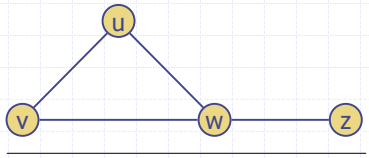
SCC-211
Lab. Algoritmos Avançados

Capítulo 9
Grafos

Adaptado por João Luís G. Rosa

Representação (Skiena & Revilla, 2003)

- Vértices rotulados:
  - Chaves (índices) são associadas aos vértices
- Arestas sem elementos.
- Matrizes e vetores dimensionados para:
  - No. de vértices (MAXV)
  - Grau (MAXDEGREE)
- Permite representar multi-grafos e ter o desempenho da lista de adjacências sem alocação dinâmica.



	v	u	w	z
0	1	2	3	4

edges

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

0 1 2

MAXDEGREE

degree

0	
1	2
2	2
3	3
4	1

MAXV

Representação (Skiena & Revilla, 2003)


```
/*  graph.h */

#define MAXV          100      /* maximum number of vertices */
#define MAXDEGREE     50       /* maximum outdegree of a vertex */

typedef struct {
    int edges[MAXV+1][MAXDEGREE]; /* adjacency info */
    int degree[MAXV+1];
    int nvertices;
    int nedges;
} graph;
```

```
#include "graph.h"
initialize_graph(graph *g) {
    int i,j;  /* counters */

    g -> nvertices = 0;
    g -> nedges = 0;

    for (i=1; i<=MAXV; i++) {
        g->degree[i] = 0;
        for (j=1; j<=MAXDEGREE; j++)
            g->edges[i][j] = 0;
    }
}
```

Representação (Skiena & Revilla, 2003)


```
#include "bool.h"
#include "graph.h"

read_graph(graph *g, bool directed) {
    int i;          /* counter */
    int m;          /* number of edges */
    int x, y;       /* vertices in edge (x,y) */

    initialize_graph(g);

    scanf("%d %d",&(g->nvertices),&m);

    for (i=1; i<=m; i++) {
        scanf("%d %d",&x,&y);
        insert_edge(g,x,y,directed);
    }
}
```

```
/*  bool.h */

#define TRUE    1
#define FALSE   0

typedef int bool;
```

```
print_graph(graph *g) {
    int i,j;            /* counters */
    for (i=1; i<=g->nvertices; i++) {
        printf("%d: ",i);
        for (j=0; j<g->degree[i]; j++)
            printf(" %d",g->edges[i][j]);
        printf("\n");
    }
}
```

Representação (Skiena & Revilla, 2003)


```
insert_edge(graph *g, int x, int y, bool directed) {
    if (g->degree[x] > MAXDEGREE)
        printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

    g->edges[x][g->degree[x]] = y;
    g->degree[x] ++;

    if (directed == FALSE) insert_edge(g,y,x,TRUE);
    else g->nedges ++;
}
```

Duas arestas direcionadas, (x,y) e (y,x), representando uma aresta não-direcionada

```
delete_edge(graph *g, int x, int y, bool directed) {
    int i;              /* counter */
    for (i=0; i<g->degree[x]; i++)
        if (g->edges[x][i] == y) {
            g->degree[x] --;
            g->edges[x][i] = g->edges[x][g->degree[x]];
            if (directed == FALSE)
                delete_edge(g,y,x,TRUE);
            return;
        }
    printf("Warning: deletion(%d,%d) not found in g.\n",x,y);
}
```

Busca em Profundidade (DFS)


```
dfs(graph *g, int v) {

    int i;              /* counter */
    int y;              /* successor vertex */

    discovered[v] = TRUE;
    process_vertex(v);

    for (i=0; i<g->degree[v]; i++) {
        y = g->edges[v][i];
        if (discovered[y] == FALSE) {
            parent[y] = v;
            dfs(g,y);
        } else if (processed[y] == FALSE) process_edge(v,y);
        if (finished) return;  /* allow for search termination */
    }
    processed[v] = TRUE;
}
```

Busca em Profundidade - Complexidade

- Quando uma **matriz de adjacências** é utilizada, o procedimento DFS (*Depth-First Search*) requer $O(|V|^2)$.
- Quando uma **lista de adjacências** é utilizada, a busca profundidade requer $O(|V|+|A|)$.
- Repare que DFS é ótima para listas de adjacências (por que?).
  - DFS não revisita vértices, *backtracking* sim.

Algoritmos baseados na Busca em Profundidade

- Teste de existência de ciclos (linear);
- Teste de conectividade fraca (linear);
- Encontrar componentes fracamente conexos (linear);
- Teste de conectividade forte (linear);
- Encontrar componentes fortemente conexos (linear);
- Fechamento transitivo ($O(|V|(|V|+|A|))$);
- Ordenação topológica (linear);
- Identificação de grafos bi-coloridos, bipartidos ou com ciclos de tamanho ímpar (linear);
- Identificação de pontes (linear);
- Identificação de vértices de articulação (linear).

Teste de Existência de Ciclos

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou se contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em profundidade em *G*, então o grafo é cíclico.

Teste de Existência de Ciclos

- Isso sugere as seguintes especializações de DFS para buscar ciclos:

```
process_vertex(int v) {
}

process_edge(int x, int y) {
    if (parent[x] != y) {    /* found back edge! */
        printf("Cycle from %d to %d:",y,x);
        find_path(y,x,parent);
        printf("\n\n");
        finished = TRUE;
    }
}
```

Encontrar Componentes Fracamente Conexos

- Componentes conexos de um grafo é um conjunto máximo de vértices tal que existe um caminho entre todos os pares de vértices.
  - Existem problemas aparentemente complexos, como testar se 15-puzzle ou cubo mágico podem ser solucionados a partir de qualquer posição, que se resumem a um teste de conectividade.
  - Componentes fracamente conexos podem ser facilmente encontrados a partir de uma busca em largura ou profundidade.
  - Basta iniciar a busca e verificar se existem vértices não descobertos. Reiniciar a busca a partir de um vértice não descoberto até que todos sejam descobertos.

Encontrar Componentes Fracamente Conexos

```
connected_components(graph *g) {
    int c, i;

    initialize_search(g);

    c = 0;
    for (i = 1; i <= g->vertices; i++) {
        if (discovered[i] == false) {
            c++;
            printf("Component %d:", c);
            dfs(g, i);
            printf("\n");
        }
    }
}
```

Encontrar Componentes Fracamente Conexos

```
connected_components(graph *g) {
    int c, i;

    initialize_search(g);

    c = 0;
    for (i = 1; i <= g->vertices; i++) {
        if (discovered[i] == false) {
            c++;
            printf("Component %d:", c);
            dfs(g, i);
            prin
        }
    }
}
```

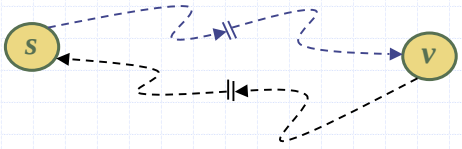
```
process_vertex(int v) {
    printf(" %d", v);
}

process_edge(int x, int y) {
}
```


Teste de Conexão Forte

- Podemos executar DFS ou BFS (*breadth-first search*) múltiplas vezes e verificar se o grafo é fortemente conexo.
- Verifica-se se a partir de cada vértice tomado como origem todos os demais vértices são alcançáveis ou não: $O(|V| (|V| + |A|))$ no pior caso.

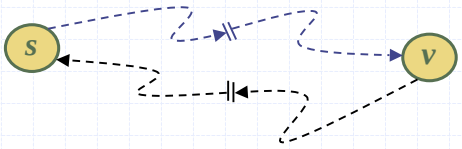
Teste de Conexão Forte Eficiente

- Basta que exista um único vértice de um dígrafo G que alcance qualquer outro e que seja alcançável por qualquer outro para que todos os vértices de G possuam essa mesma propriedade através dele $\Rightarrow$ G fortemente conexo.
  - Note que um vértice s alcança qualquer outro v e é alcançável por v se e somente se s alcança v em ambos os dígrafos G e G^T (transposto), pois o ciclo direcionado entre s e v se mantém (invertido) em G^T.
  - Logo, basta tomar qualquer vértice e executar DFS ou BFS duas vezes, uma sobre o dígrafo original G e a outra sobre G^T: $O(|V| + |A|)$.



Encontrar Componentes Fortemente Conexos

- Podemos utilizar a propriedade anterior também para calcular os componentes fortemente conexos de G:
  - Toma-se um vértice v e calcula-se o componente fortemente conexo que inclui v como todos aqueles vértices (e as respectivas arestas) que são alcançados por v em ambos os dígrafos G e G^T.
  - Faz-se isso sucessivas vezes, sempre a partir de um vértice não presente no componente fortemente conexo anterior.



Fechamento Transitivo

- Um fechamento transitivo F é um grafo construído a partir de $G = (V, A)$, tal que se existe um caminho entre 2 vértices em G, então existe uma aresta em F.
- É simples calcular o fechamento transitivo de um dígrafo G via DFS ou BFS executando o percurso a partir de cada vértice s de G e inserindo uma aresta direcionada adicional ligando a origem s a cada vértice alcançável a partir de s (se esta aresta já não existir).
  - Tempo = $|V|$ percursos $\Rightarrow$ $O(|V|(|V| + |A|))$ no pior caso.
  - Superior a algoritmo de Floyd-Warshall ($O(|V|^3)$) se G não for denso.

Floyd Warshall [3]

- Floyd-Warshall é um algoritmo de caminhos mais curtos do tipo *all-to-all*, o que significa que apresenta como resultado colateral quais vértices são alcançados por quais. Aqueles pares de vértices alcançáveis de um para o outro se caracterizam por uma distância mínima final não-infinita.
- Problema motivador: dado um grafo ponderado conectado *G* com $V \leq 100$ e dois vértices *s1* e *s2*, ache um vértice *v* em *G* que represente o melhor ponto de encontro, i.e., `dist[s1][v] + dist[s2][v]` é o mínimo para todo *v* possível. Qual é a melhor solução?

Floyd Warshall [3]

- Este problema requer a informação de menor caminho de duas fontes s1 e s2 para todos os vértices em *G*.
- Isso pode ser facilmente resolvido com duas chamadas ao algoritmo de Dijkstra.
- Uma de s1 para produzir o vetor de distância mais curta **`dist1`** de s1 e um de s2 para produzir **`dist2`**.
- Então repete através de todos os vértices possíveis no grafo para achar v tal que **`dist1[v]+dist2[v]`** é minimizada.
- Podemos fazer melhor? Floyd Warshall é a solução.

Floyd Warshall [3]

- Se o grafo tem $V \leq 100$ então há um algoritmo mais simples – em termos de implementação – para resolver este problema!
- Armazene o pequeno grafo em uma matriz de adjacência e rode o seguinte código com três laços aninhados:

```
REP (k,0,V-1)// #define REP(i,a,b)
             // for(int i=int(a); i<=int(b); i++)
  REP(i,0,V-1)
      REP(j,0,V-1)
        AdjMatrix[i][j] = min(AdjMatrix[i][j],
            AdjMatrix[i][k] + AdjMatrix[k][j]);
```

Floyd Warshall [3]

- Quando terminar, **`AdjMatrix[i][j]`** conterá a distância do caminho mais curto entre dois pares de vértices *i* e *j* em *G*.
- O problema original agora ficou fácil!
- Este algoritmo é chamado de Floyd-Warshall que roda em $O(V^3)$, mas como $V \leq 100$ isto é factível.
- O algoritmo de Floyd Warshall deve usar matriz de adjacências tal que o peso de aresta(i,j) possa ser acessado em O(1).

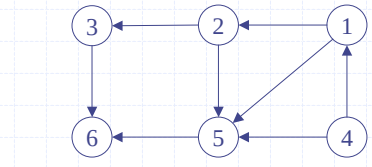
Ordenação Topológica

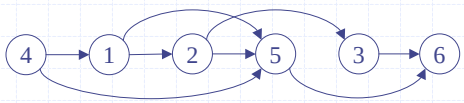
- Um grafo direcionado acíclico é também chamado de **DAG** (*directed acyclic graph*).
- Um DAG é diferente de uma árvore, uma vez que as árvores são não direcionadas.
- DAGs podem ser utilizados, por exemplo, para indicar precedências entre eventos.

Ordenação Topológica

- A ordenação topológica é uma ordenação linear de todos os vértices, tal que se *G* contém uma aresta *(u,v)* então *u* aparece antes de *v*.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.

Ordenação Topológica

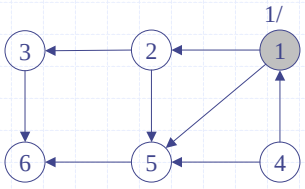




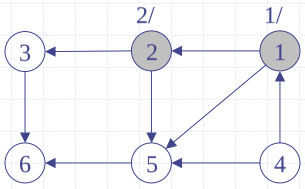
Ordenação Topológica

- A ordenação topológica de um DAG pode ser obtida utilizando-se uma busca em profundidade.
- Para isso deve-se fazer o seguinte algoritmo:
  1. Faça uma busca em profundidade;
  2. Quando um vértice é processado, insira-o numa lista de vértices;
  3. Retorne a lista de vértices.

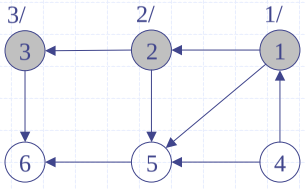
Ordenação Topológica



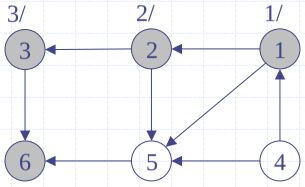
Ordenação Topológica



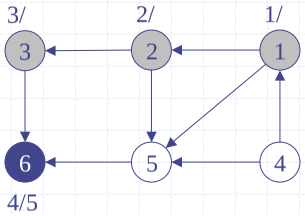
Ordenação Topológica



Ordenação Topológica

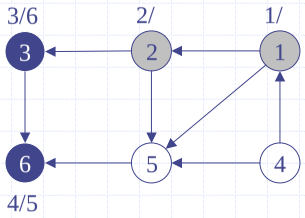


Ordenação Topológica



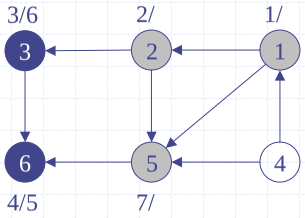
6

Ordenação Topológica



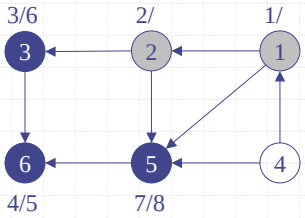
3 6

Ordenação Topológica



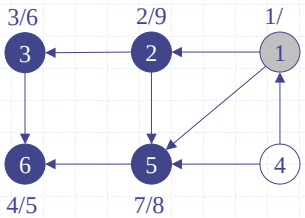
3 6

Ordenação Topológica



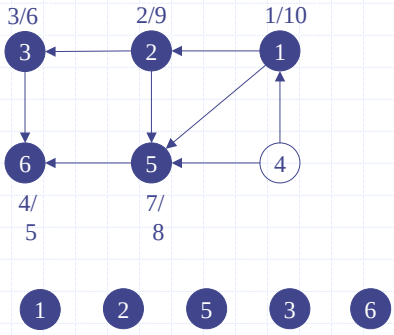
5 3 6

Ordenação Topológica

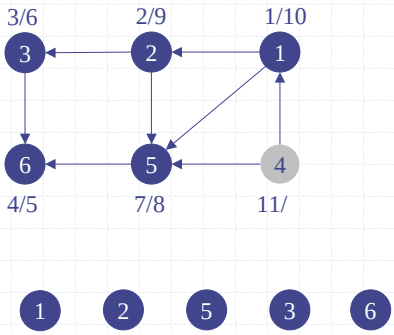


2 5 3 6

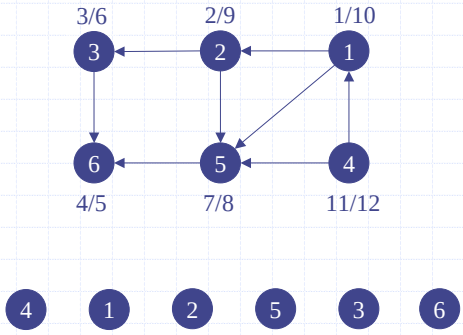
Ordenação Topológica



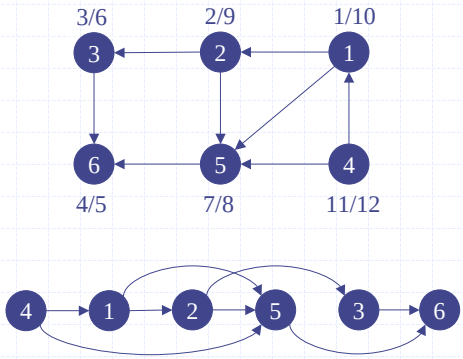
Ordenação Topológica



Ordenação Topológica



Ordenação Topológica

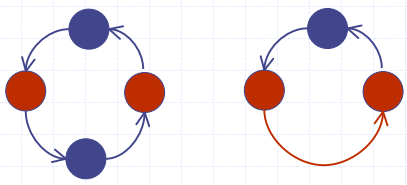


Ordenação Topológica

- A complexidade do algoritmo de ordenação topológica em um DAG é a mesma da busca em profundidade, ou seja:
  - $O(|V|^2)$ para matrizes de adjacência, e;
  - $O(|V|+|A|)$ para listas de adjacência.
- Inserir um elemento na fila é $O(1)$.

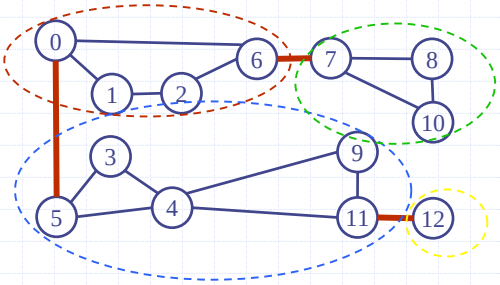
Grafos bi-coloridos, bipartidos ou com ciclos de tamanho ímpar

- Esses três problemas são equivalentes:
  - Os dois primeiros são diferentes nomenclaturas para o mesmo problema;
  - Qualquer grafo com um ciclo de tamanho ímpar é claramente não bi-colorível.

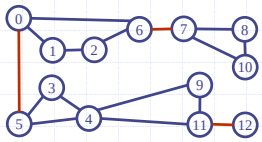


Pontes

- Uma **ponte** em um grafo é uma aresta que, se removida, separaria um grafo conexo em dois grafos disjuntos.

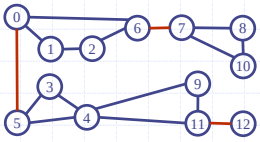


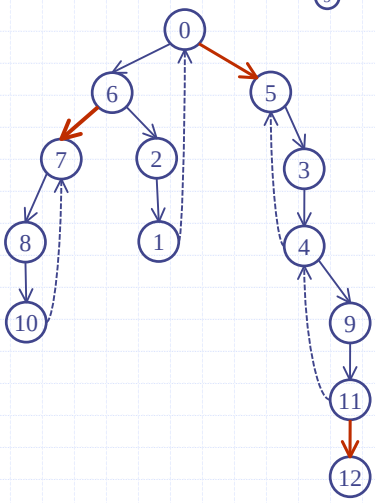
Pontes



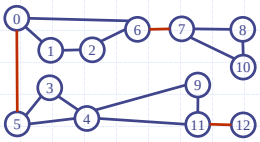
- Em uma árvore de busca em profundidade, uma aresta *v*-*w* é uma ponte se e somente se não existem arestas de retorno que conectam um descendente de *w* a um ancestral de *v*.

Pontes

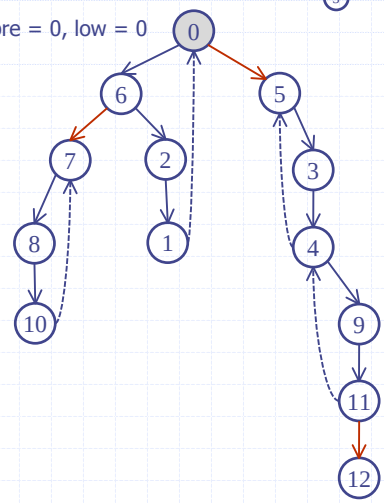




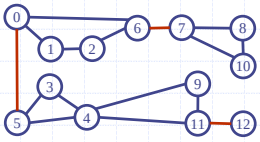
Pontes

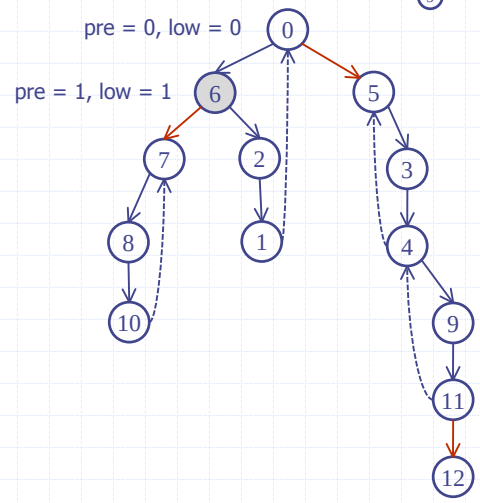


pre = 0, low = 0

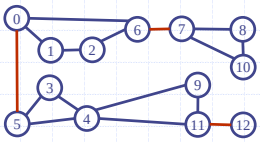


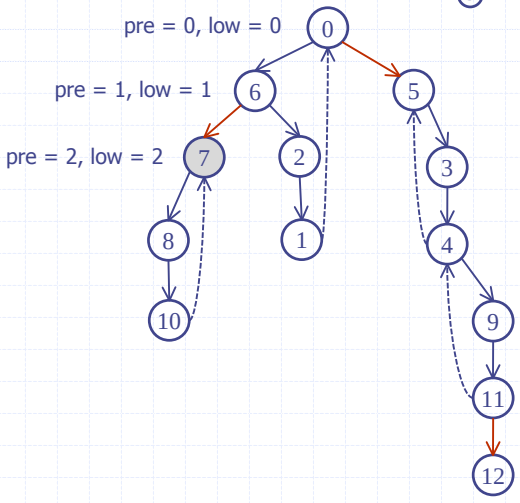
Pontes



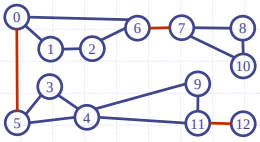


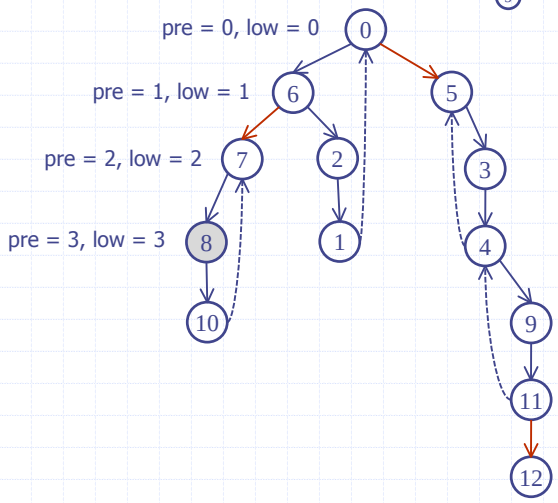
Pontes



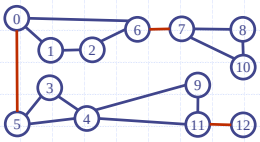


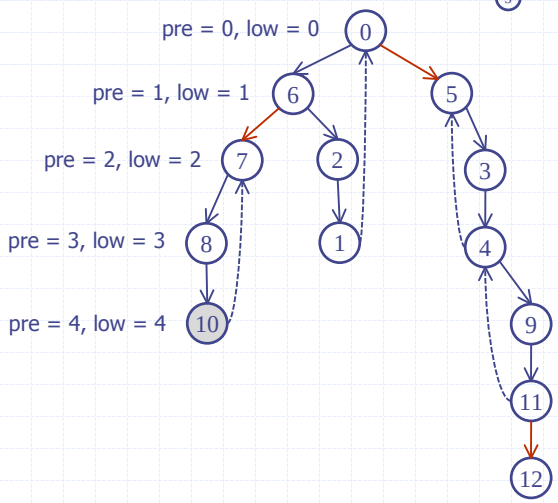
Pontes



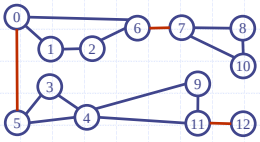


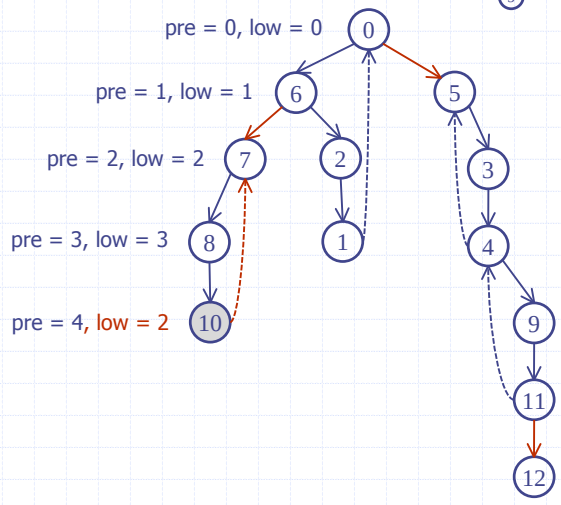
Pontes



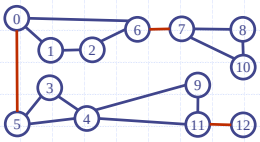


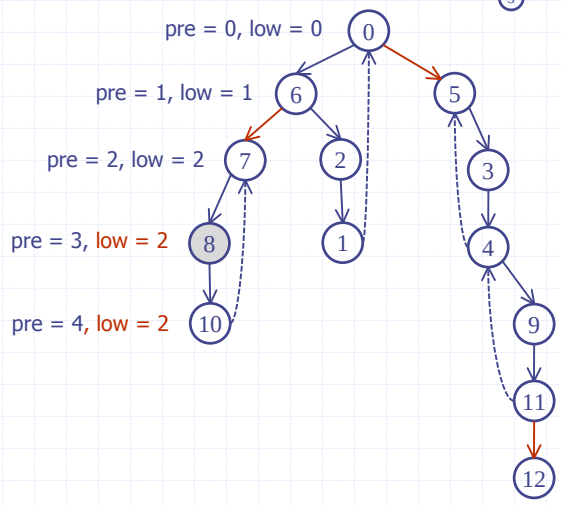
Pontes



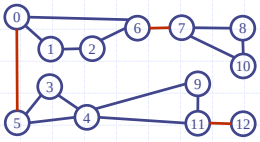


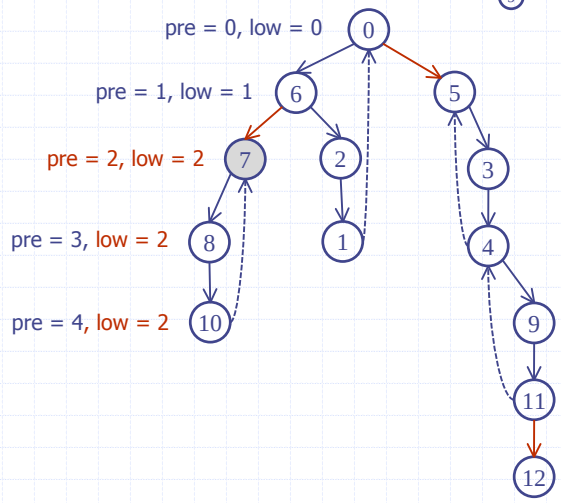
Pontes



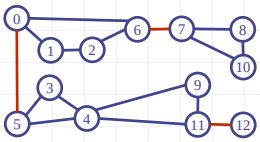


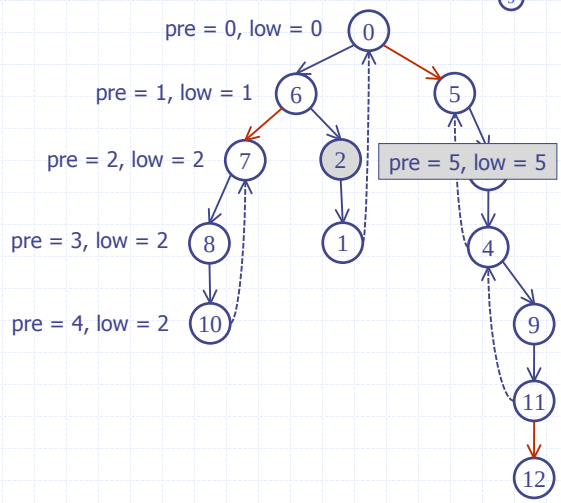
Pontes



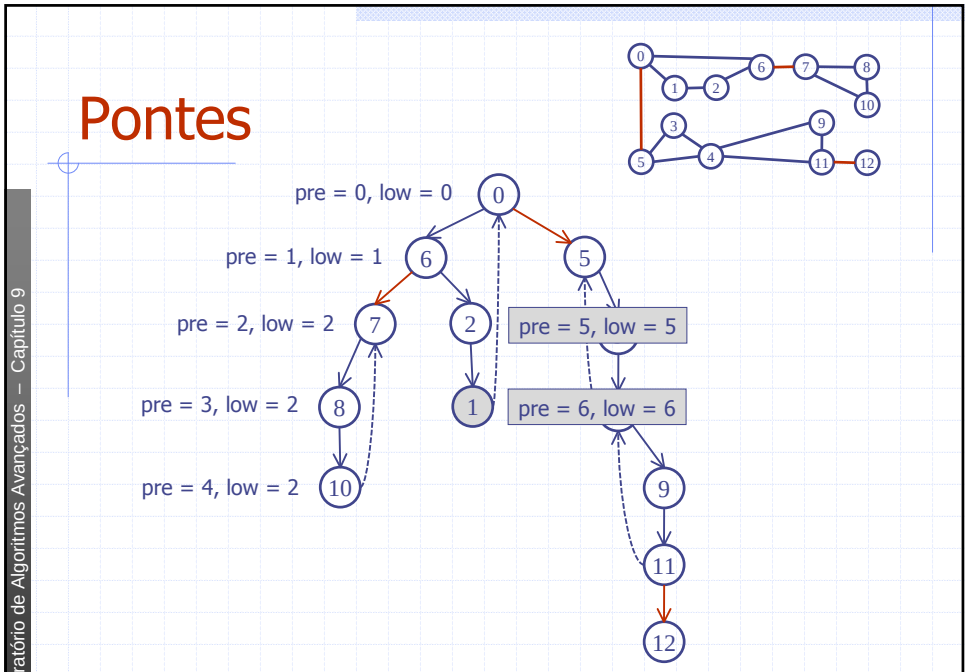


Pontes

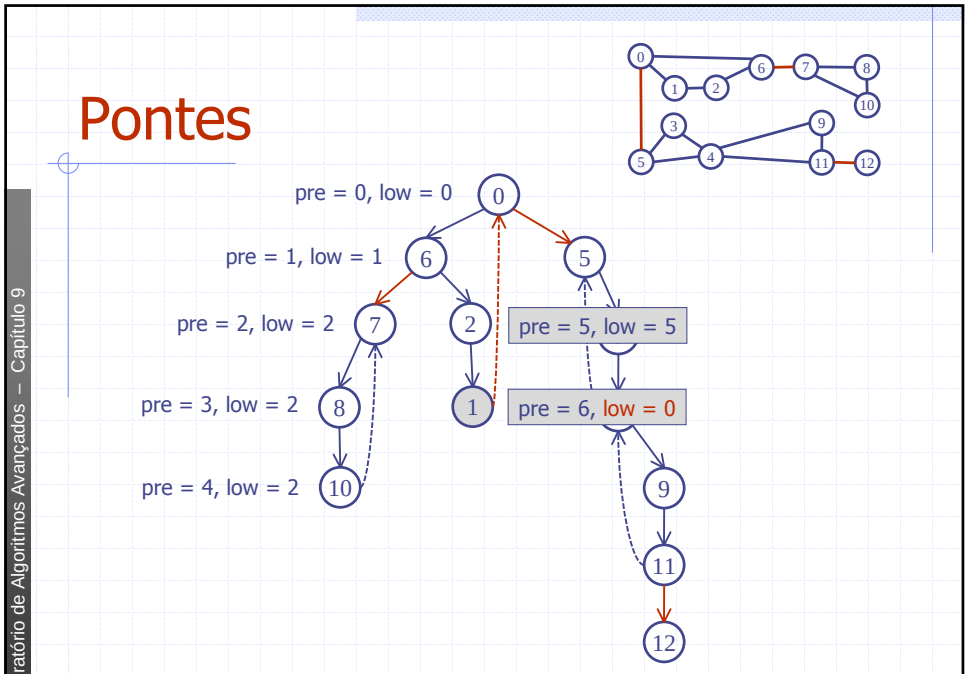




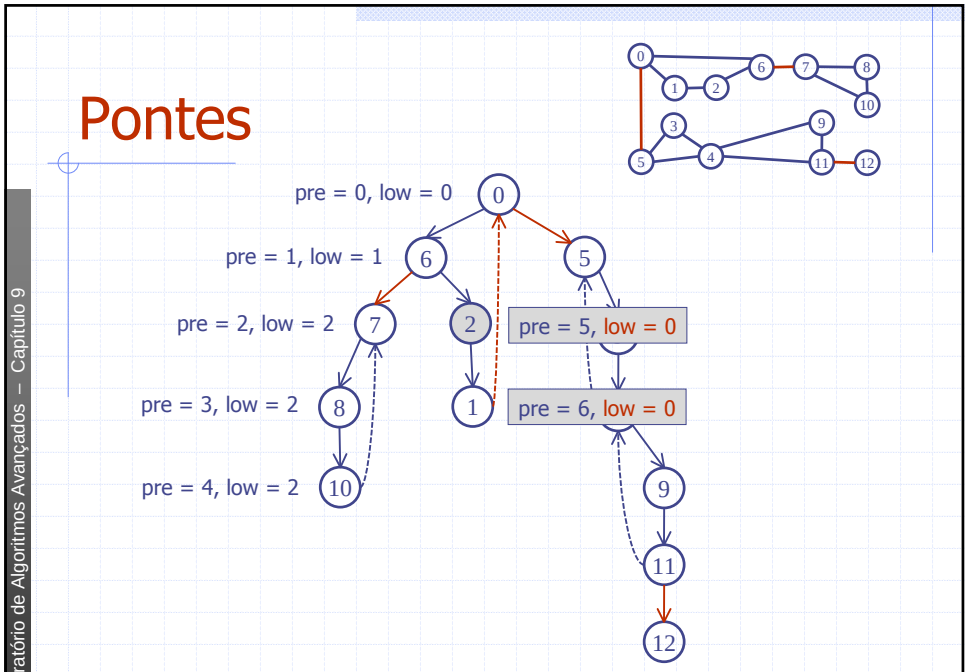
Pontes



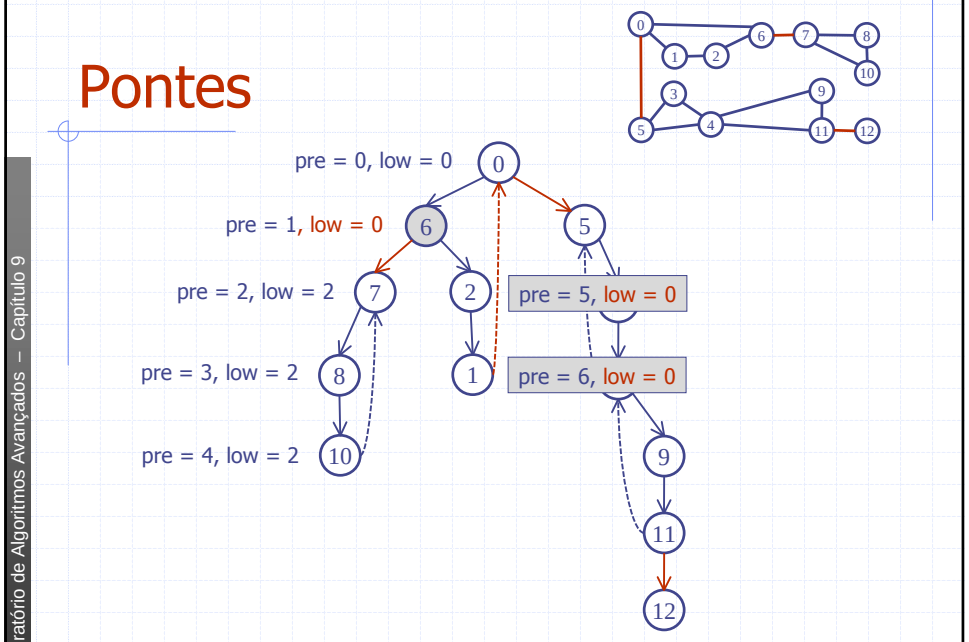
Pontes



Pontes



Pontes



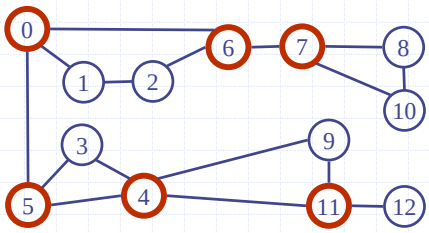
Pontes: algoritmo

```
void bridge(int v, int ant) {
    int adj;

    pre[v] = cnt++;                          // cnt iniciado com 0
    low[v] = pre[v];
    for (adj=0; adj<N; i++)
        if (m[v][adj] > 0)
            if (pre[adj] == -1) {            // pre iniciado com -1
                bridge(adj, v);
                if (low[v] > low[adj])
                    low[v] = low[adj];
                if (low[adj] == pre[adj]) {
                    bcnt++;                  // bcnt iniciado com 0
                    printf("%d-%d\n", v, adj);
                }
            } else if (adj != ant)
                if (low[v] > pre[adj])
                    low[v] = pre[adj];
}
```


Pontos de articulação

- Um vértice ou **ponto de articulação** (ou ainda vértice de separação) é um vértice que, se removido, separaria o grafo em pelo menos dois subgrafos disjuntos.
- Existem pontos de articulação associados a pontes, mas existem outros não associados.

Pontos de articulação



Pontos de articulação

- Um grafo é dito ser biconectado se cada par de vértices é conectado por dois caminhos disjuntos.
- Um grafo é biconectado se e somente se ele não possui pontos de articulação.

Pontos de articulação

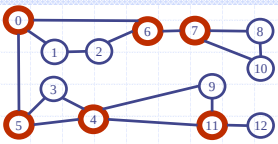
- Para encontrar pontos de articulação pode-se usar um algoritmo similar ao de pontes:
  - Um ponto de articulação é desprovido de uma aresta de retorno que liga um descendente a um ancestral.
  - O problema é a raiz da árvore de busca em profundidade:
    - A raiz de uma árvore de busca em profundidade é um ponto de articulação se e somente se ela possuir dois ou mais filhos.

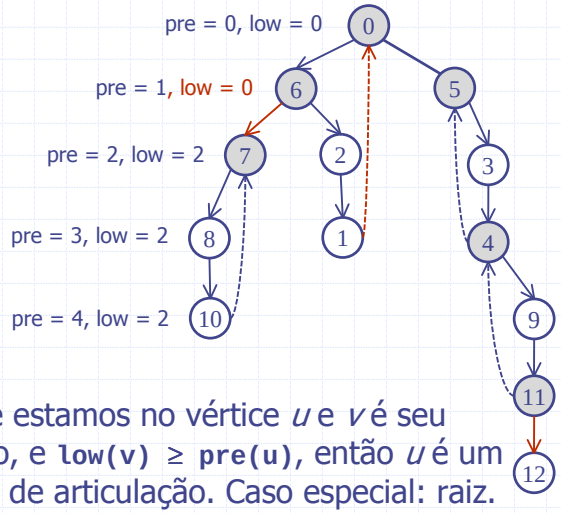
Pontos de articulação: algoritmo

```
void articulation_point(int v, int ant, int root) {
    int adj;

    pre[v] = cnt++;                              // cnt inicializado com 0
    low[v] = pre[v];
    for (adj=0; adj<N; adj++)
        if (m[v][adj] > 0)
            if (pre[adj] == -1) {                // pre inicializado com -1
                if (root)
                    child_root++;                // child_root inicializado com 0
                articulation_point(adj, v, 0);
                if (low[v] > low[adj])
                    low[v] = low[adj];
                if (low[adj] >= pre[v])
                    art_vertex[v] = 1;           // art_vertex inicializado com 0
            } else if (adj != ant)
                if (low[v] > pre[adj])
                    low[v] = pre[adj];
}
```

```
articulation_point(0, -1, 1);
art vertex[0] = child root > 1;
```

Pontos de articulação





Se estamos no vértice *u* e *v* é seu vizinho, e `low(v)` ≥ `pre(u)`, então *u* é um ponto de articulação. Caso especial: raiz.

O algoritmo de Kruskal

- O algoritmo de Kruskal é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima (MST – *minimum spanning tree*) para um grafo conexo com pesos.
- Isto significa que ele encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado.

O algoritmo de Kruskal

- Se o grafo não for conexo, então ele encontra uma floresta geradora mínima (uma árvore geradora mínima para cada componente conexo do grafo).
- O algoritmo de Kruskal é um exemplo de um algoritmo *guloso.*

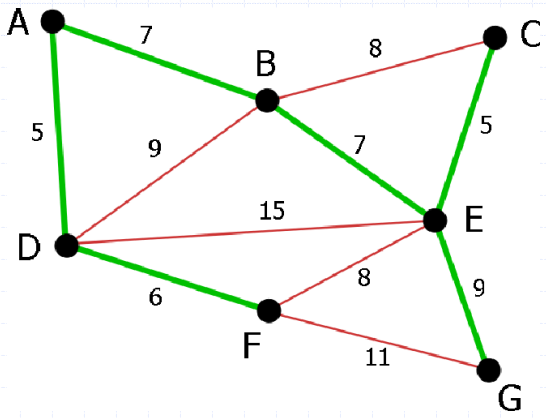
O algoritmo de Kruskal

- Seu funcionamento é mostrado a seguir:
  - crie uma floresta *F* onde cada vértice no grafo é uma árvore separada,
  - crie um conjunto *S* contendo todas as arestas do grafo,
  - enquanto *S* for não-vazio, faça:
    - remova uma aresta com peso mínimo de S,
    - se essa aresta conecta duas árvores diferentes, adicione-a à floresta, combinando duas árvores numa única árvore parcial,
    - caso contrário, descarte a aresta.
- Ao fim do algoritmo, a floresta tem apenas um componente e forma uma árvore geradora mínima do grafo.

O algoritmo de Kruskal

- Para um grafo com n vértices e m arestas, o algoritmo de Kruskal leva $O(m \log m)$ passos para ordenar as arestas pelo custo, $O(m)$ passos para decidir se a aresta deve ser adicionada (O(1) para cada aresta) e tempo $O(n^2)$ para manter uma tabela com as conexões vértice a vértice. Tempo total $O(n^2 + m \log m)$, que para grafos esparsos é $O(n^2)$.
- Esse tempo de execução é polinomial no "tamanho" da entrada, que corresponde à soma de n e m.

O algoritmo de Kruskal



Ordem de seleção das arestas: AD, CE, DF, AB, BE, EG.

Algoritmo de Dijkstra

- Um Algoritmo de Menor Caminho
  - Num grafo ponderado, ou rede, deseja-se achar o menor caminho entre dois vértices, *s* e *t*.
  - O menor caminho é definido como um caminho de *s* a *t* de modo que a soma dos pesos das arestas do caminho seja minimizada.
  - Para representar a rede, uma função **`peso(i,j)`**, que representa o peso da aresta de *i* a *j*.
  - Caso não haja aresta de *i* a *j*, **`peso(i,j)`** terá um valor grande para indicar custo infinito.
  - Se todos os pesos são positivos, o algoritmo de Dijkstra determinará o menor caminho de *s* até *t*.
  - Suponha que a variável **`infinito`** armazena o maior inteiro possível.

Algoritmo de Dijkstra

- **`distancia[i]`** = custo do menor caminho conhecido até então, de $s \rightarrow i$.
- Inicialmente, **`distancia[s]= 0`** e **`distancia[i] = infinito`**, $\forall\ i \neq s$.
- Um conjunto **`perm`** contém todos os vértices cuja distância mínima a partir de *s* é conhecida.
- Se $i \in$ **`perm`**, **`distancia[i]`** será a menor distância de $s \rightarrow i$.
- No início, **`perm`** = {*s*}.
- Quando $t \in$ **`perm`**, **`distancia[t]`** será a menor distância de $s \rightarrow t$ e o algoritmo terminará.
- O algoritmo mantém uma variável **`corrente`** que representa o vértice mais recente incluído em **`perm`**.

Algoritmo de Dijkstra

- No início, **`corrente = s`**.
- Sempre que um vértice corrente for incluído em **`perm`**, **`distancia`** deverá ser recalculada para todos os sucessores de corrente.
- Para todo sucessor i de corrente, se **`distancia[corrente] + peso[corrente,i] < distancia[i]`**, a distância de s → i por meio de **`corrente`** será menor que todas as outras distâncias de s → i encontradas até então.
- Sendo assim, **`distancia[i]`** terá de ser redefinida com esse valor menor.
- Assim que **`distancia`** for recalculada para todo sucessor de **`corrente`**, **`distancia[j]`** (∀ j),

Algoritmo de Dijkstra

representará o menor caminho de s → j que incluirá somente membros de **`perm`** (exceto o próprio j).

- Para o vértice k ∉ **`perm`**, para o qual **`distancia[k]`** é o menor, não existirá caminho de s → k cujo comprimento seja menor que **`distancia[k]`**.
- **`distancia[k]`** já é a menor distância até k que inclui somente os vértices ∈ **`perm`**, e ∀ caminho até k que inclua um vértice **`nd`** como seu primeiro vértice ∉ **`perm`** tem de ser mais longo porque **`distancia[nd] > distancia[k]`**.
- Assim, k pode ser incluído em **`perm`**. **`corrente`** é redefinida com k e o processo se repete.

Algoritmo de Dijkstra: Implementação

- Segue uma rotina em C que implementa o algoritmo de Dijkstra.
- Além de calcular distâncias, o programa encontra o menor caminho, mantendo um vetor **`precede`**, tal que **`precede[i]`** seja o vértice que precede o vértice i no menor caminho encontrado até então.
- Um vetor **`perm`** é usado para rastrear o conjunto correspondente.
- **`perm[i] = 1`**, se $i \in$ **`perm`**; **`perm[i] = 0`**, se $i \notin$ **`perm`**.

Algoritmo de Dijkstra: Implementação

- A rotina aceita uma matriz de pesos (com arestas não adjacentes com peso infinito) e dois vértices, *s* e *t*, e calcula a menor distância **`pd`** de s $\rightarrow$ t, além do vetor **`precede`** para definir o caminho.

```
void menorcaminho (int peso[ ][MAXVERT], int s,
   int t, int *pd, int precede[], int n)
{
   int distancia[MAXVERT], perm[MAXVERT];
   int corrente, i, k, dc;
   int menordist, novadist;

   // iniciação
   for (i = 0; i < n; ++i)
   {
       perm[i] = NAOMEMBRO; // NAOMEMBRO é 0
       distancia[i] = INFINITO; // INFINITO é 1000
   }

   perm[s] = MEMBRO; // MEMBRO é 1
   distancia[s] = 0;
   corrente = s;
```


Algoritmo de Dijkstra: Implementação

```
while (corrente != t)
{
   menordist = INFINITO;
   dc = distancia[corrente];
   for (i = 0; i < n; i++)
   if (perm[i] == NAOMEMBRO)
   {
       novadist = dc + peso[corrente][i];
       if (novadist < distancia[i])
       {
   // a distância de s a i através de corrente <
   // distância[i]
             distancia[i] = novadist;
             precede[i] = corrente;
       }
```

Algoritmo de Dijkstra: Implementação

```
    // determina a menor distância
        if (distancia[i] < menordist)
        {
              menordist = distancia[i];
              k = i;
        }
    }
    corrente = k;
    perm[corrente] = MEMBRO;
  }

  *pd = distancia[t];
}
```


Algoritmo de Dijkstra: Eficiência

- Para analisar a eficiência do algoritmo de Dijkstra, observe que um vértice é incluído em **`perm`** em cada iteração do **`while`**.
- Ou seja, repete n vezes, onde n = **MAXVERT**.
- Cada iteração requer o exame de todo vértice.
- Logo, o algoritmo é $O(n^2)$.

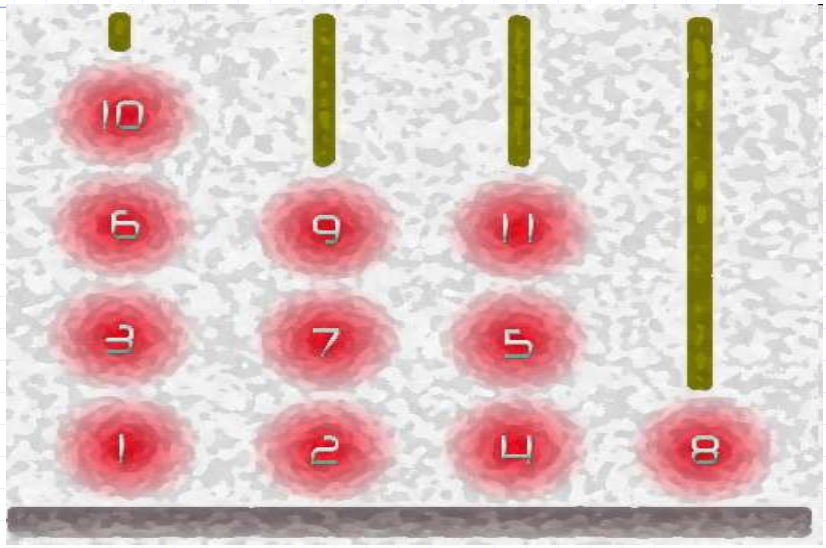
Exercícios para Nota

- Hanoi Tower Troubles Again! (10276)
- Museums (11153)

Hanoi Tower Troubles Again! (10276)

- People stopped moving discs from peg to peg after they know the number of steps needed to complete the entire task. But on the other hand, they didn't not stopped thinking about similar puzzles with the Hanoi Tower. Mr.S invented a little game on it. The game consists of N pegs and a LOT of balls. The balls are numbered 1,2,3... The balls look ordinary, but they are actually magic. If the sum of the numbers on two balls is NOT a square number, they will push each other with a great force when they're too closed, so they can NEVER be put together touching each other.

Hanoi Tower Troubles Again! (10276)



Hanoi Tower Troubles Again! (10276)

- The player should place one ball on the top of a peg at a time. He should first try ball 1, then ball 2, then ball 3... If he fails to do so, the game ends. Help the player to place as many balls as possible. You may take a look at the picture above, since it shows us a best result for 4 pegs.
- Input
  - The first line of the input contains a single integer T, indicating the number of test cases. (1<=T<=50) Each test case contains a single integer N(1<=N<=50), indicating the number of pegs available.

Hanoi Tower Troubles Again! (10276)

- Output
  - For each test case in the input print a line containing an integer indicating the maximal number of balls that can be placed. Print -1 if an infinite number of balls can be placed.
- Sample Input

  2
  4
  25

- Sample Output

  11
  337

Laboratório de Algoritmos Avançados – Capítulo 9

Museums (11153)

- You are on holiday, and being a culture lover, you decide to visit some museums this afternoon. You have the *Museums Guide Book* on hand, in which details of all museums are included. To each museum you have also assigned a "fun index", ranging from 1 to 9, describing how interesting it is.
- Everything seems fine except that you have a pretty low budget, as you need to save up money to buy manga books. What's worse, the time you can use is also limited, thus you may not be able to visit all museums. As a result, you will need to plan your trip carefully such that the "fun" you can get is maximized, i.e. the total "fun index" of all places you visit is the greatest.

Museums (11153)

- Here are some guidelines for your planning. Firstly, since you will travel in your car, you will assume the transport cost to be **$0**. Secondly, you will need to spend at least **15 minutes** at the museum for each visit - you've really got to look at the exhibits! Thirdly, you should not visit any museum more than once, although you may *travel past* it many times. Last but not least, your trip should start and end both at your home.
- You see that planning the trip is a real torture to your mind, so you need to **write a program** that helps you with it. Given the details of all museums and roads as well as the restrictions on money and time, your program should find the maximum "fun" you can get.

Laboratório de Algoritmos Avançados – Capítulo 9

Museums (11153)

- Input
  - Input consists of several test cases. The first line of the input file gives the number of cases.
  - The first line of each case gives four positive integers **d**, **t**, **n** and **m**. **d** is the amount of money you can spend (in dollars), which will not exceed **100**. **t** is the time limit for your trip (in minutes), which can be up to **600**. **n** gives the number of museums in your list, which will always be less than **13**. Finally, **m** is the number of roads, which has a upper limit of **n** (**n**+1) / 2.
  - Next comes **n** lines, each with two integers specifying the admission fee (in dollars, not greater than **50**) and "fun index" of a museum.

Museums (11153)

- Each of the following **m** lines of the input has three integers **i**, **j** and **k**, which means that a bidirectional road connects nodes **i** and **j**, and it takes **k** (≤ **200**) minutes for your car to go through it. In this problem we define **node 0** as the position of your home, and **node i** as the **i**-th museum in your above list. Please note that some (but not all) museums might NOT be reachable from your home at all, so be careful. Moreover, no roads will connect a node to itself, but there might be more than one road between two nodes.

Output

- For each test case, your program should output the maximum fun you can get. If you cannot visit any museum at all, output "**No possible trip.**" instead.

Laboratório de Algoritmos Avançados – Capítulo 9

Museums (11153)

Sample Input

```
2
50 120 4 4
15 8
5 4
0 2
15 5
0 1 3
0 4 8
1 3 5
2 4 70
50 2 4 4
15 8
5 4
0 2
15 5
0 1 3
0 4 8
1 3 5
2 4 70
```

Sample Output

```
Case 1: 15
Case 2: No possible trip.
```

Referências

[1] Batista, G. & Campello, R.
Slides *Algoritmos Avançados*, ICMC-USP, 2007.

[2] Goodrich, M. T. & Tamassia, R. & Mount, D.
http://ww3.datastructures.net

[3] Halim, S. & Halim, F.
Competitive Programming, 2010.

[4] Sedgewick, R.
Algorithms in C – Part 5 – Graph Algorithms – Third Edition, Addison-Wesley, 2002.

[5] Skiena, S. S. & Revilla, M. A.
Programming Challenges – The Programming Contest Training Manual. Springer, 2003.