EP 1

Problema 452 – Project Scheduling

Grupo 16

Arthur Paulucci Carnieto	8921047
Higor Lopes da Silva	8921760
Marcello Malagoli Ziravello	8921242
Matheus Mendes de Sant'Ana	8921666

Desafios de Programação I
ACH2107 – Turma 03

SÃO PAULO
2017

1) Variáveis

1.1) Principais estruturas

1.1.1) Globais

pesos (tipo Map) – Permite a busca ou inserção de um peso em cada nó do grafo. Essa estrutura terá como chave objetos do tipo **Character** e como valor objetos do tipo **Integer**.
– Exemplo de busca: Através da chamada pesos.get('A') poderíamos obter o peso do nó 'A'.
– Exemplo de inserção: Através da chamada pesos.put('B',7) estaríamos atribuindo peso 7 ao nó 'B'.

arestas (tipo Map) – Conterá todas as arestas do grafo. Essa estrutura terá como chave objetos do tipo **Character** e como valor objetos do tipo **LinkedList<Character>**.
– Exemplo de busca: Através da chamada arestas.get('A') poderíamos obter uma lista de todos os nós vizinhos (acessíveis) do nó 'A', como por exemplo os nós 'B','C' e 'D'.
– Exemplo de inserção: Dada uma lista de nome "lista" que contenha os objetos "Character" 'F' e 'G' (representados como nós); poderíamos realizar a chamada arestas.put('B',lista) que atribuiria os vizinhos 'F' e 'G' ao nó 'B'.

1.1.2) Locais

nos (tipo LinkedList<Character>) – Conterá todos os nós de um caso de teste (cada nó representado por um objeto do tipo Character).

1.2) Principais variáveis auxiliares

nome (tipo char) – Caractere que armazena o nome do nó atual.

peso (tipo int) – Armazena o peso do nó atual.

arestasQueDevemSerInseridas (tipo LinkedList<Object[]>) – Lista ligada de objetos que contém arestas que devem ser inseridas após a leitura completa de um caso da entrada. Cada objeto possui nós "**anteriores**" que apontarão para um nó "**atual**".

2) Algoritmo

2.1) Inicialização das estruturas

A cada caso de teste, as estruturas “pesos”, “arestas” e “nos” são recriadas; a variável global “**maiorCaminho**” também é zerada a cada caso de teste.

Para cada caso de teste recebido, teremos **n** linhas sendo que **n** representa o número de nós que o grafo possui. Para cada uma das **n** linhas, adicionaremos o nó ao grafo – adicionaremos o caracter correspondente do nó atual na lista “nos” e adicionaremos o peso correspondente ao nó atual na lista “pesos” –, além de adicionarmos as arestas no grafo. exemplo:

Entrada:
2

A 5
B 3 A

A 5
B 4 A
C 2 AB

Como ficam as estruturas no <u>1º</u> caso de teste:

“**nos**”: 'A', 'B'
“**pesos**”: [h('A') → 5], [h('B') → 3]
“**arestas**”: [f('A') → {'B'}], [f('B') → {}]

Como ficam as estruturas no <u>2º</u> caso de teste:

“**nos**”: 'A', 'B', 'C'
“**pesos**”: [h('A') → 5], [h('B') → 4], [h('C') → 2]
“**arestas**”: [f('A') → {'B', 'C'}], [f('B') → {'C'}], [f('C') → {}]

As funções **h** e **f** são funções “hash” quaisquer, que mapeiam uma chave “k” em um valor “x” – como visto no 1º caso de teste, a função h('A') retorna o inteiro 5, enquanto a função f('B') retorna uma lista vazia de objetos “Character”.

Observando a estrutura “arestas”, tendo como exemplo o 1º caso de teste, percebemos que na linha “B 3 A” o nó 'B' é dependente do nó 'A', então é criada uma aresta partindo de 'A' para 'B' (e não o contrário).

Explicando de modo “baixo nível”, um nó **x** qualquer terá uma entrada na função “hash” que nos levará a uma lista de outros caracteres (seus vizinhos). Ao obtermos a referência a essa lista de vizinhos, podemos adicionar outros vizinhos a ela ou percorrê-la para visualizar todos os vizinhos do nó **x**. Adicionando um vizinho **y** à lista, estamos criando uma aresta partindo de **x** para **y**.

2.2) Encontrando a solução

Após instanciarmos os nós, atribuirmos pesos aos nós e criarmos o grafo, podemos finalmente chegar à ideia principal do algoritmo.

O algoritmo procura **encontrar o ramo do grafo que tenha os nós com a maior somatória de pesos**: essa maior somatória de pesos será a resposta final. Como o grafo pode não ser conexo e pode não ter nó “cabeça” – ou seja, um nó que não tem dependências e a partir dele são alcançados todos os outros – devemos começar a busca começando a partir de cada um dos nós existentes. Na Fig.1 temos um exemplo de dois blocos conexos A e B em um grafo qualquer; percebe-se que ambos podem ser executados em paralelo pois não possuem dependência entre si:

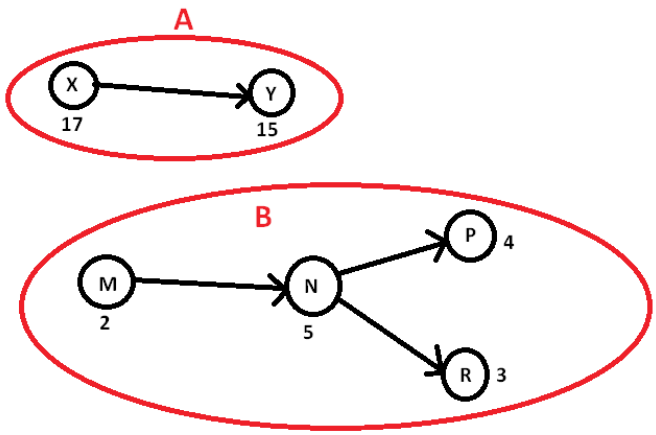

Fig.1: Um exemplo de grafo

Para encontrarmos o ramo do grafo que tenha a maior somatória de pesos, fazemos uma busca em profundidade começando a partir de cada um dos nós do grafo. Para cada ramo da busca em profundidade, é somado o peso acumulado dos nós anteriores do ramo ao peso do nó atual: se essa somatória for maior que a variável global “**maiorCaminho**”, a variável “maiorCaminho” recebe o valor dessa somatória; do contrário a busca em profundidade continua, passando para os nós vizinhos essa somatória como novo peso acumulado

do ramo.

Ao findar todas as buscas em profundidade, é imprimido na tela o valor da variável “maiorCaminho” que conterá a somatória dos pesos dos nós pertencentes ao ramo de maior peso acumulado. No nosso exemplo da Fig.1, a resposta seria **32** pois o ramo **XY** é o de maior peso acumulado no grafo. Continuando no exemplo da Fig.1: caso o grafo tivesse apenas o bloco conexo B, a resposta seria **11** pois o ramo **MNP** seria o de maior peso acumulado.

3) Dificuldades encontradas

1ª) Nas primeiras versões do EP, recebíamos o resultado “Time limit exceeded” na submissão. Para reduzirmos o tempo de execução, optamos por reduzir o número de variáveis globais e reduzir a quantidade de métodos auxiliares, executando grande parte do código no método “main”.

2ª) No princípio utilizávamos Scanners para extrair o **nome**, **peso** e **dependências** de cada nó em cada linha dos casos de teste. Após recebermos “Time limit exceeded”, optamos por utilizar métodos de String para “recortarmos” os pedaços de cada linha através dos endereços (“index”) dos espaços, ao invés de utilizarmos um novo Scanner para cada linha.

3ª) Assumimos no primeiro momento que o grafo seria conexo e que todos os elementos seriam alcançáveis a partir do primeiro nó de cada caso de teste: o algoritmo inicial utilizava apenas uma busca em profundidade começando a partir do primeiro nó. Após recebermos uma série de resultados “Wrong answer”, optamos então por realizar um busca em profundidade começando a partir de cada nó do grafo (assumindo que o grafo poderia não ser conexo), o que nos retornou o resultado “Accepted”.