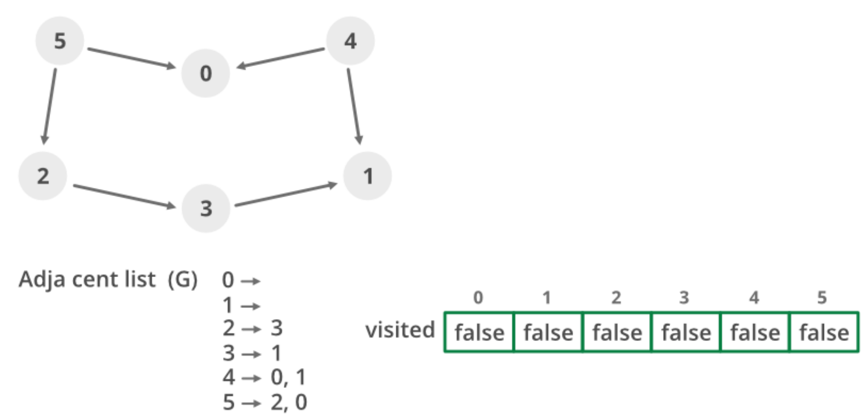
Grafos – Parte 2

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação

Relembrando... **DFS!**

```
void dfs(int u)
{
  if(visited[u]) return;
  visited[u] = true;

  for(int v : adj[u])
    dfs(v);
}

void dfs_explore()
{
  for(int i = 0; i < VERT; i++)
    if(!visited[i]) dfs(i);
}
```

```
adj[2].push_back(3);
adj[3].push_back(1);
adj[4].push_back(0);
adj[4].push_back(1);
adj[5].push_back(0);
adj[5].push_back(2);

dfs_explore();
```



O DFS pode ser executado **sem modificação** para grafos direcionados, tomando cuidado para percorrer as arestas apenas nas direções definidas.

Grafos direcionados acíclicos (**DAG**)

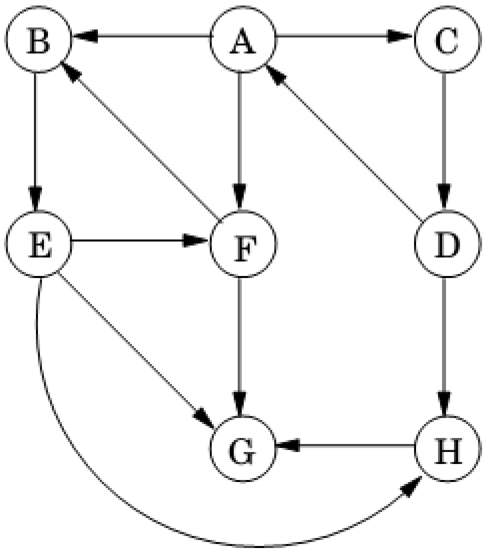
Um ciclo em um grafo direcionado é um caminho circular
$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_0$.

A figura abaixo possui alguns exemplos: (B, E, F, B) ou (A, C, D, A).

Um grafo sem ciclos é dito acíclico.

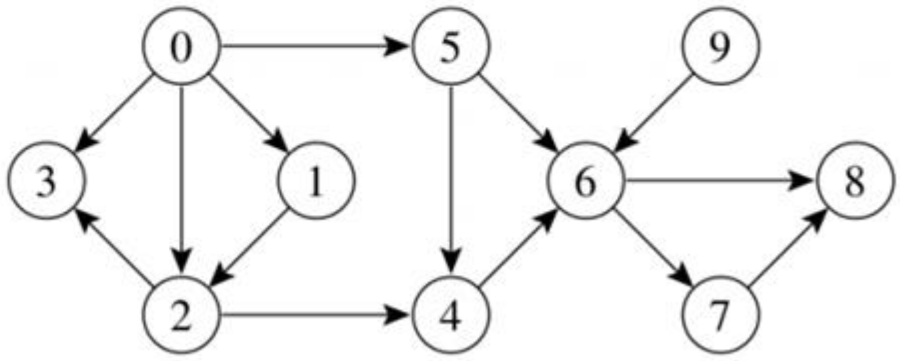
Grafos Direcionados Acíclicos (ou *dags*, do inglês *directed acyclic graphs*) aparecem o tempo todo e são bons para modelar relações de causalidades, hierarquias e dependências temporais.

É possível definir se um grafo é acíclico em tempo linear, utilizando o algoritmo de busca em profundidade. Pesquise sobre isso.



Grafos direcionados acíclicos (DAGs)

- *Dags* são muito utilizados para indicar relações de precedência entre eventos.
  - Imagine que uma pessoa precise realizar uma grande quantidade de tarefas, mas algumas delas **não podem ser feitas até que outras sejam completadas**.
  - Como exemplo, temos os **pré-requisitos** de um curso de graduação: Para fazer "estrutura de dados" é necessário haver completado com sucesso "fundamentos de programação"
- Uma aresta direcionada (u,v) indica que uma atividade u precisa ser realizada antes da atividade v.



Ordenação **Topológica**

- Como encontrar uma ordem válida para realizar as tarefas?
- Utilizando Grafos direcionados acíclicos e o algoritmo de Ordenação Topológica.

Ordenação topológica é uma ordenação linear dos vértices do grafo, de forma que, para cada aresta direcionada (u, v), u vem antes de v na ordenação (u precisa ser realizado antes de v).

O **algoritmo de ordenação topológica** consiste em:

- **Chamar o algoritmo de DFS.**
- Ao término de cada vértice, **insira-o no topo de uma pilha**.
- Imprima os vértices já ordenados topologicamente, **retirando do topo da pilha**.

Ordenação **Topológica**

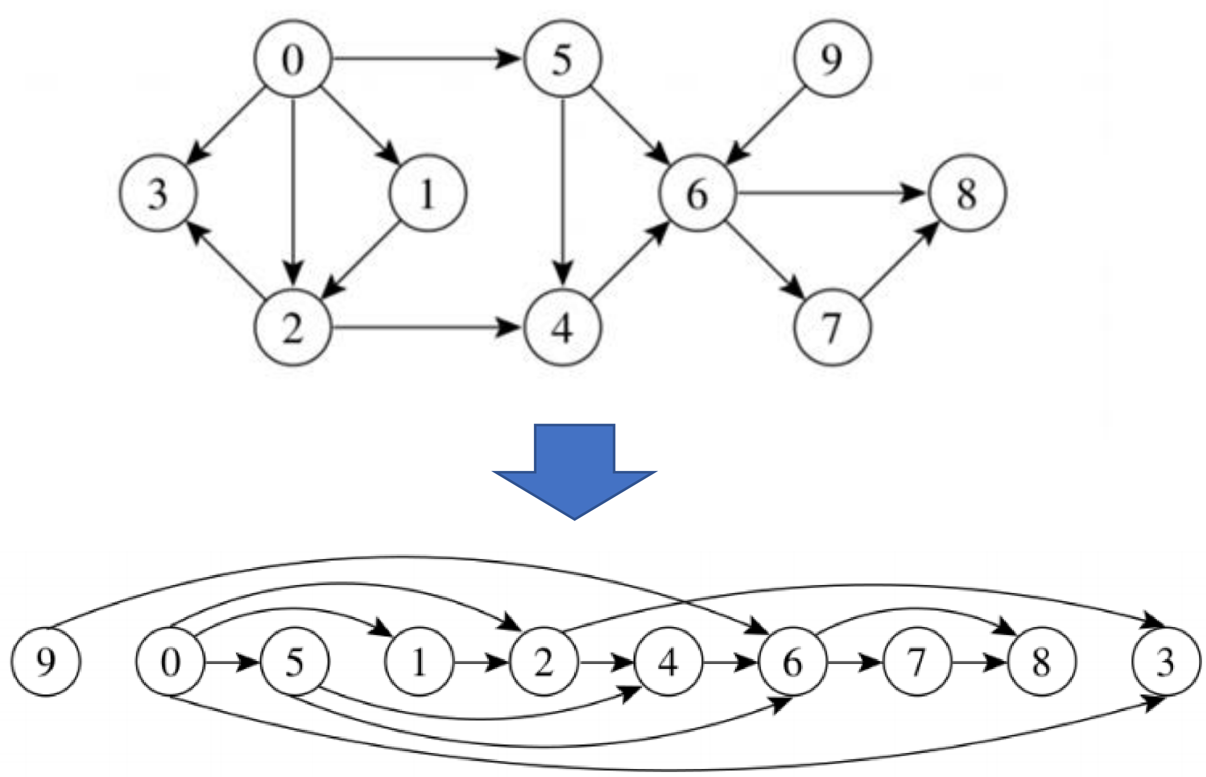
O custo é novamente **linear** **O (|V| + |E|)** , uma vez que a busca em profundidade tem essa complexidade de tempo e o custo para inserir cada um dos |V| vértices na frente da pilha custa O (1).

Basta realizar uma chamada para o procedimento de **empilhamento (push)** no procedimento **dfs**, no momento em que o vértice passa a ser totalmente explorado (após o for, depois das chamadas recursivas). Imprima a pilha ao final da execução, **desempilhando** item a item.

Grafos direcionados acíclicos (DAGs)

Exercício:

Modifique o algoritmo de DFS para realizar a ordenação topológica de um grafo acíclico. Utilize o grafo da figura como exemplo:



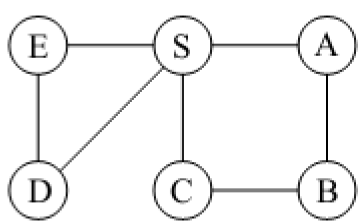
Busca em Profundidade:

- Identifica todos os vértices do grafo;
- Encontra caminhos específicos para esses vértices.

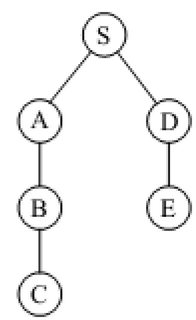
Entretanto, os caminhos podem não ser os mais econômicos possíveis (Exemplo: S → C).

- A distância entre dois vértices é o tamanho do caminho mais próximo entre eles.

(a)

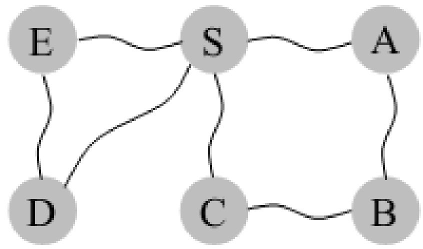


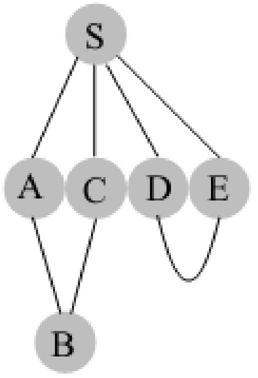
(b)



Imagine o grafo representado por bolas (vértices) e linhas (arestas).

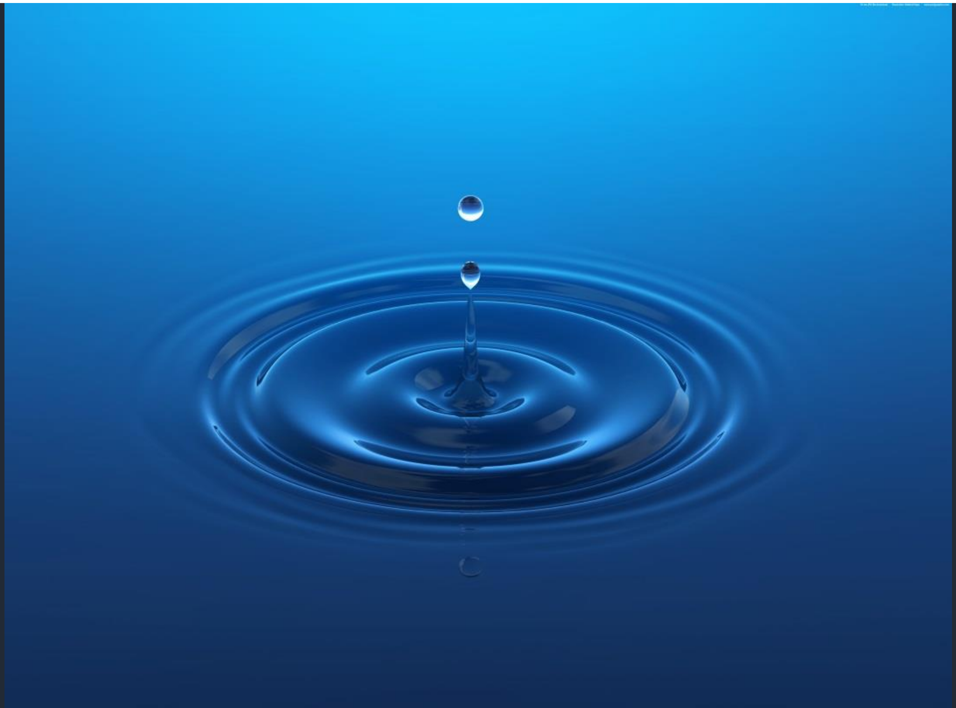
- Se você elevar a bola S alto o suficiente, as bolas que se elevarem junto são os vértices alcançáveis por S.
- As distâncias entre S e as outras bolas são facilmente calculadas (número de camadas)
  - Exemplo: A distância entre S e B é 2.



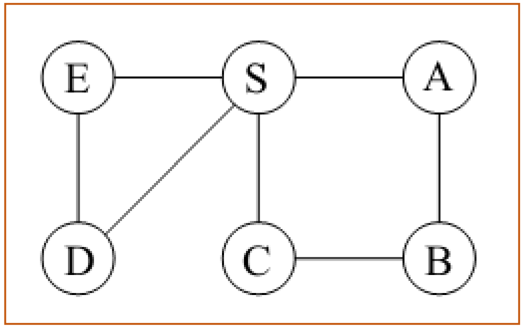


O algoritmo de Busca em Largura (BFS, do inglês *"Breadth first search"*), nos ajuda a encontrar menor caminho entre vértices.

- Descobre todos os vértices a uma distância k do vértice de origem antes de descobrir qualquer vértice a uma distância k+1.
- O grafo pode ser direcionado ou não-direcionado.

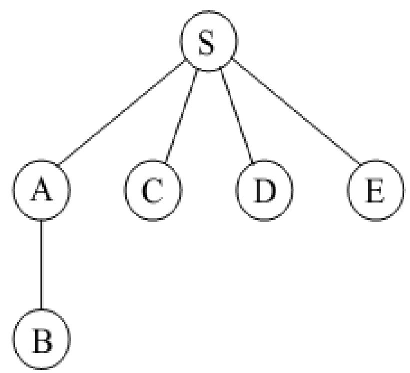


Exemplo: Considere o grafo do exemplo anterior:



A árvore de busca em largura (abaixo, à direita) contém as arestas pelas quais cada nó é descoberto. Todos os caminhos a partir de S são os menores possíveis e ela é chamada de **Árvore de Caminho Mínimo**.

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



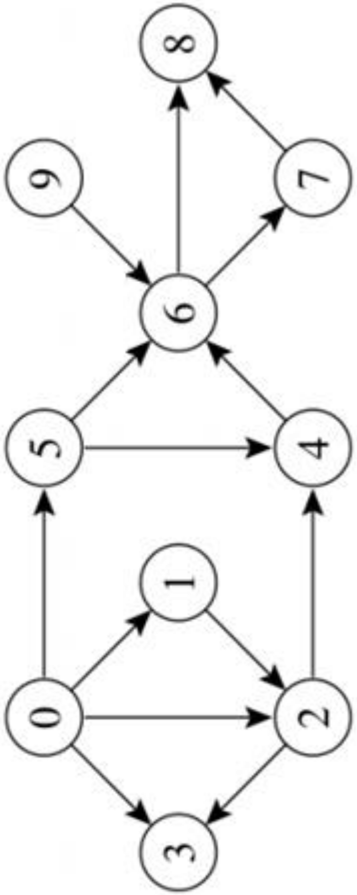
Profundidade x Largura

- A busca em profundidade faz incursões profundas no grafo e somente retorna pelo caminho que percorreu apenas quando não consegue nós mais profundos para visitar.
    - Usa uma **pilha** na sua implementação (recursão).

- A busca em largura visita os vértices em ordem crescente de suas distâncias, de maneira parecida com a propagação de uma onda na água.
    - Usa uma **fila** em sua implementação (queue).

```cpp
#include <iostream>
#include <vector>
#include <queue>

#define VERT 10

using namespace std;

vector<int> adj[1000];
vector<bool> visited(1000, false);
queue<int> fila;

void bfs(int u)
{
  fila.push(u);
  visited[u] = true;

  while(!fila.empty()) {
    int v = fila.front();
    fila.pop();

    for(int v : adj[u]) {
      if(!visited[v]) {
        fila.push(v);
        visited[v] = true;
      }
    }
  }
}
```

```cpp
void bfs_explore()
{
  for(int i = 0; i < VERT; i++)
    if(!visited[i]) bfs(i);
}

int main()
{
  adj[0].push_back(1);
  adj[0].push_back(2);
  adj[0].push_back(3);
  adj[0].push_back(5);
  adj[1].push_back(2);
  adj[2].push_back(3);
  adj[2].push_back(4);
  adj[4].push_back(6);
  adj[5].push_back(4);
  adj[5].push_back(6);
  adj[6].push_back(7);
  adj[6].push_back(8);
  adj[7].push_back(8);
  adj[9].push_back(6);

  bfs_explore();
}
```



Caminhos em Grafos

Pesos nas arestas

- Busca em Largura = Arestas de mesmo comprimento ou mesmo peso.
- Acontece raramente...

Exemplo:

Um motorista procura o caminho mais curto entre Itajubá e Uberlândia. Para isso, ele possui um mapa com as distâncias entre cada par de interseções adjacentes.

Peso de um caminho:

- Soma dos pesos de cada aresta que o motorista percorre em seu caminho.

Caminhos em Grafos

Os valores dos pesos não precisam corresponder sempre a distâncias ou comprimentos físicos. No exemplo anterior, poderíamos colocar como valores de peso:

- O tempo gasto para percorrer o caminho entre duas cidades.
- O valor gasto com pedágio em cada uma das estradas.
- A preferência do motorista de trafegar ou não por uma via, em um número entre 0 a 10.
- Condições da via, com notas entre 0 e 10. etc.

Caminho **Mínimo** (Shortest Path)

Caminhos mais curtos a partir de uma origem (SSSP): dado um grafo com pesos, obter o caminho mais curto a partir de um dado vértice origem até cada um dos vértices.

Muitos problemas podem ser resolvidos fazendo apenas pequenas modificações no algoritmo de origem única:

- Caminhos mais curtos com destino único: encontra caminho mínimo de todos os vértices até um dado vértice. Reduzido ao problema de origem única se invertermos a direção de cada uma das arestas do grafo.
- Caminhos mais curtos entre um par de vértices: algoritmo para origem única é a melhor opção conhecida.
- Caminhos mais curtos entre todos os pares de vértices: resolvido aplicando o algoritmo de origem única $|V|$ vezes, uma para cada vértice origem.

Caminho **Mínimo** (Shortest Path)

Dijkstra (1959) apresentou um algoritmo para resolver om problema de SSSP.

- Funciona apenas para grafos com custos positivos nas arestas.

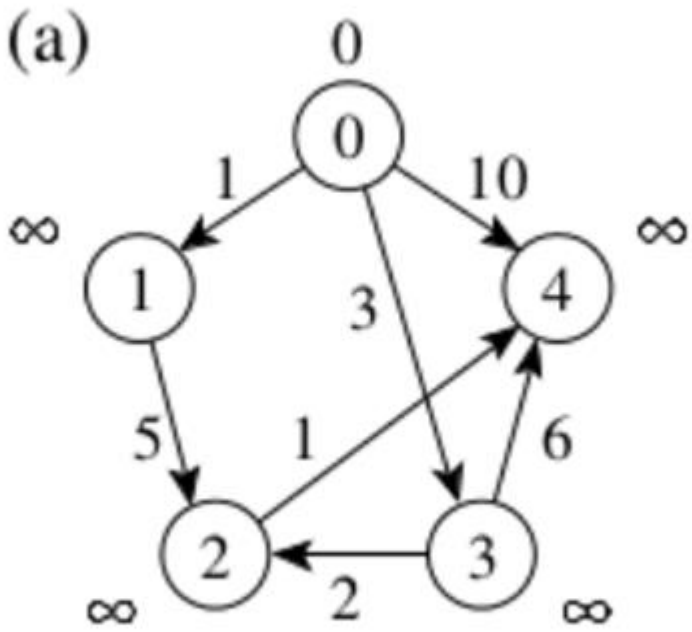
O algoritmo mantém a informação sobre o caminho mínimo entre s e v em um vetor chamado anterior[u].

- Contém um vértice que é anterior a ele no caminho mínimo entre s e v.
- Para encontrar o caminho mínimo basta voltar passo a passo até que anterior seja igual a -1 (vértice fonte).

A informação sobre o custo é mantida em outro vetor, chamado custo[v].

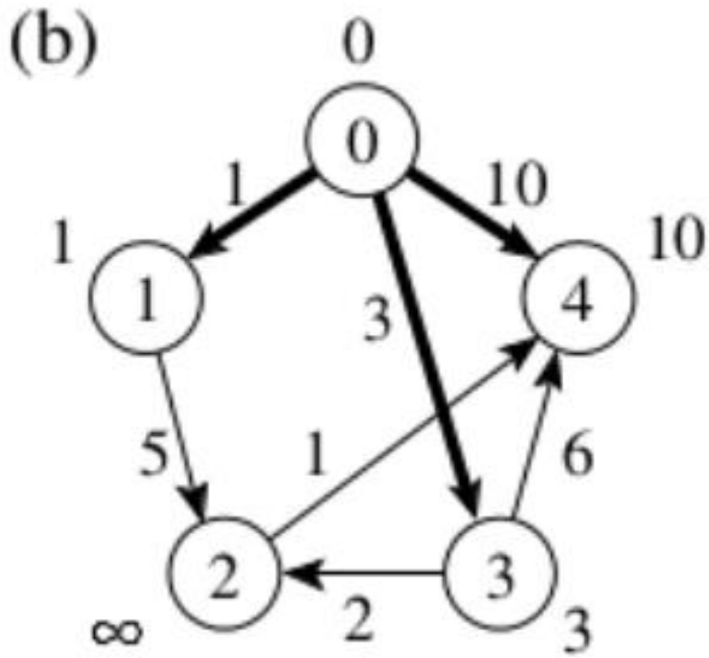
- Para saber o custo do caminho mínimo entre s e v, basta fazer a leitura do vetor na posição v.

Simulação Dijkstra



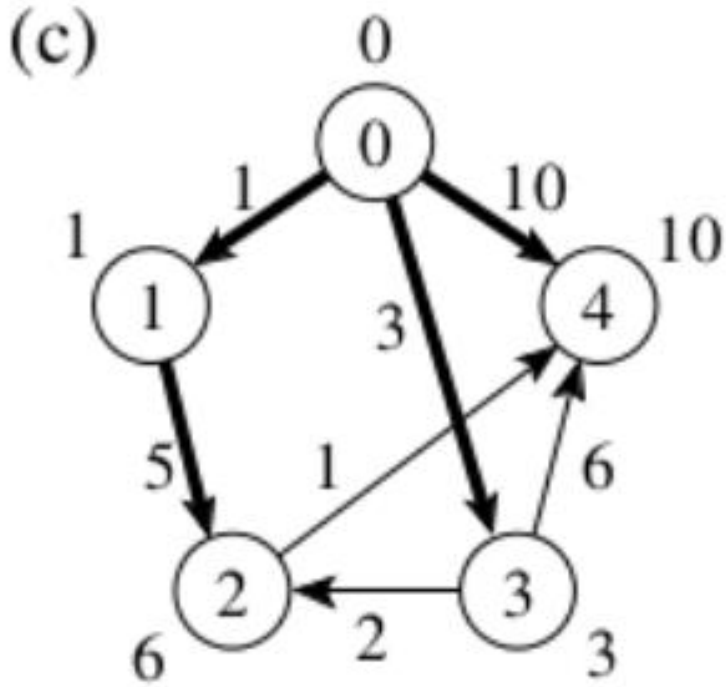
Vértice	0	1	2	3	4
Custo	0	∞	∞	∞	∞
Anterior	-1	-1	-1	-1	-1

Simulação Dijkstra



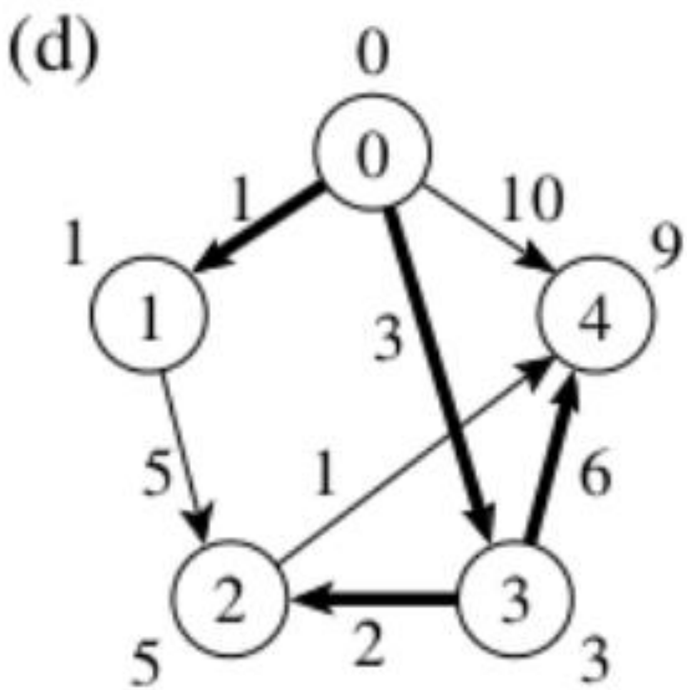
Vértice	0	1	2	3	4
Custo	0	1	∞	3	10
Anterior	-1	0	-1	0	0

Simulação Dijkstra



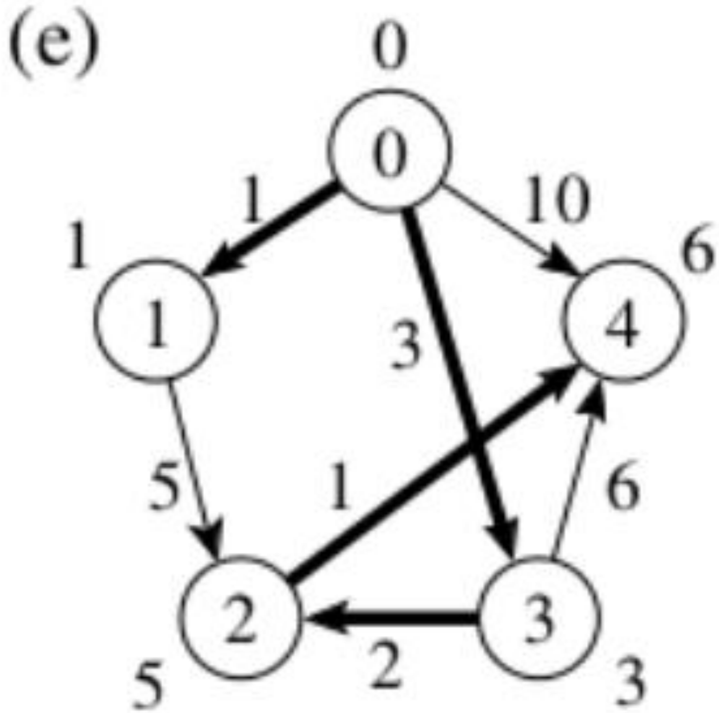
Vértice	0	1	2	3	4
Custo	0	1	6	3	10
Anterior	-1	0	1	0	0

Simulação Dijkstra



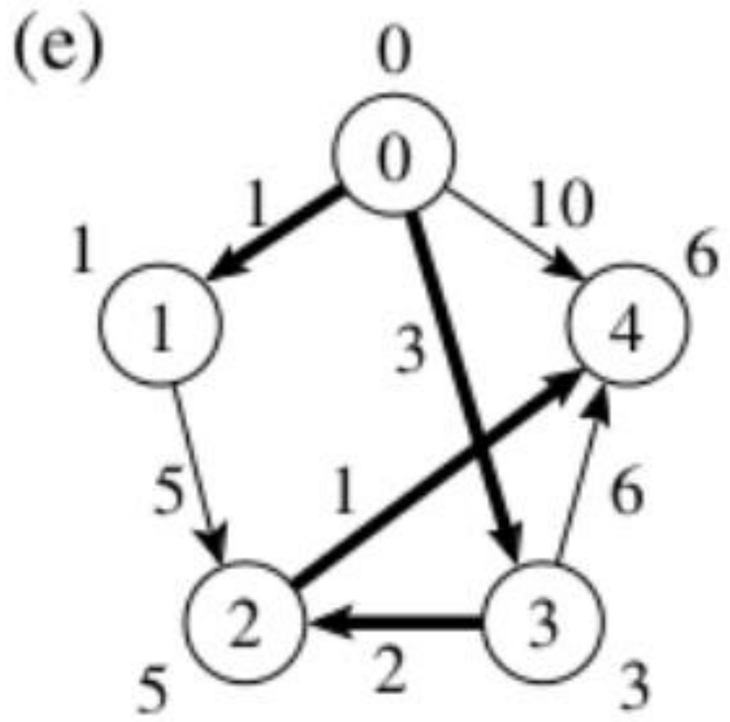
Vértice	0	1	2	3	4
Custo	0	1	5	3	9
Anterior	-1	0	3	0	3

Simulação Dijkstra



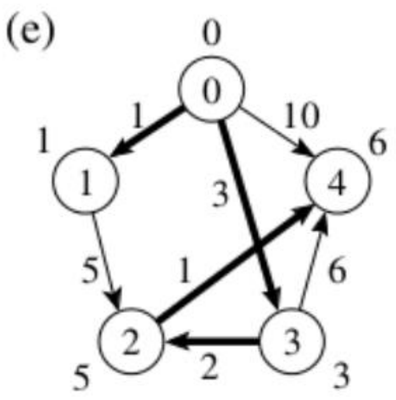
Vértice	0	1	2	3	4
Custo	0	1	5	3	6
Anterior	-1	0	3	0	2

Simulação Dijkstra



Vértice	0	1	2	3	4
Custo	0	1	5	3	6
Anterior	-1	0	3	0	2

Simulação Dijkstra



Vértice	0	1	2	3	4
Custo	0	1	5	3	6
Anterior	-1	0	3	0	2

Qual é o caminho mais curto entre os nós 0 e 4?

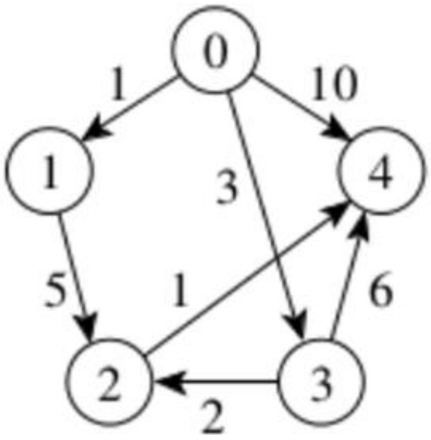
O custo mínimo é retirado diretamente do vetor custo, na posição 4 = 6.

O caminho mínimo é conseguido através do vetor anterior, e deve ser feito de trás para frente: 4 – 2 – 3 – 0

O caminho mínimo entre 0 e 4 é: 0 → 3 → 2 → 4

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <utility> // pair

using namespace std;

// Os nomes declarados com typedef são apelidos para
// tipos existentes, e não declarações de novos tipos.
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
#define INF 1000000000 // valor grande o suficiente (Infinito)
#define VERT 5 // quantidade de vértices

vii adj[VERT]; // lista de adjacencias
vi custo(VERT, INF); // vetor de custo
vi anterior(VERT, -1); // vertices anteriores
```

```cpp
void dijkstra(int s) {
  custo[s] = 0;
  priority_queue<ii, vii, greater<ii>> pq; // heap de mínimo
  pq.push(make_pair(0, s)); // par (first=custo, second=anterior)

  while(!pq.empty()) {
    ii front = pq.top(); pq.pop(); // pega elemento do topo (menor custo)
    int c = front.first, u = front.second; // pego c=custo e u=vertice anterior
    if(c > custo[u]) continue; // se c > custo atual, sai do loop
    for(ii v : adj[u]) { // pra cada vertice v adjacente a u (u -> v)
      if(custo[u] + v.second < custo[v.first]) {
        custo[v.first] = custo[u] + v.second;
        pq.push(make_pair(custo[v.first], v.first));
        anterior[v.first] = u;
      }
    }
  }
}
```




```
C:\Users\João Paulo\Deskto
Custos minimos:
0:0  1:1  2:5  3:3  4:6

Vertices anteriores:
0:-1  1:0  2:3  3:0  4:2
```

```cpp
int main() {
  adj[0].push_back(make_pair(1, 1)); // (vértice, peso)
  adj[0].push_back(make_pair(4, 10));
  adj[0].push_back(make_pair(3, 3));
  adj[1].push_back(make_pair(2, 5));
  adj[2].push_back(make_pair(4, 1));
  adj[3].push_back(make_pair(2, 2));
  adj[3].push_back(make_pair(4, 6));

  dijkstra(0);

  cout << "Custos minimos:\n";
  for(int i = 0; i < VERT; i++) {
    cout << i << ":" << custo[i] << "  ";
  }
  cout << "\n\nVertices anteriores:\n";
  for(int i = 0; i < VERT; i++) {
    cout << i << ":" << anterior[i] << "  ";
  }
  cout << endl;
}
```

Caminho Mínimo (Shortest Path)

Qual a complexidade do Algoritmo?

Para que todos os vértices sejam retirados da *heap (priority_queue)*, ou seja, para que seja encontrado o caminho mínimo para todos os vértices, são necessárias |V| iterações do laço enquanto.

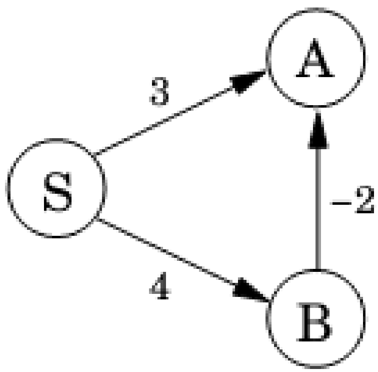
- Uma operação para remover o vértice de menor custo da *heap*, toma no máximo O(log|V|) comparações.
- A atualização do valor do custo, no último laço para também pode envolver a atualização de O(|E|log|V|) vértices.

Portanto, a complexidade do algoritmo, seria O((E + V) log V).

Caminho Mínimo (Shortest Path)

O algoritmo de Dijkstra usa uma estratégia gulosa: ele sempre escolhe o vértice de menor custo e o retira da heap, considerando que o caminho mínimo até ele já tenha sido encontrado.

Exatamente devido a essa característica, ele não irá funcionar corretamente para grafos com arestas que possuam pesos negativos. Repare no exemplo:



- **Dijkstra escolheria o caminho direto entre S e A, pois 3 é menor do que 4 (guloso, menor custo)**
- **No entanto, o menor caminho é passando por B, com custo total igual a 2.**

Teremos que usar um **algoritmo alternativo**!

Caminho Mínimo (Shortest Path)

O algoritmo de Bellman-Ford também encontra o menor caminho entre um vértice fonte S e todos os demais vértices de um grafo.

- Abre mão da possibilidade de fechar um vértice a cada iteração (retirá-lo da heap) e se obriga a examinar todos os vértices até que melhorias não sejam mais possíveis (não-guloso).
  - Complexidade aumenta ou diminui?

Capaz de encontrar caminhos mínimos em grafos com arestas negativas, pois verifica todas as possibilidades.

Caminho Mínimo (Shortest Path)

```
vii adj[100];
vi dist(100, INF);

void bellman_ford(int n, int s)
{
    dist[s] = 0;

    for(int i = 0; i < n-1; i++) // for O(|V|)
        for(int u = 0; u < n; u++) // dois fors em O(|E|)
            for(int j = 0; j < (int)adj[u].size(); j++) {
                ii v = adj[u][j];
                dist[v.first] = min(dist[v.first], dist[u] + v.second);
            }
}
```

Caminho Mínimo (Shortest Path)

O algoritmo de Bellman-Ford depende diretamente da quantidade de vértices e de arestas, tomando um tempo $O(|V|.|E|)$

- No pior caso, em grafos densos, com aproximadamente $|V|^2$ arestas, sua complexidade é cúbica.
- Portanto, o algoritmo de Bellman-Ford possui complexidade $O(n^3)$.

Hands-on!

Vamos resolver um problema básico de Dijkstra.

Uva 10986 – *Sending e-mail*...

There are n SMTP servers connected by network cables. Each of the m cables connects two computers and has a certain latency measured in milliseconds required to send an email message. What is the shortest time required to send a message from server S to server T along a sequence of cables? Assume that there is no delay incurred at any of the servers.

Input

The first line of input gives the number of cases, N. N test cases follow. Each one starts with a line containing n ($2 \leq n \leq 20000$), m ($0 \leq m \leq 50000$), S ($0 \leq S < n$) and T ($0 \leq T < n$). $S \neq T$. The next m lines will each contain 3 integers: 2 different servers (in the range $[0, n-1]$) that are connected by a bidirectional cable and the latency, w, along this cable ($0 \leq w \leq 10000$).

Output

For each test case, output the line '`Case #`x`:`' followed by the number of milliseconds required to send a message from S to T. Print '`unreachable`' if there is no route from S to T.

Sample Input

```
3
2 1 0 1
0 1 100
3 3 2 0
0 1 100
0 2 200
1 2 50
2 0 0 1
```

Sample Output

```
Case #1: 100
Case #2: 150
Case #3: unreachable
```