1) Introdução

Uma ordenação topológica consiste na ordenação de um grafo acíclico de forma que o caminho percorrido entre os vértices obedeça uma ordem de precedência determinada.

Esse tipo de ordenação se faz frequentemente necessária frente a problemas que exigem estabelecimento de dependências entre elementos de um determinado conjunto. Considerando por exemplo um conjunto X = { a, b, c } , podemos dizer que se:

- b depende de a (a <= b)
- c depende de b (b <= c)

então uma ordenação topológica para este conjunto seria:

a, b, c

sendo que os elementos que aparecem primeiro são os de maior precedência, e os que aparecem mais à direita são os de maior dependência.

A teoria nos garante que, para todo grafo acíclico, é possível determinar uma ou mais ordenações topográficas. O contrário não é válido para grafos cíclicos. Nesse caso, nenhuma ordenação topográfica é possível.

2) Solução

Existem vários métodos e algoritmos para se realizar a ordenação topológica de um conjunto de vértices num grafo, o algoritmo escolhido é caracterizado pela sua fácil implementação, e consiste na montagem de uma espécie de varal (esta foi a abstração que escolhi para ilustrar o método de funcionamento).

O que chamo de implementação em forma de varal consiste basicamente em "pendurar" listas de dependências em um arranjo que representa o intervalo a ser ordenado, e ir retirando desse "varal" os índices já ordenados.

A eficiência desta implementação poderia ser melhorada com algumas modificações, como comentarei no próximo item, mas que foram omitidas para simplificar o código e o entendimento do mesmo.

Além disso, a opção por uma busca baseada em loops, em detrimento da busca profunda usando recursividade, foi escolhida novamente para facilitar o entendimento e a leitura do código, pois possibilita a utilização de um arranjo no lugar de uma outra lista encadeada, como explicado no próximo item.

3) Estruturas e Tipos de Dados

Apenas duas estruturas de dados significativas são usadas para montar a implementação do varal, arranjos e listas encadeadas. Mas essa não era a idéia original, que consistia numa implementação usando apenas listas encadeadas.

A idéia original consistia numa implementação direta do conceito de grafos, e baseava-se na concepção de que cada nodo do grafo gerado precisaria de 2 informações:

- Um campo do tipo Inteiro, para guardar o valor daquele nodo
- Um campo de referência a uma lista encadeada onde cada nodo referenciaria os nodos dependência do nodo em questão

Dessa forma obteríamos uma lista encadeada de nodos, representando nosso grafo, que eu chamo de varal, e em cada nodo teríamos uma nova lista encadeada, representando as ligações desses nodos a cada um dos outros dos quais eles dependem, estes seriam nossos objetos "pendurados" no nosso "varal".

Essa primeira idéia seria interessante, pois usando um algoritmo recursivo de pesquisa profunda, poderíamos definir uma a uma as relações, e evitar a repetição de algumas operações. Maiores vantagens serão comentadas mais a frente.

Já a implementação utilizada utiliza, no lugar da lista principal para o varal, um arranjo de tamanho N, onde N é o intervalo a ser ordenado. Cada índice do arranjo representa o inteiro que numera os nodos, e em cada um destes indices "penduramos" as nossas dependências, através de referências para listas encadeadas de objetos do tipo Integer.

As vantagens da implementação com Lista de Listas serão mais evidentes após a explicação detalhada da implementação, mas residem principalmente no fato de que, tratando-se as listas de elementos redimensionáveis, a cada vez que extraimos um nodo dela, diminuímos a quantidade de operações que precisaremos fazer na próxima passagem do loop. Já na implementação com Arranjos, teríamos que varrer nosso arranjo no mínimo N vezes e no máximo N^2 vezes, para conseguir extrair todos os nodos do nosso grafo. Esse problema foi diminuido com a introdução de um terceiro arranjo de booleanos, que funciona como uma tabela onde deixamos marcados os nodos que já foram trabalhados pelo algoritmo.

4) Implementação

Como mencionado no item anterior, a implementação por "varal" consiste em "pendurar" num arranjo, listas com referências para os nodos dos quais depende aquele nodo representado pelo índice no arranjo. Pois bem, o que nosso algoritmo faz basicamente é verificar, a cada iteração, quais são os nodos que não possuem nenhuma dependência, ou seja, que já podem ser inseridos na nossa ordenação. O algoritmo então extrai esse nodo do "varal" e o insere numa nova lista encadeada, que chamaremos de lista-resposta.

Entrada:

A entrada do algoritmo consiste de Z+1 linhas. A primeira linha contém dois inteiros X e Y, sendo que X é o tamanho do intervalo a ser ordenado, (1 a X), e Y é a quantidade de pares de inteiros a serem digitados.

Cada uma das Z linhas seguintes contém um par de inteiros, X e Y, sendo que neste caso X é dependência de Y, ou seja, X aparecerá antes de Y na ordenação topológica.

Verificando o anexo com o código fonte do programa, temos o processo de entrada desenvolvido entre as linhas 13:30.

A seção mais importante da entrada começa na linha 23. Nesta seção, um laço for faz a recepção das entradas das Z linhas com os pares de dependentes, e os insere no arranjo de listas que chamamos de varal. Para tanto, são dados dois nomes para as variáveis que recebem temporariamente os dados.

- master -> é a variável que recebe o valor principal daquela entrada, ela será o índice do arranjo.
- slave -> é quem recebe o valor secundário, ou a dependência, do valor master.

Verificação:

Durante a produção do meu programa, inclui um método para verificação da estrutura de dados, que imprime a situação do "varal" após a entrada de todos os dados. Este método seria retirado na versão final, mas decidi deixá-lo porque ele ilustra muito bem a estrutura de dados utilizada e o funcionamento do programa, deixando muito mais fácil o entendimento de sua arquitetura.

Este método é chamado na linha 35, e sua declaração se encontra na linha 41.

Este método apenas varre o arranjo, e em cada índice varre também a lista encadeada "pendurada", imprimindo os índices, tamanhos de listas e os membros de cada lista.

A saída deste método consiste de N linhas, com N sendo o intervalo de ordenação. Cada uma dessas N linhas representa um dos inteiros a ser ordenados, que no programa são os índices do nosso "arranjo-varal".

A estrutura de cada linha é:

Master X (Y) -> A, B, C ...

Onde:

- X é o inteiro master da entrada, o índice da linha no arranjo.
- Y é o tamanho da lista "pendurada" naquele índice, que será útil na execução do algoritmo principal
- A, B, C ... são os Y nodos da lista anexada ao índice do arranjo, representando as Y dependências do nodo principal.

Ao fim da saída deste método, é fácil ter uma idéia bem concreta do funcionamento do programae da estrutura do "varal", e por conseguinte fica bem mais simples o entendimento do algoritmo principal, a ser executado logo em seguida.

Algoritmo Principal:

A alma do programa é seu algoritmo principal, que é o responsável por retornar a ordenação topográfica do intervalo recebido na entrada.

Este algoritmo consiste basicamente da varredura do "arranjo-varal" identificando, um a um, todos os nodos do grafo que não possuem nenhuma dependência. Se um nodo não possui nenhuma dependência, podemos dizer que ele pode ser "retirado" do grafo, e "adicionado" na lista-resposta.

Este procedimento inclui dois laços for, como pode ser observado no corpo do algoritmo cuja chamada se encontra na linha 32, e o corpo se estende nas linhas 53:70.

Logo no início, criamos um arranjo de booleanos do mesmo tamanho do "varal". Este arranjo já foi citado anteriormente, e sua função é evitar que um nodo que já foi "retirado" seja novamente retirado. Este arranjo se fez necessário ao migrar da implementação usando Listas-de-Listas para a implementação usando Arranjo-de-Listas. A cada vez que retirarmos um nodo do grafo, marcaremos nesse arranjo este nodo como visitado, modificando para true o valor do índice correspondente.

Em minha estrutura de dados, exclui os índices 0 de todos os arranjos, para facilitar a compreensão do código. Para evitar problemas, meu arranjo booleano já possui o seu índice 0 com o valor true, como se vê na linha 57.

O primeiro laço for varre o arranjo, buscando em seus índices referências para listas encadeadas de tamanho 0. Se uma lista tem tamanho 0, em nossa abstração significa que não possui nenhuma dependência, portanto pode ser retirada do "varal" e adicionada na lista-resposta. Caso a lista do índice k esteja vazia, e não tenha sido visitada, o algoritmo inicia o procedimento de retirada daquele nodo do grafo.

O procedimento de retirada consiste primeiramente num segundo laço for. Neste laço, o algoritmo varre novamente o arranjo, buscando todas as ocorrências dentro de cada lista encadeada, daquele nodo que será retirado. Afinal, quando retiramos um nodo do grafo e o adicionamos em nossa ordenação, todos os nodos que dependem dele passam a ter essa dependência solucionada. Isso é feito usando o método remove(Object o) da classe LinkedList do Java 1.5.0.

A escolha desse método tem um caráter especial. A outra opção para a remoção destes nodos seria usar o método remove (int i) que removeria um inteiro. Mas para isso eu teria que incluir um terceiro laço for, para varrer cada lista encadeada sozinha, comparando o valor de seus nodos com o valor a ser retirado, o que faria a complexidade do algoritmo mudar de $O(N^2)$ para $O(N^3)$; um salto considerável.

Após varrer as listas e retirar todas as ocorrências do nodo a ser retirado, marcamos ele como retirado na lista visitados (que poderia ter o nome retirados ao invés de visitados, o que ilustraria melhor seu funcionamento), e em seguida o nodo é adicionado no fim da lista resultado.

No final do algoritmo, se retiramos algum nodo, voltamos o iterador do laço for mais externo para o início, pois será necessário varrer o arranjo todo novamente. O loop não infinito, pois a cada vez que removemos um nodo, marcamos-no como retirado, de forma que ele não possa ser retirado novamente; no fim chegamos a uma situação em que não há mais nodos a serem retirados, e então a iteração do for mais externo não mais é resetada.

No fim das contas temos agora um “varal” vazio, e uma lista-resultado cheia, contendo todos os nodos originais do grafo, ordenados de acordo com as dependências estabelecidas na entrada. O último método, cuja chamada está na linha 38 e sua definição nas linhas 71:78, apenas varre a lista-resultado, imprimindo o conteúdo de seus nodos, que é a ordenação topográfica final do intervalo.

Comentários Adicionais:

Complexidade:

O algoritmo principal tem complexidade aproximada O (N^2). Cheguei a isso considerando apenas as operações mais significativas, que estão contidas nos dois laços for aninhados no algoritmo principal. Como o algoritmo tem que varrer o arranjo de tamanho N N vezes, na pior das hipóteses, a complexidade é de N^2.

Ciclos:

O comportamento do meu algoritmo para ciclos é bastante trivial. Optei por não identificar os ciclos, já que a teoria garante que se houver um ciclo, então não existe uma ordenação topográfica para aquele grafo. Na presença de um ciclo, o algoritmo simplesmente pára de ser executado, dando a saída da ordenação até aquele momento, pois ele não consegue excluir as dependências de nenhum nodo além deste ponto.

5) Anexos

Dois anexos estão incluidos.

1. Código Fonte

O código fonte, fomatado e com as linhas numeradas, está incluido como primeiro anexo, para conferência da implementação utilizada.

2. Testes Realizados

Está anexa também uma tabela contendo todas as entradas e respectivas saídas usadas como teste para verificação do funcionamento do algoritmo.