

Matematica Discreta – IFES BSI

<https://youtu.be/4v073vdQGX4>

Trabalho 1

Apresentação e explicação dos algoritmos de Dijkstra, Prim, Kruskal,
Busca em Profundidade e Busca em Nivel(Largura)

Aluno: Bruno Carvalho Caxias

Busca Sequencial

- A busca sequencial funciona indo de índice em índice dentro de uma lista
- Em listas pequenas ele é funcional, mas em grande escala o algoritmo peca em sua eficiência
- Sua complexidade é alta sendo:
 - n , no pior caso
 - $n/2$, no caso médio

Busca em Largura

- Tem como objetivo achar caminhos mínimos
- Considera a quantidade de saltos para se chegar em uma vértice, não colocando a complexidade de cada nó visitado em conta ao calcular o menor caminho entre nós.

Busca em Largura

A busca em largura funciona checando os nós diretamente ligados ao nó inicial e logo após analisando os nós ligados a esses

Pega todos os vértices tirando o inicial e os coloca como não visitados (BRANCO), distância máxima e nó pai como nulo

Coloca o nó inicial como sendo visitado (CINZA), coloca sua distância como 0 e seu nó pai como inexistente

Cria uma fila no modelo pilha, primeiro a entrar primeiro a sair

Enfileira o primeiro nó (s)

Desenfileira o primeiro nó, checa seus adjacentes, caso haja não visitados (BRANCOS) ele os visita (CINZA), dá o valor de distância (o nó anterior + 1), coloca o nó anterior como pai e o coloca na fila

Assim que visitados todos os nós ao seu redor ele coloca o nó como completamente visitado (PRETO)

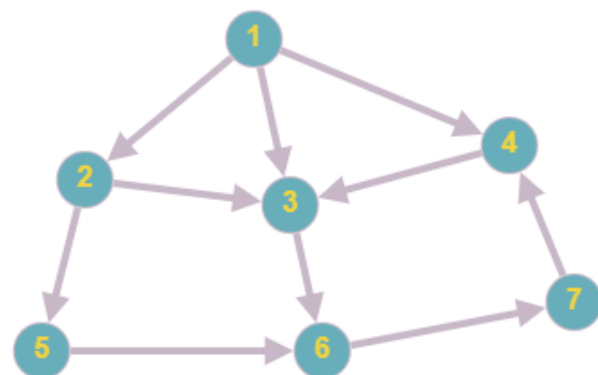
Pseudocódigo

Busca em largura

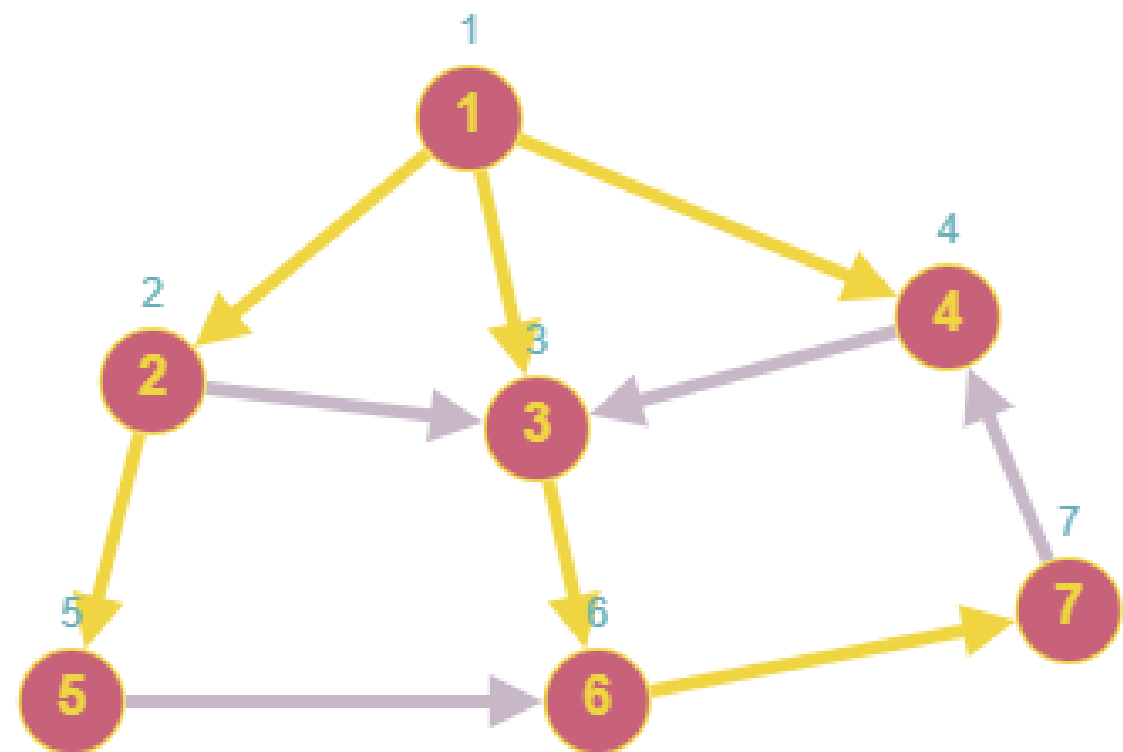
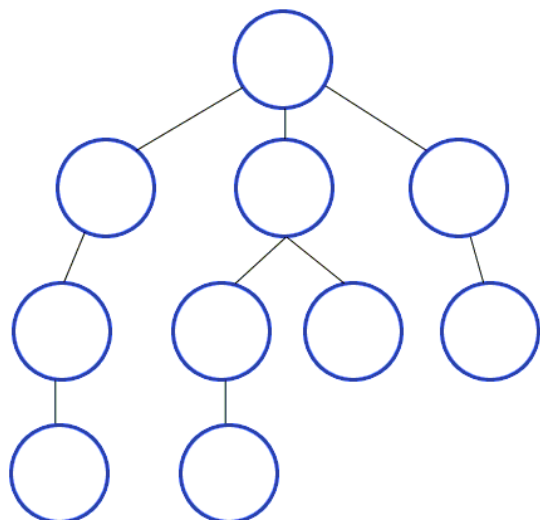
```
BFS(G,s)
  for cada vértice u ∈ V[G]-{s}{
    u.cor = BRANCO
    u.d = ∞
    u.π = NULO
  }
  s.cor = CINZA
  s.d = 0
  s.π = NULO
  Q = ∅ //Fila FiFo

  ENFILEIRAR(Q,s)
  while Q ≠ ∅ {
    u = DESENFILEIRAR(Q)
    for cada v = Adj[u]{
      if v.cor == BRANCO{
        v.cor = CINZA
        v.d = u.d+1
        v.π = u
        ENFILEIRAR(Q,v)
      }
    }
    u.cor = PRETO
  }
}
```

Busca em Largura



Outra representação:



Sequencia de nós visitados

Busca em Largura

Complexidade do Algoritmo

Busca em largura

```
BFS(G,s)
  for cada vértice  $u \in V[G]-\{s\}$  {
     $u.cor = \text{BRANCO}$ 
     $u.d = \infty$ 
     $u.\pi = \text{NULO}$ 
  }
   $s.cor = \text{CINZA}$ 
   $s.d = 0$ 
   $s.\pi = \text{NULO}$ 
   $Q = \emptyset$  //Fila FiFo

  ENFILEIRAR(Q,s)
  while  $Q \neq \emptyset$  {
     $u = \text{DESENFILAR}(Q)$ 
    for cada  $v = \text{Adj}[u]$  {
      if  $v.cor == \text{BRANCO}$  {
         $v.cor = \text{CINZA}$ 
         $v.d = u.d + 1$ 
         $v.\pi = u$ 
        ENFILEIRAR(Q,v)
      }
    }
     $u.cor = \text{PRETO}$ 
  }
}
```

$O(V)$

$O(V)$

Máx: $2|E| = O(E)$

Busca em profundidade

- A busca em profundidade funciona selecionando um nó e indo até sua ultima aresta e depois realizando o backtracking e se aprofundando novamente

Busca em Profundidade

A busca continua até se saber todos os vértices atingíveis pelo vértice inicial

Da cor aos vértices (a posição da cor será igual a posição do vertice)

Coloca todos os vértices como não visitados (BRANCO)

Se o vértice não tiver sido visitado a função/método de visita profunda é chamada

Enfileira o primeiro nó (s)

A função/método visitaP recebe o grafo, o nó inicial e o arranjo de cores. Logo após, visita a adjacência do nó, enquanto houver adjacência o algoritmo continua visitando (recursivamente) enquanto houverem nós adjacentes não visitados (BRANCOS)

Assim que se chega a um nó sem adjacências, ele é pintado de vermelho e ocorre um backtracking ao nó anterior e o processo se repete

Pseudocódigo

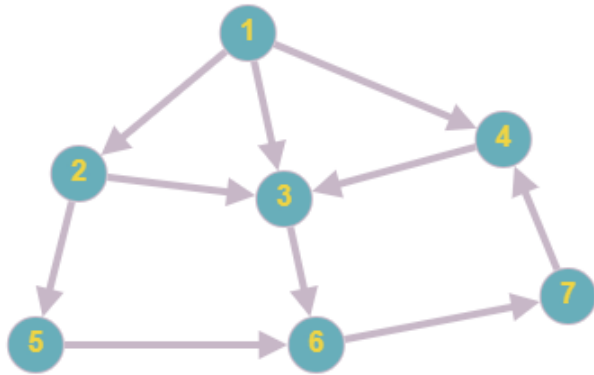
Busca em Profundidade

```
DFS(G,s){
    int cor[g->vertices];

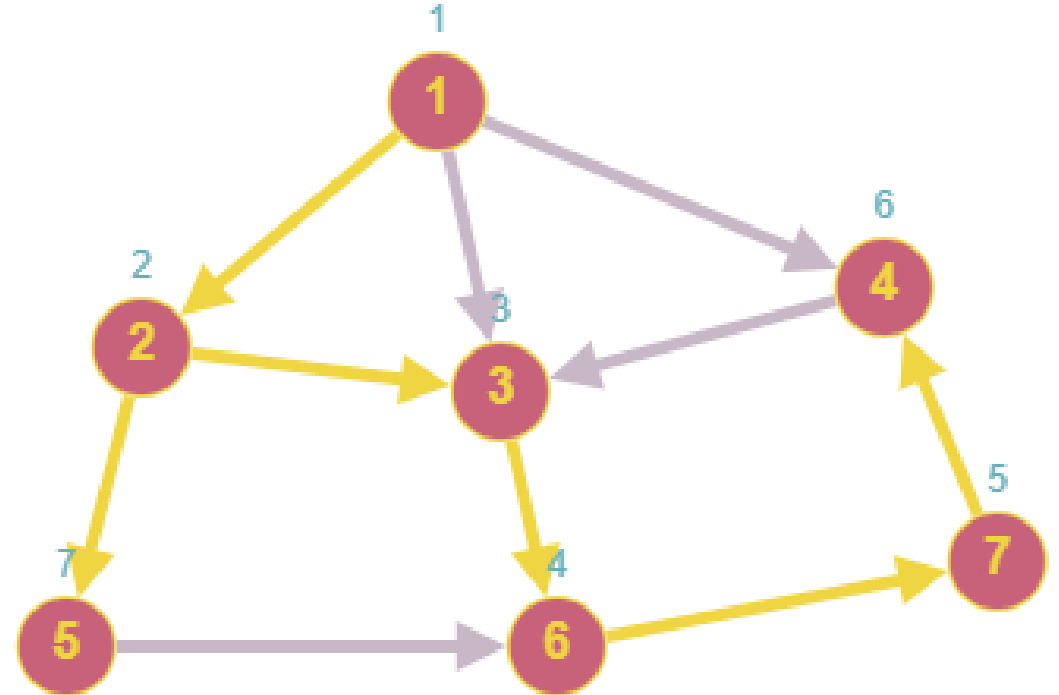
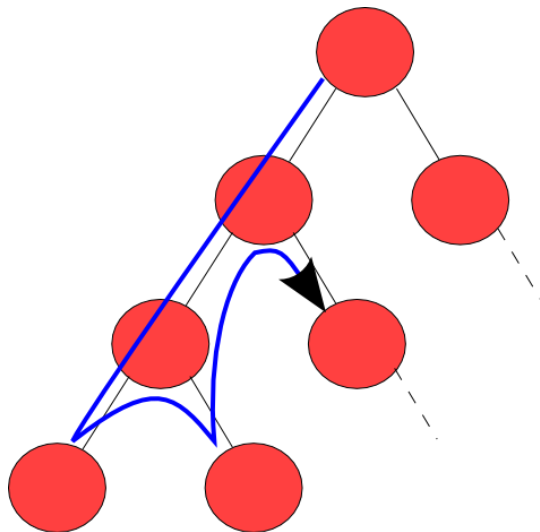
    int u;
    for (u=0;u<g->vertices;u++){
        cor[u] = BRANCO
    }
    for (u=0;u<g->vertices;u++){
        if (cor[u] == BRANCO)
            visitaP(g,u,cor);
    }
}

visitaP(GRAFO *g, int u, int cor){
    cor[u] = AMARELO;
    ADJACENCIA *v = g->adj[u].cab
    while(v){
        if (cor[v->vertice] == BRANCO)
            visitaP(g,v->vertice,cor);
        v = v->prox
    }
    cor[u] = VERMELHO
}
```


Busca em Profundidade



Outra representação:



Sequencia de nós visitados

Busca em Profundidade

Pseudocódigo

Busca em Profundidade

```
DFS(G,s){
    int cor[g->vertices];

    int u;
    for (u=0;u<g->vertices;u++){
        cor[u] = BRANCO
    }
    for (u=0;u<g->vertices;u++){
        if (cor[u] == BRANCO)
            visitaP(g,u,cor);
    }
}

visitaP(GRAFO *g, int u, int cor){
    cor[u] = AMARELO;
    ADJACENCIA *v = g->adj[u].cab
    while(v){
        if (cor[v->vertice] == BRANCO)
            visitaP(g,v->vertice,cor);
        v = v->prox
    }
    cor[u] = VERMELHO
}
```

Em Busca em Profundidade a complexidade do algoritmo se de em

$$O(|V(G)| + |E(G)|)$$

Algoritmo de Dijkstra

- O funcionamento mesmo sendo similar o Algoritmo de Dijkstra leva em consideração o peso de cada aresta diferente do de Busca em Largura ou em Profundidade.

Algoritmo de Dijkstra

Ao levar em consideração o peso de cada aresta o algoritmo consegue de forma informar de forma mais correta o menor caminho a se ter do nó inicial ao final, o caminho tomado sempre será com as arestas de menor custo.

Inicializa e dá a todos os vertices a distancia máxima e define o predecessor como -1(= INEXISTENTE)

Define a distancia do nó de começo sendo 0

Pega as adjacências de u e busca o vértice final, enquanto não for encontrado uma próxima adjacência será escolhida

Checa se a distancia aproximada para v é menor que o peso e distancia anteriormente estimado, se for ocorre uma atualização dos valores

Pseudocódigo

Algoritmo de Dijkstra

```
void inicializaD(GRAFO *g, int *d, int *p,
int s){
    int v;
    for (v=0;v<g->vertices;v++){
        d[v] = INT_MAX/2;
        p[v] = -1 ;
    }
    d[s] = 0;
}

void relaxa(GRAFO *g, int *d, int *p, int u,
int v){
    ADJACENCIA *ad = g->adj[u].cab;
    while (ad && ad->vertice != v)
        ad = ad->prox
    if (ad){
        if(d[v] > d[u] + ad->peso){
            d[v] = d[u] + ad->peso;
            p[v] = u;
        }
    }
}
```

Algoritmo de Dijkstra

Ao levar em consideração o peso de cada aresta o algoritmo consegue de forma informar de forma mais correta o menor caminho a se ter do nó inicial ao final, o caminho tomado sempre será com as arestas de menor custo.

Alocação de memória para o arranjo de distancias

Aloca o arranjo dos predecessores e o inicializa junto com as distâncias

Inicializa arranjo de abertos com todos abertos

Checa se há vértices abertos se tiver ele seleciona o menor e começa a visita colocando aberto em false

Para cada adjacência do nó o loop ira relaxar a aresta

Depois do processo, se terá todas as distancias e essas serão retornadas

Pseudocódigo

```
int *dijkstra(GRAFO *g, int s){
    int *d = (int *);
    malloc(g->vertices*sizeof(int));
    int p[g->vertices];
    bool aberto[g->vertices];
    inicializaD(g,d,p,s);

    int i;
    for (i=0; i<g->vertices;i++)
        aberto[i] = true;

    while (existeAberto(g,aberto)){
        int u = menorDist(g,aberto,d);
        aberto[u] = false;

        ADJACENCIA *ad = g->adj[u].cab;
        while (ad){
            relaxa(g,d,p,u,ad->vertice);
            ad = ad->prox;
        }
    }
    return(d);
}
```

Algoritmo de Dijkstra

Funções Auxiliares

Varre o arranjo de abertos e se encontrar um true, caso haja, é retornado true caso não, false

Busca o primeiro aberto e ao encontrar sai do loop, caso não encontre retorna -1

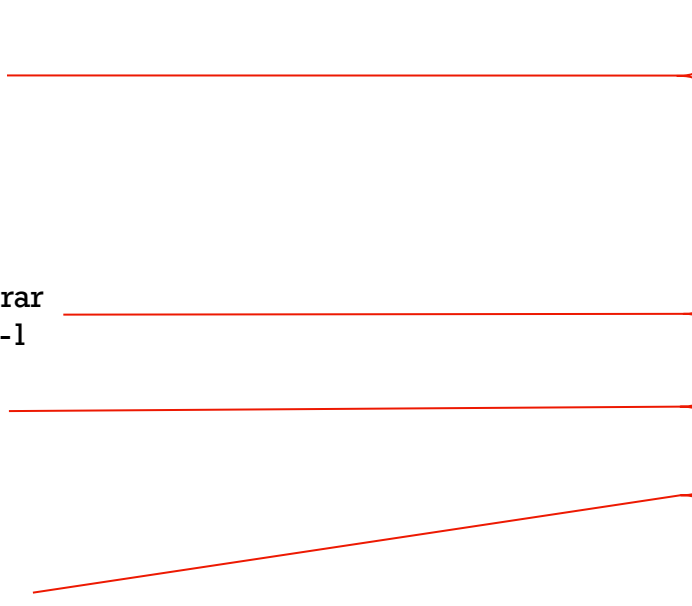
Coloca o nó encontrado como o mais perto (menor distancia)

O arranjo é varrido novamente a partir do nó definido como menor e caso se encontre outro nó de menor valor e aberto ele se torna o menor e retorna o valor

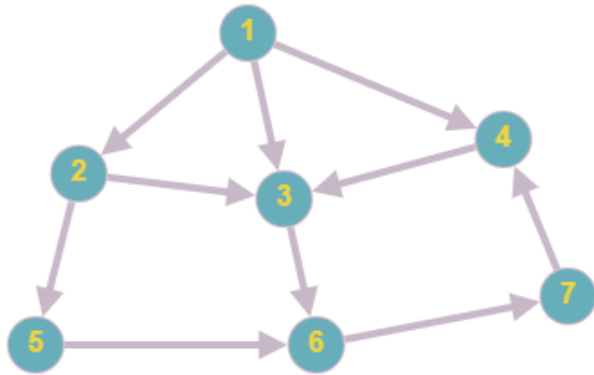
Pseudocódigo

```
bool existeAberto(GRAFO *g, int *aberto){
    int i;
    for (i=0;i<g->vertices;i++)
        if (aberto[i]) return(true);
    return(false);
}

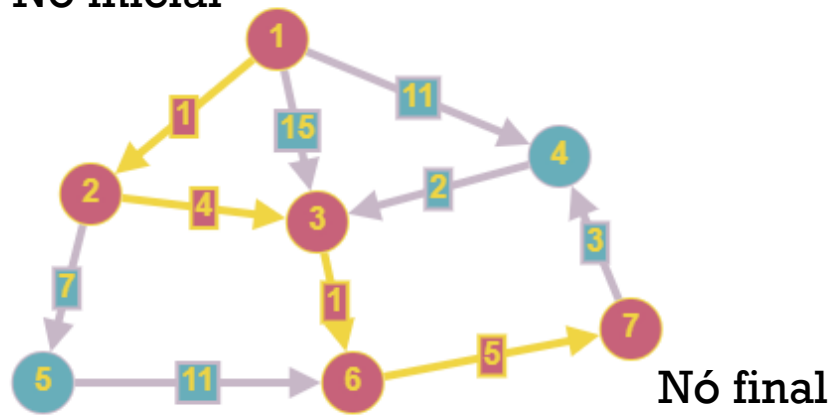
int menorDist(GRAFO *g, int *aberto, int *d){
    int i
    for (i=0;i<g->vertices;i++)
        if (aberto[i]) break;
    if (i==g->vertices) return(-1);
    int menor = i;
    for (i=menor+1;i<g->vertices;i++)
        if(aberto[i] && (d[menor]>d[i]))
            menor = i;
    return(menor);
}
```



Algoritmo de Dijkstra

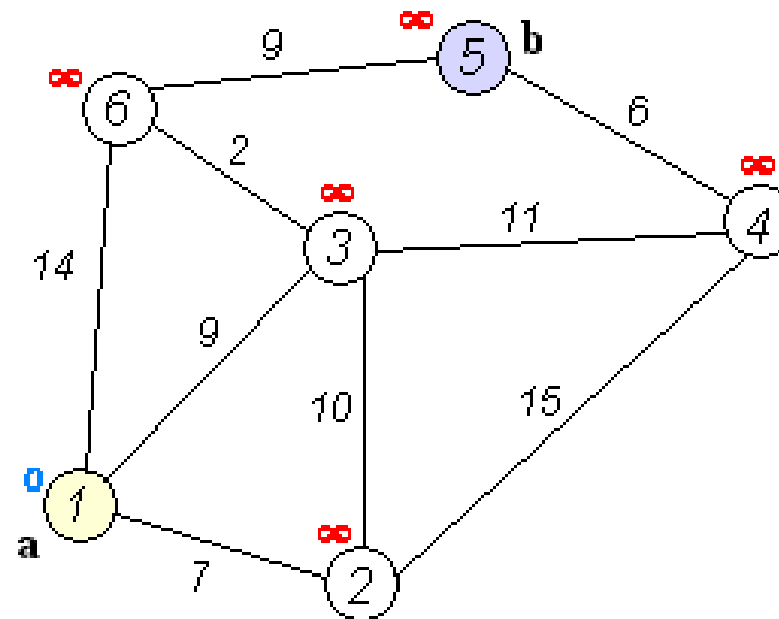


Nó inicial



Sequencia de nós visitados para
geração do menor caminho

Representação do funcionamento:



Algoritmo de Dijkstra

Complexidade do algoritmo

Suponha que nosso grafo tem V vértices e A arcos. Todas as operações sobre a fila serão executadas em tempo limitado por $\log V$. Nesse caso, `int *dijkstra()` consumirá tempo proporcional a $(V+A) \log V$ no pior caso. Se $A \geq V - 1$, o consumo de tempo no pior caso será proporcional a

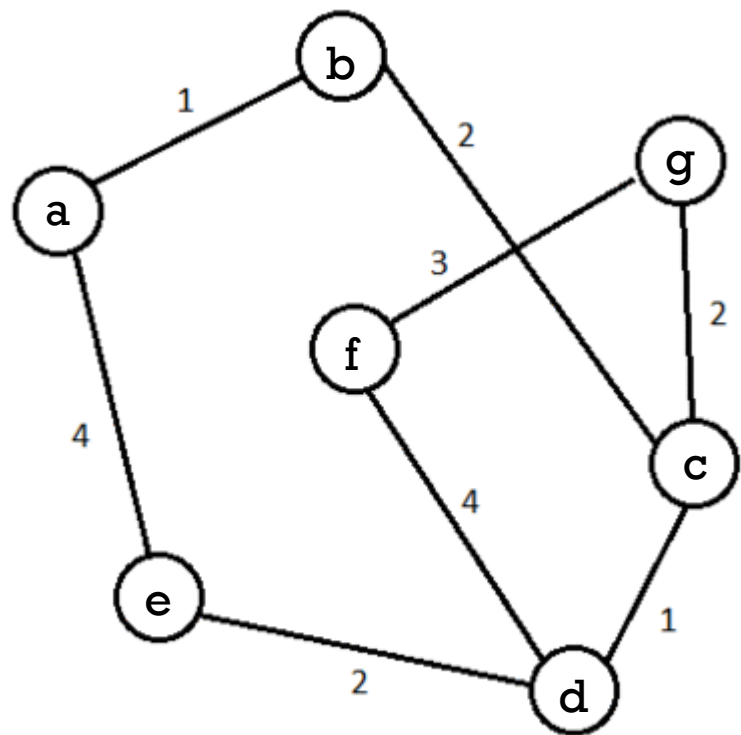
$$A \log V$$

Algoritmo de Prim

- Determina uma árvore geradora em que a soma das arestas seja mínima
- Junto com Kruskal, ele funciona somente em grafos não orientados, com valores associados às arestas
- Os valores das arestas devem ser positivos ou nulos

Algoritmo de Prim

Representação do funcionamento:



$$a-b = 1$$

$$a-e = 4$$

$$b-c = 2$$

$$c-d = 1$$

$$c-g = 2$$

$$d-f = 4$$

$$d-e = 2$$

$$f-g = 3$$

Algoritmo de Prim

Seta todos os vértices como sem pai, não estando em uma árvore, preço infinito

Seta o parentesco de todos como 0 e o valor como o valor do “caminho”

Inicia a fila e cria um loop que insere os vértices na fila

Enquanto a fila não estiver vazia, y recebe o vértice de menor valor, seta sua árvore como true e checa se os vértices são de menor valor e não participam da mesma árvore

PQinit(G->V): inicializa uma fila priorizada com capacidade para G->V vértices.

PQempty(): devolve true se e somente se a fila está vazia.

PQinsert(w,preco): insere o vértice w na fila com prioridade preco[w].

PQdelmin(preco): retira da fila um vértice y que minimiza preco[].

PQdec(w,preco): reorganiza a fila depois que o valor de preco[w] diminuiu.

```
void UGRAPHmstP2( UGraph G, vertex *pa)
{
    bool tree[1000];
    int preco[1000];
    // inicialização:
    for (vertex v = 1; v < G->V; ++v)
        pa[v] = -1, tree[v] = false, preco[v] = INFINITY;
    pa[0] = 0, tree[0] = true;
    for (link a = G->adj[0]; a != NULL; a = a->next)
        pa[a->w] = 0, preco[a->w] = a->c;

    PQinit( G->V);
    for (vertex v = 1; v < G->V; ++v)
        PQinsert( v, preco);

    while (!PQempty( )) {
        vertex y = PQdelmin( preco);
        if (preco[y] == INFINITY) break;
        tree[y] = true;
        // atualização dos preços e ganchos:
        for (link a = G->adj[y]; a != NULL; a = a->next)
            if (!tree[a->w] && a->c < preco[a->w]) {
                preco[a->w] = a->c;
                PQdec( a->w, preco);
                pa[a->w] = y;
            }
    }
    PQfree( );
}
```

Algoritmo de Prim

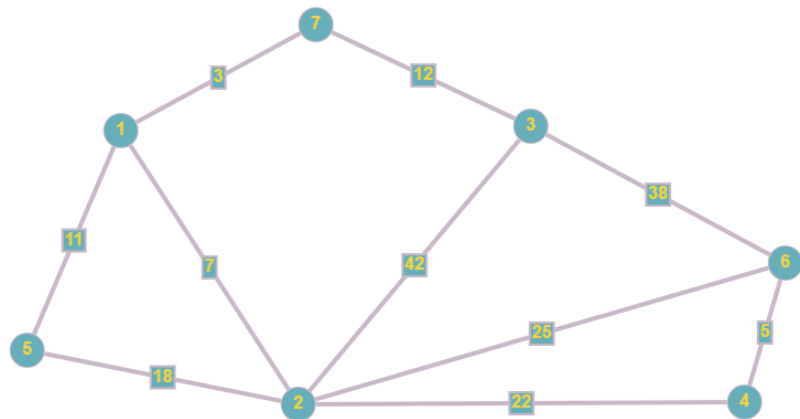
Com essa implementação da fila, a função `UGRAPHmstP2()` consome tempo proporcional a $(V+E) \log V$ no pior caso, sendo V o número de vértices e E o número de arestas de G . Como G é conexo, temos $E \geq V-1$ e portanto o consumo de tempo é proporcional a

$$E \log V$$

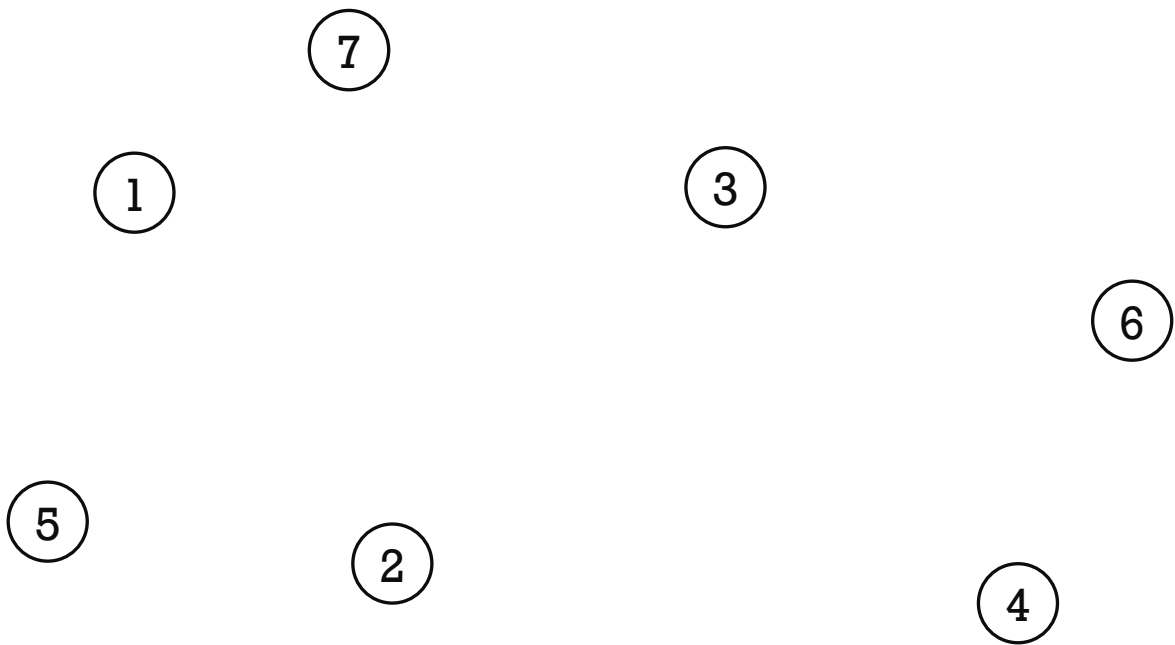
Algoritmo de Kruskal

- Cria uma floresta onde cada vértice é um árvore própria com o custo sendo o valor da aresta
- Possui solução se e somente se o grafo for conexo

Algoritmo de Kruskal



1	1	1	2	2	2	2	3	3	6
2	5	7	3	4	5	6	7	6	4
7	11	3	42	22	18	25	12	38	5



Algoritmo de Kruskal

Código

```
void UGRAPHmstK1( UGraph G, edge mst[])  
{  
    edge e[500000];
```

Armazena as arestas e as coloca em ordem crescente com a função sort()

```
    UGRAPHedges( G, e);  
    int E = G->A/2;  
    sort( e, 0, E-1);
```

Pega os pais de cada vértice e casos esses não estejam na mesma árvore os conecta por meio do UFind

```
    UFind( G->V);  
    int k = 0;  
    for (int i = 0; k < G->V-1; ++i) {  
        vertex v0 = UFind( e[i].v);  
        vertex w0 = UFind( e[i].w);  
        if (v0 != w0) {  
            UUnion( v0, w0);  
            mst[k++] = e[i];  
        }  
    }
```

Armazena as E arestas do grafo num vetor

```
static void UGRAPHedges( UGraph G, edge e[])  
{  
    int i = 0;  
    for (vertex v = 0; v < G->V; ++v)  
        for (link a = G->adj[v]; a != NULL; a = a->next)  
            if (v < a->w)  
                e[i++] = EDGE( v, a->w, a->c);  
}
```

Algoritmo de Kruskal

Funções auxiliares

inicializa a estrutura de chefes fazendo com que cada vértice seja o seu próprio chefe.

Código

```
void UFinit( int V) {  
    for (vertex v = 0; v < V; ++v) {  
        ch[v] = v;  
        sz[v] = 1;  
    }  
}
```

Devolve o chefe da componente conexa de F que contém o vértice v .

```
vertex UFfind( vertex v) {  
    vertex v0 = v;  
    while (v0 != ch[v0])  
        v0 = ch[v0];  
    return v0;  
}
```

faz a união das componentes cujos chefes são $v0$ e $w0$ respectivamente.

```
void UFunion( vertex v0, vertex w0) {  
    if (sz[v0] < sz[w0]) {  
        ch[v0] = w0;  
        sz[w0] += sz[v0];  
    }  
    else {  
        ch[w0] = v0;  
        sz[v0] += sz[w0];  
    }  
}
```


Algoritmo de Kruskal

Digamos que o grafo tem V vértices e E arestas. Então a função `sort()` consome tempo limitado por $E \log E$. O restante do código de `UGRAPHmstK1()` consiste em $2V$ invocações de `UFfind()` e V invocações de `UFunion()`, e portanto consome tempo limitado por $V \log V$. Como $\log E < 2 \log V$, podemos dizer que o consumo de `UGRAPHmstK1()` é limitado por

$$(E + V) \log V$$

no pior caso. Como o grafo é conexo, temos $E \geq V-1$ e portanto podemos dizer que o algoritmo consome tempo proporcional a $E \log V$ no pior caso.

A large red speech bubble graphic with a white border, pointing downwards. It contains the text 'FIM' and 'Obrigado pela atenção'. The background features faint, concentric circles and dashed lines.

FIM

Obrigado pela atenção