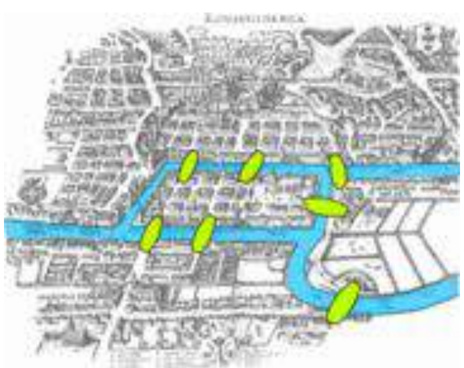
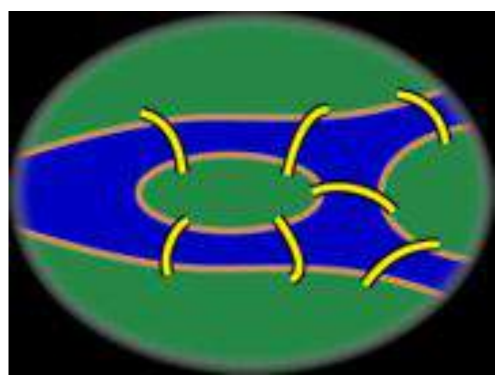
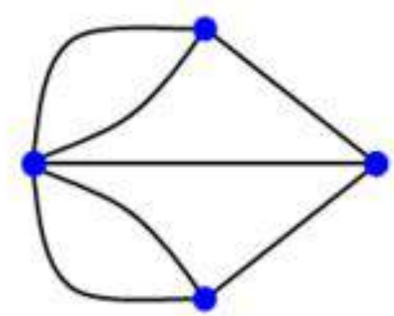
Graph Algorithms

Graph Theory - History

Leonhard Euler's paper on
"*Seven Bridges of Königsberg*",
published in 1736.







Famous problems

- "The traveling salesman problem"
  - A traveling salesman is to visit a number of cities; how to plan the trip so every city is visited once and just once and the whole trip is as short as possible ?

Famous problems

In 1852 Francis Guthrie posed the "four color problem" which asks if it is possible to color, using only four colors, any map of countries in such a way as to prevent two bordering countries from having the same color.

This problem, which was only solved a century later in 1976 by Kenneth Appel and Wolfgang Haken, can be considered the birth of graph theory.

Examples

- Cost of wiring electronic components
- Shortest route between two cities.
- Shortest distance between all pairs of cities in a road atlas.
- Matching / Resource Allocation
- Task scheduling
- Visibility / Coverage

Examples

- Flow of material
  - liquid flowing through pipes
  - current through electrical networks
  - information through communication networks
  - parts through an assembly line
- In Operating systems to model resource handling (deadlock problems)
- In compilers for parsing and optimizing the code.

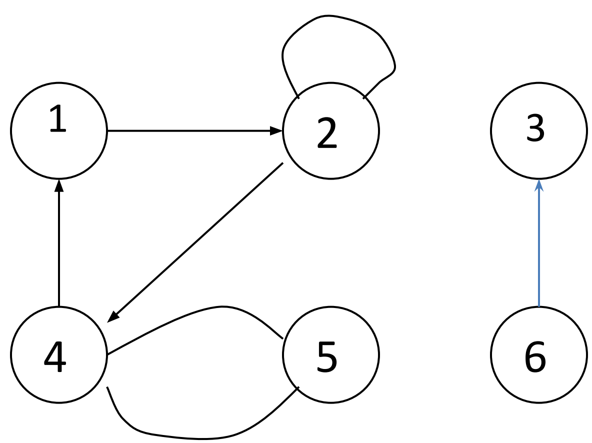
Basics

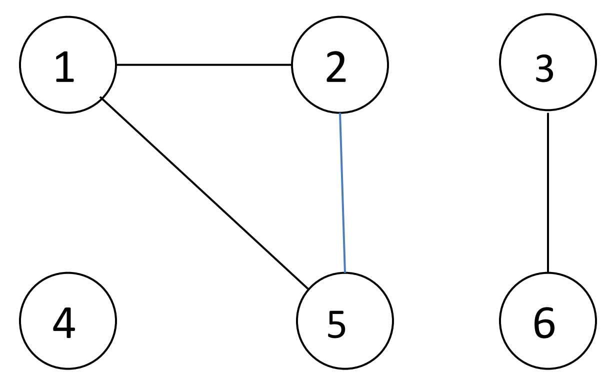
Graph Definitions and Notation

A **graph** G is a triple (V, E, g), where

(i) V is a finite nonempty set, called the **set of vertices**;
(ii) E is a finite set (may be empty), called the **set of edges**; and
(iii) g is a function, called an **incidence function**, that assigns to each edge, $e \in E$, a one-element subset $\{v\}$ or a two-element subset $\{v, w\}$, where v and w are vertices.

For convenience, we will write $g(e) = \{v, w\}$, where v and w may be the same.





Definitions

- Vertex
  - Basic Element
  - Drawn as a *node* or a *dot*.
  - V**ertex set** of G is usually denoted by $V(G)$, or V
- Edge
  - A set of two elements
  - Drawn as a line connecting two vertices, called end vertices, or endpoints.
  - The edge set of G is usually denoted by E(G), or E.

Graph Definitions and Notation

- Let $G=(V,E, g)$ be a graph and e be an edge of this graph.
- Then there are vertices v and w such that $g(e) =\{v,w\};$ vertices v and w are ***end vertices*** of e
- When a vertex v is an end point of some edge e, we say that e is an ***incident*** with the vertex v and v is incident with the edge e.
- Two vertices v and w of G are said to be ***adjacen***t if there exists an edge $e \in E$ such that $g(e)=\{v,w\}$.

Graph Definitions and Notation

- If ***e*** is an edge such that ***g(e)*** = ***{v,w}***, where ***v***=***w***, then ***e*** is an edge from the vertex ***v*** to itself. Such an edge is called a ***loop*** on the vertex ***v*** or at the vertex ***v***.
- If there is a loop on ***v***, then ***v*** is adjacent to itself.

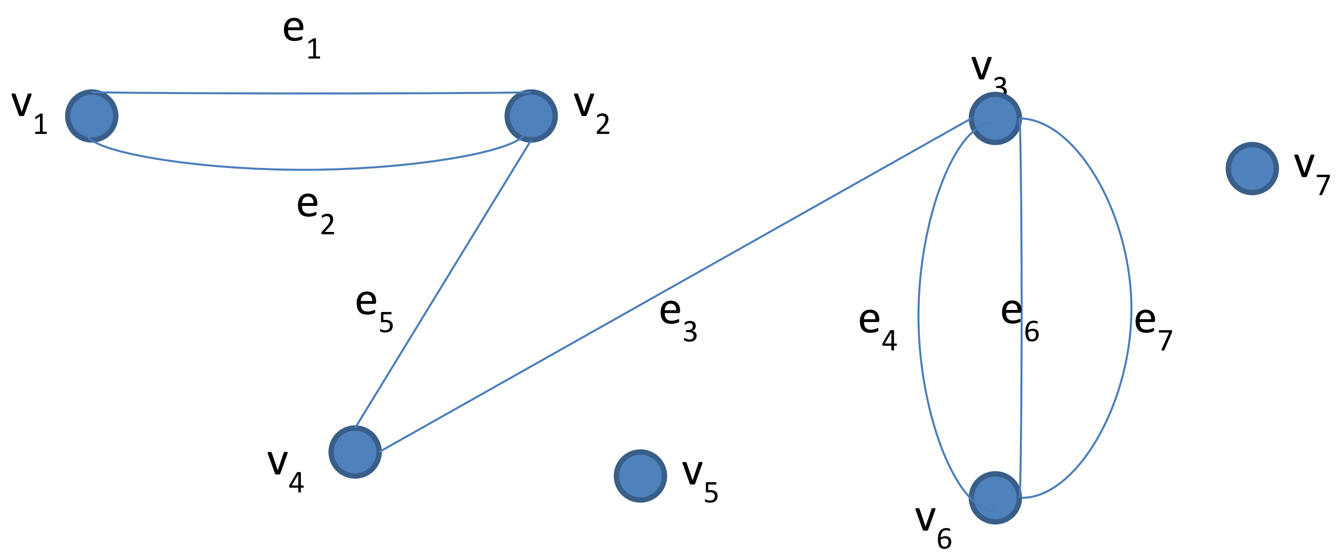
Let $G = (V, E, g)$ be a graph. If no confusion arises, we will write G as (V, E), or simply as G.

Graph Definitions and Notation

Let $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, and g be defined by

$$g(e_1) = g(e_2) = \{v_1, v_2\}$$
$$g(e_3) = \{v_4, v_3\}$$
$$g(e_4) = g(e_6) = g(e_7) = \{v_6, v_3\}$$
$$g(e_5) = \{v_2, v_4\}.$$

Then $G = (V, E, g)$ is a graph.

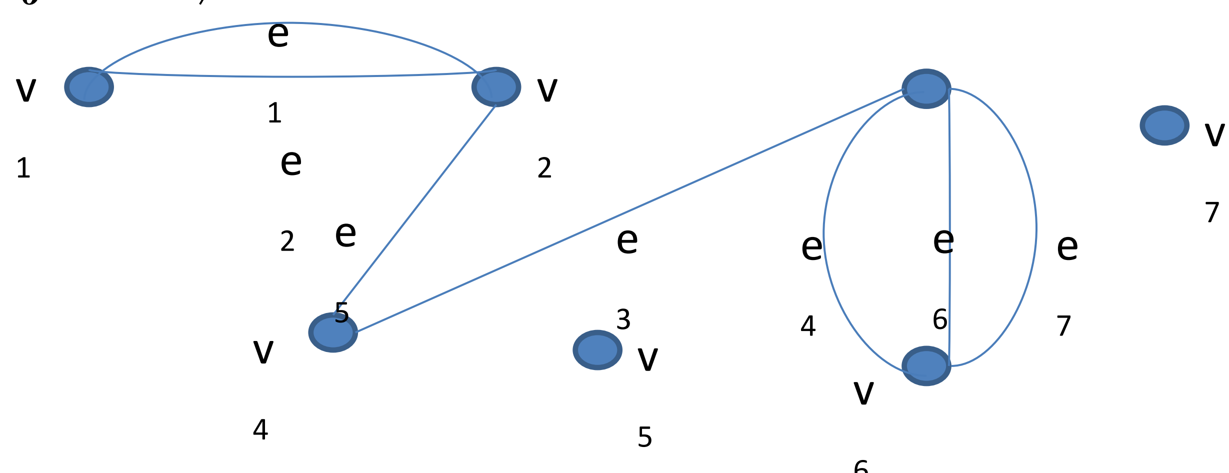


Graph Definitions and Notation

- Let $G=(V, E, g)$ be a graph.
- The incidence function g need not be one – to – one.
- Therefore, there may exist edges $e_1, e_2, \ldots, e_{n-1}, e_n$, $n \geq 2$ such that $g(e_1)= g(e_2) = \ldots\ldots = g(e_n) = \{v, w\}$.
- Such edges are called ***parallel edges***.

Example:

$g(e_1)=g(e_2)=\{v_1, v_2\}$. So edges e_1 and e_2 are parallel. Similarly edges e_4, e_6 and e_7 are parallel.



Graph Definitions and Notation

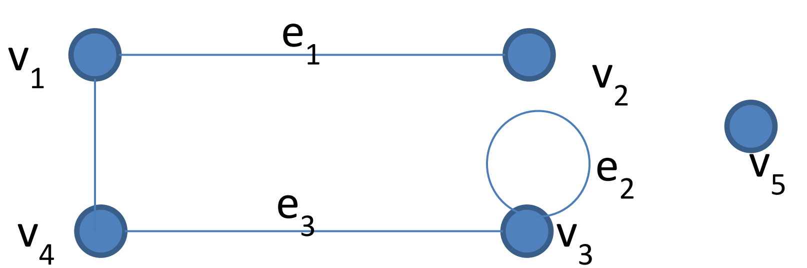
DEFINITION :: Isolated vertex

- Let G be a graph and v be a vertex in G. We say that v is an isolated vertex if it is not incident with any edge.

DEFINITION :: Degree of a vertex

- Let G be a graph and ***v*** be a vertex of ***G***. The degree of ***v***, written as ***deg(v)*** or ***d(v)*** is the number of edges incident with v.
- We make the convention that each loop on a vertex ***v*** contributes 2 to the degree of ***v.***

Example:



$deg(v_1)=2$
$deg\ (v_2)= 1$
$deg\ (v_3)=3$
$deg(v_4)=2$

$deg(v_5)=0$

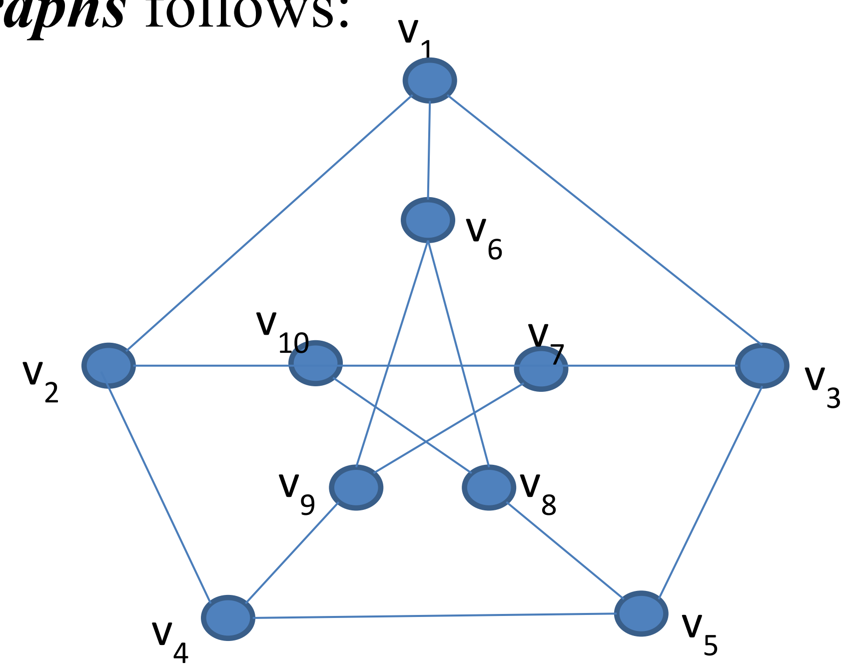
Graph Definitions and Notation

DEFINITION:: k-regular graph

- Let G be graph and ***k*** be a nonnegative integer.
- G is called a ***k-regular graph*** if the ***degree of each vertex*** of G is ***k***

Example:

- An interesting k-regular graph is the ***Petersen 3-regular graphs*** follows:

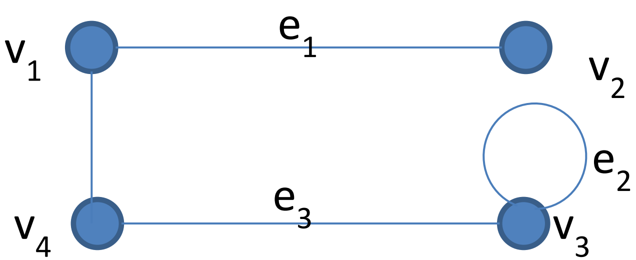


Graph Definitions and Notation

DEFINITION :: Even (odd) degree vertex

- Let G be a graph and v be a vertex in G.
- v is called an ***even (odd)*** vertex if the degree of v is ***even (odd)***

Example::



deg(v_1)=2 Even
deg (v_2)= 1 Odd
deg (v_3)=3 Odd
deg(v_4)=2 Even

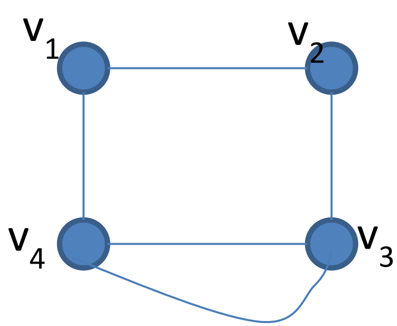
Graph Definitions and Notation

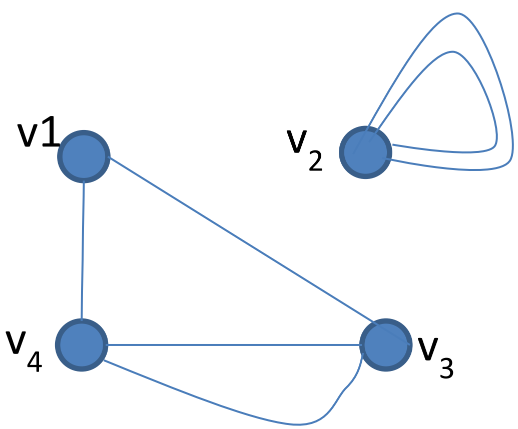
DEFINITION :: Degree Sequence

- Let $n_1, n_2, n_3, \ldots, n_k$ be the degrees of vertices of a graph G such that $n_1 \leq n_2 \leq n_3 \leq \ldots \leq n_k$.
- Then the finite sequence $n_1, n_2, n_3, \ldots, n_k$ is called the ***degree sequence of the graph***
- Every graph has a ***unique degree sequence***.
- However, we can construct completely different graphs having same degree sequence.

Example::

- The degree sequence of both of these graphs is 2, 3, 3, 4. But the graphs are different.





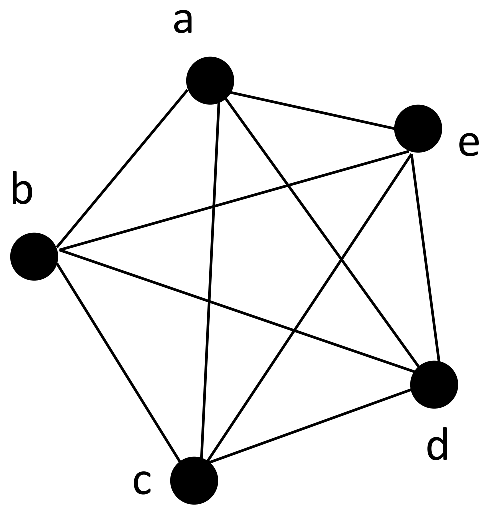
Some Theorems

- **Theorem 1:** ***Euler:*** The sum of the degrees of all vertices of a graph is twice the number of edges.
- **Corollary:** The sum of the degrees of all the vertices of a graph is an even integer.
- **Corollary:** In a graph the number of odd degree vertices is even.

Graph Definitions and Notation

DEFINITION :: Simple graph

A graph is called a simple graph if G does not contain *any parallel edges and any loop*

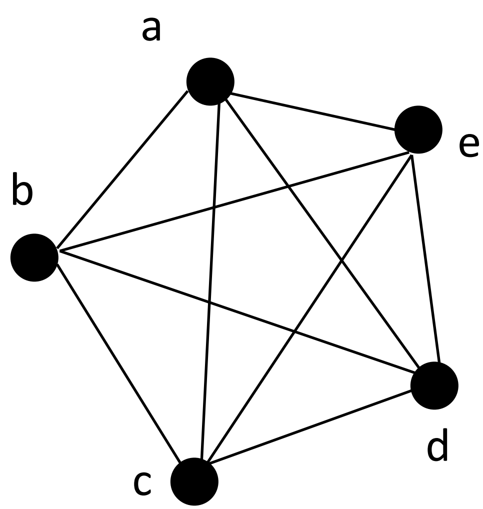


Graph Definitions and Notation

DEFINITION :: Complete Graph

A simple graph with *n* vertices in which there is an *edge between every pair of distinct vertices* is called a complete graph on *n* vertices. This is denoted as $K_{n.}$

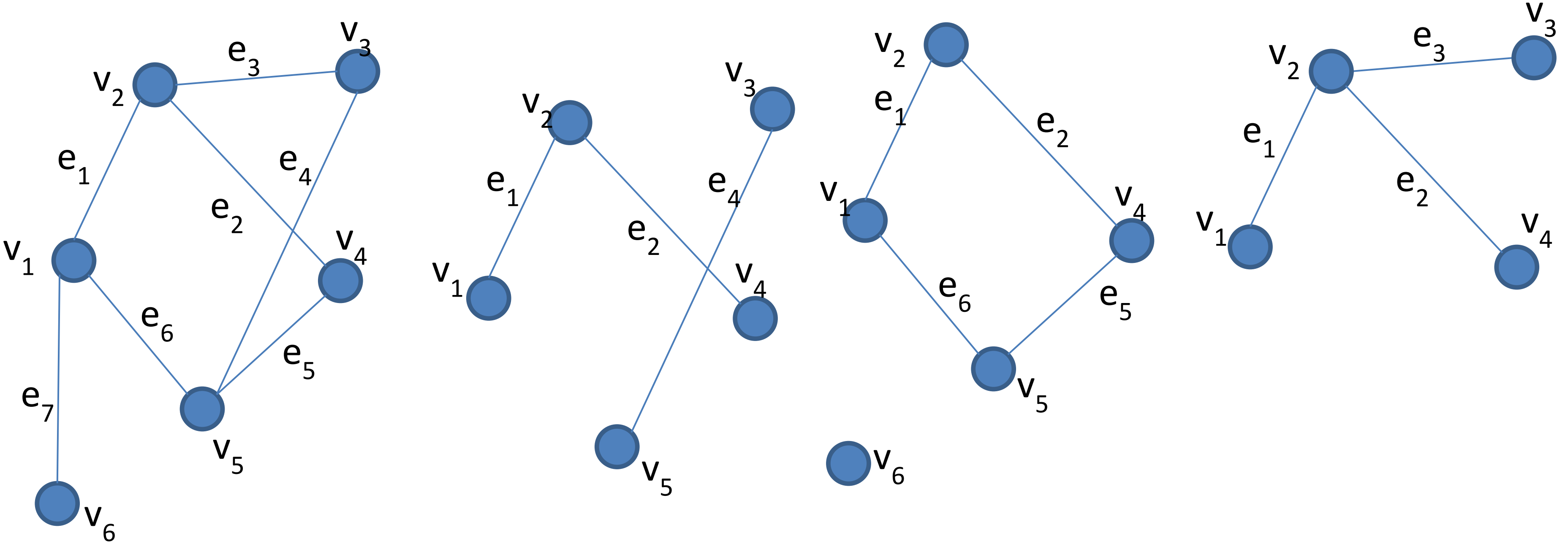
Example:



K_5

Theorem: The number of edges in a complete graph with n vertices is n(n-1)/2

Graph Definitions and Notation



G **G1** **G2** **G3**

DEFINITION :: Subgraph

Let *G= (V, E, g)* be a graph. A triple $G_1=(V_1, E_1, g_1)$ is called a subgraph of G if V_1 *is nonempty subset* of *V*, E_1 is a subset of E, and g_1 is the restriction of *g* to E_1 such that for all $e \in E_1$, if $g_1(e)= g(e) = \{u, v\}$, then $u,v \in V_1$.

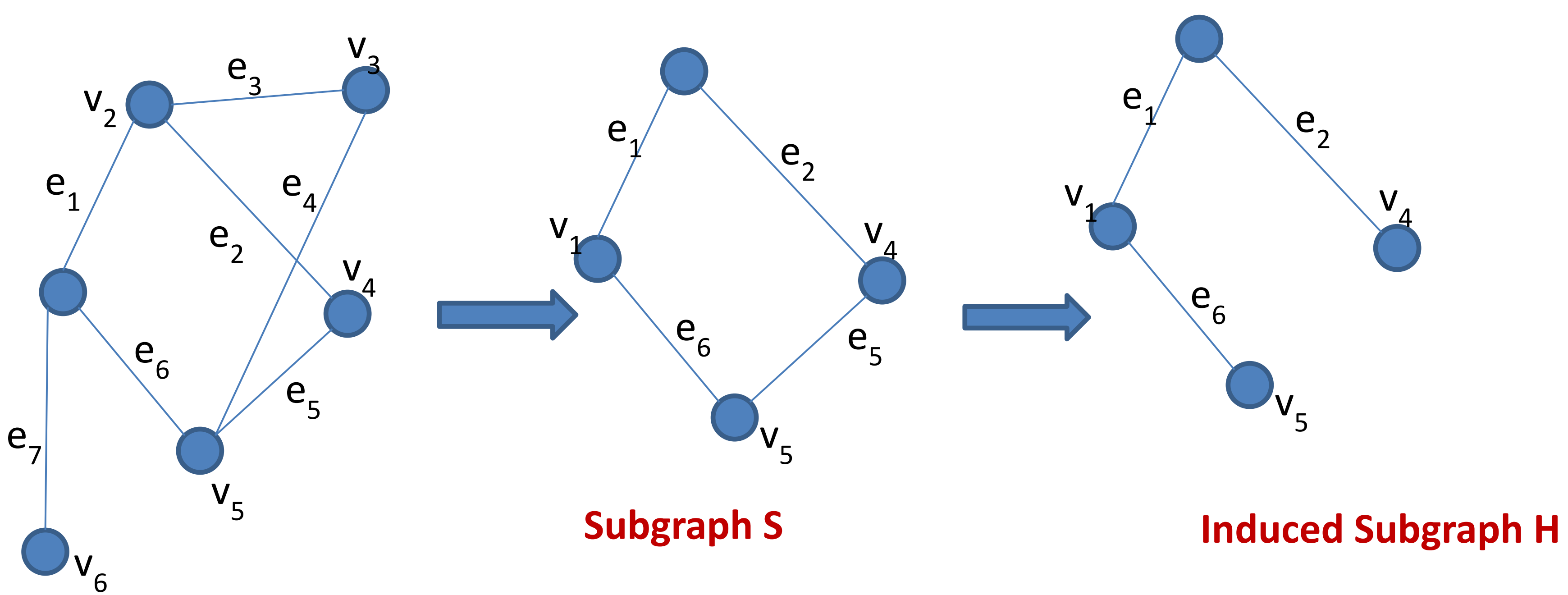
Sub-graph

- Few observations:
  - Every graph is its own sub-graph
  - A sub-graph of a sub-graph of G is a sub-graph of G
  - A single vertex in a graph G is a sub-graph of G
  - A single edge in G, together with its vertices, is also a sub-graph of G

Graph Definitions and Notation

DEFINITION :: Induced Subgraph

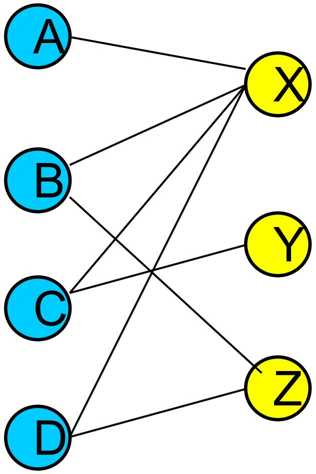
H is an induced subgraph of G on S, where S is subset of V(G); then V(H) =S and E(H) is the set of edges of G such that both the end points belong to S.



Graph Definitions and Notation

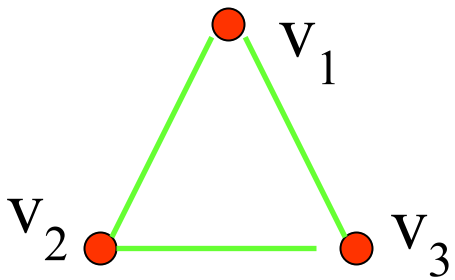
DEFINITION :: Bipartite Graph

A simple graph $\boldsymbol{G}$ is called a bipartite graph if the vertex set $\boldsymbol{V}$ of $\boldsymbol{G}$ can be partitioned into Nonempty subsets $\boldsymbol{V_1}$ and $\boldsymbol{V_2}$ such that each edge of G is incident with one vertex in $\boldsymbol{V_1}$ and one vertex in $\boldsymbol{V_2}$. $\boldsymbol{V_1} \cup \boldsymbol{V_2}$ is called a bipartition of **G**.



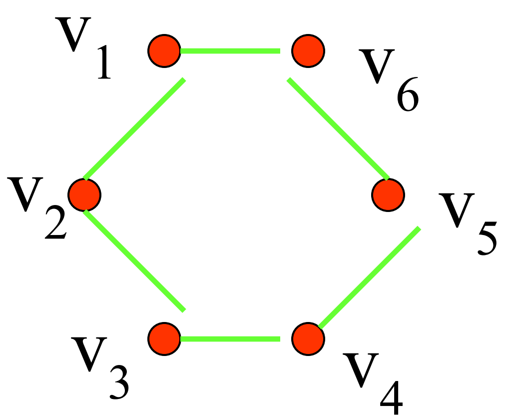
Bipartite Graphs

Example I: Is C_3 bipartite?

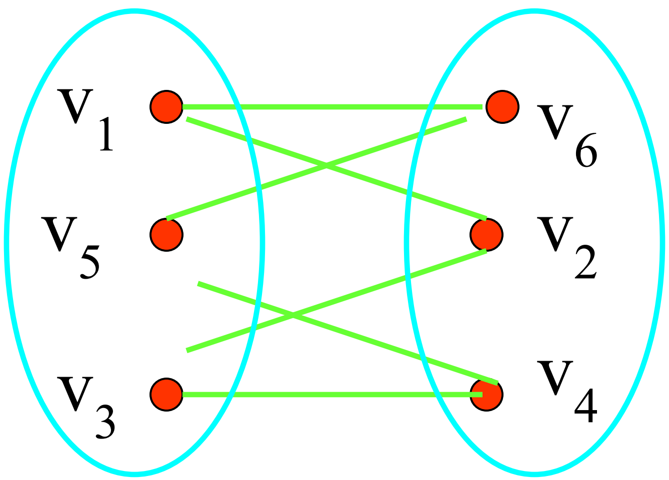


No, because there is no way to partition the vertices into two sets so that there are no edges with both endpoints in the same set.

Example II: Is C_6 bipartite?



Yes, because we can display C_6 like this:

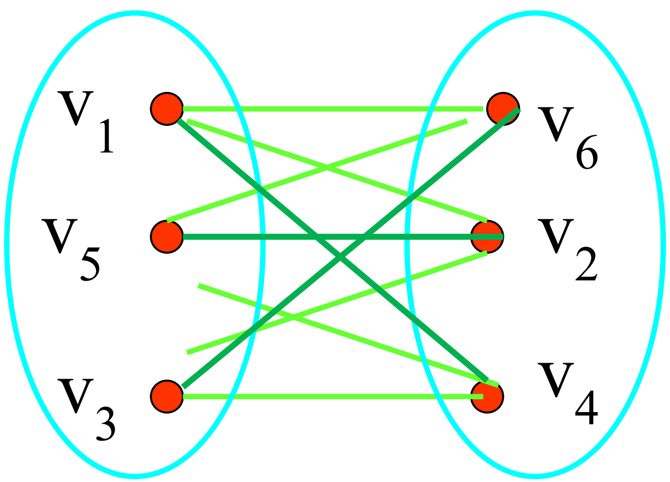


Graph Definitions and Notation

DEFINITION :: Complete Bipartite Graph

A bipartite graph $\boldsymbol{G}$ with bipartition $\boldsymbol{V_1} \cup \boldsymbol{V_2}$ is called a complete bipartite graph on m and n vertices if the subsets $\boldsymbol{V_1}$ and $\boldsymbol{V_2}$ contain $\boldsymbol{m}$ and $\boldsymbol{n}$ vertices, respectively, such that there is an edge between each pair of vertices $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$, where $\boldsymbol{v_1} \in \boldsymbol{V_1}$ and $\boldsymbol{v_2} \in \boldsymbol{V_2}$.

A complete bipartite graph on $\boldsymbol{m}$ and $\boldsymbol{n}$ vertices is denoted by $\boldsymbol{K_{m,n}}$



Isomorphism of Graphs

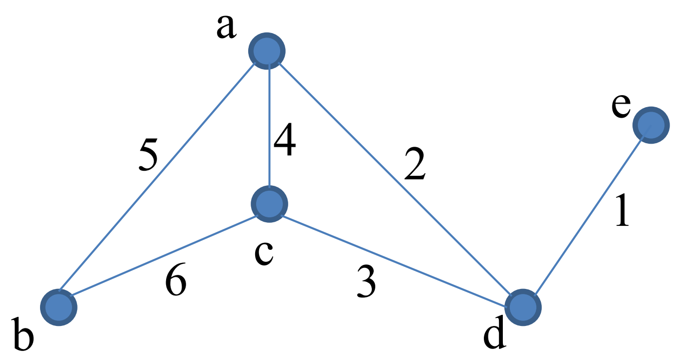
- Let G=(V,E) and G'={V', E'} be graphs.
- G and G' are said to be isomorphic if there exist a pair of functions f:V → V' and g: E→ E' that associates each element in V with exactly one element in V' and vice versa;
  - g associates each element in E with exactly one element in E' and vice versa, and
  - for each v∈ V, and each e ∈ E, if v is an end point of the edge e, then f(v) is an endpoint of the edge g(e).

Isomorphism of Graphs

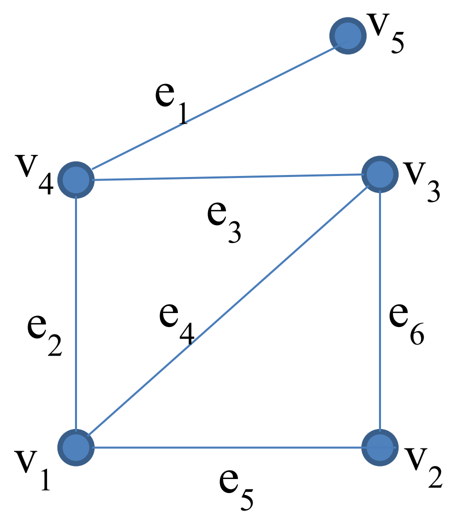
- If two graphs are isomorphic then they must have
  - The same number of vertices
  - The same number of edges
  - The same degrees for corresponding vertices
  - The same number odd connected components
  - The same number of loops
  - The same number of parallel edges.

Isomorphism of Graphs

Example:



A



B

- The vertices *a, b, c, d, e* in the graph (A) corresponds to v_1, v_2, v_3, v_4 *and* v_5 of the graph (B) respectively.
- The edges *1, 2, 3, 4, 5 and 6* correspond to e_1, e_2, e_3, e_4, e_5 *and* e_6 respectively.

Except for the labels (i.e. names) of their vertices and edges, isomorphic graphs are the same graph.

Walks, Paths, and Cycles

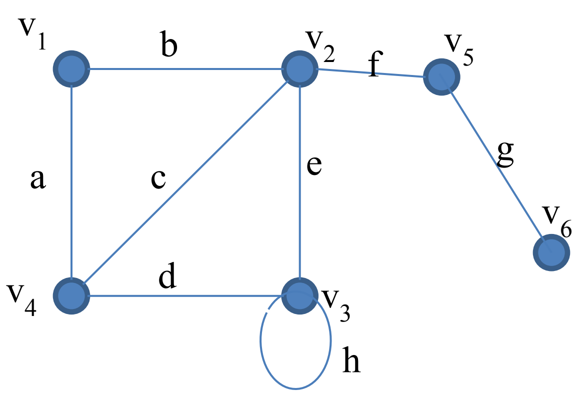
DEFINITION :: Walk

- **Let u and v be two vertices in a graph G.**
- **A walk from u to v, in G, is an alternating sequence of $n+1$ vertices and n edges of G**

$$(u = v_1, e_1, v_2, e_2, v_3, e_3, \ldots\ldots, v_{n-1}, e_{n-1}, v_n, e_n, v_{n+1} = v)$$

beginning with vertex u, called the *initial vertex*, and ending with vertex v, called the *terminal vertex*, in which v_i and v_{i+1} are endpoints of edge e_i for $i=1, 2, \ldots., n$.

Example:



Walks:

v_1, b, v_2, e, v_3, h, v_3, d, v_4

v_4, c, v_2, f, v_5 , g, v_6

v_1, b, v_2, c, v_4, a, v_1

Walks, Paths, and Cycles

DEFINITION :: Directed walk

- Let *u* and *v* be two vertices in a directed graph G.
- A directed walk from *u* to *v* in *G* is an alternating sequence of *n+1* vertices and *n* arcs of *G*

$$(u = v_1, e_1, v_2, e_2, v_3, e_3, \ldots.., v_{n-1}, e_{n-1}, v_n, e_n, v_{n+1} = v)$$

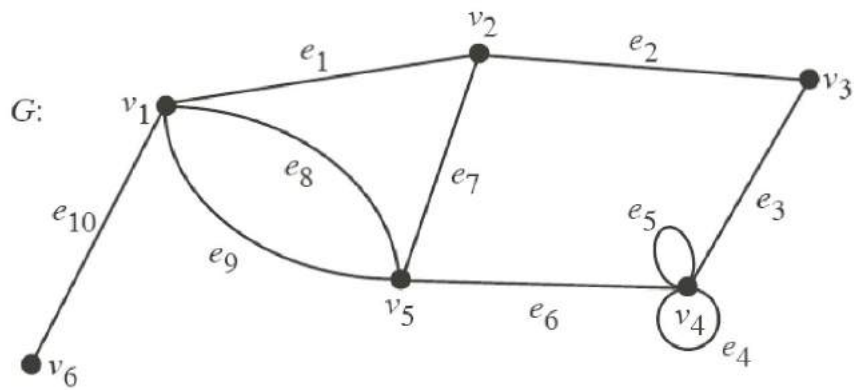
beginning with vertex u and ending with vertex v in which each edge e_i, for i=1, 2, … , n, is an arc from v_i to v_{i+1}

Walks, Paths, and Cycles

DEFINITION :: Length of a walk (directed walk)

The **length of a walk (directed walk)** is the total number of occurrences of edges (arcs) in the walk (directed walk). A walk or a directed walk of length 0 is just a single vertex.

A (directed) walk from a vertex u to a vertex v in G is also called $\boldsymbol{u - v}$ **(directed) walk**. If u and v are the same, then a $u - v$ (directed) walk is called a **closed (directed) walk**. If u and v are different, then a $u - v$ (directed) walk is called an **open (directed) walk**.



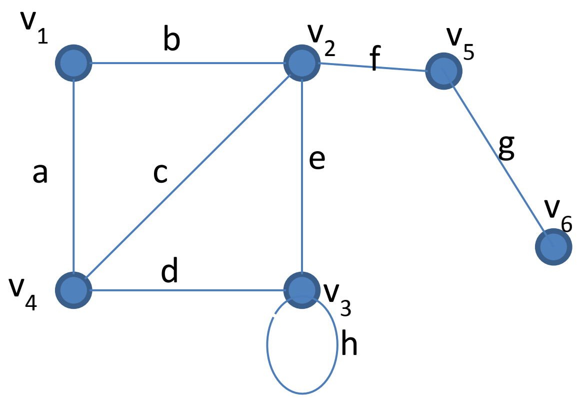
$v_2, e_7, v_5, e_8, v_1, e_8, v_5, e_6, v_4, e_5, v_4, e_5, v_4$ **Open walk**

$v_4, e_5, v_4, e_3, v_3, e_2, v_2, e_7, v_5, e_6, v_4$ **Closed walk**

Walks, Paths, and Cycles

DEFINITION :: Trail and Path

A walk with no repeated edges is called a **trail**, and a walk with no repeated vertices except possibly the initial and terminal vertices is called a **path**.



Walks:

v_1, b, v_2, e, v_3, h, v_3, d, v_4 ---- Not a path, trail

v_4, c, v_2, f, v_5 , g, v_6 ------ Path, trail

v_1, b, v_2, c, v_4, a, v_1 ----- Path (also closed path, cycle), trail (circuit)

DEFINITION :: Cycle

A circuit that does not contain any repetition of vertices except the starting vertex and the terminal vertex is called a **cycle**.

DEFINITION :: Circuit

A nontrivial closed trail from a vertex u to itself is called a **circuit**.

Hence, a circuit is a closed walk of nonzero length from a vertex u to u with no repeated edges.

Walks, Paths, and Cycles

DEFINITION :: Trivial walk, path or trail

A walk, path, or trail is called **trivial** if it has only one vertex and no edges. A walk, path, or trail that is not trivial is called **nontrivial**.

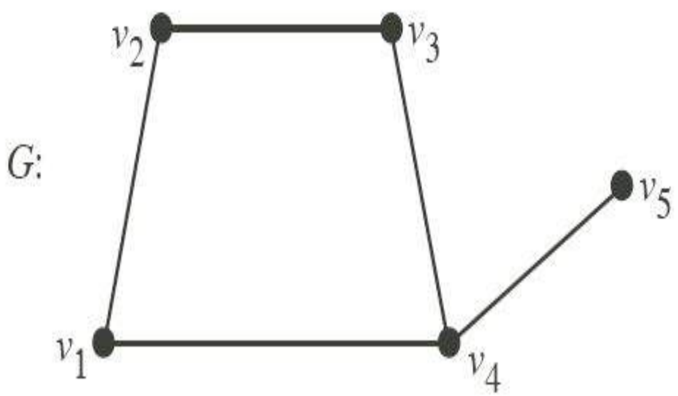
DEFINITION :: k-cycle

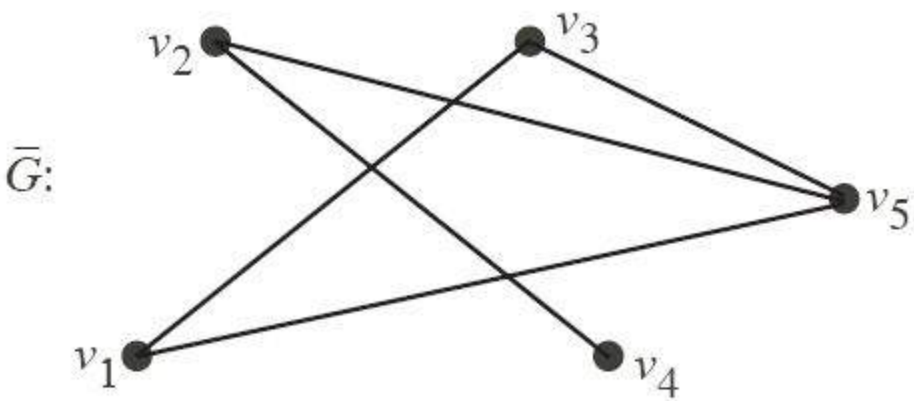
A cycle of length k is called a ***k*-cycle**. A cycle is called **even (odd)** if it contains an even (odd) number of edges.

Operations on Graph

DEFINITION :: Complement of a simple graph

The *complement* of the simple graph $G = (V, E)$ is the simple graph $\overline{G} = (V, \overline{E})$, where the edges in $\overline{E}$ are exactly the edges not in G.



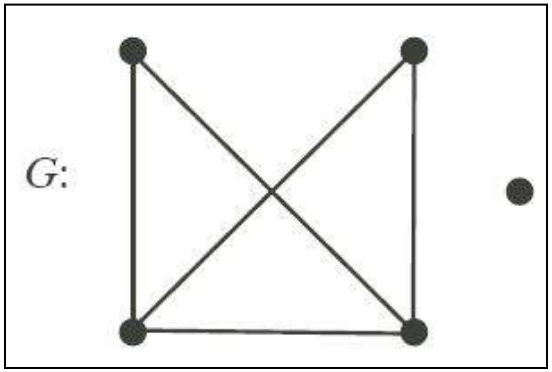


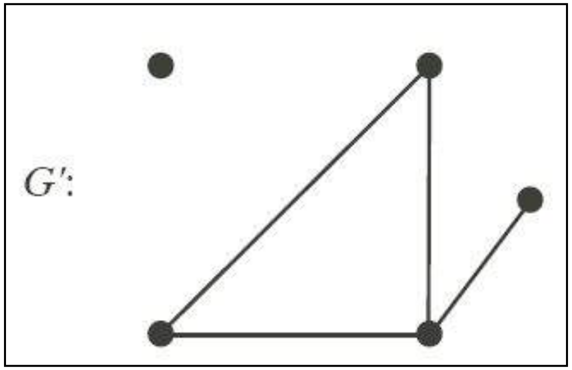
The complement of the complete graph K_n is the empty graph with n vertices.

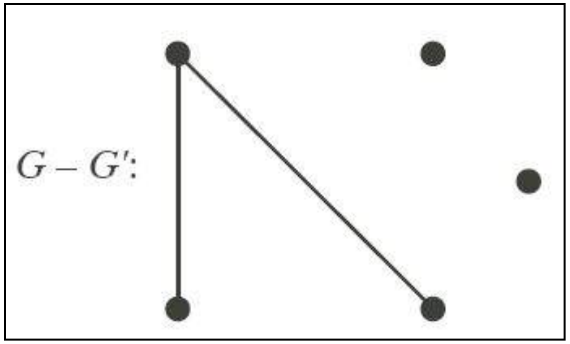
Operations on Graph

DEFINITION :: Difference operation

If the graphs G=(V, E) and G'=(V', E') are simple and V'⊆ V, then the difference graph is *G- G' = (V, E'')* where *E'' contains those edges from G that are not in G'* (simple graph).







Operations on Graph

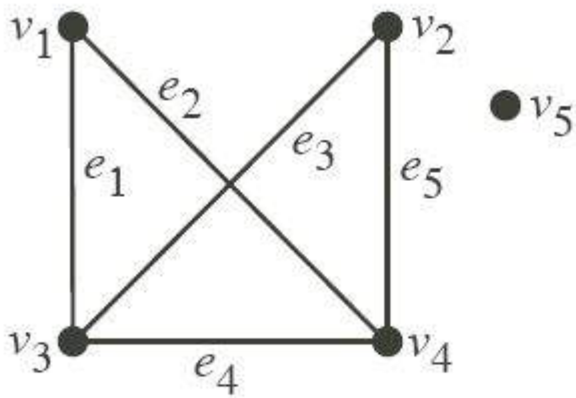
Here are some binary operations between two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$:

- The *union* is $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$ (simple graph).
- The *intersection* is $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$ (simple graph).
- The *ring sum* $G_1 \oplus G_2$ is the subgraph of $G_1 \cup G_2$ induced by the edge set $E_1 \oplus E_2$ (simple graph). *Note!* The set operation $\oplus$ is the *symmetric difference*, i.e.

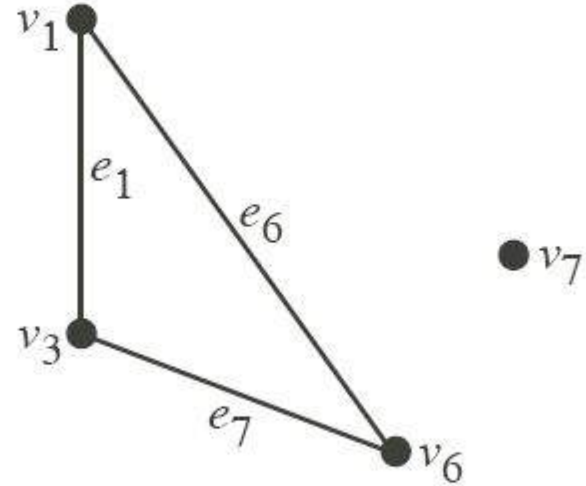
$$E_1 \oplus E_2 = (E_1 - E_2) \cup (E_2 - E_1).$$

Operations on Graph

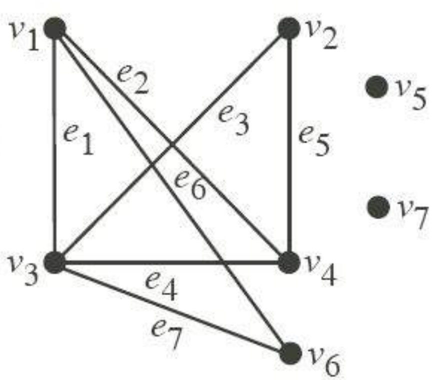
G_1:



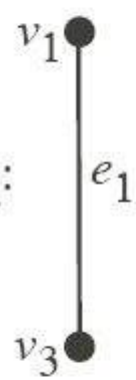
G_2:



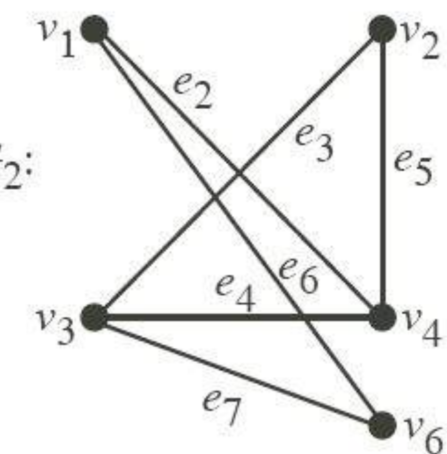
$G_1 \cup G_2$:



$G_1 \cap G_2$:



$G_1 \oplus G_2$:



Operations on Graph

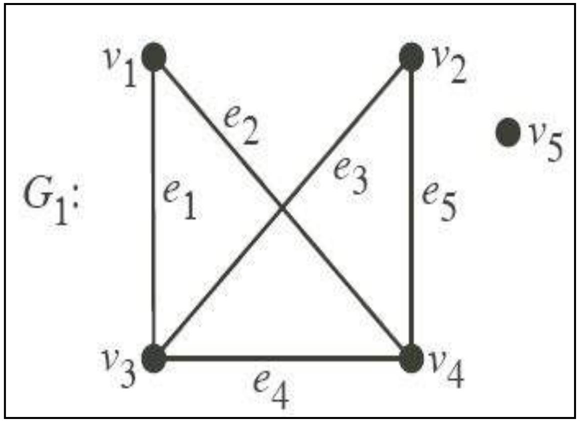
DEFINITION ::Removal of a vertex

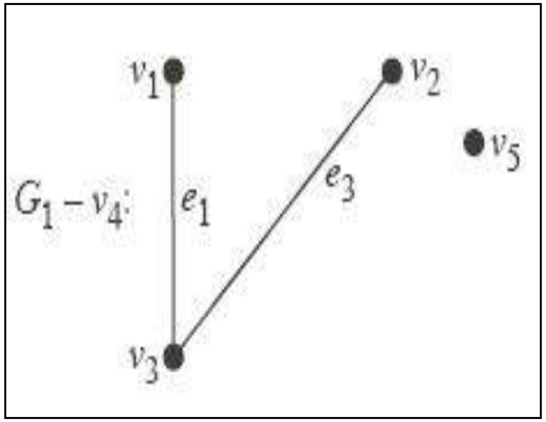
If v is a vertex of the graph $G = (V, E)$, then $G - v$ is the subgraph of G induced by the vertex set $V - \{v\}$. We call this operation the *removal of a vertex*.

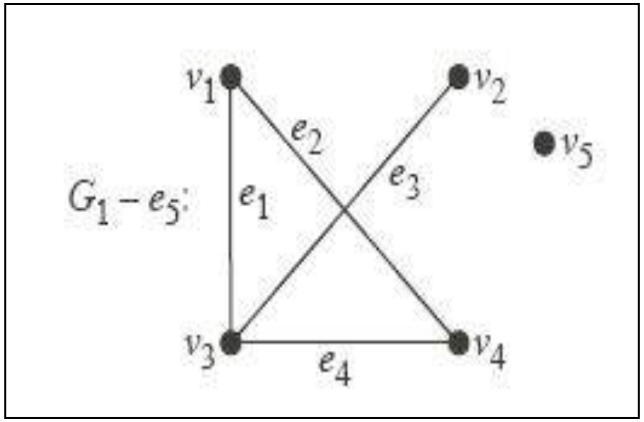
i.e. deletion of a vertex v means the deletion of the vertex v and deletion of all the edges incident on v.

DEFINITION ::Removal of a edge

If e is an edge of the graph G = (V, E) then G – e is graph (V, E') , where E' is obtained by removing e form E.







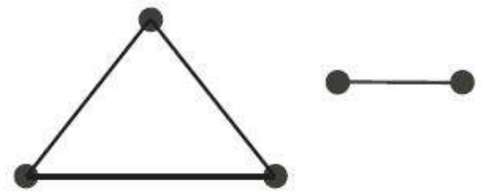
Connected graph

DEFINITION :: Connectedness

Let G be a graph. A vertex u is said to be **connected** to a vertex v if there is a $u - v$ walk in graph G.

DEFINITION :: Connected Graph

A graph G is called a **connected graph** if for any two vertices u, v of G there is a $u - v$ walk in G, otherwise the graph is called a **disconnected graph**.

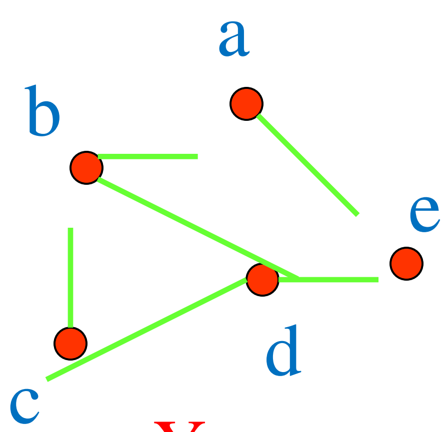


For example, any two computers in a network can communicate if and only if the graph of this network is connected.

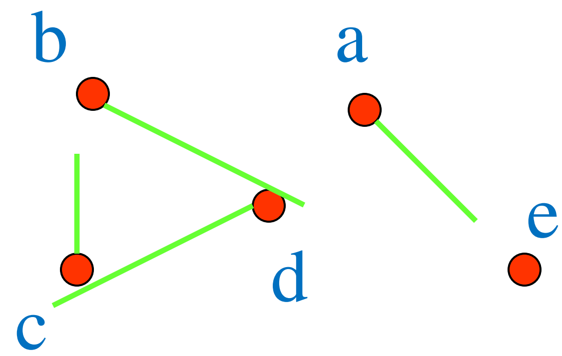
Note: **A graph consisting of only one vertex is always connected, because it does not contain any pair of distinct vertices.**

Connected graph

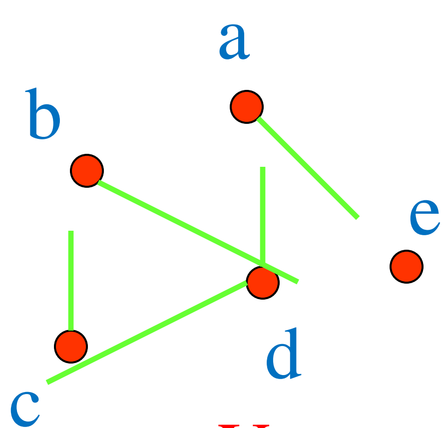
Example: Are the following graphs connected?



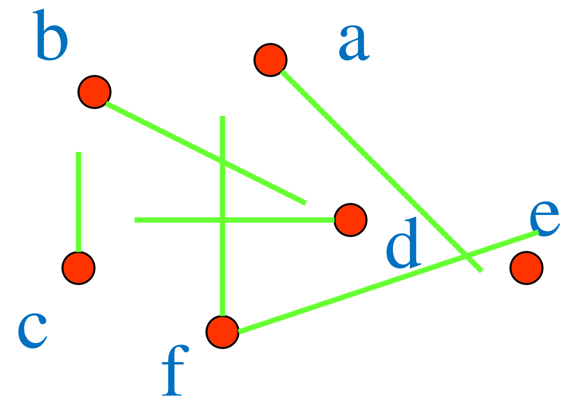
Yes.



No.



Yes.



No.

Some Theorem

Theorem:

A graph is disconnected if and only if its vertex set V can be partitioned into two nonempty, disjoint subsets V_1 and V_2 such that there exists no edge in G whose one end vertex is in subset V_1 and the other in subset V_2.

Proof:

Suppose that such a partition exists.

Consider two arbitrary vertices a and b of G, such that $a \in V_1$ and $b \in V_2$. No path can exist between vertices a and b; otherwise there would be at least one edge whose one end vertex would be in V_1 and the other in V_2. hence, if a partition exists, G is not connected.

Conversely, let G be a disconnected graph.

Consider a vertex a in G. Let V_1 be the set of all vertices that are joined by paths to a. Since G is disconnected V_1 does not include all vertices of G. The remaining vertices will form a (nonempty) set V_2. So no vertex in V_1 is joined to any in V_2 by an edge. Hence the partition.

Component

DEFINITION :: Component

A subgraph H of a graph G is called a **component** of G if

(i) any two vertices of H are connected in H, and
(ii) H is not properly contained in any connected subgraph of G.

Connected graph

Theorem: If a graph (connected or disconnected) has only two odd vertices v_1 and v_2, then there is a path between v_1 and v_2.

Proof:

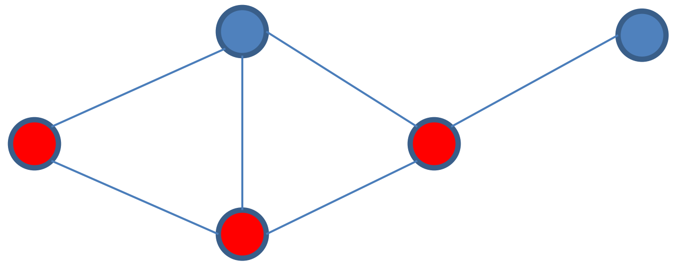
Case 1: G is connected. It is obvious that there is a path between v1 and v2.

Case 2: G has two or more connected components. For each of these connected components the theorem which states that number of odd vertices in a graph is always even is applicable. As there are only two odd vertices so, v1 and v2 must belong to the same component. Otherwise the above stated theorem would be violated. (Proved)

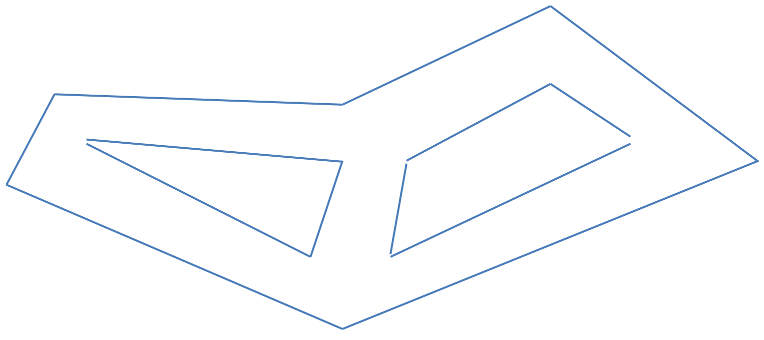
Vertex Cover

- A set S is a subset of V(G) is a vertex cover of G (of the edges of G) if every edges of G is incident with a vertex in S.
- A vertex cover of the minimum cardinality is called a minimum vertex cover denoted as MVC(G)

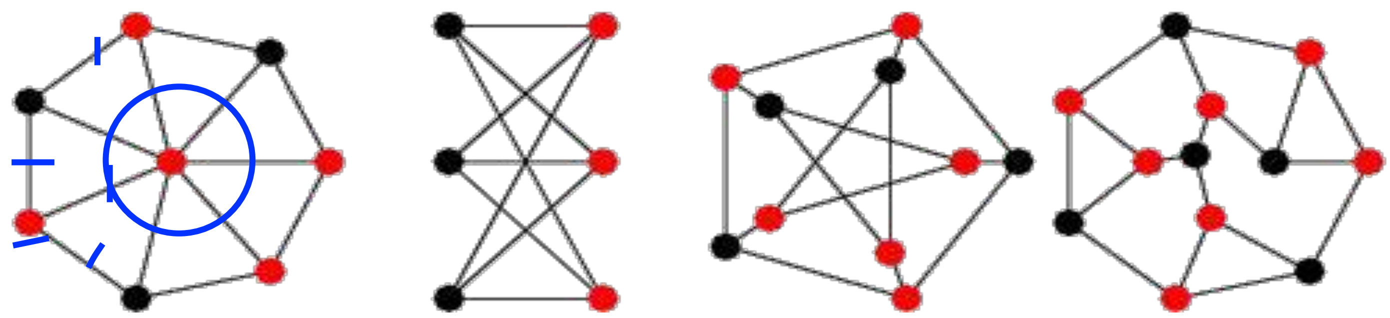
Example:



Application: Art Gallery



Some more examples of vertex cover



wheel graph W_8 $K_{3,3}$ utility graph Petersen graph and Frucht graph

Vertex Cover

- What is the cardinality of MVC in the complete graph K_n ?
- What about the complete bipartite graph $K_{m,n}$?
- The cycle C_n, where n is even or odd?

- K_n it will be (n-1).
- $K_{m,n}$ it will be m<= n in two sets A and B, then m can be MVC
- C_n it will be n/2 (it it is even) and *celing(n/2)* if it is odd.

Vertex Cover Algorithm

- Vertex Cover Problem is a known NP Complete problem
- There are ***approximate polynomial time algorithms*** to solve the problem though.

Algorithm

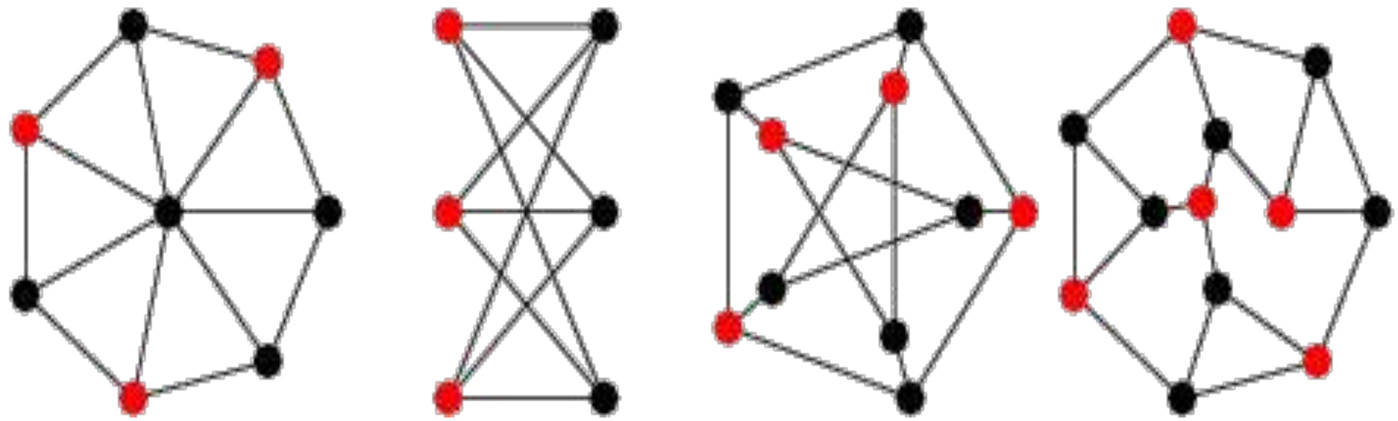
1) Initialize the results as {}
2) Consider a set of all edges in given graph. Let the set be E.
3) Do the following while E is not empty
   a) Pick an arbitrary edge (u, v) from set E and add ‘u’ and ‘v’ to result.
   b) Remove all edges from E which are either incident on u or v.
4) Return result

Independent Set

- **Definition:**

  An independent set or stable set is a set of vertices in a graph, no two of which are adjacent.
  - That is, it is a set of *l* vertices such that for every two vertices in *l,* there is no edge connecting the two.
  - Equivalently, each edge in the graph has at most one end point in *l.*
  - The size of an independent set is the number of vertices it contains.
    - The cardinality of the biggest independent set in G is called the independence number (or stability number) of G and is denoted by $\alpha(G)$.
- What will be the values for $\alpha(K_n)$, $\alpha(K_{m,n})$ and $\alpha(C_n)$??

Some examples of Independent Set



wheel graph W_8 $K_{3,3}$ utility graph Petersen graph and Frucht graph

Independent Set

Algorithm:

1. $I = \emptyset$, $V' = V$
2. While ($V' \mathrel{!}= \emptyset$) do
   a. Choose any $v \in V'$
   b. Set $I = I \cup v$
   c. Set $V' = V \setminus (v \cup N(v))$
3. Output I

Relation between MVC and $\beta(G)$

- If we remove a VC from G, then rest is an independent set.
- So, if we remove MVC from G, the rest, i.e. V- MVC is an independent set.
- So $\alpha(G) >= n - |MVC(G)|$
  - > $|MVC(G)| >= n - \alpha(G)$.
- Similarly, if we remove any independent set from G, the rest is VC, and so, $|MVC'(G) <= n- \alpha(G)$.
- Thus we get $|MVC(G)| = n- \alpha(G)$.
- If we denote $|MVC(G)|$ as $\beta(G)$ then $\beta(G)+ \alpha(G) = n$.

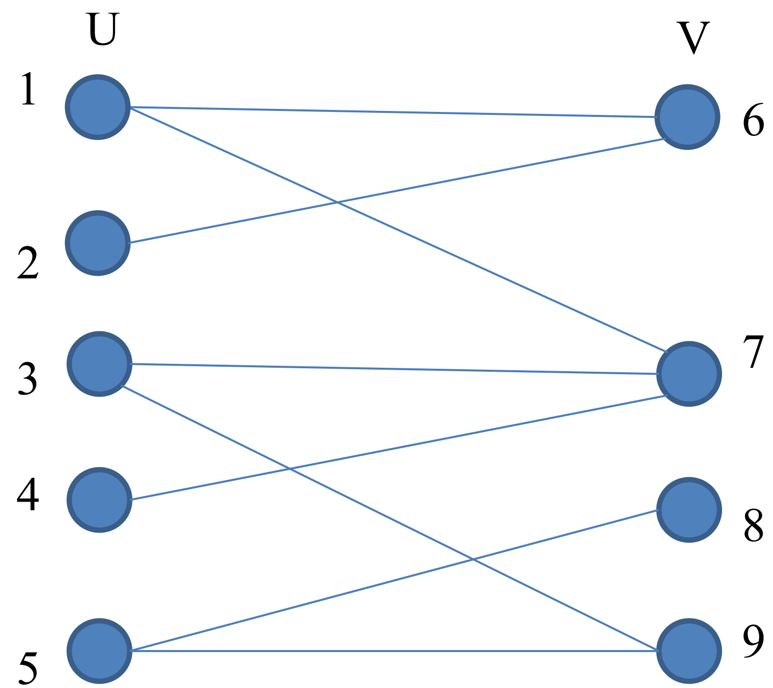
Matching

Matching

- Motivating example:
  - J jobs and C candidates
  - Li: List of jobs candidate i can do
  - Constraint: Each candidate must be given at most 1 job. Each job must be assigned to at most 1 candidate.
  - Goal: Assign candidates to jobs such that maximum number of jobs are filled.

Bipartite Maximum Matching

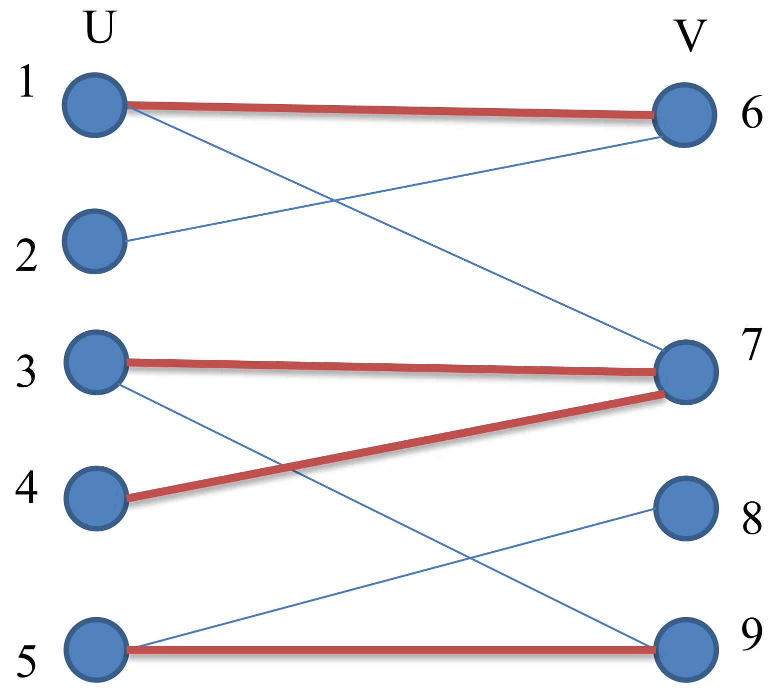
- Input:
  - Bipartite graph G=(U, V, E) where U and V are vertex sets with |U|=n1, |V|=n2 and E the set of edges.



Bipartite Maximum Matching

- Matching:
  - M is a subset of E is said to be a matching if at most one edge from M is incident on any vertex (in U or in V)

Goal: Matching of maximum possible size.



Matching

- Maximal matching:
  - A matching M is said to be maximal if M is not properly contained in any other matching.
    - Intuitively this is equivalent to say that a matching is maximal if we can not add any edge to the existing set.
- Maximum Matching:
  - A matching M is said to be maximum if for any other matching M' |M| >=|M'|

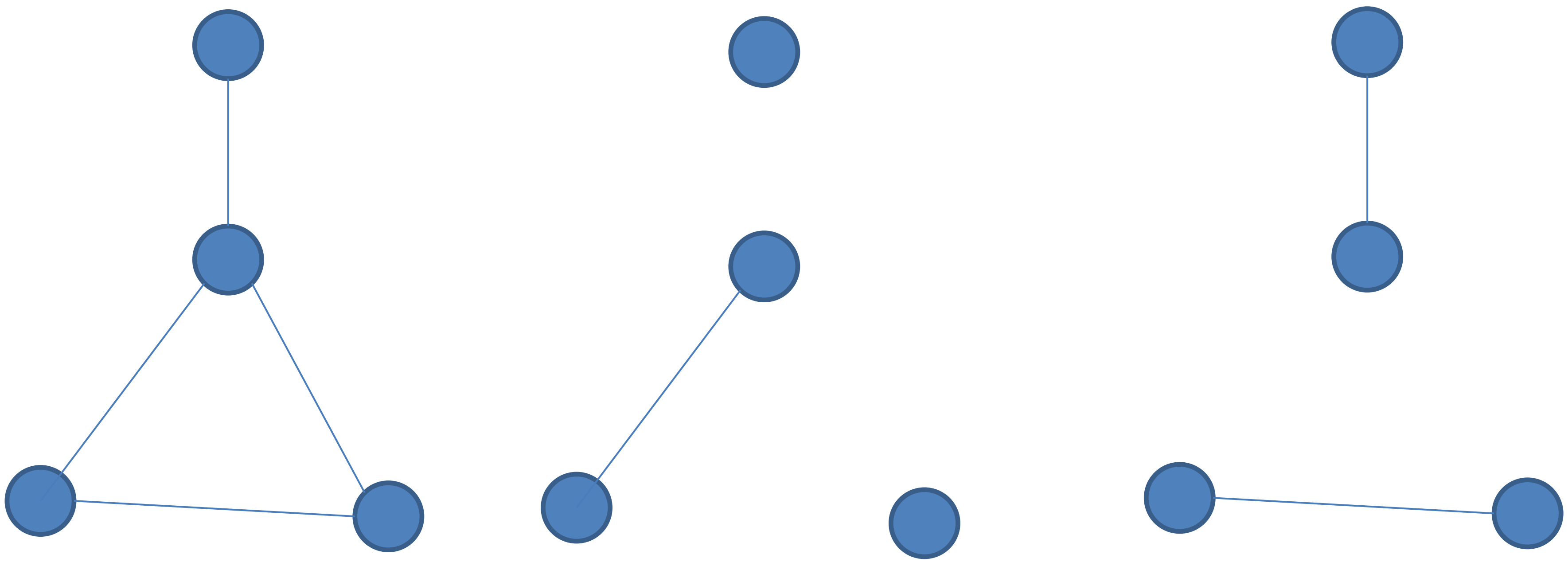
Matching

- Theorem: If a matching M is maximum □ M is maximal.
- Proof: Suppose M is not maximal
  - – > There exists M’ such that M is a subset of M’
  - – > |M|<|M’|
  - – > M is not maximum

  Therefore we have a contradiction.

Matching

- The converse of the above is not true. This can be shown using a counter example.

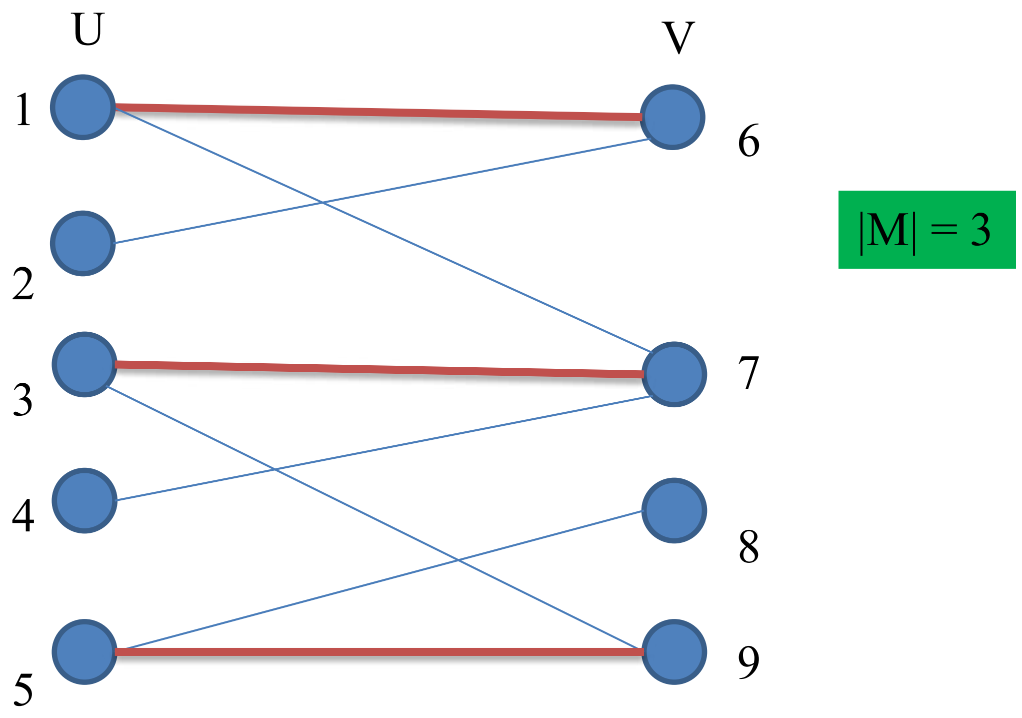


Original

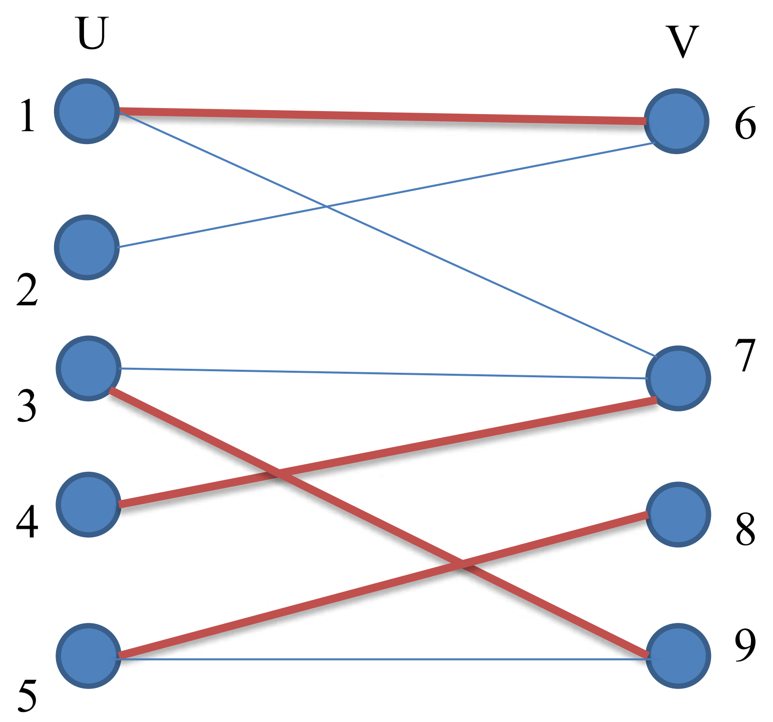
Maximal matching

Maximum matching

Matching example



Matching example

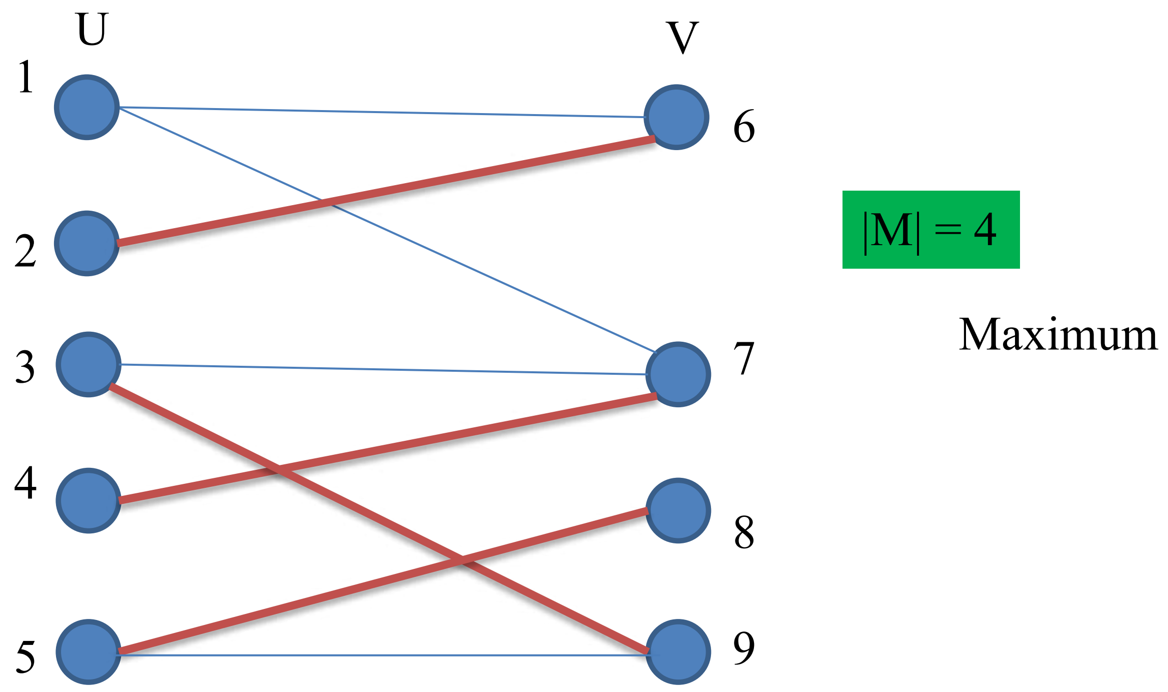


|M| = 4

Maximum

But maximum matching is not unique

Matching example

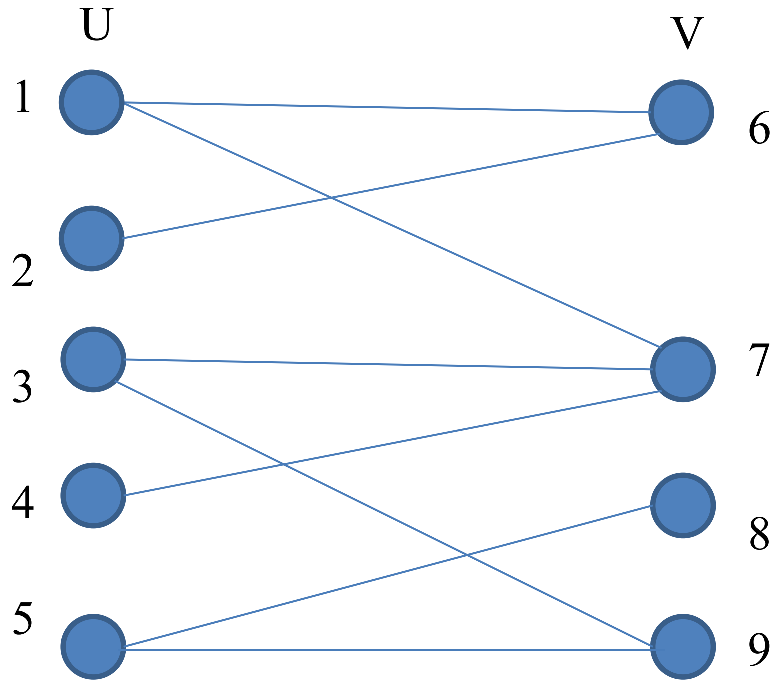


This is also a possible matching with maximum size.

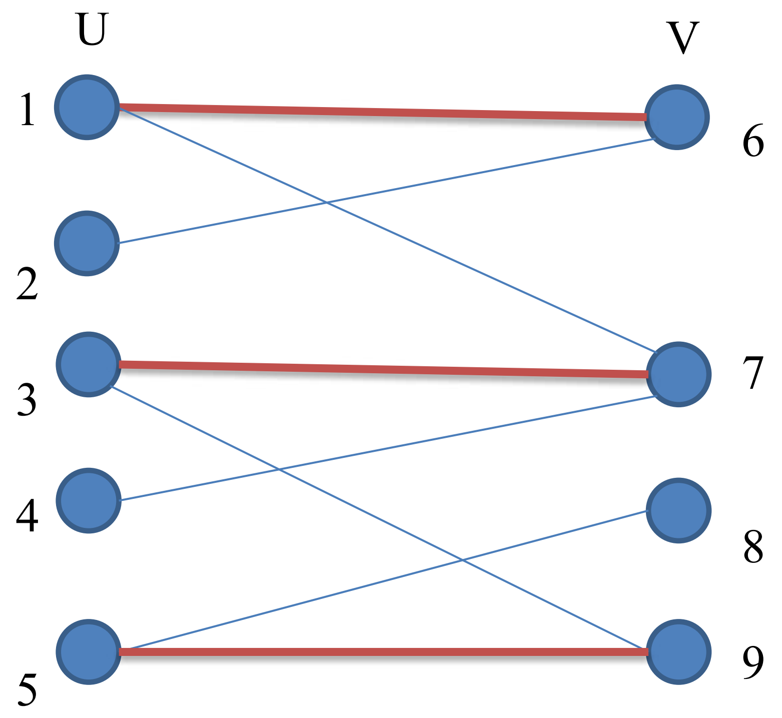
Job assignment problem

Candidates: 1, 2, 3, 4, 5
Jobs: 6, 7, 8, 9

Edge(u,v): Candidate u can do job v



Job assignment to candidates



3 candidates are assigned to 3 jobs

Matching : Job assignment

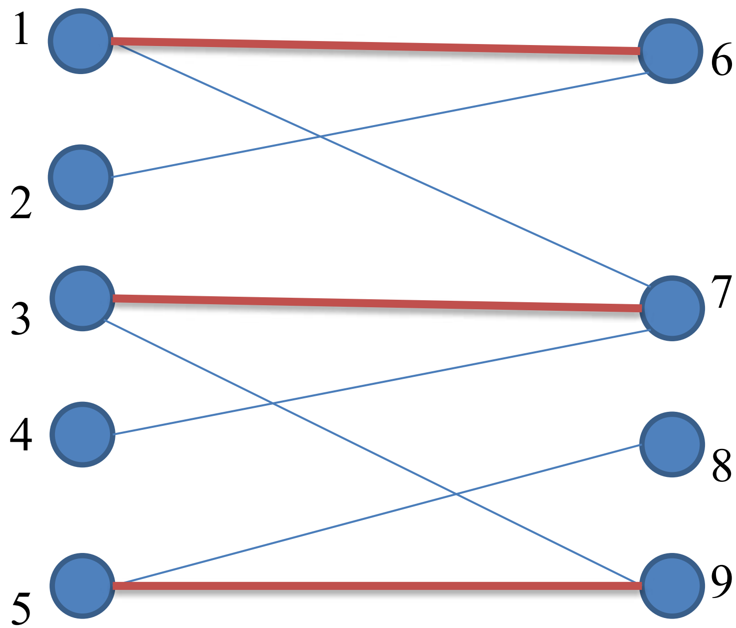
Outline

- An algorithm design idea
- Augmenting paths
- High level Algorithm
- Correctness: Berg’s Theorem
- Efficient Implementation

Matching Algorithm

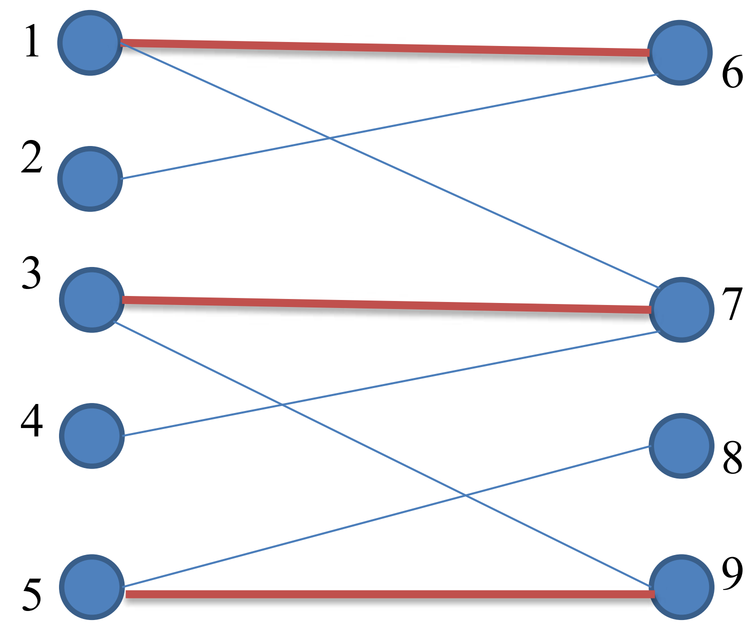
An Idea:

- Greedy: Keep on adding edges into M till no more edges.

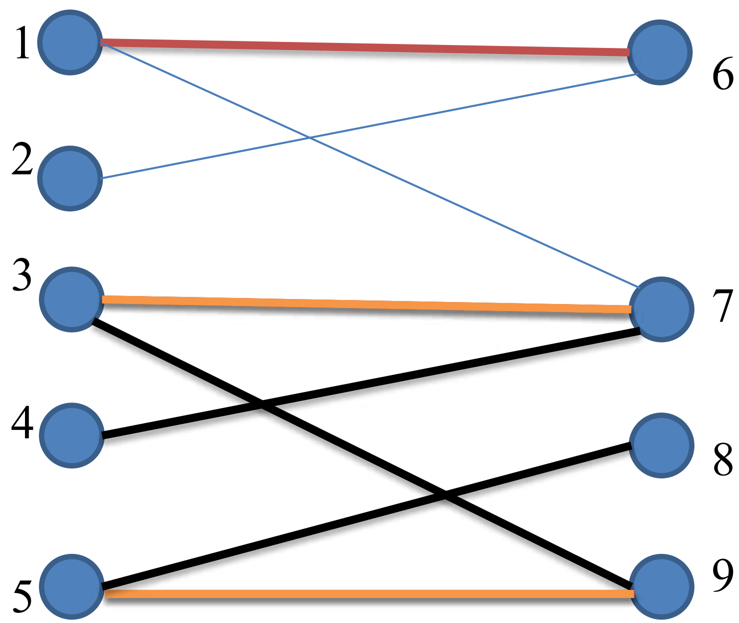


Matching Algorithm

- Kho Kho idea:
  - Free vertex for M: Not an endpoint of any edge in M
  - Match a free vertex. Disturbs matching, Remove conflict. Repeat.



Matching Algorithm



Vertices 2, 4 and 8 are free for M

Conflict with edge (3,7)

Vertex 3 is now free

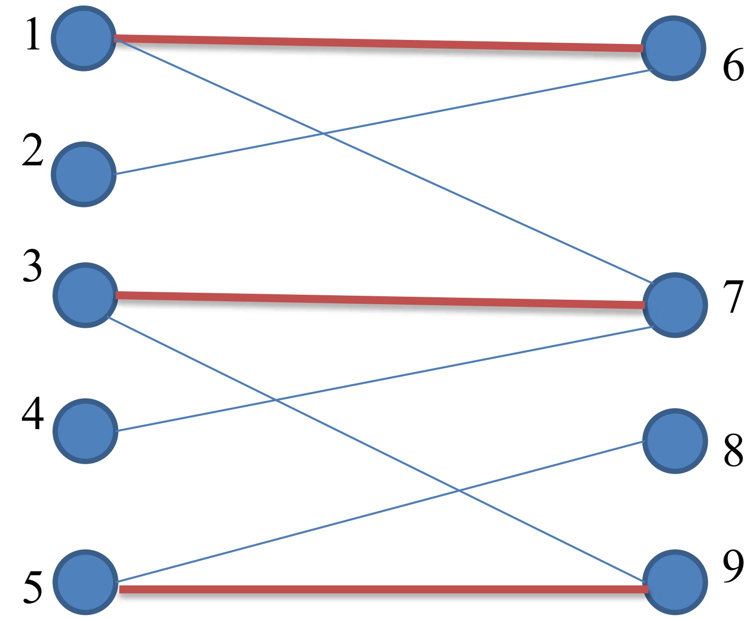
Conflict with edge (5,9)

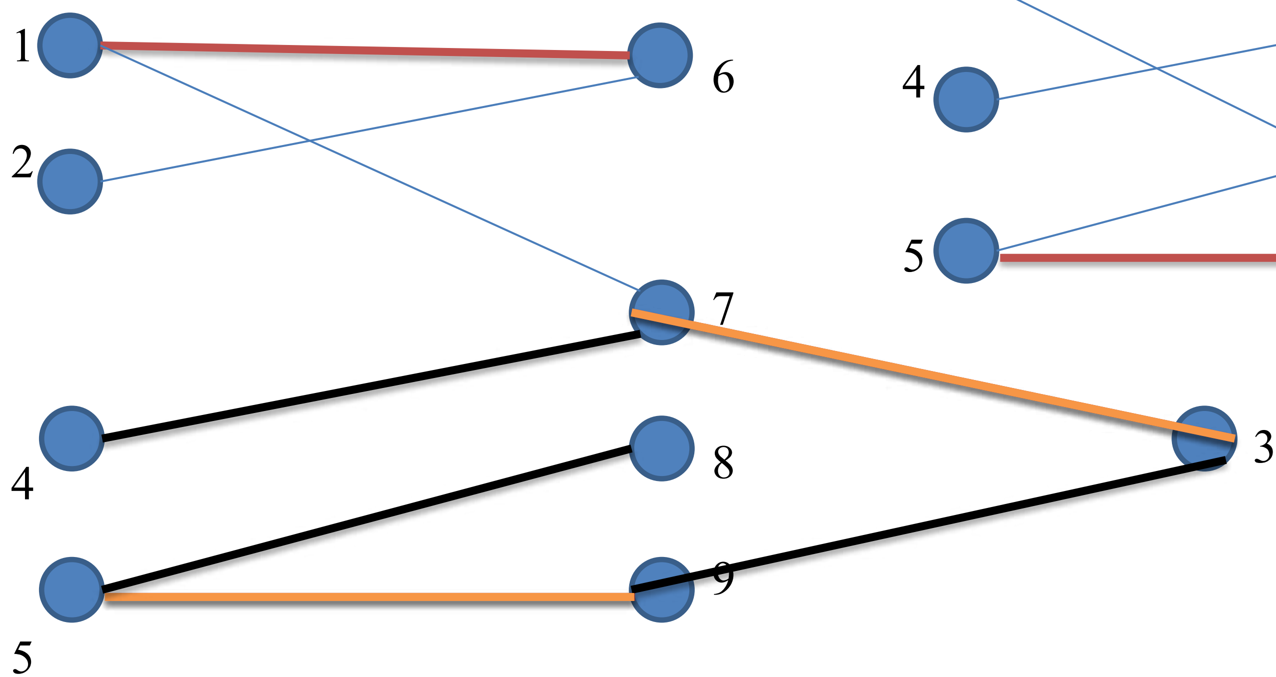
Vertex 5 is now free

No conflict!!

Added 3 edges (Black)
Removed 2 edges(Orange)

Matching Algorithm





One extra edge is
added with M

Augmenting Path for a Matching M

- Definition:
  - **Matched vertex:** Given a matching M, a vertex, v is said to be matched if there is an edge e belongs M which is incident on v.
  - **Augmenting path:** Given a graph G=(V, E) and a matching M is a subset of E. A path P is called an augmenting path for M if:
    - The two end points of P are unmatched by M.
    - The edges of P alternate between edges belongs M and edges not belongs M.
- Hence it is a sequence P of vertices $v_1, v_2, \ldots, v_k$ such that
  - $v_1 \in U$ and $v_k \in v$ are free in M.
  - $(v_1, v_2), (v_3, v_4), \ldots, (v_{k-1}, v_k) \in E-M$ go forward
  - $(v_2, v_3), (v_4, v_5), \ldots, (v_{k-2}, v_{k-1}) \in M$ go back

Augmenting Path for a Matching M

- – New bigger matching= $M \oplus P$
  - $Q \oplus R$ = Set of elements in Q, or in R, but not in both

Edmonds' Algorithm

M= empty matching

while there is an augmenting path P for M

$M = M \oplus P$

Output M.

Berg’s Theorem

- A matching M in a bipartite graph is maximum if and only if there does not exist an augmenting path for M.

Proof:

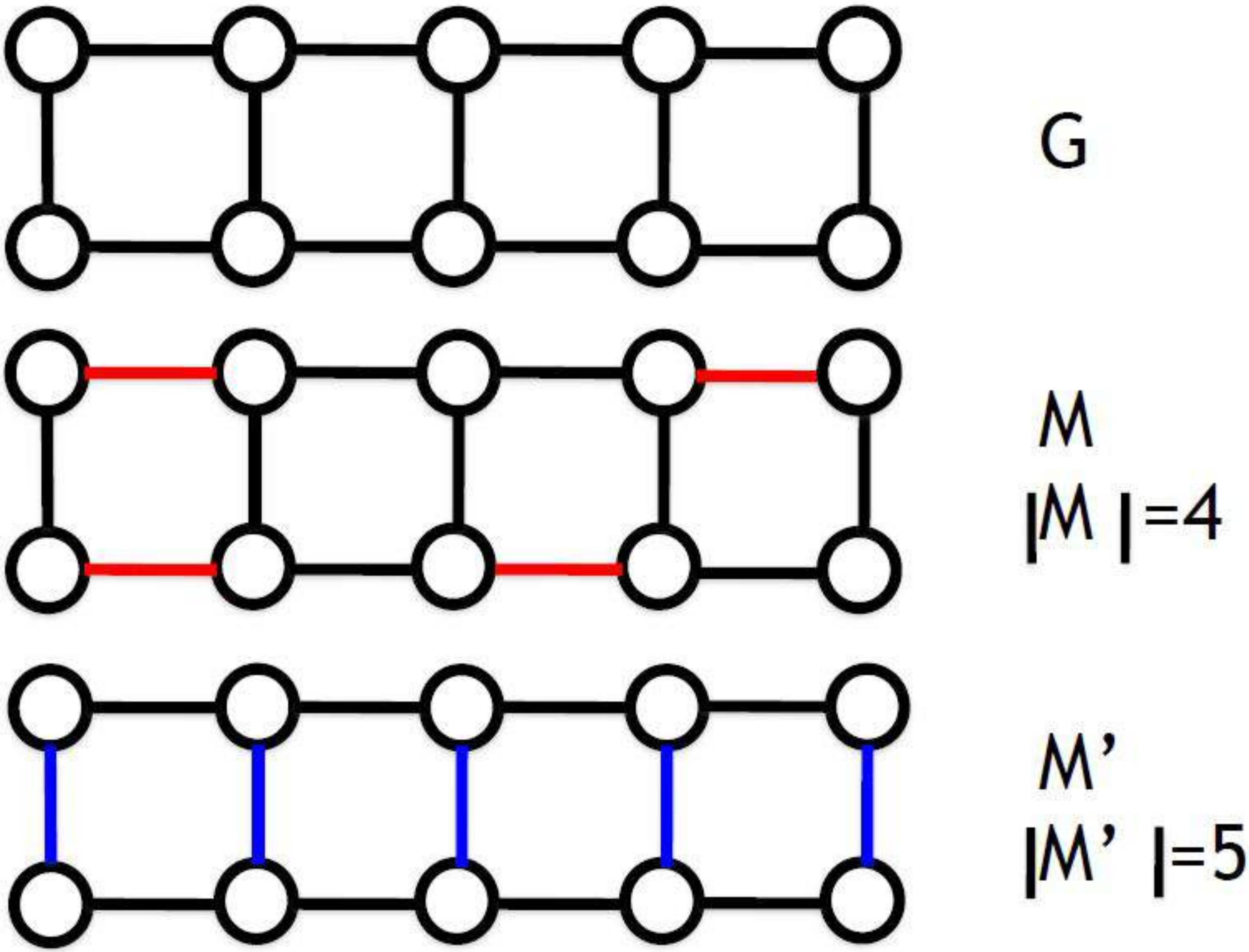
If-case: Obvious. Because Assuming G contains an augmenting path P, the set (M\E(P)) ∪ (E(P)\M) is a matching with larger cardinality than M.

  – So M is not maximum.

Only-If case: So we to prove that If there exists no augmenting path then M is maximum.

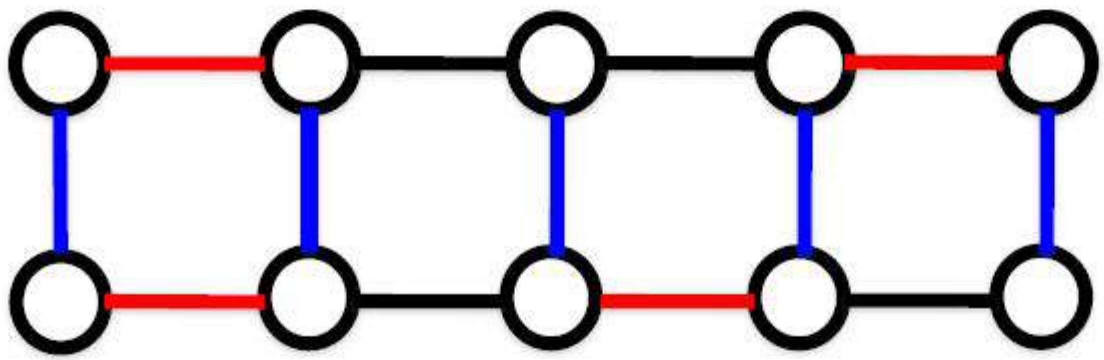
By contrapositive it is equivalent to If M is not maximum then there exist an augmenting path.

- Since M is not maximum, there exists a matching M’ with |M’|>|M|.
- Consider the subgraph H is a subset of G, with V(H)=V(G) and E(H)=M U M’.
- Every component of this graph is either a cycle of even length with edges alternately in M and M’ or a path consisting of one edge e with e ϵ M∩ M’.
- Since |M’| >|M|, there is a component of H which is a path with more edges in M’ than M.
  - Then P is an augmenting path.


G
M
|M |=4
M’
|M’ |=5

Berg’s Theorem

- Consider the set S of edges that are belongs to M or M’, but not both.
- S is called the symmetric difference of M and M’

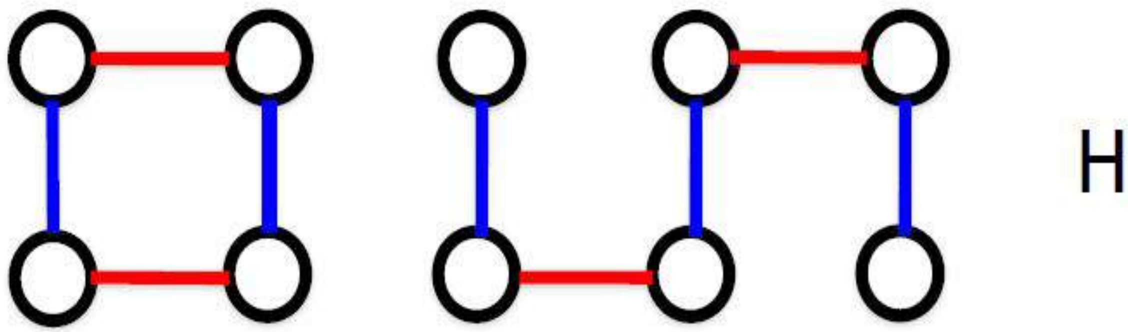


S

- In this example M and M’ share no edges so S is the union of M and M’

Berg's Theorem

- Let H be a graph formed by the edges in S and their endpoints.



- Since both M and M' are matchings, every vertex of H has degree at most 2.
- If H had a vertex of degree greater than 2, 2 edges of M or M' would have to have a vertex in common which can not happen in a matching.
- So H is a collection of disjoint paths and cycles of even length

How to find augmenting paths:

- An augmenting path starts and ends at a free vertex, and has alternate edges from M.
- Without loss of generality starting point belongs to U.
- If we knew starting point, we could grow the path from there.
- Key idea:
  - Grow paths from ALL free vertices in U
  - Paths grow forward using only edges in E-M, and backward using edges in M.
  - If we reach a free node in V through any path, we are done.
- BFS on auxiliary graph for M, in G

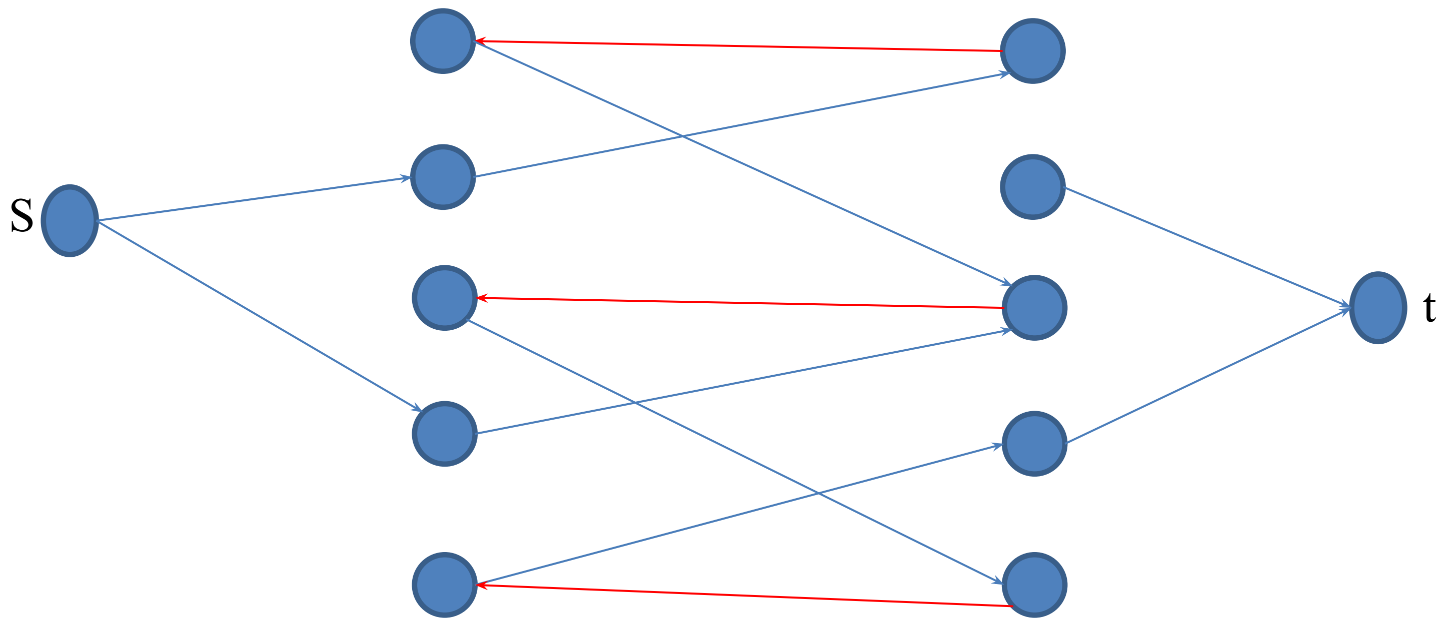
Matching algorithm

- **Edmonds' Algorithm**
  M= empty matching
  while there is an augmenting path P for M
  M= M $\oplus$ P
  Output M.

- Augmenting Path(G,M){
  G'= Auxiliary graph for G,M.
  P=Path from s to t (use BFS).
  If P!= NULL, delete s and t and return.
  else return false.
  }
  n=|U|+|V|, m=|E|, adjacency representation

Time (Constructing G')=Time(BFS of G')= O(m+n)
Time(Augmenting Path)=O(m+n)=Time(Computing M $\oplus$P)
Number of augmentations <= Matching Size <=n/2
Total time=O(n(m+n)).

Example



Starting vertices for path:
2,4

Auxiliary garph

- Auxiliary Graph for G,M
- G'=(V',E'), where V'= ∪(U,V, {s,t})
- E'=E_s, E_f, E_b, E_t where
- E_s= arcs from s to every free vertex in U
- E_f= $\{(u,v) | u \in U, v \in V, (u,v) \in E\text{-}M\}$
- E_b= $\{(v,u) | u \in U, v \in V, (v,u) \in M\}$
- E_t=arcs from every free vertex in V to vertex t.

Theorem

- **Theorem:**

    G has an augmenting path for M iff G' has a directed path from s to t.

Proof:

- **Only if part:** Let A= $v_1,\ldots,v_k$ be an augmenting path for G,M. We need to show that $s,v_1,\ldots,v_k,t$ must be a directed path in G'.
- An augmenting path in G□ Starts at a free vertex in U, goes forward and backward several times, terminates at a free vertex in V.
- In the forward direction, the path uses in E-M – these are present in G' and are directed forward.
- In the backward direction, it uses the edges of M – but these are present in G' too and are directed backward.
- Finally, s has a directed edge to every free vertex in U, and every free vertex in V has an edge to t. Thus we have a path in G'.

Proof continued…

- Only if part: by giving the same reasoning in reverse way it can be proved.

Comments

- Algorithm can be thought of as iterative refinement: We have a matching currently, can we improve it by making a small change? An augmenting path allows us to determine if a small change can be made.
- Faster algorithm are known: Best time known so far as $O(mn^{1/2})$
- Maximum matching can also be found in non-bipartite graphs in the same time as above, but the algorithm is much, much more complicated.

Shortest path algorithms

Problem statement

- Imagine you are planning a road trip to attend a music festival in another city.
- Supplied with a map of the region, your task is to find the shortest path amount of distance you have to travel to reach your destination.
- Unlike the case of finding a route through a data network, the time it takes to get from one way point to the next is not negligible.

Example

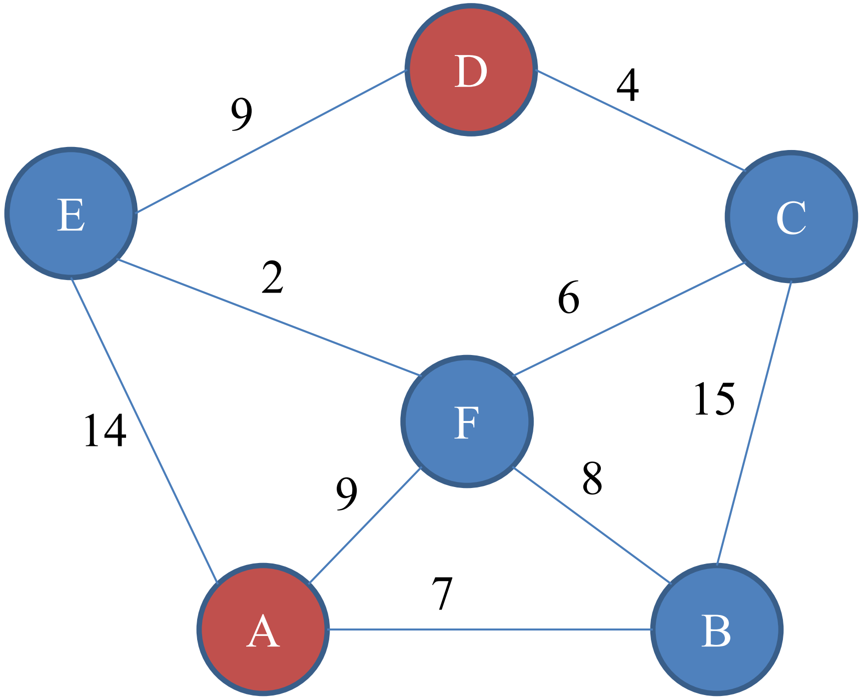
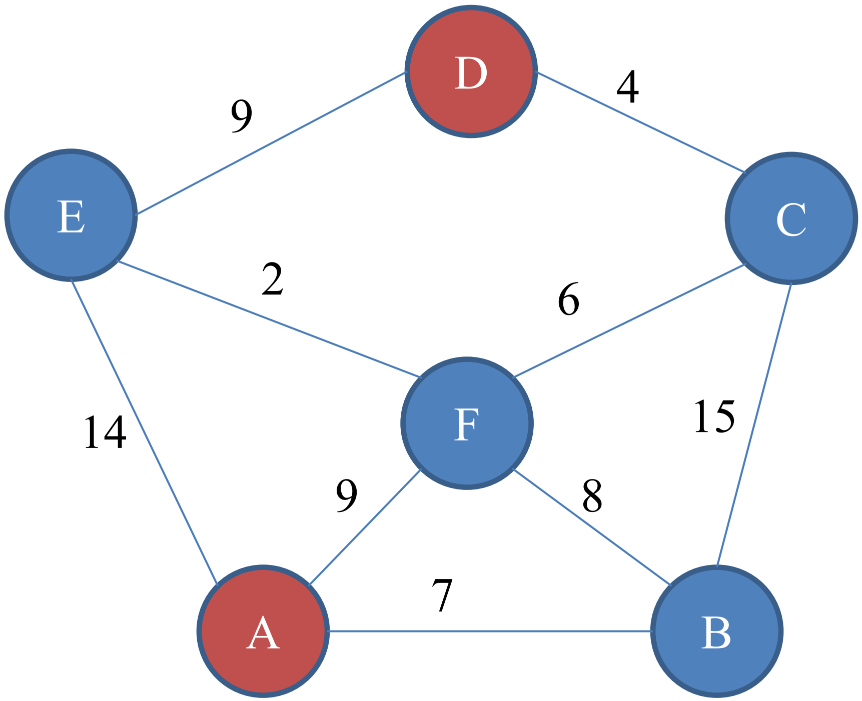


Figure: Cities in the map are circled. The distances between cities are the labelled lines.

Analysis of previous approaches

- Breadth First search (BFS)
  - We could perform a BFS on the map by using a queue.
  - We have seen how BFS could be used to construct the shortest path between two cities.
  - We could accumulate the distances as we visit each city in BFS order.
  - Once we reach the destination we know the total distance and we can reconstruct the path.

Example



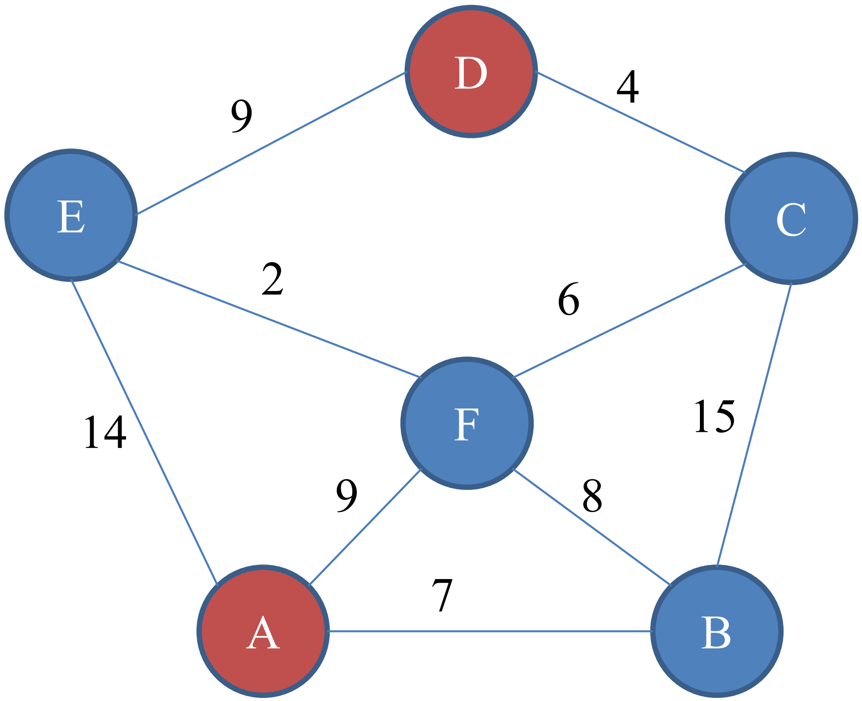
Example: Path(A,E,D) , Total distance: 14+9=23

- **Clearly, claculating the shortest path does not always result in the shortest distance.**
- **Unless the distance between all cities is the same, BFS will not always compute the shortest path.**

Analysis of previous approaches

- Depth First Search (DFS):
  - We could perform a DFS on the map by using a stack.
  - Similar to BFS we accumulate the distances as we visit each city in depth first order.
  - Once we reach the distances we know the total distance and we can reconstruct the path.

Example



Example: Path(A, B, C, D) , Total distance: 7+15+4=26

- **Clearly, calculating the shortest path does not always result in the shortest distance.**
- **This approach only works if we stumble upon the optimal distance by virtue of always choosing the correct next neighbour.**

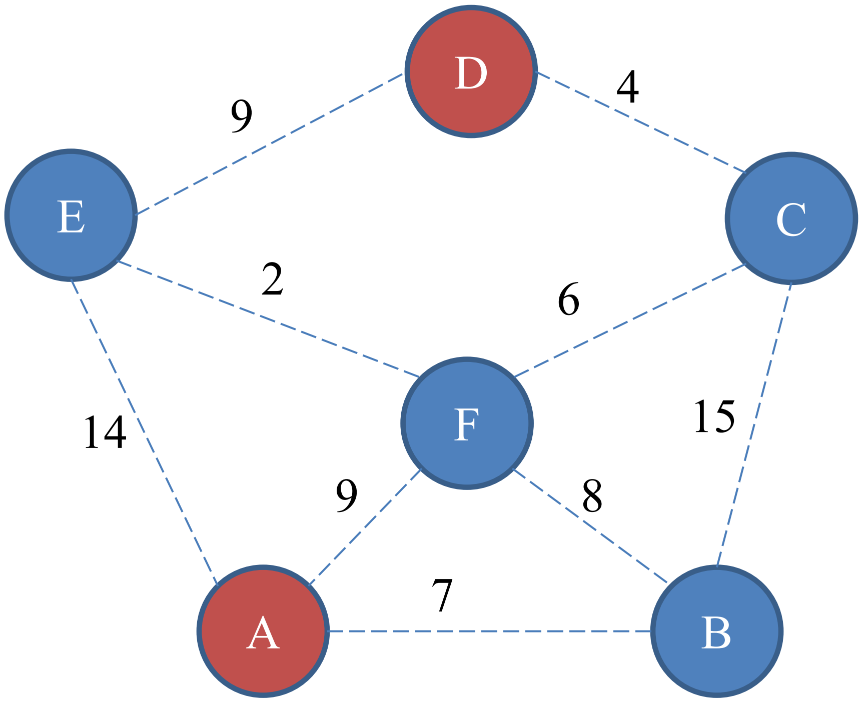
Analysis of previous approaches

- Exhaustive:
  - Exhaustive search is always an option.
  - Continue generating different paths even when a valid one is produced.
  - Once generated we would then choose the path with the lowest cumulative distance.
  - But this approach will work at the cost of enumerating all possible paths between two cities.
    - As the number of cities and roads are increase, the cost to compute the shortest distance can increase exponentially.
    - This is because each new city added causes all existing cities and paths to consider the new city.

Analysis of previous approaches

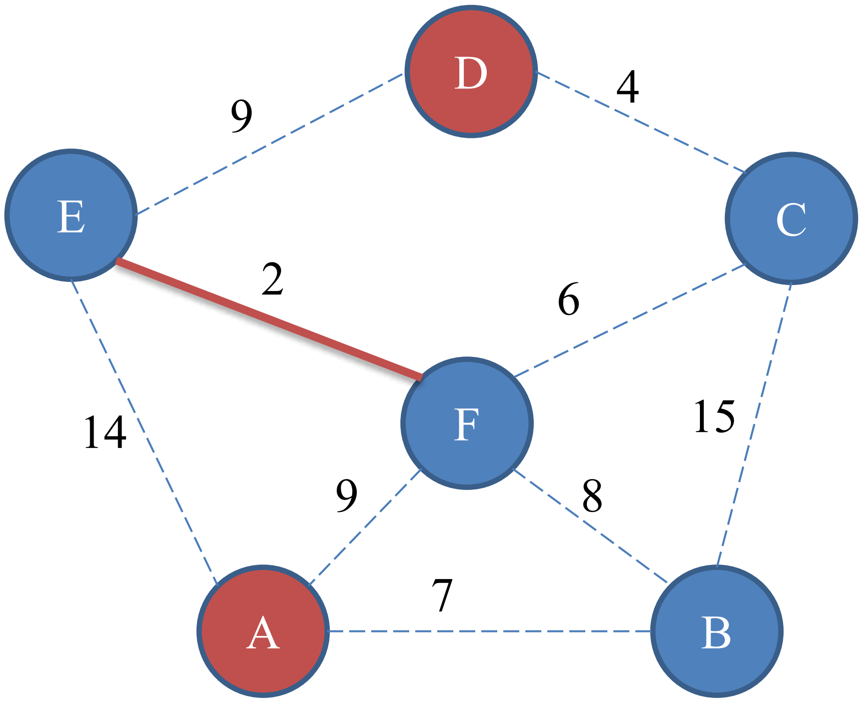
- Natural greedy approach:
  - The idea is that one can make the most progress to the goal by always choosing the best choice available at each point in time.
  - Since we are concerned about the length of the roads between cities, we could construct an algorithm that selects roads from the map based on the shortest path road length.
  - Once we have a path between the start and finish cities we can stop selecting roads and then compute the total distance between the start and finish cities.

Example



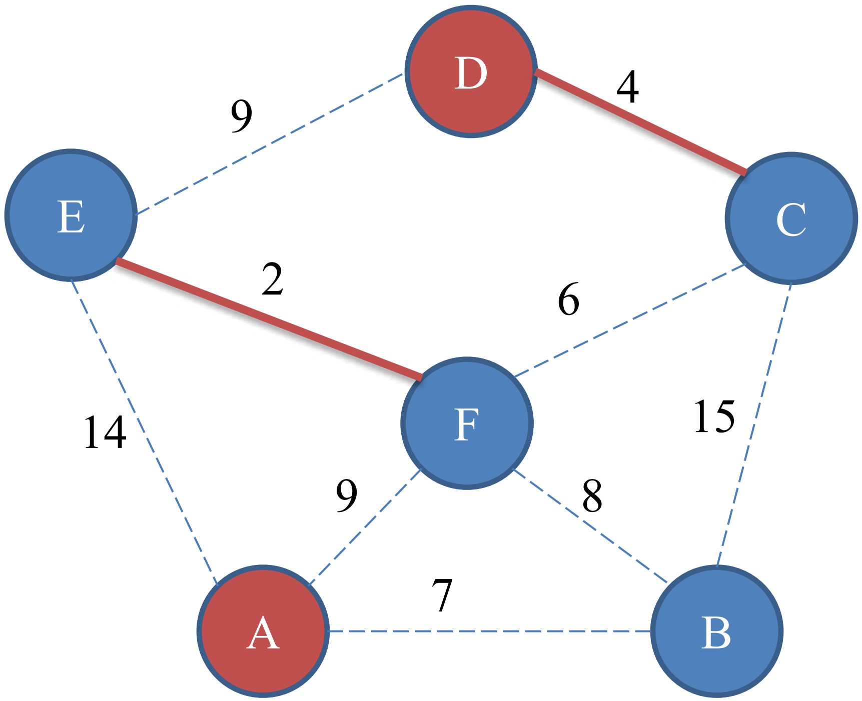
Initially no roads in the map are selected.

Example



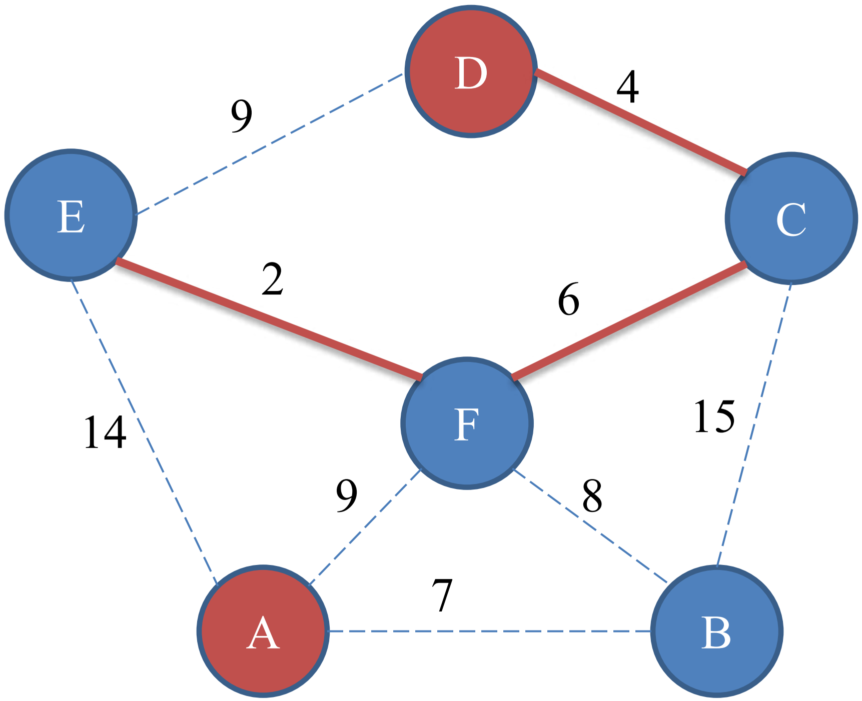
The road between E and F is selected first because it is the shortest.
No path between A and D exists – continue selecting

Example



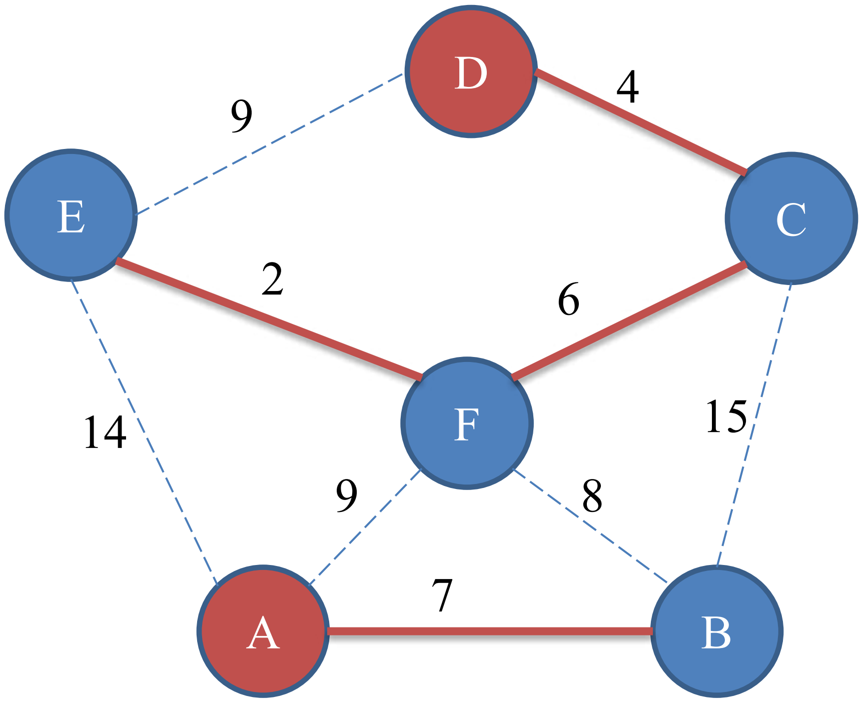
The road between C and D is selected next shortest path. No path between A and D exists – continue selecting

Example



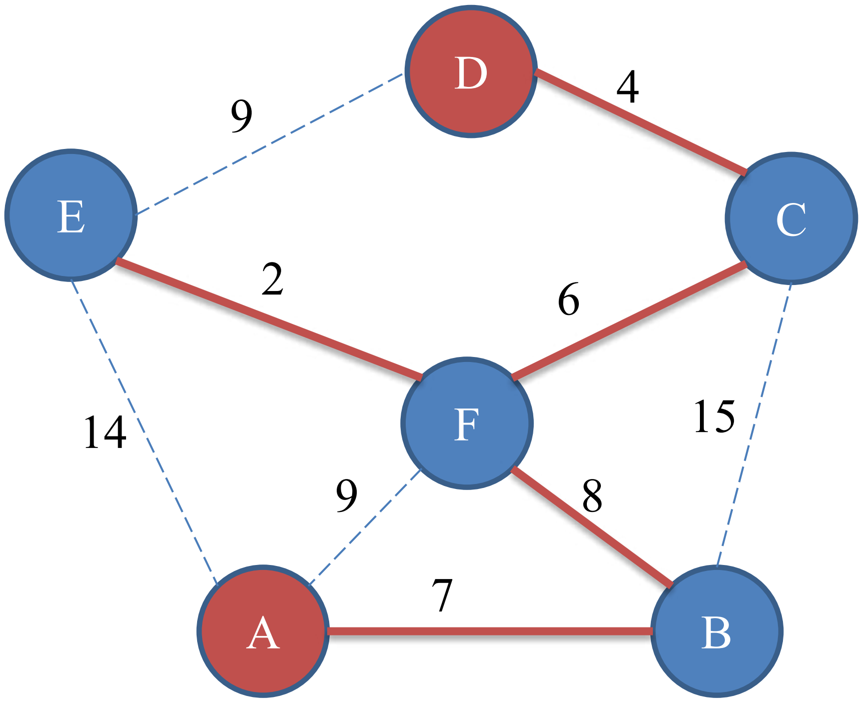
The road between C and F is selected as next shortest path. No path between A and D exists – continue selecting

Example



The road between A and B is selected as next shortest path. No path between A and D exists – continue selecting

Example



The road between B and F is selected as next shortest path. A path between A and D exists; (A, B, F, C, D): Total distance = 25

Analysis of previous approaches

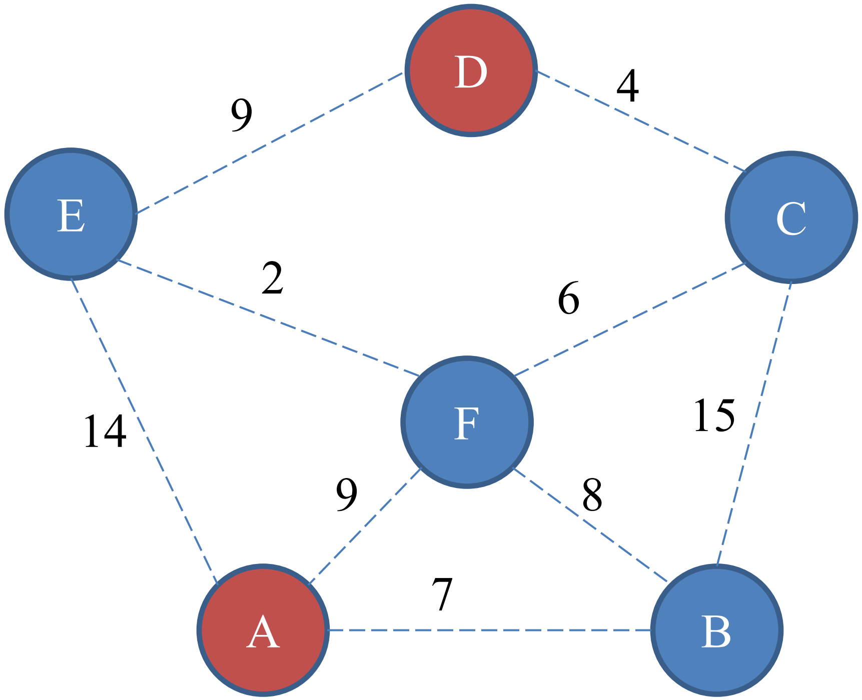
- Greedy approach: Conclusion
  - This approach also produces a non-optimal result
    - because it does not take into account the total sum of the roads needed to connect the two cities.
  - There are other greedy approaches, such as choosing the shortest road out of the current city and following it.
    - But this is also not optimal and can be easily verified.
- Note: However, one greedy approach fails, it doesn’t mean that all of them will.

Dijkstra’s Algorithm

- Main idea of this algorithm is thinking optimally
- The logic behind Dijkstar’s algorthmis based on the principle of Optimality.

“In an optimal sequence based on choices, each contiguous subsequence must also be optimal.”

Dijkstra's Shortest Path



- If we assume that C is a vertex that is part of the overall minimal path, then not only is the cost between A and C optimal (minimum cost), but all other vertices before C in the minimal path are also optimal with respect to A.

Dijkstra’s Algorithm

- Algorithm:

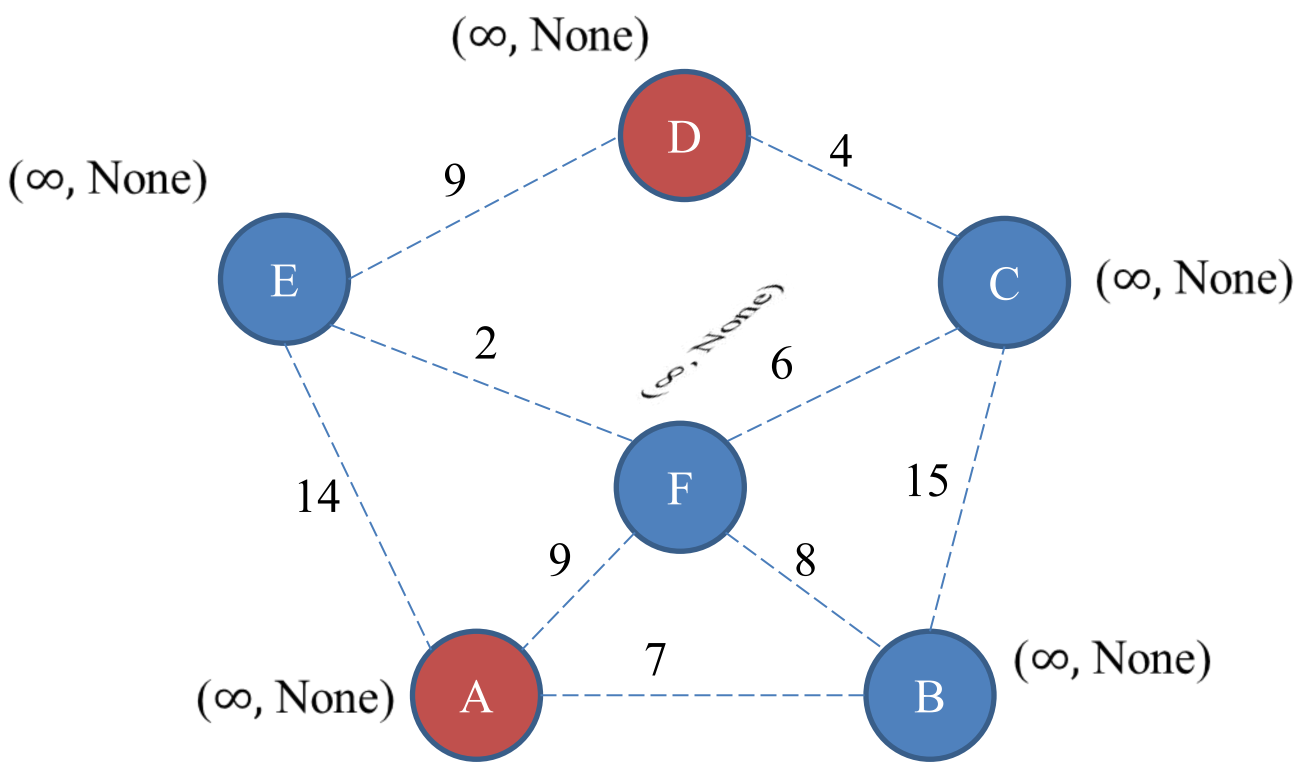
  Let S be the set of explored nodes.

  For each u belongs set of explored nodes.

For each u belongs S, store distance d[u]

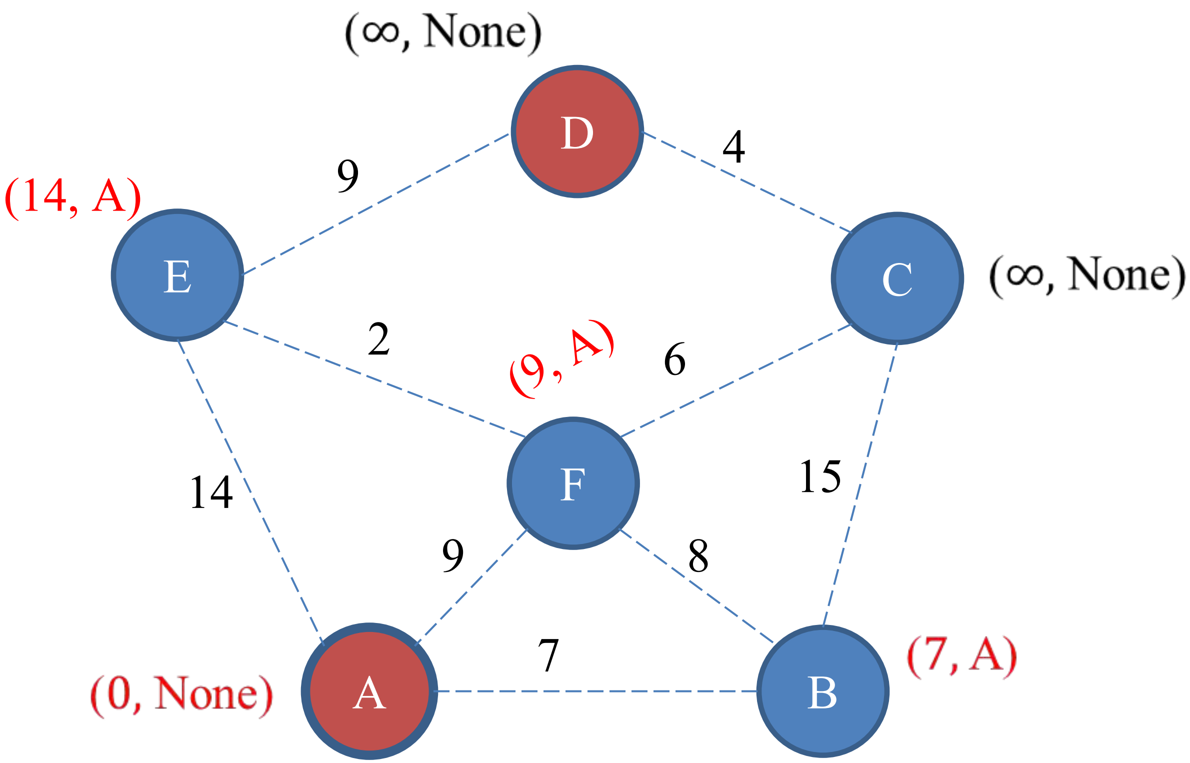
  Initilly

Example



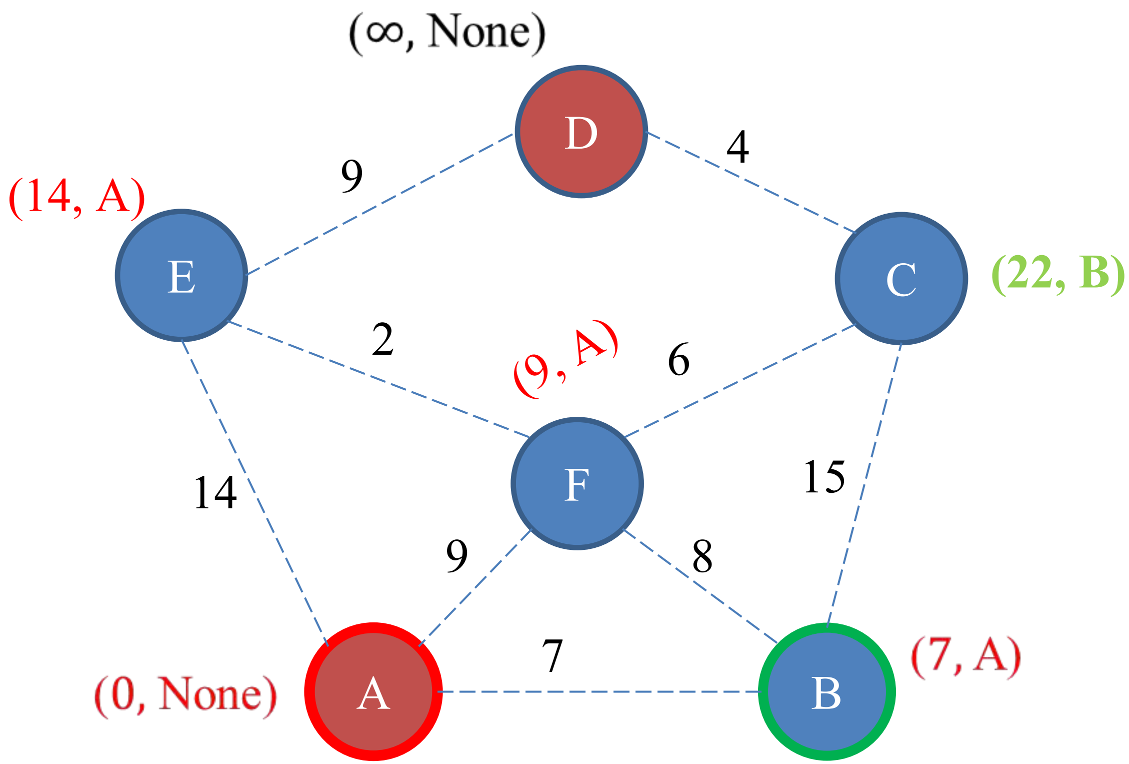
- Initial vertex is A and marked with (∞, None), all other vertices are also marked (∞, None)

Example



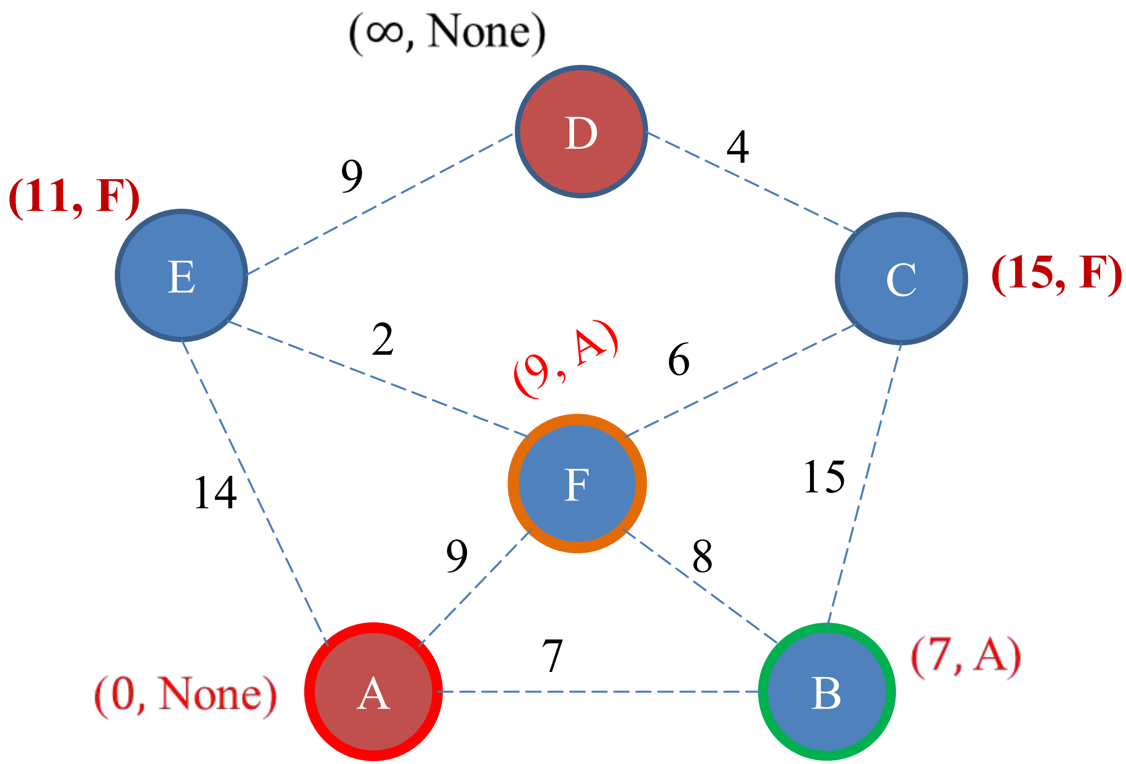
- Current vertex is A.
- A's neighbours are updated.
- A is finalized.

Example



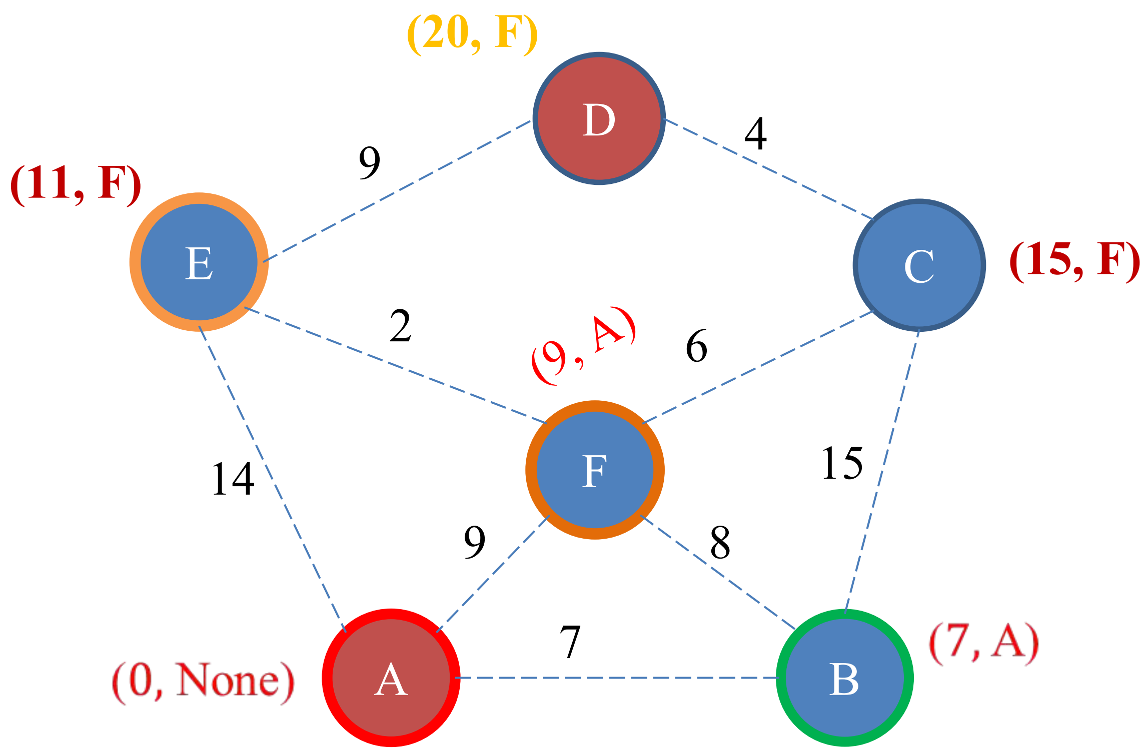
- Current vertex is B.
- B’s neighbours are updated.
- B is finalized.

Example



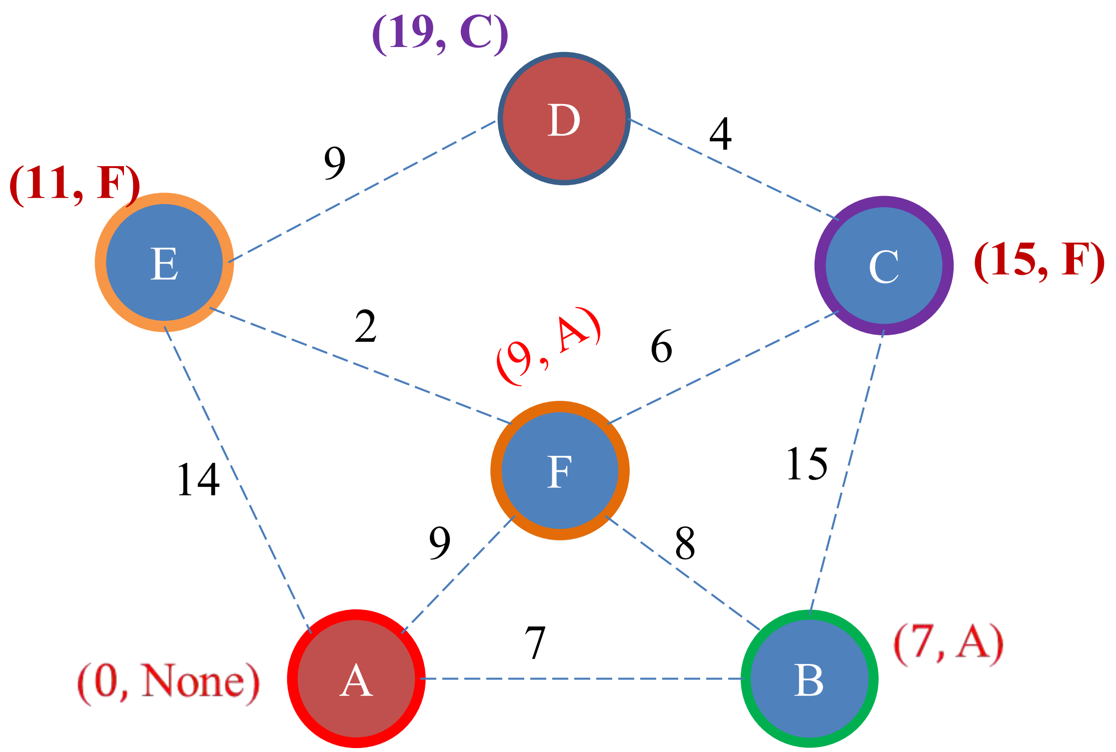
- Current vertex is F.
- F’s neighbours are updated.
- F is finalized.

Example



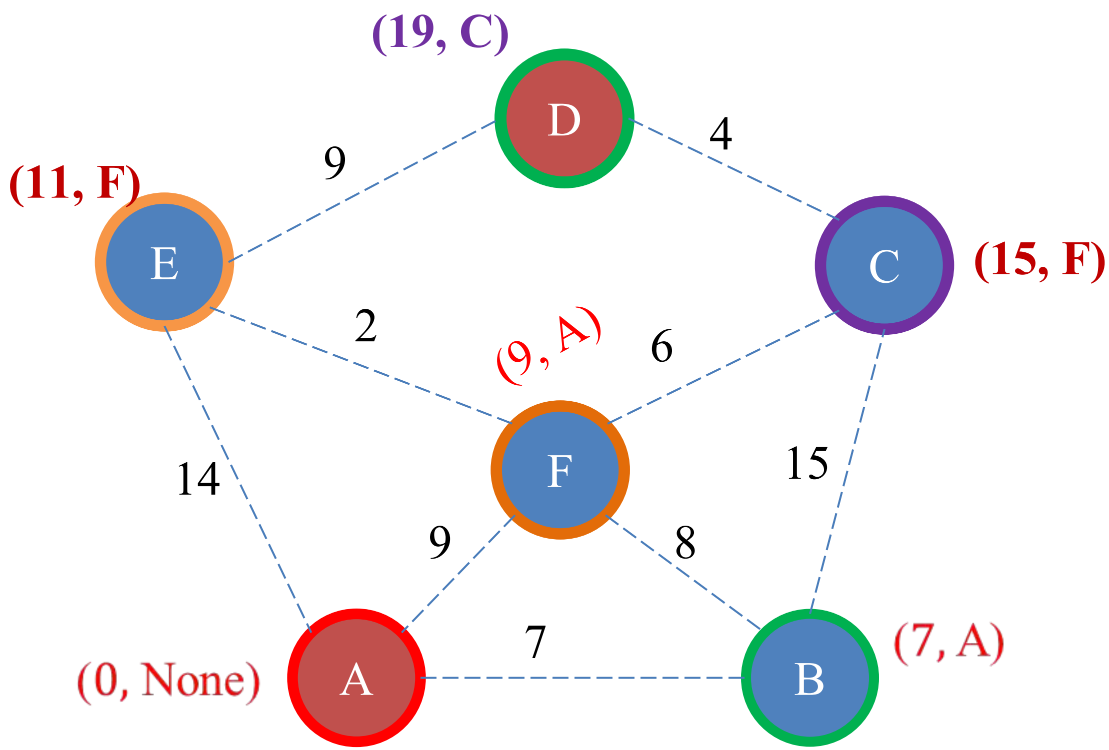
- Current vertex is E.
- E’s neighbours are updated.
- E is finalized.

Example



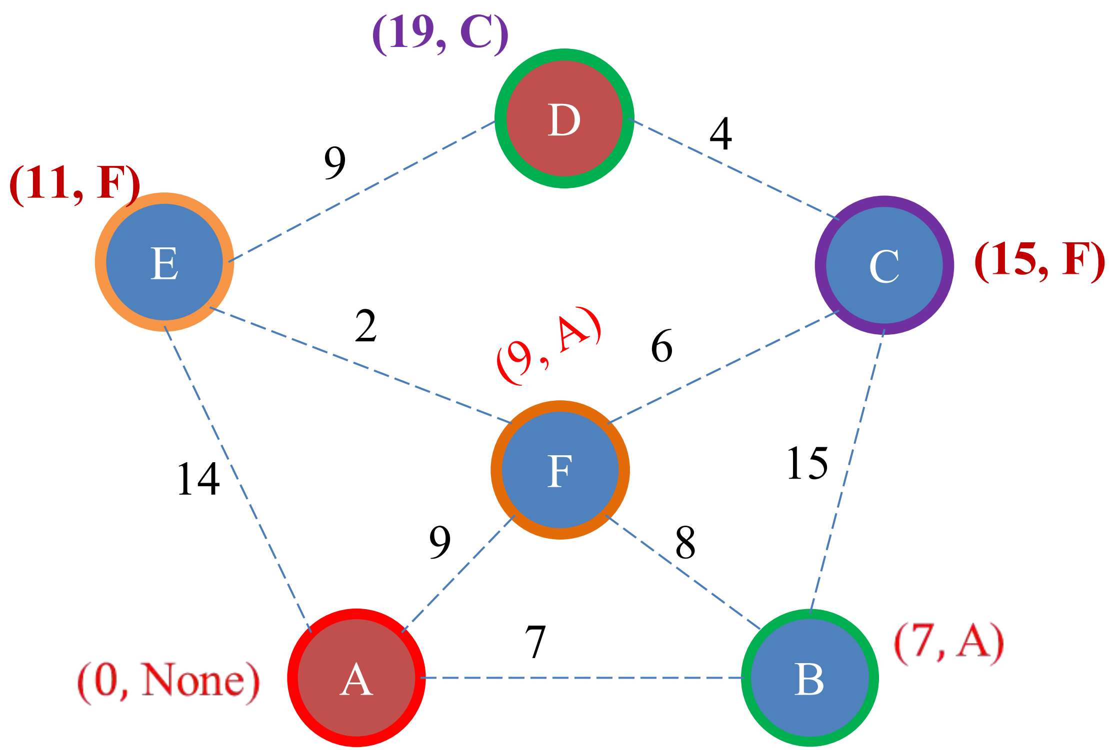
- Current vertex is C.
- C’s neighbours are updated.
- C is finalized.

Example



- Current vertex is D.
- D's neighbours are updated.
- D is finalized. **Algorithm done.**

Example



- Minimum cost from A to D is 19
- The path is found by tracing backwards from D to A using the predecessor's: (D, C, F, A).
- Reverse this to get (A, F, C, D).

Dijkstra’s Algorithm

```
Dijkstra(G, init)
     #initialization
     for every vertex v in G:
        Set its distance to infinity and predecessor to None
    set init’s distanceto 0.
    # S holds the non-initialized vertices
    Create a data structure S that contains all the vertices in G
    #main loop
     while S is not empty:
        Let U= vertex iin S with the smallest distance value
        If the distance of U is infinity:
                break;
        For each neighbour V of U
                Let distThroughU = U’s distance+ distance from U to V
                If  distThroughU < V’s distance:
                        update V’s distance to distThroughU
                        Set V’s parent to U;
                remove from S.
      Return the distances and the parents for every vertex.
```

Running time complexity

- The main loop runs once for each vertex, so that starts us with a factor on $|V|$.
- The first thing that happens in the loop is that vertex is removed from the priority queue. This requires $|V|^2$.
- In the next part of the algorithm “for each neighbor”, each edge is visited exactly twice, once form each direction. This requires $|E|$.
- So total running time is $O(|V|^2+|E|)$.

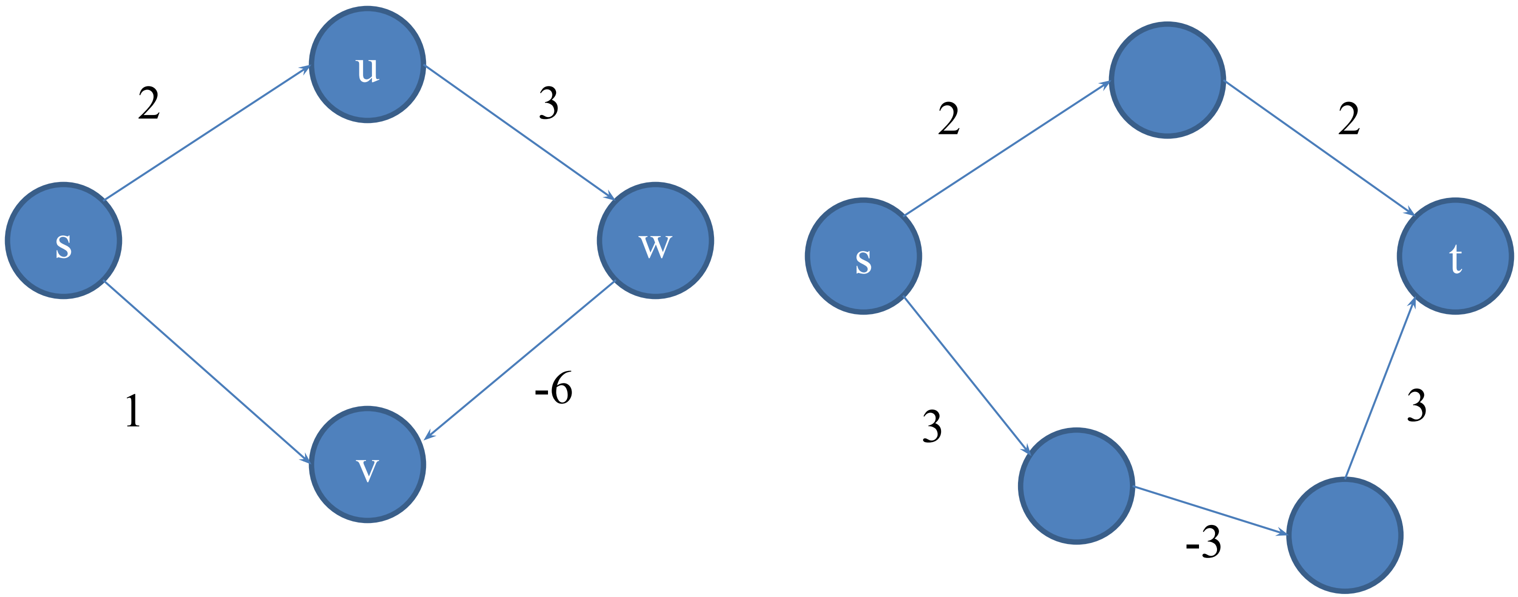
Dijkstra's Algorithm

- Main idea of this algorithm is thinking optimally
- The logic behind Dijkstar's algorthmis based on the principle of Optimality.

"In an optimal sequence based on choices, each contiguous subsequence must also be optimal."

But this idea will not work if we have negative edges.

Example



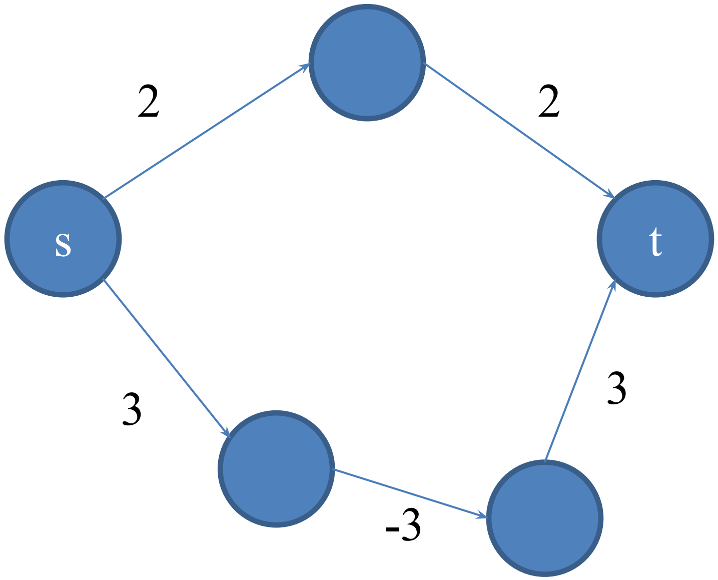
In any case Dijkstra can give the shortest cost path

Dijkstra’s Algorithm

- Natural Idea:
  - The natural idea is to first modify the costs Cij by adding some large constant M to each.
    - i.e. Cij’ = Cij +M for each edge (I, j) belongs to M.
  - If the constant M is large enough, then all modified costs are non-negative.
    - So we can apply Dijkstra’s algorithm
  - But this approach also fails to find the correct shortest path with respect to the original cost.
  - The main problem is that changing the costs from C to C’ changes the minimum cost path.

Example

- After adding 3 to each edge, the shortest path raises to the path different than the minimum cost path originally.



Graph with –ve edges : How to get the shortest paths?

- Some edges may have negative weights

- If there is a negative cycle reachable from s:
  - Shortest path is no longer well-defined
  - Example

- Otherwise, it is fine

Cycles

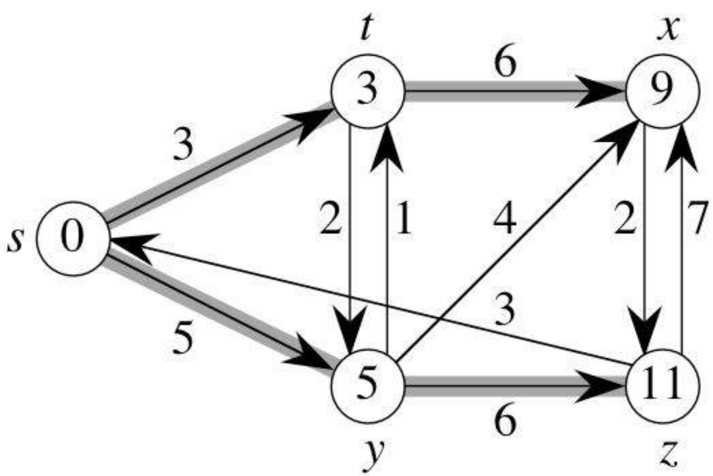
- Shortest path cannot have cycles inside
  - Negative cycles : already eliminated
  - Positive cycles: can be removed
  - 0-weight cycles: can be removed

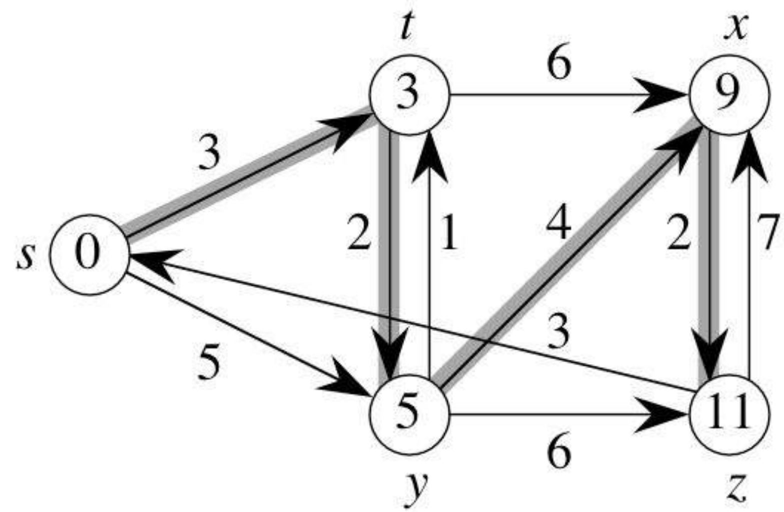
- So each shortest path does not have cycles

Shortest-paths Tree

- For every node $v \in V$, $\pi[v]$ is the predecessor of v in shortest path from source s to v
  - Nil if does not exist
- All our algorithm will output a shortest-path tree
  - Root is source s
  - Edges are $(\pi[v], v)$
  - The shortest path between s and v is the unique tree path from root s to v.

Example





Goal:

- Input:
  - directed weighted graph $G = (V, E)$, source node $s \in V$
- Output:
  - For every vertex $v \in V$,
    - $\boxed{d[v]} = \delta(s, v)$
    - $\pi[v]$
  - Shortest-paths tree induced by $\pi[v]$

Intuition

- Compared to Breadth-first search
- Main difference?

Basic Operation: Relaxation

- Maintain shortest-path estimate *d[v]* for each node

- RELAX(u, v, w)

  if $d[v] > d[u] + w(u, v)$

  then $d[v] \leftarrow d[u] + w(u, v)$

  $\pi[v] \leftarrow u$

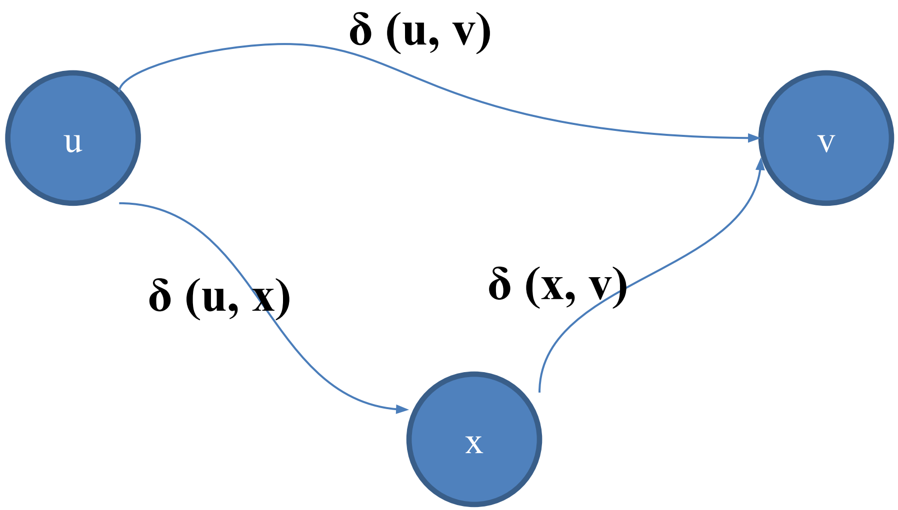
Intuition:
Do we have a
shorter path
if use edge (u,v) ?

Algorithms will repeatedly apply Relax.
Differ in the order of Relax operation

Triangle inequality

Theorem: For all $u, v, x \in V$, we have $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$

Proof:



The sum of any two sides must be greater than the third side

Upper bound property

Upper Bound property:

- **Always have $d[v] \geq \delta(s, v)$ for all v.**
- **Once $d[v] = \delta(s, v)$, it never changes.**

Proof: Initially true. Suppose there exists a vertex such that $d[v] < \delta(s, v)$.

Without loss of generality *v* is the first vertex for which this happens. Let u be the vertex that causes d[v] change. Then $d[v] = d[u]+w(u,v)$.

So

$$
\begin{aligned}
d[v] &< \delta(s, v) \\
&\leq \delta(s, u) + w(u, v) \\
&\leq d[u] + w(u, v)
\end{aligned}
$$

which implies $d[v] < d[u] + w(u, v)$. Contradicts $d[v] = d[u] + w(u, v)$. Once d[v] reaches $\delta(s, v)$, it never goes lower. It never goes up, since relaxations only lower shortest-path weights.

Properties cont.

- **Convergence property**
  - Suppose $s \Rightarrow u \rightsquigarrow v$ is the shortest path from s to v
  - Currently $d[u] = \delta(s, u)$
  - Relax (u,v) will set $d[v] = \delta(s, v)$

Proof: After relaxation

$$\begin{aligned} d[v] &\le d[u] + w(u, v) \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) \end{aligned}$$

On the other hand, we have $d[v] \ge \delta(s, v)$. Therefore, it must have $d[v] = \delta(s, v)$.

Properties cont...

Path Relaxation Property:

Let $p = v_0 \rightarrow v_1 \rightarrow \cdot \cdot \cdot \rightarrow v_k$ be a shortest-path. If we relax in order, $(v_0, v_1),(v_1, v_2), \dots,(v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(v_0, v_k)$.

Proof: Induction to show $d[v_i] = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.

- Basis step: $i = 0$. Initially $d[v_0] = \delta(s, v_0) = \delta(s, s)$
- Inductive step: Assume $d[vi-1] = \delta(s, v_{i-1})$. Relax (v_{i-1}, v_i).
- By convergence property, $d[v_i] = \delta(s, v_i)$ afterward and $d[v_i]$ never changes.

Bellman-Ford Algorithm

- Allow negative weights
- Follow the frame work that first, initialize:

$$\text{INIT-SINGLE-SOURCE}(V, s)$$

for each $v \in V$
    do $d[v] \leftarrow \infty$
        $\pi[v] \leftarrow \text{NIL}$
$d[s] \leftarrow 0$

- Then apply a set of Relax
  - compute *d[v]* and *π[v]*
  - Return *False* if there exists negative cycles

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| − 1
        do for each edge (u, v) ∈ E[G]
                do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
        do if d[v] > d[u] + w(u, v)
                then return FALSE
return TRUE
```

Pseudo-code

BELLMAN-FORD(V, E, w, s)

INIT-SINGLE-SOURCE(V, s)

for $i \leftarrow 1$ to $|V| - 1$

  do for each edge $(u, v) \in E$

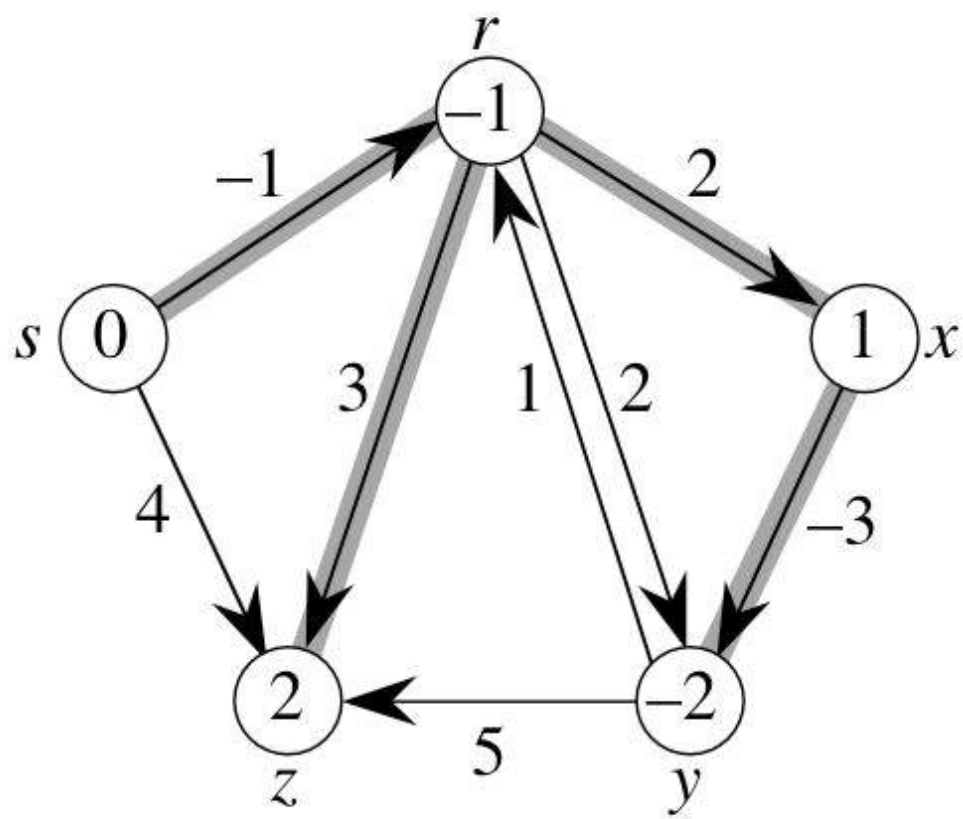
    do RELAX(u, v, w)

for each edge $(u, v) \in E$

Time complexity:
O(VE)

return TRUE

Example



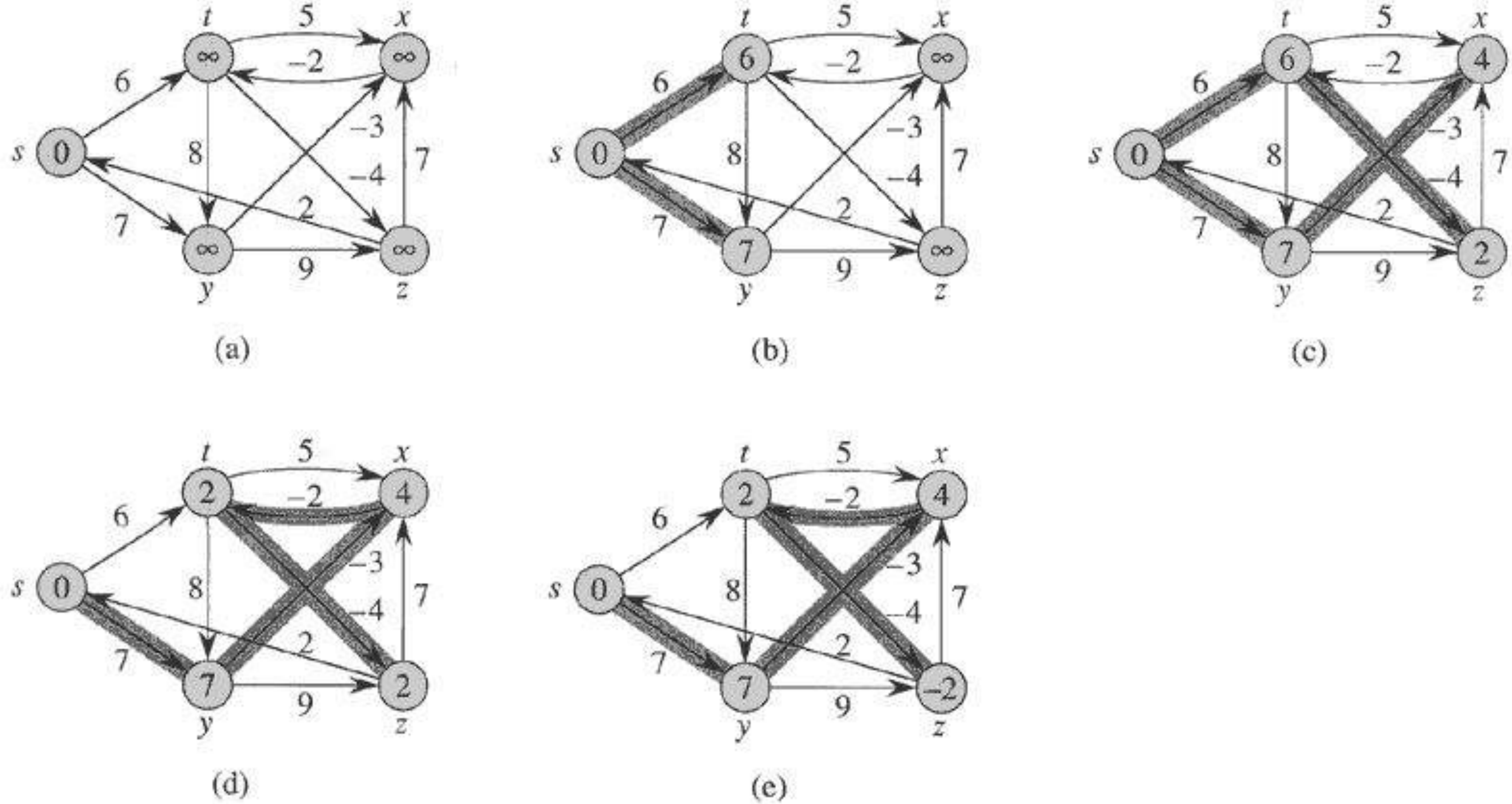


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Single source shortest path for DAG

- There is no cycle in a DAG. Hence, no negative-weight cycle can exists in a DAG, and shortest paths are well defined.
- Single source shortest paths problem for DAG can be solved more efficiently by using topological sort.

Single source shortest path for DAG

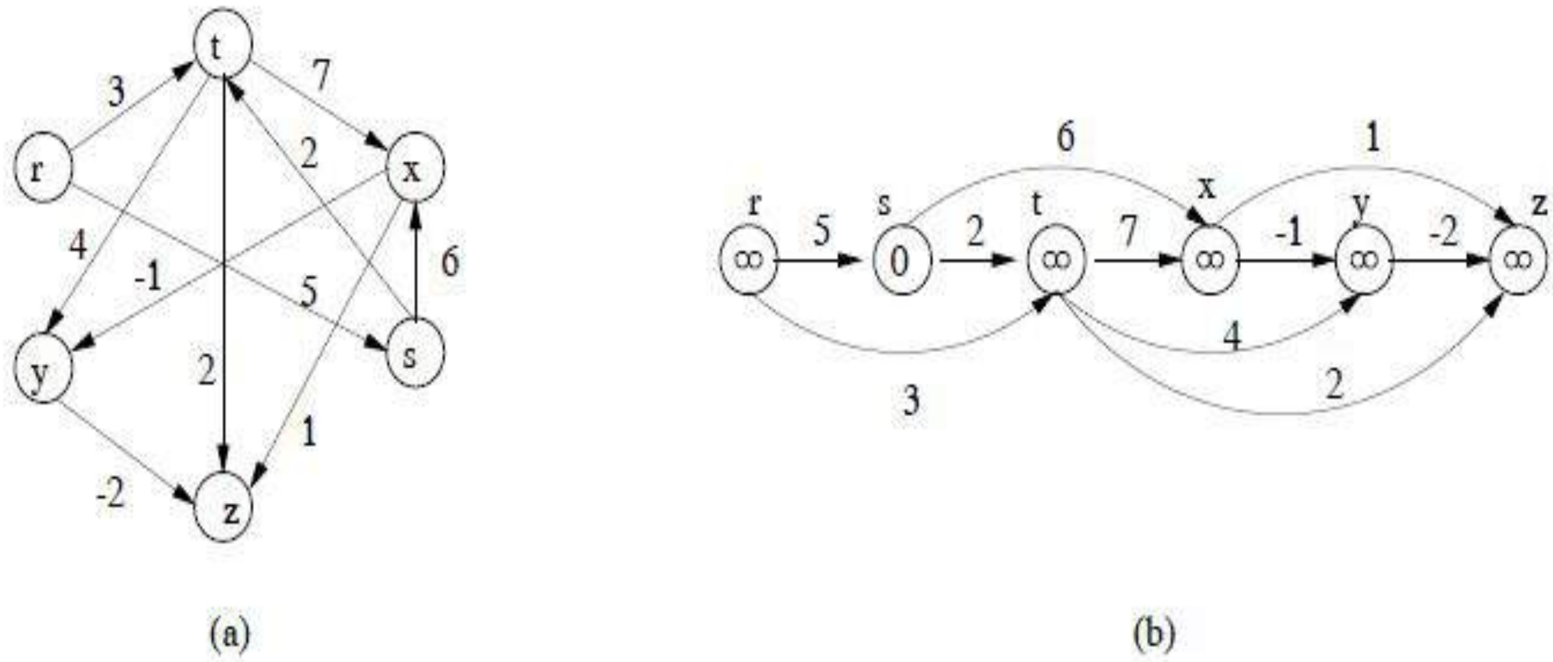
- **Algorithm**

Procedure DAG-Shortest-Paths(G, s, w)

```
{
 Do Topological sort(G);
 Initialize-single-source(G,s);
 S=NULL
 for each node u, taken in topologically sorted order do
  {
   for each node v ∈Adj[u] do RELAX(u, v, w)
   S=S∪ {u}
  }
}
```

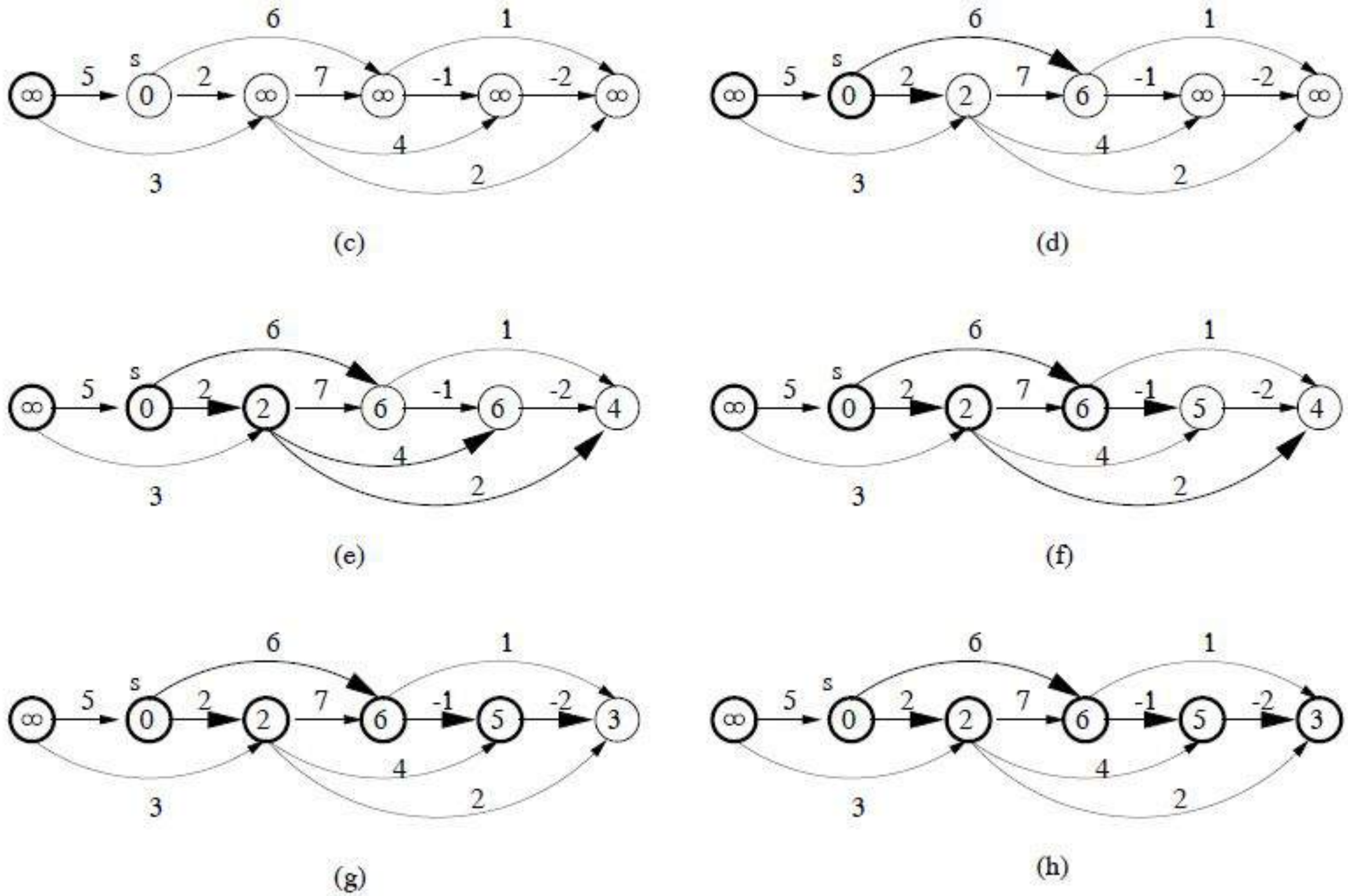
Complexity: O(V+E)

Example



DAG and its corresponding topological sort

Example



6 iterations corresponding to RELAX operations

Floyd-Warshall's Algorithm

All pairs shortest path

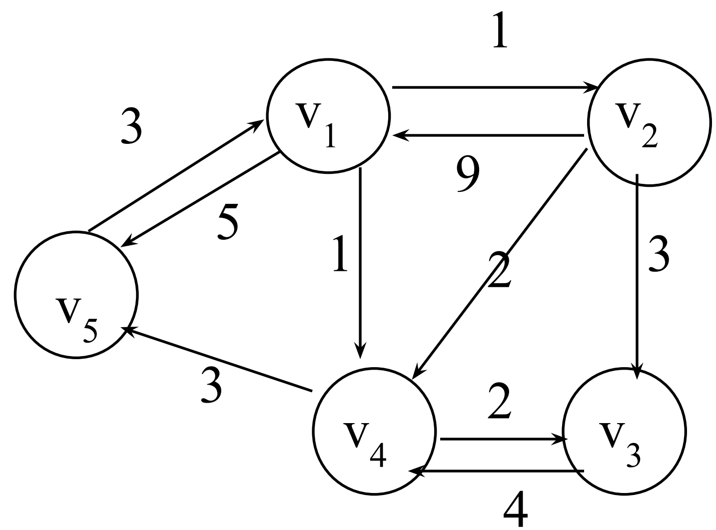
All pairs shortest path

- ***The problem:*** **F**ind the shortest path between every pair of vertices of a graph
- ***The graph*: M**ay contain negative edges but ***no negative cycles***
- ***A representation*:** A weight matrix where
  W(i,j)=0, if i=j.
  W(i,j)= ∞ ,if there is no edge between i and j.
  W(i,j)="*weight of edge*"

Note: we will also apply principle of optimality to shortest path problems

The weight matrix and the graph

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0



The subproblems

- How can we define the shortest distance $d_{i,j}$ in terms of "smaller" problems?
  - One way is to restrict the paths to only include *vertices from a restricted subset.*
  - Initially, the subset is empty.
  - Then, it is incrementally increased until it includes all the vertices.

The subproblems

- **Let $D^{(k)}[i,j]$ =** weight of a shortest path from v_i to v_j using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices in the path
  - $D^{(0)}=W$
  - $D^{(n)}=D$ **which is the goal matrix**

- **How do we compute $D^{(k)}$ from $D^{(k-1)}$?**

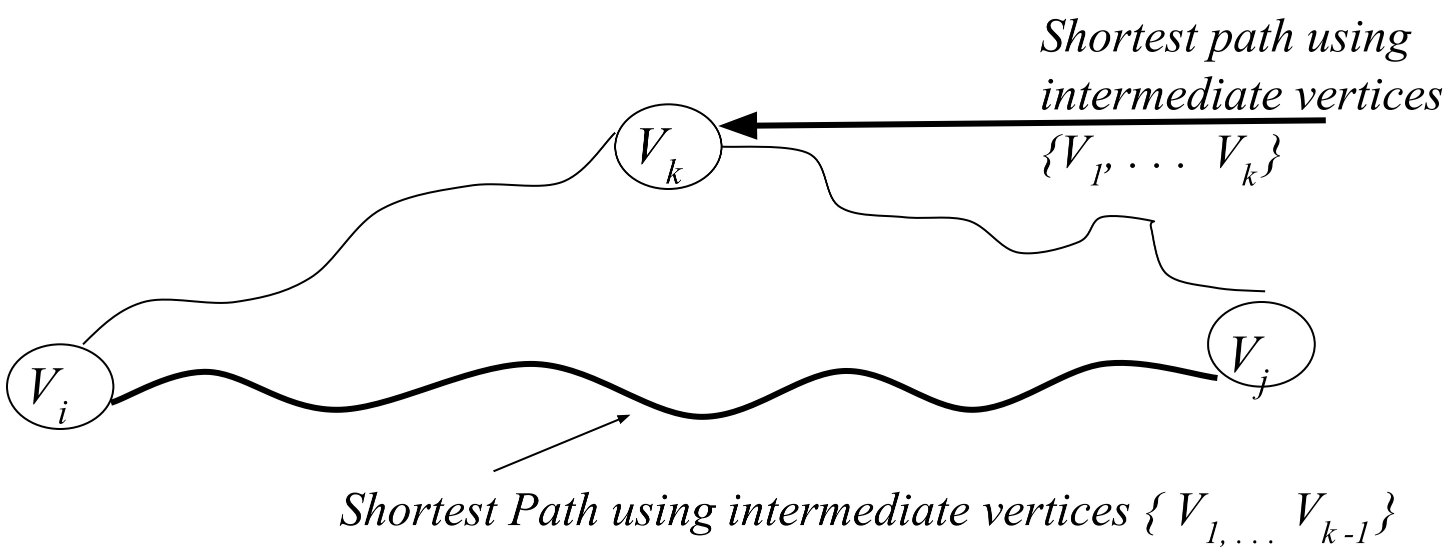
The Recursive Definition

Case 1: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices **does not use v_k.**

Then $D^{(k)}[i,j] = D^{(k-1)}[i,j]$.

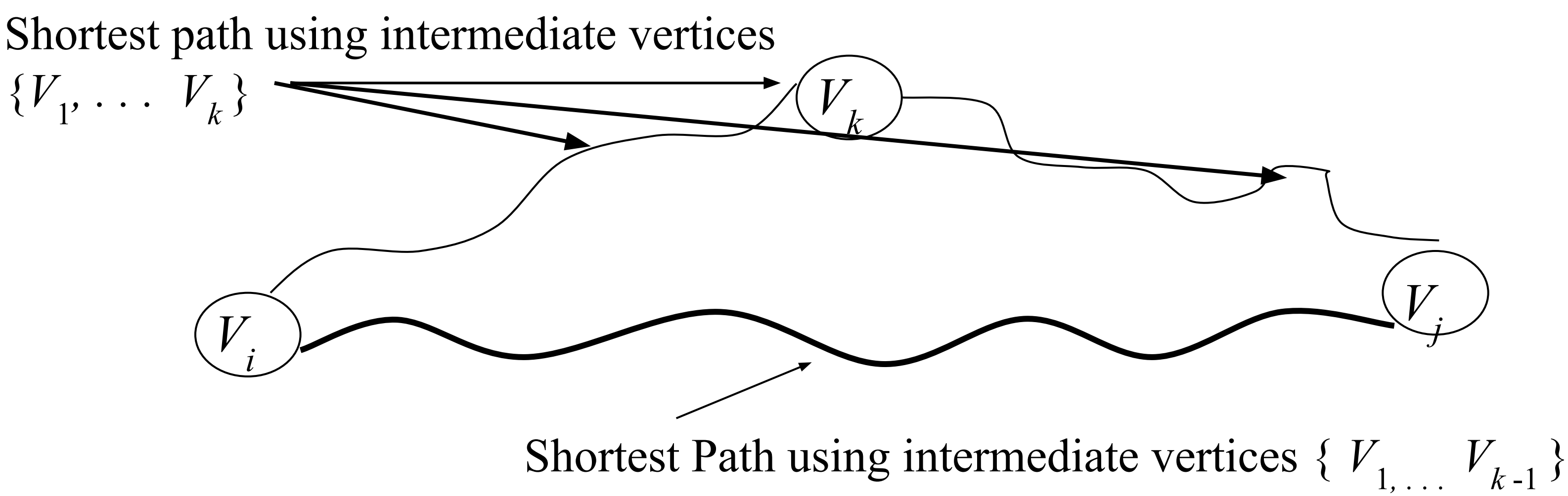
Case 2: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices **does use v_k.**

Then $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$.



The recursive definition

- **Since**
  $D^{(k)}[i,j] = D^{(k\text{-}1)}[i,j]$ or
  $D^{(k)}[i,j] = D^{(k\text{-}1)}[i,k] + D^{(k\text{-}1)}[k,j]$.
  We conclude:
  $\boldsymbol{D^{(k)}[i,j] = \min\{ D^{(k\text{-}1)}[i,j],\ D^{(k\text{-}1)}[i,k] + D^{(k\text{-}1)}[k,j] \}}$.



The pointer array P

- Used to enable finding a shortest path
- Initially the array contains 0

- Each time that a shorter path from *i* to *j* is found the *k* that provided the minimum is saved (highest index node on the path from *i* to *j*)

- To print the intermediate nodes on the shortest path a recursive procedure that print the shortest paths from *i* and *k*, and from *k* to *j* can be used

Floyd-Warshall's Algorithm Using (n+1) *D* matrices

Floyd-Warshall

//Computes shortest distance between all pairs of

//nodes, and saves P to enable finding shortest paths

1. $D^0 \leftarrow W$ // initialize *D* array to *W* [ ]

2. $P \leftarrow 0$ // initialize P array to [0]

3. **for** $k \leftarrow 1$ **to** n

4. **do for** $i \leftarrow 1$ **to** n

5. **do for** $j \leftarrow 1$ **to** n

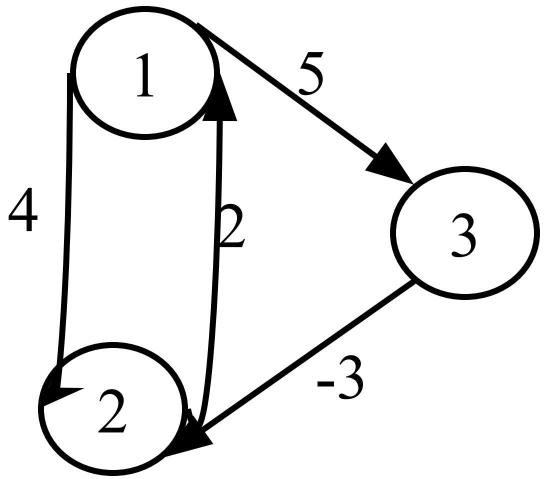
6. **if** $(D^{k-1}[\, i, j\,] > D^{k-1}[\, i, k\,] + D^{k-1}[\, k, j\,])$

7. **then** $D^{k}[\, i, j\,] \leftarrow D^{k-1}[\, i, k\,] + D^{k-1}[\, k, j\,]$

8. $P[\, i, j\,] \leftarrow k;$

9. **else** $D^{k}[\, i, j\,] \leftarrow D^{k-1}[\, i, j\,]$

Example

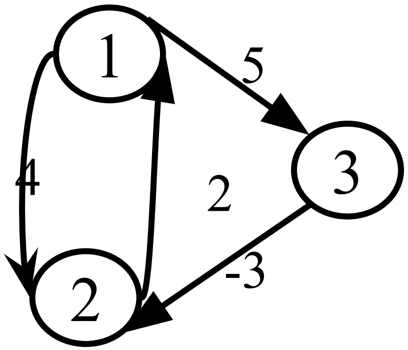


$W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$P =$

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



$$D^0 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & \infty \\ 3 & \infty & -3 & 0 \end{array}$$

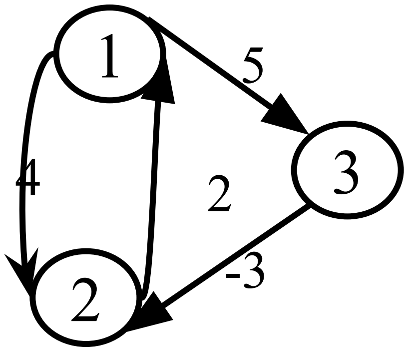
k = 1
Vertex 1 can be intermediate node

$$D^1 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & \infty & -3 & 0 \end{array}$$

$$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$$
$$= \min (\propto, 7)$$
$$= 7$$

$$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$$
$$= \min (-3, \propto)$$
$$= -3$$

$$P = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 \end{array}$$



$$D^1 = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & \infty & -3 & 0 \\ \hline \end{array}$$

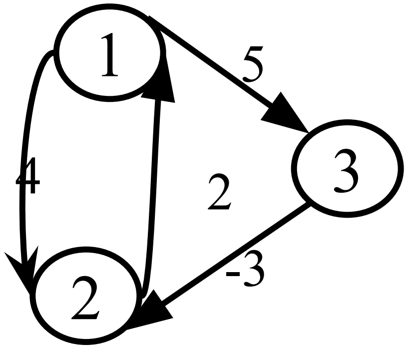
k = 2
Vertices 1, 2 can be intermediate

$$D^2 = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & -1 & -3 & 0 \\ \hline \end{array}$$

$$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3])$$
$$= \min (5, 4+7)$$
$$= 5$$

$$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1])$$
$$= \min (\propto, -3+2)$$
$$= -1$$

$$P = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 2 & 0 & 0 \\ \hline \end{array}$$



$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

k = 3
Vertices 1, 2, 3 can be intermediate

$D^3 =$

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2])$
$= \min(4, 5+(-3))$
$= 2$

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$
$= \min(2, 7+(-1))$
$= 2$

P =

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

Can we use only one D matrix?

- $D[i,j]$ depends only on elements in the kth column and row of the distance matrix.
- We see that the kth row and the kth column of the distance matrix are unchanged when D^k is computed
- This means D can be calculated *in-place*

The main diagonal values

- Before we show that k-th row and column of D remain unchanged we show that the main diagonal remains 0

- $$\begin{aligned} D^{(k)}[\,j,j\,] &= \min\{\, D^{(k\text{-}1)}[\,j,j\,],\;\; D^{(k\text{-}1)}[\,j,k\,] + D^{(k\text{-}1)}[\,k,j\,] \,\} \\ &= \min\{\, 0,\;\; D^{(k\text{-}1)}[\,j,k\,] + D^{(k\text{-}1)}[\,k,j\,] \,\} \\ &= 0 \end{aligned}$$
- ***Based on which assumption?***

-- No negative weight cycle.

The *k-th* column

- ***K*-th column of D^k is equal to the *k*-th column of $D^{k\text{-}1}$**

- ***Intuitively true -*** a path from i to k will not become shorter by adding k to the allowed subset of intermediate vertices

- ***For all i,***

$$\mathbf{D^{(k)}[i,k]} = \min\{ D^{(k-1)}[i,k],\ D^{(k-1)}[i,k]+ D^{(k-1)}[k,k] \}$$
$$= \min \{ D^{(k-1)}[i,k], D^{(k-1)}[i,k]+0 \}$$
$$= D^{(k-1)}[i,k]$$

The *k*-th row

- ***K*-th row of D^k is equal to the *k*-th row of $D^{k\text{-}1}$**

For all *j*, $D^{(k)}[k,j] =$

$$= \min\{\, D^{(k-1)}[k,j],\ D^{(k-1)}[k,k] + D^{(k-1)}[k,j] \,\}$$

$$= \min\{\, D^{(k-1)}[\,k,j\,],\ 0 + D^{(k-1)}[k,j\,] \,\}$$

$$= D^{(k-1)}[\,k,j\,]$$

Exercise