Análise e Síntese de Algoritmos
2º Projeto – 1º Semestre – 2013/2014
Tomás Alves – 75541
Miguel Santos - 75551

Introdução

O presente relatório aborda a solução e respectivos testes e análise ao problema proposto como segunda parte do projeto de Análise e Síntese de Algoritmos do primeiro semester do ano escolar 2013/2014.

O problema trata a segurança de uma cidade na realização de eventos que atraem um grande amontoado de multidões. Essas multidões podem entrar em motim, sendo o objetivo prevenir e anular o dito através do corte mínimo de ligações entre diferentes pontos da cidade, os pontos críticos. Assim, pretendemos que os pontos críticos, aqueles cuja probabilidade de iniciarem um motim caso estejam todos conectados, não estejam todos ligados, garantindo que, no fim, existe pelo menos um conjunto não vazio de pontos críticos isolados dos restantes, minimizando o número de ligações a barrar.

Através do input, é-nos dado o número de pontos a abordar da cidade, bem como todos as ligações entre eles. De seguida, estão designados h problemas, os quais são , cada um, um conjunto de pontos críticos que devem ser tratados. Por cada problema h podem haver pontos críticos desde um até ao número de pontos da cidade.

O programa tem de apresentar com output o número mínimo de ligações a serem barradas para tartar os pontos críticos, como referido anteriormente.

Descrição da solução

Optámos por programar em C++ e tirar partido das estruturas de dados já existentes e disponibilizadas pela mesma linguagem.

Para conceber o programa, associámos cada ponto da cidade a um vértice e cada ligação entre esses pontos, um arco. Assim, trabalhamos sobre um grafo não dirigido.

De forma a solucionar o problema, decidimos optar pelos teoremas de fluxos em grafos. Concebemos o programa segundo o Teorema dos Caminhos Disjuntos e a implementação do algoritmo de Ford-Fulkerson.

No caso de serem dois pontos critícos, designamos um deles por “fonte” e o outro “destino”. Todas as ligações entre pontos são um arco com capacidade um. De seguida, a partir do “destino”, iniciamos uma BFS no grafo até encontrar a “fonte”. No caso de encontrar, significa que existe um caminho entre os dois pontos críticos que tem de ser cortado. Dessa forma, através da “fonte” começamos a descrever o caminho uma vez que na execução da BFS é designado o vértice predecessor e, em cada arco desse caminho, colocamos uma unidade de fluxo, inutilizando esse arco para um futuro caminho, uma vez que a BFS apenas escolhe vértices com capacidade um. No caso de não encontrar a “fonte”, sabemos que não existe mais nenhuma forma de ligar ambos os pontos críticos e, assim, sabemos quantas ligações devem ser barradas.

No caso de serem mais que dois pontos críticos, tomamos uma abordagem diferente. Escolhemos um desses pontos críticos para ser a “fonte”

e um outro ponto para ser o"destino" até todas as combinações entre o total de pontos críticos estarem cobertas. A BFS executa de forma igual à menionada anteriormente. Após sabermos, para cada "fonte" desse conjunto, qual aquela que responde ao problema com o número menor de ligações por barrar, escolhemo-la, ou seja, a melhor combinação tendo em conta o objetivo do programa.

Esta conclusão provém do facto de que para cada caminho existente desde a "fonte" ao "destino", sendo que nenhum desses caminhos partilha arcos, existe uma ligação que deve ser cortada de forma a isolar os pontos. No conjunto de caminhos, é possível determiner o número de ligações a serem cortadas, bem como quais, mas apenas abordaremos a primeira hipótese.

As estruturas de dados utilizadas para o programa foram vetores, vetores de vetores, pares e filas (*queue*).

Existe uma classe *Graph* a qual tem como atributos um inteiro a designer o número de vértices, um vetor de vetores de pares a designar, no primeiro índice, o número do vértice compreendido de zero a V – 1 e, no segundo índice, o par cujo primeiro inteiro se refere ao número do vértice com o qual tem uma ligação e o segundo inteiro à capacidade desse arco, e, por ultimo, um vetor de inteiros para armazenar por cada BFS, o predecessor de cada vértice.

Como funções da classe, existem a que cria o próprio grafo e a que cria as ligações entre os diferentes vértices. Para além destas, a função `void flow(int u, int v)` trata de, recebendo dois vértices, diminuir a capacidade do arco que os liga, ou seja, colocando fluxo entre esses dois vértices e, no nosso programa, inutilizando o dito arco para outras possíveis BFS. A função `void refresh()` faz precisamente o contrário a cada arco do arco do grafo, fazendo com que não haja fluxo a circular pelo mesmo, dotando a capacidade em um outra vez. Assim, é possível assegurar que, depois de resolver um problema, o grafo está novamente disponível e como se estivesse acabado de ser formado para receber o novo problema.

A função `int BFS(int start, int end)` executa uma BFS normal nos vértices do grafo com um vetor de cores e um vetor de tempos de chegada, mas com a particularidade de apenas visitar vértices caso ainda seja possível passer fluxo pelos mesmos. No caso de conseguir encontrar a fonte, retorna 1, e em caso contrário, ou seja, não há mais nenhum caminho para a fonte, retorna 0.

A função `int ford_ful(std::vector<int> sol)` é comparável ao maestro deste programa. Recebe como argumento um vetor de inteiros contendo os pontos críticos a abordar no problema. Em primeiro lugar, verifica quantos pontos críticos é que devem ser abordados. Se forem dois, trata de designar um deles como a "fonte" e o outro como o "destino" e começa a fazer múltiplas BFS, uma de cada vez, até todos os caminhos disjuntos entre os pontos críticos estarem descobertos e, no fim, retorna o número de caminhos descobertos, ou seja, o total de ligações que devem ser barradas para isolar os dois pontos um do outro.

Se forem mais que dois pontos críticos, a execução é como a mencionada anteriormente, ou seja, dado o conjunto de pontos críticos (A, B, C), são executadas C_2^3 vezes. É importante referir que depois de abordar uma

combinação de pontos críticos e passar a designar a próxima, é executada a função `void refresh()` para tratar cada caso individualmente e independentemente dos outros.

A função `int main()` trata de processar o input e criar o grafo e as respetivas ligações. Depois começa por receber os problemas e colocá-los dentro de vetores, os quais vão ser processados pela função `int ford_ful(std::vector<int> sol)`, um de cada vez para cada problema. Após a execução e concretização da solução para um problema, ou seja, um conjunto de pontos críticos, as ligações do grafo são novamente refrescadas de forma a manter o grafo limpo para cada novo problema.

Análise Teórica

A construção do grafo é executada em O(V+E), assim como cada. `int BFS(int start, int end)`.

As funções de `void refresh()` e `void flow(int u, int v)` são executadas em O(E).

Tanto a BFS como as duas funções anteriores são executadas dentro da função `int ford_ful(std::vector<int> sol)`, a qual é executada uma vez para cada problema Designemos o número de arcos por E, o número de problemas por *h*, o número de pontos críticos de um problema *p* e o fluxo total do grafo por *f*. Assim, a complexidade do nosso programa é de O(EfhC_2^p). Ou seja, o algoritmo Ford-Fulkerson, o qual tem a como complexidade O(Ef), sendo cada caminho de fluxo descoberto por BFS ou DFS, é corrido *h* vezes, abordando as combinações de todos os pontos críticos designados dois a dois.

Avaliação experimental dos resultados

Para testar o programa que concebemos, primeiramente começamos com os *inputs* fornecidos pelo professor como testes públicos e comparámos os *outputs*.

Os resultados temporais foram os seguintes:

- teste 01: 0m0.002s
- teste 02: 0m0.003s
- teste 03: 0m0.002s
- teste 04: 0m0.002s
- teste 05: 0m0.004s

Após análise dos tempos de execução bem como da comparação do *output* obtido com o esperado, concluímos que o mesmo programa é extremamente eficiente e eficaz, mesmo quando o *input* cresce exponencialmente em complexidade e o expomos a ordens de grandeza de mil vértices num grafo pesado com um problema que implica que um quinto dos vértices seja crítico (0,17 segundos).