QEEE module

Graphs

G Venkatesh
IIT Madras

Introduction:

How graphs fits into
the study of algorithms and data structures

Algorithms and Data Structures

<u>What is the problem?</u>

•

Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Algorithmic techniques

-

Algorithms and Data Structures

<u>What is the problem?</u>

- Prototype problems
- Variations

<u>Algorithmic techniques</u>

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

Algorithms and Data Structures

<u>What is the problem?</u>

- Prototype problems
- Variations

<u>Algorithmic techniques</u>

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

<u>Data structures</u>

Abstract Data Types

Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Algorithmic techniques

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

Data structures

Abstract Data Types

An abstract data type specifies a set of values, the set of operations permitted on those values, and the behaviour of these operations

Algorithms and Data Structures

<u>What is the problem?</u>

- Prototype problems
- Variations

<u>Algorithmic techniques</u>

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

<u>Data structures</u>

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

Algorithms and Data Structures

<u>What is the problem?</u>

- Prototype problems
- Variations

<u>Algorithmic techniques</u>

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

<u>Data structures</u>

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

<u>Analysis</u>

-

Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Data structures

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

Algorithmic techniques

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

Analysis

- Complexity
-

Algorithms and Data Structures

<u>What is the problem?</u>

- Prototype problems
- Variations

<u>Algorithmic techniques</u>

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

<u>Data structures</u>

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

<u>Analysis</u>

- Correctness
- Complexity

Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Algorithmic techniques

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

Graphs

Data structures

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

Analysis

- Correctness
- Complexity

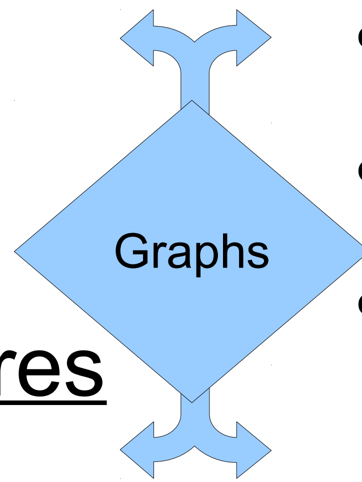
Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Data structures

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees



Algorithmic techniques

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

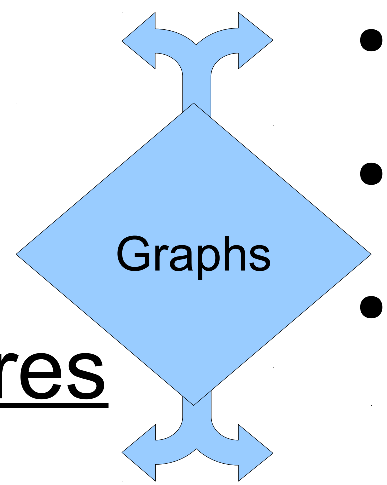
Analysis

- Correctness
- Complexity

Algorithms and Data Structures

<u>What is the problem?</u>

- Prototype problems
- Variations



<u>Data structures</u>

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

<u>Algorithmic techniques</u>

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow

<u>Analysis</u>

- Parameters:
  - No of nodes – n
  - No of edges - m

Representation of Graphs

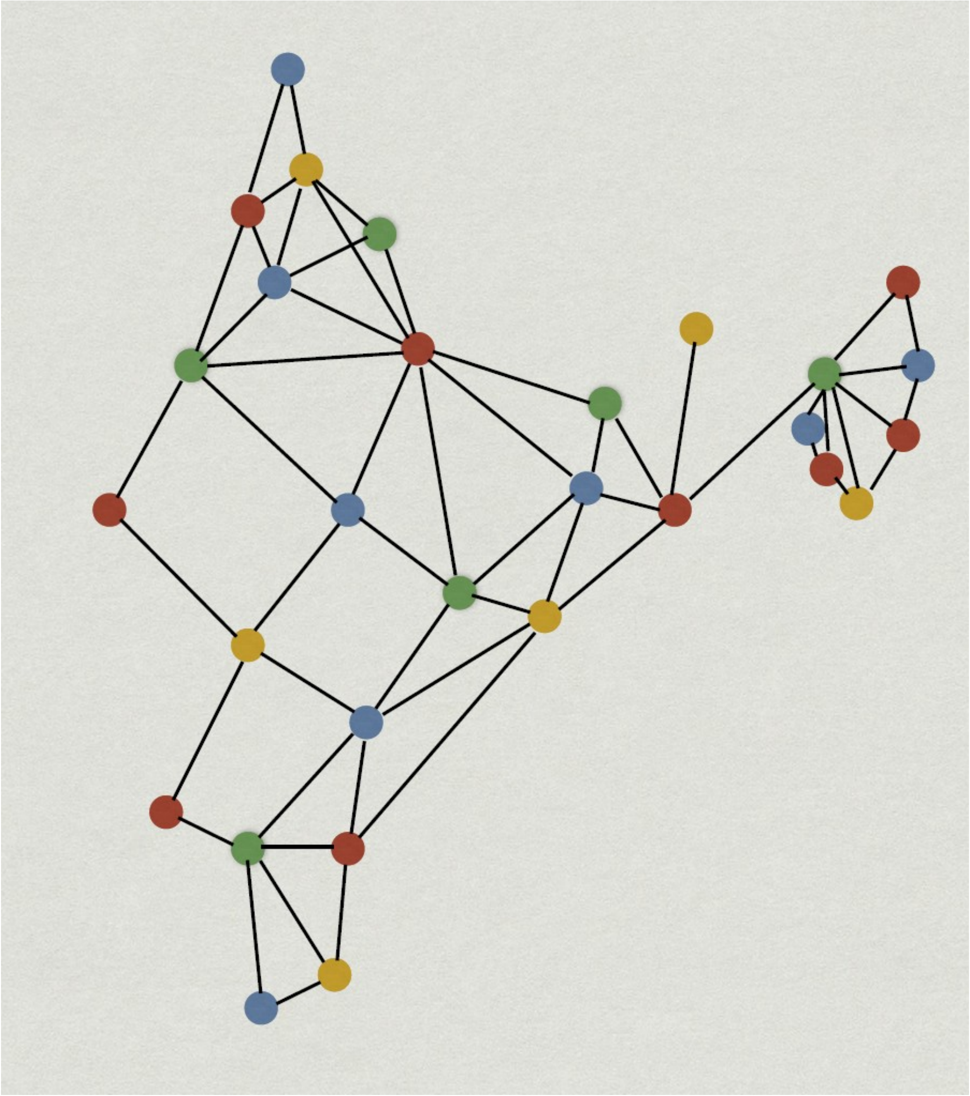
Graph problem 1: Colour a map

Mark each state

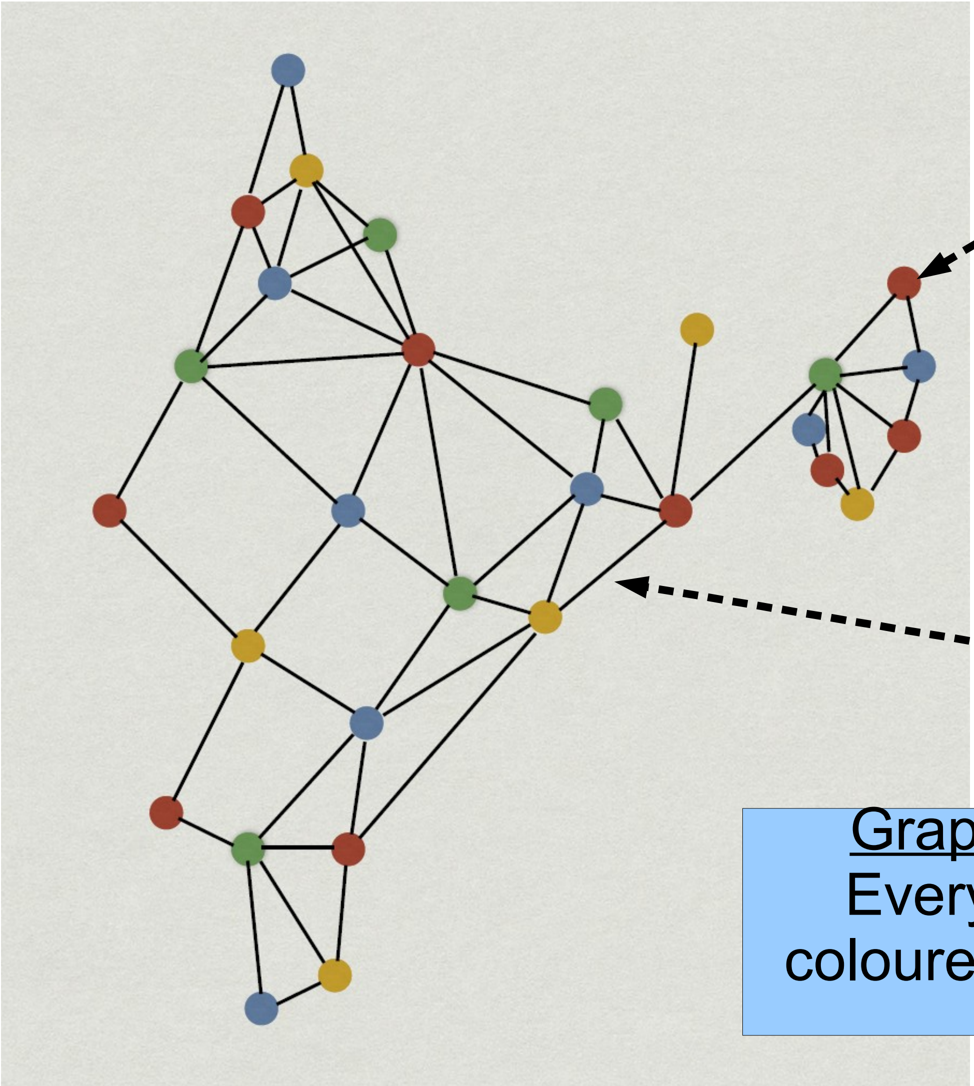
Connect states that share a border

Colour the dots … neighbouring dots should have different colour

The map itself is irrelevant



Graphs with nodes and edges



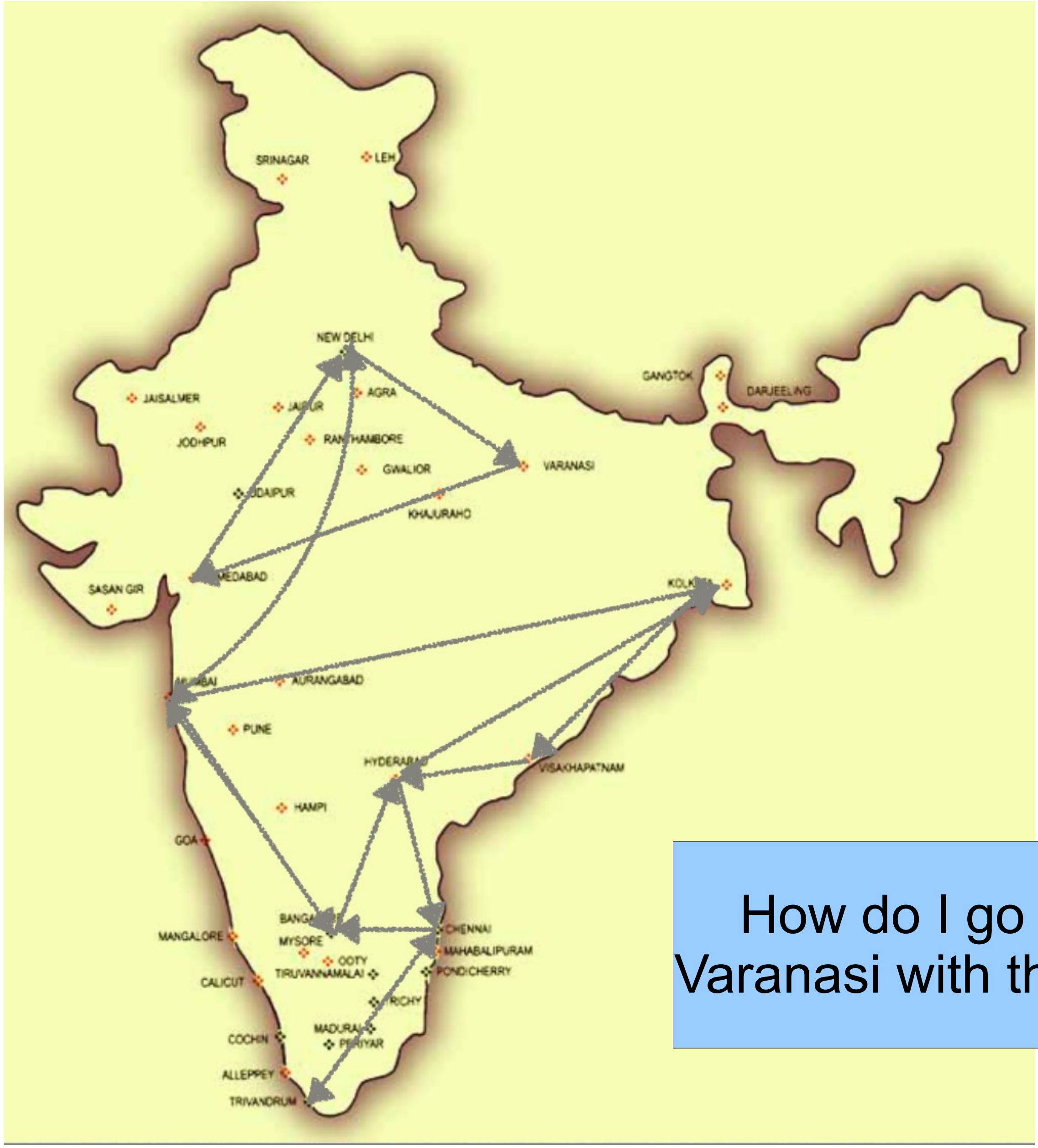
Node

Edge

Graph colouring problem:
Every planar graph can be
coloured with at most 4 colours

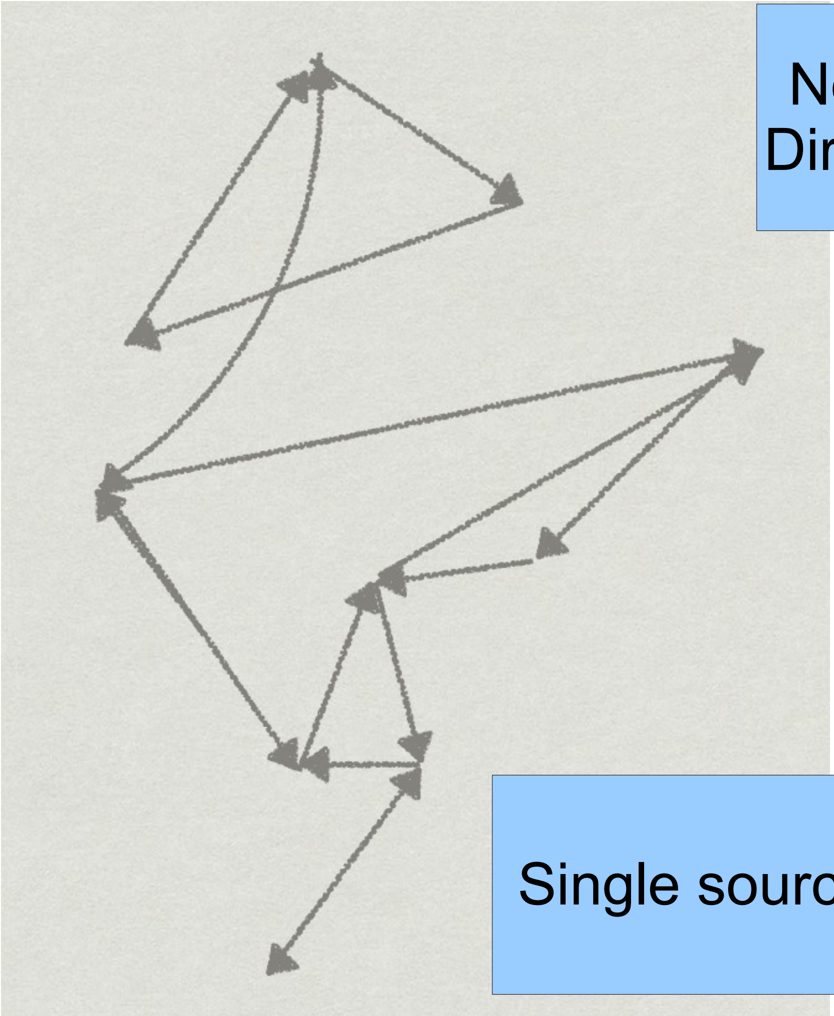
Finds applications in many fields:
for example in compiler design for register allocation

Another example: Airline routing



How do I go from Chennai to Varanasi with the least no of stops

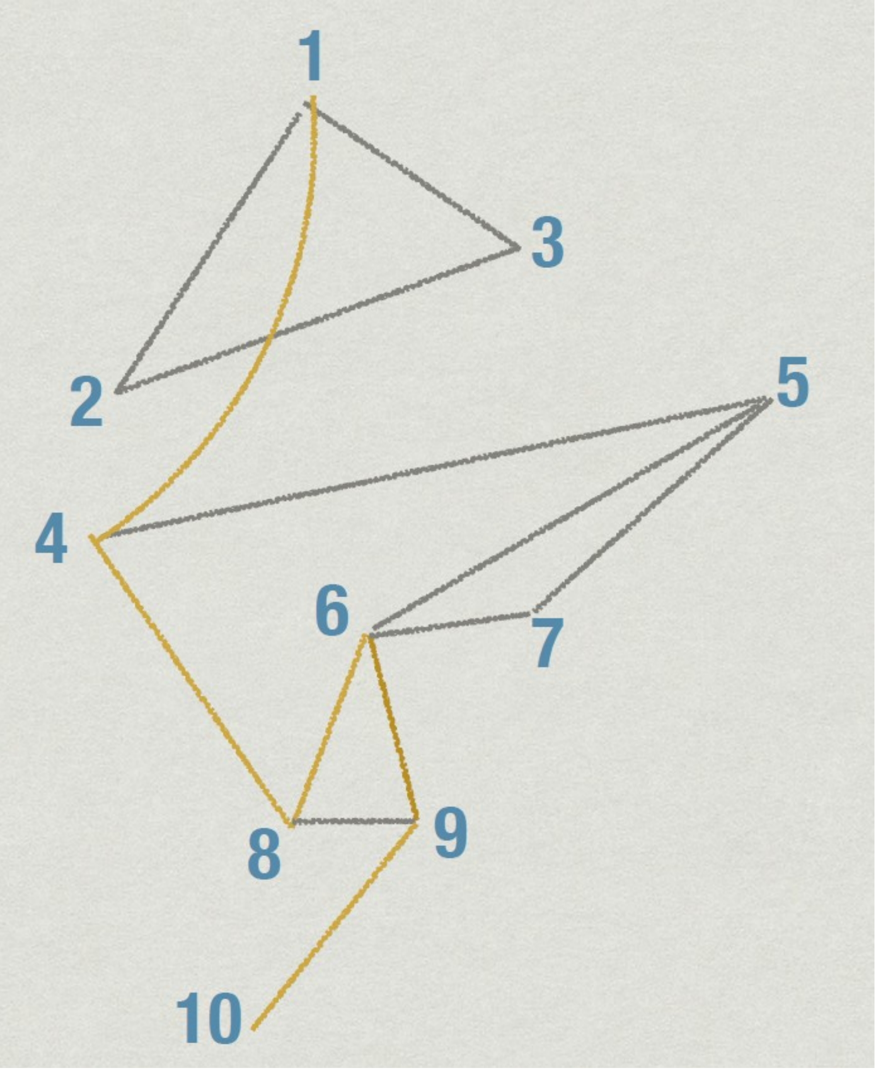
Directed graph



Nodes are cities (stops)
Directed edges are flights

Single source shortest path problem

An undirected graph may be enough



When flights fly both ways between cities

Definition of graphs

- Graph G = (V,E)
- V is a set of Vertices (or nodes)
- E is a set of edges, $E \subseteq V \times V$
- Each edge is a pair of vertices (v,v')
- If graph is undirected, then (v',v) and (v,v') are the same edge and both should be in E
- A *path from u to v* is a sequences of vertices $u=v_1,v_2,v_3,...v_n=v$ such that $(v_i,v_{i+1}) \in E$.
- A path is *simple* if the vertices in the path are distinct.

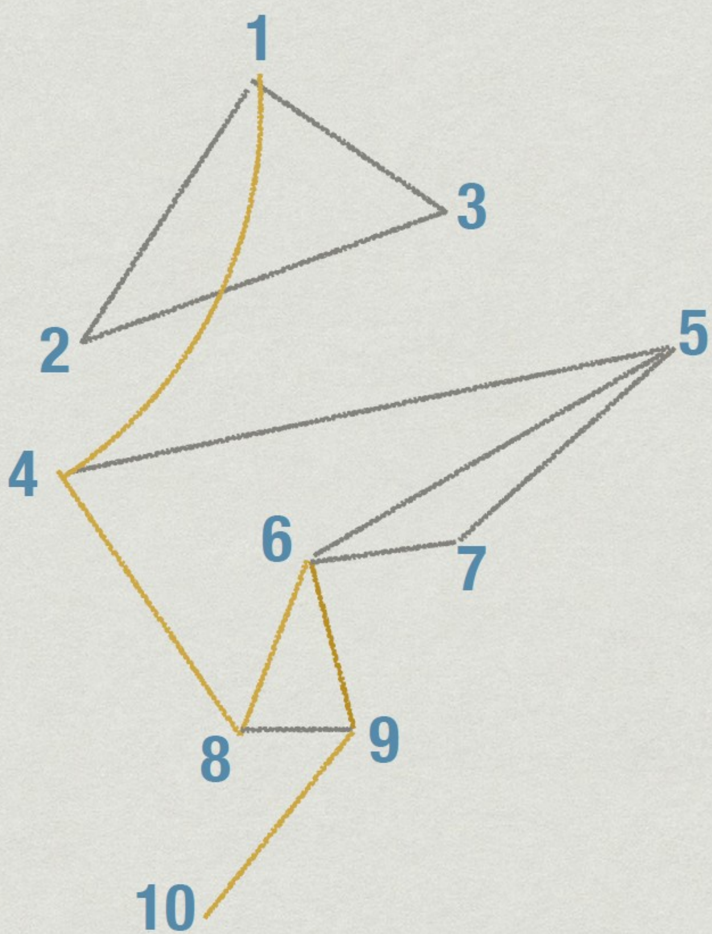
Graph algorithm complexity

- Graph G = (V,E)
- V is a set of Vertices (or nodes)
- E is a set of edges, $E \subseteq V \times V$
- Each edge is a pair of vertices (v,v')
- If graph is undirected, then (v',v) and (v,v') are the same edge and both should be in E
- Number of nodes = n, number of edges = m
- Complexity of graph data structures and algorithms evaluated with respect to n and m e.g. $O(n+m)$, $O(n^2)$, $O(nm)$, ...

Representing graphs

- Assume vertices are *named* using numbers 1,2,...n.
- Each edge is a pair (i,j), $1 \leq i,j \leq n$
- Easiest way to represent is using a matrix A:
  - $A(i,j) = 1$ if $(i,j) \in E$, $A(i,j) = 0$ otherwise
  - Called the adjacency matrix

Adjacency matrix example



	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

Using an adjacency matrix

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

Neighbour of a vertex (say 4) are obtained by scanning a row

Using an adjacency matrix

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

Neighbour of a vertex (say 4) are obtained by scanning a row

Takes O(n) time to locate all the neighbours.

Using an adjacency matrix

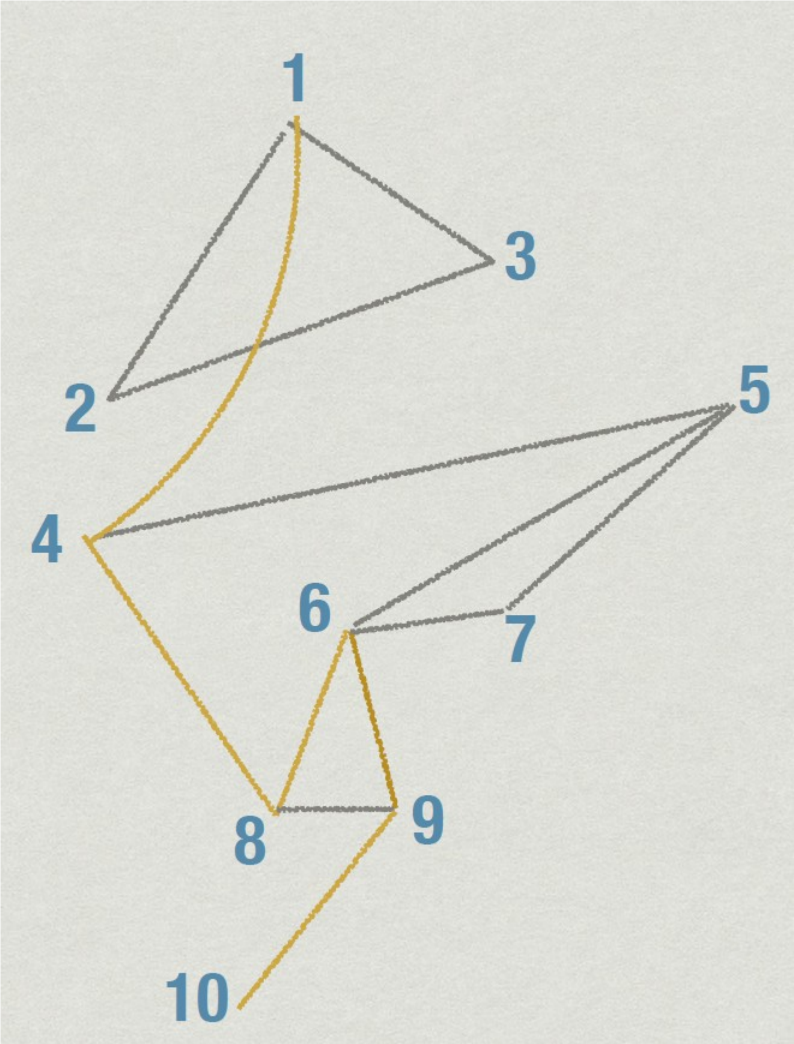
	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

Neighbour of a vertex (say 4) are obtained by scanning a row

Takes O(n) time to locate all the neighbours.

But checking if (i,j) are neighbours is constant time

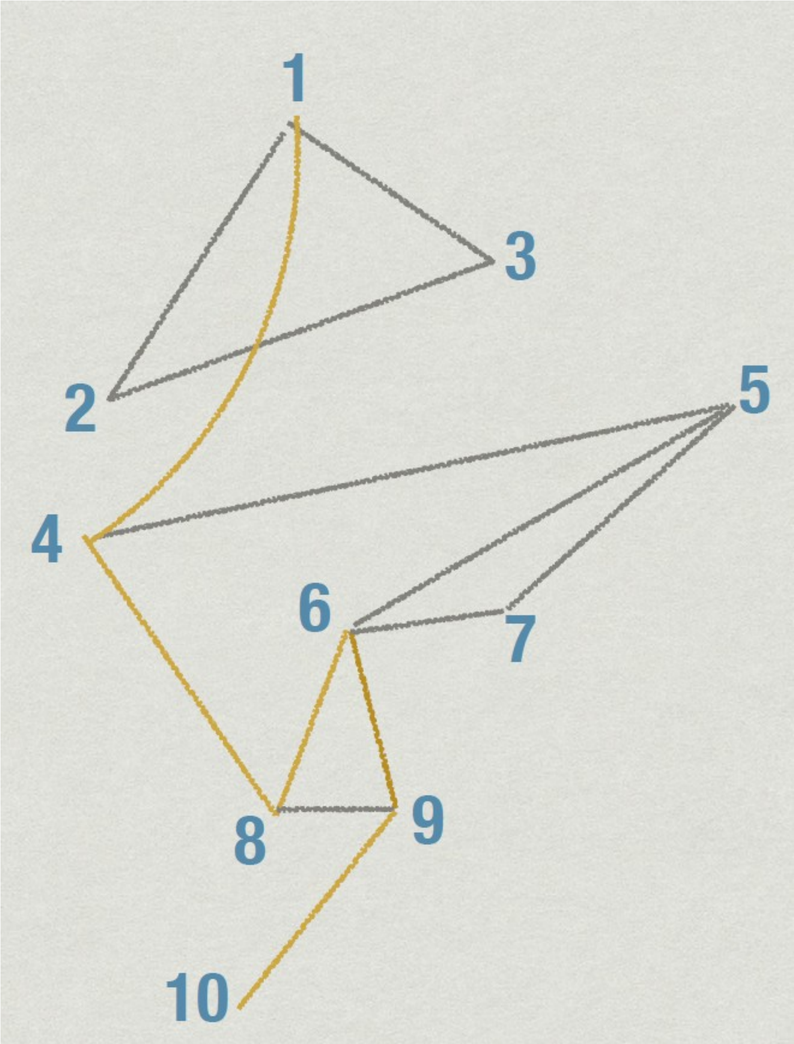
Adjacency list



1	2,3,4
2	1,3
3	1,2
4	1,5,8
5	4,6,7
6	5,7,8,9
7	5,6
8	4,6,9
9	6,8,10
10	9

For each vertex,
we maintain
list of neighbours

Adjacency list

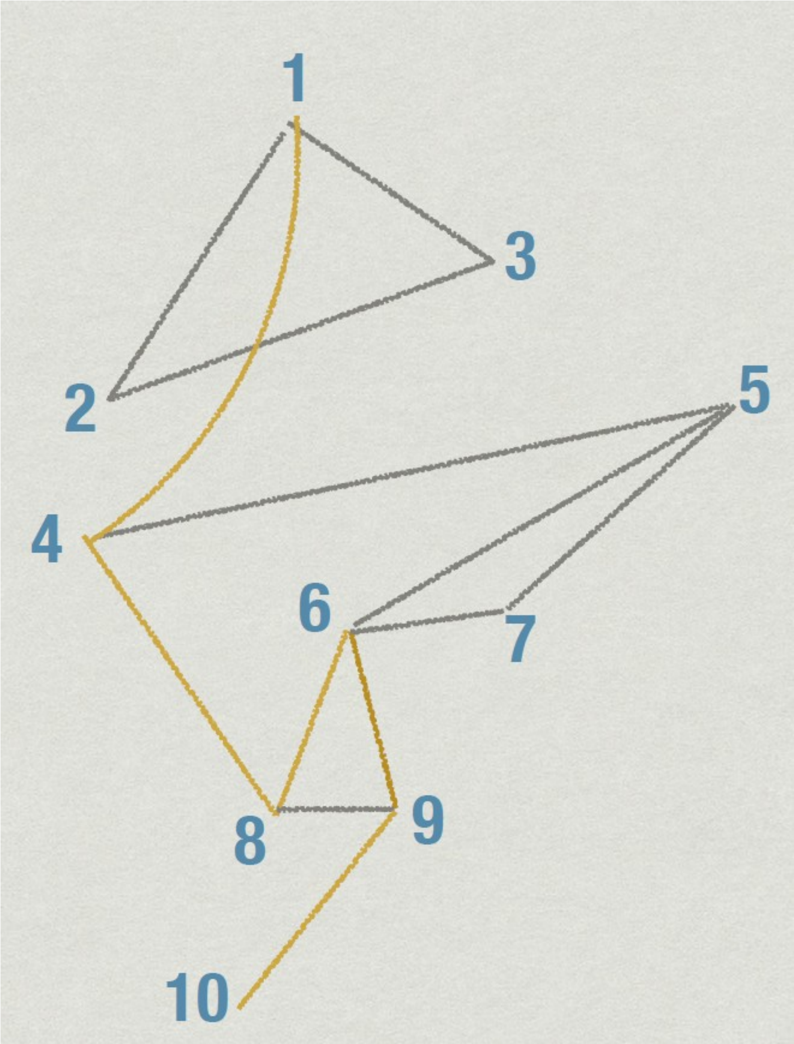


1	2,3,4
2	1,3
3	1,2
4	1,5,8
5	4,6,7
6	5,7,8,9
7	5,6
8	4,6,9
9	6,8,10
10	9

For each vertex, we maintain list of neighbours

Scanning through neighbours is easy

Adjacency list



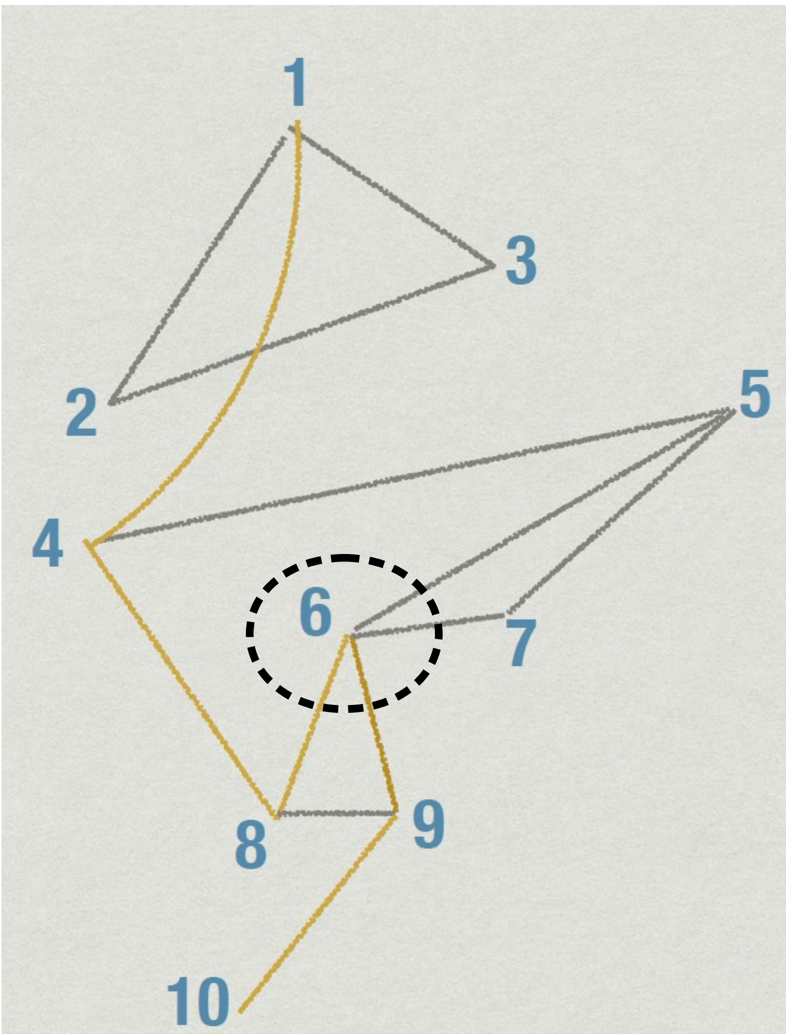
1	2,3,4
2	1,3
3	1,2
4	1,5,8
5	4,6,7
6	5,7,8,9
7	5,6
8	4,6,9
9	6,8,10
10	9

For each vertex, we maintain list of neighbours

Scanning through neighbours is easy

Finding if (i.j) are neighbours is not constant time now !

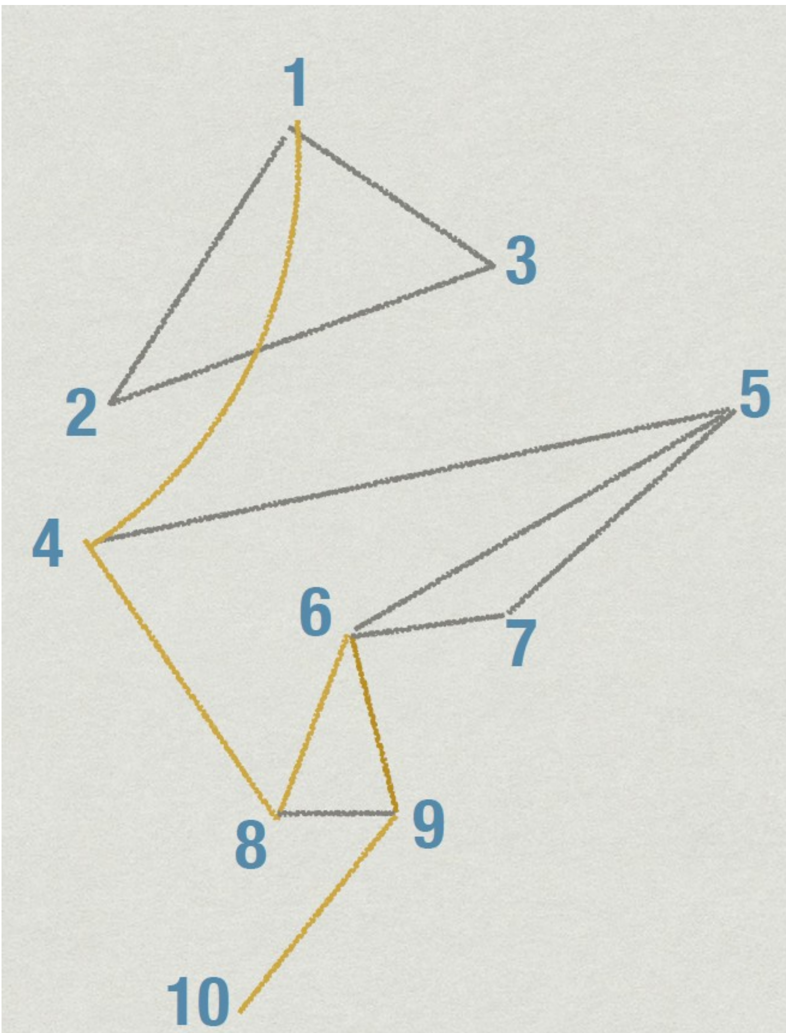
Which data structure to use?



Max degree is 4 (from node 6).

- Example 1:
  - The *degree* of a vertex v in an undirected graph G = (V,E) is the number of edges incident on v.
  - The *maximum degree* of a graph G is the maximum value of the degree of all the vertices in the graph.
  - Which representation – adjacency matrix or list is better for finding the maximum degree of a graph?

Which data structure to use?



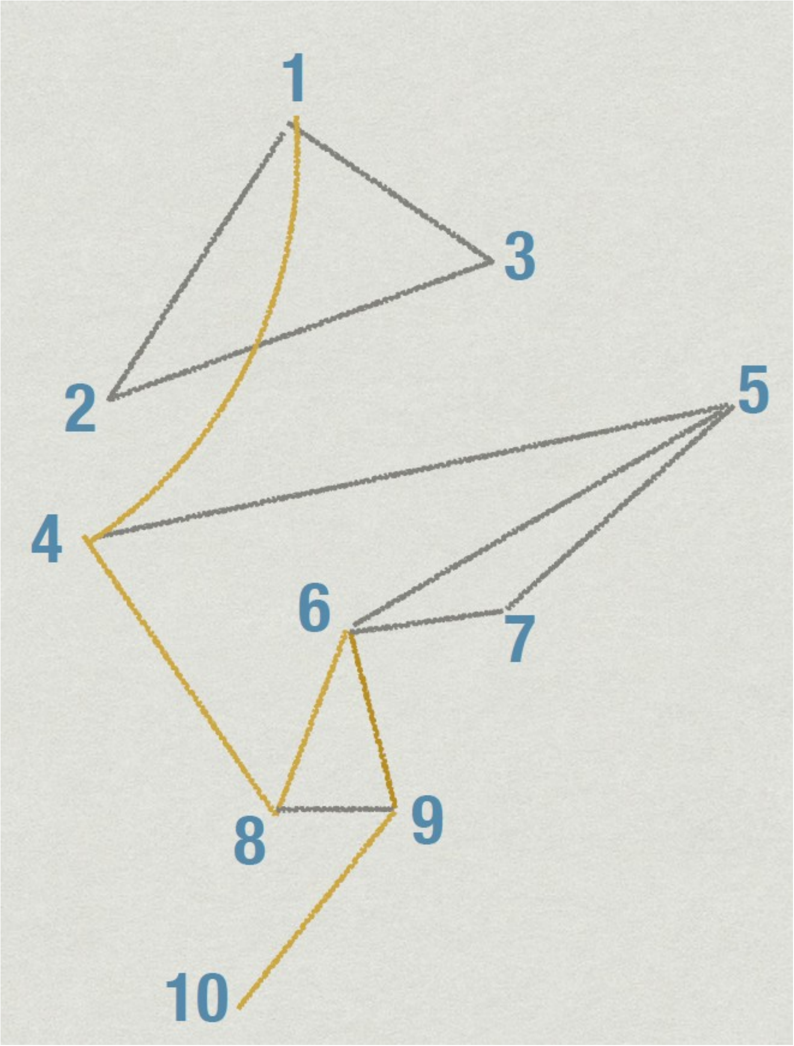
- Example 2:
  - A *clique* is a subset C of the vertices V of G such that for each pair of vertices u,v in C, (u,v) is an edge of the graph
  - For instance a clique of size 3 will form a triangle in the graph.
  - Which representation – adjacency matrix or list is better for finding out if a given set of vertices X is a clique or not?

{5,6,7} form a clique, but {1,4,5} does not

Graph traversal:

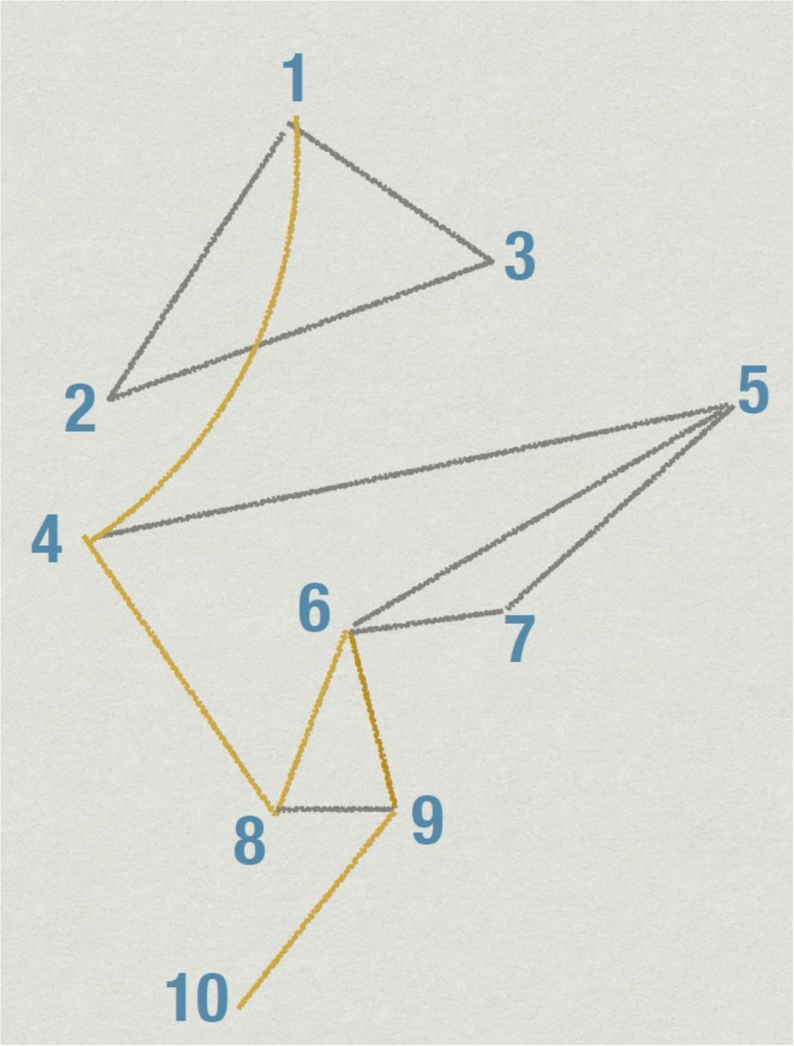
Breadth First Search

Example problem



Lets say – we want to find if there is a way to go by air from Chennai (9) to Varanasi (3)

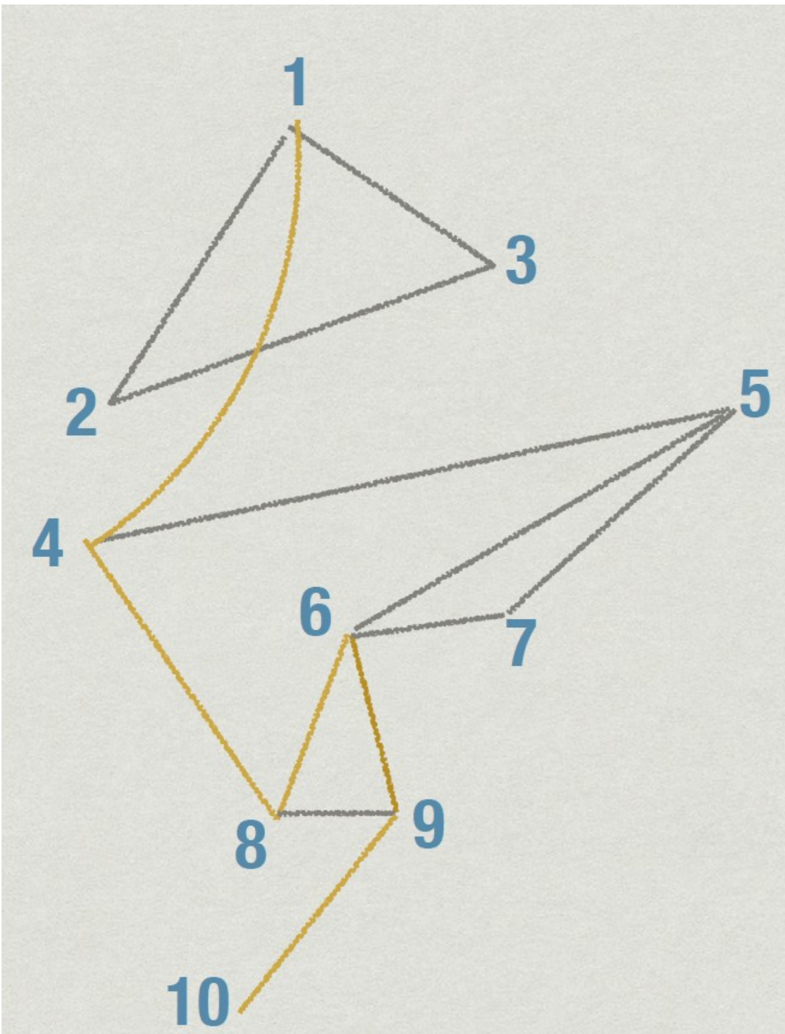
Example problem



Lets say – we want to find if there is a way to go by air from Chennai (9) to Varanasi (3)

For us it is easy !
We can visualise the path

Example problem

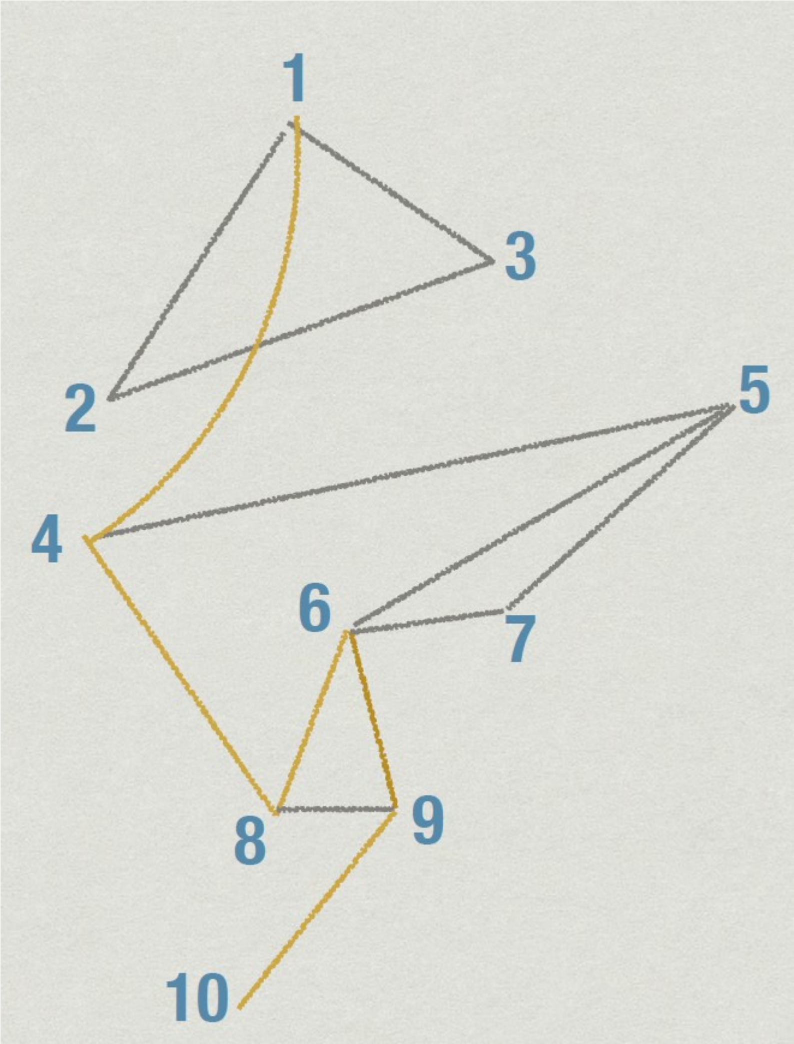


Lets say – we want to find if there is a way to go by air from Chennai (9) to Varanasi (3)

For us it is easy !
We can visualise the path

The algorithm only has information only through adjacency matrix or list data structure, nothing more

Example problem



Lets say – we want to find if there is a way to go by air from Chennai (9) to Varanasi (3)

For us it is easy !
We can visualise the path

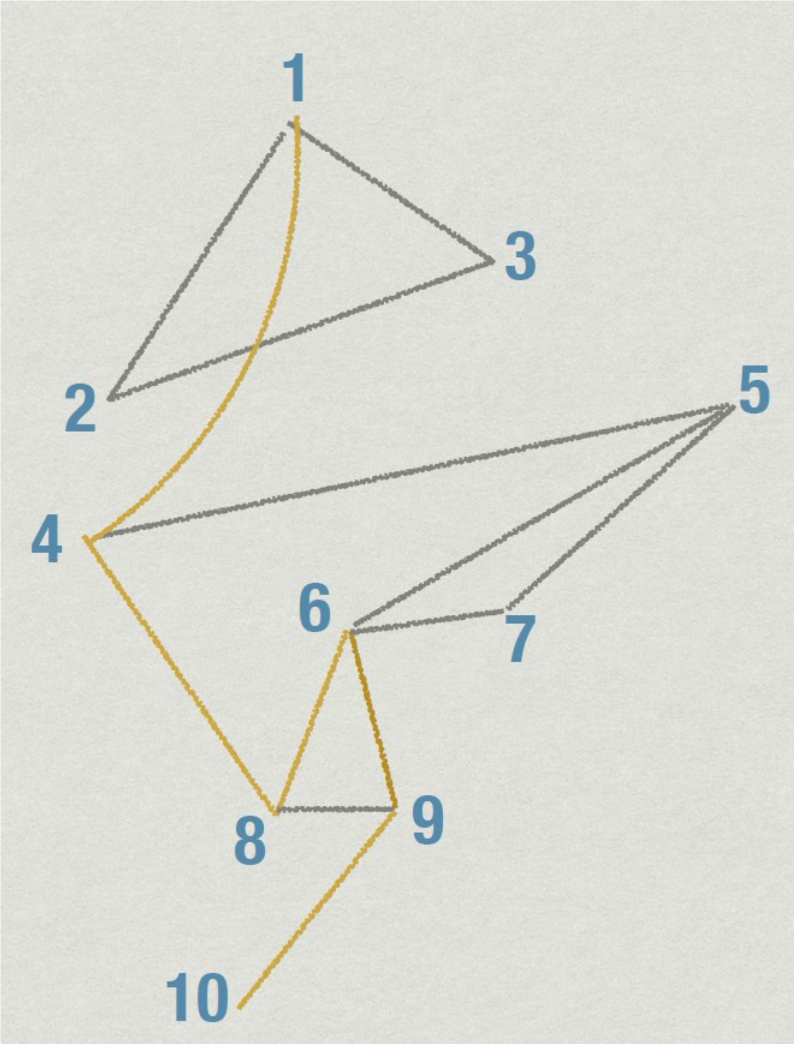
The algorithm only has information only through adjacency matrix or list data structure, nothing more

Need a systematic way of exploring the graph through only neighbour information

Exploring a graph using BFS

- In *Breadth First Search (BFS),* the graph is explored level by level starting from any vertex.
  - First visit the vertex
  - Then visit the vertices that are one step away
  - Then visit vertices two steps away ....

Breadth First Search



Start with Chennai (9).

Level 1: 6,8,10

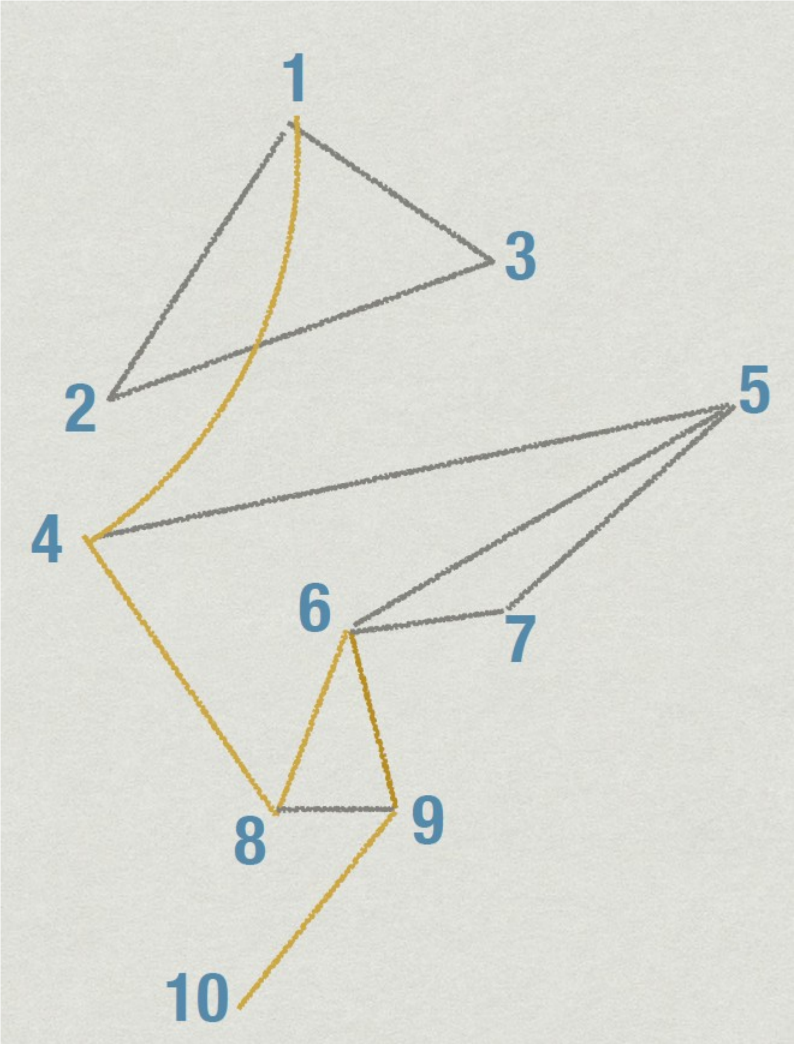
Level 2: 8,9,7,5,6,9,4

6,8,9 are being re-visited again ...

Exploring a graph using BFS

- In *Breadth First Search (BFS),* the graph is explored level by level starting from any vertex.
  - First visit the vertex
  - Then visit the vertices that are one step away
  - Then visit vertices two steps away ....
- Remember which vertices have been visited in an array called visited (visited[i] = true when i is visited)

Breadth First Search



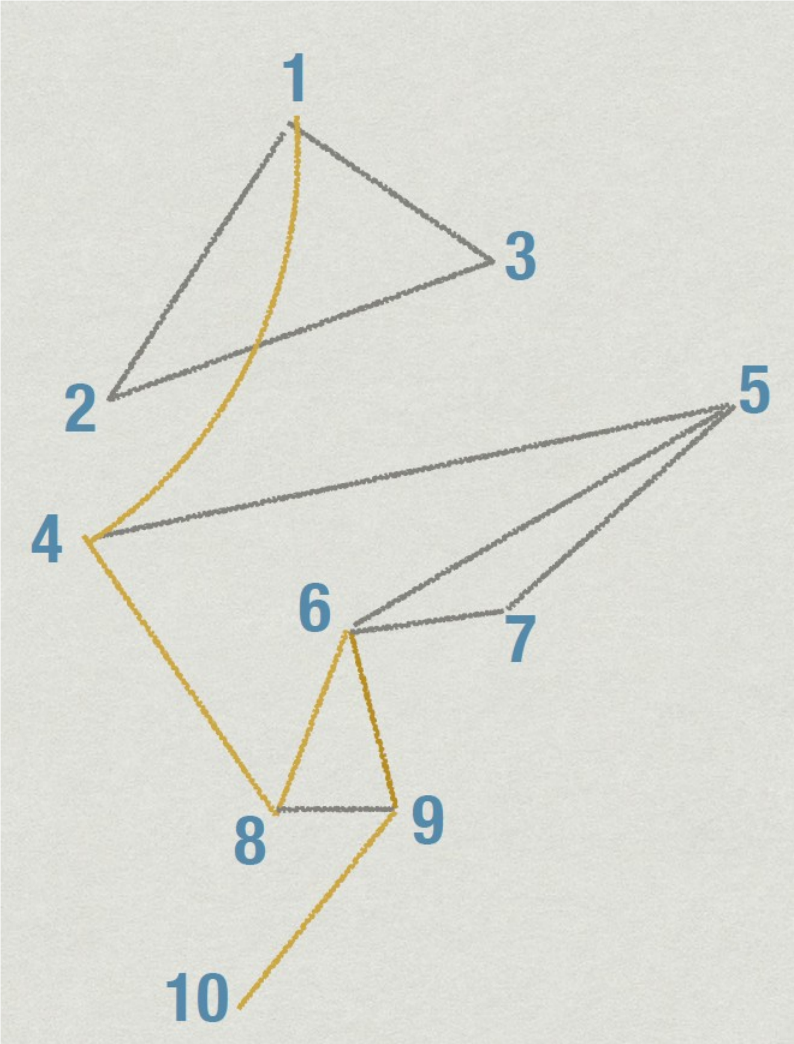
Start with Chennai (9).

Level 1: 6,8,10

Level 2: 7,5,4

Level 3: ?

Breadth First Search



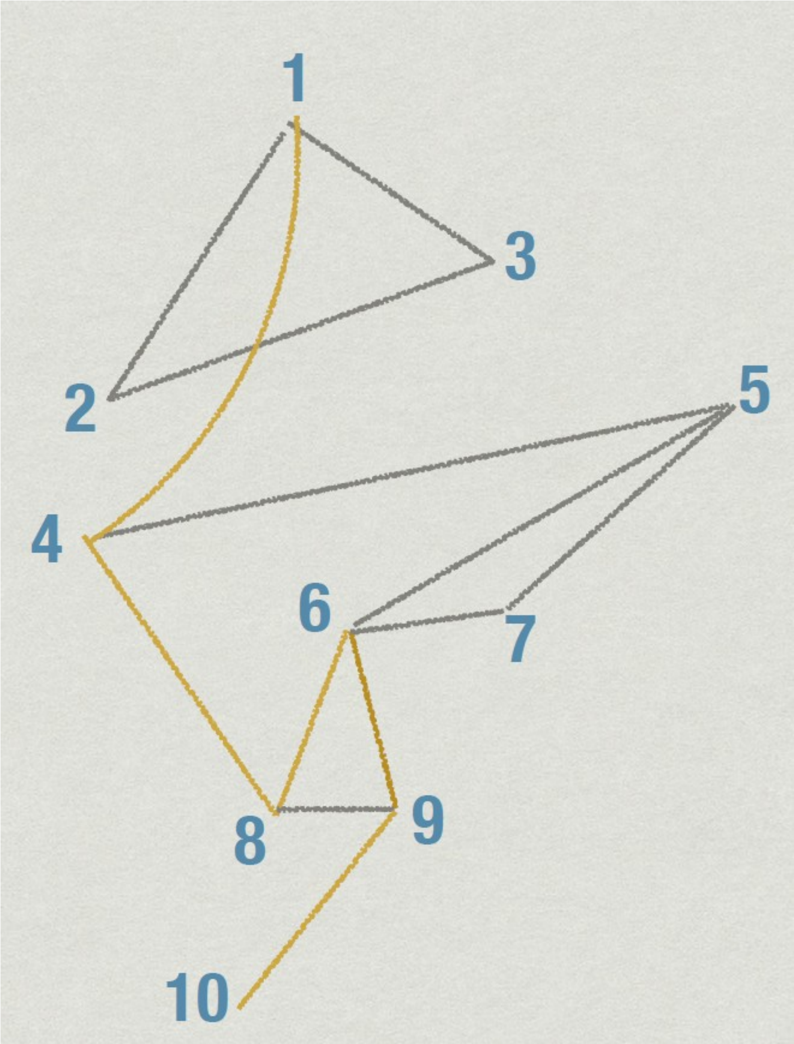
Start with Chennai (9).

Level 1: 6,8,10

Level 2: 7,5,4

Note that 6,8,10 neighbours are exhausted

Breadth First Search



Start with Chennai (9).

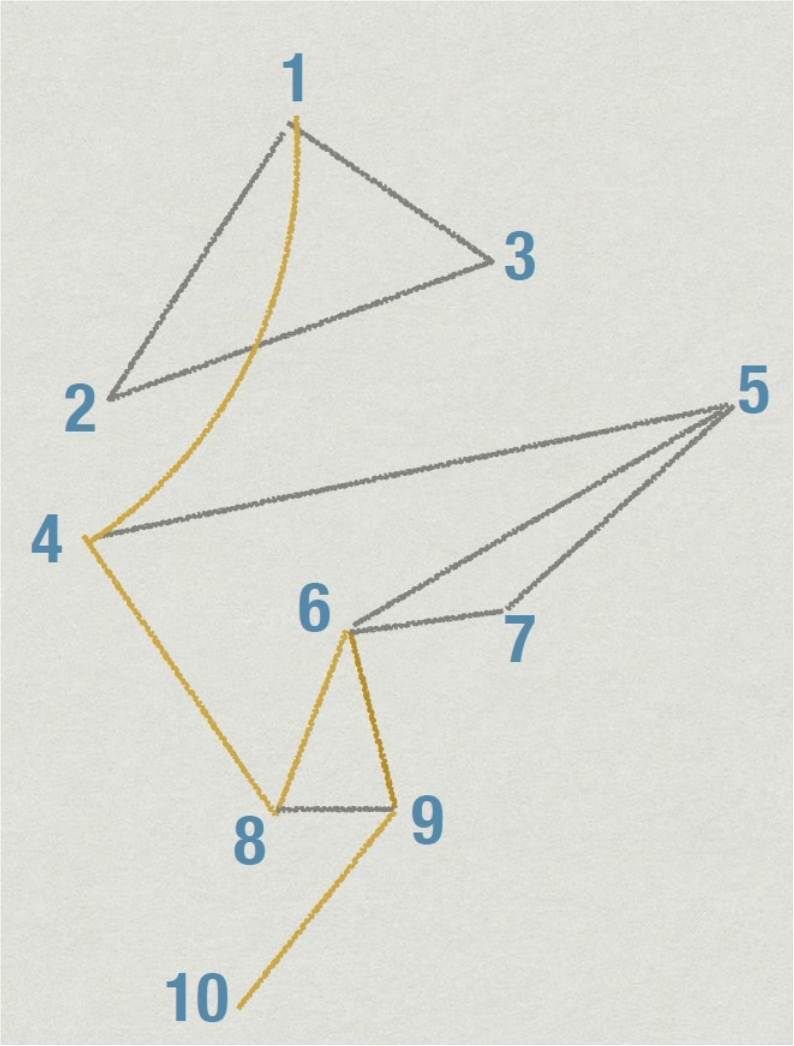
Level 1: 6,8,10

Level 2: 7,5,4

Note that 6,8,10 neighbours are exhausted

Look for unvisited neighbours of 7,5,4

Breadth First Search



Start with Chennai (9).

Level 1: 6,8,10

Level 2: 7,5,4

Note that 6,8,10 neighbours are exhausted

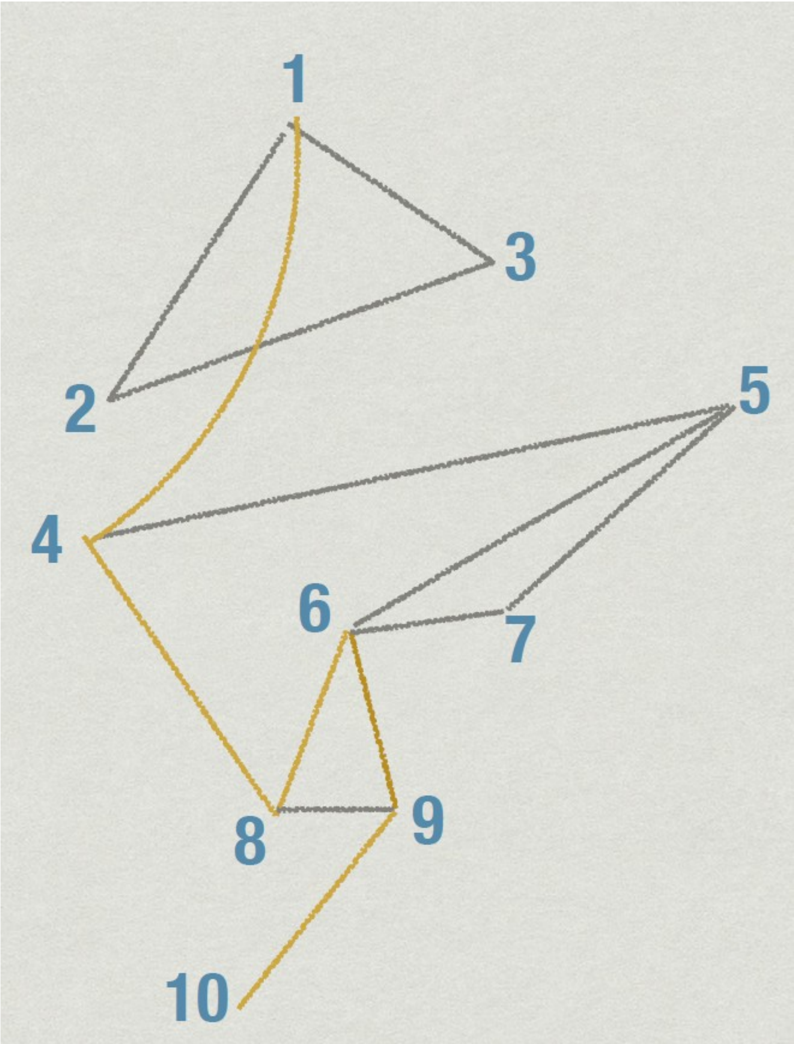
Look for unvisited neighbours of 7,5,4

Level 3: 1

Exploring a graph using BFS

- In *Breadth First Search (BFS),* the graph is explored level by level starting from any vertex.
  - First visit the vertex
  - Then visit the vertices that are one step away
  - Then visit vertices two steps away ....
- Remember which vertices have been visited in an array called visited (visited[i] = true when i is visited)
- We also have to keep track of visited vertices whose neighbours have still to be explored. Keep a queue Q for this.

Breadth First Search



Start with Chennai (9).

Level 1: 6,8,10

Level 2: 7,5,4

Level 3: 1

Level 4: 2,3 (Varanasi)

Queue Q:

{**9**} → {**6**,8,10} → {**8**,10, 7,5} → {**10**,7,5,4} → {**7**,5,4} → {**5**,4} → {**4**} → {**1**} → {**2**,3}

BFS traversal

```
function BFS(i) // BFS starting from vertex i

   //Initialization
   for j = 1..n {visited[j] = 0}; Q = []
```

Start with empty Q and by setting visited = false for all vertices

BFS traversal

```
function BFS(i) // BFS starting from vertex i

   //Initialization
   for j = 1..n {visited[j] = 0}; Q = []

   //Start the exploration at i
   visited[i] = 1; append(Q,i)
```

Append the start node i to the queue and set it as visited

BFS traversal

```
function BFS(i) // BFS starting from vertex i

  //Initialization
  for j = 1..n {visited[j] = 0}; Q = []

  //Start the exploration at i
  visited[i] = 1; append(Q,i)

  //Explore each vertex in Q
  while Q is not empty
     j = extract_head(Q)
     for each (j,k) in E
        if visited[k] == 0
           visited[k] = 1; append(Q,j)
```

Extract head of Queue and explore its neighbours

BFS traversal

```
function BFS(i) // BFS starting from vertex i

  //Initialization
  for j = 1..n {visited[j] = 0}; Q = []

  //Start the exploration at i
  visited[i] = 1; append(Q,i)


  //Explore each vertex in Q
  while Q is not empty
     j = extract_head(Q)
     for each (j,k) in E
        if visited[k] == 0
           visited[k] = 1; append(Q,j)
```

Stop when queue is empty

Complexity of BFS

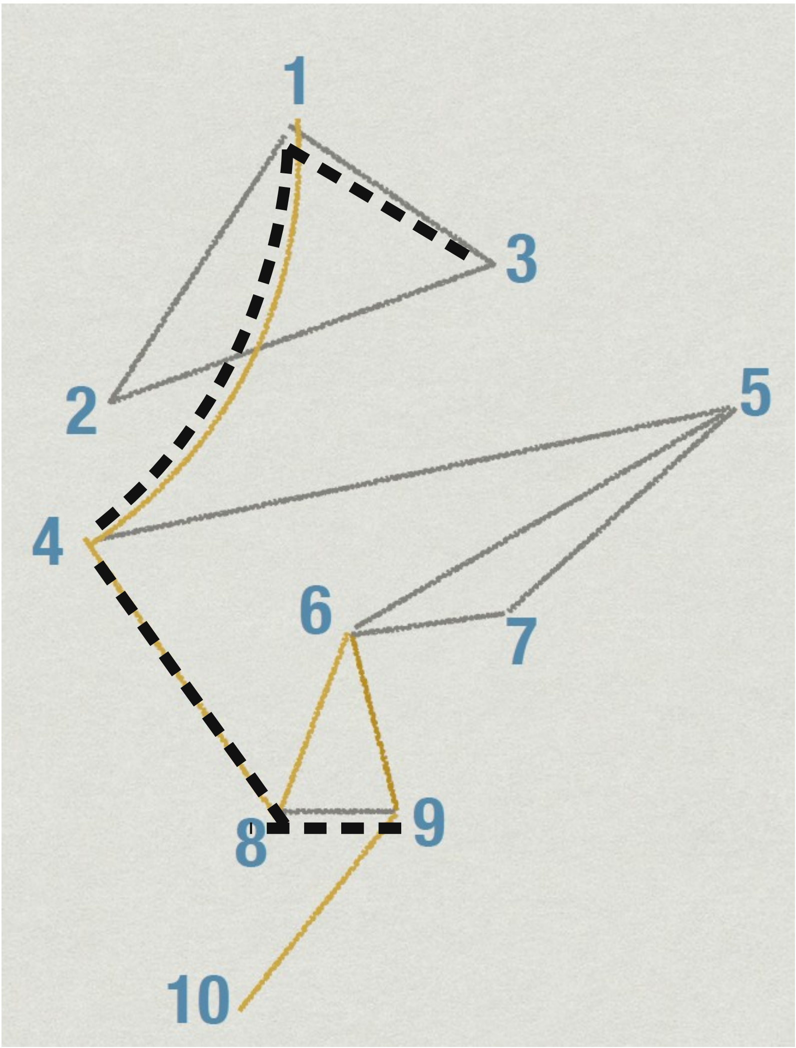
- Each vertex enters Q exactly once
- If the graph is connected, loop to process Q will iterate n times. At each loop step:
  - For each j extracted from Q, examine all its neighbours.
  - For **adjacency matrix**, this takes O(n) time. So overall, $O(n^2)$

Complexity of BFS

- Each vertex enters Q exactly once
- If the graph is connected, loop to process Q will iterate n times. At each loop step:
  - For each j extracted from Q, examine all its neighbours. For adjacency list, depends on number of neighbours. Difficult to count no of operations.
- Note: each edge (i,j) visited exactly twice – once when visiting i and once when visiting j.
  - So examining all edges takes O(m) time !
  - However each node needs to be marked: O(n) time
  - So overall complexity for adjacency list is O(m+n)

Applications of Breadth First Search

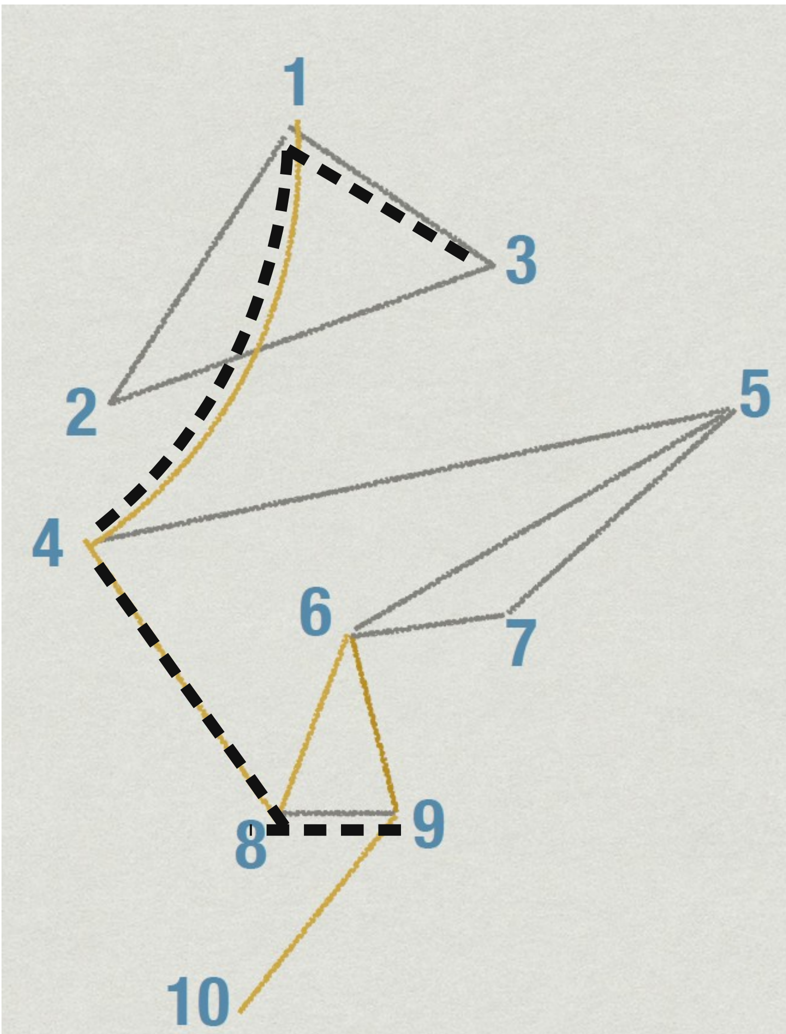
How do we find the path ?



Queue Q:

{9} → {6,8,10} → {8,10, 7,5} → {10,7,5,4} → {7,5,4} → {5,4} → {4} → {1} → {2,3}

How do we find the path ?



When visiting a vertex, record how we came to this vertex

parent[8] = 9, parent[4]=8, parent[1]=4, parent[3]=1.

When we hit the desired destination, (in this example 3), we can trace the parent chain back to source (9)

3 → 1 → 4 → 8 → 9

Queue Q:
{9} → {6,8,10} → {8,10, 7,5} → {10,7,5,4} → {7,5,4} → {5,4} → {4} → {1} → {2,3}

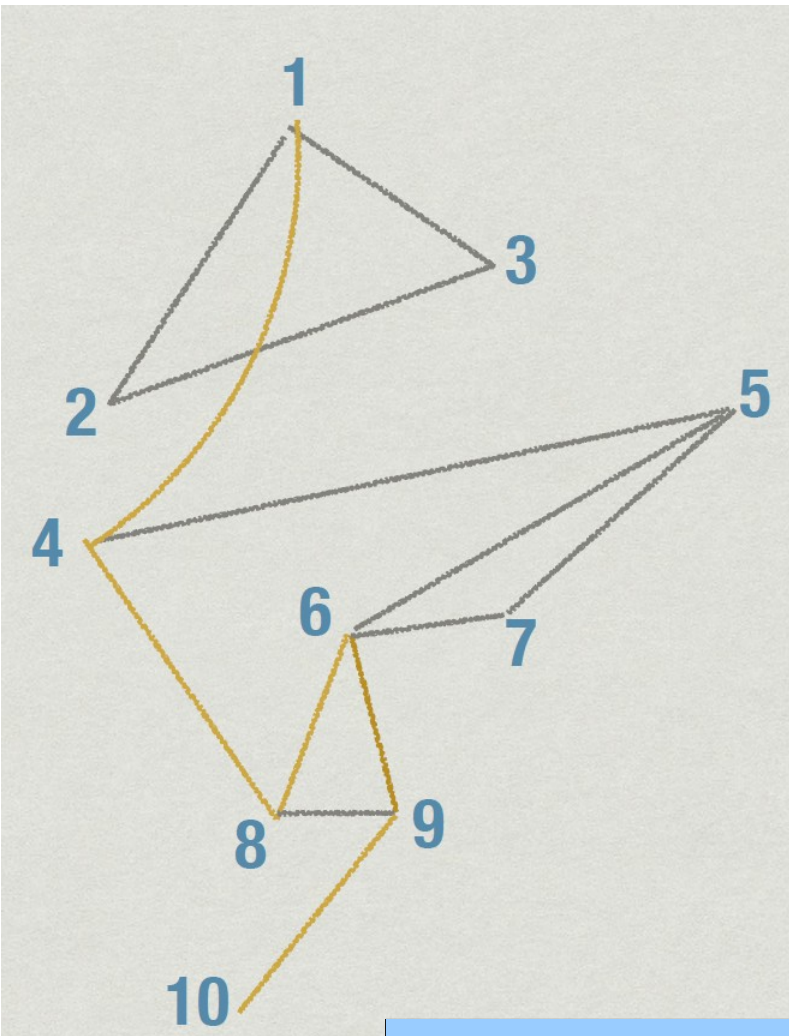
Tracing path to source

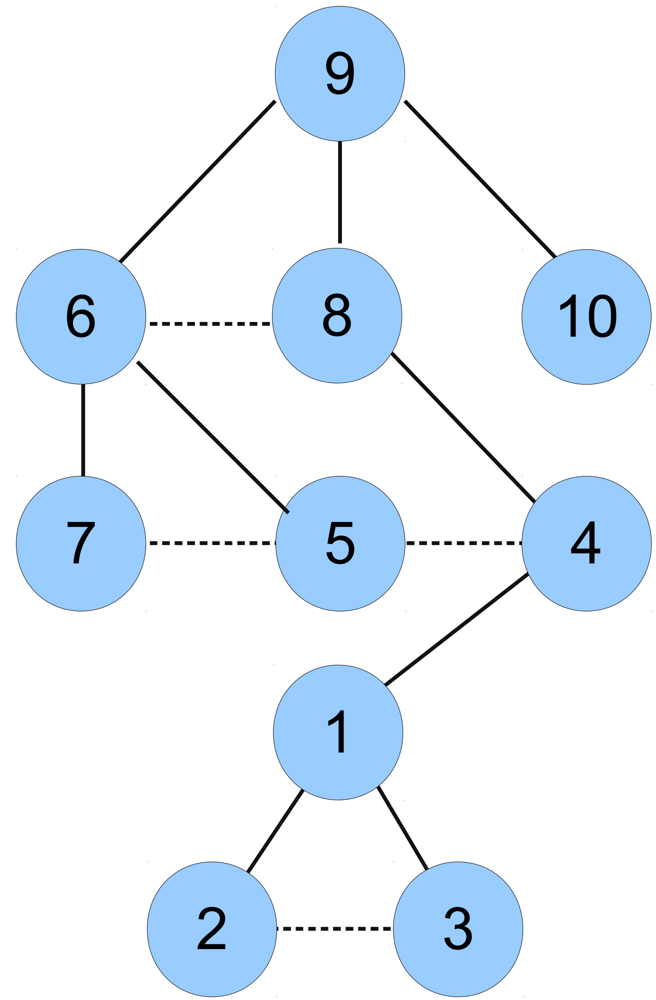
```
function BFS(i) // BFS starting from vertex i

    //Initialization
    for j = 1..n {visited[j] = 0; parent[j] = -1}
    Q = []

    //Start the exploration at i
    visited[i] = 1; append(Q,i)

    //Explore each vertex in Q
    while Q is not empty
        j = extract_head(Q)
        for each (j,k) in E
            if visited[k] == 0
                visited[k] = 1; parent[k] = j; append(Q,j);
```

How do we check if a graph is connected?

- A graph is said to be *connected* if there is a path between every pair of vertices
- Can we use BFS to check if a graph is connected?

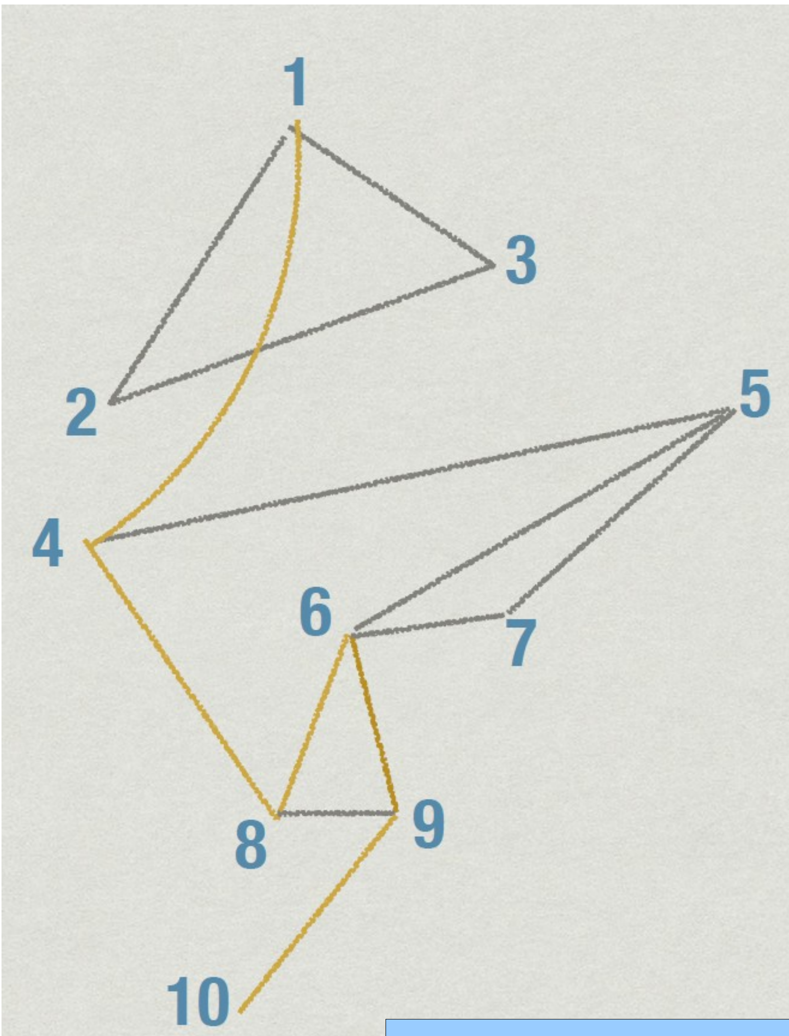
The BFS traversal structure

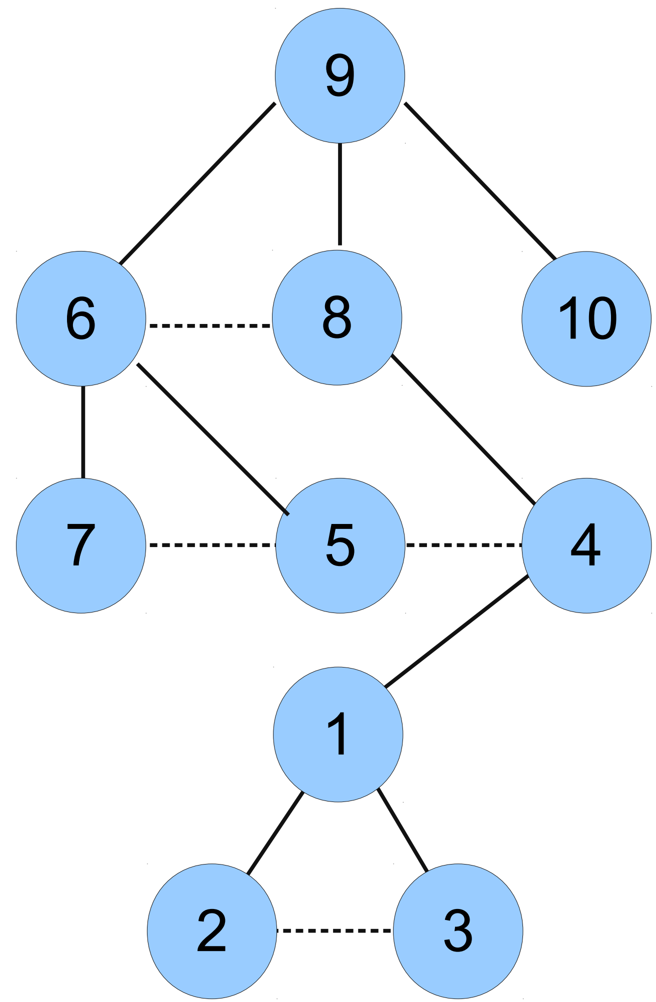




Dashed line when a visited node's neighbour is also visited

The BFS traversal structure

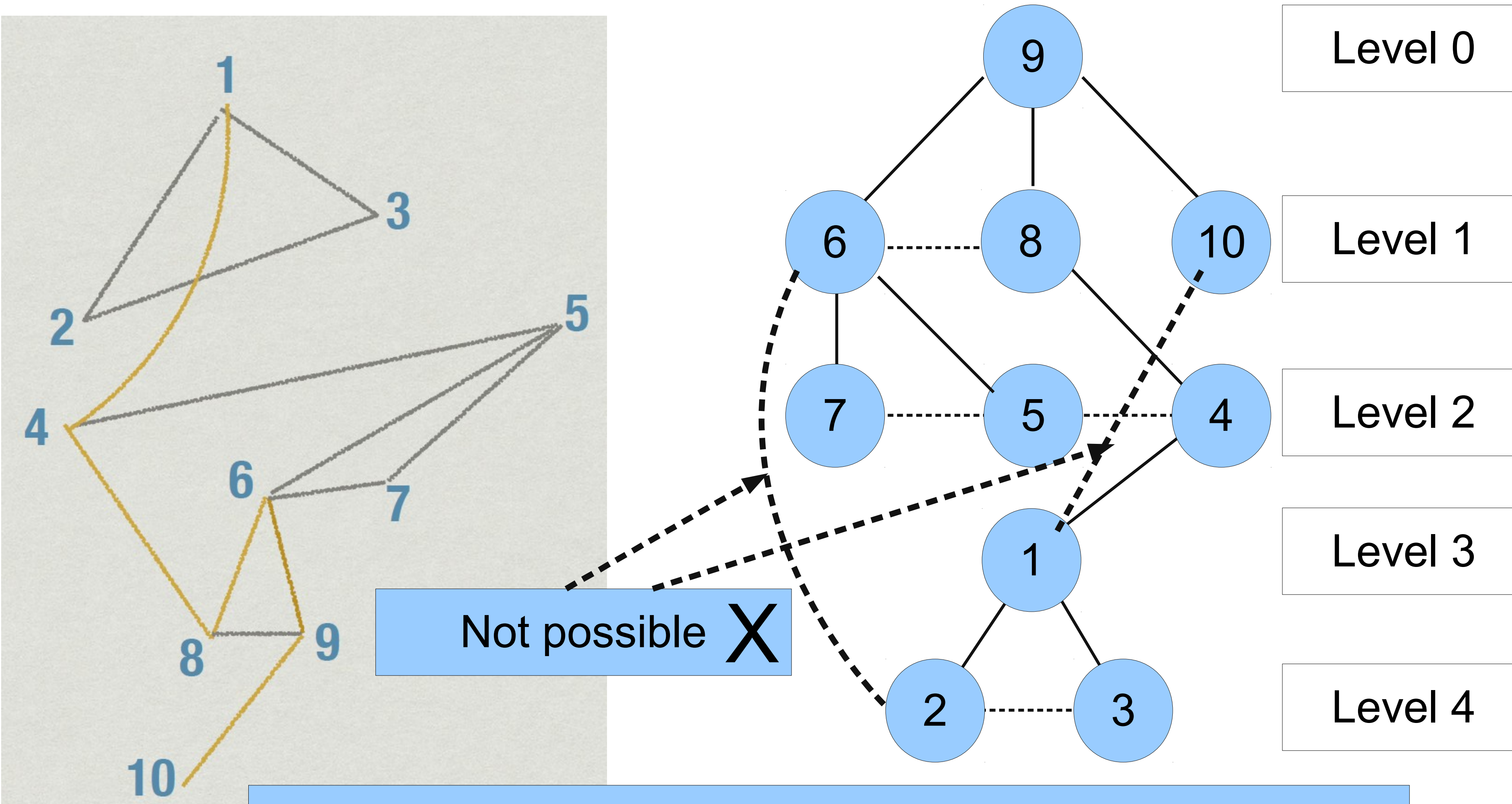




Dashed line when a visited node's neighbour is also visited

Note that a dashed line can be inside one level or to next level only

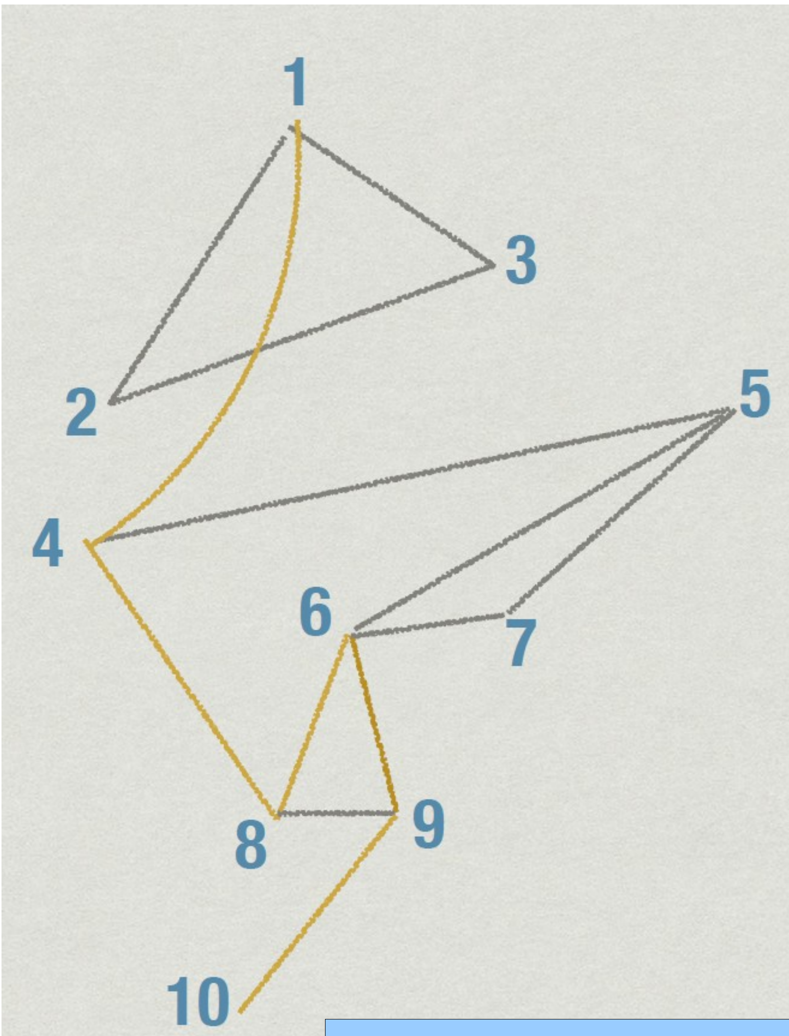
The BFS traversal structure

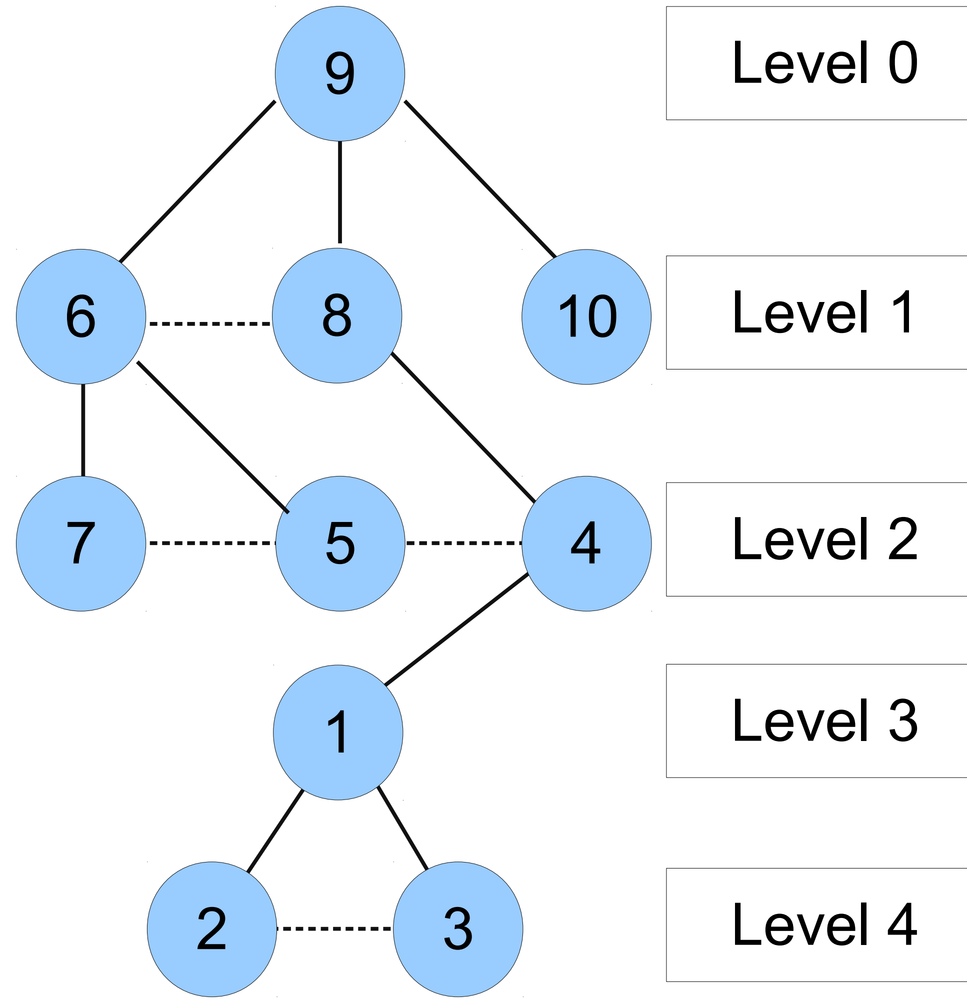


If those edges existed, then 1 and 2 would be at level 2

Note that a dashed line can be inside one level or to next level only

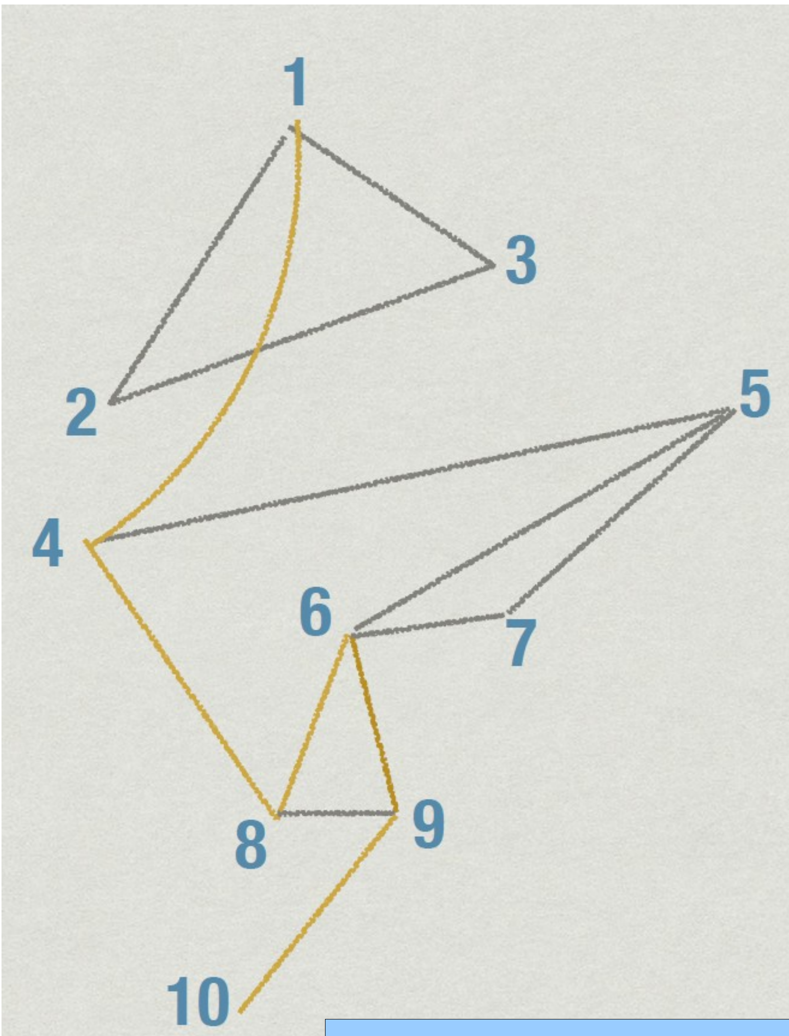
BFS can discover (odd) cycles

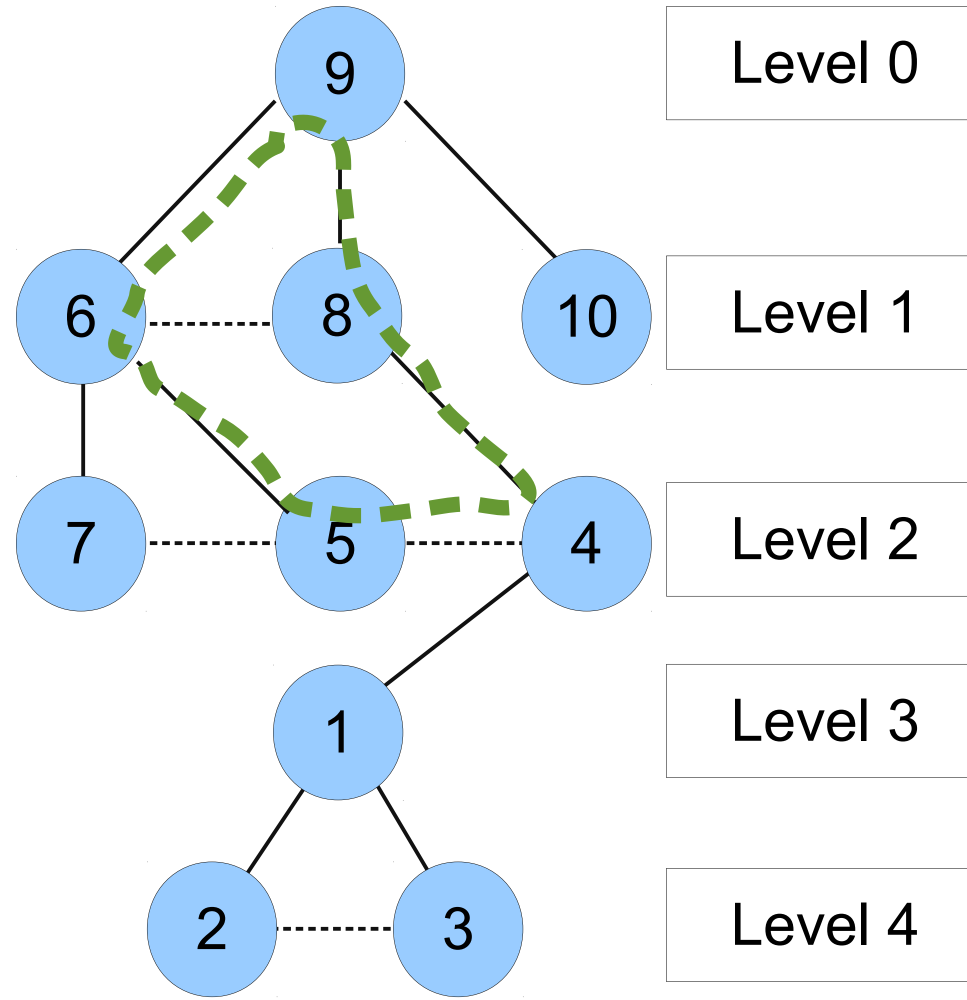




Each dashed line contributes to a cycle

BFS can discover (odd) cycles



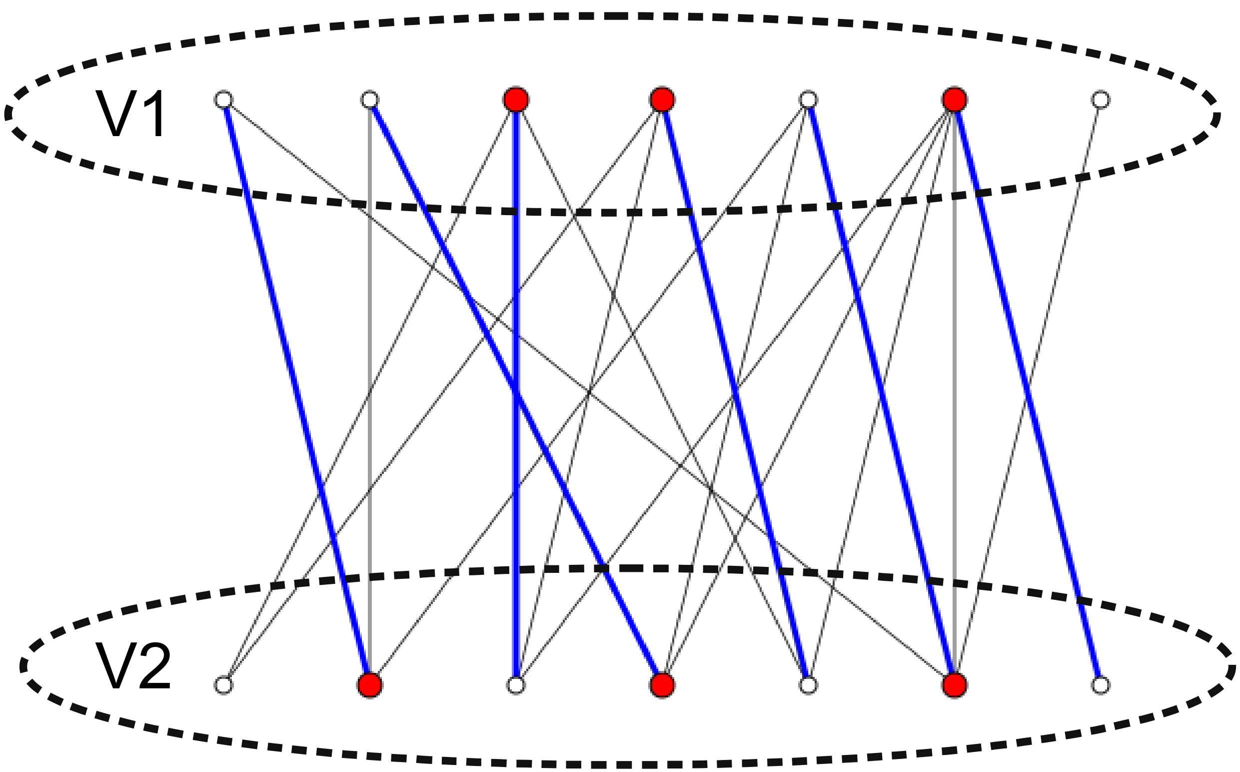


Each dashed line contributes to a cycle

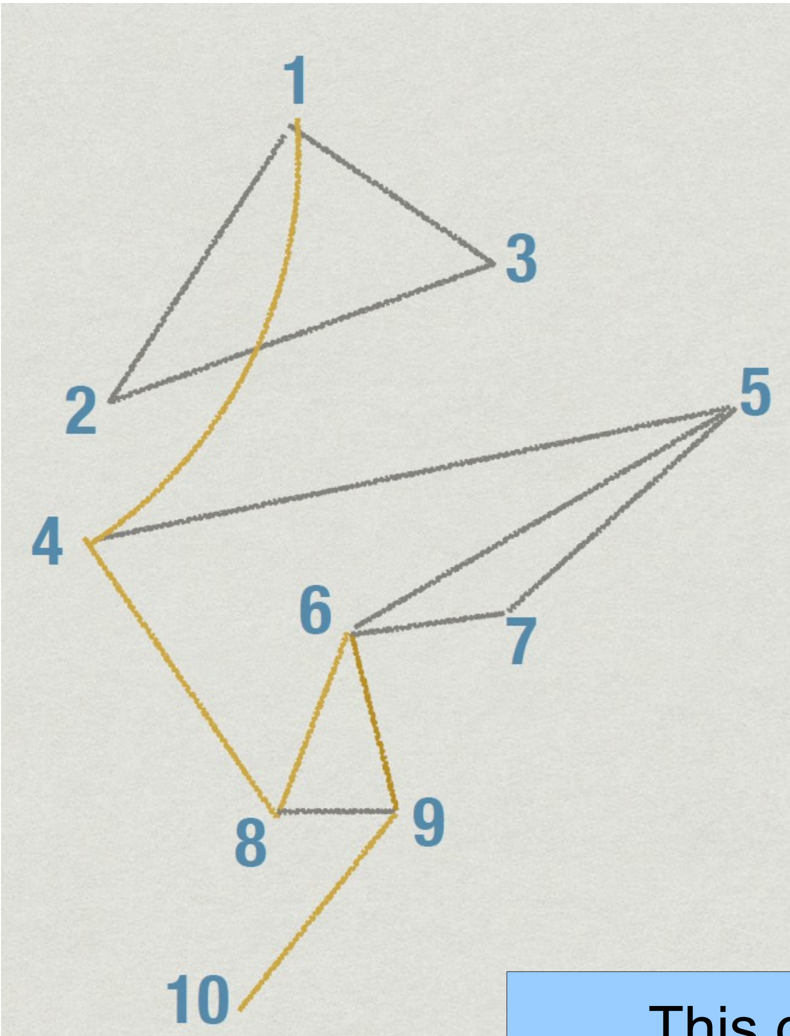
If dashed edge is within one level, then the graph has ***odd cycles***

Checking if a graph is bipartite

- A graph is called a *bipartite graph* if its vertices can be partitioned into two subsets V1 and V2, such that all its edges lie only between V1 and V2.
  - i.e. there are no edges within V1 and within V2



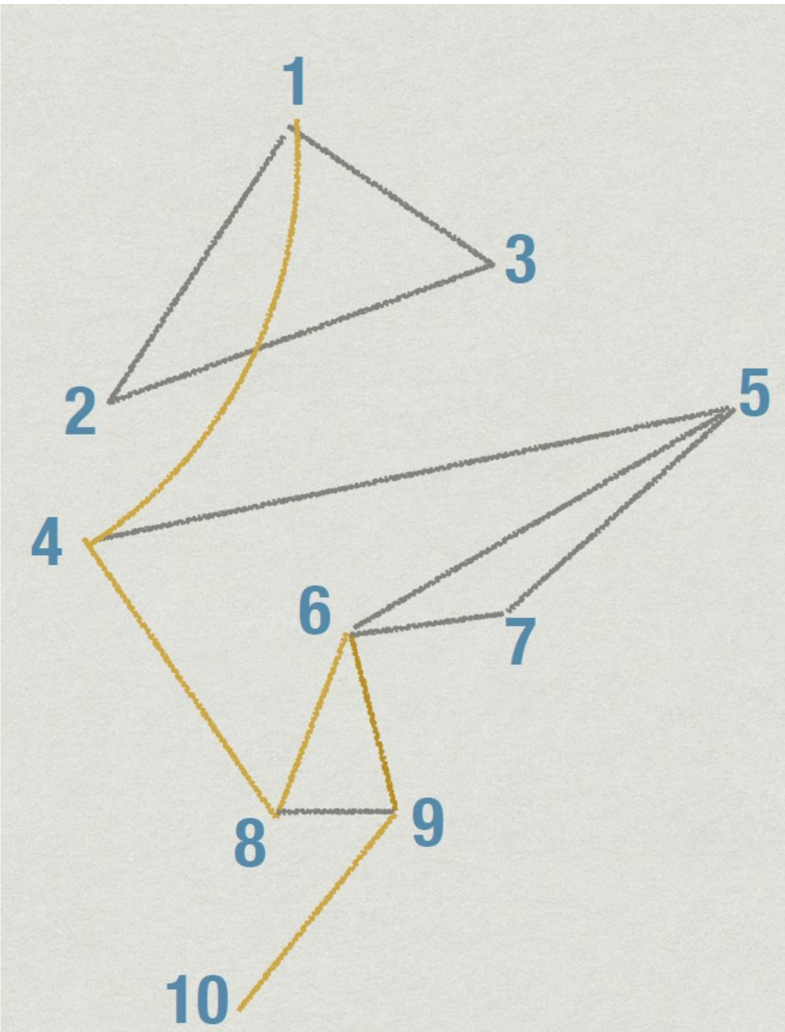
Checking if a graph is bipartite

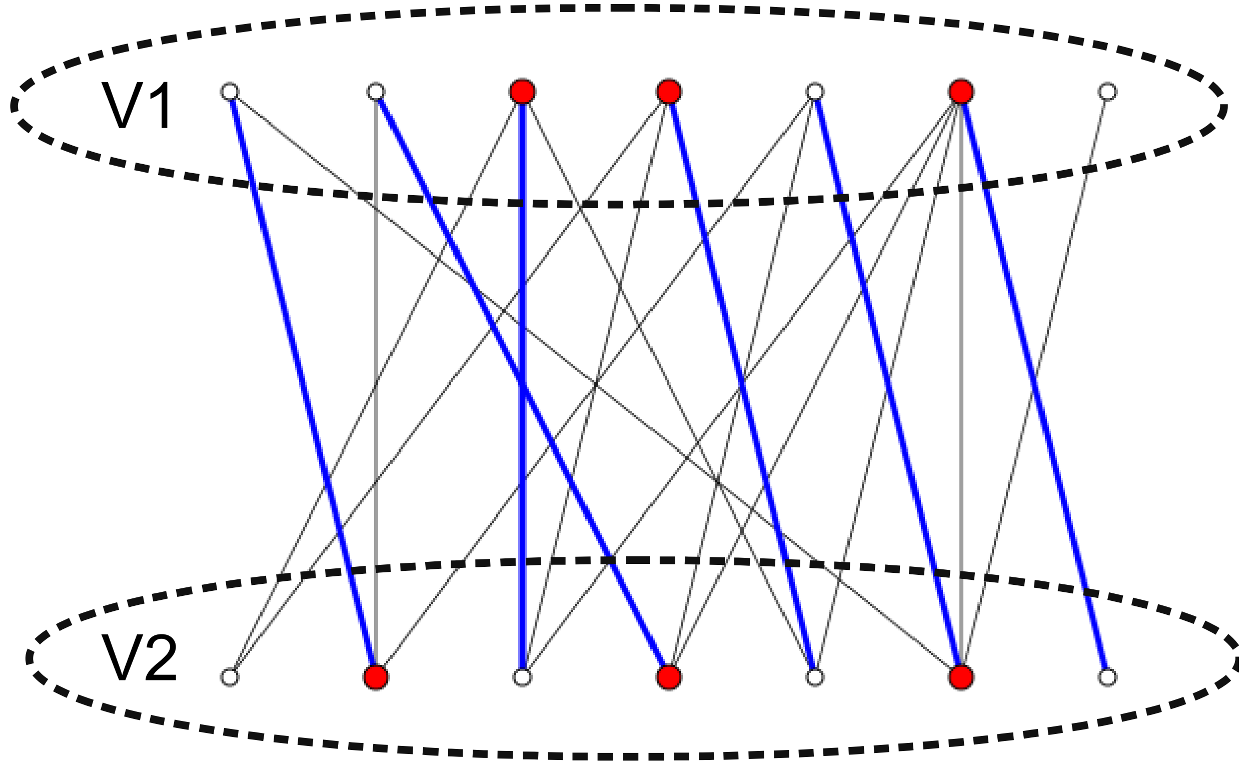


- A graph is called a *bipartite graph* if its vertices can be partitioned into two subsets V1 and V2, such that all its edges lie only between V1 and V2.
  - i.e. there are no edges within V1 and within V2

This graph is NOT bipartite

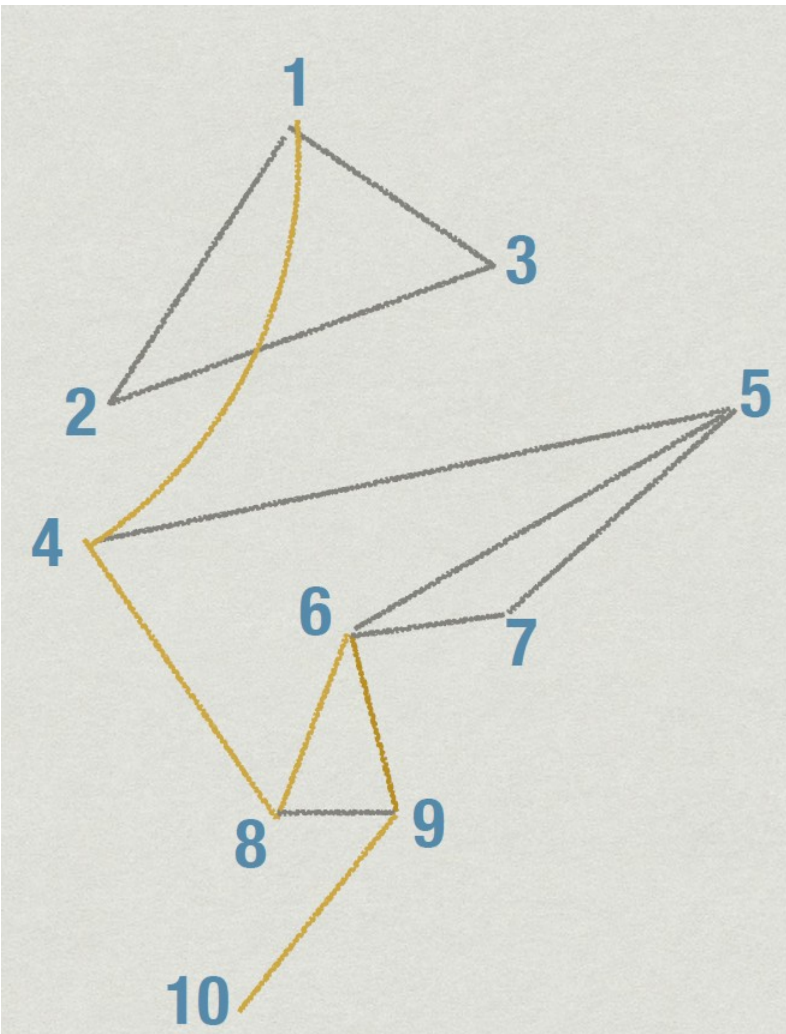
Checking if a graph is bipartite

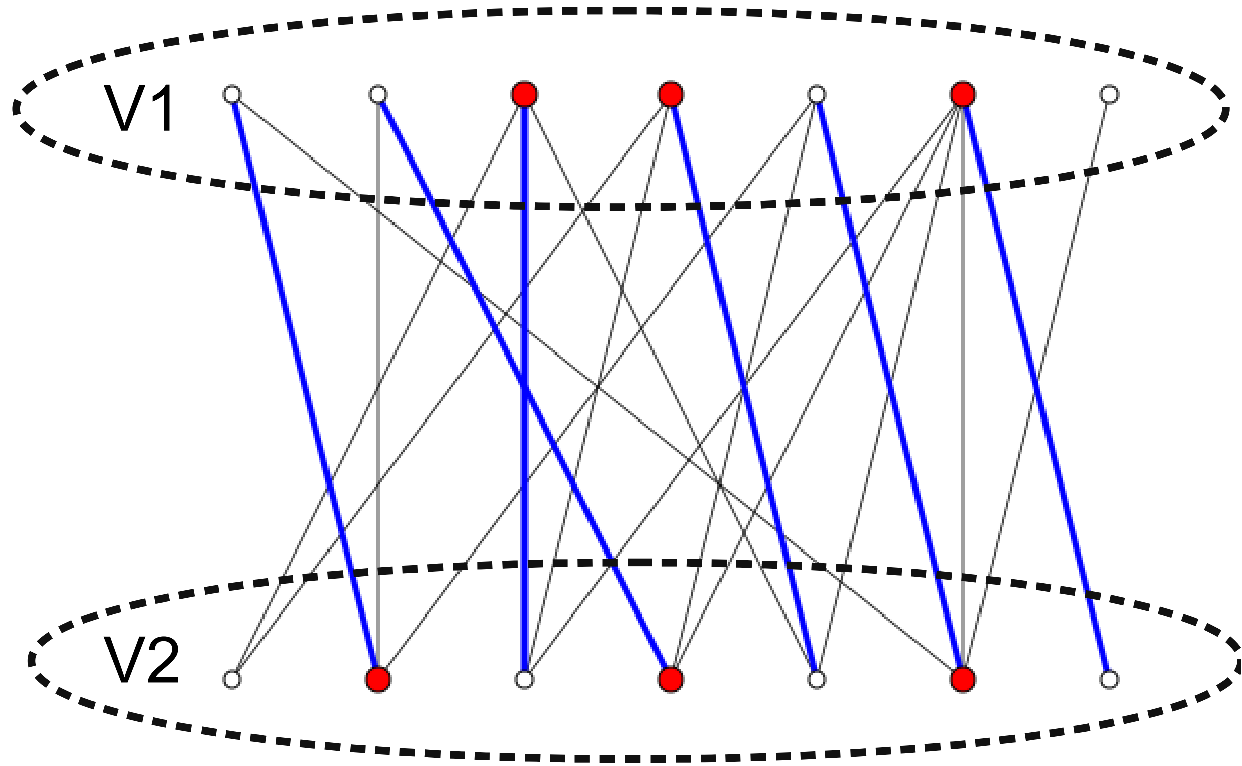




Notice something? A bipartite graph does not have any ***odd cycles***

Checking if a graph is bipartite

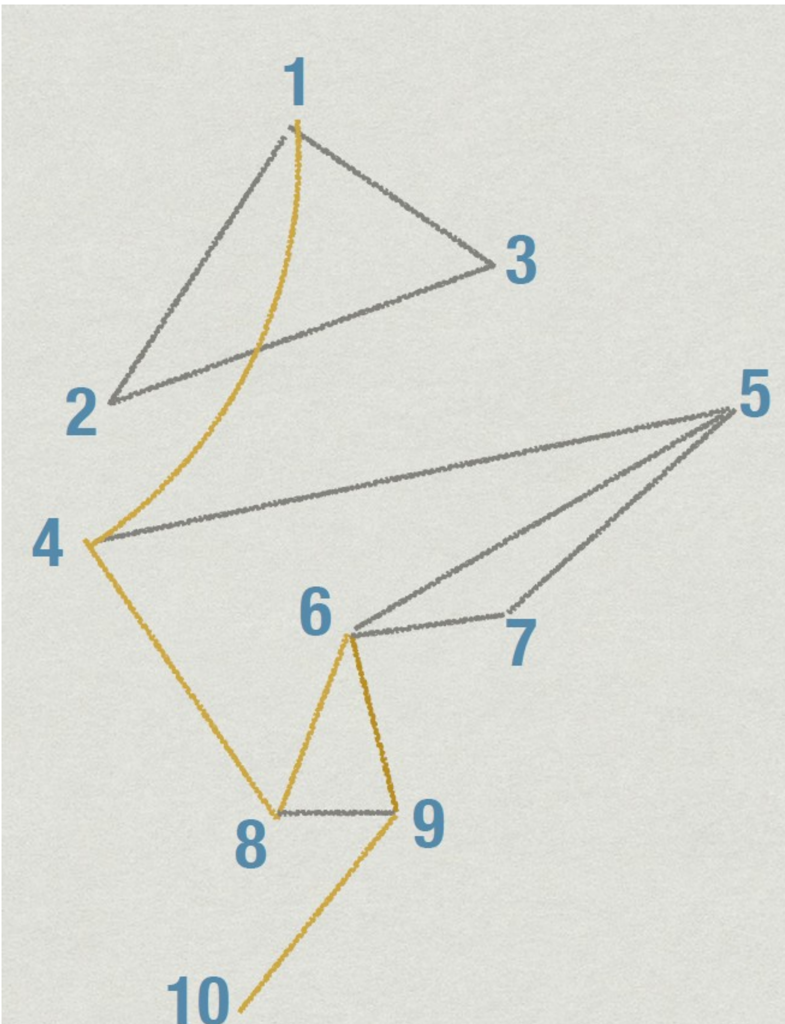


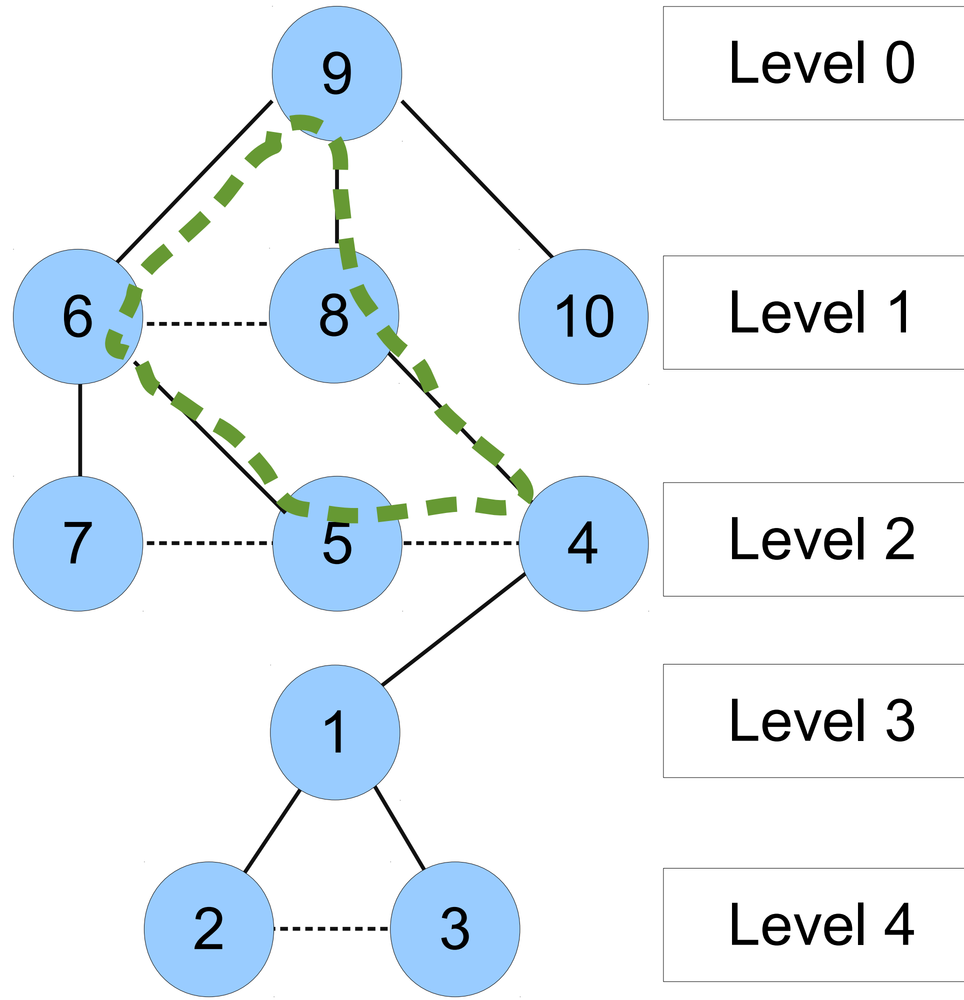


Notice something? A bipartite graph does not have any ***odd cycles***

A graph is bipartite if and only if it has no odd cycles

BFS can discover (odd) cycles



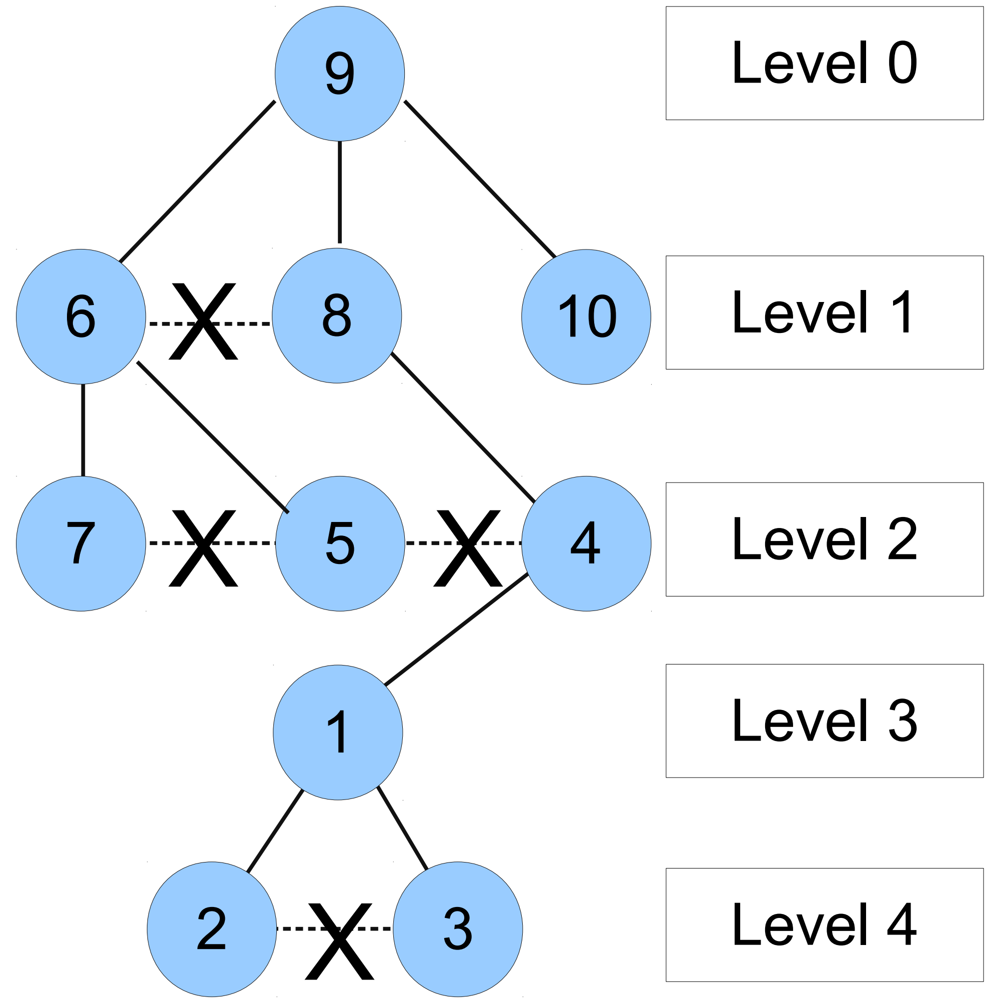


If dashed edge is within one level, then the graph has ***odd cycles***

BFS can discover (odd) cycles

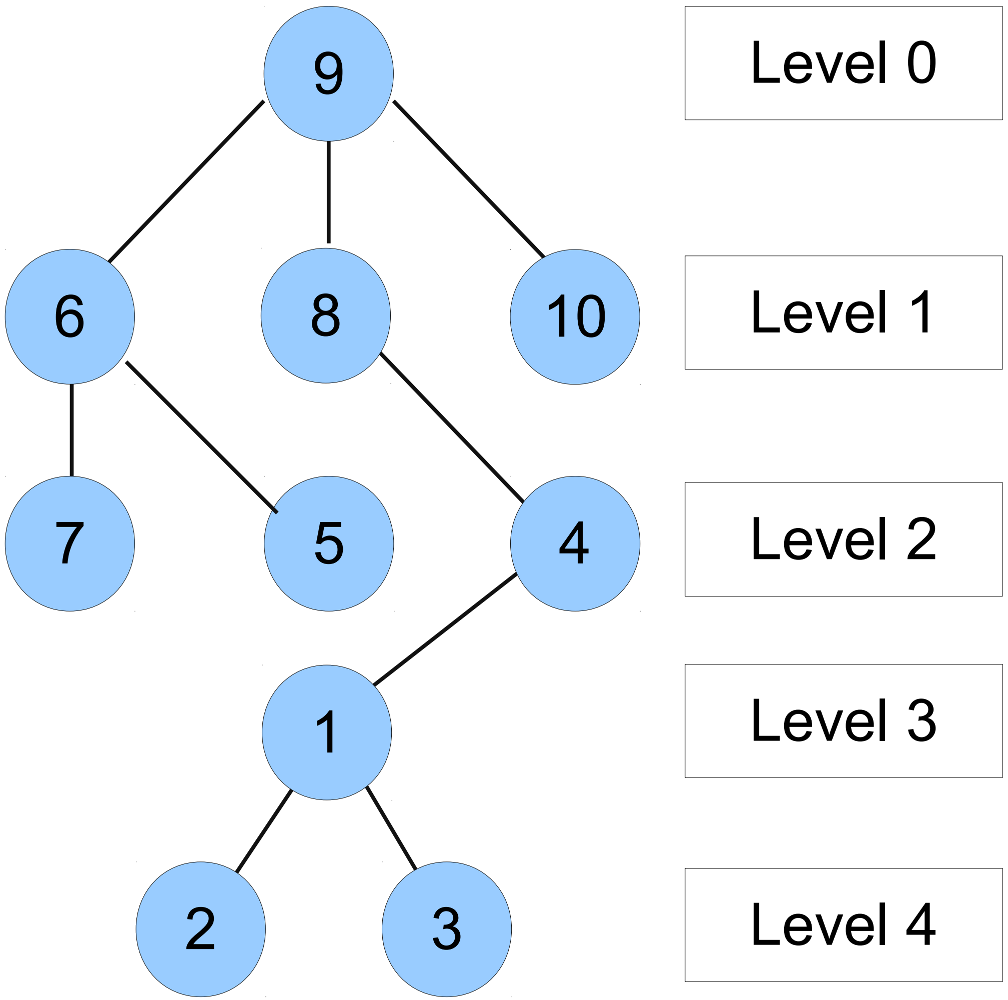
If dashed edge is within one level, then the graph has ***odd cycles***

If BFS traversal of a graph does not find edges within the same level, then graph is bipartite



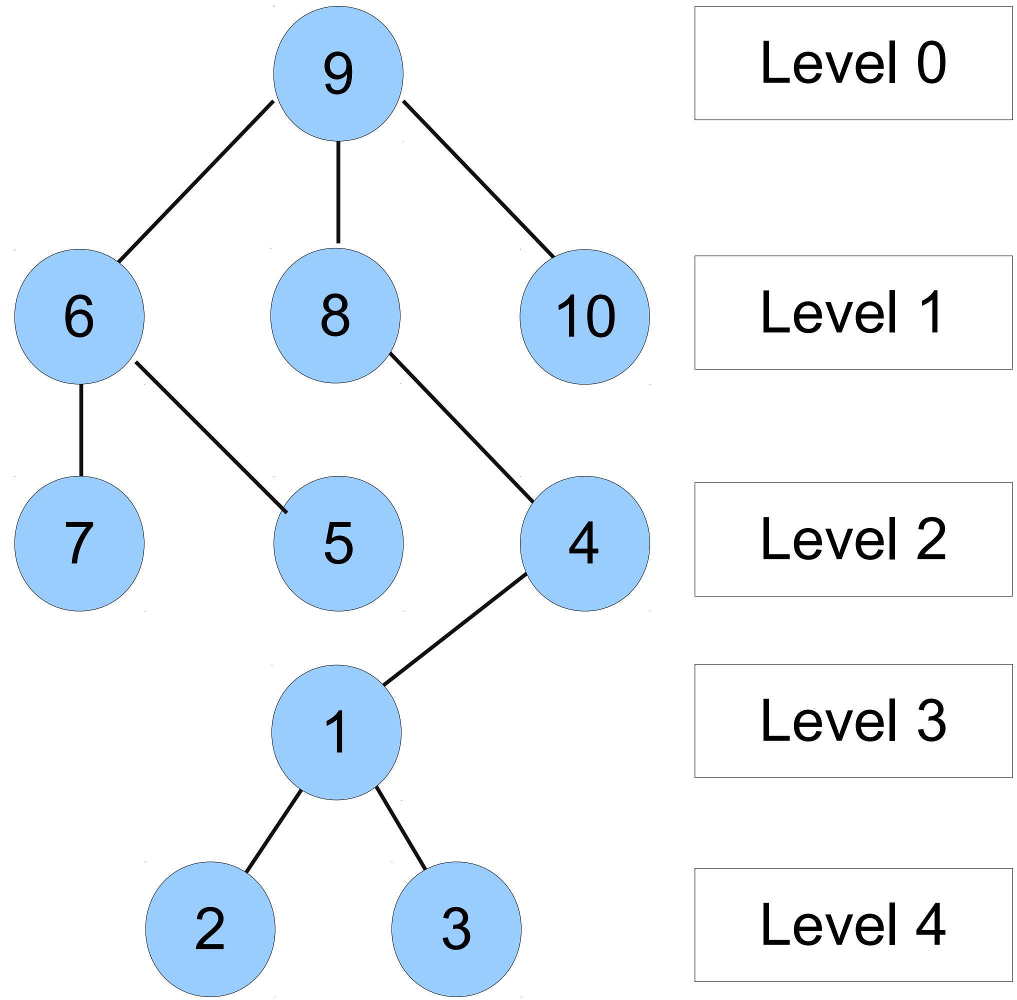
Constructing the partitions

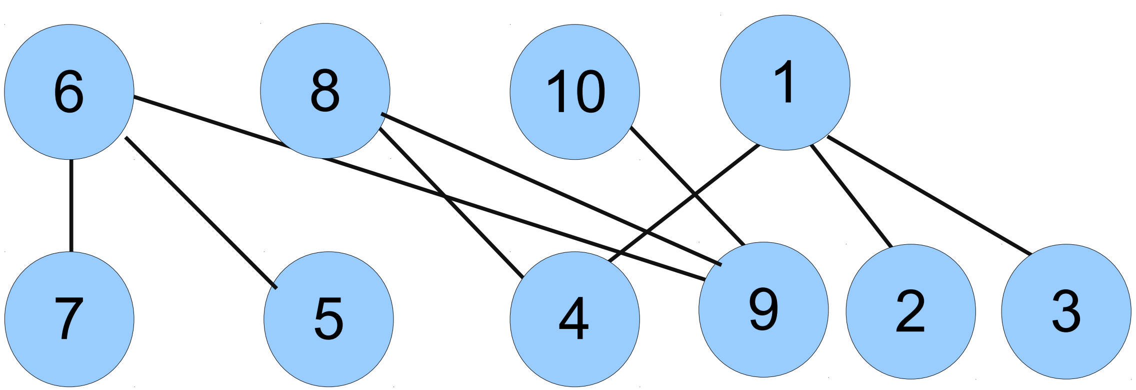
All the vertices at odd levels form one partition V1 while those at the even levels form the other partition V2



Constructing the partitions

All the vertices at odd levels form one partition V1 while those at the even levels form the other partition V2





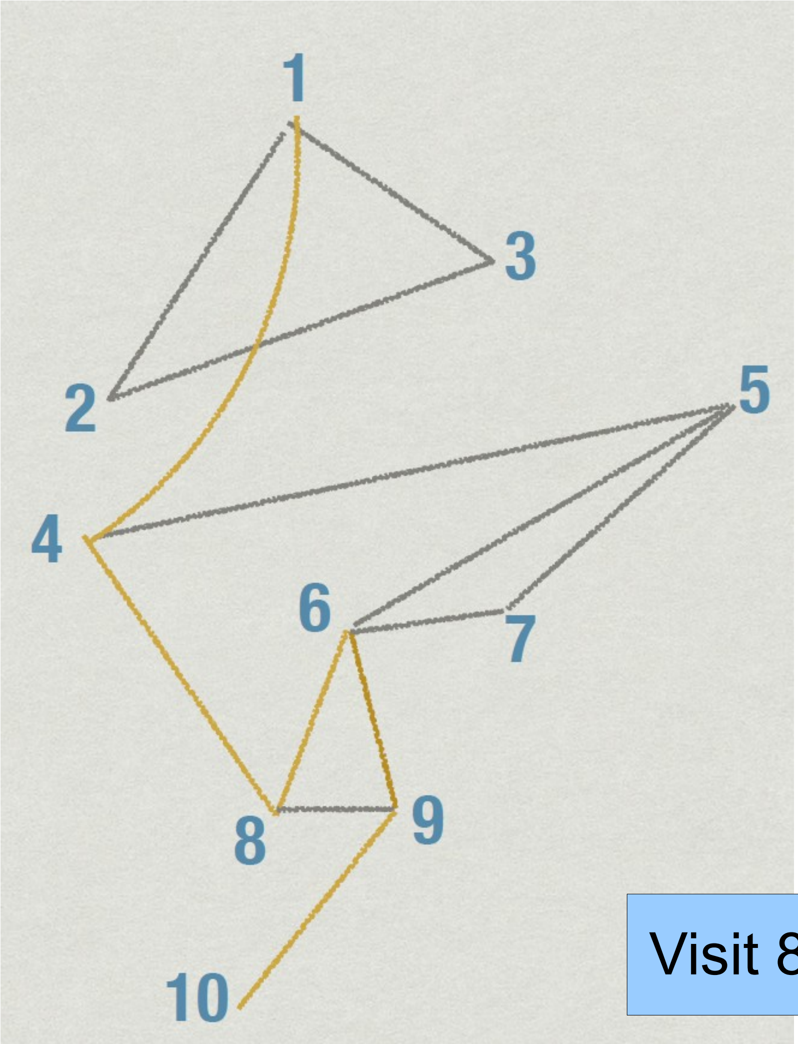
Graph traversal:

Depth First Search

Exploring a graph using DFS

- In *Depth First Search (DFS),* the graph is explored starting from any vertex by going down one direction till it can go no further. When it hits an end, it backtracks and tries another direction.
  - First visit the vertex
  - Then visit a neighbour of the vertex
  - Then visit a neighbour of that vertex ....
  - If there is no unvisited neighbour, then go back to parent vertex and try another neighbour from there

Depth First Search



Start with 9 as before. Mark it visited

Visit 6, pick its neighbour – say 7

Visit 7, pick its neighbour – only 5 is there

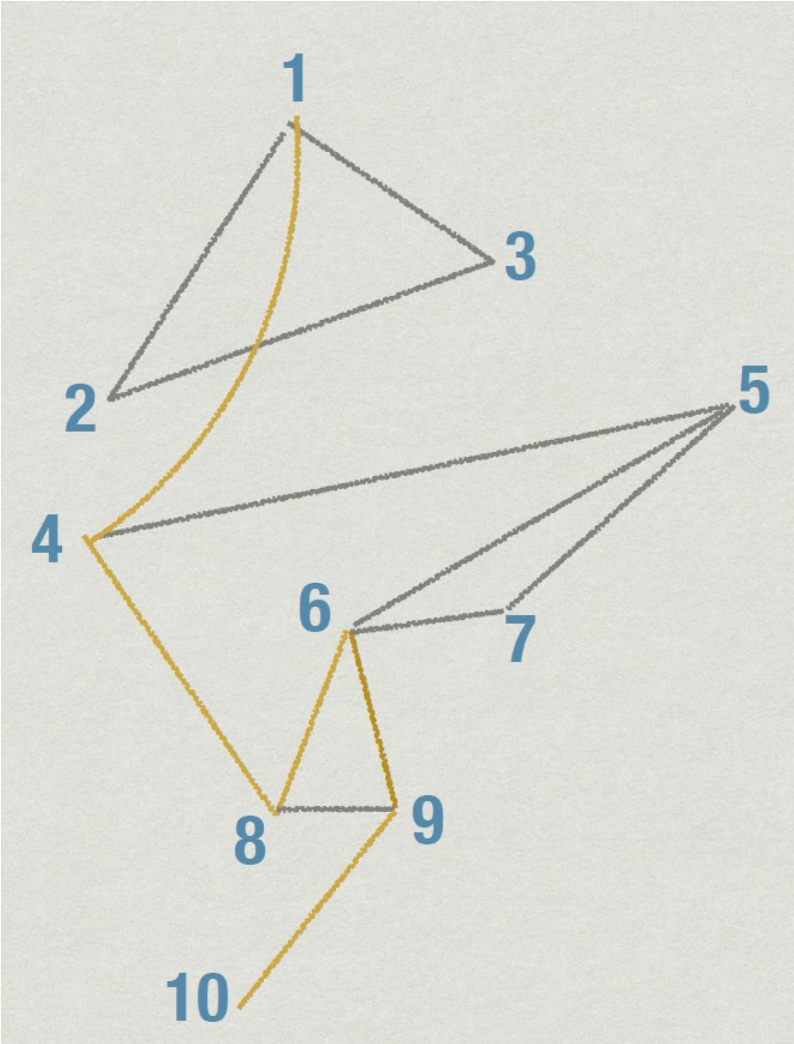
Visit 5. neighbours 6,7 are visited, so pick 4

Visit 4. pick a neighbour – say 8

Visit 8. All neighbours are visited ...go back where?

What data structure do we need to keep track of what to do when we go back?

Depth First Search and stacks

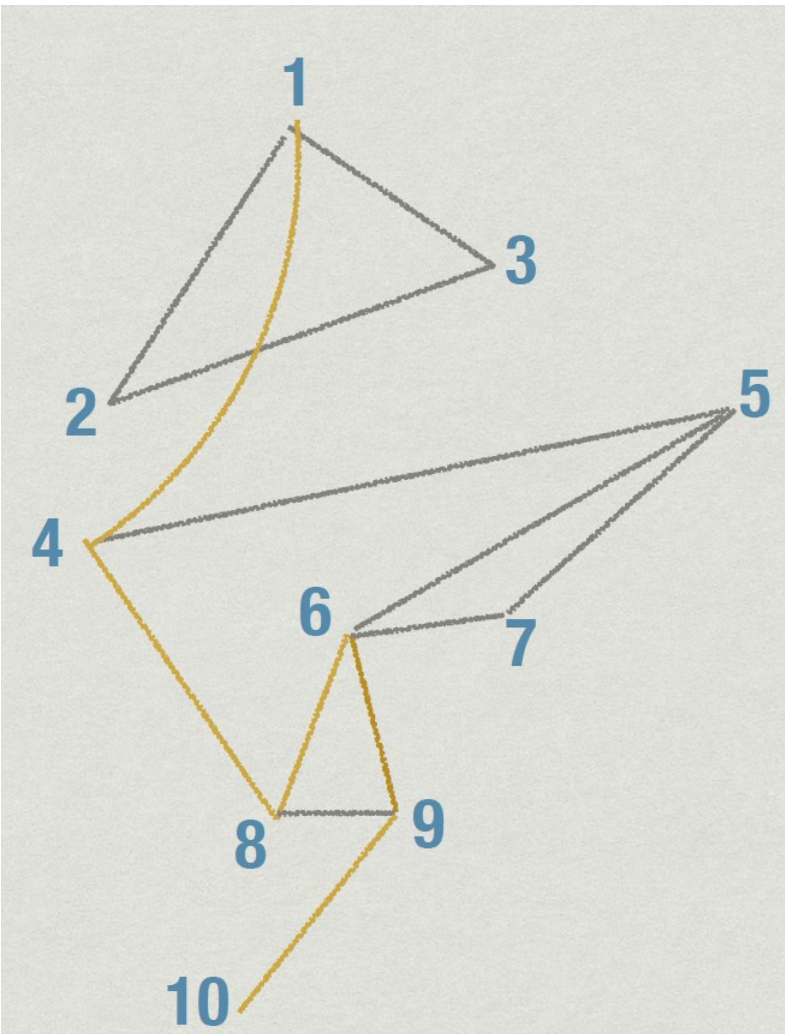


We need a stack !

Similar to returning from subroutines

Which is why DFS is best coded using recursion

Depth First Search and stacks



We need a stack !

Similar to returning from subroutines

Which is why DFS is best coded using recursion

How the stack develops in this example:
(**9**) → (**6**,8,10) → (**7**,5,8,10) → (**5**,5,8,10) → (**4**,5,8,10) → (**8**,1,5,8,10) → (**1**,5,8,10) → (**2**,3,5,8,10) → (**3**,3,5,8,10) → (3,5,8,10) → (5,8,10) → (8,10) → (**10**)

The marking of a vertex as visited is indicated by showing it in bold underlined

DFS traversal

```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}
```

Start by setting visited = false for all vertices and parent as unknown

DFS traversal

```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1
```

Call the recursive function DFS with vertex i, which first sets it as visited

DFS traversal

```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1

   //Explore each neighbour of i recursively
   for each (i,j) in E
      if visited[j] == 0
         parent[j] = i
         DFS(j)
```

Pick an unvisited neighbour of i and recursively call DFS for it

DFS traversal

```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1

   //Explore each neighbour of i recursively
   for each (i,j) in E
      if visited[j] == 0
         parent[j] = i
         DFS(j)
```

Stop when DFS exits

Complexity of DFS

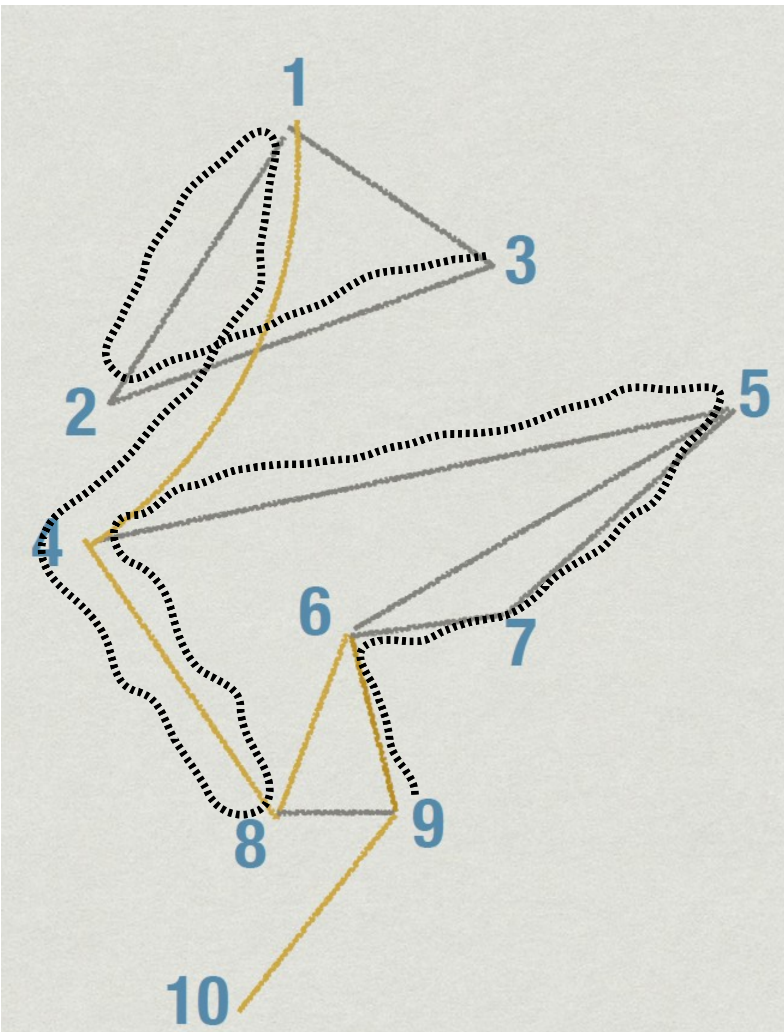
- Each vertex marked and explored exacty once
- DFS(j) will need to explore all neighbours of j
  - For **adjacency matrix**, this takes O(n) time.
  - So overall, $O(n^2)$

Complexity of DFS

- For adjacency list
  - Like in BFS, we count it differently
- Each edge (i,j) visited exactly twice – once when visiting i and once when visiting j.
  - So examining all edges takes O(m) time !
  - However each node needs to be marked: O(n) time
  - So overall complexity for adjacency list is O(m+n)

Applications of Depth First Search

How do we find the path ?



When visiting a vertex, record how we came to this vertex

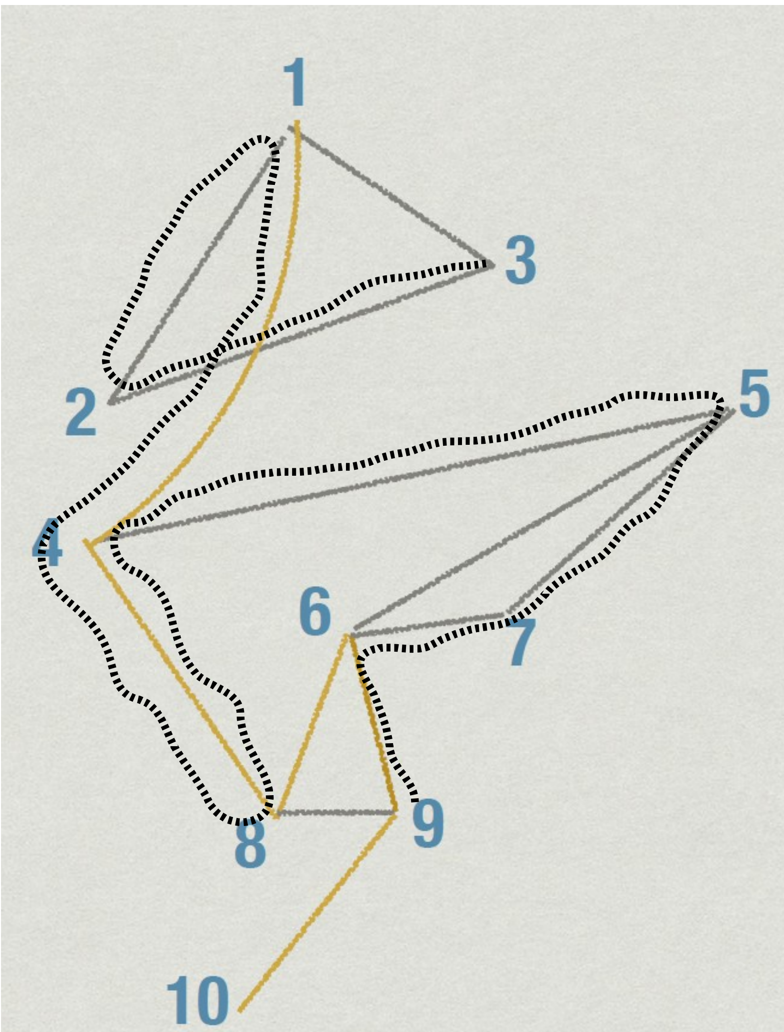
parent[6] = 9, parent[7]=6, parent[5]=7, parent[4]=5, parent[1] = 4, parent[2] = 1, parent[3] = 2

When we hit the desired destination, (in this example 3), we can trace the parent chain back to source (9)

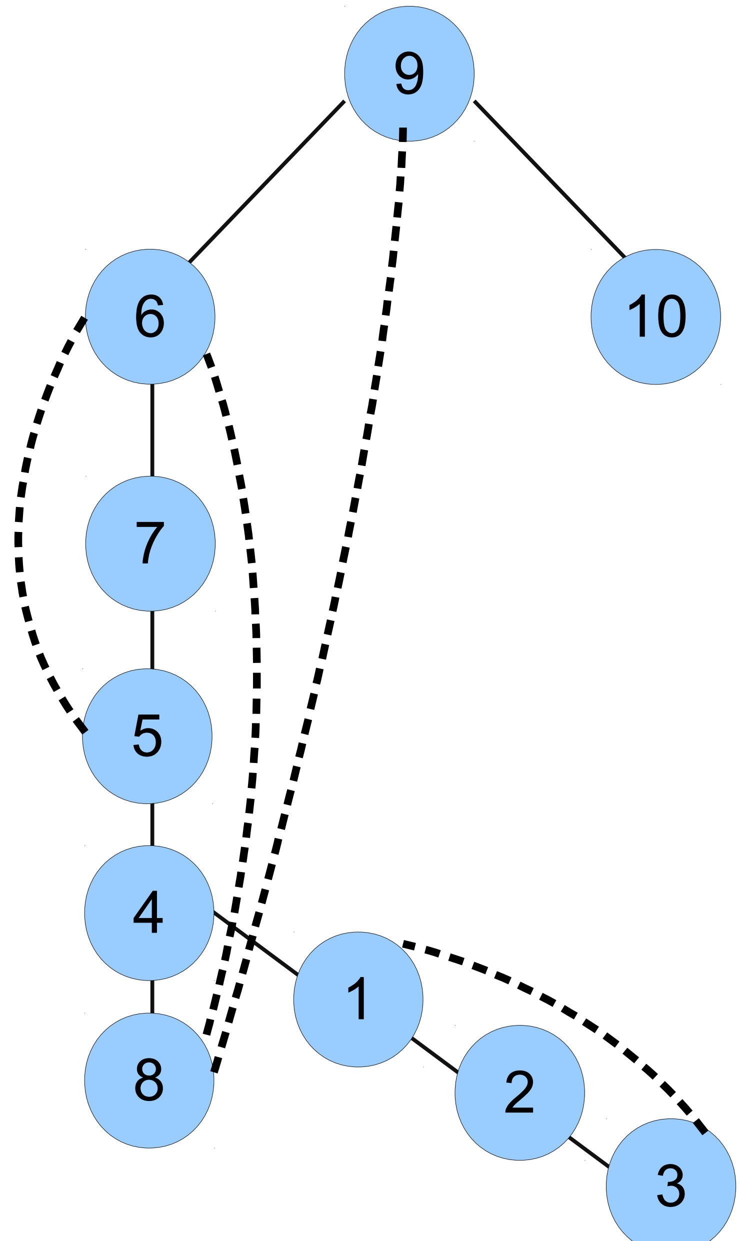
3 → 2 → 1 → 4 →5 →7 → 6 → 9

DFS finds a long winded path, then what use is it?

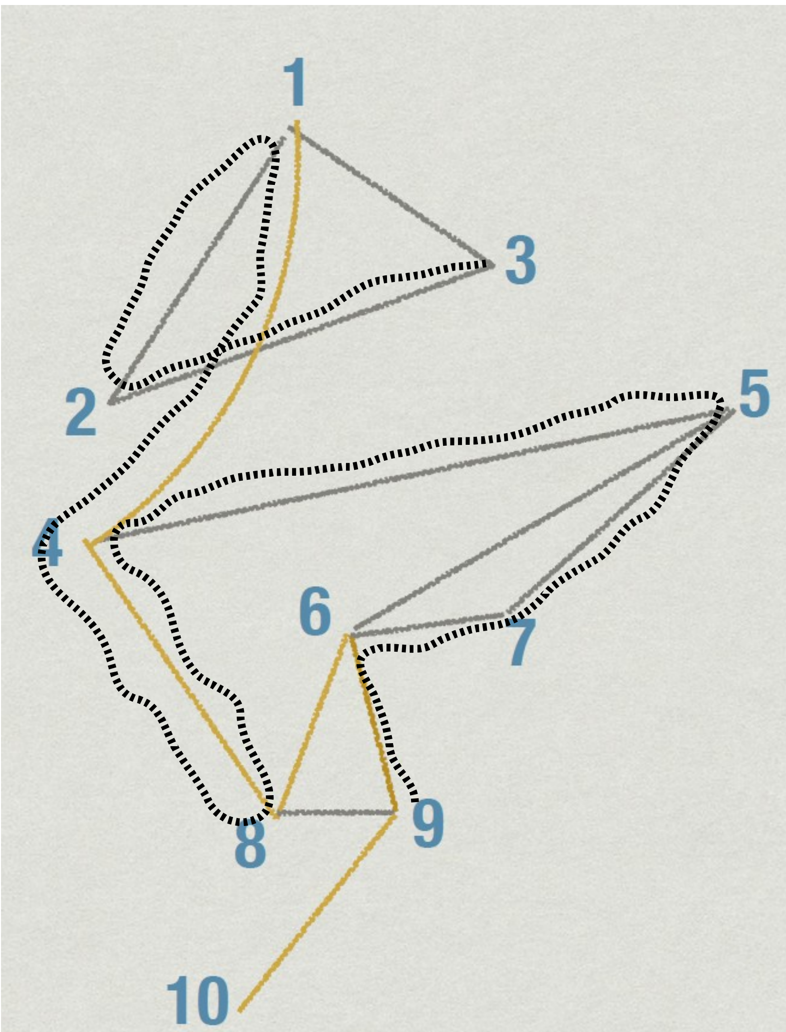
What is the DFS structure?



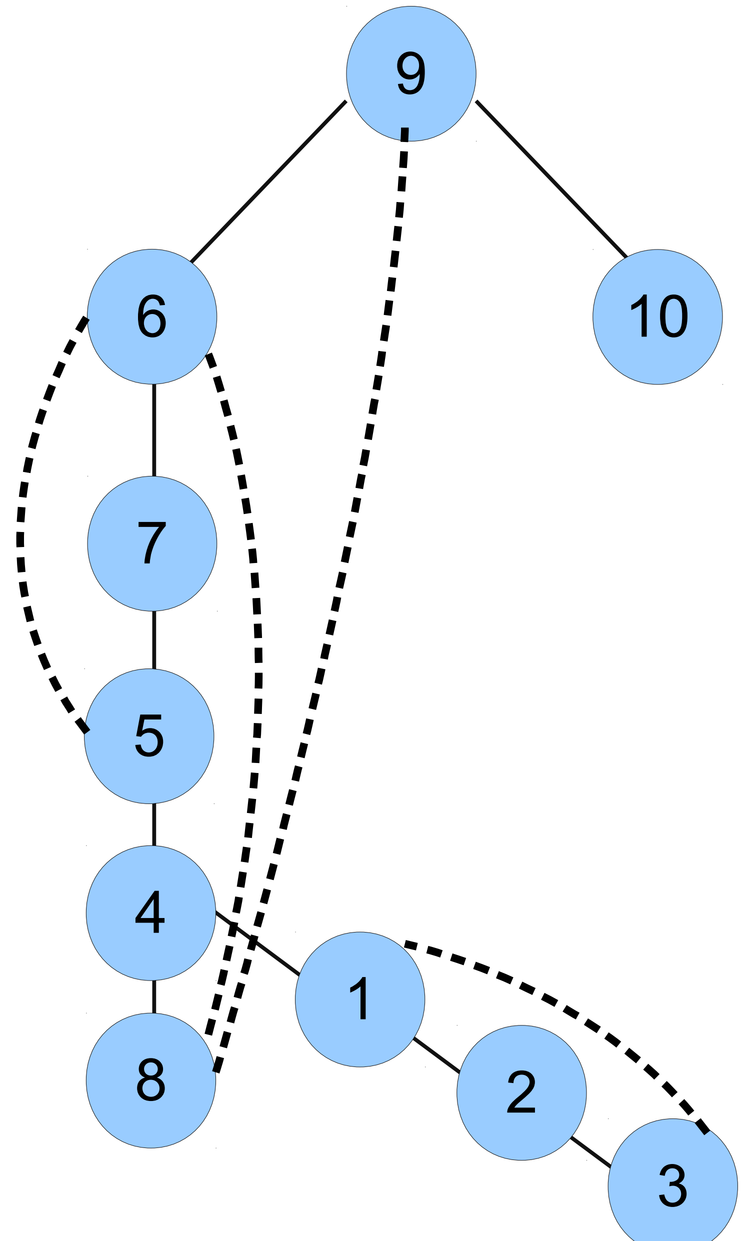
Dashed line when a visited node's neighbour is also visited



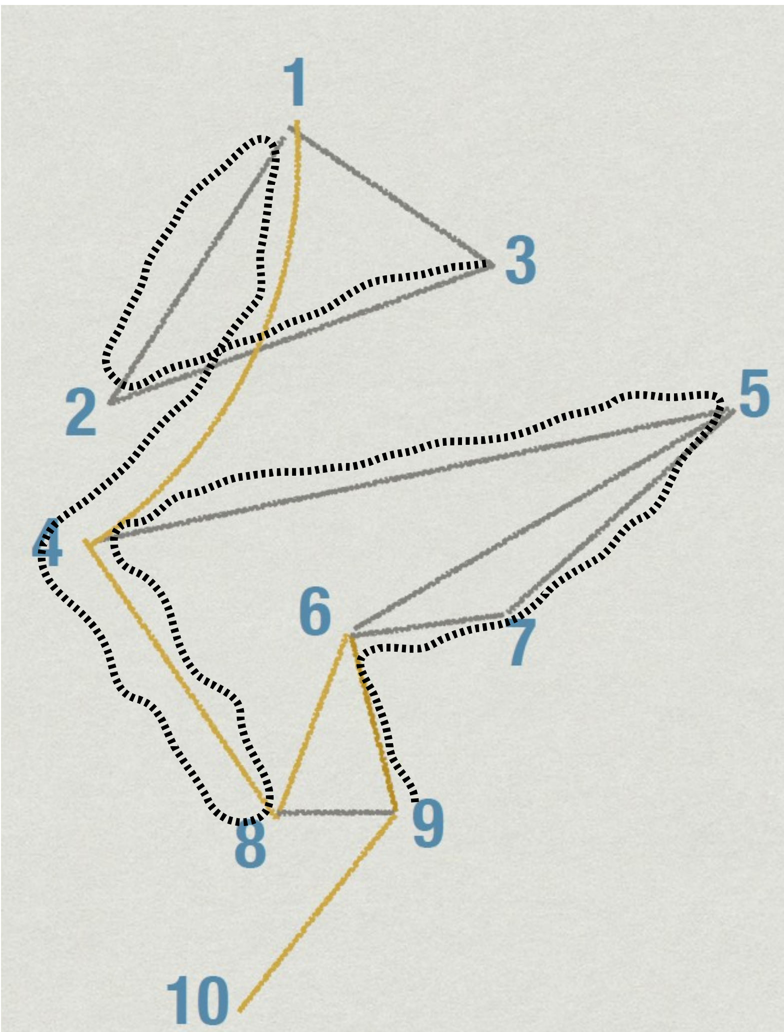
Notice anything in the diagram?



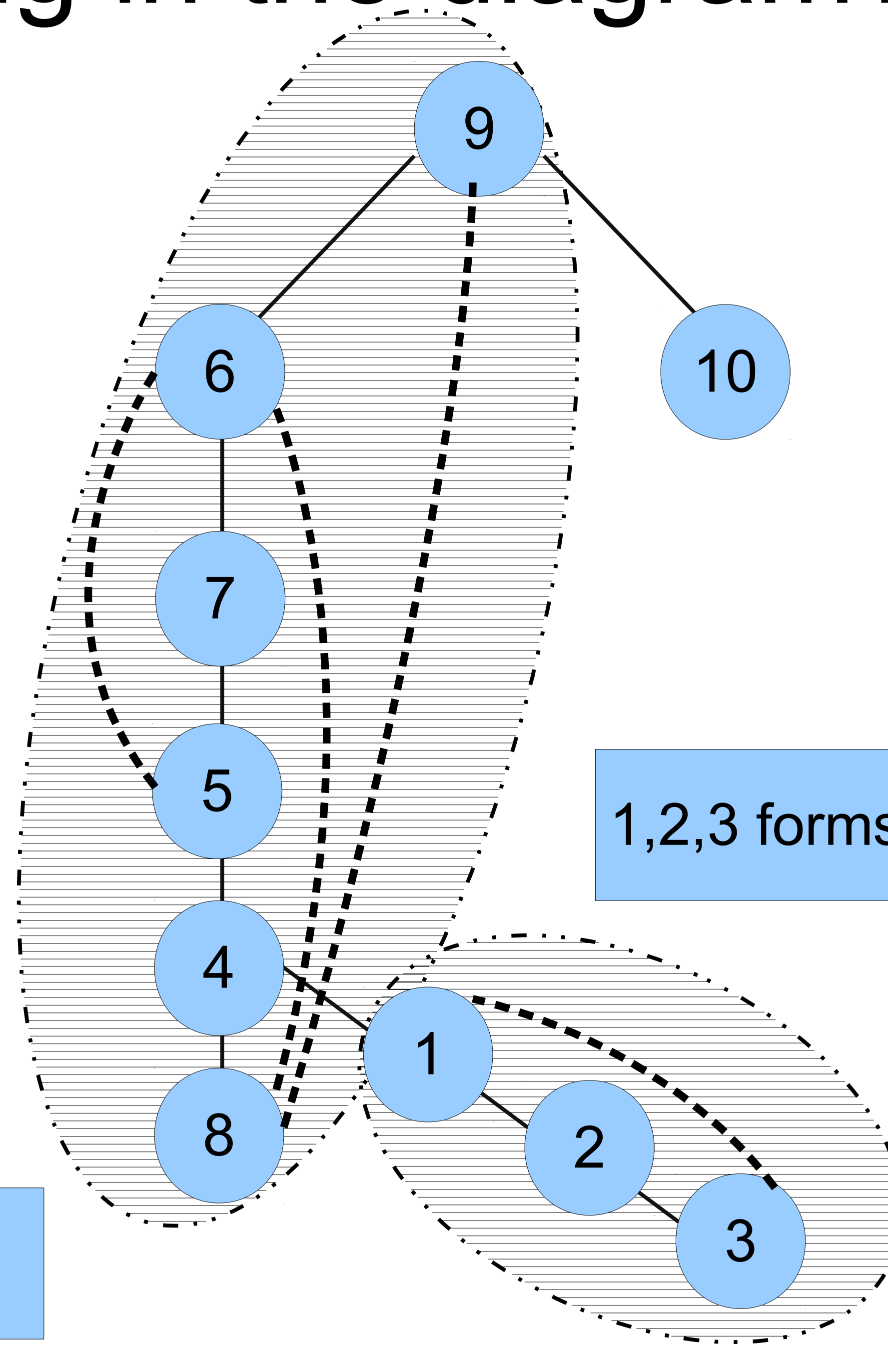
Dashed line when a visited node's neighbour is also visited



Notice anything in the diagram?

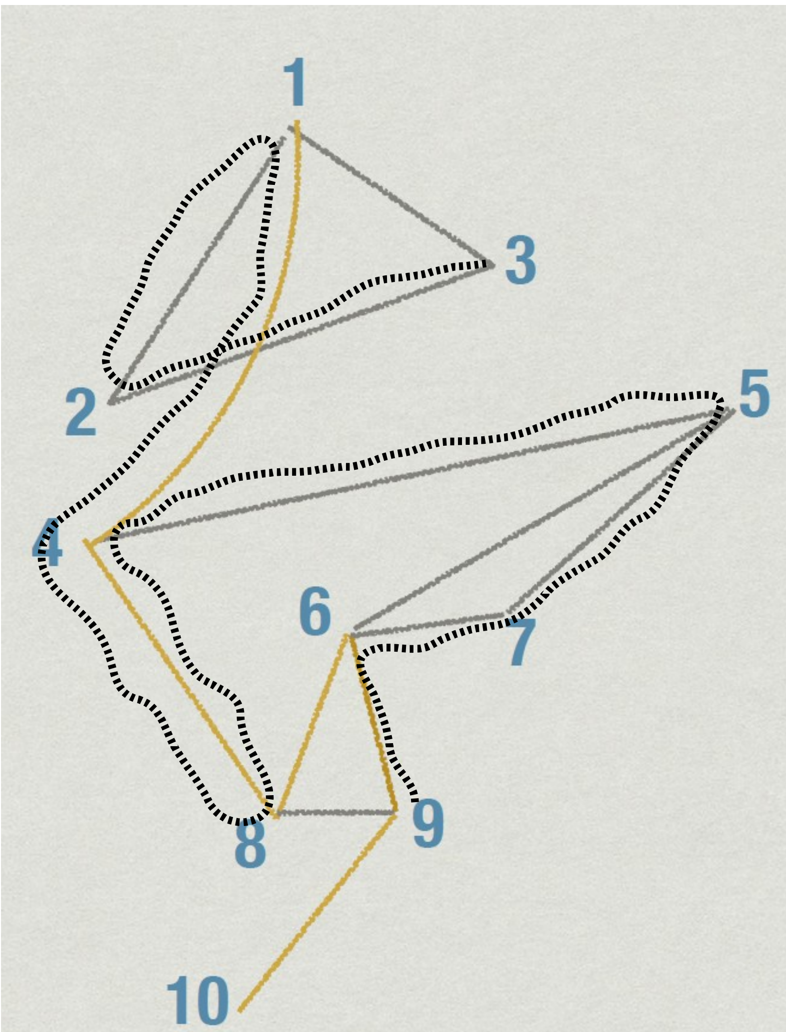


Dashed line when a visited node's neighbour is also visited

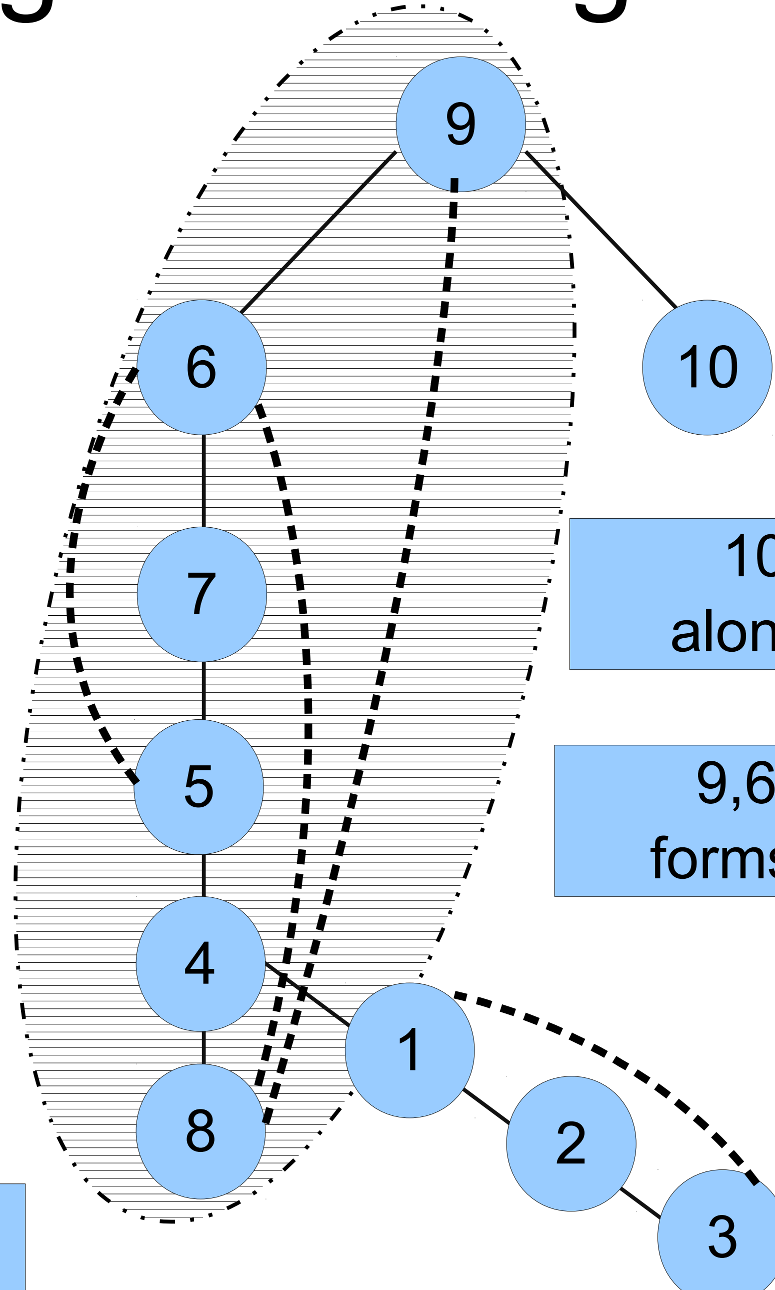


1,2,3 forms a cluster

Notice anything in the diagram?



Dashed line when a visited node's neighbour is also visited



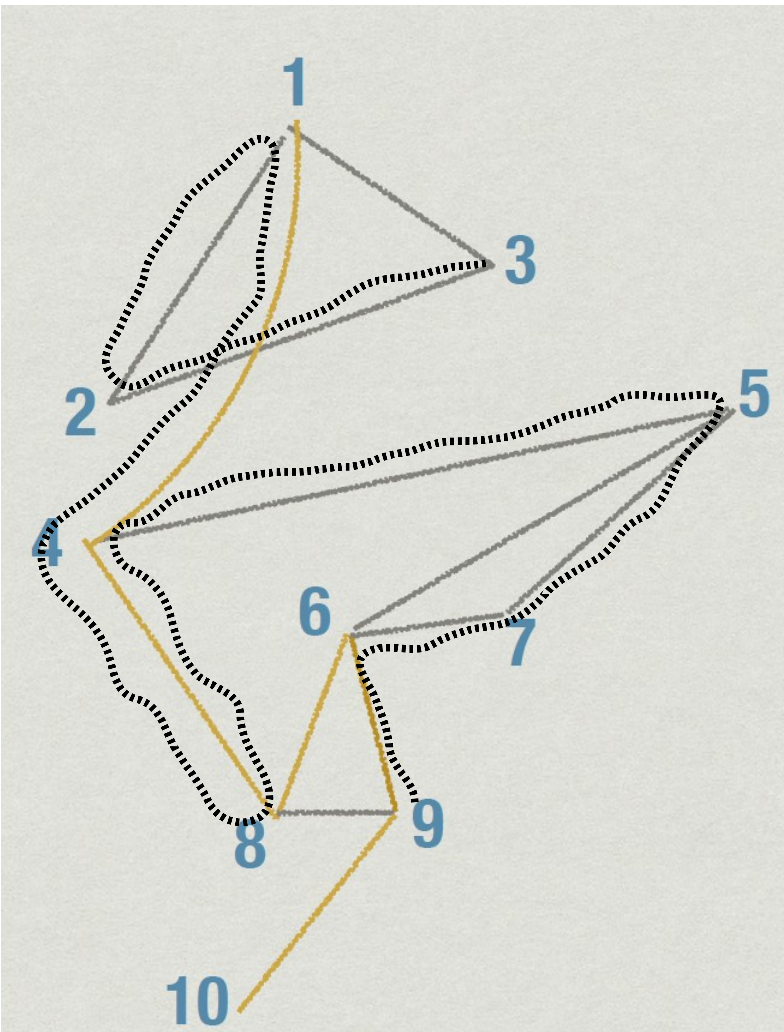
10 stands alone by itself

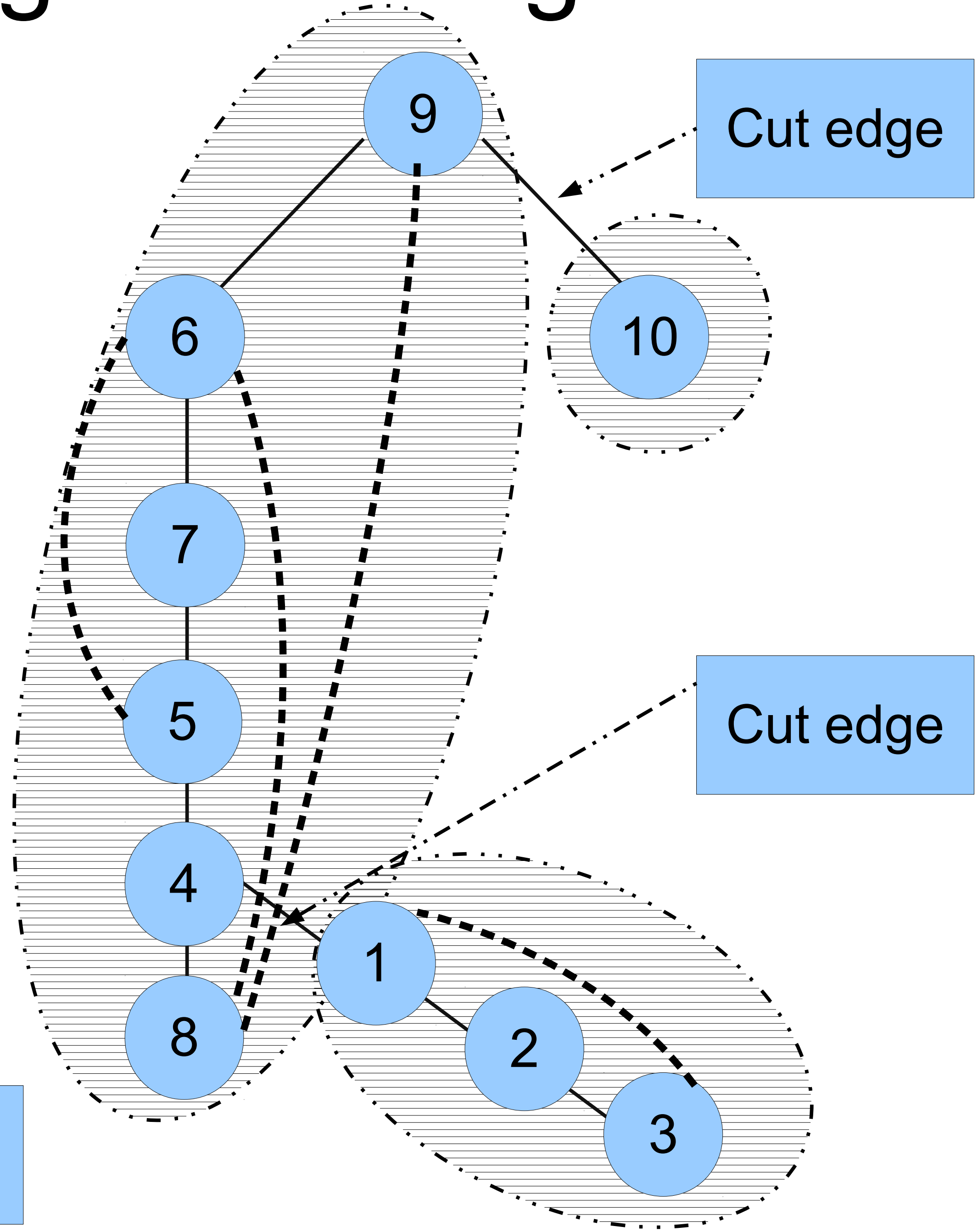
9,6,7,5,4,8 forms a cluster

DFS can discover cut edges

- A *cut edge* is an edge which if removed from a graph, will make a connected graph disconnected. If the graph is already disconnected, the cut edge increases the number of connected components by 1.

Notice anything in the diagram?



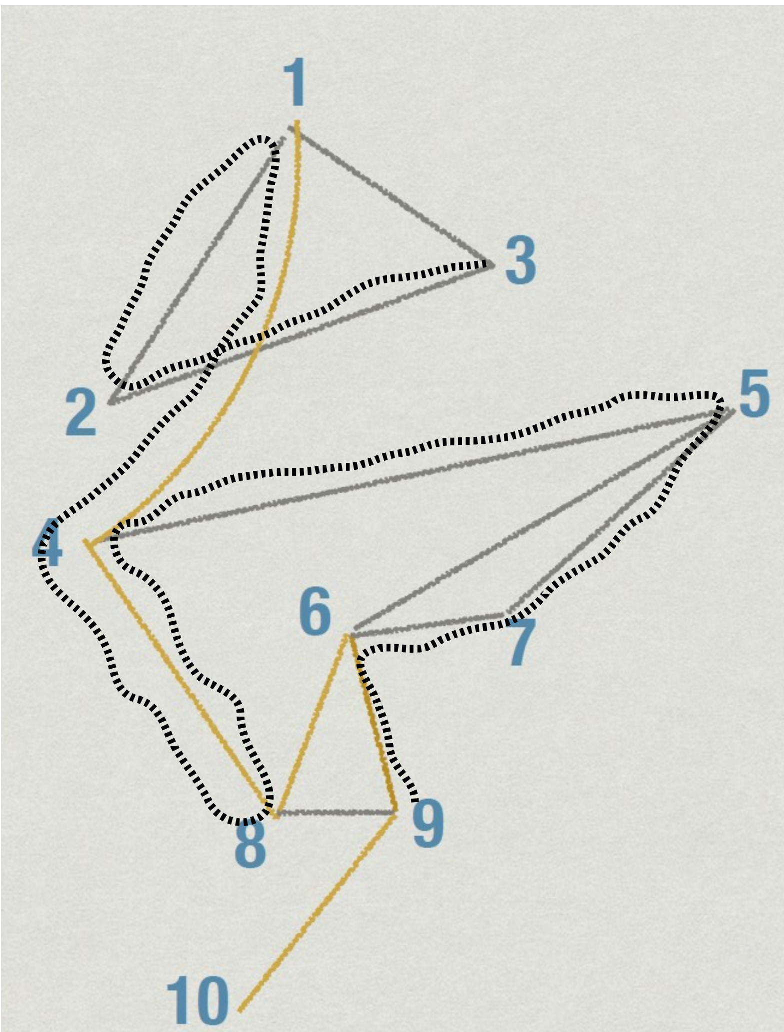


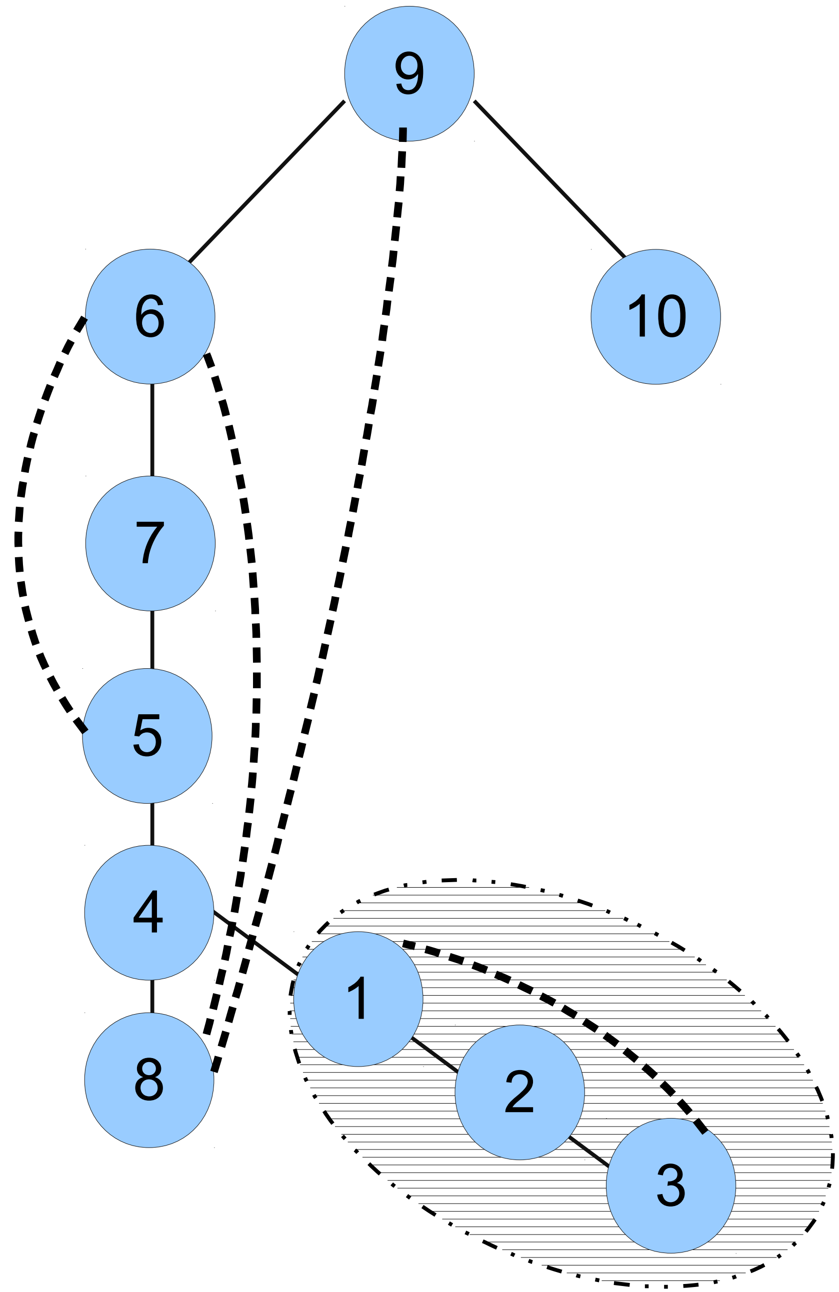
Note that the clusters dont have a cut edge

DFS can discover cut edges

- A *cut edge* is an edge which if removed from a graph, will make a connected graph disconnected. If the graph is already disconnected, the cut edge increases the number of connected components by 1.
- To find the cut edges, we would need to modify DFS so that we can find the clusters which dont have a cut edge within them (i.e. cut edges are between the clusters)

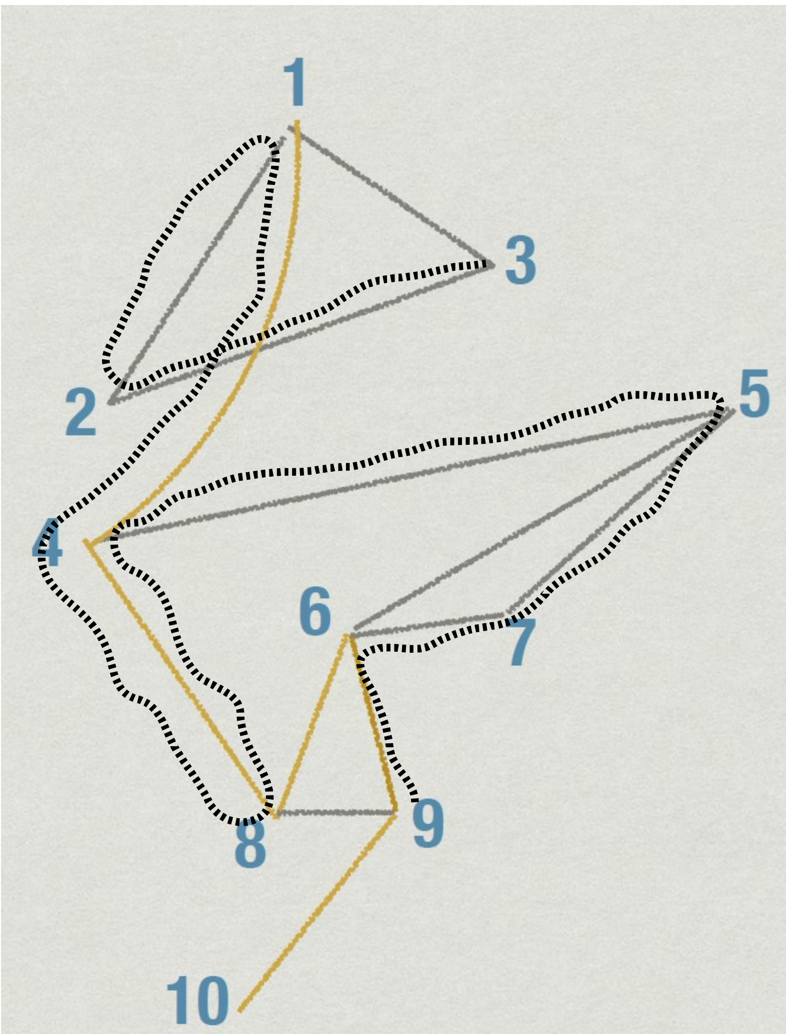
How do we find the cluster 1,2,3 ?



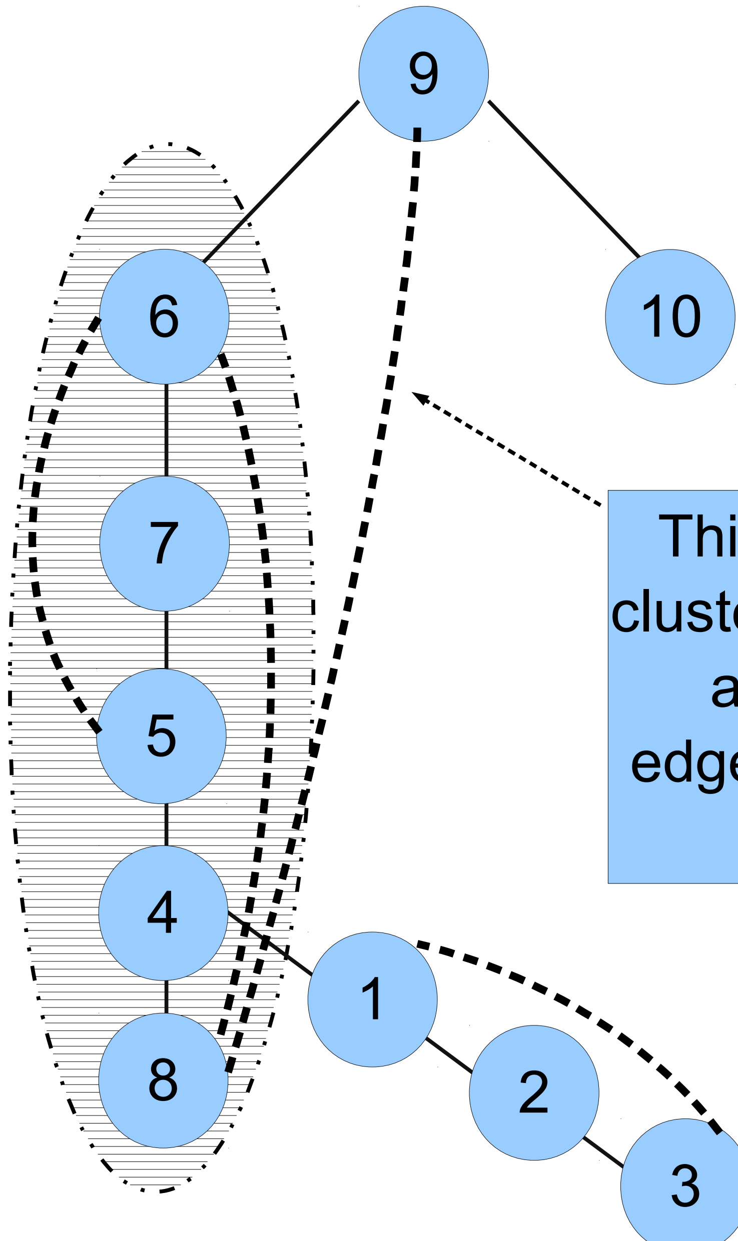


No node from inside the cluster has a dotted edge to any node outside

How do we find a cluster ?

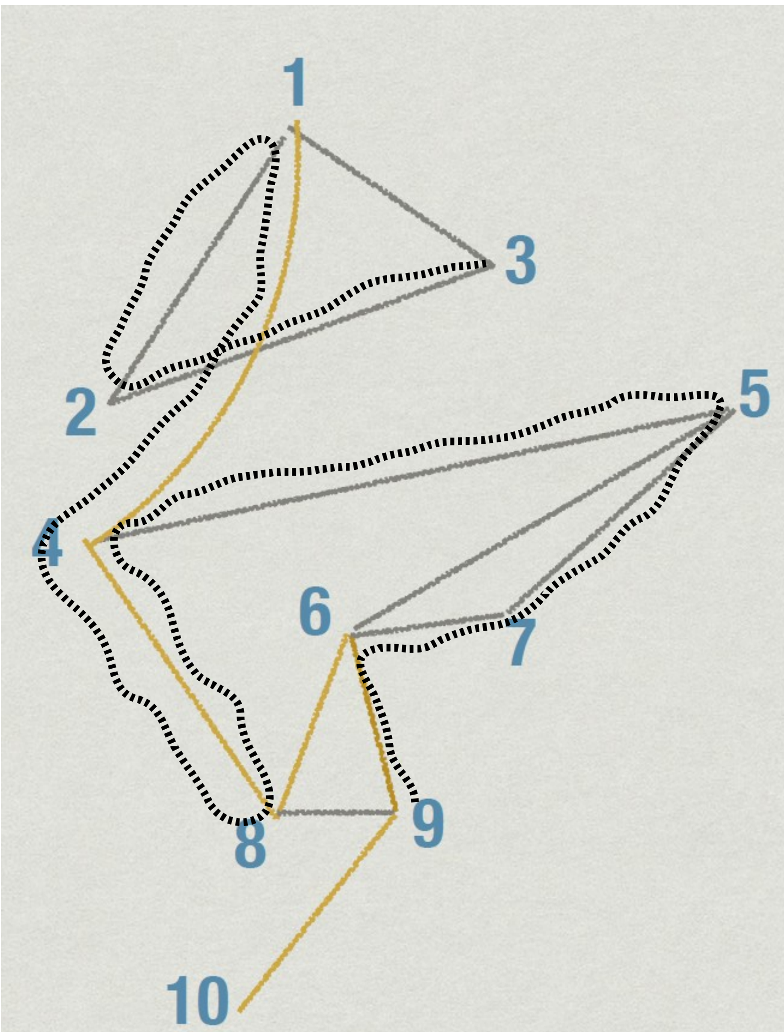


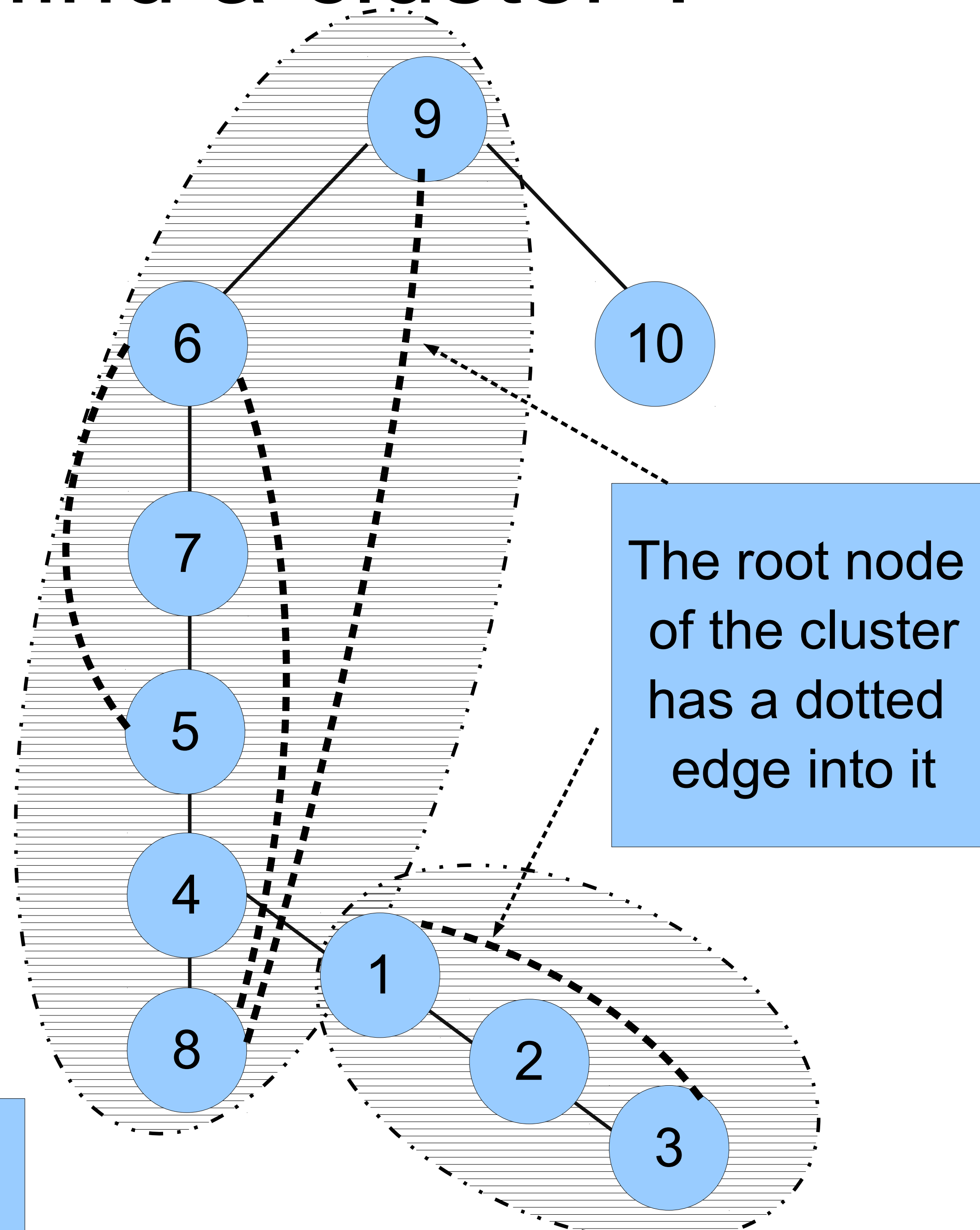
No node from inside the cluster has a dotted edge to any node outside



This is not a cluster because a dotted edge goes out of it

How do we find a cluster ?





No node from inside the cluster has a dotted edge to any node outside

Counting the nodes before and after call to DFS(i)

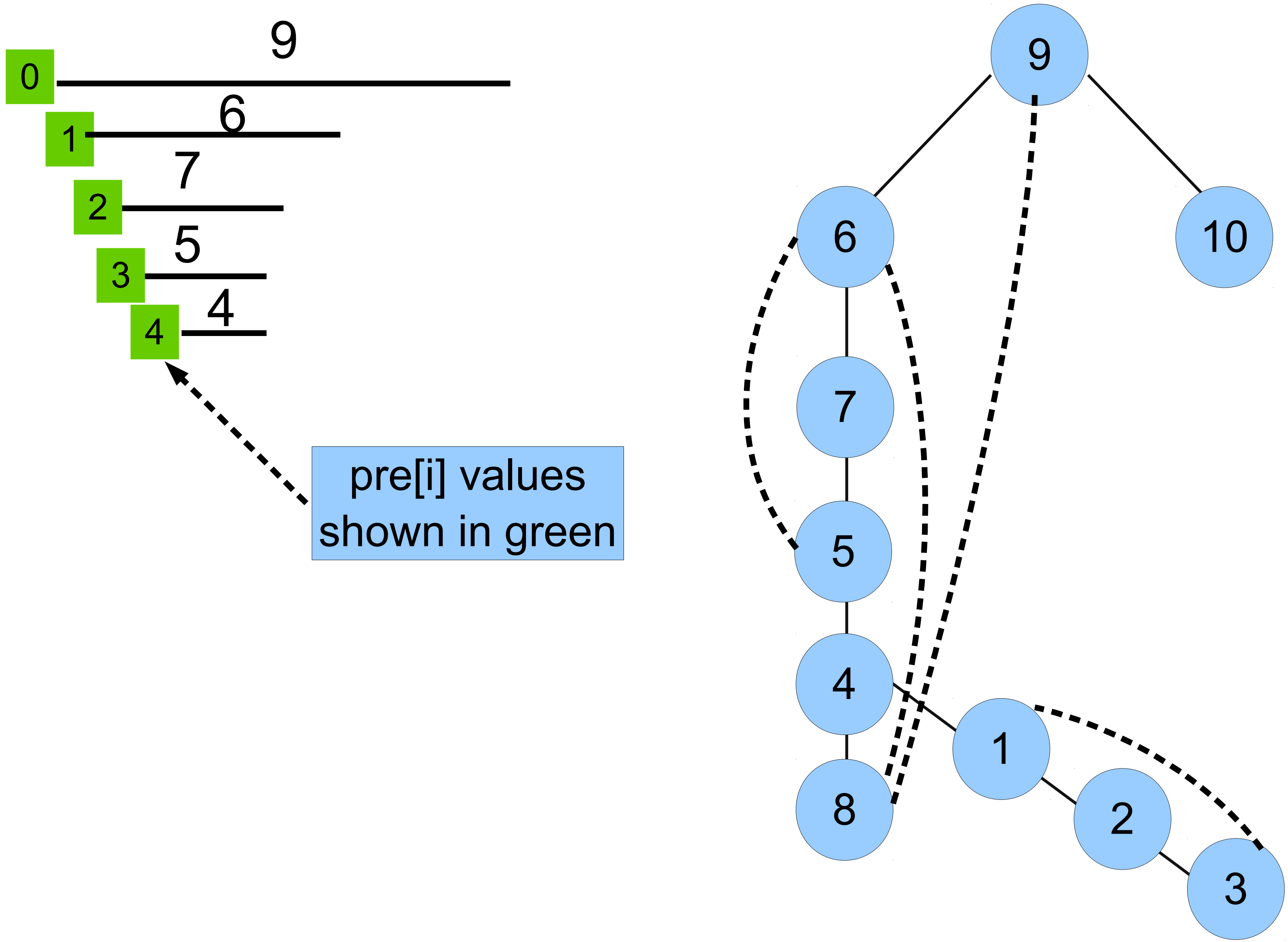
```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}
   count = 0

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1; pre[i] = count; count++

   //Explore each neighbours of i recursively
   for each (i,j) in E
      if visited[j] == 0
         parent[j] = i
         DFS(j)
         post[i] = count; count++
```

Keep a count of node in/outs

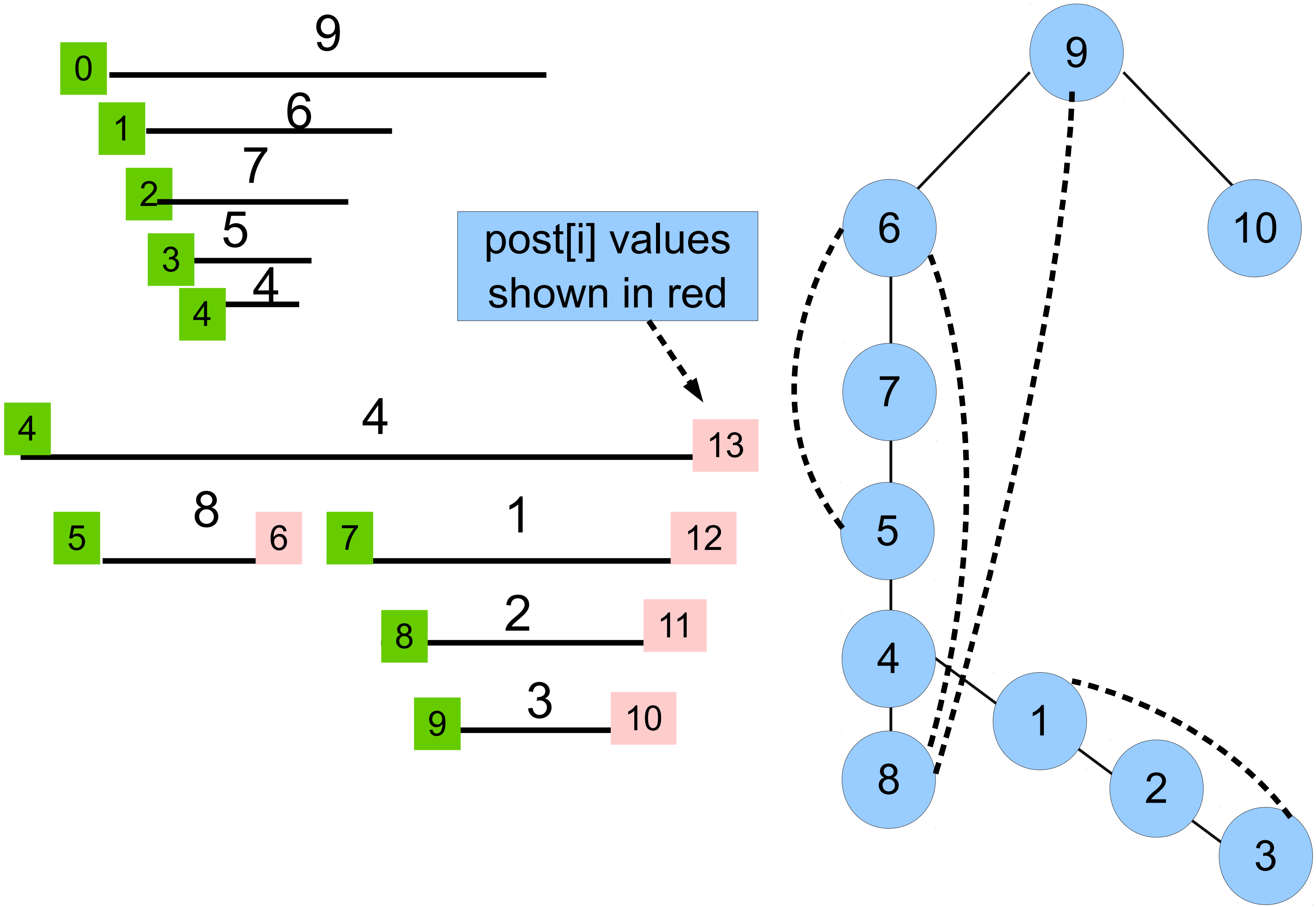
pre[i] records count value on entry

post[i] records count value on exit

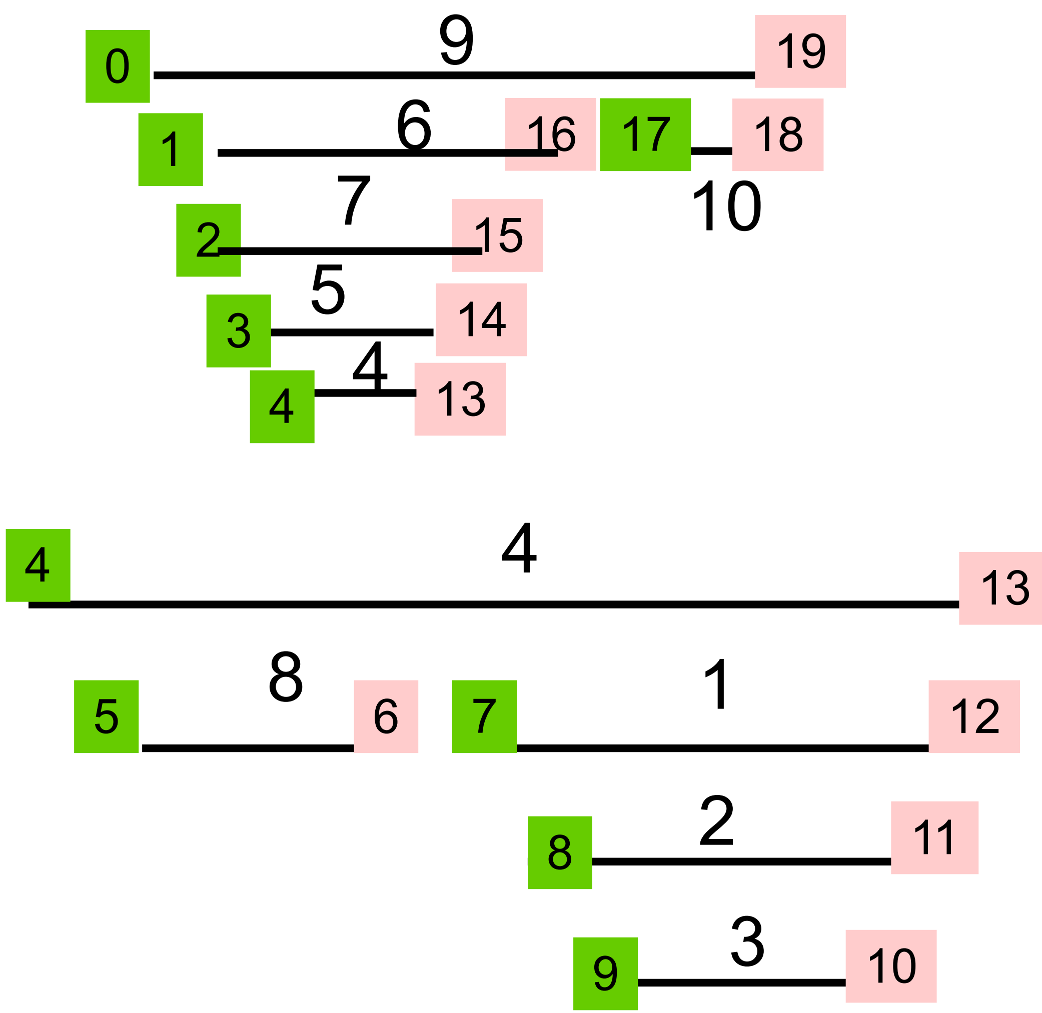
There is a nice parenthesis structure to the pre[i] to post[i] interval

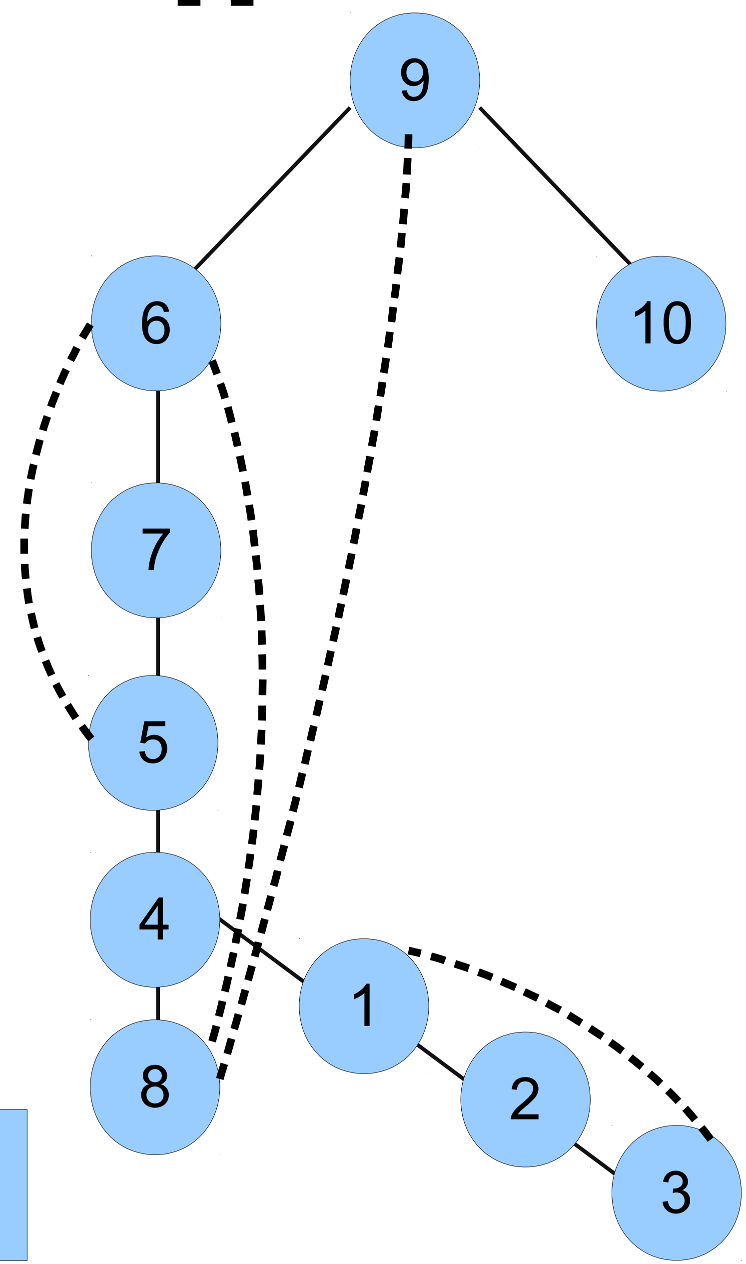


There is a nice parenthesis structure to the pre[i] to post[i] interval



There is a nice parenthesis structure to the pre[i] to post[i] interval

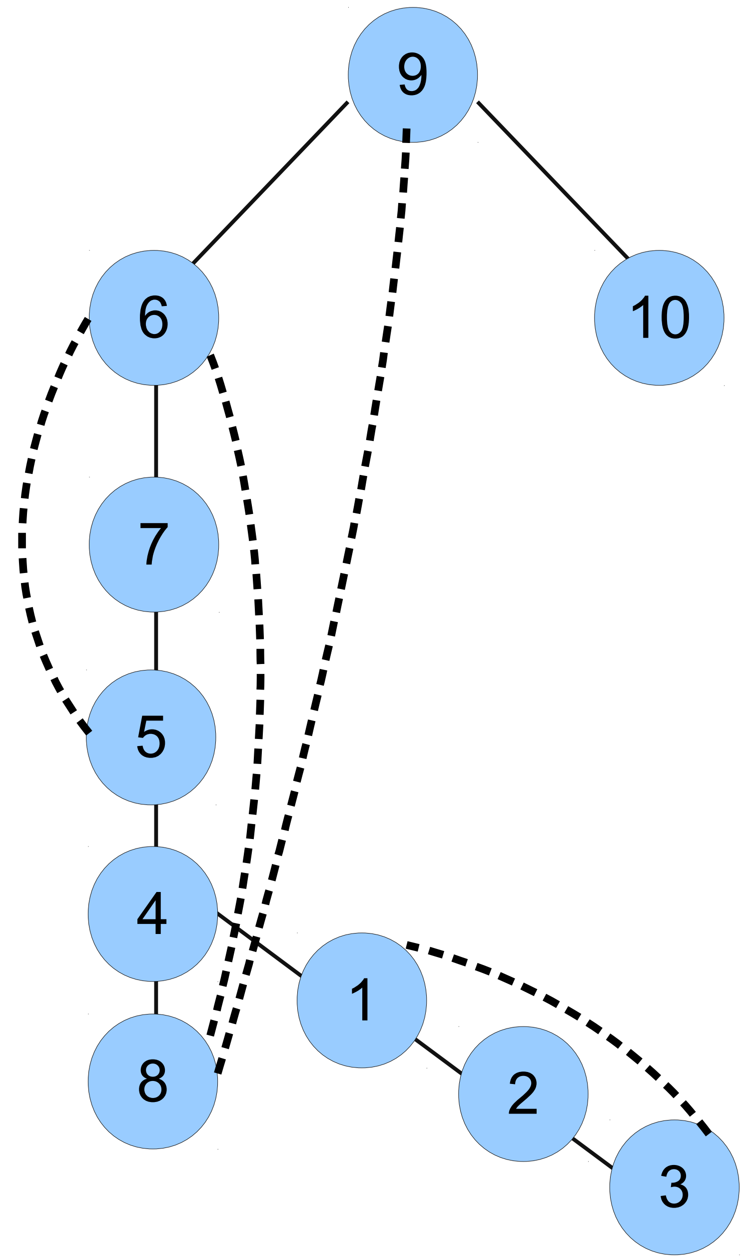




The intervals are disjoint or are enclosed in one another … parenthesis

Algorithm to find the clusters

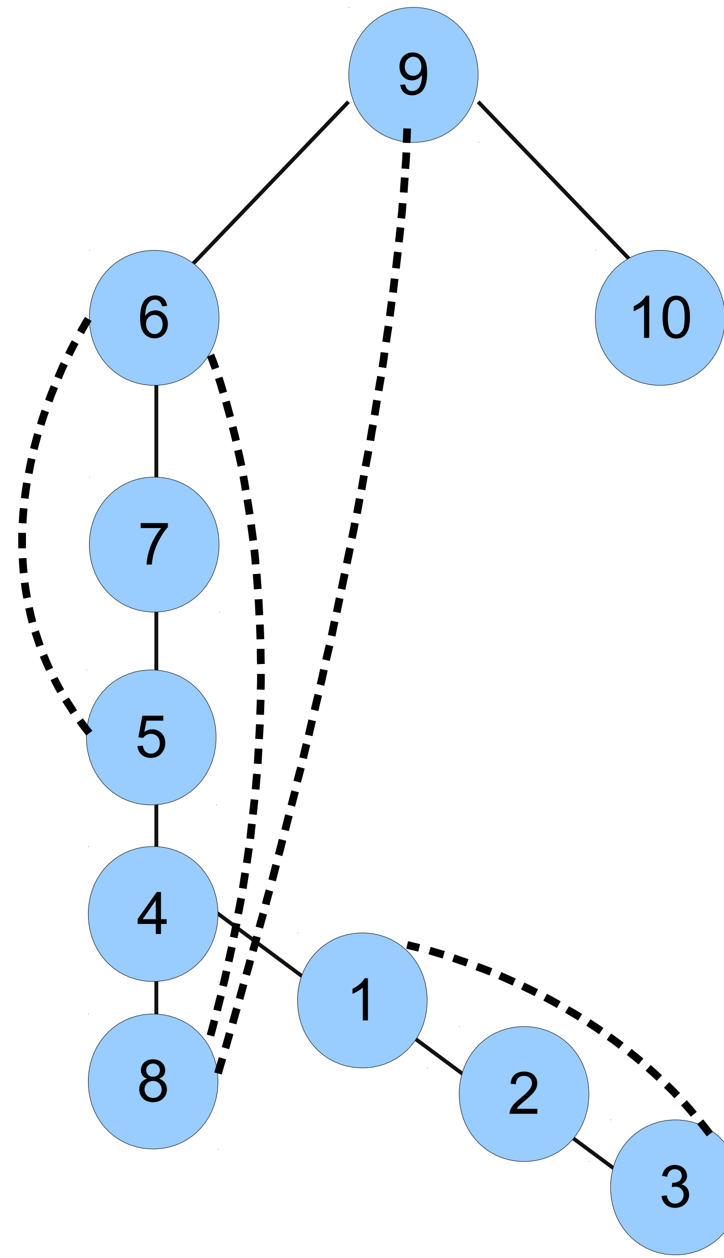
- Keep a counter called low[i] that records the lowest pre[i] value encountered by any descendant of i while traversing
- low[i] will be 0 for i=8 since it neighbours 9 (which has pre of 0)
- So low[i] will be 0 also for i=9,6,7,5,4 which all have 8 as a descendant



Exercise: Modify the pseudo-code for DFS to add low values and to print the clusters

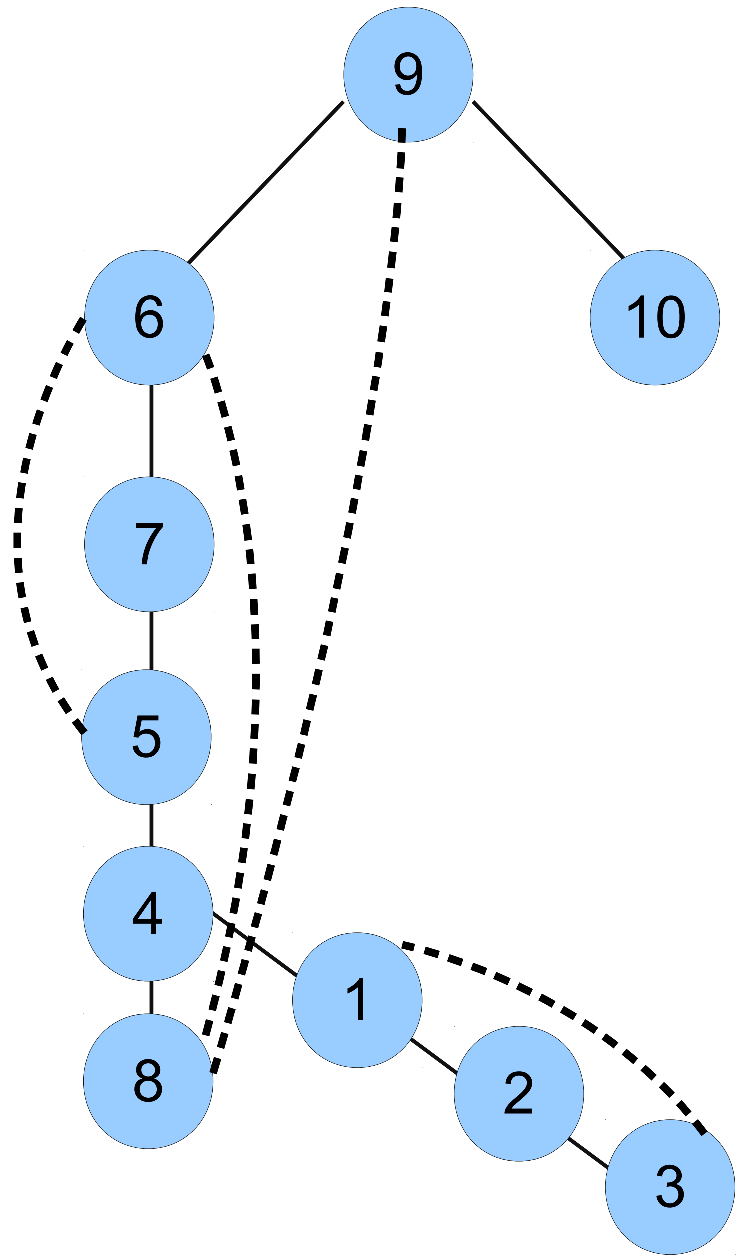
Algorithm to find the clusters

- Keep a counter called low[i] that records the lowest pre[i] value encountered by any descendant of i while traversing
- low[i] will be 0 for i=8 since it neighbours 9 (which has pre of 0)
- So low[i] will be 0 also for i=9,6,7,5,4 which all have 8 as a descendant
- low[i] will be 7 for i=3 since pre of 1 is 7
- So low[i] will be 7 also for i=1,2



Algorithm to find the clusters

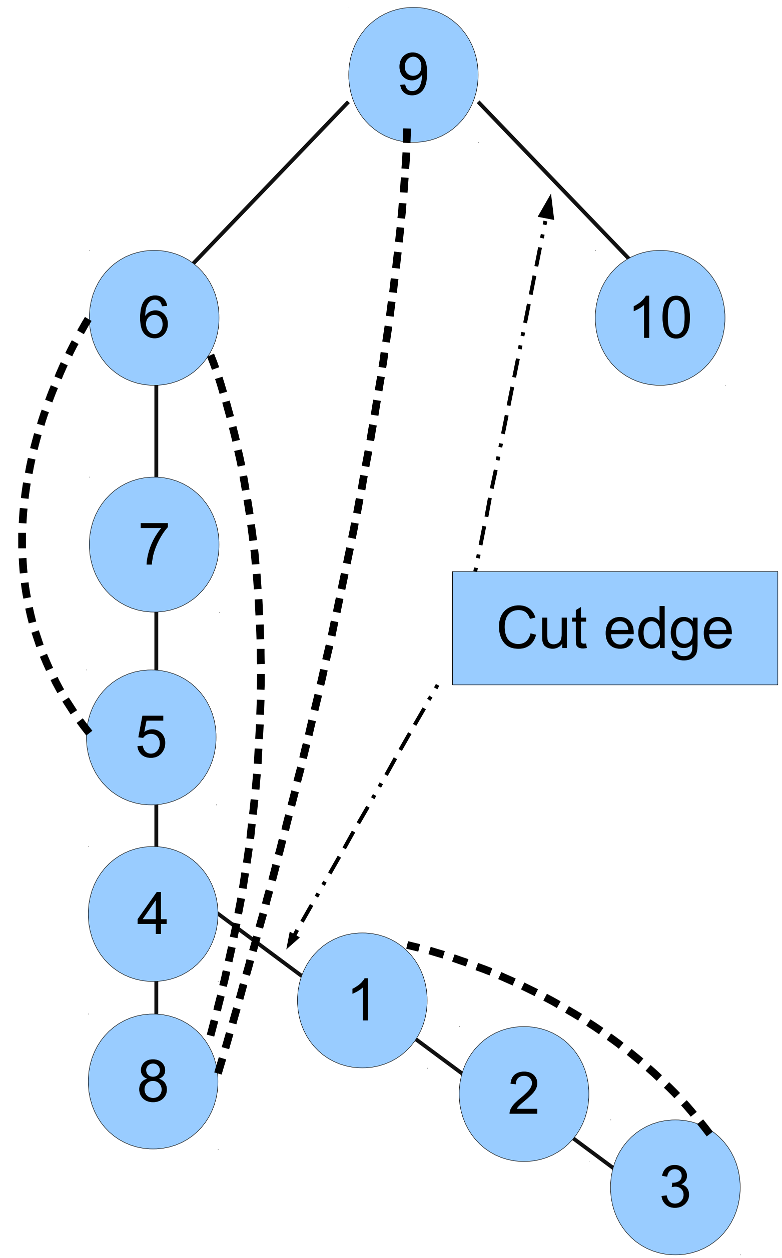
- Keep a counter called low[i] that records the lowest pre[i] value encountered by any descendant of i while traversing
- low[i] will be 0 for i=8 since it neighbours 9 (which has pre of 0)
- So low[i] will be 0 also for i=9,6,7,5,4 which all have 8 as a descendant
- low[i] will be 7 for i=3 since pre of 1 is 7
- So low[i] will be 7 also for i=1,2
- All vertices i with the same low[i] value belong to one cluster



Exercise: Modify the pseudo-code for DFS to add low values and to print the clusters

Algorithm to find the cut-edges

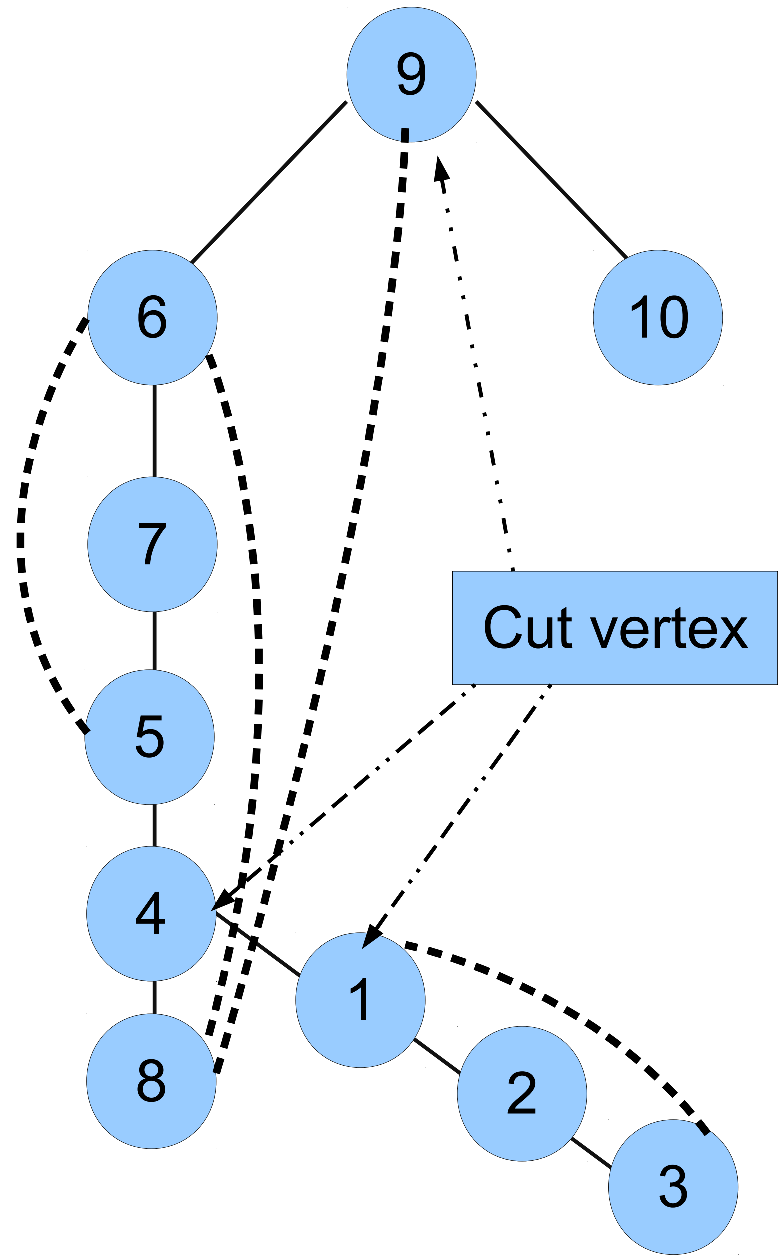
- low[i] will be 0 for i=8 since it neighbours 9 (which has pre of 0)
- So low[i] will be 0 also for i=9,6,7,5,4 which all have 8 as a descendant
- low[i] will be 7 for i=3 since pre of 1 is 7
- So low[i] will be 7 also for i=1,2
- If i is a vertex with low[i] = pre[i], then the edge (parent[i],i) is a cut-edge
- Edge (4,1) and (9.10) are cut-edges



Exercise: Modify the pseudo-code for DFS to add low values and to print the clusters

Cut vertex (or articulation point)

- A *cut-vertex* of a graph is a vertex such that if it is removed from the graph, it will disconnect the graph (or if the graph is already connected, then it increases the number of connected components by 1)
- Cut-edge will have at least one cut-vertex

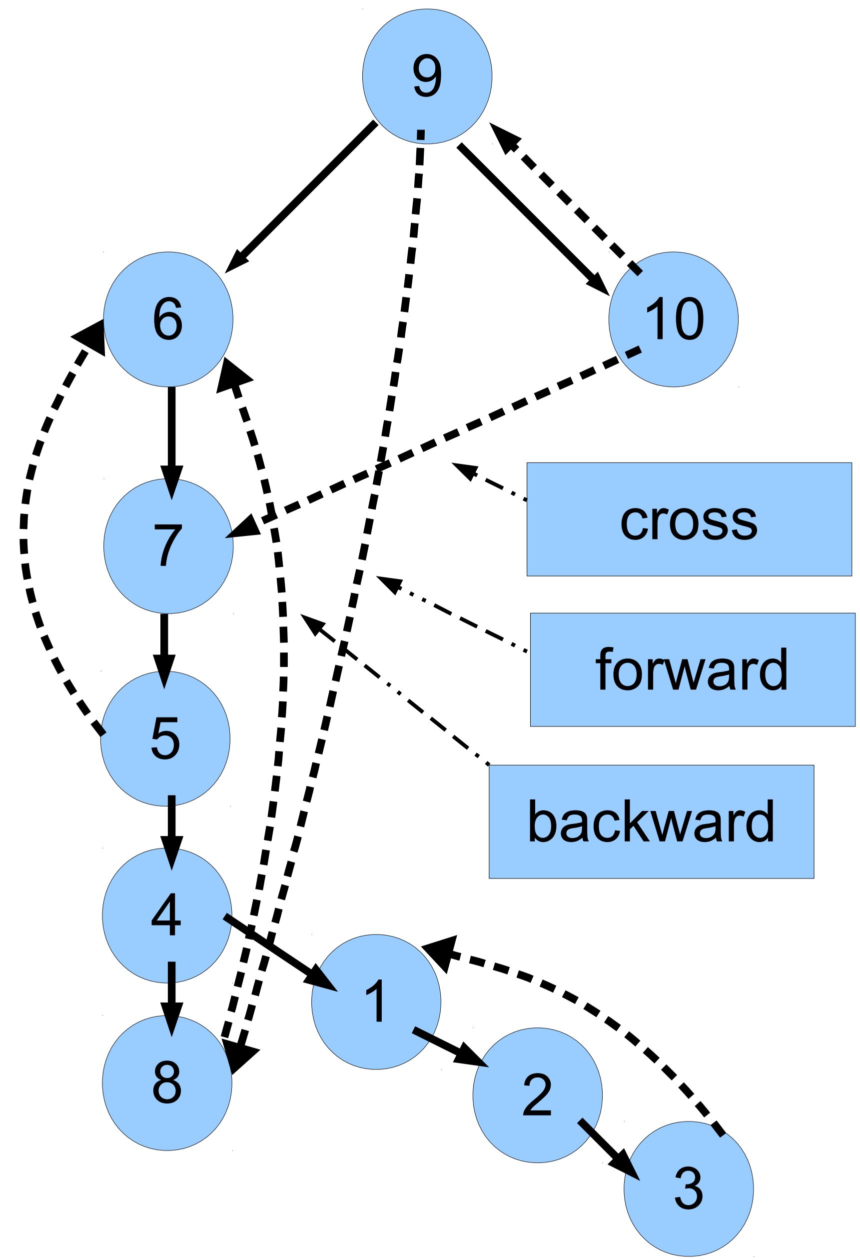


In class assignment for next class:
Find all the cut vertices of a graph using DFS

Directed graphs:
Applications of DFS

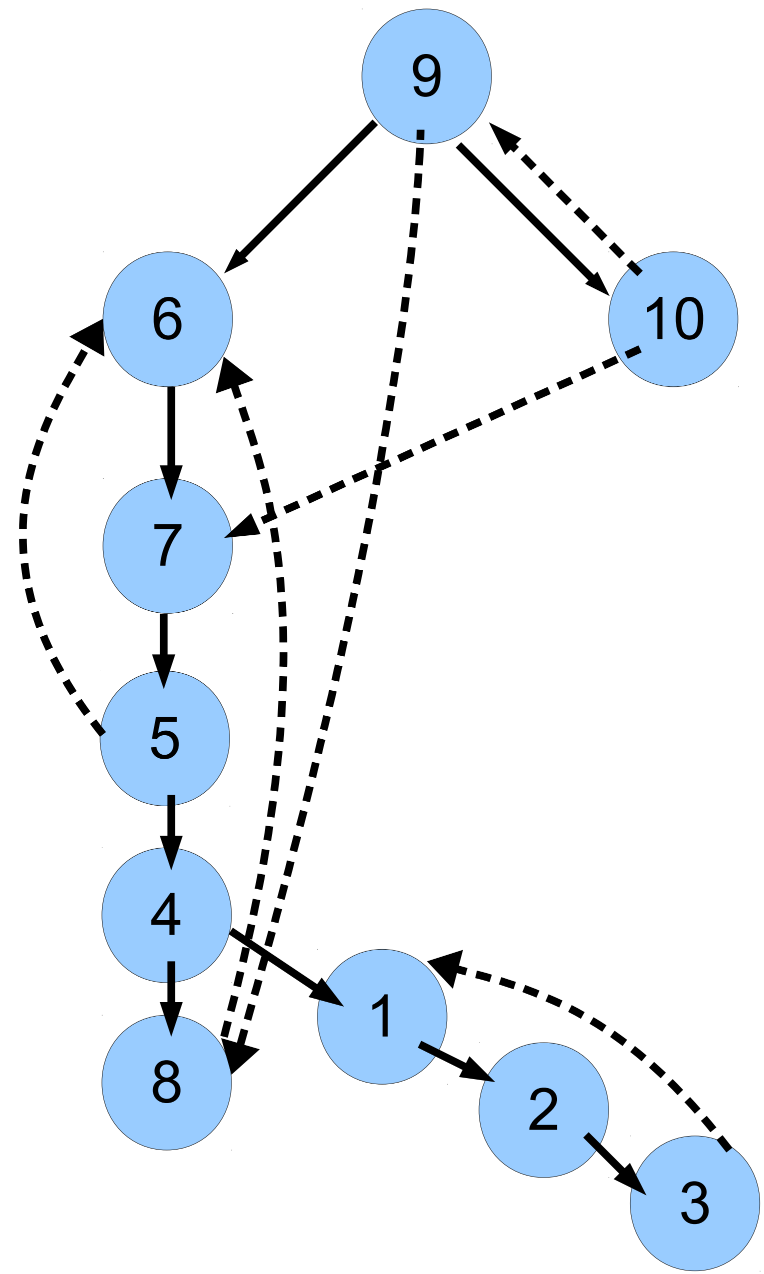
Going back to directed graphs

- Suppose the given graph was directed as shown, and we did a DFS on it using the same traversal method to get the structure as shown
- The edges that are formed when the node is first visited is shown bold as in the undirected case (these are called the *DFS tree edges, sinced they form a tree*). Edges other than these are dotted edges.
- The dotted edges can now be of three types – *backward, forward or cross.* Forward goes from ancestor to descendant. Backward from descendant to ancestor. Cross is all the other dotted edges.



Going back to directed graphs

- We had used the word "cluster" loosely without definition – we can now define it formally

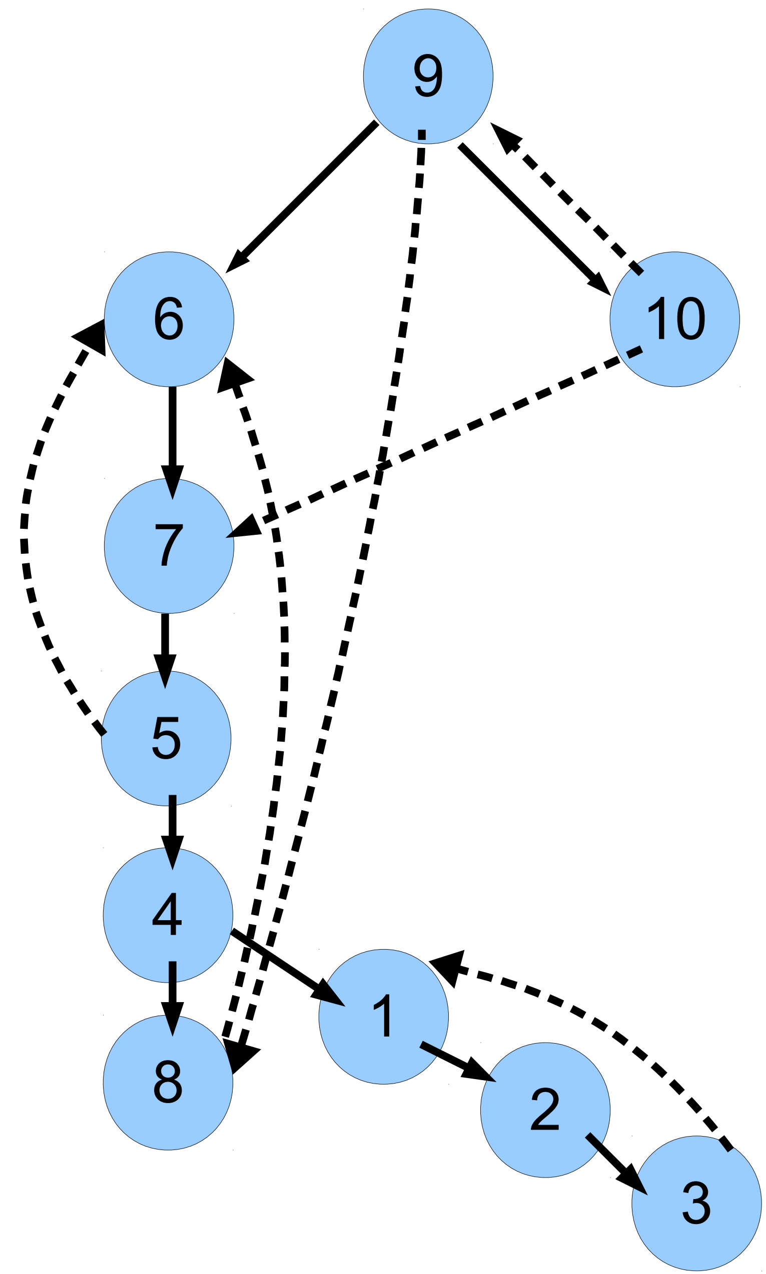


Strongly connected component

- A strongly connected component (SCC) of a graph G = (V,E) is a *maximal set* of vertices $U \subseteq V$ such that for any pair of vertices $u,v \in U$, there is a path from u to v and a path from v to u in G.

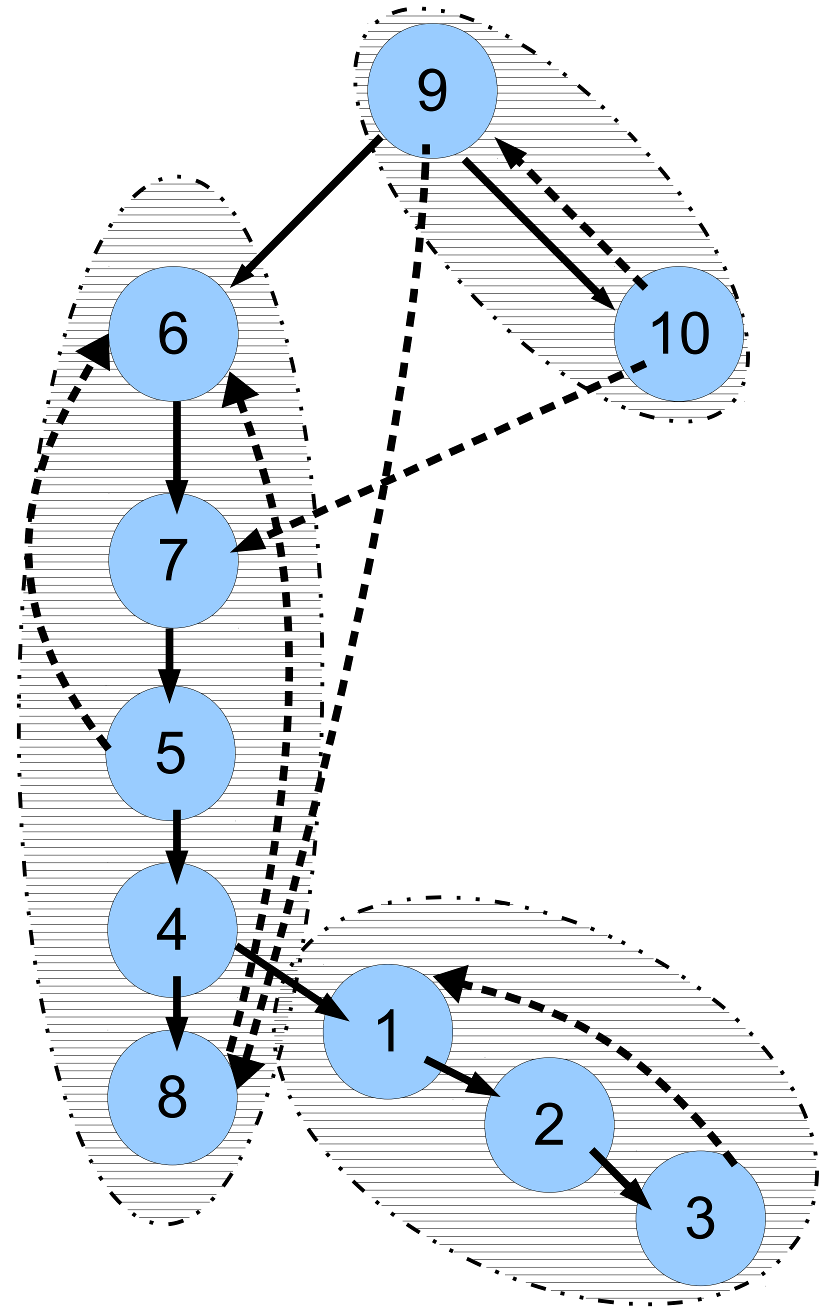
Going back to directed graphs

- Can you spot the SCCs in this directed graph?



Going back to directed graphs

- Can you spot the SCCs in this directed graph?



In class assignment for next class:
Find all the SCCs in a directed graph
using DFS

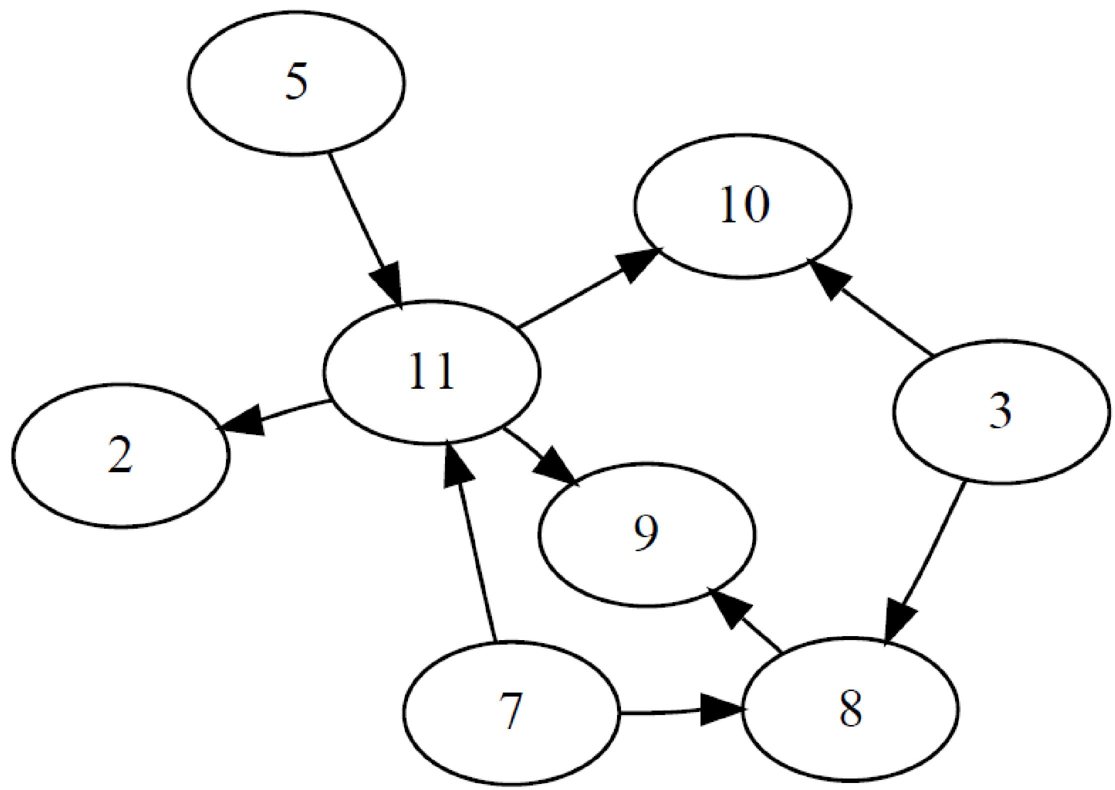
Acyclic directed graphs

- A *cycle* is a sequence of vertices $v_1, v_2, v_3, \ldots v_n$ such that $(v_i, v_{i+1}) \in E$, $v_1 = v_n$ and at least one of the v_i is distinct from v_1. It is a *simple cycle* if all the v_i are distinct from v_1
- A graph is called *acyclic* if it does not contain any (directed) cycles.
- A *topological sort* of an acyclic directed graph G is a linear ordering of its vertices such that if $(u,v) \in E$, then u appears before v in the ordering.

Topological sort of an acyclic graph

- A *topological sort* of an acyclic directed graph G is a linear ordering of its vertices such that if (u,v) $\in$ E, then u appears before v in the ordering.

Can you find a topological sort of this graph?

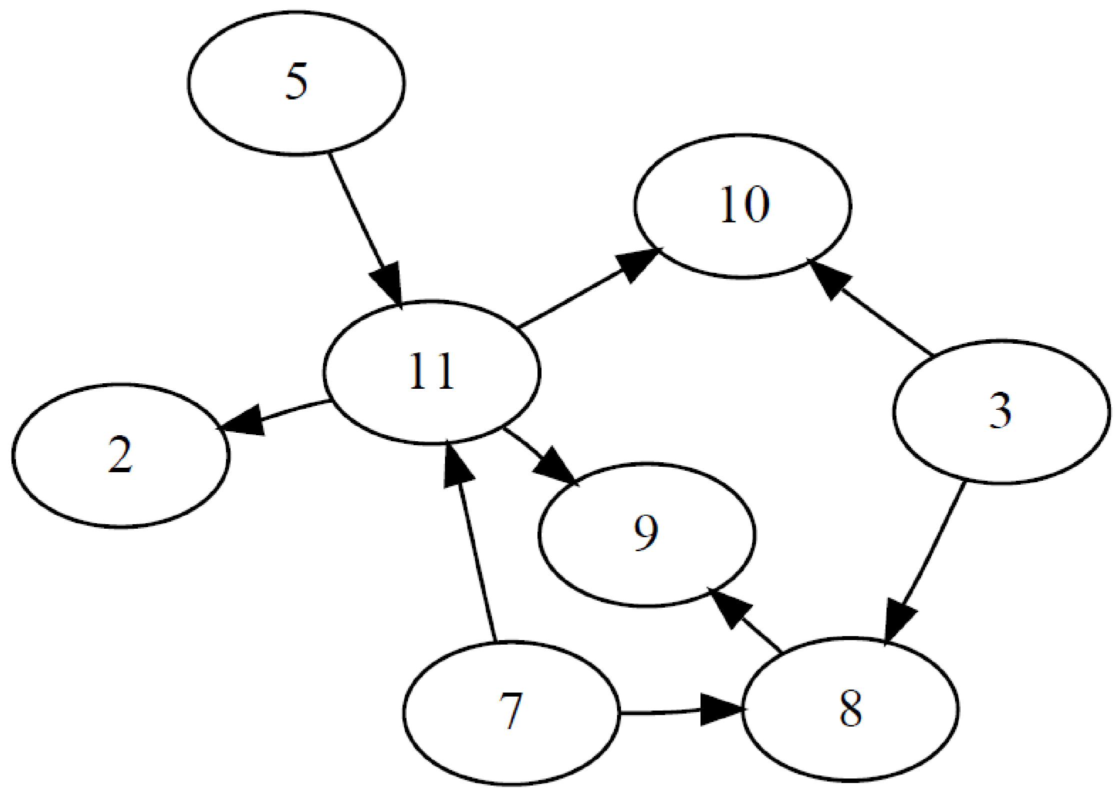


Topological sort of an acyclic graph

- A *topological sort* of an acyclic directed graph G is a linear ordering of its vertices such that if $(u,v) \in E$, then u appears before v in the ordering.

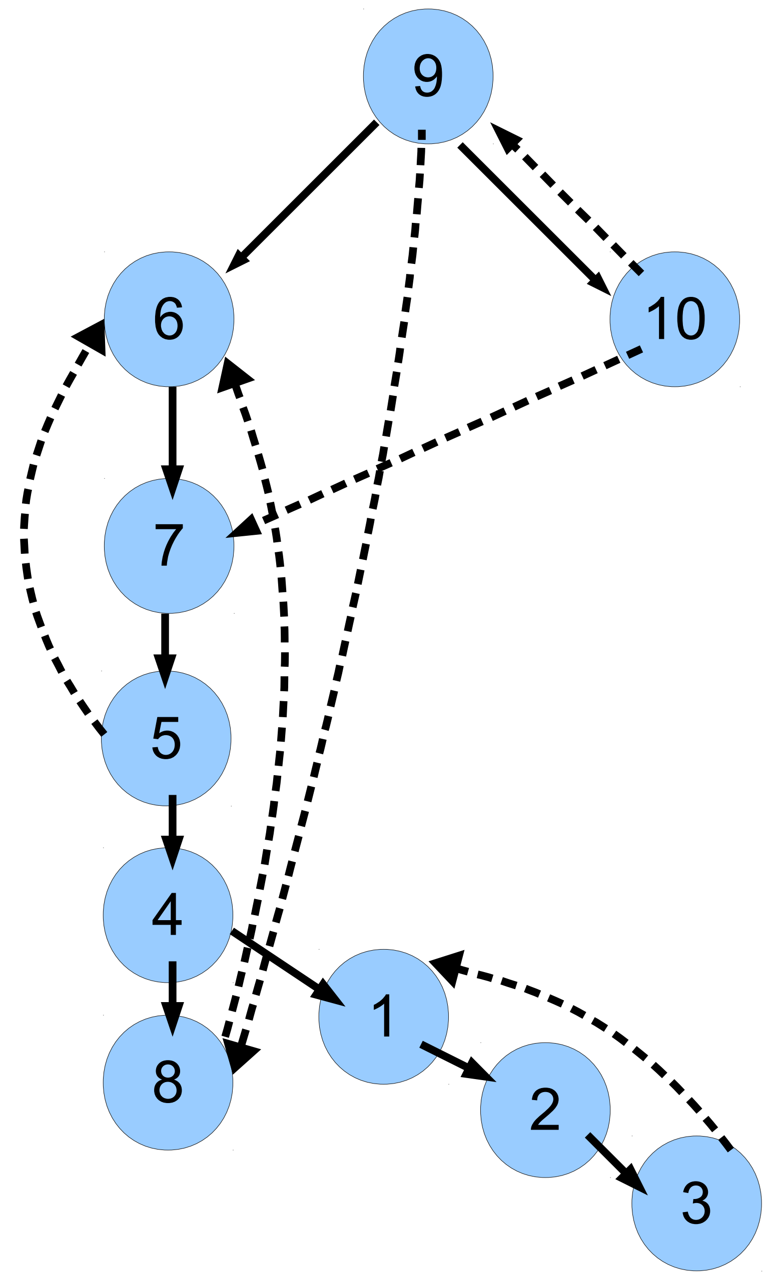
5,2,7,3,8,11,10,9

There are many others



Acyclic graphs

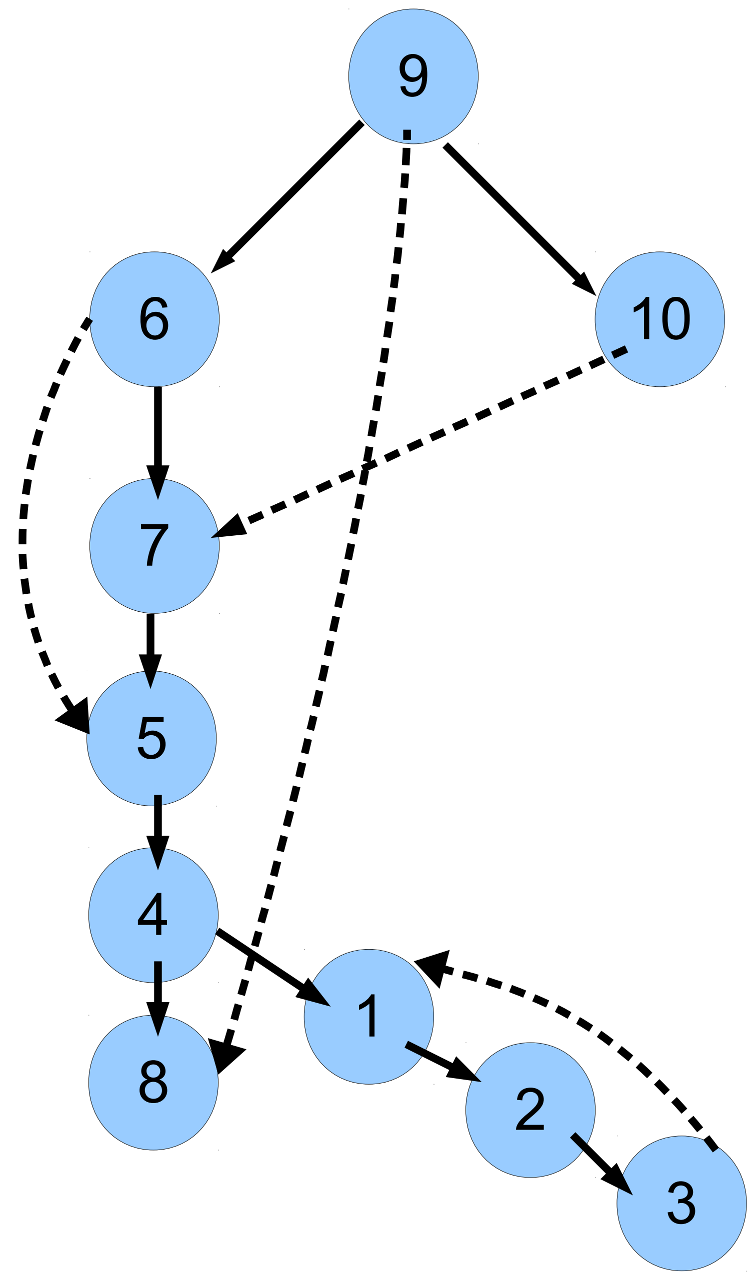
- This graph is not acyclic
- 9,10,9 is a cycle
- 1,2,3,1 is a cycle
- 6,7,5,6 is a cycle
- 6,7,5,4,8,6 is a cycle



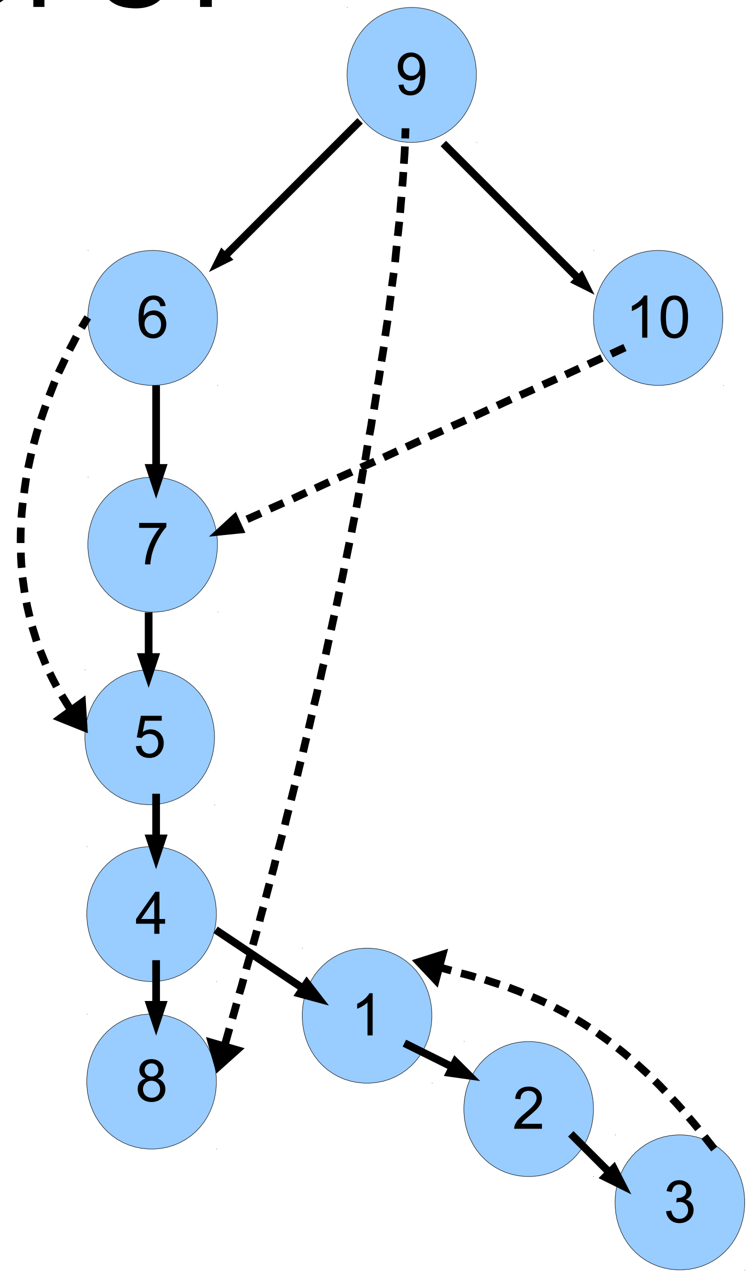
Acyclic graphs

- This graph is not acyclic
- 9,10,9 is a cycle
- 1,2,3,1 is a cycle
- 6,7,5,6 is a cycle
- 6,7,5,4,8,6 is a cycle

- Dropped two edges and changed directions on one to make it acyclic

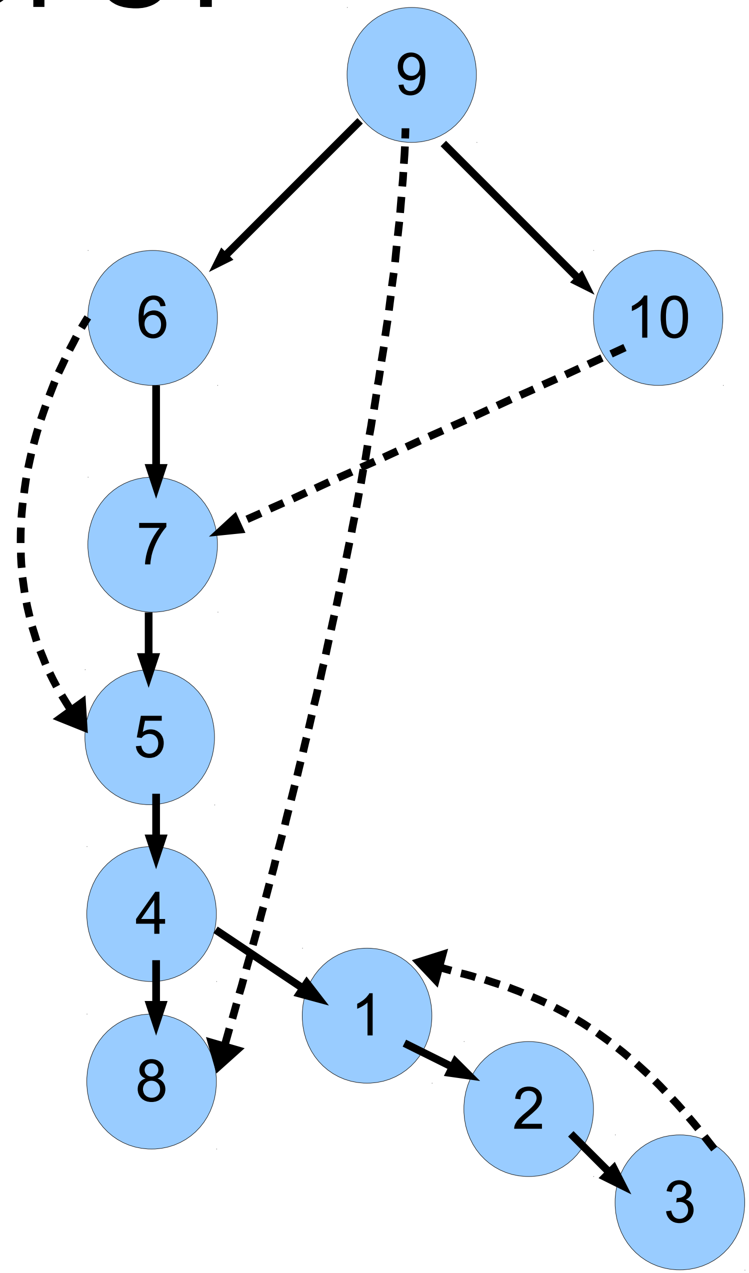


Can we do a topological sort of this graph using DFS?



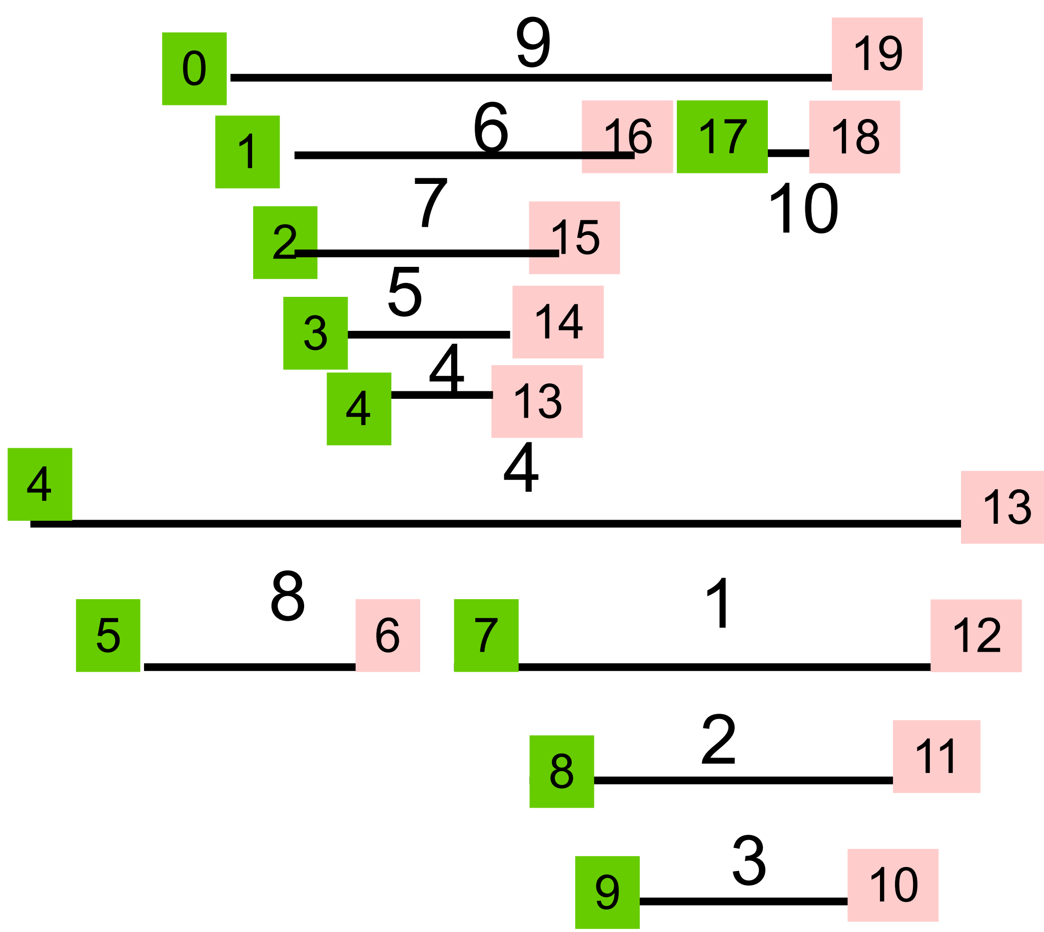
Can we do a topological sort of this graph using DFS?

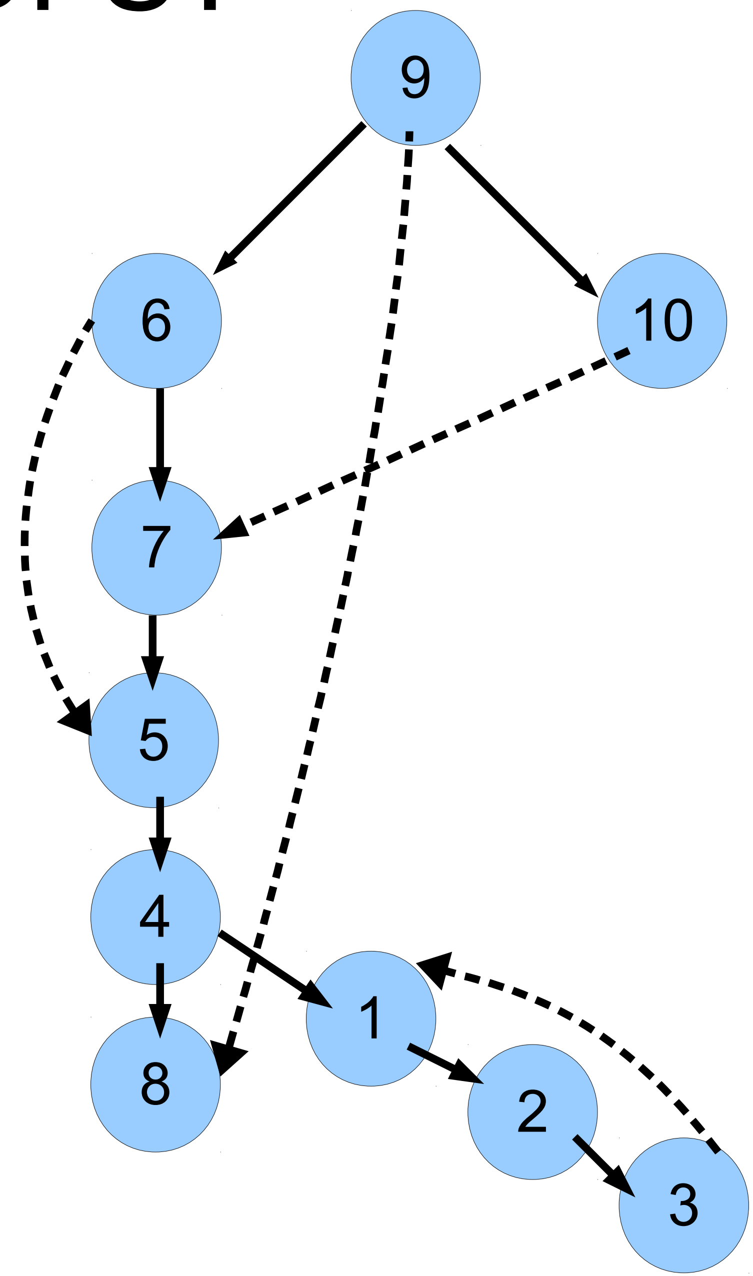
- Note that the pre and post count values are exactly the same as before, even though we have more (directed) edges.



Can we do a topological sort of this graph using DFS?

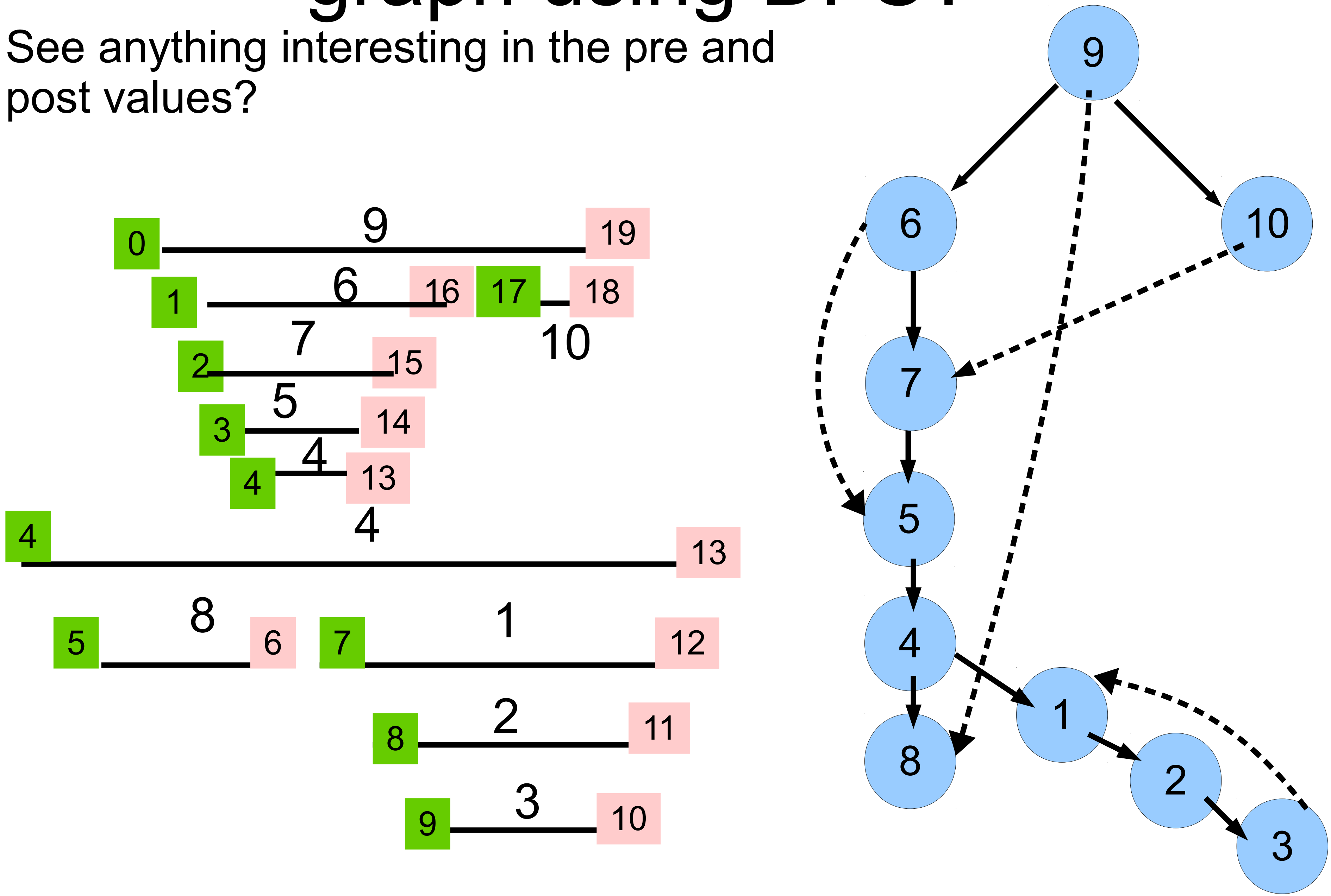
- Note that the pre and post count values are exactly the same as before, even though we have more (directed) edges.





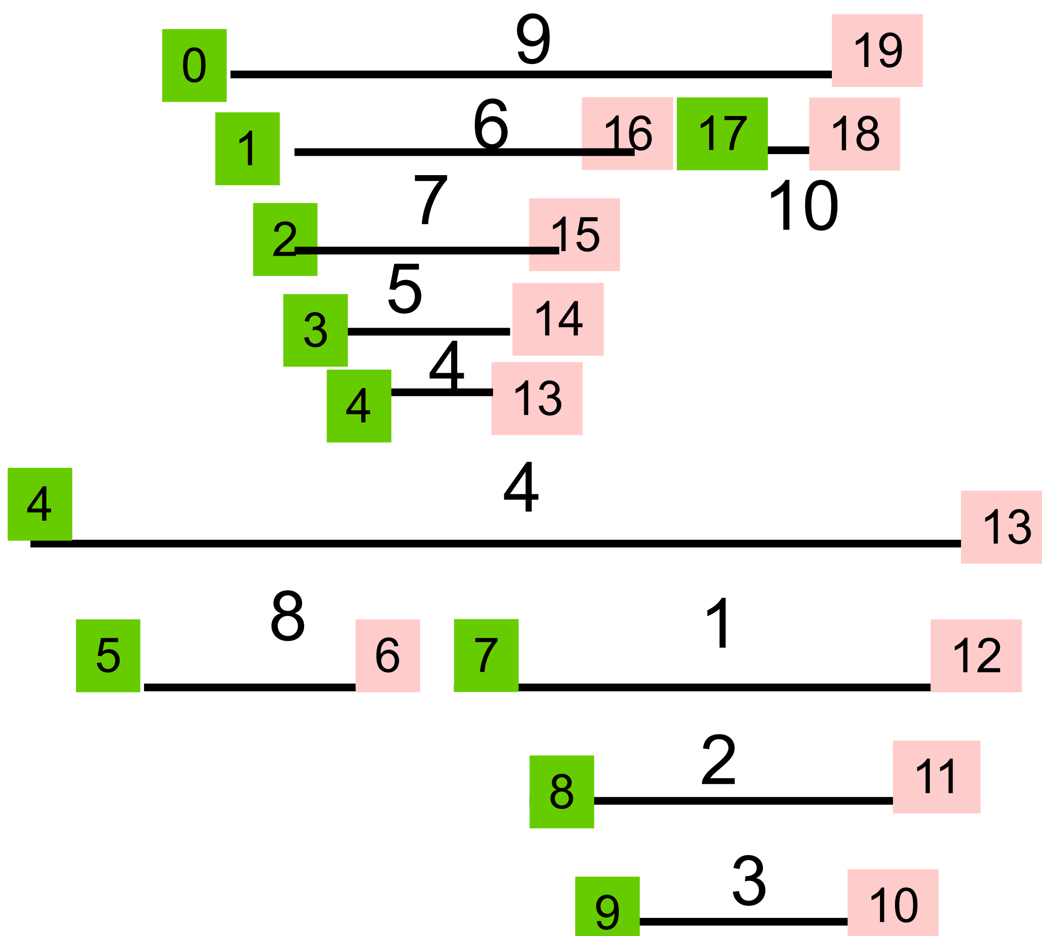
Can we do a topological sort of this graph using DFS?

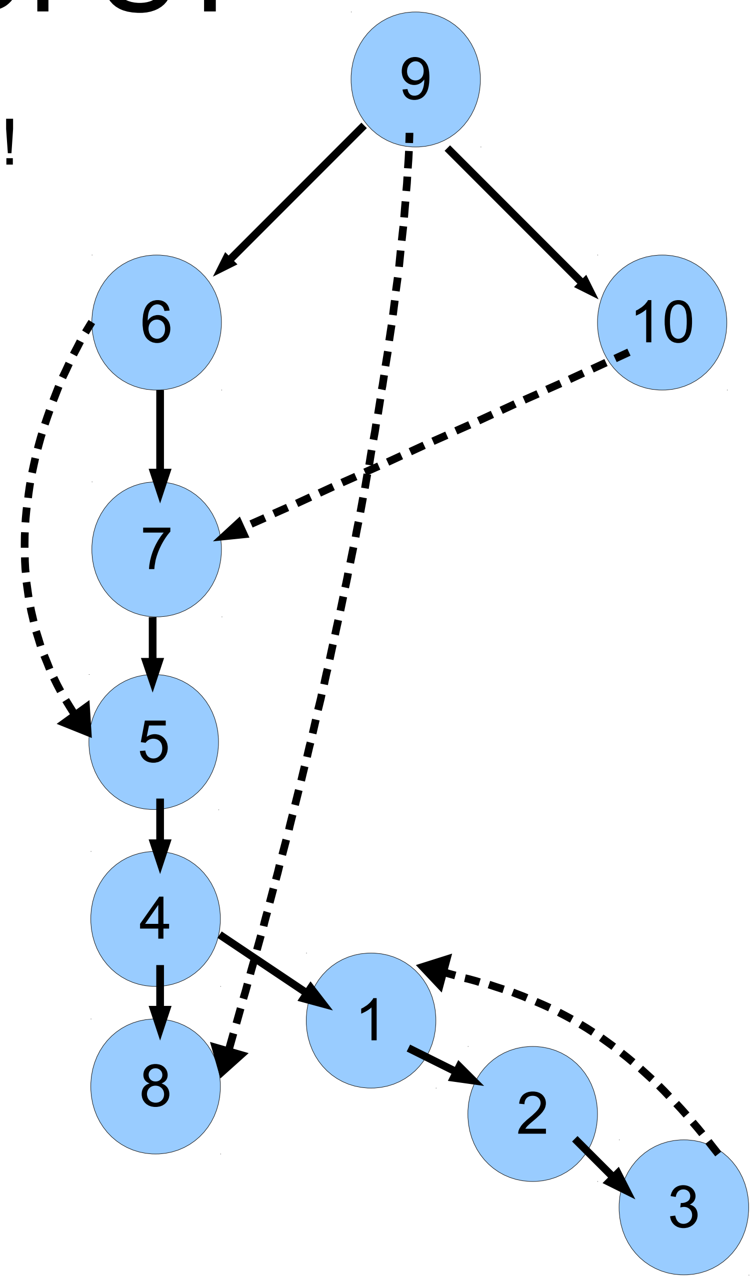
- See anything interesting in the pre and post values?



Can we do a topological sort of this graph using DFS?

- The order of the post values if reversed gives us a topological sort of the graph !!
- 9,10,6,7,5,4,1,2,3,8





The End