Trabalho Prático 01, AEDS3

Giulia M. S. G. Vieira - 2016006492

14 de Dezembro de 2018

1 Introdução

Problemas da vida real muitas vezes podem ser resolvidos por meio de mecanismos computacionais simples e conhecidos. Um exemplo disso é o desafio proposto para este primeiro trabalho da disciplina de Algoritmo de Estrutura de Dados 3, lecionada por professores do Departamento de Ciência da Computação da UFMG.

O cenário abordado é o da competição chamada "Auto-cone", evento automobilístico popular nos anos 1950. Na época, a tecnologia de telecomunicações ainda não estava bem desenvolvida, então as corridas não eram transmitidas pela TV até a década de 1970, o que dificulta a aquisição de dados acurados sobre os resultados. Como já se passaram quase 50 anos desde a última corrida não televisionada, a morte e os problemas de memória dos competidores também se tornaram impecílios nesta coleta. Como então estudar o que aconteceu?

Mesmo neste contexto nebuloso podemos adquirir alguns dados sobre os ocorridos, ainda que não acurados, e a partir daí desenvolver um modelo que ordene todas estas informações e, possivelmente, nos forneça um resultado razoável.

Se associarmos a corrida a um grafo, tudo se torna mais nítido. Sendo cada carro um vértice e cada lembraça de quem estava em sua frente uma aresta direcionada não ponderada, podemos agrupar todos os relatos em uma tupla G = (V, A).

Para formalizar a organização das informações neste sistema, usaremos aqui um arquivo entrada e um saída, ambos em formado .txt.

1.1 Entrada

Primeiramente serão apresentados dois números inteiros, P e M, onde P é o número de pilotos e M é o número de memórias registradas. No caso então, para a relação estabelecida a cima, G = (P, M).

As M linhas seguintes contém tuplas <piloto i, piloto j>, para piloto i entrevistado e piloto j a sua frente.

O fim da entrada de dados será uma tupla <0,0>.

1.2 Saída

Organizaremos os possíveis resultados por duas formas, 1. tipo de resultado adquirido e 2. qual foi o resultado. Ambos virão nesta ordem, na mesma linha, separados por espaço.

1.2.1 Tipo de resultado adquirido

0 - Não é possível definir a ordem de classificação através das memórias dos pilotos.

1 - é possível gerar uma ordem de classificação única.

2 - é possível gerar mais de uma ordem de classificação. Neste caso preservaremos a ordem crescente de identificadores.

1.2.2 Qual foi o resultado

Se o tipo foi 0, retornaremos -1 como resultado final, visto que não há ordenação possível. Se o tipo foi 1 ou 2 retornaremos a classificação do primeiro ao último identificador, separados por vírgula.

1.2.3 Imagem final da saída

X P1,P2,...,Pn

2 Modelagem do problema

Ao desenharmos o grafo do problema percebemos que as classificações explicitadas nos modelos de sáida estão diretamente associadas a possíveis características de grafos.

Para não conseguirmos definir uma ordem de classificação entre os pilotos, temos que o grafo deve conter algum ciclo, porque nele não sabemos quem está na frente de quem.

Para conseguirmos definir uma ordem, consequentemente, o grafo deve não conter ciclos. Além disto, para que haja apenas uma ordem, cada vértice deve apontar para no máximo um outro. Se apontar para mais de um temos diferentes possíveis caminhos.

Neste contexto, temos apenas dois grandes problemas para resolver, o primeiro é o do ciclos, o segundo é o do número de arestas que saem de cada vértice. Além disto, há um problema auxiliar, que é descobrir qual nó será utilizado como raíz para a ordenação.

2.1 O problema dos ciclos

Para a resolução do problema dos ciclos foi utilizada uma adaptação do algoritmo DFS (Deep First Search), que é originalmente utilizado para encontrar um vértice a partir de outro, seguindo sempre o caminho da profundidade, ou seja, na árvore, seguindo para o próximo filho ao invés de algum nó "irmão",

até que chegamos ao nó requerido ou ao fim da árvore, que signifca que o nó não existe.

Na adaptação para detecção de ciclos, então, usa um vetor auxiliar inicializado com zeros, que marca com 1 a posição com o mesmo número do vértice atual e, em seguida, entrando em sua lista de adjacência, se chama recursivamente para cada um dos próximos vértices, fazendo assim, busca em profundidade para todas as possibilidades. Este algoritmo então, caso algum destes novos vértices alcançados já tenha sido alcançado anteriormente por uma chamada anterior, ou seja, esteja marcado de 1 no vetor auxiliar, retorna 1, como quem atesta que há um ciclo no grafo. Caso terminemos todas as passagens tem retornar 1, retornamos 0, como quem diz que não há ciclos no grafo.

2.2 O problema das múltiplas arestas

Dado que o grafo não possui cilos, ou seja, é uma árvore, temos agora que resolver a ordenação dos participantes. Sendo uma árvore, então, podemos concluir que cada nível da árvore representa um nível de colocação diferente, portanto, uma busca por largura resolve.

Nisso, usamos o algoritmo BFS (Breadth Frist Search), que opera exatamente desta maneira. Usando uma lista auxiliar que salva os nós da fronteira, ou seja, os filhos do nó atual somados aos filhos ainda não visitados dos nós anteriores, seguimos vértice a vértice empilhando da raíz às folhas em uma lista que salva a ordenação, aqui chamada sorted.

2.2.1 Ordenação dentro do nível

Nas especificações deste trabalho estava definido que quando houvessem nós no mesmo nível deveríamos ordenar de forma crescente, sendo o de menor valor o mais alto na pilha sorted.

Para isto, foi usado um Quicksort simples para as listas de adjacência dos vértices durante a operação do BFS. A cada vez que entramos em um novo vértice ordenamos sua lista de adjacência em ordem decrescente, assim, pegamos sempre o último vértice possível primeiro e o empilhamos na borda, então, quando formos desenfileirar a borda e empilhar na lista sorted temos certeza de que a ordem crescente é preservada.

2.3 O problema da raíz

Para podermos usar o BFS precisamos de um vértice que será nosso ponto de partida, porque, dependendo de onde começamos a ordenação pode ficar diferente. Neste caso, temos duas possibilidades: se o grafo for um ciclo por inteiro, não temos um grafo específico por onde começar, se não, existe pelo menos um grafo para quem ninguém aponta.

Neste contexto, foi criada uma função auxiliar chamada findLast, que percorre todas as listas de adjacência, marcando no vetor auxiliar de visitados com

1 os vértices visitados, ou seja, aqueles para quem alguém aponta. Depois, verificamos este mesmo vetor e, assim que encontramos uma posição com 0, ou seja um vértice raíz, retornamos, porque este pode ser usado no BFS. Caso não encontremos, sabemos que o grafo é completamente um ciclo, então não precisamos mandar para nenhum dos algoritmos citados anteriormente e apenas printar o resultado de impossível ordenação no arquivo saída.

3 Análise teórica do custo assintótico

3.1 Tempo

3.1.1 Funções de lista

Para a resolução deste problema usamos uma lista simplesmente encadeada, sem apontador para o último elemento.

funções de criar: Para criar cada uma das células da lista e uma lista vazia temos custo constante. O(1).

funções de inserir: Para inserir célula ao final, como não temos apontador para o final da lista, temos custo O(n), visto que precisamos percorrer toda a lista. Já para inserir célula ao início o custo é constante O(1), visto que só precisamos criar uma célula nova e a apontar para o início.

funções de ordenação: Como utlizamos um Quicksort sem modificações, temos o caso médio conhecidamente O(n log n).

função de remover nó: Como não há apontador para o fim da lista, para remover um nó do seu final precisamos percorrer toda a lista, ou seja, operação O(n).

função de imprimir: Como para imprimir a lista precisamos passar por todos os elementos a serem impressos temos que a operação é O(n).

funções de liberar memória: Como para liberar a memória da lista precisamos liberar a memória de cada uma de suas células, temos que a operação é O(n).

3.1.2 Funções de grafo

função de criar: Como para cada vértice do grafo precisamos criar uma lista de adjacência, além dos custos constantes de alocação de memória dos campos do grafo, temos um custo O(n) de inicialização de cada uma das listas de adjacência no vetor de listas de adjacências. Ou seja, no final temos uma operação de custo O(n).

função de inserir: Como inserir um nó no grafo é inserir um nó no final da lista de adjacência do vértice que aponta para ele, temos que esta operação tem custo constante O(1), porque apenas chama a operação já especificada aqui de adicionar nó ao final da lista, que no caso é O(n).

função de detectar ciclos: Como usamos o DFS, temos o custo conhecidamente O(V + E), para V = número de vértices e E = número de arestas.

função de ordenar: Como usamos o BFS, temos custo conhecidamente O(V + E), para V = vértices e E = número de arestas.

função de encontrar raíz: Como percorremos a lista de adjacência de todos os n vértices, que, no pior caso, tem n-1 elementos, ou seja, ficamos com uma operação O(n) e, posteriormente, percorremos o vetor de visitados, o que no pior caso é uma operação O(n), temos um custo geral da função como O(n).

função de zerar visitas: Como para zerar todas as posições do vetor de visitas precisamos percorrer cada uma de suas posições uma vez, esta operação é O(n).

função de de liberar memória: Como precisamos liberar cada um dos vértices do grafo, temos que percorrer todas as posições do vetor de listas de adjacência e chamar a função de liberar lista, assim como executar as operações de custo constante de liberar a memória deste vetor e do de visitador, assim como do TAD grafo por inteiro. Portanto, a única operação de repetição é a de chamada da função de liberar a lista, ou seja, esta função de liberar o grafo é O(n).

3.2 Espaço

Para V = número de vértices do grafo e E o número de arestas: Aqui usamos as seguintes estruturas: o grafo e dois vetores auxiliares de tamanho V, um para guardar as "raízes"do grafo e outro para guardar os caminhamentos no grafo, sem interferir no vetor originalmente designado para isto. De auxiliares então temos complexidade 2V. Como o grafo foi salvo em um vetor de listas de adjacência temos que sua complexidade de espaço é V+E, visto que alocamos as V posições do vetor de listas para os vértices e mais E células de lista distribuídas entre eles. Além disso, temos um vetor de visitas, de tamanho V, para marcarmos quais vértices foram visitados nas operações. Somando tudo então temos 2V+E para o grafo. No geral, portanto, gastamos 4V+E de espaço.

4 Análise de experimento

A implementação da solução proposta foi compilada utilizando gcc 4.8.4. Os experimentos foram executados em um sistema com 8 GB de memória e um processador Intel® Core™ i5-5200U CPU @ 2.20GHz 4. Cada teste foi realizado em torno de 10 vezes e, a partir disso, foi feira uma média dos tempos.

	real	user	sys
Input 1	0m0,002s	0m0,002s	0m0,000s
Input 2	0m0,002s	0m0,002s	0m0,000s
Input 3	0m0,001s	0m0,001s	0m0,000s
Input 4	0m0,002s	0m0,002s	0m0,001s
Input 5	0m0,002s	0m0,002s	0m0,001s

Como podemos perceber, operamos tudo muito mais rápido que o exijido, que era 1 segundo por execussão.

5 Conclusão

A resolução do problema foi satisfatória, com tempo razoável. Algumas operações, como as de lista, poderiam ter tempo melhor, caso esta fosse implementada com elemento marcador de última posição, mas houveram problemas na tentativa de executar tal procedimento, então preferi manter da manera como foi feito.

Nos demais procedimentos foram usados os algoritmos com melhor custo que eu conhecia. Tivemos apenas uma função O(n), e este pior caso só seria alcançado em um grafo muito conectado, o que muito provavelmente não é o caso da maior parte dos testes deste problema, dadas as especificações, então na realidade temos algo girando em torno de O(n), nos casos médios, isto para mim é muito satisfatório.

No geral, giramos em torno de funções de custo linear e fixo, com um caso sublinear (Quicksort) e um superlinear (este citado a cima).