PDSA Notes

by

Gagneet Kaur

Date
January 17, 2022

PAGE NO.
DATE:

WEEK 4

→ GRAPHS ←

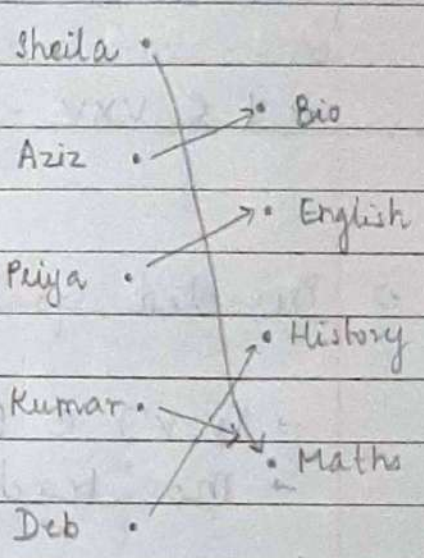
VISUALISING RELATIONS AS GRAPHS

- Teachers and courses

  → T, set of teachers in a college
  C, set of courses being offered

  → $A \subseteq T \times C$ describes the allocation of teachers to courses

  → $A = \{(t,c) \mid (t,c) \in T \times C, t \text{ teaches } c\}$

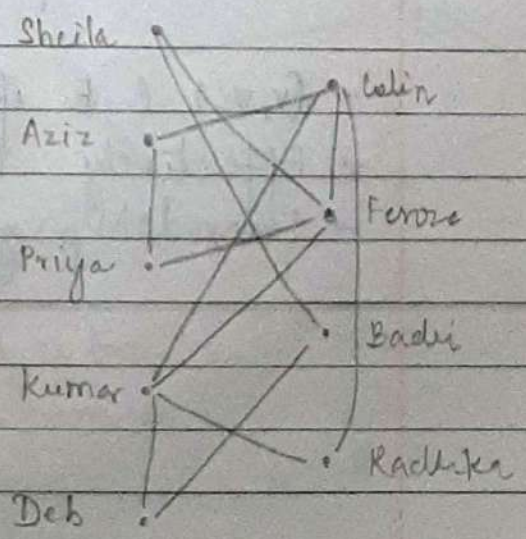


- Friendships

  → P, a set of students

  → $F \subseteq P \times P$ describes which pairs of students are friends

  → $F = \{(p,q) \mid p,q \in P, p \neq q, p \text{ is a friend of } q\}$



PAGE NO.
DATE:

→ $(p,q) \in F$ iff $(q,p) \in F$

GRAPHS

A graph represents relationships between entities
→ entities are vertices / nodes
→ relationships are edges

-> Graph : $G = (V, E)$

→ V is a set of vertices or nodes
· One vertex, many vertices

→ E is set of edges

→ $E \subseteq V \times V$ - binary relation

A graph may be directed or undirected
→ A is parent of B (directed)
→ A is friend of B (undirected)

-> Directed Graph

→ $(v, v') \in E$ does not imply $(v', v) \in E$
→ The teacher course graph is directed

-> Undirected Graph

→ $(v, v') \in E$ iff $(v', v) \in E$
→ Effectively (v, v'), (v', v) are the same edge
→ Friendship graph is undirected

PATHS

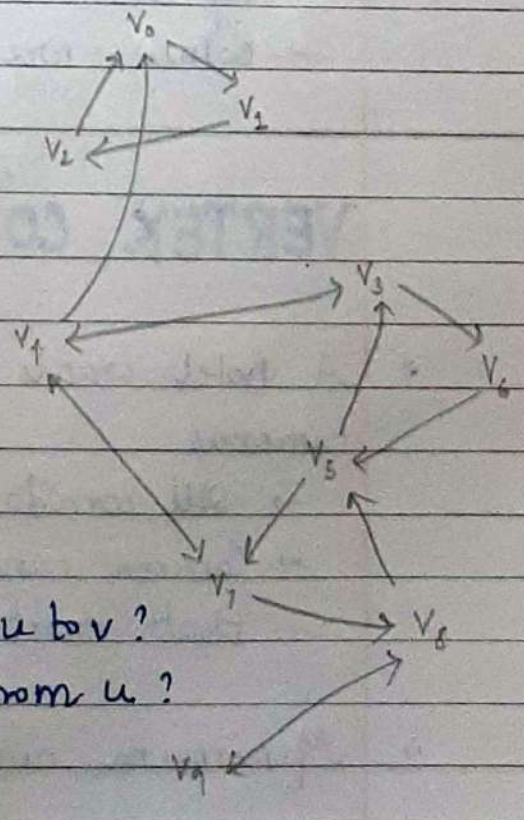
Paths are sequences of connected edges

- A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ connected by edges
  - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$
- Normally, a path does not visit a vertex twice
- A sequence that re-visits a vertex is usually called a 'walk'
  - Kumar - Feroze - Colin - Aziz - Priya - Feroze - Sheila

REACHABILITY

- Paths in directed graphs
- How can I fly from Madurai to Delhi?
  - Find a path from V_9 to V_0
- Vertex v is reachable from vertex u if there is a path from u to v.
- Typical Questions:
  - Is v reachable from u?
  - What is the shortest path from u to v?
  - What are the vertices reachable from u?
  - Is the graph connected?

AIRLINE ROUTES



* Graphs are useful abstract representations for a wide range of problems

PAGE NO.
DATE:

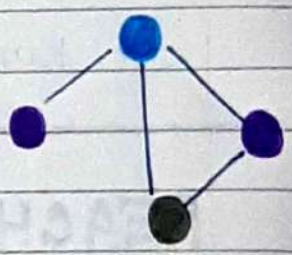
GRAPH COLOURING

- Graph $G = (V, E)$, set of colours C

English
Maths
History
Science

- Colouring is a function $c : V \to C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$

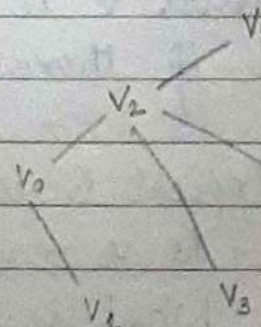
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
  - → Four colour Theorem for planar graphs derived from geographical maps, 4 colours suffice
  - → Not all graphs are planar. General case?



- How many classrooms do we need?
  - → Courses & timetable slots, edges represent overlapping slots
  - → colours are classrooms

VERTEX COVER

- A hotel wants to install security cameras
  - → All corridors are straight lines
  - → Camera can monitor all corridors that meet at an intersection

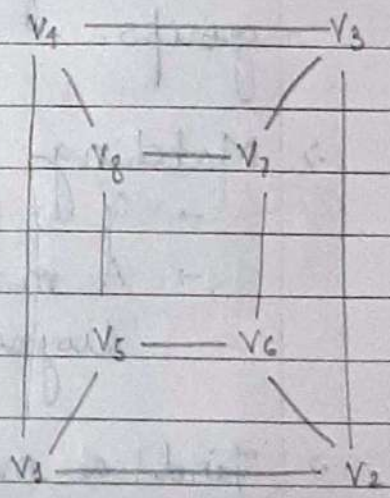


- Minimum number of cameras needed?

- Represent the floor plan as graph
  - V - intersection of corridors
  - E - corridor segments connecting intersections
- Vertex Cover
  - Marking v covers all edges from v
  - Mark smallest subset of V to cover all edges

INDEPENDENT SET

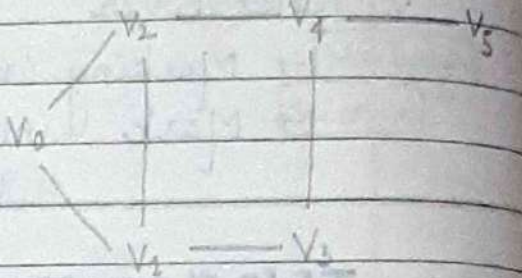
- A dance school puts up group dances
  - Each dance has a set of dancers
  - sets of dancers may overlap across dances



- Organizing a cultural programme
  - Each dancer performs at most once
  - Maximum no. of dances possible?
- Represent the dances as a graph
  - V - dances
  - E - sets of dancers overlap
- Independent Set
  - subset of vertices such that no two are connected by an edge

MATCHING

- Class project can be done by one or two people
  - If two people, they must be friends



- Assume we have a graph describing friendships
- Find a good allocation of groups.
- Matching
  - $G = (V, E)$, an undirected graph
  - A matching is a subset $M \subseteq E$ of mutually disjoint edges
- Find a maximal matching in G
- Is there a perfect matching, covering all vertices?

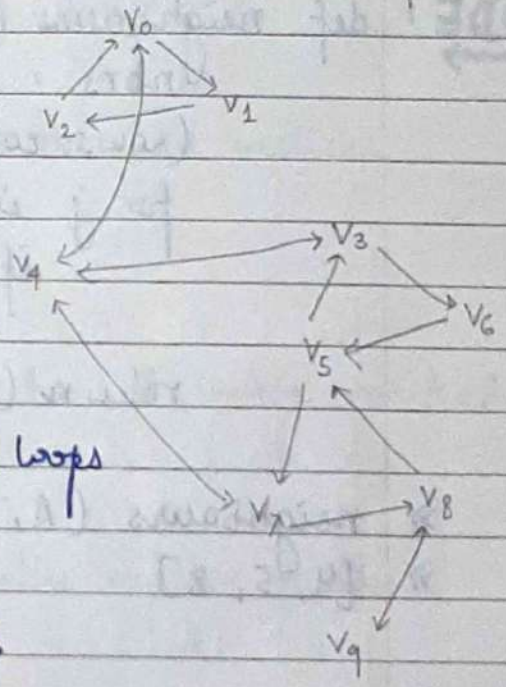
Date: January 18, 2022

Shrinidhi
PAGE NO.
DATE:

Representing Graphs

ADJACENCY MATRIX

Airline Routes



- Let $|V| = n$
  - Assume $V = \{0, 1, \ldots, n-1\}$
  - Use a table to map actual vertex 'names' to this set
- Edges are now pairs (i, j), where $0 \leq i, j < n$
  - Usually assume $i \neq j$, no self loops
- Adjacency Matrix
  - Rows and columns numbered $\{0, 1, \ldots, n-1\}$
  - $A[i, j] = 1$ if $(i, j) \in E$

CODE →

```
edges = [(0,2), (0,4), (1,2), (2,0), (3,4), (3,6), (4,0), (4,3),
         (4,7), (5,3), (5,7), (6,5), (7,4), (7,8), (8,5), (8,9),
         (9,8)]

import numpy as np
A = np.zeroes(shape = (10,10))

for (i,j) in edges:
    A[i,j] = 1
```

Shrinath
PAGE NO.
DATE:

COMPUTING WITH ADJ. MATRIX

-> Neighbours of i - column j with entry 1
  → Scan row i to identify neighbours of i
    → Neighbours of 6 are [3, 5]

CODE:

```
def neighbours (AMat, i):
    nbrs = []
    (rows, cols) = AMat.shape
    for j in range (cols):
        if AMat [i, j] == 1:
            nbrs.append(j)
    return (nbrs)
```

```
» neighbours (A, 7)
» [4, 5, 8]
```

-> Directed Graph
  → Rows represent outgoing edges
  → Columns represent incoming edges

-> Degree of a vertex i
  → No. of edges incident on i
    degree (6) = 2
  → For directed graphs, outdegree & indegree
    indegree (6) = 1 , outdegree (6) = 1

CHECKING REACHABILITY

- Is Delhi (0) reachable from Madurai (9) ?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices
- Stop when 0 becomes marked
- If marking process stops without target being marked, the target is unreachable
- Need a strategy to systematically explore marked neighbours
- Two primary strategies
  - BFS
  - DFS

ADJACENCY LISTS

- Adjacency matrix has many 0's
  - Size is n^2, regardless of no. of edges
  - Undirected graph : $|E| \leq n(n-1)/2$
  - Directed graph : $|E| \leq n(n-1)$
  - Typically $|E|$ is much less than n^2

Shrinath
PAGE NO.
DATE:

- Adjacency List
  - List of neighbours for each vertex

0	[1,4]
1	[2]
2	[0]
3	[4,6]
4	[0,3,7]
5	[3,7]
6	[5]
7	[4,8]
8	[5,9]
9	[8]

CODE:

```
AList = {}

for i in range(10):
    AList[i] = []
for (i,j) in edges:
    AList[i].append(j)

print(AList)
```

» {0: [1,4], 1: [2], 2: [0], 3: [4,6], 4: [0,3,7], 5: [3,7], 6: [5], 7: [4,8], 8: [5,9], 9: [8]}

COMPARING REPRESENTATIONS

- Adjacency list typically requires less space
- Is 'j' a neighbour of 'i'?
  - Check if A[i,j] = 1 in adj. matrix
  - Scan all neighbours of i in adj list

- Which are the neighbours of i ?
  - → Scan all n entries in row i in adjacency matrix
  - → Takes time proportional to (out) degree of i in adj. list
- Choose representation depending on requirement.

→ SUMMARY ←

- To operate on graphs, we need to represent them
- Adjacency matrix
  - → n×n matrix ; AMat[i,j] = 1 iff (i,j) ∈ E
- Adjancency List
  - → Dictionary of lists
  - → For each vertex i, AList[i] is the list of neighbours of i
- Can systematically explore a graph using these representations

Date
January 19, 2022

PAGE NO.
DATE:

Breadth First Search

- Explore the graph level by level
- Each visited vertex has to be explored
  - Extend the search to its neighbours
  - Do this only once for each vertex!
- Maintain information about vertices
  - Which vertices have been visited already
  - Among these, which are yet to be explored.
- Assume $V = \{0, 1, \ldots, n-1\}$
- visited : $V \rightarrow$ {True, False} tells us whether $v \in V$ has been visited
  - Initially, visited(v) = False for all $v \in V$
- Maintain a sequence of visited vertices yet to be explored
  - A queue - first in, first out
  - Initially empty
- Exploring a vertex i
  - For each edge (i, j), if visited(j) is False,
    - Set visited(j) to True
    - Append j to the queue

PAGE NO.
DATE:

- Initially
  - visited (v) = False for all $v \in V$
  - Queue of vertices to be explored is empty
- Start BFS from vertex j
  - Set visited (j) = True
  - Add j to the queue
- Remove and explore vertex i at the head of queue
  - For each edge (i, j), if visited (j) is False
    - Set visited (j) to True
    - Append j to the queue
- Stop when queue is Empty

→ CODE :

```
class Queue :
    def __init__(self):
        self.queue = []

    def addq(self, v):
        self.queue.append(v)

    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
```

```
def isempty(self):
    return (self.queue == [])

def __str__(self):
    return (str(self.queue))
```

```
q = Queue()
for i in range(3):
    q.addq(i)
    print(q)
print(q.isempty())
for j in range(3):
    print(q.delq(), q)
print(q.isempty())
```

OUTPUT

```
[0]
[0, 1]
[0, 1, 2]
False
0 [1, 2]
1 [2]
2 []
True
```

```
def BFS(AMat, v):
    (rows, cols) = AMat.shape
    visited = {}
    for i in range(rows):
        visited[i] = False
    q = Queue()

    visited[v] = True
    q.addq(v)

    while (not q.isempty()):
        j = q.delq()
```

```
        for k in neighbours(AMat, j):
            if (not visited[k]):
                visited[k] = True
                q.addq(k)

    return(visited)
```

COMPLEXITY OF BFS

- G = (V, E)
  - |V| = n
  - |E| = m
  - If G is connected, m can vary from (n-1) to n(n-1)/2

- In BFS, each reachable vertex is processed exactly once
  - visit the vertex : add to queue
  - Explore the vertex : remove from queue
  - visit & Explore at most n vertices

- Exploring a vertex
  - Check all outgoing edges
  - How long does it take?

ADJACENCY MATRIX

- To explore i, scan neighbours(i)
- Look up n entries in row i, regardless of degree(i)

ADJACENCY LIST

- List neighbours (i) is directly available
- Time to explore i is degree (i)
- Degree varied across vertices

SUM OF DEGREES

- Sum of degrees is $2m$
- Each edge (i, j) contributes to degree (i) & degree(j)

BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

BFS with adjacency list

- n steps to initialize each vertex
- $2m$ steps (sum of degrees) to explore all vertices (An example of amortized analysis)
- Overall time is $O(n+m)$
- If $m \ll n^2$, working with adjacency lists is much more efficient (That is why we treat m and n as separate parameters)
- For graphs, $O(m+n)$ is typically the best possible complexity.
  - Need to see each vertex & edge at least once
  - Linear Time

ENHANCING BFS TO RECORD PATHS

- If BFS from i sets visited(k) = True, we know that k is reachable from i
- How do we recover a path from i to k?
- visited(k) was set to True when exploring some vertex j
- Record parent(k) = j
- From k, follow parent links to trace back a path to i

CODE:

```
def BFSListPath(AList, v):
    (visited, parent) = ({}, {})
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    q = Queue()

    visited[v] = True
    q.addq(v)

    while (not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (not visited[k]):
                visited[k] = True
                parent[k] = j
                q.addq(k)
    return (visited, parent)
```

Enhancing BFS to record distance

- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex
- Instead of visited(j), maintain level(j)
- Initialize level(j) = -1 for all j
- Set level(i) = 0 for source vertex
- If we visit k from j, set level(k) to level(j) + 1
- level(j) is the length of the shortest path from the source vertex, in number of edges

CODE :

```
def BFSListPathLevel(AList, v):
    (level, parent) = ({}, {})
    for i in AList.keys():
        level[i] = -1
        parent[i] = -1
    q = Queue()

    level[v] = 0
    q.addq(v)

    while (not q.isemptyq()):
        j = q.delq()
```

```
        for k in AList[j]:
            if (level[k] == -1):
                level[k] = level[j] + 1
                parent[k] = j
                q.addq(k)
    return (level, parent)
```

→ SUMMARY ←

- → BFS is a systematic strategy to explore a graph, level by level
- → Record which vertices have been visited
- → Maintain visited but unexplored vertices in a queue
- → Complexity is $O(n^2)$ using adjacency matrix ; $O(m+n)$ using adjacency list
- → Add parent information to recover the path to each reachable vertex
- → Maintain level information to record length of the shortest path, in terms of no. of edges
  - In general, edges are labelled with a cost (distance, time, ticket price, ...)
  - Will look at weighted graphs, where shortest paths are in terms of cost, not no. of edges

Dat
January 20, 2022

PAGE NO.
DATE:

Depth First Search

- Start from i, visit an unexplored neighbour
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a STACK
  - Last in, first out
  - Most recently suspended is checked first

IMPLEMENTING DFS

- DFS is most natural to implement recursively.
  - For each unvisited neighbour of v, call DFS(v)
- No need to maintain a stack
  - Recursion implicitly maintains stack
  - Separate initialization step
- Can make visited and parent global
  - Still need to initialize them according to the size of input adjacency matrix / list

Shrinath
PAGE NO.
DATE:

→ Use an adjacency list instead

CODE 1:

```
def DFSInit (AMat) :
    (rows , cols) = AMat . shape
    (visited , parent) = ({}, {})
    for i in range (rows) :
        visited [i] = False
        parent [i] = -1
    return (visited , parent)

def DFS (AMat , visited , parent , v) :
    visited [v] = True

    for k in neighbours (AMat , v) :
        if (not visited [k]) :
            parent [k] = v
            (visited , parent) = DFS (AMat , visited ,
                                      parent , k)
    return (visited , parent)
```

CODE 2: (visited , parent) = ({}, {})

```
def DFSInitGlobal (AMat) :
    (rows , cols) = AMat . shape
    for i in range (rows) :
        visited [i] = False
        parent [i] = -1
    return
```

```
def DFSGlobal(AMat, v):
    visited[v] = True

    for k in neighbours(AMat, v):
        if (not visited[k]):
            parent[k] = v
            DFSGlobal(AMat, k)
    return
```

CODE 3:

```
def DFSInitList(AList):
    (visited, parent) = ({}, {})
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    return (visited, parent)

def DFSList(AList, visited, parent, v):
    visited[v] = True

    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            (visited, parent) = DFSList(AList, visited,
                                        parent, k)
    return (visited, parent)
```

CODE 4: (visited, parent) = ({}, {})

```
def DFSInitListGlobal(AList):
```

```
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    return

def DFSListGlobal(AList, v):
    visited[v] = True

    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            DFSListGlobal(AList, k)
    return
```

COMPLEXITY OF DFS

- Like BFS, each vertex is marked & explored once
- Exploring vertex v requires scanning all neighbours of v
  - → $O(n)$ time for adjacency matrix, independent of degree(v)
  - → degree(v) time for adjacency list
    - Total time is $O(m)$ across all vertices
- Overall complexity is same as BFS
  - → $O(n^2)$ using adjacency matrix
  - → $O(n+m)$ using adjacency list

PAGE NO.
DATE:

→ SUMMARY ←

- → DFS is another systematic strategy to explore a graph
- → DFS uses a stack to suspend exploration & move to unexplored neighbours
- → Paths discovered by DFS are not shortest paths, unlike BFS
- → Useful features can be found by recording the order in which DFS visits vertices

---

Applications of BFS and DFS

BFS & DFS

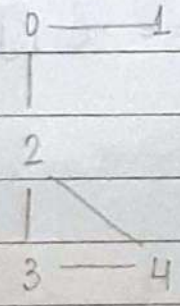
- BFS and DFS systematically compute reachability in graphs
- BFS works level by level
  - → Discovers shortest paths in terms of no. of edges
- DFS explores a vertex as soon as it is visited neighbours
  - → Suspend a vertex while exploring its neighbours
  - → DFS numbering describes the order in which vertices are ex

- Beyond reachability, what can we find out about a graph using BFS/DFS?

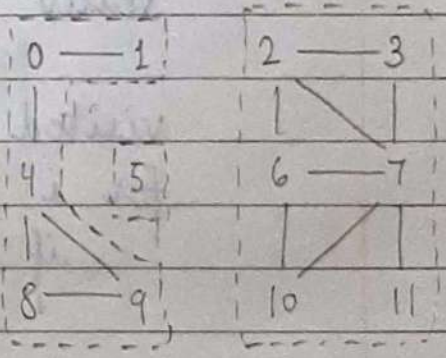
CONNECTIVITY

- An undirected graph is CONNECTED if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
  - → Maximal subsets of vertices that are connected
    - → Isolated vertices are trivial components

Connected Graph



Disconnected Graph



IDENTIFYING CONNECTED COMPONENTS

- Assign each vertex a component number
- Start BFS/DFS from 0
  - → Initialize component number to 0
  - → All visited nodes form a connected component
  - → Assign each visited node component no. 0

- Pick smallest unvisited node j
  - → Increment component number to 1
  - → Run BFS / DFS from node j
  - → Assign each visited node component number 1
- Repeat until all nodes are visited

CODE :

```
def Components (AList):
    component = {}
    for i in AList.keys():
        component[i] = -1

    (compid, seen) = (0,0)

    while seen <= max (AList.keys()):
        startv = min ([i for i in AList.keys() if
                       component[i] == -1])
        visited = BFSList (AList, startv)
        for i in visited.keys():
            if visited[i]:
                seen = seen + 1
                component[i] = compid
        compid = compid + 1

    return (component)
```

DETECTING CYCLES

- A cycle is a path (technically, a walk) that starts and ends at the same vertex

→ 4-8-9-4 is a cycle

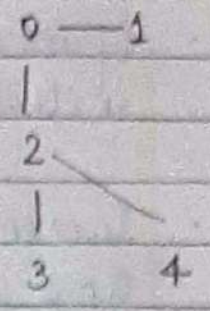
→ Cycle may repeat a vertex:
2-3-7-10-6-7-2

→ Cycle should not repeat edges: i-j-i is not a cycle, e.g. 4-2-4

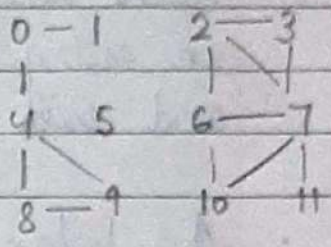
→ SIMPLE CYCLE: only repeated vertices are start & end.

- A graph is acyclic if it has no cycles.

ACYCLIC GRAPH



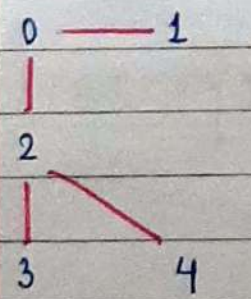
GRAPH WITH CYCLES



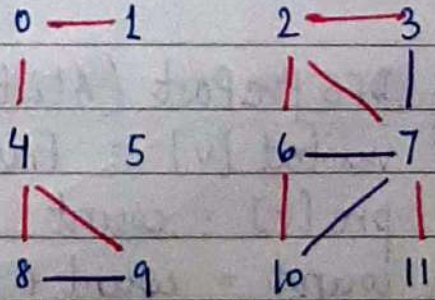
BFS Tree

- Edges explored by BFS form a tree
  → Technically, one tree per component
- Any non-tree edge creates a cycle
  → Detect cycles by searching for non tree edges

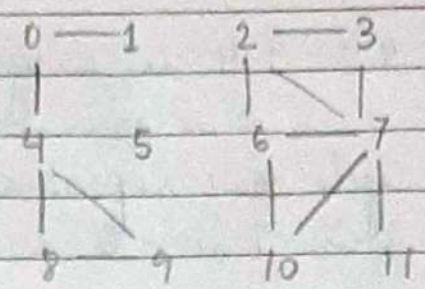
Acyclic Graph



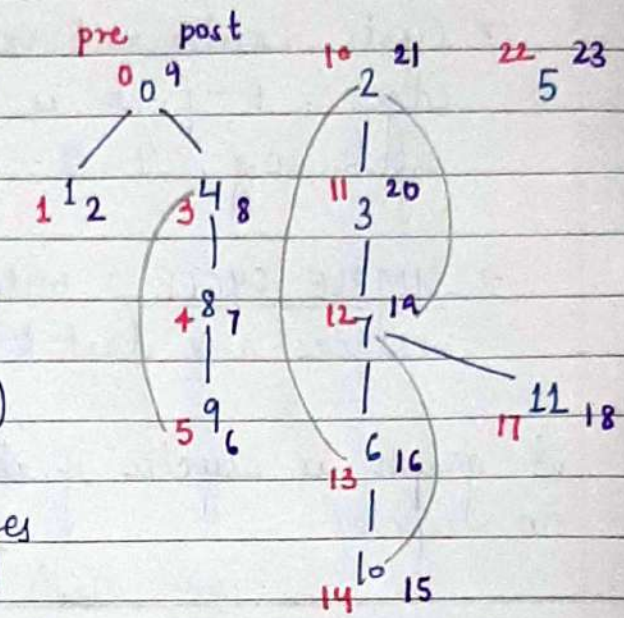
Cyclic Graph



DFS Tree



- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)
- As before, non-tree edges generate cycles
- To compute pre and post pass counter via recursive DFS calls



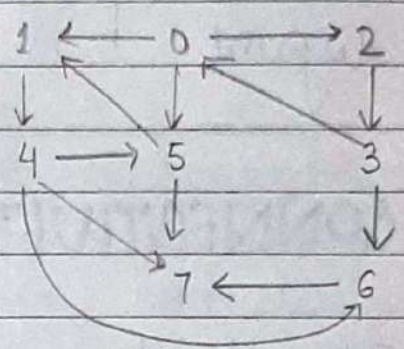
CODE: (visited, pre, post) = ({}, {}, {})

```
def DFSInitPrePost (AList):
    for i in AList.keys():
        visited[i] = False
        (pre[i], post[i]) = (-1, -1)
    return

def DFSPrePost (AList, v, count):
    visited[v] = True
    pre[v] = count
    count = count + 1
```

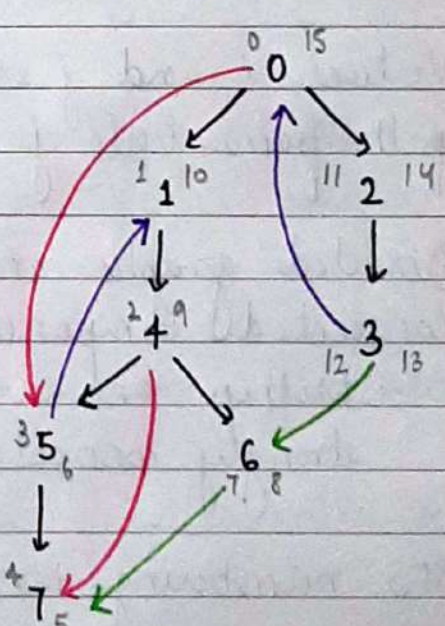
```
for k in AList[v]:
    if (not visited[k]):
        count = DFSPrePost(AList, k, count)
post[v] = count
count = count + 1
return(count)
```

DIRECTED CYCLES



- In a directed graph, a cycle must follow some direction
  - → 0 → 2 → 3 → 0 is a cycle
  - → 0 → 5 → 1 ← 0 is not

- Tree Edges
- Different types of non tree edges
  - → Forward Edges
  - → Back Edges
  - → Cross Edges



- Only back edges correspond to cycles

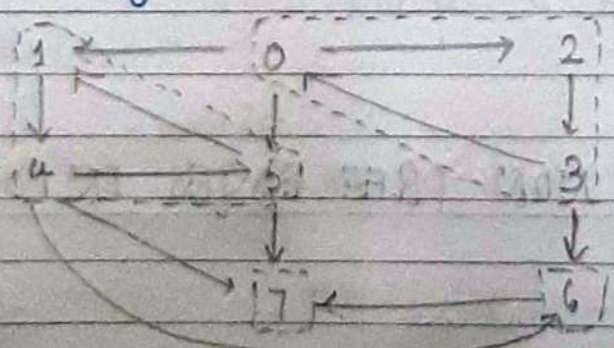
CLASSIFYING NON-TREE EDGES IN DIRECTED GRAPHS

- Use pre/post numbers

- Tree Edge / forward edge (u, v)
  Interval [pre(u), post(u)] contains [pre(v), post(v)]
- Back Edge (u, v)
  Interval [pre(v), post(v)] contains [pre(u), post(u)]
- Cross Edge (u, v)
  Intervals [pre(u), post(u)] and [pre(v), post(v)] are disjoint

CONNECTIVITY IN DIRECTED GRAPHS

- Take directions into account
- Vertices i and j are strongly connected if there is a path from i to j and a path from j to i.
- Directed graphs can be decomposed into strongly connected components (SCCs)
  - Within an SCC, each pair of vertices is strongly connected
- DFS numbering can be used to compute SCCs



→ SUMMARY ←

→ BFS and DFS can be used to identify connected components in an undirected graph.
- BFS & DFS identify an underlying tree, non-tree edges generates cycles

→ In a directed graph, non-tree edges can be forward / back / cross
- Only back edges generate cycles
- Classify non-tree edges using DFS numbering.

→ Directed graphs decompose into strongly connected components
- DFS numbering can be used to compute SCC decomposition

→ DFS numbering can also be used to identify other features such as articulation points (cut vertices) & bridges (cut edges).

→ Directed Acyclic Graphs are useful for representing dependencies
- Given course prerequisites, find a valid sequence to complete a programme

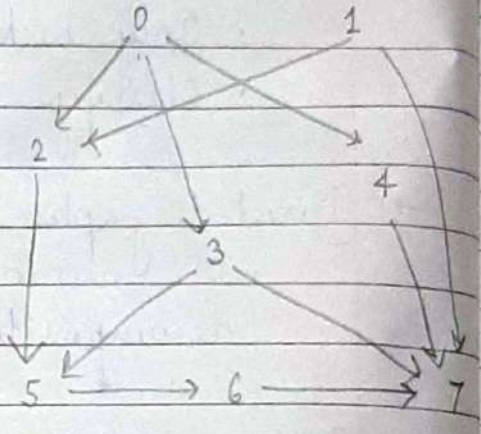
Date
January 21, 2022

PAGE NO.
DATE:

Topological Sorting

DIRECTED ACYCLIC GRAPHS

- $G = (V, E)$, a directed graph without directed cycles
- Topological Sorting
  - → Enumerate $V = \{0, 1, \ldots, n-1\}$ such that for any $(i, j) \in E$, i appears before j
  - → Feasible schedule



- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum no. of semesters to complete the programme!

TOPOLOGICAL SORT

- A graph with directed cycles cannot be ~~de~~ sorted topologically
- Path $i \rightsquigarrow j$ means i must be listed before j
- Cycle ⇒ vertices i, j such that there are paths $i \rightsquigarrow j$

and $j \rightsquigarrow i$.

- i must appear before j, and j must appear before i, impossible!

CLAIM: Every DAG can be topologically sorted.

HOW TO TOPOLOGICALLY SORT A DAG?

- Strategy
  - → First list vertices with no dependencies
  - → As we proceed, list vertices whose dependencies have already been listed
  - .....

- Questions
  - → Why will there be a starting vertex with no dependencies?
  - → How do we guarantee we can keep progressing with the listing?

ALGORITHM FOR TOPOLOGICAL SORT

- A vertex with no dependencies has no incoming edges, indegree(v) = 0

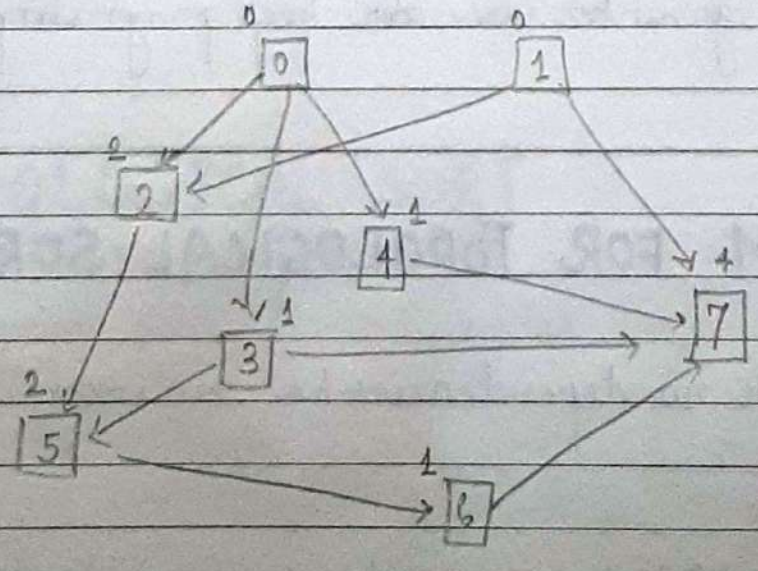
CLAIM: Every DAG has a vertex with indegree 0.

- start with any vertex with indegree > 0
- Follow edge back to one of its predecessors
- Repeat so long as indegree > 0
- If we repeat n times, we must have a cycle, which is impossible in a DAG.

TOPOLOGICAL SORT ALGORITHM

FACT: Every DAG has a vertex with indegree 0.

- List out a vertex j with indegree = 0
- Delete j & all edges from j
- What remains is again a DAG!
- Can find another vertex with indegree = 0 to list and eliminate
- Repeat till all vertices are listed



Topologically sorted sequence : 1, 0, 3, 2, 5, 6, 4, 7

Shrinath
PAGE NO.
DATE:

- compute indegree of each vertex
  → Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with indegree = 0 to list and eliminate
- Repeat till all vertices are listed

CODE :

```
def toposort (AMat) :
    (rows , cols) = AMat.shape
    indegree = { }
    toposortlist = [ ]

    for c in range (cols) :
        indegree [c] = 0
        for r in range (rows) :
            if AMat [r, c] == 1 :
                indegree [c] = indegree [c] + 1

    for i in range (rows) :
        j = min ( [k for k in range (cols) if
                   indegree [k] == 0] )
        toposortlist.append (j)
        indegree [j] = indegree [j] - 1
```

```
            for k in range (cols):
                if AMat[j,k] == 1:
                    indegree[k] = indegree[k] - 1
        return (toposortlist)
```

ANALYSIS (Using MATRIX)

- Initializing indegrees is $O(n^2)$
- Loop to enumerate vertices runs n times
  - → Identify next vertex to enumerate : $O(n)$
  - → Updating indegrees : $O(n)$
- Overall , $O(n^2)$

USING ADJACENCY LISTS

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Enumerate head of queue , update indegrees , add indegree 0 to queue
- Repeat till queue is empty

CODE :

```
def toposortlist (AList) :
    (indegree , toposortlist ) = ( {} , [])
    for u in AList.keys ():
        indegree[u] = 0
    for u in AList.keys ():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0 :
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()) :
        j = zerodegreeq.delq()
        toposortlist.append(j)
        indegree[j] = indegree[j] - 1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            if indegree[k] == 0 :
                zerodegreeq.addq(k)

    return (toposortlist)
```

ANALYSIS (Using LISTS)

- Initializing indegrees is $O(m+n)$
- doop to enumerate vertices run n times
  → Updating indegrees : amortised $O(m)$

PAGE NO.
DATE:

- Overall, $O(m+n)$

→ SUMMARY ←

→ DAGs are a natural way to represent dependencies

→ Topological sort gives a feasible schedule that represents dependencies
- At least one vertex with no dependencies, indegree 0
- Eliminating such a vertex retains DAG structure
- Repeat the process till all vertices are listed

→ Complexity
- Using adjacency matrix takes $O(n^2)$
- Using adjacency list takes $O(m+n)$

→ More than one topological sort is possible
- Choice of which vertex with indegree 0 to list next

Date
January 22, 2022

PAGE NO.
DATE:

Longest Paths in DAG

LONGEST PATH

- Find the longest path in a DAG
- If indegree(i) = 0, longest-path-to(i) = 0
- If indegree(i) > 0, longest path to i is 1 more than longest path to its incoming neighbours

$$\text{longest-path-to}(i) = 1 + \max\{\text{longest-path-to}(j) \mid (j,i) \in E\}$$

- To compute longest-path-to(i), need longest-path-to(k), for each incoming neighbour k
- If graph is topologically sorted, k is listed before i
- Hence, compute longest-path-to ( ) in topological order
- Let $i_0, i_1, \ldots, i_{n-1}$ be a topological ordering of V.
- All neighbours of i_k appear before it in this list.
- From left to right, compute longest-path-to(i_k) as $1 + \max\{\text{longest-path-to}(i_j) \mid (i_j, i_k) \in E\}$
- Overlap this computation with topological sorting

LONGEST PATH ALGORITHM

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize textlongest-path-to to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

USING ADJACENCY LISTS

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- process head of queue : update indegrees, update queue, update longest paths
- Repeat until queue is empty

CODE :

```
def longestpathlist (AList):
    (indegree, lpath) = ({}, {})
    for u in AList.keys():
        (indegree[u], lpath[u]) = (0,0)
    for u in AList.keys():
        for v in AList[u]:
```

```
indegree[v] = indegree[v] + 1

zerodegreeq = Queue()
for u in Alist.keys():
    if indegree[u] == 0:
        zerodegreeq.addq(u)

while (not zerodegreeq.isempty()):
    j = zerodegreeq.delq()
    indegree[j] = indegree[j] - 1
    for k in Alist[j]:
        indegree[k] = indegree[k] - 1
        lpath[k] = max(lpath[k], lpath[j]+1)
        if indegree[k] == 0
            zerodegree.addq(k)

return (lpath)
```

→ SUMMARY ←

→ In parallel with topological sort, we can compute the longest path

→ Notion of longest path makes sense even for graphs with cycles
  - No repeated vertices in a path, so path has at most n-1 edges

→ However computing longest paths in arbitrary graphs is much harder than for DAGs.