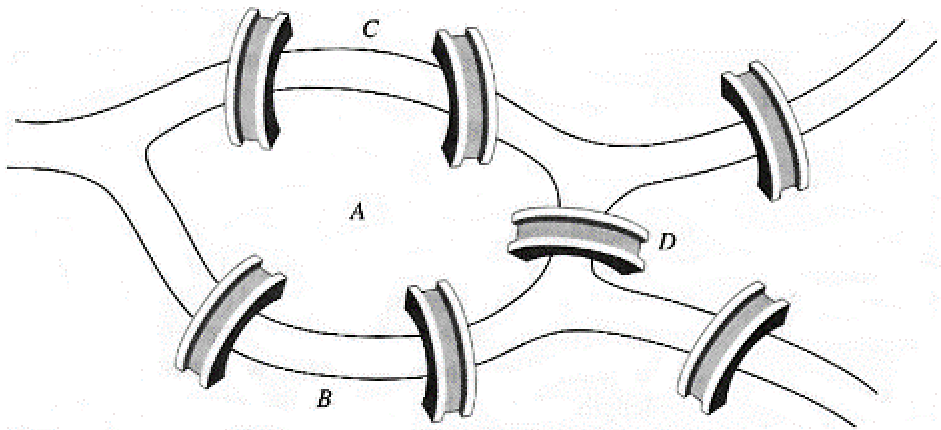
Graphs

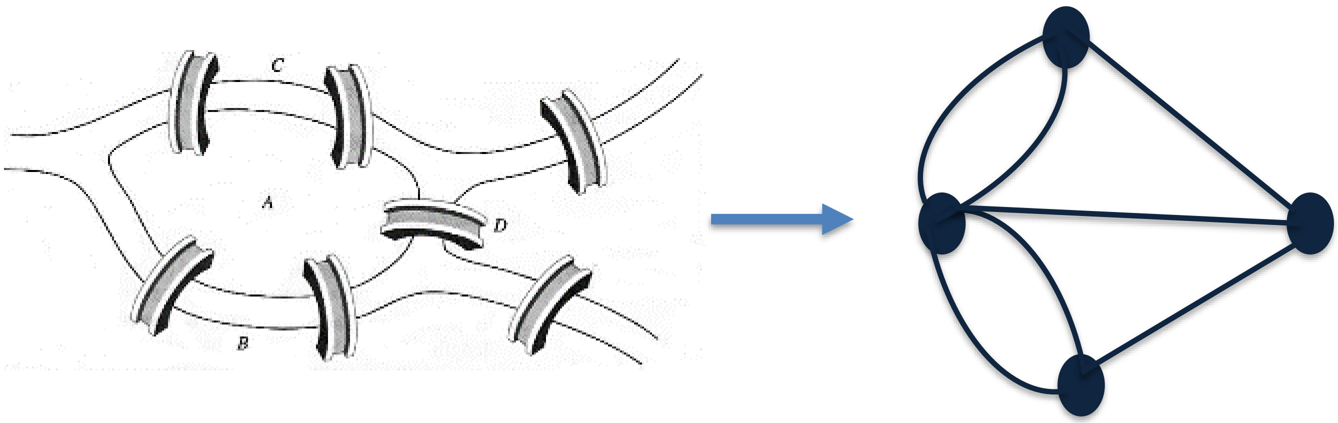
Graph Theory



- Königsberg was a city in Germany in 18th century. There was a river named Pregel that divided the city into four distinct regions.
- There was a natural question for the people of Königberg :

'Is it possible to take a walk around the city that crosses each bridge exaactly once?'

Graph Theory



- The problem was solved by Swiss mathematician Leonard Euler. His works are considered as the beginning of Graph Theory.

- Euler represented four distinct lands with four points (or nodes), and seven bridges with seven lines connecting those points.

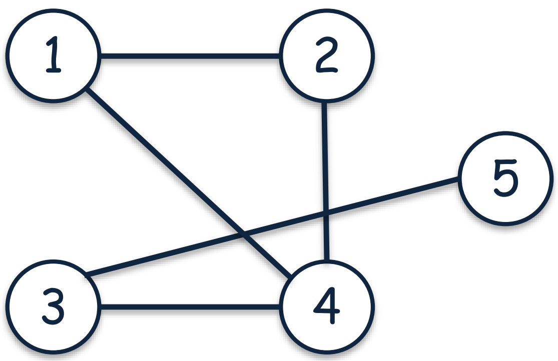
'Can you find a path that includes every edge exactly once?'
'Is the given graph traversable?'

Graph Theory

$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



- $V = \{1, 2, 3, 4, 5\}$
- $E \subseteq V \times V$

  $(1, 2) \in E$

  starting node

  ending node

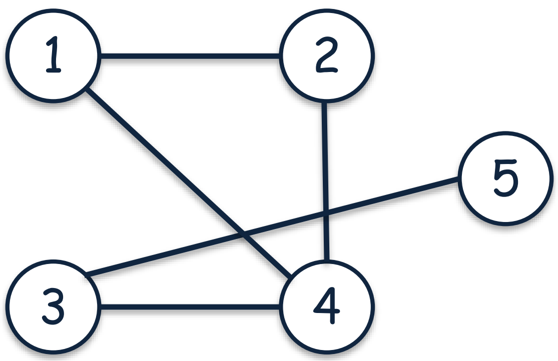
- $E = \{(1,2), (2,4), (4,3), (1,4), (3,5)$
  $(2,1), (4,2), (3,4), (4,1), (5,3)\}$
- If $(1, 2) \in E$, 1 and 2 are adjacent vertices.
- adj(4) = {1, 2, 3}

Graph Theory

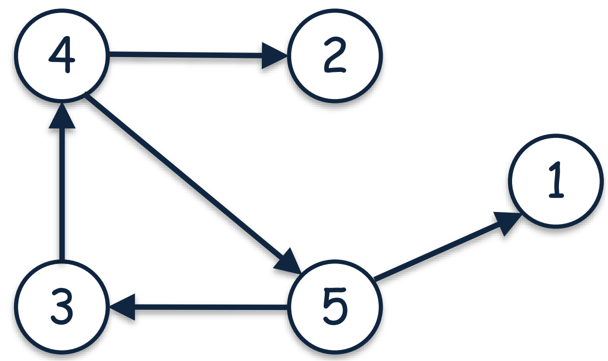
$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



undirected graph



directed graph

deg(v)= # of edges at that vertex

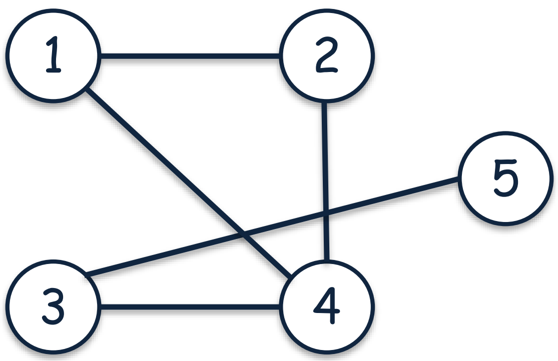
deg(1)=2

deg(4)=3

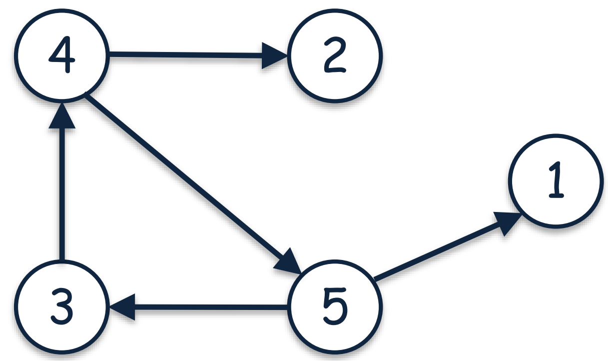
Graph Theory

$$G = (V, E)$$

set of nodes (or vertices) set of edges (or arc)



undirected graph



directed graph

deg(v)= # of edges at that vertex

$deg^{in}(v)$ = # of incoming edges

$deg^{out}(v)$ = # of outgoing edges

$deg^{in}(5) = 1$

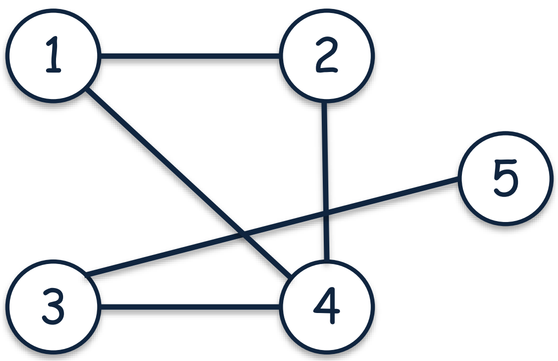
$deg^{out}(4) = 2$

Graph Theory

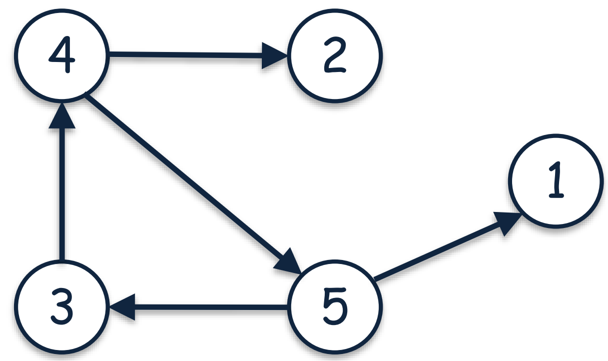
$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



undirected graph



directed graph

$deg(v)$ = # of edges at that vertex

$$\sum deg(v) = 2\,|E|$$

- a vertex v is called odd vertex if deg(v) is odd
- a vertex v is called even vertex if deg(v) is even

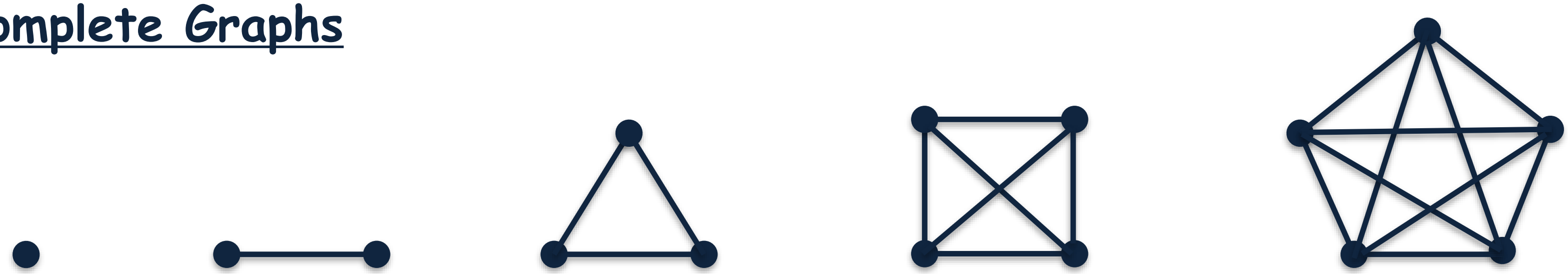
$deg^{in}(v)$ = # of incoming edges

$deg^{out}(v)$ = # of outgoing edges

$$\sum deg^{in}(v) = \sum deg^{out}(v) = |E|$$

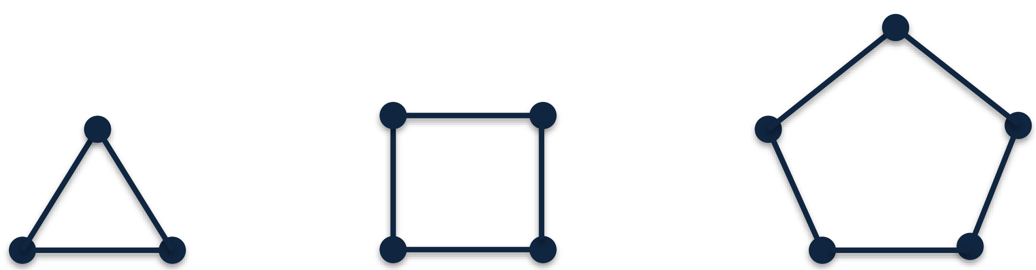
Graph Theory

Complete Graphs



K_1 K_2 K_3 K_4 K_5

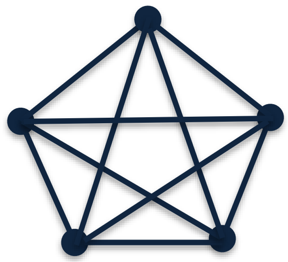
Cycle Graphs



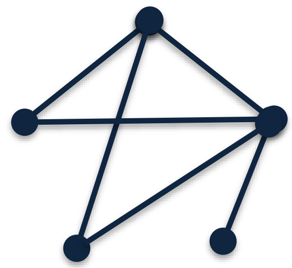
C_3 C_4 C_5

Graph Theory

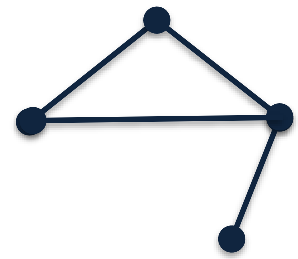
- a subgraph of a graph G = (V, E) is a graph H = (W, F) such that W⊆V and F ⊆ E.



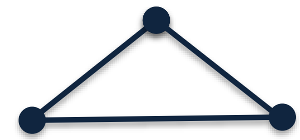
K_5



$G_1 \subseteq K_5$

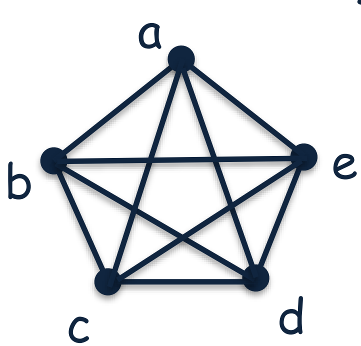


$G_2 \subseteq K_5$
$G_2 \subseteq G_1$

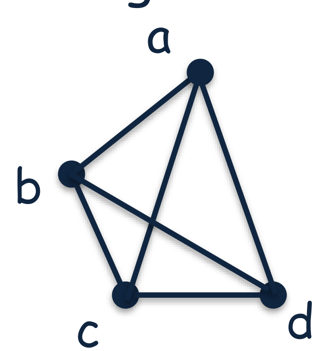


$G_3 \subseteq K_5$
$G_3 \subseteq G_2$

- the subgraph induced by a subset W of the vertex set V is the graph (W, F) where the edge set F contains an edge in E if and only if both starting node and ending node of this edge are in W.



$K_5 = (V, E)$

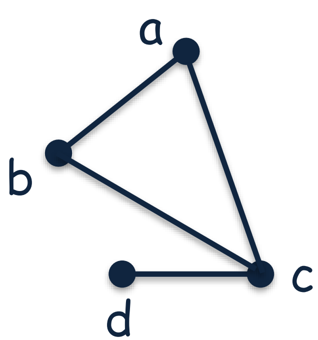


$H = \langle\{a, b, c, d\}\rangle$

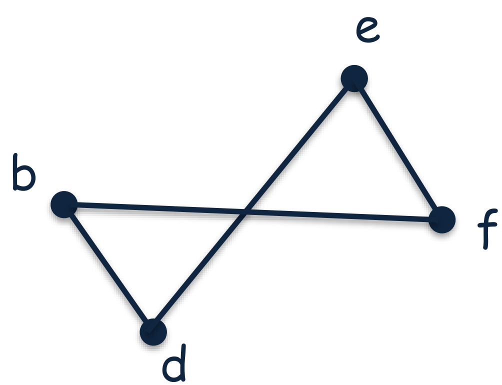
the subgraph induced by W={a, b, c, d}

this subgraph produced by removing the edge e

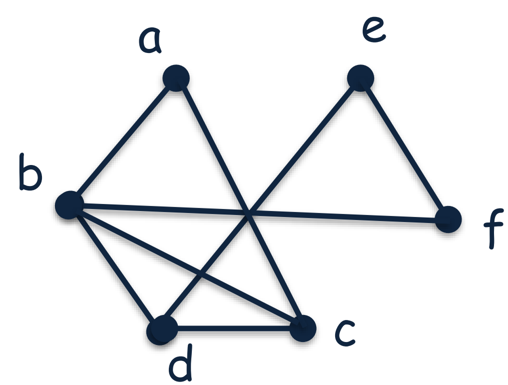
Graph Theory



$G_1 = (V_1, E_1)$

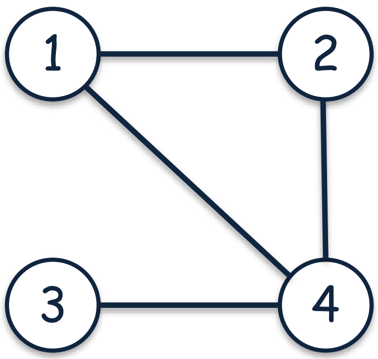


$G_2 = (V_2, E_2)$



$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$

Representation

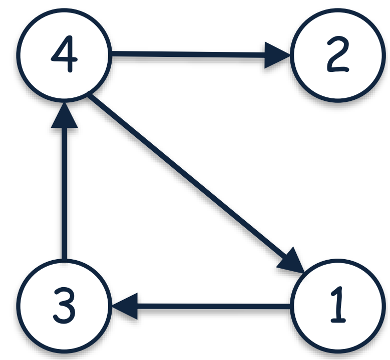


Adjacency List

1 - 2,4
2 - 1,4
3 - 4
4 - 1,2,3

Adjacency Matrix

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0



Adjacency List

1 - 3
2 -
3 - 4
4 - 1,2

Adjacency Matrix

	1	2	3	4
1	0	0	1	0
2	0	0	0	0
3	0	0	0	1
4	1	1	0	0

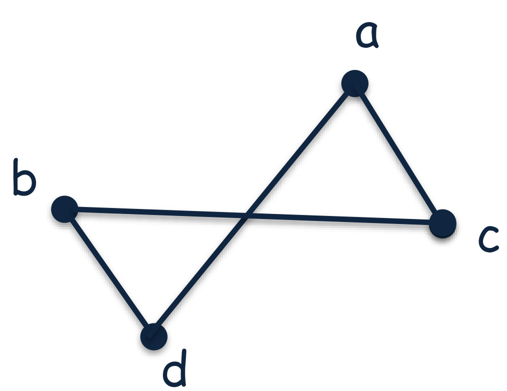
Representation

	Adjacency List	Adjacency Matrix
• retrieving all neighbors of a given node u	O(deg(u))	O(|V|)
• given nodes u and v, checking if u and v are adjacent	O(deg(u))	O(1)
• space	O(|E|+|V|)	O(|V|^2)

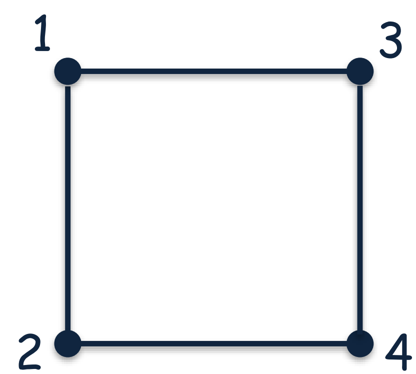
If graph is sparse, use adjacency list;
if graph is dense, use adjacency matrix

Isomorphism

- Two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection f from V_1 to V_2 such that a and b are adjacent in G_1 if and only if f(a) and f(b) are adjacent in G_2 for all $a, b \in V_1$



$G_1 = (V_1, E_1)$



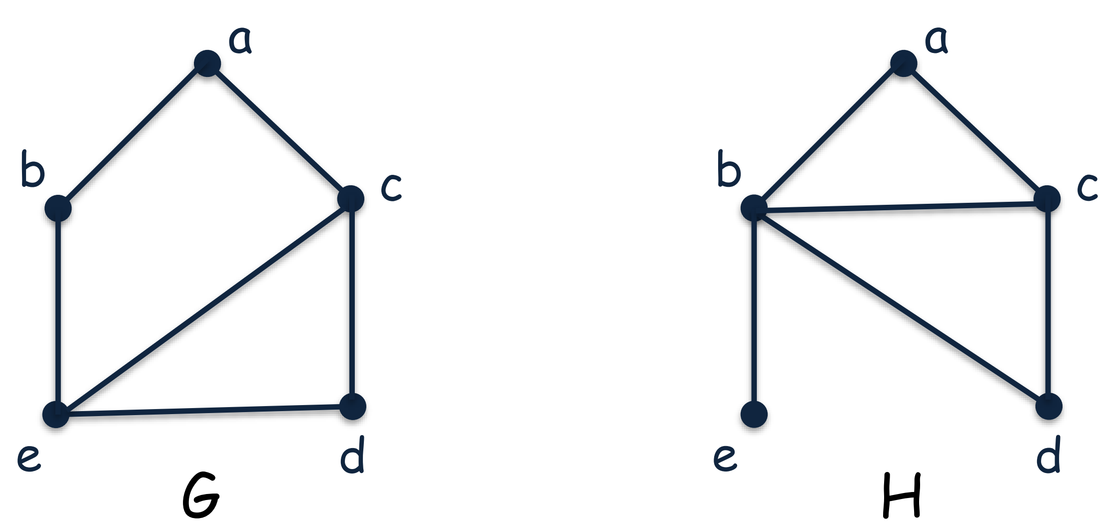
$G_2 = (V_2, E_2)$

- f : $V_1 \rightarrow V_2$, f(a) = 1, f(b) = 4, f(c) = 3, f(d) = 2

a and c are adjacent in G_1, f(a) = 1 and f(c) = 3 are adjacent in G_2
a and d are adjacent in G_1, f(a) = 1 and f(d) = 2 are adjacent in G_2
b and d are adjacent in G_1, f(b) = 4 and f(d) = 2 are adjacent in G_2

Isomorphism

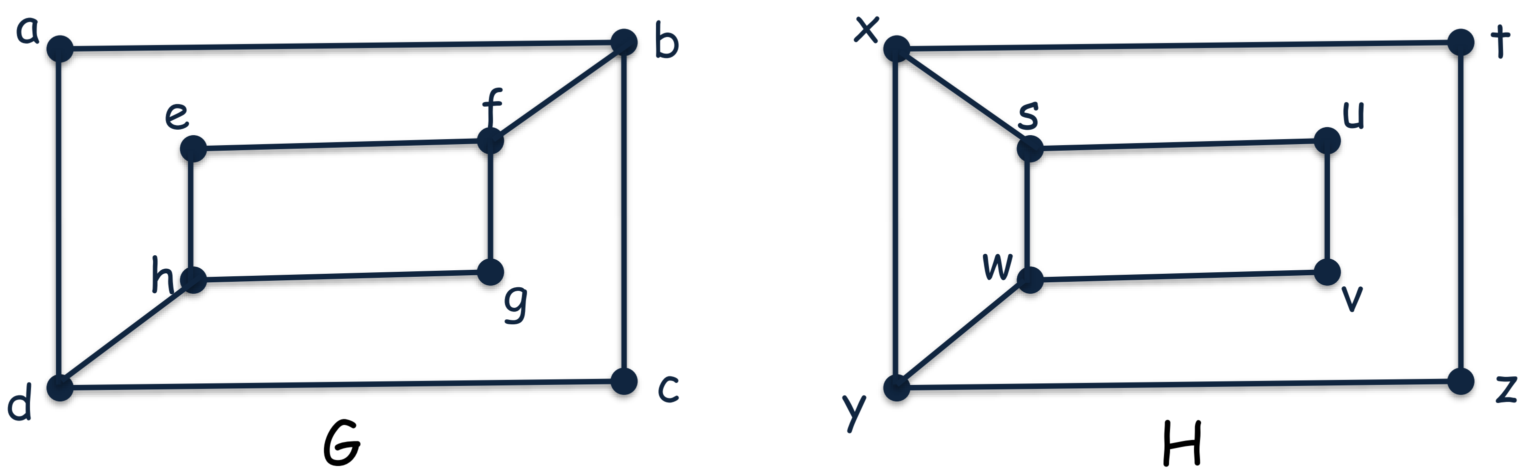
- Isomorphic graphs must have same number of edges
- The degrees of the vertices in isomorphic graphs must be same



- G and H both have 5 vertices and 6 edges
- G has 3 vertices of degree two and 2 vertices of degree three
  H has 1 vertex of degree one, 2 vertices of degree two, 1 vertex of degree three, and 1 vertex of degree 4

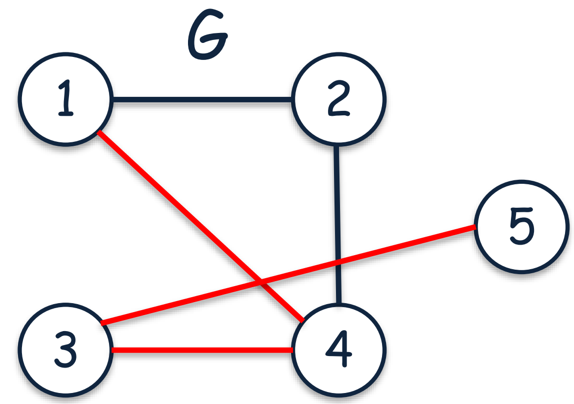
Isomorphism

- Isomorphic graphs must have same number of edges
- The degrees of the vertices in isomorphic graphs must be same



- G and H both have 8 vertices and 10 edges
- G has 4 vertices of degree two and 4 vertices of degree three
  H has 4 vertices of degree two and 4 vertices of degree three
- One of the odd vertices (s) in H has 2 adjacent odd vertices (w and x)
  We don't have such case in G

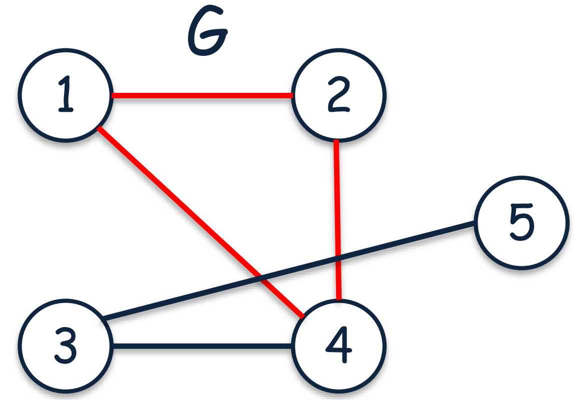
Connectivity



5, 3, 4, 1 is a simple path in G

- a path in a graph is a sequence of nodes v_1, v_2, …, v_k such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct

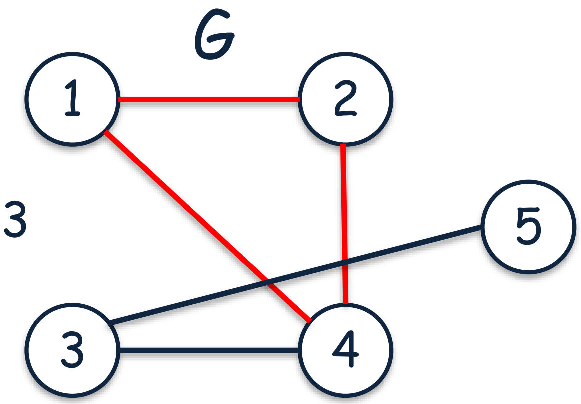
Connectivity



4, 1, 2, 4 is a simple cycle in G

- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct
- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes
- a cycle is a path $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A cycle is simple if first k-1 nodes are distinct

Connectivity



4, 1, 2, 4 is a simple cycle with length 3

- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct

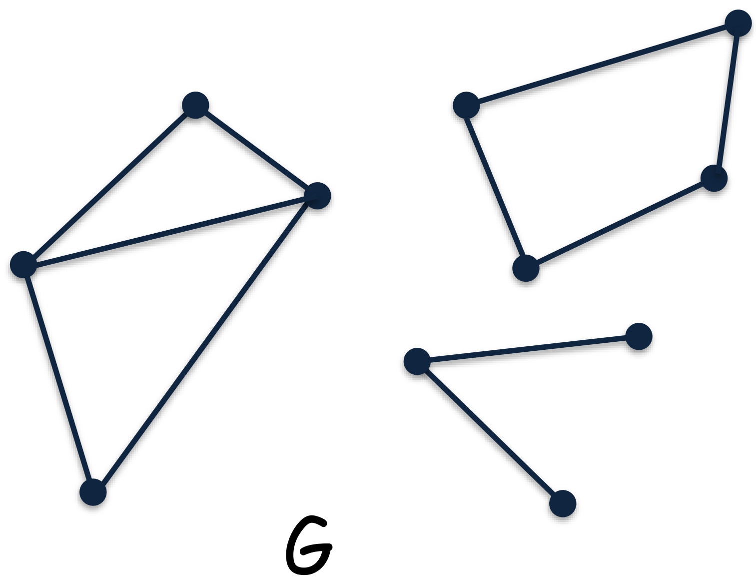
- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes

- a cycle is a path $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A cycle is simple if first k-1 nodes are distinct

- length of a path is the number of edges in the path

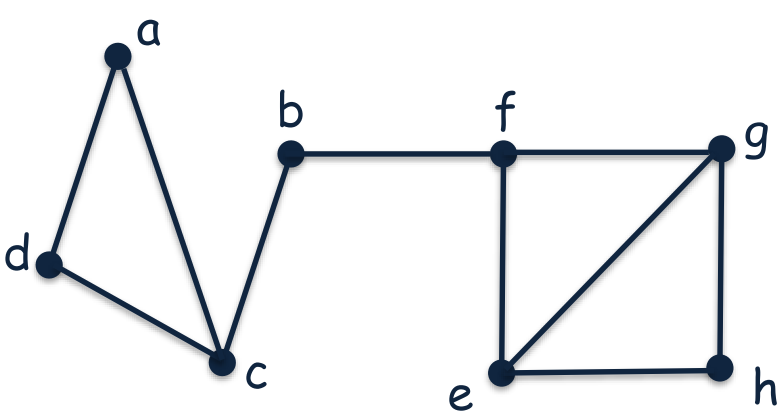
Connectivity

- Given $G = (V, E)$ and $H \subseteq G$, if there is no proper subgraph U of G ($U \subset G$) such that $H \subseteq U$, H is called a maximal subgraph of G.
- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components



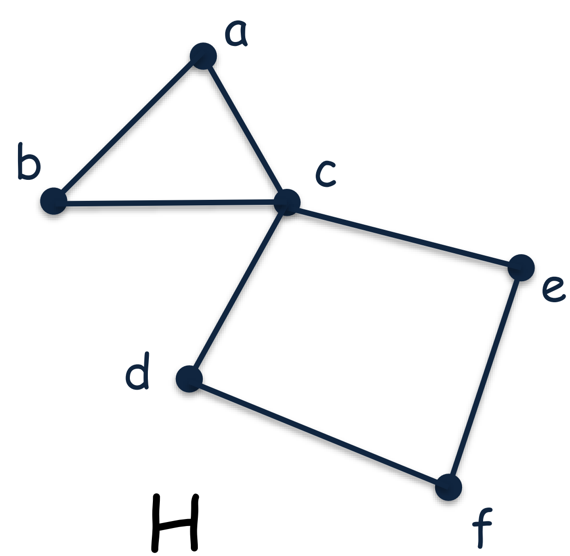
Connectivity

- Consider a vertex v of a given graph $G=(V,E)$, if removing v and all its inncident edges from the graph produces a subgraph with more connected components, v is called cut vertex (or cut vertices)
- Similarly, if removing an edge from a graph creates a subgraph with more connected components, it's called cut edge



G

cut vertices : {b, c, f}
cut edges : {(b, f), (c, b)}

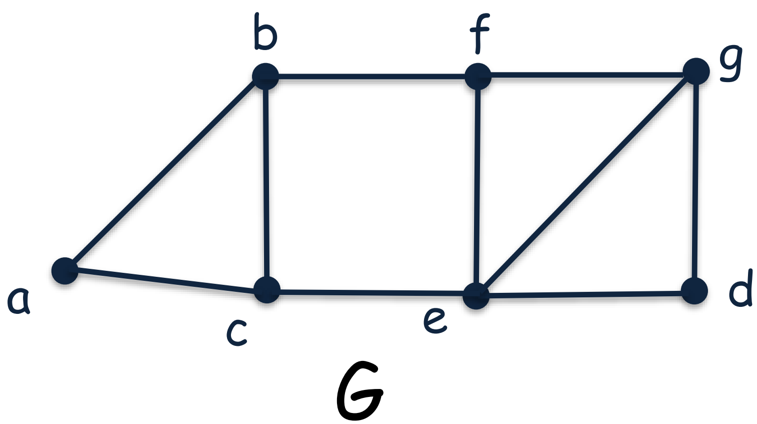


H

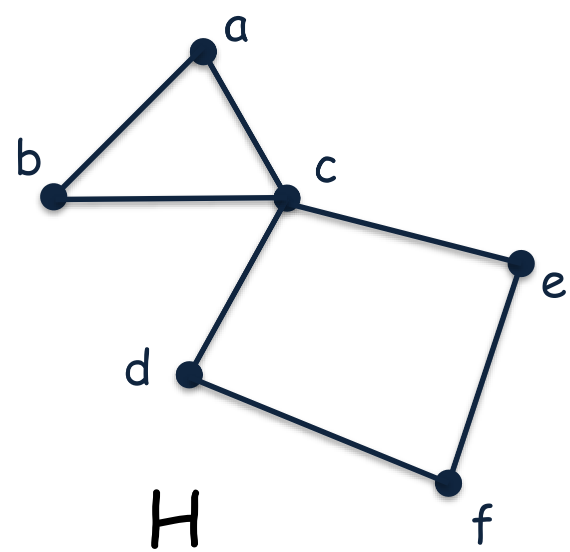
cut vertices : {c}
cut edges : { }

Connectivity

- A subset W of the vertex set V of $G = (V, E)$ is called a vertex cut or separating set, if G – W is disconnected
- Similarly, a subset F of the edge set E of $G = (V, E)$ is called a edge cut, if G – F is disconnected



vertex cut: {b, c} or {f, e}
edge cut: {(b, f), (c, e)} or {(a, c), (a, b)}
no cut vertex and no cut edge

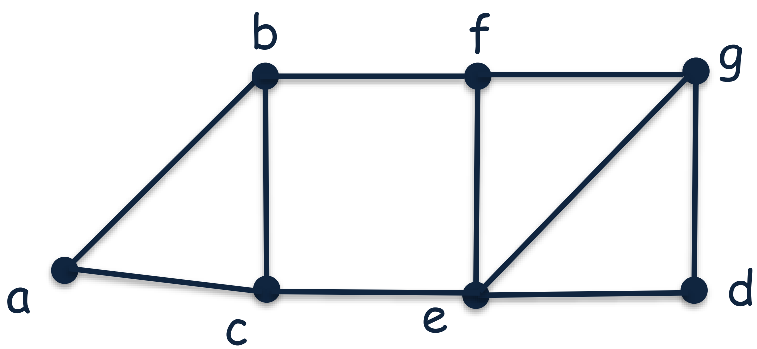


vertex cut: {c}
edge cut: {(d, c), (c, e)}
no cut edge

Connectivity

- A subset W of the vertex set V of $G = (V, E)$ is called a vertex cut or separating set, if G - W is disconnected.
- Simi edge cut,

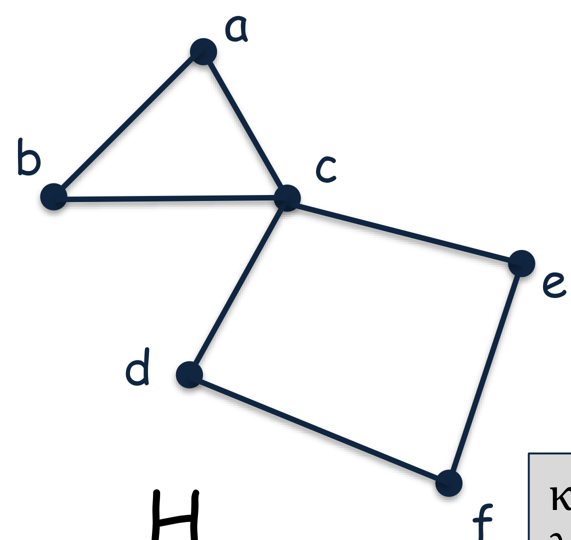
$\kappa(G)$: minimum number of vertices in a vertex cut
$\lambda(G)$: minimum number of edges in a edge cut



G

$\kappa(G) = 2$
$\lambda(G) = 2$

vertex cut: {b, c} or {f, e}
edge cut: {(b, f), (c, e)} or {(a, c), (a, b)}
no cut vertex and no cu



H

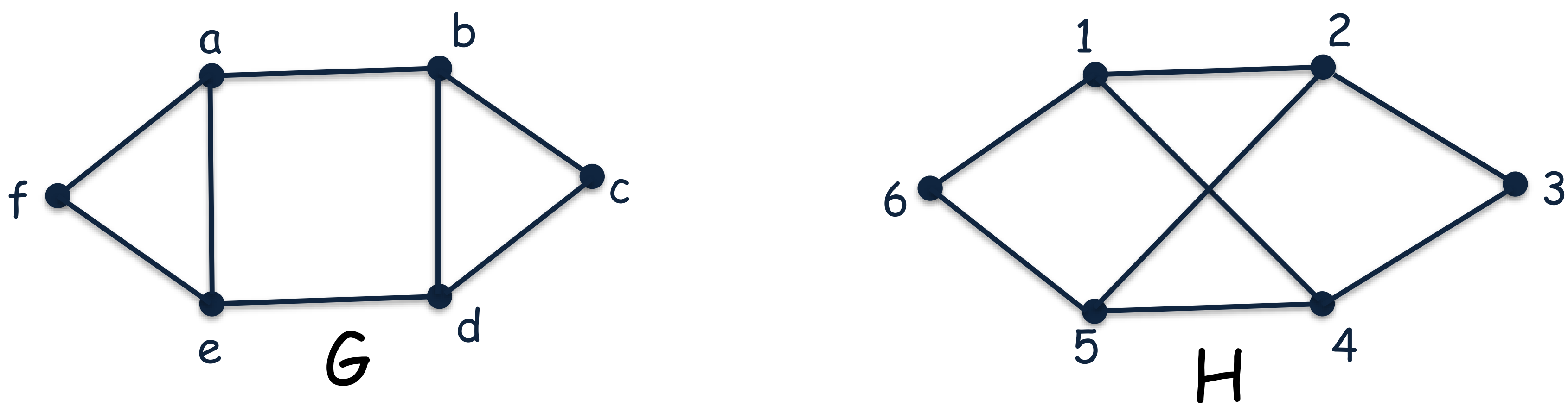
$\kappa(G) = 1$
$\lambda(G) = 2$

vertex cut: {c}
, e)}

$$\kappa(G) \leq \lambda(G) \leq min_{v \in V} \deg(v)$$

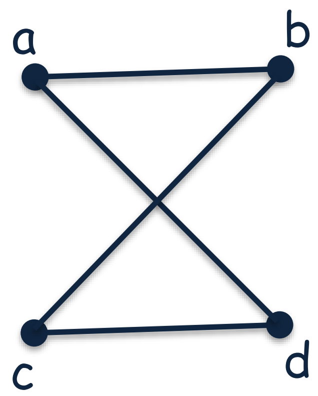
Isomorphism

- Isomorphic graphs must have same number of edges
- The degrees of the vertices in isomorphic graphs must be same
- They must have same amount of simple circuits of length k



- G and H both have 6 vertices and 8 edges
- G has 2 vertices of degree two and 4 vertices of degree three
  H has 2 vertices of degree two and 4 vertices of degree three
- G has two simple circuits of length three; however, H has no simple circuit of length three

Connectivity



- How many paths of length two from a to c ?

  a, b, c or a, d, c

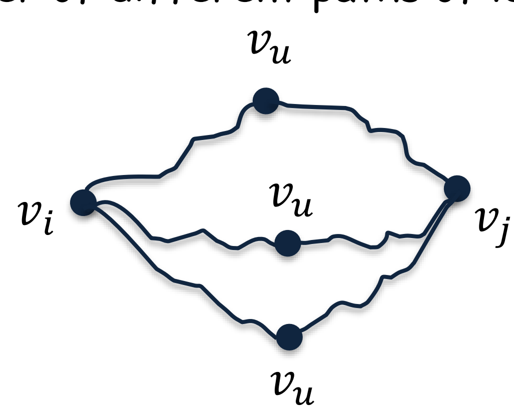
- For a given graph $G = (V, E)$, what are the number of different paths of length k from one vertex to another one ?

- Given a graph $G = (V, E)$ together with the adjacency matrix A, the number of different paths of length m from v_i to v_j will be the (i, j)-th entry of A^m

<u>Basis Step</u> $(k = 1)$ For $A = (a_{ij})$, a_{ij} will be the number of different path of length 1 from v_i to v_j (true)

<u>Inductive Step</u> Assume it's true for k, i.e. the number of different paths of length k from v_i to v_j will be the (i, j)-th entry of A^k.

For $k + 1$, $A^{k+1} = A^k . A$

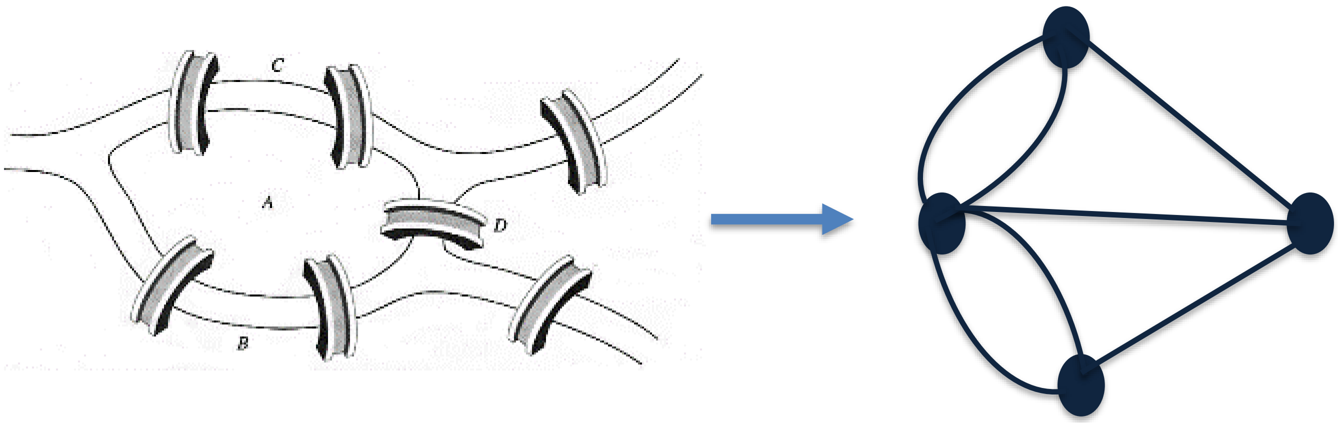


$$A^{k+1} = \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

$$c_{ij} = b_{i1}.a_{1j} + b_{i2}.a_{2j} + \ldots + b_{in}.a_{nj}$$

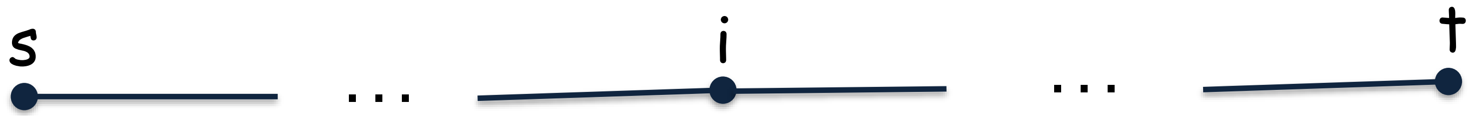
c_{ij} : the number of different paths of length (k+1) from v_i to v_j

Euler Paths and Circuits

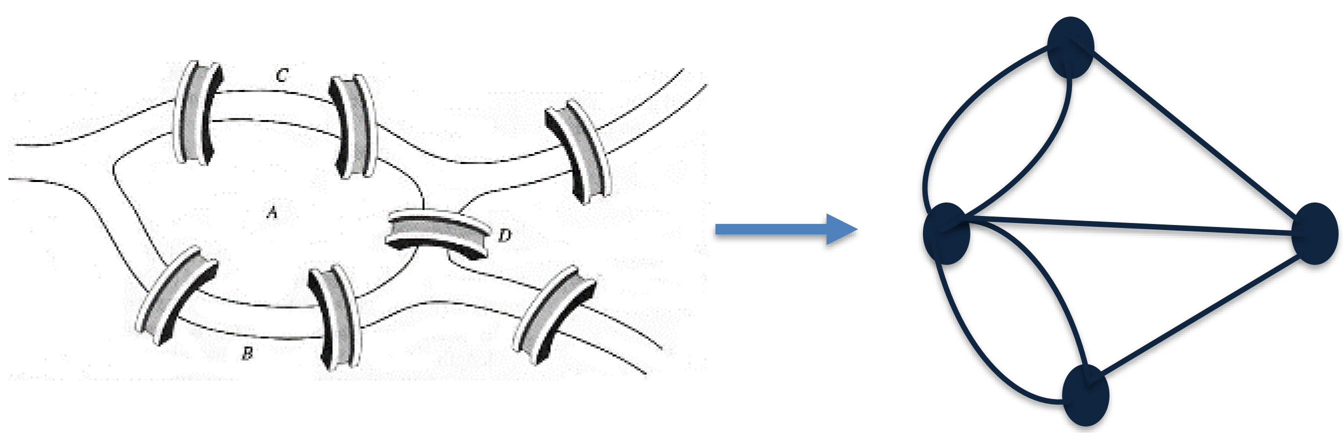


- Euler circuit is a simple circuit that contains every edge of G.
- Euler path is a simple path that contains every edge of G
- Does this graph have an Euler path or Euler circuit?

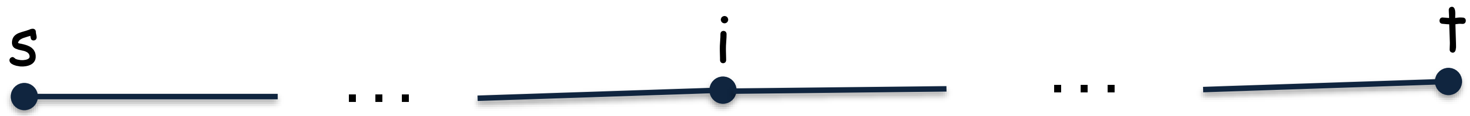
Euler Paths and Circuits



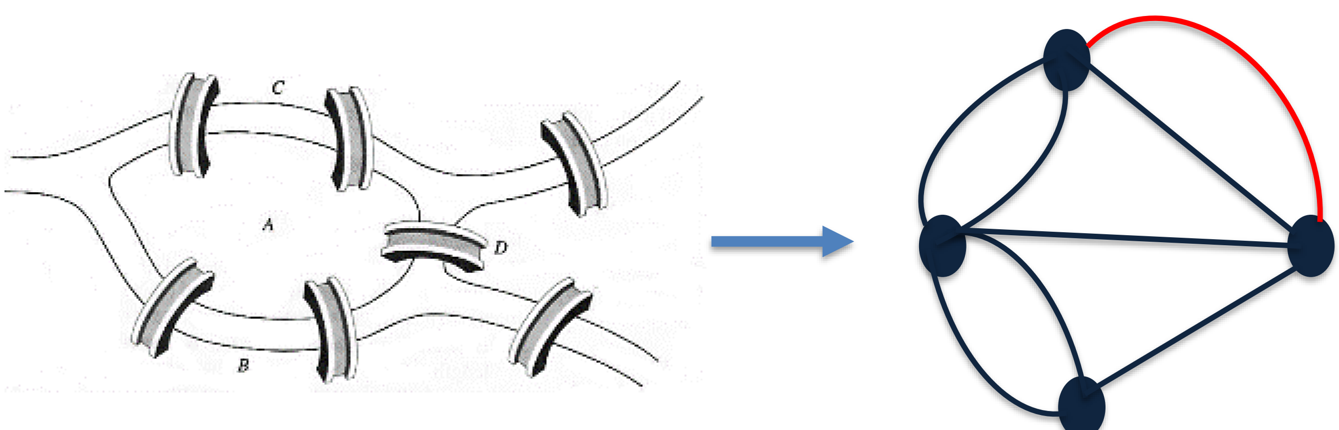
- when you pass a vertex, you add two to the degree of it.
- the degree of starting node and ending node just one or odd number
- the graph has a Euler path or Euler circuit if if it has no odd vertex or exactly two odd vertices.



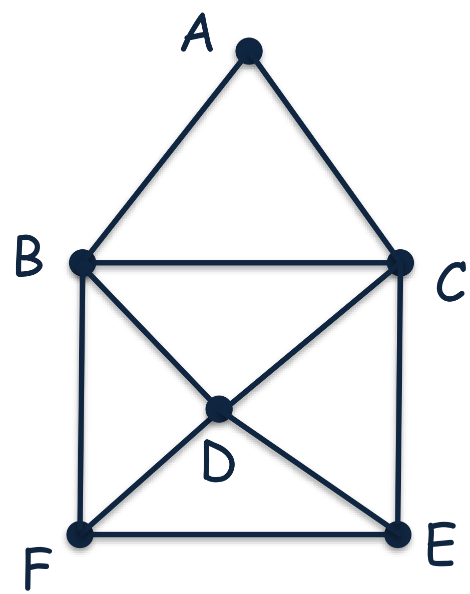
Euler Paths and Circuits



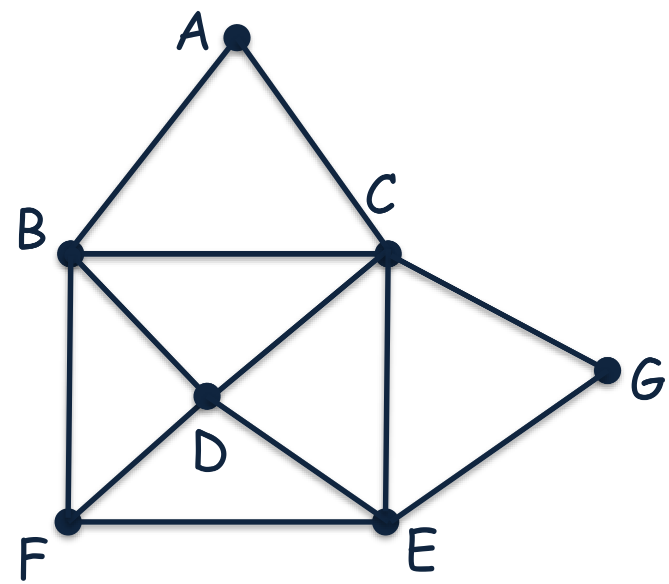
- when you pass a vertex, you add two to the degree of it.
- the degree of starting node and ending node just one or odd number
- the graph has a Euler path or Euler circuit if if it has no odd vertex or exactly two odd vertices.



Euler Paths and Circuits



F-B-A-C-B-D-F-E-D-C-E

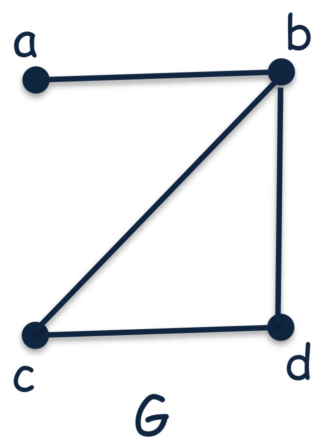


F-B-D-E-G-C-E-F-D-C-A-B-C

Hamilton Paths and Circuits

- Hamilton circuit is a simple circuit that contains every vertex of G exactly once except the starting vertex.

- Hamilton path is a simple circuit that contains every vertex of G exactly once



- Does G contain a Hamilton path or circuit ?

a – b – c – d

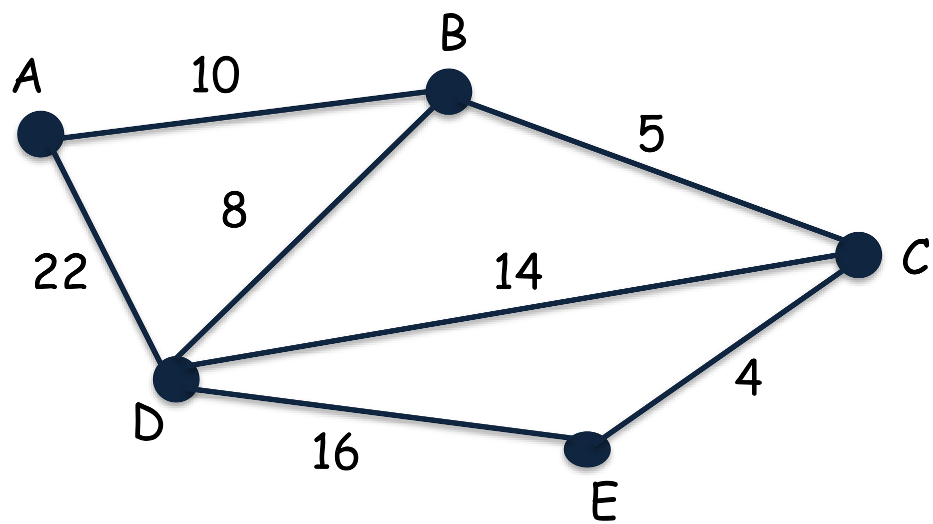
no Hamilton circuit

- There is no easy way to determine a given graph has a Hamilton circuit or Hamilton path

a graph with a vertex of degree one cannot have a Hamilton circuit

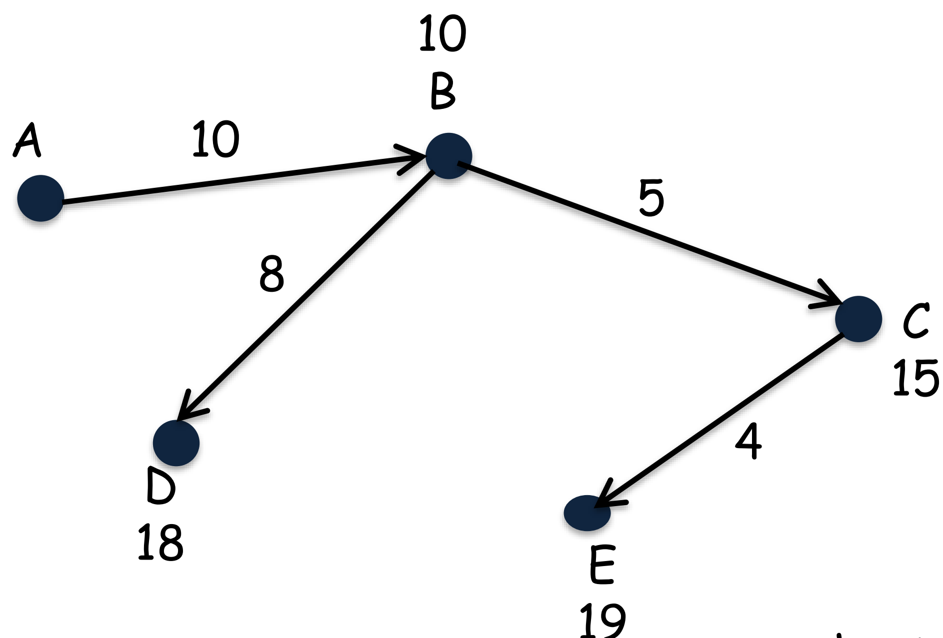
SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



shortest-paths tree

SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V

- Three cases :

  - the weight of each edge fixed as 1
    --BFS--

  - the weight of each edge non-negative
    --Dijkstra—

  - the weight of each can be negative
    --Belmann/Ford--

Relaxation

- For each vertex v in V, initialize two parameters :
  - parent pointer – indicates the predecessor of the vertex in the shortest path from s to v
  - distance – indicates the shortest-path estimate from vertex to the source

Initialize (G, s)

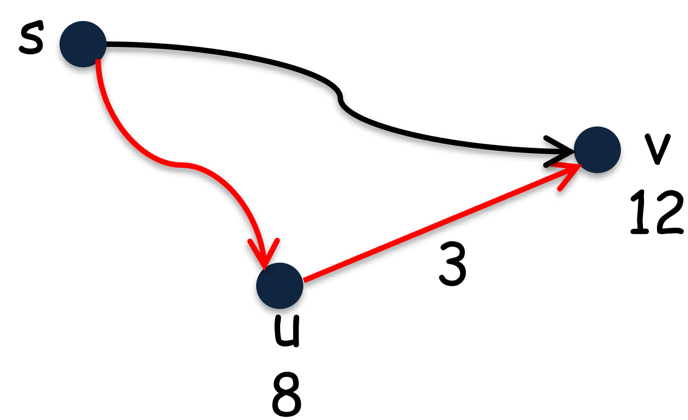
```
for each vertex v i V
    v.dis = ∞
    v.par = nil
s.dis = 0
```

Relaxation

- relaxing an edge (u,v) : testing whether the shortest path to the vertex v can be improved by going through the vertex u

Relax(u, v)

if v.dis > u.dis + w(u,v)
    v.dis = u.dis + w(u,v)
    v.par = u



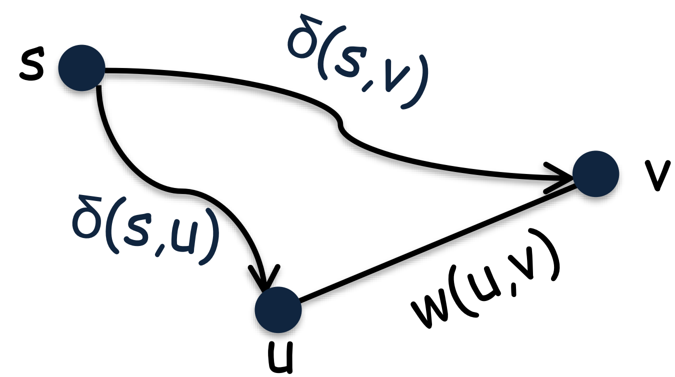
v.dis > u.dis + w(u,v) → v.dis = u.dis + w(u,v)
12 > 8 + 3   v.dis = 11
v.par = u

Relaxation

- Let $\delta(s,v)$ be the weight of the shortest path from source to the vertex v (after the termination of the program)

- For any edge (u,v) in E,

$$\delta(s,v) \le \delta(s,u) + w(u,v)$$



- For all vertices v in V,

$$v.dis \ge \delta(s,v)$$

- If there is no path from s to v, then

$$v.dis = \delta(s,v) = \infty$$

Dijkstra's Algorithm

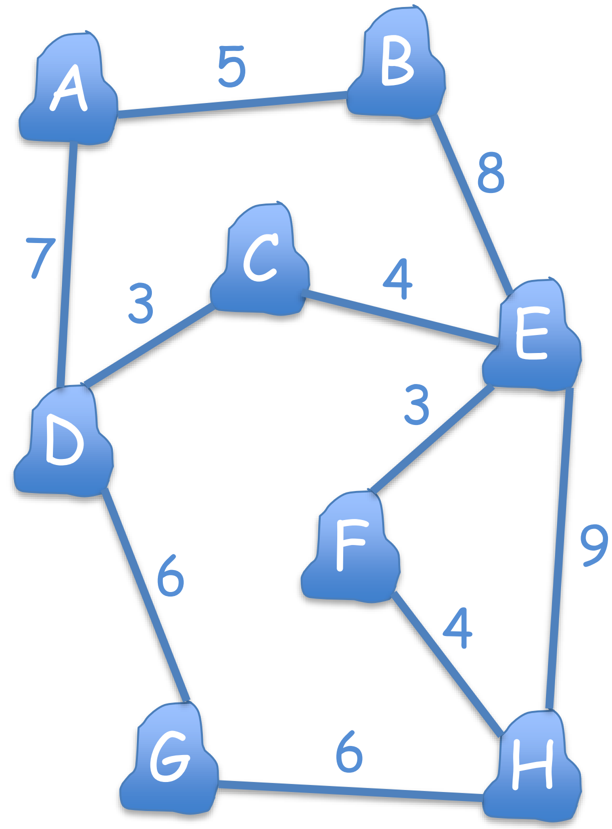
Dijkstra(G,s)

```
for each u of V                              Initialize(G,s)
      u.key = ∞                              O(|V|)
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V             O(|V|)
while Q ≠ { }
      u = ExtractMin(Q)                      O(|V|.log|V|)
      S = S ∪ {u}
      for each v of Adj(u)
            if v.dis > u.dis + w(u,v)        O(|E|.log|V|)
                v.dis = u.dis + w(u,v)
                v.par = u
           update Q
Relax(u,v)
O(1)
```

Dijkstra's Algorithm

Dijkstra(G,s)

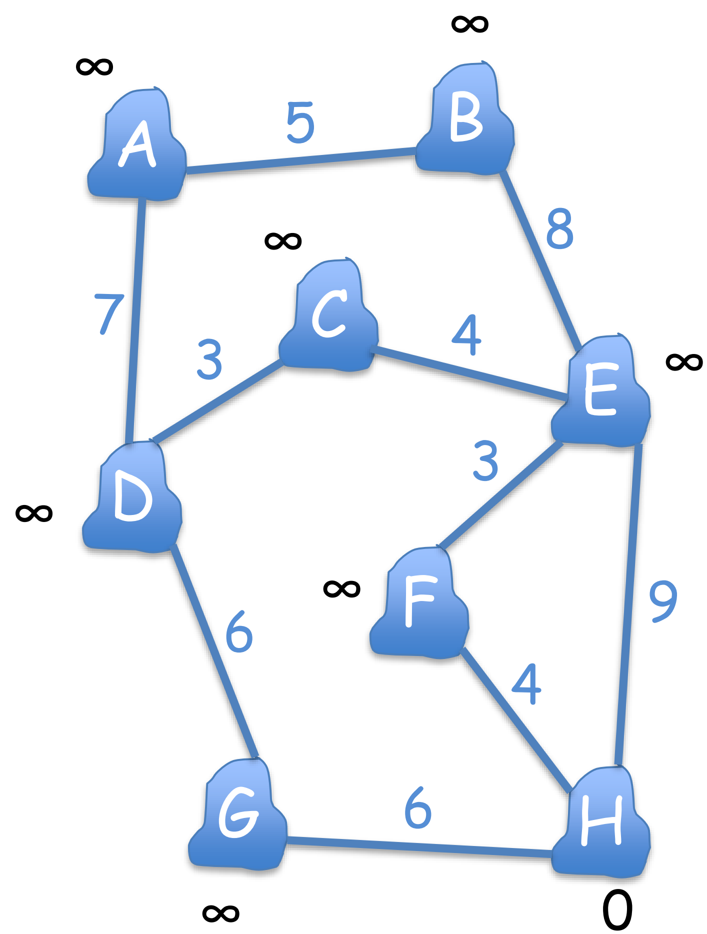
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```

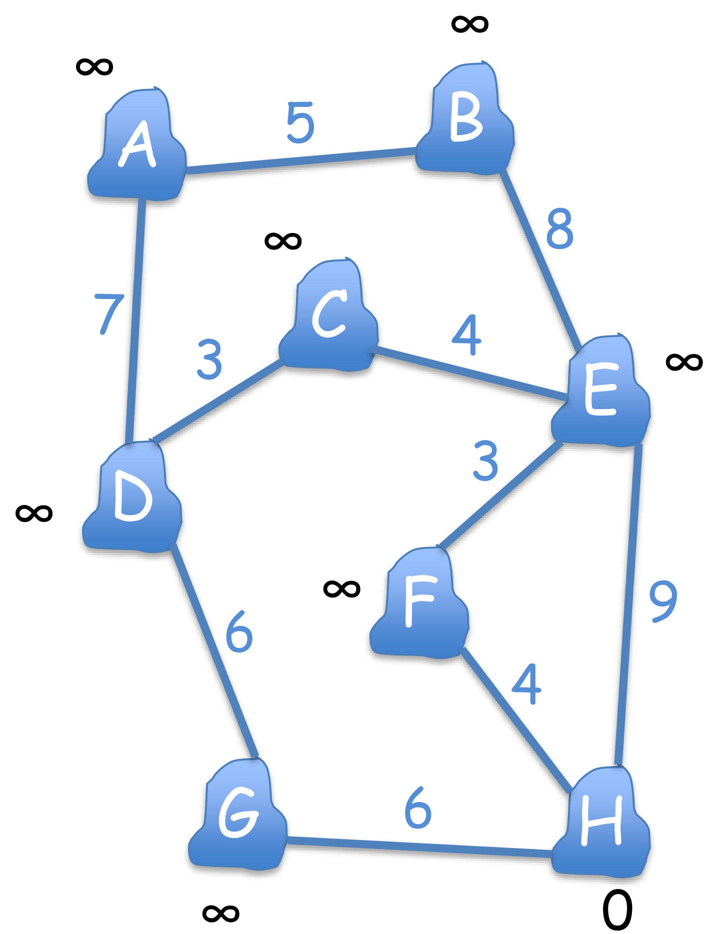


H F G E D C B A
S = { }

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



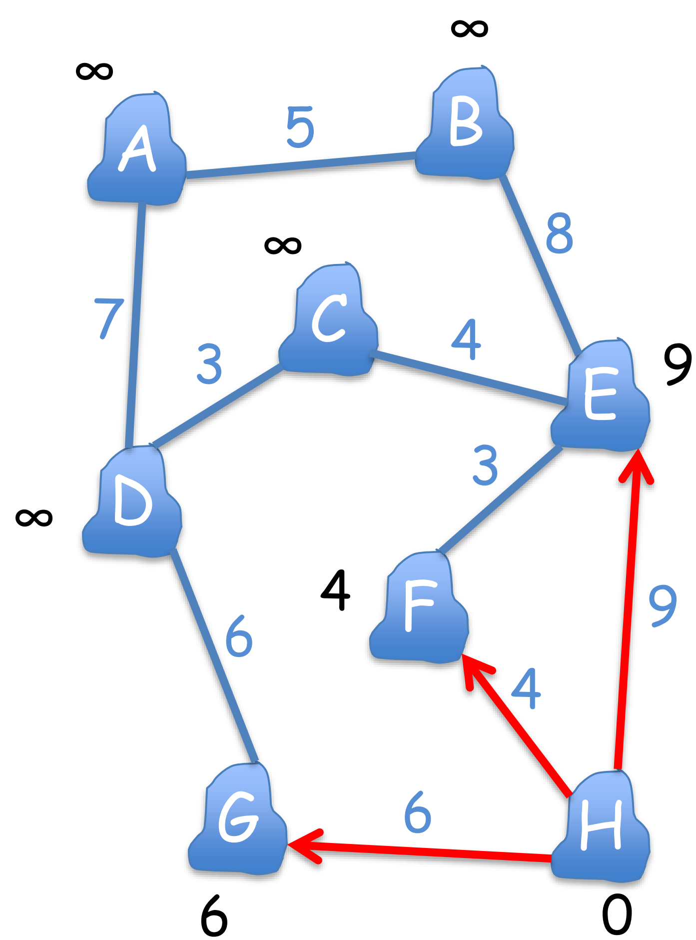
F G E D C B A

S = {H}

Dijkstra's Algorithm

```
Dijkstra(G,s)

for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
           update Q
```



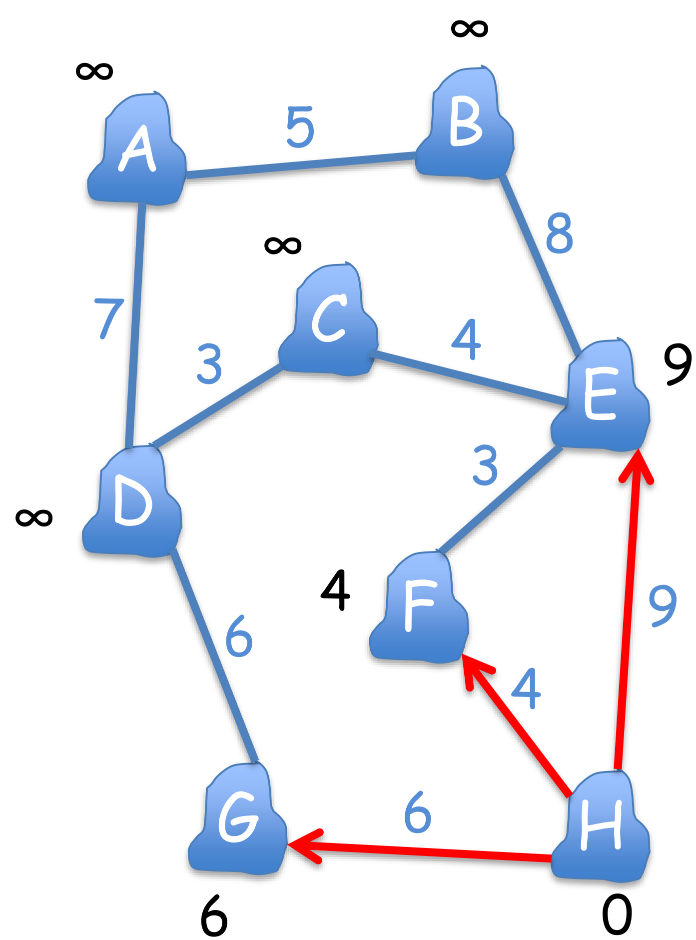
F G E D C B A

S = {H}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



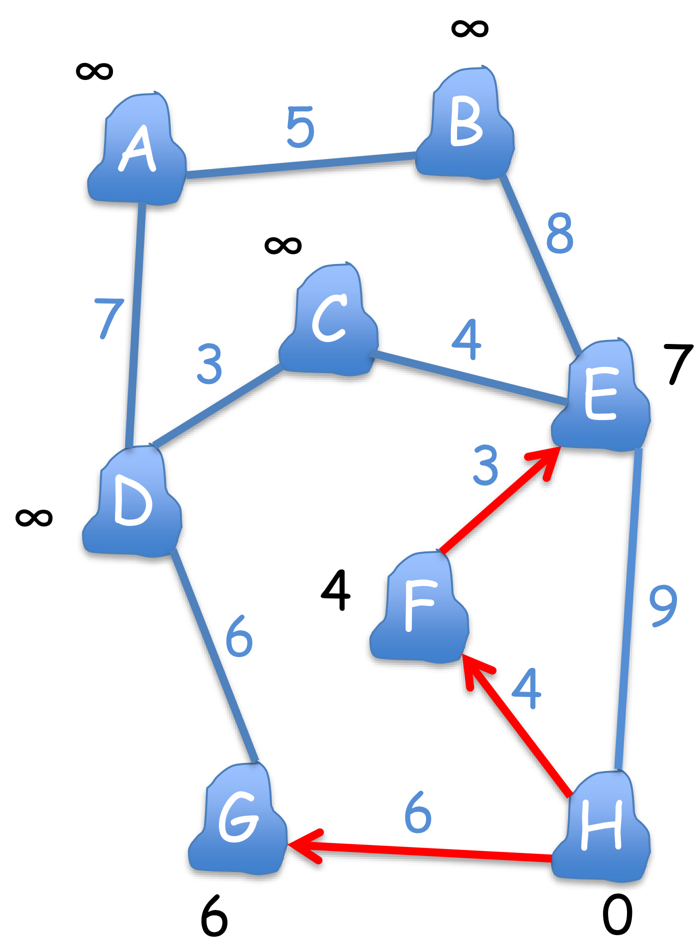
G E D C B A

S = {H,F}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



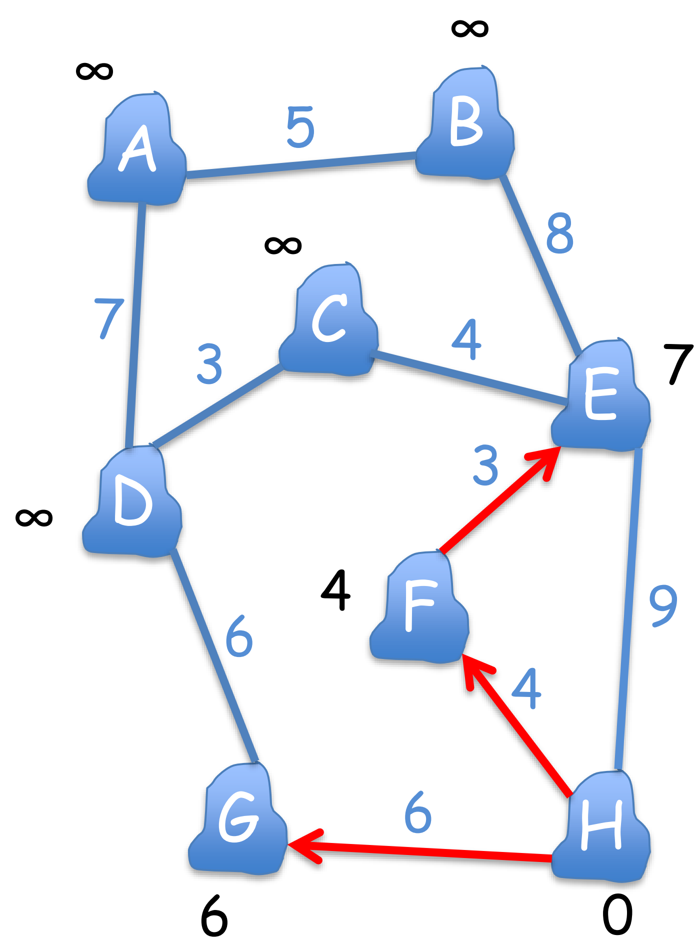
G E D C B A

S = {H,F}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```



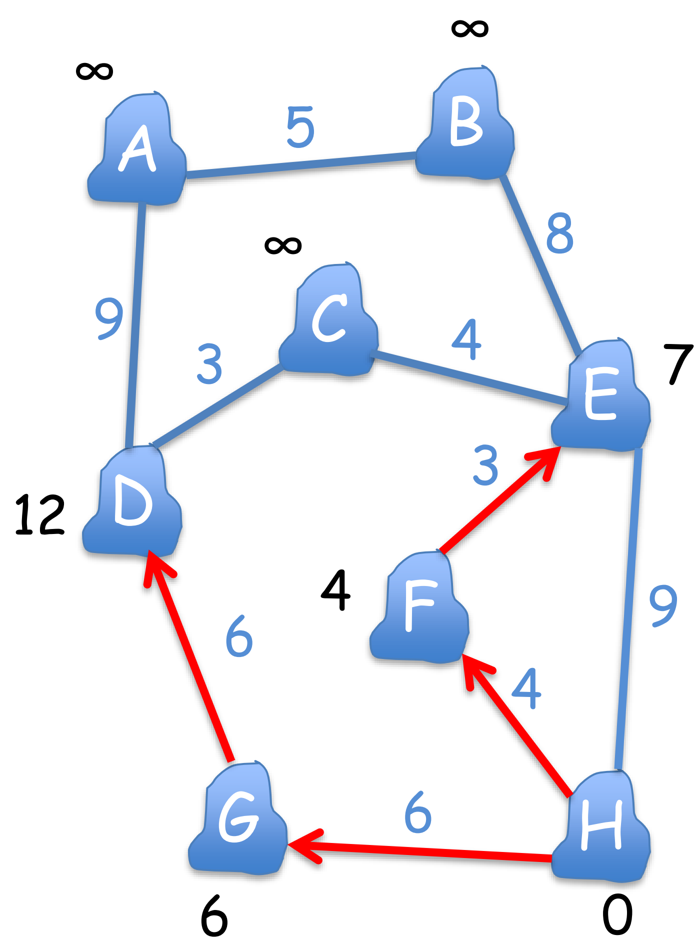
E D C B A

S = {H,F,G}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



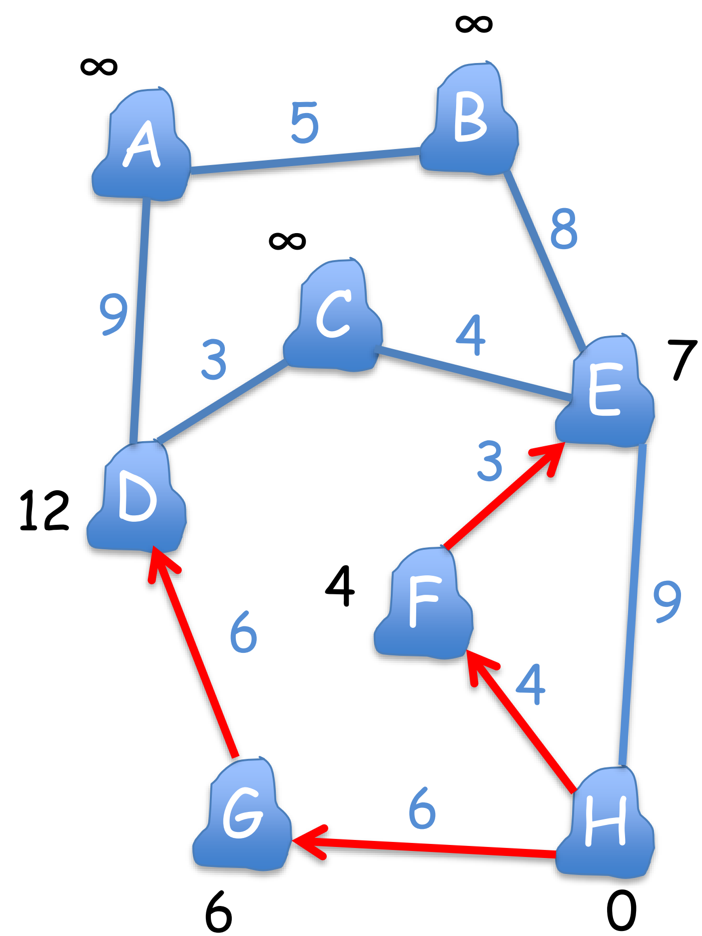
E D C B A

S = {H,F,G}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



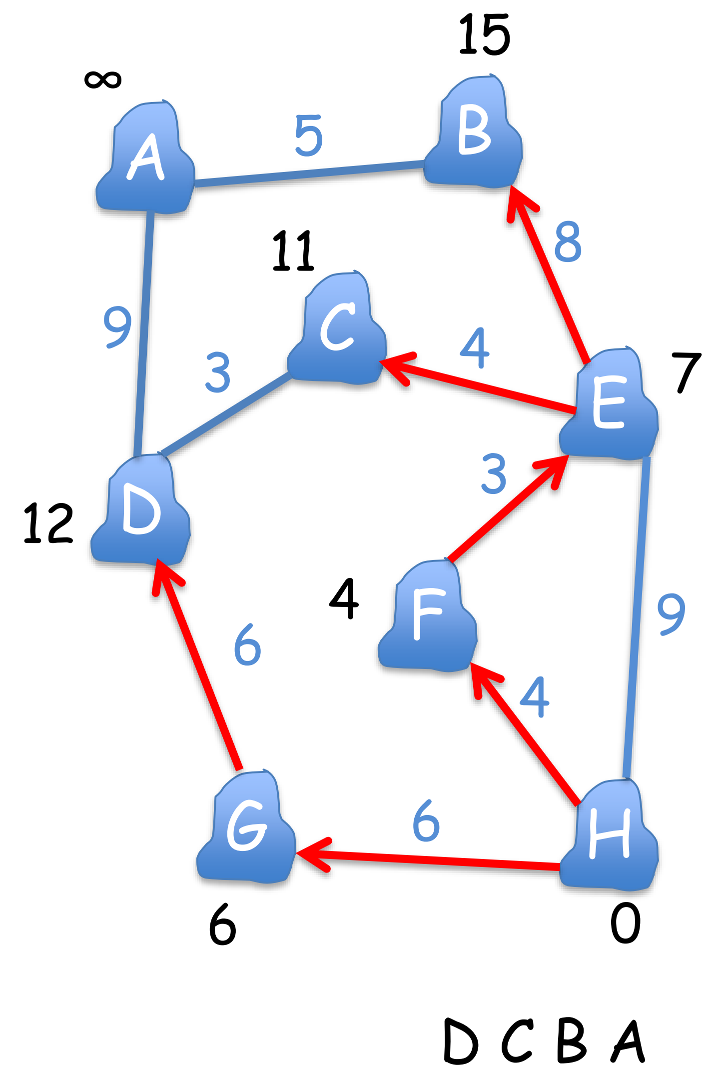
D C B A

S = {H,F,G,E}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```

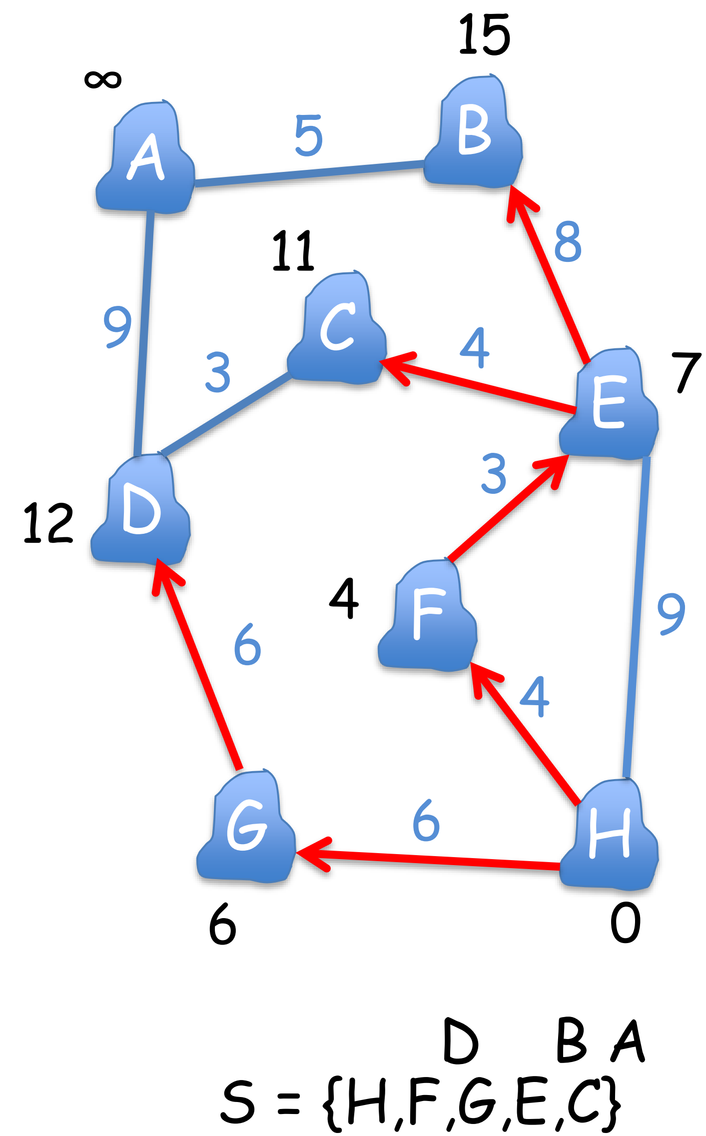


S = {H,F,G,E}

Dijkstra's Algorithm

Dijkstra(G,s)

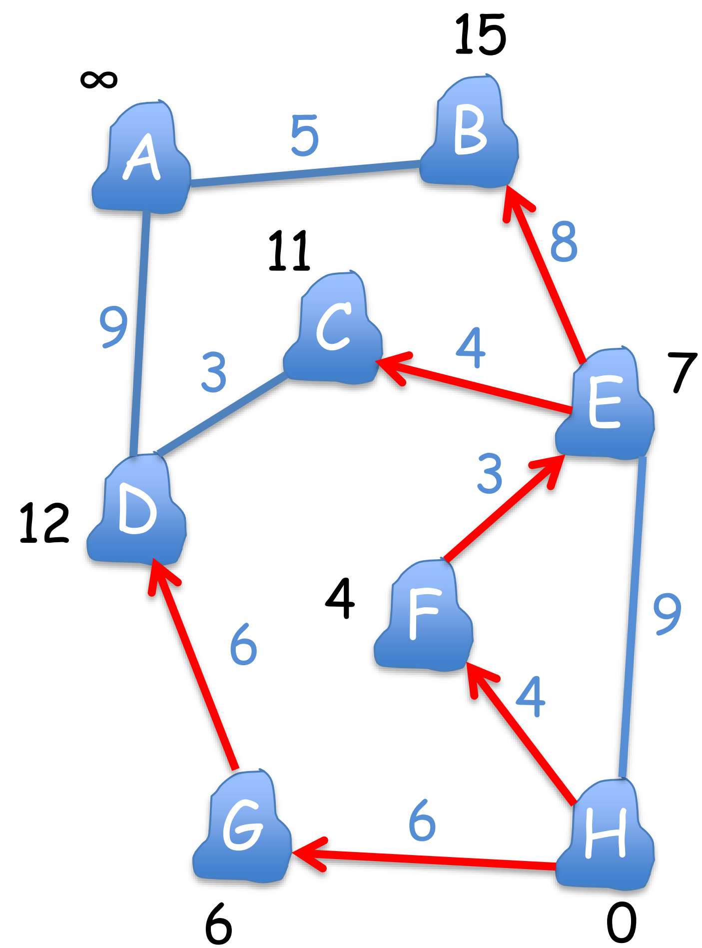
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
        update Q



Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



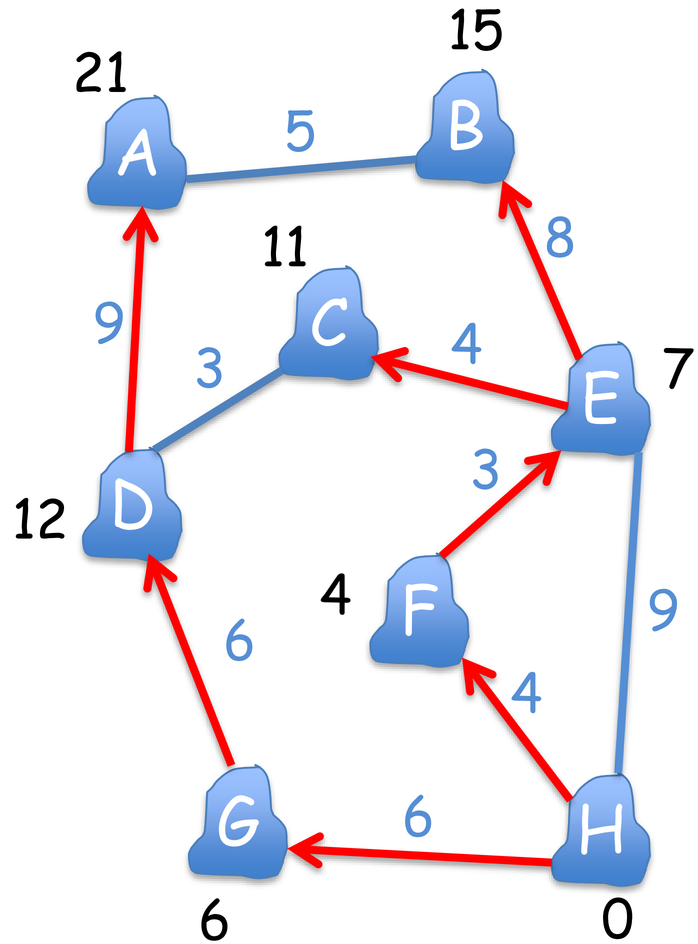
B A

S = {H,F,G,E,C}

Dijkstra's Algorithm

```
Dijkstra(G,s)

for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



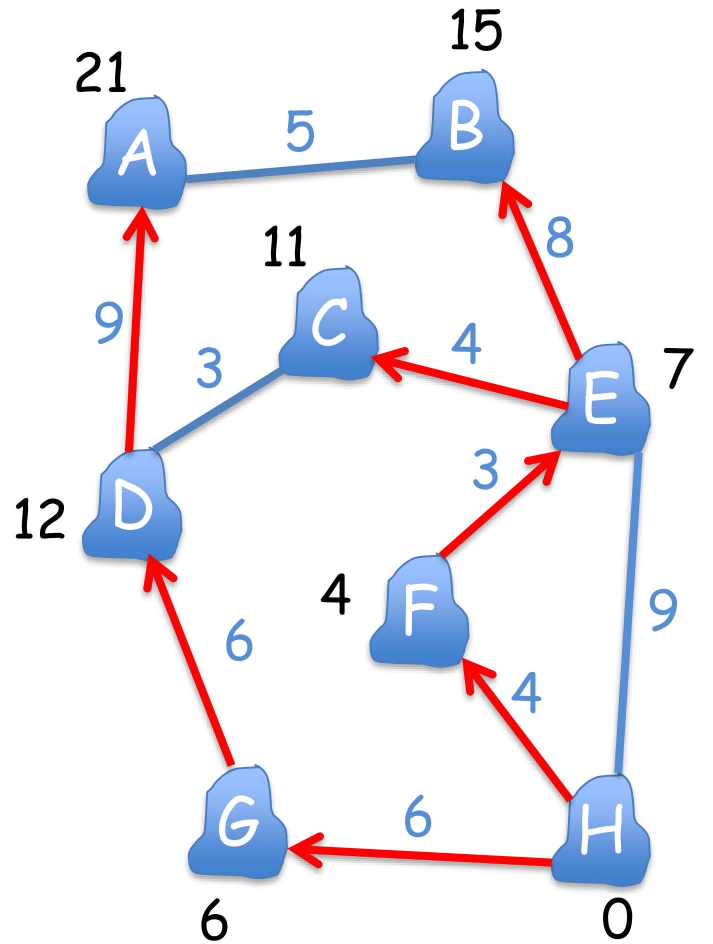
B A

S = {H,F,G,E,C,D}

Dijkstra's Algorithm

Dijkstra(G,s)

for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
        update Q

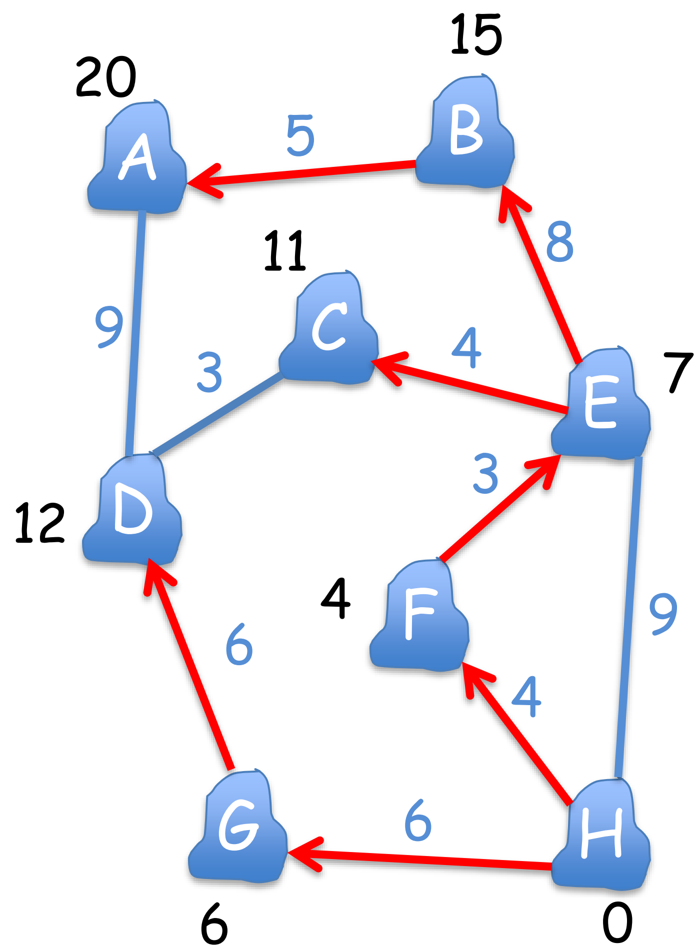


A
S = {H,F,G,E,C,D,B}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```

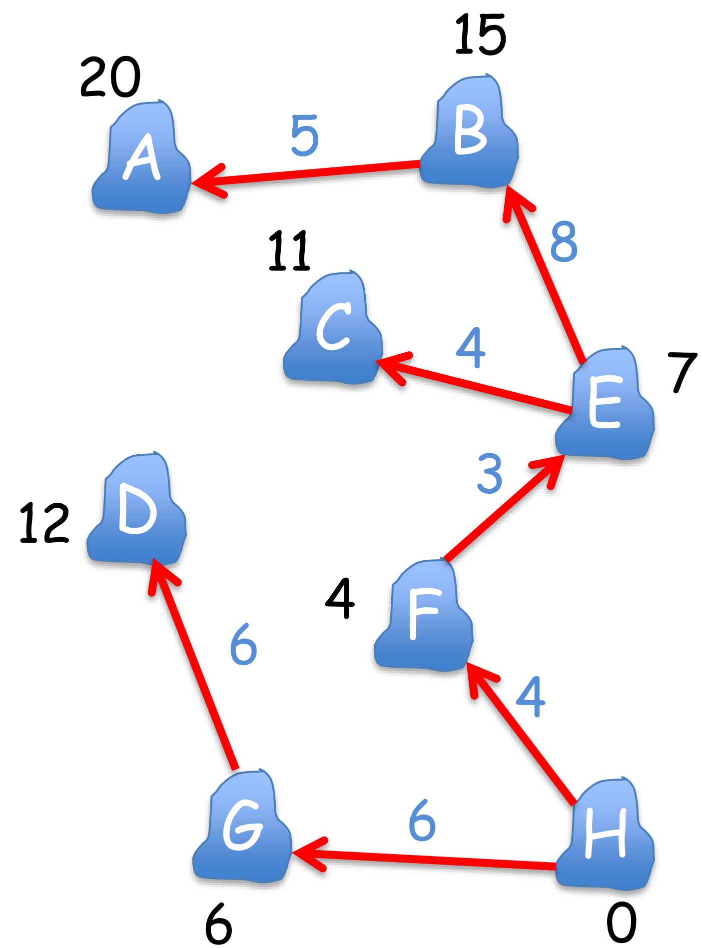


A
S = {H,F,G,E,C,D,B}

Dijkstra's Algorithm

Dijkstra(G,s)

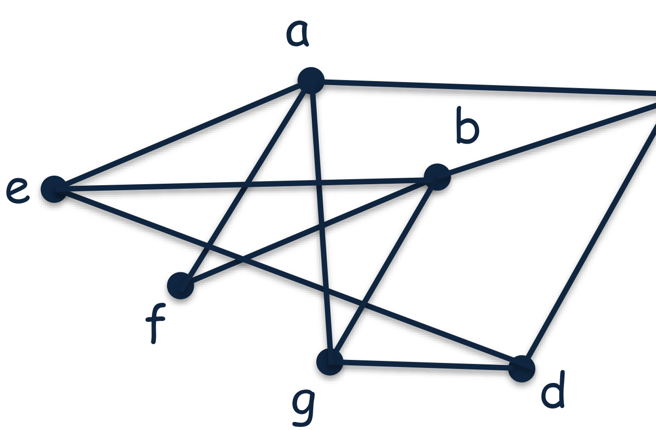
```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if  v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
     update Q
```



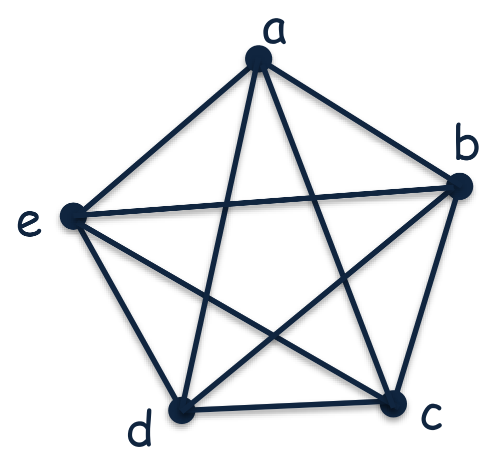
S = {H,F,G,E,C,D,B,A}

Bipartite Graphs

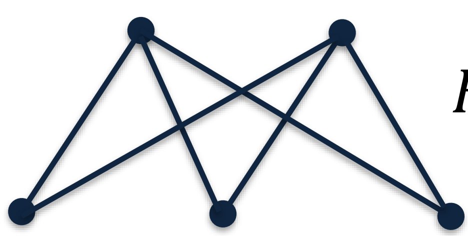
- a simple graph G is called bipartite if its vertex set V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex in V_2 (there is no edge (a,b) such that a and b are elements of same partition)



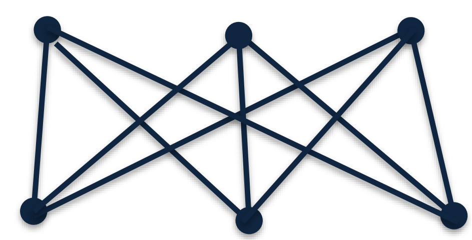
G



K_5



$K_{2,3}$

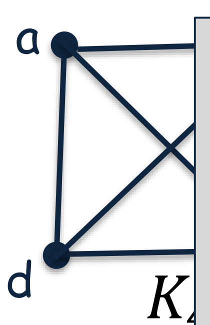


$K_{3,3}$

Planar Graphs

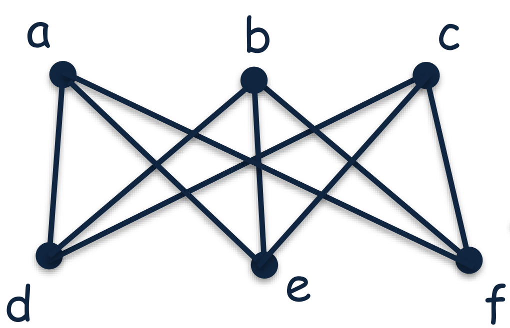
- a graph G is called planar if it can be drawn in the plane without any edge crossing.

this drawing is called planar representation of the graph

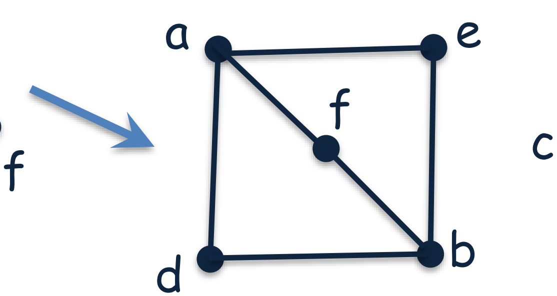


Euler Formula : Let G be connected simple graph with e edges and v vertices. Let r be the number of region in a planar representation of G. Then,

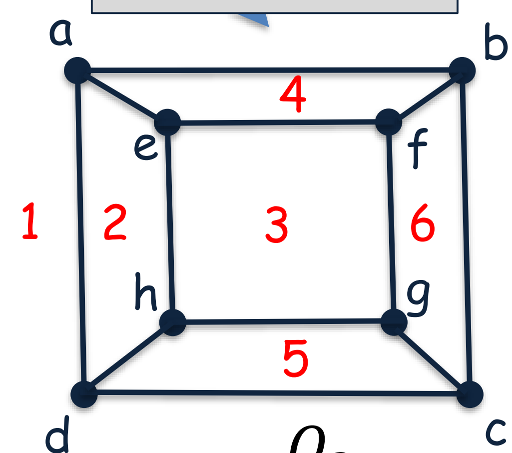
$$r = e - v + 2$$



$K_{3,3}$



$6 = 12 - 8 + 2$



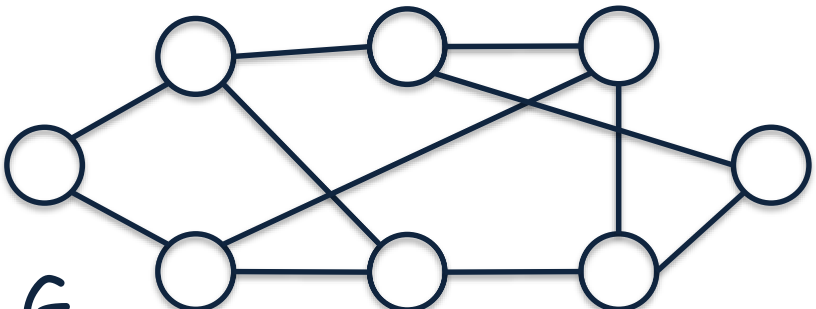
Q_3

- $K_{3,3}$ cannot be drawn as planar graph

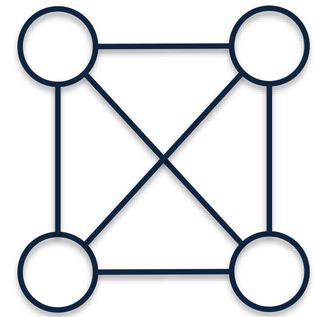
Graph Coloring

Graph Coloring

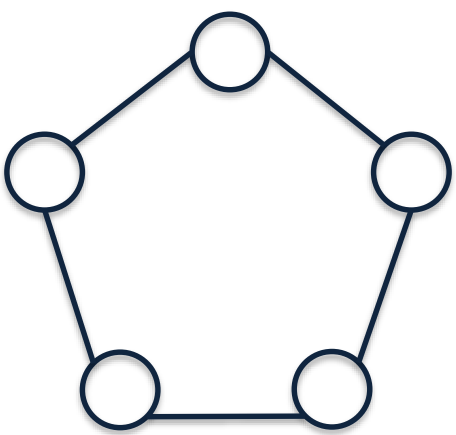
- a coloring of a simple graph is the assignment of a color to each vertex so that no two adjacent vertices are assigned the same color



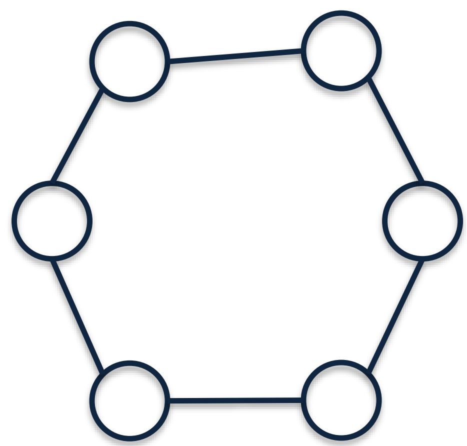
G



K_5



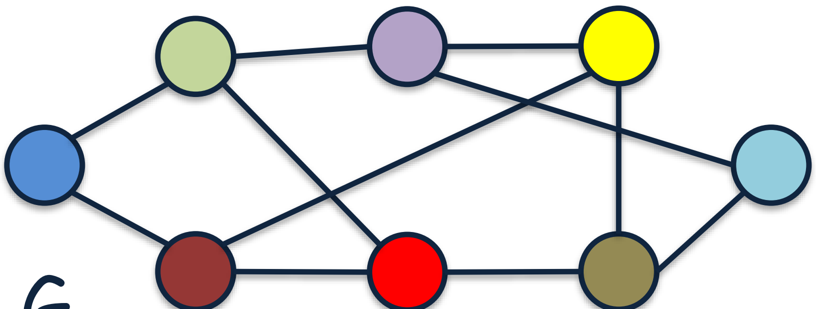
C_5



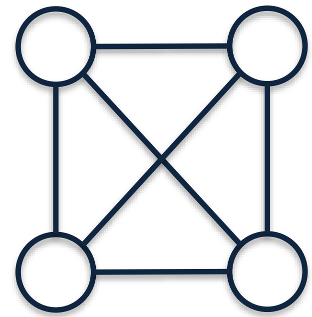
C_6

Graph Coloring

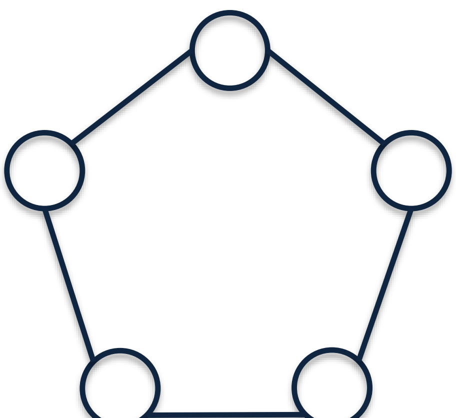
- a coloring of a simple graph is the assignment of a color to each vertex so that no two adjacent vertices are assigned the same color



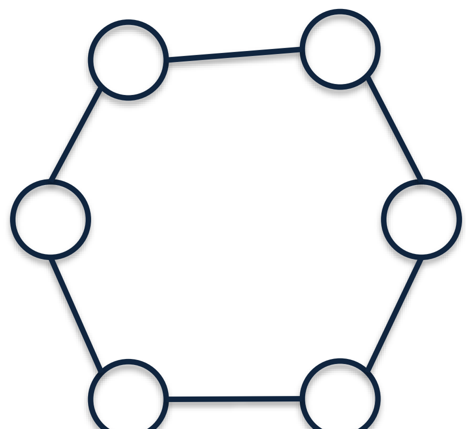
G



K_5



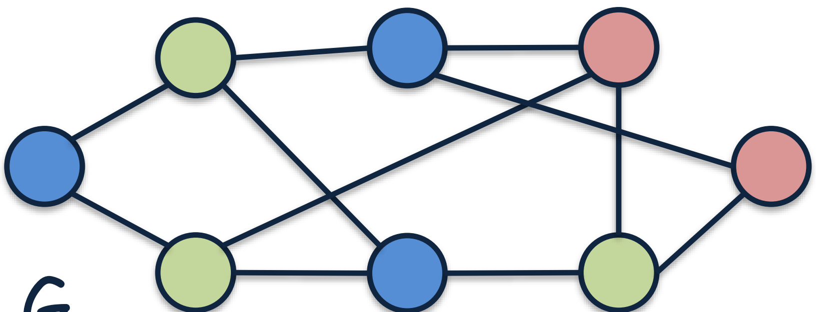
C_5



C_6

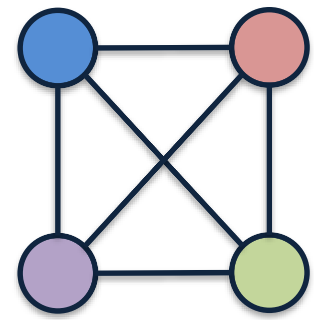
Graph Coloring

- a coloring of a simple graph is the assignment of a color to each vertex so that no two adjacent vertices are assigned the same color



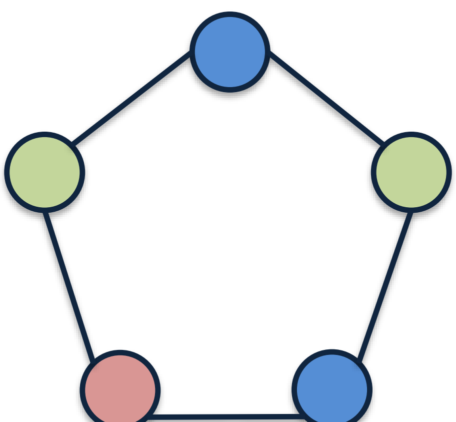
G

$\chi(G) = 3$ (chromatic number)

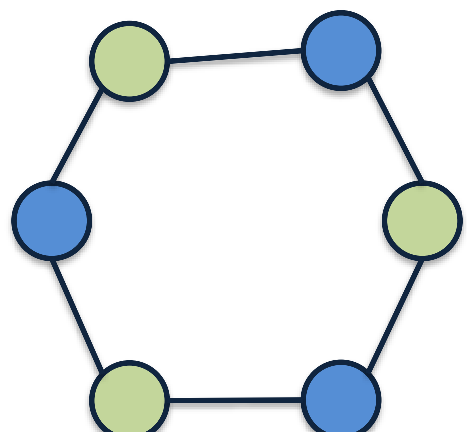


K_4

$\chi(K_4) = 4$



C_5 $\chi(C_5) = 3$



C_6 $\chi(C_6) = 2$