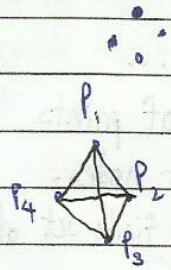
NO.

DATE

{GRAPHS}

Illustration



The graph has 4 vertices
↳ P_1, P_2, P_3, P_4

has 6 edges
↳ $P_1P_2, P_2P_3, P_3P_4, P_4P_1, P_2P_4, P_1P_3$ } each of these can be interchanged

Definition

A graph ...

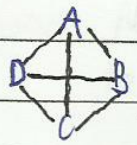
- is a finite set of points called vertices or nodes,
- together w/ a finite set of edges,
- each of which joins a pair of nodes

{COMPLETE GRAPH}

→ A graph is considered complete if every pair of vertices is joined by an edge.

→ The complete graph with n vertices is denoted by K_n

- Example

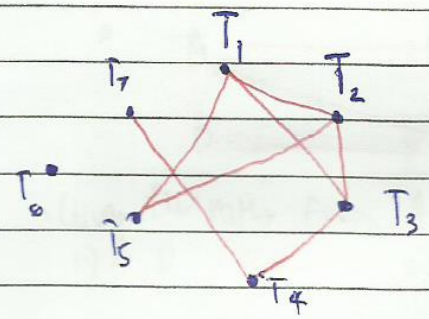


# of vertices? 4:
A, B, C, D

# of edges? 6:
AB, BC, CD, AD, BD, AC

- Exercise 1 -

• Draw a graph based on the ff. details: T_1 has played T_2, T_3 & T_5, T_2 has played T_1, T_3 & T_5 T_3 has played T_1, T_2 & T_4, T_4 has played T_3 T_7, T_5 has played T_1, T_2, T_6 withdrew, T_7 played T_4



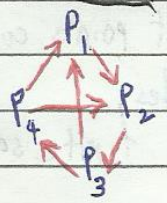
// can't determine which node won the games or has the most edges in this graph

NO.

DATE

{Directed graph}
or
Digraph

Illustration



- This digraph has 4 vertices → P_1, P_2, P_3, P_4
- has 6 arcs → (P_1, P_2), (P_2, P_3), (P_2, P_3), (P_3, P_4), (P_4, P_1), (P_4, P_2)

Definition

A directed graph...
- is a finite set of points called vertices or nodes.
- together with a finite set of directed edges or arcs.
- each of which joins an ordered pair of distinct vertices.
  - ↳ unique

{Arc in a digraph}

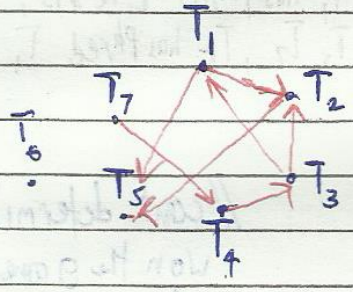
- A directed edge or ARC is an ordered pair of vertices (v, w).

where: v is the tail of the arc and
w is the head of the arc

Illustration: 

Notation: $v \to w$ means that w is adjacent to v

Exercise 2
// the same graph but w/ direction



Sterling

NO.

DATE

{ Path of a digraph }

Path in a digraph → is a sequence of vertices $V_1, V_2, V_3, \ldots V_n$, such that $V_1 \to V_2$, $V_2 \to V_3$, $\ldots$ $V_{n-1} \to V_n$ are arcs in the digraph

- Examples: Based on Previous Exc. There are two paths from T_7 to T_2
  1.) $T_7 \to T_4$, $T_4 \to T_3$, $T_3 \to T_2$
  2.) $T_7 \to T_4$, $T_4 \to T_3$, $T_3 \to T_1$, $T_1 \to T_2$

{ Length of Path }

- The length of path is the number of arcs on the path.

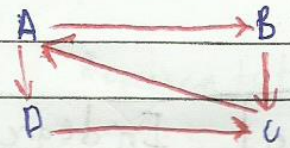
- Examples:
  - The longest path from T_7 to T_2 is length is 4
  - The shortest path from T_7 to T_2 is length is 3

Special Case: path of length zero is the # of arcs in a path from a vertex to itself.

{ Simple Path }

A path is simple if all the vertices on the path, except possibly the first and the last are distinct

- Examples:
  - The path from T_7 to T_2 is simple.
    The vertices T_7, T_4, T_3, T_2 are unique.



- Given the paths from A to D
  1.) A, D ✓ simple 2.) A, B, C, (A), D ✗ not simple
  ↳ not distinct since it is where we started

Sterling

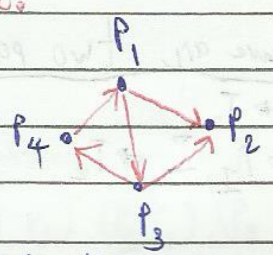
NO.

DATE

{ Simple Cycle }

- A simple cycle is a simple path that begins and ends at the same vertex.

- Example:



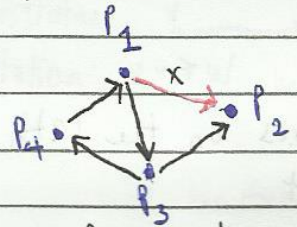
- The path from P_1 to P_1 (with vertices P_1, P_3, P_4, P_1) is simple cycle

> Cyclic and Acyclic

- if a graph contains a cycle then its cyclic; otherwise it is acyclic

{ Incident and Degree }

- Incidence
  - A node n (or vertex n) is incident to an arc x if n is one of the two nodes in the ordered pair of nodes that comprise x.



P_1 is incident to arc x
P_2 is incident to arc x

- Degree - of a node n is the number of arcs incident to it
  In degree - of a node n is the number of arcs that have node n as a head
  Out degree - of a node n is the number of arcs that have node n as a tail.

Node	Degree	Out degree	In degree
P_1	3	2	1
P_2	2	0	2
P_3	3	2	1
P_4	2	1	1

NO.

DATE

{Relation}

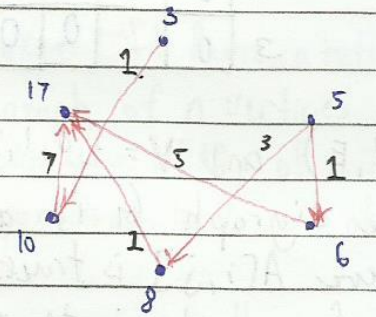
- A graph can be used to illustrate a relation.

- Example of a Relation -

Desc: $x \in A$ is related to $y \in A$ if $(x < y)$ and the remainder obtained by dividing y by x is odd.

IF $A = \{3, 5, 6, 8, 10, 17\}$

THEN Elements of the Relation: $R = \{(3,10), (5,6), (5,8), (6,17), (8,17), (10,17)\}$ is a relation on A.



* the weight of each arc is the remainder after dividing y by x.

{Weighted graph or Network}

- The Illustration above shows a weighted graph.
- A weighted graph is a graph in which a number, called weight is associated with the arc.
- A labeled digraph is a digraph in which arcs and/or vertices can have associated labels which may be names, costs, or any values of any given type.

/* Representations of Graph */

- Adjacency Matrix
- Adjacency List Representation
  a.) Linked-list
  b.) Cursor Based (w h y)

Sterling

NO.

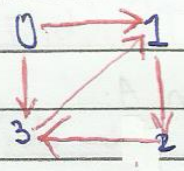
DATE

{Representations of Graph}

1.) Adjacency Matrix
- An n-by-n matrix (a.k.a array)
- first index starts at zero

// 2 ways of organizing memory: row major order, column major order

•Illustrations•



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

* Given a graph G = (V, E) and V = {0, 1, 2, 3, ...}*, the adjacency matrix of the digraph G is an n x n matrix of Booleans, where A[i, j] is true if and only if there is an arc from vertex i to vertex j

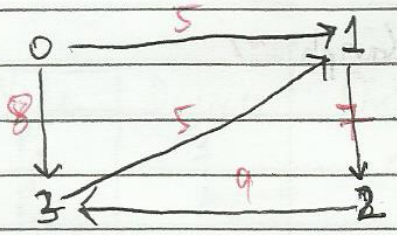
// Labeled Adjacency Matrix

Given a digraph G = (V, E) and V = {0, 1, 2, 3, ...}

A labeled adjacency matrix is an n x n matrix B of arc weights. B[i, j] is the weight of the arc from vertex i to vertex j

•Illustrations

> Graph



> Labeled Adjacency Matrix

	0	1	2	3
0	∞	5	∞	8
1	∞	∞	7	∞
2	∞	∞	∞	9
3	∞	5	∞	∞

// '∞' is a sentinel value so we could use a really large #

Sterling

NO.

DATE

*Disadvantage on Adjacency Matrix Representation

1.) For a digraph of n vertices, the storage requirement is $\Omega(n^2)$ even if digraph has arcs fewer than n^2.

2.) Examining a square matrix would require $O(n^2)$ time.

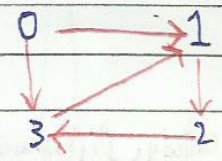
2.) Adjacency List Representation

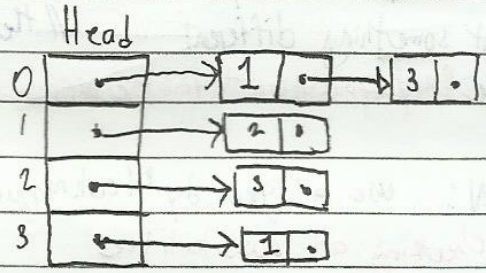
↳ The adjacency list for any vertex i in a list, in some order, of all vertices adjacent to i.

a.) Linked List Representation

*Example: A graph of n vertices is represented by an array, declared as HEAD, of n components. Storage requirement is proportional to the sum of the no. of vertices + no. of arcs

*Illustration:





b.) Cursor Representation

//same illustration

HEAD

0	0
1	3
2	5
3	7

ADJ

0	1
1	3
2	-1
3	2
4	-1
5	3
6	-1
7	1
8	-1
9	

Last: 8

// Note: In graph, there is no such thing as insert.

NO.

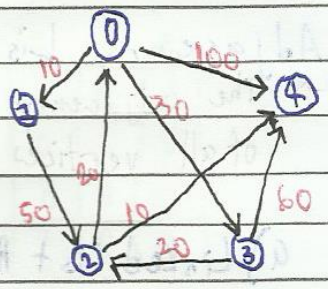
DATE

/* Dijkstra's, Floyd's and Warshall Algorithms */

{ The Single-Source Shortest Paths Problem }

- Given:
  Directed graph G=(V,E)
  in which each arc has a
  nonnegative label

Illustration:



*Problem
- Find the cost of the shortest path from a given SOURCE vertex, say Vertex 0, to the other vertices 1, 2, 3, 4
- Cost of the path may represent something different like time (any application to find the minimum value)

// The idea here is that you have
// a source vertex and u wanted to
// know the shortest path to the rest of
// the vertices

* SOLUTION: use a "greedy" technique, often called as Dijkstra Algo.
  ↳ checking all possibilities

// Note: Most codes on the web about Dijkstra's Algo are the same. Crazy programmer's code is more on storing the paths.

Sterling

NO.

DATE

/* Function Dijkstra */

Function Dijkstra

/* Computes the cost of the shortest path from vertex 0 to every vertex of a directed graph */

{

S = {0};

for (i = 1; i < n; i++)

D[i] = C[0, i]; // initializes D

for (x = 1; x < n; x++)

{

a) choose a vertex w in V-S such that

D[w] is a minimum; // except source vertex

b) add w to S;

c) for each vertex v in V-S do

D[v] = min(D[v], D[w] + C[w, v]);

}

}

2-D array, labeled Adj. matrix

1 dim. array, where the length of the shortest path to the other vertices

V minus S where V is the set containing all vertices

S = {0} is a set & an arbitrary source, we dont always start at 0

Stores all elements in row 0 (or whatever is our source)

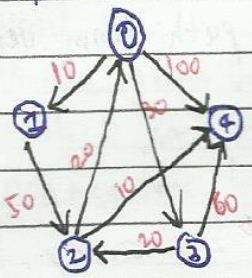
// Simulation next page

NO.

DATE

Simulation of Function Dijkstra

Graph G



Labeled Adj. Matrix C

	0	1	2	3	4
0	∞	10	∞	30	100
1	∞	∞	50	∞	∞
2	20	∞	∞	∞	10
3	∞	∞	20	∞	60
4	∞	∞	∞	∞	∞

// initialization

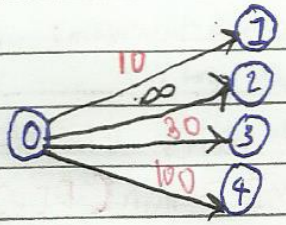
$V = \{0, 1, 2, 3, 4\}$

$S = \{0\}$

keep this at 0 as the distance between 0 & itself is 0.

D

0	
1	10
2	∞
3	30
4	100



// Simulation

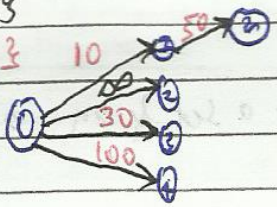
- $X = 1$

$V = \{0, 1, 2, 3, 4\}$

$S = \{0, 1\}$

D

0	
1	10
2	~~∞~~ 60
3	~~30~~ 30
4	~~100~~ 100



$X = 1$

$W = 1$

$v = 2$: $D[2] = \min(\infty, 10 + 50) = 60$

$v = 3$: $D[3] = \min(30, 10 + \infty) = 30$

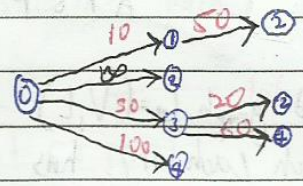
$v = 4$: $D[4] = \min(100, 10 + \infty) = 100$

NO.

DATE

• x = 2 V = {0, 1, 2, 3, 4}

S = {0, 1, 3 }



D

0	
1	10
2	60 50
3	30
4	100 90

x = 2

w = 3

v = 2: D[2] = min(60, 30 + 20) = 50

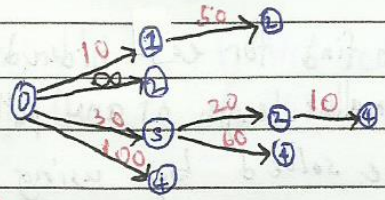
v = 4: D[4] = min(100, 30 + 60) = 90

• x = 3 V = {0, 1, 2, 3, 4}

S = {0, 1, 3, 2 }

D

0	
1	10
2	50
3	30
4	100 60



x = 3

w = 2

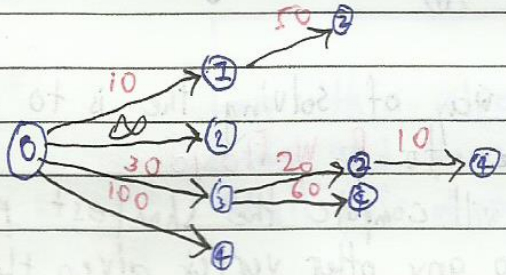
v = 4: D[4] = min(90, 50 + 10) = 60

• x = 4 V = {0, 1, 2, 3, 4}

S = {0, 1, 3, 2, 4}

D

0	
1	10
2	50
3	30
4	60



x = 4

w = 4

//end

NO. 2m

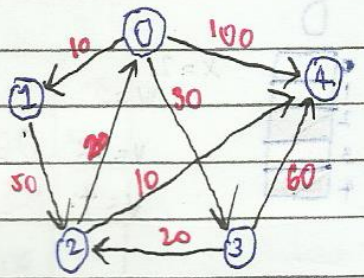
DATE

{ All-Pairs Shortest paths Problem }

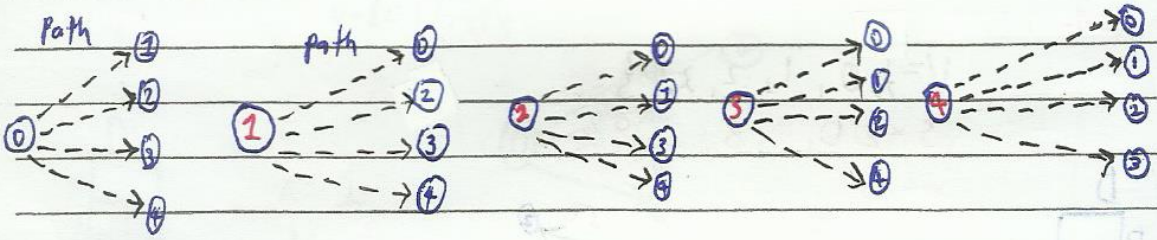
or

APSP

- Given:
  - Directed graph G=(V,E) in which each arc has a nonnegative label
- Problem:
  - Find the cost of the shortest path from any given SOURCE vertex to any DESTINATION vertex



* The APSP problem is to find for each ordered pair of vertices (v, w) the smallest length of any path from v to w.
* This problem could be solved by using Dijkstra's algorithm with each vertex in turn as source. V = {0, 1, 2, 3, 4}



* A more direct way of solving the is to use the following algorithm due to R. W. Floyd.
* This algorithm will compute the shortest path from any given vertex to any other vertex given the arc cost matrix C. The result is stored in an n x n matrix A.

Sterling

NO.

DATE

{ Floyd's Algorithm }

// used to solve all pairs shortest path problem

labeled Adj. Matrix

```
function Floyd (int A[][], int C[][])
```

is where the shortest path from any source vertex to any destination vertex will be stored

```
{
  int i, j, K;
  for(i=0; i<n; i++){
    for(j=0; j<n; j++){
      A[i][j] = C[i][j]
    }
  }
  for(i=0; i<n; i++){
      A[i][i] = 0;
  }
  for( K=0; k<n; K++){
    for(i=0; i<n; i++){
      for(j=0; j<n; j++){
        if( A[i][k] + A[k][j] < A[i][j]){
            A[i][j] = A[i][k] + A[k][j];
        }
      }
    }
  }
}
```

- will copy the labeled adj. Matrix to A (can also be done using memcpy)
- Entire matrix to matrix

Will set the diagonal to zero (like making elems from A[i][i] to A[n][n] to 0)

- loop within a loop within a loop
- finds and stores minimum cost of all pair of nodes.

Simulation →

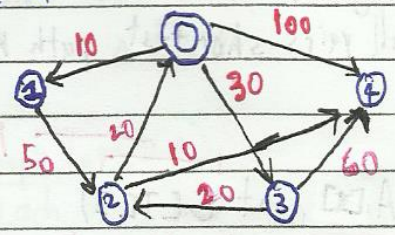
Sterling

NO.

DATE

Simulation: Floyd's Algorithm

Graph G



Adj. Matrix C

	0	1	2	3	4
0	∞	10	∞	30	100
1	∞	∞	50	∞	∞
2	20	∞	∞	∞	10
3	∞	∞	20	∞	60
4	∞	∞	∞	∞	∞

Copy →

APSP Matrix

	0	1	2	3	4
0	∞	10	∞	30	100
1	∞	∞	50	∞	∞
2	20	∞	∞	∞	10
3	∞	∞	20	∞	60
4	∞	∞	∞	∞	∞

↓ set diagonal to 0

	0	1	2	3	4
0	0	10	∞	30	100
1	∞	0	50	∞	∞
2	20	∞	0	∞	10
3	∞	∞	20	0	60
4	∞	∞	∞	∞	0

↓ Look for shortest Path

	0	1	2	3	4
0	0	10	50	30	60
1	70	0	50	100	60
2	20	30	0	50	10
3	40	50	20	0	30
4	∞	∞	∞	∞	0

Sterling

NO.

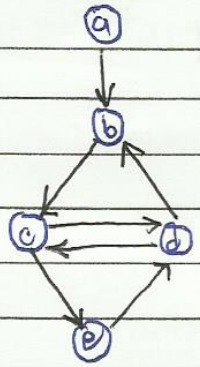
DATE

{ Transitive Closure }

* A transitive closure of the Adjacency matrix C is an nxn matrix A such that:
  - A[i,j] = 1 if there is a path from i to j
  - A[i,j] = 0 if there is no path from i to j

// Note: If there is an arc, there is a path

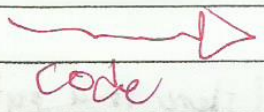
- Given:



	a	b	c	d	e
a	1	1	1	1	1
b	0	1	1	1	1
c	0	1	1	1	1
d	0	1	1	1	1
e	0	1	1	1	1

{ Warshall's Algorithm }

→ Determines the transitive closure of an adjacency matrix



Sterling

NO.

DATE

{ Warshal's Algo Code }

```
Function Warshall (int A[][], int C[][])
{
    int i, j, k;
    for(i=0; i<n; i++){
        for(j=0; j<n; j++){
            A[i][j] = C[i][j];
        }
    }
    for(k=0; k<n; k++){
        for(i=0; i<n; i++){
            for(j=0; j<n; j++){
                if(A[i][j] == 0){
                    A[i][j] = A[i][k] && A[k][j];
                }
            }
        }
    }
}
```

copies & stores contents of C onto A, similar to Floyd's, can also use memcpy

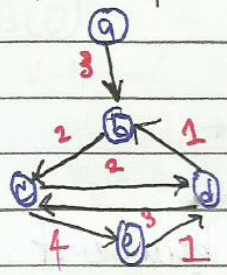
For checking if there is a path or no

NO.

//wtf is dis DATE

{ Finding the Center of the Graph }

- Weighted Graph:



Step 1: Determine the APSP Cost matrix using Floyd's Algorithm

	a	b	c	d	e
a	0	3	5	7	9
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	2	4	4	0

Step 2: Determine the eccentricity of each vertex

Ecc	∞	3	5	7	9

Note: The eccentricity of vertex v is MAX { min length of a path from w → v }

then find the min of the max

∴ B is the center of the graph!??

NO.

DATE

/* Traversals of Directed Graphs */

- 2 types of Graph Traversals:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

{ Depth First Search }

> - a systematic way of visiting vertices and arcs
> - This technique is called depth-first search because it continues searching in the forward (deeper) direction as long as possible
>   - Note: we backtrack if necessary.
> - a generalization of Preorder Traversals.

// Recursive dfs()

// initialization:

- Code:

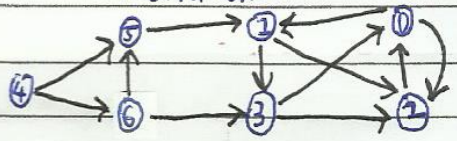
```
function dfs (vertex V){
    vertex w;
    mark[V] = visited;
    for each vertex w on L[V] do
        if (mark[w] == unvisited)
            dfs(w)
}
```

mark

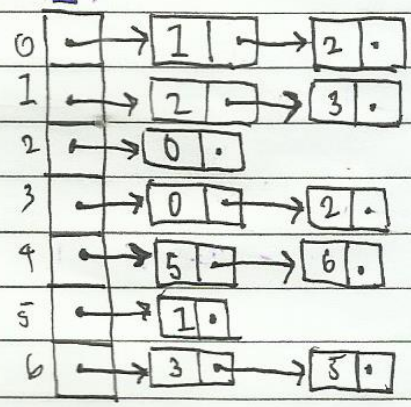
u	u	u	u	u	u	u
0	1	2	3	4	5	6

// u = unvisited
V = visited

- Illustration:



L



Sterling

NO.

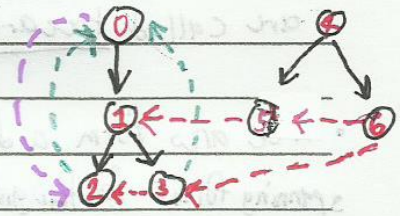
DATE

//simulation

- first call

mark: | V | V | V | V | V | V | V |
0 1 2 3 4 5 6

→ note: we can start ANYWHERE



dfs(0)

V = 0

$W_1 = 1$ → dfs(1)

$W_2 = 2$

V = 1

$W_1 = 2$ → dfs(2)

$W_2 = 3$

V = 2

$W_1 = 0$

End of dfs(2)

End of dfs(1)

dfs(3)

V = 3

$W_1 = 0$

$W_2 = 2$

End of dfs(3)

End of dfs(0)

// since Not all vertices
// are visited, make
// another call for dfs()

dfs(4)

V = 4

$W_1 = 5$ → dfs(5)

$W_2 = 6$

V = 5

$W_1 = 1$

End of dfs(5)

End of dfs(4)

dfs(6)

V = 6

$W_1 = 3$

$W_2 = 5$

End of dfs(6)

// end

NO.

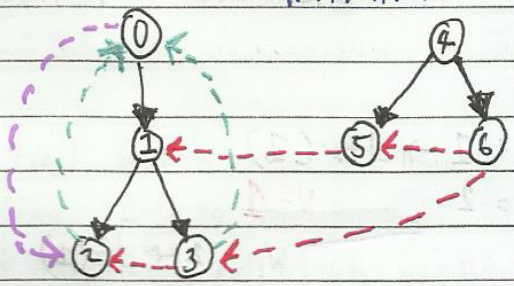
DATE

{Depth First Search Spanning Forest}

- The arcs leading to new unvisited vertices are called tree arcs.

- Tree arcs form a depth-first spanning forest for the given digraph.

- Illustration:



- Other types of arcs:

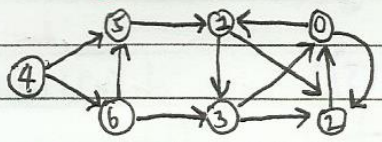
1.) **Back arc** - is a non-tree arc which goes from a vertex to one of its ancestors in the spanning forest.
↳ Example: 3 → 1

2.) **Forward arc** - is a non-tree arc which goes from a vertex to a proper descendant in the spanning forest.
↳ Example: 1 → 2

3.) **Cross arc** - is a non-tree arc which goes from a vertex to another vertex that is neither an ancestor nor a descendant
↳ Example: 3 → 2 and 5 → 1

{Exercise 2}

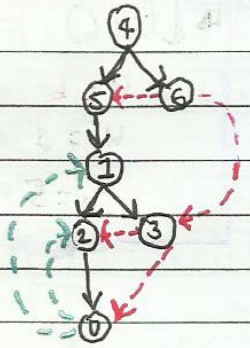
L vertices:
0: 1, 2
1: 2, 3
2: 0
3: 0, 2
4: 5, 6
5: 2
6: 3, 5



mark

u.v	u.v	u.v	u.v	u.v	u.v	u.v
0	1	2	3	4	5	6

Starting vertex: 4



dfs(4)
v=4
w1=5 → dfs(5)
w2=6
v=5
w1=1 → dfs(1)
dfs(5) end
v=1
w1=2 → dfs(2)
w2=3
dfs(1) end
v=2
w1=0 → dfs(0)
dfs(2) end
v=0
w1=1
w2=2
dfs(0) end
dfs(3)
v=3
w1=0
w2=2
dfs(3) end
end dfs(4)
dfs(6)
v=6
w1=3
w2=5
end dfs(6)
//done

Height of the tree: 4

Sterling

NO.

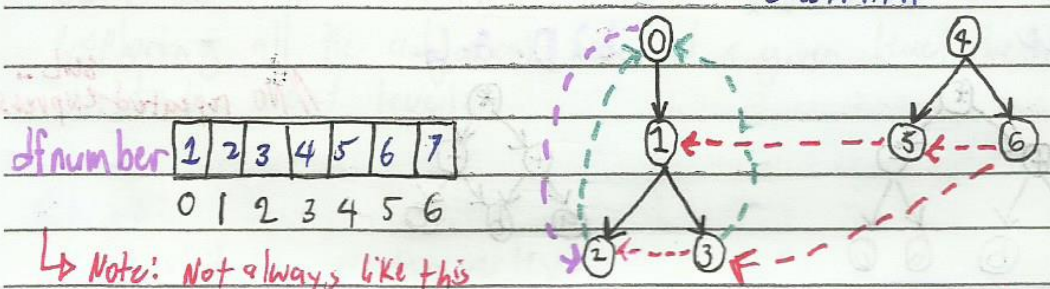
DATE

{ Depth-First Number }

A **depth-first number**:

- is a number assigned to a vertex
- according to the order
- in which the vertex was first marked
- visited during a depth-first search

• Illustration:



↳ Note: Not always like this since it's based on the tree/graph or the order of visitation

Observations:

1.) w is a descendant of v iff (if and only if):
dfnumber(v) < dfnumber(w) ≤ dfnumber(v) + number of descendants of v.

2.) Forward arcs go from low-numbered to high-numbered vertices

3.) Back arcs go from high numbered to low-numbered vertices

4.) Cross arcs go from high numbered to low-numbered vertices

NO.

DATE

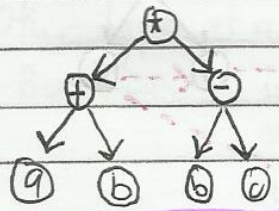
/* Directed Acyclic Graphs (DAG) */

> A directed acyclic graph is a directed graph w/ no cycles
> Are useful in representing the syntactic structure of arithmetic expressions with common subexpressions.

Representation of the Arithmetic Operation:

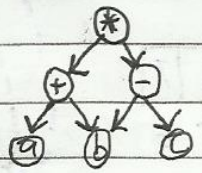
Given: $(a + b) * (b - c)$

a.) tree



b.) D. A. G.

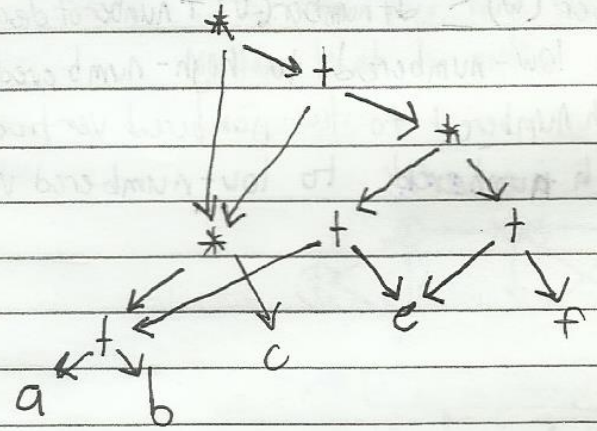
// NO repeated sub-expression



{ Exercise 4 }

Draw the DAG for the ff. expression:

$$\left((a+b)*c+((a+b)+e)*(e+f)\right)*((a+b)*c)$$



// unconfirmed though

NO.

DATE

/* Breadth First_Search */

NOTE: maam Peña barely/never discussed this. contents in this part is from Geeks for Geeks & other sources

> BFS/Breadth First Search is an algorithm used to traverse or search a graph.

> Can be implemented using queues.

> Basic idea of BFS is to traverse a graph/tree level by level, exploring all the adjacent nodes at a given level before moving on to the next level.

Steps:

1.) visit the starting vertex
2.) initialize queue to contain only the start vertex
3.) while the queue is not empty:

```
if (vertex w != visited)
    visit w & mark as visited
    Add w to queue
```

Pseudo Code by Geeks for Geeks:

```
// Here, G is the existing graph & X is the source node
BFS(Graph G, X)
  Let Q be the queue
  Q.enqueue(X) // inserting source node x into the queue
  Mark X node as visited
  while(Q != empty)
    Y = Q.dequeue() // Removing the front node from the queue
Process all the neighbors of Y. For all the neighbors Z of Y
If Z is not visited:
    Q.enqueue(Z) // stores Z in Q
    Mark Z as visited.
```

Sterling

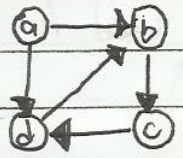
NO.

DATE

/* Undirected Graphs */

RECALL: Graph definitions

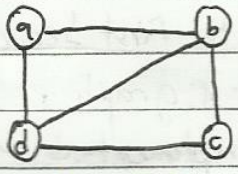
a.) Digraph D



A digraph $G = (V, E)$

- is a finite set of points called vertices, or nodes.
- together with a finite set of directed edges or arcs
- each of which joins an ordered pair of distinct vertices

b.) Undirected graph G



An undirected graph $G = (V, E)$

- is a finite set of points called vertices or nodes,
- together with a finite set of edges E.
- each of which joins a pair of vertices

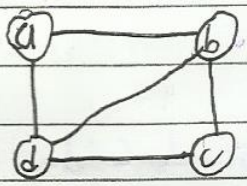
{ Basic Terminologies }

- Given & Illustration:

$G = (V, E)$

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

$E = \{(a,b), (a,d), (b,d), (b,c), (c,d)\}$



Adjacent

- In an undirected graph, vertices v and w are adjacent if (v, w) is an edge
- Example: a is adjacent to b, and b is adjacent to a

Incident

- In an undirected graph, the edge (v, w) is incident upon the vertices v and w.
- Example: (a, b) is incident upon vertex a and vertex b.

Sterling

NO.

DATE

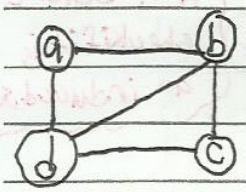
Path

- a sequence of vertices $V_1, V_2, \ldots V_n$ such that (V_i, V_{i+1}) is an edge.
- Example: a, b, d, c

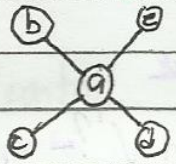
Length of a path

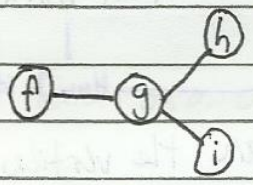
- the number of edges along the path
- Example: The length of path a, b, d, c is 3

{ Connected Graph }



a.) Connected Graph





b.) Unconnected Graph

all one graph btw

Connected Graph – // similar to complete Graph

- is a graph wherein there exist a path for every pair of its vertices
- The path $V_1, V_2, \ldots, V_n$ connects V_1 and V_n

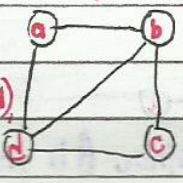
{ Sub Graph }, "prime" → reads G prime

$G = (V, E)$

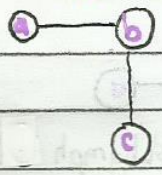
$V = \{a, b, c, d\}$

$E = \{(a,b), (a,d), (b,d), (b,c), (c,d)\}$

G



G'



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

$V' = \{a, b, c\}$

$E' = \{(a,b), (b,c)\}$

A **Subgraph** of $G = (V, E)$ is a graph $G' = (V', E')$ where:

1.) V' is a subset of V

2.) E' is a subset of E

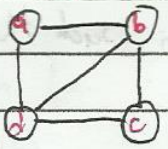
// In the ex. above, G' is a subgraph of G

NO.

DATE

{ Induced Subgraph }

G

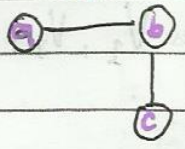


$G = (V, E)$

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

$E = \{(a,b), (a,d), (b,d), (b,c), (c,d)\}$

G'



$G' = (V, E)$

$V' = \{a, b, c\}$

$E' = \{(a,b), (b,c)\}$

G' is called an induced subgraph of G,

- if E' consists of all edges (v, w) in E,
- such that both vertices v and w are in V'

• Note: check if it is a subgraph first before check if it is an induced subg.

How 2 check

- What are the vertices in V'? → a, b, c
- Possible pair of vertices?

  a & b, a & c, b & c

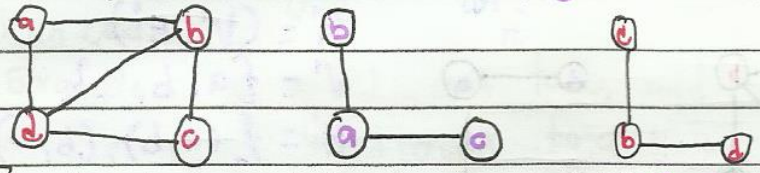
- table form:

Edges	E	E'
(a,b)	✓	✓
(a,c)	x	
(b,c)	✓	✓

∴ G' is an induced subgraph of G. E' consists of all edges (v, w) in E whose v & w are all in V'.

// Another Example!

$G = \{V, E\}$ $A = \{V_A, E_A\}$ $B = \{V_B, E_B\}$



→ ex. edge (a, c) is not found in G

- Is Graph A an induced subgraph of G? No since A is not a subgraph
- Is Graph B an induced subgraph of G? NO

Edges	E	E_B
(b,c)	✓	✓
(b,d)	✓	✓
(c,d)	✓	x

$V_B = \{b, c, d\}$

∴ B is not an induced subgraph of G. E_B does not consist of all edges (v, w) in E whose v and w are in V_B

Sterling

NO.

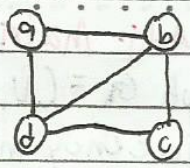
DATE

{Simple Cycle, Cyclic Graph & Free Tree}

$G = (V, E)$

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

$E = \{(a,b), (a,d), (b,d), (b,c), (c,d)\}$



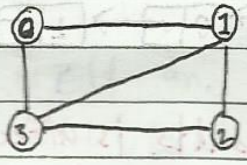
- **Simple Cycle**
  - A simple cycle in a graph is a simple path of length three or more that connects a vertex to itself
  - Example: a, b, d, a
- **Cyclic Graph**
  - A graph is cyclic if it contains at least one cycle
  - Example: Graph G above is cyclic
- **Free Tree**
  - A connected, acyclic graph is sometimes called a free tree.

/* Representations of Undirected Graphs */

1.) Adjacency matrix

Given a Graph $G = (V, E)$ and $V = \{0, 1, 2, 3, \ldots, n-1\}$. The adjacency matrix of the Graph G is an nxn matrix A of booleans (zeroes or ones), where A[i,j] is true (or 1) if and only if there is an edge from vertex i to vertex j.

- Illustration:



Adj. Matrix A

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

Sterling

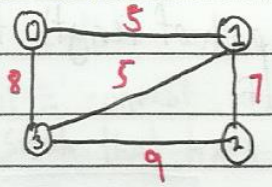
NO.

DATE

// Labeled Adj. Matrix

Given a Graph G = (V, E) and V = {0, 1, 2, 3, ..., n-1}
A labeled adjacency matrix is an nxn matrix B of arc weights, B[i, j] is the weight of the arc from vertex i to vertex j

* Illustration:



* Labeled Adj. Matrix B

	0	1	2	3
0	∞	5	∞	8
1	5	∞	7	5
2	∞	7	∞	9
3	8	5	9	∞

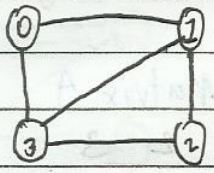
2.) Adjacency List Representation

* The adjacency list for any vertex i is a list, in some order, of all vertices adjacent to i.

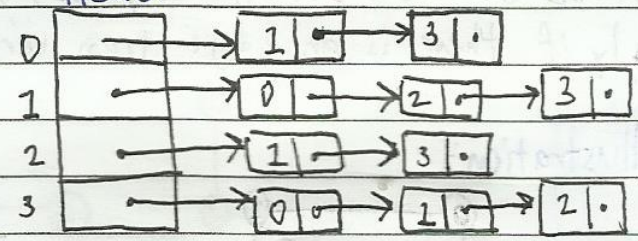
a.) Linked List Representation

A graph of n vertices is represented by an array, declared as HEAD, of n components. Storage requirement is proportional to the sum of the no. of vertices + no. of arcs

* Illustration



Head



// The results is affected by the definition in adjacency in undirected graph

Sterling

NO.

DATE

{Minimum Cost Spanning Tree} (MST)

- Suppose $G=(V,E)$ is a connected graph in which each edge (u,v) in E has a cost $c(u,v)$ attached to it.
- A spanning tree for G is a free tree that connects all the vertices in V.
- The cost of a spanning tree is the sum of the costs of the edges in the tree.
- One important application of MST is in the design of communication networks.

* Two Techniques used in constructing an MST:
  1.) Prim's Algorithm
  2.) Kruskal's Algorithm

{Prim's Algorithm}

→ Grows a spanning tree, one edge at a time

Given:
1. Graph $G=(V,E)$
2. $V=\{0,1,2,3,4,5\}$

Steps of Prim's Algorithm:
1. Initialize set U to contain {0}
2. while $(U != V)$
   a.) Find the minimum cost edge (u,v) such that $u \in U$ and $v \in V-U$
   b.) Add v to U

Simulation of steps →

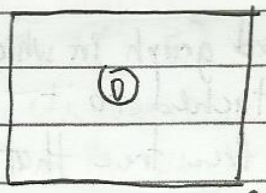
Sterling

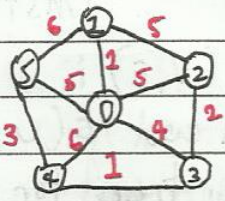
NO.

DATE

{ Steps in Prim's Algorithm }

• Phase 1: Initialize set U to contain {0}





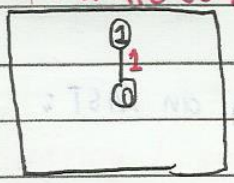
V = {0, 1, 2, 3, 4, 5}

U = {0, }

V - U = {1, 2, 3, 4, 5}

Min Cost = 0

• Phase 2 a: while (U != V)



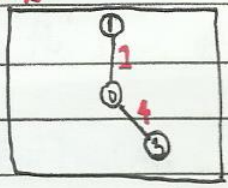
V = {0, 1, 2, 3, 4, 5} Find min. cost

U = {0, 1 } edge = (0, 1)

V - U = {1, 2, 3, 4, 5} add v to U

Min Cost = 0 + 1 = 1

• Phase 2 b

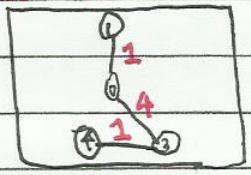


V = {0, 1, 2, 3, 4, 5} U = {0, 1, 3 }

V - U = {2, 3, 4, 5} min cost = 1 + 4 = 5

min cost edge = (0, 3)

• Phase 2 c

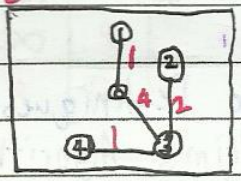


V = {0, 1, 2, 3, 4, 5} U = {0, 1, 3, 4 }

V - U = {2, 5} min cost = 5 + 1 = 6

min cost edge = (3, 4)

• Phase 2 d

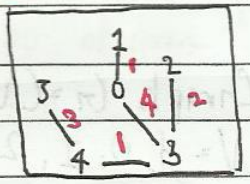


V = {0, 1, 2, 3, 4, 5} U = {0, 1, 3, 4, 2 }

V - U = {5} min cost = 6 + 2 = 8

min cost edge = (3, 2)

• Phase 2 e

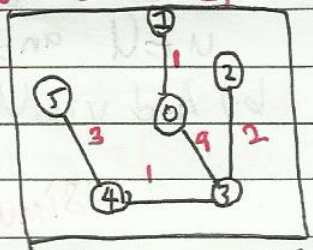


V = {0, 1, 2, 3, 4, 5} U = {0, 1, 3, 4, 2, 5}

V - U = { } min cost = 8 + 3 = 11

min cost edge = (4, 5)

• FINAL RESULT



Min Cost = 11

Sterling

NO.

DATE

{ Kruskal's Algorithm }

→ starts with a forest untill all the vertices are in 1 component

Given:

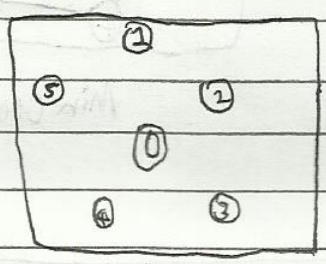
1. Graph G = (V, E)
2. V = {0, 1, 2, 3, 4, 5}

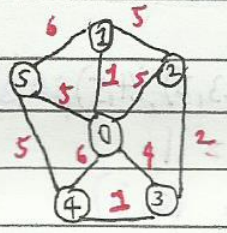
Steps of Kruskal's Algorithm:

1. Initialize Graph T = (V, ∅) consisting of:
   the vertices in V & having no edges.
   Each vertex is a connected component by itself.
2. While (all the vertices are NOT yet in 1 component)
   - Find the edge (u, v) with the minimum cost that will connect to vertices in two different components.

// SIMULATION

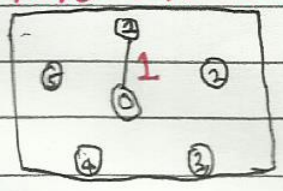
- Phase 1: Initialize Graph T = (V, ∅)





V = {0, 1, 2, 3, 4, 5}
E = {
Min Cost = 0

- Phase 2a: While Not 1 component only



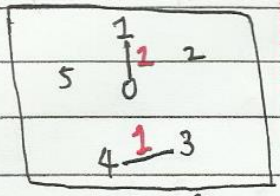
V = {0, 1, 2, 3, 4, 5}
E = {(0, 1)
Min Cost = 0 + 1 = 1

} min cost edge (u, v) connecting vertices of two diff. components - (0, 1)

NO.

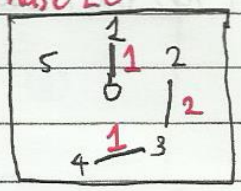
DATE

• Phase 2b



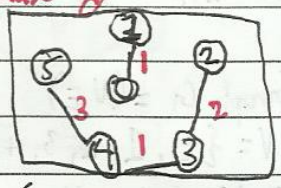
V={0,1,2,3,4,5}
E={(0,1),(3,4)}
Min Cost = 1+1=2
min cost edge = (3,4)

• Phase 2c



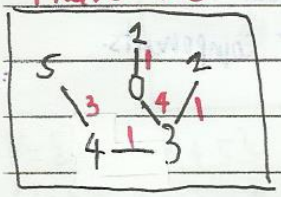
V={0,1,2,3,4,5}
E={(0,1),(3,4),(3,2)}
Min Cost = 2+2=4
Min Cost edge = (3,2)

• Phase 2d



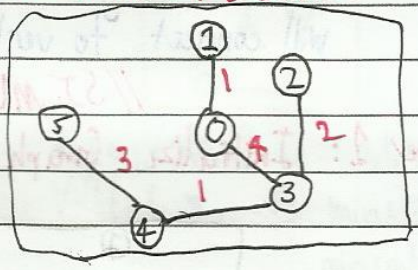
V={0,1,2,3,4,5}
E={(0,1),(3,4),(3,2),(4,5)}
Min Cost = 4+3=7
min cost edge = (4,5)

• Phase 2e



V={0,1,2,3,4,5}
E={(0,1),(3,4),(3,2),(4,5),(0,3)}
min cost = 7+4=11
min cost edge = (0,3)

• FINAL RESULT



Min Cost = 11

```
// side tangent for bfs traversal
// cuz i forgor to put this
function bfs(vertex v){
  QUEUE of vertex Q;
  vertex x, y;
  mark[v] = visited
  ENQUEUE(v, Q)
  While (not EMPTY(Q)) {
     x = Front(Q);
     DEQUEUE(Q);
        for each vertex y adjacent to x do
           if(mark[y] == unvisited){
              mark[y] = visited
              ENQUEUE(y, Q)
              INSERT((x,y), T)
```

Sterling