Graph Algorithm

Nattee Niparnan

Department of Computer Engineering
Chulalongkorn University

Overview

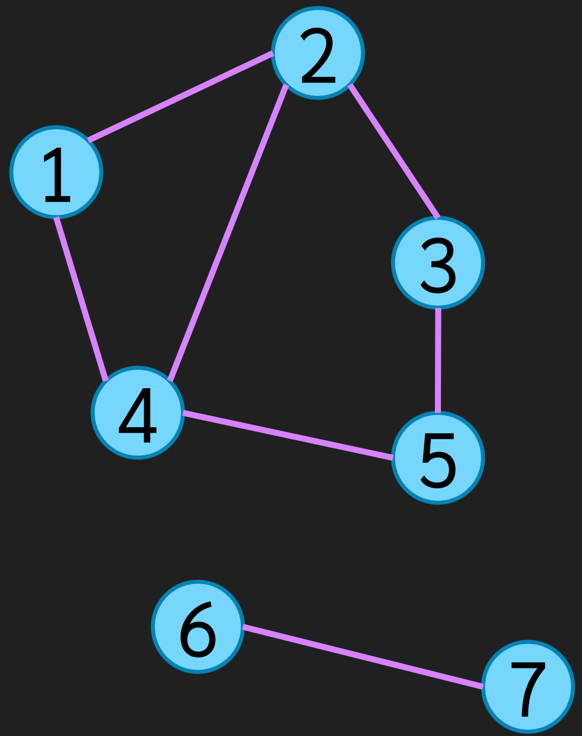
- Graph (and tree) is very useful in modelling several problem
- We will talk about several well-known algorithms relating to graphs
  - Checking connectivity: DFS, BFS, CC, SCC
  - Calculating Minimum Spanning Tree: Prim’s, Kruskal’s
  - Finding a shortest path: Dijkstra, Bellman-Ford, Floyd-Warshall

Graph Recap

- A graph G = (V,E) consist of a set of nodes V and a set of edges E
  - Each edge is a pair of vertices (a,b) indicating that there is a connectivity from a to b

V = (1,2,3,4,5,6,7)

E = ((1,2),(2,3),(3,5),(1,4),(4,5),(6,7))

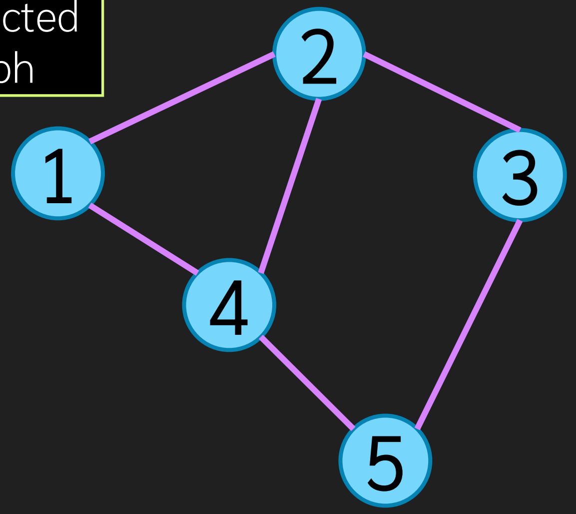


Graph Recap

- Undirected graph: having edge (a,b) is the same as having edge (b,a)
- Weighted graph: Each edge has a weight
- Path: sequence of nodes such that there is an edge for every adjacent pair of nodes in the sequence
  - Simple path: no duplicate nodes in the sequence
  - Circuit: path that the first and the last nodes in the sequence is the same
  - Simple circuit: no duplicate nodes in the sequence except the first and last
- Degree of a node: number of edges that connect to that node
- Simple graph: graph with no self-loop (edge connecting a same node) or duplicate edge (two or more edges connecting the same pair of node)
  - Most graph we use in this class is a simple graph

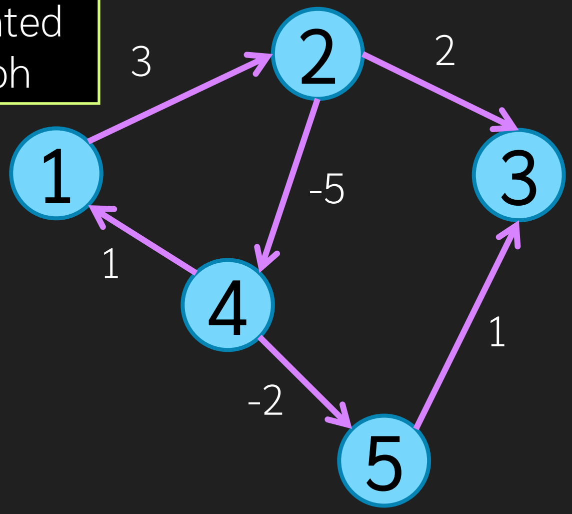
Sample Graph

Simple Undirected Graph



Path	Not a path
<1,2,4,5,3>	<1,3>
<4,2,1,4,5>	<7,8,9>
<1,4,5>	<3,2,1,5>
<1,2,1,2,1,2>	
<1,2,4,1>	
<4>	

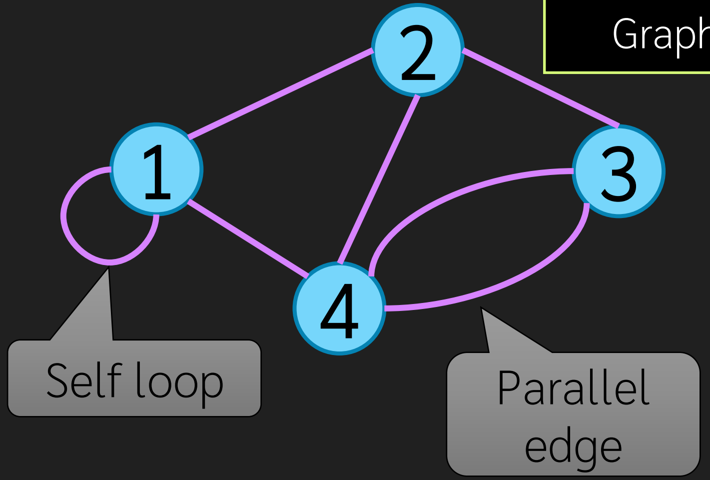
Simple Directed Weighted Graph



Path	Not a path
<1,2,4,5,3>	<1,3>
	<7,8,9>
	<3,2,1,5>
	<1,2,1,2,1,2>
<1,2,4,1>	
<4>	

Node	In Degree	Out Degree
1	0	2
2	1	2
3	2	0
4	2	1
5	1	1

Undirected Graph



Data Structure for Graph

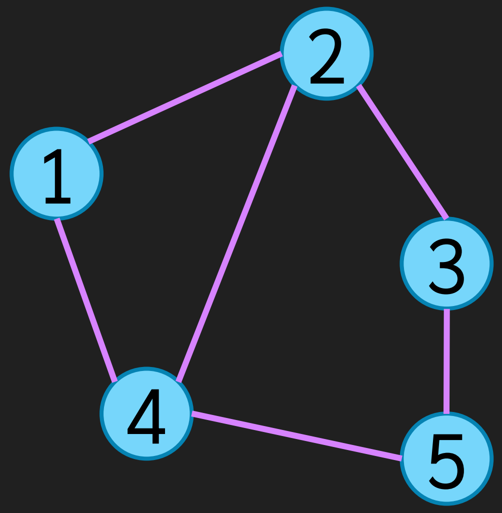
How to store a graph

Data Structure for a graph

- There is three basic operations when working with a static graph
  - nodes(): get a list of nodes in the graph
    - Usually, we name each node as 1 to n (or 0 to n-1). So, nodes() can be simplified to return just the number of nodes
  - adj(v1): Given a node v1, get a list of nodes such that there is an edge from v1 to them
  - has_edge(v1,v2): Given two nodes v1 and v2, return true only when there is an edge connecting v1 to v2
- For a dynamic graph, we also need an operation to add and remove nodes and edges

Adjacency Matrix

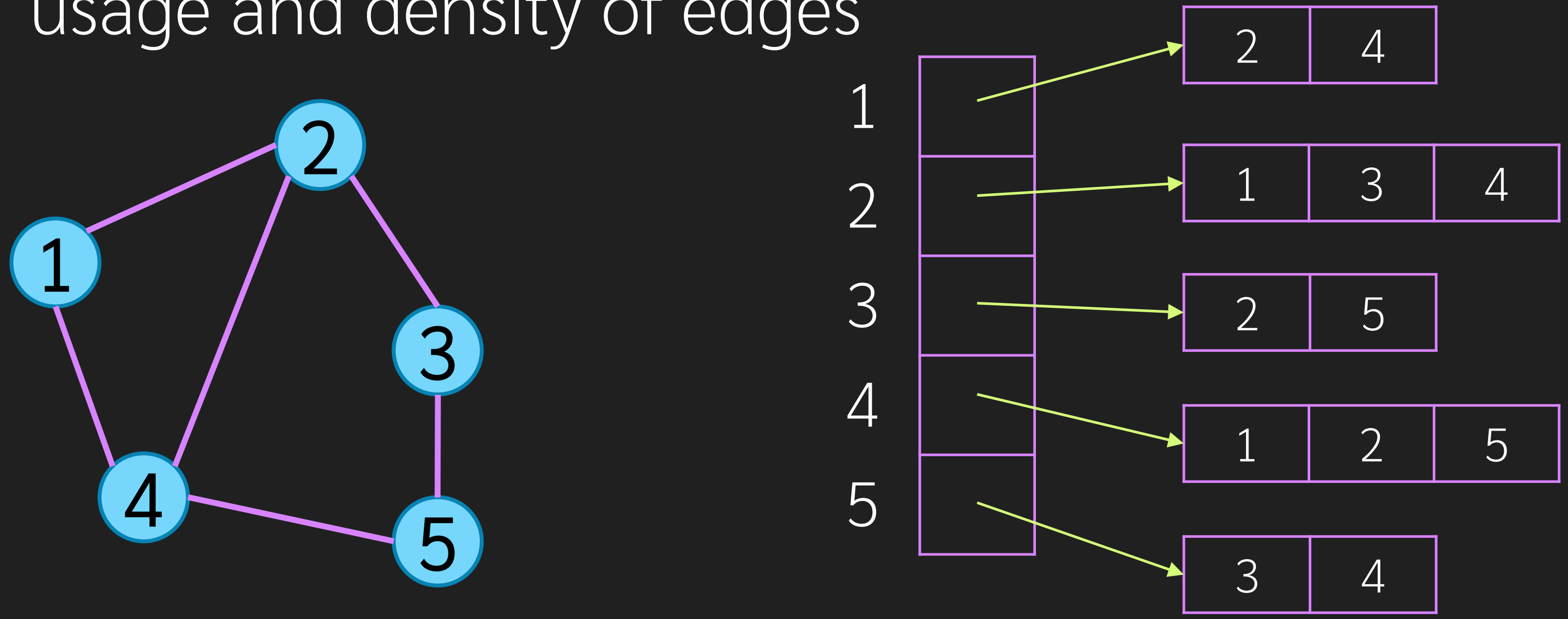
- Using 2D array A[1..n][1..n]
- A[x][y] = 1 when there is an edge connecting node x and node y



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	0	1
4	1	1	0	0	1
5	0	0	1	1	0

Adjacency List

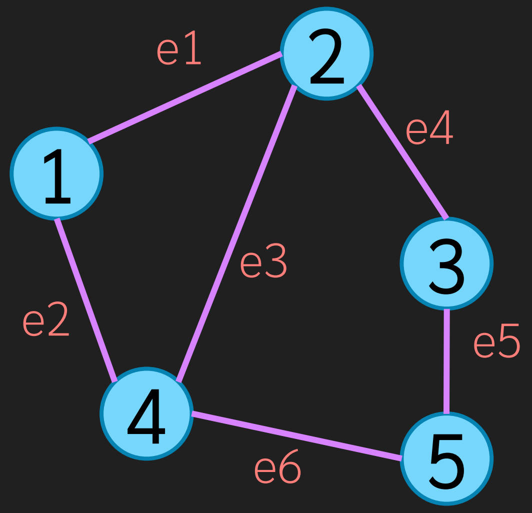
- A[1..n] = Array of a vector (or some other data structure)
- A[x] is a list of all neighbor of x
- We can use vector, BST, hash instead of list but it really depends on usage and density of edges



Incidence matrix

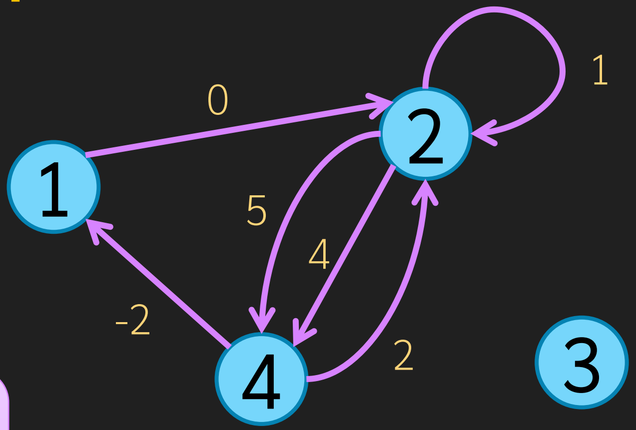
Incidence matrix is not used in our class, but it has nice property mathematical analysis

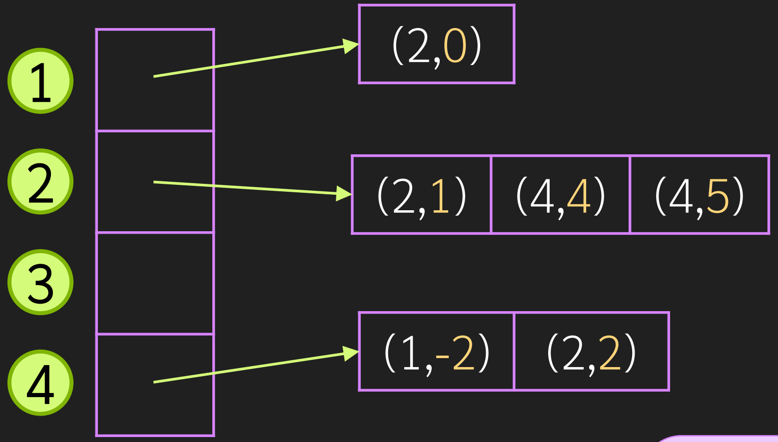
- A[1..n][1..e]
  - Row represent nodes while columns represent edges
  - A[v1][v1] = 1 when edge e1 connect nodes v1



	e1	e2	e3	e4	e5	e6
1	1	1	0	0	0	0
2	1	0	1	1	0	0
3	0	0	0	1	1	0
4	0	1	1	0	0	1
5	0	0	0	0	1	1

Example for non-simple directed weighted graph





We need to distinguish between no edge and zero weight. So, we need to use a vector

	1	2	3	4
1	<>	<0>	<>	<>
2	<>	<1>	<>	<4,5>
3	<>	<>	<>	<>
4	<-2>	<2>	<>	<>

Non-positive edges needs two scalar value, one for direction and another for weight

1	0	0	0	0	(0,0)	(1,0)
2	1	4	5	-2	(-1,-2)	(-1,0)
3	0	0	0	0	(0,0)	(0,0)
4	0	-4	-5	2	(1,-2)	(0,0)

Data Structure of a graph

- Let n = |V|, e = |E| and k be the size of the output of the operation
  - Remember that for a simple graph, e can be as high as n^2 and can be as low as 0
- Assume that we use a vector for an adjacency list

Not use

Operation	Adjacency Matrix	Adjacency List	Incidence Matrix
adj(v1)	O(n)	O(k)	O(ne)
has_edge(v1,v2)	O(1)	O(deg(v1))	O(e)
Add an edge	O(1)	O(1) (actually O(n))	O(n) (actually O(ne))
Remove an edge	O(1)	O(deg(v1) + deg(v2))	O(n) if we know the edge O(n+e) if we don't
Add a node	O(n) (actually $O(n^2)$)	O(1) (actually $O(n^2)$)	O(n) (actually O(ne))
Remove a node	$O(n^2)$	O(ne)	O(ne)

Not use

Finding a path

Given two nodes, check if there is a path between the nodes

Finding a path

- Problem: Given two nodes, check if there is a path between the nodes
- Input:
  - A graph $G = (V,E)$
  - Two nodes u and v
- Output:
  - True: when there is a path from u to v, False: otherwise
    - Sometimes, we need a path as well

Brute Force Approach

- Try all possible sequences of nodes
- What should be the maximum length of the sequence?

Candidate Solution	Set of candidate solution	Satisfaction condition
A sequence of vertex p[ ]	All permutations of length 1..n-1 of 1..N The size is $n!$	P[ ] is a path and p.first = u and p.last = v

V0.1: Super naïve brute force

```
def brute_path(p,idx,used,G)
  if idx < n
    if p[idx-1] == v
      if is_a_path(G,p)
        ans = p
        return true
    for b in 1..n
      if used[b] == false
        used[b] = true
        p[idx] = b
        if brute_path(p,idx+1,used)
          return true
        used[b] = false
    return false
```

Check if this is a path from u to v

Standard permutation

```
def is_a_path(G,p)
  for i in 1..(p.length-1)
    unless G.has_edge(p[i],p[i+1])
      return false
  return true
```

- See that a path from any two nodes has length at most n-1
- Start with `brute_path( {u} ,1, used, G )` where `used[u] = true`
- While generating all permutations, we also check if it is a path
- If we found a path, just return (and makes calling function return as well)
- Return false when all possible permutation is not the desired path

V0.2: Better brute force

```
def brute_path(p,idx,used,G)
  if idx < n
    if p[idx-1] == v


        return true
    for b in G.adj(p[idx-1])
      if used[b] == false
        used[b] = true
        p[idx] = b
        if brute_path(p,idx+1,used)
          return true
        used[b] = false
    return false
```

No need to check if P is a path (because it always is)

Generate only a path, we pick b which is adjacent to the last node in the path

- With very simple modification, we can have better (much smaller) candidate solution set
- Generate only all possible path instead of all possible sequence
- Much better when a graph has small number of edges

Observation

- When we start from u and, later, we find a path from u to some node x where x is not the target vertex (v)
  - Now, the problem is to check if there is a path from x to v
  - In other words, if there is a path from x to a node y, there must be a path from u to y as well. Similarly, How we get from u to x does not really affect the answer
- Instead of searching for a specific path from u to v, we search for any node reachable from u
  - Will this be any faster? Does it seem like we are doing more than what we have been asked to do?

Depth First Search

Visiting nodes in a graph

Actual Solution to Find a Path Problem

- The Depth First Search (DFS) algorithm
  - This will be a basis for several other algorithms in the series

- Start with `dfs(u,G,visited[1..n] = [false])`
- When done, visited[x] == true only when there is a path from u to x
- Can modify to stop when we reach the target v
- If there is multiple instance of the problem, it may be better to use the original version and check the visited[ ] directly for each question

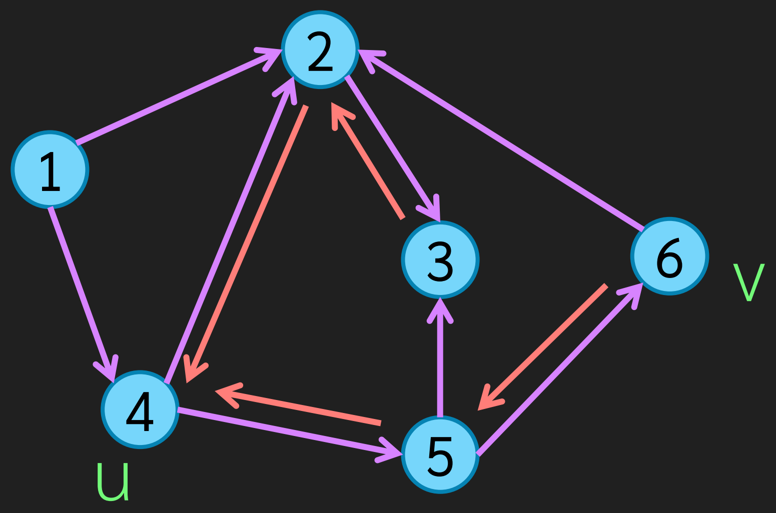
```
def dfs(a,G,visited)
  visited[a] = true
  for b in G.adj(a)
    if visited[b] == false
      dfs(b,G,state)
end
```

```
def dfs_path(a,G,visited)
  visited[a] = true
  if (a == v)
    return true
  for b in G.adj(a)
    if visited[b] == false
      if dfs_path(b,G,visited)
        return true
  return false
end
```

Finding a path

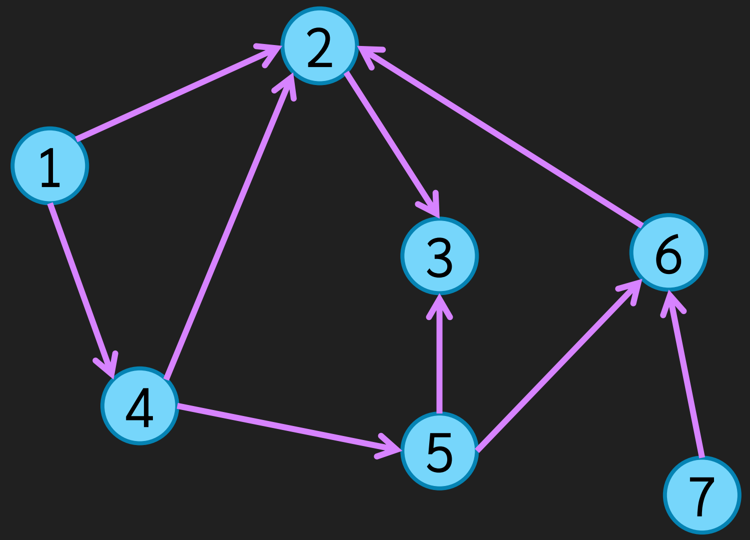
- Set up `from[1..n]`, initialized with -1
  - `from[x]` is a node that is used to reach x
  - Use `from[ ]` to re-trace the path
    - Start with v, follow from until we reach u, reverse the list of traversed path

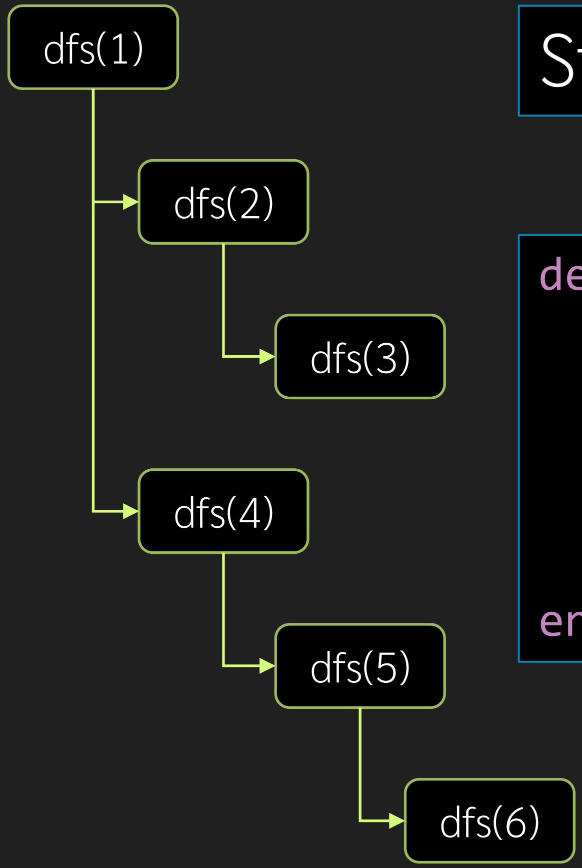
```
def dfs(a,G,visited,from)
  visited[a] = true
  for b in G.adj(a)
    if visited[b] == false
      from[b] = a
      dfs(b,G,visited)
end
```



	1	2	3	4	5	6
visited	false	true	true	true	true	true
from	-1	4	2	-1	4	5

Let's see it in action





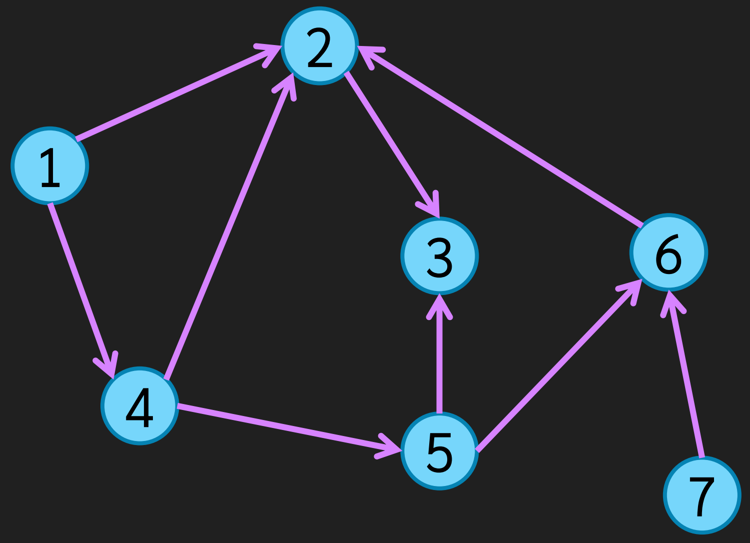
Start with `dfs(1,G)`

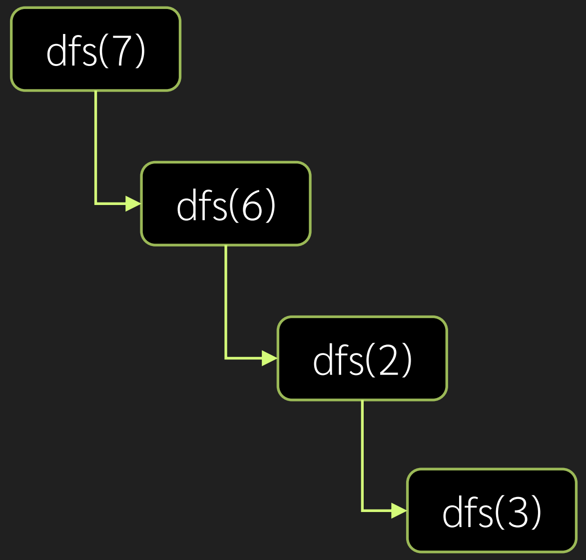
```
def dfs(a,G,visited,from)
  visited[a] = true
  for b in G.adj(a)
    if visited[b] == false
      from[b] = a
      dfs(b,G,visited,from)
end
```

	1	2	3	4	5	6	7
visited	true	true	true	true	true	true	false
from	-1	1	2	1	4	5	-1

Let's see it in action again

Start with `dfs(7,G)`






```
def dfs(a,G,visited,from)
  visited[a] = true
  for b in G.adj(a)
    if visited[b] == false
      from[b] = a
      dfs(b,G,visited,from)
end
```

	1	2	3	4	5	6	7
visited	false	true	true	false	false	true	true
from	-1	6	2	-1	-1	7	-1

DFS using stack

- It is possible to achieve the same result without recursion
- Instead of relying on a program call stack, we directly use a stack to implement

```
def dfs_by_stack(a,G)
  s = new Stack
  s.push(a)
  visited[a] = true
  while (s.size > 0)
    u = s.top
    s.pop
    for b in G.adj(u)
      if visited[b] == false
        visited[b] = true
        s.push(b)
end
```

Question:

1) Does this work just like the recursive version
2) How to modify it to calculate from[ ]

Time Complexity

```
def dfs_by_stack(a,G)
  s = new Stack
  s.push(a)
  visited[a] = true
  while (s.size > 0)
    u = s.top
    s.pop
    for b in G.adj(u)
      if visited[b] == false
        visited[b] = true
        s.push(b)
end
```

- There is a while loop
- Each iteration, we pop a node, and each node enters a stack at most once
  - So, there is $O(n)$ iterations in the loop
- There is inner for loop for each iteration
  - Each for loop runs at most $O(n)$ (because each node has at most $n - 1$ neighbors)
- So, it is $O(n^2)$
- However, a better analysis is $O(n+e)$
  - Why?
- Does recursive version give the same $O(n+e)$

Breadth First Search

Another algo for exploring nodes in a graph

Using a queue

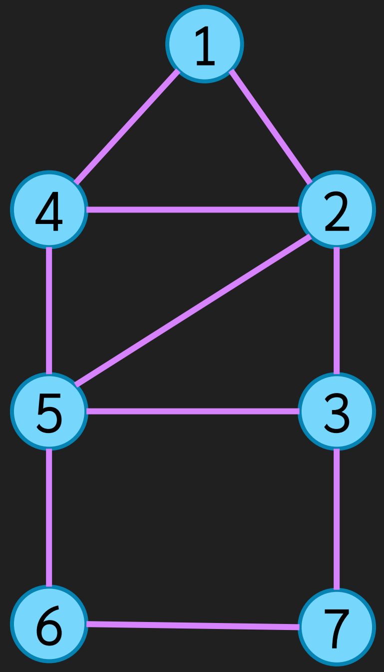
- Consider a DFS using a stack, what will happen if we replace a stack with a queue?
  - Is the result still the same? What is the different?

```
def dfs_by_stack(a,G)
  s = new Stack
  s.push(a)
  visited[a] = true
  while (s.size > 0)
    u = s.top
    s.pop
    for b in G.adj(u)
      if visited[b] == false
        visited[b] = true
        s.push(b)
end
```

```
def by_queue(a,G)
  q = new Queue
  q.push(a)
  visited[a] = true
  while (q.size > 0)
    u = q.front
    q.pop
    for b in G.adj(u)
      if visited[b] == false
        visited[b] = true
        q.push(b)
end
```

Let's try

- Start with `dfs_by_stack(1,G)`
- Track the order that a node change to visited (pushed into a stack)

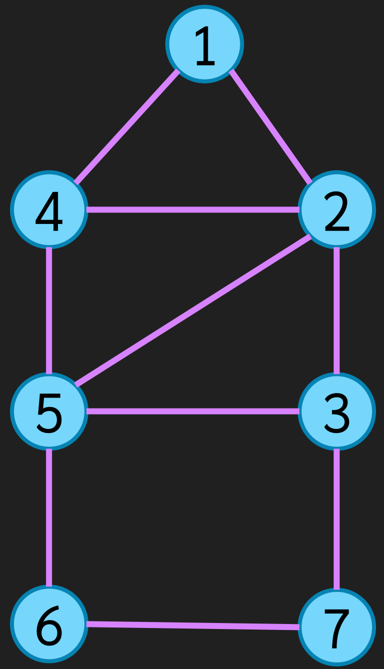



```
def dfs_by_stack(a,G)
  s = new Stack
  s.push(a)
  visited[a] = true
  while (s.size > 0)
    u = s.top
    s.pop
    for b in G.adj(u)
      if visited[b] == false
        visited[b] = true
        s.push(b)
end
```

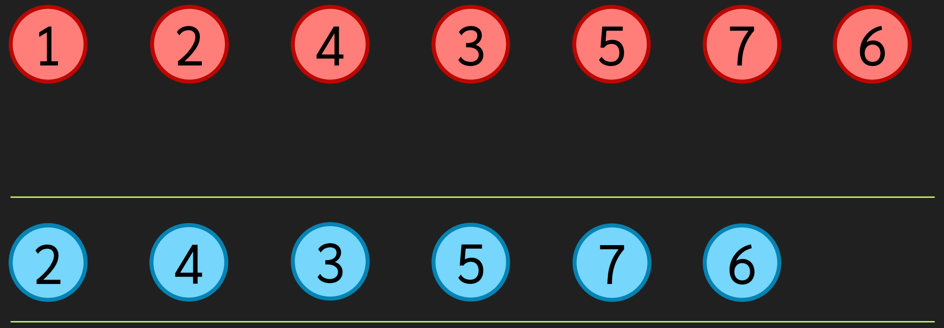
1 2 4 5 3 6 7

Let’s try

- Start with **by_queue(1,G)**
- Track the order that a node change to visited (pushed into a queue)

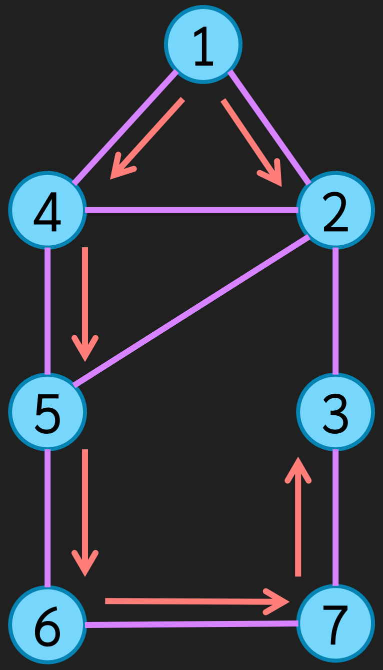


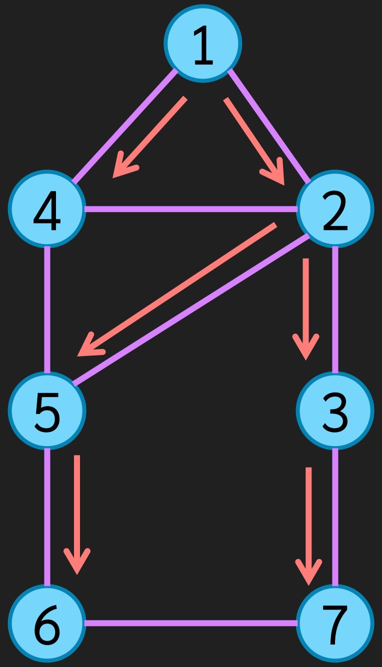

```
def by_queue(a,G)
  q = new Queue
  q.push(a)
  visited[a] = true
  while (q.size > 0)
    u = q.front
    q.pop
    for b in G.adj(u)
      if visited[b] == false
        visited[b] = true
        q.push(b)
end
```



Compare

DFS





BFS

- DFS tries to go as deep as possible
- BFS tries to cover nodes of the same distance

Breadth First Search

- The order of nodes going into the queue depends on the distance from the starting position
- Breadth First Search is used to find a shortest path on an unweighted graph

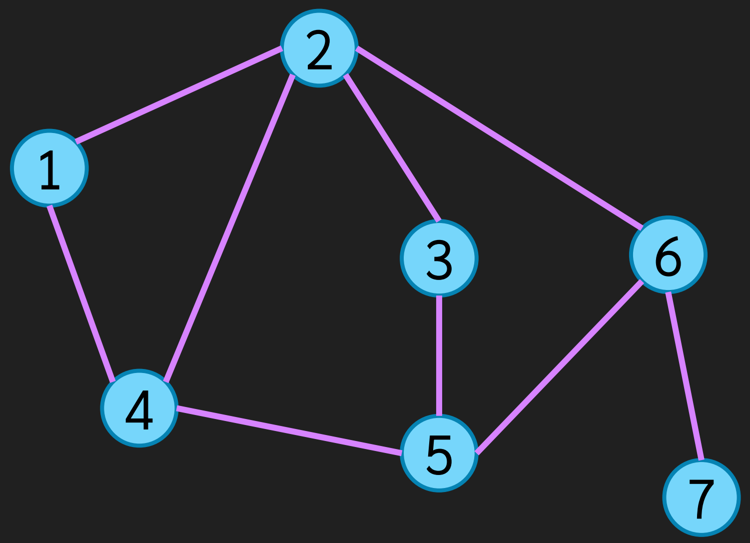
BFS with Distance

```
def bfs(a,G)
  dist[ ] = (-1,...)
  q = new Queue
  q.push(a)
  dist[a] = 0
  while (q.size > 0)
    u = q.front
    q.pop
    for b in G.adj(u)
      if dist[b] == -1
        dist[b] = dist[u] + 1
        s.push(b)
end
```

- `dist[x]` = distance (minimum number of edges we need to go through in the shortest path from u to x)
- Can you modify it to find the shortest path as well?

Distance Example

```
def bfs(a,G)
  dist[ ] = (-1,...)
  q = new Queue
  q.push(a)
  dist[a] = 0
  while (q.size > 0)
    u = q.front
    q.pop
    for b in G.adj(u)
      if dist[b] == -1
        dist[b] = dist[u] + 1
        s.push(b)
end
```



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

Analysis

- Time Complexity
  - Is it the same as in the case of a stack (DFS)?
  - Why? Why not?
- Memory Usage
  - At worse, both may require up to $O(N)$ nodes to be stored in the stack / queue
    - A line graph for DFS
    - A star graph for BFS
  - In practice, a k-ary thee with depth d is very common
    - DFS use $k * d$, because it must remember all children of each unfinished visiting node.
    - BFS use k^d , because it has to remember all nodes of the same depth

Connected Component

Graph Partitioning

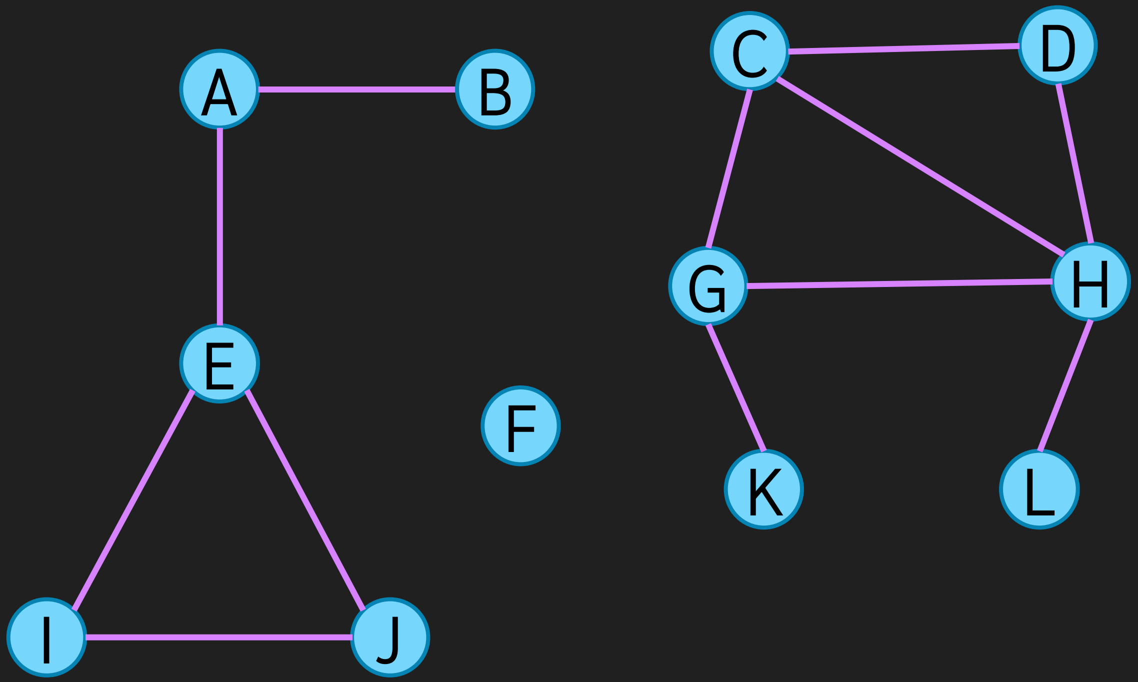
Connected Component

- Having a path from a to b in an undirected graph is the same as having a path from b to a
- Also, having a path from a to b and from b to c means having a path from a to c
- Hence, have-a-path (connected) is an equivalent relation
- We can group nodes according to this relation
- Each group is called connected component

Connected Component

- Problem: Given an undirected graph, partitions nodes into groups such that each group is connected (there is a path between every pair of nodes in the same group)
- Input:
  - An undirected graph G = (V,E)
- Output:
  - cc[v], a label for each nodes such that if cc[a] == cc[b], it means that a and b is in the same group

Example



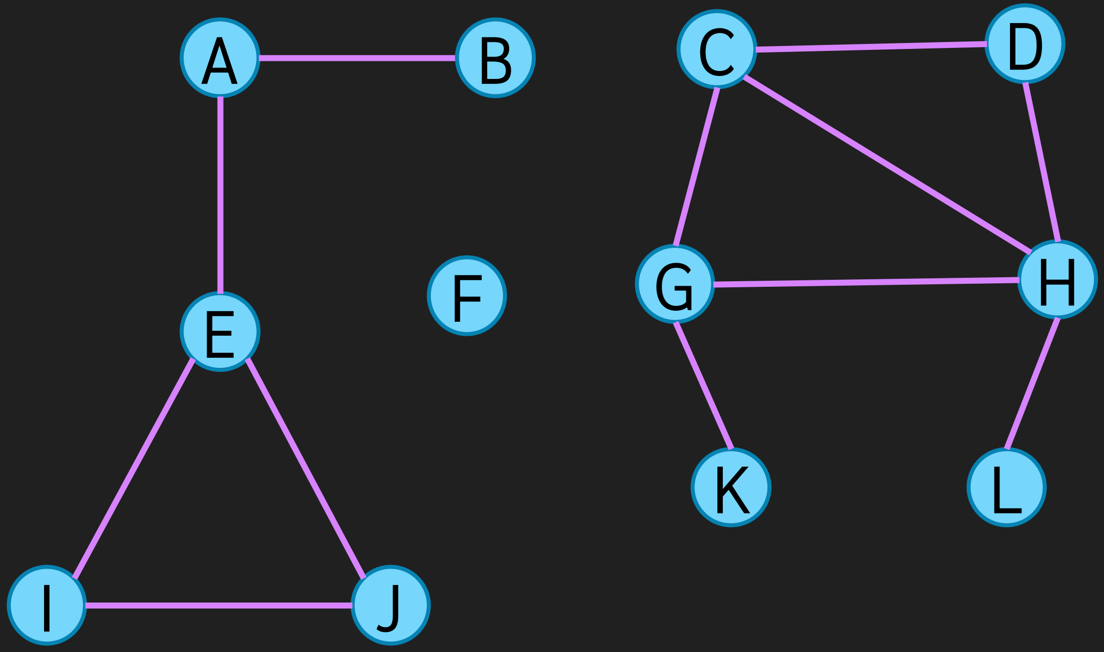
- Connected component is not applicable for a directed graph
  - We will use strongly connected component

Using DFS

```
def dfs_cc(a,G,visited)
  visited[a] = cc_num
  for b in G.adj(a)
    if visited[b] == 0
      dfs(b,G,visited)
end

def cc(G)
  cc_num = 0
  visited[] = [0,...]
  for every u in G
    if visited[u] == 0
      cc_num += 1
      dfs_cc(u,G,visited)
end
```

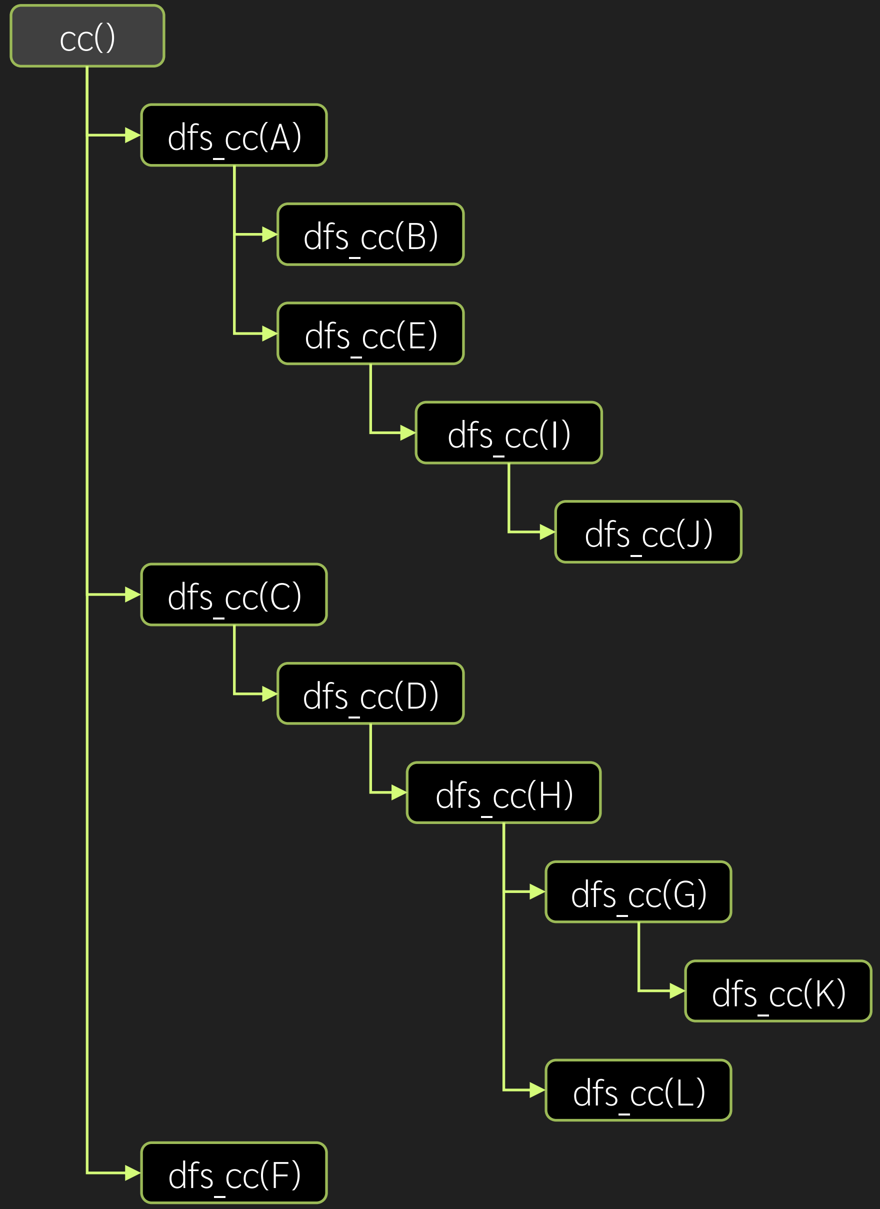
- Directly apply DFS on every nodes in the graph
- Set visited[ ] as integer instead of Boolean
  - Notice that `cc_num` is used to label any reachable nodes and it is increase when we found a node that is not reachable from earlier iteration
- `visited[x]` is the index of the component of x

Tracing Connected Component




```
def cc(G)
  cc_num = 0
  visited[] = [0,...]
  for u in V
    if visited[u] == 0
      cc_num += 1
      dfs_cc(u,G,visited)
end
```

```
def dfs_cc(a,G,visited)
  visited[a] = cc_num
  for b in G.adj(a)
    if visited[b] == 0
      dfs(b,G,visited)
end
```



Analysis

- It runs DFS multiple times
  - At worse, it must run DFS on every node.
  - Since DFS is $O(n+e)$, hence, in total, it might be $O(n\,(n+e))$
  - Is this correct?
- Actually, the entire connected component algorithms takes just $O(n+e)$
  - Can you see why?
- Can we use BFS instead of DFS?

Detecting a Cycle

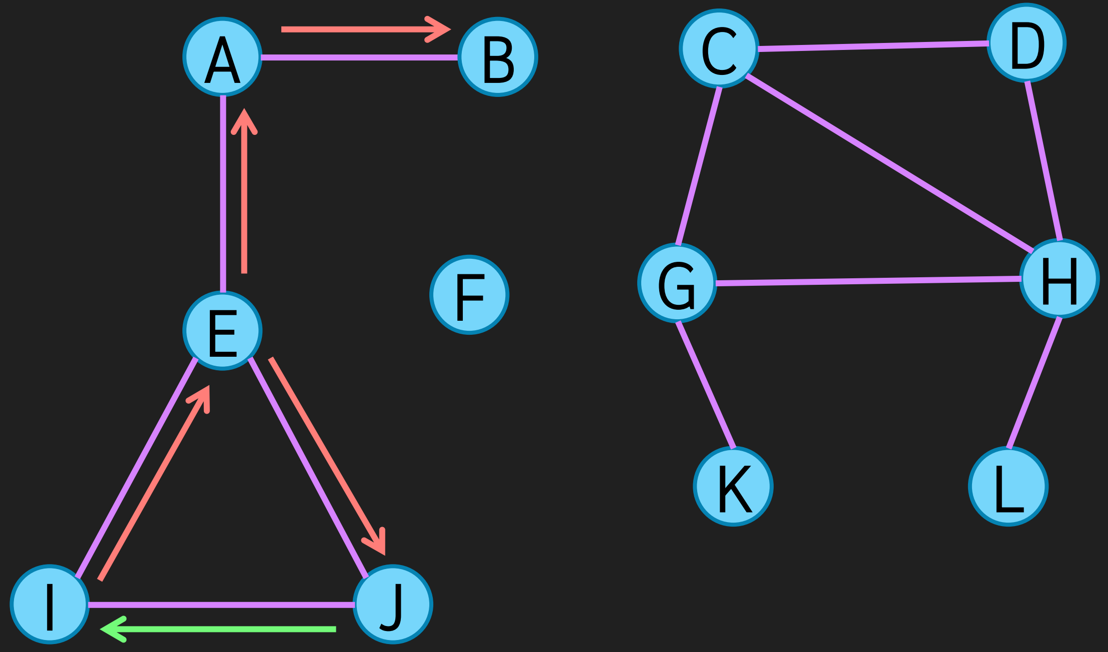
Another application of {B,D}FS

Problem Definition

- Problem: Given a graph, check whether there is a circuit in the graph
- Input:
  - A graph $G = (V,E)$
- Output:
  - True, when there is a circuit in a graph. False, otherwise

Observation

- Circuit must be in a connected component
- Use DFS, we found a cycle when G.adj(u) contains a node that is already visited which is not used to directly reach u.



The code

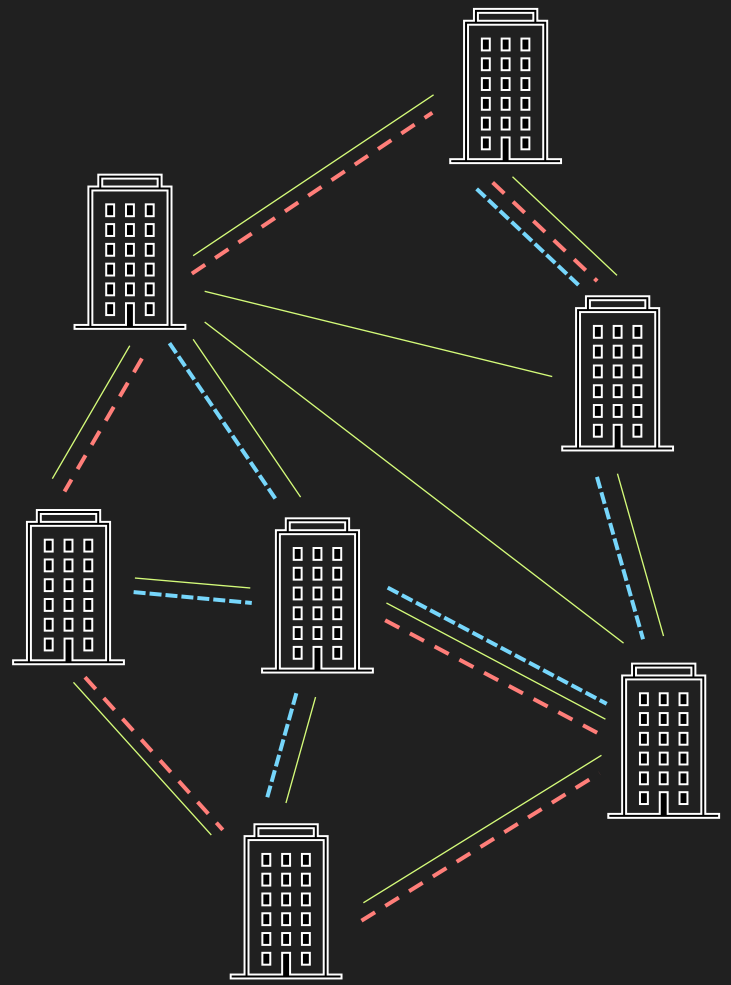
```
def dfs_cd(a,G,visited,parent)
  visited[a] = true
  for b in G.adj(a)
    if visited[b] == 0
      dfs(b,G,visited,a)
    else if b != parent
      return true
  return false
end

def circuit_detect(G)
  visited[] = [false,...]
  for every u in V
    if visited[u] == false
      if dfs_cd(u,G,visited,-1)
        return true
  return false
end
```

- `parent` is the node that is used to visit `a`.
- The time complexity is still O(n+e)
- This method works in directed graph as well.

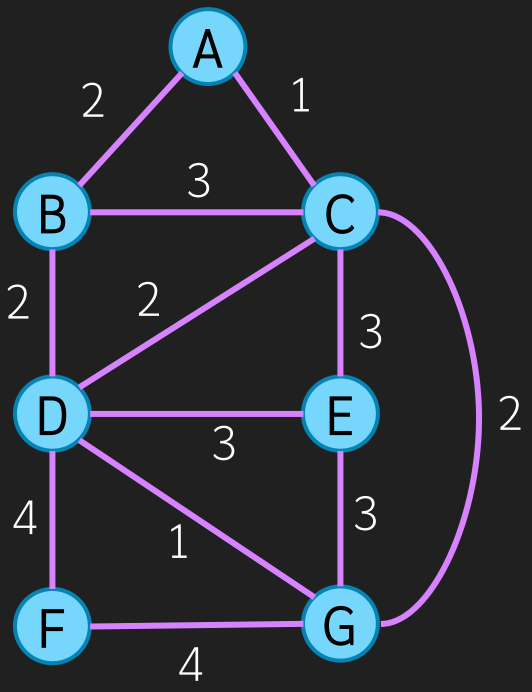
Minimum Spanning Tree

Minimum Spanning Tree

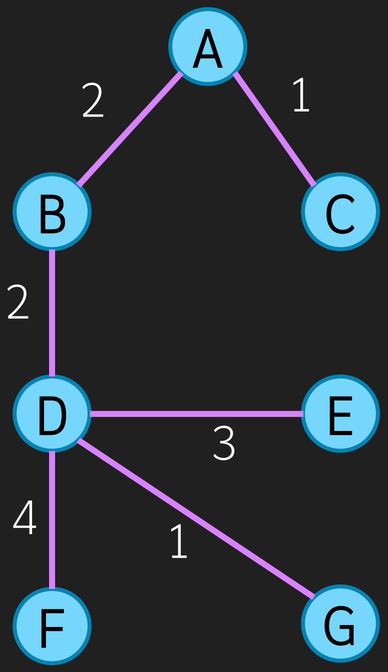
- Consider a campus of n buildings. We want to wire a computer network connecting these buildings. We have survey every pair of building to check if it is possible to put a network cable connecting the pair. Some pair of building can be wired while some can't.
    - We can model this as a graph. A node is a building and there is an edge connecting two buildings that we can wired. If there is a path in this graph, it means that we can transmit a data between building in the path.
    - The goal is to have a graph that is connected (only one connected component) so that any pair of building can communicate.
    - The graph is undirected because the cable can be used to bidirectionally.
- Since the cables are not free, we wish to minimize the cost while maintaining connectivity. The spanning tree problem is to choose as few as possible edge from the graph such that it is still connected.
    - This simply is picking edges that make a tree in the graph
    - It is a tree because it is a connected graph with least number of edges
- For Minimum spanning tree problem. The original edges has some weight associated. The weight is the cost of that edge.
    - We want to find a spanning tree such that the sum of the weighted of its edges is minimum.



Example



Input Graph



MST

- MST selects n-1 edges from the graph
  - Connected
  - Minimum summation of weight

Why Tree?

- MST should not have a cycle
  - Removing edge in a cycle does not destroy the connectivity
  - So why bother having an edge in a cycle in the MST
- Recall a property of a tree
  - A tree with n nodes has $n - 1$ edges
  - A connected graph having $n - 1$ edge is a tree
  - i.e., tree is the smallest structure that is connected

MST Problem

- Problem: find an MST from a graph
- Input:
  - A weighted graph $G = (V,E)$
  - Let assume that we have w as a weight function where $w((a,b))$ returns a weight of an edge (a,b)
- Output:
  - A subset of E that is the MST of the given graph

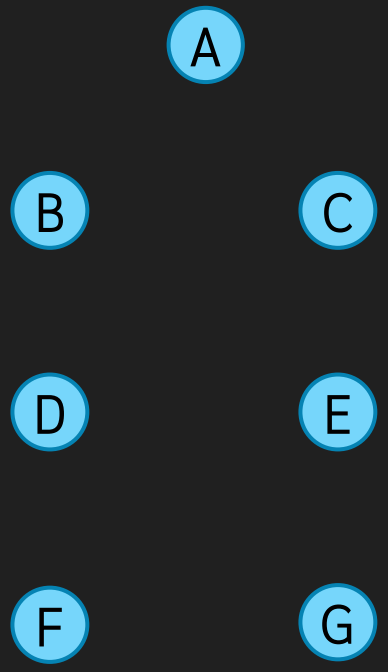
Kruskal's Algorithm

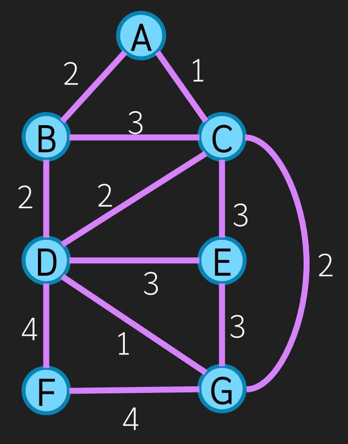
First Method to calculate an MST

Kruskal's Algorithm

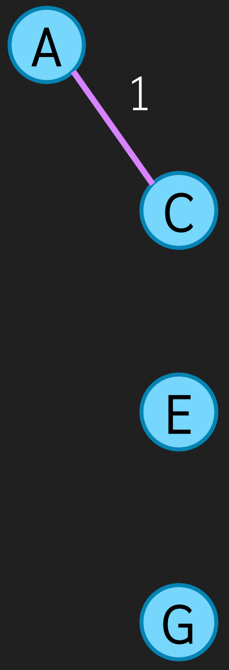
- Idea
  - Start with the graph $G = (V,\{\})$ (all nodes with no edges)
  - Since we need n-1 edges, simply pick them one by one
  - There are n-1 iterations. On each iteration, we select the edge having smallest weight that does not make a cycle in the graph

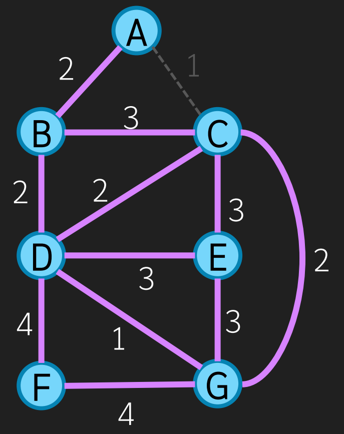
Example



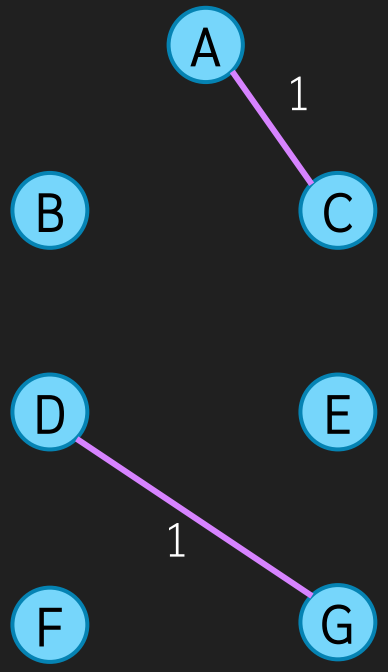


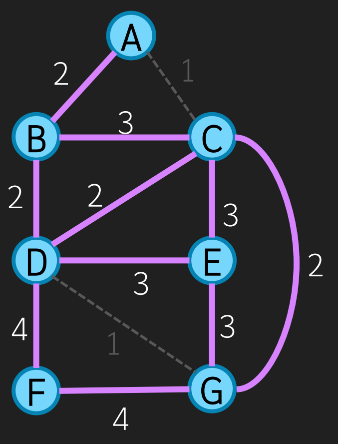
Example



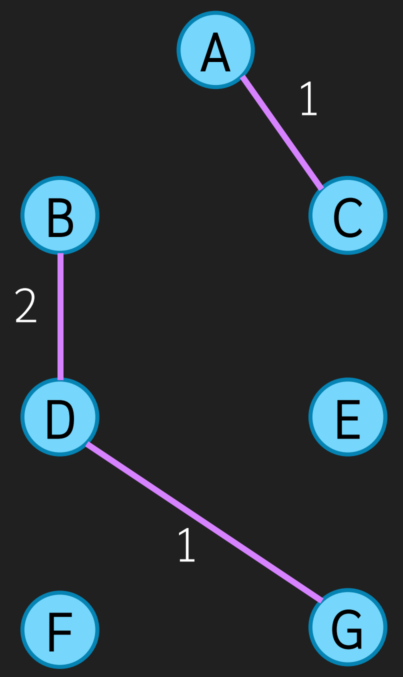


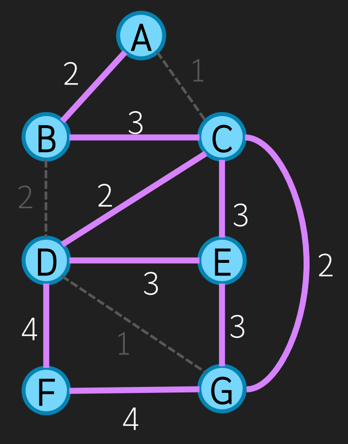
Example



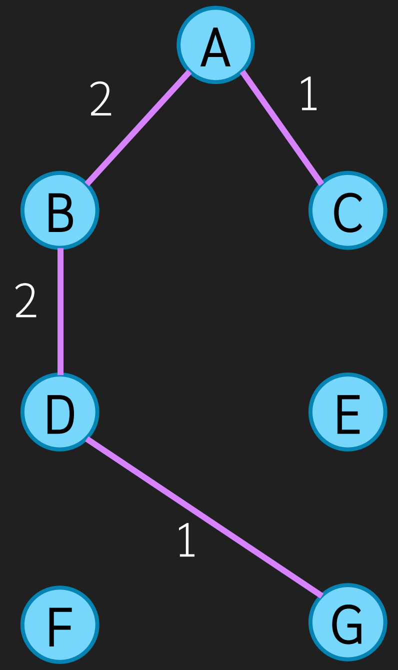


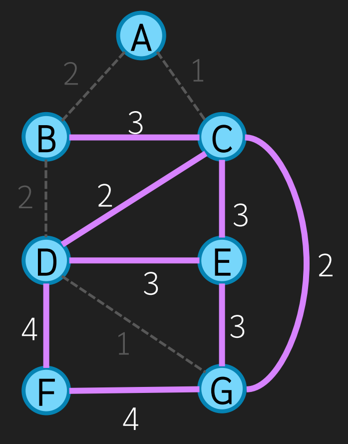
Example



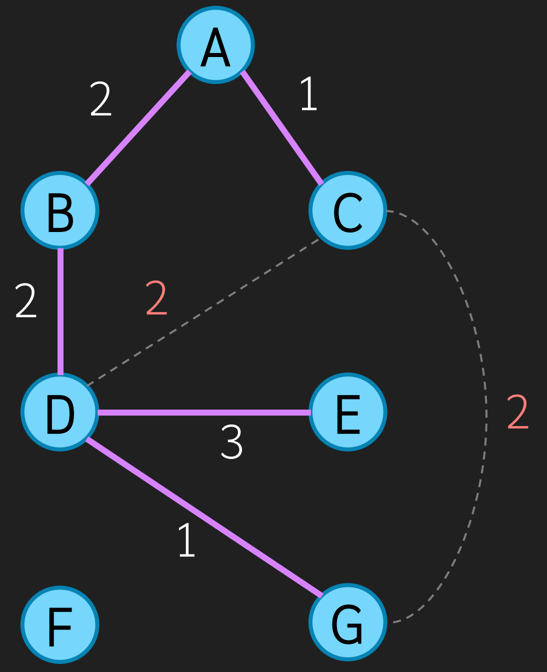


Example

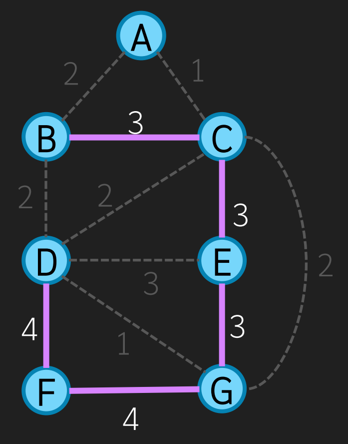




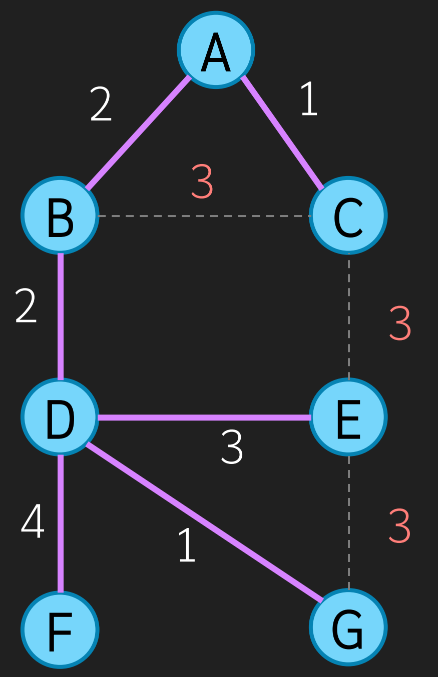
Example



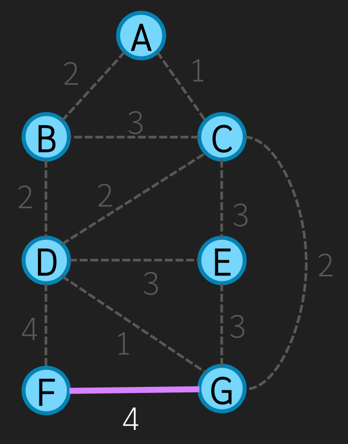
Can't select (D,C) or (C,G) even thought their weight is smallest (2), because it will cause a cycle. We have to pick (D,E) with weight 3 instead



Example



Same for
(B,C), (C,E) and (E,G)



Implementing Kruskal's

- For each iteration, we need some method to check the connected component of the graph
  - The selected edge must connect two different connected components.
  - The two components are joined into one connected component.
  - Can be done by recalculating CC on every step of Kruskal but that takes too much time $O(n(n+e))$
- We will use a data structure called Disjoint Set instead
  - Each node is a member of a set which represent a connected component
  - Initially each node is in its own set
  - When consider an edge, check if the endpoints of the edge belong to different set.
  - When choosing an edge, union the sets of its endpoints.

We need a data structure that is be able to

• findset(x)	find a set where x is a member
• union(x,y)	union a set containing x and a set y

Implementing Kruskal’s

```
def kruskal(G,w)

  ds = new DisjointSet
  ds.makeset(n)

  X = [] //X is the answer
  Sort the edges G.E by weight w
  for each edge (u,v) in G.E (in increasing order of weight)
    if ds.findset(u) != ds.findset(v)
      X.add(u,v) //add (u,v) to the answer
      ds.union(u,v)
  return X
end
```

Analysis

- Time complexity
  - O(e lg e) sorting the edges by weight
  - It also need
    - e findset
    - n-1 union
- What is the eventual complexity?
  - Depends on implementation of the set
  - We will use the Disjoint Set Data Structure

Disjoint Set Data Structure

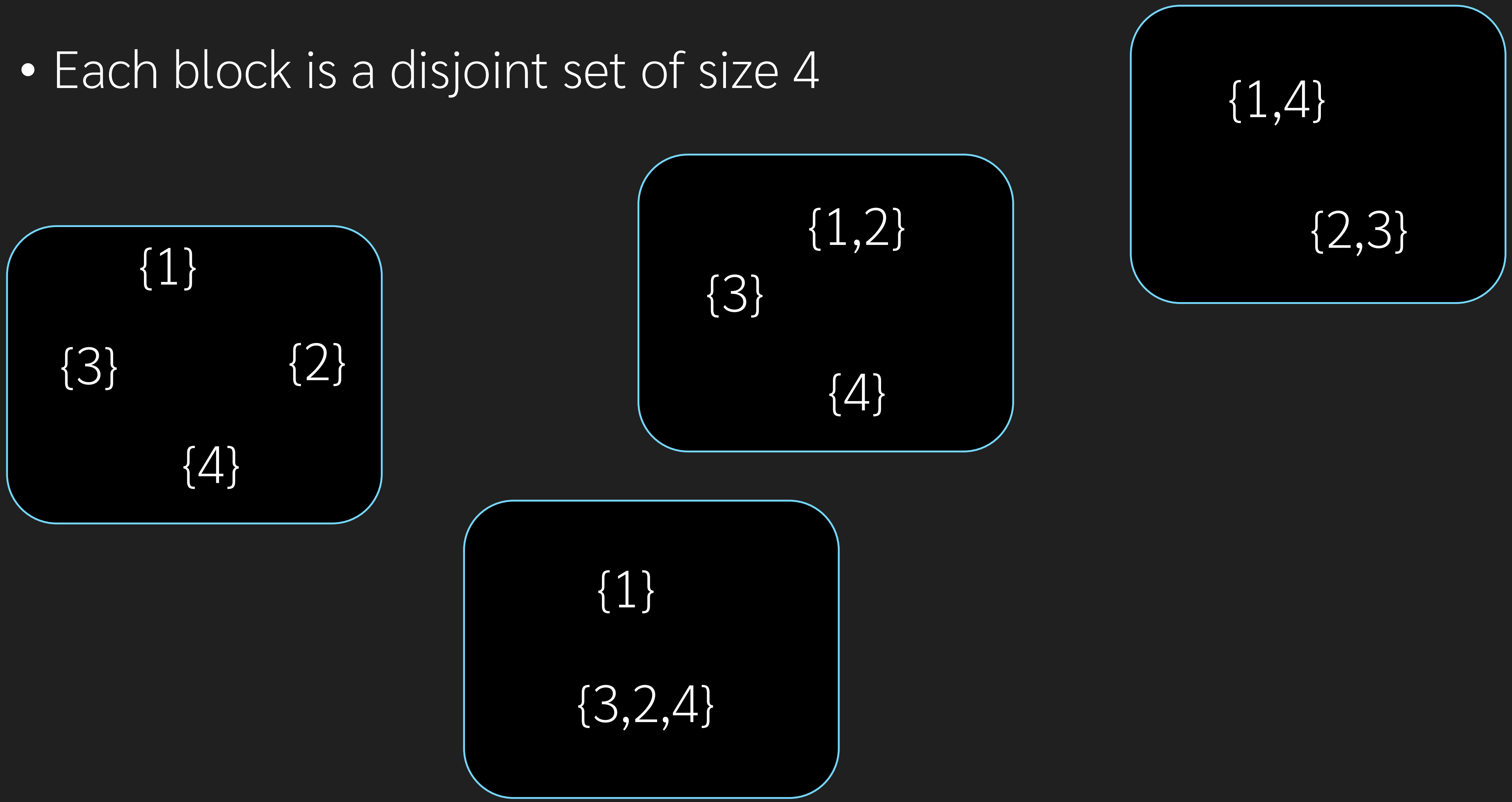
Disjoint Set Data Structure

- There are n elements, numbered 1 to n
- Each element must be in exactly one set
  - Initially each number is in its own set
- We can union two sets

- Operation
  - makeset(n) — create n sets for element 1..x, each set contain each element
  - findset(x) — return the ID of a set where x is a member
If x and y are in the same set, findset(x) must be equal to findset(y)
  - union(x,y) — union the set findset(x) with the set findset(y)

Example

- Each block is a disjoint set of size 4



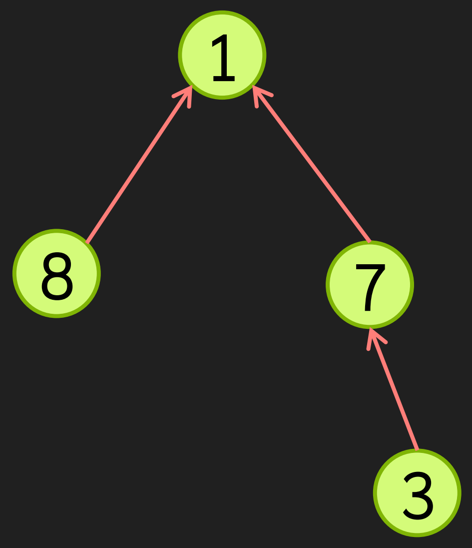
Data

- We must store
  - N to see how many element do we have and what are them (1..n)
  - What is the set of each member
- See that we never ask "what are the member of this set" but only "is X and Y in the same set"

Store Set as a Tree

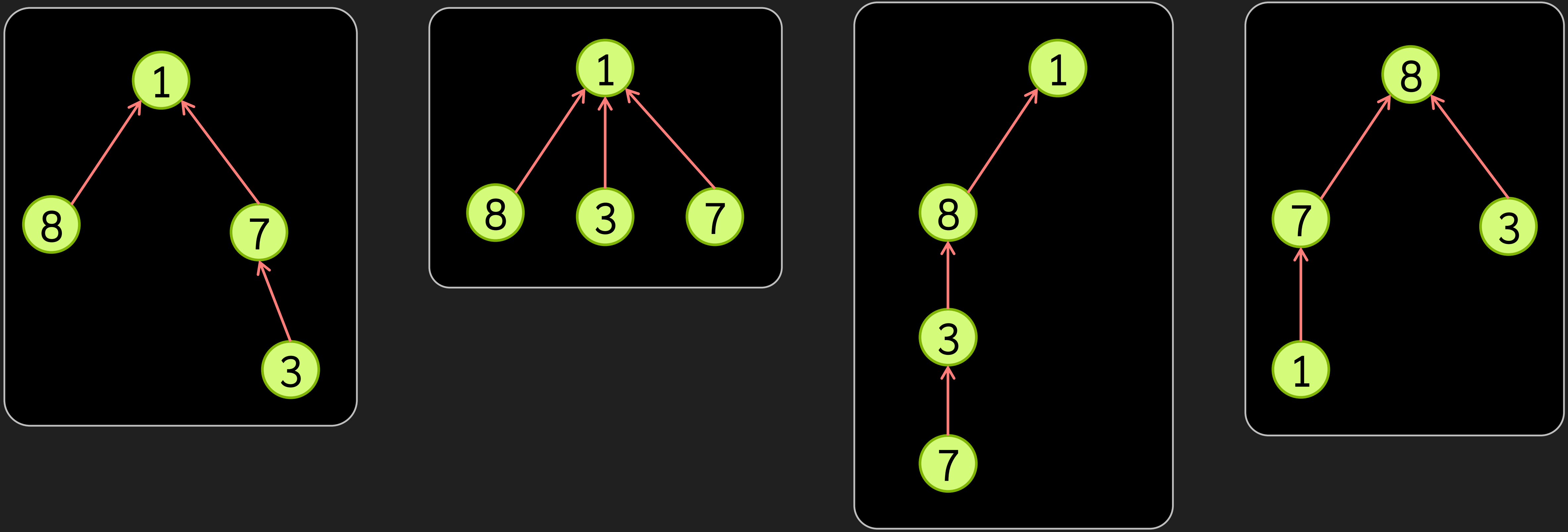
- Pick one member of a set as a root
- The other members are descendant of the root
- Store only the parent, no need to have a pointer to children

A set of {1, 8, 7, 3}



Same Set can be stored in many ways

A set of {1, 8, 7, 3}



Data

- Store as an array p where p[x] is the parent of node x
- makeset(n) is just creating an array [-1,...]

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

find(x) Operation

- Goes up the parent until we reach the root
- If p[x] is -1, x has no parent and is the root of the set. In this case we use x as the name of the set.

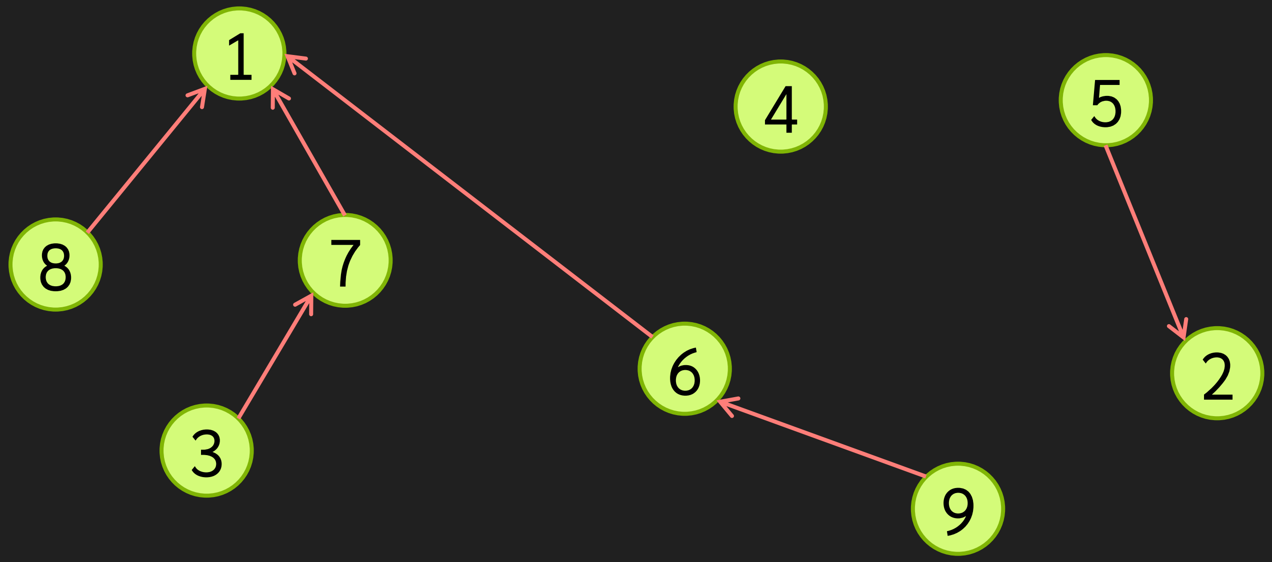
```
def findset(x)
  while p[x] != -1
    x = p[x];
  return x;
end
```

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

Union(x,y) operation

- Make root of one set points to another set

```
def union(x, y)
  s1 = findset(x)
  s2 = findset(y)
  p[s1] = s2
end
```



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

union(9,6)
union(5,2)
union(9,3)

Analysis

- findset(x) O(h)
- union(x,y) O(h)

- h is the height of the tree
    - Try to make the tree shallow

Reducing height of the trees

- When union, make shorter (smaller) tree root point to the larger tree root
  - This alone helps tremendously
  - To increase the size of the tree by 1, we must union tree of the same size
  - i.e., for each incremental of size, we double the size
  - Hence, to get a tree of height 10, we need 2^{10} elements in the tree
  - This gives h as log n
- When find, compress the path (set the parent of everything that we find to the root)
  - This is very very very fast

Actual Disjoint Set

```
def findset(x)
  if x == -1 return x;
  //path compression
  p[x] = findset(p[x]);
  return p[x];
end
```

```
def union(x, y)
  s1 = find(x)
  s2 = find(y)
  if S[s1] > S[s2]
    p[s2] = s1;
    S[s1] = S[s1] + S[s2]
  else
    p[s1] = s2;
    S[s2] = S[s1] + S[s2]
  end
end
```

- Use S[1..n] to store the size of the tree
  - Initially, S[i] = 1
  - When union, use S to determine which root should point to which root. Also update S afterward
- When find, update all parents in the path to point to the root

Kruskal Analysis

- Sorting the edges needs O(e lg e)
- For each edges, we need to do findset and union (which is just two findsets)
  - Both are O(h) where h = O(lg n)
- Total = O(e (lg e) + e (lg n))

From sorting of e edges

e iterations of 3 findset

Prim’s Algorithm

Another Algo for MST

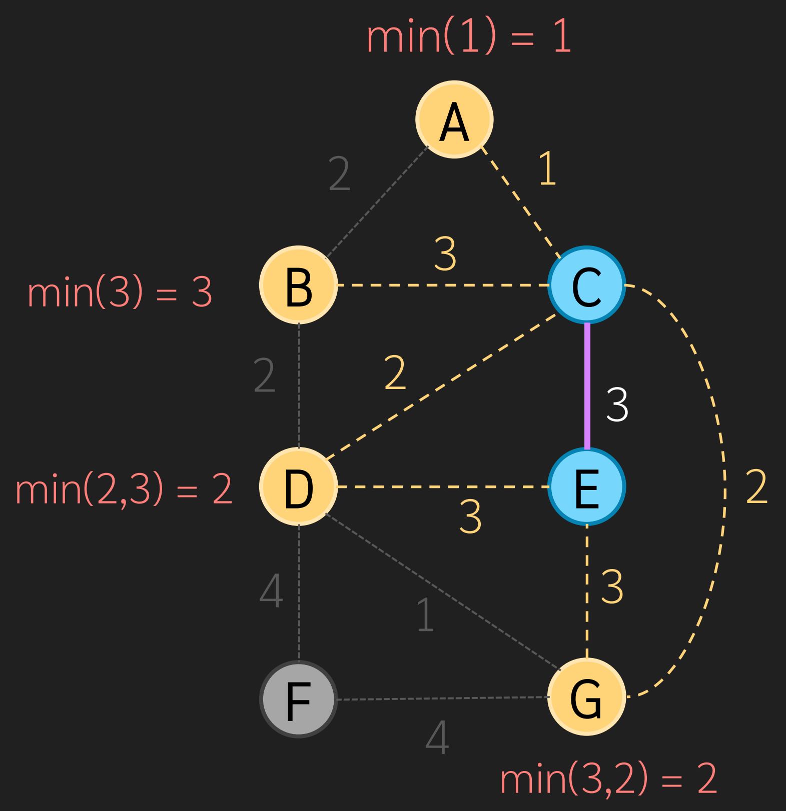
Prim’s Algorithm

- Kruskal: Select the minimal edge of all edges that does not create a cycle
- Prim: Select the minimal node & edge of all edges that connects to the original graph

Idea

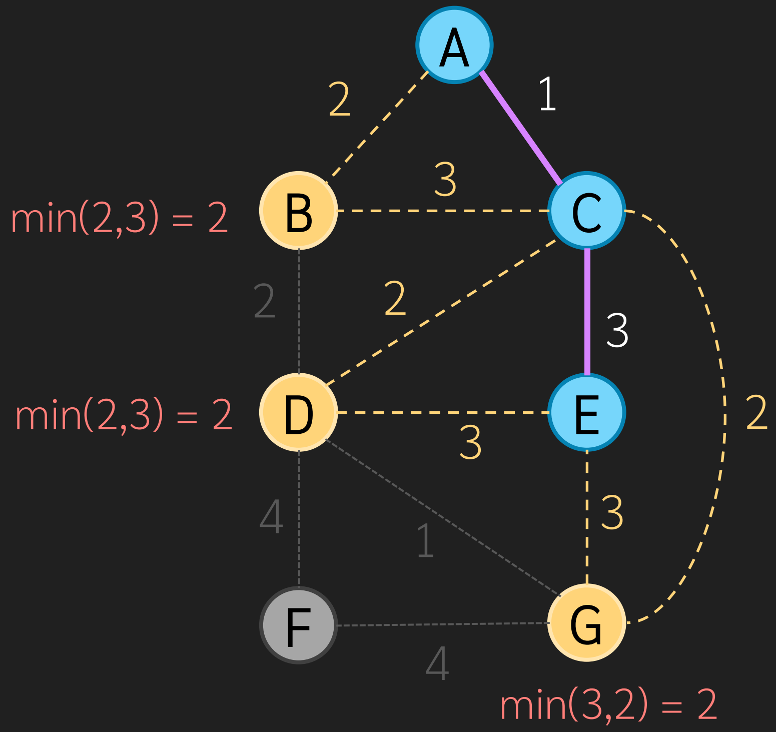
- Start with a single node (any node). This is our partial MST. We will expand this partial MST until it is the final MST of the graph
- We expand the partial MST by repeatedly adding one node and one edge to this tree, we stop when we have n nodes (and n-1 edges)
  - At each iteration, we maintain a list of connectible nodes which are nodes that can be added to our partial MST by adding just one edge.
  - We select the node with minimum cost of including that node to the tree.
  - When a node is added, we must update the list (and cost) of connectible nodes

Example



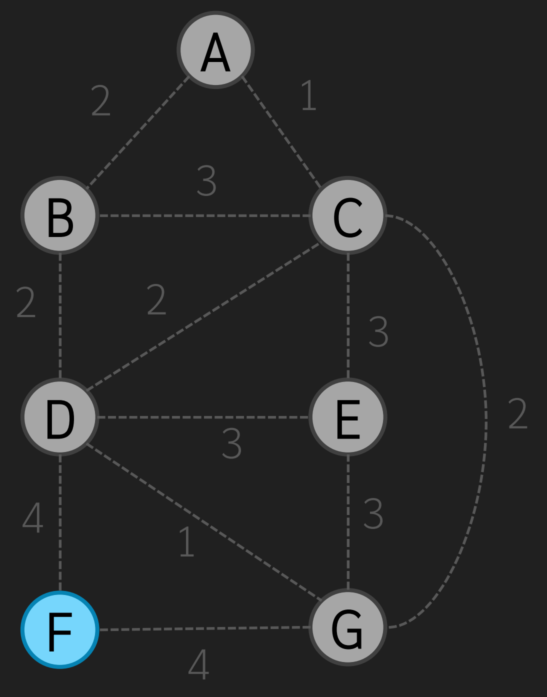
- Blue nodes and pink edges are our partial MST
- Yellow nodes are connectible nodes
  - Their cost are shown in red. The cost of each node is calculated from the minimum of weight of edges that connect our partial MST to the node (shown by yellow edges).
- Black nodes and edges are ones that is not connectible at this step
- We select nodes with minimum cost (and the edge that makes that minimum cost)
  - Update the connectible nodes and their cost

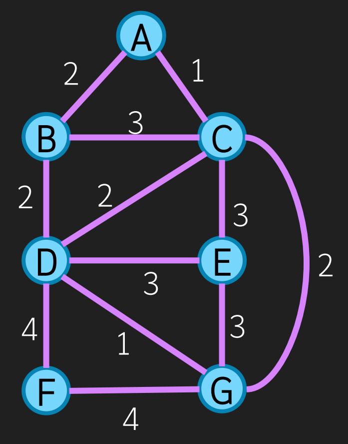
Example: adding nodes to partial MST



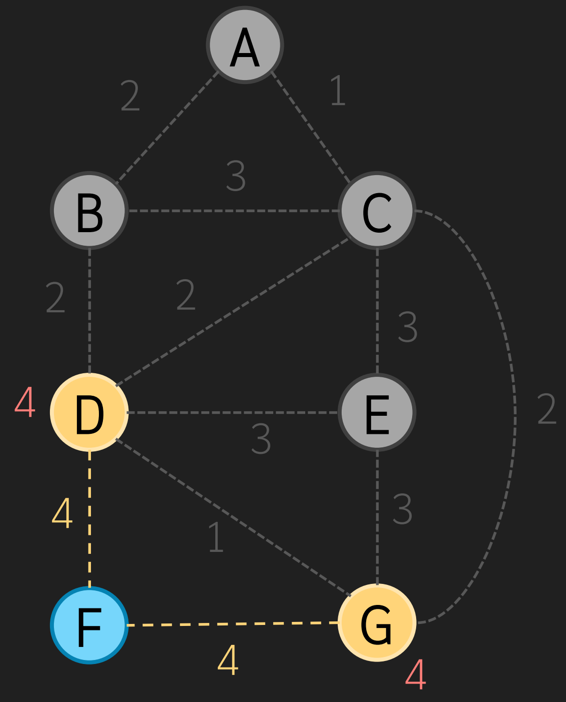
- We select node A because it has min cost amongst the yellow nodes
- Change it to blue and update
  - Might make some black nodes into yellow nodes

Full Trace: Start at F

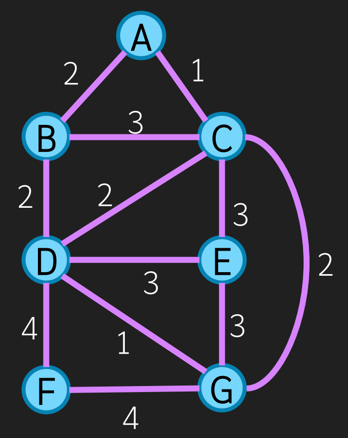




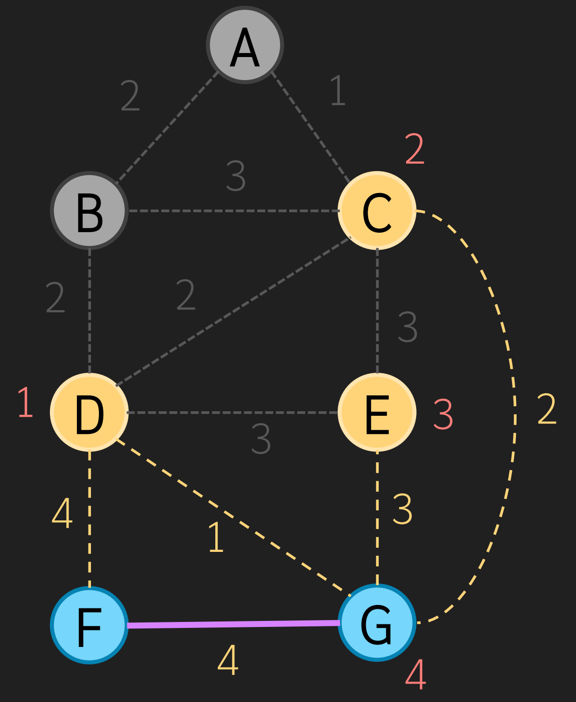
Full Trace: Update initial connectibles

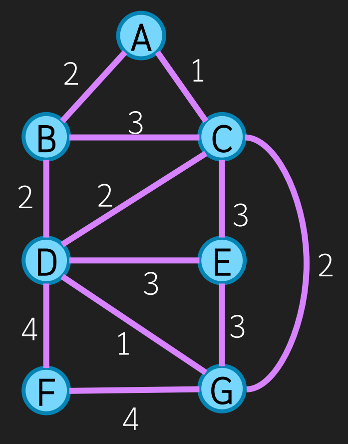


- Cost of D and G is updated

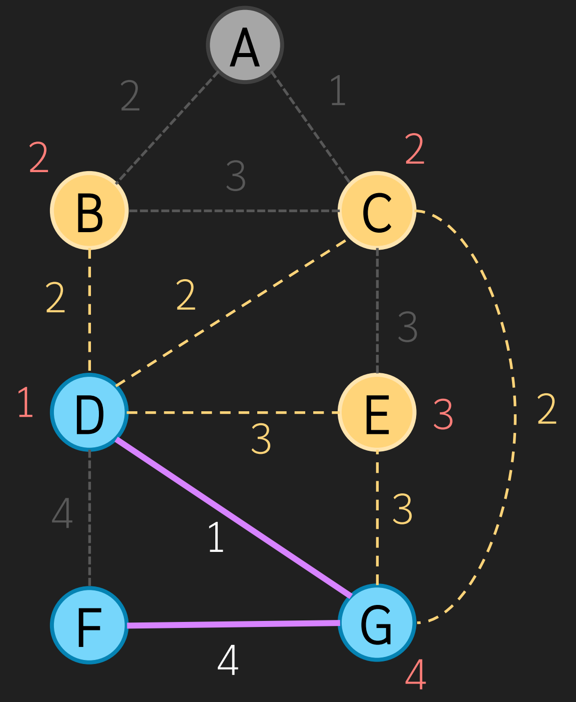


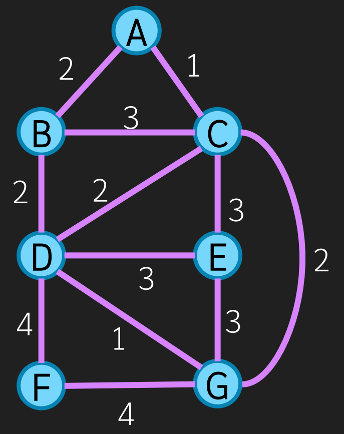
Full Trace: Select G



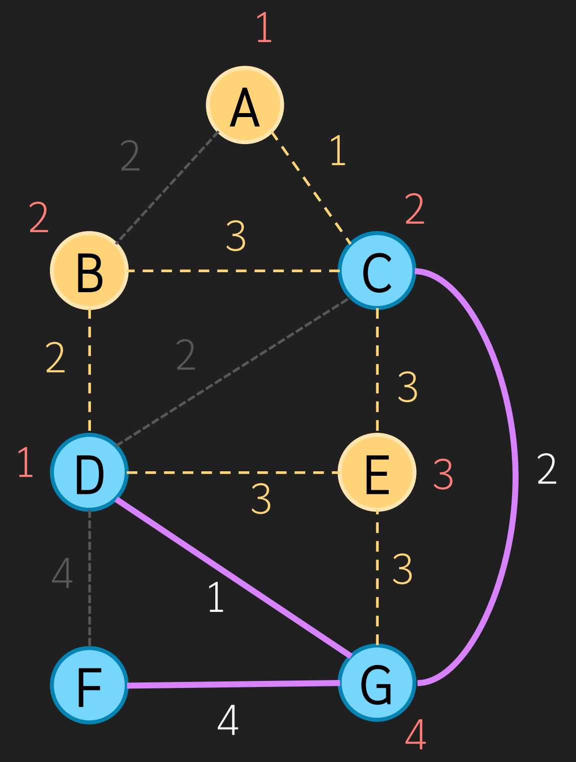


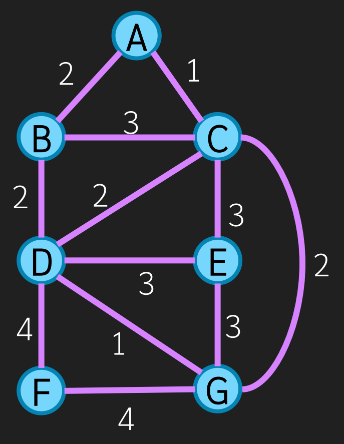
Full Trace: Select D



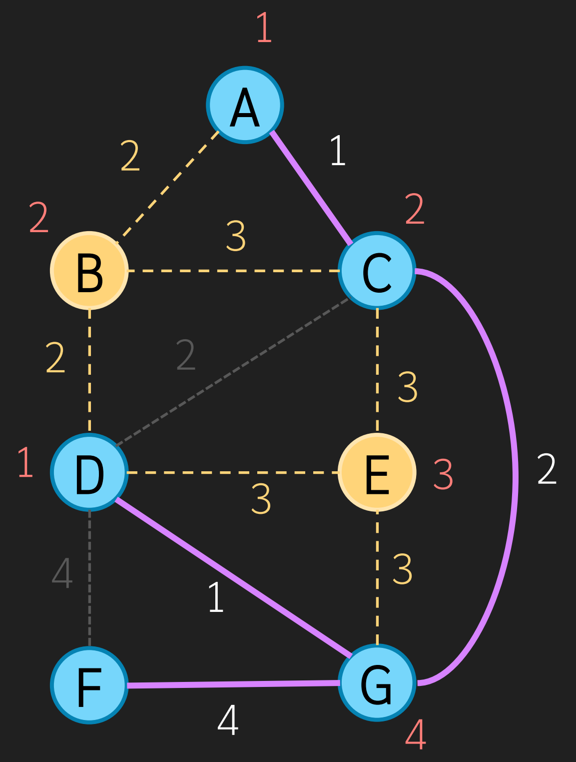


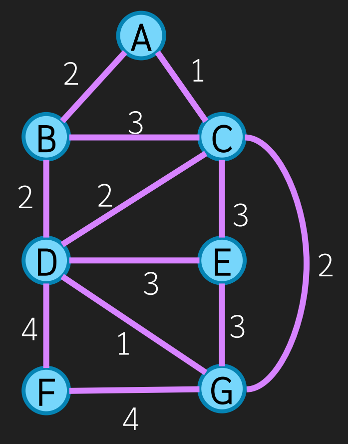
Full Trace: Select C



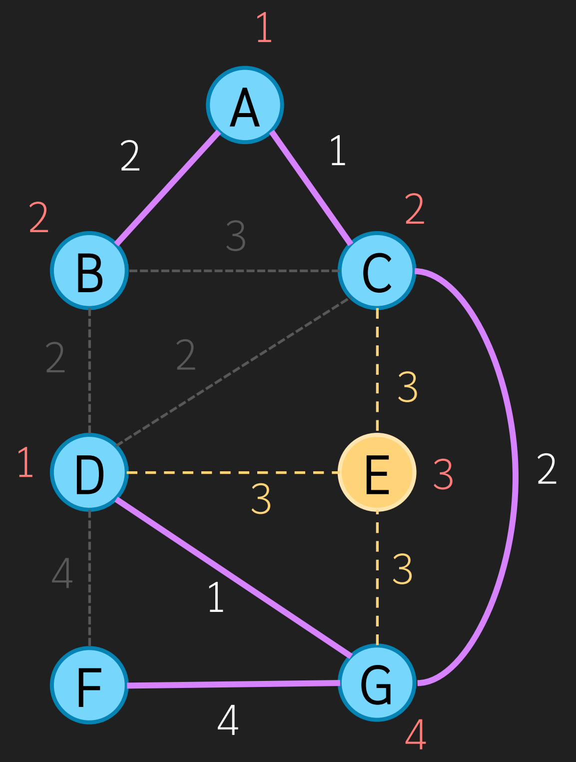


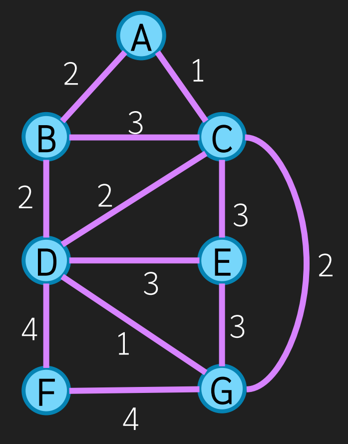
Full Trace: Select A



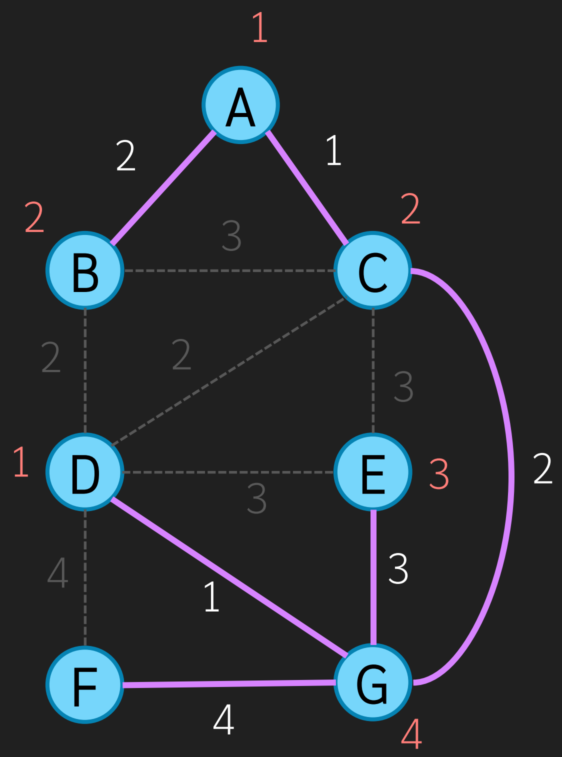


Full Trace: Select B

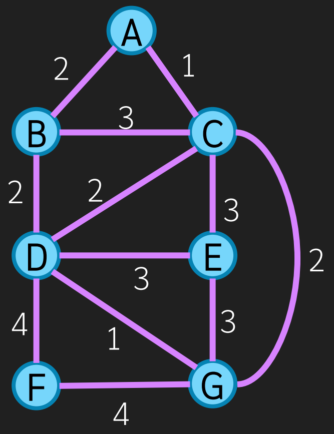




Full Trace: Select E



- Total cost = 4 + 1 + 2 + 1 + 2 + 3



Prim’s Implementation

- Iterate n-1 rounds
- Use some data structure that we can update and select min
  - Using a set
- `prev[x]` is the node that is used to connect x with.
  - The edge (x, prev[x]) is the selected edge
- `taken[x]` indicate whether node x is in the partial MST

```
def prim(G,w)
  for u in G.V
    cost[u] = ∞
    prev[u] = -1
    taken[u] = false
  S = new Set
  Pick any initial node u1
  S.insert( (0,u1) )
  while S is not empty
    (c,u) = S.min
    S.delete_min
    taken[u] = true
    for v in G.adj(u)
      if cost[v] > w(u,v) && taken[v] == false
        # change the value of v to the new cost[v]
        S.remove_if_exist( (cost[v],v) )
        cost[v] = w(u,v)
        prev[v] = u
        S.insert( (cost[v],v) )
      end
    end
  end
end
```

Prim's Implementation

- Better implementation
- Use (min) priority queue
- Must support decrease
    - Fibonacci Heap

Can do **top** and **decrease** in O(1)
pop is O(lg n)

Not covered in this class

```
def prim(G,w)
  for u in G.V
    cost[u] = ∞
    prev[u] = -1
  Pick any initial node u0
  cost[u0] = 0
  // H is a heap, using cost[] as keys, sort from min to max
  H = new priority_queue of all nodes in G.V
  while H is not empty
    u = H.top #pick nodes x with minimal cost[x]
    H.pop
    for v in G.adj(u)
      if cost[v] > w(u,v)
        cost[v] = w(u,v)
        prev[v] = u
        # change the value of v to the new cost[v]
        H.decrease(v,cost[v])
      end
    end
  end
end
```

Analysis

- There are exactly n-1 iteration in the while loop
- Each loop has
  - 1 delete min O(lg n)
  - Every neighbor of the selected node is removed and inserted O(lg n)
- Total is O(n (lg n) + e (lg n))

From delete min n nodes

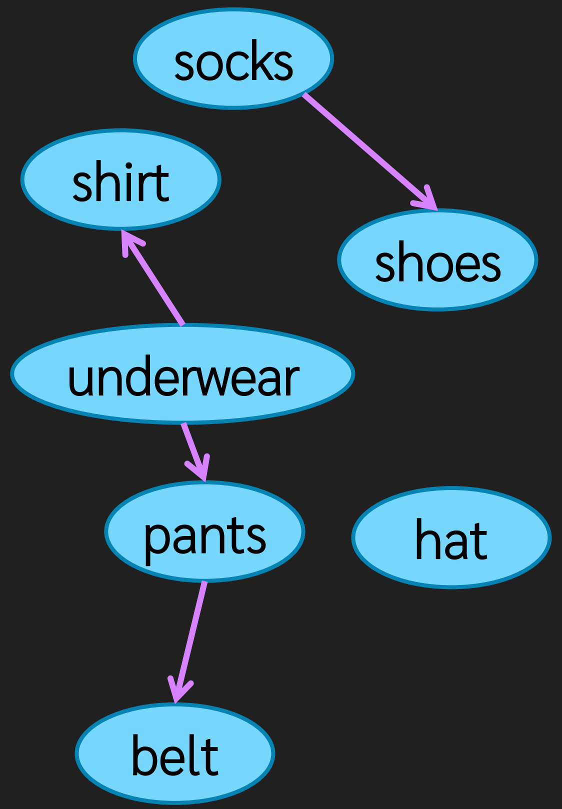
e times of remove and insert of nodes

Topological Sorting

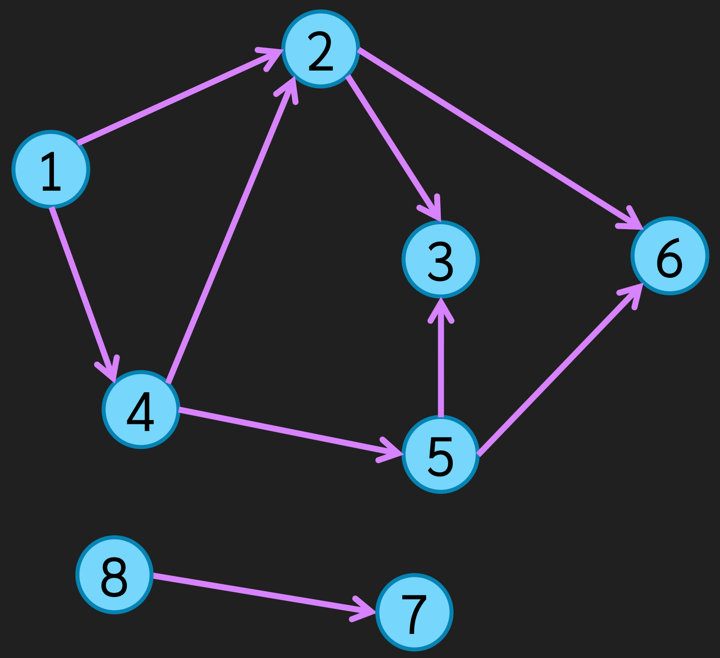
Ordering of nodes

Modelling Dependency as Directed Graph

- Consider a Task Scheduling Problem
  - For example, in a computer with multiple tasks running, the CPU can pick and run any task. Some program must
  - However, some task might need another task to finish first. This is called dependency. Some other task might not have dependency
  - Example, morning dressing routine
    - We must put on socks before shoes, i.e., shoes depends on socks
    - But socks and shirt aren't depends on each other
  - Another example, formula in the spreadsheet (excel), some cell might require result from another cell.
- These dependencies can be modelled as a directed graph. A node is a task and a directed edge from A to B means that B depends on A to finished first.
- The topological sorting problem is to calculate the ordering of the nodes such that all dependencies are met. It is a sorting of nodes



Example



- Is a topological sorting

1 4 2 5 3 6 8 7

8 7 1 4 5 2 3 6

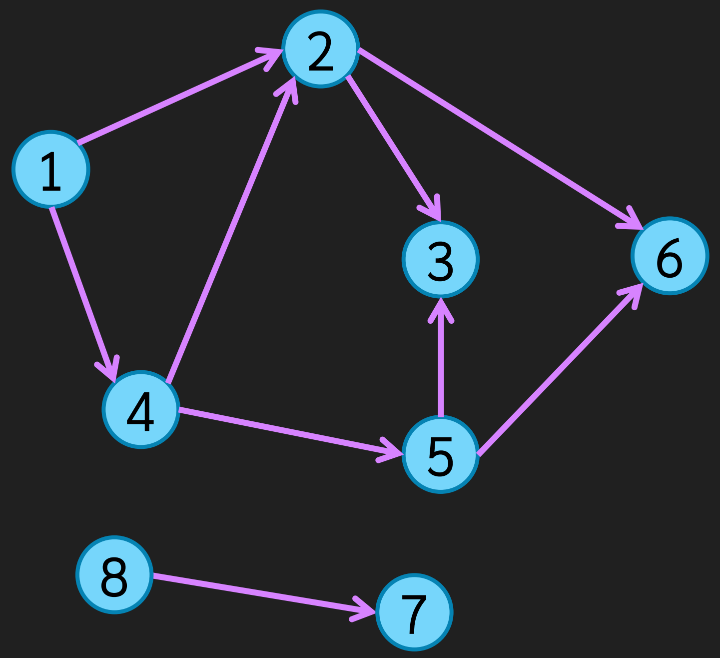
8 1 4 2 5 7 6 3

- Is not

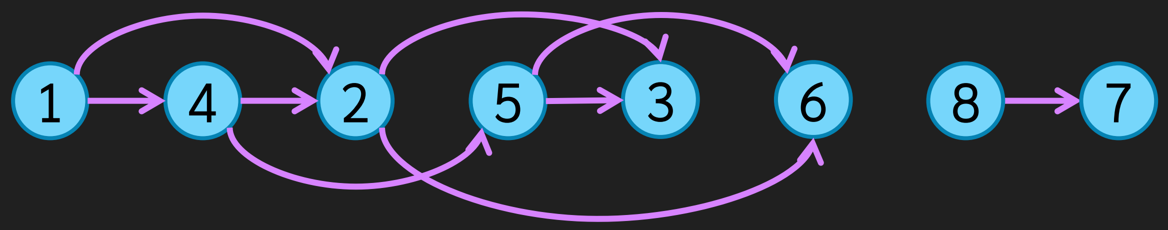
7 8 1 4 2 5 6 3

8 7 1 2 2 5 6 3

As Drawing of Graph



- Topological sorting is an arrangement of nodes from left to right such that every edges go to the right



Directed Acyclic Graph

- Job Scheduling requires that there is no cycle in the directed graph.
  - Because cycle means that the job cannot be done, job A needs job B to finish first but B also needs A to finish before as well. This create deadlock.
- Directed Acyclic Graph (DAG) is a directed graph without a cycle
- While we calculate a topological sort, we can detect whether a graph is acyclic
  - Or we can directly use the cycle detection algorithm

Problem Definition

- Problem: Given a directed acyclic graph, compute the topological sorting of the graph
- Input:
  - A directed acyclic graph $G = (V,E)$
- Output:
  - A sequence $T[1..n]$ where $T[i]$ is a member of V such that there is no edge connecting $T[a]$ and $T[b]$ where $a > b$

Observation

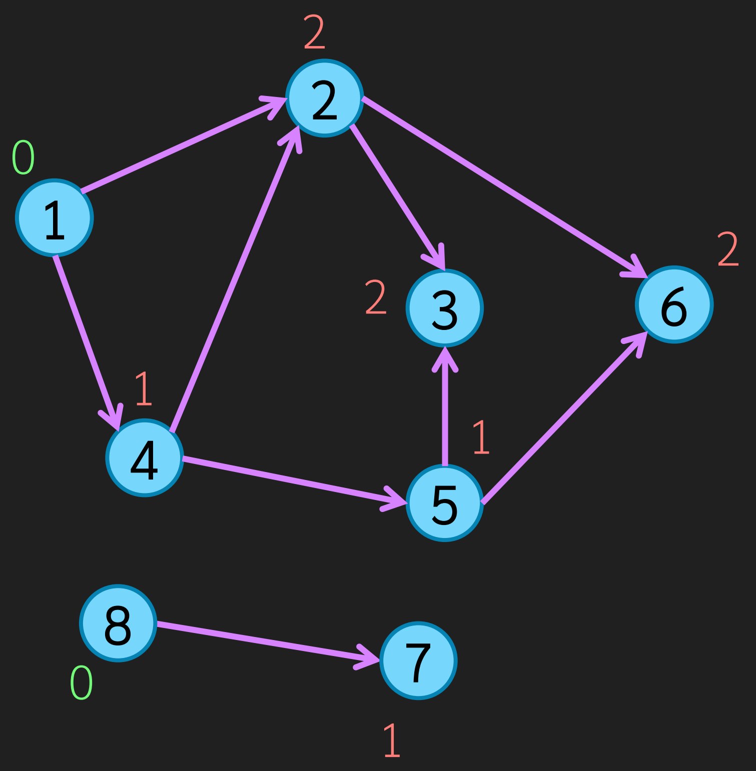
- In DAG, There must be nodes with in-degree of 0
- They are the nodes that we can pick as the starting points, i.e., select as job that is done first
  - After that we can remove these nodes and looking for any other nodes that has 0 in-degree

Kahn’s Algorithm

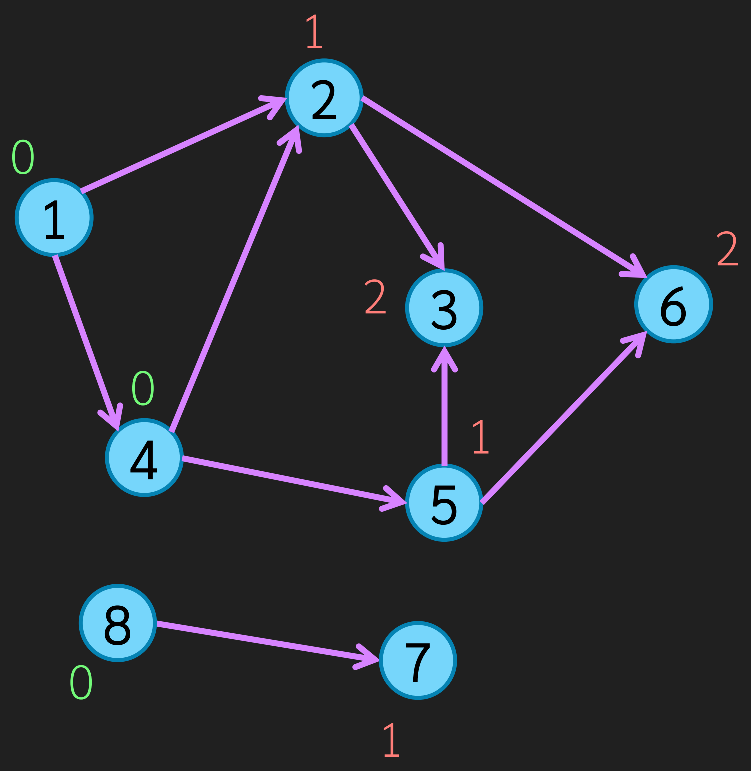
- In actual implementation, we can count the in-degree instead of remove edge

```
def topo_sort(G)
  ans = []
  q = new Queue
  let Z be any node in G.V that has zero in-degree
  for each z in Z
    q.push(z)
  while q is not empty
    u = q.front
    q.pop
    ans.push_back(u)
    for v in G.adj(u)
      remove edge (u,v) from G.E
      if v has zero in-degree
        q.push(v)
  if G.E is not empty
    return “error”
  return ans
end
```

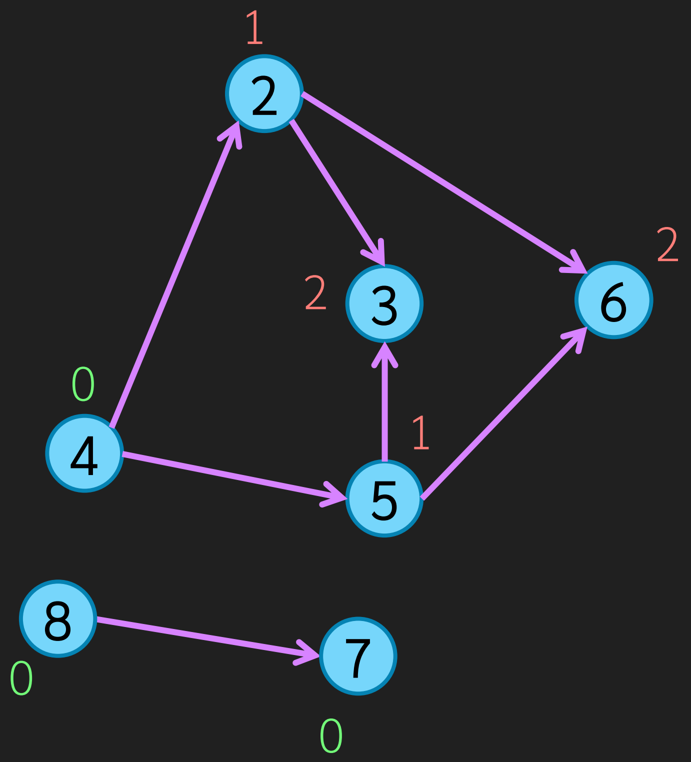
Example



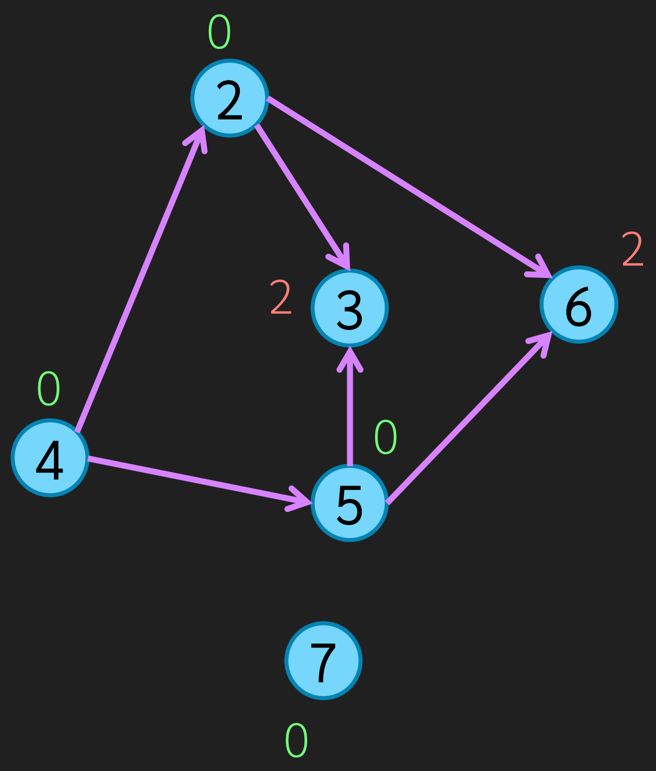
Example



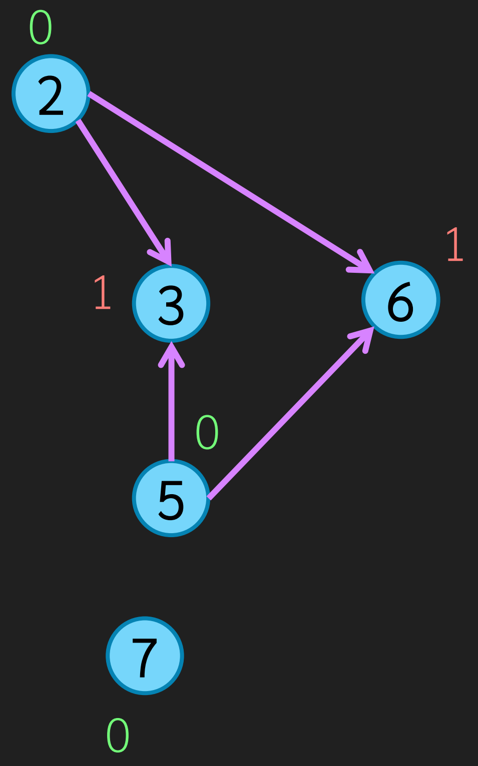
Example



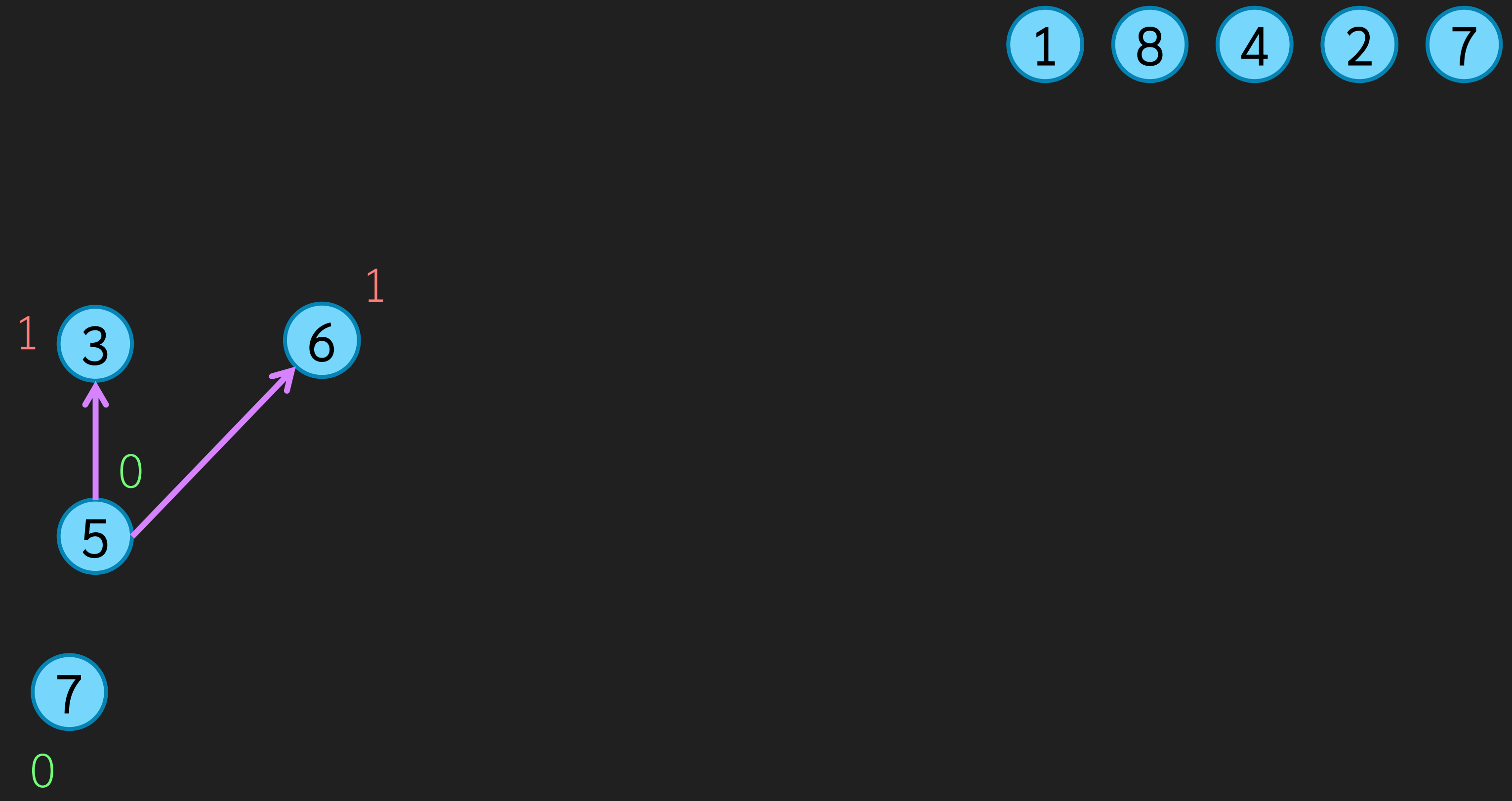
Example



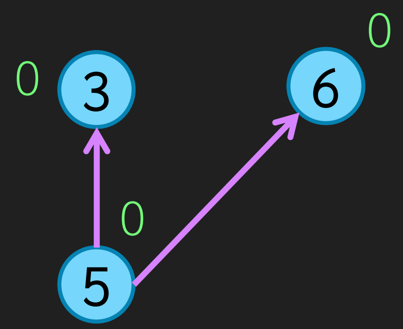
Example



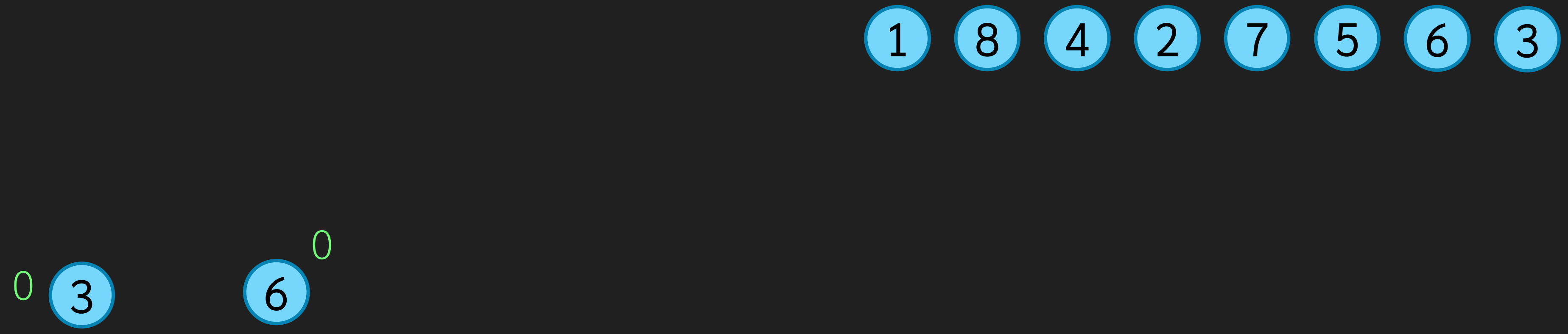
Example



Example



Example



Analysis of Kahn’s

- There are n rounds
- Each round, we remove one nodes and its edge
  - If we just count the in-degree, this is just decreasing the in-degree of the neighbor of the removing node
- $O(n+e)$