Estruturas de Información

Grafos

Departamento de Engenharia Informática (DEI/ISEP)
Fátima Rodrigues
mfc@isep.ipp.pt

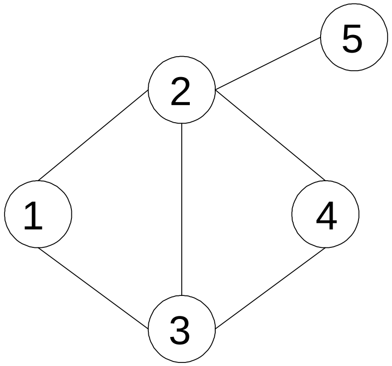
Definição

Informalmente um grafo é um conjunto não vazio de pontos (vértices, nós, nodos) podendo ou não haver ligações entre eles através de linhas (ramos, arcos, arestas)

$G = (V, E)$ V conjunto dos Vértices
E conjunto dos Ramos

A cada ramo do grafo existe associado um par de vértices do grafo

$\forall_{r \in E} \quad r \rightarrow (u,v)$ tal que $u,v \in V$



Ordem de um grafo é o número de vértices que o grafo contem

Tamanho de um grafo é o número de ramos que o grafo contem

Aplicações

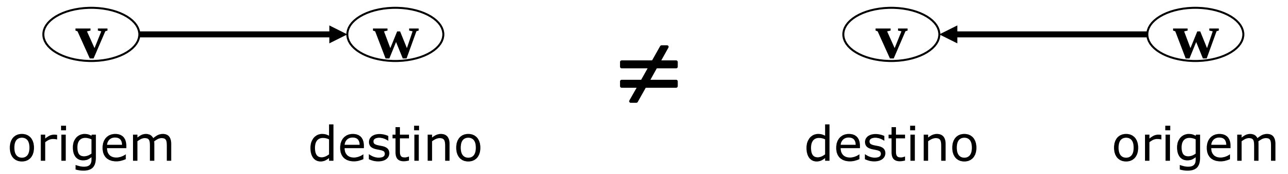
Os grafos são usados para modelar uma infinidade de problemas do mundo real, tudo o que tem subjacente conectividade de informação:

- mapas (em sistemas de informação geográfica)
- transportes (redes viárias e aéreas)
- em engenharia electrotécnica (distribuição eléctrica, circuitos eléctricos), redes de computadores, redes telefónicas,...
- escalonamento de tarefas em projectos, conhecido pelo método PERT/CPM
- Outra aplicação importante é no algoritmo "mark-sweep" do mecanismo de Garbage Collection, usado na gestão de memória do Java

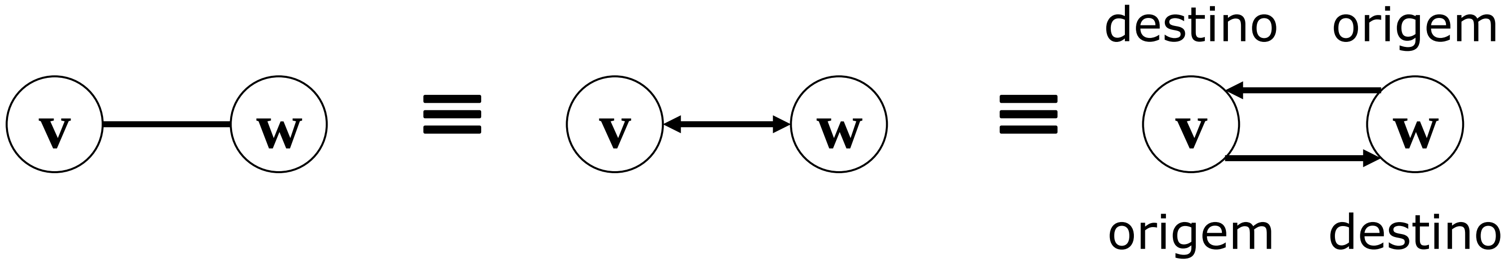
Ramos

Existem dois tipos de ramos, segundo sua orientação:

- Direcionados, tais que (v,w) ≠ (w,v)

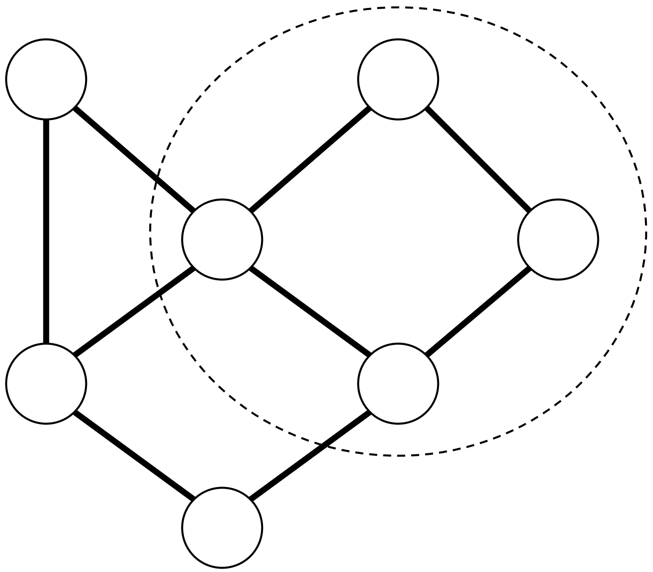


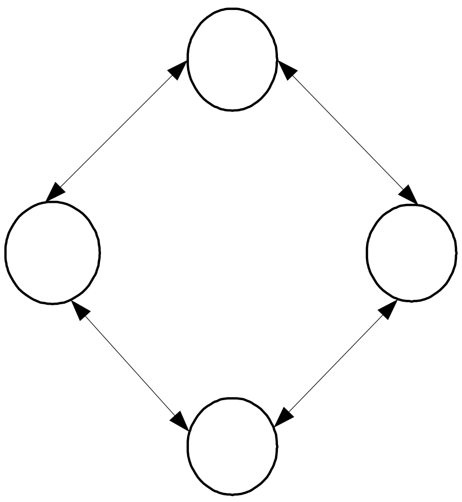
- Não-direcionados, tais que (v,w) = (w,v)

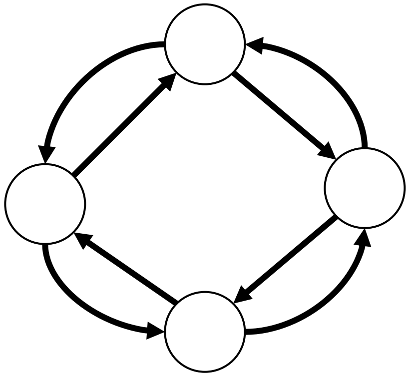


Grafo Não-Dirigido

Todos os ramos são bi-direcionais, ou seja, não importa o sentido escolhido, (v,w) = (w,v):



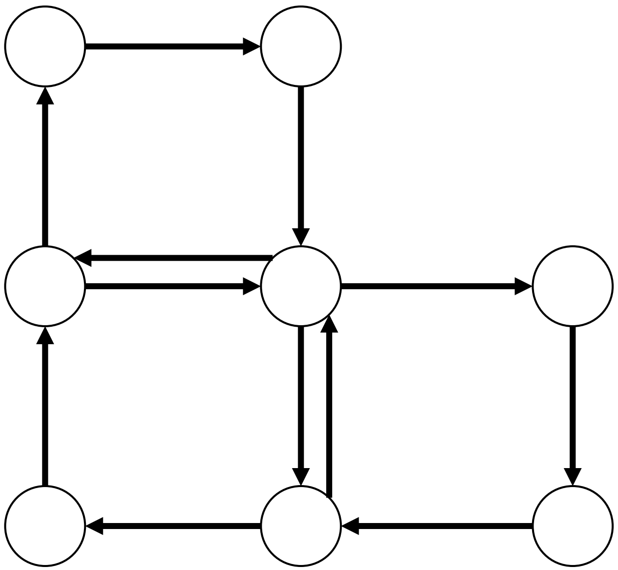




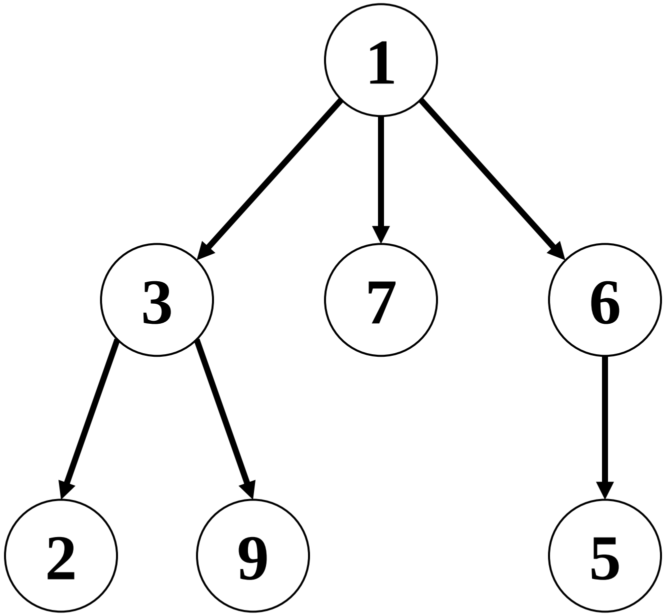
Grafo Dirigido ou Digrafo

Seja G = (V, E) um grafo em que os vértices V = {v_1, v_2, ..., v_n} estão ordenados de v_1 a v_n, e ligados através de **ramos orientados** **-** G é um **grafo dirigido**, ou **digrafo**

Cada ramo tem um sentido único, partindo do vértice de **origem** e chegando ao vértice de **destino**



Ex: Ruas de uma cidade



Ex: Árvore genealógica

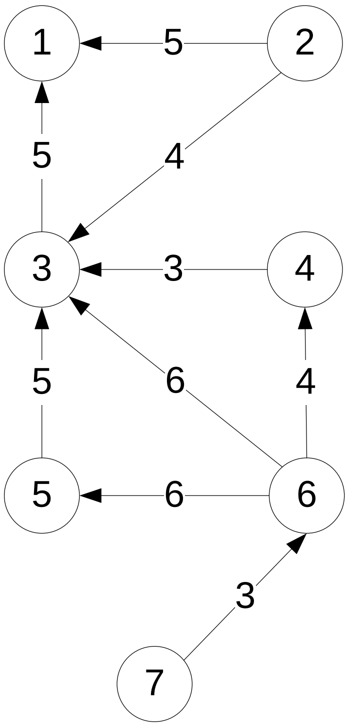
(ramos pai-filho)

Grafo Valorizado

É o grafo que possui valores nos vértices e nos ramos

Na prática estes valores podem representar:

- Custos, distâncias, capacidades, e/ou limitações e procuras;
- Tempo (trânsito, permanência, etc)
- Fiabilidade de transmissão
- Probabilidade de ocorrer falhas
- Capacidade de carga
- Outros



Caminho ou Percurso

- **Caminho** de um vértice v_1 para o vértice v_n é uma sequência de ramos $<v_1,v_2>, \ldots, <v_{n-1},v_n>$ tais que $(v_i, v_{i+1}) \in E,\ 1 \le i \le n$
- **Caminho Elementar** todos os vértices por onde passa são distintos
- **Caminho Simples** todos os ramos que o constituem são distintos
- **Comprimento** de um percurso:
  - num grafo valorizado é a soma dos custos de percorrer cada ramo
  - num grafo não valorizado é igual ao número de ramos que o compõem
- **Ciclo** caminho de comprimento ≥ 1 com $v_1 = v_n$, este é chamado **Circuito** se o grafo for orientado
- **Anel** – ramo $v, v \Rightarrow (v, v) \in E$

Relação de Adjacência e de Incidência

Grafo Não orientado

- Dois **vértices** v e w $\in$ V de um grafo G = (V,E) são **adjacentes** se existe um ramo (v,w) $\in$ E
- **Vizinhança** de um vértice v é o conjunto de vértices adjacentes de v
- Dois **ramos** são **adjacentes** se possuem uma extremidade (vértice) comum
- Um **ramo é incidente** a um vértice, num grafo não orientado, se este vértice é um dos seus extremos. Os ramos (u,v) e (v,w) são incidentes ao vértice v
- **Grau** de um vértice v num **grafo não orientado,** é o número de ramos adjacentes a v
- Num grafo não orientado com ramo (v,w) e, logo (w,v) w é **adjacente** a v e v é **adjacente** a w

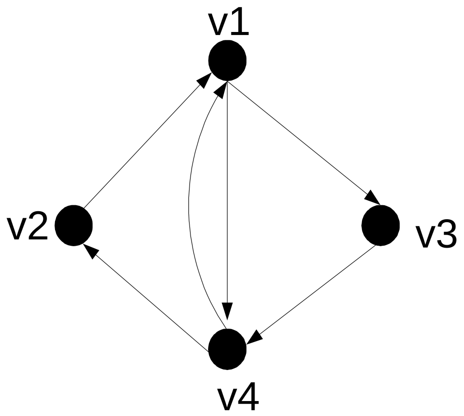
Relação de Adjacência e de Incidência

Grafo Orientado

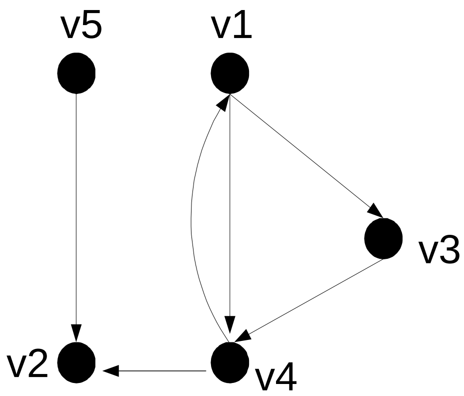
- Num grafo orientado, **ramo incidente** num vértice v, é qualquer ramo para o qual v é vértice destino
- Vértice w é **adjacente** a um vértice v se e só se $(v,w) \in E$
- Vértice $v \in V$ é **sucessor** do vértice $w \in V$ se $\exists_{(w,v) \in E}$
- Vértice $v \in V$ é **antecessor** do vértice $w \in V$ se $\exists_{(v,w) \in E}$
- **Grau de Saída** de um vértice v é o número de ramos de que v é vértice origem
- **Grau de Entrada** de um vértice v é o número de ramos de que v é vértice destino
- **Grau de um vértice** v = grau de entrada + grau de saída

Conectividade

- um grafo não dirigido é **conexo** se para todo o par de vértices u e v $\in$ V, existe um caminho entre u e v
- digrafo com a mesma propriedade – **fortemente conexo**
- **digrafo conexo** – para qualquer vértice u e v $\in$ V existe caminho **u** $\rightarrow$ **v ou v** $\rightarrow$ **u**



Grafo fortemente conexo



Grafo Desconexo

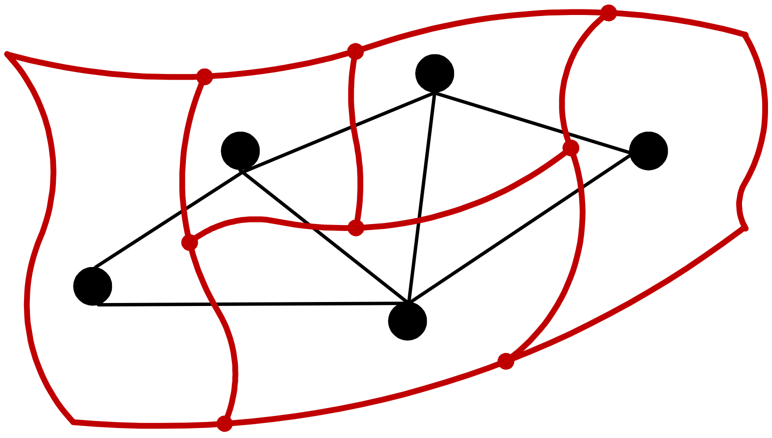
Densidade

- **grafo completo** – existe um ramo entre qualquer par de nós
- grafo denso – $|E| = O(V^2)$
- grafo esparso – $|E| = O(V)$

Ramos Paralelos são ramos que ligam os mesmos nós

Multigrafo é um grafo que contem ramos paralelos, caso contrário é um **Grafo Simples**

Grafo Planar é possível dispor os seus vértices num plano de forma a que não haja cruzamento de ramos

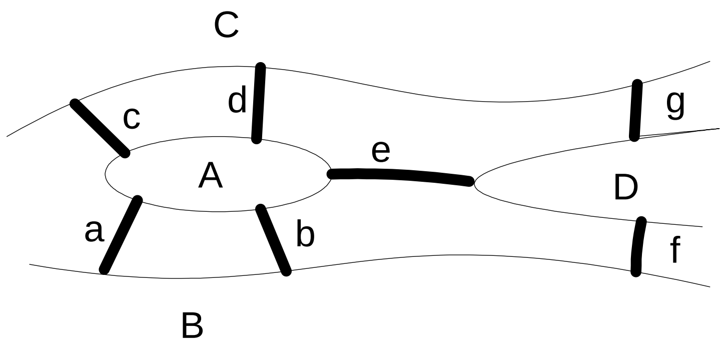


Pode-se fazer uma associação entre um grafo planar e um mapa

- Regiões ⇔ vértices
- Fronteiras ⇔ ramos

Ciclo/Grafo Euleriano

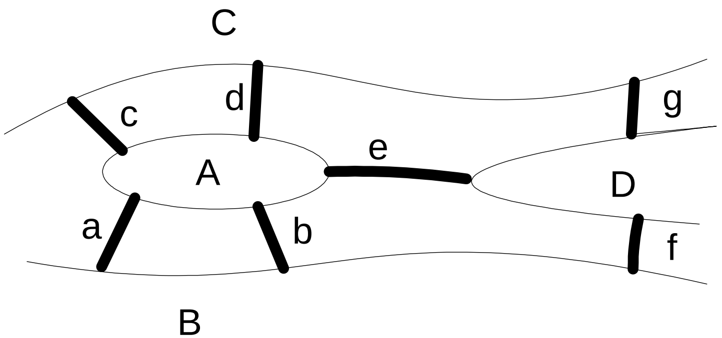
Circuito Euleriano é um circuito que passa exactamente uma só vez por todos os ramos de um grafo conexo

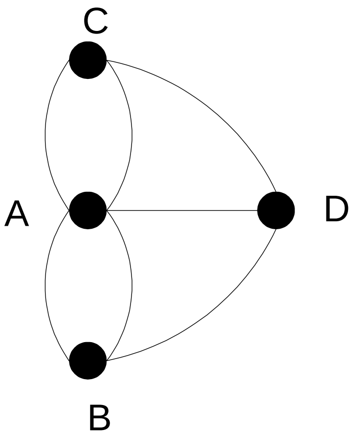


História:

Problema das pontes de Koenigsberg (hoje Kaliningrado): determinar se a partir de alguma área de terra (A, B, C, D) é possível atravessar todas as pontes (a, b, c, d, e, f, g) exactamente uma só vez, e retornar ao ponto de origem

Ciclo/Grafo Euleriano





Euler provou que um circuito assim é possível se e somente se o grafo for conexo e todos os seus vértices tiverem grau par

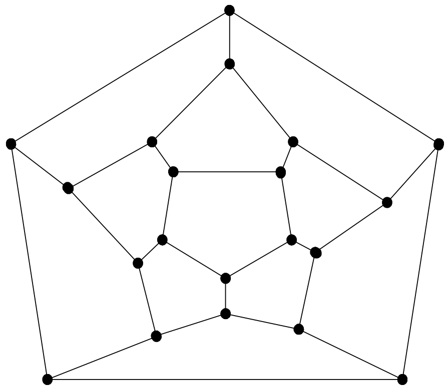
Grafo Euleriano é um grafo conexo cujos vértices têm grau par e logo possui um circuito Euleriano

Este problema foi solucionado por Euler em 1736 e marca o início da Teoria dos Grafos

Circuito Hamiltoniano

Circuito Hamiltoniano é um circuito que passa exactamente uma e uma só vez por todos os vértices de um grafo não orientado e conexo

Este nome foi dado a tais grafos após Hamilton ter descrito um jogo matemático no dodecaedro, no qual um jogador coloca 5 marcas em 5 vértices consecutivos e o outro jogador deve completar o caminho, até então criado, formando um ciclo gerador, ou seja, um que contenha todos os vértices do dodecaedro



O **problema do caixeiro viajante** que pretende visitar um certo número de cidades e voltar ao ponto de partida é resolvido com um **ciclo hamiltoniano de comprimento mínimo**

Grafos

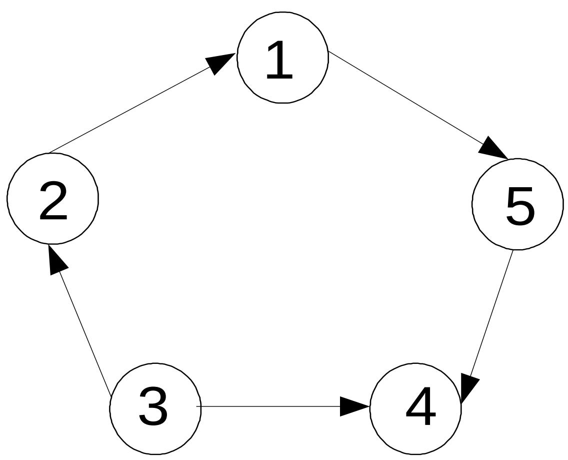
Representação Matriz de Adjacências

Representação Matriz de Adjacências

Seja G = (V, E) um grafo em que os vértices V = $\{v_1, v_2, \ldots. v_n\}$ estão ordenados de v_1 a v_n

A **matriz de Adjacências** é uma matriz quadrada (n x n) cujos elementos m_{ij} são:

$$M = \begin{cases} 1 & \text{se existe ramo entre i e j} \\ 0 & \text{se não existe ramo entre i e j} \end{cases}$$

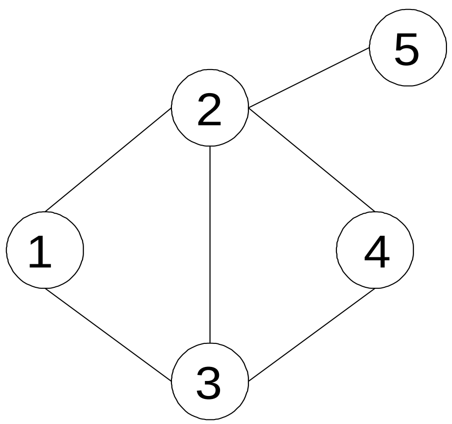


M =

	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	0
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

Representação Matriz de Adjacências

- Quando o grafo é não-orientado existe uma simetria em relação à diagonal principal da matriz
- Como (u,v) e (v,u) representam a mesma aresta num grafo não-orientado, a matriz de adjacências será a sua própria transposta: $M = M^T$



M =

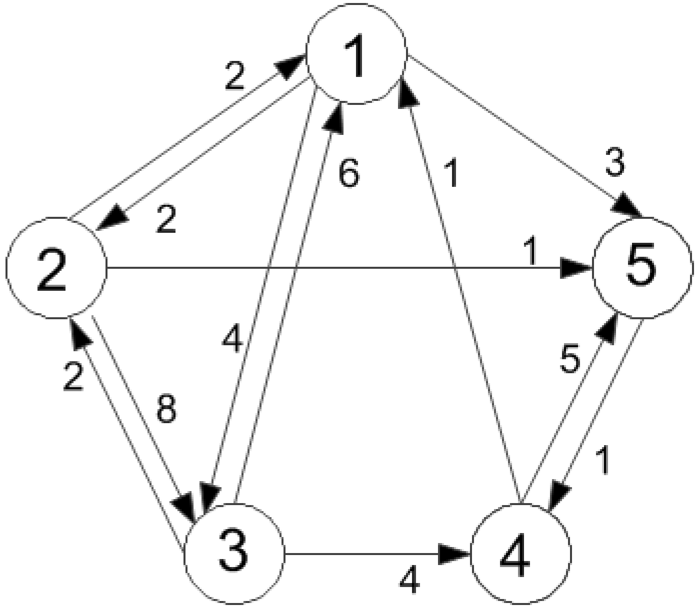
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	0
5	0	1	0	0	0

- Independentemente do número de ramos, o espaço de memória necessário para guardar a matriz → $O(V^2)$
- A matriz de adjacências é vantajosa na representação de grafos densos, nos quais $|E| \approx |V|^2$, ou quando se tem grafos razoavelmente pequenos

Representação Matriz de Pesos

A **matriz de Pesos** é uma matriz quadrada (n x n) cujos elementos m_{ij} são:

$$W = \begin{cases} m_{ij} = p_{ij} & \text{se existe ligação entre i e j com peso } p_{ij} \\ 0 & \text{se } i = j \\ m_{ij} = \infty & \text{se não existe ligação entre i e j com peso } p_{ij} \end{cases}$$



$$W = \begin{vmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 8 & \infty & 1 \\ 6 & 2 & 0 & 4 & \infty \\ 1 & \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

Algoritmo de Floyd-Warshall

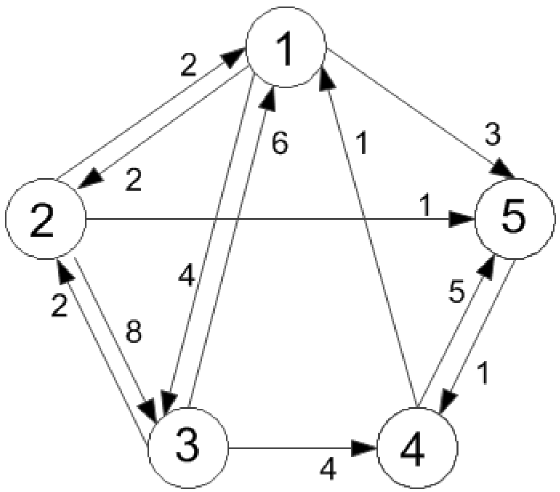
- Este algoritmo começa por determinar o caminho mais curto entre todos os vértices não utilizando nenhum vértice intermédio – **matriz de pesos**
- Em seguida calcula o caminho mais curto usando como vértice intermédio o vértice um
- Na iteração seguinte considera o vértice um ou dois como intermédios
- Em seguida os vértices um, dois, três e assim sucessivamente... até usar todos os vértices

Fórmula para atualização dos vértices em cada iteração:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & se\ k = 0 \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}) & \mathrm{se\ k} \geq 1 \end{cases}$$

Aplicação do Algoritmo de Floyd-Warshall

A matriz D^0 é igual à matriz de pesos(W)



$$D^0 = \begin{vmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & \mathbf{8} & \infty & 1 \\ 6 & 2 & 0 & 4 & \boldsymbol{\infty} \\ 1 & \boldsymbol{\infty} & \boldsymbol{\infty} & 0 & \mathbf{5} \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

Em seguida calcula-se D^1

$$D^1 = \begin{vmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & \mathbf{6} & \infty & 1 \\ 6 & 2 & 0 & 4 & \mathbf{9} \\ 1 & \mathbf{3} & \mathbf{5} & 0 & \mathbf{4} \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

Calculo de $\mathbf{d_{2,3}^{(1)}}$

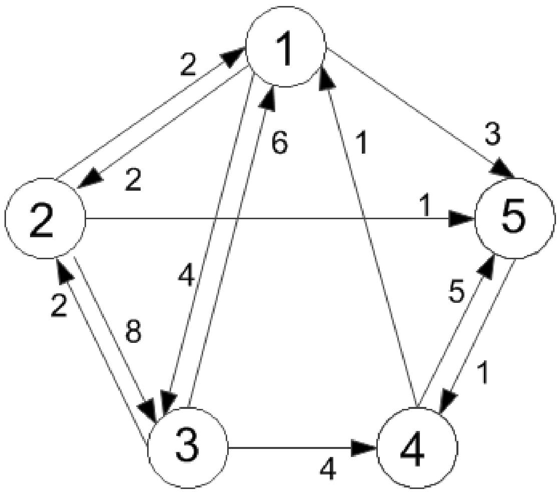
$$d_{2,3}^{(1)} = \min(d_{2,3}^{0}, d_{2,1}^{0} + d_{1,3}^{0}) = \min(8, 2+4) = 6$$

Calculo de $\mathbf{d_{4,2}^{(1)}}$

$$d_{4,2}^{(1)} = \min(d_{4,2}^{0}, d_{4,1}^{0} + d_{1,2}^{0}) = \min(\infty, 1+2) = 3$$

Aplicação do Algoritmo de Floyd-Warshall

Depois calcula-se a matriz D^2 partindo da matriz D^1



$$D^1 = \begin{vmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \infty & 1 \\ \mathbf{6} & 2 & 0 & 4 & \mathbf{9} \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

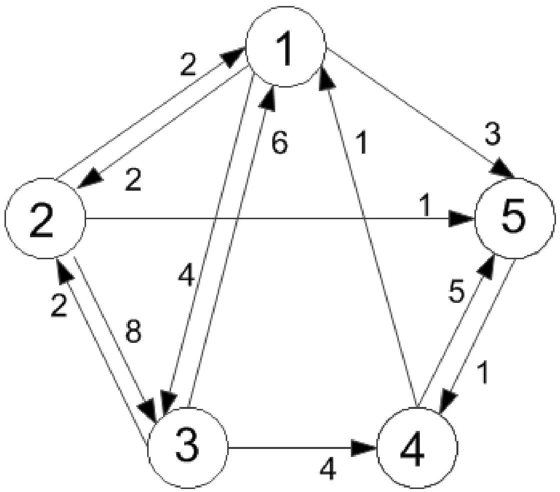
$$D^2 = \begin{vmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \infty & 1 \\ \mathbf{4} & 2 & 0 & 4 & \mathbf{3} \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

Calculo de $d_{3,1}^{(2)}$

$$d_{3,1}^{(2)} = \min\,(d_{3,1}^{1}, d_{3,2}^{1} + d_{2,1}^{1}) = \min\,(6, 2+2) = 4$$

Aplicação do Algoritmo de Floyd-Warshall

Calculo da matriz D^3 a partir da matriz D^2



$$D^2 = \begin{vmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \boldsymbol{\infty} & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

$$D^3 = \begin{vmatrix} 0 & 2 & 4 & 8 & 3 \\ 2 & 0 & 6 & \mathbf{10} & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{vmatrix}$$

Calculo de $d_{2,4}(3)$

$$d_{2,4}^{(3)} = \min(d_{2,4}^2, d_{2,3}^2 + d_{3,4}^2) = \min(\infty, 6+4) = 10$$

Aplicação do Algoritmo de Floyd-Warshall

Seguindo o mesmo método calculam-se as matrizes D^4 e D^5

$$D^4 = \begin{vmatrix} 0 & 2 & 4 & 8 & 3 \\ 2 & 0 & 6 & \mathbf{10} & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ 2 & 4 & 6 & 1 & 0 \end{vmatrix} \qquad D^5 = \begin{vmatrix} 0 & 2 & 4 & 4 & 3 \\ 2 & 0 & 6 & \mathbf{2} & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ 2 & 4 & 6 & 1 & 0 \end{vmatrix}$$

Obtém-se assim a matriz de pesos dos caminhos mínimos entre todos os pares de vértices

Fecho Transitivo

Por vezes interessa saber se existe ou não caminho entre dois nós e não o caminho mínimo

O **fecho transitivo** de um grafo é o grafo com os mesmos vértices que o grafo original e com um arco entre os pares de vértices que no grafo original têm um caminho a uni-los

O fecho transitivo de um grafo é calculado usando o algoritmo Floyd-Warshall, mas substitui-se:

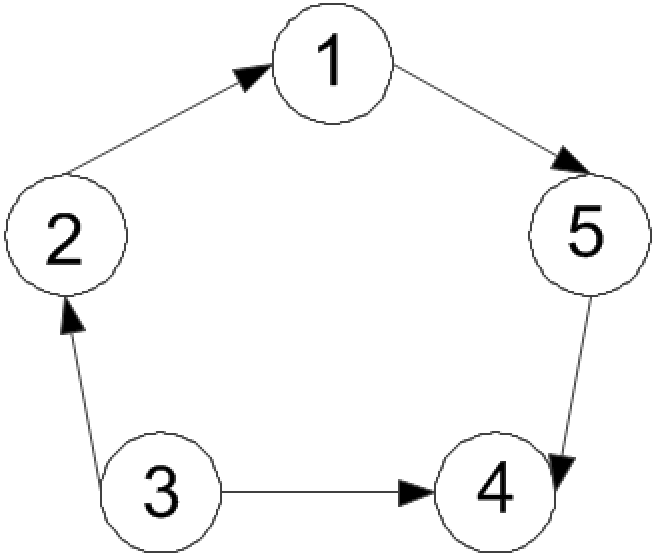
- a matriz de pesos pela matriz de adjacências
- as operações algébricas sobre as matrizes por operações lógicas (o operador *min* por $\vee$ e o operador $+$ por $\wedge$)

A fórmula para atualização dos vértices em cada iteração:

$$t_{i,j}^{(k)} = \begin{cases} w_{i,j} & se\ k = 0 \\ t_{i,j}^{(k-1)} \vee (t_{i,k}^{(k-1)} \wedge d_{k,j}^{(k-1)}) & se\ k \geq 1 \end{cases}$$

Exemplo de Aplicação do Fecho Transitivo

A matriz T^0 é igual à matriz de adjacências - matriz de caminhos de comprimento 1



T^0 =

	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	0
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

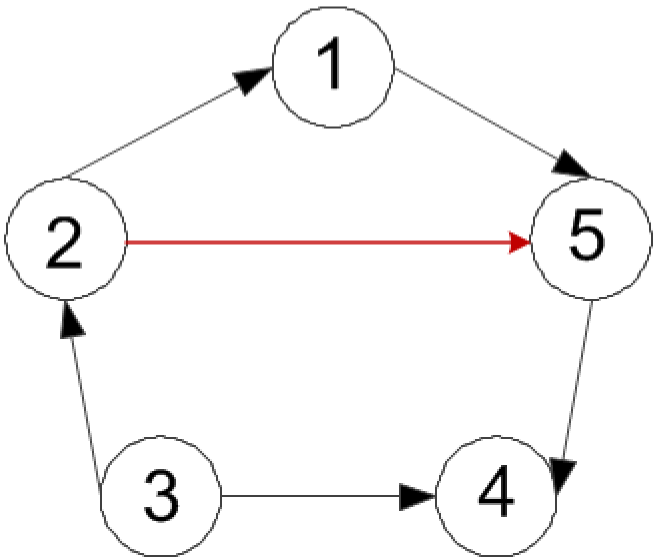
Calculo de $d_{2,5}(1)$

$$d_{2,5}^{(1)} = d_{2,5}^{0} \vee (d_{2,1}^{0} \wedge d_{1,5}^{0}) = 0 \vee (1 \wedge 1) = 1$$

Exemplo de Aplicação do Fecho Transitivo

T^1 =

	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	**1**
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0



Exemplo de Aplicação do Fecho Transitivo

$T^1 =$

	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	1
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

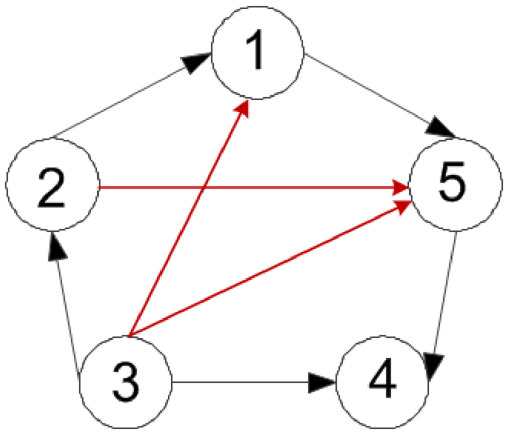
$T^2 =$

	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	1
3	**1**	1	0	1	**1**
4	0	0	0	0	0
5	0	0	0	1	0

Calculo de $d_{3,1}(2)$ e $d_{3,5}(2)$

$$d_{3,1}^{(2)} = d_{3,1}^{1} \vee (d_{3,2}^{1} \wedge d_{2,1}^{1}) = 0 \vee (1 \wedge 1) = 1$$

$$d_{3,5}^{(2)} = d_{3,5}^{1} \vee (d_{3,2}^{1} \wedge d_{2,5}^{1}) = 0 \vee (1 \wedge 1) = 1$$



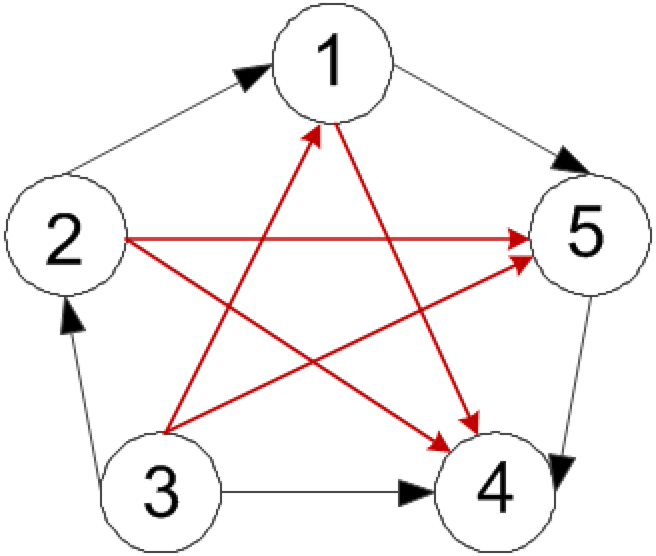
- O acréscimo dos ramos que partem do vértice 3 não conduz a novos ramos, pelo que $T^2 = T^3$
- Idem para o vértice 4 $\Rightarrow$ $T^3 = T^4$

Exemplo de Aplicação do Fecho Transitivo

- A inserção do ramo que parte do vértice 5 permite acrescentar os ramos (1,4) e (2,4)

T^5 =

	1	2	3	4	5
1	0	0	0	1	1
2	1	0	0	1	1
3	1	1	0	1	1
4	0	0	0	0	0
5	0	0	0	1	0



A matriz final - Matriz dos Caminhos - cada elemento ou é um, ou é zero, significando respetivamente se existe ou não existe caminho de comprimento menor ou igual a n, entre cada par de vértices

Pseudo-Código Algoritmo Floyd-Warshall

```
Para k = 1 até n
        Para  i = 1 até n
                Para  j = 1 até n
                     se (P[i, j] ≠ 1 ∧ (P [i, k]  = 1 ∧ P [k, j] = 1) )
                        P[i, j] = 1
                Fpara
        Fpara
Fpara
```

O algoritmo de Floyd-Warshall é um algoritmo simples

Complexidade Temporal → O ($|V|^3$)

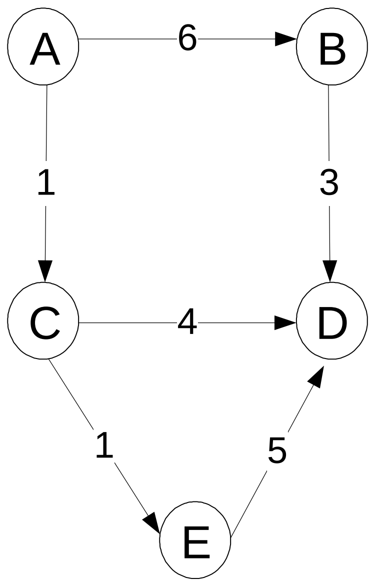
Grafos

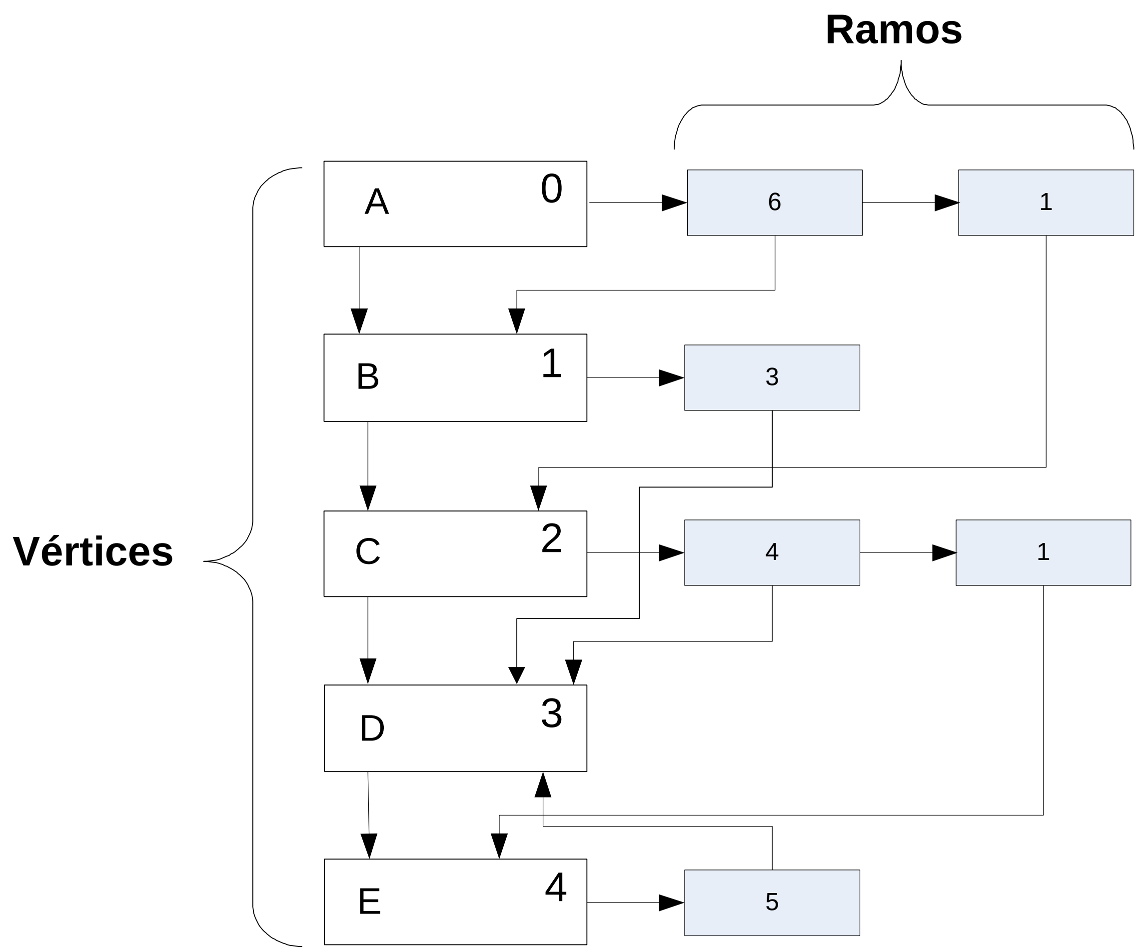
Representação Lista de Adjacências

Representação Lista de Adjacências

- Lista de vértices
- Cada vértice tem uma lista dos ramos que partem dele

Exemplo:





Representação Lista de Adjacências

- Vantajosa para representar **grafos esparsos**, nos quais $|E|$ é bem menor do que $|V|^2$

- Ideal para determinar os sucessores imediatos de um vértice
  - E os antecessores de um vértice ?

- Este tipo de representação não é vantajoso para verificar os antecessores de um vértice
  - necessário percorrer a lista de adjacência completa

Classe Vértice

```
template<class TV, class TE>
class graphVertex
{
  private:
    int vKey;  //Vertex unique key in the graph
    TV vContent;  // Vertex content
    list < graphEdge<TV,TE> > eList;  //Vertex list of edges (each edge
                                        contains its adjacent vertices)
public:
    graphVertex();
    graphVertex(const TV& vContent, int vKey=-1);
    graphVertex(const graphVertex<TV,TE>& v);
    ~graphVertex();

    TV getVContent() const ;
    void setVContent(const TV& vContent);

    int getVKey() const ;
    void setVKey(int vKey) ;
```

Classe Vértice

```
template<class TV, class TE>
class graphVertex
{

    bool getEdgeContentByVDestination(TE& eContent,
        typename list < graphVertex <TV,TE> >::iterator vDestination);

    typename list < graphEdge <TV,TE> >::iterator getAdjacenciesBegin() ;
    typename list < graphEdge <TV,TE> >::iterator getAdjacenciesEnd() ;

    int getAdjacenciesSize() const ;

    void addAdjacency(const TE& econtent,
            typename list < graphVertex <TV,TE> >::iterator vDestination );

    bool operator == (const graphVertex <TV,TE> &v) const;
    void write(ostream &o) const;
};
```

Classe Ramo

```
template<class TV, class TE>
class graphEdge
{
  private:
    TE eContent;  // Edge content
    typename list < graphVertex <TV,TE> >::iterator vDestination;

 public:
    graphEdge();
    graphEdge(const TE& eContent, typename
                        list < graphVertex <TV,TE> >::iterator vDestination);
    graphEdge(const graphEdge<TV,TE>& e);
    ~graphEdge();

    TE getEContent() const ;
    void setEContent(const TE& eContent);
```

Classe Ramo

```
template<class TV,class TE>
class graphEdge
{

   typename list < graphVertex <TV,TE> >::iterator getVDestination();
   void setVDestination(typename list <graphVertex < TV,TE > >::iterator
                        vDestination) ;

   void write(ostream &o) const;
};
```

Classe Grafo

```
template<class TV, class TE>
class graphStl
{
 private:
        TE infinite; // Dijkstra's Infinite
        int keys;

 protected:
    list < graphVertex <TV,TE> > vlist;  //Vertice list

    TE getInfinite() const;


    virtual bool compareVertices(TV const &vContent1, TV const &vContent2);

    bool getVertexIteratorByContent(typename list < graphVertex <TV,TE> >::
                              iterator &vIterator, const TV &vContent );
    bool getVertexIteratorByKey(typename list < graphVertex <TV,TE> >::
                                    iterator &itv, int vKey );
```

Classe Grafo

```
template<class TV, class TE>
class graphStl
{
  public:
      graphStl();
      bool getVertexContentBySubContent(TV &vContent, const TV &vSubContent);
      bool getVertexContentByKey(TV &vContent, int vKey );
      bool getVertexKeyByContent(int &vKey, const TV &vContent);
      bool getEdgeByVertexContents(TE &eContent, const TV &vOrigin,
                                   const TV &vDestination);
      bool getEdgeByVertexKeys(TE &eContent, int vKeyOrigin,
                               int vKeyDestination);

      int getEntranceDegree(const TV& vContent);
      int getExitDegree(const TV& vContent);
```

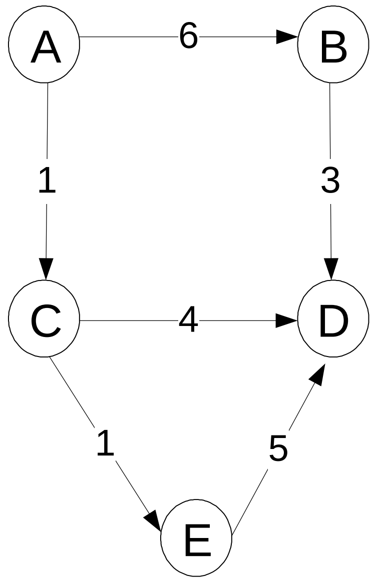
Classe Grafo

```
template<class TV, class TE>
class graphStl
{
     bool addGraphVertex(const TV& vContent);
     bool addGraphEdge(const TE& eContent, const TV& vOrigin,
                       const TV& vDestination);

     virtual void write(ostream &o);
};
```

Construir o grafo exemplo

Exemplo:




```
int main()
{
   graphStlPath <char, int> g ;

   g.addGraphEdge(6,'A','B');
   g.addGraphEdge(1,'A','C');
   g.addGraphEdge(3,'B','D');
   g.addGraphEdge(4,'C','D');
   g.addGraphEdge(1,'C','E');
   g.addGraphEdge(5,'E','D');

   cout << g ;

   return 0 ;
}
```

Grafos

Visitas em Largura e Profundidade

Território de um Vértice

- Um vértice v pode ser alcançado a partir de um vértice r se existe um caminho de r a v

- O **território** de um vértice v é o conjunto de todos os vértices alcançáveis a partir de v

Algoritmos de Visita em Largura e em Profundidade

- Os algoritmos de visita (em Largura e em Profundidade) dão os nós alcançáveis a partir de um nó origem
- A ordem de apresentação dos vértices é irrelevante
  - ⤷ esta é definida pela ordem dos vértices das listas de adjacências

Visita em Largura (Breadth First Search)

Passos:

1. a partir de um dado vértice origem, ver quais os adjacentes
2. caso ainda não tenham sido visitados colocam-se numa fila auxiliar
3. O próximo nó origem é o que está no início da fila sendo retirado desta
4. são inseridos na fila os nós adjacentes (do nó retirado da fila) caso ainda não tenham sido visitados

Visita em Largura → **Para cada vértice percorrem-se todos os seus ramos**

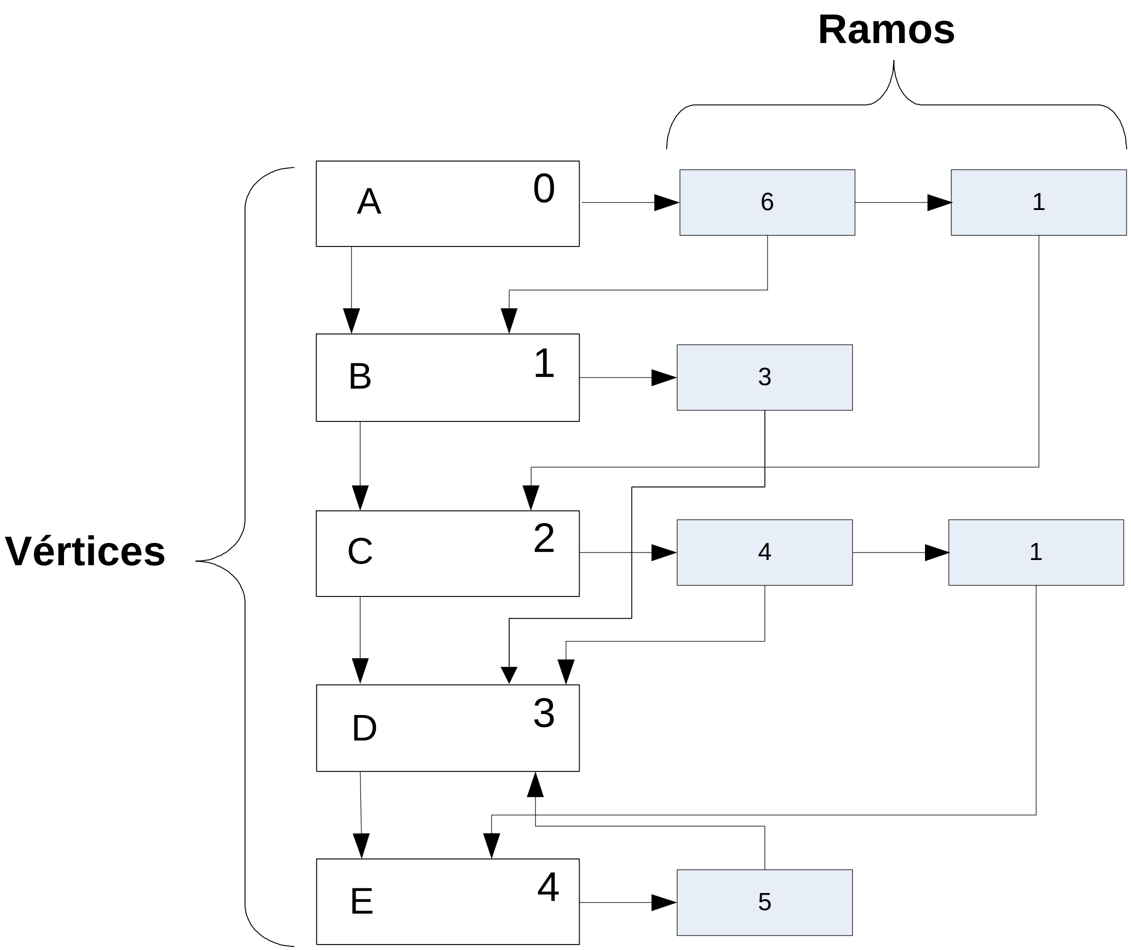
A **Visita em Largura** é uma busca em amplitude, processa os vértices por níveis, começando pelos vértices mais próximos do vértice inicial

↪ Similar à **visita por níveis** nas árvores

Visita em Largura (Breadth First Visit)

```
BFS (vértice, qbfv)
  marca vértice visitado
  insere_qbfv(conteúdo vértice)          //qbfv fila c/ vértices da visita
  insere_fila(vértice)
  Enqto fila não vazia
      retira vértice fila
      Enqto Vértices Adjacentes
         encontra vértice_adjacente
         Se vértice_adjacente não visitado
            marca vértice_adjacente visitado
            insere_qbfv(conteúdo vértice_adjacente)
            insere_fila(vértice_adjacente)
         Fse
      Fenqto //avança na lista de vértices adjacentes
  FEnqto //avança para o próximo vértice da fila
Fim BFS
```

Visita em Largura



Vértice Origem 1 → A B C D E

vector [1,0,0,0,0]
qbfv: A
fila: A
retira-fila (A)
vector [1,1,0,0,0]
qbfv: A, B
fila: B
vector [1,1,1,0,0]
qbfv:A,B, C
fila: B, C
retira-fila (B)
vector [1,1,1,1,0]
qbfv:A,B,C,D
fila: C, D
retira-fila (C)
vector [1,1,1,1,0]
vector [1,1,1,1,1]
qbfv:A,B,C, D, E
fila: D, E
retira-fila (D)
retira-fila (E)
vector [1,1,1,1,1]

Análise de Complexidade

- As operações de colocar e tirar da fila de espera têm complexidade O(1) - estas são realizadas no máximo uma vez para cada vértice
  - a complexidade total com as operações de fila é O(V)

- A complexidade do varrimento total das listas dos ramos é a soma do comprimento total de cada lista, que é igual a O(E)

- Daí que o limite superior para o número de operações realizadas na Visita em Largura seja $O(V \times E)$

- Em teoria de grafos o tamanho do grafo é dado por $|V| \times |E|$

  Assim como O(n) indica execução em tempo linear para algoritmos com entrada de tamanho n, a complexidade $O(V \times E)$ indica também **complexidade linear** para Grafos

Visita em Profundidade (Depth First Search)

Passos

1. a partir de um nó origem avança-se para um seu nó adjacente, caso não tenha sido visitado
2. a partir deste último, que passa a ser origem, passa-se para o seu adjacente não visitado, e a partir deste, para o seu adjacente, e assim sucessivamente percorrendo um caminho
3. quando não puder avançar mais, regressa ao último vértice (backtraking) que ainda tem adjacentes não visitados
4. avança para o próximo adjacente e assim sucessivamente

Visita em Profundidade → Para cada vértice percorrem-se todos os seus adjacentes

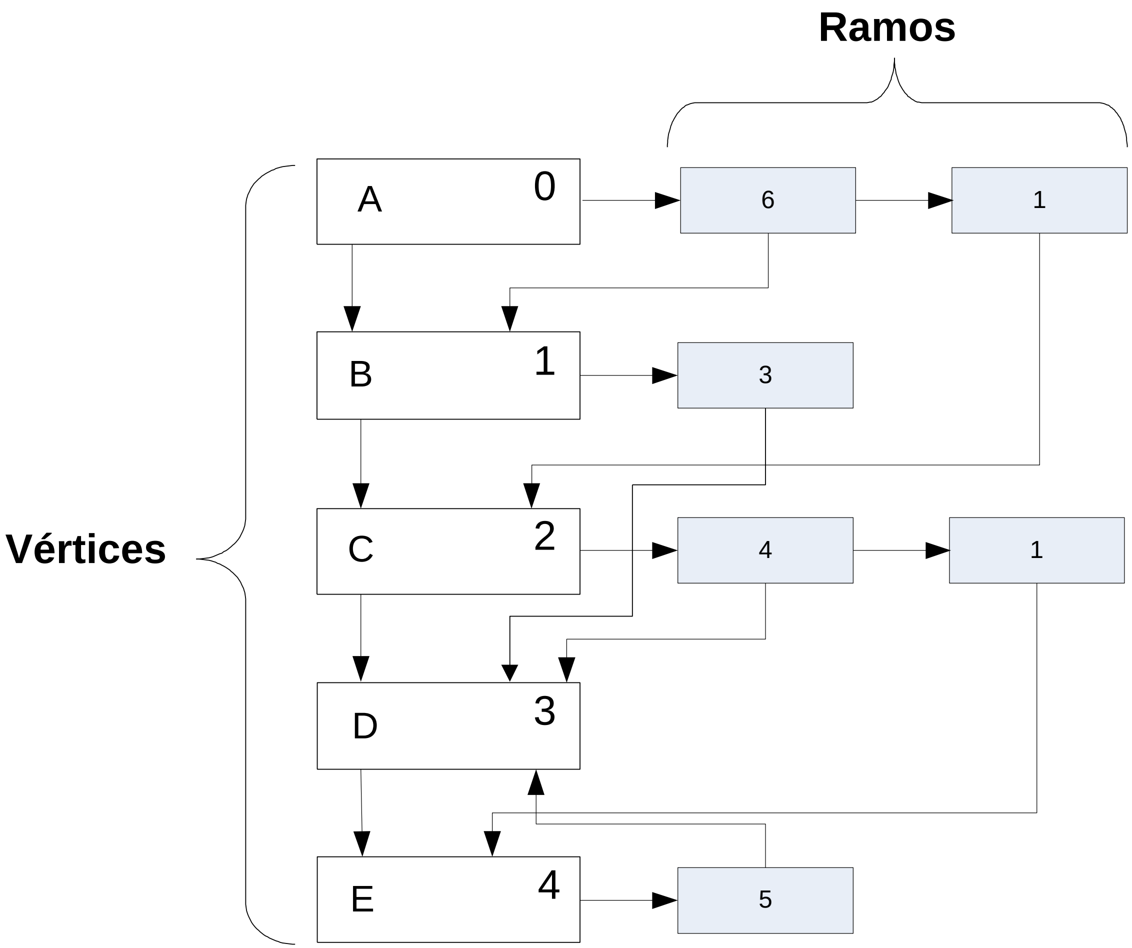
A **Visita em Profundidade** é uma busca vertical

- ⤷ Similar à **visita em pré-ordem** nas árvores

Visita em Profundidade (Depth First Search)

```
DFS (vértice, vtvisits, qdps)
   marca vértice visitado
   qdps.push(conteúdo vértice)              //qdfs fila c/ vértices da visita
   Enqto Vértices Adjacentes
        Encontra vértice_adjacente
        Se vértice_adjacente não visitado
          DFS (vértice_adjacente, vtvisits, qdps)
        Fse
   FEnqto //avança na lista de vértices adjacentes
Fim DFS
```

Visita em Profundidade



Vértice Origem A → A B D C E

DFS (A, vtvisits, qdps)
vtvisits [1,0,0,0,0]
qdps: A
DFS (B, vtvisits, qdps)
vectvisits [1,1,0,0,0]
qdps: A, B
DFS (D, vtvisits, qdps)
vectvisits[1,1,0,1,0]
qdps: A,B, D
DFS (C, vtvisits, qdps)
vectvisits [1,1,1,1,0]
qdps: A,B,D, C
DFS (D, vtvisits, qdps)
vectvisits [1,1,1,1,0]
DFS (E, vtvisits, qdps)
vectvisits [1,1,1,1,1]
qdps: A,B,D,C, E

Análise de Complexidade

Visita em Profundidade

O ciclo Enq^{to} Vértices Adjacentes

é executado |Adj(V)| vezes $\rightarrow$ grau de saída do vértice V, para cada vértice do grafo

Este ciclo para todos os vértices do grafo demora $\rightarrow$ O(V$\times$E)

↪ **Complexidade Linear**

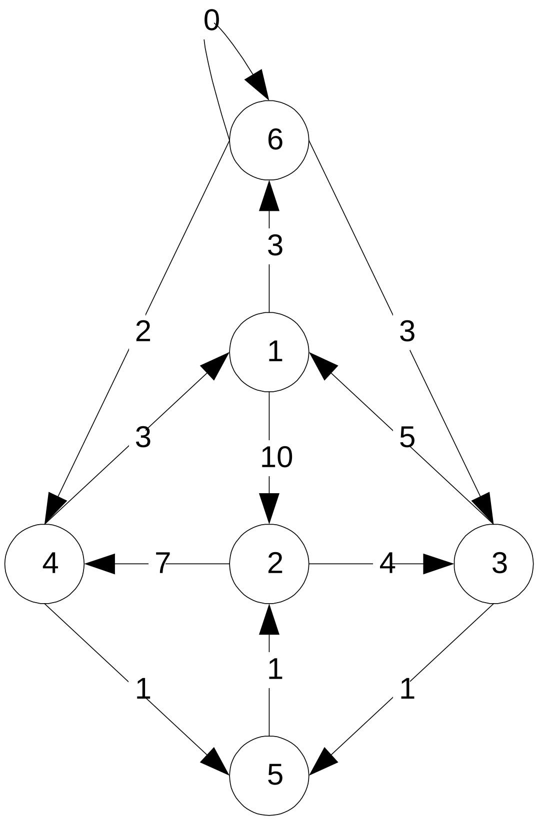
Grafos

Caminhos, Caminho Mínimo

Caminhos

- Um **caminho** num grafo é uma sequência de vértices $(v_0, v_1, v_2, ..., v_k)$ em que para cada v_i existe um ramo que sai do vértice $v_{[i-1]}$ e entra no vértice $v_{[i]}$
- Um caminho pode passar duas ou mais vezes por um mesmo vértice ou por um mesmo arco
- **Caminho Simples** todos os ramos que o constituem são distintos
- **Caminho de Euler** – caminho simples que contém todos os ramos do grafo
- **Caminho Elementar** todos os vértices que o constituem são distintos
- **Caminho de Hamilton** – caminho elementar que contém todos os vértices do grafo

Todos os Caminhos Simples entre dois Vértices



Lista de Adjacências

1 → 2 , 6
2 → 3 , 4
3 → 1 , 5
4 → 1 , 5
5 → 2
6 → 3, 4, 6

Todos os caminhos entre os Vértices 1 - 5:

- 1, 2, 3, 5
- 1, 2, 4, 5
- 1, 6, 3, 5
- 1, 6, 4, 5

Contar Caminhos Simples entre dois Vértices

```
Conta-Caminhos (Vert_Inicial, Vert_Final, vector, cont)
  marca Vert_Inicial visitado
  Enqto Vertices_Adjacentes
     Se (Vert_Adjac == Vert_Final)
        cont++
     Senão
        Se (Vert_Adjac não visitado)
          Conta-Caminhos(Vert_Adjac,Vert_Final,vector,cont)
     Avança para o próximo Vert_Adjac
  FEnqto
  Marca Vert_Inicial não visitado;
Fim Conta-Caminhos
```

Como adaptar este algoritmo para apresentar todos os caminhos entre dois vértices ?

Todos os Caminhos Simples entre dois Vértices

```
Todos-Caminhos (VOrig, VDst, vector, cam, caminhos)
  marca VOrig visitado
  insere VOrig cam                       //cam: stack c/ o caminho atual
  Enqto Vertices_Adjacentes
     Se (VAdj == VDst)
        insere VAdj cam
        insere cam caminhos         //caminhos: fila c/ todos os caminhos
        retira VAdj cam
     Senão
        Se (VAdj não visitado)
          Todos-Caminhos(VAdj,VDst,vector,cam,caminhos)
     Avança para o próximo VAdj
  FEnqto
  Marca VOrig não visitado
  retira Vorig cam
  Fim Mostra-Caminhos
```

Caminhos Mais Curtos de Origem Única

Dado um grafo G = (V, E) dirigido, com uma função de **pesos positivos** W: E $\rightarrow$ R, e um vértice inicial v, encontrar todos os caminhos mais curtos entre esse vértice e qualquer outro do grafo

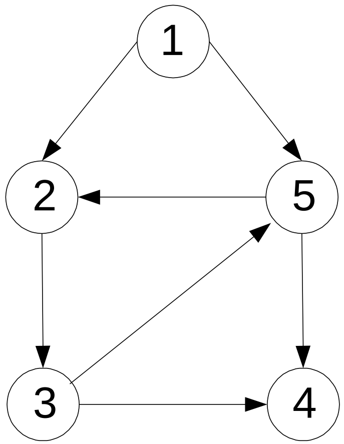
O **peso de um caminho** $(v_0, v_1, v_2, ..., v_k)$ é definido pela soma do peso de cada ramo $<v_{i-1}, v_i>$ i = 0,..., k que compõe o caminho

Para **grafos não pesados** (os ramos têm peso 1) o caminho mais curto é aquele que minimiza o números de ramos que compõem o caminho

⤷ é um caso particular de um grafo pesado (peso de cada ramo é unitário)

Caminhos Mais Curtos - Algoritmo não Pesado

Dado um grafo G = (V, E) dirigido, não pesado, e um vértice inicial v, encontrar todos os caminhos mais curtos entre esse vértice e qualquer outro do grafo



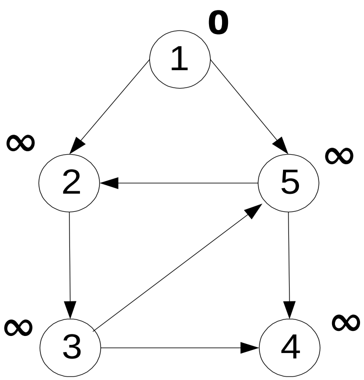
Caminhos mais curtos com origem no vértice 1 : **Peso**

- 1-2 1
- 1-5 1
- 1-2-3 2
- 1-5-4 2

Algoritmo

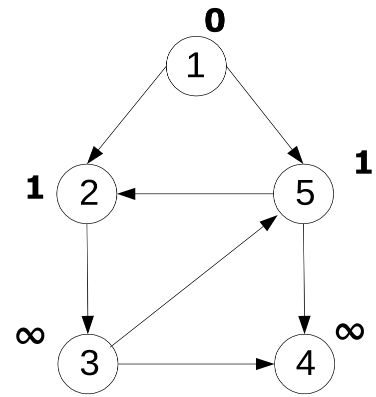
- começa-se por marcar o vértice inicial **v** com comprimento 0
- passa-se aos que lhe são adjacentes e marcam-se com mais 1 do valor do caminho do seu antecedente – **Visita em Largura**
- código usa uma tabela em que regista, para cada vértice v
  - a distância de cada vértice ao inicial (dist)
  - qual o antecessor no caminho mais curto (path)

Evolução do Algoritmo não Pesado



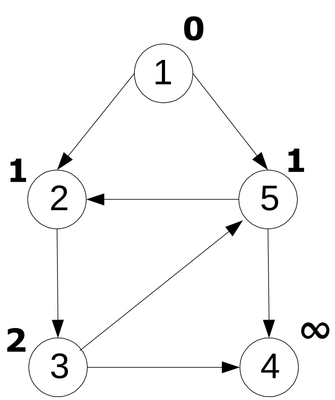
dist	0	∞	∞	∞	∞
	1	2	3	4	5
path	0	0	0	0	0
	1	2	3	4	5

queue		1			



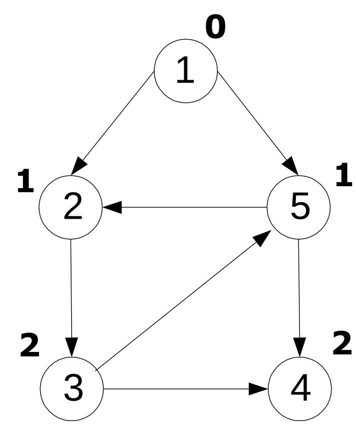
dist	0	1	∞	∞	1
	1	2	3	4	5
path	0	1	0	0	1
	1	2	3	4	5

queue		2	5		



dist	0	1	2	∞	1
	1	2	3	4	5
path	0	1	2	0	1
	1	2	3	4	5

queue		5	3		



dist	0	1	2	2	1
	1	2	3	4	5
path	0	1	2	5	1
	1	2	3	4	5

queue		3	4		

.... termina

queue					

Pseudo-Código Algoritmo Não-Pesado

```
CamMin_NãoPesado (vinicio)
   Para cada v ∈ V
      dist[v] = ∞
      path[v] = 0
   FPara
   dist[vinicio] = 0
   fila.insere (vinicio)
   Engto (fila não vazia)
        fila.retira(vorig)
        Para cada vdest ∈ Adj[vorig]
            Se (dist[vdest] = ∞)
              dist[vdest] = dist[vorig] + 1
              path[vdest] = vorig
              fila.insere(vdest)
            FSe
        FPara
   F_Engto
Fim CamMin_NãoPesado
```

Complexidade → $O(|V| \times |E|)$

Caminhos Mais Curtos - Algoritmo Pesado

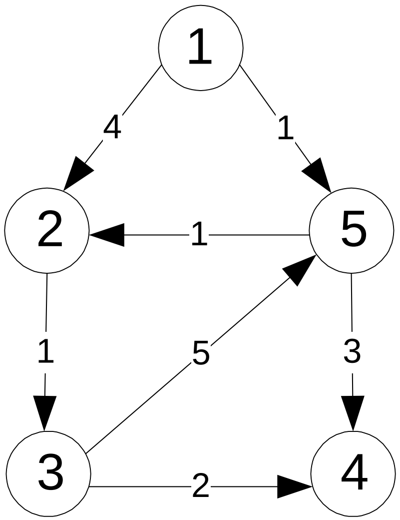
A solução é uma modificação da anterior:

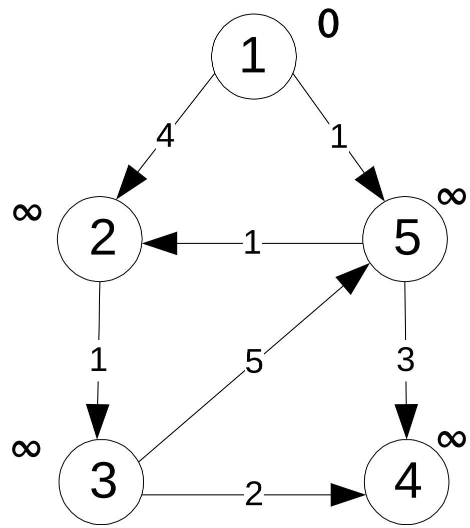
- cada vértice mantém uma distância ao inicial, obtida somando os pesos nos ramos
- quando se declara um vértice processado, exploram-se os seus adjacentes; se o caminho através deste nó é melhor que o já registado, modifica-se este
- **distância corrente em cada vértice**: a melhor usando apenas os vértices já processados
- **o ponto crucial**: escolher para declarar como processado o vértice que tiver o menor custo até ao momento
  - é o único cujo custo não pode diminuir
  - todas as melhorias de caminhos que usam este vértice são exploradas

Este é um exemplo de um **algoritmo ganancioso**: em cada passo faz o que melhora o ganho imediato

Restrição: só é válido se não existirem ramos com pesos negativos

Evolução do Algoritmo Pesado (Dijkstra)

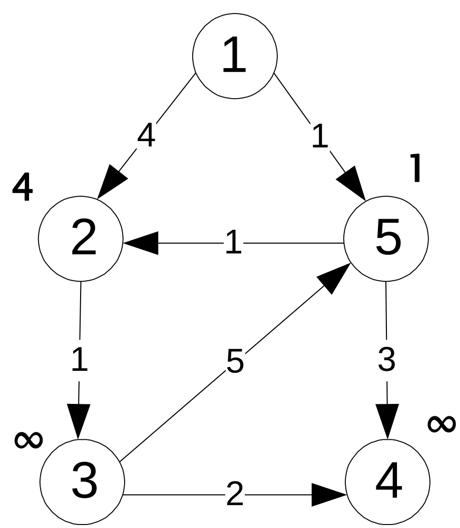




path	0	0	0	0	0
	1	2	3	4	5

dist	0	∞	∞	∞	∞
	1	2	3	4	5

proc	0	0	0	0	0
	1	2	3	4	5

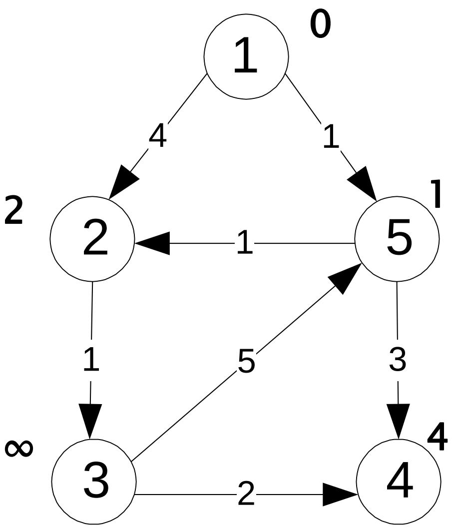


path	0	1	0	0	1
	1	2	3	4	5

dist	0	4	∞	∞	1
	1	2	3	4	5

proc	1	0	0	0	0
	1	2	3	4	5

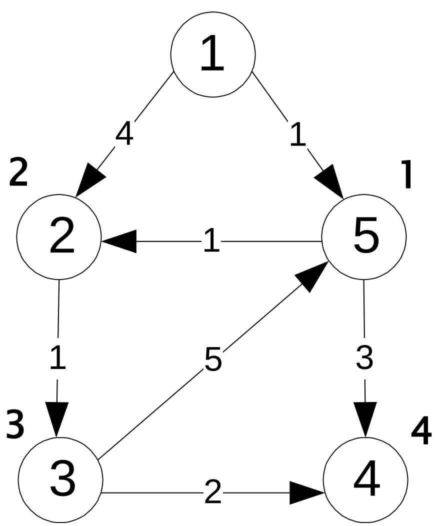
Evolução do Algoritmo Pesado (Dijkstra)



path	**0**	**5**	**0**	**5**	**1**
	1	**2**	**3**	**4**	**5**

dist	**0**	**2**	**∞**	**4**	**1**
	1	**2**	**3**	**4**	**5**

proc	**1**	**0**	**0**	**0**	**1**
	1	**2**	**3**	**4**	**5**



path	**0**	**5**	**2**	**5**	**1**
	1	**2**	**3**	**4**	**5**

dist	**0**	**2**	**3**	**4**	**1**
	1	**2**	**3**	**4**	**5**

proc	**1**	**1**	**0**	**0**	**1**
	1	**2**	**3**	**4**	**5**

...

path	**0**	**5**	**2**	**5**	**1**
	1	**2**	**3**	**4**	**5**

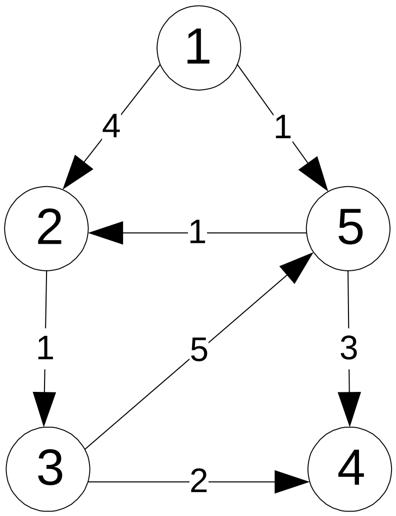
dist	**0**	**2**	**3**	**4**	**1**
	1	**2**	**3**	**4**	**5**

proc	**1**	**1**	**1**	**1**	**1**
	1	**2**	**3**	**4**	**5**

Pseudo-Código Algoritmo Pesado

```
CamMin_Pesado (vinicio, dist, path, process)
  Para cada v ∈ V
       dist[v] = ∞
       path[v] = -1
       process[v] = 0
  Fpara
  vorig = vinicio.key
  dist[vorig] = 0
  Engto vorig /= -1
    process[vorig] = 1
    Para  cada vdest ∈ Adjacentes[vorig]
      Se (process[vdest]=0 ∧ dist[vdest]>dist[vorig]+Peso_Ramo)
          dist[vdest] = dist[vorig] + Peso_Ramo
          path[vdest] = vorig
      FSe
    FPara
    vorig = vert_min_dist(dist, process)
  FEngto
  Fim CamMin_Pesado
```

Caminhos Mais Curtos - Algoritmo Pesado



path	0	5	2	5	1
	1	2	3	4	5

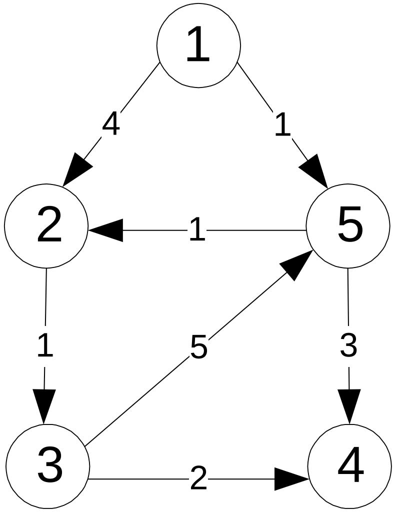
dist	0	2	3	4	1
	1	2	3	4	5

Caminhos mais curtos

com origem no vértice 1 : **Peso**

- 1-5-2 2
- 1-5-2-3 3
- 1-5-4 4
- 1-5 1

Escreve_Caminho



path					
	0	5	2	5	1
	1	2	3	4	5

dist					
	0	2	3	4	1
	1	2	3	4	5

```
getPath (path, vOrig, vDst)
   se (vOrig != vDst)
      apvert = encvert_conteudo(vfim) ;
      getPath (path, vOrig, vDst]) ;
        p.push(;
  }
  cout << vfim << " "  ;
```

Análise de Complexidade

Inicialização → O(|V|)

1º ciclo Enq^{to} → O(|V|)

Tempo de corrigir a distância é constante por actualização e há no máximo uma actualização por aresta, num total de O(|E|)

Pesquisa do mínimo: método de percorrer a tabela até encontrar o mínimo é O(|V|) em cada fase

⤷ gasta-se O($|V|^2$) ao longo de todo o processo

Escreve_Caminhos → O(|V|)

⤷ Tempo de execução → O(|V|×|E|)

Grafos

Ciclos ou Circuitos

Ciclos ou Circuitos em Grafos

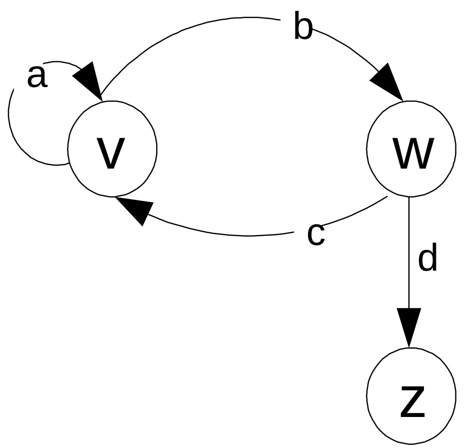
Um ciclo num grafo é um caminho fechado **sem vértices repetidos**

Um ciclo é um caminho (v_0, v_1, v_2, ..., v_k) com K >= 1, onde $v_k = v_0$, mas v_0, v_1, v_2, ..., v_{k-1} são distintos dois a dois

Se (u, v, w, ..., z, u) é um ciclo, então (v, w, ..., z, u, v) também é um ciclo

Qualquer permutação cíclica de um ciclo é um ciclo equivalente ao original

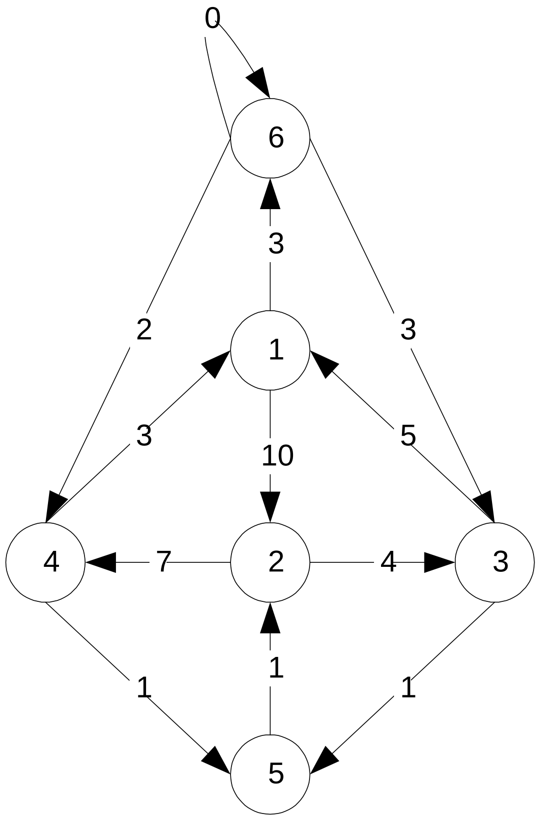
Exemplo:



A sequência:

- (v,v) é um ciclo de comprimento 1
- (v,w,v) é um ciclo de comprimento 2
- (w,v,w) é um ciclo equivalente ao anterior
- (w,v,v,w) não é um ciclo - repete o vértice v

Ciclos em Grafos



Lista de adjacências

1 → 2, 6
2 → 3, 4
3 → 1, 5
4 → 1, 5
5 → 2
6 → 3, 4, 6

Ciclos presentes no grafo (– 9 –) :

- 1-2-3-1
- 1-2-4-1
- 1-6-3-1
- 1-6-3-5-2-4-1
- 1-6-4-1
- 1-6-4-5-2-3-1
- 2-3-5-2
- 2-4-5-2
- 6-6

Ciclos em Grafos

```
Conta-Ciclos-Grafo ( )
     Vertice_Inic Lista_Vértices
     Enqto Lista_Vértices    //ciclo para percorrer todos os vértices
        origem = Vertice_Inic.key
        Para i=1 até origem       // os vértices já processados são marcados
                                  // como visitados para evitar ciclos equivalentes
           vector[i]=1
           Conta-Caminhos(Vertice_Inic,Vertice_Inic,vector,cont)
        Próximo Vértice Lista_Vértices
      FEnqto
Fim Conta-Ciclos-Grafo
```

Circuito de Euler

Circuito de Euler é circuito simples (sem ramos repetidos) que contém todos os ramos do grafo

De acordo com o Teorema Euler (1736)

- um **grafo orientado** admite um circuito de Euler sse for **fortemente conexo** e **pseudo-simétrico** ou seja, se para todos os vértices v_i grau_entrada (v_i) = grau_saida (v_i)

- um **grafo não orientado** admite um circuito de Euler sse for **conexo** e **não tiver vértices de grau ímpar**

Circuito de Euler

Prova de Euler (simplificada !)

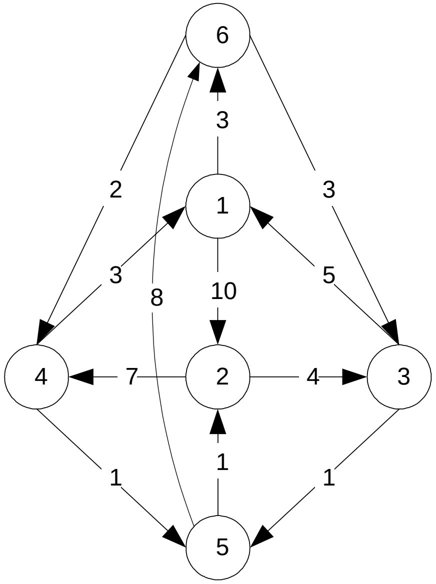
- como todos os vértices têm grau par é sempre possível entrar e sair de um vértice
- como cada vértice de G possui grau ≥ 2, G contém necessariamente algum ciclo simples C_1
- se C_1 contém todos os ramos de G
  - **C_1 é o circuito de Euler**
- senão
  - remove-se de G todas as arestas de C_1
  - no grafo resultante, cada vértice possui ainda grau par
  - determina-se novo ciclo ...... até não haver mais ramos em G
  - no final, os ramos de G encontram-se particionados em ciclos simples
  - como G é conexo, cada ciclo C_i tem pelo menos um vértice em comum
  - fazendo a união dos ciclos C_i obtém-se um ciclo que contém todas as arestas de G exatamente uma vez

Circuito de Euler – Alg. Hierholzer

Algoritmo Hierholzer, baseado na prova matemática de Euler

```
nciclos = 0
Enq^to existirem arestas em G
   escolher um vértice v
   realizar uma busca em profundidade em G, a partir de v,
   até encontrar um ciclo C_i
   retirar de G os ramos do ciclo C_i
   nciclos++
FEnq^to
Enq^to nciclos > 1
   Juntar C_i e C_j num único ciclo fechado
   nciclos--
FEnq^to
```

Circuito de Euler – Alg. Hierholzer

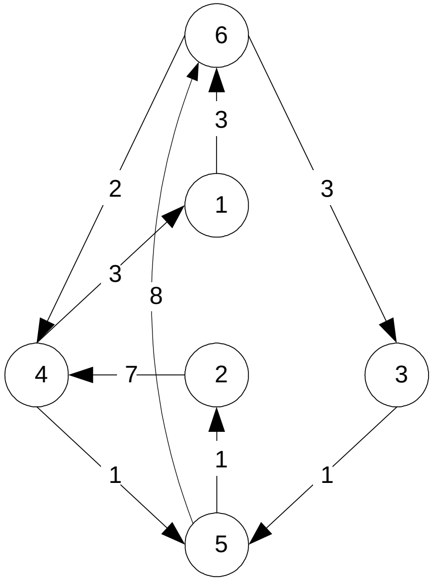


Lista de adjacências

1 → 2, 6
2 → 3, 4
3 → 1, 5
4 → 1, 5
5 → 2, 6
6 → 3, 4

1ª iteração:

1 – 2 – 3 – 1

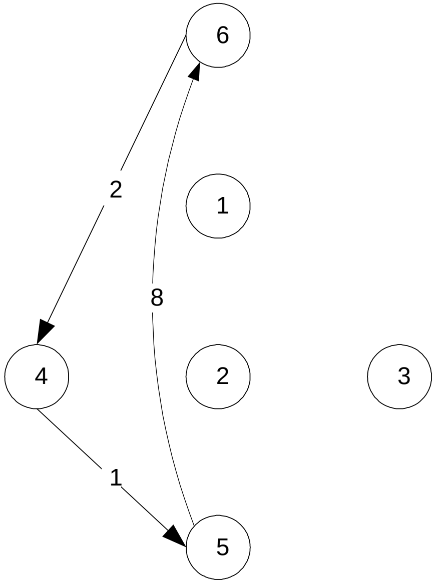


Lista de adjacências

1 → 6
2 → 4
3 → 5
4 → 1, 5
5 → 2, 6
6 → 3, 4

2ª iteração:

2 – 4 – 1 – 6 – 3 – 5 – 2



Lista de adjacências

1 →
2 →
3 →
4 → 5
5 → 6
6 → 4

3ª iteração:

4 – 5 – 6 – 4

Circuito de Euler – Alg. Hierholzer

Ciclos encontrados

1 – 2 – 3 – 1
2 – 4 – 1 – 6 – 3 – 5 – 2
4 – 5 – 6 – 4

Circuito Euler = União dos ciclos

1 – 2 – 3 – 1
2 – 4 – 1 – 6 – 3 – 5 – 2

1 – 2 – 4 – 1 – 6 – 3 – 5 – 2 – 3 – 1
4 – 5 – 6 – 4

Circuito Euler:

1 – 2 – 4 – 5 – 6 – 4 – 1 – 6 – 3 – 5 – 2– 3 – 1

Nota: Grafo com todos os vértices grau par, excepto dois vértices, apresenta um **caminho de Euler**, com início e fim nos dois vértices sem grau par

Grafos

Ordenação Topológica

Ordenação Topológica

É uma ordenação linear dos vértices num **grafo acíclico, dirigido** de tal forma que se existe um caminho do vértice v para o vértice w, então na ordenação, o vértice v aparece antes de w

Para que serve a ordenação topológica ?

Se os vértices de um grafo são tarefas, o arco (v,w) significa que a tarefa v tem de ser executada antes da tarefa w

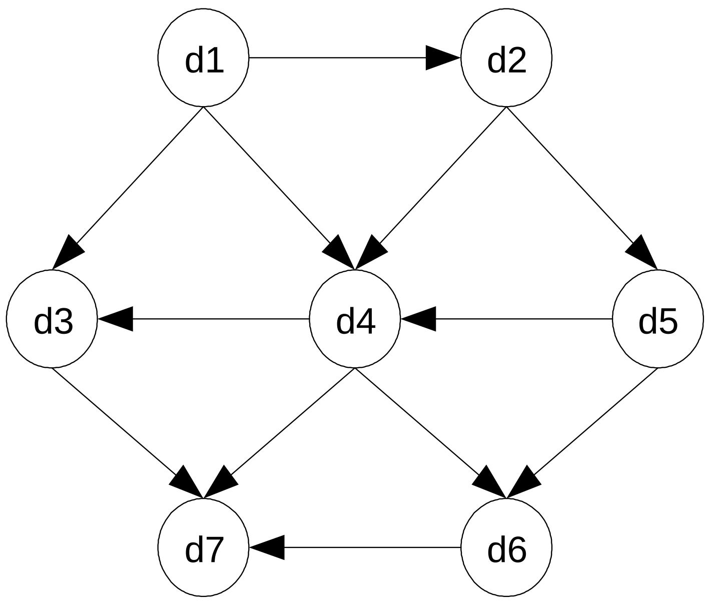
- a ordenação topológica fornece a programação sequencial das tarefas

Ordenação topológica:

- impossível se o grafo for cíclico
  - para dois vértices v_i e v_j do ciclo, se v_i precede v_j, também v_j precede v_i
  - só os grafos acíclicos apresentam numeração topológica
- não é necessariamente única

Ordenação Topológica - Exemplo

O grafo abaixo representa a estrutura de precedências de um curso



Os vértices representam as disciplinas e qualquer ramo (d_i,d_j) indica que a disciplina d_i deve ser concluída antes de se inscrever na disciplina d_j

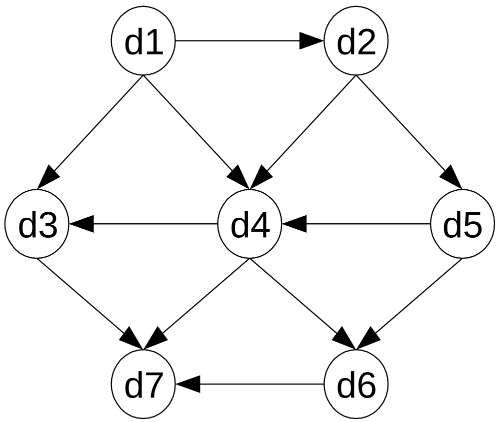
Algoritmo de Ordenação Topológica

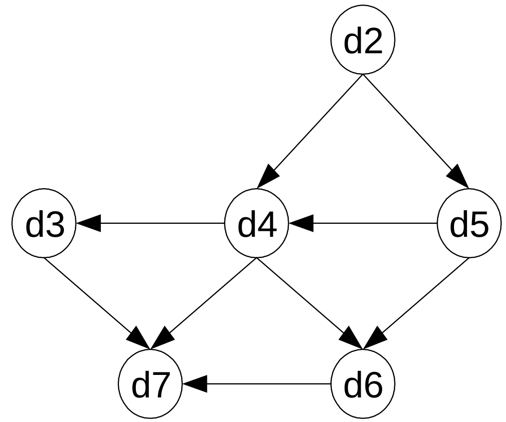
Lema: Todo o grafo dirigido acíclico tem pelo menos um vértice de grau de entrada zero

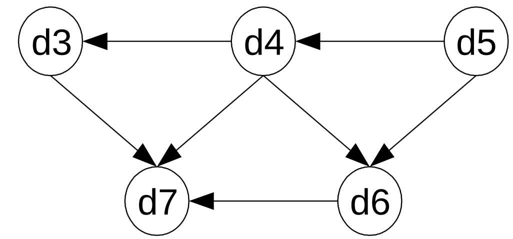
Algoritmo

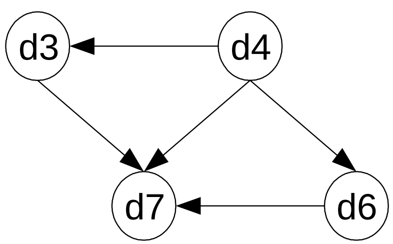
1. Construir um vector com os graus de entrada de todos os vértices no grafo
2. Inserir numa fila todos os vértices com grau de entrada zero
3. Enquanto a fila não estiver vazia
   - retirar um vértice – v – da fila
   - imprimir vértice v
   - decrementar 1 unidade o grau de entrada de todos os vértices adjacentes de v
   - se o grau de entrada de um desses vértices passa a zero
     - o vértice é colocado na fila

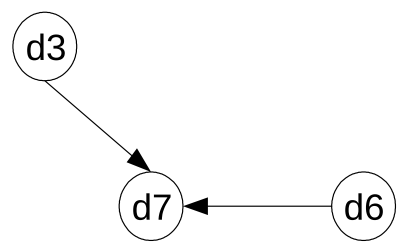
Exemplificação











O grafo acima apresenta as seguintes ordenações topológicas:

$d_1, d_2, d_5, d_4, d_3, d_6, d_7$

$d_1, d_2, d_5, d_4, d_6, d_3, d_7$

Pseudo-código Ordenação Topológica

```
Ordenação_Topologica ( )
  Para i=1 até nvertices
     apvertice = encvert_key(i)
     vector[Vertice:key]=grau_entrada_Vertice
     Se vector[Vertice:key] == 0
        insere_fila(apvertice)
  Fpara
  Enqto fila não vazia
     retira-fila(Vertice)
     imprime Vertice
     vertprocess++
     Enqto Ramos Adjac.Vertice
         outrovert=Vertice_Adjac
         vector[outrovert]--
         Se vector[outrovert]==0
            insere_fila(outrovert)
         Ramo Seg.
     FEnqto
  FEnqto
  Se vertprocess < numvert
    "Erro: Grafo cíclico"
Fim Ordenação_Topologica
```

Análise de Complexidade

Ordenação Topológica

- Inicialização do vector com grau de entrada dos vértices
  - $O(V) \times O(|V| \times |E|)$
- Inicialização da fila com os vértices grau de entrada = 0
  - proporcional ao número de vértices $O(|V|)$

- **1º Ciclo → $O(|V|^2 \times |E|)$**

- Enq^to fila não vazia → é executado no máximo **nº vértices do Grafo**
- O ciclo de actualização dos graus de entrada é executado no máximo uma vez por aresta → $O(|E|)$
- **2º Ciclo → $O(|V| \times |E|)$**

Complexidade → $O(|V|^2 \times |E|)$

Ordenação Topológica – Versão mais Simples

Algoritmo

- Utiliza-se o algoritmo DFS para processar os vértices ainda não visitados
- À medida que cada vértice termina (não tem vértices adjacentes) é colocado numa stack (o que garante que ele aparece depois dos seus antecedentes)
- No final do algoritmo, a stack contém a ordenação topológica do Grafo

Nota

- Este algoritmo pode ter início por qualquer vértice (independentemente do seu grau de entrada), consequentemente, é possível encontrar diferentes ordenações topológicas correctas para um mesmo grafo orientado acíclico

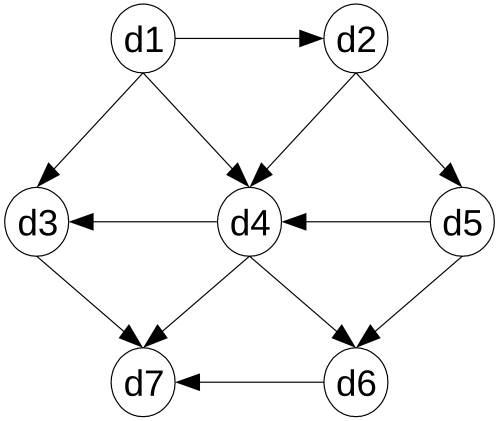
Ordenação Topológica – Versão mais Simples

```
Topological_Sort( )
VertInic = graf
Engto (VertInic /= Nulo)
   Se (vector[VertInic:key]==0)
      topolog_sort(VertInic,vector,topsort)
   FSe
   VertInic = Próximo_Vert
FEngto
Imprimir topsort
Fim Topological_Sort

topolog_sort (VertInic,vector,topsort)
  vector[VertInic:key] = 1
  Engto (Ramos Adjac.Vertice)
     Se (vector[VertAdjac:key]==0)
        topolog_sort(VertAdjac,vector,topsort)
     Fse
     apramo = apramo→apr
  FEngto
  push topsort(VertInic)
Fim topological_sort
```

Tempo de execução → $O(|V| \times |E|)$

Exemplificação



topolog_sort (1, vector, topsort)
topolog_sort (2, vector, topsort)
topolog_sort (4, vector, topsort)
topolog_sort (3, vector, topsort)
topolog_sort (7, vector, topsort)

vector	1	1	1	1	0	0	1
	1	2	3	4	5	6	7

topsort (7)
topsort (7, 3)

Lista de adjacências

1 → 2, 3, 4
2 → 4, 5
3 → 7
4 → 3, 6, 7
5 → 4, 6
6 → 7
7 →

topolog_sort (6, vector, topsort)

vector	1	1	1	1	0	1	1
	1	2	3	4	5	6	7

topsort (7, 3, 6)
topsort (7, 3, 6, 4)

topolog_sort (5, vector, topsort)

vector	1	1	1	1	1	1	1
	1	2	3	4	5	6	7

topsort (7, 3, 6, 4, 5)
topsort (7, 3, 6, 4, 5, 2)
topsort (7, 3, 6, 4, 5, 2, 1)

Grafos

Árvore de Cobertura de Custo Mínimo

Árvore de Cobertura de Custo Mínimo

- **Árvore** ((sub)grafo acíclico e conexo)
- **de Cobertura** (contendo todos os vértices e alguns ramos – os suficientes para ligar todos os vértices)
- **de Custo Mínimo** (nenhuma árvore de cobertura tem menor custo)

↪ *"Minimum Spanning Tree"* – MST

Exemplo de aplicação: cablamento de uma casa, minimização do comprimento total da ligação

- os vértices são as tomadas
- os ramos são os comprimentos dos troços de ligação

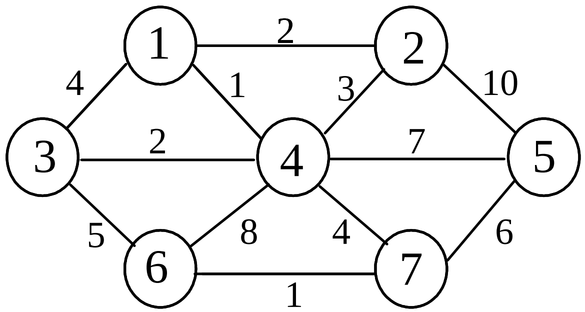
- Dado um grafo **não orientado** e **conexo**, como encontrar uma **Árvore Mínima de Cobertura** ?
  - Algoritmo de Kruskal
  - Algoritmo de Prim

Algoritmo de Kruskal

Dado um grafo $G = (V, E)$ ligado e não dirigido, com uma função de pesos $W: E \rightarrow R$

- considera-se cada vértice de G como pertencendo a uma árvore em G, ou seja, $v_i \in T_i$, $1 < i < |V|$, ou seja, inicialmente cada vértice é uma árvore
- ordenam-se todos os ramos por ordem crescente de peso
- e inserem-se numa fila de prioridade
- seguidamente, pega-se no menor ramo de G e verifica-se se este une 2 vértices pertencentes a árvores diferentes. Se sim, unificam-se as duas árvores
- repete-se a operação até todos os vértices terem sido ligados
- quando o algoritmo termina há uma só árvore - **de expansão mínima**

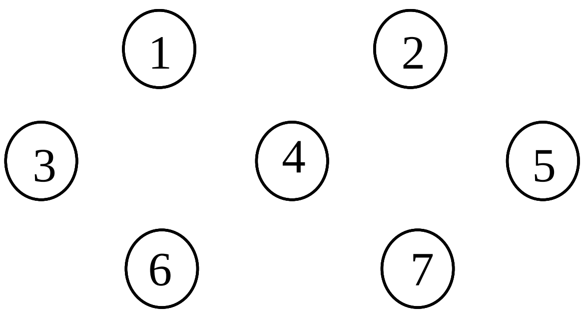
Evolução do Algoritmo de Kruskal

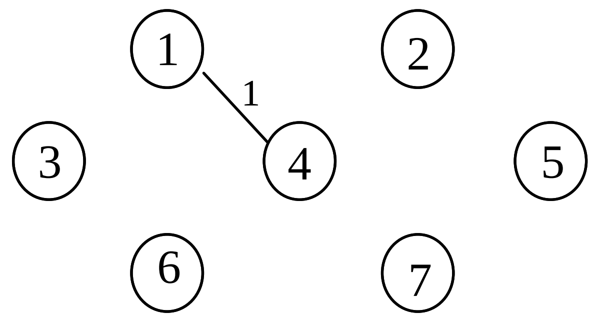


Conjunto ordenado de ramos:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}

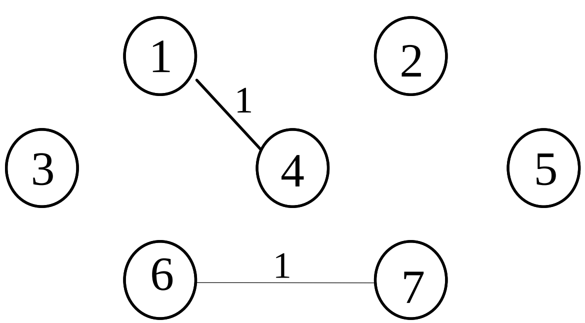
Q = {}





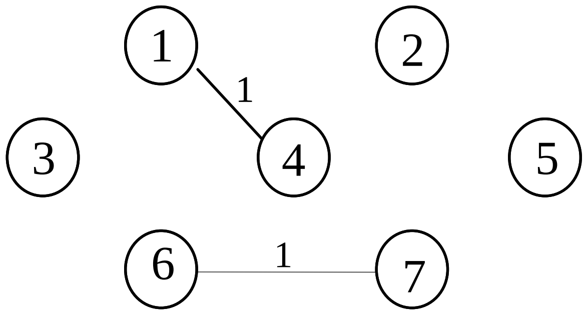
Q = {{1},{2},{3},{4},{5},{6},{7}}

Q = {{1,4},{2},{3},{5},{6},{7}}



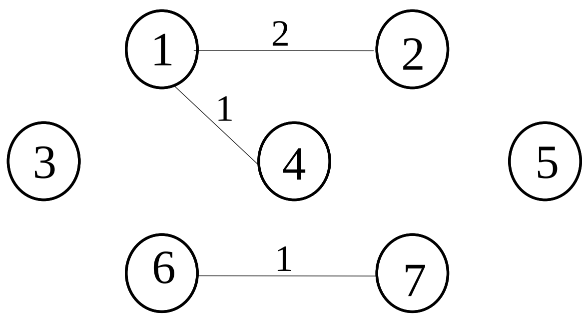
Q = {{1,4},{6,7},{2},{3},{5}}

Evolução do Algoritmo de Kruskal

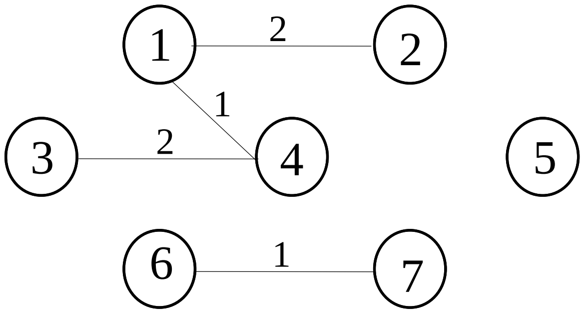


Conjunto ordenado de ramos:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}

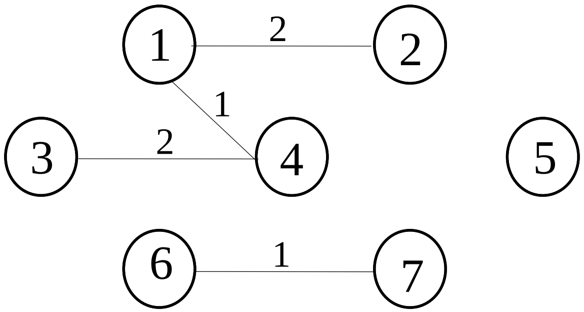


Q = {{1,4,2},{6,7},{3},{5}}

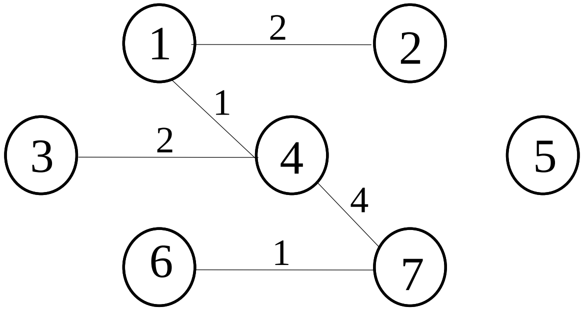


Q = {{1,4,2,3},{6,7},{5}}

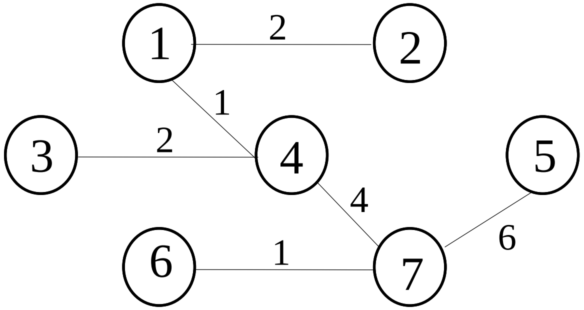
Evolução do Algoritmo de Kruskal



Conjunto ordenado de ramos:
{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}

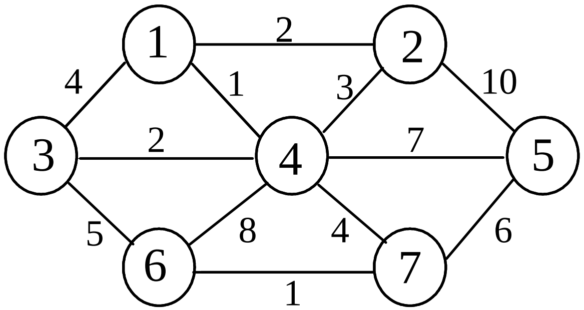


Q = {{1,4,2,3,6,7},{5}}



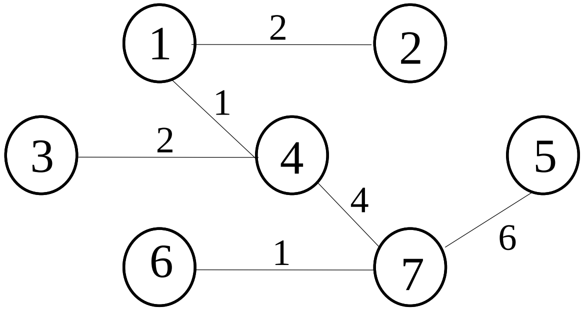
Q = {{1,4,2,3,6,7,5}}

Evolução do Algoritmo de Kruskal



Conjunto ordenado de ramos:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}



Árvore de Cobertura Mínima

A = {(1,4), (7,6), (1,2), (4,3), (4,7), (5,7)}

O peso da árvore A é dado pela soma dos seus arcos W(A) = 16

Pseudo-Código Algoritmo de Kruskal

```
Kruskal( )
   Q = ∅
   Para cada v ∈ V
     G.juntar-vertice(v)
   FPara
   construir fila de prioridade com os ramos por ordem cresc. peso w
   Para cada ramo(u,v) ∈ fila de prioridade
        Conta-Caminhos(u,v,vector,cont)
        Se  cont == 0
         G.juntar_ramo (w, v, u)
         G.juntar_ramo (w, u, v)
        FSe
   FPara
  devolve G
Fim Kruskal
```

Análise de Complexidade

- O algoritmo gasta $|V|$ para criar a lista inicial de vértices
- Construir Fila de Prioridade com os ramos do grafo é $O(|V| \times |E|)$
- Último ciclo $O(|E| \times |V| \times |E|)$

↪ **Complexidade do algoritmo** $\rightarrow O(|V| \times |E|^2)$

Algoritmo de Prim

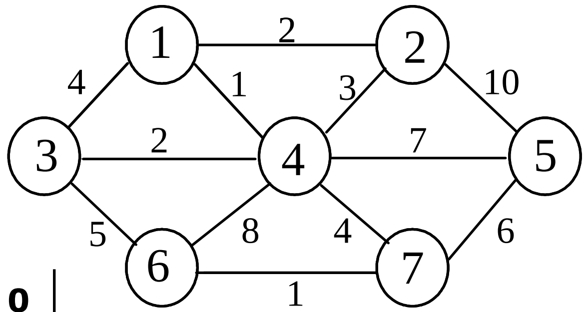
Dado um grafo G=(V,E) ligado e não dirigido, função de pesos W: E → R

Este algoritmo funciona com três vectores auxiliares:

- pesos [ ] → contém o menor peso até à altura conseguido para alcançar o vértice i
- pai [ ] → contém o vértice predecessor relativo ao peso do vector pesos
- vertices [ ] → indica os vértices já processados

- Escolhe-se o vértice ainda não processado com menor peso → vorigem
- Em seguida, actualizam-se todos os pesos dos vértices adjacentes ao vorigem e ainda não processados
- Repete-se a operação até todos os vértices terem sido processados
- Quando o algoritmo termina a **árvore de expansão mínima** encontra-se no vector pai [ ]

Evolução do Algoritmo de Prim



Vértice inicial: 7

pai	0	0	0	7	7	7	0
	1	2	3	4	5	6	7
pesos	∞	∞	∞	4	6	1	0
	1	2	3	4	5	6	7
vertices	0	0	0	0	0	0	1
	1	2	3	4	5	6	7

u = minimo(pesos), u = 6

pai	∞	∞	6	7	7	7	0
	1	2	3	4	5	6	7
pesos	∞	∞	5	4	6	1	0
	1	2	3	4	5	6	7
vertices	0	0	0	0	0	1	1
	1	2	3	4	5	6	7

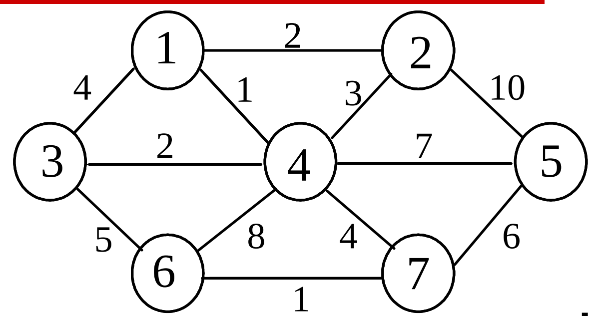
u = minimo(pesos), u = 4

pai	4	4	4	7	7	7	0
	1	2	3	4	5	6	7
pesos	1	3	2	4	6	1	0
	1	2	3	4	5	6	7
vertices	0	0	0	1	0	1	1
	1	2	3	4	5	6	7

u = minimo(pesos) , u = 1

pai	4	1	4	1	7	7	0
	1	2	3	4	5	6	7
pesos	1	2	2	1	6	1	0
	1	2	3	4	5	6	7
vertices	1	0	0	1	0	1	1
	1	2	3	4	5	6	7

Evolução do Algoritmo de Prim



u = minimo (pesos)
u = 2 → não altera vector pesos

pai	4	1	4	1	7	7	0
	1	2	3	4	5	6	7

pesos	1	2	2	1	6	1	0
	1	2	3	4	5	6	7

vertices	1	1	0	1	0	1	1
	1	2	3	4	5	6	7

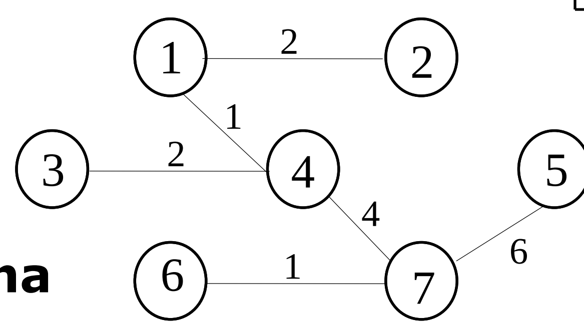
u = minimo (pesos) → u = 3
Adj[3] = {1, 4, 6} → vértices já processados
→ não altera vector pesos

vertices	1	1	1	1	0	1	1
	1	2	3	4	5	6	7

u = minimo (pesos) → u = 5
Adj[5] = {2, 4, 7} → vértices já processados
→ não altera vector pesos

vertices	1	1	1	1	1	1	1
	1	2	3	4	5	6	7

Árvore de Cobertura Mínima



Pseudo-Código Algoritmo de Prim

```
Prim (TV vinicio)
    Para cada v ∈ V
        pesos[v] = pvertices
        pai[v] = 0
        vertices[v] = 0
    FPara
    peso[vinicio] = 0          // vértice inicial pode ser um qualquer
    Engto  vertices[i] /= 1    // vértices não processados
        u = minimo(pesos)      // selecciona vértice com menor peso
        vertices[u] = 1
        Para  cada v ∈ Adjacentes[u]
            Se  (vertices[v]==0 ∧ w(u,v) < peso[v])
                pai[v] = u
                pesos[v] = w(u,v)
            FSe
        FPara
    FEngto
Fim Prim
```

Análise de Complexidade

- O algoritmo gasta O(|V|) para criar o conjunto inicial de vértices

- O ciclo Enq^{to} é executado |V| vezes e como cada operação de extracção do mínimo de pesos leva |V| vezes → O($|V|^2$)

- A actualização de pesos leva O(|V|×|E|) pois o ciclo Para é no máximo executado O(|E|) vezes

    - **Complexidade do algoritmo** → O (|V|×|E|)

Outros problemas

- Fluxo Máximo numa Rede de Transporte
  Aplicações:
  - abastecimento de líquido ponto a ponto
  - tráfego entre dois pontos
  - sistemas de rega
    - Algoritmo Ford-Fulkerson
- Coloração de um mapa
  ...