II. Algoritmos sobre Grafos

Tópicos:

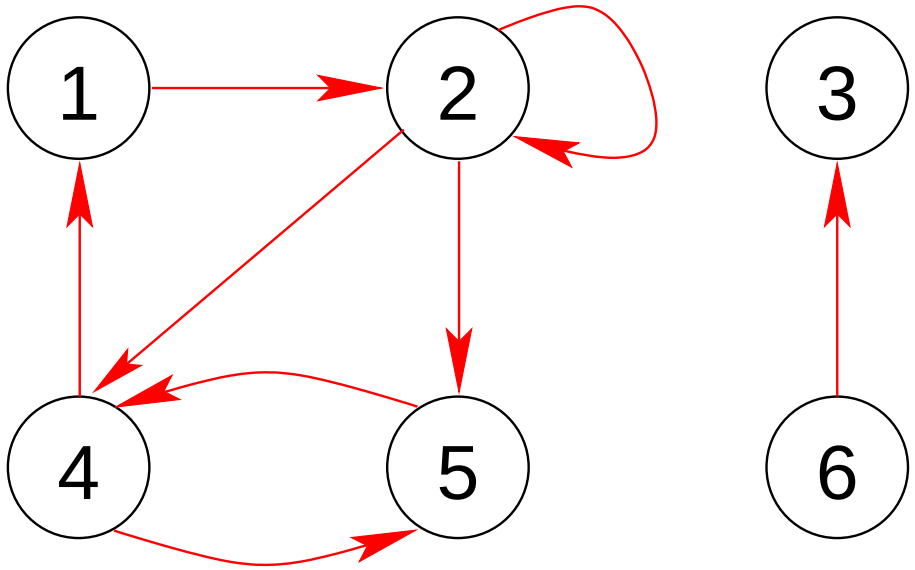
- Grafos: conceitos básicos e representação em memória de computador
- Algoritmos elementares: pesquisas (em largura e em profundidade)
- Análise amortizada de algoritmos; análise agregada
- Algoritmos para determinação de árvores geradoras mínimas
- Algoritmos para determinação de caminhos mais curtos
- Estratégias algorítmicas: algoritmos "greedy"
- Algoritmos para determinação do fecho transitivo de um grafo

Conceitos Básicos

Um grafo orientado é um par (V, E) com V um conjunto finito de *vértices* ou *nós* e E uma relação binária em V – o conjunto de *arcos* ou *arestas* do grafo.

Exemplo: $V = \{1, 2, 3, 4, 5, 6\}$,

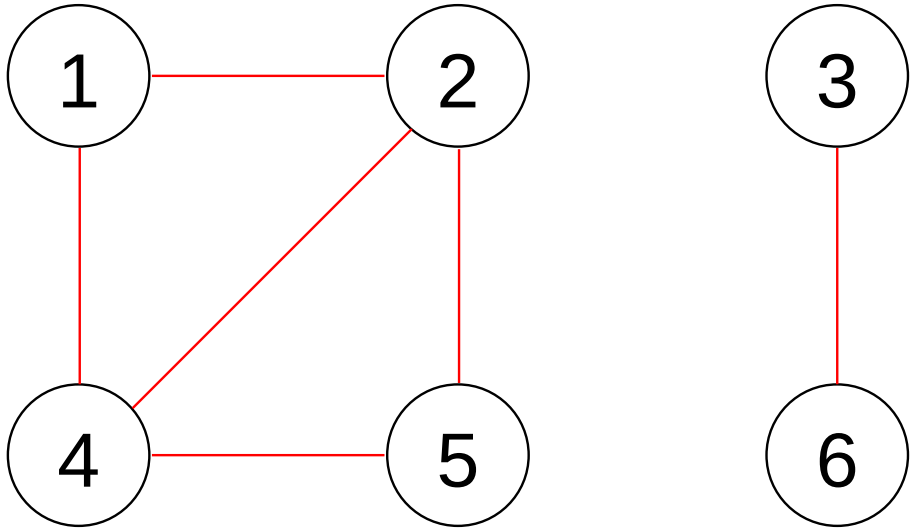
$E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$



Num grafo não-orientado o conjunto E é constituído por pares não-ordenados (conjuntos com 2 elementos). Assim, (i, j) e (j, i) correspondem ao *mesmo arco*.

Exemplo: $V = \{1, 2, 3, 4, 5, 6\}$,

$E = \{(1, 2), (1, 4), (2, 4), (2, 5), (3, 6), (4, 5)\}$



Notas e terminologia

- Um arco (i, i) designa-se por *anel*. Os anéis são interditos nos grafos não-orientados.

- i e j são respectivamente os vértices de *origem* e de *destino* do arco (i, j). j diz-se *adjacente* a i.

- A relação de adjacência pode não ser simétrica num grafo orientado.

- O *grau de entrada* (resp. *de saída*) de um nó num grafo orientado é o número de arcos com destino (resp. origem) no nó. O *grau* do nó é a soma de ambos.

Caminhos em Grafos

Num grafo (V, E), um caminho do vértice m para o vértice n é uma sequência de vértices $\langle v_0, v_1, \ldots v_k \rangle$ tais que

- $v_i \in V$ para todo o $i \in \{0, \ldots, k\}$
- $(v_i, v_{i+1}) \in E$ para todo o $i \in \{0, \ldots, k-1\}$
- $v_0 = m$ e $v_k = n$

Um caminho diz-se *simples* se todos os seus vértices são distintos. O *comprimento* de um caminho é o número de arcos nele contidos: $(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$.

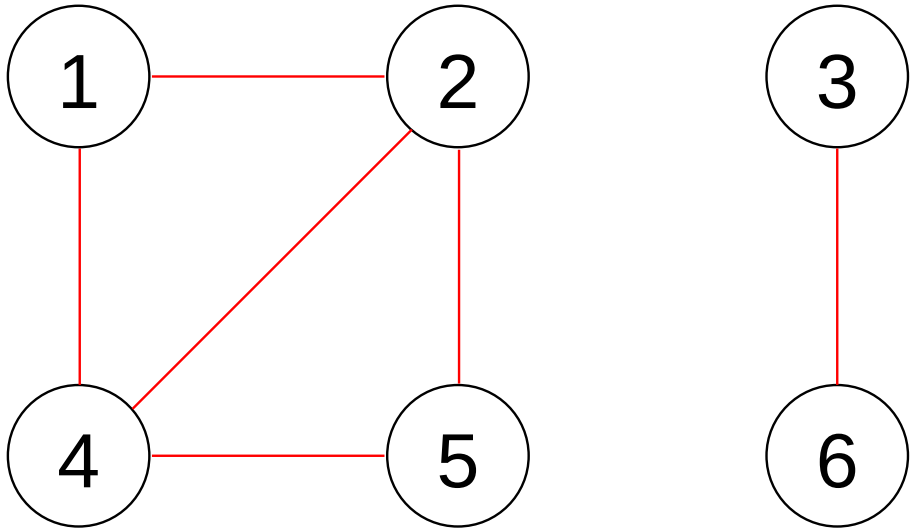
Um *sub-caminho* de um caminho é uma qualquer sua sub-sequência contígua.

Existe sempre um caminho de comprimento 0 de um vértice para si próprio; um *ciclo* é um caminho de comprimento ≥ 1 com início e fim no mesmo vértice. Um grafo diz-se *acíclico* se não contém ciclos.

Componentes Ligados

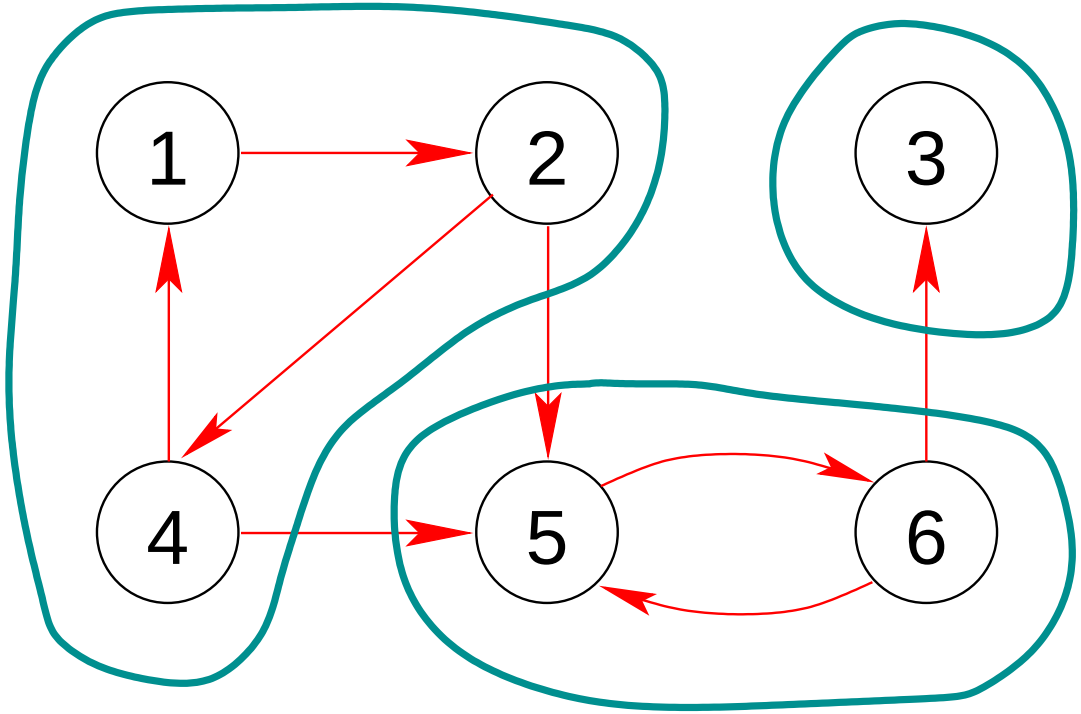
Um grafo (V', E') é um sub-grafo de (V, E) sse $V' \subseteq V$ e $E' \subseteq E$.

Um grafo não-orientado diz-se *ligado* se para todo o par de vértices existe um caminho que os liga. Os *componentes ligados* de um grafo são os seus maiores sub-grafos ligados.



Componentes Fortemente Ligados

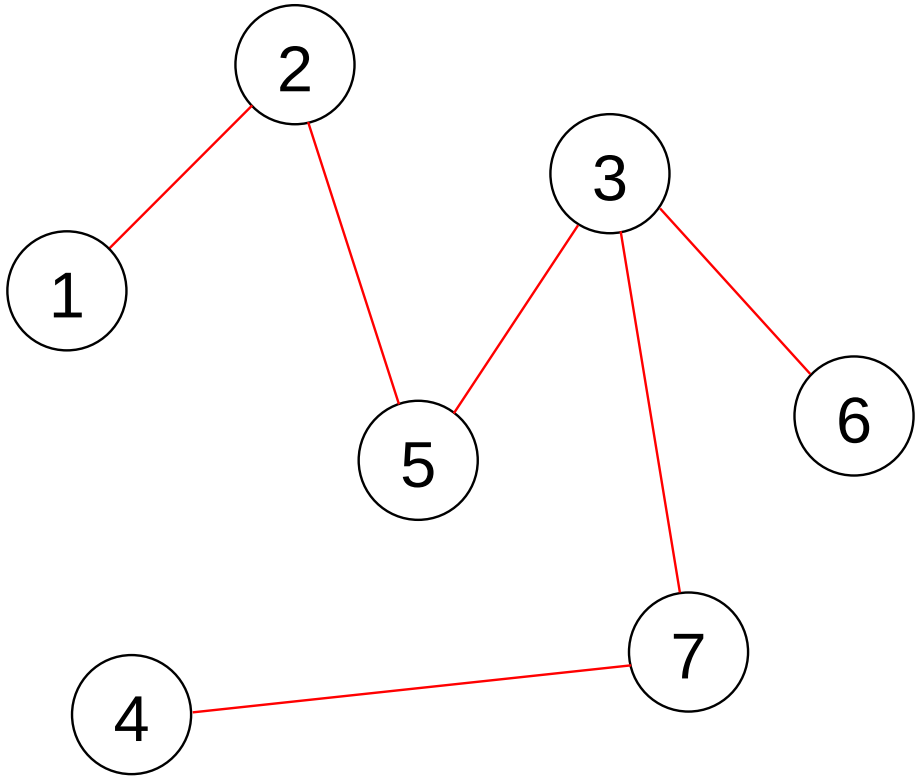
Um grafo orientado diz-se *fortemente ligado* se para todo o par de vértices m, n existem caminhos de m para n e de n para m. Os *componentes fortemente ligados* de um grafo são os seus maiores sub-grafos fortemente ligados.



Árvores

Uma floresta é um grafo *não-orientado acíclico*.

Uma árvore é um grafo *não-orientado*, *acíclico*, e *ligado*.



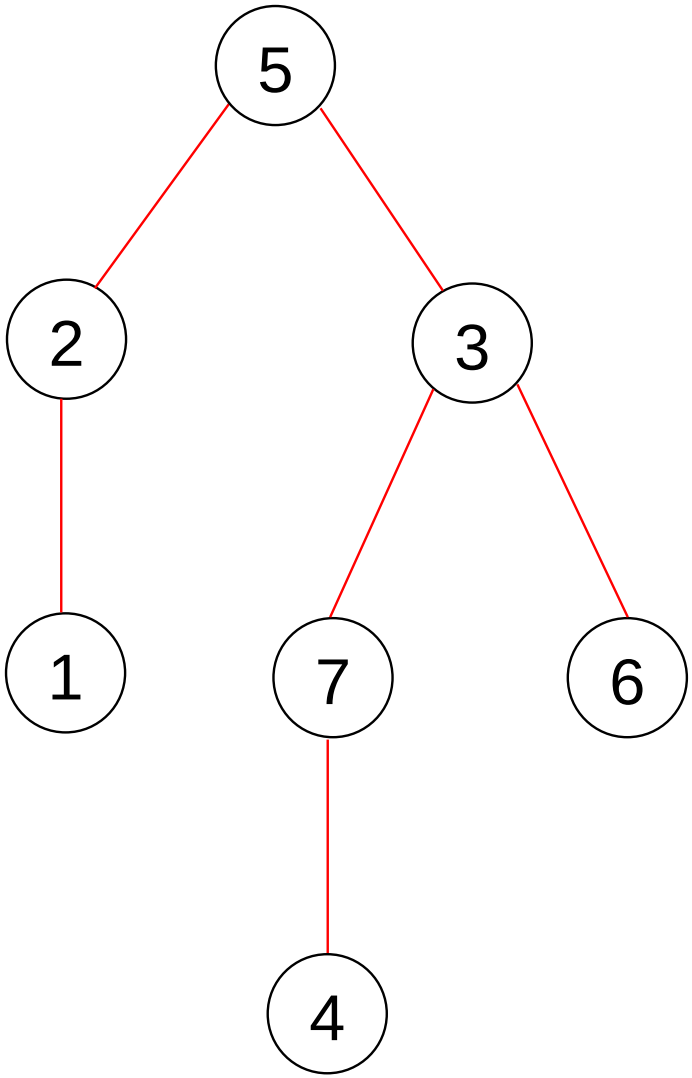
Árvores com Raíz

A escolha de um nó arbitrário para *raíz* de uma árvore define noções de *descendentes* de um nó e de *sub-árvore* com raíz num nó.

Existe um caminho único da raíz para qualquer nó.

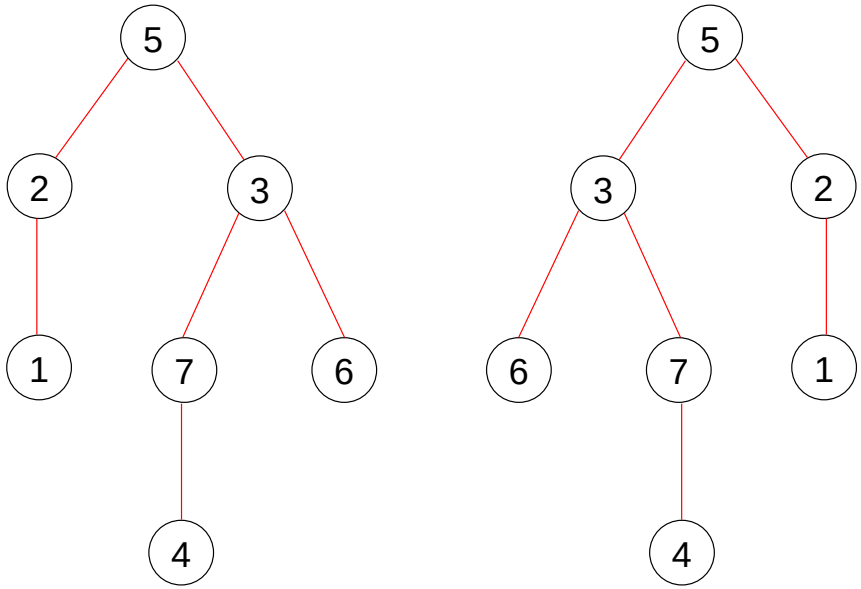
Uma árvore com raíz pode também ser vista como um grafo orientado.

$\Rightarrow$ com que restrição?



Árvores Ordenadas

Uma árvore ordenada é uma árvore com raíz em que a ordem dos descendentes de cada nó é relevante. As figuras seguintes representam a mesma árvore com raíz, mas árvores ordenadas diferentes.

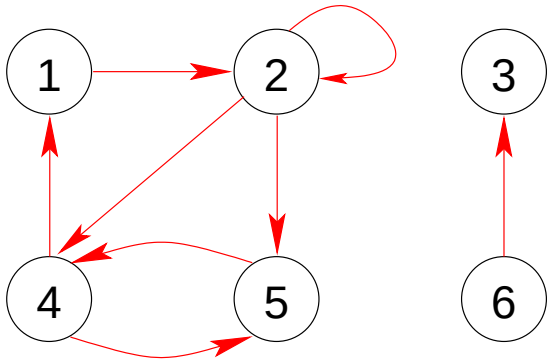


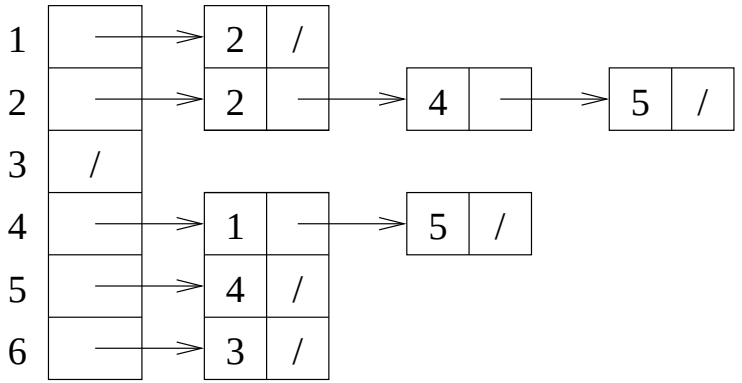
$\Rightarrow$ Como descrever uma árvore binária de pesquisa?

Representação de Grafos em Computador

Como representar um grafo orientado?

A representação mais usual é como um conjunto de listas de adjacências. Trata-se de uma representação compacta, útil para grafos em que o número de arcos $|E|$ seja pequeno (muito menor que $|V|^2$) – grafos *esparsos*.





Consiste num vector Adj de $|V|$ listas ligadas. A lista $Adj[i]$ contém todos os vértices j tais que $(i, j) \in E$, i.e., todos os vértices adjacentes a i.

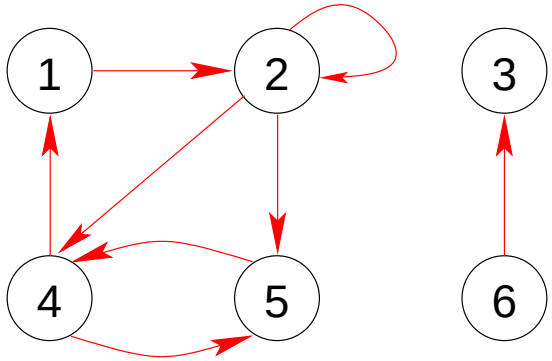
Representação por Listas de Adjacências

Notas:

- A soma dos comprimentos das listas ligadas é $|E|$.
- Se o grafo for *pesado* (i.e., se contiver informação associada aos arcos), o peso de cada arco pode ser incluído no nó respectivo numa das listas ligadas.
- No caso de um grafo não-orientado, esta representação pode ser utilizada desde que antes se converta o grafo num grafo orientado, substituindo cada arco (não-orientado) por um par de arcos (orientados).
- Neste caso a representação contém informação redundante e o comprimento total das listas é $2|E|$.
- Em qualquer dos casos o espaço de memória necessário é $\Theta(V + E)$.

Representação de Grafos em Computador

Uma outra possibilidade é a representação por uma matriz de adjacências. Trata-se de uma representação estática, e por isso apropriada para grafos *densos* – em que $|E|$ se aproxima de $|V|^2$.



	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	1	0	1	1	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0

Trata-se de uma matriz de dimensão $|V| \times |V|$, $A = (a_{ij})$ com $a_{ij} = 1$ se $(i, j) \in E$; $a_{ij} = 0$ em caso contrário.

Representação por Matrizes de Adjacências

Notas:

- Se o grafo for *pesado* o peso de cada arco pode ser incluído na respectiva posição da matriz, em vez de 1.

- No caso de um grafo não-orientado a matriz de adjacências é *simétrica*, e é possível armazenar apenas o triângulo acima da diagonal principal.

- Uma vantagem em relação às listas de adjacências: é imediato verificar a adjacência de dois nós (sem ter que percorrer uma lista ligada).

- O espaço de memória necessário é $\Theta(V^2)$ – independente do número de arcos.

Pesquisas em Grafos: Pesquisa em Largura

Dado um grafo $G = (V, E)$ e um seu vértice s, um algoritmo de pesquisa explora o grafo passando por todos os vértices alcançáveis a partir de s. O algoritmo de pesquisa em largura em particular:

- Calcula a distância (= menor número de arcos) de s a cada vértice;
- Produz uma árvore (sub-grafo de G) com raíz s contendo todos os vértices alcançáveis a partir de s;
- Nessa árvore o caminho da raíz s a cada vértice corresponde ao caminho mais curto entre os dois nós.
- Algoritmo para grafos orientados e não-orientados.

Estes algoritmos designam-se também por algoritmos de travessia de grafos.

Pesquisa em Largura num Grafo ("Breadth-first Search")

Este algoritmo utiliza a seguinte estratégia para efectuar a travessia do grafo:

- *Todos os vértices à distância k de s são visitados antes de qualquer vértice à distância $k+1$ de s.*

O algoritmo pinta os vértices (inicialmente brancos) de cinzento ou preto:

- um vértice branco ainda não foi *descoberto*;
- um vértice cinzento já foi visitado mas pode ter adjacentes ainda não descobertos (brancos);
- um vértice preto só tem adjacentes já descobertos (i.e., cinzentos ou pretos).

Os cinzentos correspondem à *fronteira* entre descobertos e não-descobertos.

A árvore de travessia em largura é expandida atravessando-se a lista de adjacências de cada vértice cinzento u; para cada vértice adjacente v acrescenta-se à árvore o arco (u, v).

Algoritmo de Pesquisa em Largura

```
void BFS((V,E), s) {                    /* G = (V,E) */
  for (u ∈ V, u ≠ s)
    { color[u] = WHITE; d[u] = ∞; parent[u] = NIL; }
  color[s] = GRAY; d[s] = 0; parent[s] = NIL;
  initialize_queue(Q); enqueue(Q,s);
  while (!is_empty(Q)) {
    u = dequeue(Q);
    for (v ∈ Adj(u))
      if (color[v] == WHITE) {
        color[v] = GRAY;
        d[v] = d[u]+1;
        parent[v] = u;
        enqueue(Q,v);
      }
    color[u] = BLACK;
  }
}
```

Notas (implementação)

- Utiliza-se uma fila de espera – estrutura linear FIFO – que se manipula através das funções
  - `void initialize_queue(Queue)`
  - `void enqueue(Queue, Vertex)`
  - `Vertex dequeue(Queue)`
  - `bool is_empty(Queue)`

- Outras estruturas de dados: vectores
  - `color[]` para guardar as *cores* dos vértices;
  - `d[]` para as *distâncias* a s;
  - `parent[]` para o *pai* de cada vértice na árvore construída.

- Normalmente utiliza-se uma representação por listas de adjacências (daí a utilização dos vectores anteriores . . .);

- $Adj(u)$ denota o conjunto dos vértices adjacentes a u.

Observações

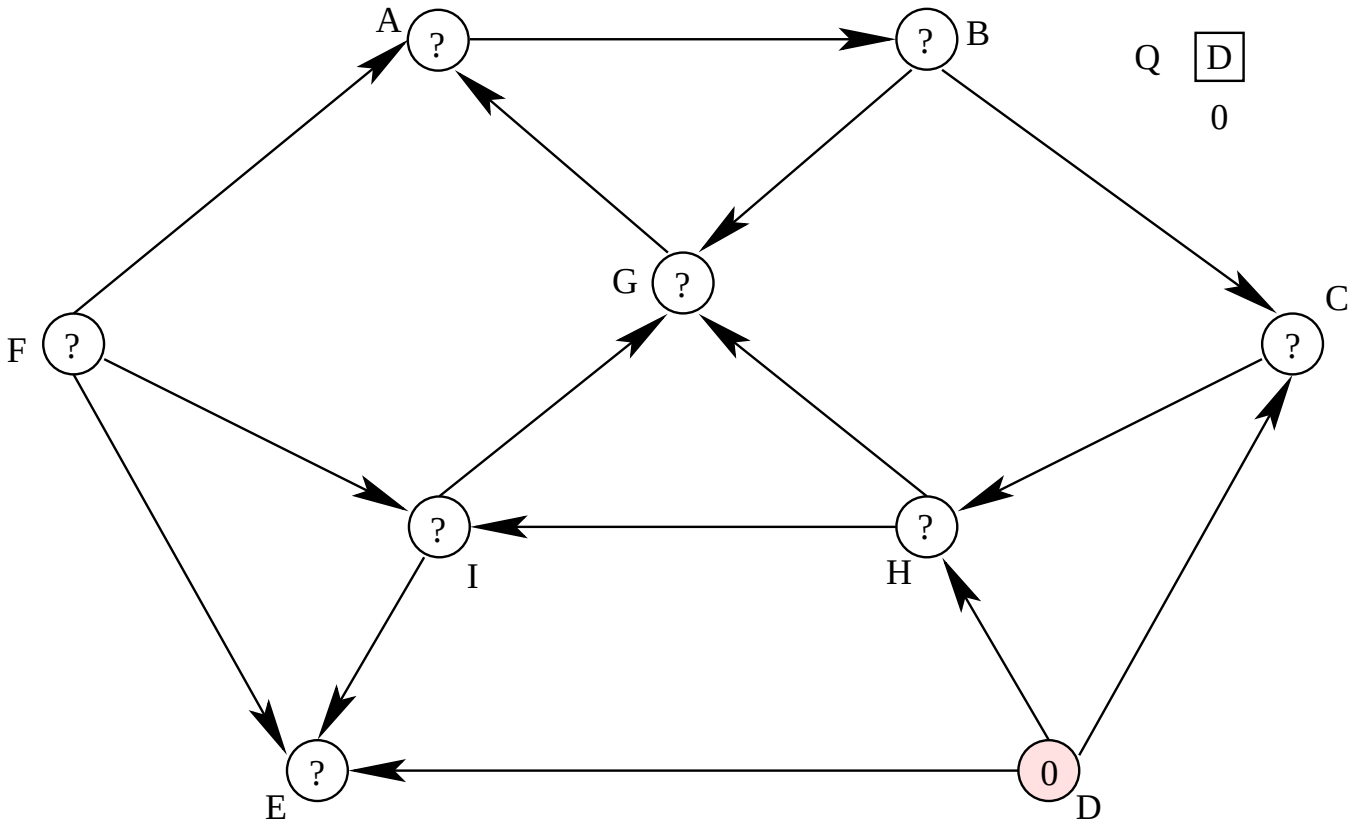
Invariante do ciclo `while`: *no início de cada iteração Q contém exactamente os vértices cinzentos.*

- **Inicialização**: Antes da primeira iteração Q contém apenas o vértice s.
- **Preservação**: Quando um vértice muda para cinzento entra para a fila; quando sai passa a ser preto.

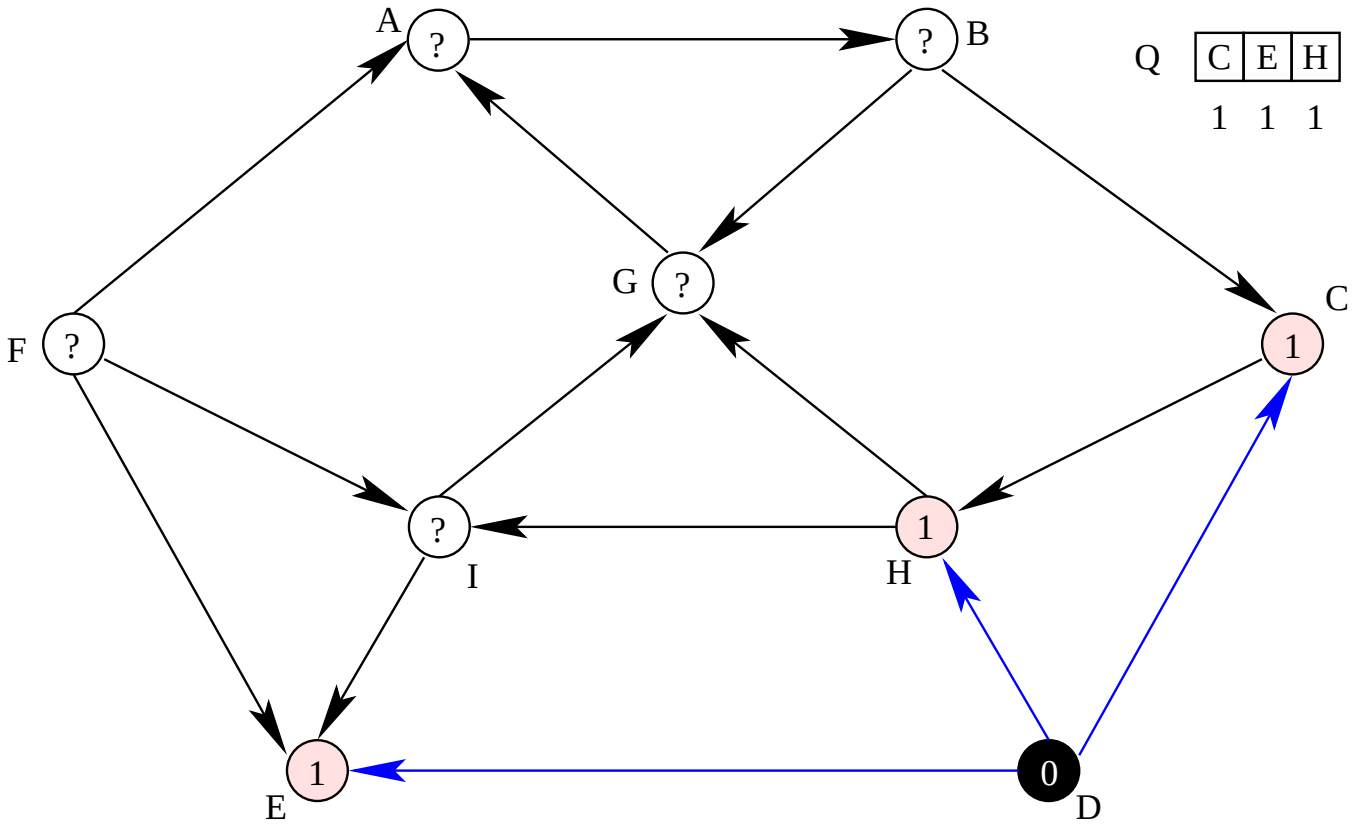
A árvore construída não é única, dependendo da ordem pela qual são visitados os vértices adjacentes a um determinado vértice.

No entanto as distâncias de cada vértice a s são únicas.

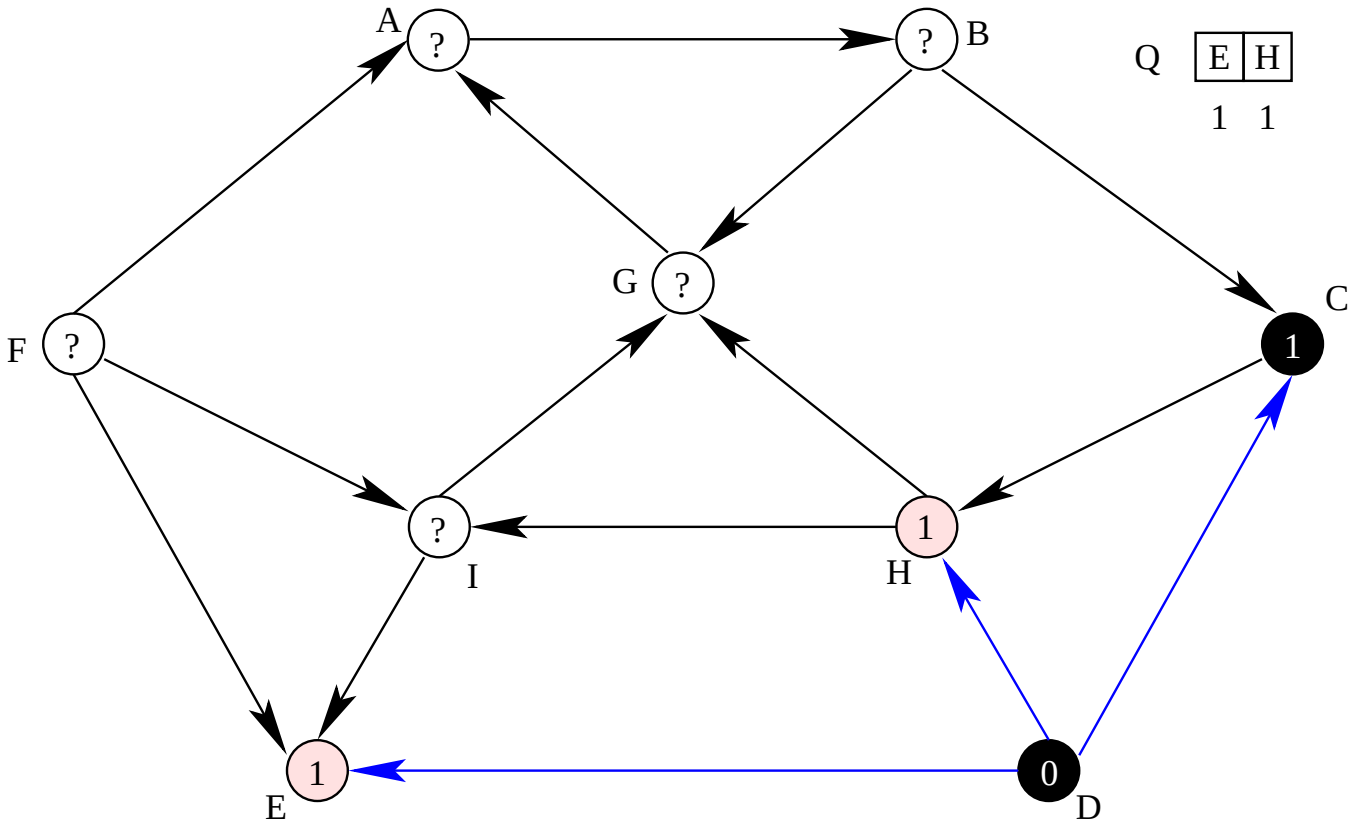
Exemplo de Execução



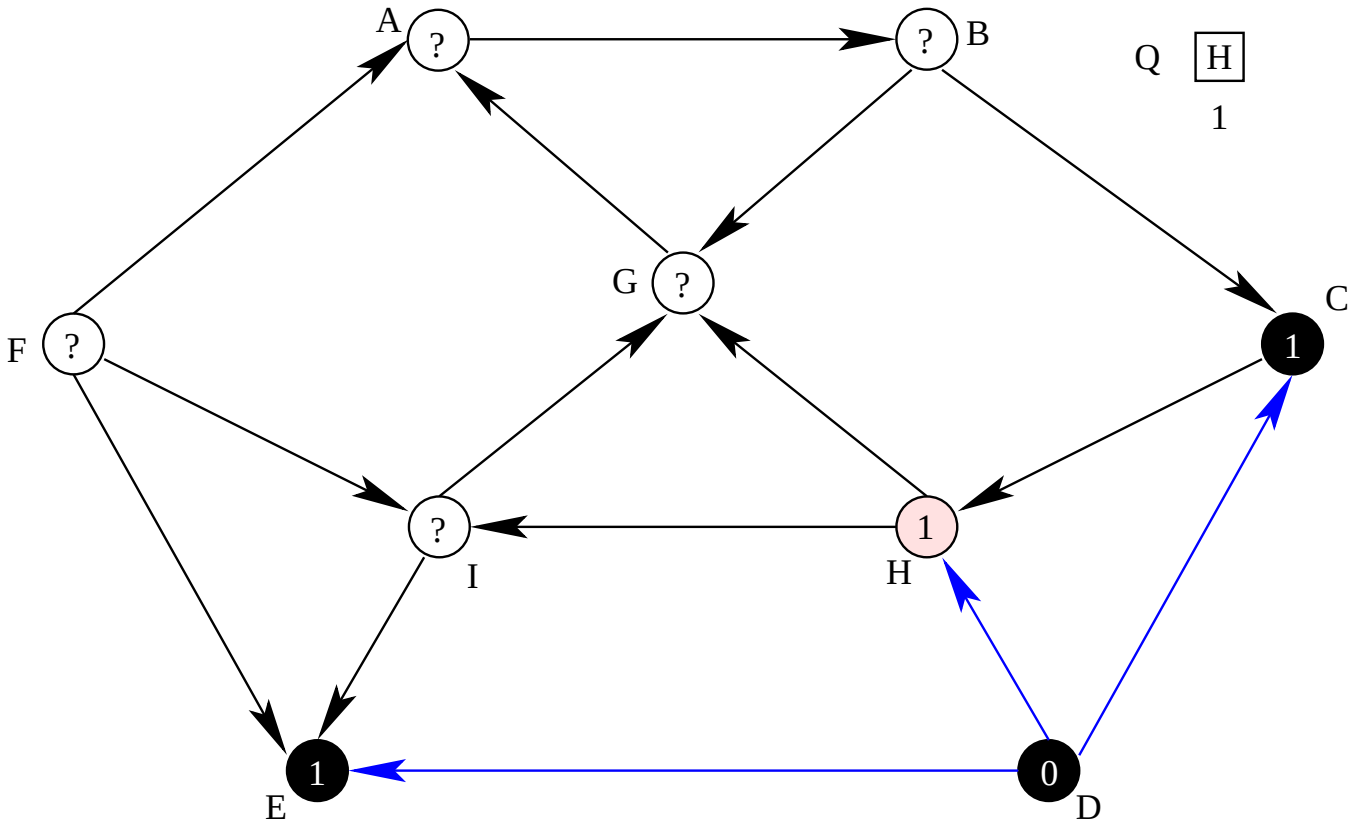
Exemplo de Execução



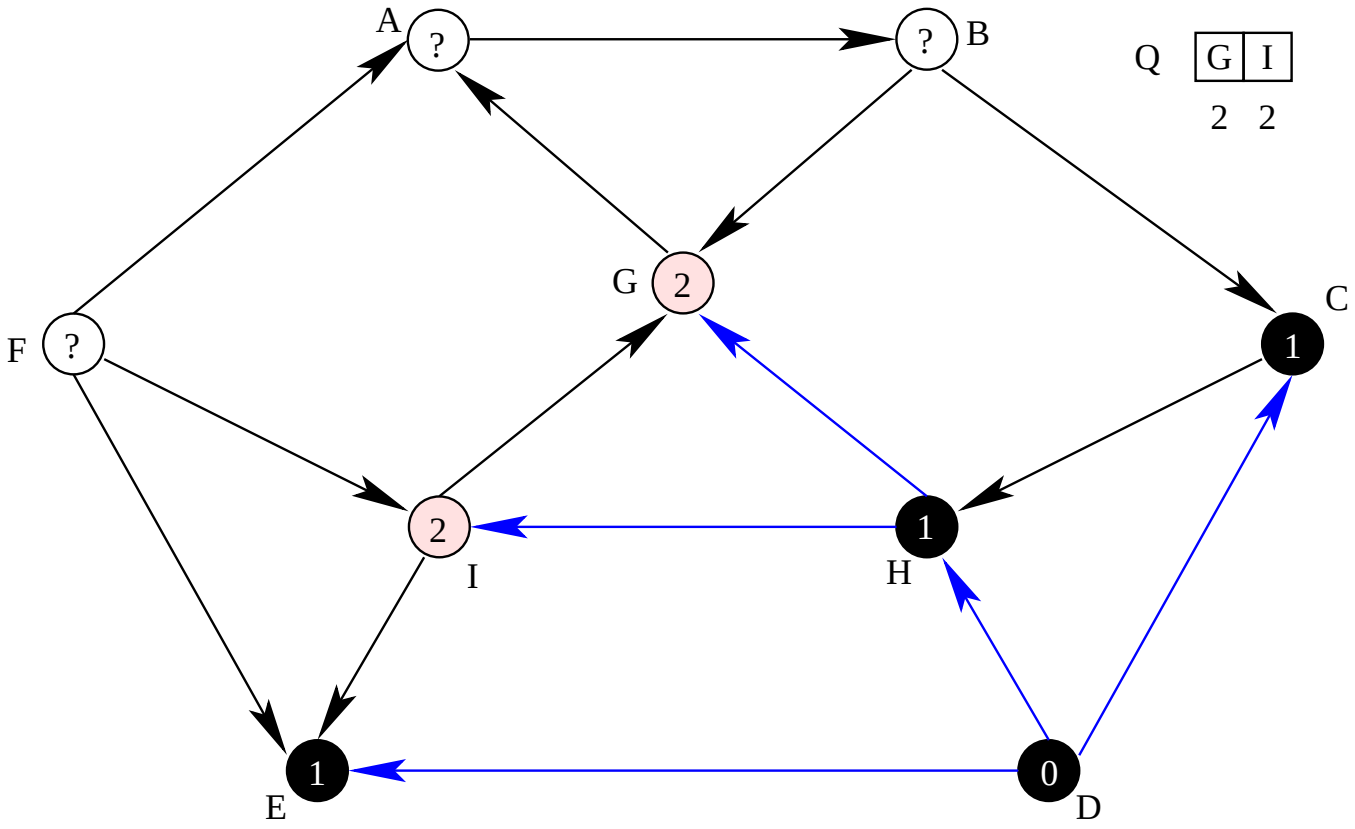
Exemplo de Execução



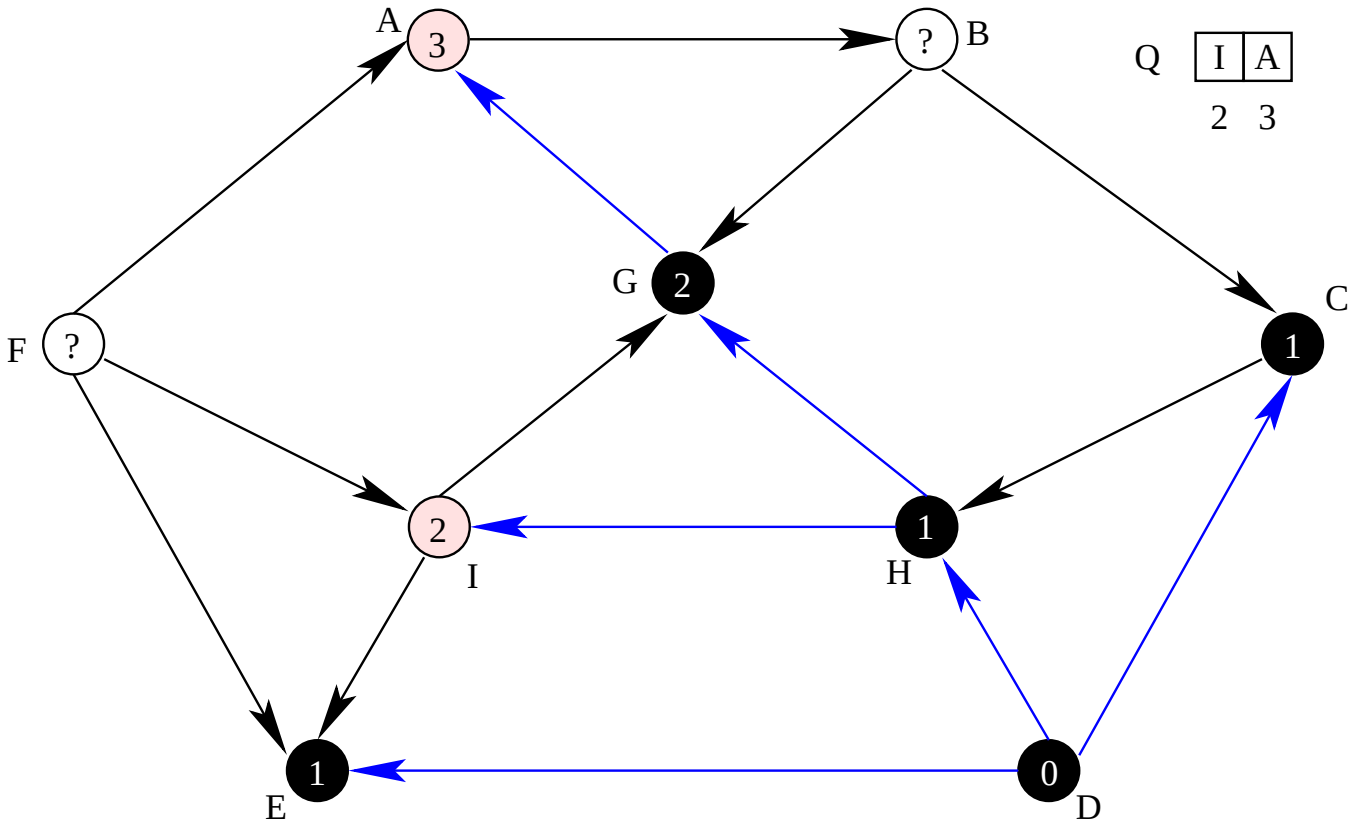
Exemplo de Execução



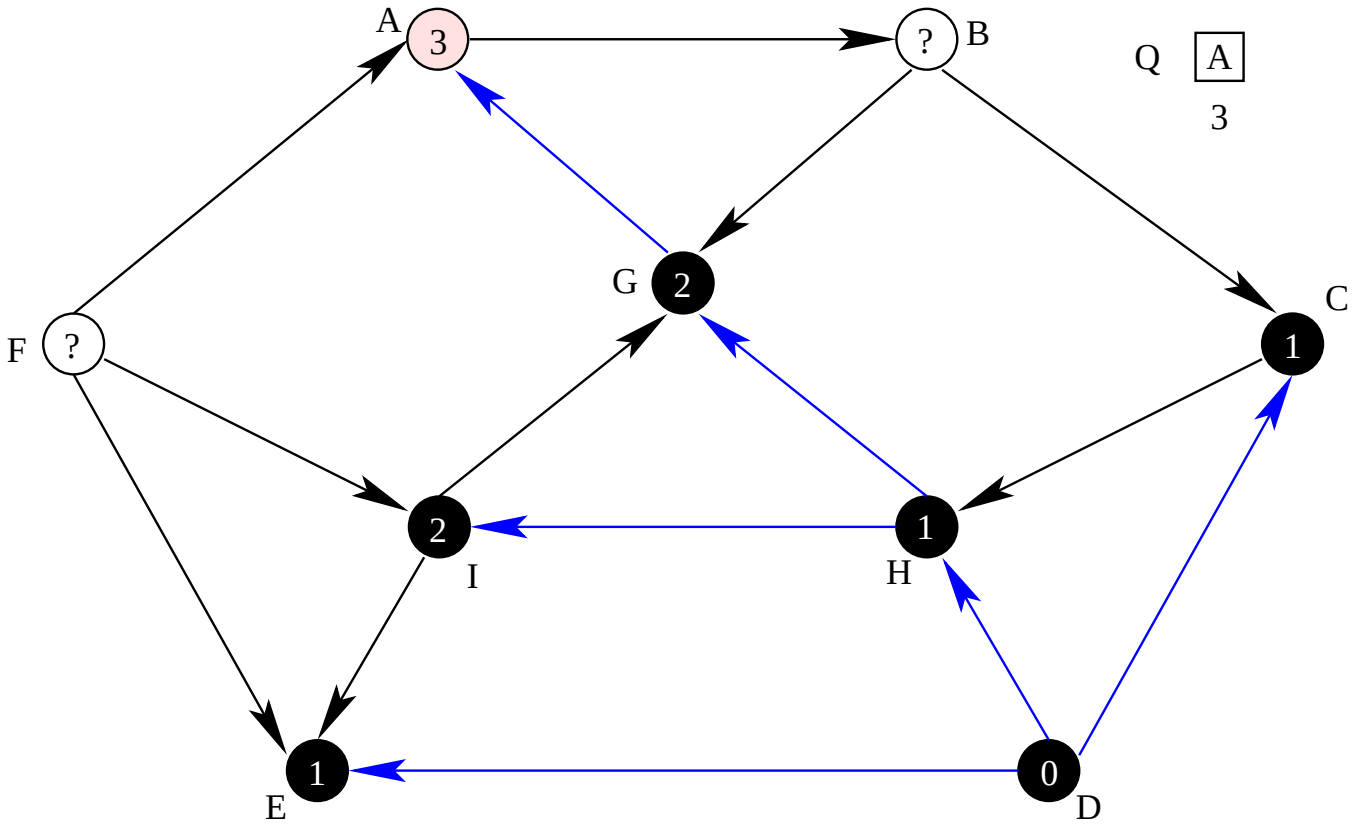
Exemplo de Execução



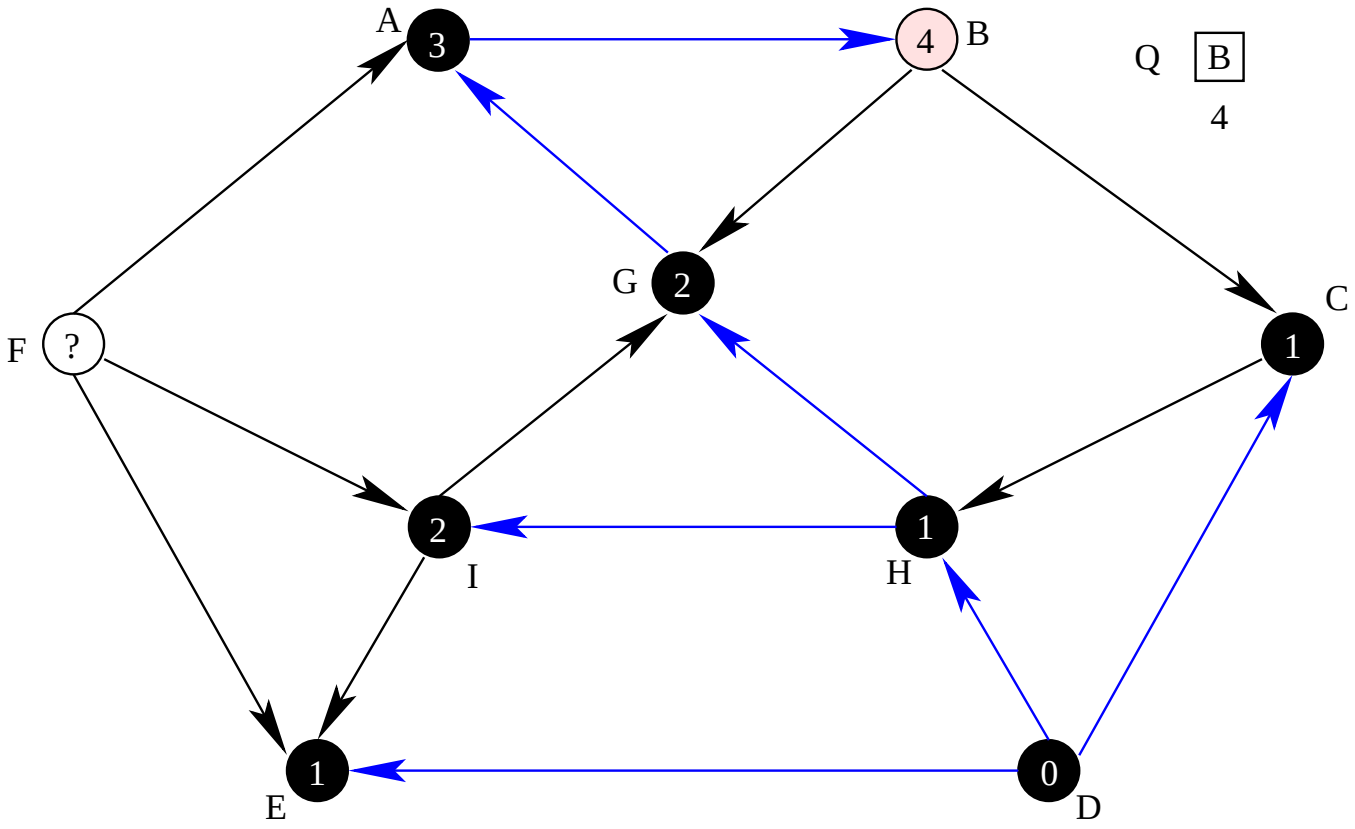
Exemplo de Execução



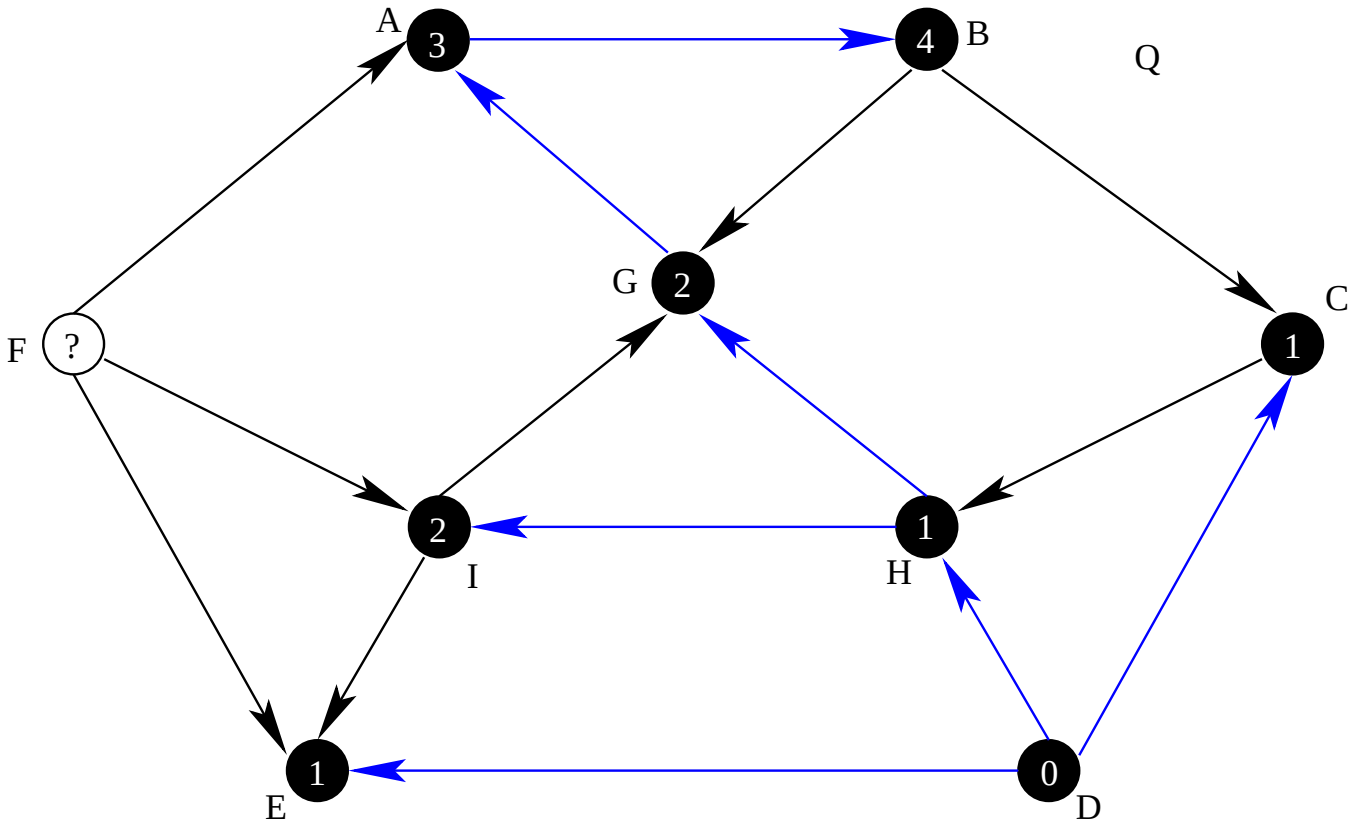
Exemplo de Execução



Exemplo de Execução



Exemplo de Execução



Exemplo de Execução – Notas

- O vértice F não é alcançável a partir de D, logo o algoritmo não passa por este vértice. Em geral, num grafo não fortemente ligado, é necessário iniciar uma nova pesquisa em cada componente ligado, para garantir que todos os vértices são alcançados.

- A árvore de pesquisa em largura do grafo (a azul no exemplo) fica definida pelo vector `parent[]`. $\Rightarrow$ Como?

- Valores de `d[]` aparecem por baixo dos elementos na queue. Observe-se que nunca há mais de dois valores diferentes! Veremos que esta é uma propriedade do algoritmo.

Caminhos Mais Curtos Entre Dois Vértices

A *distância* $\delta(s, v)$ do vértice s ao vértice d define-se como o menor número de arcos em qualquer caminho de s para d. É infinita caso não existam caminhos de s para d.

Um caminho de comprimento $\delta(s, v)$ diz-se um *caminho mais curto* entre os dois vértices.

Lema. *Seja $G = (V, E)$ orientado ou não, e $s \in V$. Então para qualquer arco $(u, v) \in E$ tem-se $\delta(s, v) \leq \delta(s, u) + 1$.*

Prova.

- *Se u é alcançável a partir de s, então v também o é. O caminho mais curto de s para v não pode ser mais comprido do que o mais curto de s para u seguido de (u, v).*

- *Se u não é alcançável a partir de s, então $\delta(s, u) = \infty$.*

Alguns Lemas . . .

Lema. *Seja $G = (V, E)$ orientado ou não, e $s \in V$. Então após execução de* $\texttt{BFS}(G, s)$, *cada valor $d[v]$ calculado para $v \in V$ verifica $d[v] \geq \delta(s, v)$.*

Prova. *Por indução no número de operações* `enqueue` *efectuadas (caso base: queue contém apenas s; caso indutivo: v é descoberto como adjacente a u).*

Lema. (na mesma execução) *Se a queue Q ($\longleftarrow$) contém $\langle v_1, v_2, \ldots v_r \rangle$, então*

$$d[v_r] \leq d[v_1] + 1 \qquad \text{e} \qquad d[v_i] \leq d[v_{i+1}], i \in \{1, 2, \ldots r-1\}$$

Prova. *Indução no número de operações de acesso à queue (incl.* `dequeue`*).*

Lema. *Se v_i e v_j são colocados em Q* por esta ordem *durante a execução do algoritmo, então $d[v_i] \leq d[v_j]$ no instante em que v_j entra em Q.*

Prova. *Imediato pelo Lema anterior.*

Correcção do Algoritmo

Teorema. [Correcção] *Seja $G = (V, E)$ orientado ou não, e $s \in V$, e considere-se a execução de $\texttt{BFS}(G, s)$. Então:*

1. *O algoritmo descobre todos os vértices $v \in V$ alcançáveis a partir de s;*

2. *no fim da execução tem-se $\texttt{d}[v] = \delta(s, v)$ para todo $v \in V$;*

3. *para todo $v \neq s$ alcançável a partir de s, um dos caminhos mais curtos de s para v é um caminho mais curto de s para $\texttt{parent}[v]$, seguido do arco $(\texttt{parent}[v], v)$.*

Prova. *2. prova-se por contradição considerando, de entre todos os vértices v com valores $d[]$ errados, o caso do que tem menor valor $\delta(s, v)$. 1. segue naturalmente (se $\delta(s, v)$ é finito então $d[v]$ também, logo v foi descoberto), e 3. é uma consequência de $d[v] = d[parent[v]] + 1$.*

Árvore de Pesquisa em Largura de um Grafo

Dado $G = (V, E)$ e um vector $\pi[\,]$ dos antecessores de cada vértice, define-se $G_\pi = (V_\pi, E_\pi)$, o sub-grafo dos antecessores de G, como se segue:

$$\begin{aligned} V_\pi &= \{v \in V \mid \pi[v] \neq NIL\} \cup \{s\} \\ E_\pi &= \{(\pi[v], v) \mid v \in V_\pi - \{s\}\} \end{aligned}$$

Este grafo será uma árvore de pesquisa em largura (APL) de G a partir de s se V_π for o conjunto de vértices alcançáveis a partir de s, e para cada $v \in V_\pi$ o caminho mais curto (em G) de s até v pertencer a G_π.

Teorema. *(Para G orientado ou não-orientado) O algoritmo de pesquisa em largura constrói um vector* `parent` *tal que* $G_{\texttt{parent}}$ *é uma APL de* G.

Prova.

- $V_{\texttt{parent}}$ *consiste nos vértices alcançáveis (a partir de* s*) de* V*.*
- *Sendo* $G_{\texttt{parent}}$ *uma árvore, contém caminhos únicos de* s *para qualquer* $v \in V_{\texttt{parent}}$*.*
- *Aplicação do Teorema de Correcção indutivamente a cada um desses caminhos. . .*

Exercício: Escrever um algoritmo que, depois de executado BFS(G, s), recebe um vértice v e devolve (caso exista) o caminho (sequência de vértices) mais curto entre s e v.

Qual o tempo de execução do algoritmo?

Pesquisa em Profundidade ("Depth-first Search")

Este algoritmo utiliza a seguinte estratégia para efectuar a travessia do grafo:

- Os próximos arcos a explorar têm origem no mais recente vértice descoberto que ainda tenha vértices adjacentes não explorados.

Assim, quando todos os adjacentes a v tiverem sido explorados, o algoritmo recua ("backtracks") para explorar vértices com origem no nó a partir do qual v foi descoberto.

Estudaremos a versão deste algoritmo que percorre *todos* os nós do grafo. Depois de terminada a pesquisa com origem em s, serão feitas novas pesquisas para descobrir os nós não-alcançáveis a partir de s.

O *grafo dos antecessores* de G é neste caso uma *floresta* composta de várias árvores de pesquisa em profundidade (APP).

Pesquisa em Profundidade ("Depth-first Search")

A coloração (branco, cinzento, preto) garante que cada vértice pertence a *uma única árvore*. Estas são pois *disjuntas*.

O algoritmo usa ainda uma noção de marcas temporais 'timestamps": $d[v]$ para o instante em que o vértice é descoberto (passa a cinzento) e $f[v]$ quando todos os seus adjacentes são descobertos (passa a preto). Logo $d[v] < f[v]$.

Estas marcas são inteiros entre 1 e $2|V| \Rightarrow$ porquê?

Algoritmo de Pesquisa em Profundidade

```
void DFS((V,E), s) {                    /* G = (V,E) */
  color[s] = GRAY;
  time = time+1;
  d[s] = time;
  for (v ∈ Adj(s))
    if (color[v] == WHITE) {
      parent[v] = s;
      DFS((V,E), v);
    }
  color[s] = BLACK;
  time = time+1;
  f[s] = time;
}
```

Inicia pesquisa no nó s; assume que todas as inicializações foram feitas – nomeadamente da variável global `time`.

Algoritmo de Pesquisa em Profundidade

```
void CDFS((V,E)) {                    /* G = (V,E) */
  for (u ∈ V) {
    color[u] = WHITE;
    parent[u] = NIL;
  }
  time = 0;
  for (u ∈ V)
    if (color[u] == WHITE)
      DFS((V,E), u);
}
```

Utiliza o algoritmo anterior como sub-rotina; efectua todas as inicializações e garante que todos os nós são descobertos.

Cada invocação `DFS((V,E), u)` em `CDFS` gera uma nova APP com raíz `u`.

Observações

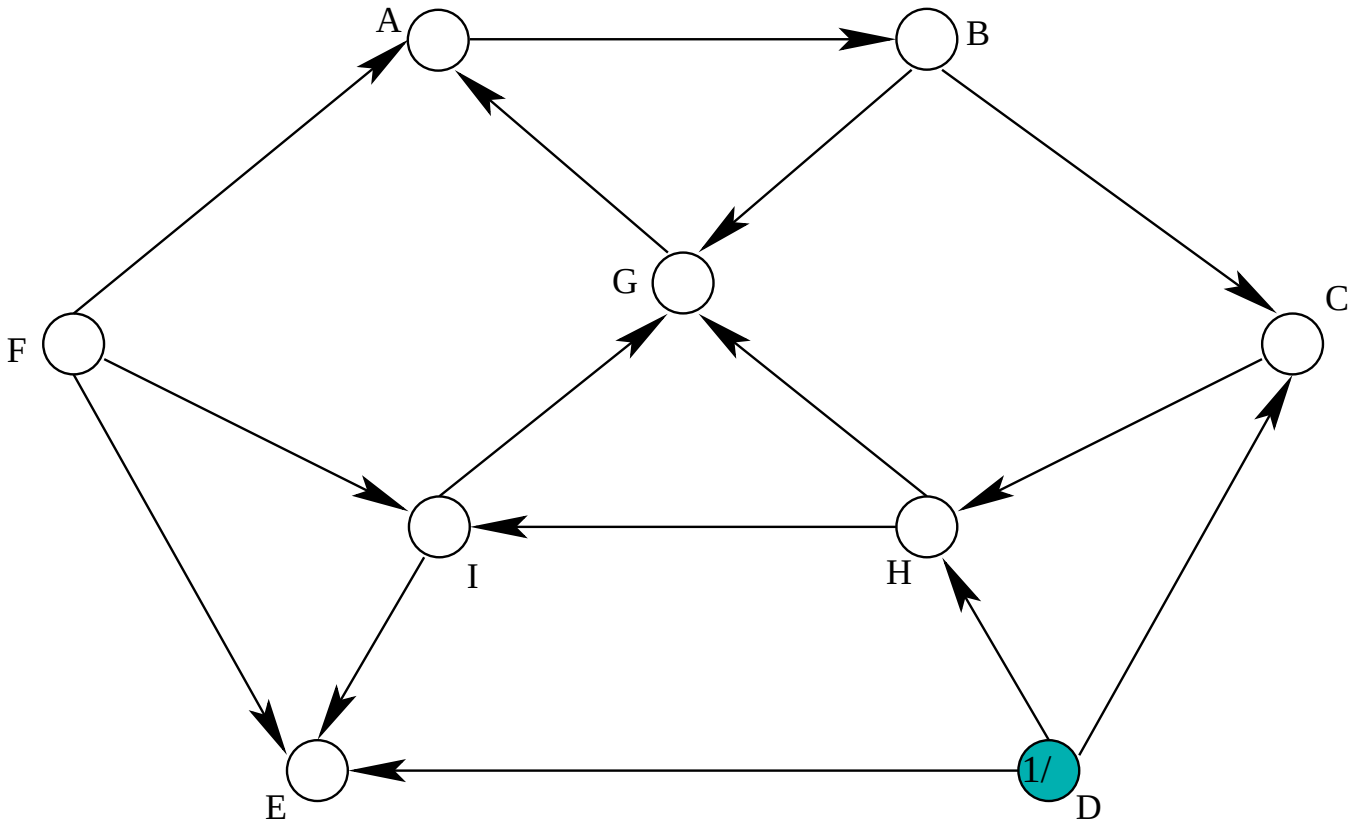
- No fim da execução de `CDFS` foram atribuídas a todos os nós u marcas temporais $d[u]$ e $f[u]$.

- Os resultados produzidos pelo algoritmo (floresta gerada e marcas temporais) não são únicos, dependendo da ordem pela qual são percorridos os nós nos diversos ciclos `for`.

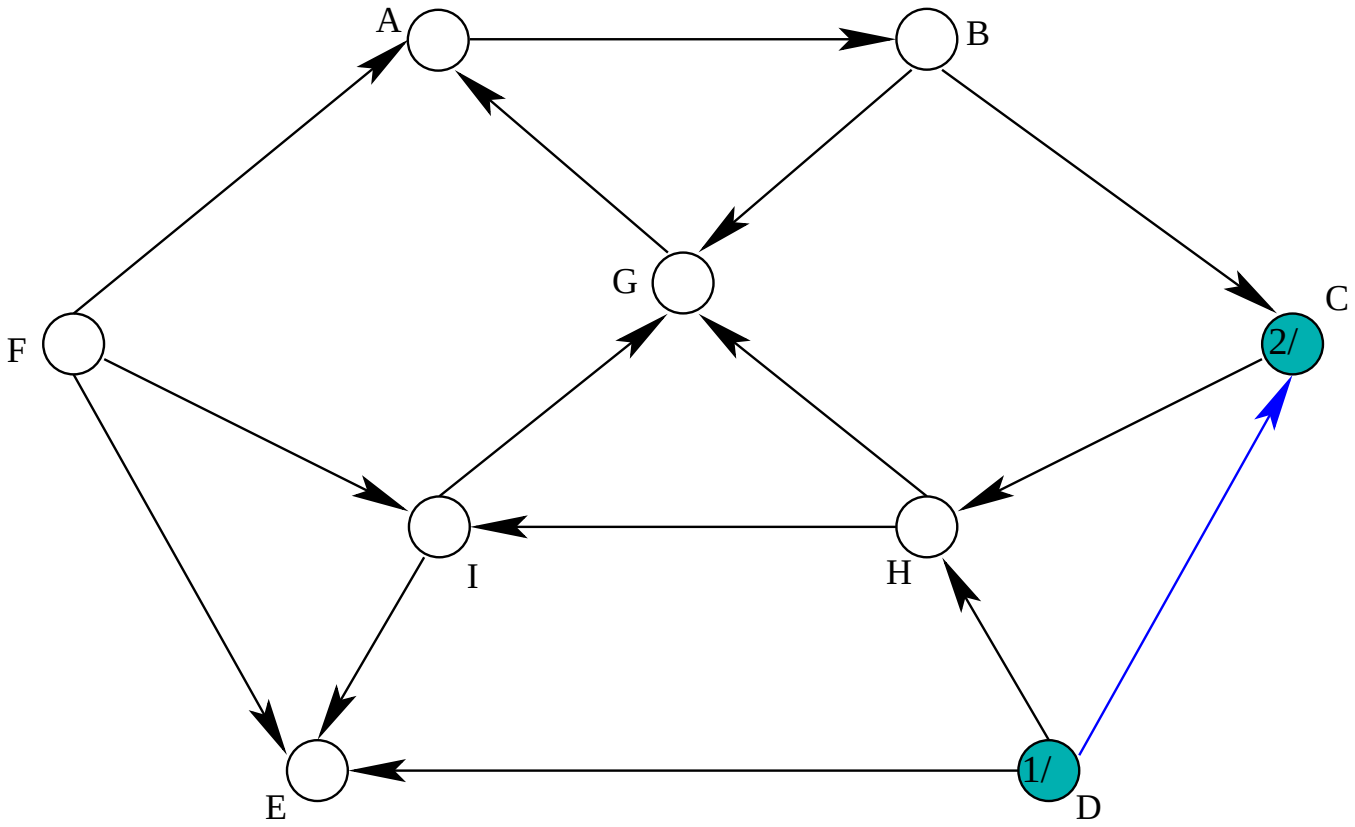
- No entanto, nas aplicações típicas deste algoritmo, os diversos resultados possíveis são equivalentes.

$\Rightarrow$ Por que razão este algoritmo não calcula as distâncias de cada vértice à origem da travessia?

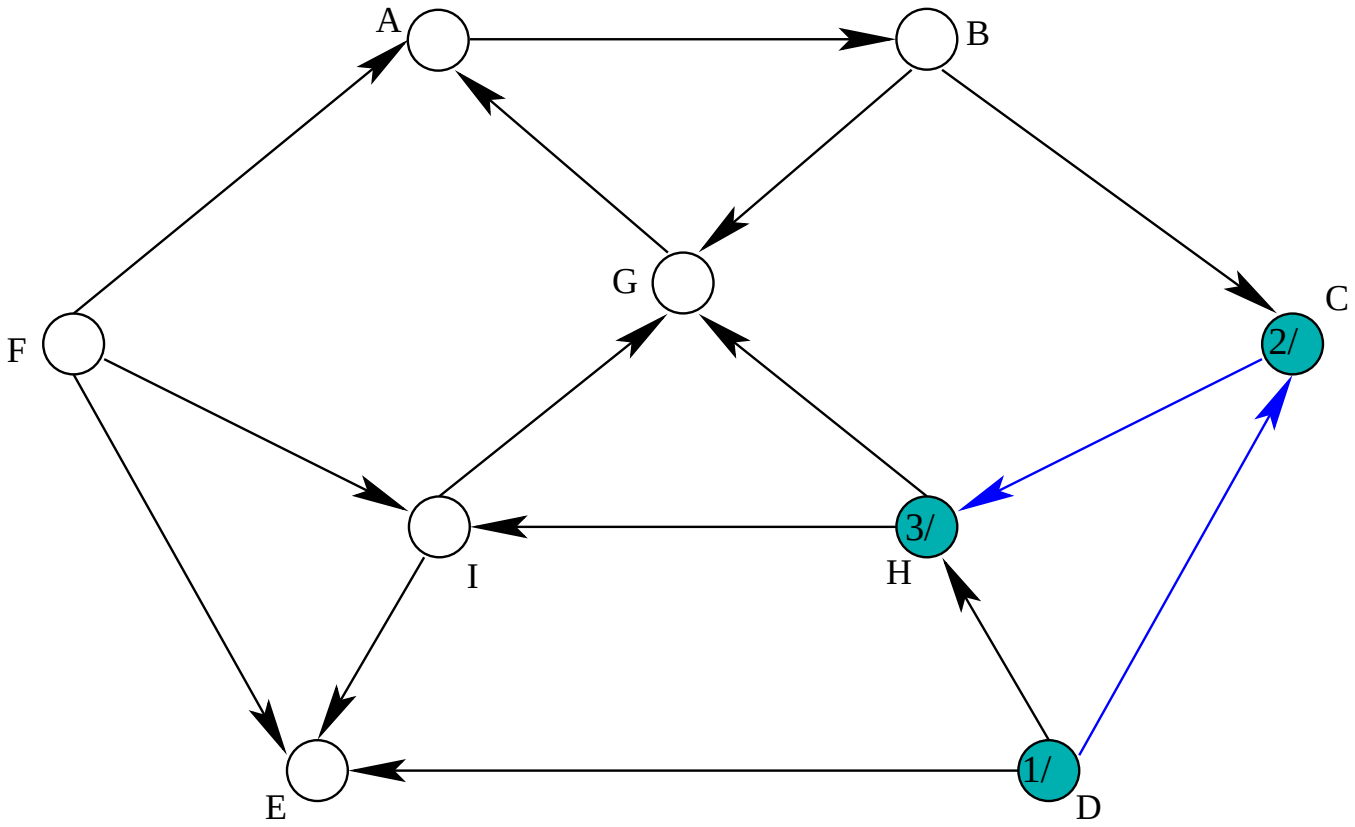
Exemplo de Execução



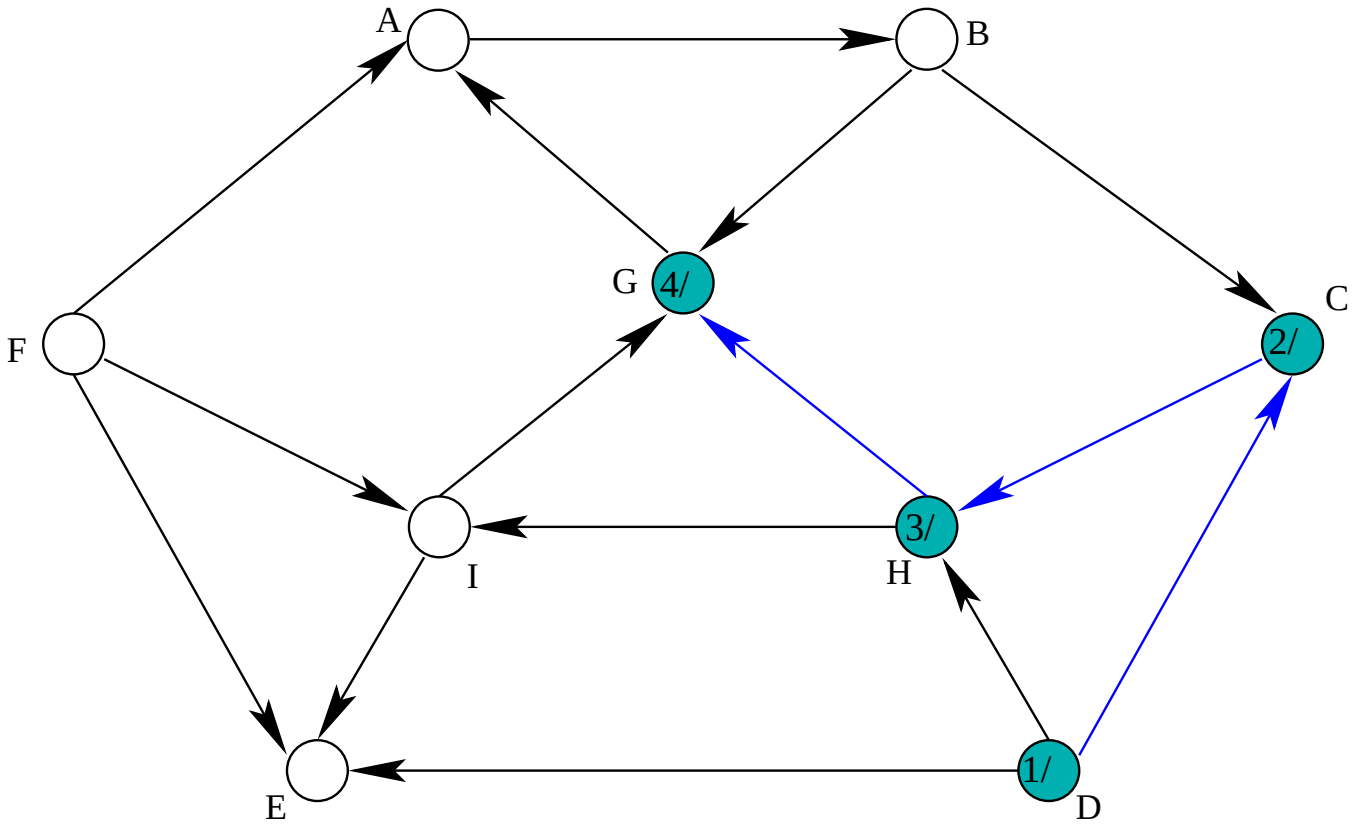
Exemplo de Execução



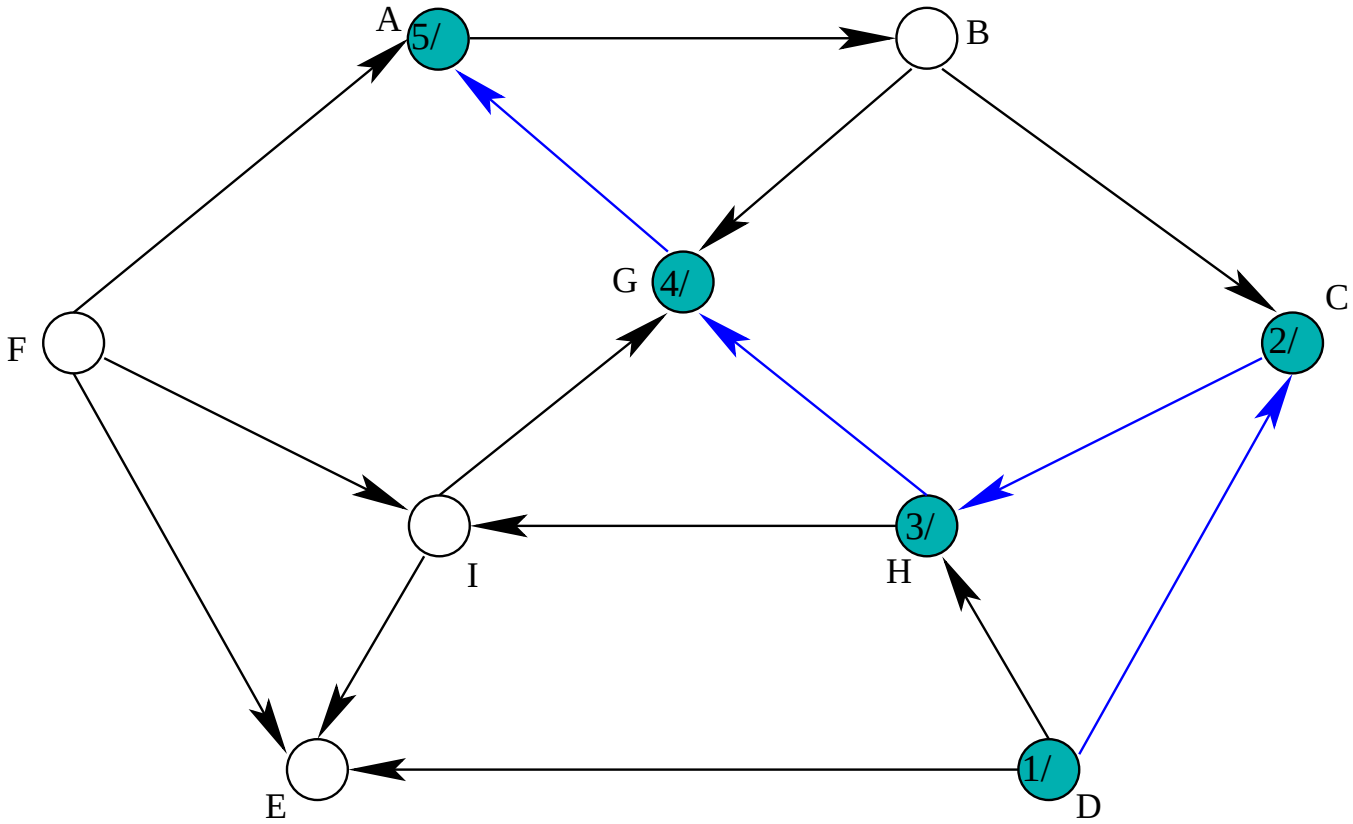
Exemplo de Execução



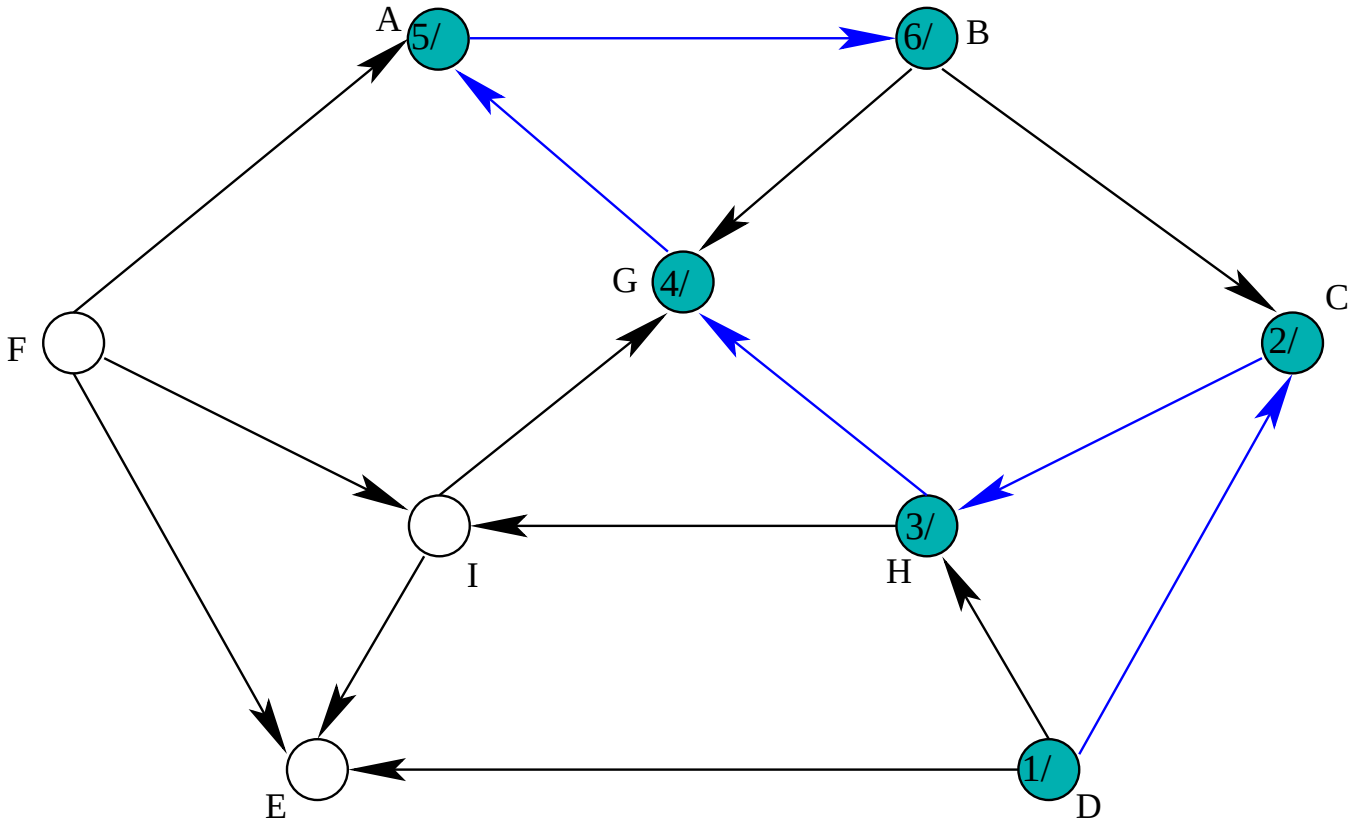
Exemplo de Execução



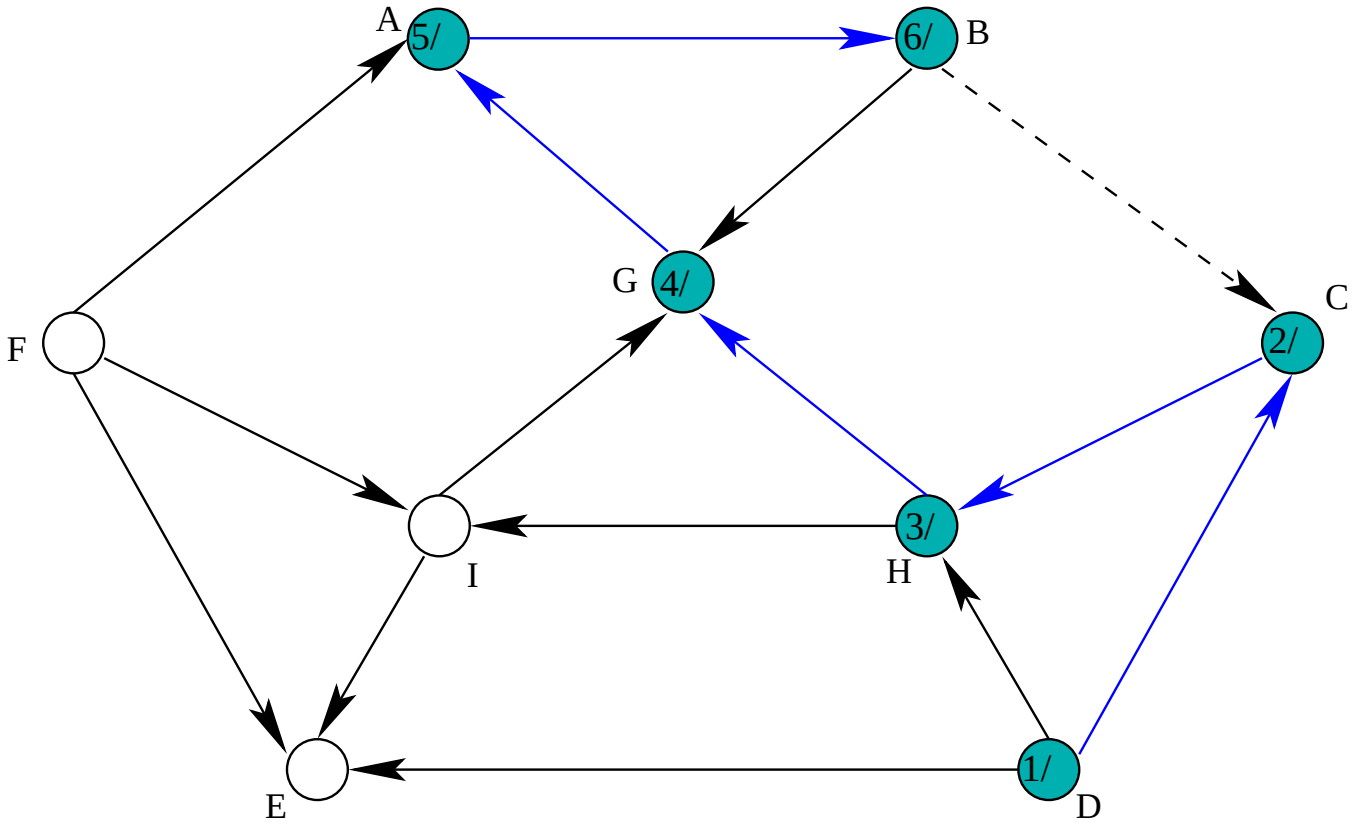
Exemplo de Execução



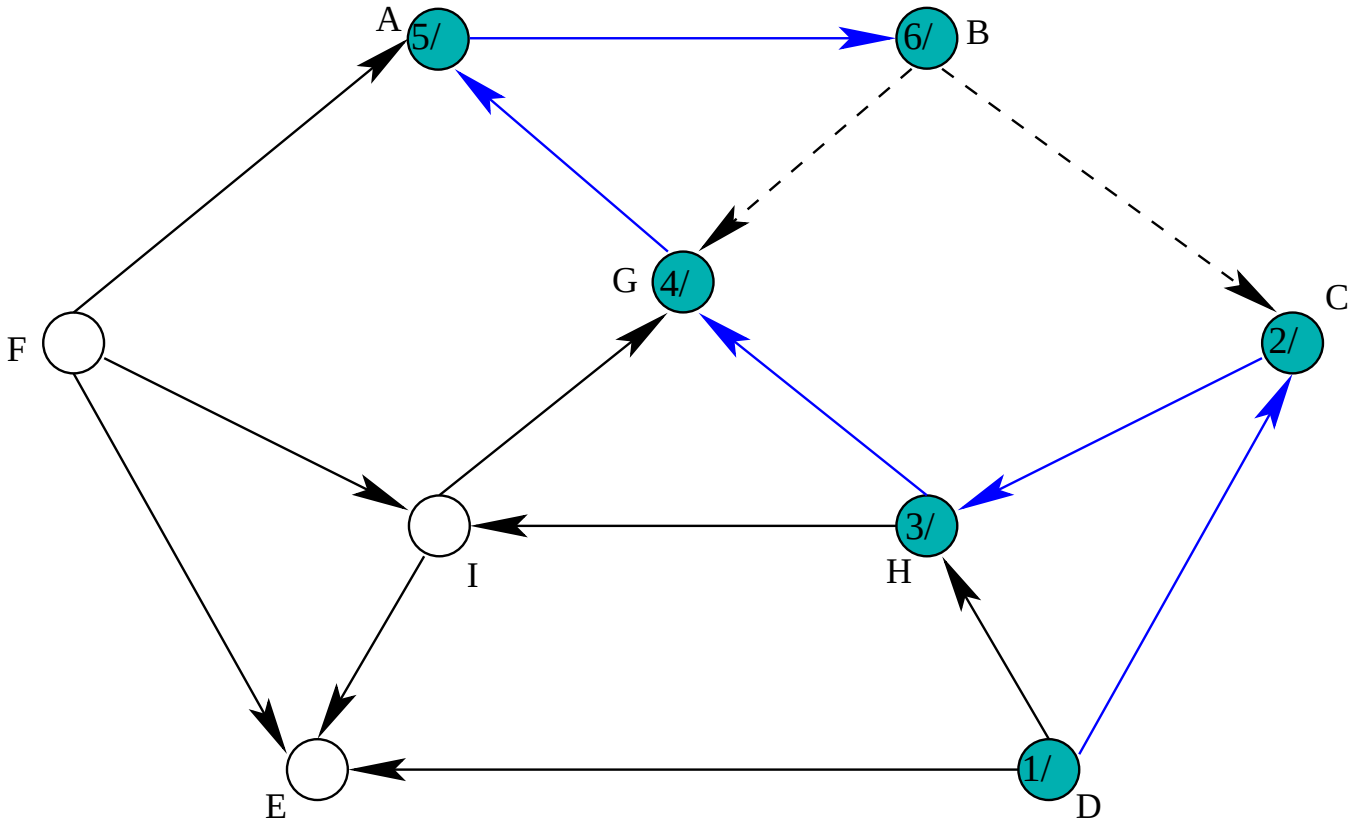
Exemplo de Execução



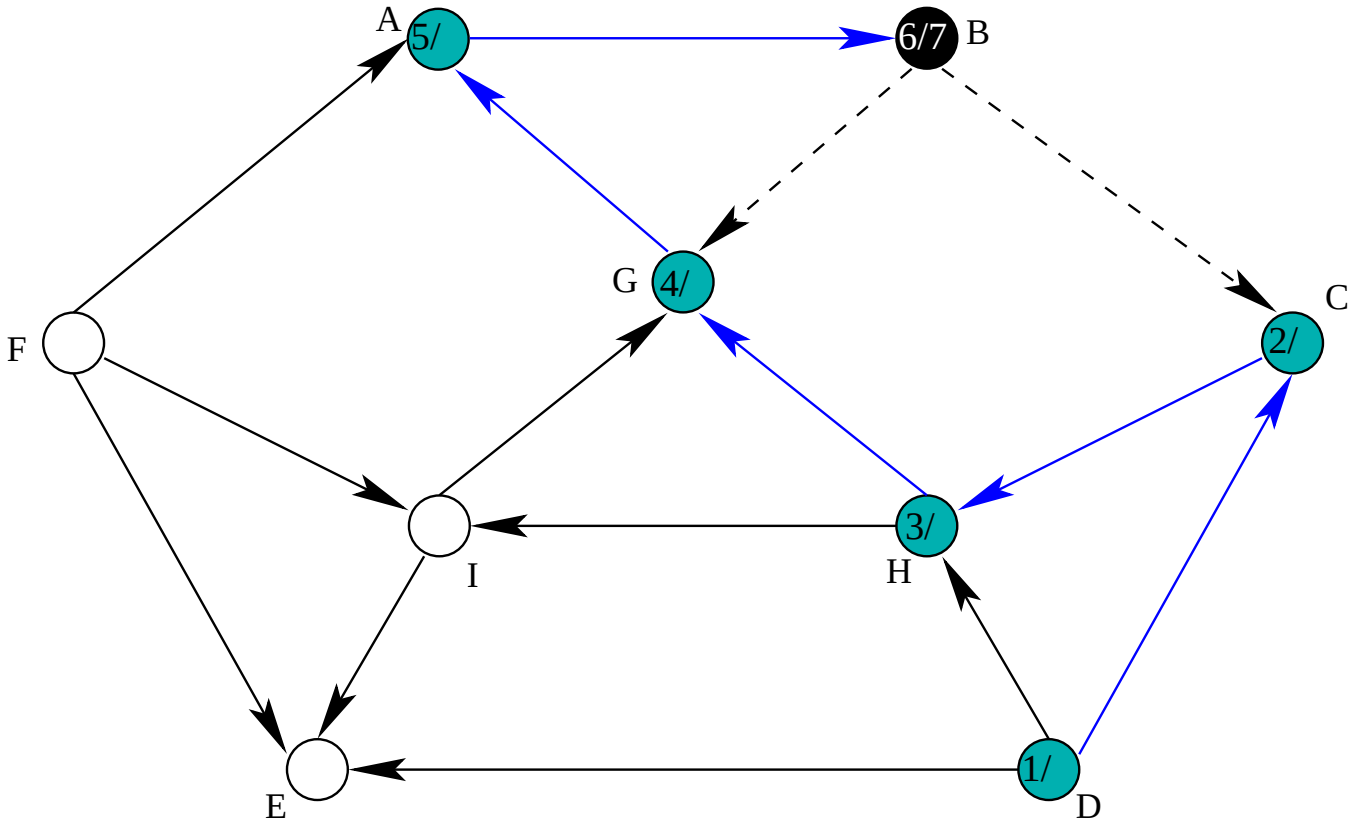
Exemplo de Execução



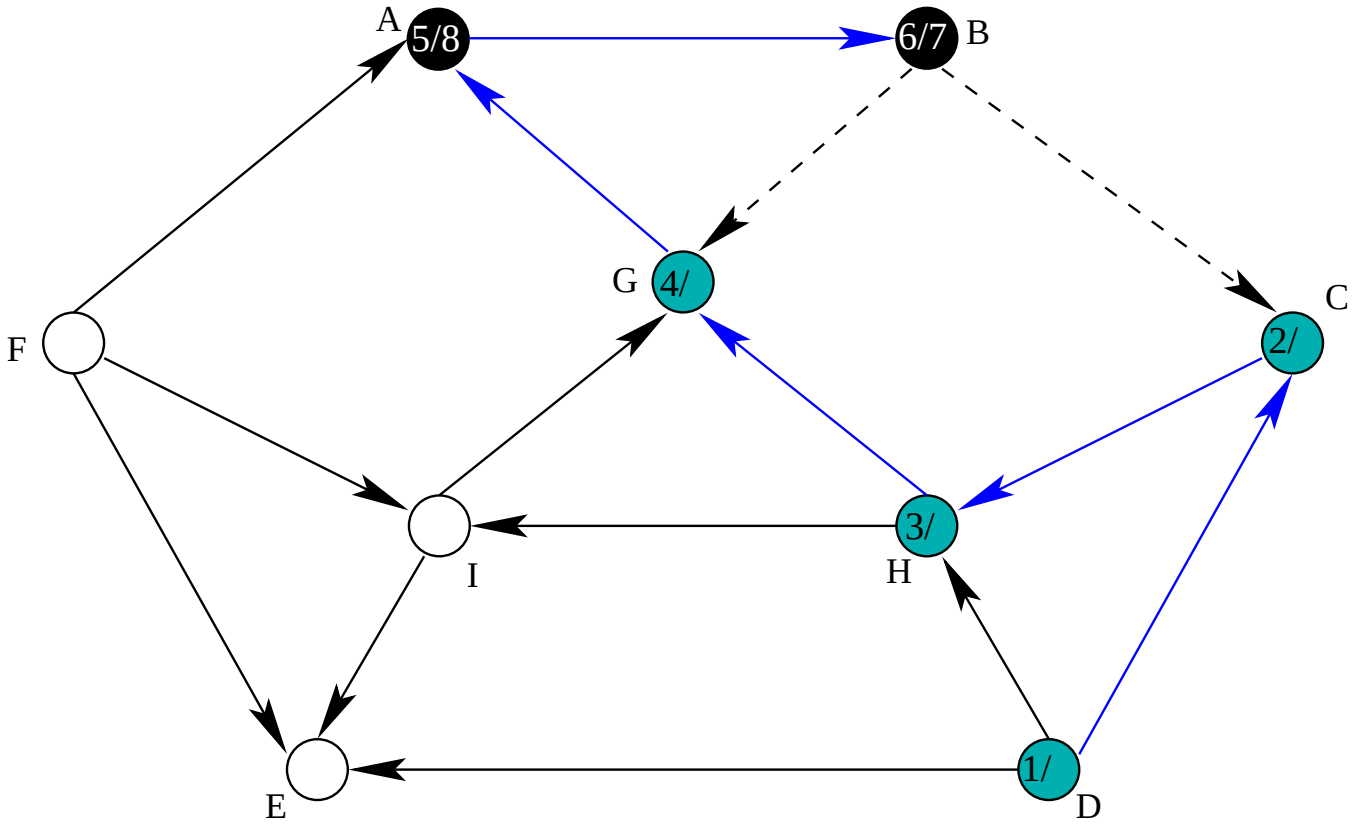
Exemplo de Execução



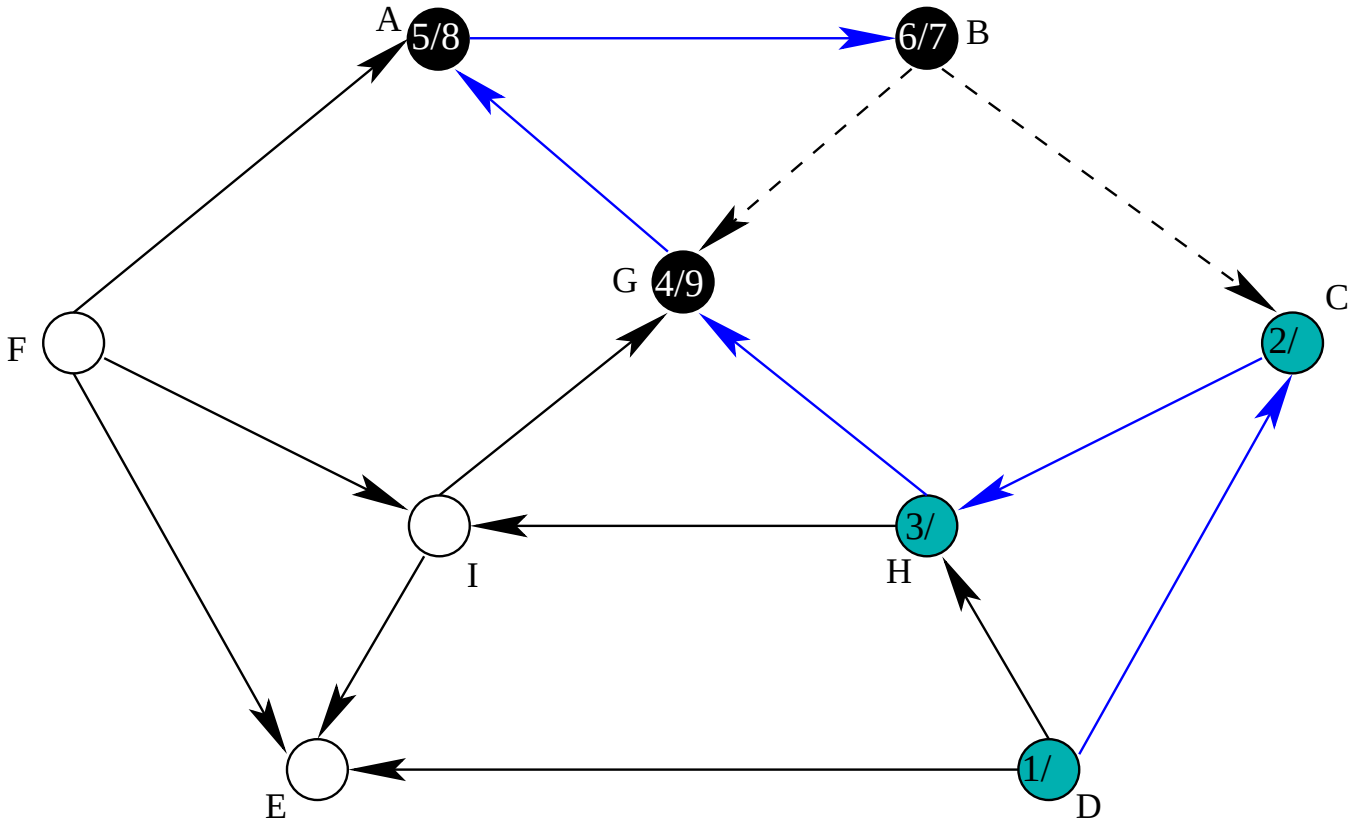
Exemplo de Execução



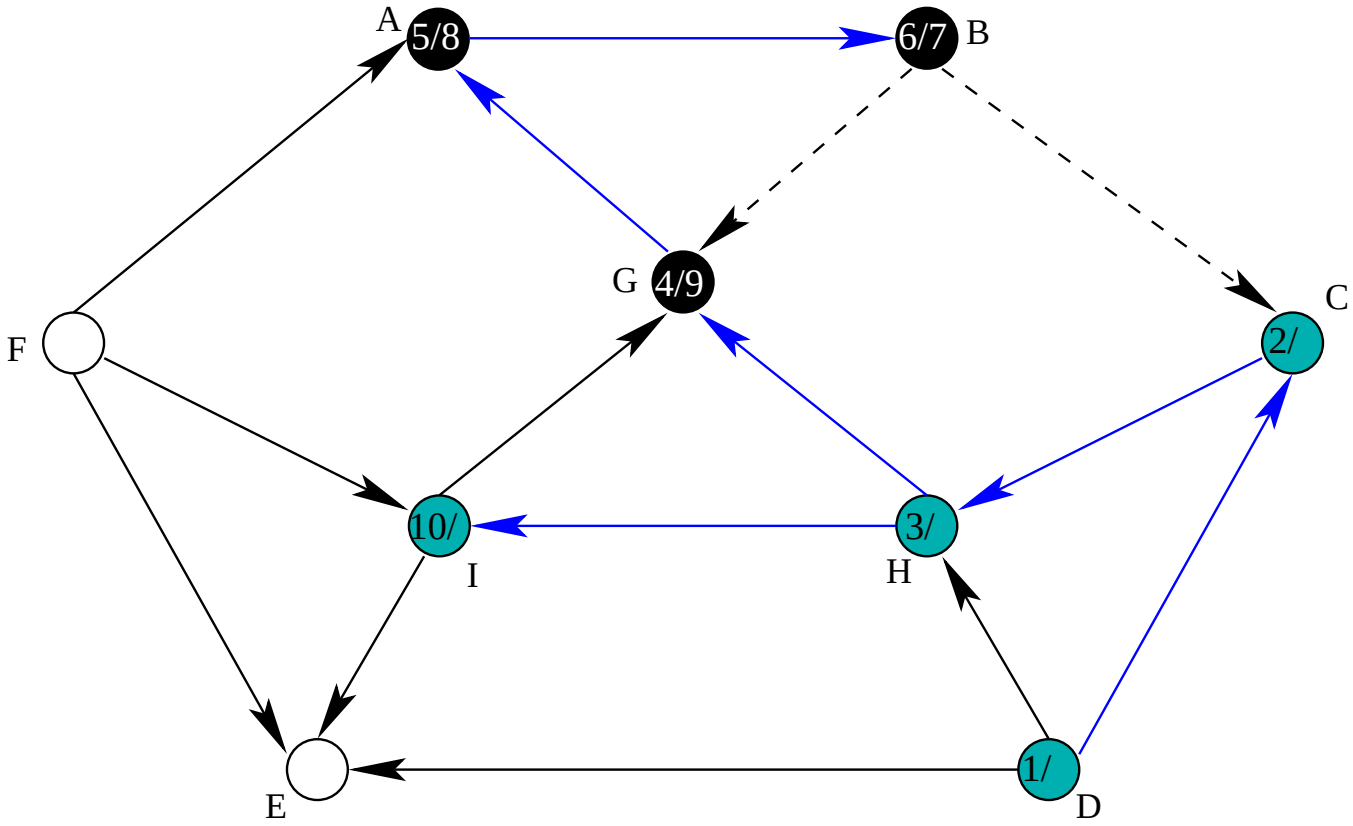
Exemplo de Execução



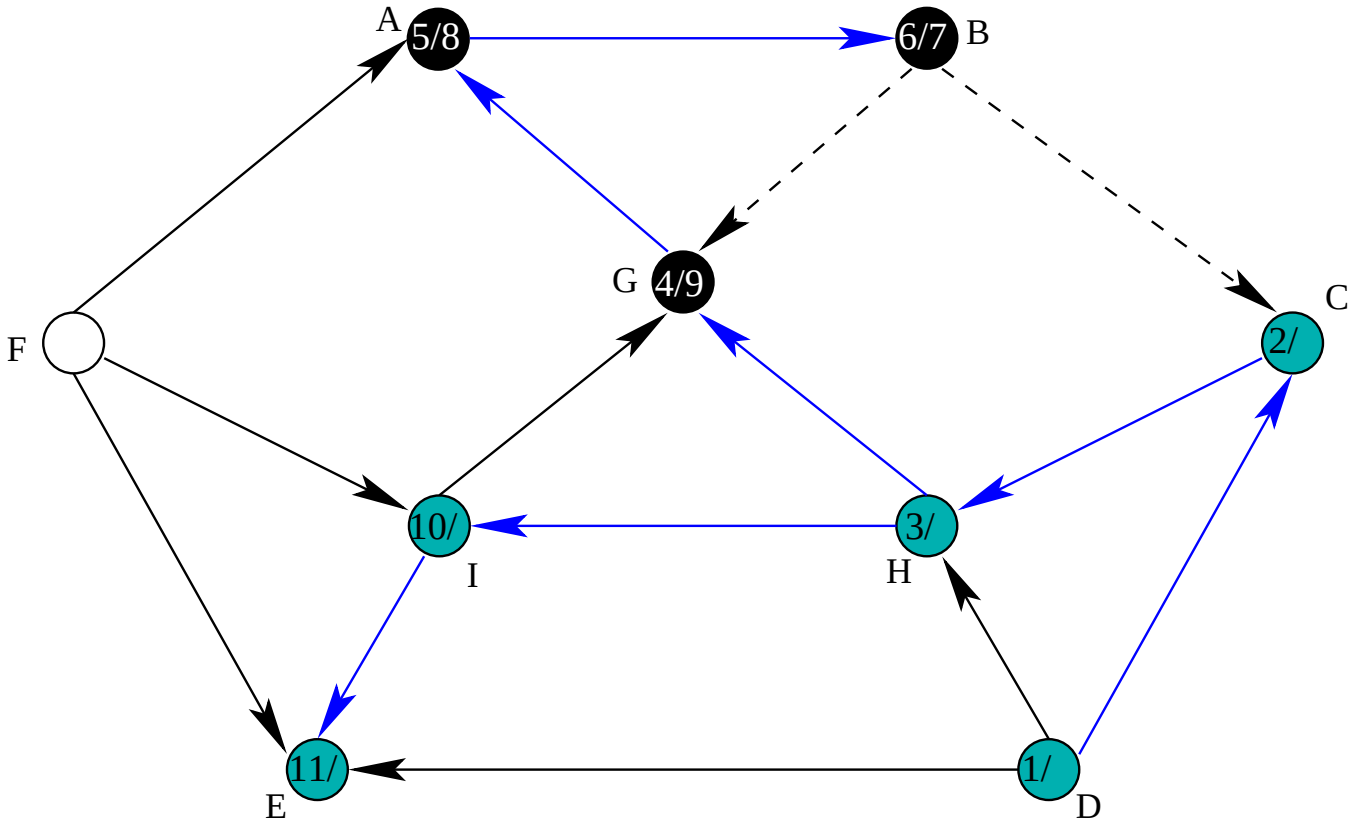
Exemplo de Execução



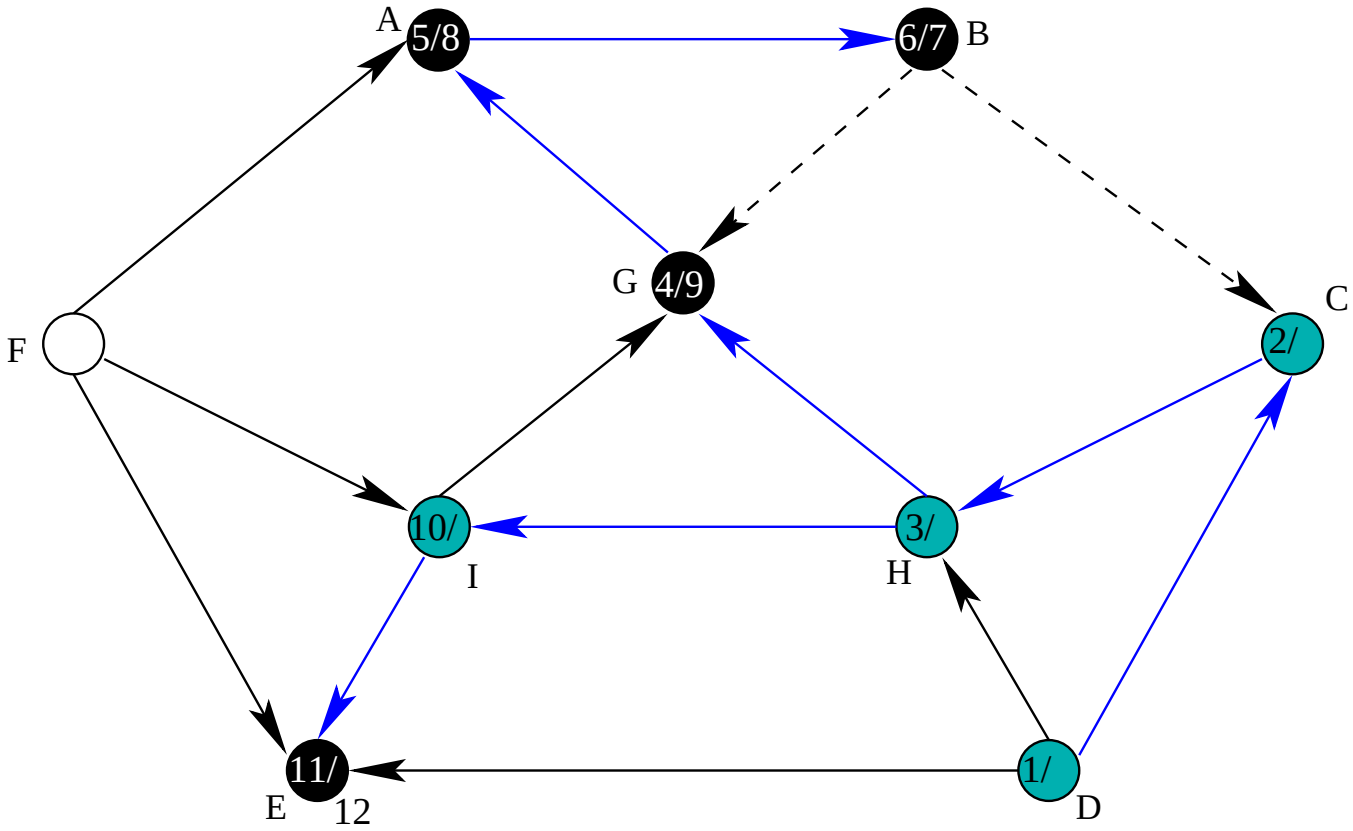
Exemplo de Execução



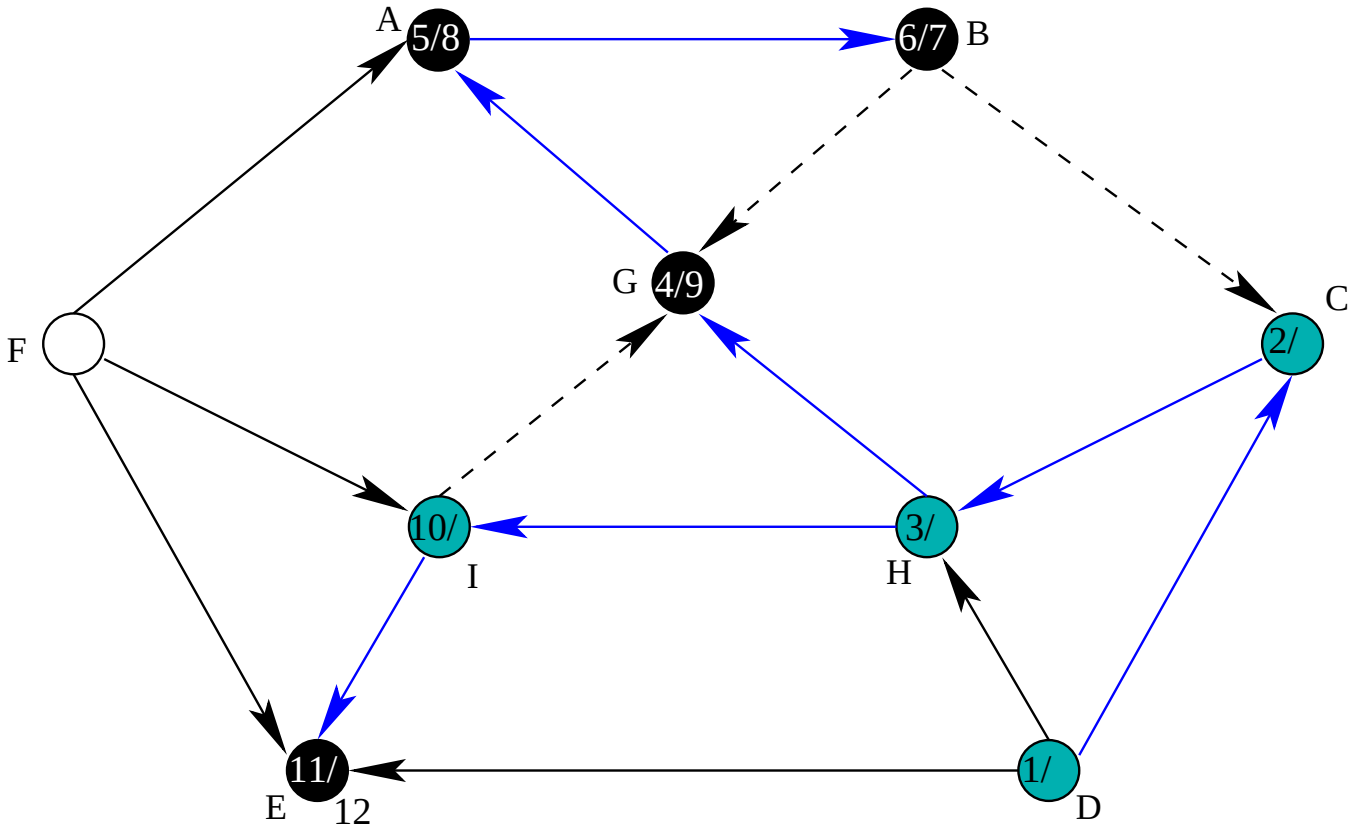
Exemplo de Execução



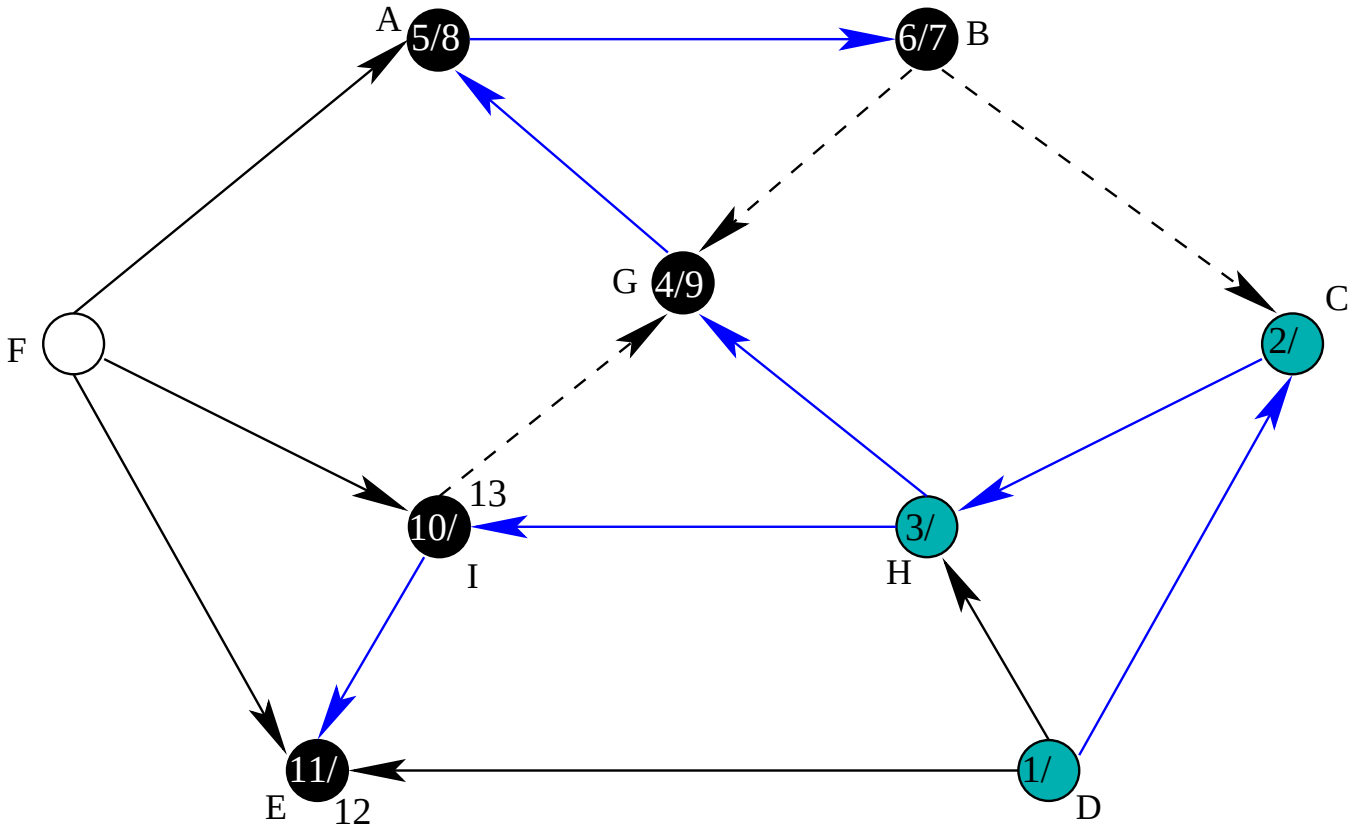
Exemplo de Execução



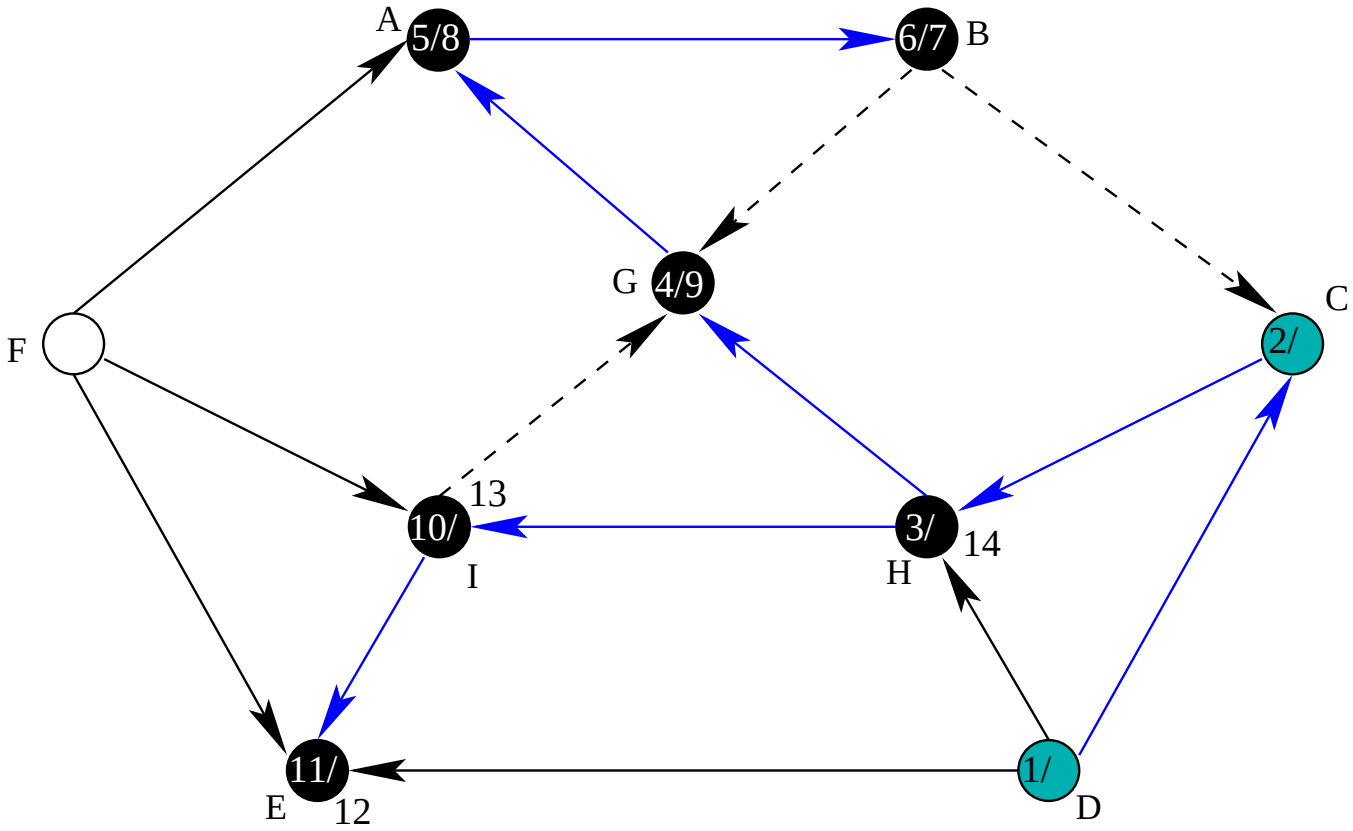
Exemplo de Execução



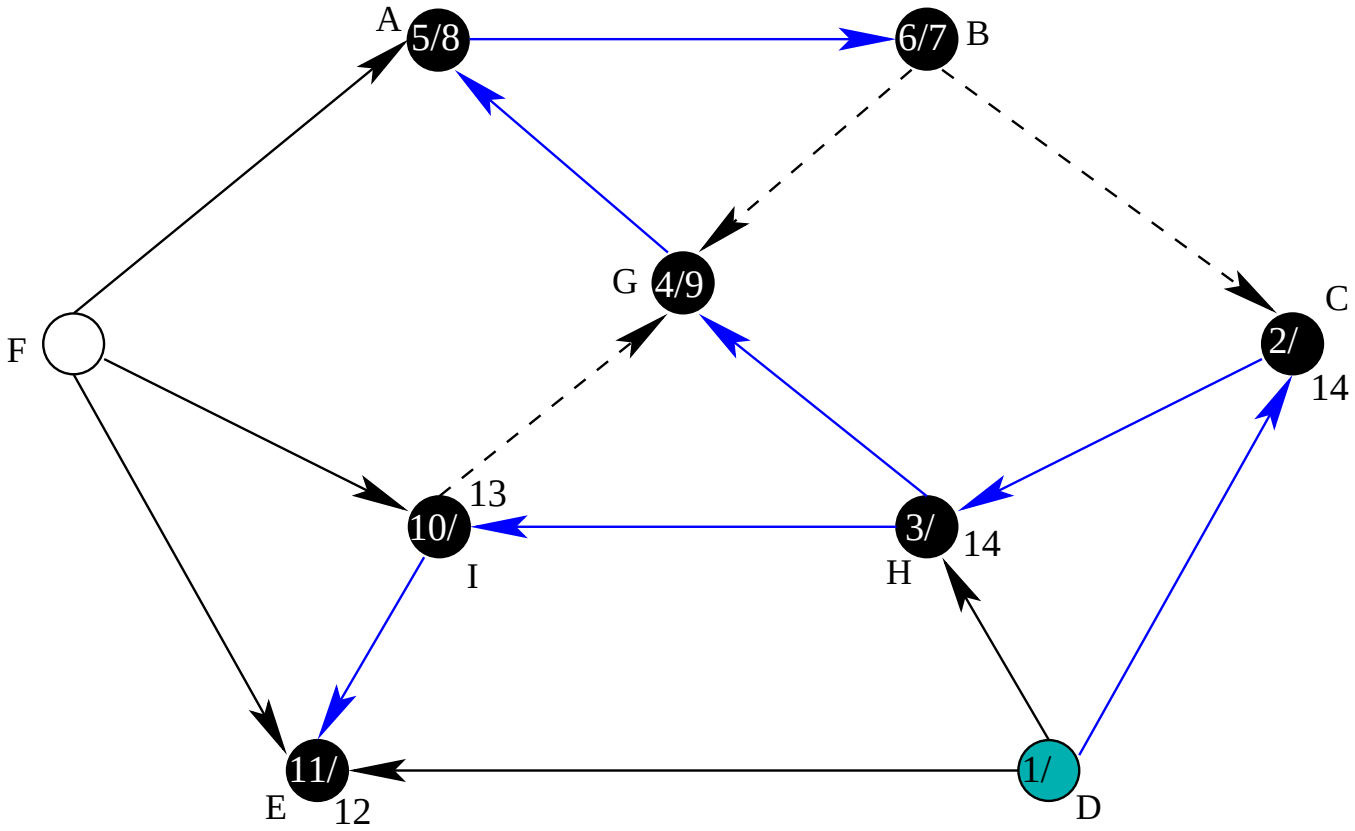
Exemplo de Execução



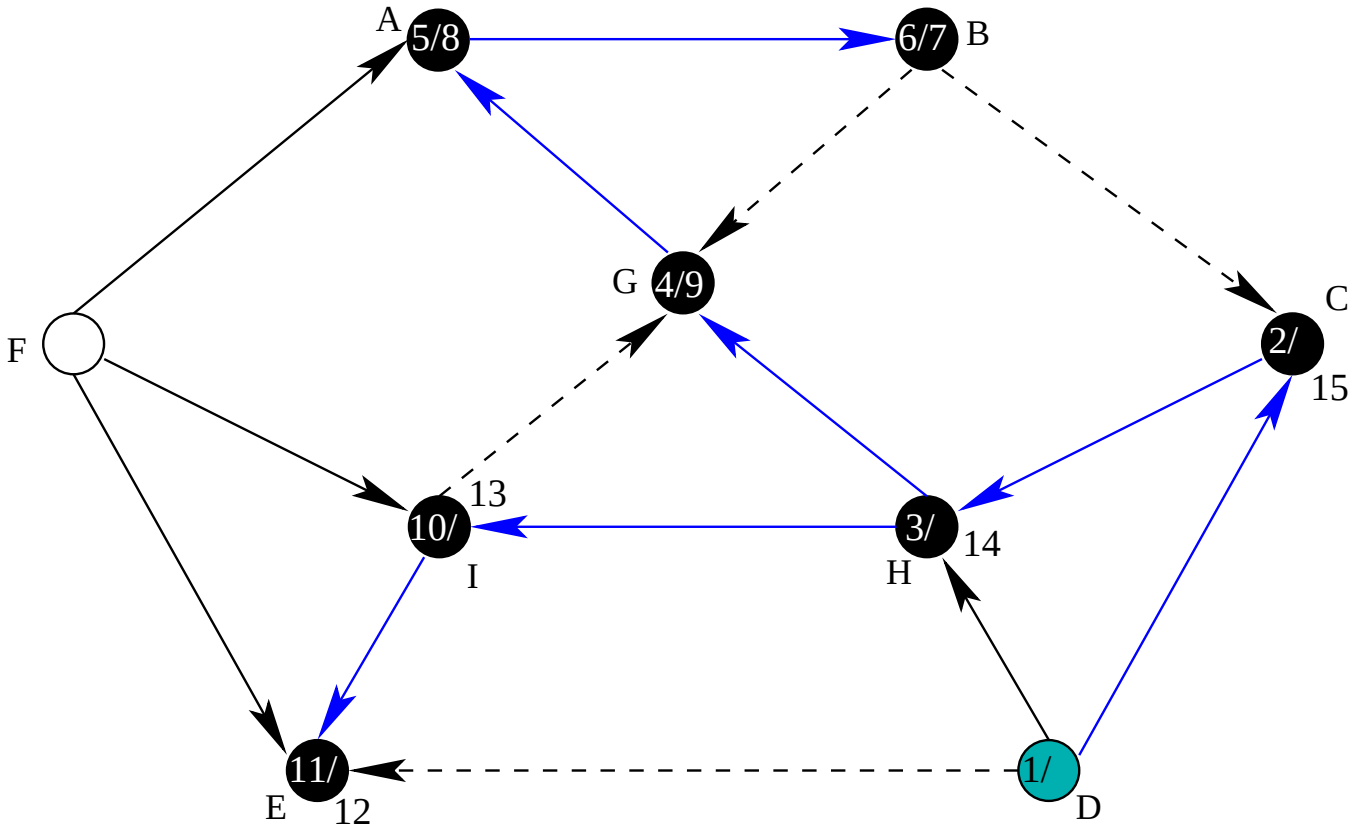
Exemplo de Execução



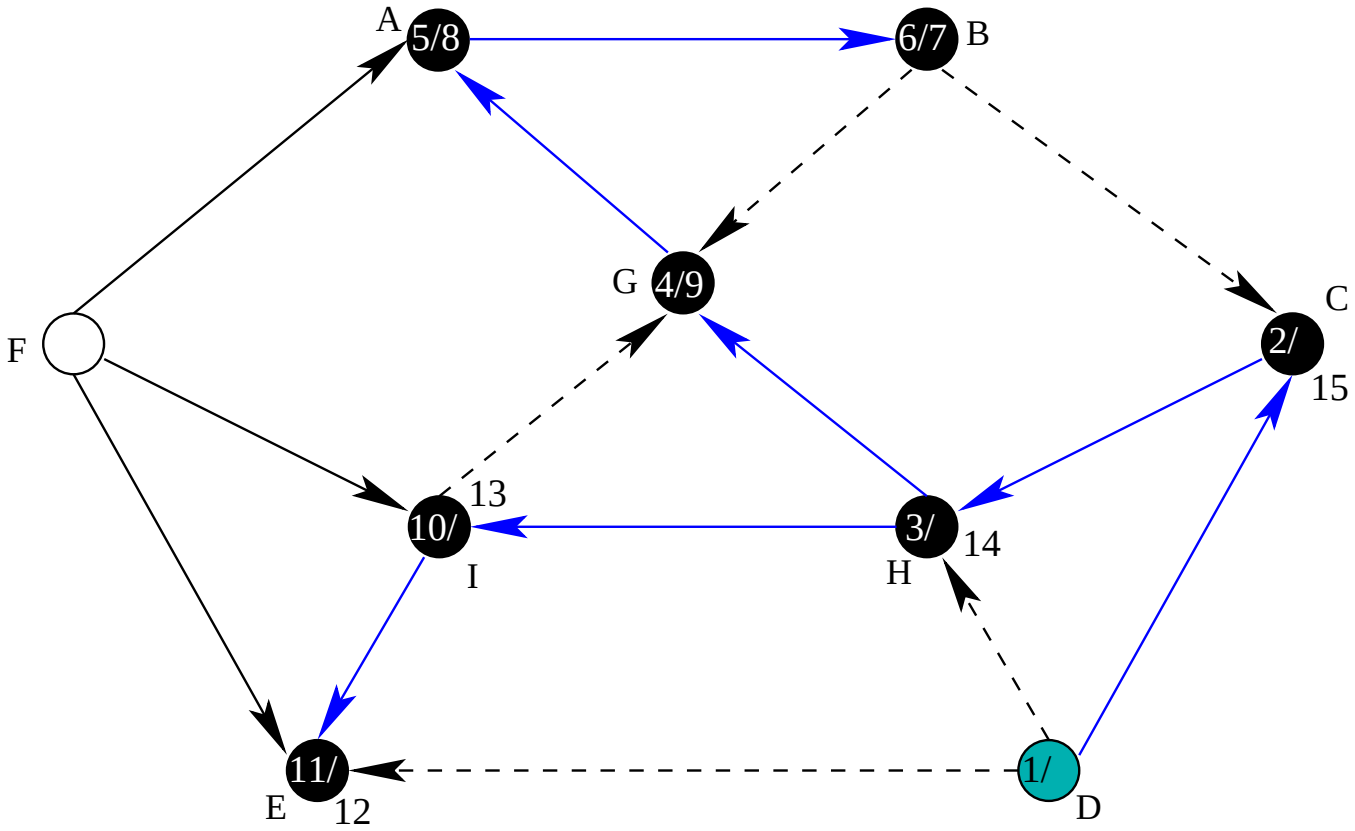
Exemplo de Execução



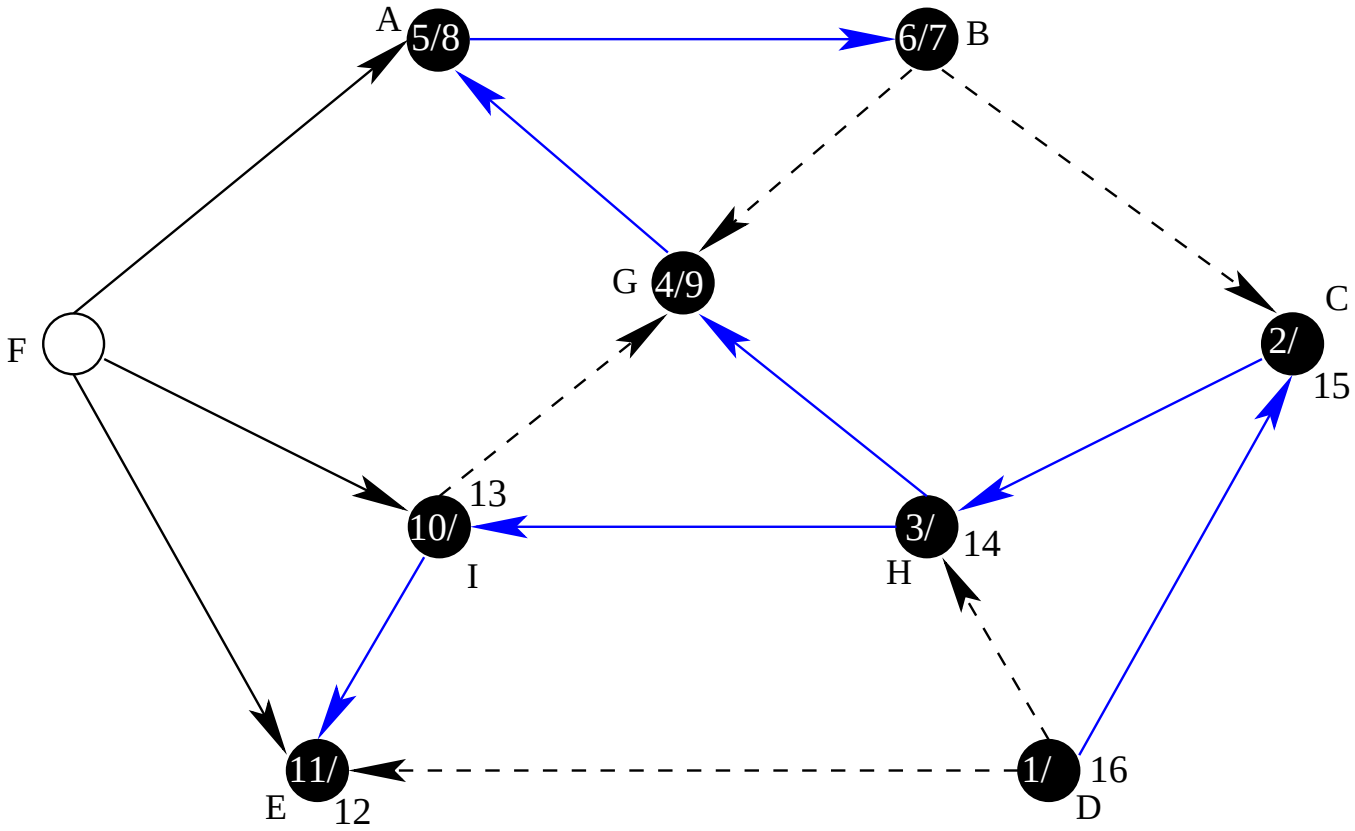
Exemplo de Execução



Exemplo de Execução

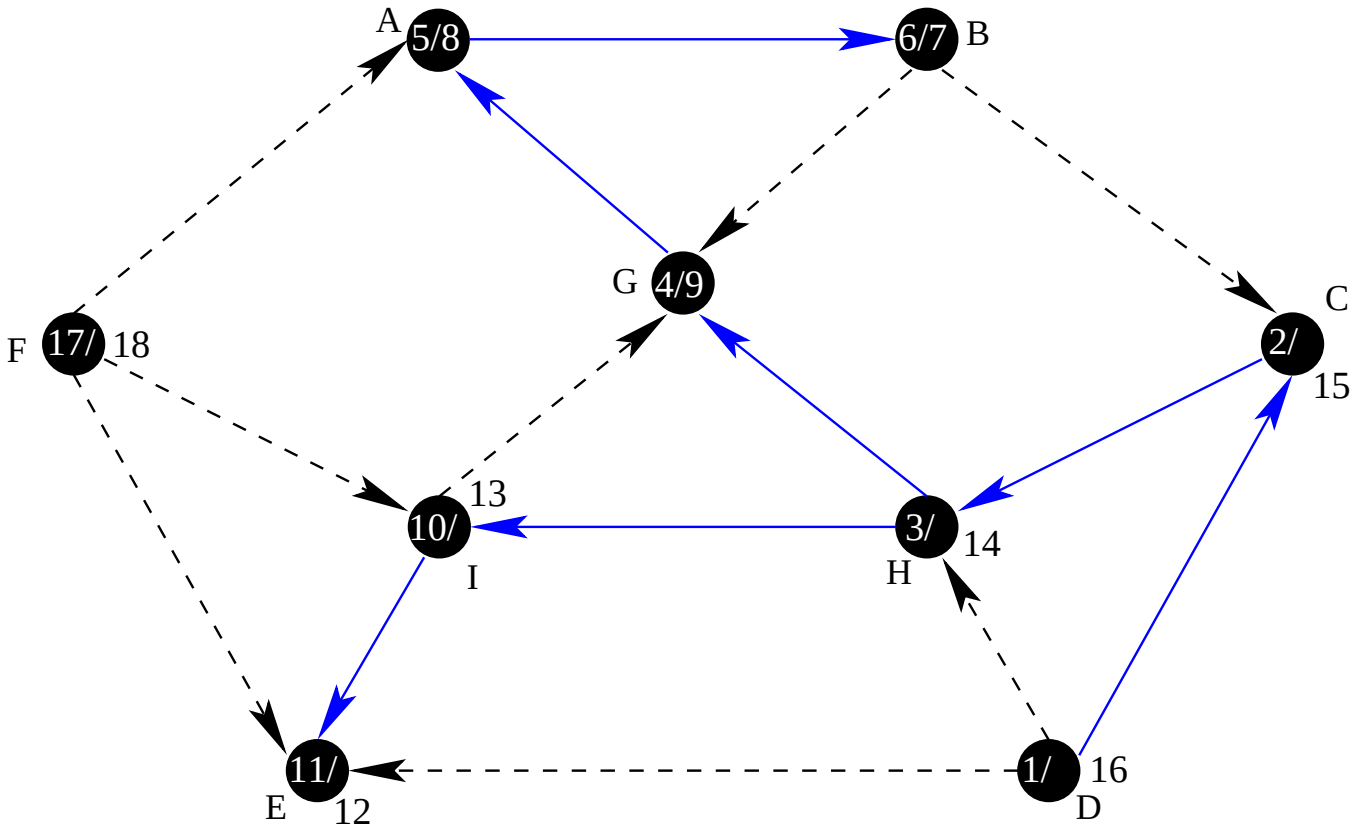


Exemplo de Execução



Exemplo de Execução

E depois da nova invocação DFS(G,F)



(D (C (H (G (A (B B) A) G) (I (E E) I) H) C) D) (F F)

Propriedades da Pesquisa em Profundidade

Teorema. [Parêntesis] *Em qualquer pesquisa em profundidade de* $G = (V, E)$, *tem-se para qualquer par de vértices* u, v *que uma e só uma das seguintes situações é válida:*

- *O intervalo* $[d[u], f[u]]$ *está contido em* $[d[v], f[v]]$ *e* u *é descendente de* v *numa APP;*
- *O intervalo* $[d[v], f[v]]$ *está contido em* $[d[u], f[u]]$ *e* v *é descendente de* u *numa APP;*
- *Os dois intervalos são disjuntos e nem* u *é descendente de* v *nem o contrário.*

Corolário. v *é um descendente de* u *em* G *se e só se*

$$d[u] < d[v] < f[v] < f[u]$$

Propriedades da Pesquisa em Profundidade

Teorema. [Caminhos Brancos] *Na floresta de pesquisa em profundidade de um grafo $G = (V, E)$, o vértice v é um descendente de u se e só se no instante $d[u]$ de descoberta de u, v é alcançável a partir de u por um caminho inteiramente constituído por vértices brancos.*

Por exemplo: B é descendente de G, mas E não o é (apesar de haver um caminho de G para E, este caminho não é inteiramente branco no momento em que G passa a cinzento).

Interlúdio – Análise Amortizada de Algoritmos

Uma nova ferramenta de análise, que permite estudar o tempo necessário para se efectuar uma *sequência de operações sobre uma estrutura de dados*.

- A ideia chave é considerar a média em relação à sequência de operações: uma operação singular pode ser pesada; no entanto, quando considerada globalmente como parte de uma sequência, o seu tempo médio de execução pode ser baixo.

- Trata-se do estudo do custo médio de cada operação no pior caso, e não de uma análise de caso médio!

3 técnicas: Análise agregada, Método contabilístico, Método do potencial

Análise agregada

Princípios:

- Mostrar que para qualquer n, uma sequência de n operações executa no pior caso em tempo $T(n)$.
- Então o *custo amortizado* por operação é $T(n)/n$.
- Considera-se que todas as diferentes operações têm o mesmo custo amortizado (as outras técnicas de análise amortizada diferem neste aspecto).

Análise agregada – Exemplo de Aplicação

Consideremos uma estrutura de dados do tipo **Pilha** ("Stack") com as operações `Push(S,x)`, `Pop(S)`, e `IsEmpty(S)` habituais. Cada uma executa em tempo constante (arbitraremos tempo 1).

Considere-se agora uma extensão deste tipo de dados com uma operação `MultiPop(S,k)`, que remove os `k` primeiros elementos da stack, deixando-a vazia caso contenha menos de `k` elementos.

```
MultiPop(S,k) {
  while (!IsEmpty(S) && k != 0) {
    Pop(S);
    k = k-1;
  }
```

Qual o tempo de execução de `MultiPop(S,k)`?

Análise de `MultiPop(S,k)`

- número de iterações do ciclo `while` é $\min(s, k)$ com s o tamanho da stack.
- em cada iteração é executado um `Pop` em tempo 1, logo o custo de `MultiPop(S,k)` é $\min(s, k)$.

Consideremos agora uma sequência de n operações `Push`, `Pop`, e `MultiPop` sobre a stack.

- Como o tamanho da stack é *no máximo* n, uma operação `MultiPop` na sequência executa no *pior caso* em tempo $O(n)$.
- Logo *qualquer operação* na sequência executa no pior caso em $O(n)$, e a sequência é executada em tempo $O(n^2)$.

Esta estimativa considera isoladamente o tempo no pior caso de cada operação. Apesar de correcta dá um limite superior demasiado "largo".

Análise agregada de `MultiPop(S,k)`

A análise agregada permite obter um limite mais apertado para o tempo de execução de uma sequência de n operações `Push`, `Pop`, e `MultiPop`

- Cada elemento é `popped` no máximo uma vez por cada vez que é `pushed`.
- Na sequência, `Pop` pode ser invocado (incluindo a partir de `MultiPop`) no máximo tantas vezes quantas as invocações de `Push` – no máximo n.
- O custo de execução da sequência é então $\leq 2n$ (1 por cada `Push` e `Pop`), ou seja, $O(n)$.
- O custo médio, ou custo amortizado, de cada operação, é então $O(n)/n = O(1)$.

Apesar de uma operação `MultiPop` isolada poder ser custosa ($O(n)$), o seu custo amortizado é $O(1)$.

Análise do Tempo de Execução de `BFS` e `DFS`

Utilizamos **Análise Agregada**. No caso da pesquisa em largura:

- Cada vértice é `enqueued` e `dequeued` exactamente uma vez. Isto é garantido pelo facto de os vértices nunca serem pintados de branco depois da inicialização.
- `enqueue` e `dequeue` executam em tempo $O(1)$, logo o tempo total gasto em operações sobre a Queue é $O(|V|)$.
- A lista de adjacência de cada vértice é percorrida no máximo uma vez (quando o vértice é dequeued), e o comprimento *total* das listas é $\Theta(|E|)$. Logo, o tempo total tomado pela travessia das listas de adjacências é $O(|E|)$.
- A *inicialização* do algoritmo é feita em tempo $O(|V|)$.
- Assim, o tempo de execução de `BFS` é $O(|V| + |E|)$
  – linear no tamanho da representação por listas de adjacências de G.

Análise do Tempo de Execução de `BFS` e `DFS`

E para a pesquisa em profundidade:

- Em `CDFS`, os ciclos `for` executam em tempo $O(|V|)$, excluindo o tempo tomado pelas invocações de `DFS`.
- `DFS` é invocada *exactamente uma vez* para cada vértice do grafo (a partir de `CDFS` ou da própria `DFS`) – garantido porque é invocada apenas com vértices brancos e a primeira coisa que faz é pintá-los de cinzento (e nenhum vértice volta a ser pintado de branco).
- Em `DFS(`G`,s)`, o ciclo `for` é executado $Adj(s)$ vezes. No total das invocações de `DFS`, este ciclo é então executado $\sum_{v \in V} |Adj(v)| = \Theta(|E|)$.
- O tempo de execução de `CDFS` é então $\Theta(|V| + |E|)$ – também linear no tamanho da representação.

Aplicações da Pesquisa em Grafos

- **Ordenação topológica** de um grafo acíclico orientado: determinar uma ordem linear dos nós tal que $(u, v) \in E$ implica que u aparece antes de v nessa ordem.

  Este algoritmo permite determinar ordens possíveis de execução de tarefas (representadas pelos nós) quando os arcos representam restrições de precedência.

- **Identificação de componentes fortemente ligados**: pode ser resolvido efectuando-se duas pesquisas, uma no grafo original e outra no seu transposto.

  $\Rightarrow$ Aulas teórico-práticas

Árvores Geradoras Mínimas

("Minimum spanning trees" – MST)

Seja $G = (V, E)$ um grafo *não-orientado*, *ligado*. Uma árvore geradora de G é um sub-grafo (V, T) acíclico e ligado de G.

Observe-se que (V, T) é necessariamente uma árvore ($\Rightarrow$ porquê?), e que liga todos os vértices de G entre si.

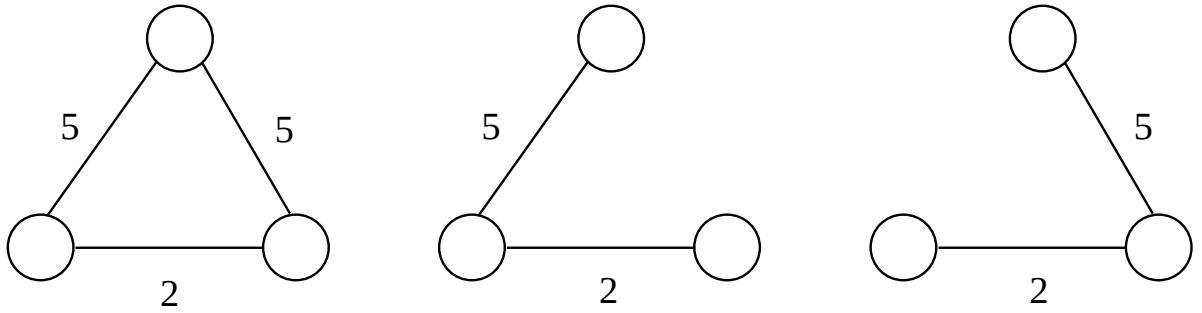
Associe-se agora a cada arco $(u, v) \in E$ um peso $w(u, v)$ numérico. As árvores geradoras mínimas de G são aquelas para as quais o peso total

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

é mínimo.

Árvores Geradoras Mínimas

As MSTs não são únicas:



Exemplo de aplicação: ligação eléctrica de um número de "pins" num circuito integrado. Cada fio liga um par de "pins"; o peso corresponde à quantidade de cobre necessária na ligação. Pretende-se minimizar a quantidade total de cobre.

A determinação de uma MST ocorre como sub-problema de muitos outros!

Algoritmo de Prim para Determinação de MSTs

Considera em cada instante da execução o conjunto de vértices dividido em 3 conjuntos disjuntos:

1. os vértices da árvore construída até ao momento;

2. os vértices na orla (alcançáveis a partir dos anteriores);

3. os restantes vértices.

Em cada passo selecciona-se um arco (com origem em 1 e destino em 2) para acrescentar à árvore. O vértice destino desse arco é também acrescentado.

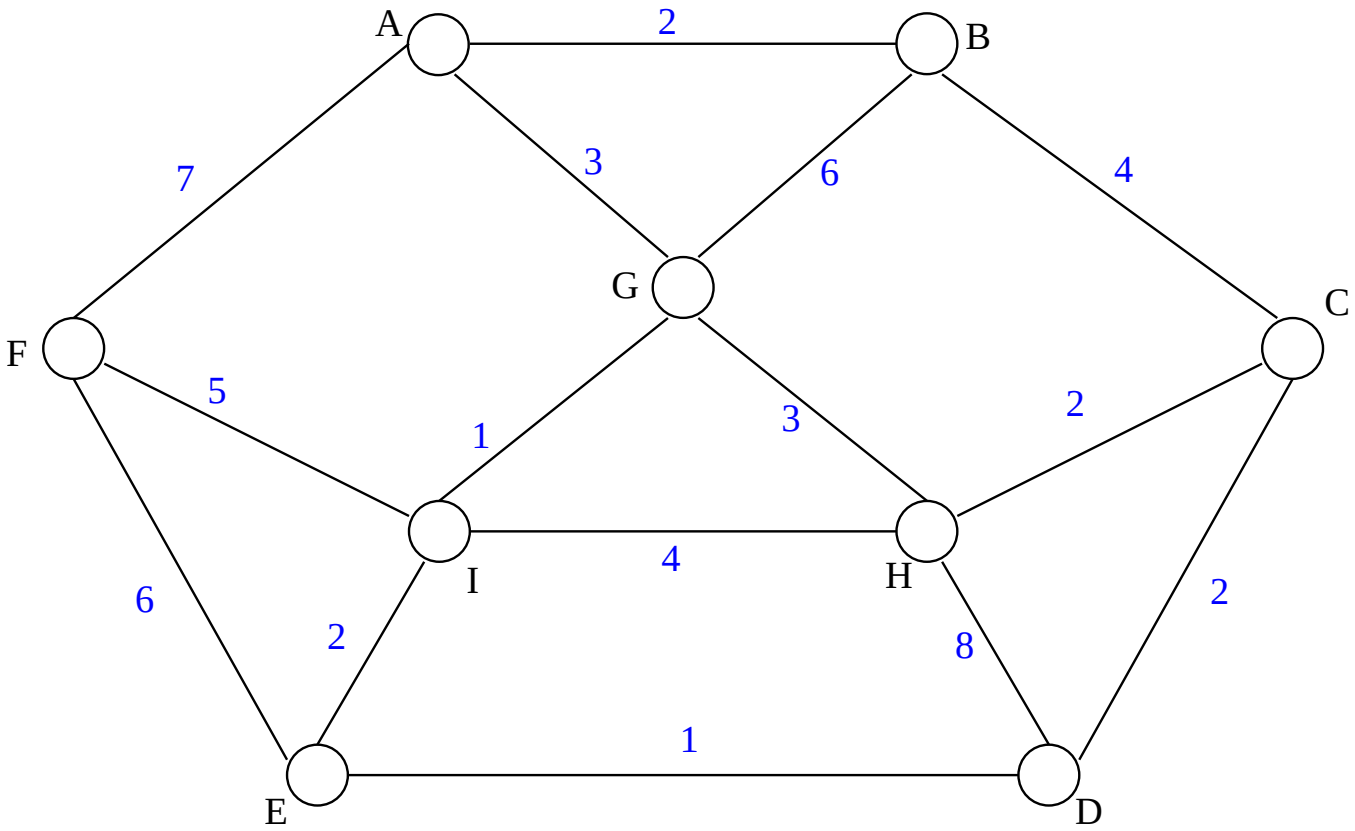
O algoritmo de Prim selecciona sempre o arco com *menor peso* nestas condições.

Estrutura Geral do Algoritmo

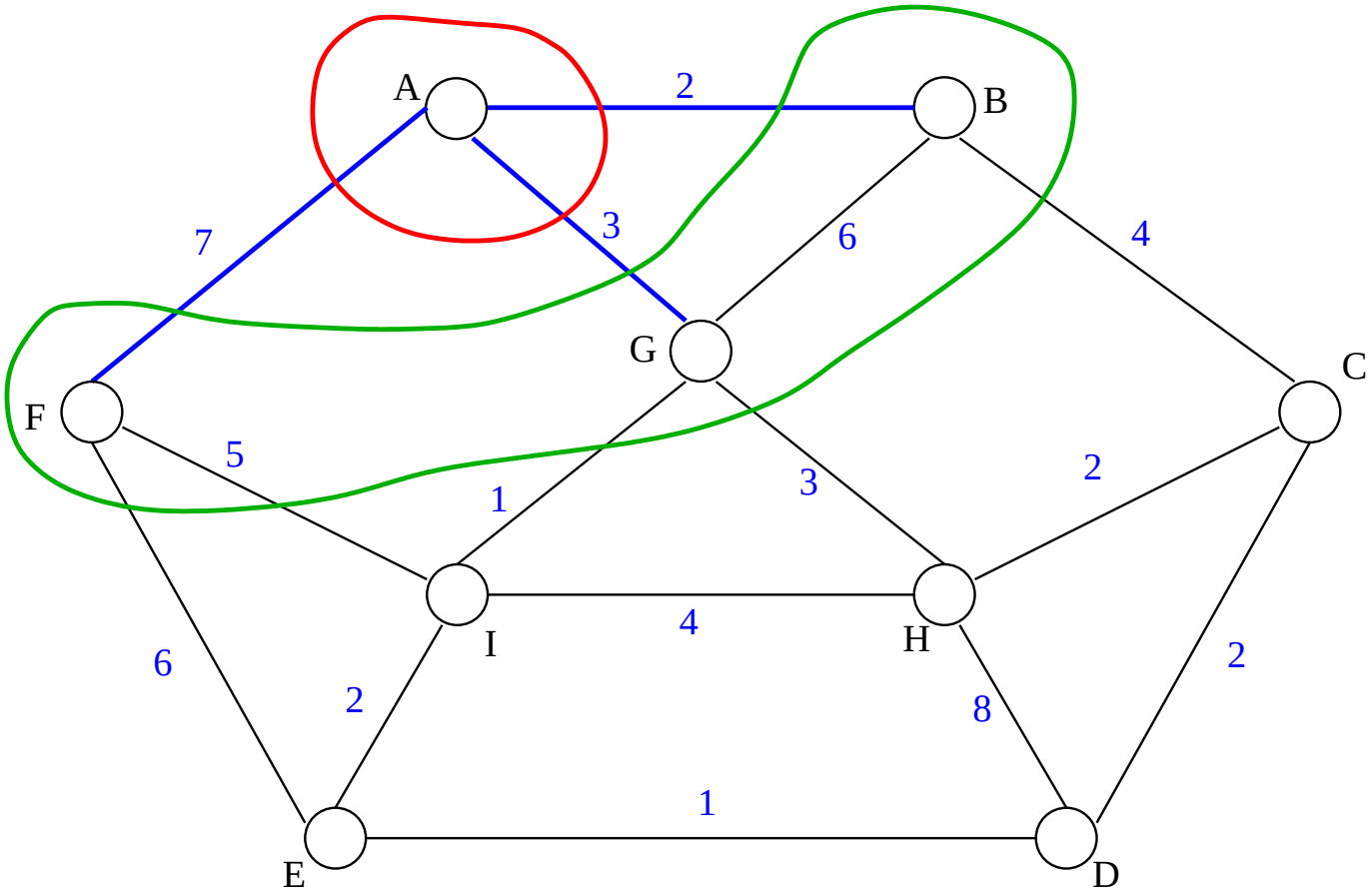
```
void MST((V, E)) {                          /* G = (V, E) */
  seleccionar nó arbitrário x para início da árvore;
  while (existem vértices na orla) {
    seleccionar um arco de peso mínimo entre um nó da árvore
      e um nó na orla;
    acrescentar esse arco e o respectivo vértice-destino à árvore;
  }
}
```

O exemplo seguinte mostrará que não é necessário examinar *todos* os arcos com origem na árvore actual e destino na orla. Para cada nó da orla, basta considerar um arco (o de menor peso) com origem na árvore – o *arco candidato* desse nó.

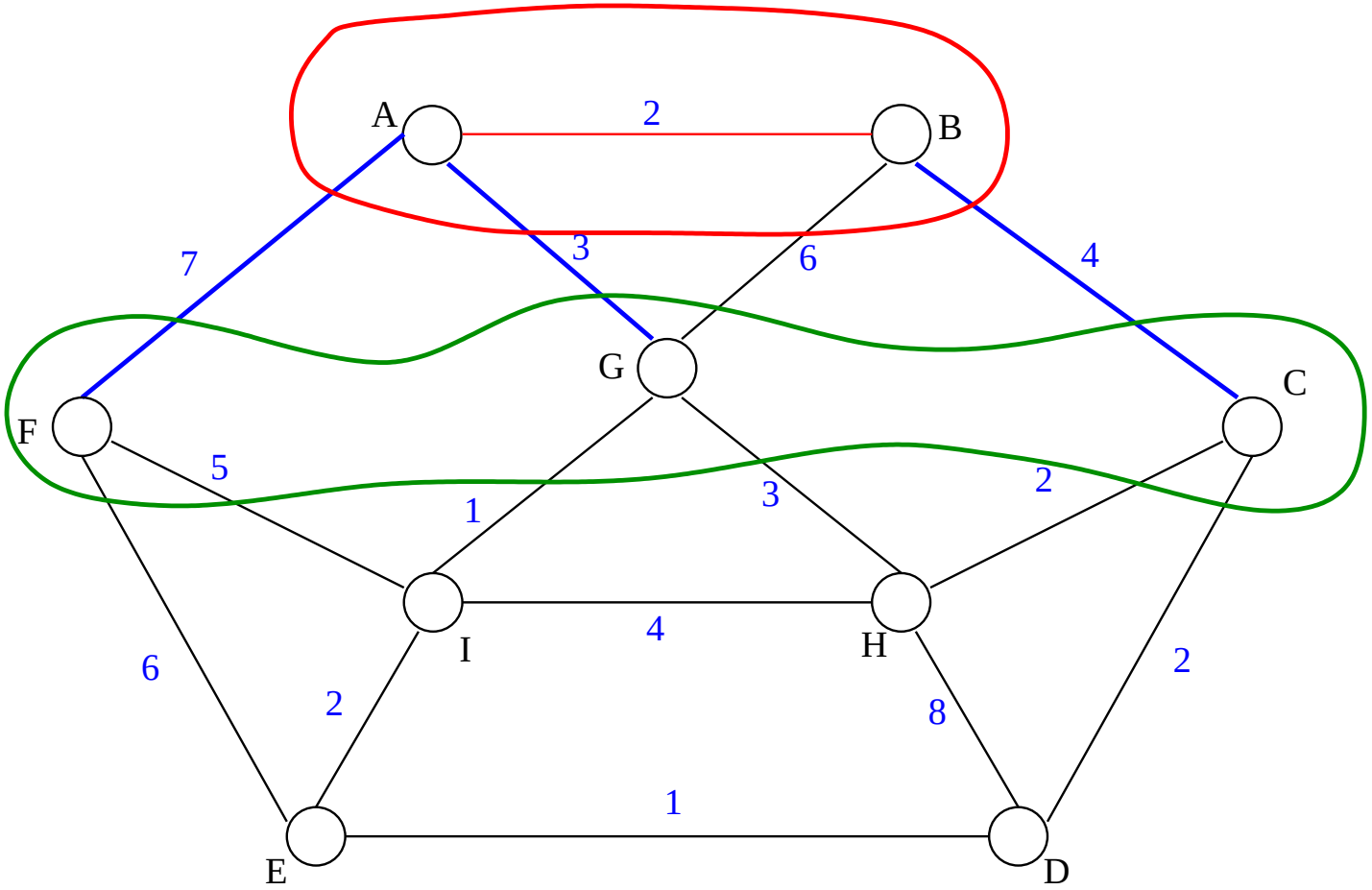
Algoritmo MST de Prim – Exemplo



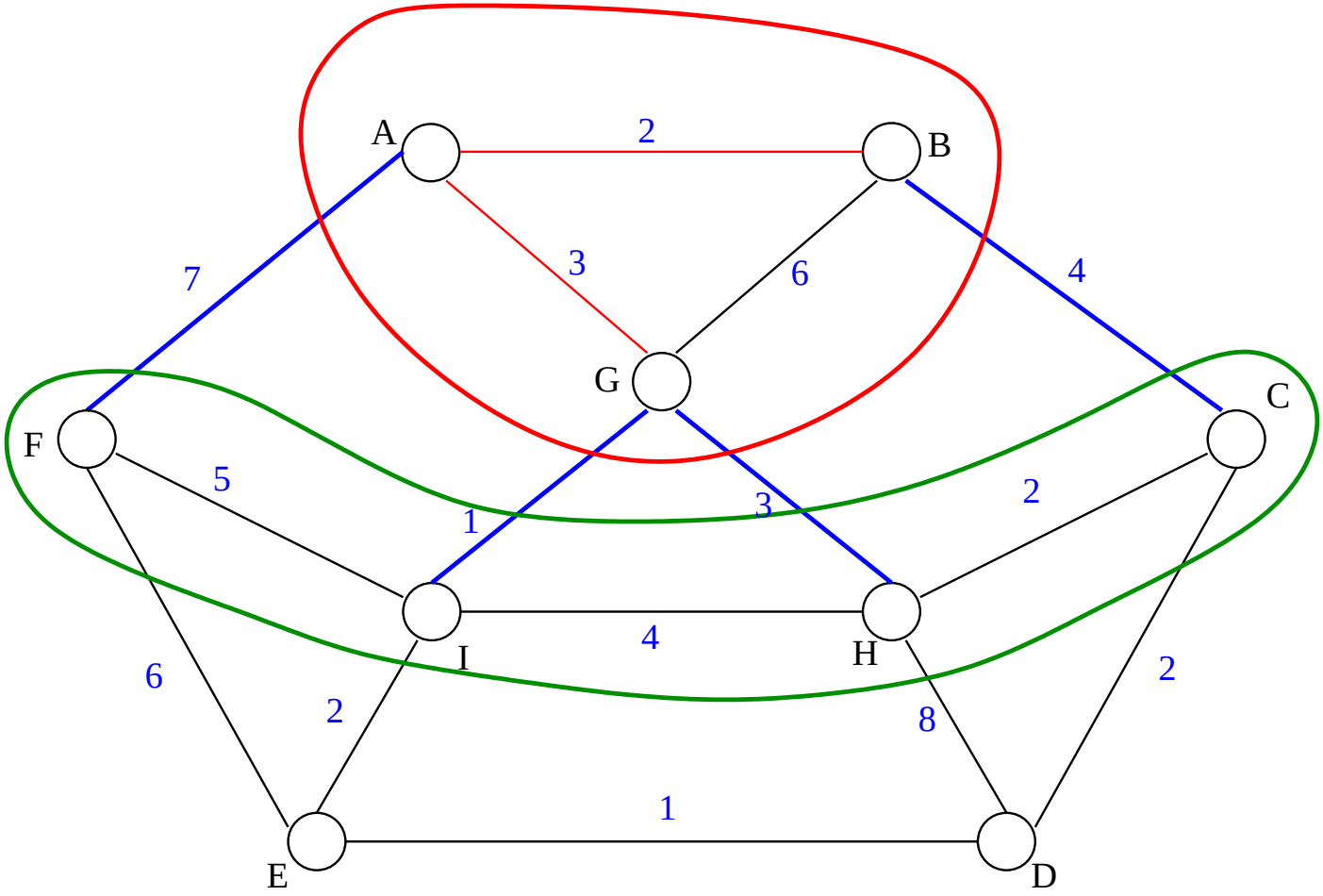
Algoritmo MST de Prim – Exemplo



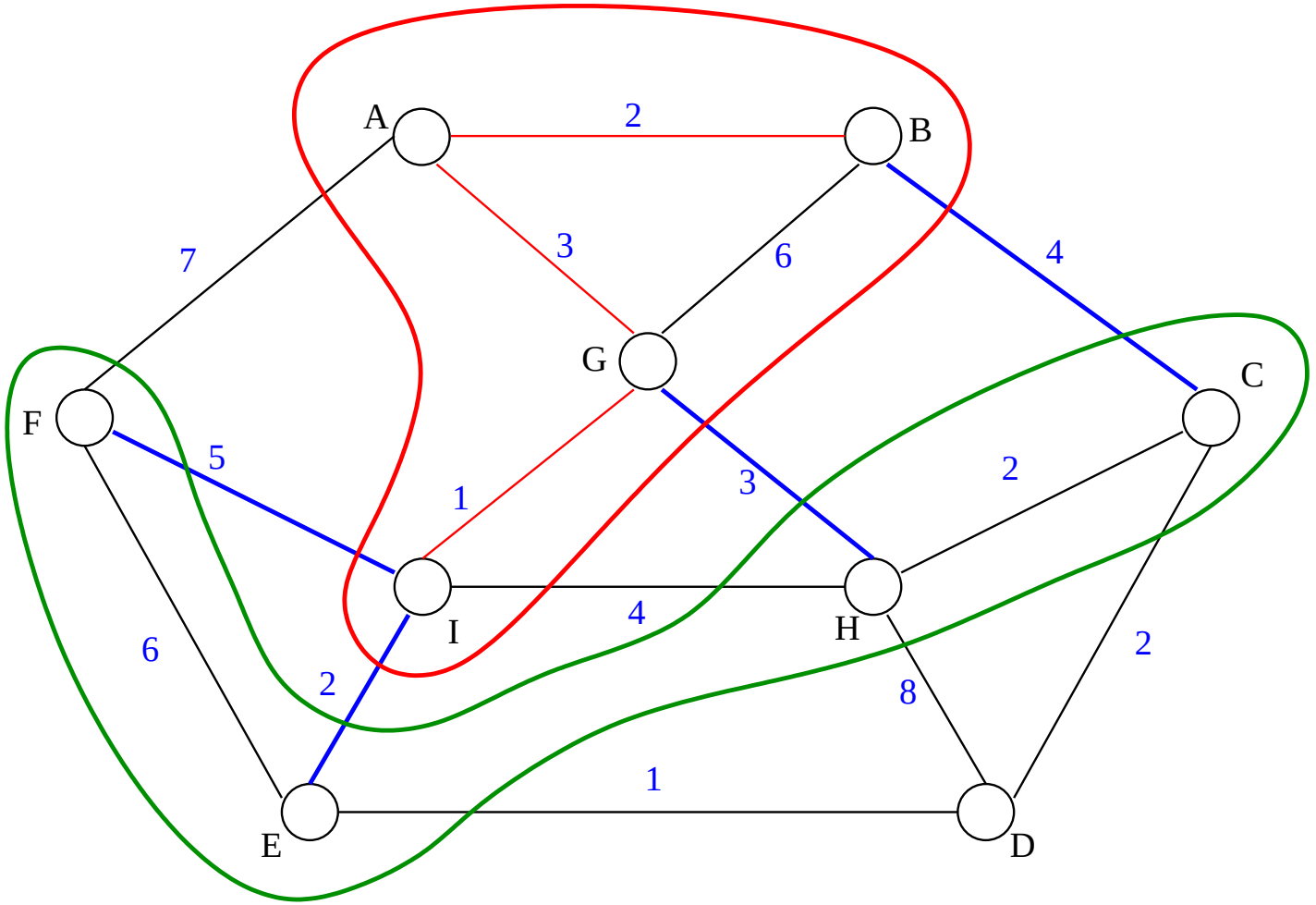
Algoritmo MST de Prim – Exemplo



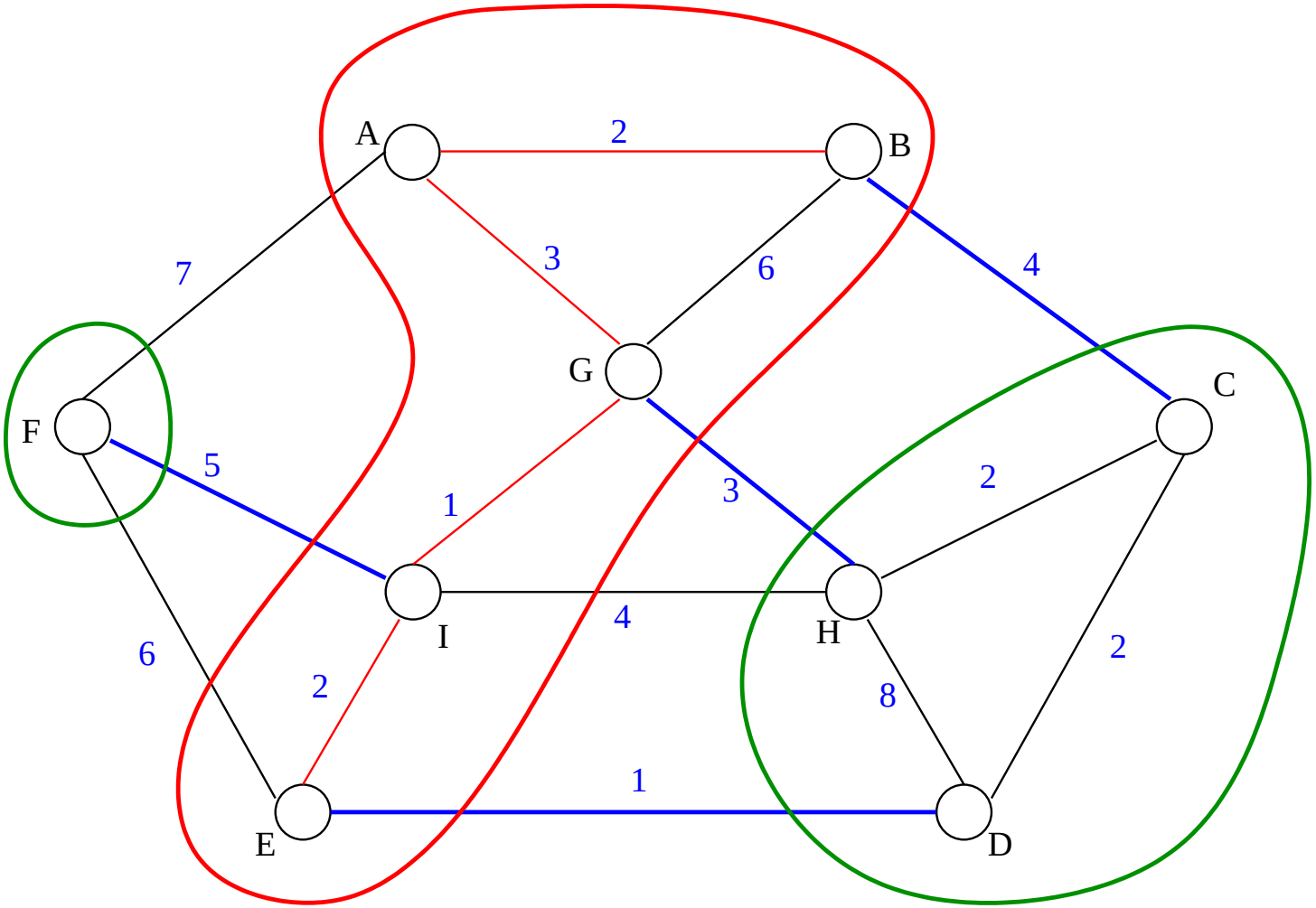
Algoritmo MST de Prim – Exemplo



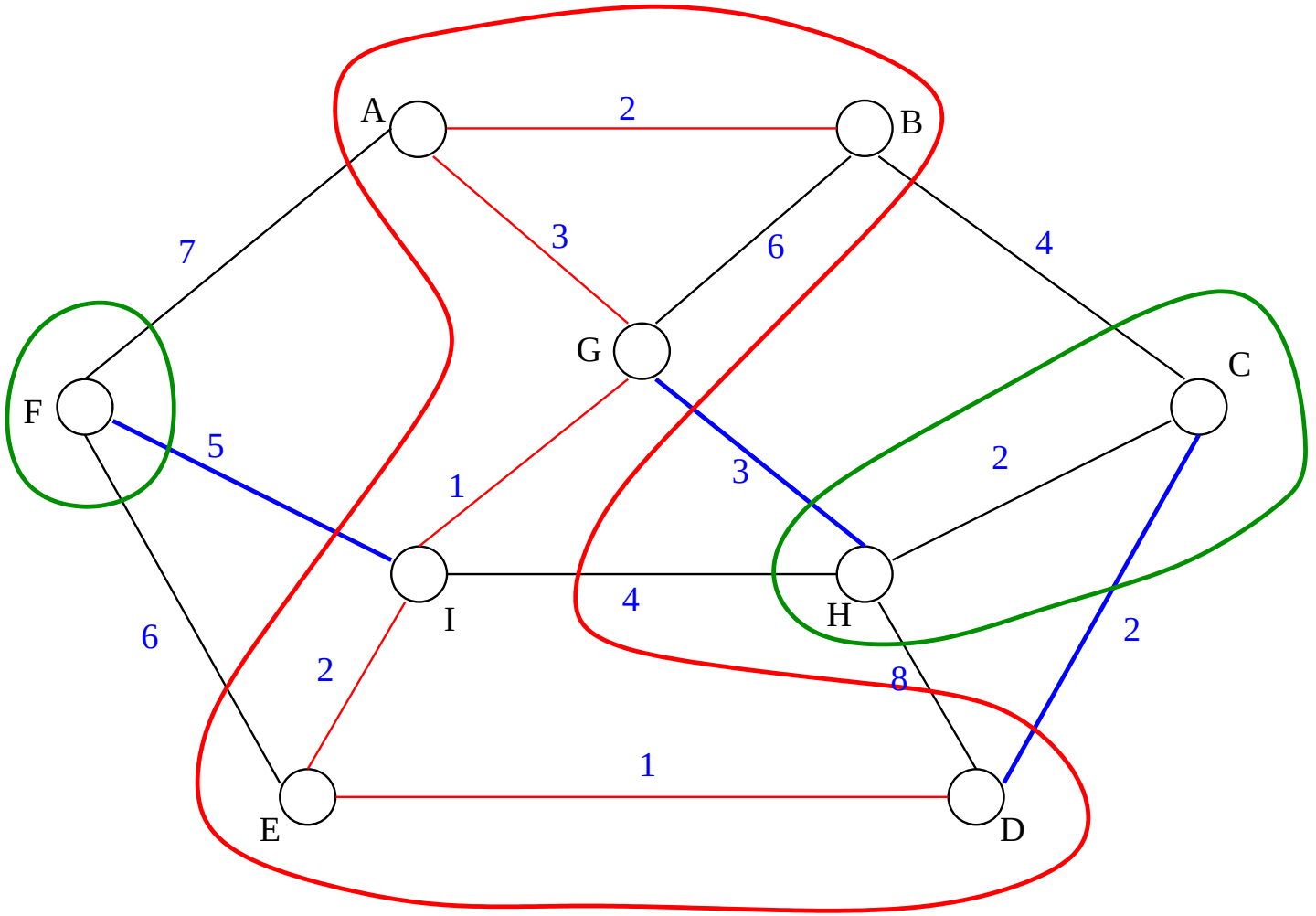
Algoritmo MST de Prim – Exemplo



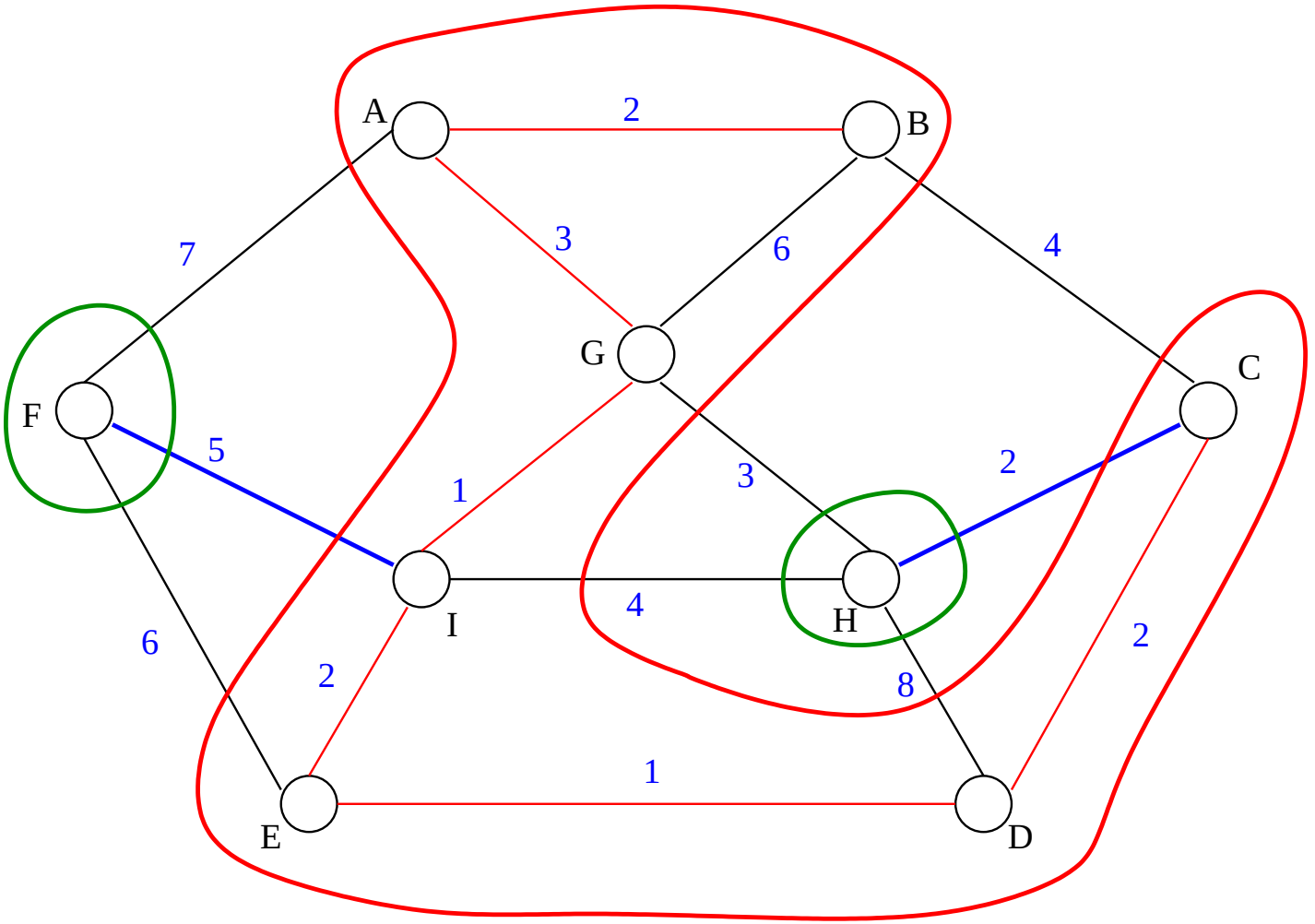
Algoritmo MST de Prim – Exemplo



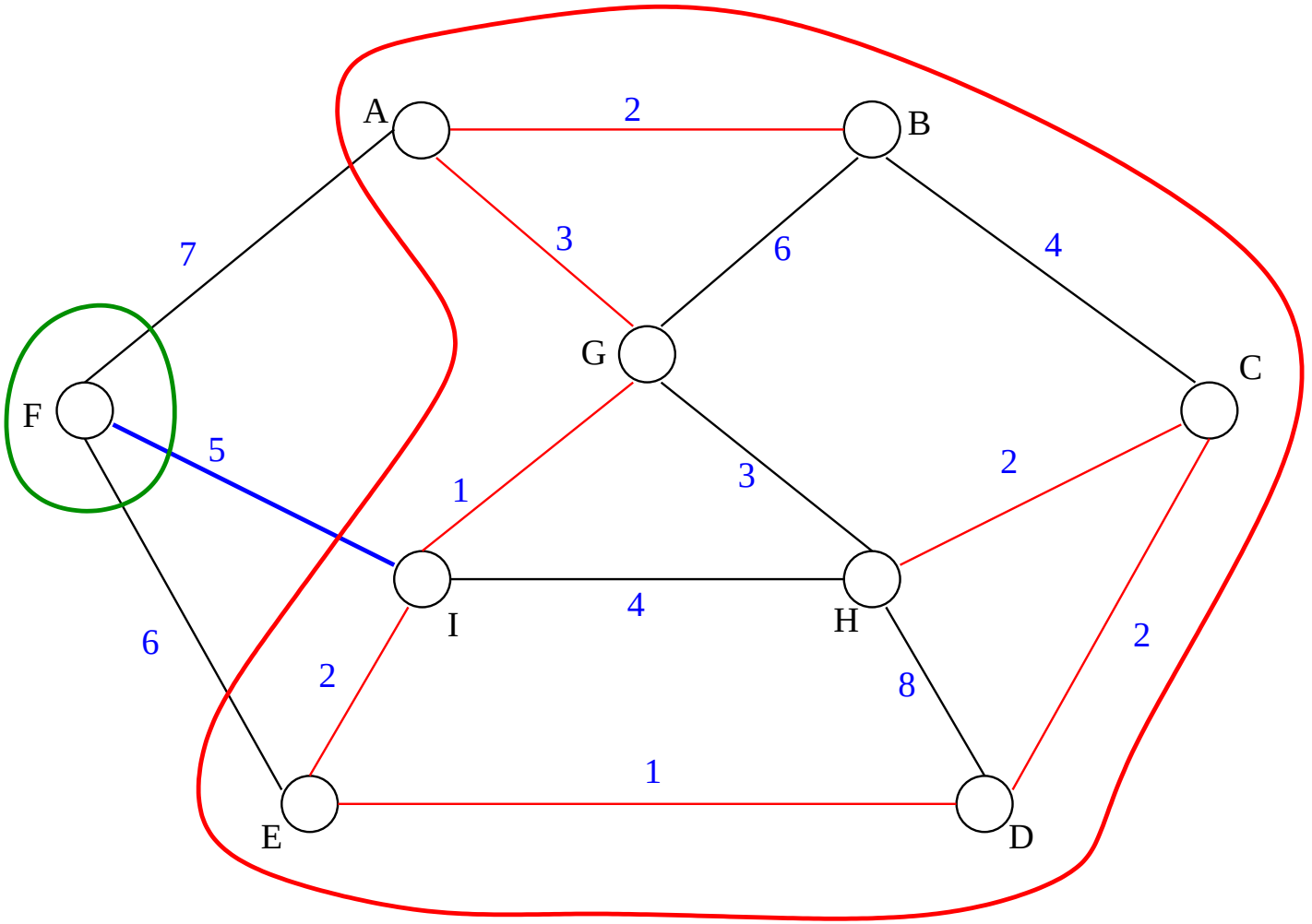
Algoritmo MST de Prim – Exemplo



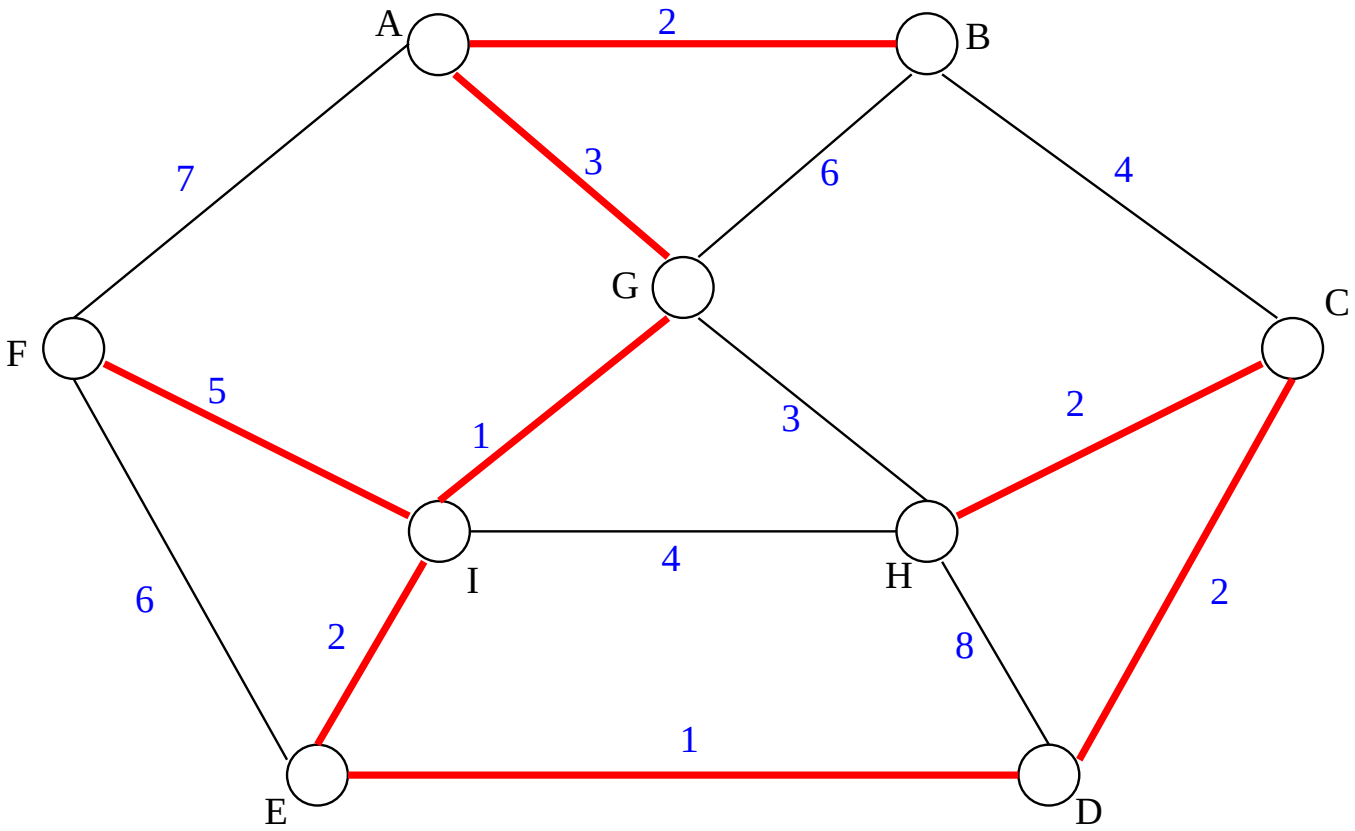
Algoritmo MST de Prim – Exemplo



Algoritmo MST de Prim – Exemplo



Algoritmo MST de Prim – Exemplo



Correcção do Algoritmo de Prim

Teorema. *Seja $G = (V, E)$ um grafo orientado ligado, e (V, T) uma árvore geradora mínima de G. Seja ainda (V', T') uma sub-árvore ligada de (V, T). Se (u, v) é um arco de peso mínimo tal que $u \in V'$ e $v \notin V'$, então $(V' \cup \{v\}, T' \cup (u, v))$ é uma sub-árvore de uma árvore geradora mínima $(V, \widehat{T})$ de G.*

Prova.

- *Se $(u, v) \in T$, a prova é imediata e $(V, \widehat{T}) = (V, T)$.*

- *Caso contrário, existe um caminho de u para v em (V, T); os primeiros nós desse caminho estão em V'. Seja (w, x) o primeiro arco tal que $w \in V'$ e $x \notin V'$, e $\widehat{T} = T - \{(w, x)\} \cup \{(u, v)\}$. Então:*
  - *$(V, \widehat{T})$ é uma árvore geradora ($\Rightarrow$ porquê?);*
  - *$(V, \widehat{T})$ tem custo mínimo: $w(u, v) \leq w(x, y)$, logo $w(V, \widehat{T}) \leq w(V, T)$;*
  - *$(V' \cup \{v\}, T' \cup (u, v))$ é uma sub-árvore de $(V, \widehat{T})$.*

Algoritmo Detalhado

```
void MST((V, E)) {                          /* G = (V, E) */
  V' = {x}; T' = ∅;                  /* x escolhido arbitrariamente*/
  stuck = 0;
  while (V' ≠ V && !stuck) {
    for (y ∈ orla, y adjacente a x)
      if (w(x, y) < w(arco candidato de y))
        substituír arco candidato de y por (x, y);
    for (y ∉ V', y ∉ orla, y adjacente a x) {
      colocar y na orla;
      marcar (x, y) arco candidato;
    }
    if (não há arcos candidatos) stuck = 1;
    else { escolher arco candidato (u, v) de custo mínimo; x = v;
           V' = V' ∪ {x}; T' = T' ∪ {(u, v)};
           remover x da orla;
           desmarcar (u, v) como candidato;
} } }
```

Observações

- No fim da execução, se `stuck == 0`, (V', T') é uma MST.
- Em que circunstâncias termina o algoritmo com `stuck == 1`?
- Como proceder para obter uma floresta de MSTs para um grafo não ligado?

Correcção do Algoritmo de Prim – Invariante de Ciclo

No início de cada iteração do ciclo `while`, (V', T') é uma sub-árvore de uma árvore geradora mínima de G.

Inicialização $(\{x\}, \emptyset)$ é sub-árvore de todas as árvores geradoras de G . . .

Preservação dada pelo Teorema anterior

Terminação no fim da execução do ciclo, se `stuck != 0`, $V' = V$ logo $(V'T')$ é uma árvore geradora mínima de G

Tempo de Execução: Limite Inferior

Teorema: *Qualquer algoritmo* de construção de MSTs examina necessariamente todos os arcos do grafo, logo executa em tempo $\Omega(|E|)$.

Prova. *Por contradição:*

- *Seja $G = (V, E)$ um grafo ligado cujos arcos têm todos peso 2.*
- *Imaginemos que existe um algoritmo que constrói uma árvore geradora mínima (V, T) sem examinar um arco (x, y) de G. Então T não contém (x, y).*
- *T contém forçosamente um caminho c de x para y; G contém um ciclo constituído por esse caminho e pelo arco (x, y).*
- *Se alterarmos o peso $w(x, y)$ para 1, isso não altera a acção do algoritmo (uma vez que não considera esse arco). No entanto podemos construír uma árvore geradora com peso inferior a (V, T): basta substituír qualquer arco de c por (x, y) – contradição!*

Detalhes de implementação do Algoritmo de Prim

- Grafo implementado por listas de adjacências;
- são mantidos vectores adicionais para o estado (na árvore, na orla, nos restantes) de cada vértice, para a construção da árvore (vector *parent* como nas pesquisas), e para o peso dos arcos candidatos;
- mantida uma lista ligada correspondente aos nós na orla.

Estas escolhas utilizam bastante espaço mas permitem acelerar a execução:

- A pesquisa e remoção do arco candidato de menor peso é feita por uma travessia da orla e consulta directa dos pesos dos arcos candidatos;
- a substituição de um arco candidato é feita facilmente alterando-se o vector *parent* e o vector de pesos dos arcos candidatos.

Tempo de Execução do Algoritmo de Prim

Análise agregada (pior caso) sobre $G = (V, E)$:

- Número de operações de inicialização é linear em $|V|$.
- Os dois ciclos `for` podem ser fundidos num único que atravessa vértices adjacentes a x. Este ciclo atravessa completamente todas as listas de adjacências, e como o seu corpo é executado em tempo constante, demora $\Theta(|E|)$.
- Teste `if` $(\text{não há arcos candidatos})$ é executado no máximo $|V| - 1$ vezes.
- Número total de comparações está em $O(|V|^2)$. $(\Rightarrow$ porquê?)

Então, o algoritmo executa em tempo $O(|V|^2 + |E|)$.

Caminhos Mais Curtos

("Shortest Paths" – SP)

Seja $G = (V, E)$ um grafo pesado orientado ou não-orientado, e $P = v_0, v_1, \ldots v_k$ um caminho nesse grafo. O *peso* do caminho P define-se como

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Um caminho P do vértice v para o vértice w diz-se um caminho mais curto entre v e w se não existe nenhum caminho de v para w com peso inferior a $w(P)$. P não é necessariamente único.

O nome vem da analogia geográfica, quando os vértices representam *locais* e os pesos *distâncias* entre eles. Outras aplicações: pesos podem representar *custos*, por exemplo de viagens entre locais.

Caminhos Mais Curtos

O problema: dado um grafo G e dois vértices v e w nesse grafo, determinar um caminho mais curto entre eles.

Questão prévia: se construírmos uma *árvore geradora mínima* com origem em v, será o caminho (único) de v para w contido nessa árvore um caminho mais curto? A resposta pode ser encontrada no exemplo anterior . . .

Uma estratégia imediata: força bruta – construír *todos* os caminhos de v para w ($\Rightarrow$ como?); seleccionar o mais curto.

Veremos em seguida um algoritmo muito mais eficiente.

Uma definição necessária: a *distância* $d(x, y)$ do vértice x ao vértice y é o peso de um caminho mais curto de x para y.

Algoritmo de Dijkstra para Determinação de SPs

- Muito semelhante ao algoritmo de PRIM para MSTs.
- Selecciona em cada passo um vértice da orla para acrescentar à árvore que vai construindo.
- O algoritmo vai construindo caminhos cada vez mais longos (no sentido do peso maior) a partir de v, dispostos numa árvore; pára quando encontrar w.
- A grande diferença em relação ao algorito de Prim é o *critério de selecção do arco candidato*.
- A *análise do tempo de execução* é análoga.

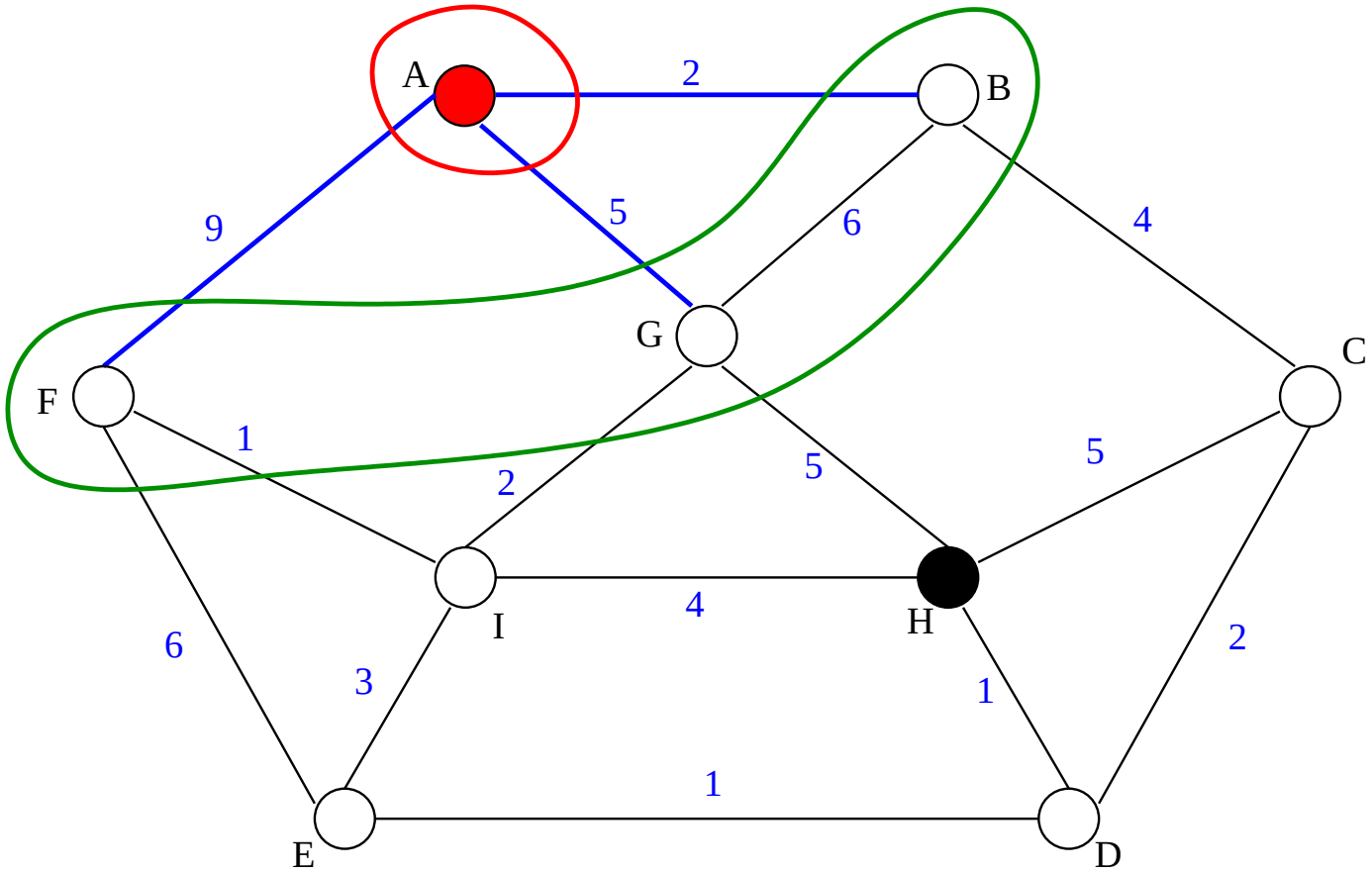
Algoritmo de Dijkstra – Arcos Candidatos

- Para cada nó z na orla, existe um caminho mais curto $v, v_1, \ldots v_k$ na árvore construída, tal que $(v_k, z) \in E$.

- Se existirem vários caminhos $v, v_1, \ldots v_k$ e arco (v_k, z) nas condições anteriores, o arco candidato (único) de z será aquele para o qual $d(v, v_k) + w(v_k, z)$ fôr mínimo.

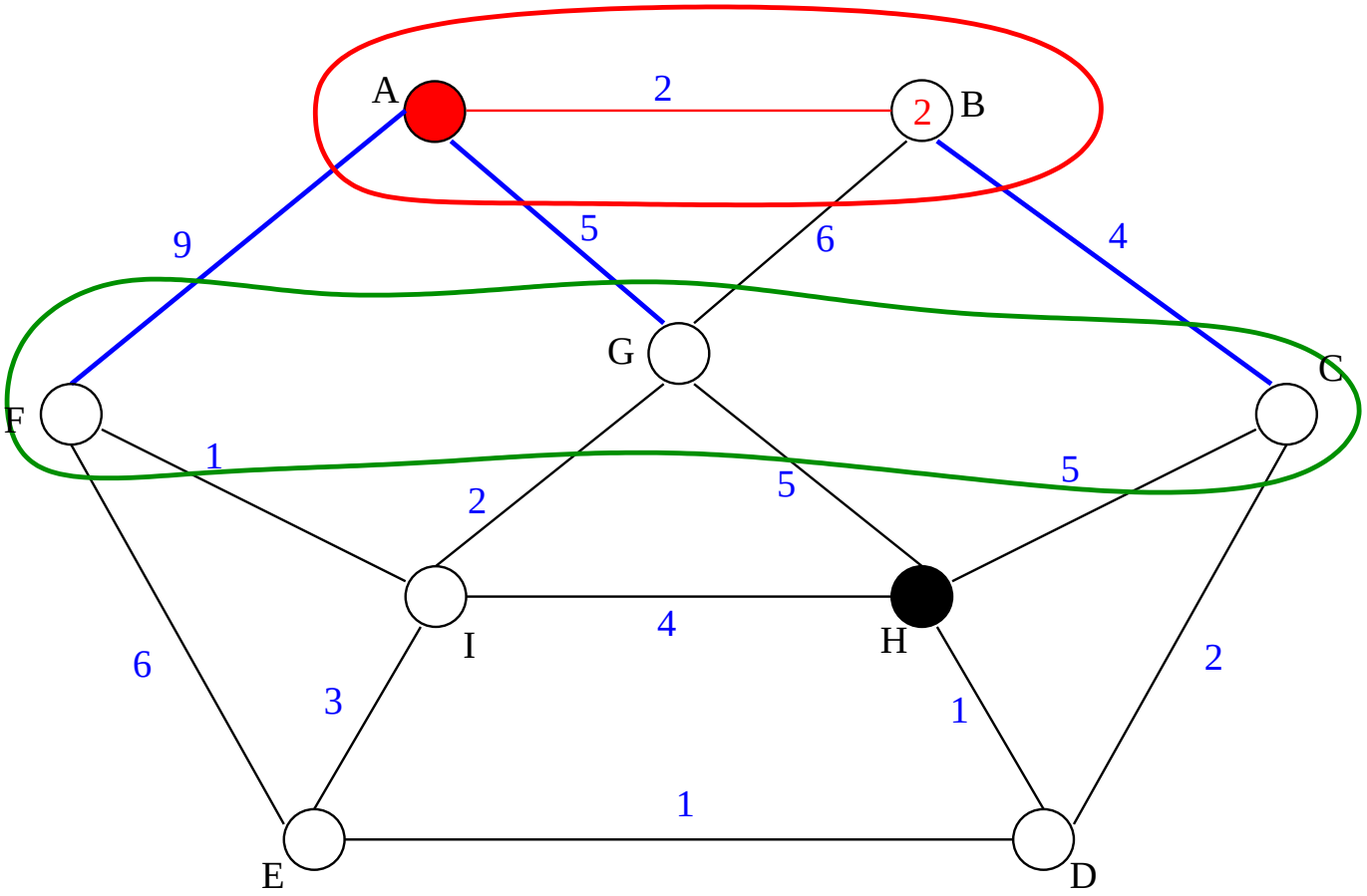
- Em cada passo, o algoritmo selecciona um vértice da orla para acrescentar à árvore. Este será o vértice z com valor $d(v, v_k) + w(v_k, z)$ mais baixo.

Algoritmo SP de Dijkstra – Exemplo



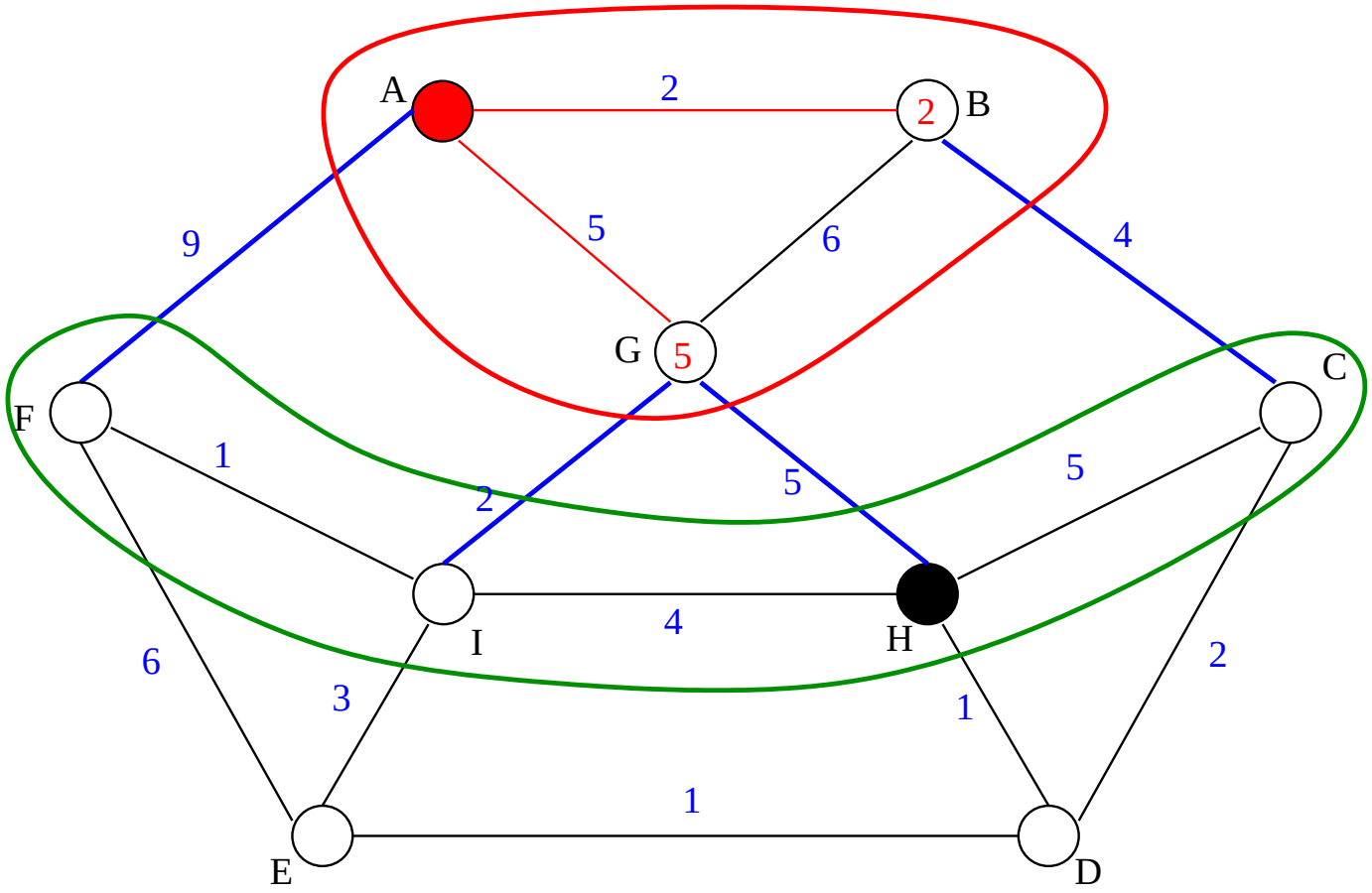
$$d(A,A)+w(A,B)=2; \quad d(A,A)+w(A,F)=9; \quad d(A,A)+w(A,G)=5.$$

Algoritmo SP de Dijkstra – Exemplo



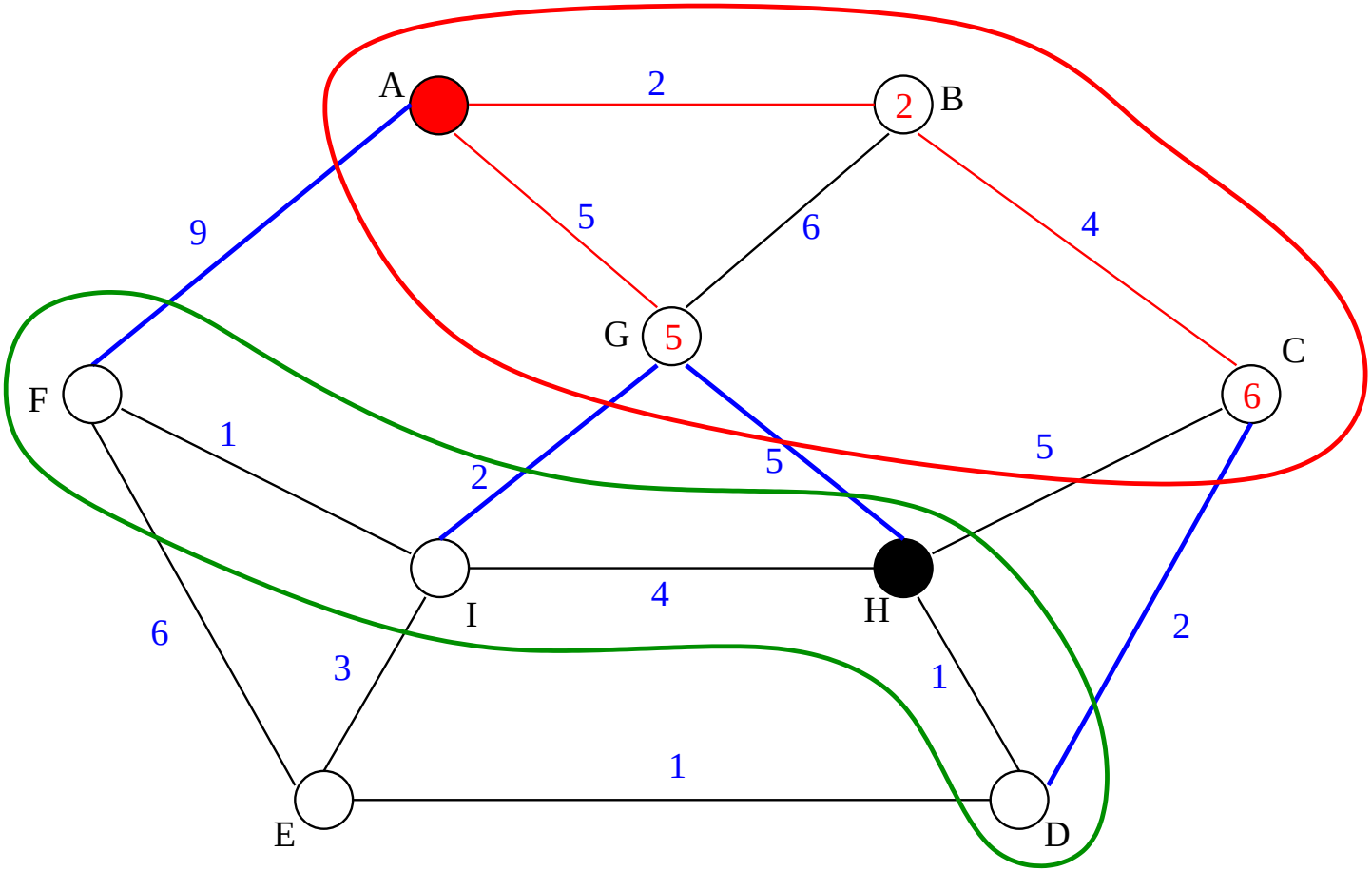
$$d(A,B) + w(B,C) = 6; \quad d(A,A) + w(A,F) = 9; \quad d(A,A) + w(A,G) = 5.$$

Algoritmo SP de Dijkstra – Exemplo



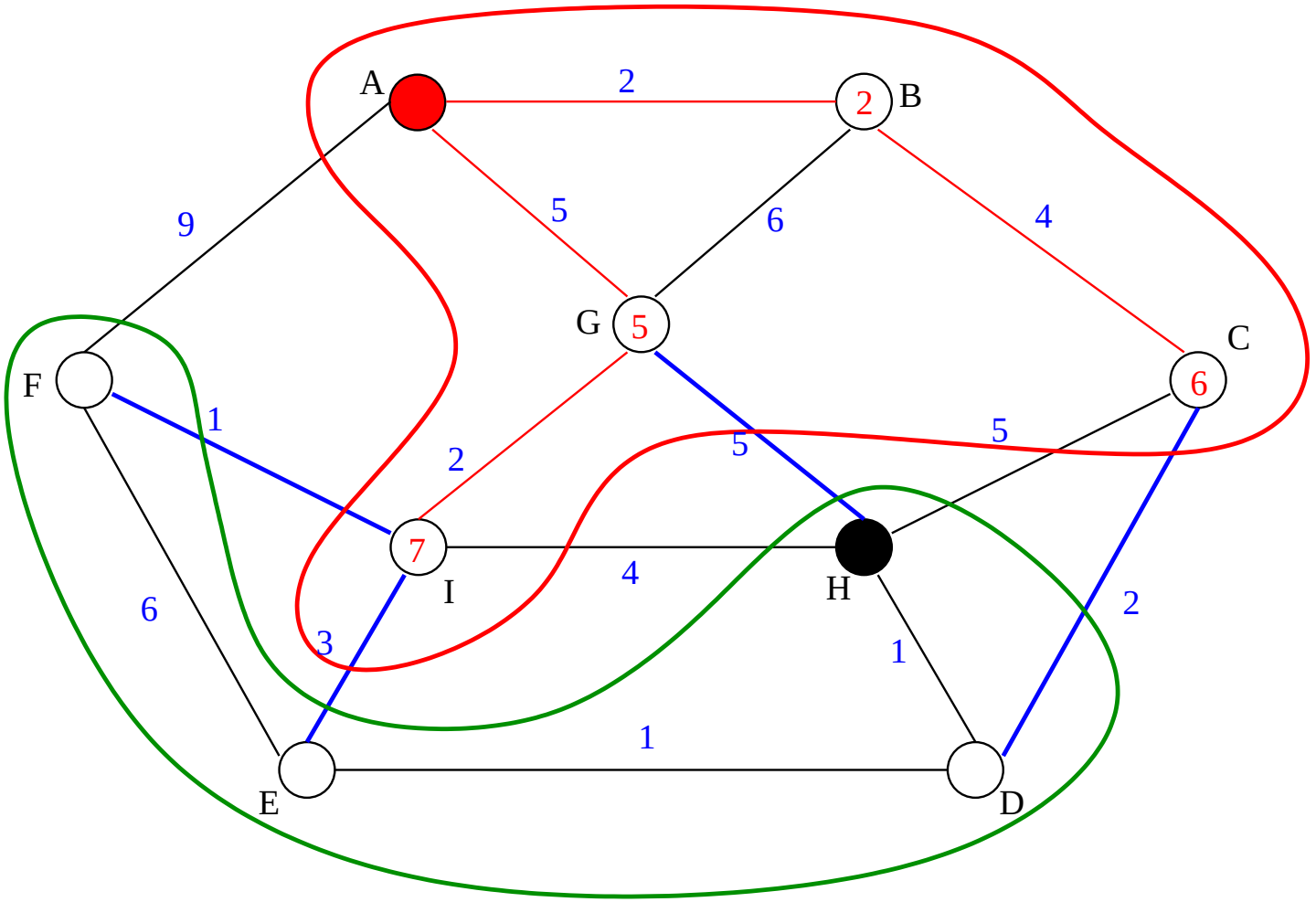
$$d(A,B) + w(B,C) = 6; \quad d(A,A) + w(A,F) = 9;$$
$$d(A,G) + w(G,H) = 10; \quad d(A,G) + w(G,I) = 7.$$

Algoritmo SP de Dijkstra – Exemplo

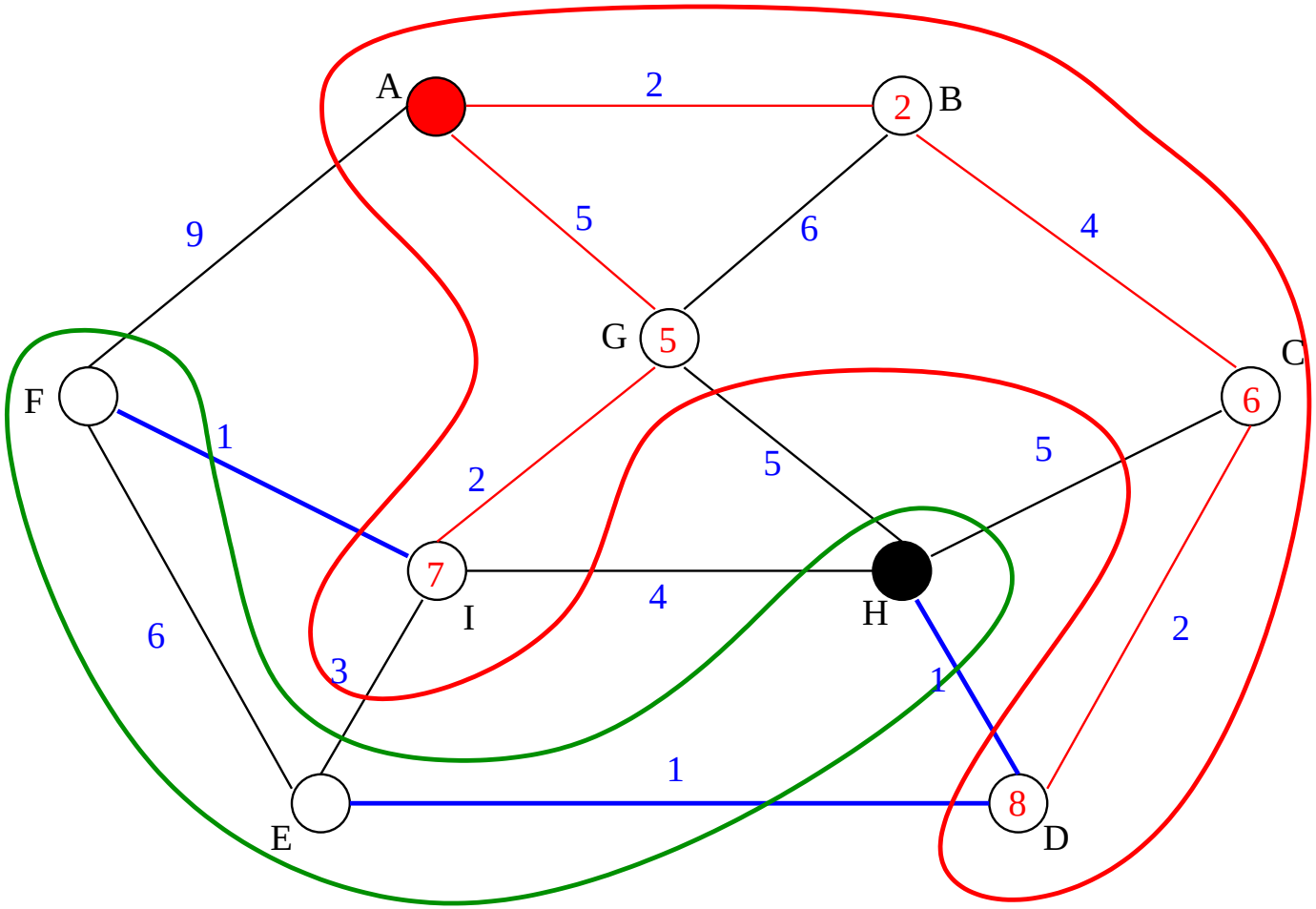


$$d(A,C) + w(C,D) = 8; \quad d(A,A) + w(A,F) = 9;$$
$$d(A,G) + w(G,H) = 10; \quad d(A,G) + w(G,I) = 7.$$

Algoritmo SP de Dijkstra – Exemplo

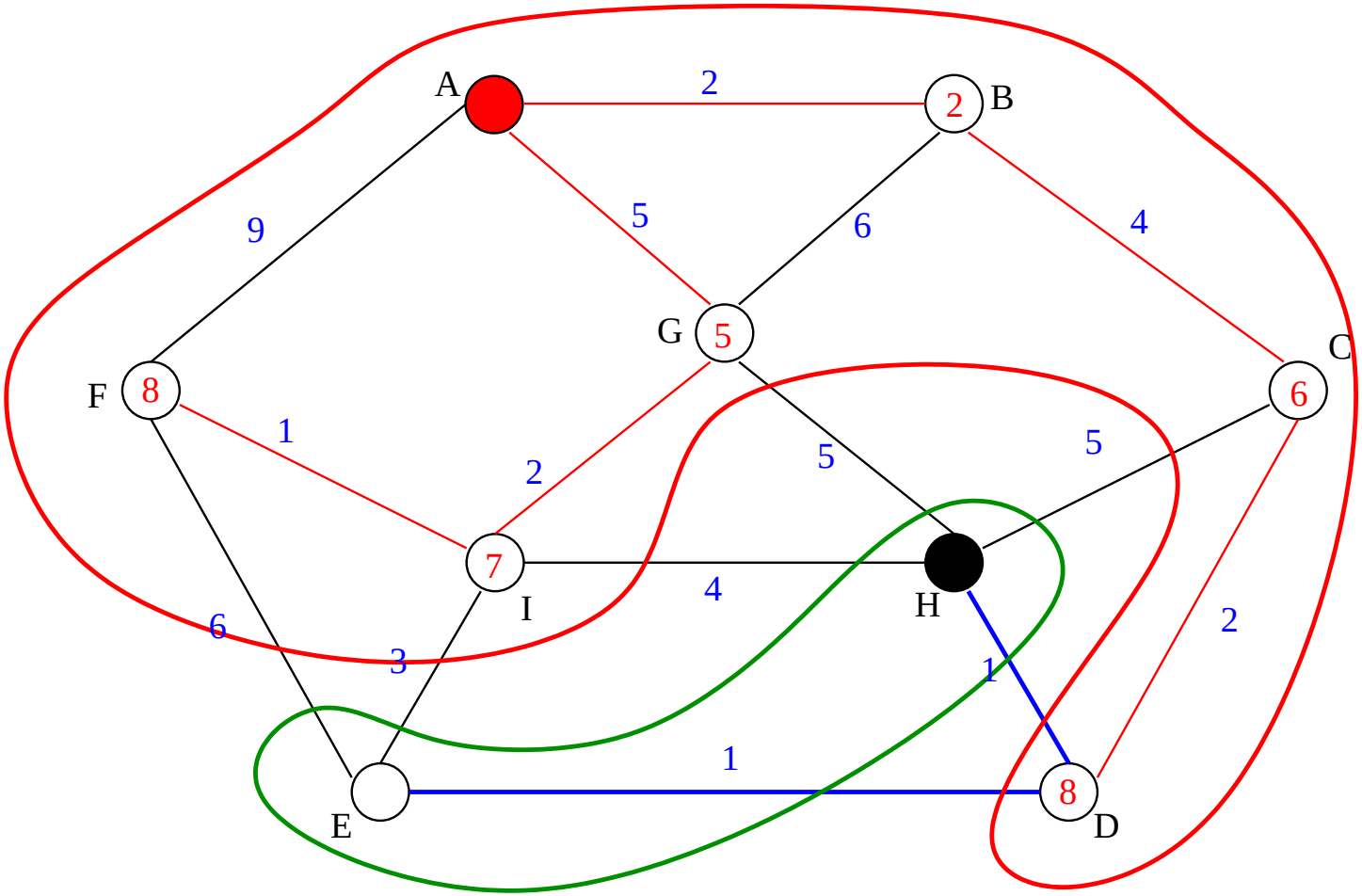


Algoritmo SP de Dijkstra – Exemplo

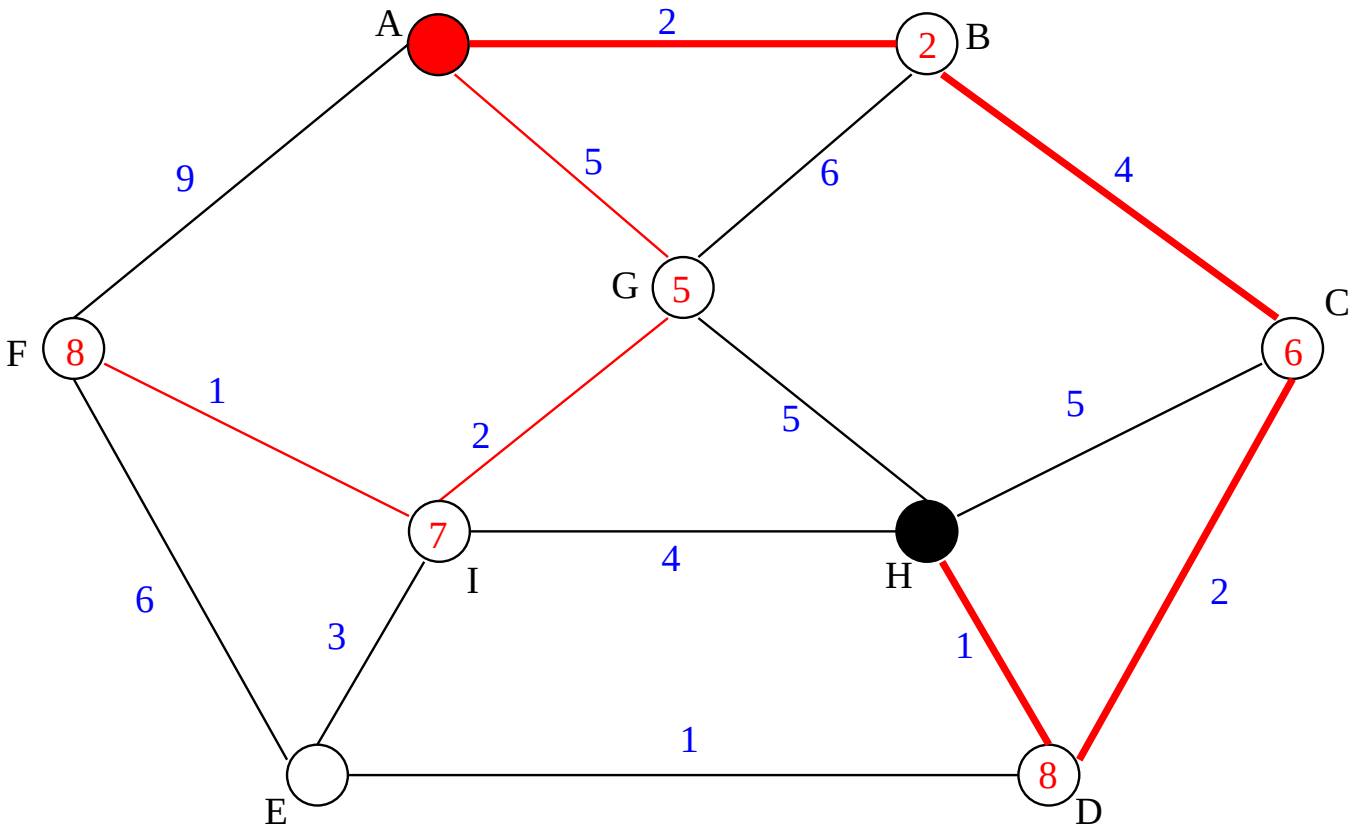


Houve uma alteração do arco candidato de H!

Algoritmo SP de Dijkstra – Exemplo



Algoritmo SP de Dijkstra – Exemplo



Implementação

Utiliza-se um vector $dist[\,]$ para guardar a seguinte informação:

- $dist[y] = d(v, y)$ se y está na árvore construída até ao momento;
- $dist[z] = d(v, y) + w(y, z)$ se z está na orla e (y, z) é o seu arco candidato.

Observe-se que se trata de informação que é reutilizada várias vezes pelo que deve ser armazenada quando é calculada pela primeira vez.

Algoritmo Detalhado

```
void SP((V,E), v, w) {                    /* G = (V,E) */
  x = v; V' = {x}; T' = ∅;
  stuck = 0;
  while (x ≠ w && !stuck) {
    for (y ∈ orla, y adjacente a x)
      if (dist[x] + w(x,y) < dist[y]) {
        substituír arco candidato de y por (x,y);
        dist[y] = dist[x]+w(x,y);
      }
    for (y ∉ V', y ∉ orla, y adjacente a x) {
      colocar y na orla;
      marcar (x,y) arco candidato;
      dist[y] = dist[x]+w(x,y);
    }
```

Algoritmo Detalhado

```
        if (não há arcos candidatos) stuck = 1;
        else { escolher arco candidato (u, z) com dist[z] mínimo;
               x = z;
               V' = V' ∪ {x}; T' = T' ∪ {(u, z)};
               remover x da orla;
               desmarcar (u, z) como candidato;
        }
    }
}
```

Nota: O grafo G é em geral orientado.

Correcção do Algoritmo de Dijkstra

Teorema. *Seja $G = (V, E)$ um grafo pesado e $V' \subseteq V$ contendo o nó v. Se (u, z), com $u \in V'$, $z \notin V'$, é escolhido por forma a minimizar $d(v, u) + w(u, z)$, então o caminho obtido acrescentando-se (u, z) no fim de um caminho mais curto de v para u é um caminho mais curto de v para z.*

Prova. $\Rightarrow$ *Exercício!!*

Correcção do Algoritmo de Dijkstra – Invariante de Ciclo

No início de cada iteração do ciclo `while`, (V', T') é uma árvore com raíz em v tal que todo o caminho de v para y nela contido é um caminho mais curto em G.

Inicialização trivial para $(\{v\}, \emptyset)$

Preservação dada pelo Teorema anterior

Terminação no fim da execução do ciclo, se `stuck != 0`, V' contém w logo contém um caminho mais curto de v para w.

Variantes do Problema dos Caminhos Mais Curtos

Seja G um grafo orientado. O problema estudado designa-se também por "**Single-pair Shortest Paths**" e pode ser visto como um caso particular de:

1. **Single-source Shortest Paths**: determinar todos os caminhos mais curtos com origem num nó v dado e destino em cada nó de G. Poder ser resolvido por uma versão ligeiramente modificada do algoritmo de Dijkstra. $\Rightarrow$ como?

2. **Single-destination Shortest Paths**: determinar todos os caminhos mais curtos com destino num nó w dado e origem em cada nó de G. Pode ser resolvido por um algoritmo de resolução do problema 1, operando sobre um grafo obtido do original invertendo o sentido de todos os arcos.

3. **All-pairs Shortest Paths**: determinar SPs entre todos os pares de nós de G.

Os algoritmos para 1. e 2. são assimptoticamente tão rápidos quanto o algoritmo de Dijkstra. O problema 3. pode ser resolvido de forma mais eficiente do que pela resolução do problema 1 repetidamente.

Estratégias Algorítmicas – Algoritmos "Greedy"

A estratégia "greedy" pode ser utilizada na resolução de problemas de optimização. Um algoritmo greedy efectua uma sequência de escolhas; Em cada ponto de decisão no algoritmo, esta estratégia elege a solução que "parece" melhor:

> *Fazer escolhas localmente óptimas esperando que elas resultem numa solução final globalmente óptima.*

Nota: Esta estratégia não resulta sempre em soluções globalmente óptimas, pelo que tipicamente é necessário *provar* que a estratégia é adequada.

Sub-estrutura Óptima

Diz-se que um problema possui *sub-estrutura óptima* se uma solução óptima para o problema contém *soluções óptimas para sub-problemas* do problema original.

- **Árvores Geradoras Mínimas**: Cada sub-árvore de uma MST do grafo G é uma MST de um sub-grafo de G.

- **Caminhos Mais Curtos**: Todos os sub-caminhos do caminho mais curto de v até w são caminhos mais curtos.

Para que um problema seja resolúvel por uma estratégia "greedy" é condição necessária que ele possua sub-estrutura óptima. . .

Algoritmos Greedy

. . . e que possa ser reformulado como se segue:

- É efectuada uma escolha, da qual resulta um (único) sub-problema que deve ser resolvido.
- Essa escolha é efectuada localmente *sem considerar soluções de sub-problemas* – de facto o novo sub-problema *resulta* da escolha efectuada, e não o contrário.
- A escolha greedy pode então depender das escolhas efectuadas até ao momento, mas nunca de escolhas futuras.
- Trata-se pois de um método *top-down*: cada problema é reduzido a um mais pequeno por uma escolha greedy e assim sucessivamente.
- Isto contrasta com a Programação Dinâmica – uma estratégia *bottom-up* em que cada escolha efectuada depende já de soluções de sub-problemas.

Prova de Correcção Típica

Por exemplo no caso do algoritmo de Prim:

1. O sub-problema actual: extender a árvore já construída (V', T') até conter todos os nós de V.

2. Assume-se uma solução globalmente óptima deste sub-problema: a MST (V, T); (V', T') é sub-árvore desta.

3. A escolha greedy reduz o problema a um mais pequeno extendendo (V', T') com um vértice e um arco; seja (V'', T'') a árvore resultante.

4. Novo problema: extender (V'', T'') até conter todos os nós de V. (V, T) não contém necessariamente (V'', T'').

5. Há então que provar que (V'', T'') é sub-árvore de uma outra solução globalmente óptima $(V, \widehat{T})$, i.e., uma solução para o sub-problema obtido depois da escolha greedy é globalmente óptima.

Fecho Transitivo de um Grafo Não-pesado

Considere-se uma relação binária A sobre um conjunto S ($A \subseteq S \times S$). Escreveremos xAy quando $(x, y) \in A$. Dada uma enumeração $s_1, s_2, \ldots$ de S, a relação A pode ser representada por uma matriz de dimensão $|S|$:

$$a_{ij} = \begin{cases} 1 \text{ se } s_i A s_j \\ 0 \text{ caso contrário} \end{cases}$$

Se A for a *relação de adjacência* de um grafo G, a respectiva matriz é a matriz de adjacências de G. A determinação de elementos do *fecho transitivo* de A:

$$xAy \quad \text{e} \quad yAz \quad \Longrightarrow \quad xAz$$

corresponde à *inserção de arcos em* G:

$$x \longrightarrow y \longrightarrow z \quad \Longrightarrow \quad x \longrightarrow z$$

Algoritmo de Fecho Transitivo de um Grafo

```
void TC (int A[][], int R[][], int n) {
  /* G representado por A, fecho transitivo em R */
  R = A;                          /* copia matriz...  */
  for (i=1 ; i<=n; i++)
    for (j=1 ; j<=n ; j++)
      for (k=1 ; k<=n ; k++)
        if (R[i][k] && R[k][j]) R[i][j] = 1;
}
```

Considere-se a situação seguinte com $i < k'$ e $j < k$:

$$s_i \longrightarrow s_{k'} \longrightarrow s_k \longrightarrow s_j$$

Então o algoritmo tenta acrescentar o arco (s_i, s_j) antes de (s_i, s_k) e $(s_{k'}, s_j)$.

Este algoritmo é incorrecto porque não processa os vértices pela ordem adequada.

Algoritmo de Warshall

Uma correcção possível do algoritmo consiste na introdução de um novo ciclo (mais exterior), `while`(houver arcos a acrescentar...).

Uma solução mais eficiente é o Algoritmo de Warshall, que difere do anterior apenas na disposição dos ciclos:

```
void WarshallTC (int A[][], int R[][], int n) {
  /* G representado por A, fecho transitivo em R */
  R = A;                        /* copia matriz...  */
  for (k=1 ; k<=n; k++)
    for (i=1 ; i<=n ; i++)
      for (j=1 ; j<=n ; j++)
        if (R[i][k] && R[k][j]) R[i][j] = 1;
}
```

Este algoritmo executa em tempo $\theta(|V|^3)$.

Correcção do Algoritmo de Warshall – Invariante de Ciclo

Seja $S_k = \{s_1, \ldots, s_k\}$, para $k \leq |V|$. No *fim* da execução da iteração de índice k do ciclo `for` mais exterior, `R[i][j]==1` sse existe um caminho (de comprimento > 0) de s_i para s_j contendo além destes apenas nós de S_k.

Inicialização $k = 0$: o ciclo `for` não foi ainda executado, `R` contém apenas os arcos iniciais do grafo.

Preservação Assuma-se válido o invariante no fim da iteração $k-1$; Para prová-lo válido no fim da iteração k, há que estudar quais os pares (i, j) para os quais `R[i][j]` se vai tornar 1, tendo em conta os vários casos possíveis ($k \geq j$ ou $k < j$; $k \geq i$ ou $k > i$).

Terminação No fim da última execução do ciclo, tem-se `R[i][j]==1` sse existe um caminho (de comprimento > 0) de s_i para s_j em G.

"All-pairs Shortest Paths"

Uma generalização do algoritmo de Warshall para grafos pesados permite calcular uma matriz contendo os pesos dos caminhos mais curtos entre todos os pares de vértices de um grafo.

```
void Distances (Weight W[][], Weight D[][], int n) {
  /* G representado por A, fecho transitivo em R */
  D = W;                        /* copia matriz...  */
  for (k=1 ; k<=n; k++)
    for (i=1 ; i<=n ; i++)
      for (j=1 ; j<=n ; j++)
        D[i][j] = min(D[i][j],D[i][k]+D[k][j]);
}
```

Trata-se de facto de um algoritmo de Programação Dinâmica.