Graphs

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

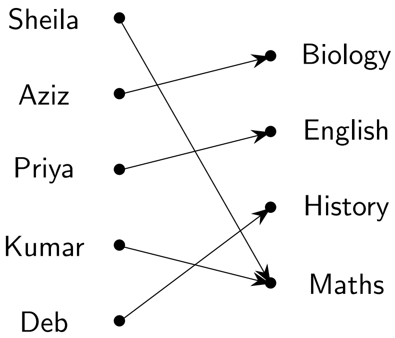
Programming, Data Structures and Algorithms using Python

Week 4

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

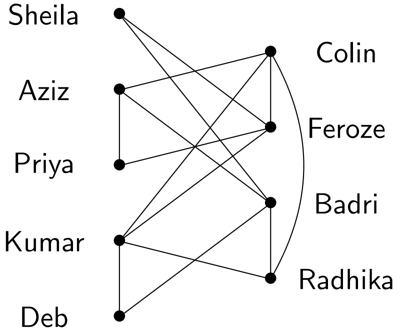
Teachers and courses



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

Friendship



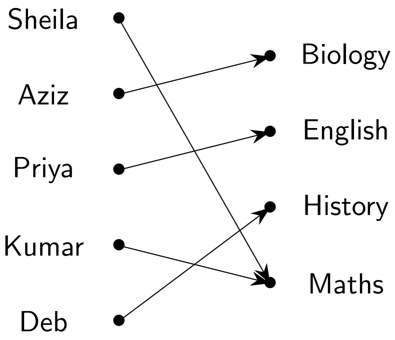
Graphs

- Graph: $G = (V, E)$
  - V is a set of vertices or nodes
    - One vertex, many vertices
  - E is a set of edges
  - $E \subseteq V \times V$ — binary relation

Graphs

- Graph: $G = (V, E)$
  - V is a set of vertices or nodes
    - One vertex, many vertices
  - E is a set of edges
  - $E \subseteq V \times V$ — binary relation
- Directed graph
  - $(v, v') \in E$ does not imply $(v', v) \in E$
  - The teacher-course graph is directed

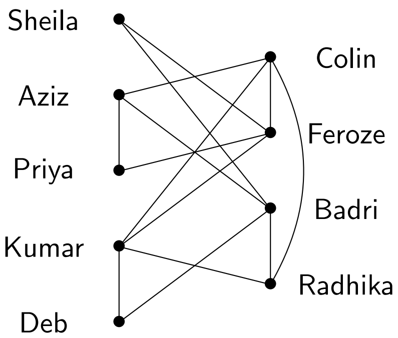
Teachers and courses



Graphs

- Graph: $G = (V, E)$
  - V is a set of vertices or nodes
    - One vertex, many vertices
  - E is a set of edges
  - $E \subseteq V \times V$ — binary relation
- 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

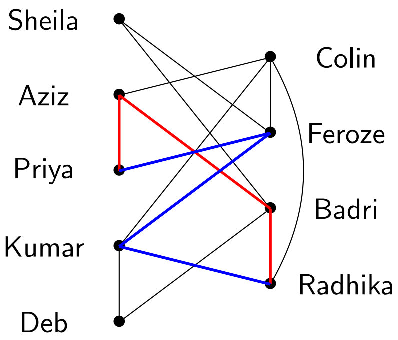
Friendship



Paths

- A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ connected by edges
  - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$

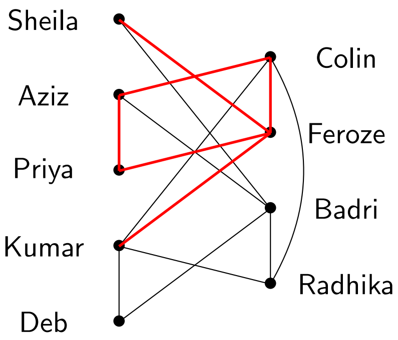
Friendship graph



Paths

- 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

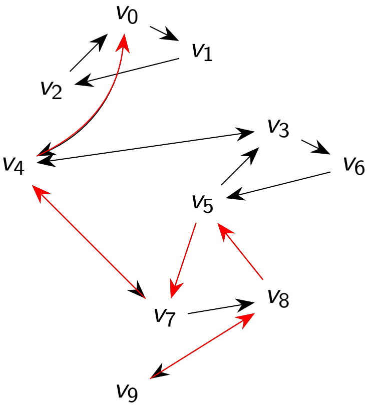
Friendship graph



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

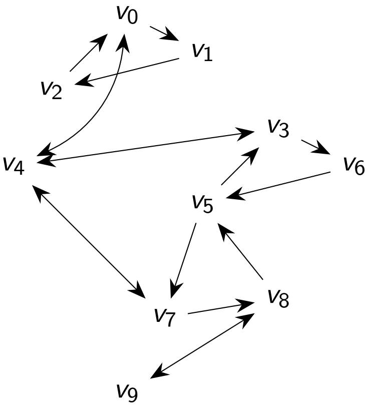
Airline routes



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? Are all vertices reachable from each other?

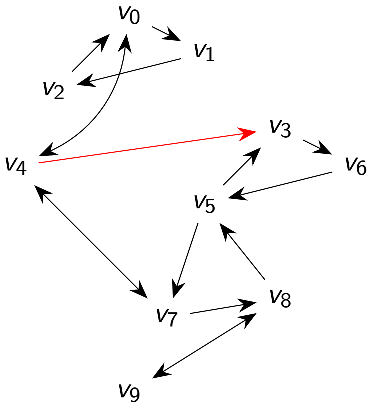
Airline routes



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? Are all vertices reachable from each other?

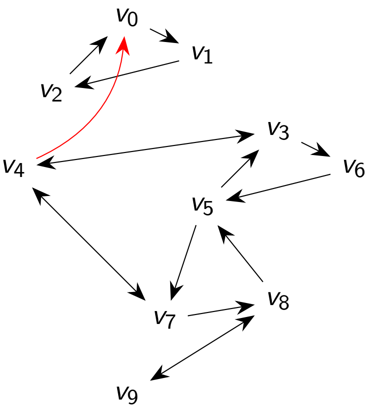
Airline routes



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? Are all vertices reachable from each other?

Airline routes



- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?

Map colouring

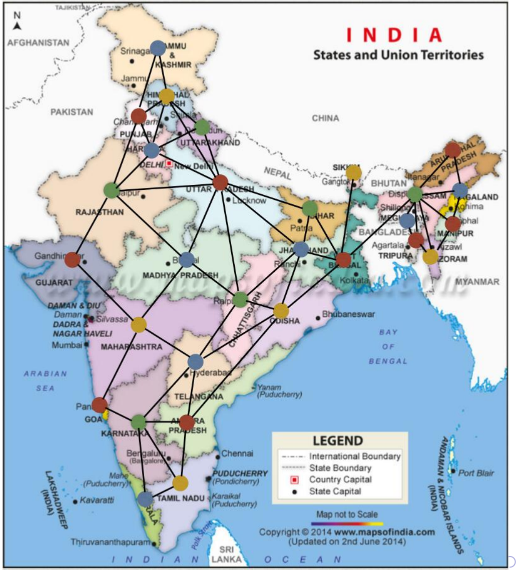
- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
  - Each state is a vertex

Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
  - Each state is a vertex
  - Connect states that share a border

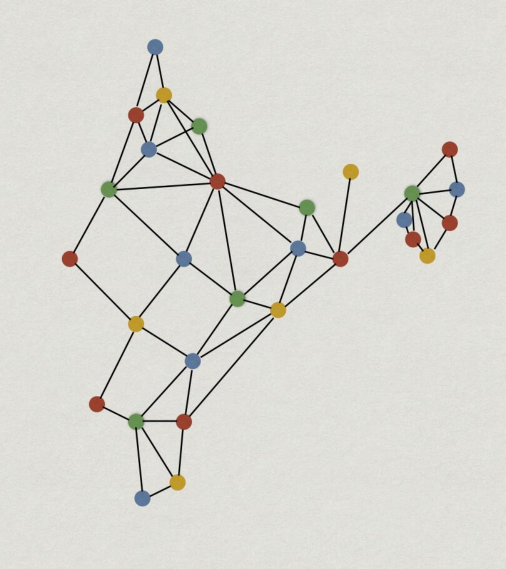
Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
    - Each state is a vertex
    - Connect states that share a border
- Assign colours to nodes so that endpoints of an edge have different colours



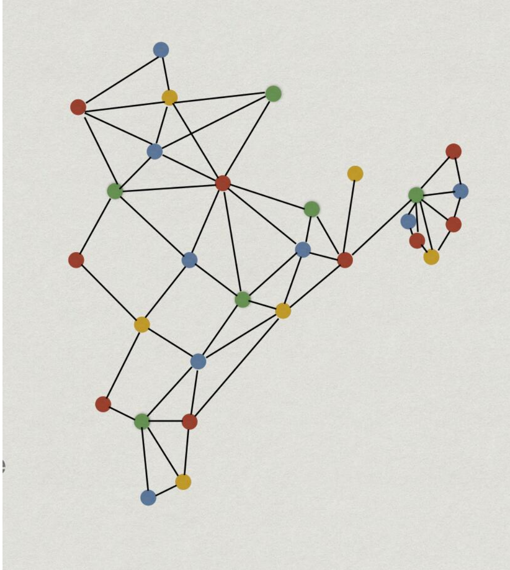
Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
  - Each state is a vertex
  - Connect states that share a border
- Assign colours to nodes so that endpoints of an edge have different colours
- Only need the underlying graph



Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
  - Each state is a vertex
  - Connect states that share a border
- Assign colours to nodes so that endpoints of an edge have different colours
- Only need the underlying graph
- Abstraction: if we distort the graph, problem is unchanged



Graph colouring

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

Graph colouring

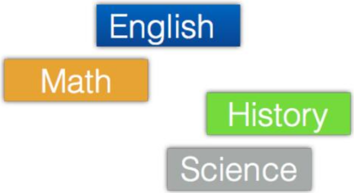
- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \to C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
  - Four Colour Theorem For planar graphs derived from geographical maps, 4 colours suffice

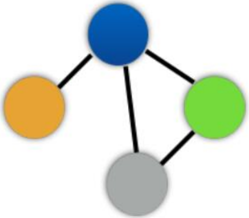
Graph colouring

- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \rightarrow C$ 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? Why do we care?

Graph colouring

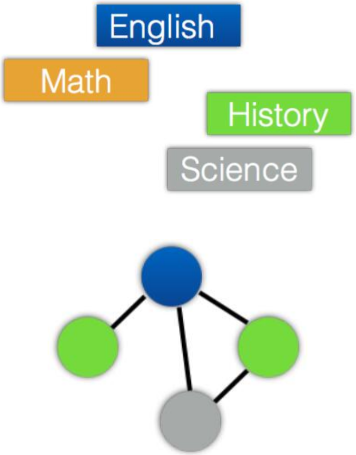
- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \to C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
  - Four Colour Theorem For planar graphs derived from geographical maps, 4 colours suffice
  - Not all graphs are planar. General case? Why do we care?
- How many classrooms do we need?
  - Courses and timetable slots, edges represent overlapping slots
  - Colours are classrooms





Graph colouring

- Graph $G = (V, E)$, set of colours C
- 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? Why do we care?
- How many classrooms do we need?
  - Courses and timetable slots, edges represent overlapping slots
  - Colours are classrooms

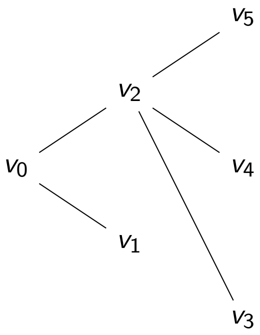


Vertex cover

- 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?

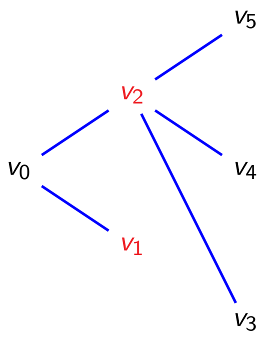
Vertex cover

- 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 a graph
  - V — intersections of corridors
  - E — corridor segments connecting intersections



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 a graph
  - V — intersections 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

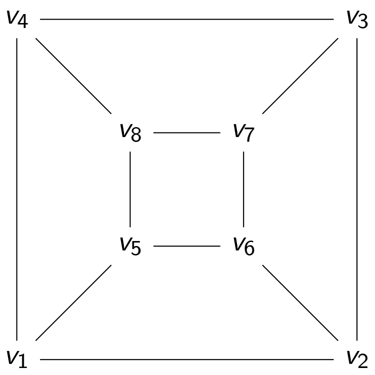


Independent set

- 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 number of dances possible?

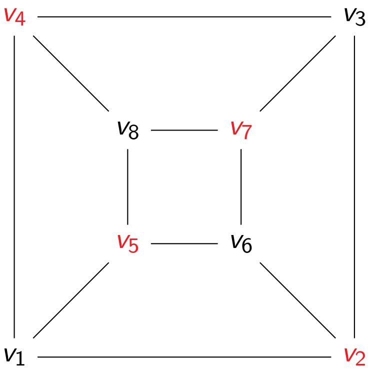
Independent set

- 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 number of dances possible?
- Represent the dances as a graph
  - V — dances
  - E — sets of dancers overlap



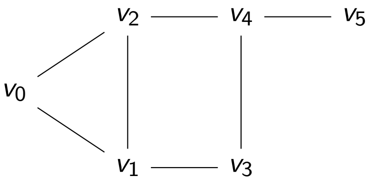
Independent set

- 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 number 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



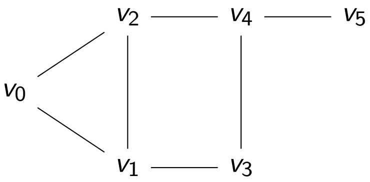
Matching

- Class project can be done by one or two people
  - If two people, they must be friends
- Assume we have a graph describing friendships



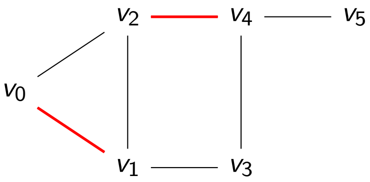
Matching

- 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



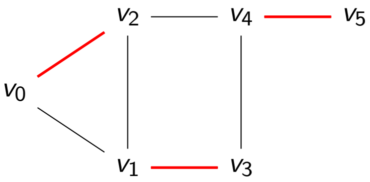
Matching

- 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



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?



Summary

- A graph represents relationships between entities
  - Entities are vertices/nodes
  - Relationships are edges
- A graph may be directed or undirected
  - A is a parent of B — directed
  - A is a friend of B — undirected

Summary

- A graph represents relationships between entities
  - Entities are vertices/nodes
  - Relationships are edges
- A graph may be directed or undirected
  - A is a parent of B — directed
  - A is a friend of B — undirected
- Paths are sequences of connected edges
- Reachability: is there a path from u to v?

Summary

- A graph represents relationships between entities
  - Entities are vertices/nodes
  - Relationships are edges
- A graph may be directed or undirected
  - A is a parent of B — directed
  - A is a friend of B — undirected
- Paths are sequences of connected edges
- Reachability: is there a path from u to v?
- Graphs are useful abstract representations for a wide range of problems
- Reachability and connectedness are not the only interesting problems we can solve on graphs
  - Graph colouring
  - Vertex cover
  - Independent set
  - Matching
  - ...

Representing Graphs

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

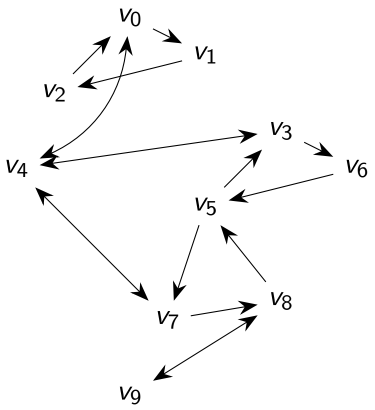
Programming, Data Structures and Algorithms using Python

Week 4

Working with graphs

- Graph $G = (V, E)$
  - V — set of vertices
  - $E \subseteq V \times V$ — set of edges
- A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ connected by edges
  - For $1 \le i < k$, $(v_i, v_{i+1}) \in E$
- Vertex v is reachable from vertex u if there is a path from u to v

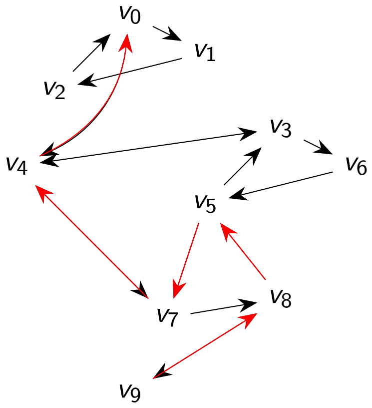
Airline routes



Working with graphs

- Graph $G = (V, E)$
  - V — set of vertices
  - $E \subseteq V \times V$ — set of edges
- A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ connected by edges
  - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$
- Vertex v is reachable from vertex u if there is a path from u to v
- Looking at the picture of G, we can "see" that v_0 is reachable from v_9
- How do we represent this picture so that we can compute reachability?

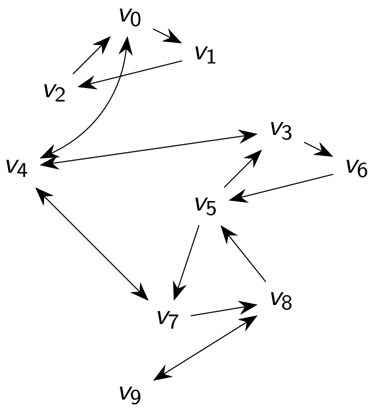
Airline routes



Adjacency matrix

- Let $|V| = n$
  - Assume $V = \{0, 1, \ldots, n-1\}$
  - Use a table to map actual vertex "names" to this set

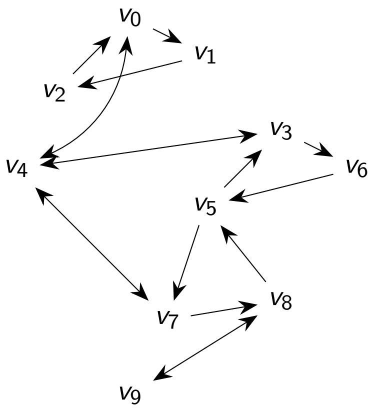
Airline routes



Adjacency matrix

- 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

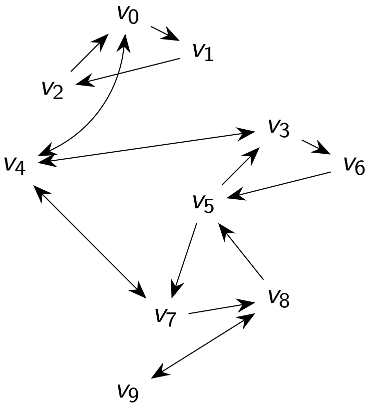
Airline routes



Adjacency matrix

- 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 $(i, j) \in E$

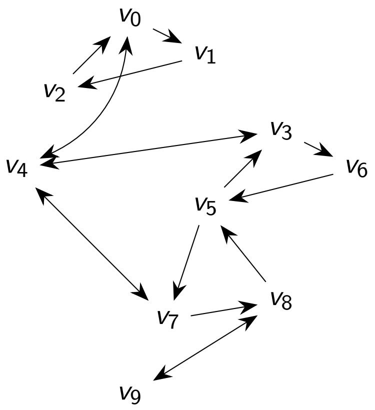
Airline routes



Adjacency matrix

- Adjacency matrix
  - Rows and columns numbered $\{0, 1, \ldots, n-1\}$
  - $A[i,j] = 1$ if $(i,j) \in E$

Airline routes



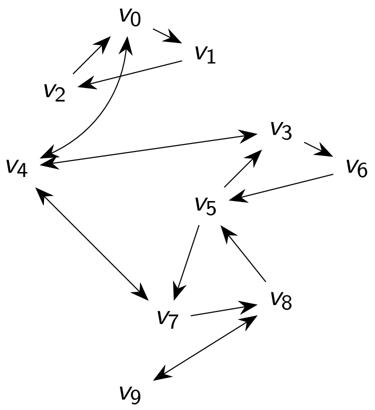
Adjacency matrix

- Adjacency matrix
  - Rows and columns numbered $\{0, 1, \ldots, n-1\}$
  - $A[i, j] = 1$ if $(i, j) \in E$

```
edges = [(0,1),(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.zeros(shape=(10,10))

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

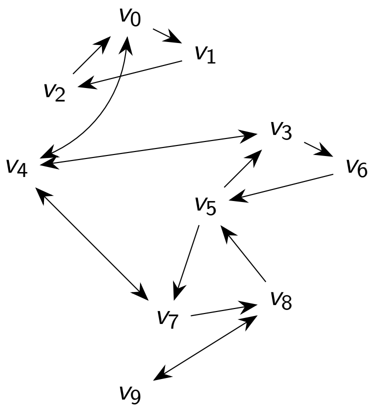
Airline routes



- Adjacency matrix
  - Rows and columns numbered $\{0, 1, \ldots, n-1\}$
  - $A[i,j] = 1$ if $(i,j) \in E$

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

Airline routes

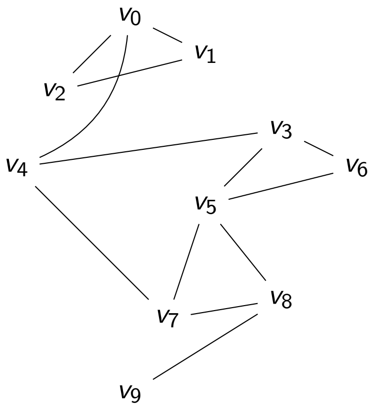


Adjacency matrix

- Undirected graph
  - $A[i,j] = 1$ iff $A[j,i] = 1$
  - Symmetric across main diagonal

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

Airline routes, all routes bidirectional



Computing with the adjacency matrix

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

```
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]
```

Undirected airline routes

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

Computing with the adjacency matrix

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

- Directed graph

Directed airline routes

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

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
  - Scan row i to identify neighbours of i
  - Neighbours of 6 are $[3, 5]$
- Directed graph
  - Rows represent outgoing edges

Directed airline routes

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

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
  - Scan row i to identify neighbours of i
  - Neighbours of 6 are $[3, 5]$
- Directed graph
  - Rows represent outgoing edges
  - Columns represent incoming edges

Directed airline routes

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

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
  - Scan row i to identify neighbours of i
  - Neighbours of 6 are $[3, 5]$
- Directed graph
  - Rows represent outgoing edges
  - Columns represent incoming edges
- Degree of a vertex i
  - Number of edges incident on i

    $\text{degree}(6) = 2$

Undirected airline routes

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

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
  - Scan row i to identify neighbours of i
  - Neighbours of 6 are $[3, 5]$
- Directed graph
  - Rows represent outgoing edges
  - Columns represent incoming edges
- Degree of a vertex i
  - Number of edges incident on i

    $\text{degree}(6) = 2$
  - For directed graphs, outdegree and indegree

    $\text{indegree}(6) = 1$, $\text{outdegree}(6) = 1$

Directed airline routes

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

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?

Undirected airline routes

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

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable

Undirected airline routes

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

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable

Undirected airline routes

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

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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

Undirected airline routes

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

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 becoming marked, the target is unreachable

Undirected airline routes

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

Checking reachability

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked

Undirected airline routes

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

Checking reachability

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Need a strategy to systematically explore marked neighbours

Undirected airline routes

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

Checking reachability

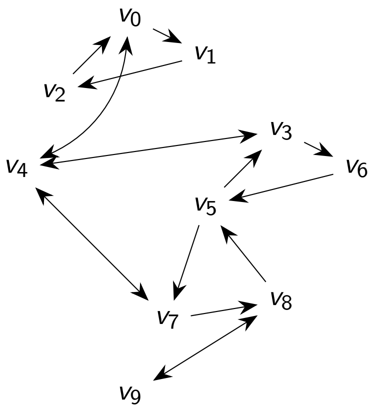
- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Need a strategy to systematically explore marked neighbours
- Two primary strategies
  - Breadth first — propagate marks in "layers"
  - Depth first — explore a path till it dies out, then backtrack

Undirected airline routes

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

- Adjacency matrix has many 0's
  - Size is n^2, regardless of number of edges
  - Undirected graph: $|E| \leq n(n-1)/2$
  - Directed graph: $|E| \leq n(n-1)$
  - Typically $|E|$ much less than n^2

Airline routes



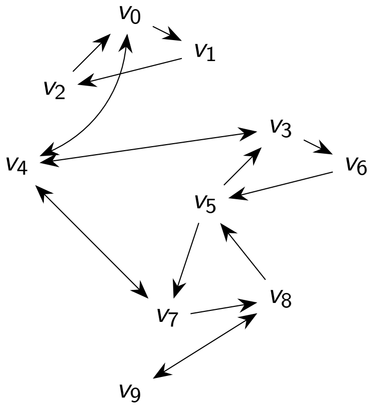
Adjacency lists

- Adjacency matrix has many 0's
  - Size is n^2, regardless of number of edges
  - Undirected graph: $|E| \leq n(n-1)/2$
  - Directed graph: $|E| \leq n(n-1)$
  - Typically $|E|$ much less than n^2
- 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]

Airline routes



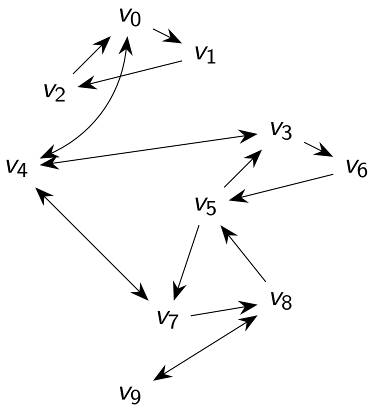
Adjacency lists

- Adjacency matrix has many 0's
  - Size is n^2, regardless of number of edges
  - Undirected graph: $|E| \leq n(n-1)/2$
  - Directed graph: $|E| \leq n(n-1)$
  - Typically $|E|$ much less than n^2
- Adjacency list

```
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]}
```

Airline routes



Comparing representations

- Adjacency list typically requires less space

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

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 adjacency matrix
  - Scan all neighbours of i in adjacency list

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

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 adjacency matrix
  - Scan all neighbours of i in adjacency 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 adjacency list

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

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 adjacency matrix
  - Scan all neighbours of i in adjacency 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 adjacency list
- Choose representation depending on requirement

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

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]

Summary

- To operate on graphs, we need to represent them
- Adjacency matrix
  - $n \times n$ matrix, $AMat[i,j] = 1$ iff $(i,j) \in E$
- Adjacency 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
  - For reachability, propagate marking to all reachable vertices

Breadth First Search

Madhavan Mukund

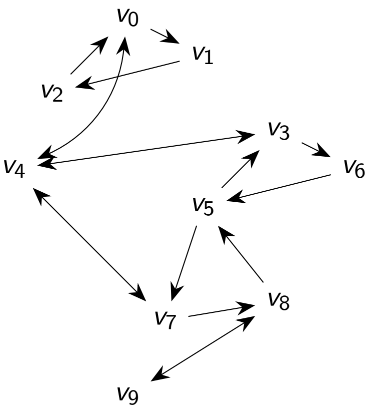
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

Reachability in a graph

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked



Reachability in a graph

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Choose an appropriate representation
  - Adjacency matrix
  - Adjacency list

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

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}

Reachability in a graph

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Choose an appropriate representation
  - Adjacency matrix
  - Adjacency list
- Strategies for systematic exploration
  - Breadth first — propagate marks in "layers"
  - Depth first — explore a path till it dies out, then backtrack

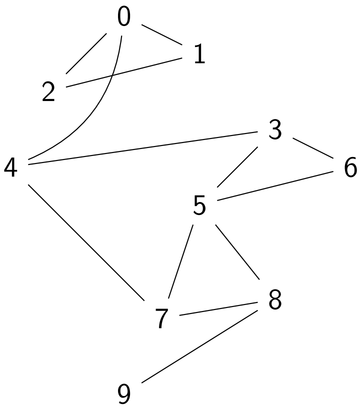
	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

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}

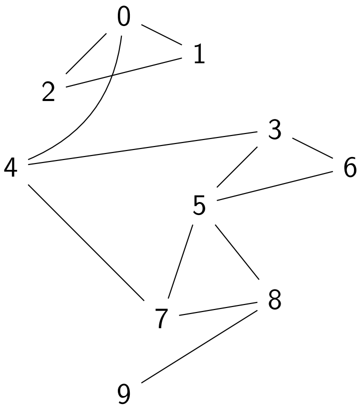
Breadth first search (BFS)

- Explore the graph level by level
  - First visit vertices one step away
  - Then two steps away
  - ...



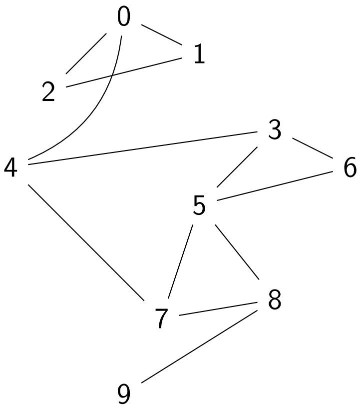
Breadth first search (BFS)

- Explore the graph level by level
  - First visit vertices one step away
  - Then two steps away
  - . . .
- Each visited vertex has to be explored
  - Extend the search to its neighbours
  - Do this only once for each vertex!



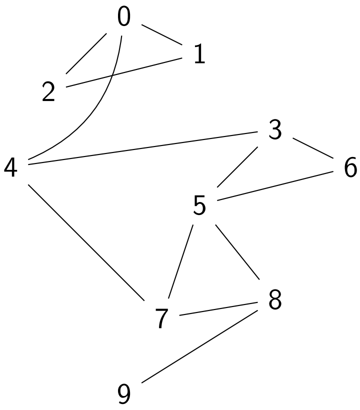
Breadth first search (BFS)

- Explore the graph level by level
  - First visit vertices one step away
  - Then two steps away
  - ...
- 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



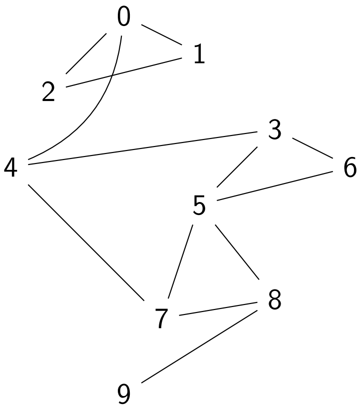
Breadth first search (BFS) . . .

- Assume $V = \{0, 1, \ldots, n-1\}$



Breadth first search (BFS) . . .

- Assume $V = \{0, 1, \ldots, n-1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
  - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$



Breadth first search (BFS) . . .

- Assume $V = \{0, 1, \ldots, n-1\}$
- $\text{visited} : V \to \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
  - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet be explored
  - A queue — first in, first out
  - Initially empty

```
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))
```

Breadth first search (BFS) . . .

- Assume $V = \{0, 1, \ldots, n-1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
  - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet be explored
  - A queue — first in, first out
  - Initially empty

```
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())
```

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

Breadth first search (BFS) ...

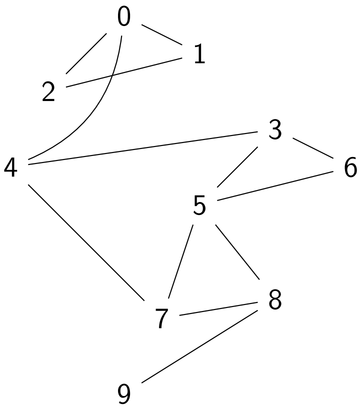
- Assume $V = \{0, 1, \ldots, n-1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
  - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet be explored
  - A queue — first in, first out
  - Initially empty
- Exploring a vertex i
  - For each edge (i, j), if $\text{visited}(j)$ is False,
    - Set $\text{visited}(j)$ to True
    - Append j to the 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())
```

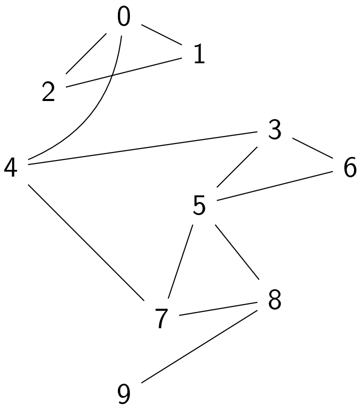
```
[0]
[0, 1]
[0, 1, 2]
False
0 [1, 2]
1 [2]
2 []
True
```

Breadth first search (BFS) ...

- Initially
  - visited(v) = False for all $v \in V$
  - Queue of vertices to be explored is empty

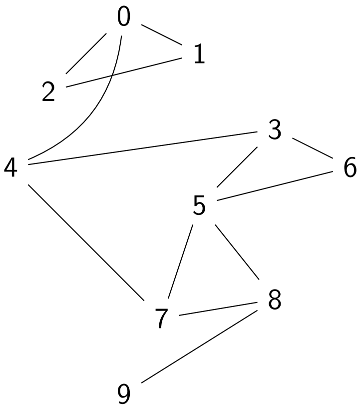


- Initially
  - visited(v) = False for all $v \in V$
  - Queue of vertices to be explored is empty
- Start BFS from vertex j
  - Set visited(j) = True
  - Add j to the queue



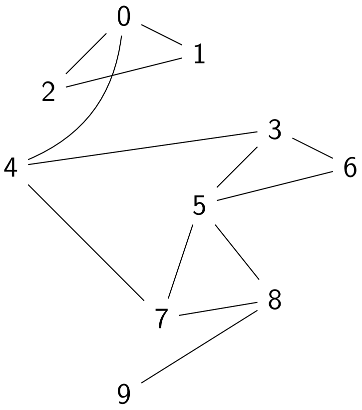
Breadth first search (BFS) ...

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



Breadth first search (BFS) ...

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



Breadth first search (BFS) . . .

- 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 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

```
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)
```

Breadth first search (BFS) ...

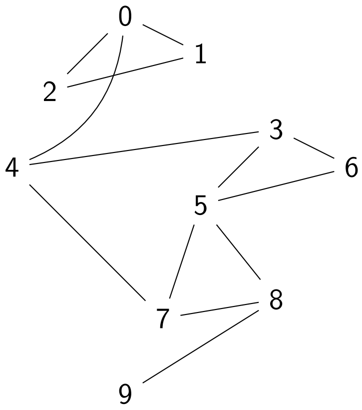
- Initially
  - visited(v) = False for all $v \in V$
  - Queue of vertices to be explored is empty
- Start BFS from vertex j
  - Set visited(j) = True
  - Add j to the queue
- Remove and explore vertex i at head of queue
  - For each edge (i, j), if visited(j) is False,
    - Set visited(j) to True
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

```
def BFSList(AList,v):
    visited = {}
    for i in AList.keys():
        visited[i] = False
    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
                q.addq(k)

    return(visited)
```

BFS from vertex 7

Visited	
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False

To explore queue									

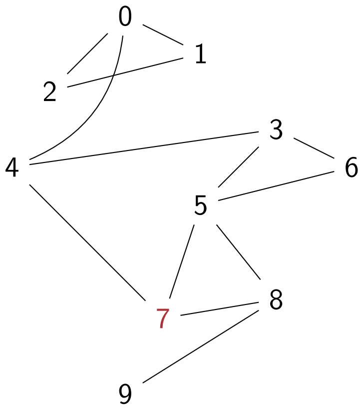


BFS from vertex 7

Visited	
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	True
8	False
9	False

To explore queue									
7									

- Mark 7 and add to queue

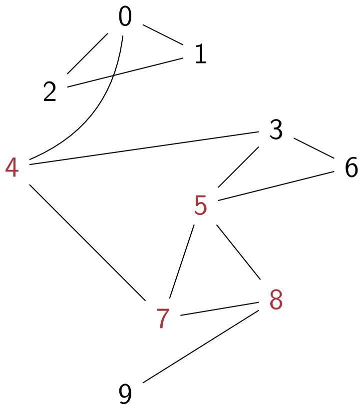


BFS from vertex 7

Visited	
0	False
1	False
2	False
3	False
4	True
5	True
6	False
7	True
8	True
9	False

To explore queue									
4	5	8							

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}

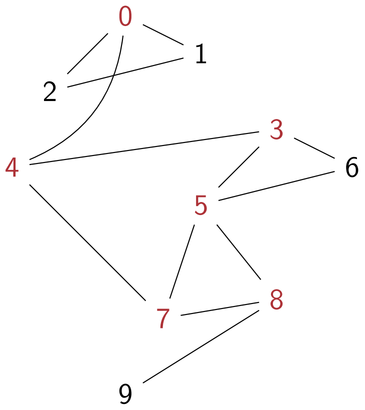


BFS from vertex 7

Visited	
0	True
1	False
2	False
3	True
4	True
5	True
6	False
7	True
8	True
9	False

To explore queue									
5	8	0	3						

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}

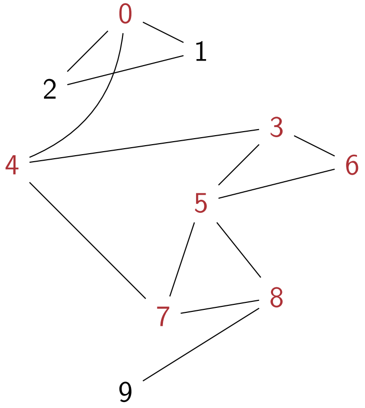


BFS from vertex 7

Visited	
0	True
1	False
2	False
3	True
4	True
5	True
6	True
7	True
8	True
9	False

To explore queue									
8	0	3	6						

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}

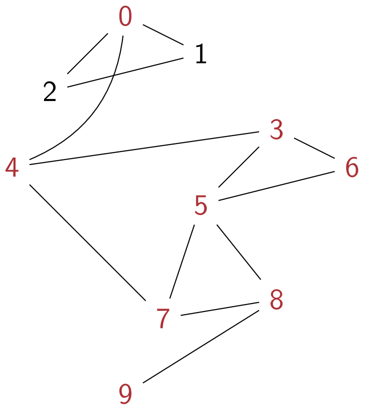


BFS from vertex 7

Visited	
0	True
1	False
2	False
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue									
0	3	6	9						

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}

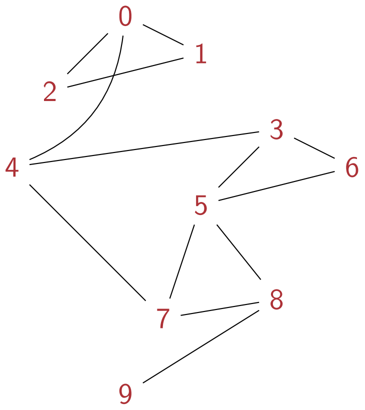


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue									
3	6	9	1	2					

- Mark 7 and add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$

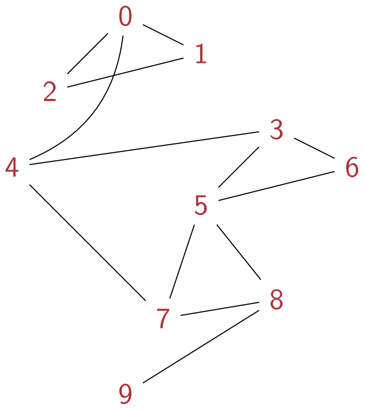


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue									
6	9	1	2						

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3

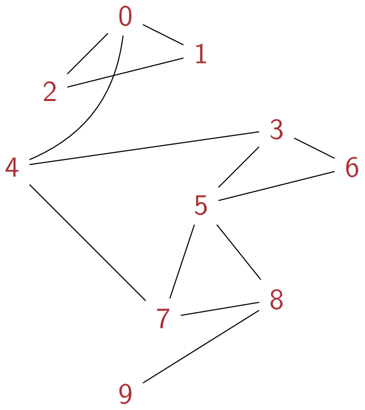


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue										
9	1	2								

- Mark 7 and add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6

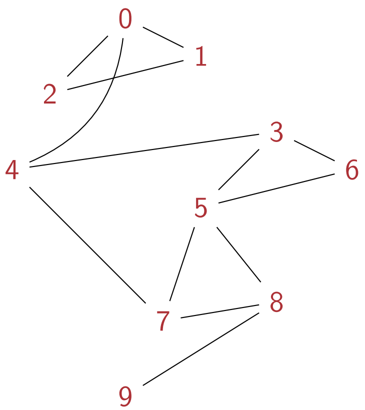


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue										
1	2									

- Mark 7 and add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9

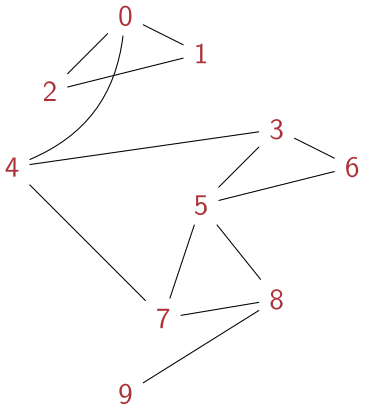


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue									
2									

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1

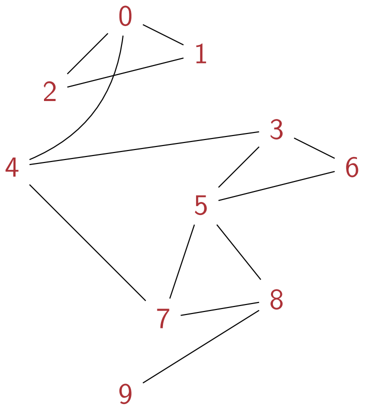


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue									

- Mark 7 and add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Complexity of BFS

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

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 and explore at most n vertices

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 and explore at most n vertices
- Exploring a vertex
  - Check all outgoing edges
  - How long does this take?

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 and explore at most n vertices
- Exploring a vertex
  - Check all outgoing edges
  - How long does this take?

Adjacency matrix

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

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 and explore at most n vertices
- Exploring a vertex
  - Check all outgoing edges
  - How long does this 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 varies across vertices

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 and explore at most n vertices
- Exploring a vertex
  - Check all outgoing edges
  - How long does this 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 varies across vertices

Sum of degrees

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

Complexity of BFS

BFS with adjacency matrix

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

Complexity of BFS

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 $O(n+m)$

Complexity of BFS

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
  - This is why we treat m and n as separate parameters

Complexity of BFS

BFS with adjacency matrix

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

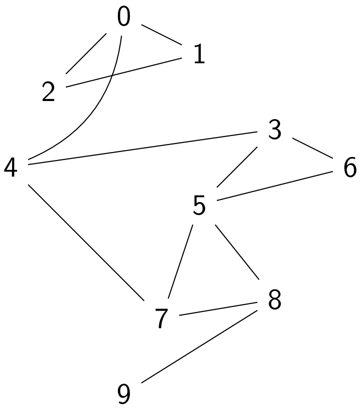
BFS with adjacency list

- 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
  - This 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 each vertex and edge at least once
  - Linear time

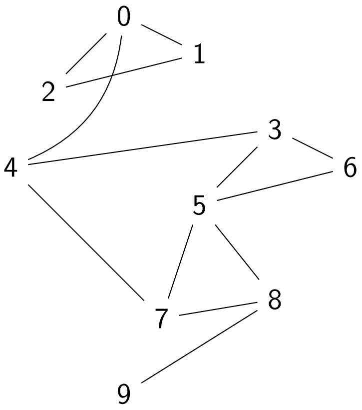
Enhancing BFS to record paths

- 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?



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



Enhancing BFS to record paths

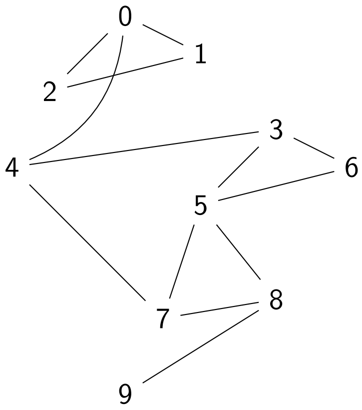
- If BFS from i sets $\text{visited}(k) = \text{True}$, we know that k is reachable from i
- How do we recover a path from i to k?
- $\text{visited}(k)$ was set to True when exploring some vertex j
- Record $\text{parent}(k) = j$
- From k, follow parent links to trace back a path to i

```
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)
```

BFS from vertex 7 with parent information

	Visited	Parent
0	False	-1
1	False	-1
2	False	-1
3	False	-1
4	False	-1
5	False	-1
6	False	-1
7	False	-1
8	False	-1
9	False	-1

To explore queue									

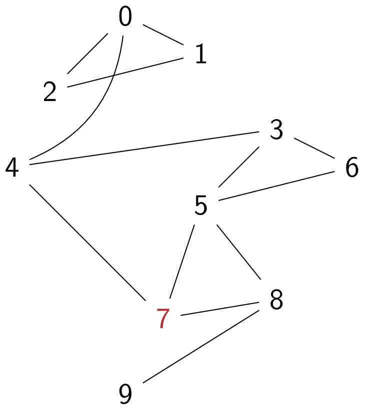


BFS from vertex 7 with parent information

	Visited	Parent
0	False	−1
1	False	−1
2	False	−1
3	False	−1
4	False	−1
5	False	−1
6	False	−1
7	True	−1
8	False	−1
9	False	−1

To explore queue									
7									

- Mark 7, add to queue

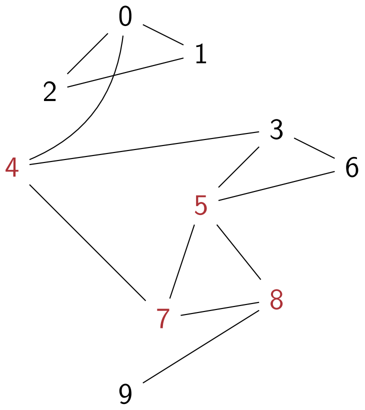


BFS from vertex 7 with parent information

	Visited	Parent
0	False	-1
1	False	-1
2	False	-1
3	False	-1
4	True	7
5	True	7
6	False	-1
7	True	-1
8	True	7
9	False	-1

To explore queue										
4	5	8								

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$

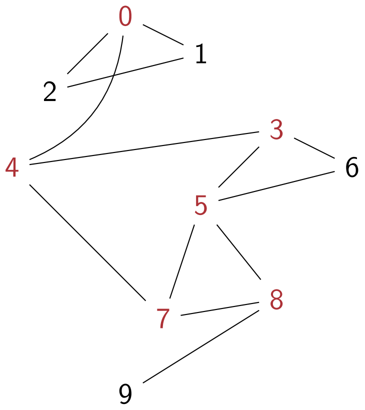


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	False	-1
2	False	-1
3	True	4
4	True	7
5	True	7
6	False	-1
7	True	-1
8	True	7
9	False	-1

To explore queue									
5	8	0	3						

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$

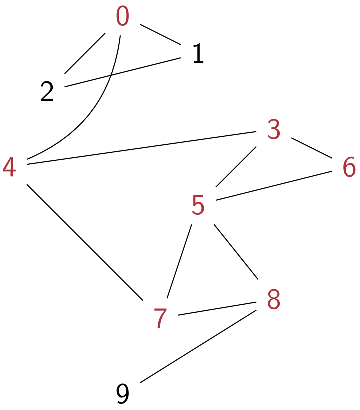


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	False	-1
2	False	-1
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	False	-1

To explore queue									
8	0	3	6						

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$

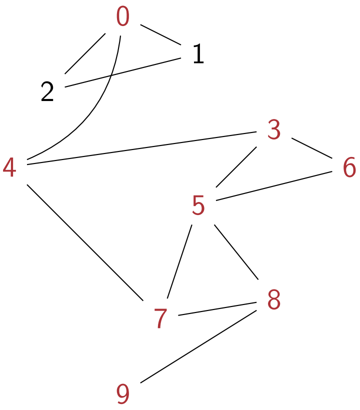


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	False	-1
2	False	-1
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue										
0	3	6	9							

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$

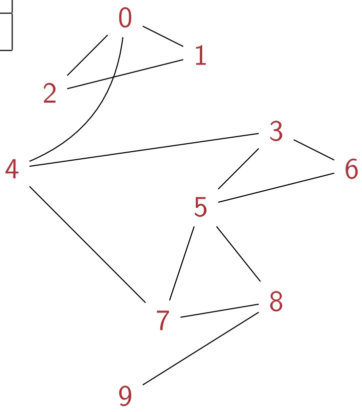


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue									
3	6	9	1	2					

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}

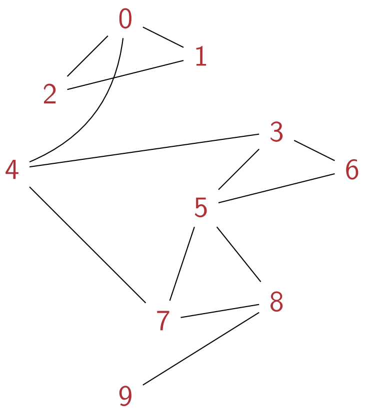


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue										
6	9	1	2							

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3

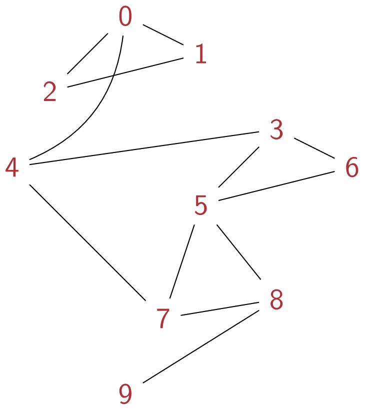


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue										
9	1	2								

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6

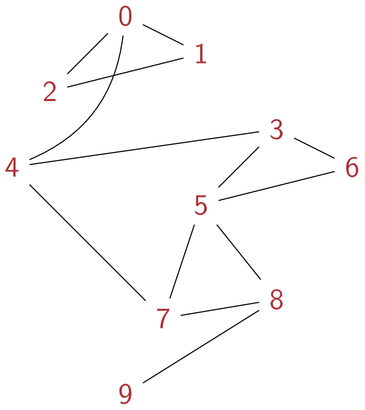


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue										
1	2									

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9

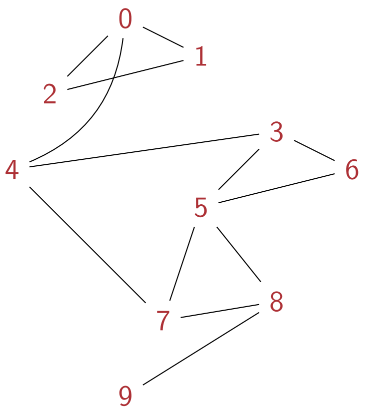


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue									
2									

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9
- Explore 1

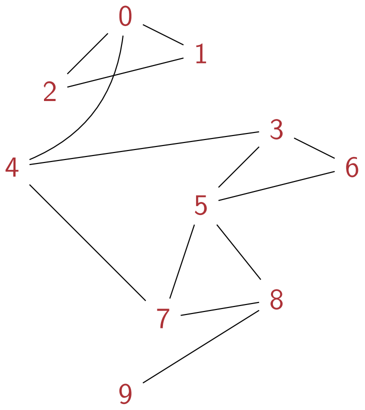


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue									

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



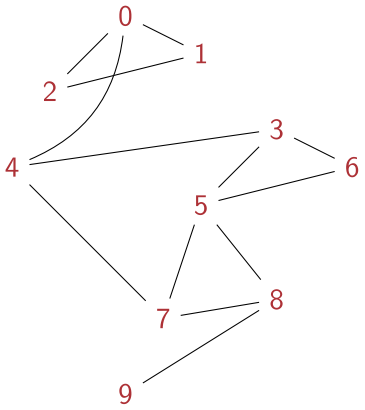
BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

Path from 7 to 6 is
7–5–6

To explore queue									

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



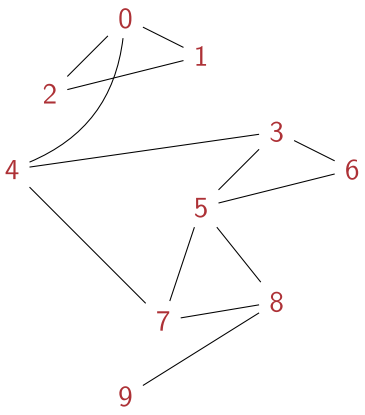
BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

Path from 7 to 2 is
7–4–0–2

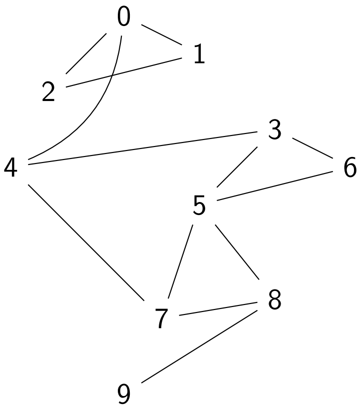
To explore queue									

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



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



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)

```
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.isempty()):
        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)
```

Enhancing BFS to record distance

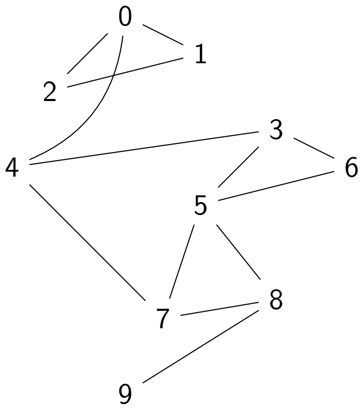
- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex
- Instead of visited(j), maintain level$)(j)$
- Initalize level$(j) = -1$ for all j
- Set level$(i) = 0$ for source vertex
- If we visit k from j, set level(k) to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

```
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.isempty()):
        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)
```

BFS from vertex 7 with parent and distance information

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

To explore queue									

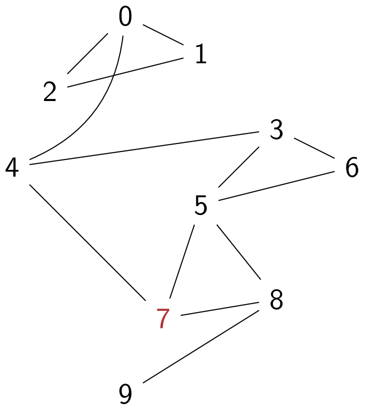


BFS from vertex 7 with parent and distance information

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

To explore queue										
7										

- Mark 7, add to queue

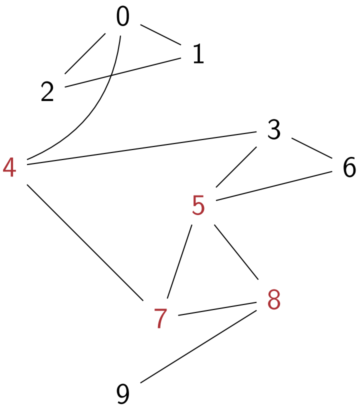


BFS from vertex 7 with parent and distance information

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

To explore queue										
4	5	8								

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$

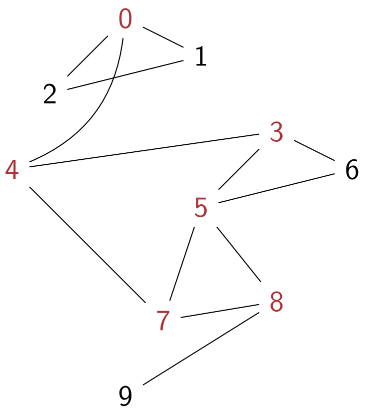


BFS from vertex 7 with parent and distance information

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

To explore queue									
5	8	0	3						

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$

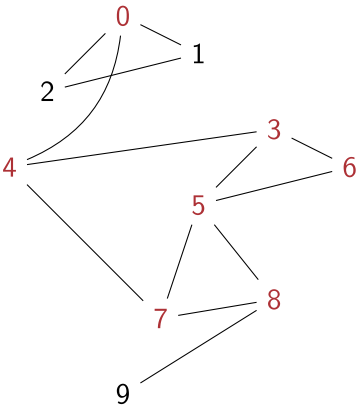


BFS from vertex 7 with parent and distance information

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

To explore queue									
8	0	3	6						

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$

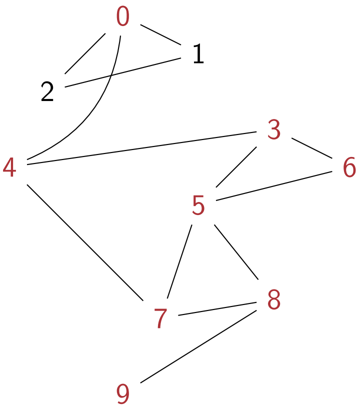


BFS from vertex 7 with parent and distance information

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

To explore queue									
0	3	6	9						

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$

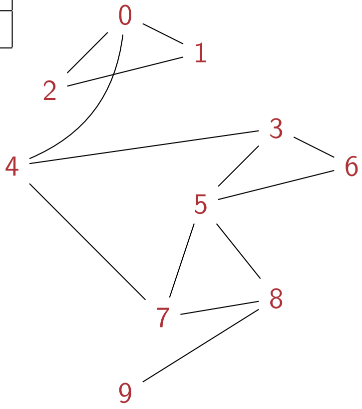


BFS from vertex 7 with parent and distance information

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

To explore queue									
3	6	9	1	2					

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$

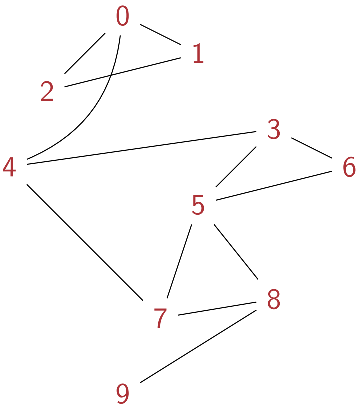


BFS from vertex 7 with parent and distance information

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

To explore queue									
6	9	1	2						

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3

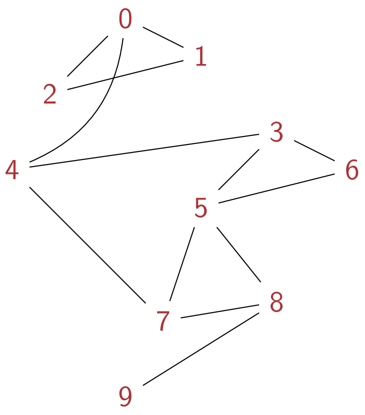


BFS from vertex 7 with parent and distance information

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

To explore queue										
9	1	2								

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6

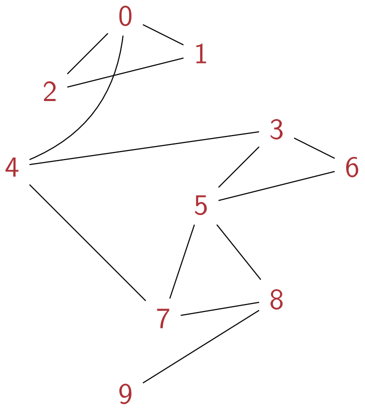


BFS from vertex 7 with parent and distance information

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

To explore queue										
1	2									

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9

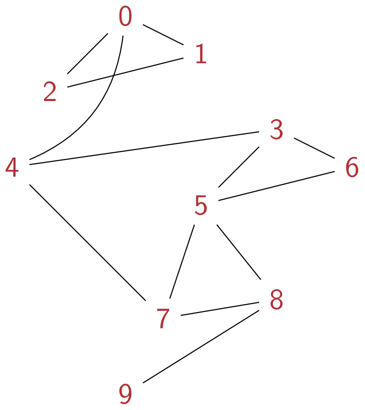


BFS from vertex 7 with parent and distance information

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

To explore queue									
2									

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9
- Explore 1

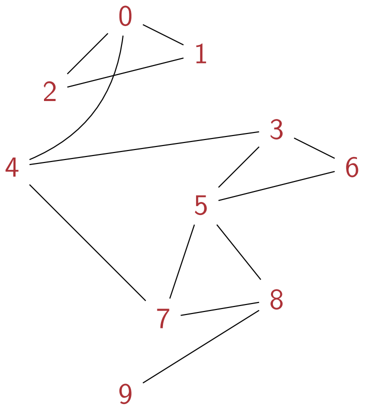


BFS from vertex 7 with parent and distance information

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

To explore queue									

- Mark 7, add to queue
- Explore 7, visit $\{4,5,8\}$
- Explore 4, visit $\{0,3\}$
- Explore 5, visit $\{6\}$
- Explore 8, visit $\{9\}$
- Explore 0, visit $\{1,2\}$
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Summary

- Breadth first search is a systematic strategy to explore a graph, level by level

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited

Summary

- Breadth first search 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

Summary

- Breadth first search 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

Summary

- Breadth first search 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

Summary

- Breadth first search 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 number of edges

Summary

- Breadth first search 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 number of edges
  - In general, edges are labelled with a cost (distance, time, ticket price, ...)

Summary

- Breadth first search 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 number 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 number of edges

Depth First Search

Madhavan Mukund

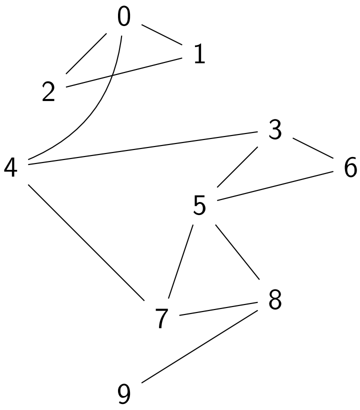
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

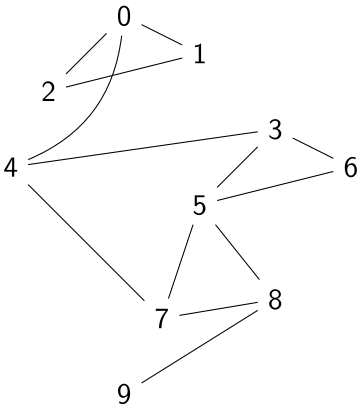
Depth first search (DFS)

- Start from i, visit an unexplored neighbour j



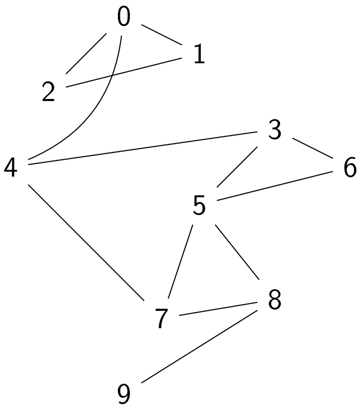
Depth first search (DFS)

- Start from i, visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead



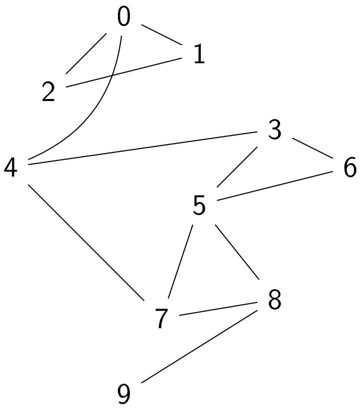
Depth first search (DFS)

- Start from i, visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours



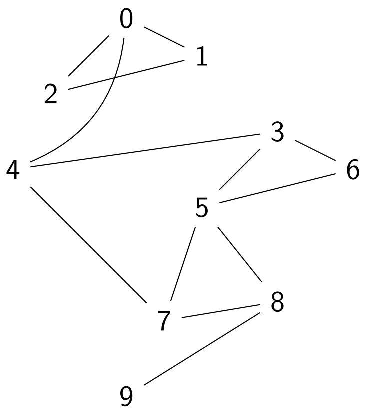
Depth first search (DFS)

- Start from i, visit an unexplored neighbour j
- 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



Depth first search (DFS)

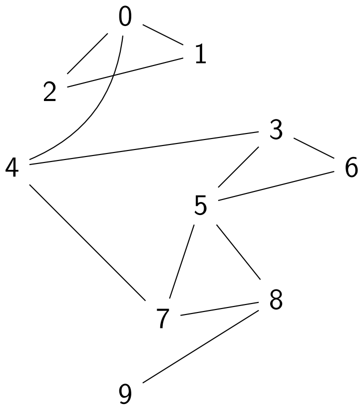
- Start from i, visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a stack
    - Last in,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



DFS from vertex 4

Visited	
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									

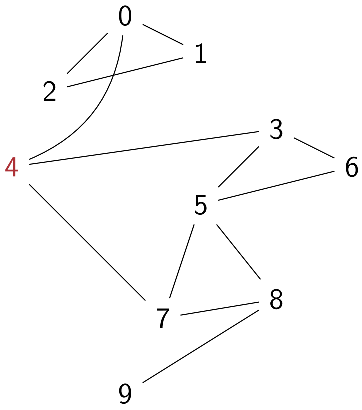


DFS from vertex 4

Visited	
0	False
1	False
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									

- Mark 4,

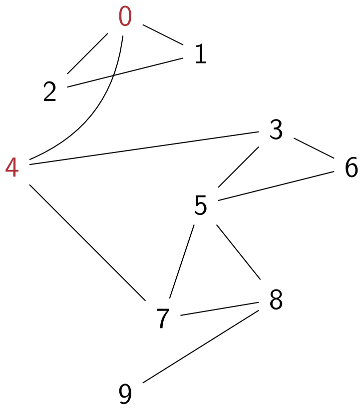


DFS from vertex 4

Visited	
0	True
1	False
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4									

- Mark 4, Suspend 4, explore 0

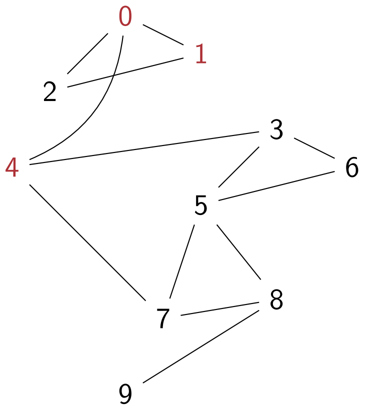


DFS from vertex 4

Visited	
0	True
1	True
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4	0								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1

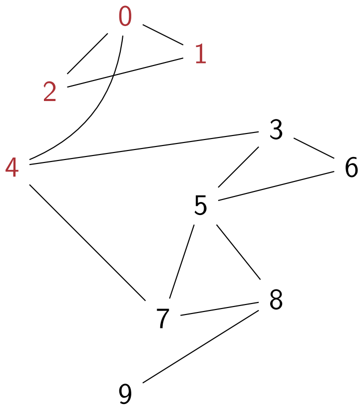


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4	0	1							

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2

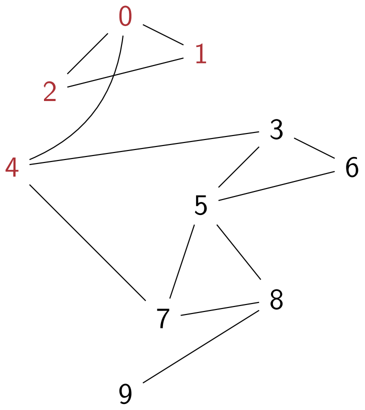


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4	0							

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1,

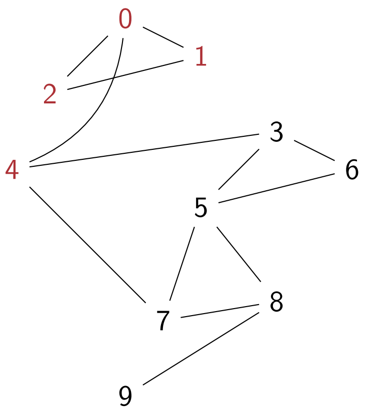


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4									

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0,

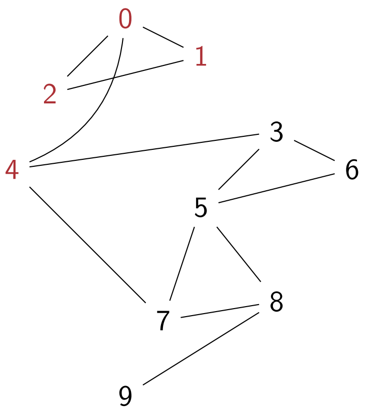


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4

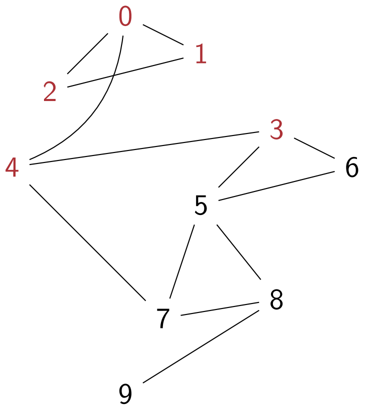


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4									

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3

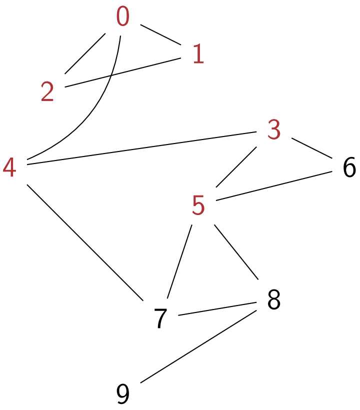


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	False
7	False
8	False
9	False

Stack of suspended vertices									
4	3								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5

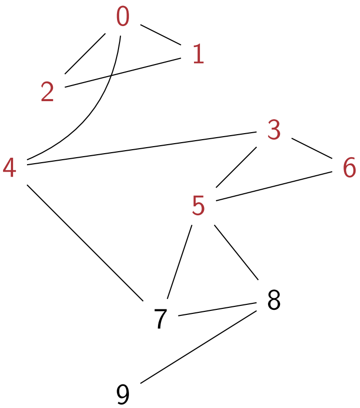


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	False
8	False
9	False

Stack of suspended vertices									
4	3	5							

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6

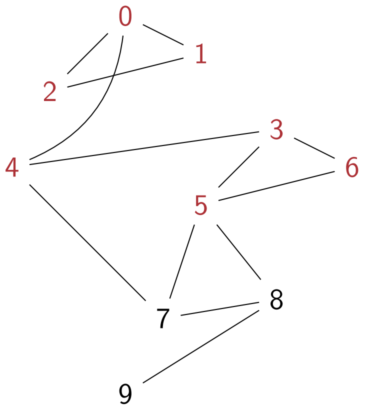


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	False
8	False
9	False

Stack of suspended vertices									
4	3								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5,

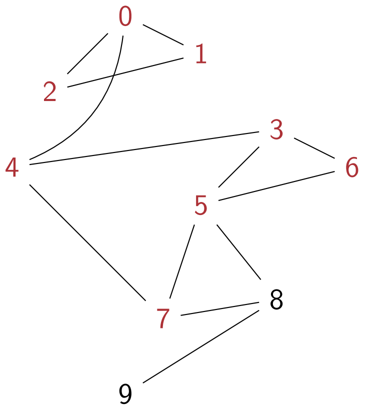


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	False
9	False

Stack of suspended vertices										
4	3	5								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7

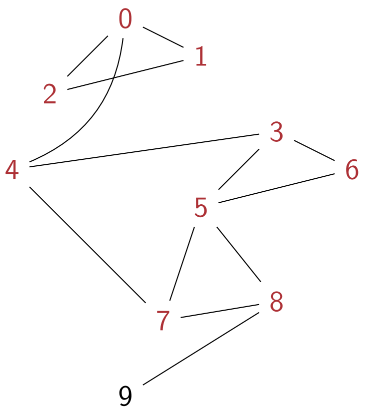


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	False

Stack of suspended vertices									
4	3	5	7						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8

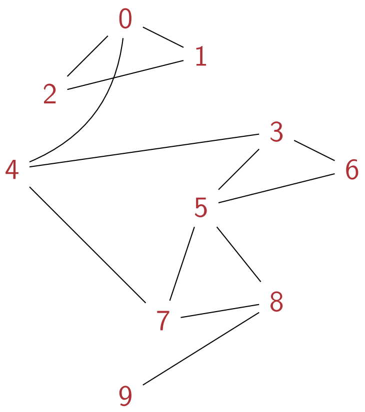


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									
4	3	5	7	8					

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9

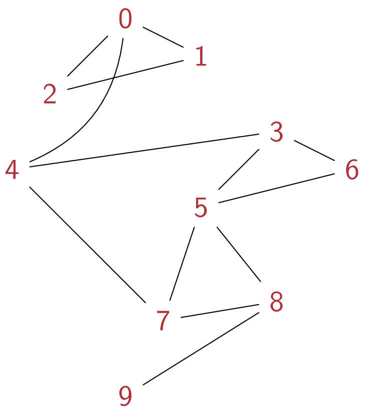


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									
4	3	5	7						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8,

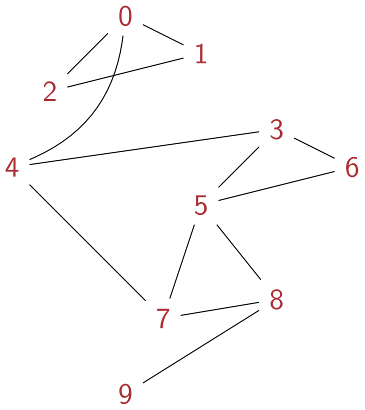


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									
4	3	5							

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7,

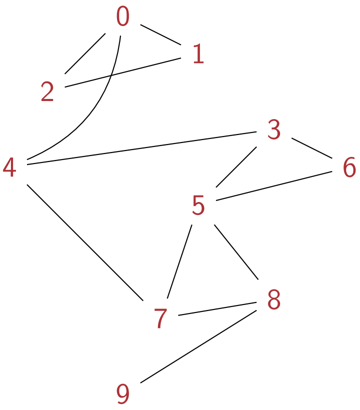


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									
4	3								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5,

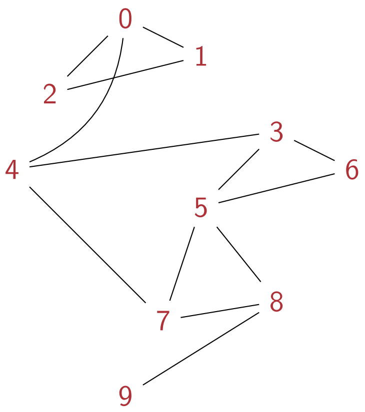


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									
4									

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5, 3,

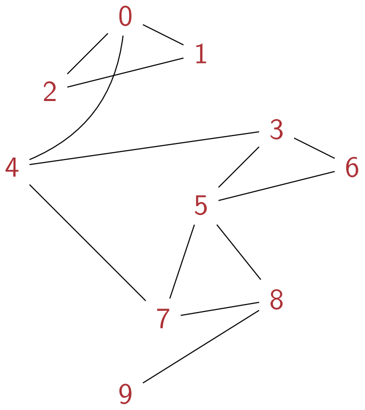


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices									

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5, 3, 4



Implementing DFS

- DFS is most natural to implement recursively
  - For each unvisited neighbour of v, call *DFS*(v)

```
def DFSInit(AMat):
  # Initialization
  (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)
```

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 implicilty maintains stack
  - Separate initialization step

```
def DFSInit(AMat):
  # Initialization
  (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)
```

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 implicilty 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

```
(visited,parent) = ({},{})

def DFSInitGlobal(AMat):
  # Initialization
  (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
```

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 implicilty 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
- Use an adjacency list instead

```
def DFSInitList(AList):
  # Initialization
  (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)
```

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 implicilty 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
- Use an adjacency list instead

```
(visited,parent) = ({},{})

def DFSInitListGlobal(AList):
  # Initialization
  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 and explored once

Complexity of DFS

- Like BFS, each vertex is marked and 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

Complexity of DFS

- Like BFS, each vertex is marked and 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
  - $O(m+n)$ using adjacency list

Summary

- DFS is another systematic strategy to explore a graph

Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours

Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours
- Paths discovered by DFS are not shortest paths, unlike BFS

Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and 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

Madhavan Mukund

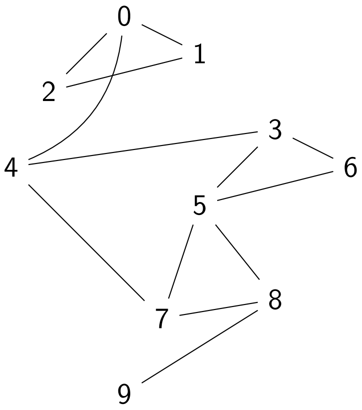
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

BFS and DFS

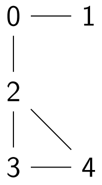
- BFS and DFS systematically compute reachability in graphs
- BFS works level by level
  - Discovers shortest paths in terms of number of edges
- DFS explores a vertex as soon as it is visited neighbours
  - Suspend a vertex while exploring its neighbours
  - DFS numbering describes the order in which vertices are explored
- Beyond reachability, what can we find out about a graph using BFS/DFS?



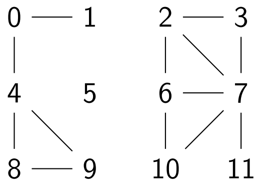
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex

Connected Graph



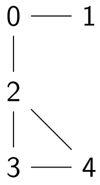
Disconnected Graph



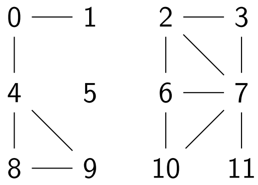
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components

Connected Graph



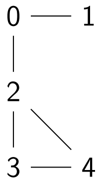
Disconnected Graph



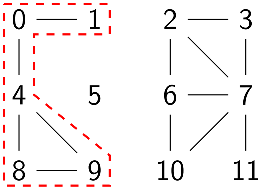
Connectivity

- 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

Connected Graph



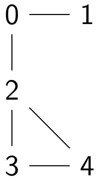
Disconnected Graph



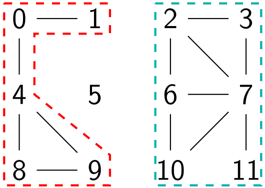
Connectivity

- 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

Connected Graph



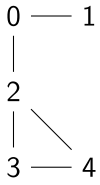
Disconnected Graph



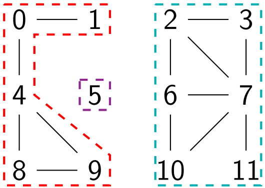
Connectivity

- 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



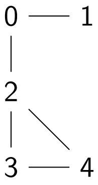
Disconnected Graph



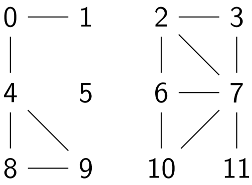
Identifying connected components

- Assign each vertex a component number

Connected Graph



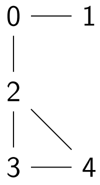
Disconnected Graph



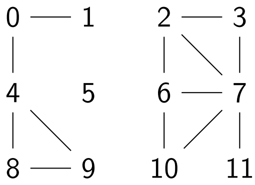
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0

Connected Graph



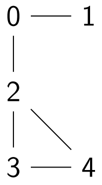
Disconnected Graph



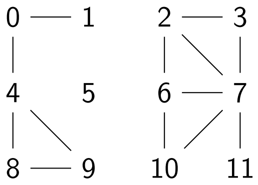
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0

Connected Graph



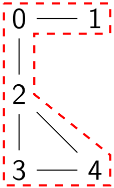
Disconnected Graph



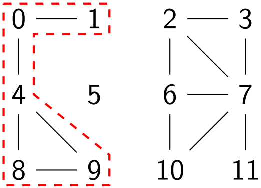
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component

Connected Graph



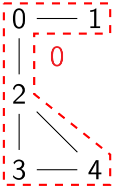
Disconnected Graph



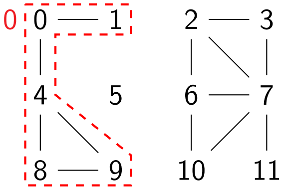
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0

Connected Graph



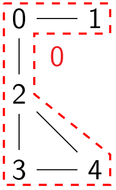
Disconnected Graph



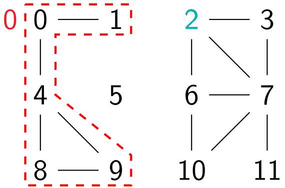
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j

Connected Graph



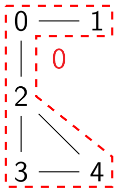
Disconnected Graph



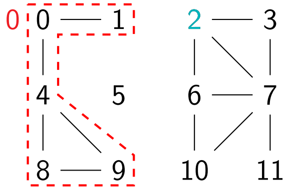
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node *j*
  - Increment component number to 1

Connected Graph



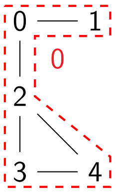
Disconnected Graph



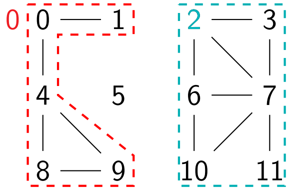
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j
  - Increment component number to 1
  - Run BFS/DFS from node j

Connected Graph



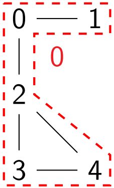
Disconnected Graph



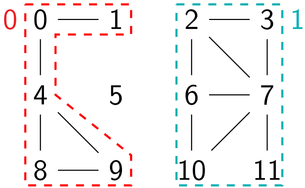
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node *j*
  - Increment component number to 1
  - Run BFS/DFS from node *j*
  - Assign each visited node component number 1

Connected Graph

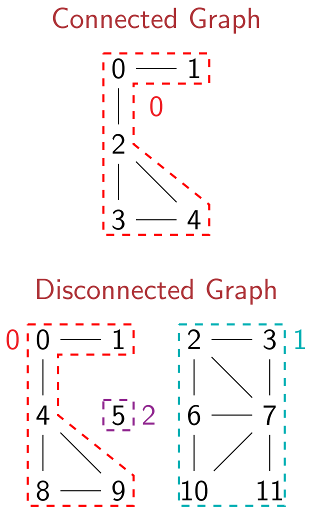


Disconnected Graph



Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 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

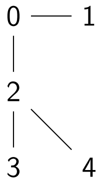


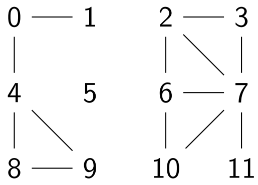
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 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

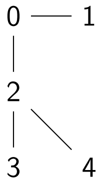
```
def Components(AList):
  component = {}
  for i in AList.keys():
    component[i] = -1

  (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)
```

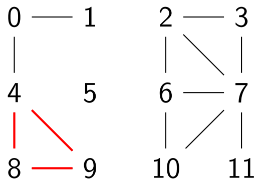
- A cycle is a path (technically, a walk) that starts and ends at the same vertex





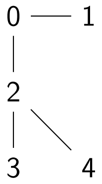
- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - 4 — 8 — 9 — 4 is a cycle

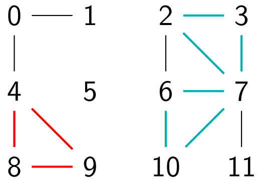




Detecting cycles

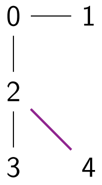
- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - $4 - 8 - 9 - 4$ is a cycle
  - Cycle may repeat a vertex: $2 - 3 - 7 - 10 - 6 - 7 - 2$

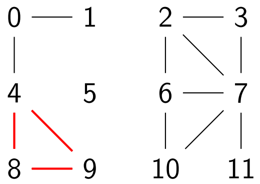




Detecting cycles

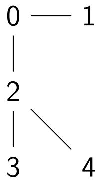
- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - 4 — 8 — 9 — 4 is a cycle
  - Cycle may repeat a vertex: 2 — 3 — 7 — 10 — 6 — 7 — 2
  - Cycle should not repeat edges: $i - j - i$ is not a cycle, e.g., 2 — 4 — 2

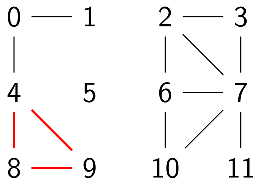




Detecting cycles

- 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., 2 — 4 — 2
  - Simple cycle — only repeated vertices are start and end

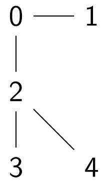




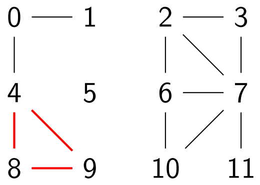
Detecting cycles

- 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., $2 - 4 - 2$
  - Simple cycle — only repeated vertices are start and end
- A graph is acyclic if it has no cycles

Acyclic Graph



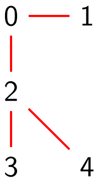
Graph with cycles



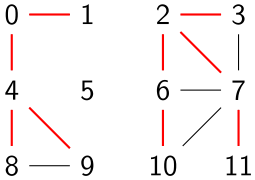
BFS tree

- Edges explored by BFS form a tree
  - Technically, one tree per component
  - Collection of trees is a forest

Acyclic Graph



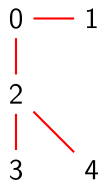
Graph with cycles



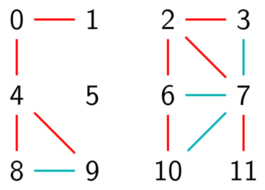
BFS tree

- Edges explored by BFS form a tree
  - Technically, one tree per component
  - Collection of trees is a forest
- Any non-tree edge creates a cycle
  - Detect cycles by searching for non-tree edges

Acyclic Graph

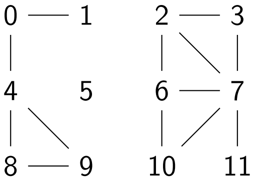


Graph with cycles



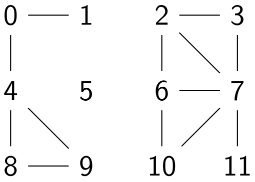
DFS tree

- Maintain a DFS counter, initially 0



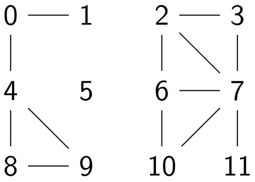
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node



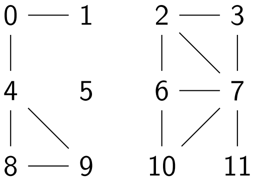
DFS tree

- 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)



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)

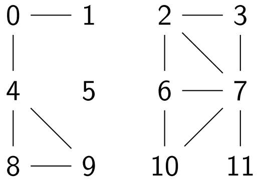


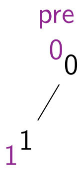
pre

0_0

DFS tree

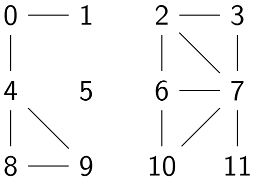
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

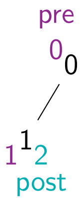




DFS tree

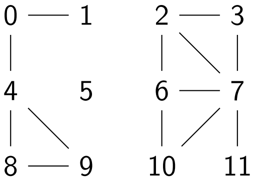
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

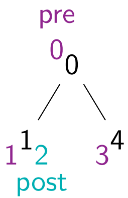




DFS tree

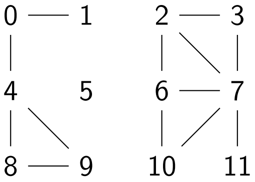
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

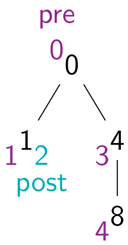




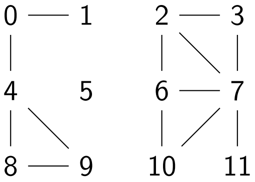
DFS tree

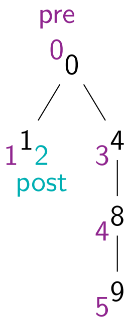
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)



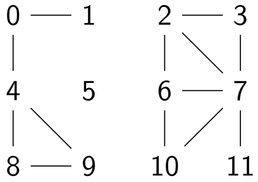


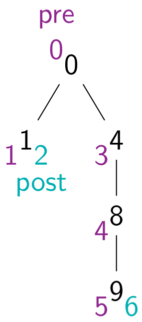
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)





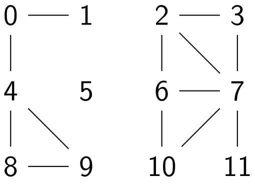
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

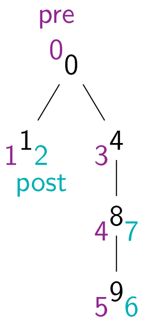




DFS tree

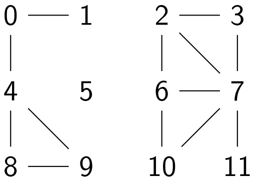
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

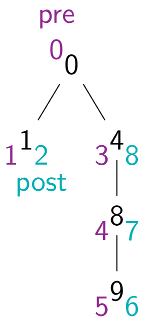




DFS tree

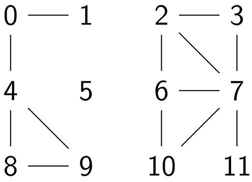
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

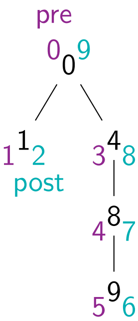




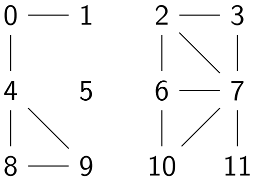
DFS tree

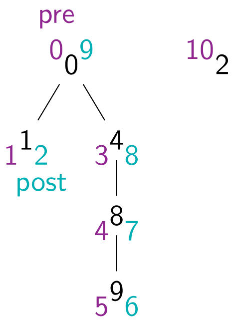
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)





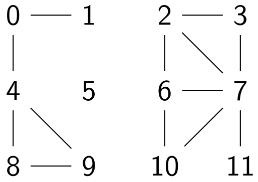
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

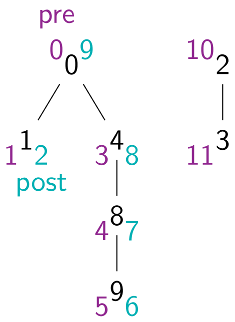




DFS tree

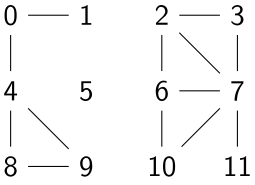
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

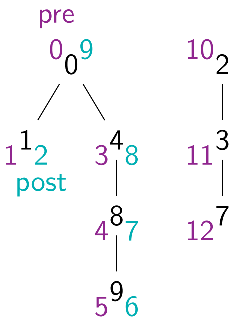




DFS tree

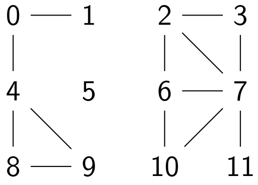
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

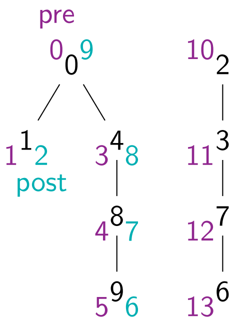




DFS tree

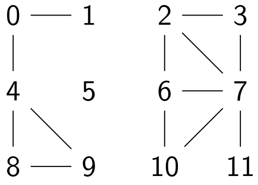
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

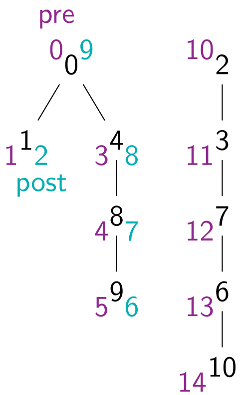




DFS tree

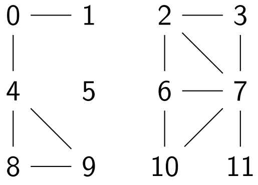
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

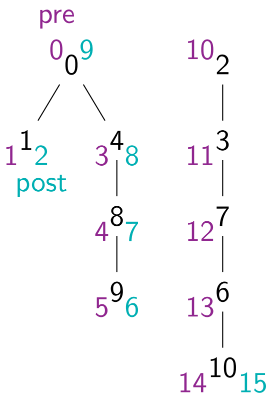




DFS tree

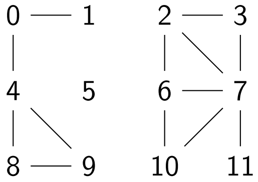
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

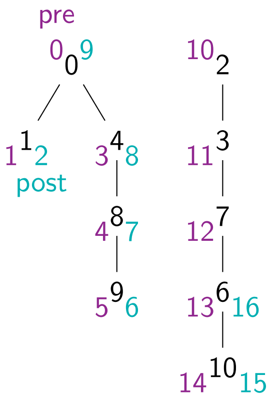




DFS tree

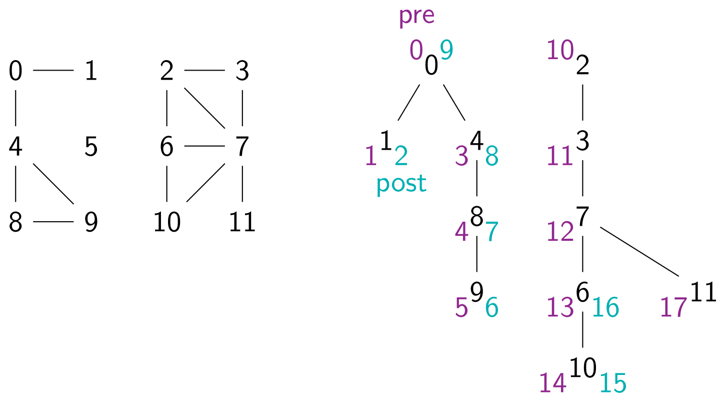
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)





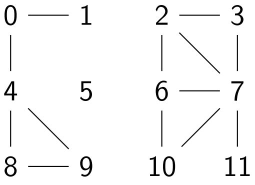
DFS tree

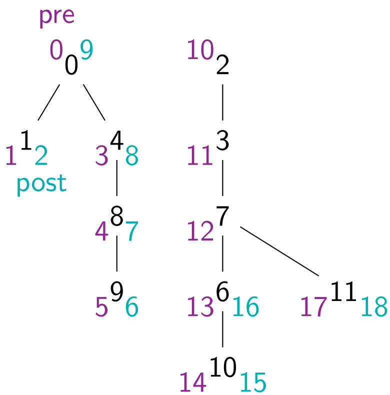
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)



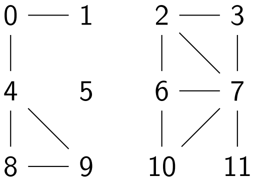
DFS tree

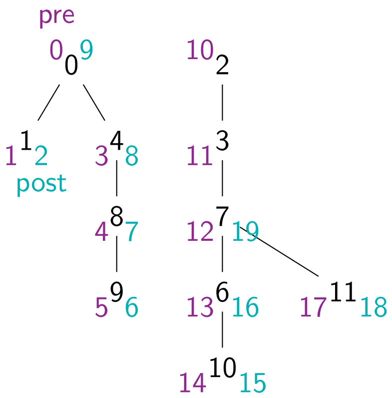
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)





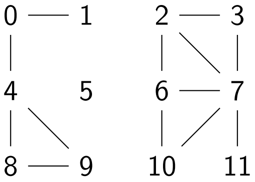
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

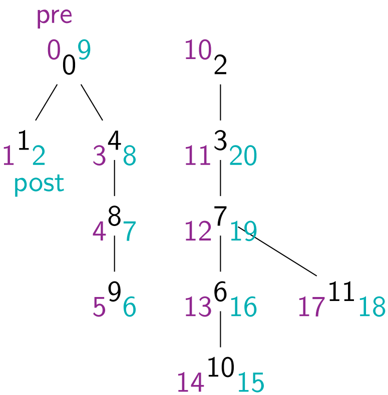




DFS tree

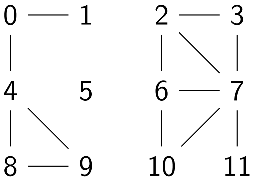
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

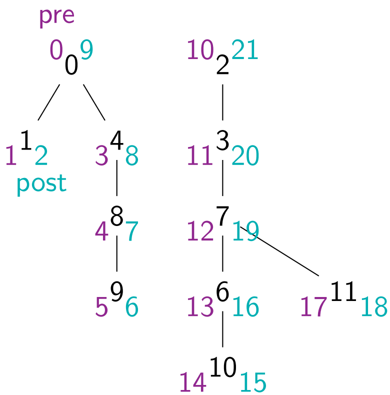




DFS tree

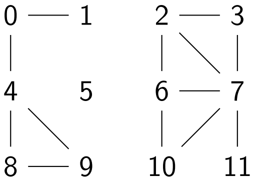
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

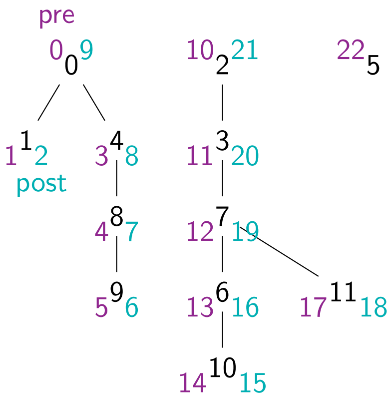




DFS tree

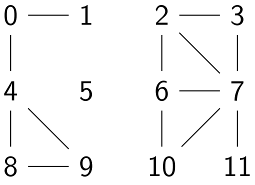
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

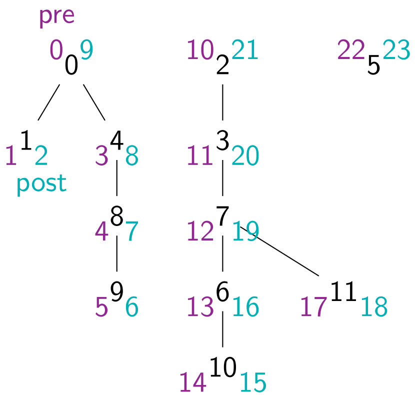




DFS tree

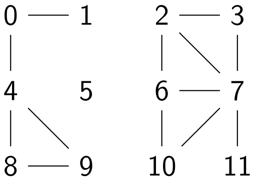
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)



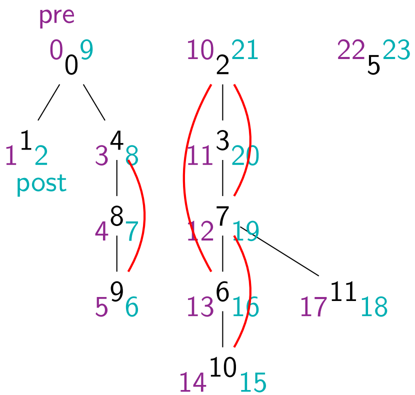


DFS tree

- 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)

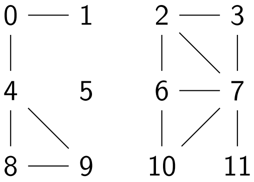


- As before, non-tree edges generate cycles



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)

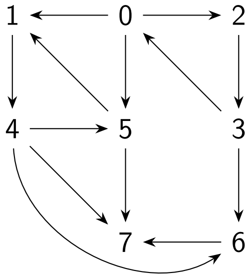


- As before, non-tree edges generate cycles
- To compute pre and post pass counter via recursive DFS calls

```
(visited,pre,post) = ({},{},{})

def DFSInitPrePost(AList):
  # Initialization
  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)
```

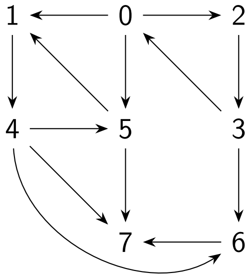
Directed cycles

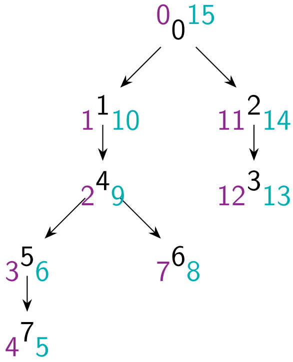
- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not



Directed cycles

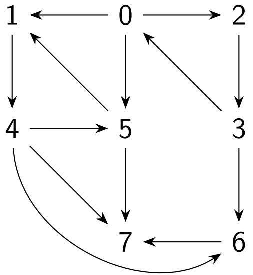
- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not

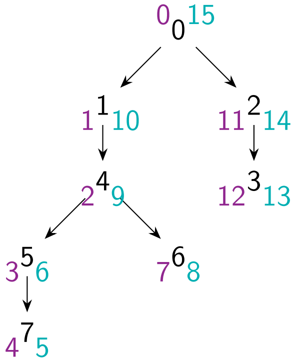




Directed cycles

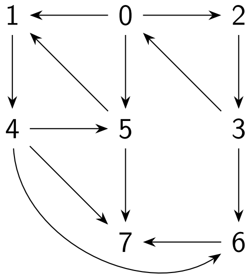
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges

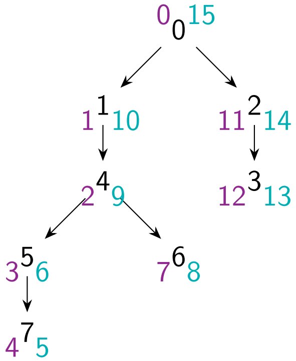




Directed cycles

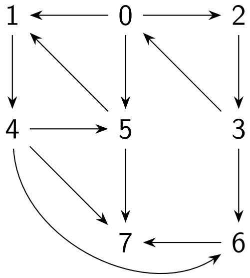
- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges

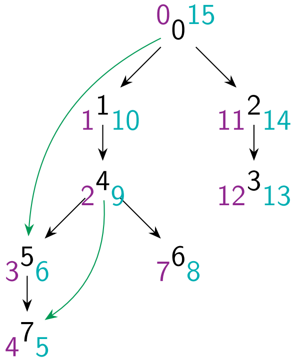




Directed cycles

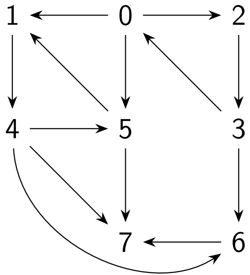
- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges

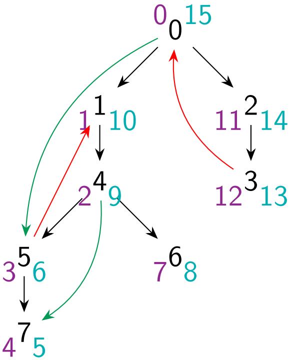




Directed cycles

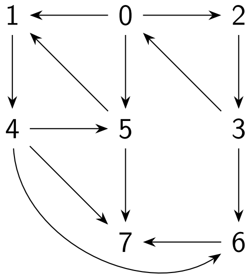
- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges
  - Back edges

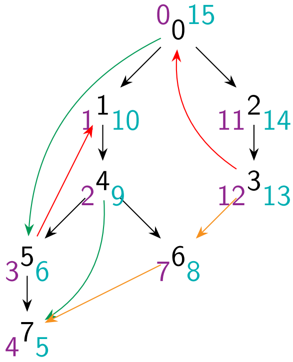




Directed cycles

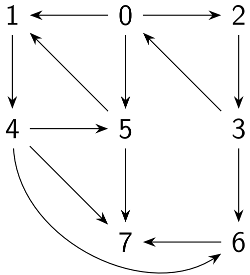
- In a directed graph, a cycle must follow same direction
    - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
    - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
    - Forward edges
    - Back edges
    - Cross edges

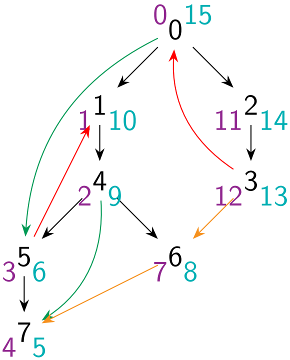




Directed cycles

- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges
  - Back edges
  - Cross edges

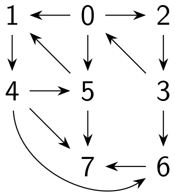


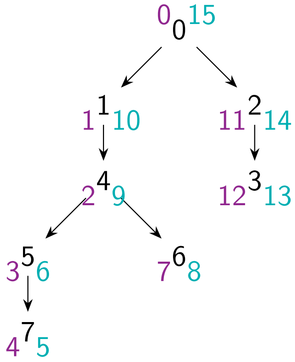


- Only back edges correspond to cycles

Classifying non-tree edges in directed graphs

- Use pre/post numbers

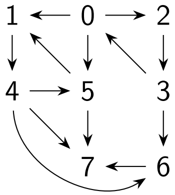


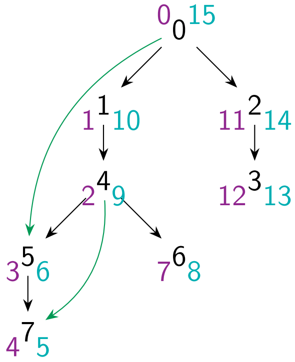


Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)

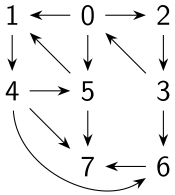
  Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$

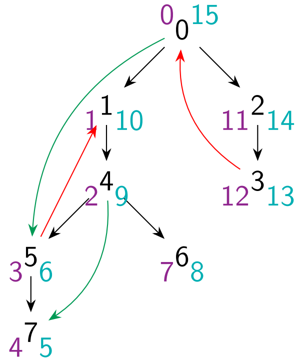




Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)
  Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$
- Back edge (u, v)
  Interval $[\text{pre}(v), \text{post}(v)]$ contains $[\text{pre}(u), \text{post}(u)]$





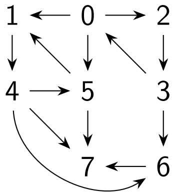
Classifying non-tree edges in directed graphs

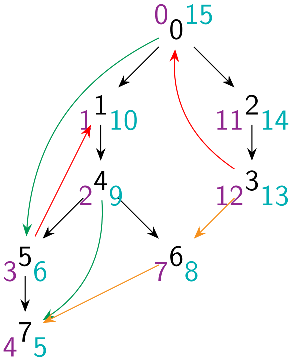
- Use pre/post numbers
- Tree edge/forward edge (u, v)

  Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$
- Back edge (u, v)

  Interval $[\text{pre}(v), \text{post}(v)]$ contains $[\text{pre}(u), \text{post}(u)]$
- Cross edge (u, v)

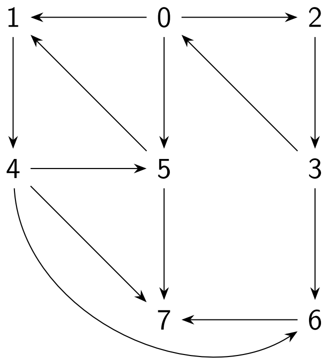
  Intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint





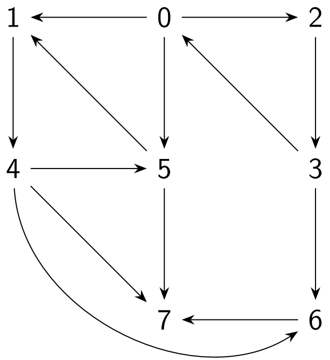
Connectivity in directed graphs

- Take directions into account



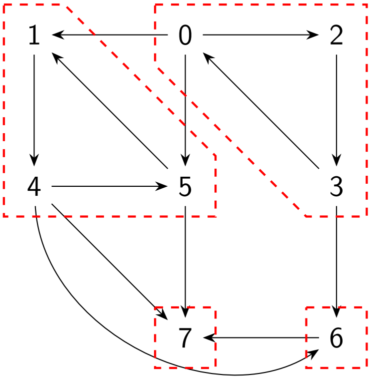
Connectivity in directed graphs

- 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



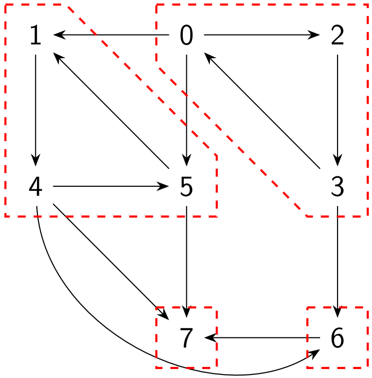
Connectivity in directed graphs

- 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



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 and DFS identify an underlying tree, non-tree edges generate cycles

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate 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

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate 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

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate 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) and bridges (cut edges)

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate 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) and bridges (cut edges)
- Directed acyclic graphs are useful for representing dependencies
  - Given course prerequisites, find a valid sequence to complete a programme

Directed Acyclic Graphs (DAGs)

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

Tasks and dependencies

- Startup moving into new office space
- Major tasks for completing the interiors
  - Lay floor tiles
  - Plaster the walls
  - Paint the walls
  - Lay conduits (pipes) for electrical wires
  - Do electrical wiring
  - Install electrical fittings
  - Lay telecom conduits
  - Do phone and network cabling

Tasks and dependencies

- Startup moving into new office space
- Major tasks for completing the interiors
  - Lay floor tiles
  - Plaster the walls
  - Paint the walls
  - Lay conduits (pipes) for electrical wires
  - Do electrical wiring
  - Install electrical fittings
  - Lay telecom conduits
  - Do phone and network cabling
- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings

Tasks and dependencies

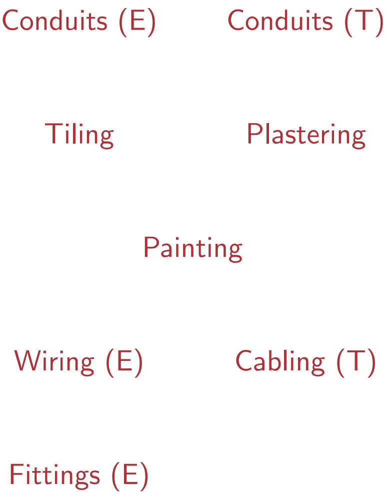
- Startup moving into new office space
- Major tasks for completing the interiors
  - Lay floor tiles
  - Plaster the walls
  - Paint the walls
  - Lay conduits (pipes) for electrical wires
  - Do electrical wiring
  - Install electrical fittings
  - Lay telecom conduits
  - Do phone and network cabling
- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u

Tasks and dependencies

- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u

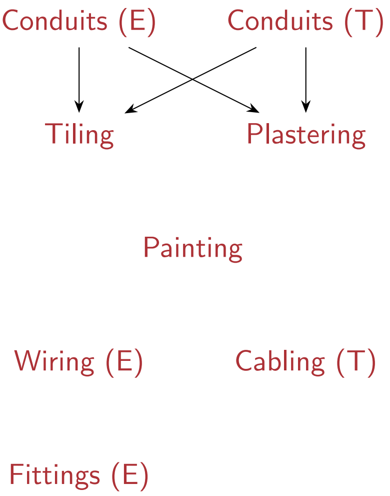
Tasks and dependencies

- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u



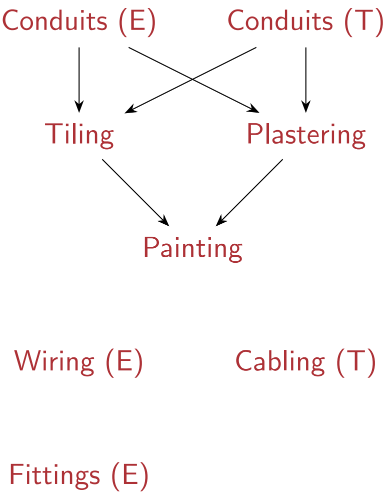
Tasks and dependencies

- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u



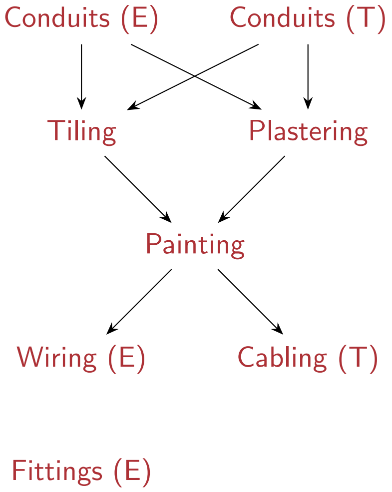
Tasks and dependencies

- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u



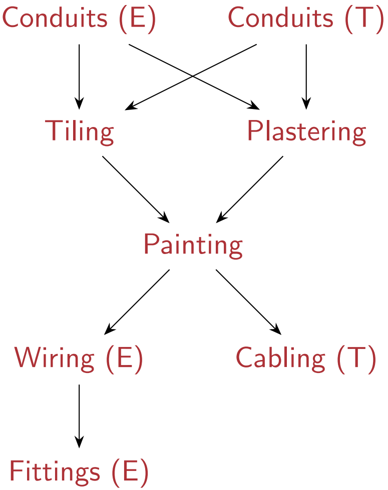
Tasks and dependencies

- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u



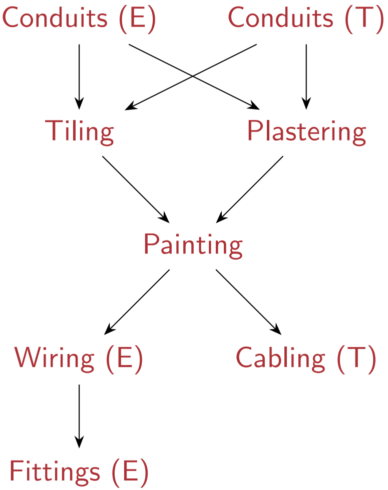
Tasks and dependencies

- Constraints on the sequence
  - Lay conduits before tiles and plastering
  - Lay tiles, plaster wall before painting
  - Finish painting before any cabling/wiring work
  - Electrical wiring before installing fittings
- Represent constraints as a directed graph
  - Vertices are tasks
  - Edge (t, u) if task t has to be completed before task u



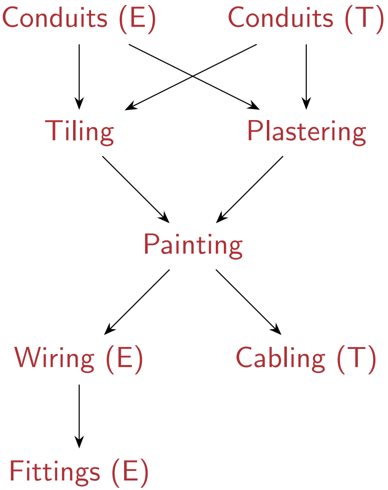
Typical questions

- Schedule the tasks respecting the dependencies



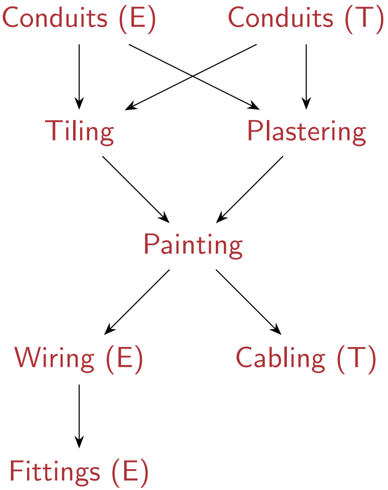
Typical questions

- Schedule the tasks respecting the dependencies
  - Conduits (E) – Conduits (T) – Tiling – Plastering – Painting – Wiring (E) – Cabling (T) – Fittings (E)



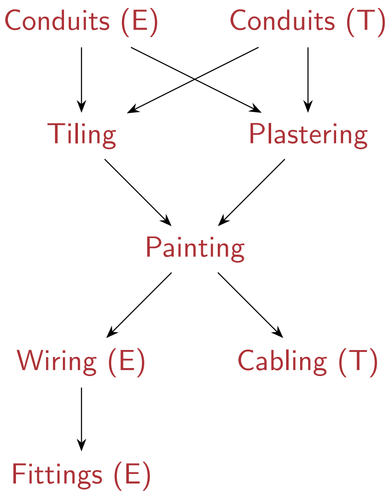
Typical questions

- Schedule the tasks respecting the dependencies
  - Conduits (E) – Conduits (T) – Tiling – Plastering – Painting – Wiring (E) – Cabling (T) – Fittings (E)
  - Conduits (T) – Conduits (E) – Plastering – Tiling – Painting – Wiring (E) – Fittings (E) – Cabling (T)
  - ...



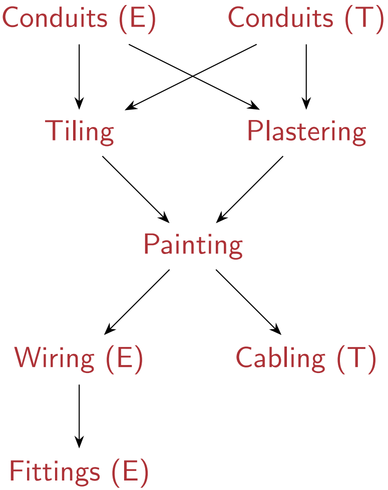
Typical questions

- Schedule the tasks respecting the dependencies
  - Conduits (E) – Conduits (T) – Tiling – Plastering – Painting – Wiring (E) – Cabling (T) – Fittings (E)
  - Conduits (T) – Conduits (E) – Plastering – Tiling – Painting – Wiring (E) – Fittings (E) – Cabling (T)
  - ...
- How long will the work take?



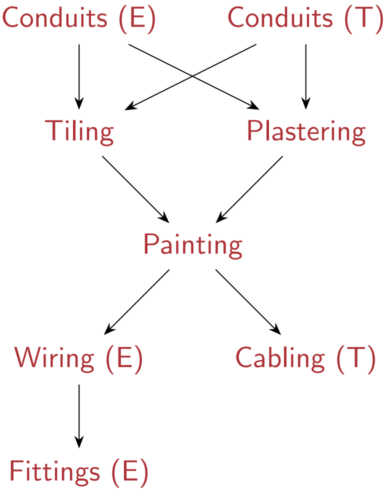
Directed Acyclic Graphs

- Formally, we have a directed acyclic graph (DAG)
- $G = (V, E)$, a directed graph without directed cycles



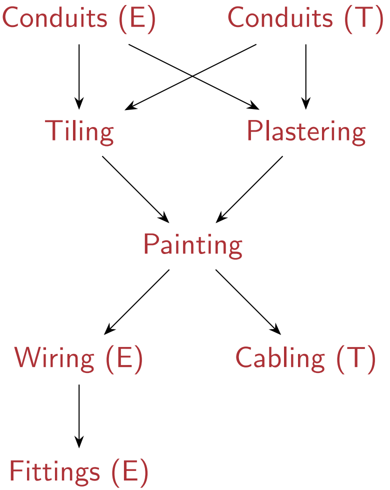
Directed Acyclic Graphs

- Formally, we have a directed acyclic graph (DAG)
- $G = (V, E)$, a directed graph without directed cycles
- Find a schedule
  - Enumerate $V = \{0, 1, \ldots, n-1\}$ such that for any $(i, j) \in E$, i appears before j



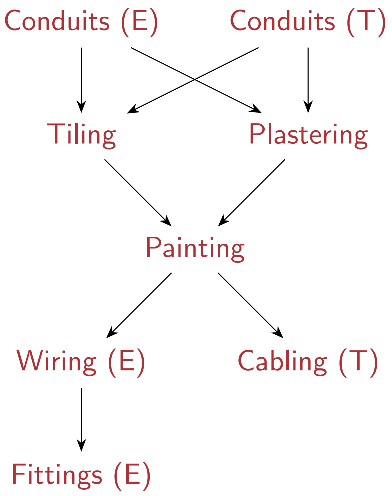
Directed Acyclic Graphs

- Formally, we have a directed acyclic graph (DAG)
- $G = (V, E)$, a directed graph without directed cycles
- Find a schedule
  - Enumerate $V = \{0, 1, \ldots, n-1\}$ such that for any $(i, j) \in E$, i appears before j
  - Topological sorting



Directed Acyclic Graphs

- Formally, we have a directed acyclic graph (DAG)
- $G = (V, E)$, a directed graph without directed cycles
- Find a schedule
  - Enumerate $V = \{0, 1, \ldots, n-1\}$ such that for any $(i, j) \in E$, i appears before j
  - Topological sorting
- How long with the work take?
  - Find the longest path in the DAG



Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Arise in many contexts
  - Pre-requisites between courses for completing a degree
  - Recipe for cooking
  - Construction projects
  - ...
- Problems to be solved on DAGS
  - Topological sorting
  - Longest paths

Topological Sorting

Madhavan Mukund

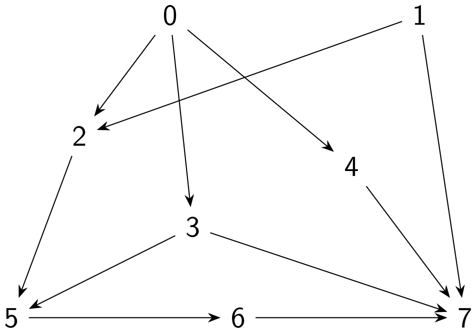
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

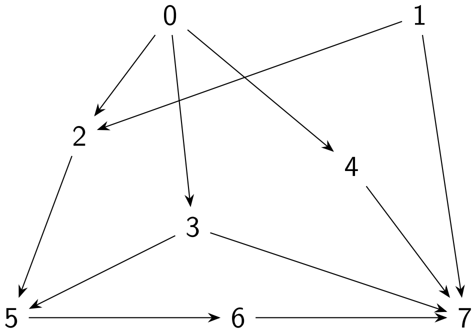
Directed Acyclic Graphs

- $G = (V, E)$, 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
- Represents a feasible schedule



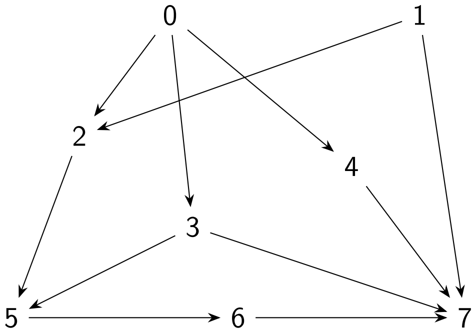
Topological Sort

- A graph with directed cycles cannot be sorted topologically
- Path $i \rightsquigarrow j$ means i must be listed before j
- Cycle $\Rightarrow$ 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!



Topological Sort

- A graph with directed cycles cannot be sorted topologically
- Path $i \rightsquigarrow j$ means i must be listed before j
- Cycle $\Rightarrow$ 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!



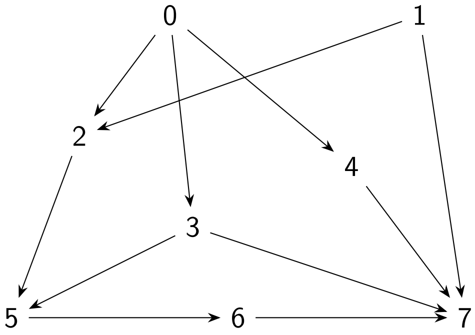
Claim

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
- ...



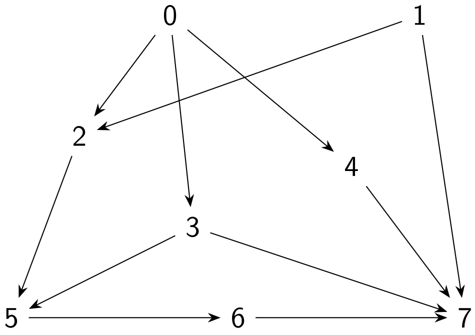
How to topologically sort a DAG?

Strategy

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

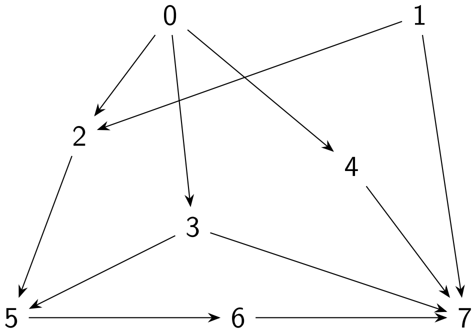
Questions

- 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, $\text{indegree}(v) = 0$

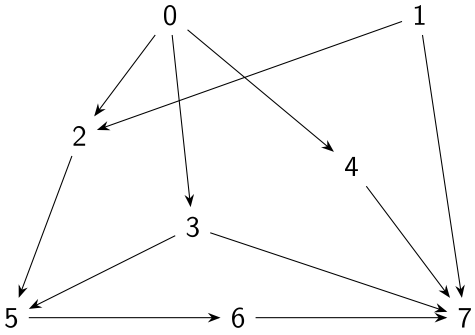


Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

Claim

Every DAG has a vertex with indegree 0



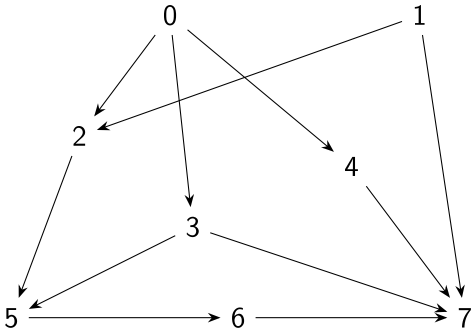
Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

Claim

Every DAG has a vertex with indegree 0

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

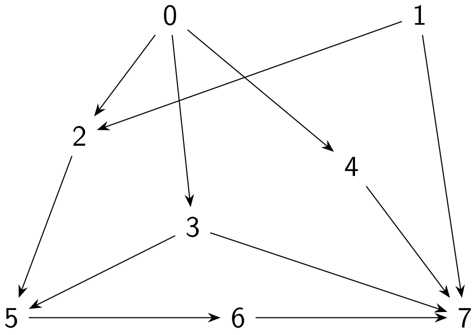


Topological sort algorithm

Fact

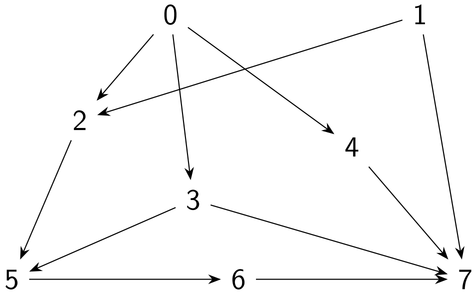
Every DAG has a vertex with indegree 0

- List out a vertex j with indegree $= 0$
- Delete j and 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



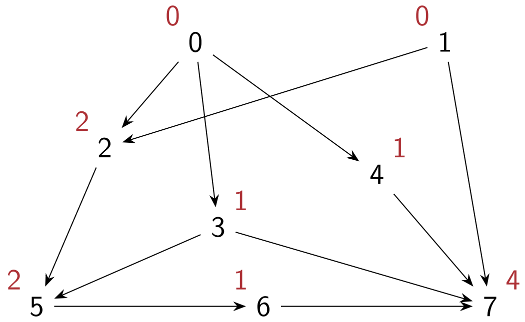
Topological sort algorithm

- Compute indegree of each vertex



- Compute indegree of each vertex
  - Scan each column of the adjacency matrix

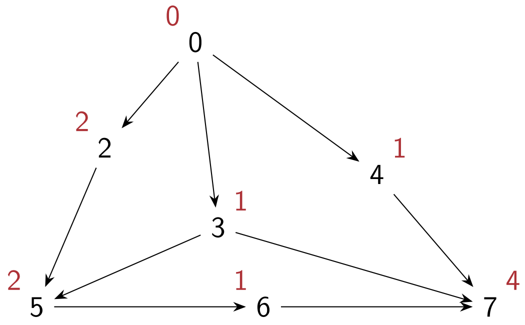
Indegree



Topological sort algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG

Indegree



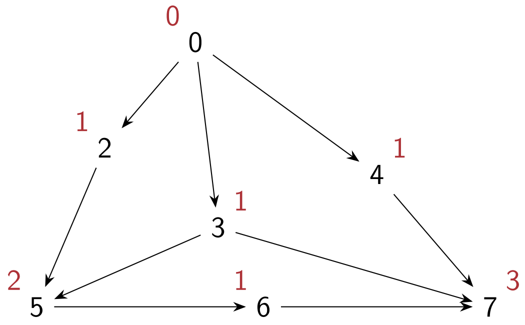
Topologically sorted sequence

1,

Topological sort algorithm

- 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

Indegree



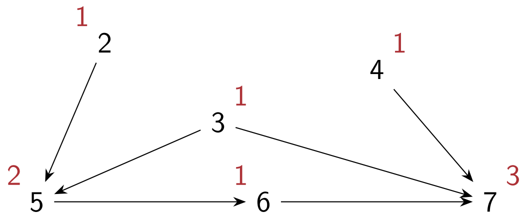
Topologically sorted sequence

1,

Topological sort algorithm

- 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

Indegree



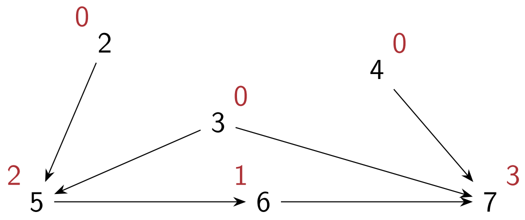
Topologically sorted sequence

1, 0,

Topological sort algorithm

- 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

Indegree



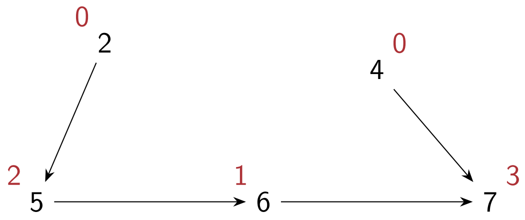
Topologically sorted sequence

1, 0,

Topological sort algorithm

- 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

Indegree



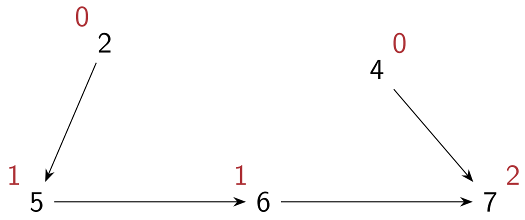
Topologically sorted sequence

1, 0, 3,

Topological sort algorithm

- 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

Indegree



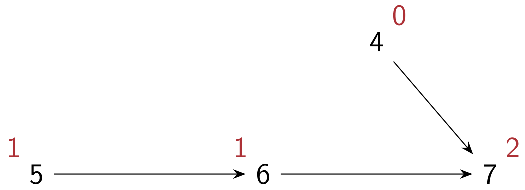
Topologically sorted sequence

1, 0, 3,

Topological sort algorithm

- 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

Indegree



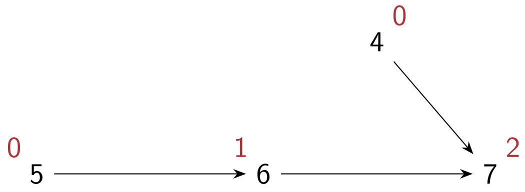
Topologically sorted sequence

1, 0, 3, 2,

Topological sort algorithm

- 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

Indegree



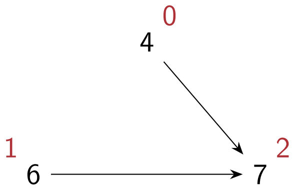
Topologically sorted sequence

1, 0, 3, 2,

Topological sort algorithm

- 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

Indegree



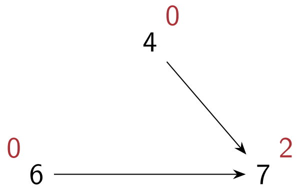
Topologically sorted sequence

1, 0, 3, 2, 5,

Topological sort algorithm

- 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

Indegree



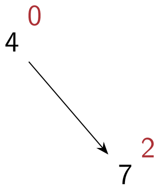
Topologically sorted sequence

1, 0, 3, 2, 5,

Topological sort algorithm

- 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

Indegree

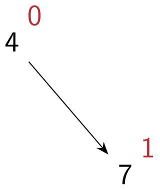


Topologically sorted sequence

1, 0, 3, 2, 5, 6,

- 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

Indegree



Topologically sorted sequence

1, 0, 3, 2, 5, 6,

Topological sort algorithm

- 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

Indegree

7^{1}

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4,

Topological sort algorithm

- 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

Indegree

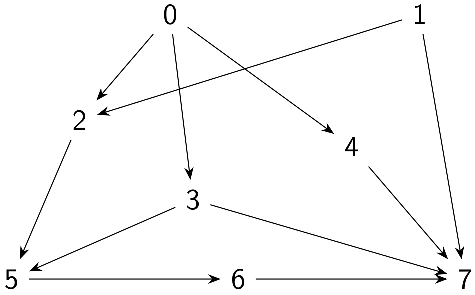
7 0

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4,

Topological sort algorithm

- 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



Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4, 7

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

```
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)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

```python
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)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

- Initializing indegrees is $O(n^2)$

```
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)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

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

```
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)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

- 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, $O(n^2)$

```
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)
```

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

```
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)
```

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

Analysis

```
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)
```

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

Analysis

- Initializing indegrees is $O(m+n)$

```python
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)
```

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

Analysis

- Initializing indegrees is $O(m+n)$
- Loop to enumerate vertices runs n times
  - Updating indegrees: amortised $O(m)$

```python
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)
```

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

Analysis

- Initializing indegrees is $O(m+n)$
- Loop to enumerate vertices runs n times
  - Updating indegrees: amortised $O(m)$
- Overall, $O(m+n)$

```
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)
```

Summary

- Directed acyclic graphs 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

Longest Paths in DAGs

Madhavan Mukund

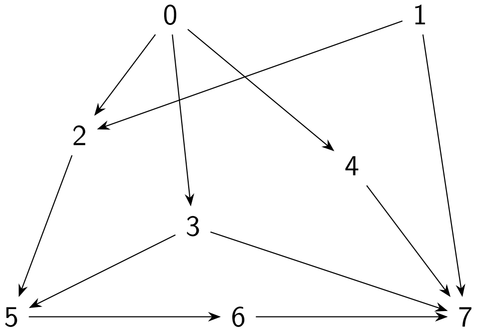
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

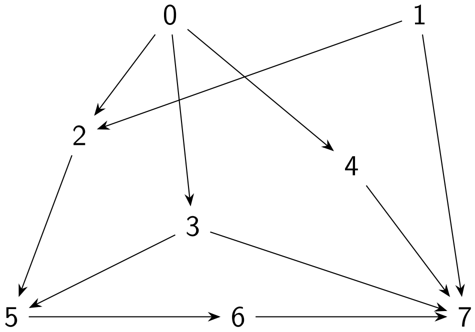
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles



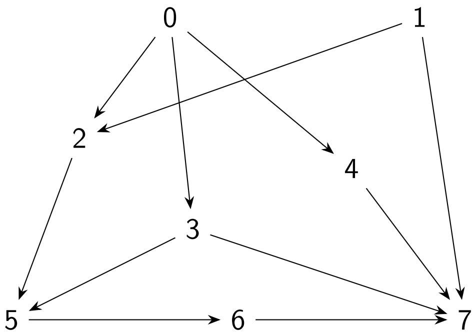
Directed Acyclic Graphs

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



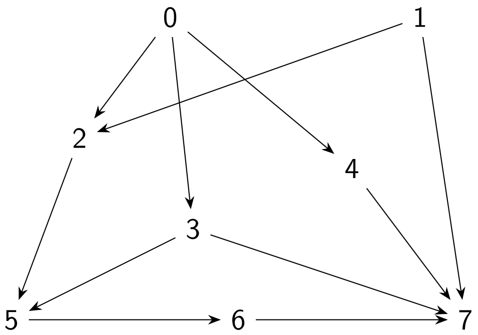
Directed Acyclic Graphs

- $G = (V, E)$, 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



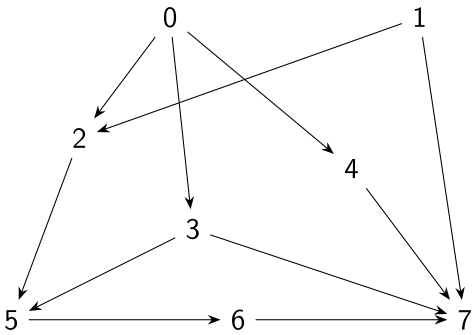
Directed Acyclic Graphs

- $G = (V, E)$, 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



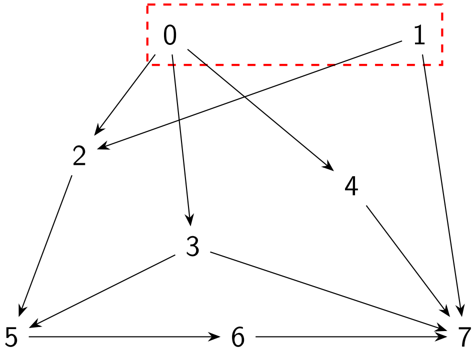
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
  - Enumerate $V = \{0, 1, \dots, n-1\}$ 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 number of semesters to complete the programme?



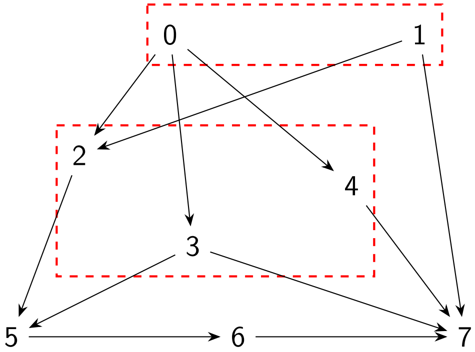
Directed Acyclic Graphs

- $G = (V, E)$, 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 number of semesters to complete the programme?



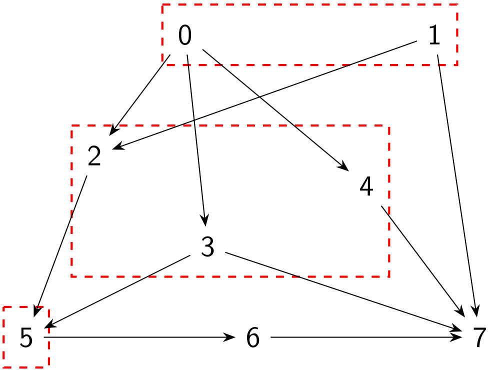
Directed Acyclic Graphs

- $G = (V, E)$, 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 number of semesters to complete the programme?



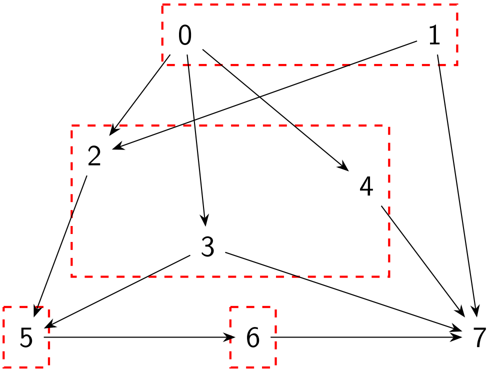
Directed Acyclic Graphs

- $G = (V, E)$, 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 number of semesters to complete the programme?



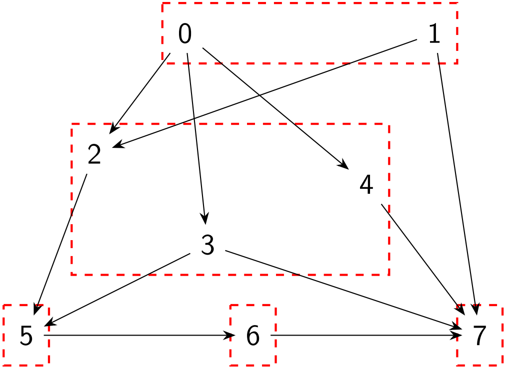
Directed Acyclic Graphs

- $G = (V, E)$, 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 number of semesters to complete the programme?



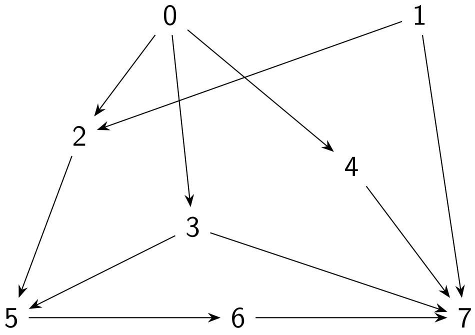
Directed Acyclic Graphs

- $G = (V, E)$, 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 number of semesters to complete the programme?



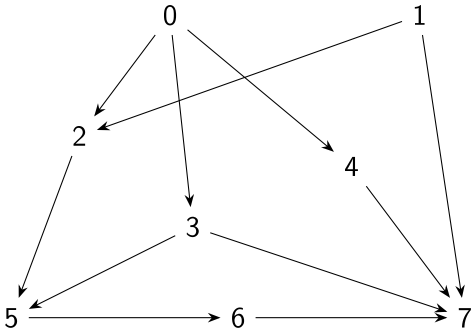
Longest Path

- Find the longest path in a DAG



Longest Path

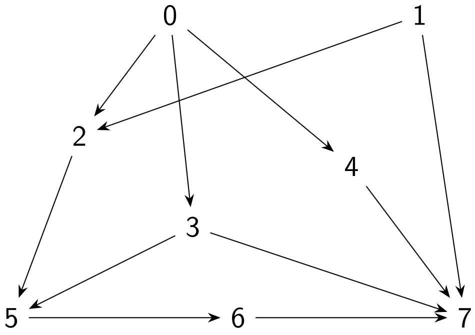
- Find the longest path in a DAG
- If indegree$(i) = 0$,
  longest-path-to$(i) = 0$



Longest Path

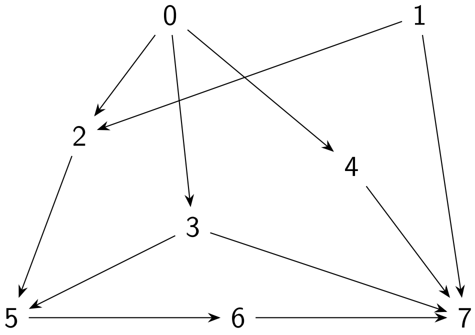
- Find the longest path in a DAG
- If $\text{indegree}(i) = 0$,
  $\text{longest-path-to}(i) = 0$
- If $\text{indegree}(i) > 0$, longest path to i is 1 more than longest path to its incoming neighbours

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



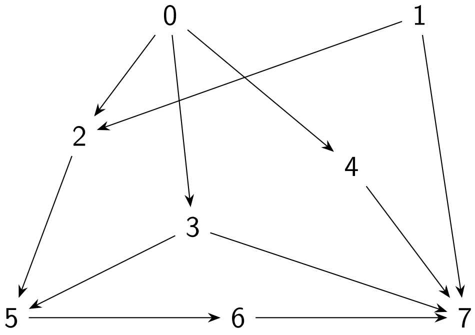
Longest Path

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



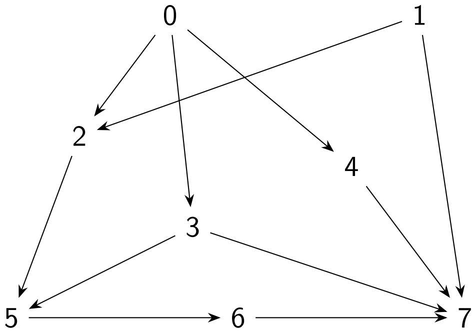
Longest Path

- 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



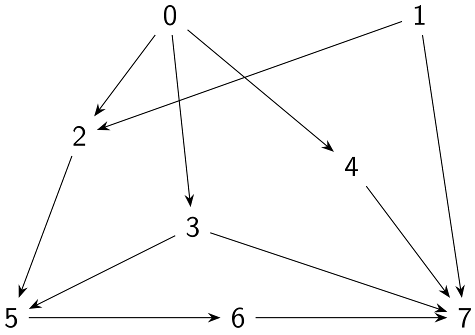
Longest Path

- 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



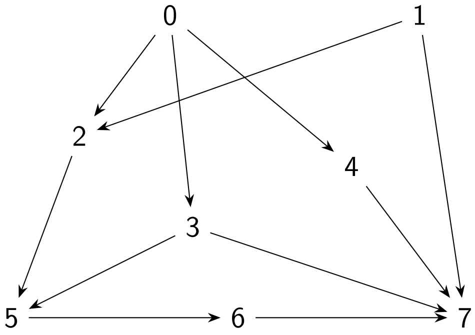
Longest Path

- longest-path-to(i) $=$
  $1 + \max\{$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



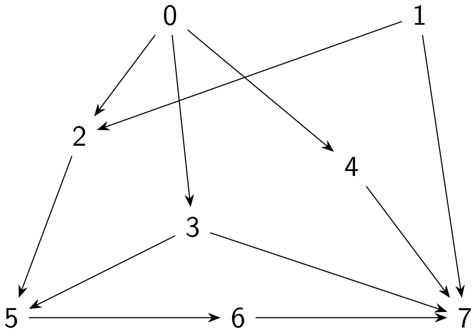
Longest Path

- Let $i_0, i_1, \ldots, i_{n-1}$ be a topological ordering of V



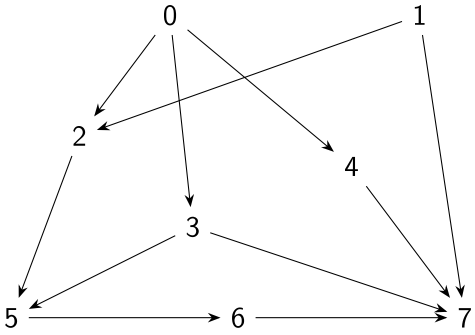
Longest Path

- 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



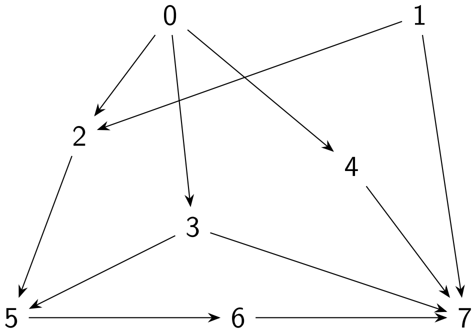
Longest Path

- 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 $\text{longest-path-to}(i_k)$ as $1 + \max\{\text{longest-path-to}(i_j) \mid (i_j, i_k) \in E\}$



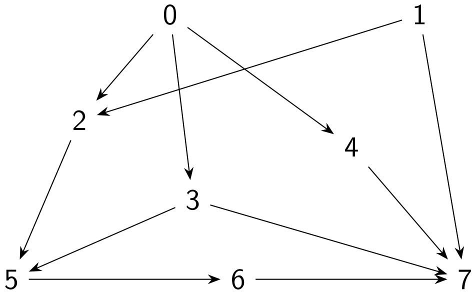
Longest Path

- 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

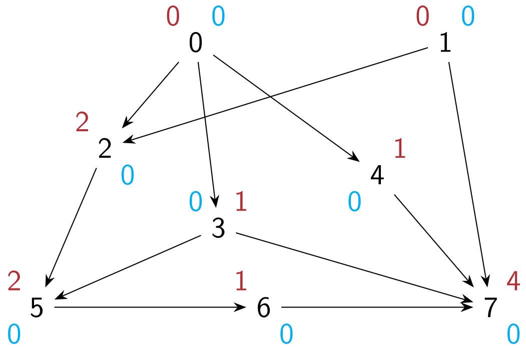
- Compute indegree of each vertex



Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize $textlongest - path - to$ to 0 for all vertices

Indegree, Longest path



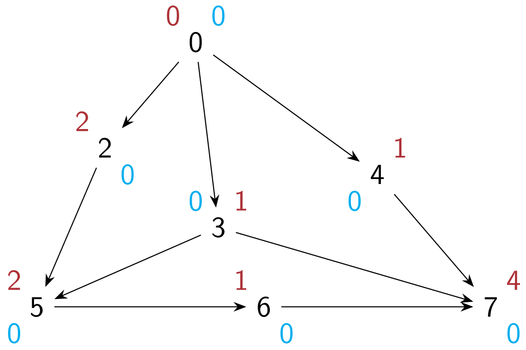
Topological order

Longest path to

Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize $text longest - path - to$ to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG

Indegree, Longest path

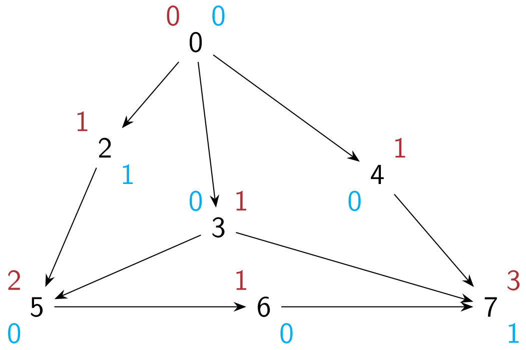


Topological order	1
Longest path to	0

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

Indegree, Longest path

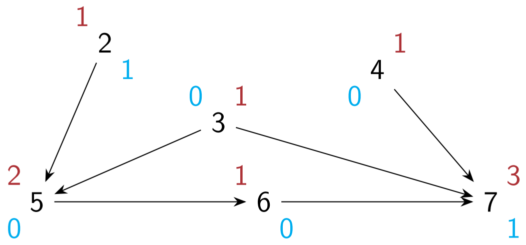


Topological order	1
Longest path to	0

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

Indegree, Longest path

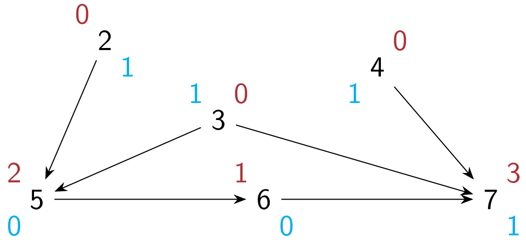


Topological order	1	0
Longest path to	0	0

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

Indegree, Longest path

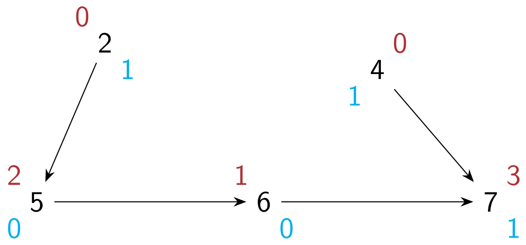


Topological order	1	0
Longest path to	0	0

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

Indegree, Longest path

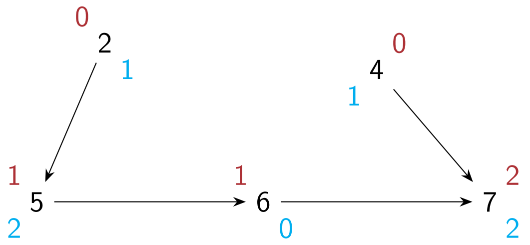


Topological order	1	0	3
Longest path to	0	0	1

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

Indegree, Longest path

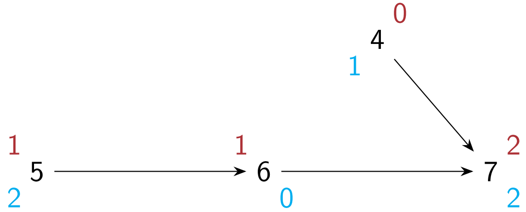


Topological order	1	0	3
Longest path to	0	0	1

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

Indegree, Longest path

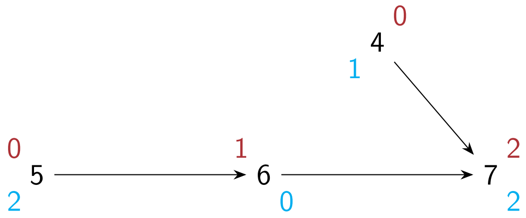


Topological order	1	0	3	2
Longest path to	0	0	1	1

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

Indegree, Longest path

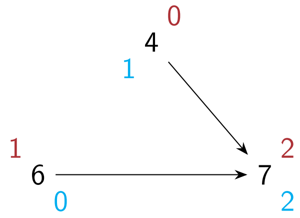


Topological order	1	0	3	2
Longest path to	0	0	1	1

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

Indegree, Longest path

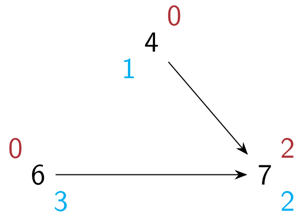


Topological order	1	0	3	2	5
Longest path to	0	0	1	1	2

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

Indegree, Longest path

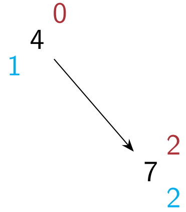


Topological order	1	0	3	2	5
Longest path to	0	0	1	1	2

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

Indegree, Longest path

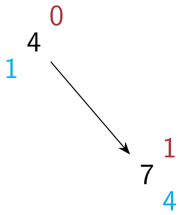


Topological order	1	0	3	2	5	6
Longest path to	0	0	1	1	2	3

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

Indegree, Longest path



Topological order	1	0	3	2	5	6
Longest path to	0	0	1	1	2	3

Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize $text longest - path - to$ 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

Indegree, Longest path

1
7
4

Topological order	1	0	3	2	5	6	4
Longest path to	0	0	1	1	2	3	1

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

Indegree, Longest path

0
7
4

Topological order	1	0	3	2	5	6	4
Longest path to	0	0	1	1	2	3	1

Longest path algorithm

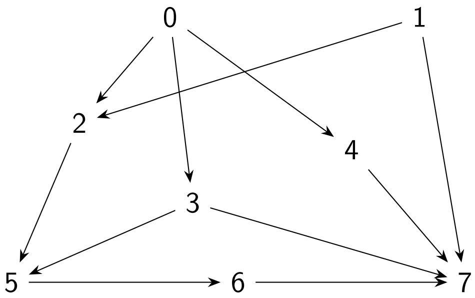
- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize $textlongest-path-to$ to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

Topological order	1	0	3	2	5	6	4	7
Longest path to	0	0	1	1	2	3	1	4

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



Topological order	1	0	3	2	5	6	4	7
Longest path to	0	0	1	1	2	3	1	4

Longest path 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 till queue is empty

```
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:
        zerodegreeq.addq(k)
  return(lpath)
```

Longest path 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 till queue is empty

Analysis

```
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:
        zerodegreeq.addq(k)
  return(lpath)
```

Longest path 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 till queue is empty

Analysis

- Initializing indegrees is $O(m+n)$

```
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:
        zerodegreeq.addq(k)
  return(lpath)
```

Longest path 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 till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$
- Loop to enumerate vertices runs n times
  - Updating indegrees, longest path: amortised $O(m)$

```
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:
        zerodegreeq.addq(k)
  return(lpath)
```

Longest path 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 till queue is empty

Analysis

- Initializing indegrees is $O(m+n)$
- Loop to enumerate vertices runs n times
  - Updating indegrees, longest path: amortised $O(m)$
- Overall, $O(m+n)$

```
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:
        zerodegreeq.addq(k)
  return(lpath)
```

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- In parallel with topological sort, we can compute the longest path

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- 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

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- 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
  - No better strategy known than exhaustively enumerating paths