Algoritmos em Grafos

Ana Paula Tomás

Desenho e Análise de Algoritmos 2017/18

Outubro 2017

Determina os nós acessíveis de s no grafo $G = (V, A)$

Estratégia: **pesquisa em largura** a partir do nó s

```
BFS_Visit(s, G)          // Breadth-First Search
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
    visitado[s] ← true;
    Q ← MkEmptyQueue();
    Enqueue(s, Q);

    Repita
        v ← Dequeue(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                Enqueue(w, Q);
                visitado[w] ← true;
                pai[w] ← v;          // v precede w no caminho de s para w
    até (QueueIsEmpty(Q) = true );
```

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se designa por **árvore de pesquisa em largura a partir de s**.
- Os ramos da árvore são os pares $(pai[v], v)$ tais que $pai[v] \neq \texttt{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número de ramos possível). Os nós são visitados por ordem crescente de distância a s.
- Se G for não dirigido, os vértices visitados na chamada $\text{BFS\_Visit}(s, G)$ são os que definem a componente conexa a que s pertence.
- Para G dado por **listas de adjacências** e as operações MkEmptyQueue, Enqueue, QueueIsEmpty e Dequeue suportadas em $O(1)$:
  - a complexidade temporal de $\text{BFS\_Visit}(s, G)$ é $O(|V| + |A|)$.
  - a complexidade espacial é $O(|V|)$, se a fila for suportada por uma lista ligada, inicialmente está vazia (além de $\Theta(|V| + |A|)$ para G).

Escrita do caminho de s para v (se existe), $s \neq v$

Assumindo que $pai[v] \neq$ `NULL`, a função seguinte determina a sequência de vértices no caminho de s para v

```
EscreveCaminho(v, pai)
    Se pai[v] ≠ NULL então
        EscreveCaminho(pai[v], pai);
    escrever(v);
```

Visitar o grafo $G = (V, A)$ em largura

```
BFS(G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
    Q ← MkEmptyQueue();
    Para cada v ∈ G.V fazer
        Se visitado[v] = false então
            BFS_Visit(v, G, Q);
```

```
BFS_Visit(s, G, Q)
    visitado[s] ← true;
    Enqueue(s, Q);
    Repita
        v ← Dequeue(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                Enqueue(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QueueIsEmpty(Q) = true );
```

Assume que as variáveis $pai[\cdot]$ e $visitado[\cdot]$ são globais.

Distância mínima do nó s a cada nó (minimizar número de ramos)

```
BFS_Visit_Distancia(s, G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
        dist[v] ← ∞;
    visitado[s] ← true;
    dist[s] ← 0;
    Q ← MkEmptyQueue();
    Enqueue(s, Q);
    Repita
        v ← Dequeue(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                dist[w] ← dist[v] + 1;
                Enqueue(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QueueIsEmpty(Q) = true );
```

BFS visita vértices por ordem crescente de distância

Proposição: Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Caso de Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita todos os seus adjacentes.
- **Hereditariedade**
  - Suponhamos agora, como hipótese de indução, que para todo o u tal que $dist[u] < d$ se tem $dist[u] = \delta(s, u)$.
  - Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$.
  - Quando v' sai da fila, v está por visitar. Se não, $dist[v] \leq dist[v']$.
  - Logo, v' visita v e, portanto, $dist[v] = dist[v'] + 1$. Como $dist[v'] = d - 1 < d$, tem-se, pela hipótese, $dist[v'] = \delta(s, v')$. Logo, $dist[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Proposição: Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Caso de Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita todos os seus adjacentes.
- **Hereditariedade**
  - Suponhamos agora, como hipótese de indução, que para todo o u tal que $dist[u] < d$ se tem $dist[u] = \delta(s, u)$.
  - Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$.
  - Quando v' sai da fila, v está por visitar. Se não, $dist[v] \leq dist[v']$.
  - Logo, v' visita v e, portanto, $dist[v] = dist[v'] + 1$. Como $dist[v'] = d - 1 < d$, tem-se, pela hipótese, $dist[v'] = \delta(s, v')$. Logo, $dist[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Proposição: Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Caso de Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita todos os seus adjacentes.
- **Hereditariedade**
  - Suponhamos agora, como hipótese de indução, que para todo o u tal que $dist[u] < d$ se tem $dist[u] = \delta(s, u)$.
  - Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$.
  - Quando v' sai da fila, v está por visitar. Se não, $dist[v] \leq dist[v']$.
  - Logo, v' visita v e, portanto, $dist[v] = dist[v'] + 1$. Como $dist[v'] = d - 1 < d$, tem-se, pela hipótese, $dist[v'] = \delta(s, v')$. Logo, $dist[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Proposição: Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Caso de Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita todos os seus adjacentes.
- **Hereditariedade**
  - Suponhamos agora, como hipótese de indução, que para todo o u tal que $dist[u] < d$ se tem $dist[u] = \delta(s, u)$.
  - Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$.
  - Quando v' sai da fila, v está por visitar. Se não, $dist[v] \leq dist[v']$.
  - Logo, v' visita v e, portanto, $dist[v] = dist[v'] + 1$. Como $dist[v'] = d - 1 < d$, tem-se, pela hipótese, $dist[v'] = \delta(s, v')$. Logo, $dist[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Proposição: Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Caso de Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita todos os seus adjacentes.
- **Hereditariedade**
  - Suponhamos agora, como hipótese de indução, que para todo o u tal que $dist[u] < d$ se tem $dist[u] = \delta(s, u)$.
  - Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$.
  - Quando v' sai da fila, v está por visitar. Se não, $dist[v] \leq dist[v']$.
  - Logo, v' visita v e, portanto, $dist[v] = dist[v'] + 1$. Como $dist[v'] = d - 1 < d$, tem-se, pela hipótese, $dist[v'] = \delta(s, v')$. Logo, $dist[v] = \delta(s, v') + 1 = \delta(s, v)$.

Obter as componentes conexas de um grafo não dirigido

- Em $\mathrm{BFS}(G)$, o valor de $pai[v]$ identifica o primeiro nó que descobriu v durante a procura.
- No fim, o vetor $pai[.]$ define uma floresta de árvores pesquisa em largura.
- Por análise para trás a partir de v, podemos localizar a raíz da árvore de pesquisa em largura a que v pertence (podendo esta ser v se $pai[v] = \texttt{NULL}$).
- Para grafos **não dirigidos**, essa árvore define a componente conexa a que v pertence. Se G for conexo, a floresta reduz-se a uma árvore, à qual pertencem todos os vértices de G.

Obter as componentes conexas de um grafo não dirigido

Proposição Se G for um grafo não dirigido, cada árvore da floresta obtida por $\mathrm{BFS}(G)$ identifica uma componente conexa do grafo.

Ideia da prova:

- G pode ser representado por um grafo dirigido simétrico G' (que designamos por adjunto de G)
- Os vértices que constituem a árvore a que w pertence não dependem do nó raíz (embora a estrutura da árvore possa ser diferente os vértices são os mesmos).
- Na chamada de $\mathrm{BFS}(v, G, Q)$ no segundo ciclo de $\mathrm{BFS}(G)$, serão visitados todos os vértices acessíveis de v em G.
- Como o grafo adjunto de G é simétrico, se algum w acessível de v tivesse sido visitado numa chamada anterior, então também v teria de ter sido marcado como visitado por algum dos descendentes de w.

Ordenação topológica dos nós de um DAG

Dado um **grafo dirigido acíclico** $G = (V, A)$ finito, determinar uma função bijectiva σ de V em $\{0 \ldots, |V| - 1\}$ tal que $\sigma(v) < \sigma(w)$, para todo $(v, w) \in A$.
Ou seja, σ atribui um número distinto a cada nó e satisfaz a condição indicada.

Para todo $v \in G.V$ fazer $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$i \leftarrow 0$;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    $sigma[v] \leftarrow i$;
    $i \leftarrow i + 1$;
    Para todo $w \in G.Adjs[v]$ fazer
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;

Recordar que: qualquer DAG tem sempre algum vértice com grau de entrada zero.

Justificar que: se G não for um DAG, o ciclo "Enquanto" termina sempre mas com $i < n$ em vez de $i = n$.

Ordenação topológica dos nós de um DAG

Dado um **grafo dirigido acíclico** $G = (V, A)$ finito, determinar uma função bijectiva σ de V em $\{0 \ldots, |V| - 1\}$ tal que $\sigma(v) < \sigma(w)$, para todo $(v, w) \in A$.
Ou seja, σ atribui um número distinto a cada nó e satisfaz a condição indicada.

Para todo $v \in G.V$ fazer $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$i \leftarrow 0$;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    $sigma[v] \leftarrow i$;
    $i \leftarrow i + 1$;
    Para todo $w \in G.Adjs[v]$ fazer
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;

Recordar que: qualquer DAG tem sempre algum vértice com grau de entrada zero.

Justificar que: se G não for um DAG, o ciclo "Enquanto" termina sempre mas com $i < n$ em vez de $i = n$.

Caminho máximo num DAG

Determinar um caminho de comprimento máximo num grafo dirigido acíclico $G = (V, A)$, sendo o comprimento dado pelo número de ramos do caminho.

Para todo $v \in G.V$ fazer $ES[v] \leftarrow 0$; $Prec[v] \leftarrow$ `Nenhum`; $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$Max \leftarrow -1$; $v_f \leftarrow$ `Nenhum`;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S;
    $S \leftarrow S \setminus \{v\}$;
    Se $Max < ES[v]$ então $Max \leftarrow ES[v]$; $v_f \leftarrow v$; /* $ES[v]$ é o número de ramos do caminho */
    Para todo $w \in G.Adjs[v]$ fazer
        Se $ES[w] < ES[v] + 1$ então
            $ES[w] \leftarrow ES[v] + 1$; $Prec[w] \leftarrow v$;
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;
escreveCaminho(v_f, $Prec$); escrever(Max);

Escrever um caminho dado *Prec*[·] e o vértice final

Abordagem recursiva para escrever o caminho encontrado, sendo dados *Prec*[·] e o vértice v que é o último nesse caminho:

```
ESCREVECAMINHO(v, Prec)
    Se Prec[v] ≠ Nenhum então
        ESCREVECAMINHO(Prec[v], Prec);
    escrever(v);
```

Caminho máximo num DAG com pesos associados aos nós

Problema de escalonamento de tarefas: Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem que as que a precedem estejam concluídas. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível. Dado $G = (V, A, D)$ em que $D(v) \in \mathbb{R}_0^+$ é a duração da tarefa $v \in V$, obter MAX e agendar v o mais cedo possível (ES designa "earliest start").

Para todo $v \in G.V$ fazer $ES[v] \leftarrow 0$; $Prec[v] \leftarrow$ `Nenhum`; $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$Max \leftarrow -1$; $v_f \leftarrow$ `Nenhum`;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    Se $Max < ES[v] + D[v]$ então $Max \leftarrow ES[v] + D[v]$; $v_f \leftarrow v$;
    Para todo $w \in G.Adjs[v]$ fazer
        Se $ES[w] < ES[v] + D[v]$ então
            $ES[w] \leftarrow ES[v] + D[v]$; $Prec[w] \leftarrow v$;
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;
escreveCaminho$(v_f, Prec)$; escrever(Max);

Visita grafo $G = (V, A)$ em profundidade

DFS(G)

```
instante ← 0;
Para cada v ∈ G.V fazer cor[v] ← branco; Prec[v] ← Nenhum;
Para cada v ∈ G.V fazer
    Se cor[v] = branco então DFS_Visit(v, G);
```

DFS_Visit(v, G)

```
/* Vértices por visitar ainda a branco */
instante ← instante + 1;
t_inicial[v] ← instante;
cor[v] ← cinzento;        // cor útil para detetar ciclos
Para cada w ∈ G.Adjs[v] fazer
    Se cor[w] = branco então
        Prec[w] ← v;
        DFS_Visit(w, G);
cor[v] ← preto;
instante ← instante + 1;
t_final[v] ← instante;
```

Assume que as variáveis $cor[\cdot]$, $Prec[\cdot]$, $t\_inicial[\cdot]$, $t\_final[\cdot]$ e $instante$ são globais.

Visita grafo $G = (V, A)$ em profundidade (outra versão)

Versão simplificada e adaptada para que DFS retorne *stack* com os nós ordenados por ordem decrescente de tempo de finalização

```
DFS(G)
    stack ← Mk_Empty_Stack();
    Para cada v ∈ G.V fazer
        visitado[v] ← false;

    Para cada v ∈ G.V fazer
        Se visitado[v] = false então
            DFS_Visit(v, G, visitado, stack);
    return stack;
```

```
DFS_Visit(v, G, visitado, Stack)
    visitado[v] ← true;
    Para cada w ∈ G.Adjs[v] fazer
        Se visitado[w] = false então
            DFS_Visit(w, G, visitado, stack);
    Push(v,Stack);
```

Ordenação topológica por pesquisa em profundidade

```
TopSort_DFS(G)
    Stack ← {};        // definir stack vazia
    Para cada v ∈ G.V fazer cor[v] ← branco;
    Para cada v ∈ G.V fazer
        Se cor[v] = branco então TopSort_DFSVisit(v, G);
    /* Enquanto (S ≠ {}) fazer escrever(Pop()); */
    i ← 0;
    Enquanto (S ≠ {}) fazer
        sigma[Pop()] ← i;    /* como anteriormente */
        i ← i + 1;

TopSort_DFSVisit(v, G)
    cor[v] ← cinzento;
    Para cada w ∈ G.Adjs[v] fazer
        Se cor[w] = branco então
            TopSort_DFSVisit(w, G);
        senão se cor[w] = cinzento então    // retirar se sabe que G é DAG
            Termina com indicação de erro (G não é DAG);    // retirar se sabe que G é DAG
    cor[v] ← preto;
    Push(v);
```

Componentes fortemente conexas

Algoritmo de Kosaraju-Sharir

Usar $DFS(G)$ para obter pilha S com os nós por ordem decrescente de $t\_final[\cdot]$
Para $v \in G.V$ fazer $cor[v] \leftarrow$ **branco**;
Enquanto $(S \neq \{\ \})$ fazer
    $v \leftarrow POP(S)$;
    Se $cor[v] =$ **branco** então $\mathrm{DFS\_Visit}(v, G^T)$ e indica os nós visitados;

$G^T = (V, A^T)$ denota o **grafo transposto** de $G = (V, A)$, obtém-se de G se se trocar o sentido dos arcos, sendo, $A^T = \{(y, x) \mid (x, y) \in A\}$.

Complexidade temporal do algoritmo de Kosaraju-Sharir

O algoritmo de Kosajaru-Sharir tem complexidade $\Theta(|V| + |A|)$, (ou seja, linear na estrutura do grafo), se o grafo for representado por listas de adjacências.

Correção do Algoritmo Kosaraju-Sharir

- O *grafo das componentes fortemente conexas* G_{scc} de um grafo dirigido é um grafo dirigido acíclico (DAG).

  $\mathbf{G}_{scc}$: os nós correspondem às componentes fortemente conexas de G e os ramos são os pares (C, C') tais que $C \neq C'$ e existe algum ramo de algum nó de C para algum nó de C' em G.

  Se G_{scc} não fosse acíclico, quaisquer dois nós x e y que estivessem em componentes C_x e C_y (distintas) envolvidas num ciclo seriam acessíveis um do outro em G. Isso é absurdo, pois contradiz a noção de componente fortemente conexa.

- As componentes fortemente conexas de G e G^T têm os mesmos nós.

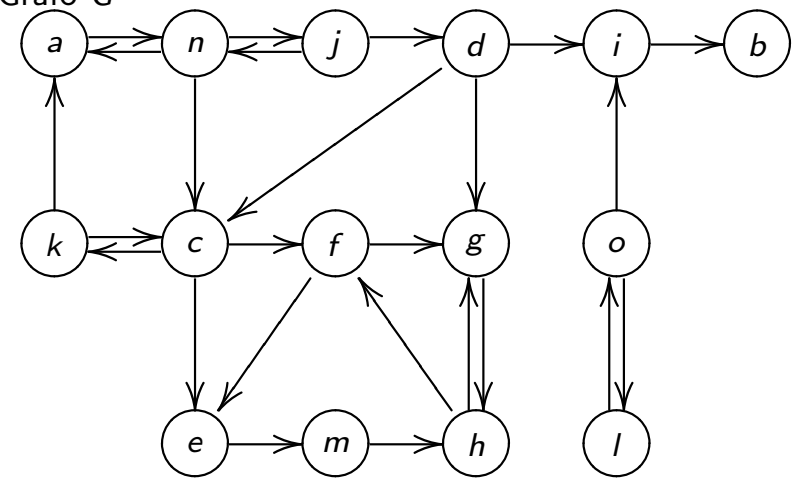
  Qualquer percurso de x para y em G é um percurso de y para x em G^T (e vice-versa).

- Uma ordenação topológica das componentes de G corresponde a uma ordenação topológica por ordem inversa (da cronológica) para as componentes de G^T. O DAG de componentes de G^T é o transposto do DAG de componentes de G.

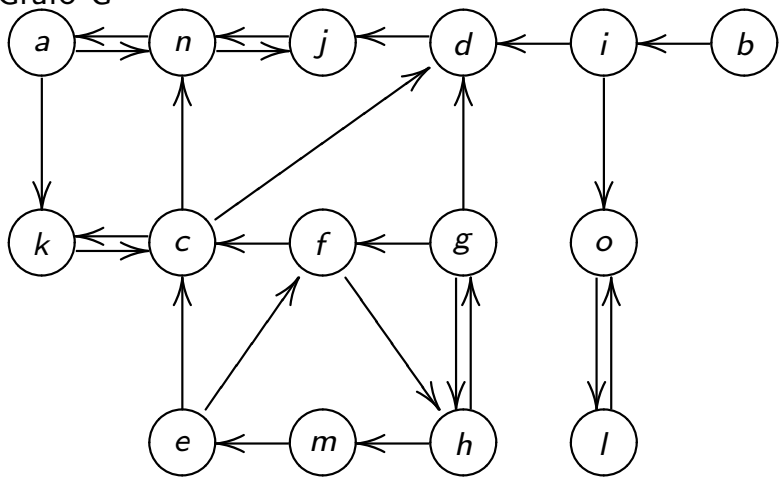
- A ordem dada por S para visita de G^T faz com que as componentes acessíveis de uma dada componente já estejam visitadas quando entra nessa componente.

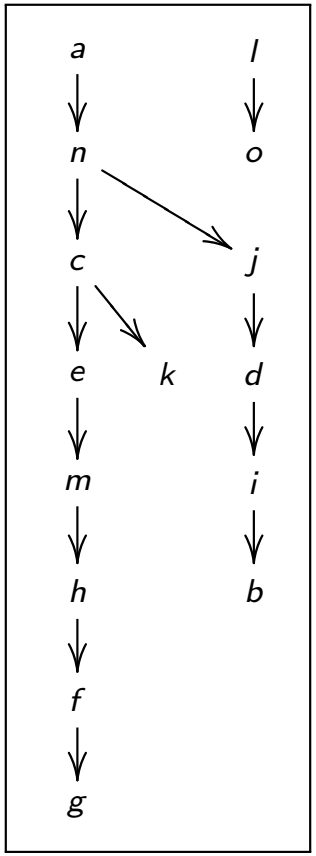
Componentes fortemente conexas (exemplo)

Grafo G



Grafo G^T





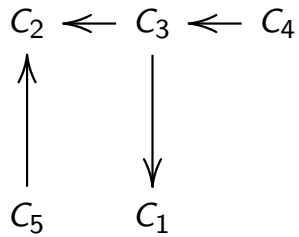
Pilha

l
o
a
n
j
d
i
b
c
k
e
m
h
f
g

Componentes fortemente conexas

$C_1 = \{l, o\}$
$C_2 = \{a, n, j, k, c, d\}$
$C_3 = \{i\}$
$C_4 = \{b\}$
$C_5 = \{e, f, h, g, m\}$

DAG componentes em G^T



Árvores geradoras com peso mínimo/máximo

Exemplo

Uma companhia de distribuição de gás natural pretende construir uma rede que assegure a distribuição a um certo número de locais a partir de um dado local. Dados os custos da ligação entre cada par de locais, há que determinar as ligações a efetuar de modo a reduzir os custos globais.

Resolução

- As árvores são os grafos não dirigidos conexos com menos ramos.
- Determinar uma **árvore geradora de peso mínimo** (*minimum spanning tree*) num grafo $G = (V, E, d)$ não dirigido, finito e **conexo** com valores associados aos ramos, em que $d : E \longrightarrow \mathbb{R}_0^+$ indica o valor associado a cada ramo.
- Designação alternativa: árvore de suporte de peso mínimo/máximo.
- **Algoritmos de Prim** (1957) e de **Kruskal** (1956).

  Baseiam-se em **estratégias "greedy"** (gulosas, ávidas, gananciosas).

  Em cada iteração, a seleção localmente ótima. Não haverá retrocesso para analisar outras possibilidades.

Algoritmo de Kruskal [1956] - minimum spanning tree

- São escolhidos sucessivamente os $|V| - 1$ ramos da árvore de suporte;
- os ramos são analisados por ordem crescente de valores de peso, e
- o ramo corrente só não fará parte da árvore de suporte se o grafo resultante da sua junção à floresta construída até esse passo ficar com um ciclo.

Algoritmo de Kruskal - pseudocódigo

Dados: Um grafo $G = (V, E, d)$ não dirigido, conexo, com valores nos ramos.
Resultado: O conjunto de ramos T na árvore mínima de suporte de G.

Ordenar E por ordem crescente de valores nos ramos.
$T \leftarrow \emptyset$; $\mathcal{C} \leftarrow \{\{v\} \mid v \in V\}$;
Enquanto $|T| \neq |V| - 1$ fazer
  Seja $<u, v> \in E$ o primeiro ramo não escolhido (na ordem considerada).
  Sejam C_u e C_v os elementos de $\mathcal{C}$ tais que $u \in C_u$ e $v \in C_v$.
  Se $C_u \neq C_v$ então $T \leftarrow T \cup \{<u, v>\}$; $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C_u, C_v\}) \cup \{C_u \cup C_v\}$;

Algoritmo de Kruskal - pseudocódigo

[CLRS 23.2]

MST_KRUSKAL(G,w)

$T \leftarrow \emptyset$
Para cada vértice $v \in G.V$
  MAKE_SET(v)
Ordenar as arestas de $G.E$ por ordem não decrescente de peso w
Para cada aresta $(u, v) \in G.E$ (por ordem não decrescente de peso) fazer
  Se FIND_SET(u) $\neq$ FIND_SET(v)
    $T \leftarrow T \cup \{(u, v)\}$
    UNION(u,v)

Algoritmo de Kruskal - pseudocódigo

Árvore geradora com peso mínimo:

AlgoritmoKruskal(G)

1. $Q \leftarrow$ Fila que representa E por ordem crescente de valores nos ramos;
2. $T \leftarrow \emptyset$;
3. $\mathcal{C} \leftarrow$ Init_Singletons(V);
4. Enquanto ($|T| \neq |V| - 1 \wedge$ QueueIsEmpty(Q) = `false`) fazer
5. $\quad \langle u, v \rangle \leftarrow$ Dequeue(Q);
6. $\quad$ Se FindSet($u, \mathcal{C}$) $\neq$ FindSet($v, \mathcal{C}$) então
7. $\quad\quad T \leftarrow T \cup \{\langle u, v \rangle\}$;
8. $\quad\quad$ Union($u, v, \mathcal{C}$);

A condição QueueIsEmpty(Q) = `false` é redundante se o grafo for conexo. Contudo, permite que o algoritmo possa ser aplicado a um grafo não conexo para obter a **floresta de árvores geradoras mínimas das suas componentes conexas**.

Árvore geradora com peso máximo: obtém-se por aplicação do algoritmo se se ordenar os ramos por ordem decrescente de peso inicialmente.

Algoritmo de Prim [1957] - Minimum spanning tree

- São escolhidos sucessivamente os $|V|$ vértices da árvore;
- em cada passo, é ligado, à sub-árvore já construída, o vértice que está **mais próximo** dos já nela incluídos.
- O primeiro vértice pode ser qualquer um dos vértices do grafo.

Árvore geradora com peso máximo: obtém-se por aplicação do algoritmo se em cada iteração se ligar o nó **mais afastado** dos que já estão na árvore (na implementação, usa *"heap de máximo"*).

Algoritmo de Prim – pseudocódigo

AlgoritmoPrim(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[u] \leftarrow \infty$; **ok[v]** $\leftarrow$ `false`; }	$\Theta(\lvert V\rvert)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmin($dist, \lvert V\rvert$);	$\Theta(\lvert V\rvert)$
$T \leftarrow \{\}$;	$O(1)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMin(Q);	$O(\log_2 \lvert V\rvert)$
$T \leftarrow T \cup \{(pai[v], v)\}$; **ok[v]** $\leftarrow$ `true`; /* $ok[v]$ indica se v já está em T */	$O(1)$
Para cada $w \in Adjs[v]$ fazer	
Se **ok[w]** $=$ `false` e $d(v, w) < dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
DecreaseKey($Q, w, dist[w]$);	$O(\log_2 \lvert V\rvert)$

NB: Apenas as distâncias dos nós que estão na fila podem ser alteradas

Complexidade Temporal $O(|E| \log |V|)$, se for suportado por uma **heap de mínimo**. Como G é conexo, $|E| \geq |V| - 1$. O ciclo "Enquanto" domina a complexidade, sendo:

$$O(\sum_{v \in V}(1+\log_2 |V|+|Adjs[v]| \log_2 |V|)) = O(|V| \log_2 |V|+|E| \log_2 |V|) = O(|E| \log_2 |V|)$$

Algoritmo de Prim – pseudocódigo

AlgoritmoPrim(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[u] \leftarrow \infty$; **ok[v]** $\leftarrow$ `false`; }	$\Theta(\|V\|)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmin($dist, \|V\|$);	$\Theta(\|V\|)$
$T \leftarrow \{\}$;	$O(1)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMin(Q);	$O(\log_2 \|V\|)$
$T \leftarrow T \cup \{(pai[v], v)\}$; **ok[v]** $\leftarrow$ `true`; /* $ok[v]$ indica se v já está em T */	$O(1)$
Para cada $w \in Adjs[v]$ fazer	
Se **ok[w]** $=$ `false` e $d(v, w) < dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
DecreaseKey($Q, w, dist[w]$);	$O(\log_2 \|V\|)$

NB: Apenas as distâncias dos nós que estão na fila podem ser alteradas

Complexidade Temporal $O(|E| \log |V|)$, se for suportado por uma **heap de mínimo**. Como G é conexo, $|E| \geq |V| - 1$. O ciclo "Enquanto" domina a complexidade, sendo:

$$O(\sum_{v \in V}(1+\log_2 |V|+|Adjs[v]| \log_2 |V|)) = O(|V| \log_2 |V|+|E| \log_2 |V|) = O(|E| \log_2 |V|)$$

Algoritmo de Prim – pseudocódigo

AlgoritmoPrim(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[u] \leftarrow \infty$; **ok[v]** $\leftarrow$ `false`; }	$\Theta(\lvert V\rvert)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmin($dist, \lvert V\rvert$);	$\Theta(\lvert V\rvert)$
$T \leftarrow \{\}$;	$O(1)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMin(Q);	$O(\log_2 \lvert V\rvert)$
$T \leftarrow T \cup \{(pai[v], v)\}$; **ok[v]** $\leftarrow$ `true`; /* $ok[v]$ indica se v já está em T */	$O(1)$
Para cada $w \in Adjs[v]$ fazer	
Se **ok[w]** $=$ `false` e $d(v, w) < dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
DecreaseKey($Q, w, dist[w]$);	$O(\log_2 \lvert V\rvert)$

NB: Apenas as distâncias dos nós que estão na fila podem ser alteradas

Complexidade Temporal $O(|E| \log |V|)$, se for suportado por uma **heap de mínimo**. Como G é conexo, $|E| \geq |V| - 1$. O ciclo "Enquanto" domina a complexidade, sendo:

$$O(\sum_{v \in V}(1+\log_2 |V|+|Adjs[v]| \log_2 |V|)) = O(|V| \log_2 |V|+|E| \log_2 |V|) = O(|E| \log_2 |V|)$$

Correção dos algoritmos de Prim e Kruskal

A correção dos algoritmos descritos resulta da propriedade seguinte

Propriedade das árvores geradoras de peso mínimo

Seja T uma árvore geradora mínima de um grafo $G = (V, E, d)$ não dirigido e conexo. Para toda a partição $\{V_1, V_2\}$ do conjunto de vértices V, a árvore T tem algum ramo $\langle v_1, v_2 \rangle$ tal que $v_1 \in V_1$, $v_2 \in V_2$, e $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Prova: (por redução ao absurdo) Seja T uma árvore geradora mínima de G e suponhamos que $\{V_1, V_2\}$ é uma partição de V tal que T não contém nenhum ramo $\langle v_1, v_2 \rangle$ com $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$, $v_1 \in V_1$ e $v_2 \in V_2$. Seja $\langle v_1, v_2 \rangle$ um tal ramo. Como T é uma árvore geradora de G, existe um e um só caminho entre v_1 e v_2 em T. Esse caminho tem que ter algum ramo $\langle x, y \rangle$ com $x \in V_1$ e $y \in V_2$, pois, caso contrário, os nós em V_1 (respectivamente, em V_2) só estariam ligados em T a nós em V_1 (respectivamente, em V_2), e a árvore T não seria conexa (o que é absurdo). Note-se que é possível que ou $x = v_1$ ou $y = v_2$. Pela hipótese inicial, $d(x, y) > d(v_1, v_2)$. Por outro lado, se substituirmos $\langle x, y \rangle$ em T por $\langle v_1, v_2 \rangle$, o grafo resultante ainda é uma árvore geradora de G e tem "peso" menor do que a árvore T, o que contradiz o facto de T ser mínima. Portanto, a árvore T tem de ter algum dos ramos de menor peso nesse corte (por definição, o **corte** determinado pela partição $\{V_1, V_2\}$ de V é o conjunto de ramos que ligam vértices de V_1 a vértices de V_2).

Correção dos algoritmos de Prim e Kruskal

A correção dos algoritmos descritos resulta da propriedade seguinte

Propriedade das árvores geradoras de peso mínimo

Seja T uma árvore geradora mínima de um grafo $G = (V, E, d)$ não dirigido e conexo. Para toda a partição $\{V_1, V_2\}$ do conjunto de vértices V, a árvore T tem algum ramo $\langle v_1, v_2 \rangle$ tal que $v_1 \in V_1$, $v_2 \in V_2$, e $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Prova: (por redução ao absurdo) Seja T uma árvore geradora mínima de G e suponhamos que $\{V_1, V_2\}$ é uma partição de V tal que T não contém nenhum ramo $\langle v_1, v_2 \rangle$ com $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$, $v_1 \in V_1$ e $v_2 \in V_2$. Seja $\langle v_1, v_2 \rangle$ um tal ramo. Como T é uma árvore geradora de G, existe um e um só caminho entre v_1 e v_2 em T. Esse caminho tem que ter algum ramo $\langle x, y \rangle$ com $x \in V_1$ e $y \in V_2$, pois, caso contrário, os nós em V_1 (respectivamente, em V_2) só estariam ligados em T a nós em V_1 (respectivamente, em V_2), e a árvore T não seria conexa (o que é absurdo). Note-se que é possível que ou $x = v_1$ ou $y = v_2$. Pela hipótese inicial, $d(x, y) > d(v_1, v_2)$. Por outro lado, se substituirmos $\langle x, y \rangle$ em T por $\langle v_1, v_2 \rangle$, o grafo resultante ainda é uma árvore geradora de G e tem "peso" menor do que a árvore T, o que contradiz o facto de T ser mínima. Portanto, a árvore T tem de ter algum dos ramos de menor peso nesse corte (por definição, o **corte** determinado pela partição $\{V_1, V_2\}$ de V é o conjunto de ramos que ligam vértices de V_1 a vértices de V_2).

Correção dos algoritmos de Kruskal e de Prim

Da propriedade anterior conclui-se que:

- no algoritmo de Prim, é *seguro* ligar o vértice v à sub-árvore já construída.

  Em cada iteração do ciclo "Enquanto", V_1 seria o conjunto dos vértices que já estão na sub-árvore e V_2 seria o conjunto dos restantes.

  Para cada $v \in V_2$, o valor de $dist[v]$ é o custo dos ramos mais leves com extremidade em v e que estão no corte definido por $\{V_1, V_2\}$. Este invariante é preservado pelo ciclo.

- no algoritmo de Kruskal, quando $\langle u, v \rangle$ é escolhido para ligar duas componentes, se tomarmos V_1 como os nós da componente que contém u e V_2 como os restantes nós, podemos concluir que $\langle u, v \rangle$ é *seguro*.

  Alguma árvore geradora mínima contém $\langle u, v \rangle$, pois este ramo tem peso mínimo no corte definido por $\{V_1, V_2\}$.

Caminhos mínimos em grafos com pesos positivos

Seja $G = (V, E, d)$ um grafo dirigido, finito e com pesos (*distâncias* ou *valores*) positivos associados aos ramos, $d(u, v) > 0$, para todo $(u, v) \in E$.

- A **distância associada a um percurso de u para v** é a soma das distâncias associadas aos ramos que constituem o percurso.
- Assumimos que a distância mínima de um nó do grafo a si mesmo é zero.

Dependendo da aplicação, podemos querer encontrar um:

- caminho mínimo de s para t, para **um par** $(s, t) \in V \times V$, com $s \neq t$;
- caminho mínimo **de s para cada um** dos outros nós, para $s \in V$ **fixo**;
- caminho mínimo de s para t, para **todos os pares** $(s, t) \in V \times V$, $s \neq t$.

Algoritmo de Dijkstra (1959)

Restrição de aplicabilidade: Os valores nos ramos têm de ser positivos.

Caminhos mínimos a partir de um nó origem s:

AlgoritmoDijkstra(G, s)

1. Para cada $v \in G.V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow \infty$;}
2. $dist[s] \leftarrow 0$;
3. $Q \leftarrow$ Mk_PQ_Heapmin$(dist, G.V)$;
4. Enquanto (PQ_Not_Empty(Q)) fazer
5. $\quad v \leftarrow$ ExtractMin(Q);
6. $\quad$ Para cada $w \in G.Adjs[v]$ fazer
7. $\quad\quad$ Se $dist[v] + G.d(v, w) < dist[w]$ então
8. $\quad\quad\quad dist[w] \leftarrow dist[v] + G.d(v, w)$;
9. $\quad\quad\quad pai[w] \leftarrow v$;
10. $\quad\quad\quad$ DecreaseKey$(Q, w, dist[w])$;

Melhoramento: Sair se $dist[v] = \infty$ na linha 5. (pois não há caminho de s os nós em $Q \cup \{v\}$)

Caminho mínimo de s para t, com s e t fixos: sair quando t é extraído de Q, colocando Se $(v = t)$ então retornar; entre as linhas 5 e 6.

Algoritmo de Dijkstra (1959)

Restrição de aplicabilidade: Os valores nos ramos têm de ser positivos.

Caminhos mínimos a partir de um nó origem s:

AlgoritmoDijkstra(G, s)

1. Para cada $v \in G.V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow \infty$;}
2. $dist[s] \leftarrow 0$;
3. $Q \leftarrow$ Mk_PQ_Heapmin$(dist, G.V)$;
4. Enquanto (PQ_Not_Empty(Q)) fazer
5. $\quad v \leftarrow$ ExtractMin(Q);
6. $\quad$ Para cada $w \in G.Adjs[v]$ fazer
7. $\quad\quad$ Se $dist[v] + G.d(v, w) < dist[w]$ então
8. $\quad\quad\quad dist[w] \leftarrow dist[v] + G.d(v, w)$;
9. $\quad\quad\quad pai[w] \leftarrow v$;
10. $\quad\quad\quad$ DecreaseKey$(Q, w, dist[w])$;

Melhoramento: Sair se $dist[v] = \infty$ na linha 5. (pois não há caminho de s os nós em $Q \cup \{v\}$)

Caminho mínimo de s para t, com s e t fixos: sair quando t é extraído de Q, colocando Se $(v = t)$ então retornar; entre as linhas 5 e 6.

Complexidade temporal do algoritmo de Dijkstra

```
AlgoritmoDijkstra(G, s)
1.  | Para cada v ∈ G.V fazer { pai[v] ← NULL; dist[v] ← ∞;}
2.  | dist[s] ← 0;
3.  | Q ← Mk_PQ_Heapmin(dist, G.V);
4.  | Enquanto (PQ_Not_Empty(Q)) fazer
5.  |       v ← ExtractMin(Q);
6.  |       Para cada w ∈ G.Adjs[v] fazer
7.  |           Se dist[v] + G.d(v, w) < dist[w] então
8.  |               dist[w] ← dist[v] + G.d(v, w);
9.  |               pai[w] ← v;
10. |               DecreaseKey(Q, w, dist[w]);
```

Se G for dado por **listas de adjacências** e a fila de prioridade Q for suportada por uma **heap de mínimo**, tem complexidade temporal $O((|V| + |E|) \log_2 |V|)$, pois é dominada pelo ciclo "Enquanto":

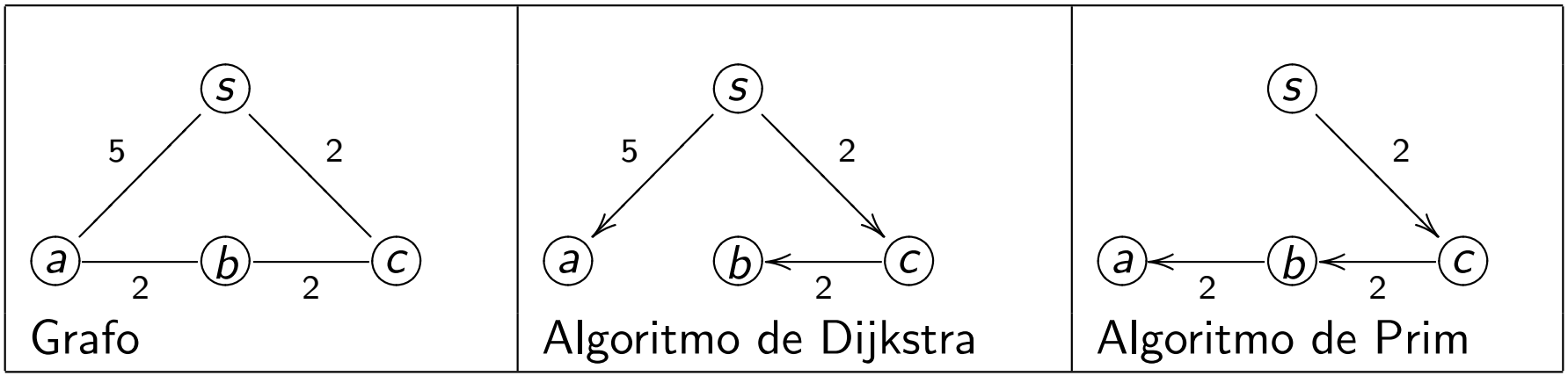
$$O(\sum_{v \in V} (1 + \log_2 |V| + |Adjs[v]| \log_2 |V|)) = O((|V| + |E|) \log_2 |V|).$$

Mantemos a expressão assim pois não sabemos qual é a ordem de grandeza de $|E|$ relativamente a $|V|$.

Árvores de peso mínimo / Árvores de caminhos mínimos

O algoritmo de Dijkstra pode ser aplicado a grafos $G = (V, E, d)$ não dirigidos (que podem ser vistos como grafos dirigidos simétricos). Quando G é conexo, **a árvore dos caminhos mínimos com origem em** s contém todos os nós mas **nem sempre é uma árvore geradora mínima de** G. Consequentemente,

o algoritmo de Prim **não pode ser usado** para determinar os caminhos mínimos com origem em s.



Prova de correção do algoritmo de Dijkstra

Usamos $\delta(s, v)$ para denotar **a distância mínima** de s a v em G, para $v \in V$.

O algoritmo de Dijkstra mantém o invariante seguinte, para $k \geq 1$.
No **final da iteração k do ciclo "Enquanto"**, seja $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q. Tem-se:

1. $dist[r] = \delta(s, r)$, para todo $r \in \mathcal{M}_k$;
2. $dist[r]$ é a distância mínima de s a r em G se os percursos só puderem passar por vértices de $\mathcal{M}_k \cup \{r\}$, para todo $r \in \mathcal{Q}_k$.

Prova (por indução sobre $k \geq 1$):
(Caso de base) Para $k = 1$, o nó que sai de Q (linha 5) é s e, no bloco 6-10, $dist$ é atualizado, ficando $dist[w] = d(s, w)$, para todo $w \in Adjs[s]$ (e mantém $dist[r] = \infty$ para os restantes $r \neq s$). Logo, as condições 1. e 2. verificam-se, já que, $\mathcal{M}_1 = \{s\}$, $dist[s] = 0 = \delta(s, s)$, por definição, e para $r \in \mathcal{Q}_1 = V \setminus \{s\}$, o caminho mínimo de s para r que só pode passar por $M_1 \cup \{r\} = \{s, r\}$ é dado por (s, r) ou não existe, para $r \in V \setminus \{s\}$.

Prova de correção do algoritmo de Dijkstra

Usamos $\delta(s, v)$ para denotar **a distância mínima** de s a v em G, para $v \in V$.

O algoritmo de Dijkstra mantém o invariante seguinte, para $k \geq 1$.
No **final da iteração k do ciclo "Enquanto"**, seja $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q. Tem-se:

1. $dist[r] = \delta(s, r)$, para todo $r \in \mathcal{M}_k$;
2. $dist[r]$ é a distância mínima de s a r em G se os percursos só puderem passar por vértices de $\mathcal{M}_k \cup \{r\}$, para todo $r \in \mathcal{Q}_k$.

Prova (por indução sobre $k \geq 1$):
(Caso de base) Para $k = 1$, o nó que sai de Q (linha 5) é s e, no bloco 6-10, $dist$ é atualizado, ficando $dist[w] = d(s, w)$, para todo $w \in Adjs[s]$ (e mantém $dist[r] = \infty$ para os restantes $r \neq s$). Logo, as condições 1. e 2. verificam-se, já que, $\mathcal{M}_1 = \{s\}$, $dist[s] = 0 = \delta(s, s)$, por definição, e para $r \in \mathcal{Q}_1 = V \setminus \{s\}$, o caminho mínimo de s para r que só pode passar por $M_1 \cup \{r\} = \{s, r\}$ é dado por (s, r) ou não existe, para $r \in V \setminus \{s\}$.

Prova de correção do algoritmo de Dijkstra (cont.)

(cont.) Prova (por indução sobre $k \geq 1$):
(Hereditariedade) Suponhamos, como hipótese de indução, que o invariante se verifica no final da iteração k, para $k \geq 1$ fixo, e que $\mathcal{M}_k \neq V$, ou seja, $Q \neq \{\}$. Vamos mostrar que então o invariante se verifica no final da iteração $k+1$.

Iremos analisar os dois casos seguintes:

- (Caso A) não existem vértices em $\mathcal{Q}_k$ acessíveis de s
- (Caso B) existem vértices em $\mathcal{Q}_k$ acessíveis de s

Caso A

Todo $r \in \mathcal{Q}_k$ está, por convenção, a distância mínima ∞ de s e, de acordo com o invariante, no final da iteração k, tem-se $dist[r] = \infty$ para todo $r \in \mathcal{Q}_k$ (como se definiu inicialmente). O vértice v que sai da fila na iteração $k+1$ tem $dist[v] = \infty$ e, assim, como $dist[v] + d(v, w) = \infty + d(v, w) = \infty$, não altera o valor de $dist[w]$, para nenhum $w \in Adjs[v]$. Logo, todos os vértices em $r \in \mathcal{Q}_{k+1}$ se manterão com $dist[r] = \infty$, e a condição 2. do invariante mantém-se.

Prova de correção do algoritmo de Dijkstra (cont.)

(cont.) Prova (por indução sobre $k \geq 1$):
Seja $\hat{d}(\gamma)$ o comprimento de um percurso γ, ou seja, $\hat{d}(\gamma) = \sum_{(x,y)\in\gamma} d(x,y)$.

Caso B

- Seja $w \in \mathcal{Q}_k$ um vértice que se encontra a distância mínima de s, ou seja tal que $\delta(s,w) = \min\{\delta(s,r) \mid r \in \mathcal{Q}_k\}$. Seja $\gamma_{s,w}$ um **caminho mínimo** de s para w em G. Logo, tem-se $\hat{d}(\gamma_{s,w}) = \delta(s,w)$.
- Se $\gamma_{s,w}$ **só tem um ramo**, então $w \in Adjs[s]$, e do (caso de base, $k = 1$) segue $dist[w] = \delta(s,w) = \hat{d}(\gamma_{s,w})$.
- Se $\gamma_{s,w}$ **tem mais do que um ramo**, seja u **o vértice que precede** w no caminho $\gamma_{s,w}$, e $\gamma_{s,u}$ o sub-caminho até u.
  - $\delta(s,u) = \hat{d}(\gamma_{s,u})$
  - $u \notin Q_k$ pois $\delta(s,w) = \delta(s,u) + d(u,w)$ implica que $\delta(s,u) < \delta(s,w)$. Se u estivesse em Q_k seria escolhido em vez de w. Então $u \in M_k$, e, pela hipótese de indução, $dist[u] = \delta(s,u)$ e $dist[w] \leq \hat{d}(\gamma_{s,w})$. Logo, $dist[w] = \hat{d}(\gamma_{s,w}) = \delta(s,w)$.

Prova de correção do algoritmo de Dijkstra (cont.)

Caso B (cont.)

- Portanto, no algoritmo de Dijkstra, o **vértice v que se retira de Q na iteração $k+1$** satisfaz $dist[v] = \delta(s, v)$ (é um vértice de Q_k que está a distância mínima de s, dado que $dist[v]$ não seria alterado em nenhuma das iterações seguintes). Como $\mathcal{M}_{k+1} = \mathcal{M}_k \cup \{v\}$, a condição 1. do invariante verifica-se no final da iteração $k+1$.
  - Importa observar que se $r \in Adjs[v] \cap \mathcal{M}_k$, o valor de $dist[r]$ não pode ser reduzido na iteração $k+1$ pois, pela hipótese de indução, $dist[r] = \delta(s, r)$ e, portanto, $dist[v] + d(v, r) \geq dist[r]$, por definição de caminho mínimo.
- Resta ver que, para todo $r \in \mathcal{Q}_{k+1}$, no final da iteração $k+1$, o valor de $dist[r]$ é a distância mínima de s a r se os percursos só puderem passar por vértices de $\mathcal{M}_{k+1} \cup \{r\}$. Tais percursos são caminhos sendo:
  - ou *caminhos mínimos de s para r que não passam por* $Q_k \setminus \{r\}$

    Nesse caso, pela hipótese de indução, $dist[r]$ é já a distância mínima de s a r com essa restrição.
  - ou *caminhos mínimos que passam em v mas não em* $Q_k \setminus \{r, v\}$

    Notar que $Q_k \setminus \{r, v\} = Q_{k+1} \setminus \{r\}$. Se se verificar o segundo caso (mas não o primeiro), tal caminho mínimo passa por v e, por ser mínimo terá de ser da forma $\Gamma_{s,v}, (v, r)$, para $\Gamma_{s,v}$ mínimo. Mas, r é adjacente a v e $dist[v] + d(v, r) = \delta(s, v) + d(v, r) = \hat{d}(\Gamma_{s,v}, (v, r)) < dist[r]$. Portanto, $dist[r]$ é atualizado no bloco 6-10 na iteração $k+1$ ficando com $\hat{d}((\Gamma_{s,v}, (v, r)))$.

Caminhos de capacidade máxima (adaptação do Algoritmo de Dijkstra)

Seja $G = (V, E, c)$ um grafo dirigido, finito, $c(u, v) \geq 0$ indica a **capacidade** do ramo (u, v). A **capacidade de um percurso** é o **mínimo** das capacidades dos ramos que constituem o percurso.

Problema:
Para um nó origem s, determinar um *percurso com capacidade máxima* de s para v, para cada $v \neq s$.

CaminhosCapacidadeMaxima(G, s)

```
1.  Para cada v ∈ V fazer { pai[v] ← NULL; cap[v] ← 0;}
2.  cap[s] ← ∞;
3.  Q ← Mk_PQ_HeapMax(cap, V);
4.  Enquanto (PQ_Not_Empty(Q)) fazer
5.      v ← ExtractMax(Q);
6.      Para cada w ∈ Adjs[v] fazer
7.          Se min(cap[v], c(v, w)) > cap[w] então
8.              cap[w] ← min(cap[v], c(v, w));
9.              pai[w] ← v;
10.             IncreaseKey(Q, w, cap[w]);
```

Caminhos de capacidade máxima (adaptação do Algoritmo de Dijkstra)

- **Complexidade temporal**
  - Se G for dado por **listas de adjacências** e a fila de prioridade Q for suportada por uma **heap de máximo**, tem complexidade temporal $O((|V| + |E|) \log_2 |V|)$, como o algoritmo de Dijkstra.

- **Correção:** o ciclo "Enquanto" preserva o invariante seguinte, para todo $k \geq 1$: sendo $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q, no final da iteração k, tem-se
  1. para $r \in \mathcal{M}_k$, o valor $cap[r]$ é a capacidade máxima dos percursos de s para r em G, para todo $r \in \mathcal{M}_k$;
  2. para $r \in Q_k$, o valor $cap[r]$ é a capacidade máxima dos percursos de s para r em G se os percursos só puderem passar por nós de $\mathcal{M}_k \cup \{r\}$.

- **Propriedade que explora:** um percurso γ_{st} de capacidade máxima **não tem de ter subestrutura ótima**. Mas, é verdade que se $\gamma_{st} = \gamma_{sv}\gamma_{vt}$, para algum v, **podemos substituir** cada um dos percursos γ_{sv} e γ_{vt} por caminhos $\gamma^{\star}_{sv}$ e $\gamma^{\star}_{vt}$ de capacidade máxima.

Caminhos de capacidade máxima em grafos não dirigidos

Propriedade

Se $G = (V, E, c)$ for um grafo não dirigido e conexo, a árvore geradora de **peso máximo criada a partir da raíz** s por adaptação do algoritmo de Prim contém um caminho de capacidade máxima de s para v, para cada $v \in V \setminus \{s\}$.

- Por isso, em instâncias deste tipo, o algoritmo de Prim (adaptado para obter árvores de peso máximo) seria uma alternativa ao que apresentámos acima.
- Esta propriedade resulta da definição de caminho de capacidade máxima e da seguinte propriedade estrutural das árvores de suporte de peso máximo:

  Seja T uma árvore geradora de peso máximo de um grafo $G = (V, E, d)$ não dirigido e conexo. Qualquer que seja a partição $\{V_1, V_2\}$ do conjunto de vértices V, a árvore T tem algum ramo $\langle v_1, v_2 \rangle$ com $v_1 \in V_1$ e $v_2 \in V_2$ e tal que $d(v_1, v_2) = \max\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Algoritmo de Floyd-Warshall

Problema:
Determinar o comprimento do caminho mínimo de s para t, para **todos os pares** $(s,t) \in V \times V$, $s \neq t$.

- Pode ser resolvido usando o algoritmo de Dijkstra
  - Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para determinar $D^{\star}_{ij}$, para todo j. Complexidade: $O(|V|(|E|+|V|)\log_2|V|)$.
  - Para grafos densos, com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.
- Mas, o **algoritmo de Floyd-Warshall** (1962), tem complexidade $\Theta(n^3)$.

  Supõe-se que inicialmente $D_{ii} = 0$, $D_{ij} = d(i,j)$, se $(i,j) \in E$; e, caso contrário, $D_{ij} = \infty$ se $i \neq j$

```
AlgoritmoFloyd-Warshall(D, n)
    Para k ← 1 até n fazer
        Para i ← 1 até n fazer
            Para j ← 1 até n fazer
                Se D[i,j] > D[i,k] + D[k,j] então D[i,j] ← D[i,k] + D[k,j];
```

Algoritmo de Floyd-Warshall

Problema:
Determinar o comprimento do caminho mínimo de s para t, para **todos os pares** $(s,t) \in V \times V$, $s \neq t$.

- Pode ser resolvido usando o algoritmo de Dijkstra
  - Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para determinar $D^{\star}_{ij}$, para todo j. Complexidade: $O(|V|(|E|+|V|)\log_2|V|)$.
  - Para grafos densos, com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.
- Mas, o **algoritmo de Floyd-Warshall** (1962), tem complexidade $\Theta(n^3)$.

  Supõe-se que inicialmente $D_{ii} = 0$, $D_{ij} = d(i,j)$, se $(i,j) \in E$; e, caso contrário, $D_{ij} = \infty$ se $i \neq j$

```
AlgoritmoFloyd-Warshall(D, n)
    Para k ← 1 até n fazer
        Para i ← 1 até n fazer
            Para j ← 1 até n fazer
                Se D[i,j] > D[i,k] + D[k,j] então D[i,j] ← D[i,k] + D[k,j];
```