEmpty()){

**int** union1 = union.find(p.remove());//Encontra o representante do primeiro ponto

**int** union2 = union.find(p.remove());//Encontra o representante do segundo ponto

**if**(union1!= union2){ //Caso em que os representantes sao diferentes

union.union(union1, union2); //Executa a uniao dos representantes

result++;

}

}

**return** result;

}

}

**Classe UnionFindInArray**

**package** dataStructures;

**public** **class** UnionFindInArray **implements** UnionFind

{

**static** **final** **long** *serialVersionUID* = 0L;

// The partition is a forest implemented in an array.

**protected** **int**[] partition;

// Definition of the range of valid elements.

**protected** String validRangeMsg;

// Creates the partition {{0}, {1}, ..., {domainSize-1}}.

**public** UnionFindInArray( **int** domainSize )

{

partition = **new** **int**[domainSize];

**for** ( **int** i = 0; i < domainSize; i++ )

partition[i] = -1;

**int** lastElement = domainSize - 1;

validRangeMsg = "Range of valid elements: 0, 1, ..., " + lastElement;

}

**protected** **boolean** isInTheDomain( **int** number )

{

**return** ( number >= 0 ) && ( number < partition.length );

}

// Pre-condition: 0 <= element < partition.length.

**protected** **boolean** isRepresentative( **int** element )

{

**return** partition[element] < 0;

} // Returns the representative of the set that contains

// the specified element.

//

// Without side effects - Iterative.

// Returns the representative of the set that contains

// the specified element.

//

// With path compression.

**public** **int** find( **int** element ) **throws** InvalidElementException

{

**if** ( !**this**.isInTheDomain(element) )

**throw** **new** InvalidElementException(validRangeMsg);

**return** **this**.findPathCompr(element);

}

// Pre-condition: 0 <= element < partition.length.

**protected** **int** findPathCompr( **int** element )

{

**if** ( partition[element] < 0 )

**return** element;

**else**

{

partition[element] = **this**.findPathCompr( partition[element] );

**return** partition[element];

}

}

// Removes the two distinct sets S1 and S2 whose representatives are

// the specified elements, and inserts the set S1 U S2.

// The representative of the new set S1 U S2 can be any of its members.

//

// Union by size.

**public** **void** union( **int** rep1, **int** rep2 ) **throws** InvalidElementException,

NotRepresentativeException, EqualSetsException

{

**if** ( !**this**.isInTheDomain(rep1) || !**this**.isInTheDomain(rep2) )

**throw** **new** InvalidElementException(validRangeMsg);

**if** ( !**this**.isRepresentative(rep1) )

**throw** **new** NotRepresentativeException("First argument");

**if** ( !**this**.isRepresentative(rep2) )

**throw** **new** NotRepresentativeException("Second argument");

**if** ( rep1 == rep2 )

**throw** **new** EqualSetsException("The two arguments are equal");

**if** ( partition[rep1] <= partition[rep2] )

{

// Size(S1) >= Size(S2).

partition[rep1] += partition[rep2];

partition[rep2] = rep1;

}

**else**

{

// Size(S1) < Size(S2).

partition[rep2] += partition[rep1];

partition[rep1] = rep2;

}

}

}

**Interface UnionFind**

**package** dataStructures;

**import** java.io.Serializable;

**public** **interface** UnionFind **extends** Serializable

{

// Creates the partition {{0}, {1}, ..., {domainSize-1}}.

// UnionFind( int domainSize );

// Returns the representative of the set that contains

// the specified element.

**int** find( **int** element ) **throws** InvalidElementException;

// Removes the two distinct sets S1 and S2 whose representatives are

// the specified elements, and inserts the set S1 U S2.

// The representative of the new set S1 U S2 can be any of its members.

**void** union( **int** representative1, **int** representative2 ) **throws**

InvalidElementException, NotRepresentativeException,

EqualSetsException;

}

**Classe EqualSetsException**

**package** dataStructures;

**public** **class** EqualSetsException **extends** RuntimeException

{

**static** **final** **long** *serialVersionUID* = 0L;

**public** EqualSetsException( )

{

**super**();

}

**public** EqualSetsException( String message )

{

**super**(message);

}

}

**Classe InvalidElementException**

**package** dataStructures;

**public** **class** InvalidElementException **extends** RuntimeException

{

**static** **final** **long** *serialVersionUID* = 0L;

**public** InvalidElementException( )

{

**super**();

}

**public** InvalidElementException( String message )

{

**super**(message);

}

}

**Classe NotRepresentativeException**

**package** dataStructures;

**public** **class** NotRepresentativeException **extends** RuntimeException

{

**static** **final** **long** *serialVersionUID* = 0L;

**public** NotRepresentativeException( )

{

**super**();

}

**public** NotRepresentativeException( String message )

{

**super**(message);

}

}