Ordenação Externa

A ordenação externa envolve arquivos compostos por um número de registros que é maior do que a memória interna do computador pode armazenar.

Os métodos de ordenação externa possuem particularidades que os diferenciam dos métodos de ordenação interna.

Em ambos os casos o problema é o mesmo: rearranjar os registros de um arquivo em ordem ascendente ou descendente.

Ordenação Externa

Entretanto, na ordenação externa as estruturas de dados têm que levar em conta o fato de que os dados estão armazenados em unidades de memória externa, relativamente muito mais lentas do que a memória principal.

Nas memórias externas, tais como **fitas** e **discos magnéticos**, os dados são armazenados como um arquivo seqüencial, onde apenas um registro pode ser acessado em um dado momento.

Esta é uma restrição forte se comparada com as possibilidades de acesso da estrutura de dados do tipo vetor.

UNIVASF

Ordenação Externa

Existem três importantes fatores que tornam os algoritmos para ordenação externa diferentes dos algoritmos para ordenação interna, sendo:

1. O custo para acessar um item é algumas ordens de grandeza maior do que os custos de processamento na memória interna.

   Custo principal ➔ o custo de transferir dados entre a memória interna e a memória externa.

2. Existem restrições severas de acesso aos dados.

   (a) Itens armazenados em fita magnética só podem ser acessados de forma seqüencial.

Ordenação Externa

(b) Itens armazenados em disco magnético podem ser acessados diretamente, mas a um custo maior do que o custo para acessar seqüencialmente, o que contra-indica o uso do acesso direto.

3. O desenvolvimento de métodos de ordenação externa é muito dependente do estado atual da tecnologia. A grande variedade de tipos de unidades de memória externa pode tornar os métodos de ordenação externa dependentes de vários parâmetros que afetam seus desempenhos.

UNIVASF

Ordenação Externa

Fatores associados à eficiência dos Métodos de Ordenação Externa:

O aspecto sistema de computação deve ser considerado no mesmo nível do aspecto algorítmico.

A grande ênfase deve ser na minimização do número de vezes que cada item é transferido entre a memória interna e a memória externa.

Cada transferência deve ser realizada de forma tão eficiente quanto as características dos equipamentos disponíveis permitam.

UNIVASF

Ordenação Externa

Método de ordenação externa mais importante

⬇

Ordenação por Intercalação

Como vimos anteriormente, intercalar significa combinar dois ou mais blocos ordenados em um único bloco ordenado através de seleções repetidas entre os itens disponíveis em cada momento.

A intercalação é utilizada como uma operação auxiliar no processo de ordenação.

Ordenação Externa

Como veremos, a estratégia geral é a mesma. Contudo, algumas particularidades existem na implementação da intercalação na memória externa:

1. É realizada uma primeira passada sobre o arquivo, quebrando-o em blocos do tamanho da memória interna disponível. Cada bloco é então ordenado na memória interna.

2. Os blocos ordenados são intercalados, fazendo-se varias passadas sobre o arquivo. A cada passada são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

UNIVASF

Ordenação Externa

Os algoritmos para ordenação externa devem procurar reduzir o número de passadas sobre o arquivo.

Os bons métodos de ordenação geralmente envolvem no total menos do que 10 passadas sobre o arquivo.

Uma boa medida de complexidade de um algoritmo de ordenação por intercalação é o número de vezes que um item é lido ou escrito na memória auxiliar.

UNIVASF

Ordenação Externa

Ordenação por Intercalação – Mergesort

Princípio:

1. Partir o arranjo em dois.
2. Intercalar dois arranjos independentes.

Vantagens:

1. Pior caso: **O** (n log n)
2. Só precisa de acesso seqüencial aos dados.
3. Boa opção quando os dados estão em uma lista encadeada.

Desvantagem: Espaço de armazenamento adicional é necessário.

UNIVASF

Ordenação Externa

Ordenação de Seqüências - Fusão (intercalação ou merge) Direta

Corresponde a combinar duas ou mais seqüências ordenadas para formar uma única seqüência ordenada através da aplicação de repetidas seleções entre os componentes acessíveis em cada ocasião.

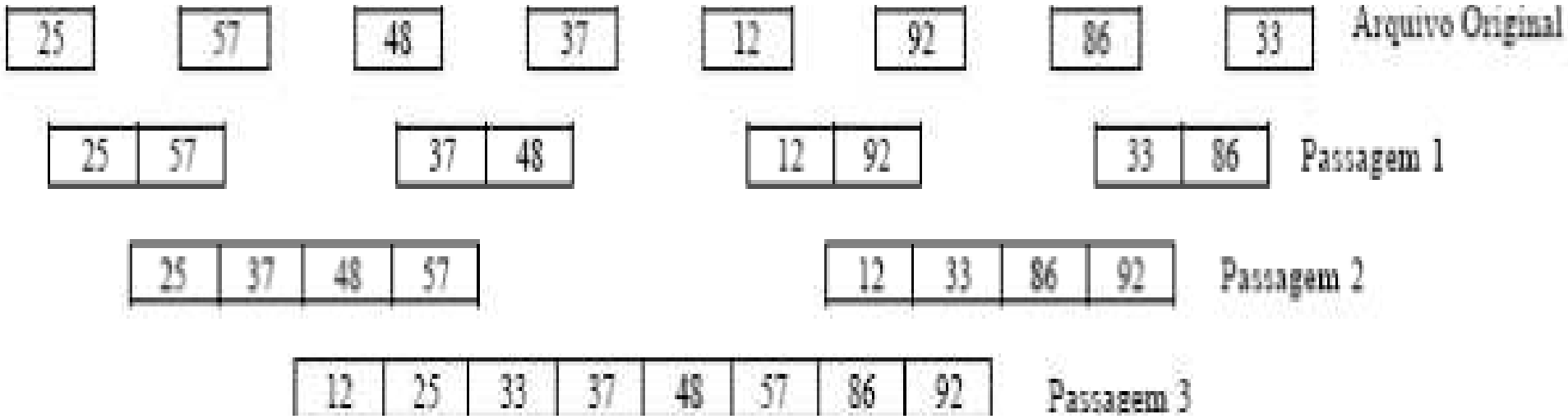
A operação de fusão é muito mais simples que a de ordenação, sendo empregada, como operação auxiliar, no processo mais complexo de ordenação seqüencial.

Ordenação Externa

O método Merge Sort (intercalação direta) funciona da seguinte maneira: divida o arquivo em **n** sub-arquivos de tamanho **1** e intercale pares de arquivos adjacentes.

Temos, então, aproximadamente **n/2** arquivos de tamanho **2**.

Repita esse processo até restar apenas um arquivo de tamanho **n**. Exemplo:



Ordenação Externa

As fases de particionamento não oferecem nenhuma contribuição ao processo de ordenação propriamente dito, uma vez que elas, de modo algum, efetuam a permutação de elementos.

Estas operações podem ser eliminadas, através da combinação da fase de particionamento com a de fusão.

Ao invés de se efetuar uma fusão para produzir uma seqüência única, o resultado do processo de fusão é imediatamente redistribuído em duas ou mais sequências (fitas), as quais constituirão as fontes de dados que alimentarão o passo subseqüente. Este método é chamado: **Fusão de uma Única Fase** ou **Fusão Balanceada**.

UNIVASF

Ordenação Externa

Estudaremos agora o processo de **Intercalação Balanceada de Vários Caminhos**, para tal consideraremos o processo de ordenação externa quando o arquivo a ser ordenado encontra-se armazenado em fita magnética.

Exemplo:

Considere um arquivo composto pelos registros com as seguintes chaves:

A S O R T I N G A N D M E R G I N G E X A M P L E

Os 25 registros devem ser ordenados de acordo com as chaves e colocados em uma fita de saída. Neste caso os registros são lidos um após o outro.

Ordenação Externa

Assuma que a memória interna do computador a ser utilizado só tem espaço para 3 registros, e o número de unidades de fita magnética é 6.

Chaves: A S O R T I N G A N D M E R G I N G E X A M P L E

Na 1ª etapa o arquivo é lido de 3 em 3 registros. Cada bloco de 3 registros é ordenado e escrito em uma das fitas de saída.

Para o exemplo são lidos os registros:

1. **A S O** e escrito o bloco **A O S** na fita 1.
2. A seguir são lidos os registros **R T I** e escrito o bloco **I R T** na fita 2, e assim por diante, conforme mostra o exemplo a seguir.

Ordenação Externa

Chaves: A S O R T I N G A N D M E R G I N G E X A M P L E

fita 1:	A O S	D M N	A E X
fita 2:	I R T	E G R	L M P
fita 3:	A G N	G I N	E

Observação: Quando 3 fitas são utilizadas denomina-se intercalação-de-3-caminhos.

Na 2ª etapa os blocos ordenados devem ser intercalados.

1. O 1º registro de cada uma das 3 fitas é lido para a memória interna, ocupando toda a memória interna.

Ordenação Externa

2. O registro contendo a menor chave dentre as 3 é retirado e colocado em uma fita de saída; e o próximo registro da fita que continha tal chave é lido para a memória interna.

3. Repete-se o processo até que o 3º registro de um dos blocos é lido, o que faz com que a fita em questão fique inativa até que o 3º registro das outras fitas também sejam lidos e escritos na fita de saída, formando um bloco de 9 registros ordenados.

A seguir o 2º bloco de 3 registros de cada fita é lido para formar outro bloco ordenado de 9 registros, o qual é escrito em uma outra fita.

Ordenação Externa

Ao final 3 novos blocos ordenados são obtidos, conforme mostra o exemplo:

Fitas obtidas na primeira etapa:

fita 1:	A O S	D M N	A E X
fita 2:	I R T	E G R	L M P
fita 3:	A G N	G I N	E

Fitas gerada na segunda etapa:

fita 4:	A A G	I N O	R S T
fita 5:	D E G	G I M	N N R
fita 6:	A E E	L M P	X

A seguir mais uma intercalação-de-3-caminhos das fitas 4, 5 e 6 para as fitas 1, 2 e 3 completa a ordenação.

Ordenação Externa

Considerações sobre a Intercalação

Se o arquivo exemplo tivesse um número maior de registros, então vários blocos ordenados de 9 registros seriam formados nas fitas 4, 5 e 6.

Neste caso, a segunda passada produziria blocos ordenados de 27 registros nas fitas 1, 2 e 3.

A terceira passada produziria blocos ordenados de 81 registros nas fitas 4, 5 e 6, e assim sucessivamente, até obter-se um único bloco ordenado.

UNIVASF

Ordenação Externa

Neste ponto cabe a seguinte pergunta: quantas passadas são necessárias para ordenar um arquivo de tamanho arbitrário?

Considere um arquivo contendo **n** registros (neste caso cada registro contém apenas uma palavra) e uma memória interna de **m** palavras.

A passada inicial sobre o arquivo produz **n/m** blocos ordenados (se cada registro contiver **k** palavras, **k > 1**, então teríamos **n/m/k** blocos ordenados.)

Ordenação Externa

Seja **P** uma função de complexidade tal que **P**(**n**) é o número de passadas para a fase de intercalação dos blocos ordenados. Seja **f** o número de fitas utilizadas em cada passada.

Então, para uma intercalação-de-f-caminhos o número de passadas é: $\mathbf{P}(\mathbf{n}) = \log_{\mathbf{f}} (\mathbf{n}/\mathbf{m})$

No exemplo anterior, temos **n**=25, **m**=3 e **f**=3.

Logo,

$$\mathbf{P}(\mathbf{n}) = \log_{\mathbf{3}} (25/3) = 2$$

Grafos

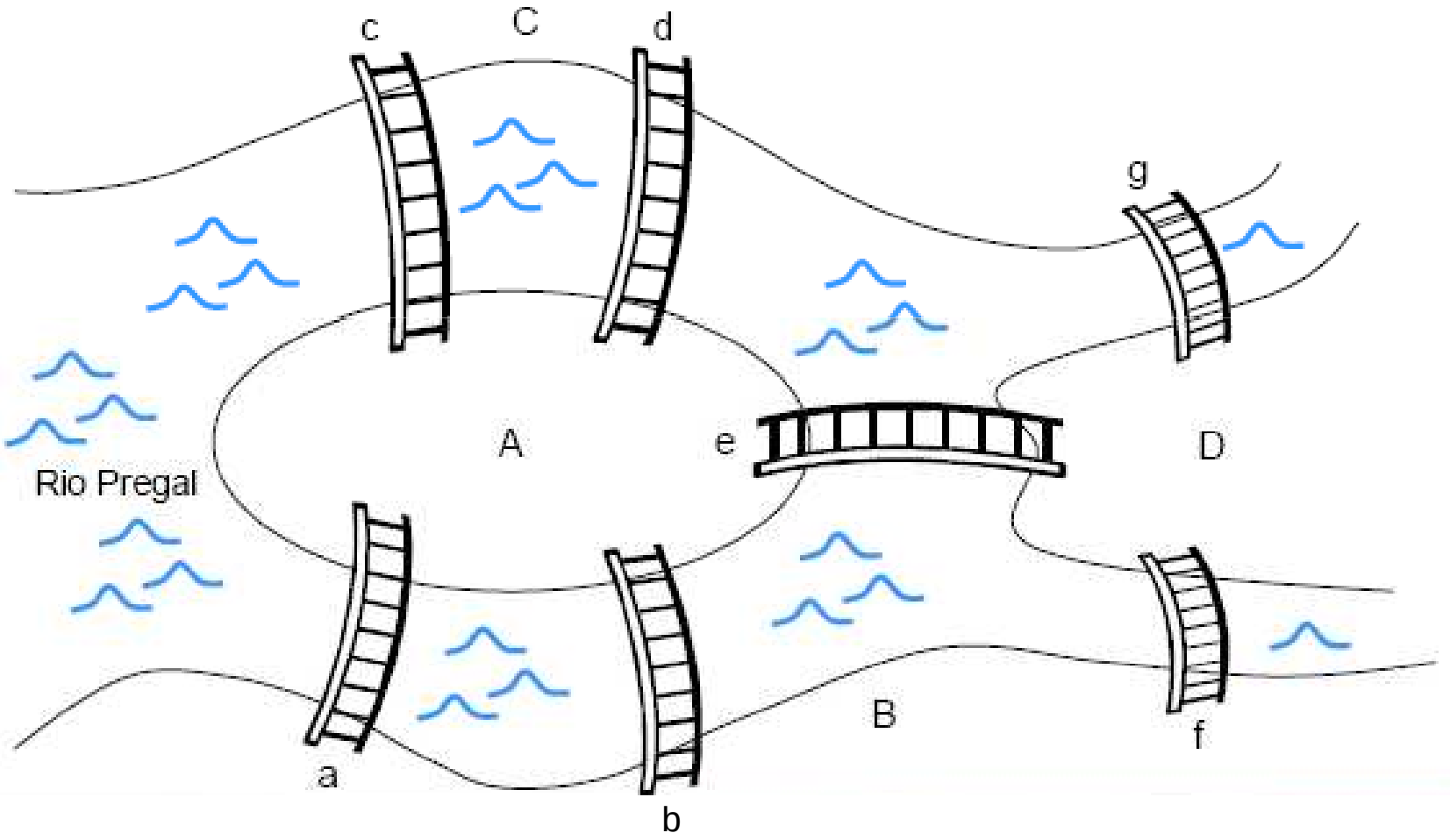
Os primeiros slides relacionados histórico dos grafos foram baseados nos slides do professor José Augusto Baranauskas do Departamento de Física e Matemática – FFCLRP-USP.

Grafos - Histórico

- A primeira evidência sobre grafos remonta a 1736, quando Euler fez uso deles para solucionar o problema clássico das pontes de Koenigsberg;
- Na cidade de Koenigsberg (na Prússia Oriental), o rio Pregal flui em torno da ilha de Kneiphof, dividindo-se em seguida em duas partes;
- Assim sendo, existem quatro áreas de terra que ladeiam o rio: as áreas de terra (A-D) estão interligadas por sete pontes (a-g);
- O problema das pontes de Koenigsberg consiste em determinar se, ao partir de alguma área de terra, é possível atravessar todos as pontes exatamente uma vez, para, em seguida, retornar à área de terra inicial.

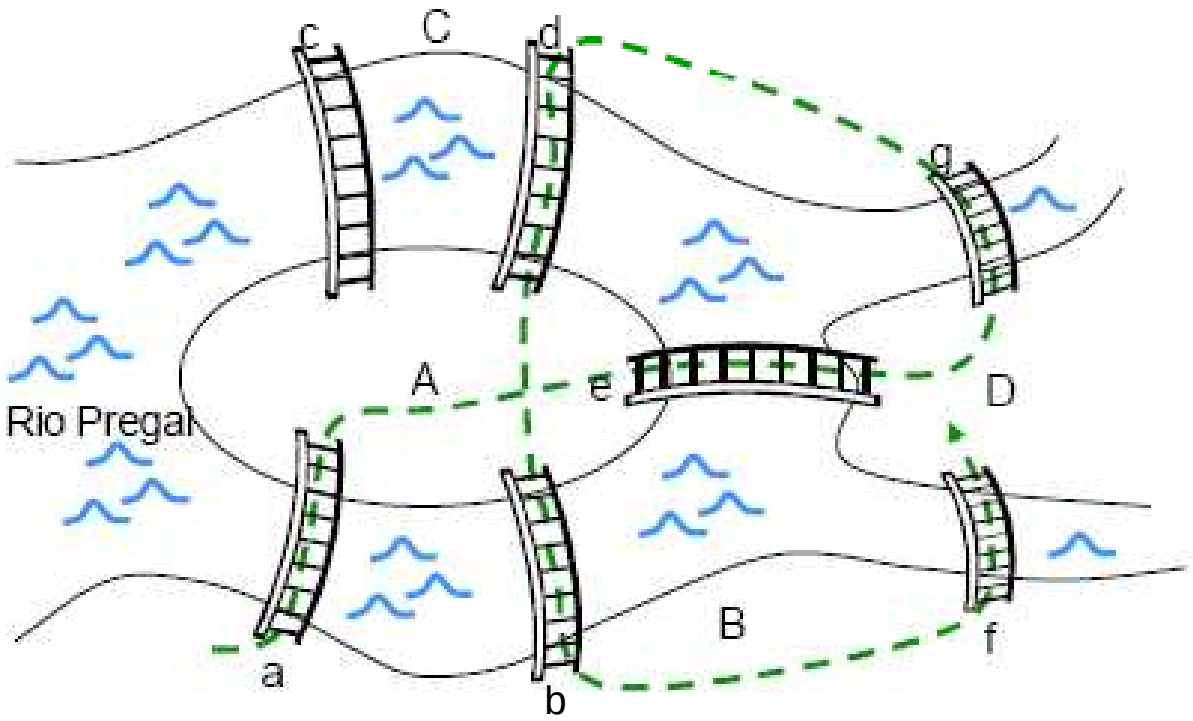
Grafos - Histórico

- É possível caminhar sobre cada ponte exatamente uma única vez e retornar ao ponto de origem?



Grafos - Histórico

• Um caminho possível consistiria em iniciar na área de terra **B**, atravessar a ponte **a** para a ilha **A**; pegar a ponte **e** para chegar à área **D**, atravessar a ponte **g**, chegando a **C**; cruzar a ponte **d** até **A**; cruzar a ponte **b** até **B** e a ponte **f**, chegando a **D.**



Grafos - Histórico

- Esse caminho não atravessa todas as pontes uma vez, nem tampouco retorna à área inicial de terra B.

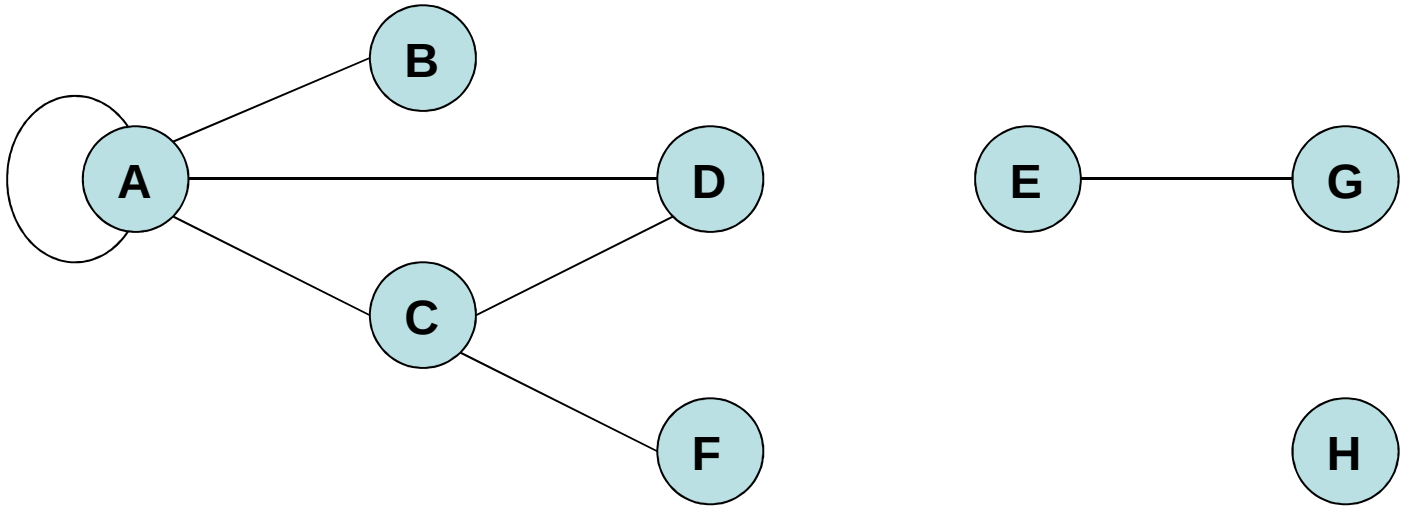
- Euler provou que não é possível o povo de Koenigsberg atravessar cada ponte exatamente uma vez, retornando ao ponto inicial.

- Ele resolveu o problema, representando as áreas de terra como vértices e as pontes como arestas de um grafo (na realidade, um multigrafo).

Grafos - Definições

• Um grafo consiste num conjunto de nós (ou vértices) e num conjunto de arcos (ou arestas).

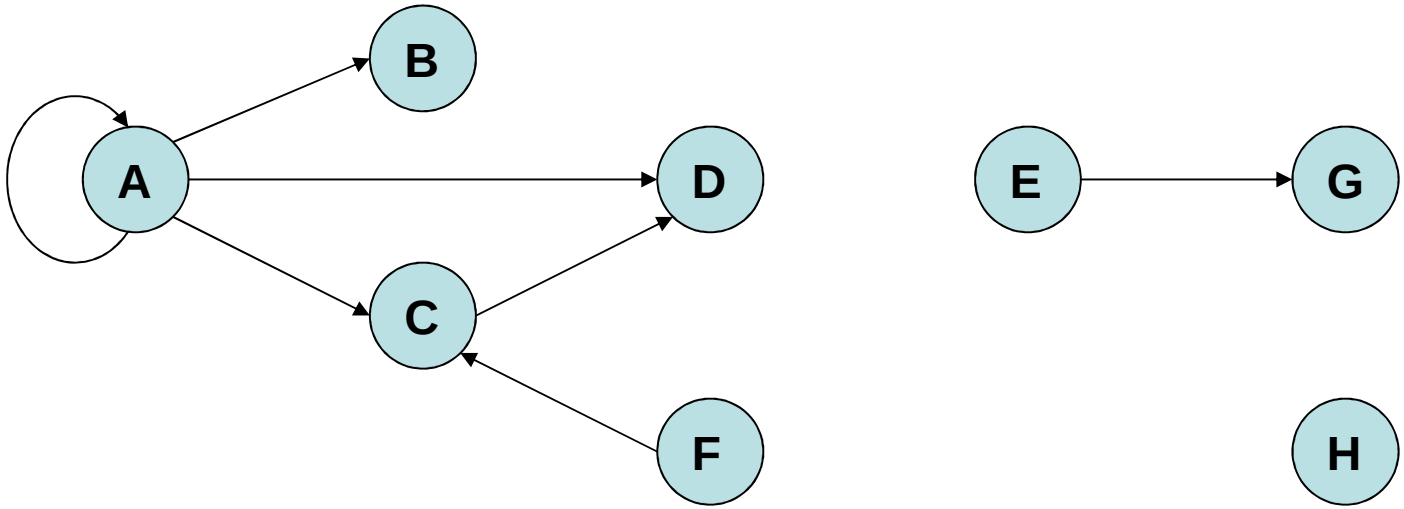
• Cada arco num grafo é especificado por um par de nós.



• A seqüência de nós é {A,B,C,D,E,F,G,H},e o conjunto de arcos é {(A,B), (A,D), (A,C), (C,D), (C,F), (E,G), (A,A)}.

Grafos - Definições

• Se os pares de nós que formam os arcos forem pares ordenados, diz-se que o grafo é um grafo orientado (ou dígrafo). Exemplos:



• As setas entre os nós representam arcos. A ponta de cada seta representa o segundo nó no par ordenado de nós que forma um arco, e o final de cada seta representa o primeiro nó no par. O conjunto de arcos do grafo acima é {<A,B>, <A,C>, <A,D>, <C,D>, <F,C>, <E,G>, <A,A>}.

Grafos - Definições

• Note que foram usados parênteses para indicar um par não-ordenado (desordenado) e chaves angulares para indicar um par ordenado.

• Concentraremos inicialmente nosso estudo nos dígrafos.

• Observe que, como vimos, um grafo não precisa ser uma árvore, mas uma árvore tem de ser um grafo. Note também que um nó não precisa ter arcos associados a ele (por exemplo, nó *H* no grafo anterior).

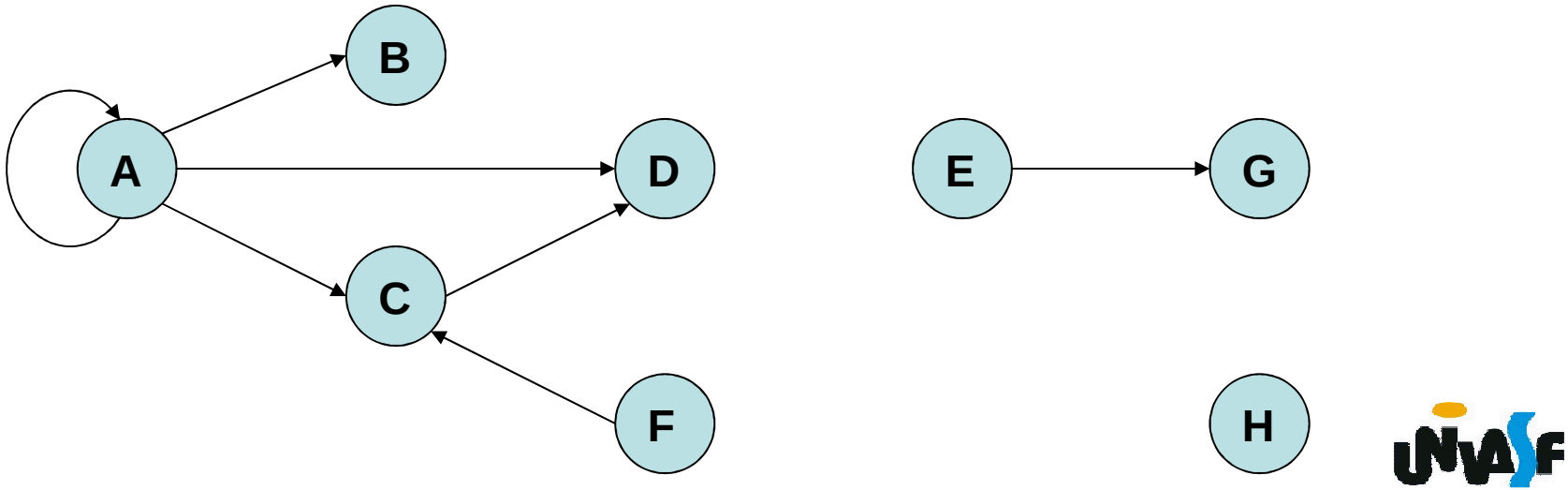
• Um nó *n* ***incide*** em um arco *x* se *n* for um de seus dois nós no par ordenado de nós que constituem *x*. (Dizemos também que *x* incide em *n*.)

Grafos - Definições

• O ***grau*** de um nó é o número de arcos incidentes nesse nó.

• O ***grau de entrada*** de um nó *n* é o número de arcos que têm *n* como cabeça, e o ***grau de saída*** de *n* é o número de arcos que têm *n* como terminação da seta.

Por exemplo, o nó C no grafo abaixo tem grau de entrada 2, grau de saída 1 e grau 3.



Grafos - Definições

• Um nó *n* será ***adjacente*** a um nó *m* se existir um arco de *m até n.* Se *n* for adjacente a *m, n* será chamado ***sucessor*** de *m* e *m* será um ***predecessor*** de *n.*

• Uma ***relação*** *R* num conjunto A é uma seqüência de pares ordenados de elementos de A.

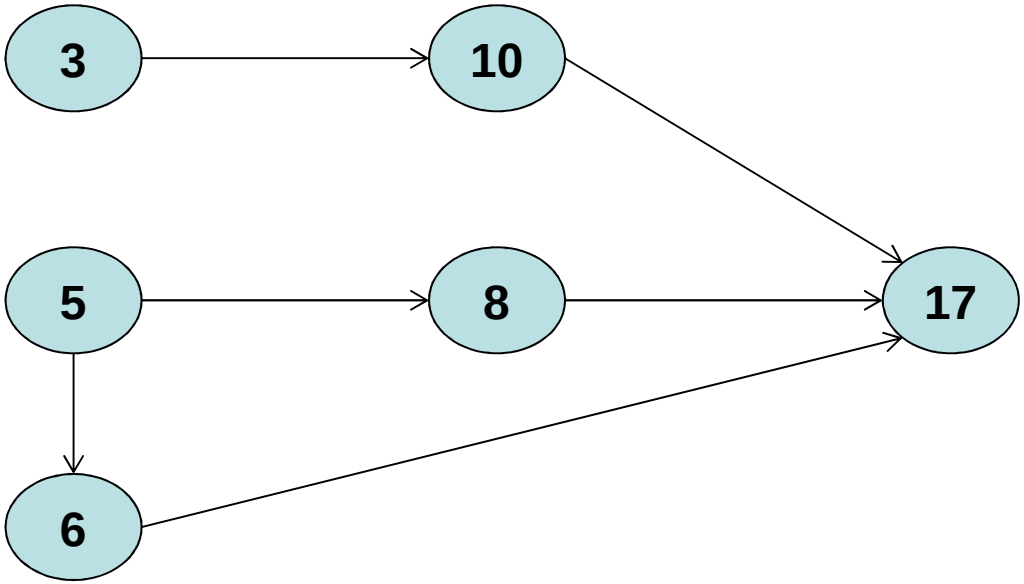
Por exemplo, se *A* = {3, 5, 6, 8, 10, 17}, o conjunto *R* = {<3,10>, <5,6>, <5,8>, <6,17>, <8,17>, <10,17>} será uma relação. Se <*x, y*> for um membro de uma relação *R,* diz-se que *x* está relacionado a *y* em *R.*

A relação *R* anterior pode ser descrita dizendo-se que *x* está relacionado com y se x for menor que *y e o* resto obtido a partir da divisão de *y* por *x* for ímpar.

Grafos - Definições

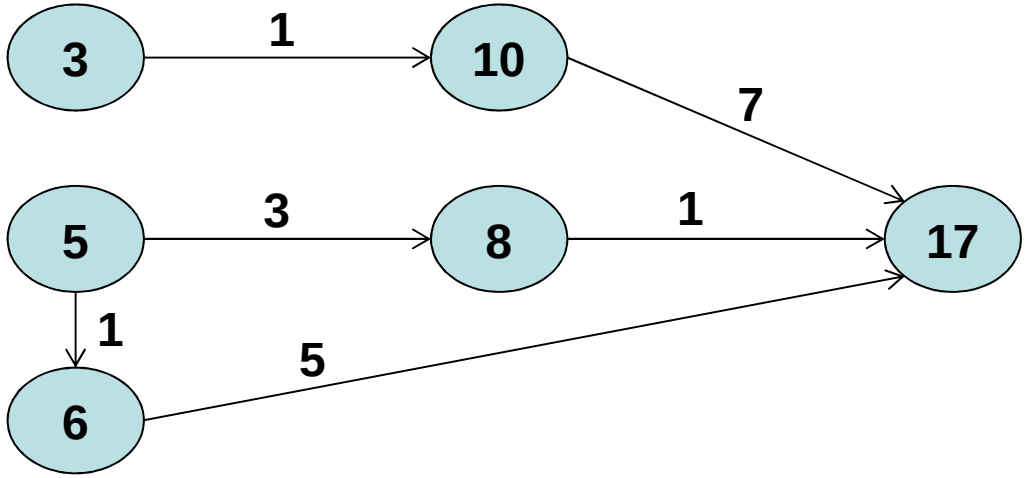
Uma relação pode ser representada por um grafo no qual os nós representam o conjunto básico e os arcos representam os pares ordenados da relação.

O grafo abaixo representa a relação anterior.



Grafos - Definições

Um número pode ser associado a cada arco de um grafo, como no grafo abaixo.



Neste grafo, o número associado a cada arco é o resto obtido da divisão do inteiro posicionado na cabeça do arco pelo inteiro posicionado em sua terminação. Um grafo desse tipo, no qual existe um número associado a cada arco, é chamado ***grafo ponderado*** ou ***rede.*** O número associado a um arco é chamado ***peso.***

Grafos - Definições

Identificaremos várias operações primitivas que serão úteis ao lidar com grafos. A operação *ligar(a,b)* introduz um arco do nó *a* até o nó *b* se ainda não existir *um; ligarComPeso(a,b,x)* insere um arco de *a* até *b* com peso *x* num grafo ponderado; *remover(a,b)* e *removerComPeso (a,b,x)* eliminam um arco de *a* até *b*, caso exista *(removerComPeso* define também *x* com seu peso). Embora possamos também acrescentar ou eliminar nós de um grafo, discutiremos essas possibilidades posteriormente. A função *adjacente(a,b)* retorna *true* se *b* for adjacente a *a*, e *false,* caso contrário.

Grafos - Definições

Um ***caminho de comprimento k*** do nó *a* ao nó *b* é definido como uma seqüência de *k + 1* nós *n1, n2,..., nk + 1,* tal que *n1 = a, nk + 1 = b* e *adjacente(ni, ni + i)* é *true* para todo *i* entre *1* e *k.* Se, para algum inteiro *k*, existir um caminho de comprimento *k* entre *a* e *b,* existirá um ***caminho*** de *a* até *b.*

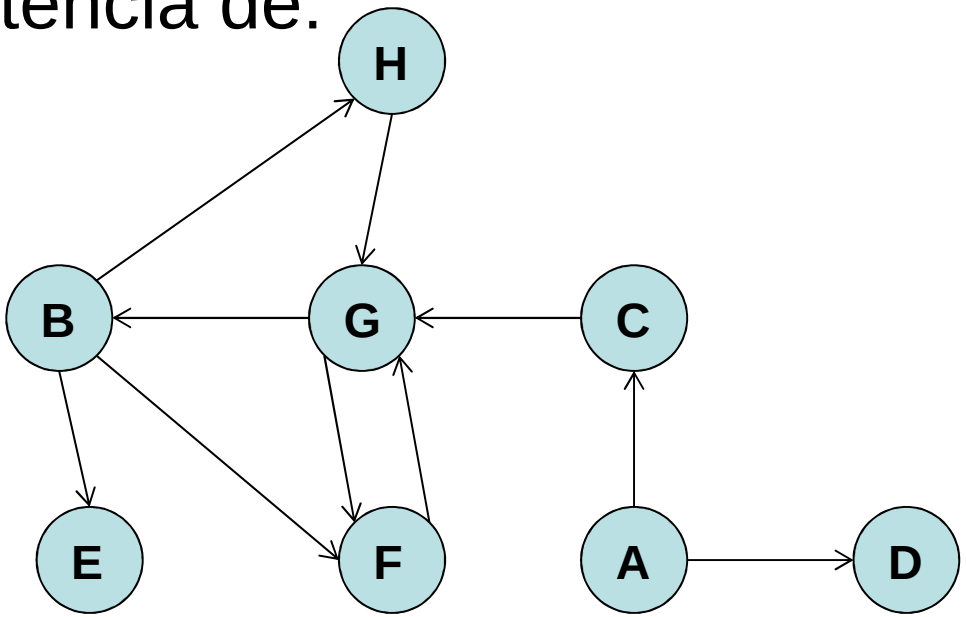
Um caminho de um nó para si mesmo é chamado ***ciclo***.

Se um grafo contiver um ciclo, ele será ***cíclico***; caso contrário, será ***acíclico***.

Um grafo acíclico orientado é chamado ***dag***, uma aglutinação das iniciais de *directed acyclic graph.*

Grafos - Definições

Uma análise do grafo abaixo possibilita perceber a existência de:



- um caminho de comprimento 1 de A até C;
- dois caminhos de comprimento 2 de *B até G;*
- *e um caminho de* comprimento 3 de A até F*.*

Note que não existe um caminho de B até C.

Existem ciclos de B para B, de F para F e de H para H.

Grafos - Representação

Com base no que foi exposto e considerando que o número de nós no grafo seja constante, ou seja, os arcos podem ser acrescentados ou eliminados, mas os nós não.

Que estrutura poderia ser utilizada para sua representação?

Um vetor bidimensional.

Se considerarmos que os nós de um grafo serão numerados de 0 a *MAXNODES - 1* e nenhuma informação é atribuída (associada) a eles. Além disso, também consideraremos a existência de arcos sem a associação de pesos ou outras informações aos mesmos.

UNIVASF

Grafos - Representação

Nesse caso, o grafo poderia ser declarado simplesmente por:

#define MAXNODES valorInteiroPositivo

int adj[MAXNODES][MAXNODES];

O valor de *adj[i][j]* é *TRUE* ou *FALSE,* dependendo de o nó *j* ser ou não adjacente ao nó *i.* O vetor bidimensional *adj[ ][ ]* é chamado ***matriz de adjacência.***

Grafos - Representação

Seguindo esta linha de raciocínio, podemos vislumbrar uma representação mais completa para um grafo, como sendo:

```
#define MAXNODES valorInteiroPositivo
struct node {
   /* informacao associada a cada noh */
};
struct arc {
   int adj;
   /* informacao associada a cada arco */
};
struct graph {
   struct node nodes[MAXNODES];
   struct arc arcs[MAXNODES][MAXNODES];
};
struct graph g;
```

Grafos - Representação

Onde, assim como na representação anterior, cada nó do grafo é representado por um inteiro entre 0 e *MAXNODES* - 1, e o campo vetor *nodes representa as informações corretas associadas a* cada nó. O campo vetor *arcs* é um vetor bidimensional representando todo possível par ordenado de nós. O valor de *g.arcs[i][j].adj* é *TRUE* ou *FALSE,* dependendo de o nó *j* ser ou não adjacente ao nó *i.* Neste caso, o vetor bidimensional *g.arcs[][].adj* é a matriz de adjacência. No caso de um grafo ponderado, cada arco poderá também receber a atribuição de informações.

Grafos - Representação

Considerando a representação de um grafo apenas pela matriz de adjacência, ou seja, onde não serão atribuídos pesos e outras informação à arestas e nem informações ao nós. Implemente a operação *ligar(a,b)*.

```
void ligar (int adj[][MAXNODES], int node1, int
node2)
{
   adj[node1][node2] = 1;
}
```

Grafos - Representação

Considerando a representação de um grafo em questão. Implemente a operação *remover (a,b)*.

```
void remover (int adj[][MAXNODES], int node1, int node2)
{
   adj[node1][node2] = 0;
}
```

Grafos - Representação

Considerando a representação de um grafo em questão. Implemente a operação *adjacente (a,b)*.

```
int adjacente (int adj[][MAXNODES], int node1, int
node2)
{
   return adj[node1][node2];
}
```

Grafos - Representação

Proponha uma estrutura capaz de representar um grafo ponderado com um número fixo de nós.

```
#define MAXNODES valorInteiroPositivo
struct arc {
   int adj;
   int peso;
};
struct arc g[MAXNODES][MAXNODES];
```

Grafos - Representação

Considerando a representação para um grafo ponderado apresentada, implemente a operação *ligarP(...)*.

```
void ligarP (struct arc adj[][MAXNODES],
int node1, int node2, int peso)
{
   adj[node1][node2].adj = 1;
   adj[node1][node2].peso = peso;
}
```

Grafos - Representação

Considerando a representação para um grafo ponderado apresentada, implemente a operação *removerP(...)*.

```
void removerP (struct arc adj[][MAXNODES],
int node1, int node2, int peso)
{
   adj[node1][node2].adj = 0;
   adj[node1][node2].peso = peso;
}
```

Grafos - Representação

Considerando a representação para um grafo ponderado apresentada, implemente a operação *adjacenteP(...)*.

```
void adjacenteP (struct arc adj[][MAXNODES],
int node1, int node2)
{
   return adj[node1][node2].adj;
}
```

Grafos - Aplicação

Examinaremos agora um exemplo de aplicação de um grafo. Vamos supor o seguinte problema: Considerando a existência de **n** cidades. Considerando que alguns pares destas cidades possuem estradas que as ligam. Determine se é possível sair de uma cidade **A** e chegar em uma cidade **B** utilizando exatamente **nr** estradas.

Determine uma estratégia para solucionar o problema apresentado.

Grafos - Aplicação

Uma estratégia para a solução é a seguinte: crie um grafo com as cidades como nós e as estradas como arcos. Para achar um caminho de comprimento *nr* do nó *A* ao nó *B,* procure um nó *C* de modo que exista um arco de *A* até *C* e um caminho de comprimento *nr - 1* de *C* até *B.* Se essas condições forem atendidas para um nó *C*, o caminho desejado existirá; se elas não forem atendidas para qualquer nó *C*, o caminho não existirá.

O algoritmo usará uma função recursiva auxiliar, *procurarCaminho(k,a,b)*. Essa função retornará *true* se existir um segmento de comprimento *k* de *A* até *B,* e *false*, caso contrário.

Grafos - Aplicação

Implemente na linguagem C a função procurarCaminho(...).

```
procurarCaminho(int k, int a, int b)
{
   if (k == 1)
      return (adjacente (a, b));
   for (c = 0; c < n; ++c)
      if (adjacente(a,c) && procurarCaminho (k - 1,
c, b))
         return(1);
   return(0);
}
```

Grafos - Aplicação

Visando possibilitar o teste da função apresentada implemente, na linguagem C, uma função *main()*, que receba uma linha de entrada contendo quatro inteiros seguidos por um número qualquer de linhas de entrada com dois inteiros cada uma. O primeiro inteiro na primeira linha, *n,* representa um número de cidades, que, para simplificar, serão numeradas de 0 a *n* - 1*. O* segundo e o terceiro inteiro nessa linha estão entre 0 e *n* - 1 e representam duas cidades. Queremos sair da primeira cidade para a segunda usando exatamente *nr* estradas, onde *nr* é o quarto inteiro na primeira linha de entrada. Cada linha de entrada subsequente contém dois inteiros representando duas cidades, indicando que existe uma estrada da primeira cidade até a segunda. A última linha na sequência conterá dois valores inteiros negativos. O problema é determinar se existe um percurso do tamanho solicitado pelo qual se possa viajar da primeira cidade para a segunda.