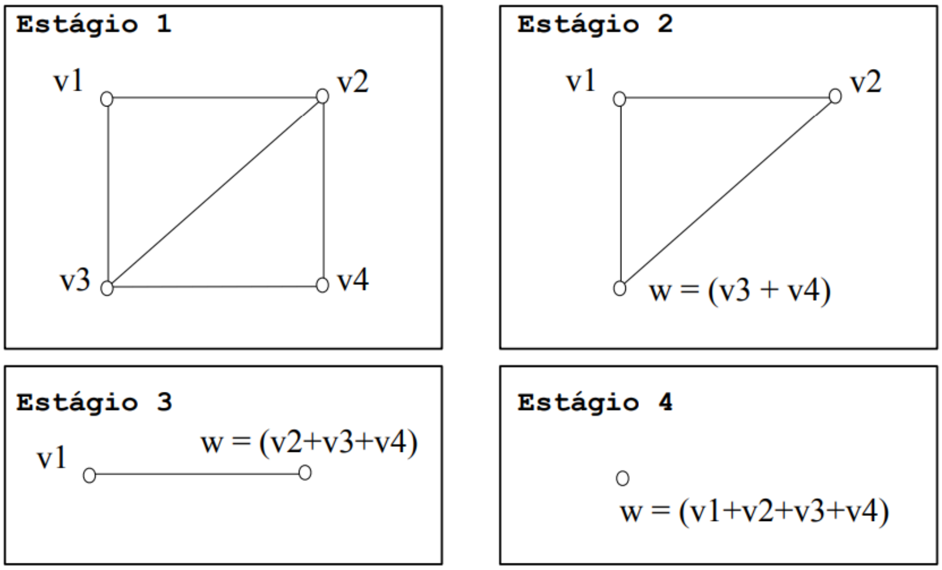
Resumo

Tópicos:

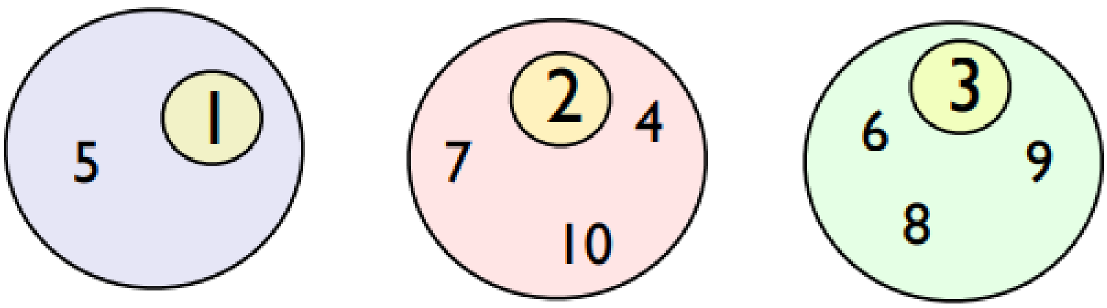
- Conexidade.
- Caminhamento em Grafos.

Conexidade

- Conexidade == atingibilidade de um vértice a partir de outro.
- Grafo Conexo == quando existe pelo menos um caminho entre cada par de vértices do grafo.
- Componente Conexa == é um subgrafo conexo.
- Algoritmos de conexidade em grafos não dirigidos: busca em largura, busca em profundidade, algoritmo de Goodman, estruturas de conjuntos.
- Algoritmo de Goodman: redução sequencial do grafo pela fusão de vértices, até que cada componente conexa seja reduzida a um único vértice.
    - A fusão de dois vértices adjacentes resulta na eliminação da aresta que existia entre eles e a criação de um novo vértice que é adjacente a todos que já eram adjacentes antes da fusão.



- Estrutura de Conjuntos Disjuntos (Ck): mantém uma coleção de conjuntos que não se sobrepõem, onde cada elemento pertence a exatamente um conjunto e cada conjunto é identificado por um representante único. Suas operações básicas permitem unir conjuntos e determinar a qual conjunto um elemento pertence de forma eficiente.



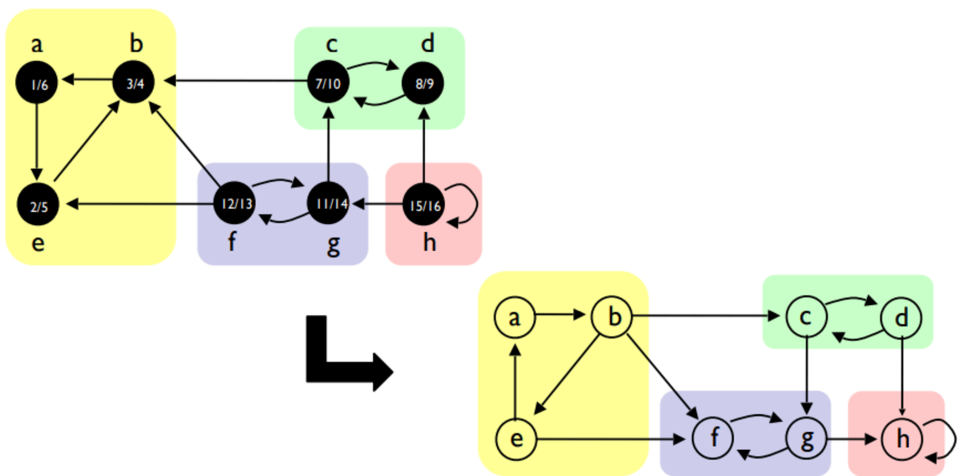
Operações:

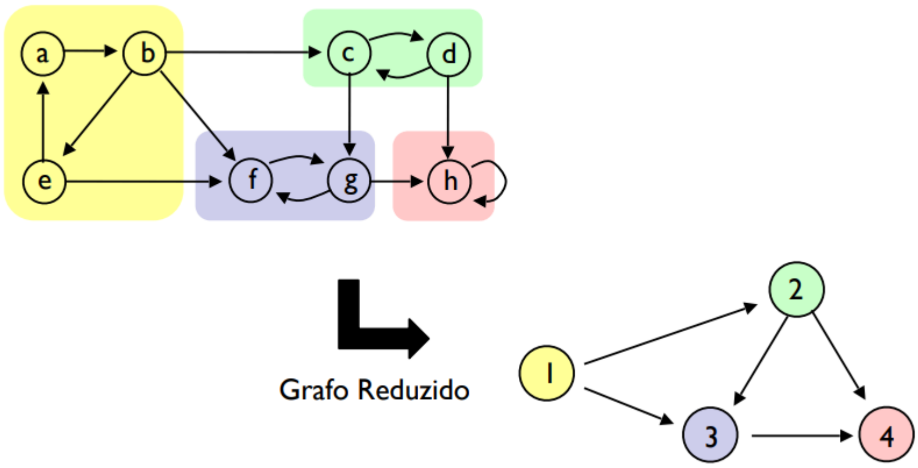
`Make-Set(x):`	cria um novo conjunto cujo único elemento é apontado por x. x não pode pertencer a outro conjunto da coleção
`Union(x, y):`	executa a união dos conjuntos que contêm x e y, digamos Sx e Sy ,em um conjunto único. – $Sx \cap Sy = \varnothing$ – O representante de $S = Sx \cup Sy$ é um elemento de S
`Find-Set(x):`	retorna um ponteiro para o representante (único) do conjunto que contém x

- Grafo Subjacente: para um digrafo, seu grafo subjacente é um grafo não dirigido a partir da substituição de todas as arestas dirigidas por arestas não dirigidas.
- Digrafo Conexo: um digrafo é conexo se seu grafo subjacente dor conexo.
- Componente Fortemente Conexa: é um subgrafo de um digrafo que é um conjunto maximal (abrange o maior número de vértices possíveis sem perder a propriedade). Sua propriedade diz que deve existir um caminho entre o vértice v e u e vice-versa.
- Grafo Transposto: é um segundo digrafo que é igual ao primeiro, com exceção de que a direção de suas arestas esta invertida. Para um digrafo G=(V,E), seu grafo transposto é definido por

$$G^T = (V, E^T)$$

  - Passo a Passo:
    - Faça uma busca em profundidade e calcule o tempo de finalização em cada vértice u.
    - Gere o grafo transposto.
    - Faça a busca em profundidade no grafo transposto, mas considerando os vértices acessíveis na ordem decrescente ao seu tempo de finalização encontrado no passo 1.
    - Cada floresta encontrada no passo 3, corresponde a um componente fortemente conexo.





Caminhamento em Grafos - Dijkstra

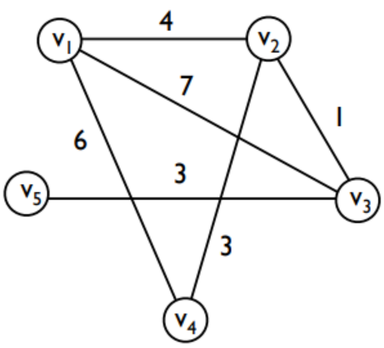
- Grafo Valorado: é um grafo cuja arestas possuem um valor numérico. Esse custo representa alguma grandeza numérica relevante para o problema (distância, tempo, valor monetário).

  valor numérico $w(u, v)$ ou w_{uv}, chamado de custo da aresta (u, v).

- Matriz de Custos: os custos de um grafo valorado podem ser armazenados em uma matriz W, definida por:

$$W_{i,j} = \begin{cases} 0, & \text{se } v_i = v_j \\ \infty, & \text{se } (v_i, v_j) \notin E \\ \text{custo}, & \text{se } (v_i, v_j) \in E \end{cases}$$

  - Exemplo:



$$W = \begin{vmatrix} 0 & 4 & 7 & 6 & \infty \\ 4 & 0 & 1 & 3 & \infty \\ 7 & 1 & 0 & \infty & 3 \\ 6 & 3 & \infty & 0 & \infty \\ \infty & \infty & 3 & \infty & 0 \end{vmatrix}$$

- Custo Mínimo: o custo de um caminho p=<V0,V1,...,Vk> entre dois vértices é igual ao somatório dos custos de todas as arestas valoradas do caminho.

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

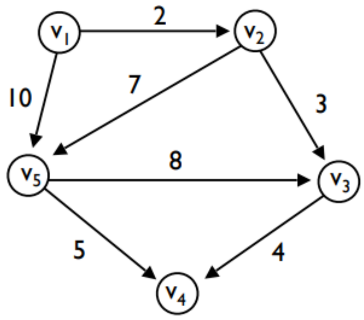
- Caminho Mínimo: é o menor custo do conjunto entre os vértices u e v, quando for conexo. Caso contrário, é infinito.
    - Problema do caminho mínimo como origem: encontrar os caminhos mais curtos a partir de um ponto inicial até todos os outros pontos num grafo com pesos.

$$\delta_{u,v} = \delta(u,v) = \begin{cases} \min\{\ w(p): u \rightarrow v\ \}, & \text{se } \exists \text{ caminho de u para v} \\ \infty, & \text{caso contrário} \end{cases}$$

- Algoritmo de Dijkstra: resolve esse problema porque ele foi projetado justamente para explorar o grafo de maneira "gananciosa" (greedy), escolhendo sempre o próximo vértice acessível de menor custo, e assim garante que a primeira vez que chega a um vértice, esse é o caminho mais curto até ele (desde que não existam pesos negativos).

Elemento	Descrição
s	vértice inicial
v	quaisquer outros vértices
u	vértice "pivô", que pode representar uma mudança no caminho mínimo até o vértice v
d[v]	custo estimado do caminho mínimo de s até v
w(u,v)	custo da aresta (u, v), tem valor ∞ se não existir aresta (u, v)
π[v]	vértice predecessor do vértice v
Q	fila de prioridades mínimas de vértices (o vértice com menor valor de d[v] tem prioridade para sair da fila)
S	conjunto dos vértices cujo caminho mínimo já foi calculado

Exemplo




```
INITIALIZE-SINGLE-SOURCE(G,s)
01. for each vertex v ∈V[G]
02.     do d[v] ← ∞
03.        ∏[v] ← NULL
04. d[s] ← 0;
```

```
RELAX(u, v, w)
01. if d[v] > d[u] + w(u,v)
02.     then d[v] ← d[u] + w(u,v)
03.          ∏[v] ← u
```

```
DIJKSTRA(G, w, s)
01. INITIALIZE-SINGLE-SOURCE(G,s)
02. S ← ∅
03. Q ← V[G]
04. while Q ≠ ∅
05.     do u ← EXTRACT-MIN(Q)
06.        S ← S ∪ [u]
07.        for each vertex v ∈Adj[u]
08.            do RELAX(u,v,w)
```

$$W = \begin{vmatrix} 0 & 2 & \infty & \infty & 10 \\ \infty & 0 & 3 & \infty & 7 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 8 & 5 & 0 \end{vmatrix}$$

vértice	v1	v2	v3	v4	v5
d	0	2	5	9	9
∏	nil	V1	V2	V3	V2
Q					
S	x	x	x	x	x

INITIALIZE-SINGLE-SOURCE(G,s)
01. for each vertex v ∈ V[G]
    02. do d[v] ← ∞
    03. ∏[v] ← NULL
04. d[s] ← 0;

RELAX(u, v, w)
01. if d[v] > d[u] + w(u,v)
    02. then d[v] ← d[u] + w(u,v)
        03. ∏[v] ← u

DIJKSTRA(G, w, s)
01. INITIALIZE-SINGLE-SOURCE(G,s)
02. S ← ø
03. Q ← V[G]
04. while Q ≠ ø
    05. do u ← EXTRACT-MIN(Q)
        06. S ← S ∪ [u]
        07. for each vertex v ∈ Adj[u]
            08. do RELAX(u,v,w)

INITIALIZE-SINGLE-SOURCE(G,s) → Inicializa todas as variáveis antes de começar o algoritmo.

- 01 → para cada vértice v do grafo.
  - 02 → define o custo do caminho (d) da origem s até o destino v como ∞, isto é, desconhecido.
  - 03 → Inicialmente não há predecessor conhecido para v.
- 04 → distância de origem até ela mesma é 0.

RELAX(u, v, w) → verifica se passar pelo vértice u gera um caminho mais curto até v, e atualiza os valores se sim.

- 01 → verifica se passando por u chegamos a v com cust melhor do que conhecemos.
  - 02 → Se sim, atualiza a distância de s para v com o novo melhor valor.
    - 03 →atualiza o predecessor de v. Agora v é alcaçado vindo de u. Isso permite reconstruir o caminho no final.

DIJKSTRA(G, w, s) → controlar todo o processo (escolher o próximo vértice a processar e aplicar o relaxamento em seus vizinhos).

- 01 → Chama a rotina INITIALIZE-SINGLE-SOURCE(G,s) para inicializar as variáveis.
- 02→ S é o conjunto de vértices cuja distância mínima já foi definitivamente determinada, por isso, inicialmente é vazio.
- 03 → Q é a estrutura que contém os vértices ainda não processados. Incialmente, contém todos os vértices.
- 04 → Repete enquanto ainda houver vértices não finalizados.
  - 05 → remove de Q o vértice u com menor valor d[u] entre os que restam.
    - 06 → marca u como finalizado acrescentando e s. A distância de u não mudará mais.
    - 07 → percorre todos os vizinhos v de u (aresta u→v).

- 08 → tenta melhorar a estimativa d[v] usando o caminho que chega em v através de. Se d[u] + w(u,v) for menor que d[v], atualiza d[v] e π[v].

Caminhamento em Grafos - Floyd-Warshall

- Encontra os menores caminhos entre todos os pares de vértices de um grafo valorado, mesmo quando existem arestas com pesos negativos (mas sem ciclos negativos).
  - A questão dos loops negativos é que a cada volta o custo para chegar ao destino diminui, mesmo que o caminhamento não tenha andado.
- “Se eu permitir usar o vértice k como ponto intermediário, consigo achar um caminho mais curto entre i e j?”
- Matriz de custos W:
  - Wi,j = 0 → loop.
  - Wi,j = ∞ → não há arestas entre i e j.
  - Wi,j = custo → existe aresta direta entre i e j.
- D^o→ matriz inicial.
- Vantagens: descobre os menores caminhos entre os pares, excelente para grafos pequenos, aceita valores negativos com exceção de loops.
- Desvantagnes: quando o grafo é muito grando, o tempo e a memória crescem cúbicamente.
- Π → Matriz de Roteamento.
- D → Matrizes de distância.
- Algoritmo:

```
FLOYD-WARSHALL(G, W)
para cada i ← 1 até n faça
  para cada j ← 1 até n faça
    D0ij ← Wij
    se vi ≠ vj e wij < ∞ então
      Πij ← vi
```

```
06.     senão
07.       Πij ← NIL

08. para cada k ← 1 até n faça
09.   para cada i ← 1 até n faça
10.     para cada j ← 1 até n faça
11.       se (di,k + dk,j) < di,j então
12.         dij ← di,k + dk,j
13.         Πij ← Πkj
14. retorno Dn, Πn
```

- 01-02 → Percorre todas as combinações possíveis de vértices.
- 03 → copia a matris de custos W para a matriz de distâncias inicial D^0. Se há aresta direta `i->j` , `D0[i][j] = w(i,j)` . Se i = j, W[i][i] = 0. Se não há aresta, w[i][j] = ∞.
- 04 → Se o vértice origem não for o mesmo que o vértice destino (não faz sentido ter um predecessor de um vértice para ele mesmo) e se existe uma aresta direta de i para j (menor que infito = custo; só há predecessor se existir ligação).
- 05 → no caminho direto de i para j, o nó anterior de j é i.
- 06-07 → se não existe aresta direta ou i==j, então não há predecessor. Marca NIL.
- 08 → a cada iteração de k (k+1), permitimos que os caminhos usem qualquer vértice entre 1...k como nós intermediários.
- 09-10 →Para o k atual, testamos todos os pares de vértices para ver se melhora a matriz de distâncias D[i][j].
- 11 → Se o caminho i→k→ é mais curto do que o caminho i→j.
  - 12 →Atualiza a distância de menor custo.
  - 13 → Se o melhor caminho de i até j passa por k,
    então o predecessor de j no caminho que começa em i é o mesmo predecessor que j tem no caminho que começa em k.
- 14 → Ao fim, retorna a matriz de menores distâncias entre todos os pares e Π permite reconstruir os caminhos. O algoritmo devolve essas duas matrizzes.

Grafos e Ciclos Eulirianos e Hamiltonianos

- Ciclo Euliriano: é um caminho fechado, ou seja, começa e termina no mesmo vértice, que passa por todas as arestas do grafo exatamente uma vez. O percurso pode repetir vértices, mas nunca arestas.
    - Condição de existência: em um grafo conexo não direcionado, todos os vértices devem ter grau par.
- Ciclo Hamiltoniano: é um caminho fechado que passar por todos os vértices do grafo exatamente uma vez. Ele não precisa usar todas as arestas, mas não pode repetir vértices (exceto o primeiro, que é igual ao último).