departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Análise e Desenho de Algoritmos – 2013-2014

Trabalho Prático 3: *Polygon Phobia*

Eduardo Lopes (nº 35469), Ricardo Gaspar (nº 42038)

Departamento de Informática – FCT/UNL

Apresentação do Problema

O objectivo do trabalho é saber o número de linhas que um artista com fobia a polígonos pode desenhar. Esta pintura é feita com base num modelo "*the luck of the draw*" em que o artista escolhe dois pontos aleatórios que representam as extremidades de uma linha e escreve-os num papel. Faz isso para todas as linhas que pensa inicialmente desenhar, escrevendo as coordenadas em papéis diferentes. No fim coloca os papéis num jarro onde as linhas são todas misturadas. O artista começa a tirar e a pintar as linhas num sistema de coordenadas cartesianas, tendo sempre atenção à sua fobia para não formar um polígono. Ou seja, se sair uma linha que forme um polígono esta não é desenhada e não será contabilizada para o número final a retornar.

Resolução do Problema

Para resolver o problema é preciso construir o grafo à medida que se vai lendo as linhas a desenhar. Cada linha representa um arco e os pontos da linha são os vértices do arco. Logo, a forma de evitar o desenho de um polígono é detectar se a introdução de um dado arco provoca um ciclo no grafo.

Para se conseguir isto é necessário colocar os vértices dentro de conjuntos. Cada conjunto é composto por números inteiros designados de índices que representam univocamente cada vértice. Assim cada vértice tem um único índice com o qual possa ser identificado. Com isto o grafo é representado como um conjunto de conjuntos de vértices, sendo que os vértices só ocorrem uma única vez em todo o grafo. Isto é, não pode ocorrer o mesmo vértice em conjuntos diferentes. Ao ter vértices no mesmo conjunto significa que existe um caminho entre eles. Por exemplo, tendo os vértices (5,1) e (7,8), com os índices 0 e 1 respectivamente. Ao ter o conjunto {0,1} significa que existe um caminho de o vértice (5,1) para o vértice (7,8). Inicialmente, antes da construção do grafo, apenas existem conjuntos singulares. Utilizando o exemplo anterior o grafo seria o conjunto {{0},{1}}.

Em cada um dos conjuntos existe um vértice que se designa o representante do conjunto. Desta forma consegue-se distinguir conjuntos diferentes pelos seus representantes. Alargando o exemplo anterior para incluir o arco com os vértices (1,2) e (3,4) com os respectivos índices 2 e 3. Neste caso o grafo seria representado pelo conjunto {{0,1},{2,3}}. Para efeitos de exemplo pode-se assumir que os representantes são os menores índices de cada conjunto. Assim, os representantes de cada conjunto seriam 0 e 2, respectivamente.

Ao representar o grafo desta forma é possível verificar se um novo arco, par de vértices, introduz um ciclo no grafo pois basta verificar se ambos pertencem ao mesmo conjunto, ou seja, se têm o mesmo representante. Caso não tenham, significa que o arco não introduz ciclo e o arco pode ser adicionado. Para a adição de um arco basta realizar a união dos conjuntos dos seus vértices. Mais concretamente basta fazer com que um dos conjuntos passe a ter como representante o representante do outro conjunto. Para o exemplo anterior, se um novo arco a adicionar fosse entre os vértices (1,2) e (7,8) então o grafo passaria a ser representado pelo conjunto {{0,1,2,3}} visto os seus representantes serem diferentes. Desta forma, o grafo passaria a ter apenas um conjunto cujo representante seria o 0.

Esta ideia permitiu que cálculo do número de linhas que o artista pode desenhar se baseasse na subtracção do número total de arcos lidos pelo do número de arcos que introduziam ciclos no grafo.

Implementação do Algoritmo

Para a implementação do algoritmo foram utilizadas várias classes. A classe *Vertex* representa um ponto no plano constituído por duas coordenadas *xx* e *yy*.

A interface *UnionFind* assim como a respectiva sua implementação através da classe *UnionFindInArray* que representam o grafo sobre a forma de um conjunto de conjuntos de vértices.

Toda a implementação do Tipo Abstracto de Dados (TAD) *UnionFind* foi disponibilizado na página da cadeira e permite realizar exactamente o que foi descrito na secção anterior. No entanto, esta implementação tinha disponíveis várias implementações ao nível do método *find.* Devido às condições do problema foi escolhida a implementação que mais se adequava, o *find* por tamanho com compressão de caminho. A escolha do *find* com compressão de caminho é justificada com facto de no *input* poderem vir vértices repetidos e a consulta do seu representante ser mais rápida para este tipo de casos. Estes casos ocorrem com grande frequência visto o objectivo do programa ser evitar polígonos e os polígonos terem vértices partilhados.

Uma vez que o *UnionFindInArray* utiliza um *array* de inteiros para identificar os vértices foi necessário utilizar uma estrutura auxiliar que permitisse estabelecer a correspondência unívoca entre índices e vértices mencionada anteriormente. A estrutura escolhida foi um *hashmap* cuja chave é uma *String* e o valor é um *Integer*. Esta estrutura designada de *vertices* serviu para guardar todos os vértices lidos do *input* ao mesmo tempo que filtrava os vértices já lidos. Assim garante-se que o vértice só existe uma vez e tem apenas um índice que o identifica. Este índice é o número de ordem de ocorrência do

vértice no input, excluindo duplicados. Exemplos: se o vértice (5,1) é o primeiro e ocorre pela primeira vez *input* então este tem o índice 0 associado; se o vértice (3,4) é o quarto vértice e ocorre pela primeira vez no *input* então tem o índice 3. Para a obtenção ou inserção de um índice no *hashmap vertices* utilizou-se como chave o método *toString* definido na classe *Vertex*.

Para se tirar partido do TAD *UnionFind* foi necessário criar um objecto deste tipo que foi designado *partition*. Este é implementado utilizando um *array*, pelo que a indexação dos vértices mencionada anteriormente serve para se poder identificar os vértices pelo seu índice neste *array*. Como se pode perceber pelo que foi descrito, os métodos *find* e *union* são os que permitem a manipulação do grafo sob a forma de conjuntos.

O método *find* retorna para um dado vértice , mais concretamente o seu índice, o representante do conjunto ao qual pertence. Assim, para cada linha, após lidos os dois pontos da mesma, criados os vértices e as respectiva indexação consulta-se o representante de cada um destes. De seguida, comparam-se os representantes.

No caso de serem diferentes é chamado o método *union* que forma um adiciona ambos os vértices ao mesmo conjunto. Isto é conseguido através da colocação de um dos representantes como filho do outro. Neste caso o representante cujo tamanho for menor será o filho do outro. Como a implementação é feita em *array* esta operação implica a manipulação de índices e soma do tamanho. Quer isto dizer que o seu custo é constante.

Caso os representantes sejam iguais, então os vértices já se encontram no mesmo conjunto e, portanto, o que se faz é decrementar em uma unidade o número de linhas lido inicialmente.

No final, após iterados todas linhas, ou arcos, do input é devolvido o número de linhas que quando desenhadas não formam qualquer polígono.

Análise do Algoritmo

O estudo da complexidade do algoritmo foi realizado para as suas duas componentes:

Estudo da complexidade temporal do algoritmo.

Operação	**Complexidade no pior caso**
Criação do *hashmap vertices*	**Θ(1)**
Criação da partição *partition*	**Θ(|V|)**
Ciclo (executado |A| vezes)	
1. Verificar se vértice existe no *hashmap vertices*	**Θ(1)**
2. Inserir vértice no *hasmap vertices*	**Θ(1)**
3. Descobrir representante (executado duas vezes)	**O(log |V|)**
4. Unir representantes	**Θ(1)**
Total:	**O(|A|*log |V|)**

O custo total do algoritmo traduz-se no número de vezes que se realiza a operação de descobrir o representante de um dado vértice (*find*). Esta é da ordem de log|V| devido à implementação do método *find* com compressão de caminho.

Estudo da complexidade espacial do algoritmo.

vertices.size() = **|2*numLines|**

Array da partição = **|2*numLines|**

O tamanho do *hashmap vertices* é inicializado com uma capacidade de |2*numLines| pois não é possível saber de antemão quantos vértices serão inseridos. Sabe-se que no pior caso os vértices de todos os arcos do *input* são diferentes. Assim assegura-se uma taxa de ocupação de pelo menos 50%, visto que no mínimo há um vértice que tem um arco para cada um dos restantes vértices. Ou seja, é uma raiz.
O *array* da partição é inicializado com uma capacidade igual à do *hashmap vertices* pela mesma razão.

numLines – representa o número de linhas lidas do input, consequentemente o número possível de arcos do grafo.
V – número de vértices do grafo. No pior caso, pode ser igual a 2**numLines*.
A – número de arcos do grafo. No pior caso, pode ser igual a *numLines*.

Conclusões

Este trabalho revelou-se fácil após a compreensão do problema enunciado. Inicialmente houve alguns problemas.

Um deles teve relacionado com a criação desnecessária de uma lista que servia para armazenar os arcos para mais tarde serem iterados por forma a se realizar as operações sobre os vértices. Com isto, surgia o problema de o programa demorar demasiado tempo a executar (*TLE – Time Limited Exceeded* na plataforma de submissão *Mooshak*). Após a correcção deste problema, o programa passou em todos os testes, sendo aceite na página de submissão do trabalho.

Outro dois erros foram o facto se utilizar um tamanho demasiado baixo para o *array* da partição aliado à utilização de uma cláusula *try-catch* cercava as operações de *find* e *union* sobre os vértices. Esta cláusula, apanhava uma excepções do tipo *EqualSetsException* seguido do tipo genérico, *Exception*. O que acontecia é que o decremento do total de linhas era realizado quando a excepção *EqualSetsException* era apanhada e quando outro tipo de excepção era apanhada era ignorada e continuava-se para a leitura da próxima linha. Esta conjunção de erros fazia com que o programa produzisse resultados correctos, mas estando mal formulado. Isto porque as outras excepções apanhadas pelo tipo genérico resultavam do tamanho diminuto do *array* da partição o que fazia com que esta lançasse excepções do tipo *InvalidElementException*.

Contudo, após resolvidos os erros mencionados o programa desenvolvido produz resultados correctos e apresenta-se como uma solução eficiente para o problema enunciado.

Anexo

O código fonte encontra-se nas páginas seguintes.

NOTAS:

1. A classe *Edge* ainda consta no trabalho uma vez que a última submissão do mesmo a inclui. No entanto esta não é utilizada desde a correcção do problema da iteração desnecessária da lista que causava *TLE*.
2. Existem *imports* na classe *Main* que não são utilizados.