maratona de programação

UNIFEI

# 7 – Grafos II

## Alguns Algoritmos Clássicos

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação
sites.google.com/site/unifeimaratona/

# Veremos alguns algoritmos:

1) Caminhos Mínimos
    1) BFS
    2) Dijkstra
    3) Bellman-Ford
    4) Floyd-Warshall
2) Árvore Geradora Mínima
    1) Prim
    2) Kruskal
3) Fluxo
    1) Ford-Fulkerson
    2) Edmond-Karp

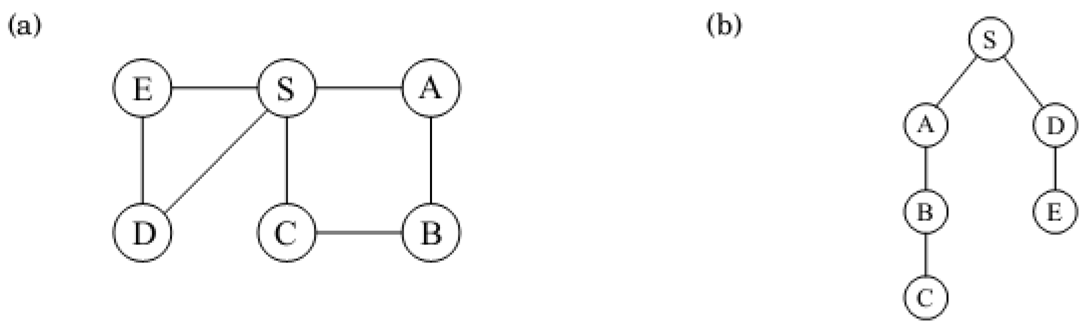
* Seções para auto-estudo, com exercícios relacionados na tarefa do *URI Academic*. Fique atento para o prazo de entrega.

Busca em Profundidade:

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

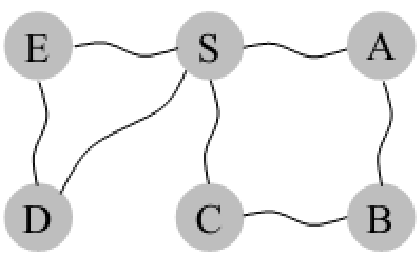
Entretanto, os caminhos podem não ser os mais econômicos possíveis (Exemplo: S → C).

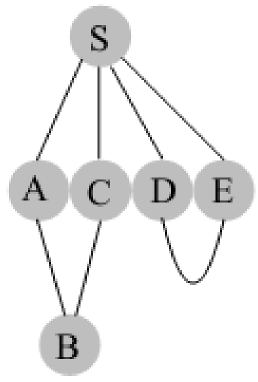
- A distância entre dois vértices é o tamanho do caminho mais próximo entre eles.



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.



Repare que a elevação de S divide o grafo em camadas, onde a distância de S para os vértices da primeira camada é 1, para os da segunda camada a distância é 2, e assim por diante.

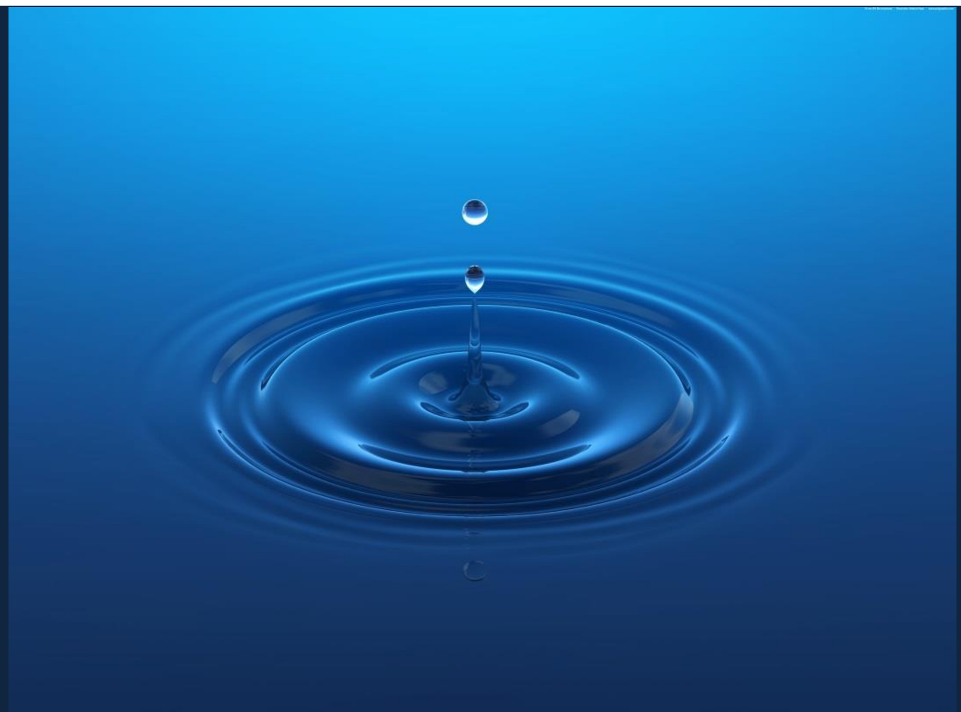
Isso sugere um algoritmo iterativo que percorra o grafo camada por camada, identificando a distância

- Diferente da DFS, que percorre o grafo em profundidade e ignora as camadas.

Busca em Largura!

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

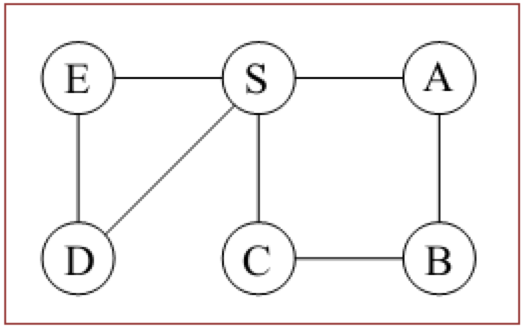
- Descobre todos os vértices a uma distância k do vértice de origem antes de descobrir qualquer vértice a uma distância k+1.
- O grafo pode ser direcionado ou não-direcionado.



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

queue<int> Q;
Q.push(start);
visited[start] = true;

while (!Q.empty()) {
    int u = Q.front(); Q.pop();

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (!visited[v]) {
            Q.push(v);
            visited[v] = true;
        }
    }
}
```

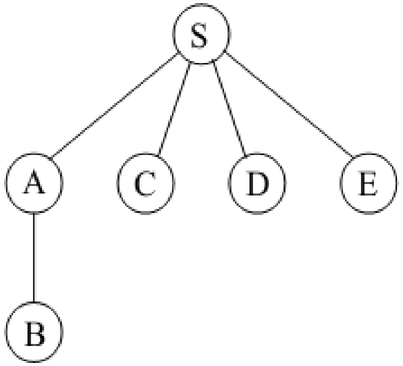
# Busca em Largura

## 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 |
|---|---|
| | $[S]$ |
| $S$ | $[A\ C\ D\ E]$ |
| $A$ | $[C\ D\ E\ B]$ |
| $C$ | $[D\ E\ B]$ |
| $D$ | $[E\ B]$ |
| $E$ | $[B]$ |
| $B$ | $[\,]$ |



# Busca em Largura

Profundidade x Largura

- A busca em profundidade faz incursões profundas no grafo e somente retorna 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).

# 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 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.
  - Assim poderíamos depois encontrar um caminho com tempo mínimo.
- O valor gasto com pedágio em cada uma das estradas.
  - Encontrando os caminhos mais baratos.

# Caminho Mínimo (Shortest Path)

Caminhos mais curtos a partir de uma origem (SSSP): dado um grafo ponderado, desejamos 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 aresta 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.

- Somente 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).

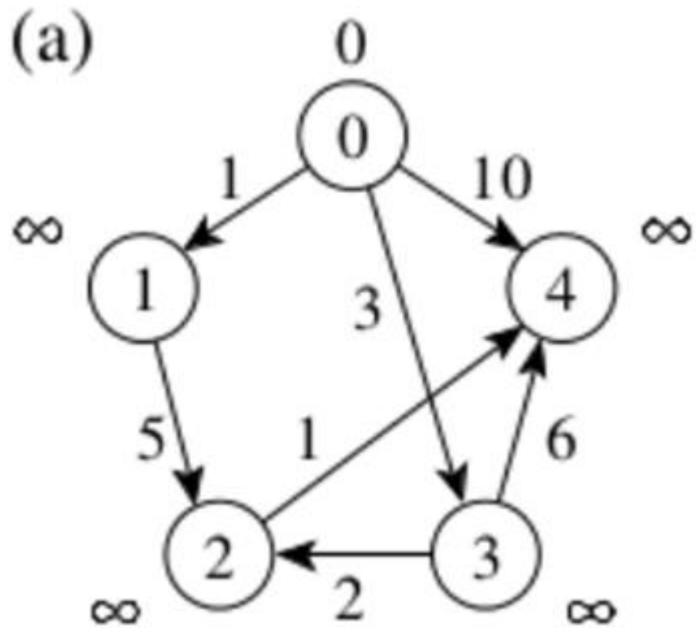
A informação sobre o custo é mantida em outro vetor, chamado custo[v].

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

# Caminho Mínimo (Shortest Path)

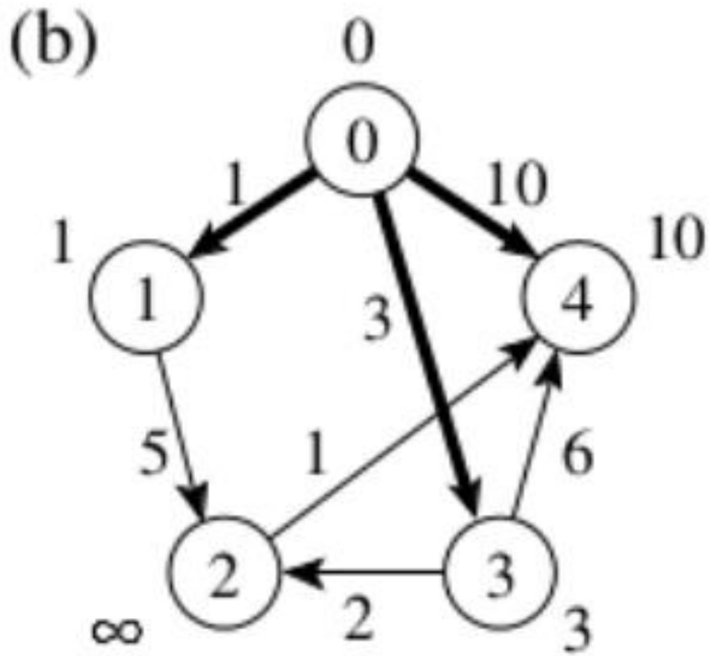
```
Dijkstra(G, s)

Para todo u ϵ V
    custo[u] = ∞ // inicialmente não se sabe o custo para nenhum vértice
    anterior[u] = -1 // não conhecemos ainda nenhum caminho

custo[s] = 0 // o custo de s para ele mesmo é 0!
H = ConstroiHeap(V) // inicialmente com todos os vértices

enquanto H for não-vazio // enquanto não houver caminho mínimo para todos
    u = RetiraMin(H) // vértice u com menor custo estimado até agora
    Para cada aresta (u,v) ϵ E
        se custo[v] > custo[u] + peso_aresta(u,v)
            custo[v] = custo[u] + peso_aresta(u,v)
            anterior[v] = u
            DiminuiChave(H, u) // Retira vértice u da heap (fechado)
```

# Simulação Dijkstra



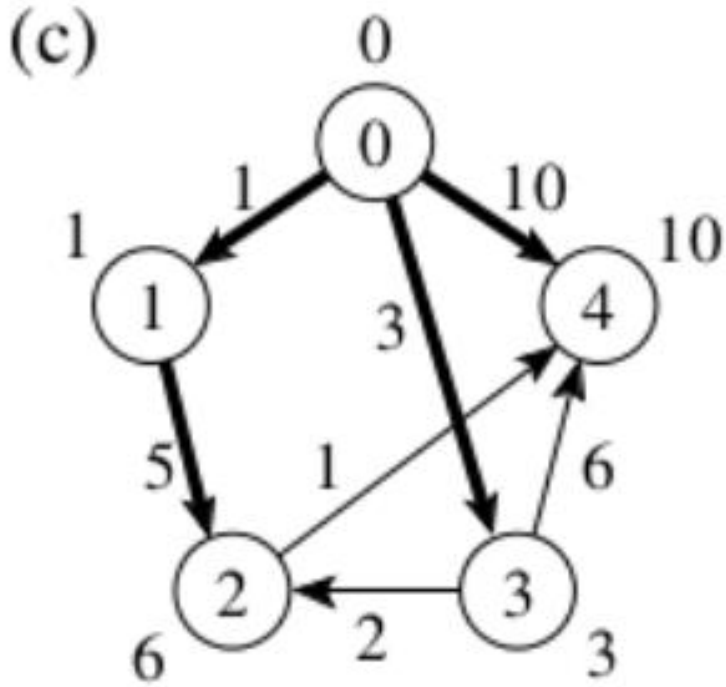

| Vértice | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Custo | 0 | ∞ | ∞ | ∞ | ∞ |
| Anterior | -1 | -1 | -1 | -1 | -1 |

# Simulação Dijkstra



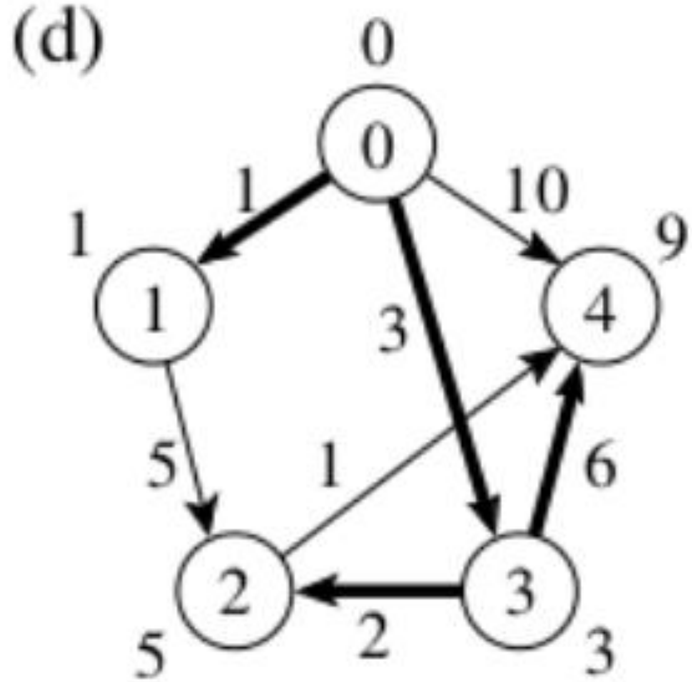

| Vértice | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Custo | 0 | 1 | ∞ | 3 | 10 |
| Anterior | -1 | 0 | -1 | 0 | 0 |

# Simulação Dijkstra



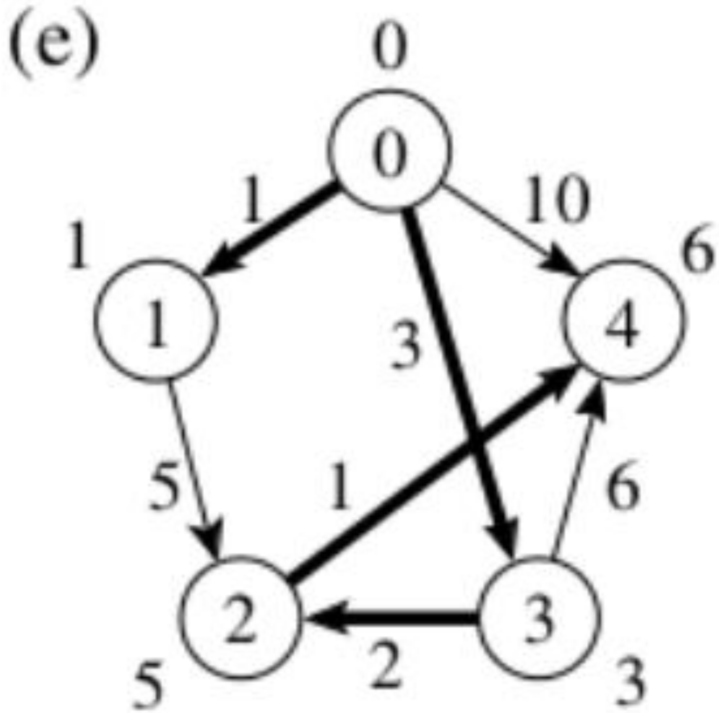

| Vértice | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Custo | 0 | 1 | 6 | 3 | 10 |
| Anterior | -1 | 0 | 1 | 0 | 0 |

# Simulação Dijkstra



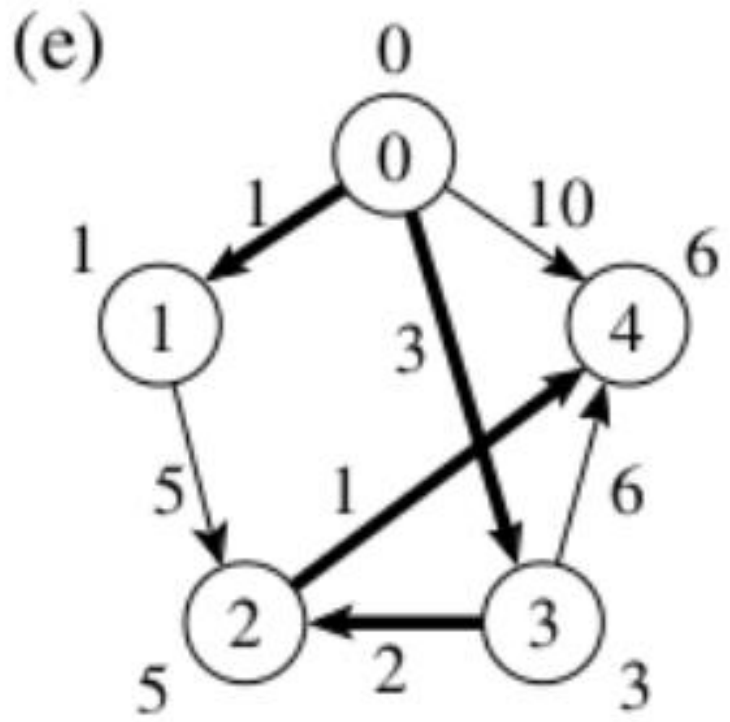

| Vértice | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Custo | 0 | 1 | 5 | 3 | 9 |
| Anterior | -1 | 0 | 3 | 0 | 3 |

# Simulação Dijkstra



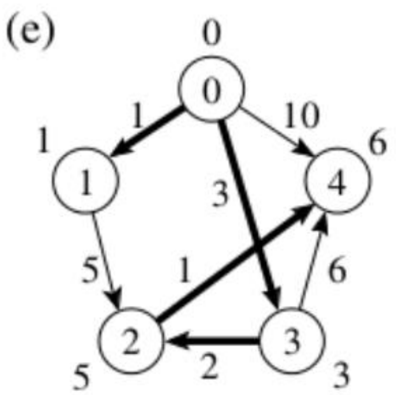

| Vértice | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Custo | 0 | 1 | 5 | 3 | 6 |
| Anterior | -1 | 0 | 3 | 0 | 2 |

# 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 <vector>
#include <queue>
#include <utility>

using namespace std;

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
#define INF 1000000000
```

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

void dijkstra(int s)
{
    dist[s] = 0;
    priority_queue<ii, vii, greater<ii> > pq;

    pq.push(make_pair(0, s));

    while(!pq.empty())
    {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if(d > dist[u]) continue;
        for(int i = 0; i < (int)adj[u].size(); i++) {
            ii v = adj[u][i];
            if(dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
}
```

# Caminho Mínimo (Shortest Path)

## Qual a complexidade do Algoritmo?

Para que todos os vértices sejam retirados da heap, 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 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).

# Hands-on!

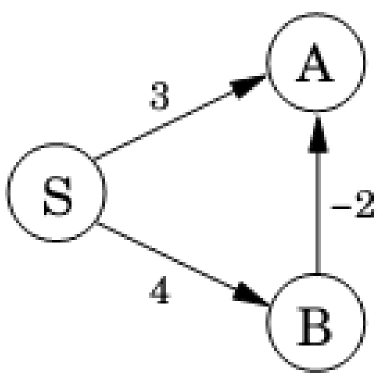
Vamos resolver um problema básico de Dijkstra.

Uva 10986 – *Sending e-mail*...

# 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 $O(|V|.|E|)$

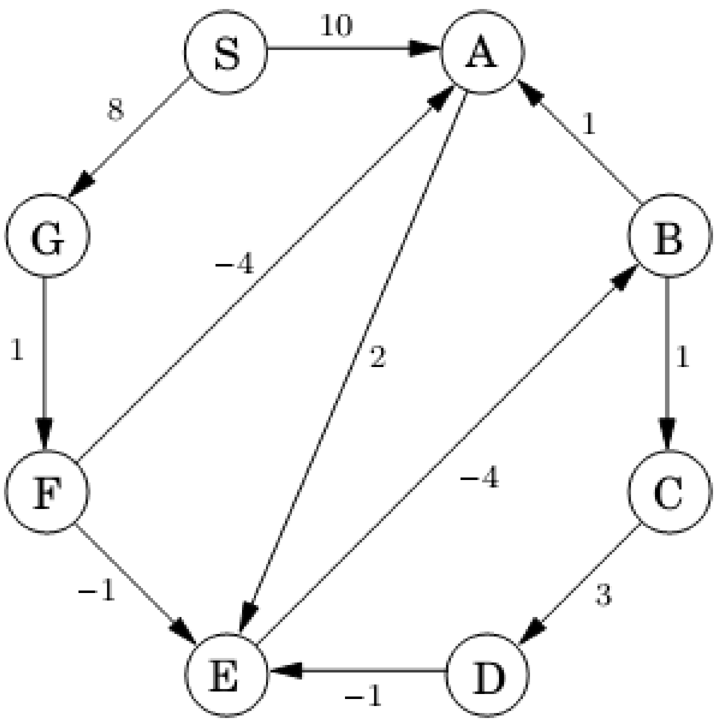
- No pior caso, em grafos densos, com aproximadamente $|V|^2$ arestas, sua complexidade é cúbica.
- Portanto, o algoritmo de Bellman-Ford possui complexidade $O(n^3)$.

# Caminho Mínimo (Shortest Path)

- O problema do caminho mínimo não pode ser bem definido para grafos com ciclos negativos.
  - Um ciclo desse tipo possibilitaria a atualização de custos infinitamente, reduzindo o custo em toda iteração.
- Portanto, para verificar se existe um ciclo negativo, basta realizar uma rodada extra:
  - Repita o laço externo |V| vezes ao invés de |V|-1
  - Existirá um ciclo negativo se e somente se houver atualização de algum custo na última rodada.

# Caminho Mínimo (Shortest Path)

Ciclos negativos: Ciclos onde a soma dos pesos das arestas que o formam é um número negativo.




**Exemplo:**
No grafo ao lado, existe um ciclo entre A → E → B → A, e a soma dos pesos da aresta é 2 + (-4) + 1 = -1

**É um ciclo negativo.**

Não conseguimos obter o custo mínimo utilizando **Bellman-Ford**.

# Árvores Geradoras Mínimas

Por último, veremos um outro problema clássico, o das Árvores Geradoras Mínimas.

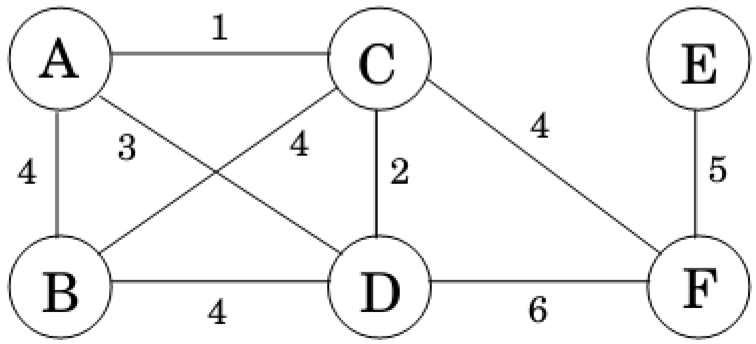
– Suponha que você precise conectar um conjunto de computadores em rede (vértices), ligando pares selecionados deles com cabos de rede (arestas).
– *Minimum Spanning Tree* (MST).

Objetivo: Selecionar um número mínimo de arestas de maneira que o grafo seja conexo.

– Além disso, cada link possui um custo de manutenção, indicado pelo peso da aresta. Precisamos encontrar a rede mais barata possível!

# Árvores Geradoras Mínimas

Veja o exemplo abaixo:



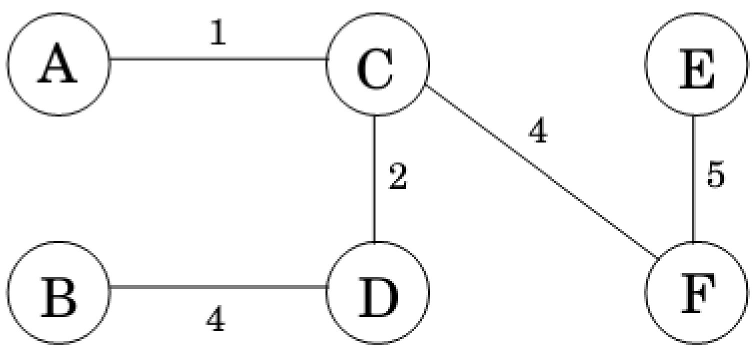

Uma observação imediata é que o conjunto ótimo de arestas não pode conter um ciclo, pois a remoção de uma aresta desse ciclo reduz o custo geral e não compromete a conectividade.

A solução portanto é **conexa e acíclica**: grafos não direcionados desse tipo são chamados de “árvores”!

# Árvores Geradoras Mínimas

A árvore que desejamos encontrar é aquela com peso total mínimo, e a chamamos de **Árvore Geradora Mínima**.

- No exemplo anterior, a árvore geradora mínima possui um custo de 16:




É a única solução? Identifique outra!

# Árvores Geradoras Mínimas

- Algumas propriedades:
  – Remover uma aresta de um ciclo não desconecta um grafo;
  – Uma árvore com n nós possui n – 1 arestas;
  – Qualquer grafo não direcionado e conexo, $G = (V, E)$ com $|E| = |V| - 1$ é uma árvore;
  – Um grafo não direcionado é uma árvore se e somente se existir um único caminho entre qualquer par de nós. Caso contrário, existiria um ciclo.

# Árvores Geradoras Mínimas

O Algoritmo de Kruskal é um dos algoritmos utilizados para encontrar a árvore geradora mínima. Funciona da seguinte maneira:

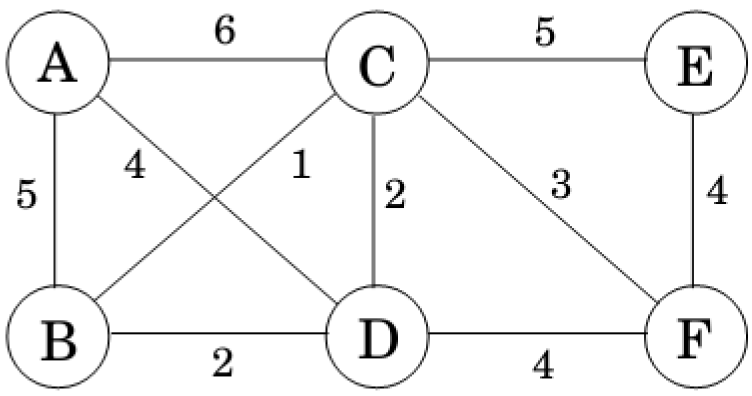
  – Repetidamente adicione a próxima aresta mais leve que não produz um ciclo.

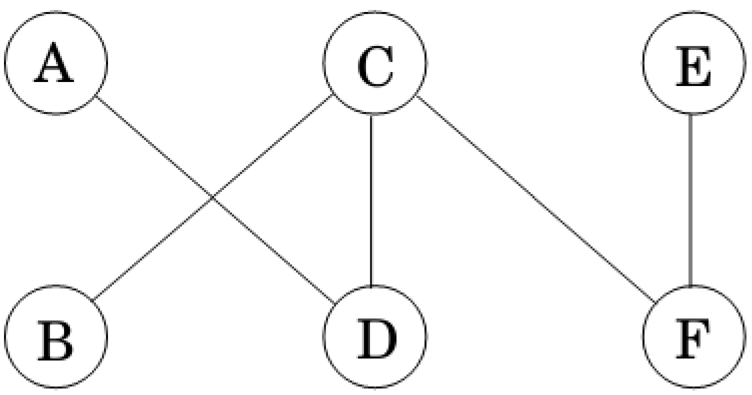
- É uma abordagem com estratégia gulosa.
  – Sempre pegue o menor peso.
- Em outras palavras, o algoritmo irá construir a árvore aresta por aresta e, além de tomar cuidado de evitar ciclos, simplesmente seleciona a aresta de menor peso no momento.
  – Vantagem imediata obvia.

# Árvores Geradoras Mínimas

No exemplo abaixo, começamos com um grafo vazio e, então, tentamos adicionar as arestas em ordem crescente de peso:

B-C, C-D, B-D, C-F, D-F, E-F, A-D, A-B, C-E, A-C




Peso Total: 1 + 2 + 3 + 4 + 4 = **14**

# Árvores Geradoras Mínimas

Na implementação, iremos utilizar um vetor de inteiros para guardar a componente à qual pertence cada vértice.

- Iniciamos com o grafo completamente desconexo, com componentes = {0, 1, 2, 3, 4, 5}

Assumindo que as arestas estão previamente ordenadas, tomamos sempre a menor, ou seja, a próxima na lista de arestas.

- A aresta irá fazer parte da árvore geradora mínima caso os dois vértices que a compõem pertencerem a componentes conexas diferentes.
- Nesse caso adicionamos seu custo ao custo total e fazemos a união das duas componentes conexas em uma só.

```
// prepara lista de arestas
vector< pair<int, ii> > edge_list;
for(int i = 0; i < E; i++)
{
    scanf("%d %d %d", &u, &v, &w);
    edge_list.push_back(make_pair(w, ii(u,v)));
}
sort(edge_list.begin(), edge_list.end()); // O(ElogE)

int mst_cost = 0;
union_find uf(V); // todos os vertices sao inicialmente desconexos

for(int i = 0; i < E; i++) { // O(E)
    pair<int, ii> front = edge_list[i];
    if(!uf.is_same_set(front.second.first, front.second.second)) {
        mst_cost += front.first; // adiciona peso a arvore
        uf.union_set(front.second.first, front.second.second); // conecta
    }
}

printf("MST cost = %d (Kruskal)\n", mst_cost);
```

A complexidade ficaria igual á da ordenação, que é O(E log E).

# Nessa implementação, utilizamos union_find.

```
struct union_find{
    vi parent;

    // initializes structure
    union_find(int n) {
        parent = vector<int>(n);
        for(int i = 0; i < n; i++)
            parent[i]= i;
    }

    int find(int x) { // find its parent until it is a root
        if(parent[x] == x) return x;
        else {
            parent[x] = find(parent[x]);
            return parent[x];
        }
    }

    void union_set(int x, int y) { // connects the roots of x and y
        parent[find(x)] = find(y);
    }

    bool is_same_set(int x, int y) { // x and y are in the same set?
        return find(x) == find(y);
    }
};
```

# Hands-on!

Vamos resolver um problema básico de MST.

Uva 00908 – *Re-connecting Computer Sites*