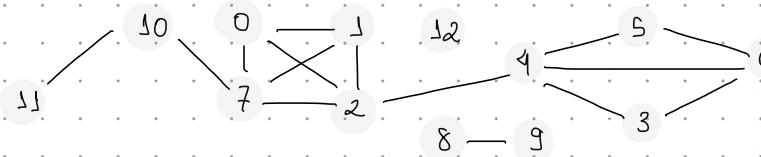


Grafos

Um grafo é formado por um conjunto de vértices e um conjunto de arestas. Cada aresta conecta um par de vértices.



O grafo tem V vértices e E arestas. Os vértices usam os números $\{0, 1, \dots, V-1\}$ e as arestas usam pares de inteiros:

Exemplo: $V = 13$ $E = 15$

$0-1, 0-2, 0-7, 1-2, 1-7, 2-4, 2-7, 3-4, 3-6, 4-5, 4-6, 5-6, 7-10,$
 $8-9, 10-11$

Um caminho é uma sequência de vértices ligados por arestas

$11 - 10 - 7 - 5 - 2 - 7 - 0$

Se não repete vértices, chamamos de caminho simples.

Não consideramos arestas em que as duas pontas são iguais (loops) nem arestas múltiplas: várias arestas com as mesmas pontas.

Um circuito é um caminho em que o vértice inicial é igual ao final: $4 - 2 - 1 - 7 - 2 - 4$

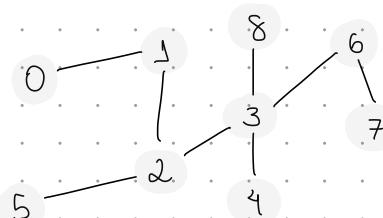
Um circuito simples não repete vértices (a menos do inicial).

Frequentemente consideramos caminhos e circuitos simples e nem usamos o adjetivo.

Dizemos que um grafo é conexo se para todo par de vértices u, v caminho ligando os vértices.

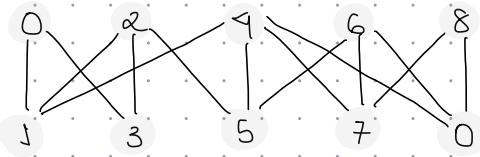
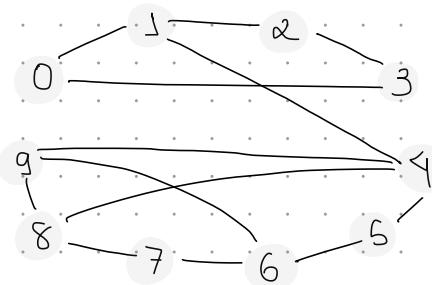
Um grafo é acíclico se não tiver circuitos.

Um grafo conexo e acíclico é chamado de árvore.

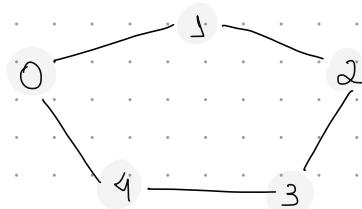


to die many

Dizemos que um grafo é bipartido se o conjunto de vértices pode ser dividido em duas partes e cada aresta tem uma ponta em cada.



Nem todo grafo é bipartido



Como representar um grafo?

Temos de guardar V, E e as arestas.

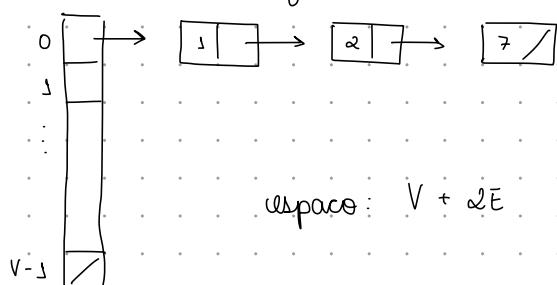
Temos duas formas de armazenar as arestas

⑤ matriz de adjacência: matriz $V \times V$ em que na posição u, v temos \mathbb{I} (verd).

$$\begin{matrix} & 0 & \cdots & \cdots & v_{n-1} \\ \vdots & & & & \\ 4 & & & & \\ \vdots & & & & \\ v-1 & & & & \end{matrix}$$

use $u - v$ é aresta do grafo. Usa espaço V^2 .

② Lista de adjacência:



Vetor com V listas ligadas. Cada lista contém os adjacentes ao vértice v .

espaco: $V + \lambda E$

matriz de incidência

Exemplo : $V = 13$ $E = 15$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	0	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	1	0	0	0	0	0
2	1	1	0	0	1	0	0	1	0	0	0	0	0

Algoritmos de busca em profundura

Um algoritmo de busca examina sistematicamente os vértices e arestas de um grafo. Cada aresta é examinada uma vez. Dois principais:

- busca em profundidade
- busca em largura

busca em profundidade

Em cada passo examino um vértice, marco que a busca já o examinou e visito cada um de seus vizinhos que ainda não foi visitado.

Ex: ① ② ③ ④

```
void dfs(){  
    bool * marked = new bool[v];  
    for(int i = 0; i < V; i++)  
        marked[i] = false;  
    for(int i = 0; i < V; i++)  
        if(!marked[i])  
            dfsR(i, marked);  
    delete [] marked;  
}
```

```
void dfsR(int v, bool marked[]){  
    marked[v] = true;  
    for(int i = 0; i < adj[v].size; i++){  
        w = adj[v].at(i);  
        if(!marked[w])  
            dfsR(w, marked);  
    }  
}
```

Aplicações:

① Têm caminho?

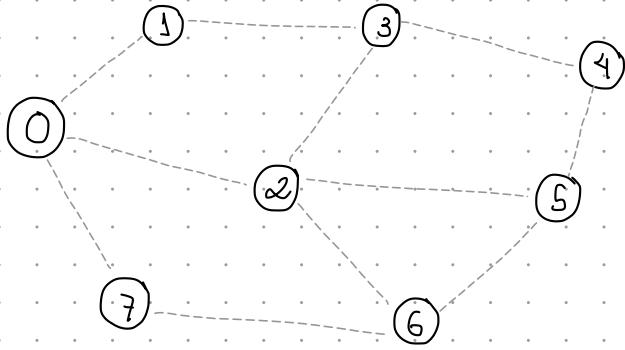
Dados vértices u, v existe caminho em G ligando u a v ?

```
bool temCaminho(int u, int v){  
    bool * marked = new bool[V]  
    for(int i = 0; i < V; i++)  
        marked[i] = false;  
    dfsR(u, marked[v]);  
    return;  
}
```

② Qual caminho?

Dados u, v , escreba, se existir um caminho de u a v em G .

19 de maio



$$0 - 1 - 3 - 2 - 5 - 4 \\ \quad \quad \quad \quad \quad \quad | \\ \quad \quad \quad \quad \quad \quad 6 - 7$$

Como encontrar o menor caminho entre os vértices $u-v$?

caminho de $0-7$

Busca em Largura

A ideia da busca em largura é visitar o vértice, em seguida seus vizinhos e assim por diante.

Isto pode ser implementado usando uma fila.

```
void bfs(){
    bool * marked = new bool[V];
    int * pred = new int[V];
    Fila <int> fila = new Fila();
    for(int i=0; i<V; i++)
        marked[i] = false;
        pred[i] = -1;
    for(int i=0; i<V; i++){
        if(!marked[i]){
            marked[i] = true;
            pred[i] = i;
            fila.insere(i);
            while(!fila.empty()){
                int u = fila.remove();
                for(int j=0; j<adj[u].size(); j++){
                    int v = adj[u].at(j);
                    if(!marked[v]){
                        marked[v] = true;
                        pred[v] = u;
                        fila.insere(v);
                    }
                }
            }
        }
    }
}
```

19 de maio

Mas, como provar que é o caminho mais certo de $u \rightarrow v$ quando faço o bfs a partir do vértice u ?

Se $\text{pred}[v] = -1$, então não existe caminho de u a v no grafo.

Suponha que existe um caminho de u a v : $u = w_0 - w_1 - w_2 - \dots - w_n = v$

$$\text{pred}[u] \neq -1$$

$$\text{pred}[v] = 1$$

Existe $i \in \{1, \dots, n\}$ tal que $\text{pred}[w_i] \neq -1$ e $\text{pred}[w_{i+1}] = -1$

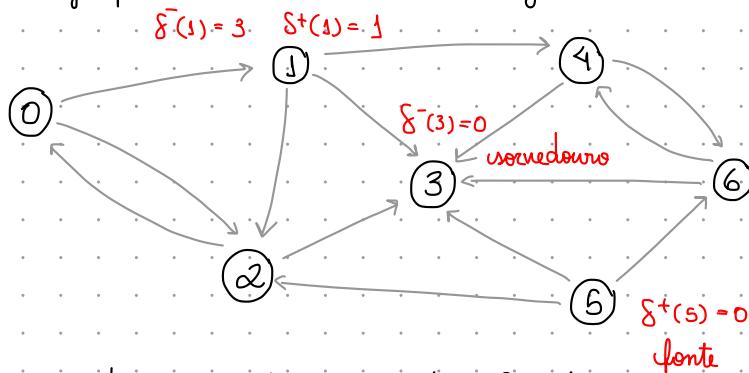
segunda parte. \uparrow ABSURDO

Se $\text{pred}[v] \neq -1$ então $\text{pred}[v]$ é o predecessor de v em um menor caminho de u a v . Vamos provar por indução nas iterações do algoritmo. Na primeira iteração é verdade.

Em uma iteração (qualquer) pegamos um vértice não marcado e marcamos. Usando a hip de indução e o fato de que os vértices à distância $i+1$ só são tirados da fila depois que todos os vértices à distância i são examinados, conclui-se o que se deseja provar.

Grafos dirigidos - digrafo

Em um digrafo as arestas são dirigidas e chamadas de arcos.



Chamamos de grau de entrada δ^+ de um vértice o número de arestas que entram em v . O grau de saída é denotado δ^- .

Da mesma forma que em grafos podemos considerar caminhos e circuitos. Porém, considerando as direções dos arcos.

$0 - 1 - 4 - 6 - 3$ é caminho

$0 - 1 - 2 - 0$ é circuito

19 de maio

Podemos usar matrizes e listas de adjacência para representar digrafos.

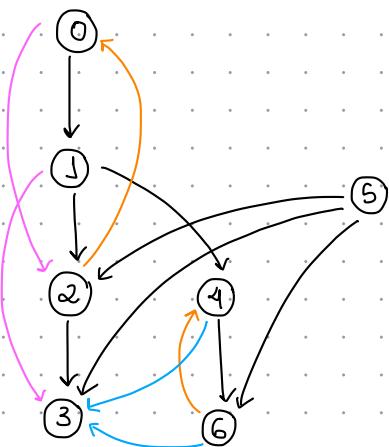
Note que as matrizes de digrafos não são necessariamente simétricas. Além disso, o arco $u-v$ é armazenado uma única vez, na lista de adjacências de u .

Podemos visitar os vértices e arcos de um digrafo usando dfs e bfs.

Exemplo: decidir se existe caminhos de u a v em um digrafo.

```
bool temCaminho(int u, int v){  
    bool * marked = new bool[V];  
    for(int i=0; i<V; i++) marked[v] = false;  
    dfsR((u, marked);  
    bool resp = marked[v];  
    delete[] marked;  
    return resp;  
}
```

```
void dfsR( int u, bool * marked){  
    marked[u] = true;  
    for(int i=0; i<adj[u].size(); i++){  
        int v = adj[u].at(i);  
        if(!marked[v])  
            dfsR(v, marked)  
    }  
}
```



→ arborescência (esqueleto)

→ retorno (ancestral)

→ descendente (encurta o caminho)

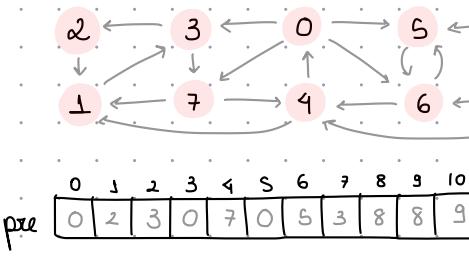
→ cruzado (nem descendente nem ancestral)

Os arcos da arborescência são os usados na recursão da dfs.

O arco $u-v$ é chamado de **retorno** se v foi visitado antes de u e é ancestral dele.

O arco $u-v$ é **descendente** se v foi visitado depois de u e é descendente dele.

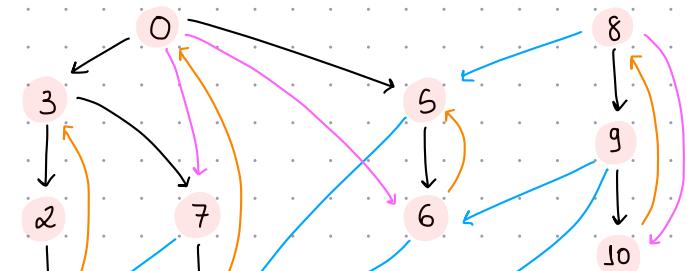
O arco $u-v$ é **cruzado** se v foi visitado antes do u e não é descendente nem ancestral.



pre [0 2 3 0 7 0 5 3 8 8 9]

beg [1 4 3 2 8 12 13 7 17 18 19]

end [16 5 6 11 9 15 19 10 22 21 20]



→ arborescência

```

void dfs(){
    bool * marked = new bool[V];
    int * pre = new int[V];
    int * beg = new int[V];
    int * end = new int[V];
    int tempo = 0;
    for(int i = 0; i<V; i++){
        marked[i] = false;
        beg[i] = pre[i] = end[i] = -1;
    }
    for(int v = 0; v<V; v++){
        if(!marked[v]){
            pre[v] = v;
            dfsR(v, marked, pre, beg, end, tempo);
        }
    }
}

void dfsR(int v, bool * marked, int * pre, int * beg, int * end, int & t{
    marked[v] = true;
    beg[v] = t++;
    for(int i=0; i<adj[v].size(); i++){
        int w = adj[v].at(i);
        if(!marked[w]){
            pre[w] = v;
            dfsR(w, marked, pre, begin, end, t);
        } // v-w
        else{ if(end[w] == -1) //retorno
            else if(end[w] < beg[v]) //cruzada
            else if(pre[w] != v) //descendente
            }
        end[v] = t++;
    }
}

```

24 de maio

tem circuito?

```
bool temCircuito(){
    for(v=0; v<V; v++){
        if(!marked[v])
            if(circR(v, marked, pre, beg, end, tempo))
                return true;
    }
    return false;
}

bool circR(int v, bool * marked, int * pre, int * beg, int * end, int & t){
    marked[v] = true; beg[v] = t++;
    for(int i=0; i<adj[v].size(); i++){
        w = adj[v].at(i);
        if(!marked[w]){
            pre[w] = v;
            if(circR(w, marked, pre, beg, end, t)) return true;
        } else if(end[w] == -1) return true;
    }
    end[v] = t++;
    return false;
}
```

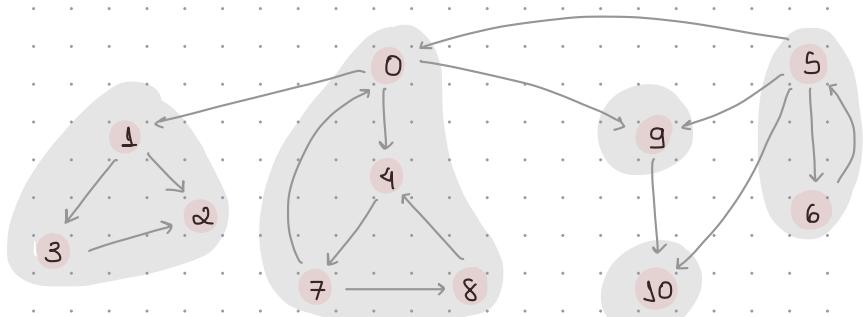
Dizemos que um digrafo é totalmente conexo se para todos pares de vértices $u, v \in V$, existem caminhos de u a v e de v a u em G .

bool totalConexo()

para cada $u, v \in V$ existe caminho (u, v) ? $V^2(V+E)$

na verdade, basta um dfs para cada vértice $V(V+E)$

Melhor ainda, roda dfs(0) $(V+E)$ e então constroi o grafo reverso que tem o mesmo conjunto de vértices e os arcos são invertidos no seu sentido $(V+E)$. Execute agora dfs(0) no reverso $(V+E) \rightarrow$ total $3(V+E)$. Se existe caminho de $0-u$ e para todo u de $0-u$ para todo u no grafo reverso, o digrafo é totalmente conexo.



Paulo Feofilloff

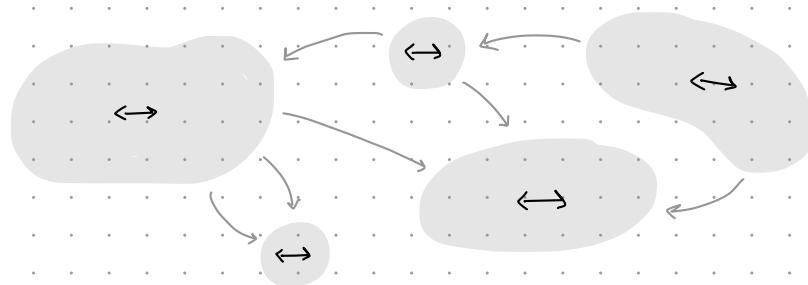
algoritmos em grafos

24 de maio

Dizemos que um vértice u é fortemente ligado a v se existe caminho de u a v e de v a u no digrafo.

Observe que essa é uma relação de equivalência.

As componentes fortemente conexas do digrafo são as classes de equivalência.



26 de maio

ALGORITMO DE TARJAN para componentes fortemente conexas

A ideia do algoritmo é identificar os vértices "cabeça" das componentes.

Dizemos que um arco $x-y$ abraça um vértice v se:

i) x é descendente de v

ii) y não é descendente de v

iii) existe caminho de y a ancestral próprio de v .

Um vértice é cabeça de uma componente se e só se nenhum arco do digrafo abraça v .

Para identificar as cabeças das componentes conexas vamos guardar para cada vértice

$\text{low}[v] = \min_{x-y} \text{beg}[y]$, $x-y$ abraça v

se $\text{low}[v] = \text{beg}[v]$, v é cabeça de componente

$\text{low}[v]$ é o mínimo entre

$$\begin{cases} \text{beg}[v], \\ \text{beg}[w], v-w & (\text{arco que abraça } v, \text{ arco de retorno}) \\ \text{low}[w], w \text{ filho de } v \end{cases}$$

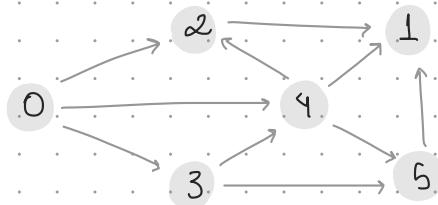
```
Pilha p = new Pilha();
int * compForConexas(){
    int * sc = new int[V];
    int * beg = new int[V];
    int * end = new int[V];
    int * low = new int[V];
    int tempo = 0; c = 0;
    for(int i=0; i<V; i++)
        sc[i] = beg[i] = end[i] = -1;
    for(int v=0; v<V; v++)
        if(beg[v] == -1)
            cfcR(v, sc, beg, end, low, tempo, c);
    return sc;
}
```

```
cfcR(int v, int * sc, int * beg, int * end, int * low, int &t, int &c){
    int u;
    beg[v] = t++;
    p.push(v);
    for(int i = 0; i<adj[v].size(); i++){
        int w = adj[v].at(i);
        if(beg[w] == -1){
            cfcR(w, sc, beg, end, low, t, c);
            if(low[w] < low[v])
                low[v] = low[w];
        }
        else if(end[w] == -1) //retorno
            if(beg[w] < low[v]) low[v] = beg[w];
    }
    end[v] = t++;
    if(beg[v] == low[v])
        do{
            u = p.pop();
            sc[u] = c;
        } while(u != v);
    c++;
}
```

3.1 de maio

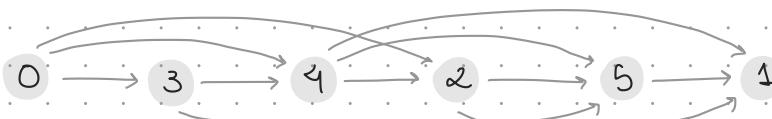
ORDENAÇÃO TOPOLOGICA

Considere um grafo em que os vértices correspondem a tarefas e os arcos são relações de precedência entre elas:

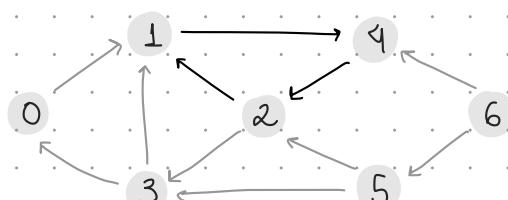


Uma ordenação topológica é uma ordenação dos vértices em que todas as tarefas que devem ser executadas antes de uma dada tarefa aparecem antes na ordenação.

Em outras palavras, todos os arcos apontam "para frente".



Todo digrafo tem uma ordenação topológica?



Se o grafo tem um ciclo, não admite uma ordenação topológica

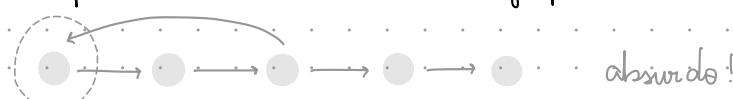


A recíproca também é verdadeira.

Se G_1 é acíclico, G_1 tem uma ordenação topológica.

Lema de Pedro: Se G_1 é acíclico, existe um vértice v com $\delta^+(v) = 0$.

Tome o caminho de comprimento máximo no digrafo:



Usando o lema, se G_1 é acíclico, tem vértice com $\delta^+(v) = 0$. $G_1 \setminus \{v\}$ é acíclico e pela HI, tem uma ordenação topológica. A ordenação de G_1 é dada incluindo v na primeira posição.

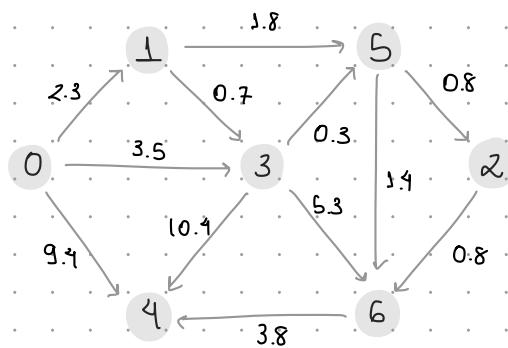
31 de maio.

Ao fazer a dfs podemos obter uma ordenação topológica, no momento em que o vértice termina a dfs, está determinada a posição na ordenação. Além disso, se identificarmos um arco de retorno, sabemos que o digrafo não tem ordenação topológica.

GRAFOS (DIGRAFOS) COM PESOS NAS ARESTAS (ARCOS)

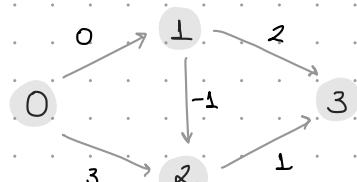
Em várias aplicações os arcos têm pesos associados à distância, custo de construção, ou custo para percorrer a aresta. Assim, faz sentido procurar por respostas que levem em conta os pesos.

Um exemplo é a busca de um caminho mais curto em um grafo (digrafo). Pode ser que o caminho com menos arcos não seja o melhor. Lembrem-se, com dfs descobrimos se existe caminho. A bfs encontra o caminho com menos arestas (arcos). Mas, como encontrar o caminho mais curto, considerando os pesos?



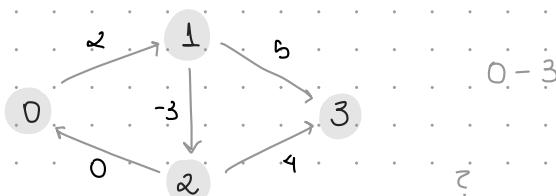
caminho mais curto $0 \rightarrow 4 : 9.4$

(minimum shortest path) $0 - 1 - 3 - 6 - 6 - 4 : 8.5$



MSP(0-3) : $0 - 1 - 2 - 3$

problema:



$0 - 3$

31 de maio

Como representar os custos:

- listas de adjacências
- matriz de adjacências

$+\infty$: inexistência da aresta/arcos

ALGORITMO DE DIJKSTRA (1959) para encontrar caminho de custo mínimo.

Vários algoritmos se baseiam na ideia de relaxar um arco $u-v$.

Considere um vetor

$\text{dist}[v]$: o comprimento do menor caminho que se conhece de u a v .

Se existir um arco $v-w$ tal que

$$\text{dist}[w] > \text{dist}[v] + \text{peso}[u][w]$$

posso atualizar o valor de $\text{dist}[w]$ passando por v .

```
void relaxaArco(int v, int w){  
    if(dist[w] > dist[v] + peso[v][w]){  
        dist[w] = dist[v] + peso[v][w];  
        pred[w] = v;  
    }  
}
```

Dizemos que um vértice u é relaxado se relaxamos todos os arcos que saem de u .

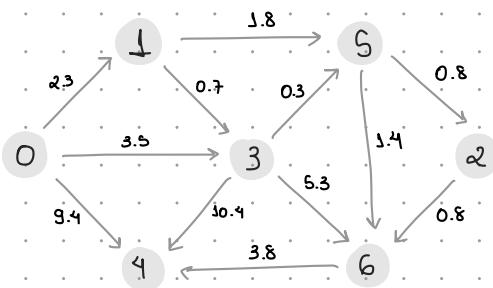
PROP. Seja G um digrafo, s um vértice origem e $\text{dist}[v]$ uma estimativa para a distância de s a v , com $\text{dist}[v] = +\infty$ se não conhecemos caminho de s a v .

Os valores de $\text{dist}[\cdot]$ são os comprimentos dos caminhos mais curtos usados para todo arco $u-v$, $\text{dist}[v] \leq \text{dist}[u] + \text{peso}[u][v]$.

(\Rightarrow) Vamos supor que $\text{dist}[v]$ estão corretos e, por absurdo, existe $u-v$ com $\text{dist}[v] > \text{dist}[u] + \text{peso}[u][v]$. Mas poderíamos atualizar $\text{dist}[v]$ para um valor menor.

(\Leftarrow) Vamos considerar que existe um caminho mínimo de s a um vértice v . $s = v_0 - v_1 - v_2 - \dots - v_{k-1} - v_k = v$

02 de junho



ALGORITMO DE DIJKSTRA para caminho de custo mínimo

c_{uv} curv $\rightarrow v$ dist $[u]$

dist $[v]$: estimativa para distância

dist $[v] > \text{dist}[u] + c_{uv}$

(relaxar o arco uv)

O algoritmo começa com

dist $[s] = 0$

dist $[v] = +\infty$ para $v \neq s$.

E com o conjunto T de vértices para os quais dist $[v]$ está correto. $T \leftarrow \emptyset$

$T \leftarrow$ $\{s\}$ $V \setminus T$ Em cada passo o algoritmo seleciona o vértice v em $V \setminus T$ com dist $[v]$ mínima

$0_{23} \quad 0_{12} \quad 0_{18} \quad 0_{13} \quad T \leftarrow T \cup \{v\}$

relaxa os arcos que saem de v

O algoritmo de Dijkstra funciona quando os arcos têm custo ≥ 0

PROPOSIÇÃO. O algoritmo de Dijkstra calcula concretamente as distâncias quando o grafo não tem arcos negativos.

PROPOSIÇÃO. Se v não é atingível a partir de s $\text{dist}[v] = +\infty$.

Caso contrário, em algum momento ele vai ser selecionado e seus arcos serão relaxados. Para esses arcos a desigualdade

$$\text{dist}[w] \leq \text{dist}[v] + \text{peso}[v][w]$$

sempre será satisfeita.

02 de maio

Quando v foi escolhido $\text{dist}[v]$ é mínimo entre os vértices de VIT . Não pode haver um caminho mais curto chegando em v por outros vértices de VIT .

O algoritmo para quando $VIT = \emptyset$ ou o vértice v com distância mínima é inatingível a partir de u ($\text{dist}[v] = +\infty$).

Para implementar o algoritmo precisamos de uma fila de prioridade para VIT . As operações que faremos são:

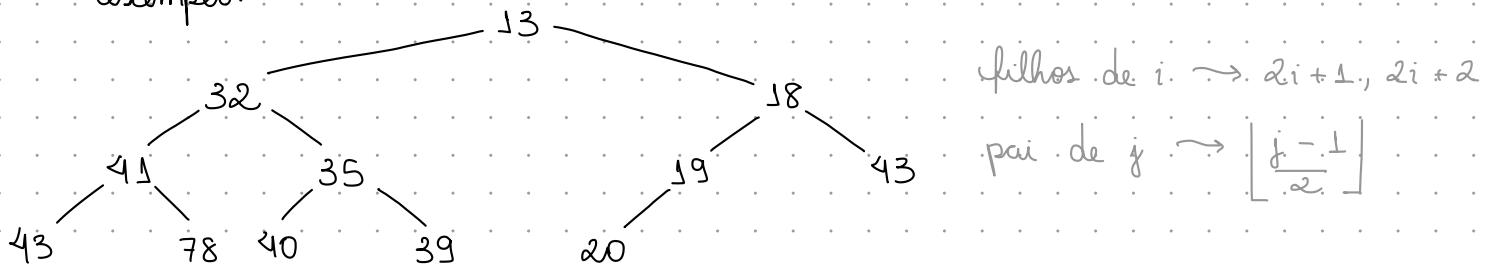
- remove v_{\min} ;
- muda prioridade;

A fila de prioridade pode ser implementada eficientemente usando min-heap.

Um min heap é uma árvore binária com as seguintes propriedades:

- completa até o penúltimo nível;
- as folhas do último nível estão à esquerda possível;
- o conteúdo do pai é menor que o dos filhos.

exemplo:



operações:

- remove mínimo: $v[0] = v[n-1]$ $O(\log n)$

- diminuir a prioridade $O(\log n)$

O algoritmo de Dijkstra executa V iterações e todos os E arcos são relaxados, no pior caso. Em cada iteração temos que remover do heap o vértice com distínacia $[V \log V]$. Para cada arco que é relaxado posso ter que arrumar o heap $(E \log V)$. A complexidade é $O((V+E) \log V)$ acabou a parte 2.