Divide and conquer (D&C) analysis

- General method
- Solving recurrence equation
- Master theorem
- Recursion tree

General method

- Given a function to compute on n inputs, the divide and conquer strategy suggests splitting the input in to k distinct subsets, yielding k subproblems.
- These sub-problems must be solved, then a method must be found to combine sub-solutions into a solutions of the whole.
- Often the subproblems are of <u>the same type </u>as the original problem. If the sub-problems are still large, apply D&C to the sub-problem (use <u>recursion algorithm</u>).
- Smaller and smaller subproblems are produced until the problem is small enough, which can be solved without splitting.

```
Algorithm D&C(P)
    if Small(P) return Solve(P)
    else{
            Divide P into smaller instances $P_1, P_2, \dots P_k$
            Apply D&C to each of these subproblems
        return Combine(D&C($P_1$), D&C($P_2$), ...,D&C($P_k$))
    }
```

- *Small(P)* is a Boolean valued function that determines whether the problem is small enough.
- If yes, the function *Solve(P)* is invoked.
- otherwise, each of subproblems is solved by *D&C* algorithm.
- *Combine* is a function that determines the solution of P using the solutions to k sub-problems.

- ➢ Recursion algorithm summary
  a) needs a simple case (to terminate)
  b) a recursive function calling itself
  c) Harder to program as it needs deeper understanding than iterative algorithms
  d) Simpler (shorter) code.

Matrix multiplication

Strassen Matrix multiplication

Convex hull

Master theorem et al

Divide and Conquer

max-min

Selection

Multiplication of two integers

Recurrence equation

- Typically in D&C we have complexity equations of the nonlinear non-homogenous type

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- Problem is divided into \`$a$' similar problems of size \` $\frac{n}{b}$ ', and $f(n)$ is the non-recursive cost.
- which is to split the problem or combine solutions of subproblems.

➢ Solving recursion in math

Example: Solve $x(n)= x(n-1)+n$, base case $x(0)=4$

The equation is for any n, so

$$x(n)- \cancel{x(n-1)}=n$$
$$\cancel{x(n-1)}- \cancel{x(n-2)}=n-1,$$
$$\cdots\cdot$$
$$\cancel{x(2)}- \cancel{x(1)}=2,$$
$$\cancel{x(1)}- x(0)=1,$$

Adds up $x(n)- x(0) = (1+2+ \ldots n) =n(n+1)/2$

$x(n)= n(n+1)/2 +4$

- In class exercise:

  Verify $x(n)=2^n+1$

  is a solution to $x(n)=2x(n-1)-1$

  with $x(1)=3$

Check : $x(1) =$ [redacted]

$x(n-1) =$ [redacted]

$2x(n-1)-1 =$ [redacted]

Substitution

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- In D&C analysis, we use the method of substitution to find out a closed form expression
- Repeatedly substitute occurrences of $T(\,)$ on the right side until all such occurrences disappear.

- e.g. $a = 2$; $b = 2$, $f(n) = n$ and $T(1) = 2$, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$
$$= 4T\left(\frac{n}{4}\right) + 2n$$
$$= 8T\left(\frac{n}{8}\right) + 3n$$
$$= \dots \dots$$

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- $T(n) = 2T\left(\frac{n}{2}\right) + n$

  $= 2(2T\left(\frac{n}{4}\right) + \frac{n}{2}) + n$

  $= 4T\left(\frac{n}{4}\right) + 2n \ = 8T\left(\frac{n}{8}\right) + 3n = \dots\dots$

- We observe that, for $1 \le k \le \log_2 n$

$$T(n) = 2^k * T\left(\frac{n}{2^k}\right) + kn$$
$$= nT(1) + \ n\log_2 n$$
$$= 2n + \ n\log_2 n$$

Hint: $n = 2^k$

In class exercise: Using substitution to find the closed expression for following recursions

1. $T(n) = T\left(\frac{n}{2}\right) + C$
2. $T(n) = 2T\left(\frac{n}{2}\right) + n$
3. $T(n) = 8T\left(\frac{n}{2}\right) + Cn$
4. $T(n) = 4T\left(\frac{n}{3}\right) + Cn^2$

[redacted]

In class exercise: Using substitution to find the closed expression for following recursions

Hint: $n = 2^k$

1. $T(n) = T\left(\frac{n}{2}\right) + C$
2. $T(n) = 2T\left(\frac{n}{2}\right) + n$
3. $T(n) = 8T\left(\frac{n}{2}\right) + Cn$
4. $T(n) = 4T\left(\frac{n}{3}\right) + Cn^2$

[redacted]

In class exercise: Using substitution to find the closed expression for following recursions

Hint: $n = 2^k$

1. $T(n) = T\left(\frac{n}{2}\right) + C$
2. $T(n) = 2T\left(\frac{n}{2}\right) + n$
3. $T(n) = 8T\left(\frac{n}{2}\right) + Cn$
4. $T(n) = 4T\left(\frac{n}{3}\right) + Cn^2$

[redacted]

Hint: $n = 3^k$

In class exercise: Using substitution to find the closed expression for following recursions

1. $T(n) = T\left(\frac{n}{2}\right) + C$
2. $T(n) = 2T\left(\frac{n}{2}\right) + n$
3. $T(n) = 8T\left(\frac{n}{2}\right) + Cn$
4. $T(n) = 4T\left(\frac{n}{3}\right) + Cn^2$

[redacted]

Master theorem

- Consider

  $T(n) = aT\left(\frac{n}{b}\right) + O(n^r)$

  Let $c = \log_b a$ be the critical exponent, then

  $$T(n) = \begin{cases} O(n^c) & \text{if} \quad r < c \\ O(n^c \log n) & \text{if} \quad r = c \\ O(n^r) & \text{if} \quad r > c \end{cases}$$

- Example: Merge sort

  $T(n) = 2T\left(\frac{n}{2}\right) + n$

  a=2, b=2, c=1, r=1, $T(n)$= $O(n^c \log n) = O(n \log n)$

$$T(n) = \begin{cases} O(n^c) & \text{if} \quad r < c \\ O(n^c \log n) & \text{if} \quad r = c \\ O(n^r) & \text{if} \quad r > c \end{cases}$$

- Example: Strassen's Multiplication

$$T(n) = 7T\left(\frac{n}{2}\right) + 18n^2$$

a=7, b=2, c= $\log_2 7$, r=2,

$T(n)$= $O(n^c) = O\left(n^{\log_2 7}\right) = O(n^{2.81})$

- Example: $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

a=4, b=2, c= $\log_2 4$, r=3,

$T(n)$= $O(n^r) = O(n^3)$

Sketch of Proof :

Note: $a^k = a^{\log_b n} = n^{\log_b a}$

From $T(n) = aT\left(\frac{n}{b}\right) + O(n^r)$

- We assume $n = b^k$, then

$$T(n) = aT\left(b^{k-1}\right) + f\left(b^k\right)$$

$$= a(aT\left(b^{\boldsymbol{k-2}}\right) + f\left(b^{\boldsymbol{k-1}}\right)) + f\left(b^k\right)$$

$$= a^2T\left(b^{\boldsymbol{k-2}}\right) + af\left(b^{\boldsymbol{k-1}}\right) + f\left(b^k\right) = \ldots\ldots$$

$$= a^kT(1) + \sum_{i=0}^{k-1} a^i f\left(b^{k-i}\right)$$

$$= n^{\log_b a}T(1) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)$$

$$T(n) = n^{\log_b a} T(1) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right) \Rightarrow T(n) = \begin{cases} O(n^c) & \text{if } r < c \\ O(n^c \log n) & \text{if } r = c \\ O(n^r) & \text{if } r > c \end{cases}$$

- Consider the following cases:

<u>Case 1</u>: First term dominates, drop the second term.

<u>Case 2</u>: Second term dominates, drop the first term. The summation is a geometric series starting with *f(n)*. Hence *T(n)* is proportional to *f(n)*.

<u>Case 3</u>: When each of the term is proportional to each other. *T(n)* is *f(n)* times a log-arithmetic factor.

In class exercise: $c = \log_b a$, $\log_b a = \frac{\log b}{\log a}$

Using Master's theorem to obtain time complexity of the following recursions

1. $T(n) = T\left(\frac{n}{2}\right) + C$
2. $T(n) = 2T\left(\frac{n}{2}\right) + n$
3. $T(n) = 8T\left(\frac{n}{2}\right) + Cn$
4. $T(n) = 4T\left(\frac{n}{3}\right) + Cn^2$

[redacted]

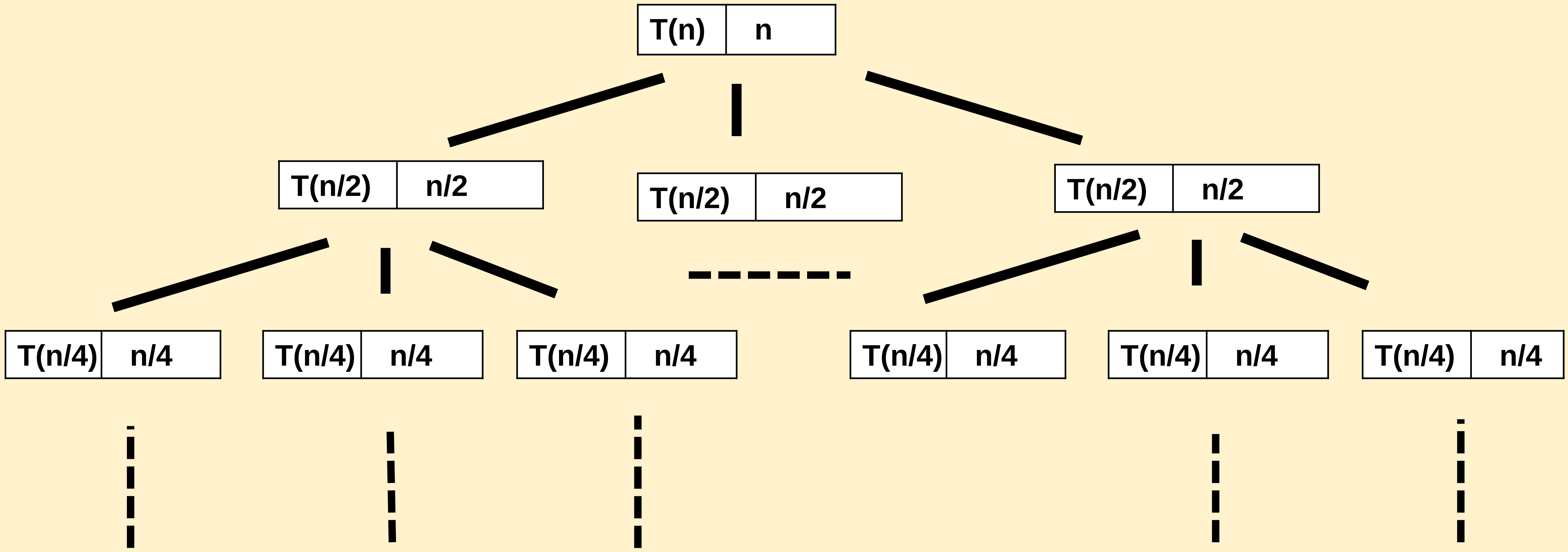
Recursion

- Typically in D&C, problem is divided into `$a$' similar problems of size `$\frac{n}{b}$', and $f(n)$ is the non-recursive cost.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- A recursion tree is a diagram of recursive calls with the amount of work in each call.
- Thus it is useful for visualizing the recursion function call.
- useful only for gaining intuition, but not recommended for use of proof complexity.

- The recursion tree of $T(n) = 3T(\frac{n}{2}) + n$

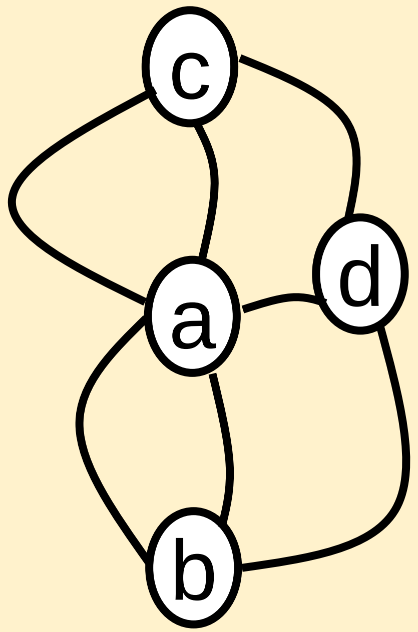


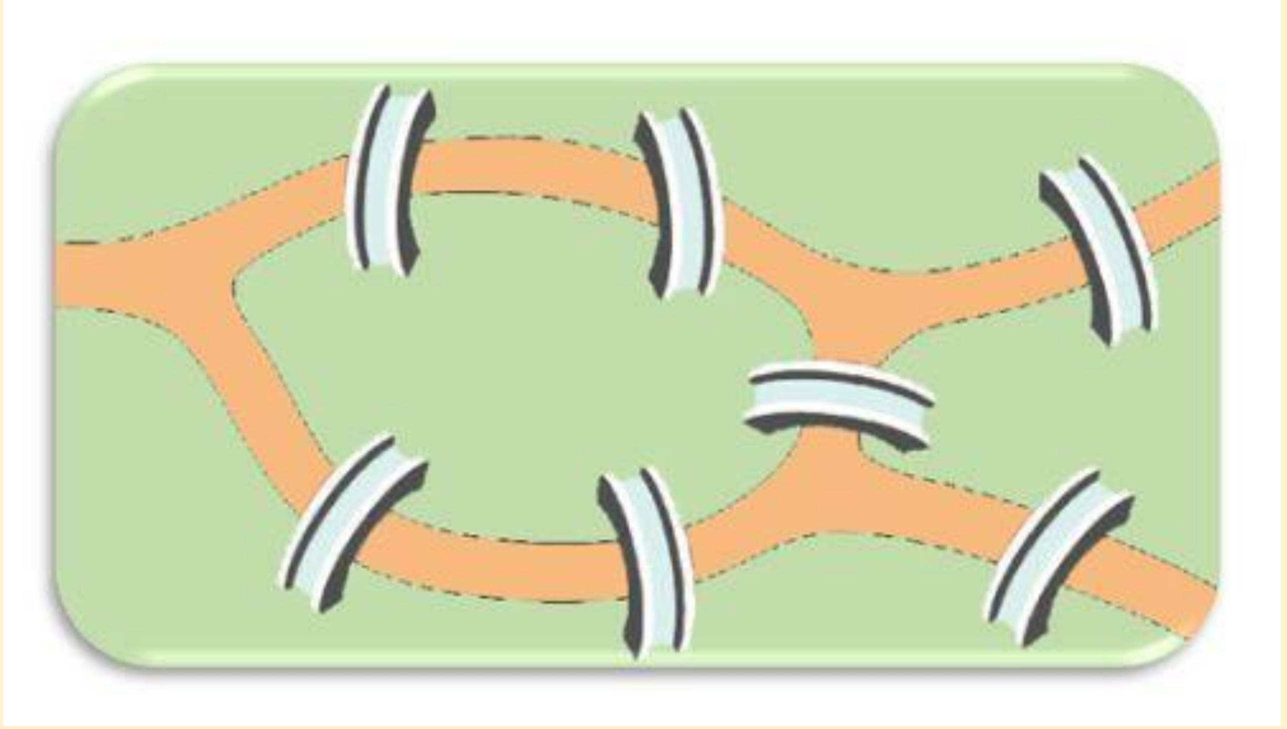
- **Additional data structure**
  - **Graph basics**
  - **Adjacency matrix**
  - **Adjacency list**

Graph basics

- ➢ Graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects.
- ➢ The subject that expresses and understands the real world systems as a network is dependent on graph theory.
- ➢ In computer science, a graph is an abstract data type to implement concepts in graph theory within mathematics.
- ➢ Graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc

- Graphs are used in
  - Modelling of networks, e.g. Traveling salesman problem (TSP)
  - Class hierarchy in an objected oriented program can be modelled as a graph
  - Data flow analysis in the optimization phase of a compiler uses a graph model
  - resource allocation in operating systems can be modelled as a graph.
  - Others: analysis of electric circuits, project planning, social networks
- The development of algorithms to handle graphs is therefore of major interest .



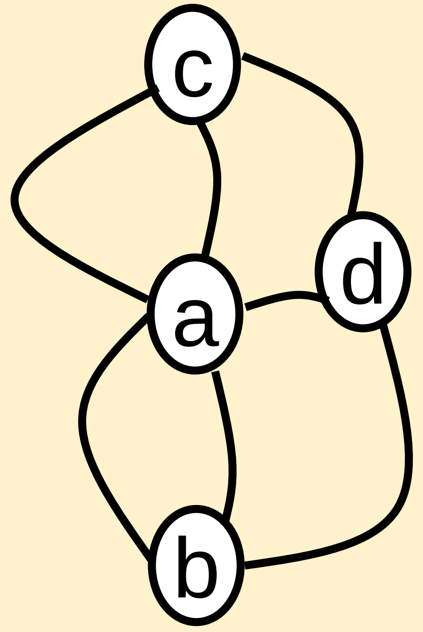


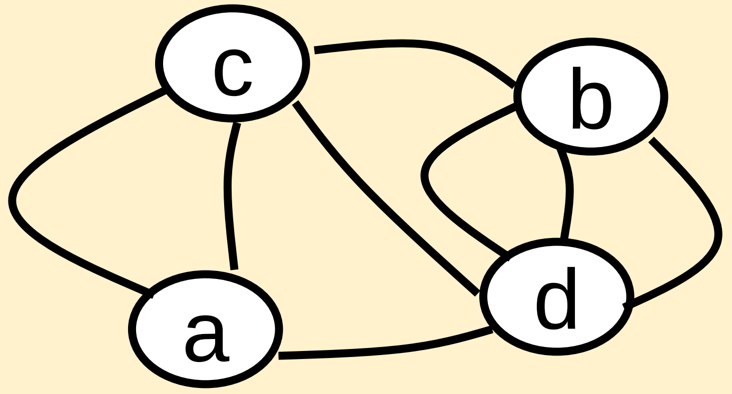
"Konigsberg" bridge example:

- Is it possible to walk across all the bridges exactly once and return to the starting point? or is it a Eulerian walk?
- Define the land area as vertices, the bridges as edges.

- The degree of a vertex is the number of edges incident to it.
- Eulerian walk:

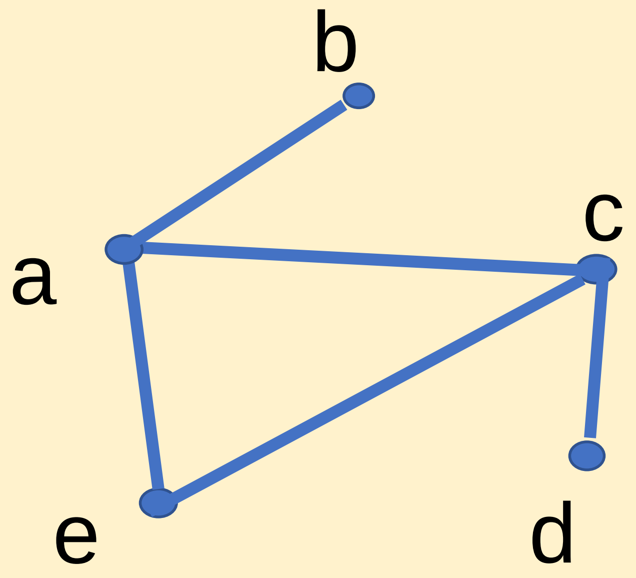
  Euler has shown that it is possible to walk across all the bridges exactly once and return to the starting point, if and only if, the degree of each vertex is even.
- Hence the walk is not feasible.
- Does this graph have an Eulerian walk?





Definitions

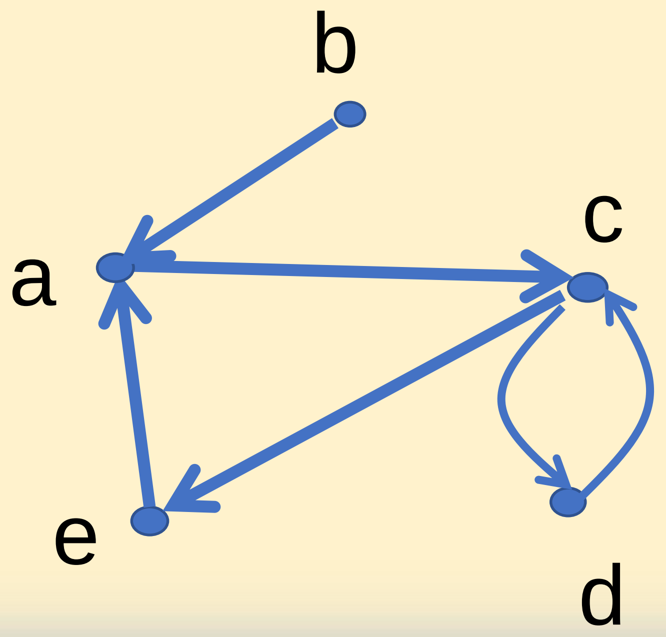
- Graph: a graph *G(V, E)* consists of a finite set *V* of vertices and a set of *E* of edges where each edge is a pair of vertices *(i, j)*



$G= (V, E), V= (a, b, c, d, e); E=\{(a, b), (a, c), (a, e),(e, c), (c, d) \}$

- Undirected graph: a graph is undirected if *(i, j) = (j, i)*, i.e., pair of vertices are unordered for all edges.

- Otherwise it is directed.
- We also use (i, j) to denote undirected, $< i, j >$ to denote directed edge (from i to j)

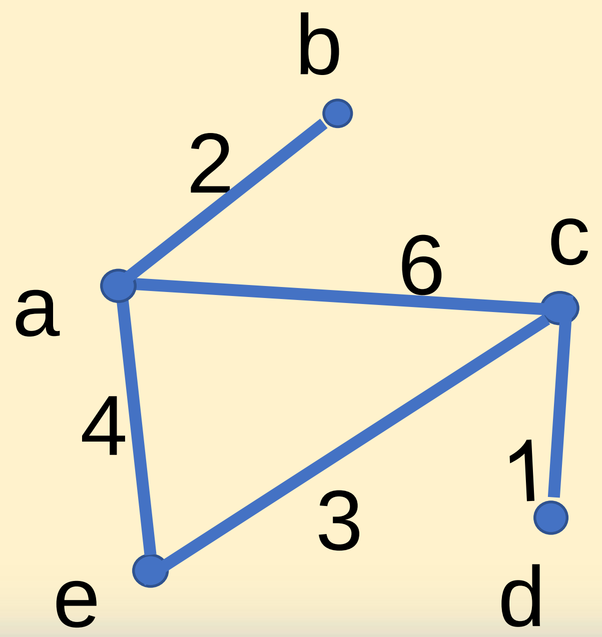


$G= (V,E)$, $V= ?$, $E=?$

$V= \{a,b,c,d,e\}$,

$E=\{ <b,a>, <a,c>,<e,a>,<d,c>,<c,d>,<c,e> \}$

- Weighted graph (network): a weighted graph G(V; E) has a cost, a real number, attached to an edge.



G= (V,E), V= ?, E=?

V= {a,b,c,d,e},

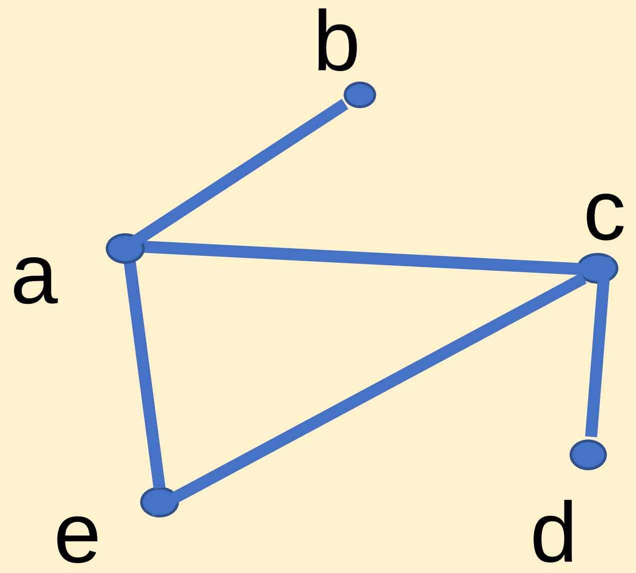
E={((a, b), 4), ((a, c), 6), ((a, e), 4),((e, c),3), ((c, d),1) }

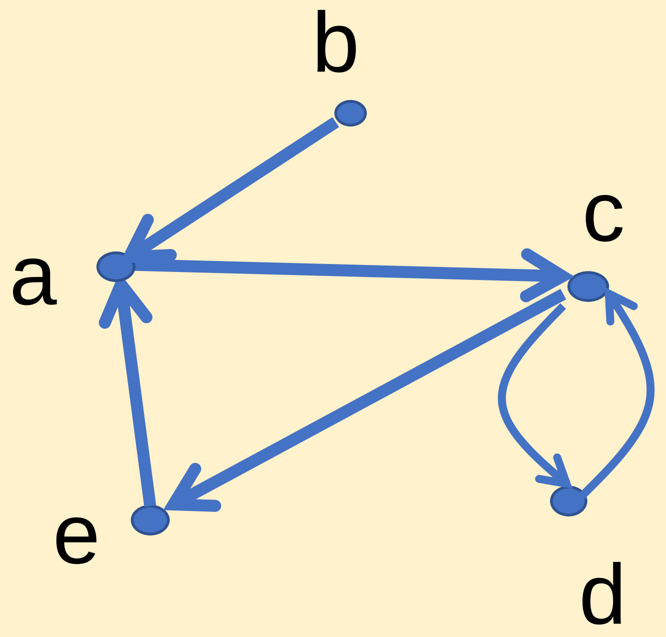
- Degree: The degree of a vertex is the number of edges attached to it.
- In-degree of vertex j is the number of edges ending at j.
- Out-degree of vertex j is the number of edges leaving j.
- In an undirected graph vertex i and j are adjacent if it is possible to get from vertex i to j by tracing along an edge.
- In a complete (or full) graph every pair of vertices are adjacent.
- A path from vertex v_P to vertex v_q is an unbroken sequence along edges

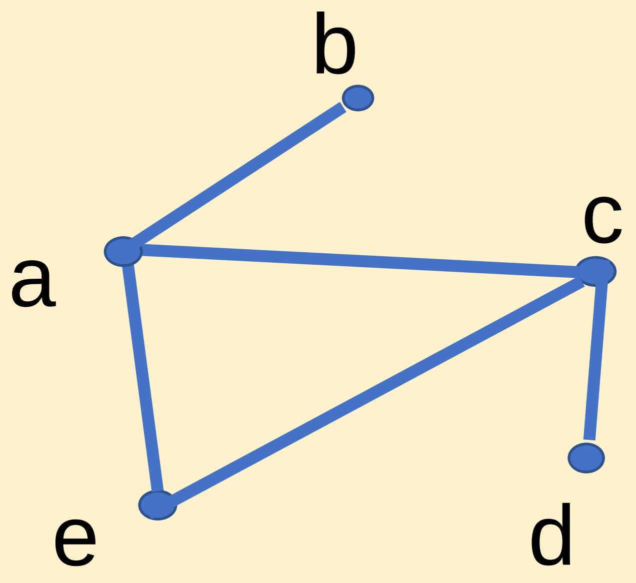
  $(v_p, v_1), (v_{1/v_2}), \cdots, (v_k, v_q)$ from edges in $E(G)$

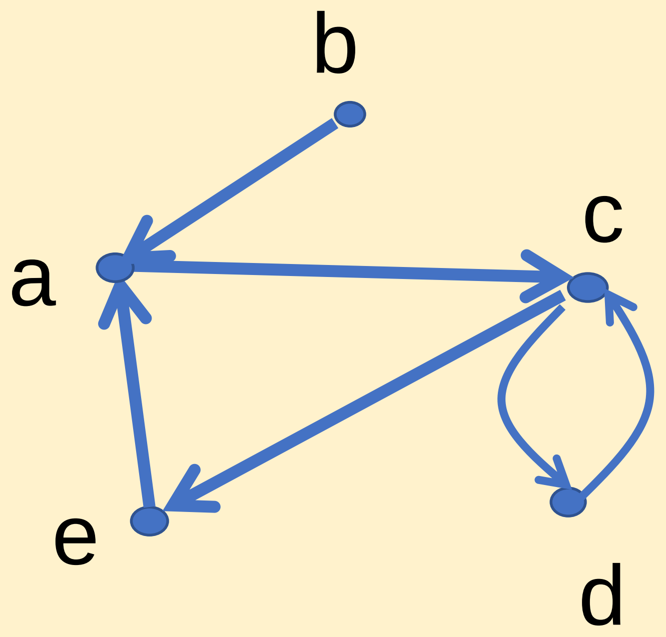
- A simple path occurs when all the vertices are distinct
- A cycle is a simple path in which the first and last vertex are identical.

Exercise: Find paths and cycles in the following graphs.



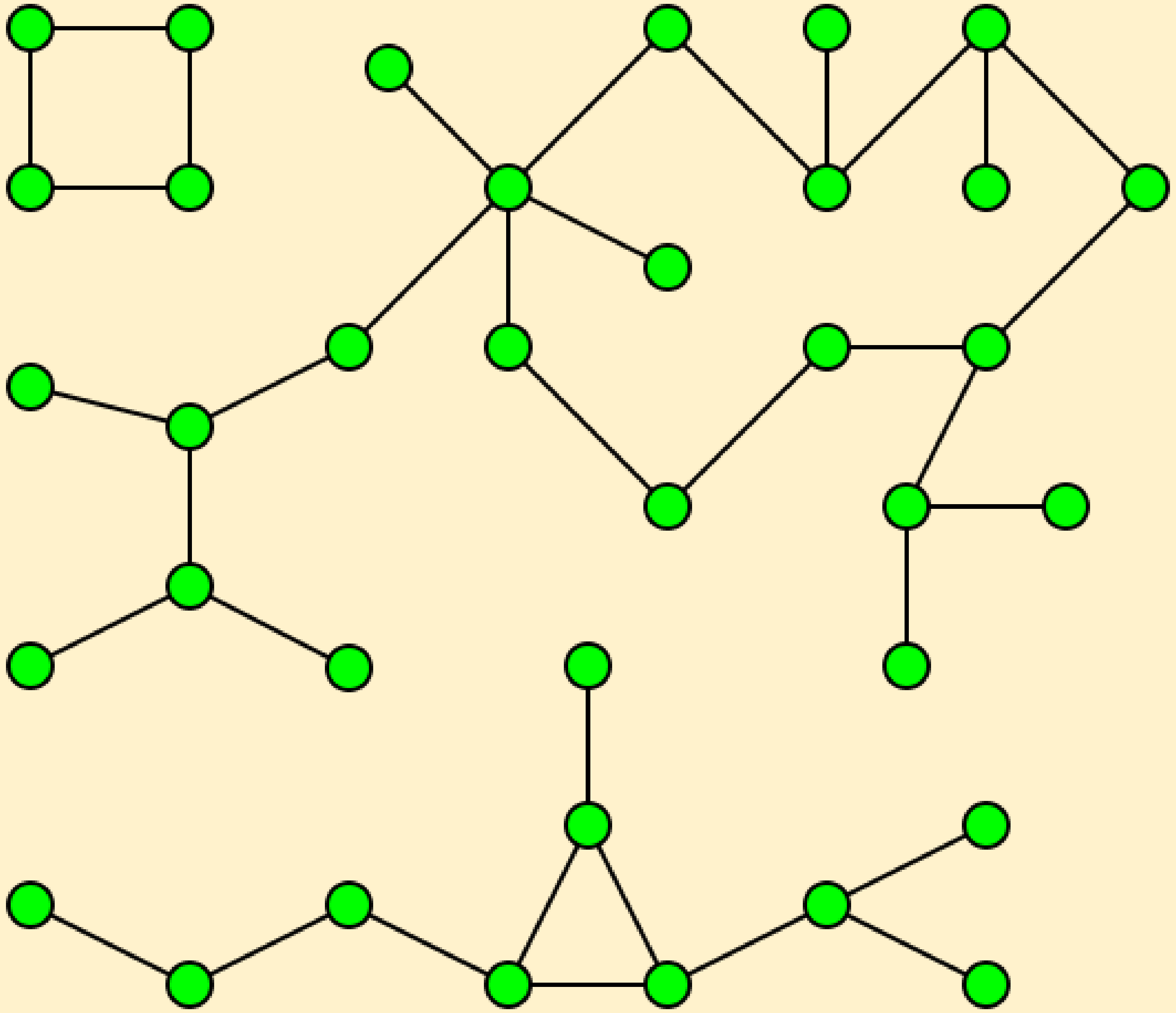



b
c
a
e
d


b
c
a
e
d

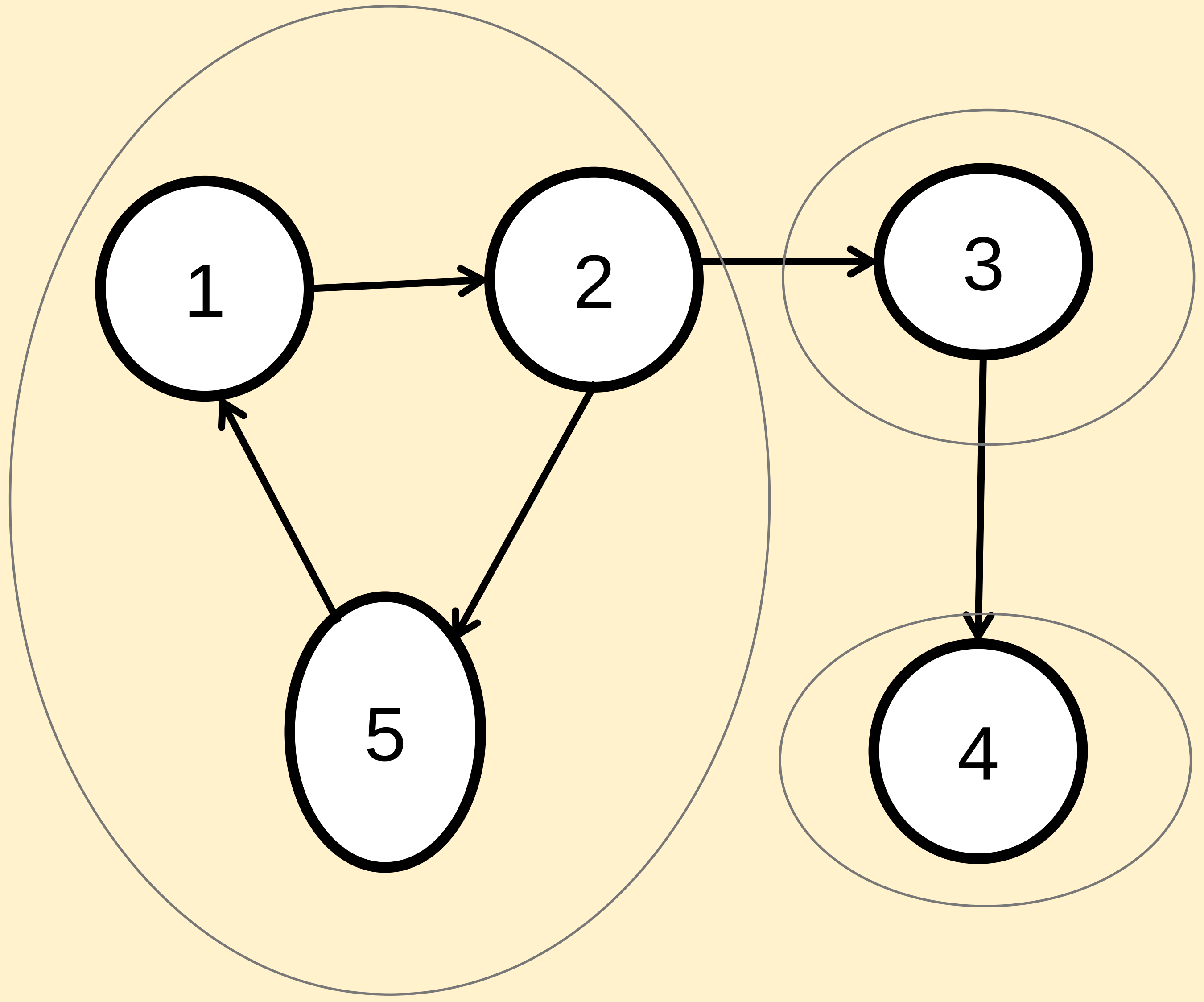
- An undirected graph is <u>connected</u> if for every pair of vertices there exists a path between them.

- A connected component 'C' is a maximal connected subgraph.

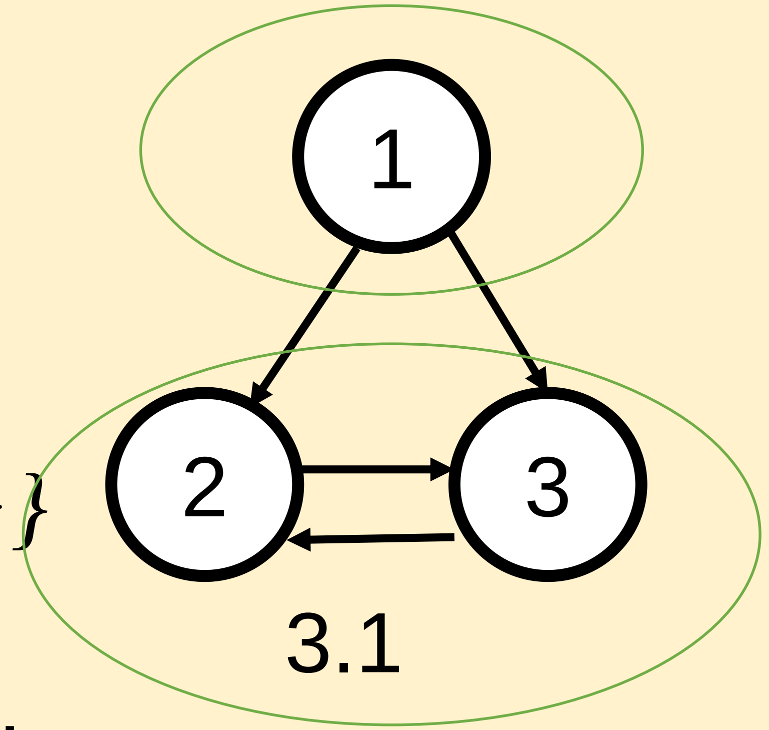


- A directed graph is strongly connected if for a path between v_P and v_q there also exists a path from v_q to v_P.

- A strongly connected component 'S' is a maximal subgraph that is strongly connected



Example: Directed Graph



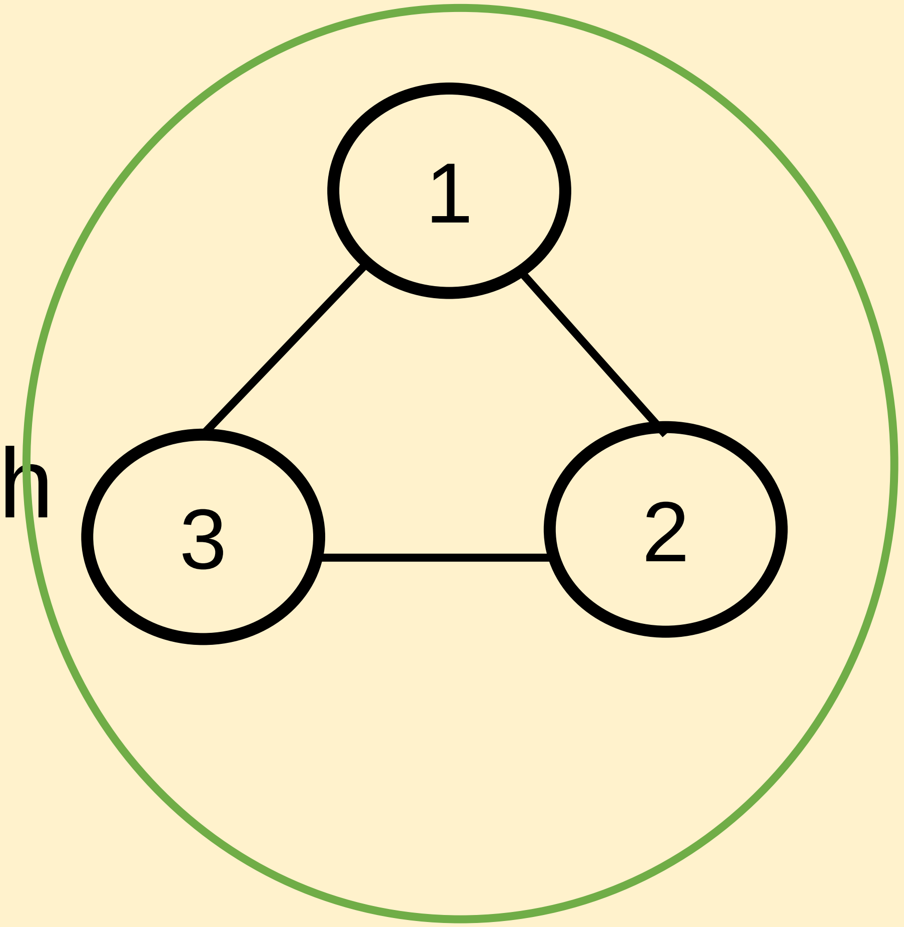
$V=\{1,2,3\}$

$E=\{<1,2>, <2,3>, <1;3>, <3;2>\}$

- The value 3.1 is a cost.
- { <1,2>, <2,3> } is a simple path
- Graph is not connected because cannot get from vertex 2 to 1.
- Strongly connected component ?

Example: Undirected Graph:

$V=\{1,2,3\}$
$E=\{(1,2), (2,3), (3,1)\}$



- (1,2), (2,3) is a simple path
- (1,2) (2,3) (3,1) a cycle.
- This Graph is connected
- Connected component ?

Adjacency matrix

- An adjacency matrix is a square matrix used to represent a finite graph.
- The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.
- 2-D matrix $A[n, n]$, where n is the number of vertices.
- Undirected graph

$$A[i, j] = \begin{cases} 1 & \text{if edge } (i, j) \text{ is in } E(G) \\ 0 & \text{otherwise} \end{cases}$$

- directed graph

$$A[i, j] = \begin{cases} 1 & \text{if edge } \langle i, j \rangle \text{ is in } E(G) \\ 0 & otherwise \end{cases}$$

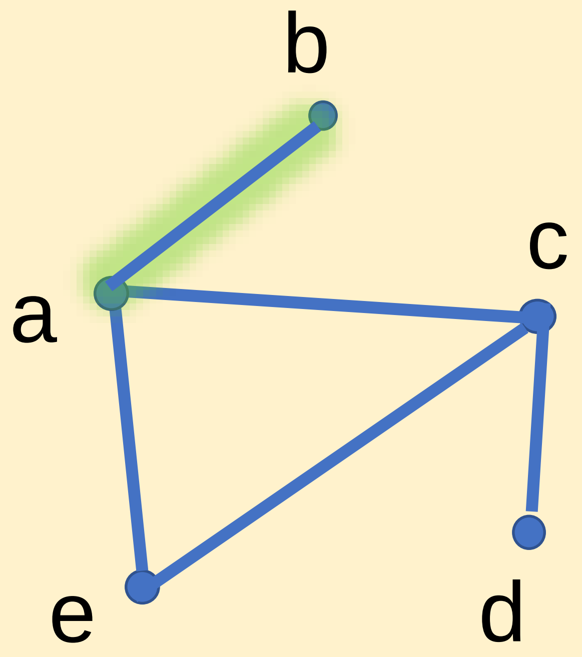
- Weighted graph

$$A[i, j] = \begin{cases} c & \text{if edge } <i, j> \text{ is in } E(G) \\ \infty & \text{otherwise} \end{cases}$$

c is the edge (i,j) cost

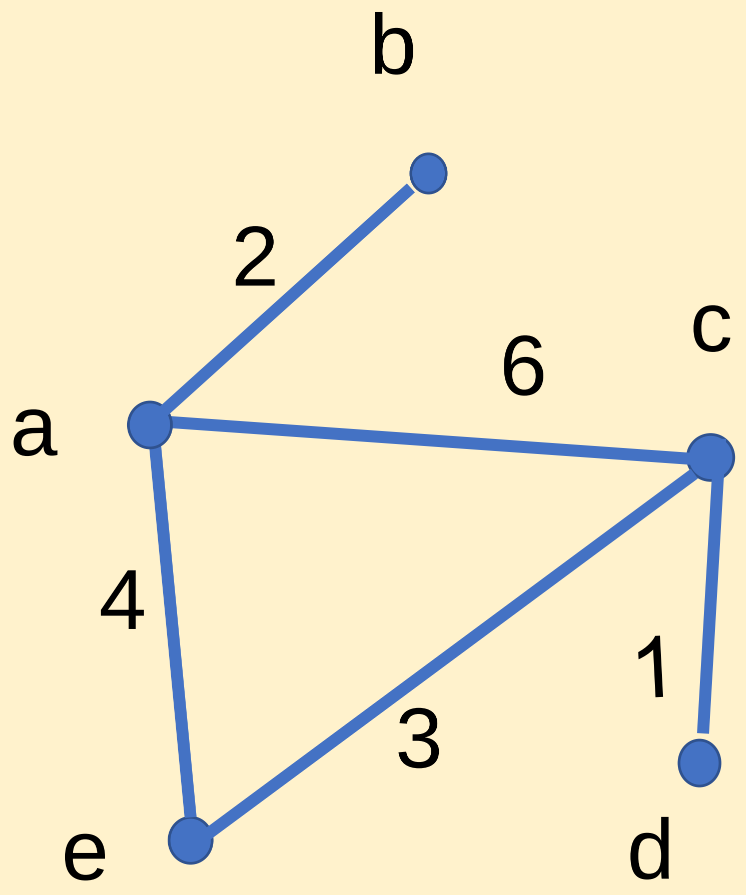
- For undirected graph, the degree of any node is its row sum
- For directed graph, the in-degree is the column sum, the out-degree is the row sum
- O(1) to check for Adjacency
- O(n2) space complexity

Example



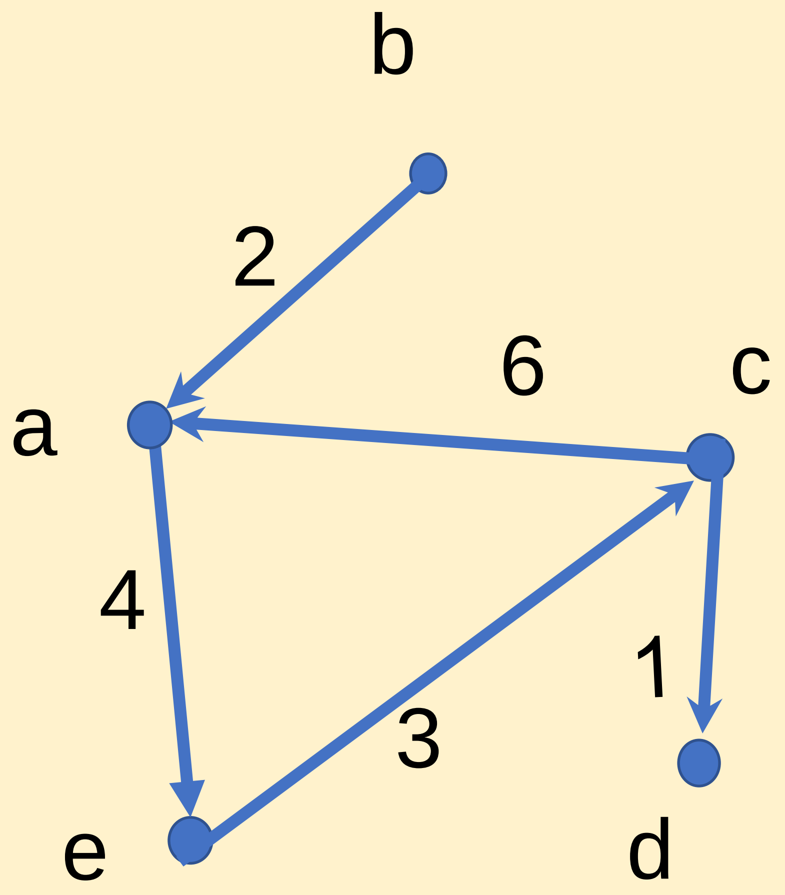
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Example



$$\begin{bmatrix} 0 & 2 & 6 & \infty & 4 \\ 2 & 0 & \infty & \infty & \infty \\ 6 & \infty & 0 & 1 & 3 \\ \infty & \infty & 1 & 0 & \infty \\ 4 & \infty & 3 & \infty & 0 \end{bmatrix}$$

Example

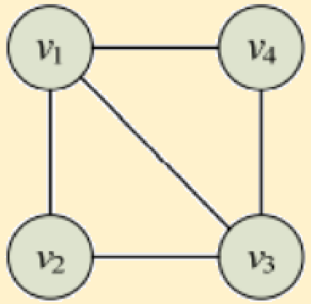


$$\begin{bmatrix} 0 & \infty & \infty & \infty & 4 \\ 2 & 0 & \infty & \infty & \infty \\ 6 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

Adjacency matrix

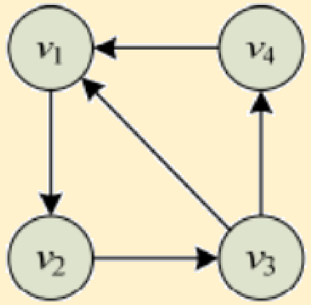
$$a_{ij} = \begin{cases} 1 & 若 e(v_i, v_j) \in E 或者 e < v_i, v_j > \in E \\ 0 & 若 e(v_i, v_j) \notin E 或者 e < v_i, v_j > \notin E \end{cases}$$

$$a_{ij} = \begin{cases} w_{ij} & 若 e(v_i, v_j) \in E 或者 e < v_i, v_j > \in E \\ \infty & 若 e(v_i, v_j) \notin E 或者 e < v_i, v_j > \notin E \\ 0 & 若 v_i = v_j \end{cases}$$



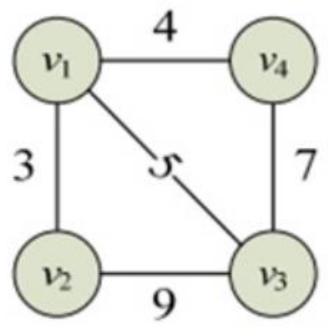
$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

a. 无向图的邻接矩阵



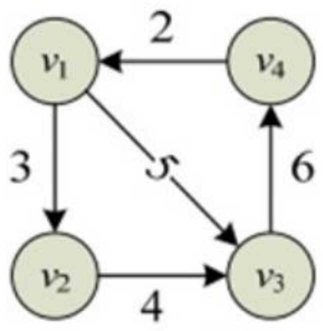
$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

b. 有向图的邻接矩阵



$$\mathbf{A} = \begin{bmatrix} 0 & 3 & 5 & 4 \\ 3 & 0 & 9 & \infty \\ 5 & 9 & 0 & 7 \\ 4 & \infty & 7 & 0 \end{bmatrix}$$

a. 带权无向图的邻接矩阵

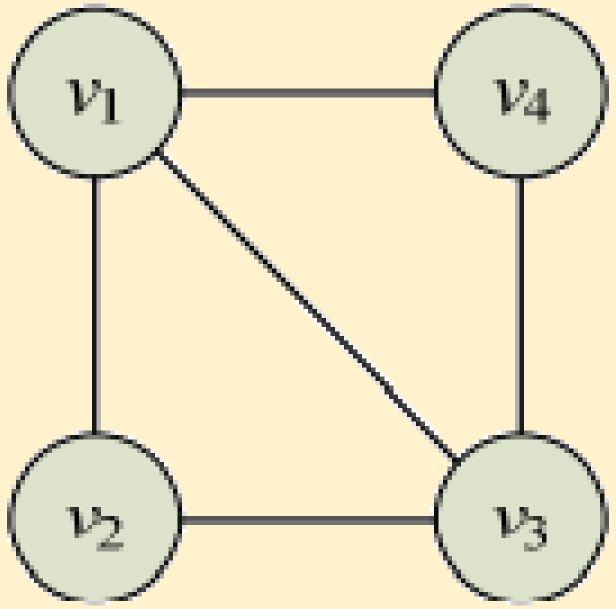


$$\mathbf{A} = \begin{bmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & 4 & \infty \\ \infty & \infty & 0 & 6 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

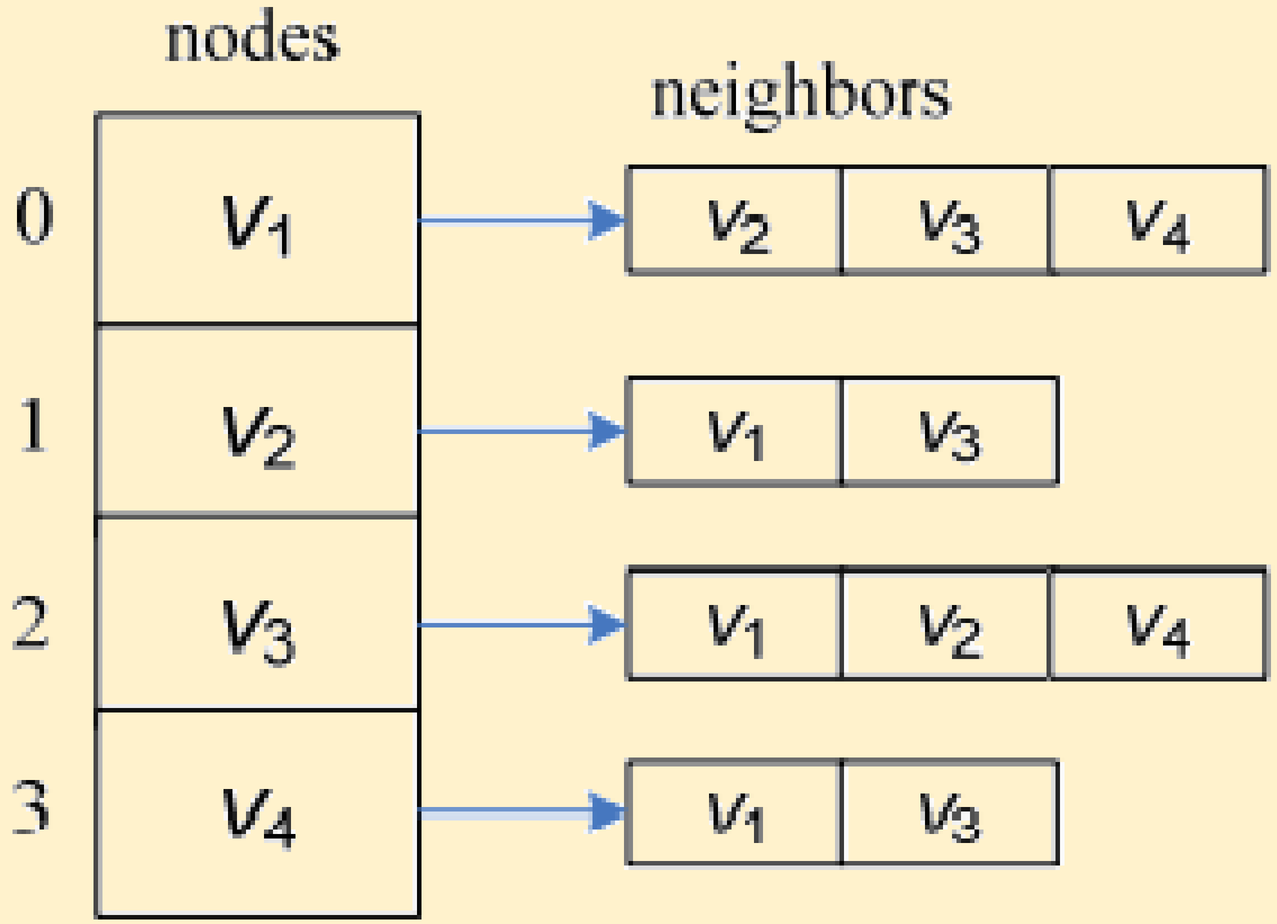
b. 带权有向图的邻接矩阵

Adjacency list

- associates each vertex in the graph with the collection of its neighbouring vertices or edges. Each list describes the set of neighbours of a vertex in the graph.
- A n node graph represented as n linked lists
- Declare an Array of size n e.g. $A[1, \ldots, n]$ of Lists.
- $A[i]$ is a pointer to the edges in $E(G)$ starting at vertex i
- For weighted graph, data in each list cell is a pair(vertex, weight)
- the degree of any node is the number of elements in the list.
- O(e) to check for adjacency for e edges.
- O($n+e$) space for graph with n vertices and e edges

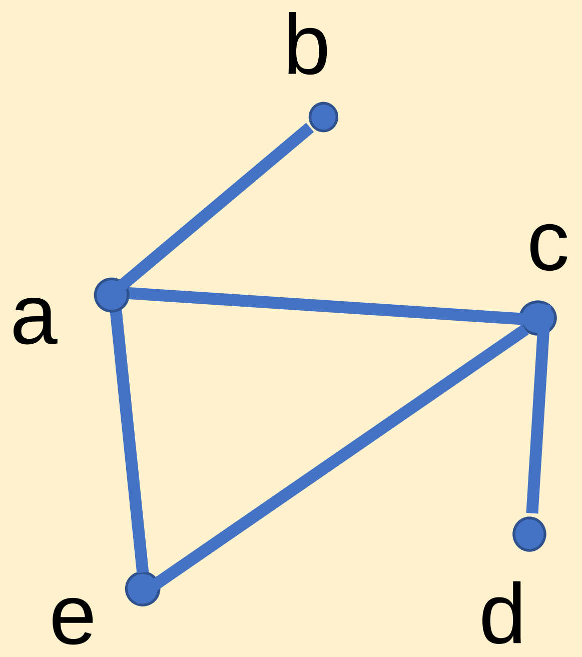


v_1
v_4
v_2
v_3



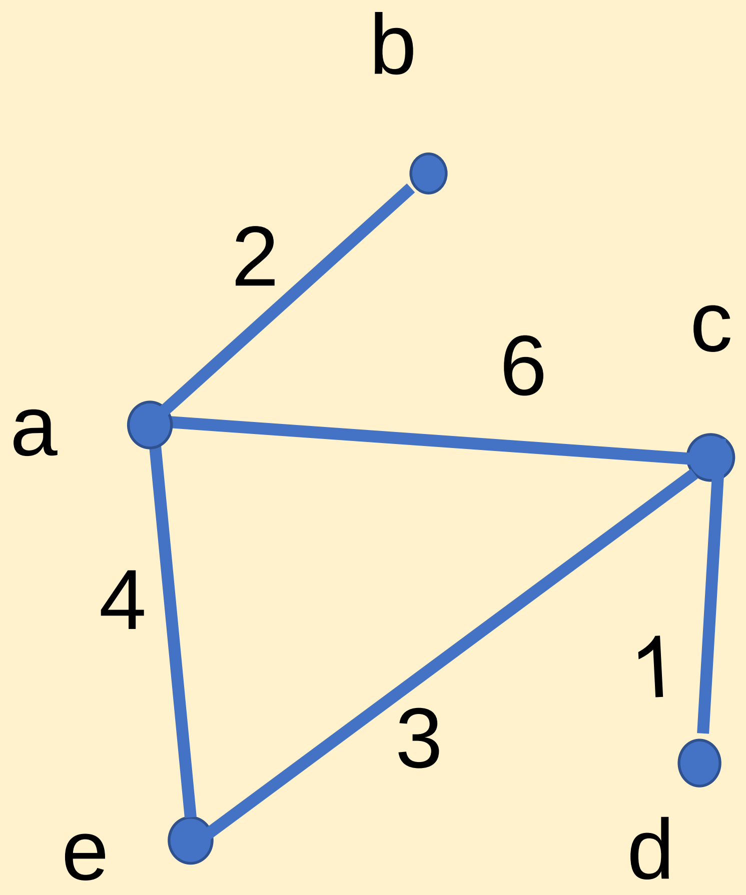
nodes
neighbors
0
1
2
3
v_1
v_2
v_3
v_4
v_2 v_3 v_4
v_1 v_3
v_1 v_2 v_4
v_1 v_3

Example



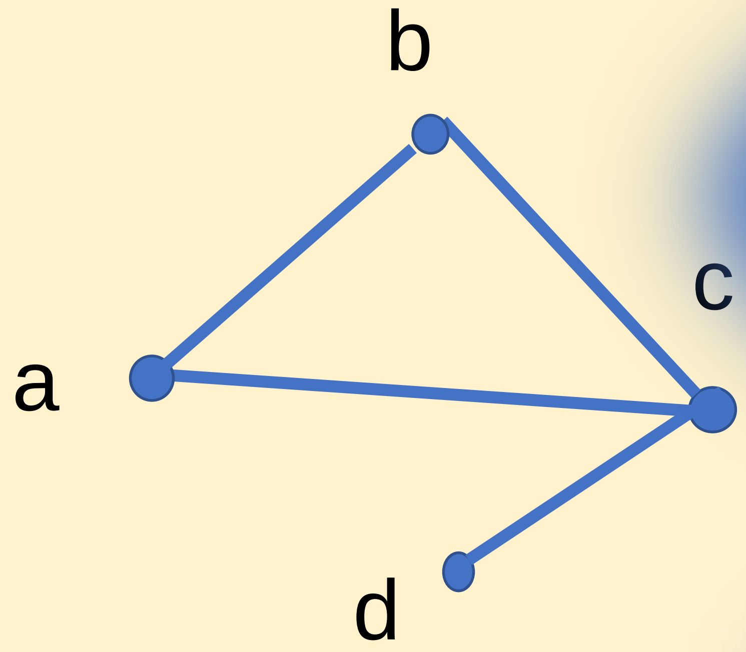
a	*b*	*c*	*e*	
b	*a*			
c	*a*	*e*	*d*	
d	*c*			
e	*a*	*c*		

Example



a	*[b,2]*	*[c,6]*	*[e, 4]*	
b	*[a,2]*			
c	*[a,6]*	*[e,3]*	*[d,1]*	
d	*[c,1]*			
e	*[a,4]*	*[c,3]*		

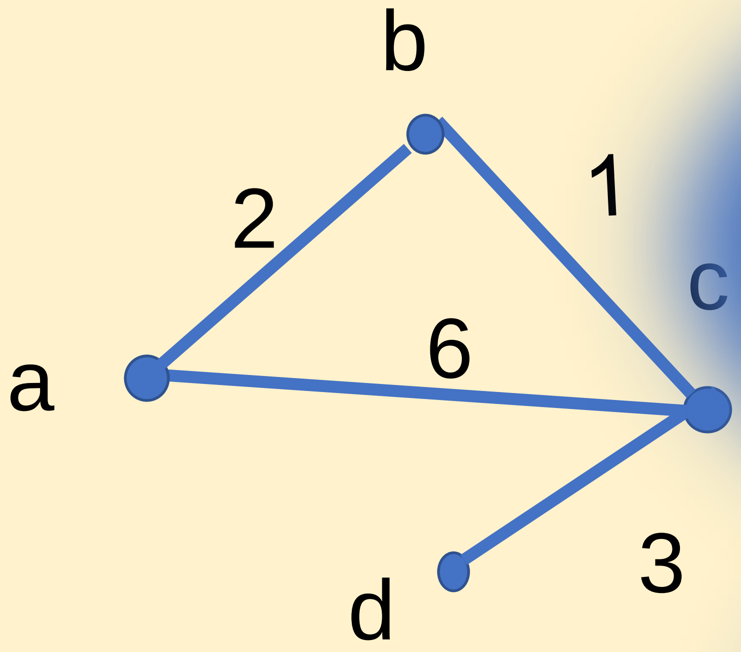
In class exercise: Find matrix and list representations for graph



[redacted]

a	[redacted]	[redacted]	[redacted]	
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
d	[redacted]	[redacted]	[redacted]	

In class exercise: Find matrix and list representations for graph



[redacted]