Disciplina: TIN0225 – Estruturas de Dados
Professor: Pedro Moura
Data: 03/11/2023

Trabalho Final

Ordenação Topológica é um processo de ordenação de elementos no qual é definida uma ordem parcial, isto é, no qual uma ordenação é efetuada somente sobre alguns pares de elementos e não sobre todo o seu conjunto. Como exemplo de aplicações que utilizam Ordenação Topológica, podemos citar:

1. Uma tarefa (por exemplo, um projeto) é dividida em subtarefas. Em geral, o término de certas subtarefas deve preceder a execução de outras subtarefas. Denota-se por $v < w$ quando a subtarefa v deve preceder a subtarefa w. Neste caso, a ordenação topológica consiste em dispor as subtarefas em uma ordem tal que, antes do início de cada tarefa, garante-se que todas as subtarefas de que ela depende tenham sido previamente executadas;

2. Em um currículo de um curso universitário, algumas disciplinas devem ser realizadas antes das outras, uma vez que se baseiam nos tópicos apresentados nas disciplinas que são seus pré-requisitos. Denota-se por $v < w$ quando a disciplina v é pré-requisito da disciplina w. A ordenação topológica, neste caso, corresponde a arranjar as disciplinas em uma ordem tal que nenhuma delas exija como pré-requisito uma outra que não tenha sido previamente cursada.

3. Em um programa, alguns métodos (funções) podem ter chamadas a outros métodos. Denota-se também por $v < w$ quando o método v é chamado pelo método w. A ordenação topológica, neste caso, implica o arranjo das declarações dos procedimentos de tal forma que não haja referências a métodos não-declarados anteriormente.

Em geral, uma ordenação parcial de um conjunto S corresponde a uma relação binária entre os elementos de S. Esta relação, denotada pelo símbolo $<$, verbalmente

lida como precede, deve possuir as seguintes propriedades para quaisquer elementos distintos de S:

1. se $x < y$ e $y < z$, então $x < z$ (transitividade);

2. se $x < y$, então não ocorre $y < x$ (assimetria); e

3. não ocorre $z < z$ (irreflexiva).

Assume-se que os conjuntos S a serem ordenados topologicamente por um algoritmo são finitos. Portanto, uma ordenação parcial pode ser ilustrada desenhando-se um grafo direcionado acíclico em que os vértices denotam os elementos de S e os arcos representam as relações de ordem. Um exemplo está mostrado na Figura 1.

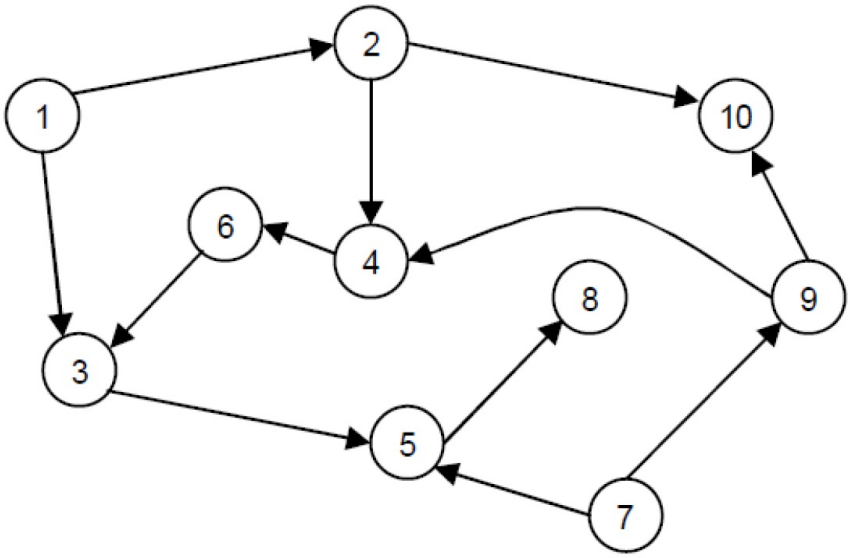

Figura 1: Grafo direcionado acíclico representando um conjunto parcialmente ordenado.

O problema de ordenação topológica consiste em obter uma ordem linear a partir da ordem parcial. Graficamente, isto implica o arranjo linear dos vértices do grafo de tal maneira que todas os arcos tenham o mesmo sentido (para a direita), conforme está mostrado na Figura 2. As propriedades (1), (2) e (3) da ordenação parcial garantem a

ausência de ciclos no grafo. Isto corresponde exatamente à condição de pré-requisito com base na qual é possível fixar uma ordem parcial em ordem linear.

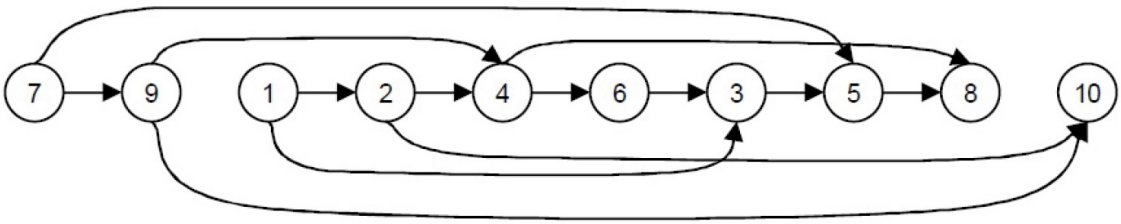

Figura 2: Arranjo linear do conjunto parcialmente ordenado da Figura 1.

Como então se deve proceder para encontrar uma das possíveis ordenações lineares em questão? A estratégia é bastante simples. Inicialmente, escolhe-se um elemento que não seja precedido por qualquer outro (tem que existir pelo menos um elemento nessas condições; caso contrário, existiria um ciclo). Este elemento é posicionado no início da nova lista e eliminado do conjunto *S*. O conjunto resultante ainda se encontra parcialmente ordenado, devendo então a estratégia ser aplicada sucessivamente até que o conjunto *S* se esgote.

Para se poder descrever mais rigorosamente esse algoritmo, deve-se escolher uma estrutura de dados para representar o conjunto *S*, bem como a representação de suas ordenações. A escolha desta representação é determinada pelas operações a serem realizadas, em particular a operação de seleção de elementos sem predecessores. Dado que o número *n* de elementos de *S* não é conhecido *a priori*, o conjunto pode ser organizado convenientemente na forma de uma lista encadeada, denominada de *Lista de Adjacência*.

Cada elemento da lista (classe `Elo`) é representado por quatro atributos:

a) `int chave`: uma chave de identificação que é um número inteiro, não necessariamente inteiros consecutivos de `1` a `n;`

b) `EloSuc listSuc`: uma lista encadeada com seus sucessores;

c) `int contador`: um contador do seu número de predecessores; e

d) `Elo prox`: aponta para o próximo elemento da lista

Como é possível perceber, o conjunto de sucessores de cada elemento da classe `Elo` é representado, muito adequadamente, também por uma lista encadeada. Cada elemento da lista de sucessores (classe `EloSuc`) possui os seguintes atributos:

a) `Elo id`: uma identificação; e
b) `EloSuc prox`: uma referência para o próximo elemento dessa lista de sucessores.

Note que o atributo `id` guarda uma referência para um elemento da classe `Elo`. Por último, a classe `OrdenacaoTopologica` é representada pelo primeiro elemento da lista encadeada que representa o conjunto *S*, denotado pelo atributo `Elo prim`, e pelo contador de elementos da lista, denotado pelo atributo `int n`. As classes Elo e EloSuc são internas à classe OrdenacaoTopologica. A especificação de tais classes estão disponíveis em `java` no Moodle da disciplina.

Assume-se que o conjunto *S* e suas relações de ordenação sejam representados por uma sequência de pares de chaves em um arquivo de entrada. Abaixo, é exibido o arquivo de entrada para o exemplo da Figura 1, em que o símbolo < denota a ordem parcial.

```
1 < 2
2 < 4
4 < 6
2 < 10
4 < 8
6 < 3
1 < 3
3 < 5
5 < 8
7 < 5
7 < 9
9 < 4
9 < 10
```

A seguir, descrevem-se os procedimentos necessários para criar a ordenação topológica. A primeira parte do programa de ordenação topológica deve ler os dados de entrada e transformá-los em uma estrutura de *Lista de Adjacência*, representando o conjunto *S*, daqui em diante chamado simplesmente de `lista`. Isto é feito através de sucessivas leituras de pares de chaves *x* e *y* (*x* < *y*).

Para cada par *x, y*, deve-se:

- Procurar *x* na `lista`. Se *x* não existir, incluir o novo elemento *x* no fina da lista e incrementar o contador `n`.

- Procurar *y* na `lista`. Se *y* não existir, incluir o novo elemento *y* no final da lista e incrementar o contador `n`.

- Atualizar os dados do elemento *x*. O atributo `EloSuc listaSuc` de *x* representa um ponteiro para o primeiro elemento da sua lista de sucessores. Este atributo deve ser atualizado incluindo-se no início desta lista um novo elemento do tipo `EloSuc`. Para tal, cria-se um novo elemento do tipo `EloSuc`, atualiza-se seu atributo para conter o elemento `Elo id` para conter um ponteiro para o elemento *y* da `lista` e seu atributo `EloSuc prox` para conter o antigo primeiro elemento desta mesma lista de sucessores (isto é, adiciona o elemento no início da lista de sucessores).

- Atualizar os dados do elemento *y*. O elemento *x* é predecessor de *y*, logo deve-se incrementar o atributo `contador` do elemento *y* para contabilizar que ele tem mais um predecessor.

Considere o método `debug()` da classe `OrdenacaoTopologica` que, após a leitura dos dados de entrada, imprime na tela a estrutura de dados resultante. A execução deste método para a estrutura de dados obtida a partir da leitura do arquivo do conjunto parcialmente ordenado contido na Figura 1 resultaria na saída ilustrada a seguir:

```
Debug
```

```
1 predecessores: 0, sucessores: 3 -> 2 -> NULL
2 predecessores: 1, sucessores: 10 -> 4 -> NULL
4 predecessores: 2, sucessores: 8 -> 6 -> NULL
6 predecessores: 1, sucessores: 3 -> NULL
10 predecessores: 2, sucessores: NULL
8 predecessores: 2, sucessores: NULL
3 predecessores: 2, sucessores: 5 -> NULL
5 predecessores: 2, sucessores: 8 -> NULL
7 predecessores: 0, sucessores: 9 -> 5 -> NULL
9 predecessores: 1, sucessores: 10 -> 4 -> NULL
```

Note que a inserção dos elementos da `lista` foram incluídos sempre no final da `lista` e os elementos da lista de sucessores foram incluídos no início da lista de sucessores.

Após a construção da estrutura de dados, nesta fase de entrada, o processo real de ordenação topológica pode ser realizado conforme descrito anteriormente. Uma vez que o processo consiste em selecionar repetidamente elementos com número nulo de predecessores, parece razoável selecionar de início todos estes elementos e colocá-los em uma outra lista encadeada.

Note que a lista encadeada original o conjunto *S* posteriormente não será mais necessária e por isso é possível reutilizar o mesmo atributo `prox` da `lista` para encadear os elementos que estarão nessa nova lista de elementos com zero predecessores. Esta operação de substituição de uma cadeia por outra ocorre com frequência em programas que envolvem o processamento de listas encadeadas e é expressa no algoritmo abaixo. Por conveniência, esta operação constrói a nova cadeia em ordem reversa.

```
/* busca elementos sem predecessores */
p = prim; prim = NULL;
WHILE p # NULL DO
     q = p;
     p = q->prox;
     IF q->contador == 0 THEN
          /* insere q na nova cadeia */
```

```
            q->prox = prim;
            prim = q;
      END
END
```

Note que o encadeamento entre os elementos não é perdido, pois aqueles que não possuem um número nulo de predecessores, e portanto não estão na nova configuração da lista, são acessíveis pela lista `EloSuc` dos elementos com zero predecessores.

Após isso, é possível continuar com a tarefa efetiva de ordenação topológica, isto é, gerar a sequência de saída. De forma resumida, isto pode ser feito como mostra o algoritmo abaixo:

```
para cada elemento q da lista de elementos com zero
predecessores
      imprimir a chave de q;
      decrementar o número de elementos da lista (n);
      para cada elemento t da lista de sucessores de q
            decrementar o contador do predecessor de t
            se o contador de t se tornar zero,
                  insere este elemento no fim da lista de
elementos com zero predecessores
            remover o elemento t da lista de sucessores de q
      remover o elemento q da lista de elementos com zero
predecessores
fim-para
```

Isto completa o programa para realizar a **ordenação topológica**. Note que foi introduzido um contador `n` para contabilizar os elementos do conjunto *S* gerados na fase de entrada. Este contador é decrementado cada vez que um elemento é inserido na sequência de saída, durante a fase de saída. No final da execução do programa, espera-se que `n` seja igual a zero para todo elemento do conjunto. Caso isto não ocorra, é uma constatação de que restaram elementos na estrutura de modo tal que nenhum deles deixa

de apresentar um predecessor. Neste caso, é evidente que o conjunto *S* não apresenta a propriedade da ordenação parcial.

A fase de saída, acima descrita, é um exemplo de um processo em que é mantida uma lista que se expande e contrai, isto é, cujos elementos são inseridos e removidos em uma ordem predeterminada. Trata-se, portanto, de um exemplo de um processo que utiliza a total flexibilidade que a **Lista Encadeada** oferece.

Considere o arquivo disponibilizado no Moodle `OrdenacaoTopologica.java` com a classe `OrdenacaoTopologica`, além das classes internas `Elo` e `EloSuc`. Pede-se para implementar os métodos `realizaLeitura(String nomeArq)`, `executa()` e `debug()`, além de demais métodos auxiliares para a execução do programa. O método `executa()` retorna `true` se a entrada de dados é um conjunto parcialmente ordenado e `false` caso contrário.

Além disso, o método `executa()` é encarregado de chamar todos os demais métodos auxiliares (incluindo o método `debug()`) necessários para a execução do programa. Por fim, o formato de arquivo de entrada está igualmente disponível no Moodle.

Para a entrada de dados mostrada anteriormente, a execução do método `main` da classe `Main.java` resultaria na saída exibida abaixo:

```
Debug
1 predecessores: 0, sucessores: 3 -> 2 -> NULL
2 predecessores: 1, sucessores: 10 -> 4 -> NULL
4 predecessores: 2, sucessores: 8 -> 6 -> NULL
6 predecessores: 1, sucessores: 3 -> NULL
10 predecessores: 2, sucessores: NULL
8 predecessores: 2, sucessores: NULL
3 predecessores: 2, sucessores: 5 -> NULL
5 predecessores: 2, sucessores: 8 -> NULL
7 predecessores: 0, sucessores: 9 -> 5 -> NULL
9 predecessores: 1, sucessores: 10 -> 4 -> NULL

Ordenacao topologica
```

```
7 1 9 2 10 4 6 3 5 8
Conjunto é parcialmente ordenado.
```

Após implementar o algoritmo supracitado, você deve implementar um método que **gere um grafo direcionado acíclico de N vértices e um determinado número de arestas (certifique-se de que, durante o processo de geração, a inserção de uma aresta não crie um ciclo)**. Você deve então usar esse método desenvolvido para gerar grafos artificiais com os seguintes números de vértices **V**: 10, 20, 30, 40, 50, 100, 200, 500, 1.000, 5.000, 10.000, 20.000, 30.000, 50.000 e 100.000.

Neste ponto, você deve realizar os seguintes experimentos computacionais: executar o algoritmo de Ordenação Topológica para todos os grafos artificialmente criados. **Para minimizar eventuais ruídos na medição de tempo, rode o algoritmo 10 vezes consecutivas para cada configuração de grafo e tome o tempo médio, conforme explicado em sala de aula.**

Se, ao rodar os experimentos acima, você encontrar problemas ocasionados por limitação de poder computacional a partir de um determinado tamanho de grafo, deve rodar os experimentos até o último tamanho factível, isto é, que seja possível de rodar no computador utilizado. Ademais, não se esqueça de discriminar no relatório o ambiente computacional adotado: quantidade e tipo de memória, processador, sistema operacional, versão do Java, IDE e versão, etc.

Por fim, gere gráficos ilustrando o tempo de execução do algoritmo de Ordenação Topológica como uma função do tamanho da entrada. Compare os gráficos obtidos com o gráfico esperado referente à complexidade teórica do algoritmo.

Algumas considerações para o trabalho:

1. Escrever um breve relatório contextualizando a estrutura de dados, a implementação (funcionalidade de cada método) e os experimentos realizados;

2. Você deve apresentar os tempos e gráficos obtidos, analisando-os e expondo as conclusões obtidas;

3. Não deixe de relacionar as referências utilizadas ao longo do desenvolvimento do trabalho;

4. Não deixe para tirar suas dúvidas no último momento;

5. Documente qualquer problema que não conseguir resolver;

6. O programa deve ser bem testado!

7. O trabalho deve ser **realizado em trios**;

8. A entrega deve ser realizada na respectiva **tarefa criada no Moodle** da disciplina até as 15:59 do **dia 07/12**;

9. O trabalho deve ser apresentado na aula do dia **07/12 às 16h**.

OBSERVAÇÕES:

1. O trabalho é obrigatório e vale 30% da Nota 2;

2. Trabalhos com alto grau de semelhança levarão nota zero; e

3. Não entregar o código, ou não entregar o relatório, ou não apresentar o trabalho implica em nota zero no trabalho.

Referências

[1] Niklaus Wirth. *Algoritmos e Estruturas de Dados*. LTC Editora, 2008.

[2] Adaptado do enunciado de trabalho de “Estruturas de Dados 1” adotado pela Profa. Adriana Alvim.