CT-234

Estruturas de Dados, Análise de Algoritmos e Complexidade Estrutural

Carlos Alberto Alonso Sanches

CT-234

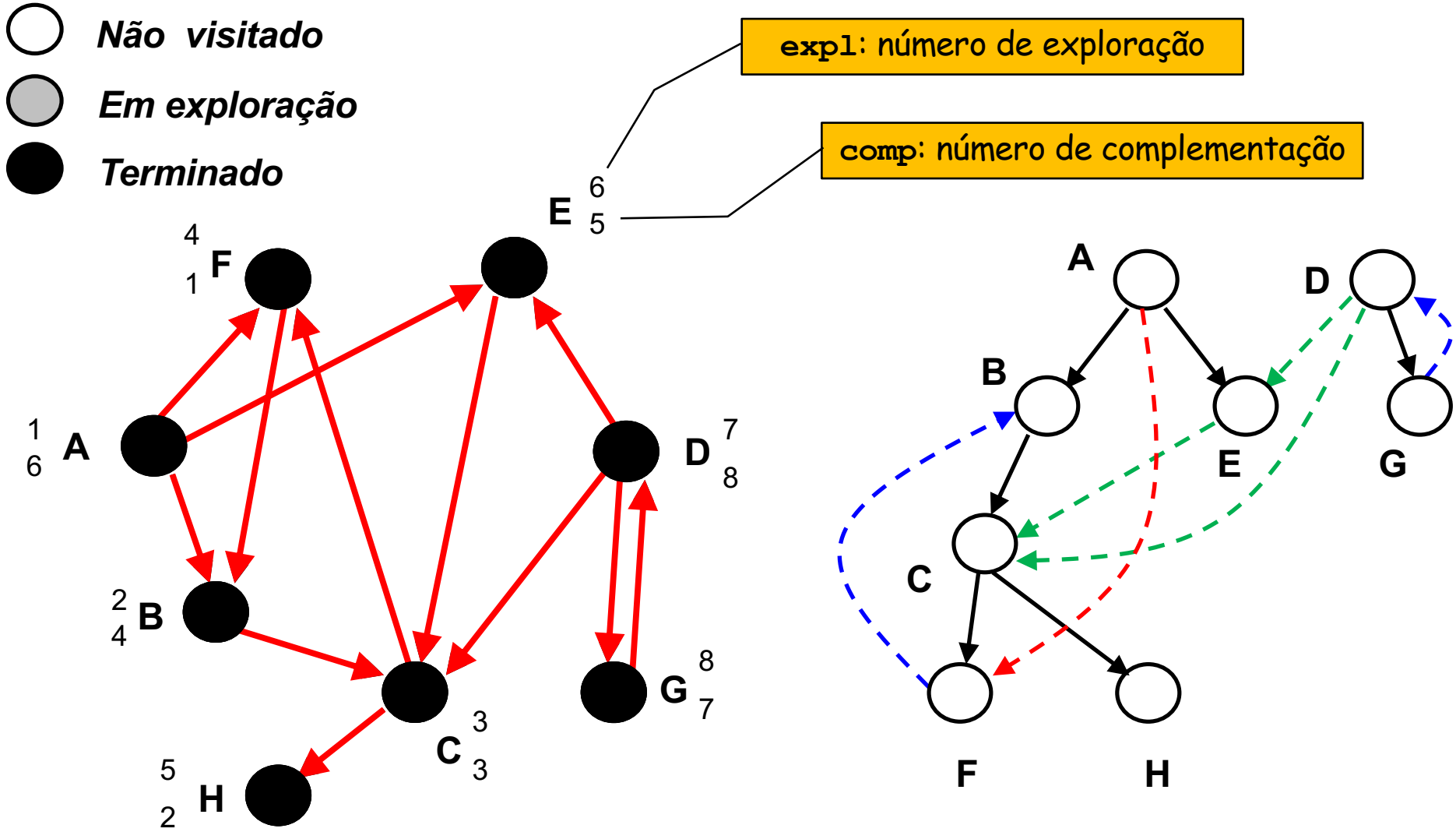
9) Algoritmos em grafos

Tarjan, Dijkstra, Kruskal, Prim

Ideia de Tarjan (1972)

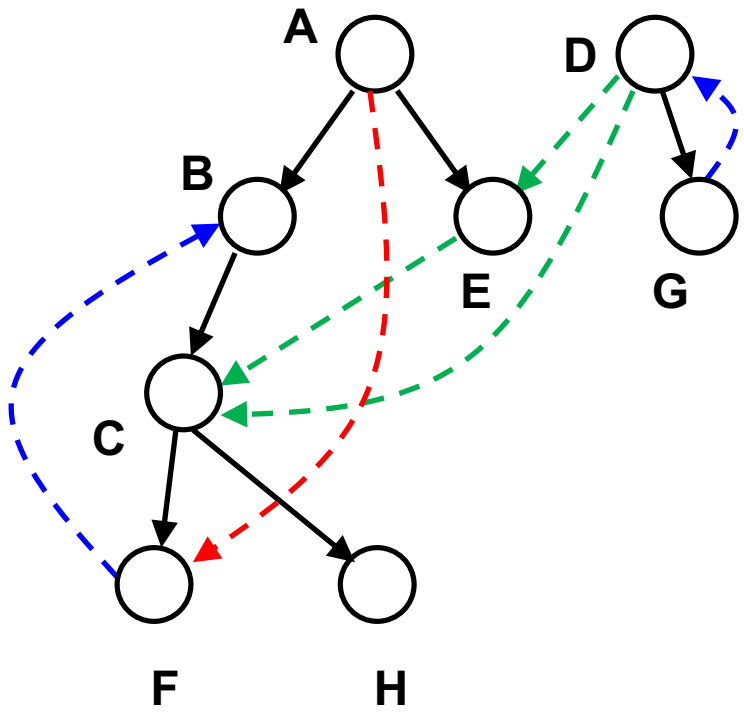
- Durante a *exploração em profundidade* de um digrafo, numerar seus vértices de acordo com o início e o término dessa exploração.
- As diferentes situações permitem estabelecer uma classificação dos arcos.

Exemplo


Não visitado
Em exploração
Terminado
expl: número de exploração
comp: número de complementação
E 6 5
F 4 1
A 1 6
B 2 4
C 3 3
D 7 8
G 8 7
H 5 2
A
B
C
D
E
F
G
H

Classificação dos arcos

- Classificação do arco <v,u>:
  - Árvore (T): u ainda não havia sido explorado, e será filho de v em T (`expl[u]=0`)
  - Retorno (B): u é antecessor de v em T, pois começou antes de v e ainda está em exploração (`expl[u]<expl[v]` e `comp[u]=0`)
  - Cruzamento (C): u está em outra árvore ou sub-árvore, pois começou antes de v e já foi explorado (`expl[u]<expl[v]` e `comp[u]>0`)
  - Avanço (F): u é descendente de v em T, pois começou depois de v e já foi explorado (`expl[u]>expl[v]` e `comp[u]>0`)



Algoritmo de Tarjan

```
Tarjan(G) {
  int ce = 0;
  int cc = 0;
  for v ∈ V {
      expl[v] = 0;
      comp[v] = 0;
  }
  for v ∈ V
      if (expl[v] == 0)
         DFST(v);
}
```

```
DFST(v) {
    expl[v] = ++ce;
    for <v,u> ∈ E
       if (expl[u] == 0) {
              tipo[<v,u>] = T;
              DFST(u);
       }
       else if (expl[u] > expl[v])
                tipo[<v,u>] = F;
            else if (comp[u] > 0)
                     tipo[<v,u>] = C;
                 else tipo[<v,u>] = B;
    comp[v] = ++cc;
}
```

Complexidade de tempo: $\Theta(n+m)$

Exercício

- Considere um *grafo não orientado, sem laços e sem arestas repetidas*. Se aplicarmos nele o algoritmo de Tarjan, somente haverá arestas de <u>árvore</u> e de <u>retorno</u>.
- Por que neste caso não existem arestas de <u>avanço</u> e de <u>cruzamento</u>?

Teste de aciclicidade

- Em certas aplicações, como a ordenação topológica, uma tarefa importante é o reconhecimento de um *digrafo acíclico* (conhecido como DAG).
- A exploração em profundidade pode nos dar a solução desse problema em tempo O(n+m).
- Concretamente, basta uma variação do algoritmo de Tarjan: se um arco de retorno for encontrado durante a exploração, então o digrafo será cíclico.

Algoritmo

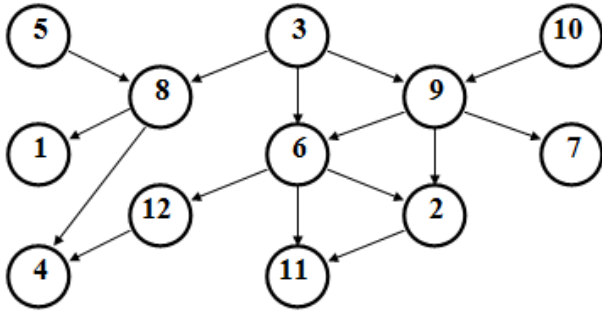
```
Aciclicidade(G) {
    stack P;
    bool aciclico = true;
    int ce = 0;
    int cc = 0;
    for v ∈ V {
         expl[v] = 0;
         comp[v] = 0;
    }
    for v ∈ V
      if (expl[v] == 0)
          DFSA(v);
    if (aciclico)
        digrafo é acíclico
    else
        digrafo é cíclico
}
```

```
DFSA(v) {
    expl[v] = ++ce;
    push(P,v);
    for <v,u> ∈ E
      if (expl[u] == 0)
          DFSA(u);
      else
          if (expl[u]<expl[v]
                  && comp[u]==0) {
              aciclico = false;
              // ciclo está em P
              // desde o topo até u
              stop;
          }
    pop(P);
    comp[v] = ++cc;
}
```

Arco de retorno

Ordenação topológica

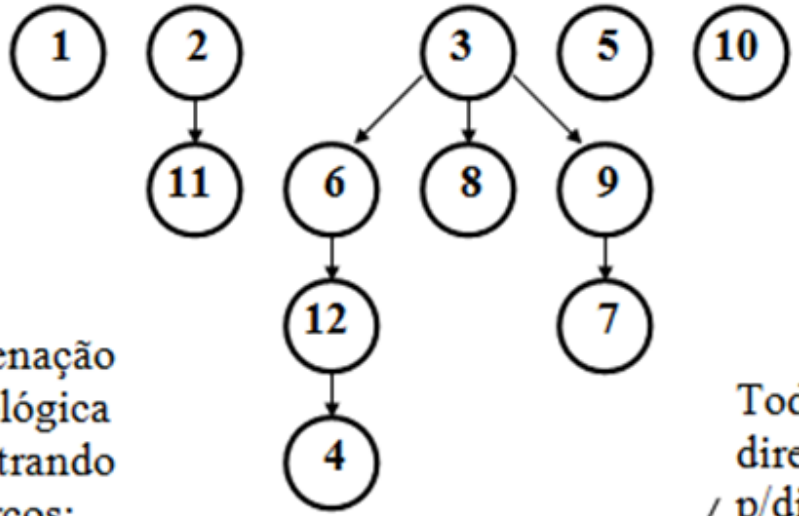
- Considere o DAG abaixo:



- Vamos fazer sua exploração em profundidade dando prioridade aos vértices de menor valor.

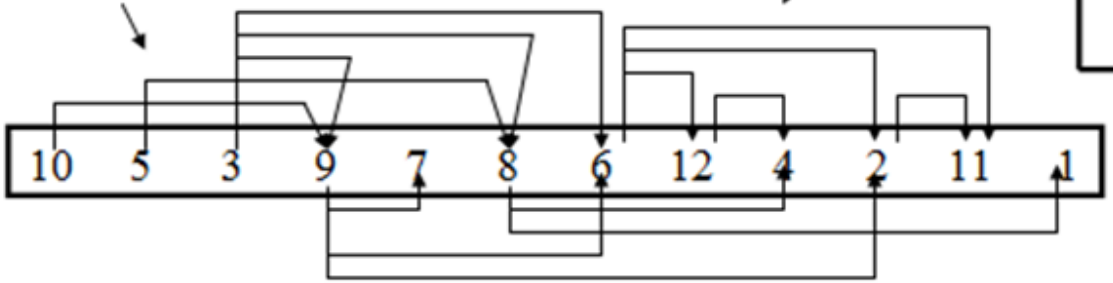
Sequência de término de exploração

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



Ordenação topológica mostrando os arcos:

Todas as direções p/direita



Algoritmo

```
OrdemTopol(G) {
   // supõe digrafo acíclico
   int ce = 0;
   int cc = 0;
   for v ∈ V {
      expl[v] = 0;
      comp[v] = 0;
   }
   for v ∈ V
      if (expl[v] == 0)
         DFSOT(v);
   for v ∈ V
      f[v] = n-comp[v]+1;
}
```

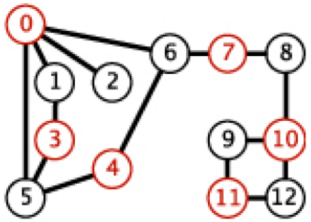
```
DFSOT(v) {
   expl[v] = ++ce;
   for <v,u> ∈ E
      if (expl[u] == 0)
         DFSOT(u);
   comp[v] = ++cc;
}
```

Usando uma pilha, seria possível imprimir os vértices já na ordem topológica. Como?

Complexidade de tempo: $\Theta(n+m)$

Bipartição (ou bicoloração) de vértices

- Vimos que um grafo G é bipartido (ou bicolorido) quando existe uma bipartição de seus vértices em subconjuntos disjuntos V_1 e V_2 tais que qualquer aresta de G possui uma extremidade em V_1 e outra em V_2.
- Exemplo:



$V_1 = \{0, 3, 4, 7, 10, 11\}$

$V_2 = \{1, 2, 5, 6, 8, 9, 12\}$

- É possível demonstrar que um grafo admite bipartição se e somente se não tiver ciclos de tamanho ímpar.
- Uma simples variação no algoritmo de exploração em profundidade permite encontrar uma bipartição de um grafo, se existir.
- O algoritmo a seguir produz uma bipartição (atribui "1" ou "2" ao número de exploração de cada vértice) ou informa que o grafo não pode ser bipartido.

Algoritmo

Ambos pseudocódigos retornam 0 se não existir bipartição

```
TarjanBP(G) {
   for v ∈ V
      expl[v] = 0;
   for v ∈ V
      if (expl[v] == 0)
         if (DFSBP(v,2) == 0)
            return 0;
   return 1;
}
```

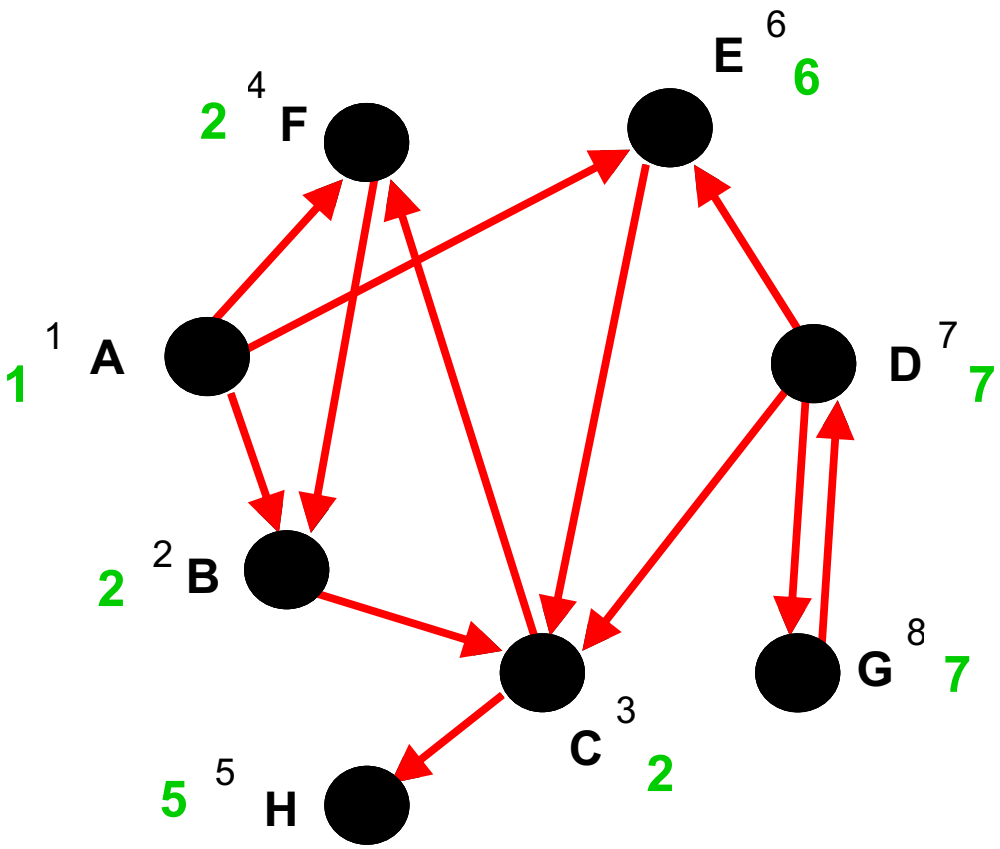
```
DFSBP(v,color) {
    c = (color % 2) + 1;
    expl[v] = c; // troca cor
    for <v,u> ∈ E
       if (expl[u] == 0) {
          if (DFSBP(u,c) == 0)
             return 0; }
       else if (expl[u] == c)
             return 0;
    return 1;
}
```

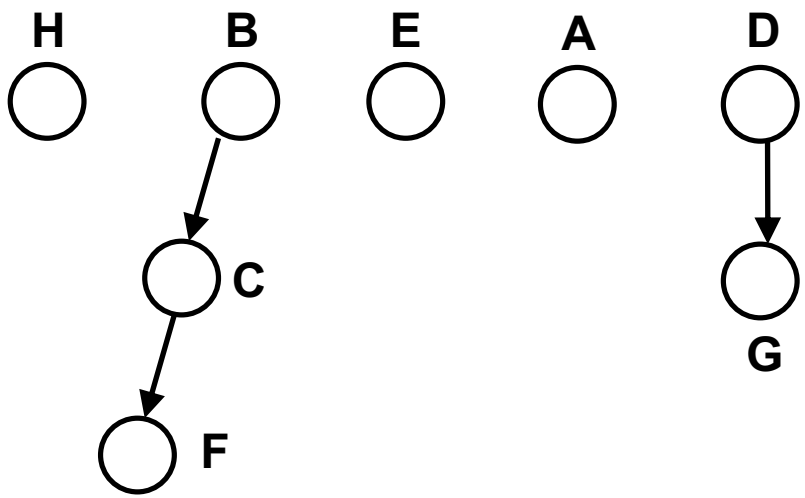
Complexidade de tempo: $\Theta(n+m)$

Componentes fortemente conexas (SCC)

- É possível encontrar as componentes fortemente conexas de um digrafo através de uma variante do algoritmo de Tarjan.
- Ideia:
  - Considere a árvore T de exploração em profundidade e a numeração expl[v] para cada $v \in V$.
  - Os vértices que estão em exploração são empilhados (permanecerão nessa pilha até que seja encontrada a sua componente conexa).
  - Cada vértice v guardará CFC[v], que é o menor número de exploração entre os vértices na pilha que atingir durante sua exploração. Desse modo, ficará automaticamente associado a uma componente conexa.
  - Quando a exploração do vértice v terminar, se expl[v] = CFC[v] então todos os vértices na pilha (desde o topo até v) pertencem a uma mesma componente, e podem ser desempilhados.

Exemplo





- <u>Importante:</u> nem todos os vértices de uma mesma componente terminam com o mesmo valor de CFC.
- Exemplo: acrescente um arco <C,A> nesse mesmo digrafo, e visite <C,F> antes.

Algoritmo

```
TarjanCFC(G) {
  stack P;
  int ce = 0;
  for v ∈ V
     expl[v] = 0;
  for v ∈ V
     if (expl[v] == 0)
         DFSCFC(v);
}
```

```
DFSCFC(v) {
    expl[v] = ++ce;
    push(P,v);
    CFC[v] = expl[v];
    for <v,u> ∈ E
       if (expl[u] == 0) {
          DFSCFC(u);
          CFC[v] = min{CFC[v],CFC[u]};
       }
       else if (u ∈ P)
                CFC[v] = min{CFC[v],expl[u]};
    if (CFC[v] == expl[v])
     do {
        x = top(P);
        pop(P);
     } while (x != v);
}
```

Arco de árvore

Teste em tempo constante com vetor de *flags*

Complexidade de tempo: $\Theta(n+m)$

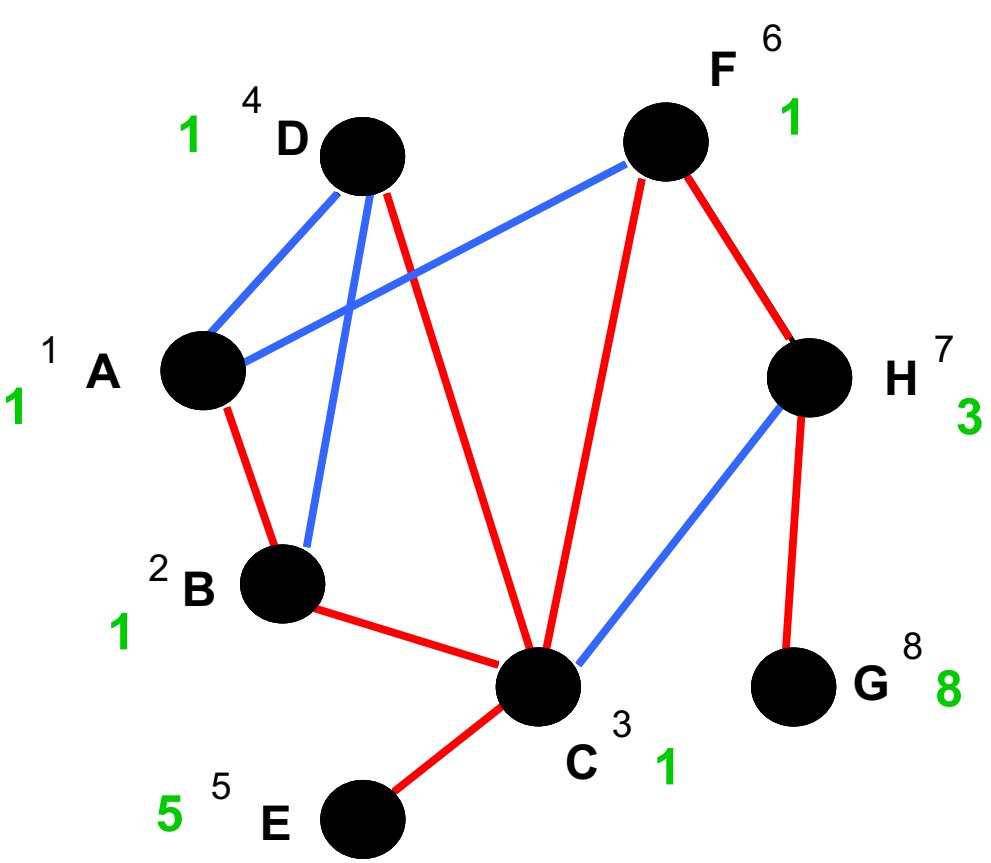
Vértices de corte

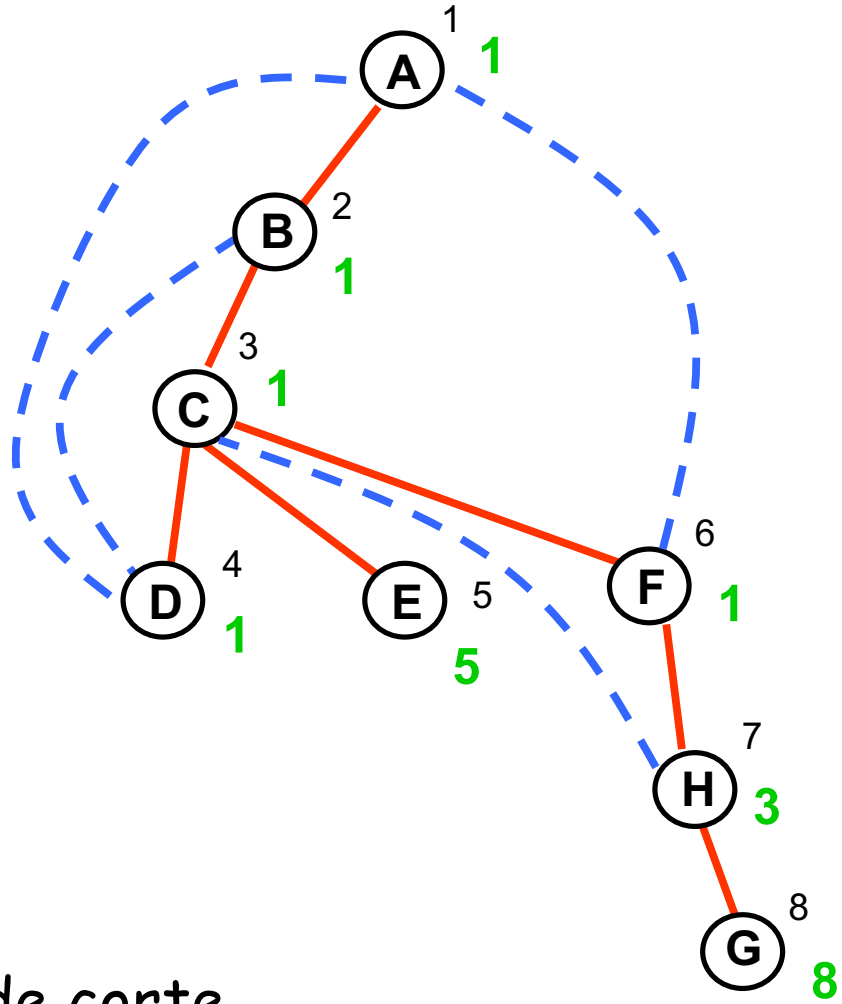
- Uma variante do algoritmo de Tarjan pode encontrar os vértices de corte (ou pontos de articulação) de um grafo G=(V,E) *conexo não-orientado, sem laços ou arestas repetidas*.

Desse modo, se fizéssemos uma exploração em profundidade a partir de cada vértice do grafo, poderíamos identificar todos os vértices de corte (no entanto, há outra solução mais eficiente)

- Ideia:
  - Considere a árvore T de exploração em profundidade e a numeração expl[v] para cada $v \in V$.
  - **Raiz:** será vértice de corte se tiver pelo menos dois filhos em T.
  - **Demais vértices**:
    - v será vértice de corte se tiver algum filho sem retorno para nenhum dos ancestrais de v.
    - É calculado m[v] = mín{ expl[v], expl[x] }, onde x é um vértice que v (ou um de seus descendentes) atinge em T através de uma *única aresta de retorno*.
    - Portanto, v será vértice de corte se tiver algum filho u tal que $m[u] \geq expl[v]$.

Exemplo





C e H são vértices de corte

Algoritmo

```
TarjanVC(r) {
  // válido se for conexo e
  // não tiver laços ou
  // arestas repetidas
  int ce = 0;
  for v ∈ V {
     expl[v] = 0;
     pai[v] = null;
     nfilhos[v] = 0;
     VC[v] = false;
  }
  DFSVC(r);
  for v ∈ V-{r} {
     p = pai[v];
     VC[p] = VC[p] || (m[v]>=expl[p]);
  }
  VC[r] = (nfilhos[r] > 1);
  for v ∈ V
     if (VC[v]) v é vértice de corte;
}
```

```
DFSVC(v) {
  expl[v] = ++ce;
  m[v] = expl[v];
  for <v,u> ∈ E
    if (expl[u] == 0) {
      pai[u] = v;
      nfilhos[v]++;
      DFSVC(u);
      m[v] = min{m[v],m[u]};
    }
    else // arestas de retorno
     if(u != pai[v])
      m[v] = min{m[v],expl[u]};
}
```

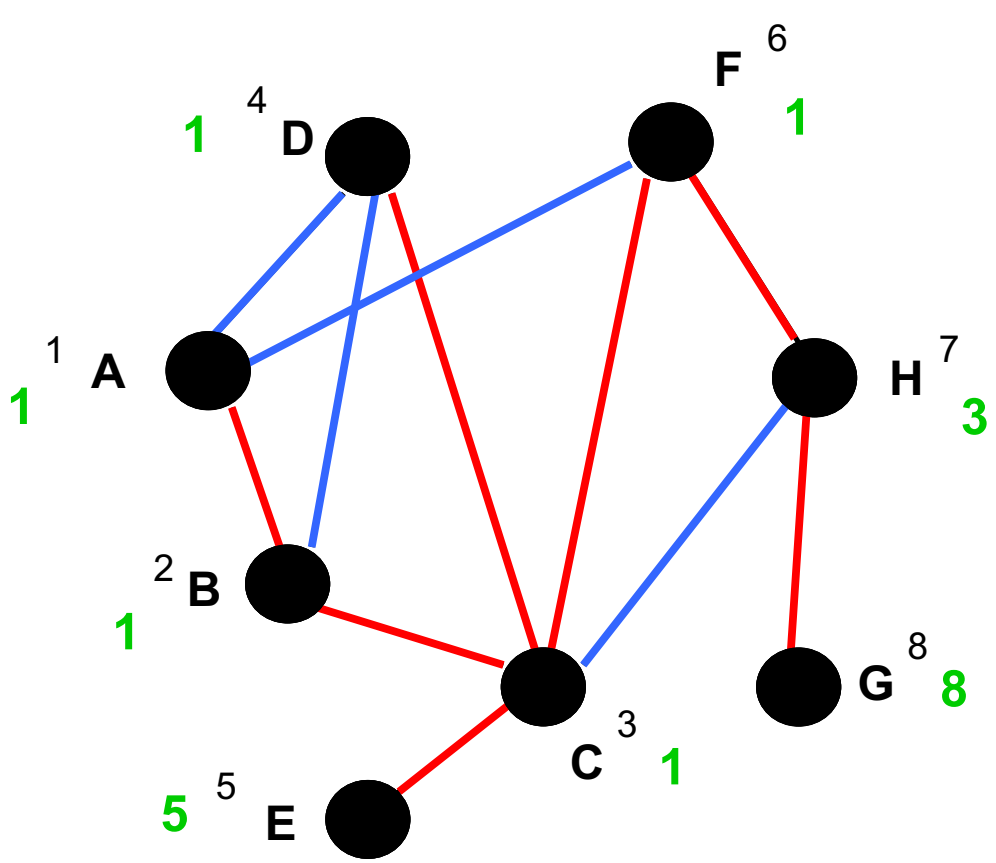
Trata as arestas de retorno, tanto na ida como na volta

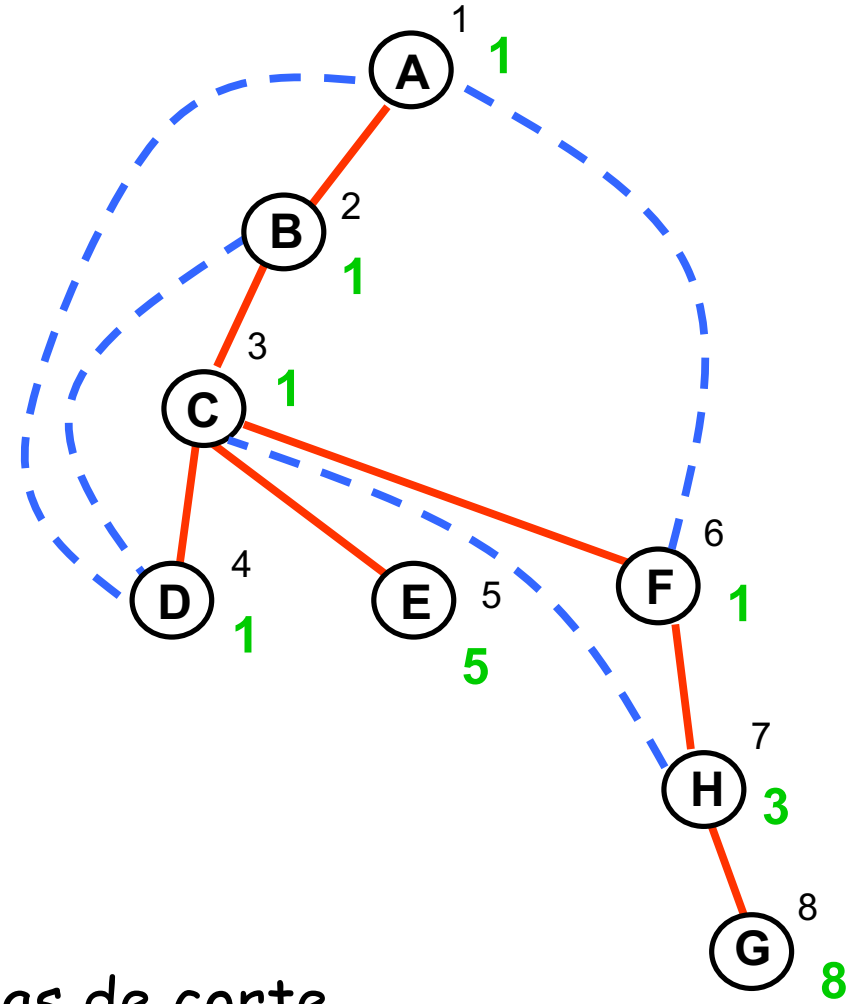
Complexidade de tempo: $\Theta(n+m)$

Arestas de corte

- A identificação das arestas de corte (ou pontes) é realizada de maneira semelhante:
  - Encontrar uma árvore de exploração T, calculando as mesmas numerações expl e m para os vértices.
  - É fácil constatar que nenhuma aresta de retorno dessa exploração pode ser de corte.
  - Uma aresta <v,u> ϵ T será de corte se m[u] = expl[u].

No exemplo anterior





<C,E> e <H,G> são arestas de corte

Algoritmo

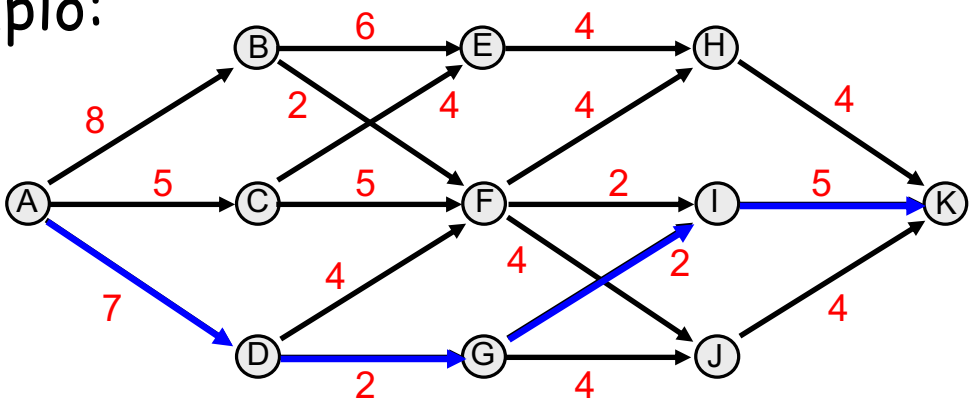
```
TarjanAC(r) {
  // válido se for conexo e
  // não tiver laços ou
  // arestas repetidas
  int ce = 0;
  for v ∈ V {
     expl[v] = 0;
     pai[v] = null;
  }
  DFSAC(r);
}
```

```
DFSAC(v) {
  expl[v] = ++ce;
  m[v] = expl[v];
  for <v,u> ∈ E
    if (expl[u] == 0) {
      pai[u] = v;
      DFSAC(u);
      m[v] = min{m[v],m[u]};
      if (m[u] == expl[u])
           <v,u> é aresta de corte;
    }
    else // arestas de retorno
     if(u != pai[v])
       m[v] = min{m[v],expl[u]};
}
```

Complexidade de tempo: $\Theta(n+m)$

Caminhos mais curtos

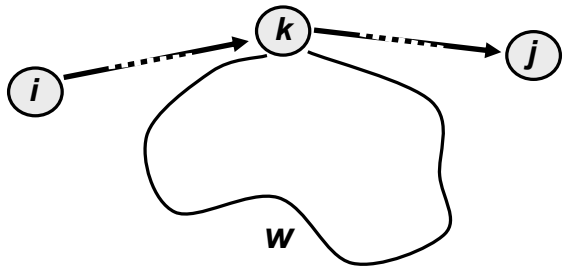
- Um digrafo (ou grafo) $G=(V,E)$, onde $V = \{v_1, v_2, ..., v_n\}$, é chamado de <u>ponderado</u> se cada arco (ou aresta) $(v_i, v_j) \in E$ tem custo c_{ij}.
- Distância entre dois vértices é a somatória dos custos dos arcos (ou arestas) de um caminho que os une.
- Um problema clássico é encontrar o caminho mais curto ou a distância mínima entre dois vértices.
- Exemplo:



Distância mínima entre A e K:
7+2+2+5 = 16

Uma condição de existência

- Considere o caminho abaixo entre *i* e *j*, e o ciclo *w* :



- Se o comprimento de *w* for negativo, qual seria a distância mínima entre *i* e *j* ?
- Uma condição de existência para o caminho mais curto é que seja elementar, isto é, sem ciclos em seu interior.

Arcos (ou arestas) de mesmo peso

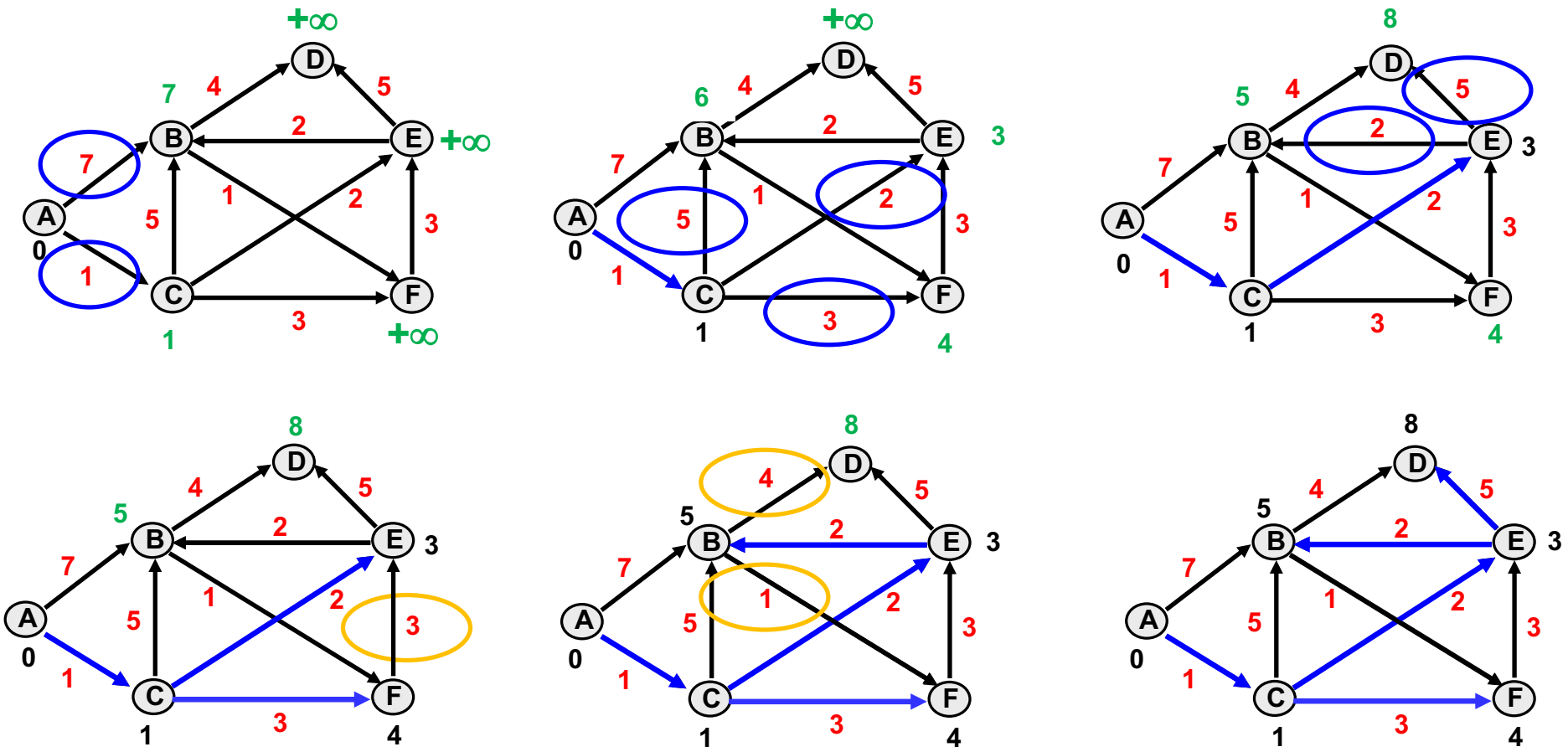
- Quando todas os arcos (ou arestas) têm pesos iguais, basta uma simples alteração na *exploração em largura*.
- Afinal, os vértices vão sendo enfileirados seguindo a ordem de proximidade: portanto, basta incrementar a distância do vértice antecessor.
- O algoritmo de Dijkstra generaliza essa ideia.

```
BFSMinCam(r) {
    queue q;
    int ce = 0;
    for v ∈ V - {r} {
        d[v] = +∞;
        expl[v] = 0;
    }
    expl[r] = ++ce;
    d[r] = 0;
    enqueue(q,r);
    while (!isEmpty(q)) {
        u = dequeue(q);
        for <u,v> ∈ E {
            if (expl[v] == 0) {
                expl[v] = ++ce;
                d[v] = d[u] + 1;
                enqueue(q,v);
            }
        }
    }
}
```

Exemplo do algoritmo de Dijkstra

Pesos dos arcos, distâncias provisórias, distâncias definitivas

Os arcos indicados são os últimos que atualizaram a distância



Algoritmo de Dijkstra (1959)

```
Dijkstra(u) {
    d[u] = 0;          // vértice inicial u: distância nula
    for v ∈ V - {u}
        d[v] = +∞; // demais vértices: distância +∞
    S ← V;             // S: vértices com distância provisória
    while S ≠ ∅ {
        selecionar j tal que d[j] == min_{i∈S}{d[i]};
        S ← S - {j};  // j passa a ter distância definitiva
        for <j,w> ∈ E, onde w ∈ S
            if (d[w] > d[j] + c_jw) {
                d[w] = d[j] + c_jw;
                pred[w] = j;
            }
}
```

Predecessor: permite reconstruir o caminho mínimo entre u e w

Assemelha-se a uma exploração em largura

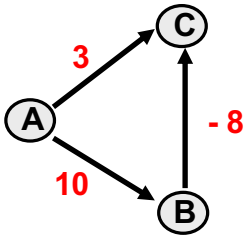
Encontra todos os caminhos mínimos a partir do vértice u

Complexidade de tempo

- *Grosso modo*, o algoritmo de Dijkstra gasta tempo de pior caso $\Theta(n^2+m) = \Theta(n^2)$.
- No entanto é possível melhorar essa complexidade de tempo se o conjunto S for implementado com um *heap* de mínimo.
- Nesse caso, passaria a ser $\Theta((n+m).\log n)$:
  - O *heap* possuirá inicialmente n elementos.
  - No total, são realizadas n extrações de mínimo e até m modificações de valor (será preciso manter um vetor auxiliar que armazena a posição corrente que cada vértice ocupa no *heap*).

Arcos (ou arestas) com custos negativos

- No digrafo abaixo, qual a distância mínima entre os vértices A e C?



- O algoritmo de Dijkstra daria como resposta 3, mas o valor correto é 2... Por que isso acontece?
- No processo de construção do caminho mínimo, o algoritmo de Dijkstra *supõe que o custo acumulado sempre cresce*. Isso não ocorre se admitirmos arcos (ou arestas) com custos negativos...
- Em um algoritmo mais geral, cada vez que se altera a distância até um vértice, também deve ser recalculada a distância até todos os seus adjacentes.

Algoritmo de Dijkstra modificado

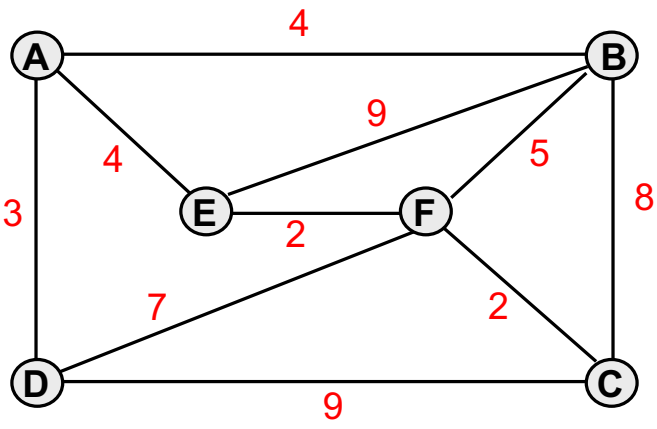
```
Dijkstra2(u) {
    for v ∈ V - {u}
        d[v] = +∞;
    d[u] = 0;
    S ← {u};
    while S ≠ ∅ {
        selecionar j ∈ S;
        S ← S - {j};
        for <j,w> ∈ E
            if (d[w] > d[j] + c_jw) {
                d[w] = d[j] + c_jw;
                pred[w] = j;
                S ← S ∪ {w};    // w volta para S
            }
}
```

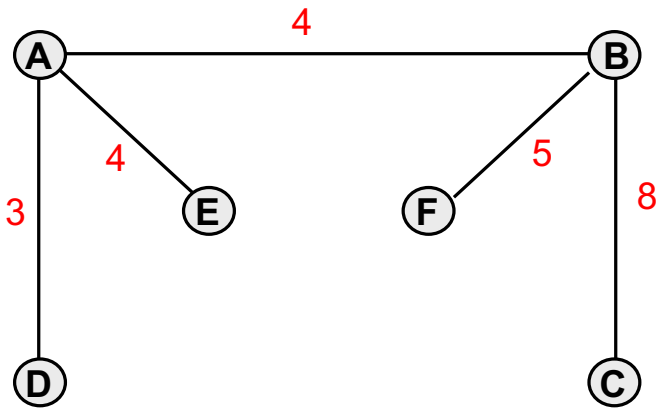
- Com uma estrutura adequada para S, a complexidade de tempo desse algoritmo pode ser o(n.m).

Árvore geradora de custo mínimo (MST)

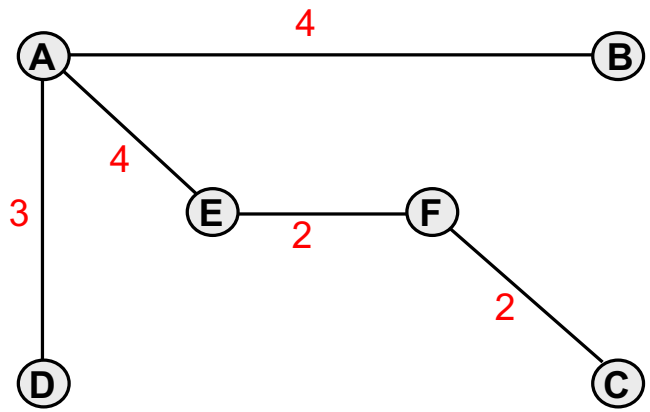
- Dado um grafo $G=(V,E)$, conexo e ponderado, com custo associado $c(e)$, $e \in E$, deseja-se encontrar um subgrafo T tal que:
  - seja gerador de G (isto é, possua todos os vértices);
  - seja acíclico e conexo (isto é, uma árvore);
  - tenha custo total $c(T) = \Sigma_{e \in E} c(e)$ que seja mínimo.
- T costuma ser chamado de *árvore de espalhamento de custo mínimo.*
- Em um grafo conexo, sabemos que $m \geq n-1$.

Exemplo





Árvore geradora
com custo 24

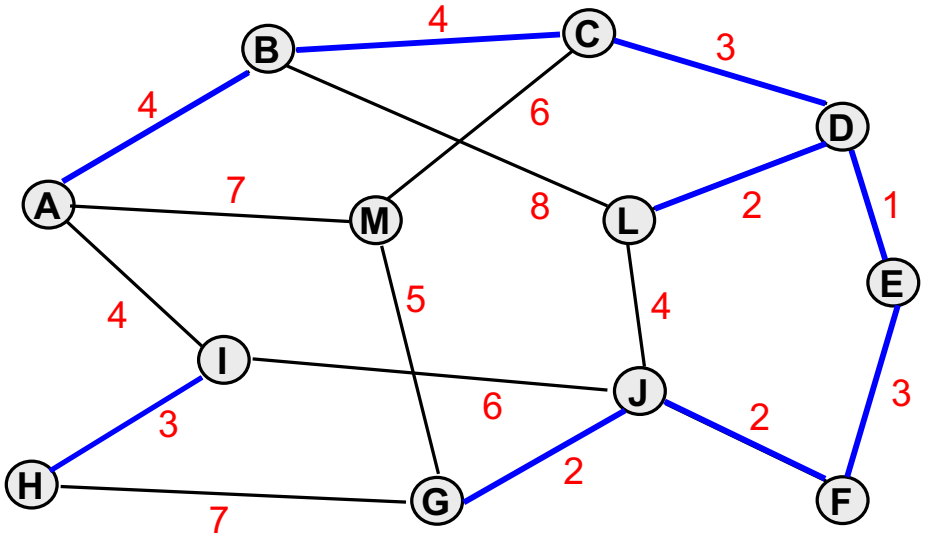


Árvore geradora
com custo 15

Ideia de Kruskal (1956)

- Princípio: a aresta de menor custo sempre pertence à árvore geradora de custo mínimo.
- Demonstração:
  - Suponha, por absurdo, que a aresta de custo mínimo não esteja na solução ótima.
  - A inserção desta aresta na solução ótima gera um ciclo.
  - Removendo-se a aresta de maior custo neste ciclo (que não é a inserida), obtém-se uma nova árvore geradora.
  - Essa nova árvore tem um custo inferior à solução ótima inicial: contradição.

Exemplo



c(T) = 24

Componentes

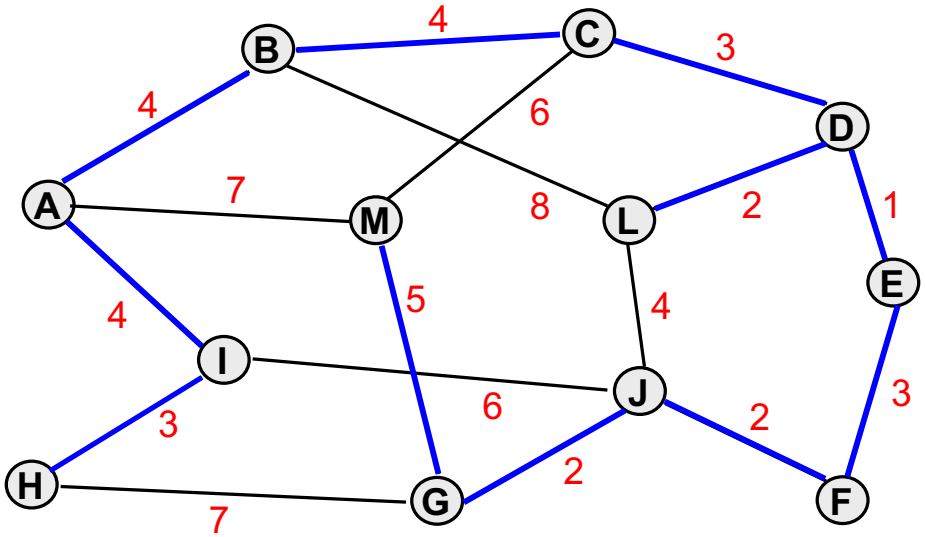
{ A, B, C, D, E, F, G, J, L }

{ H, I } { M }

Lista ordenada

e	c(e)
(D,E)	1
(D,L)	2
(F,J)	2
(G,J)	2
(C,D)	3
(E,F)	3
(H,I)	3
(A,B)	4
(B,C)	4
...	...

Exemplo



c(T) = 33

Componentes

{ A, B, C, D, E, F, G, H, I, J, L, M }

Lista ordenada

e	c(e)
...	...
(A,I)	4
(J,L)	4
(G,M)	5
(C,M)	6
(I,J)	6
(A,M)	7
(G,H)	7
(B,L)	8

Algoritmo de Kruskal

```
Kruskal(G) {
    T ← ∅;
    A ← vetor com as arestas em ordem crescente de custo;
    criar n componentes com os n vértices isolados;
    for (i=1; i<=m && |T|<n-1; i++) {
        <u,v> = A[i];
        if (u e v não estão na mesma componente) {
            T ← T ∪ {<u,v>};
            unir as componentes que contêm u e v;
        }
    }
}
```

- Operações executadas:
  - Ordenação de m valores
  - 2m testes de componentes
  - n-1 uniões de componentes

Complexidade de tempo

- Implementação simples: utilizar um vetor de tamanho n que indica a componente de cada vértice.
- Tempo de pior caso:
  - Ordenação dos custos: $\Theta(m.\log m)$
  - 2m testes de componentes: $\Theta(m)$
  - n-1 uniões de componentes: $\Theta(n^2)$
- Total: $\Theta(m.\log m + n^2)$

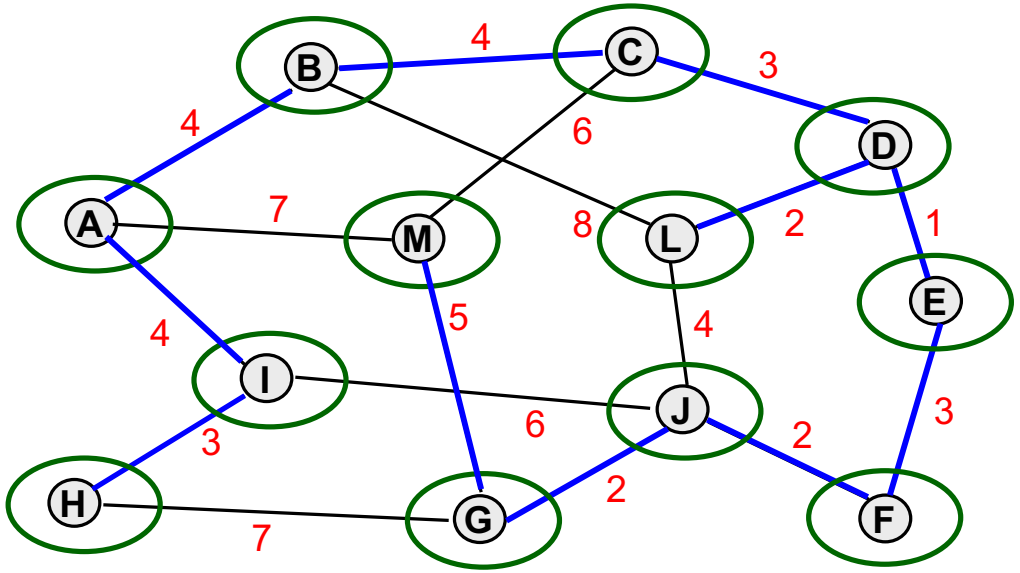
Árvores com compressão de caminhos

- Os vértices de cada componente são colocados em uma estrutura de árvore e representados pela raiz.
- Deste modo, na união de componentes, a raiz da árvore mais baixa torna-se filha da raiz da árvore mais alta.
- Encontrar a componente de um vértice corresponde a percorrer o caminho dele até sua raiz.
- Na procura da componente de um vértice, aproveita-se o percurso para que depois todos apontem para a raiz.
- Com o uso desta técnica e considerando valores práticos de n, todas as operações de testes e de união deste algoritmo gastam tempo total $O(n+m)$.
- Supondo $m > n$, o algoritmo gastaria tempo $\Theta(m.\log m)$.

Ideia de Prim (1957)

- Inicialmente, T será um vértice arbitrário de G.
- Critério de inclusão de vértices e arestas em T:
  - Dentre todas as arestas de G incidentes em T, escolhe-se a de menor custo.
  - Essa nova aresta e seu vértice adjacente serão incluídos em T somente se esse novo vértice ainda não estiver em T.
  - O processo termina quando T ficar com n vértices.
- É preciso utilizar uma estrutura de dados que armazene em ordem crescente de custo os vértices ainda não incluídos na árvore.

Exemplo



c(T) = 33

Algoritmo de Prim

```
Prim(r) {
      T ← ∅;        // árvore de espalhamento de custo mínimo
      U ← {r};      // vértices que já estão na árvore
      while (U ≠ V) {
            <u,v> = aresta de custo mínimo | u∈U e v∈V-U;
            T ← T ∪ {<u,v>}; // aqui, T contém só arestas
            U ← U ∪ {v};
      }
}
```

- Com *heap* de mínimo, o tempo de pior caso será $\Theta(m.\log n)$:
  - O *heap* possuirá apenas os vértices vizinhos da árvore em construção, cada um com sua distância corrente (começará com o vértice r, com distância nula).
  - Quando um vértice é retirado desse *heap*, modificam-se as distâncias que seus vizinhos têm em relação à árvore (será preciso manter um vetor auxiliar que armazena a posição corrente de cada vértice no *heap*).
  - No total, são realizadas n extrações de mínimo e até m modificações de valor.

Outros problemas em grafos

- Vimos algumas resoluções algorítmicas eficientes de determinados problemas em grafos.
  - Teste de planaridade: também pode ser realizado em tempo polinomial no tamanho do grafo: algoritmos APG (Auslander, Parter e Goldstein) e LEC (Lempel, Even e Cederbaum), com suas diferentes implementações
- No entanto, na teoria de grafos há ainda muitos problemas importantes que são intratáveis, ou seja, se desconhecem resoluções de tempo polinomial:
  - Clique máximo
  - Coloração
  - Caixeiro viajante
  - Cobertura de vértices
  - Conjunto de vértices dominantes
  - etc.

Exercícios

- Simular (e implementar) em diversos grafos:
  - Variantes do algoritmo de Tarjan:
    - Classificação de arcos
    - Teste de aciclidade
    - Ordenação topológica dos vértices
    - Bipartição dos vértices
    - Determinação de componentes fortemente conexas
    - Identificação de vértices e arestas de corte
  - Algoritmo de Dijkstra (com custos não negativos)
  - Algoritmo de Kruskal
  - Algoritmo de Prim