# Synthesis Assignment #2

# CS 5800, Fall 2022
# Dr. Lindsay Jamieson

# Chapter 4: Graphs

## 4.1 Basic Definition

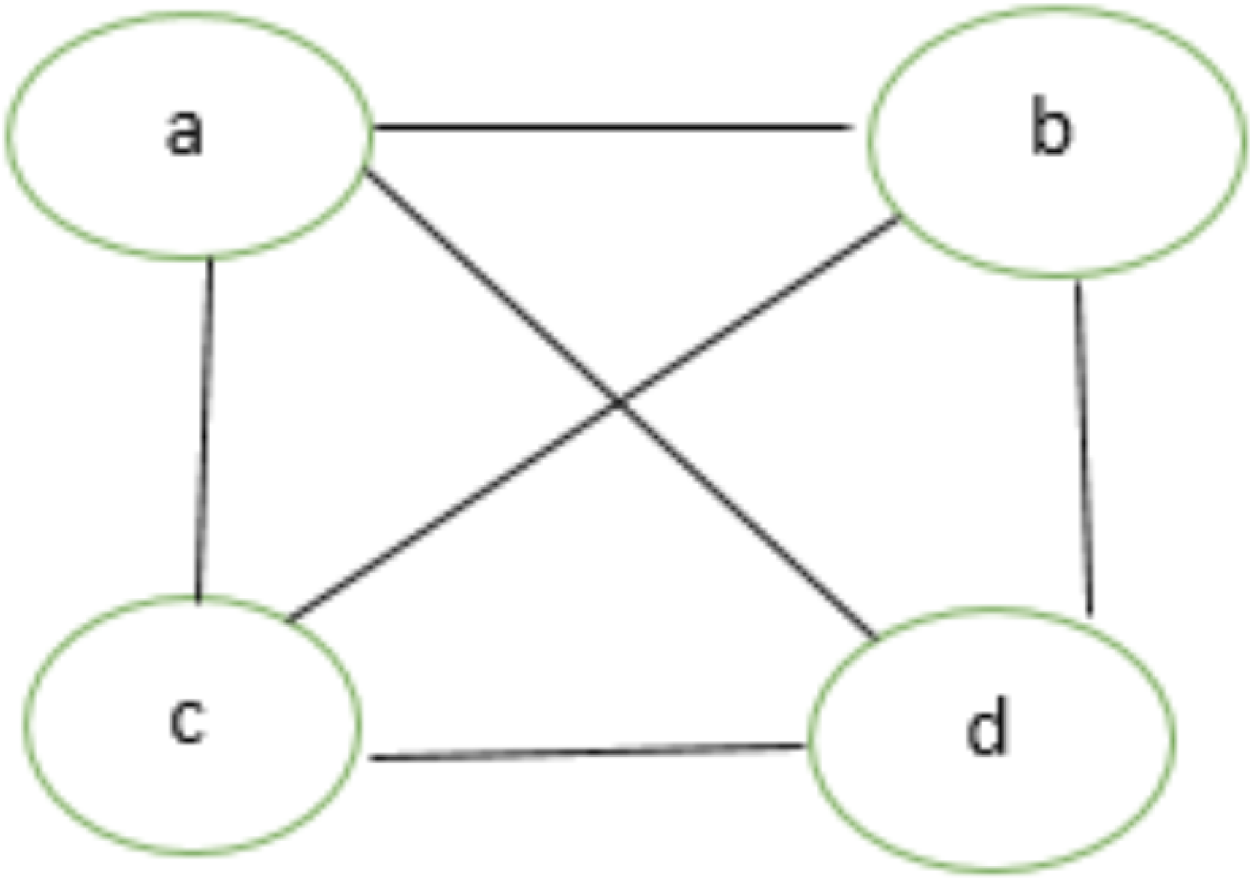
Basically, a graph is a structure composed of vertexes and edges. In computer science, a variety of graphs have been developed. First of all, I'll have a brief introduction about the classes of graphs.

- ### Types of graphs

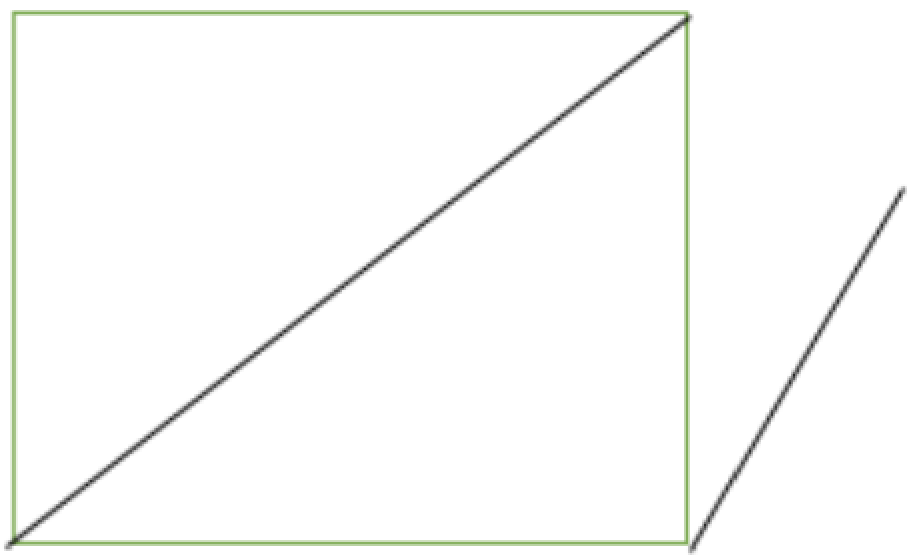
  In General, graphs are categorized into 2 types: undirected graphs and directed graphs. Literally, undirected graphs refer to the graphs that are conneted without directed edges while directed graphs are linked with directed edges. Basically, all types of graphs are derived from undirected graphs and directed graphs and here are some examples listed below:

- ### Complete graph

  A graph is a complete graph only when the degrees of each vertex is (n-1), n is the number of verticesimage.



- Connected or Disconnected Graph#### A connected graph refers to a graph that at least exsists a path between every pair of vertices in the graph. If not, it is a disconnected graph.

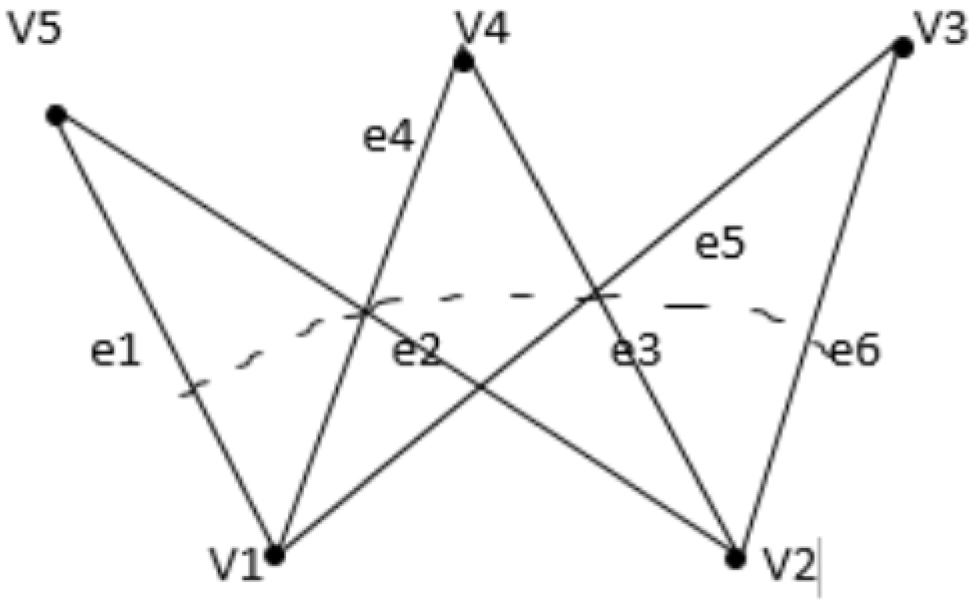

(a)

- Cyclic Graph

  A cyclic graph indicates a graph with at leaat one circle.

- Bipartite Graph

  A bipartite graph means the vertices are grouped into 2 sets of nodes where vertices in the same set are not connceted.



## 4.2 Graph representation

In general, there are 2 basic representations for graphs: one is the adjacency matrix and the other one is the adjacency list.

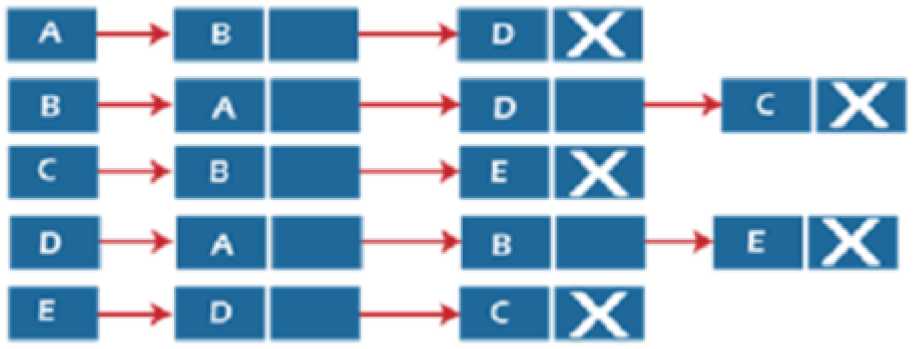
1. **Adjacency matrix**

   In computer science, adjacency matrixes of graphs are represented in terms of two-dimension arrays where rows and columns represent vertices in the graph and the pair(i,j) represents the connection of two vertices i and j. In this case, if (i,j) == 0, it indicates that there's no edge between node i and node j. Otherwise, if the graph is a weighted graph, then (i,j) will be the weight of the edge between i and j. Here's an example of a un-weighted adjacency list representation for a graph.

---

2. **Linked list**

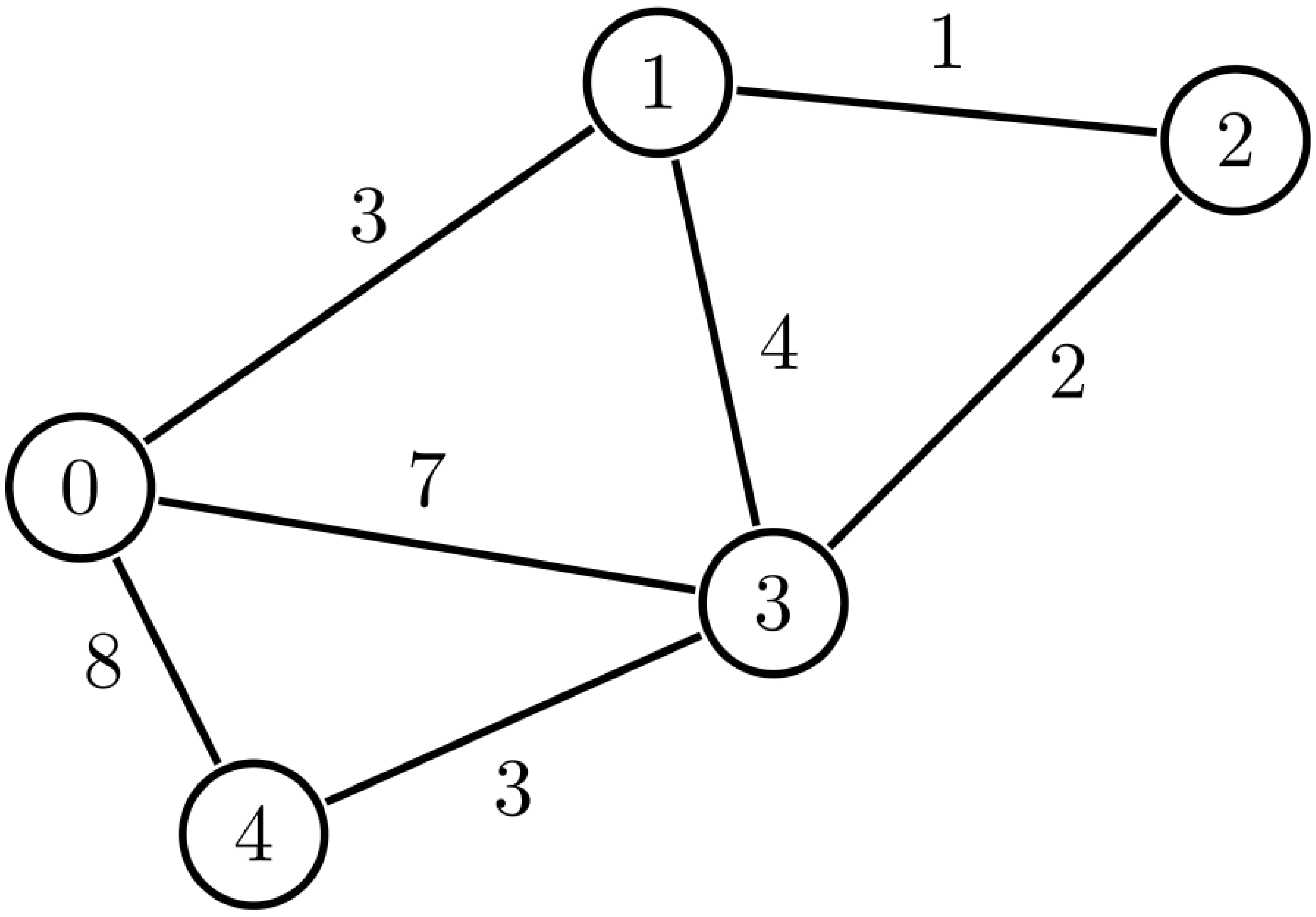
   The linked list representation of graphs is usually implemented by using linked list structures. In each row, the header of the linked list is the starting node, while the nodes linked to the header are nodes in the graph that connected to the header node. Here's a figure for the linked list representation.



## Exercise

translating a weighted graph from or to an adjacency matrix and list.

the chart for this graph is shown below:



**Solution:**

## 4.3 Graph traversal algorithm

In general, there are two graph traversal algorithm --- breadth first search and depth first search. Each of the seach algorithm can traverse all the nodes in the tree or graph, where the main difference lies in the fact that each one's goal is different.

- **BFS algorithm**

  Essentially, BFS aims at visiting the nodes at a deeper level. Therefore, this algorithm would traverse the graph as deeply as possible until it reaches the "bottom" of the tree(the node that has no un-visited neighnors). Generally, the DFS for graphs are implemented in terms of recursion. The codes in Java listed below can be a very good example.

  The time complexity for the DFS algorithm is O(V + E) while V is the number of vertices and E refers to the number of edges.

```
public void depthFirstSearch(Node node) {
    node.visit();
    System.out.print(node.name + " ");

    LinkedList<Node> allNeighbors = adjacencyMap.get(node);
    if (allNeighbors == null)
        return;

    for (Node neighbor : allNeighbors) {
        if (!neighbor.isVisited())
            depthFirstSearch(neighbor);
    }
}
```

- **BFS algorithm**

  BFS is short for breadth first search in computer science. Literally, it allows the cursor to traver the nodes with the priority of breadth in the graph. Different from DFS, BFS visit the all neighbors until it jumps into next iteration. And the implementation of the BFS in computer science would prefer the queue to represent the algorithm, since it conforms the FIFO(first in first out) mechanism. Each time when we are visiting a new node,we would pull out the element in the queue and add the neighbors to the queue in order to guarantee all nodes are traversed in a good order. Below is listed some Java implementation of BFS for graphs.

}

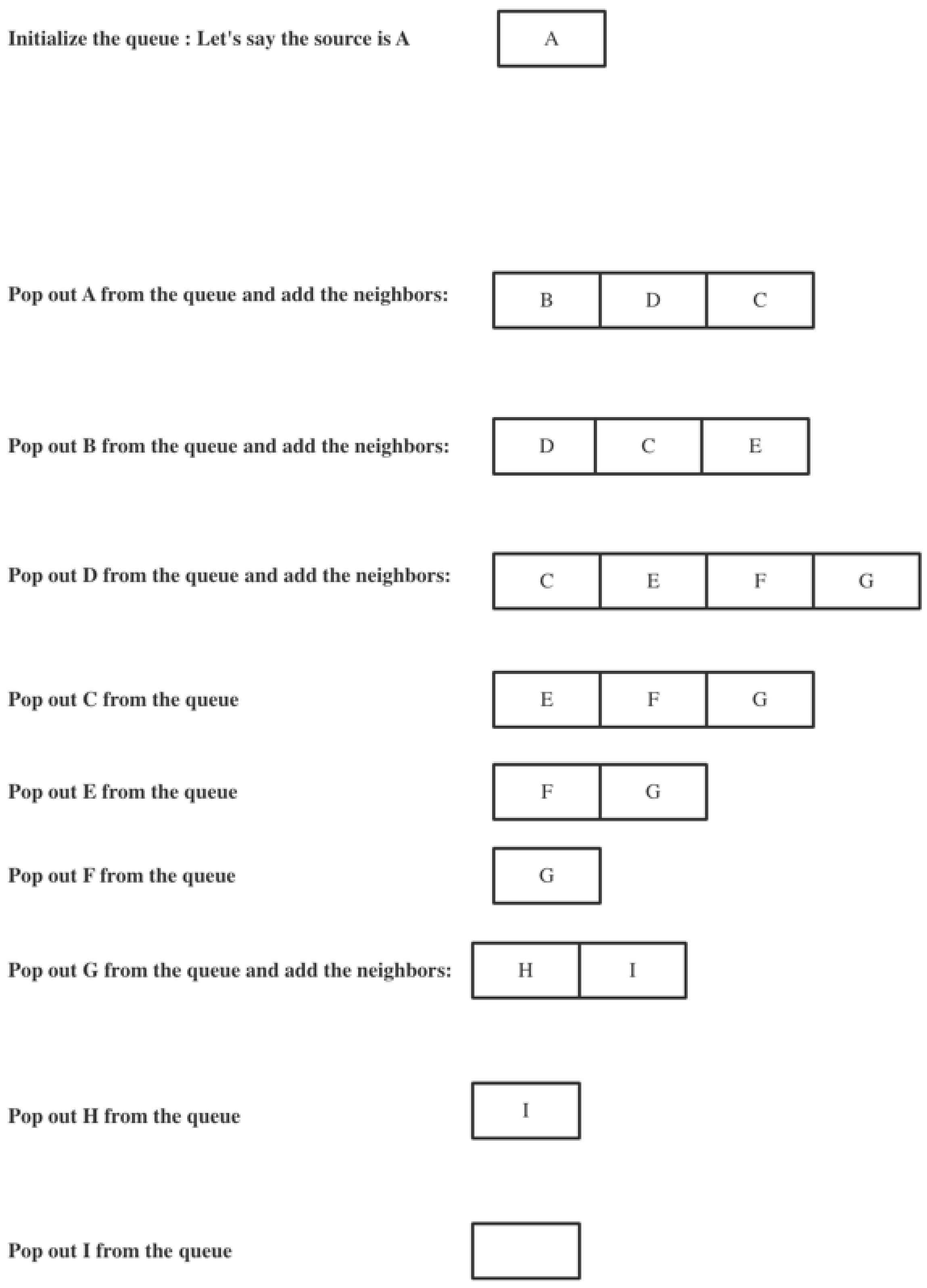
```
void breadthFirstSearch(Node node) {

  // Just so we handle receiving an uninitialized Node, otherwise an
  // exception will be thrown when we try to add it to queue
  if (node == null)
      return;

  // Creating the queue, and adding the first node (step 1)
  LinkedList<Node> queue = new LinkedList<>();
  queue.add(node);

  while (!queue.isEmpty()) {
      Node currentFirst = queue.removeFirst();

      // In some cases we might have added a particular node more than once
before
      // actually visiting that node, so we make sure to check and skip
that node if we have
      // encountered it before
      if (currentFirst.isVisited())
          continue;

      // Mark the node as visited
      currentFirst.visit();
      System.out.print(currentFirst.name + " ");

      LinkedList<Node> allNeighbors = adjacencyMap.get(currentFirst);

      // We have to check whether the list of neighbors is null before
proceeding, otherwise
      // the for-each loop will throw an exception
      if (allNeighbors == null)
          continue;

      for (Node neighbor : allNeighbors) {
          // We only add unvisited neighbors
          if (!neighbor.isVisited()) {
              queue.add(neighbor);
          }
      }
  }
  System.out.println();
```

}

```
/*
In some cases, we may traver an unconnected graph with disjointed nodes, If we
start with one random node in the graph, we may not be traverse all the nodes.
Therefore, we should use a loop to traverse all the nodes in the graph, even if
it's an un-connected one.
*/
void breadthFirstSearchModified(Node node) {
    breadthFirstSearch(node);

    for (Node n : adjacencyMap.keySet()) {
        if (!n.isVisited()) {
            breadthFirstSearch(n);
        }
    }
}
```

## Exercise

**Visit all the nodes in the given graph by using BFS.**



Initialize the queue : Let's say the source is A

Pop out A from the queue and add the neighbors:

Pop out B from the queue and add the neighbors:

Pop out D from the queue and add the neighbors:

Pop out C from the queue

Pop out E from the queue

Pop out F from the queue

Pop out G from the queue and add the neighbors:

Pop out H from the queue

Pop out I from the queue

## 4.4 Connectivity and strongly connected regions

- **Connectivity of graphs**

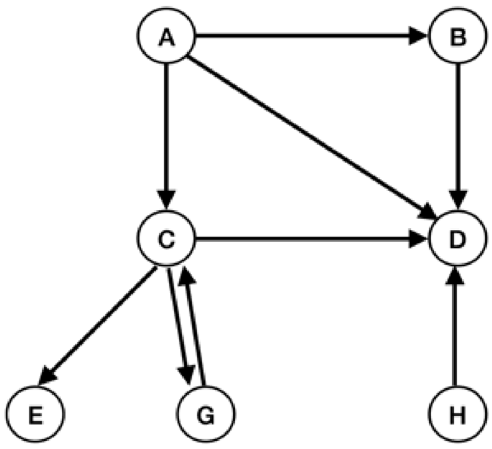
A connected graph refers to a graph in which there's always a path between every vertex. And the connectivity of a graph denotes a connected graph. On the contrary, dis-connectivity is a graph with disjointed nodes.

- **Strongly connected regions**

  A Strongly connected region is a part of a graph where every vertex is connected. Here's an example for a strongly connected graph. IMG/Stronglyconnectedgraph

## Exercise

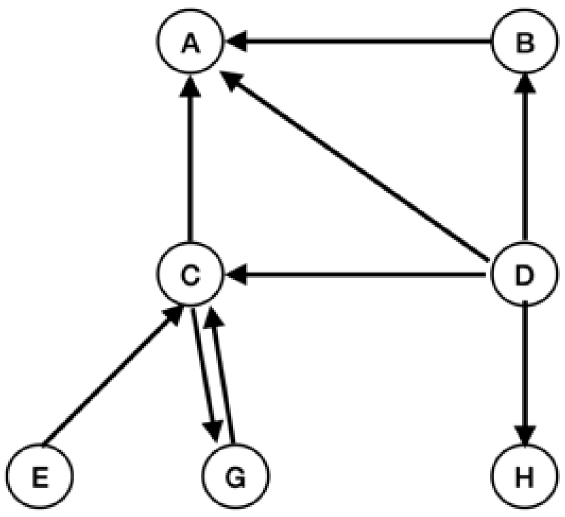
Run a step-by-step strongly connected component algorithm on the graph given below.



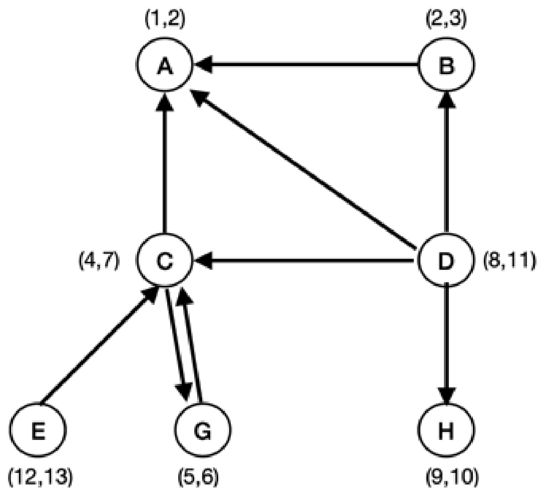
**Solution:**

Generally, we need to run DFS on the graph twice, and the only difference is that we need to run the first DFS on the reversed graph(all edges in the original graph are reversed).

**1. Reverse the graph**



**2. Run the first DFS on the given reversed graph**

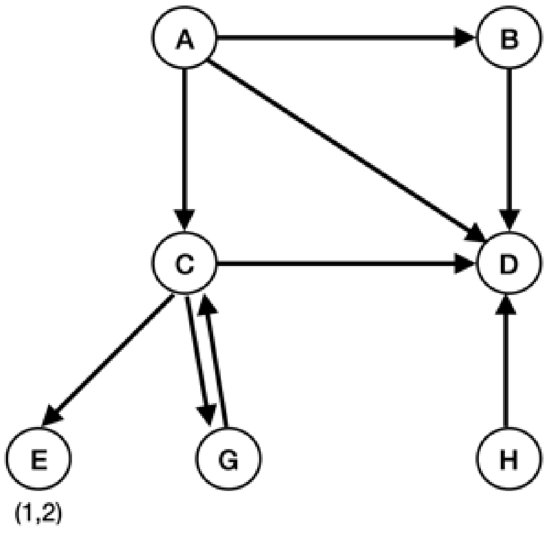


**3. Run the second DFS on the original graph**

After execution of DFS, it requires the other DFS on the original graph in descending order of post-visit number. In this case, the order of implementing DFS on the graph should be : E, D, H, C, G, B, A

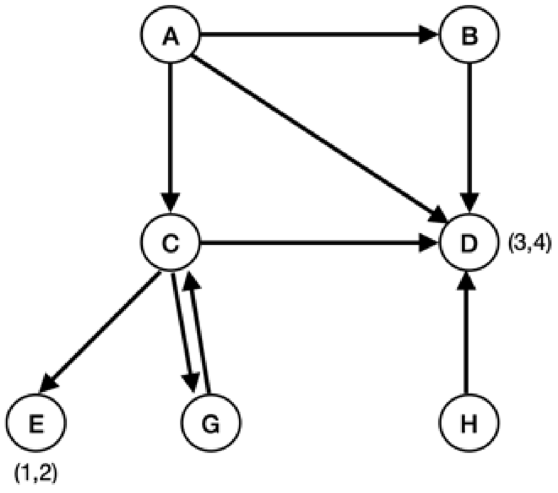
- We start off with E and since there are no edges coming off from E, we end at E.

  In this case, node E is a strongly connected component. Therefore, SSC set = {{E}}



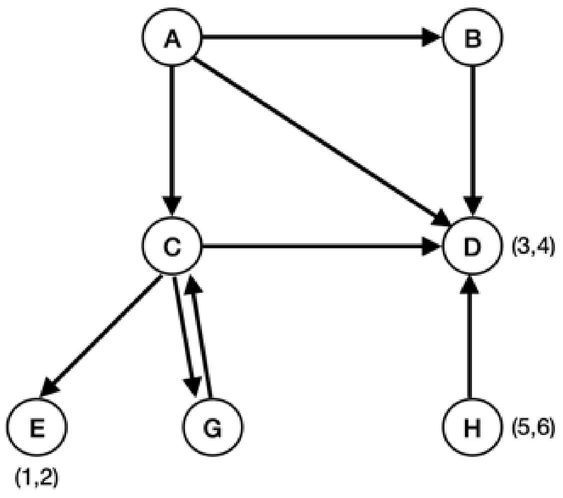
- Then we pick the next node -- D and similar to E, we end at D

  In this case, node D is a strongly connected component. Therefore, SSC set = {{E}, {D}}



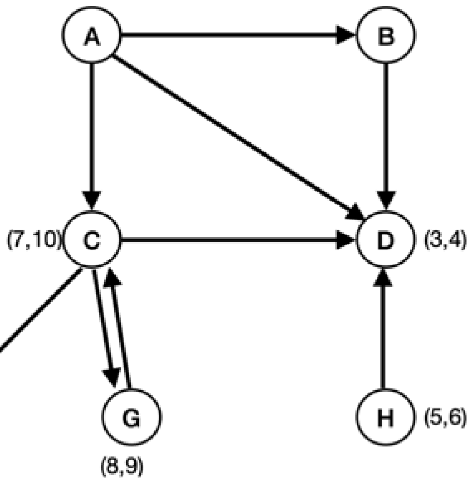
- Next, we start off another SSC at node H, and same as before, we end at node H.

  Therefore, another SSC is added to the SSC set. SSC set = {{E}, {D}, {H}}



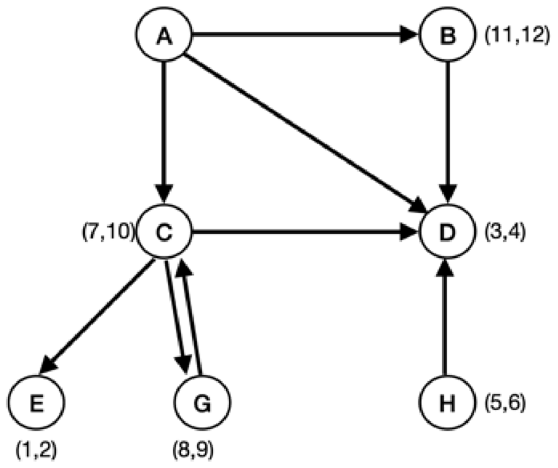
- Starting off C, and there is an edge C->G. After traversing G, there's no edge coming off, so we end at C.

  Therefore, another SSC is added to the SSC set, SSC set = {{E}, {D}, {H}, {C,G}}



- Starting off B, and there is no edge coming off from B.So we end at B.

  Therefore, another SSC is added to the SSC set, SSC set = {{E}, {D}, {H}, {C,G}, {B}}



- Finally, we come to the last node in the graph, since all nodes are already visited, we just end at A.

  Therefore, the last SSC is added to the SSC set, SSC set = {{E}, {D}, {H}, {C,G}, {B}, {A}}

# Chapter 5: Graph Algorithms

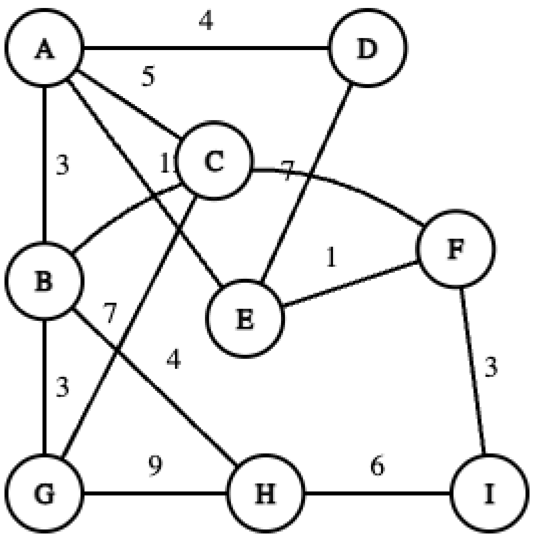
## 5.1 Dijkstra's Algorithm

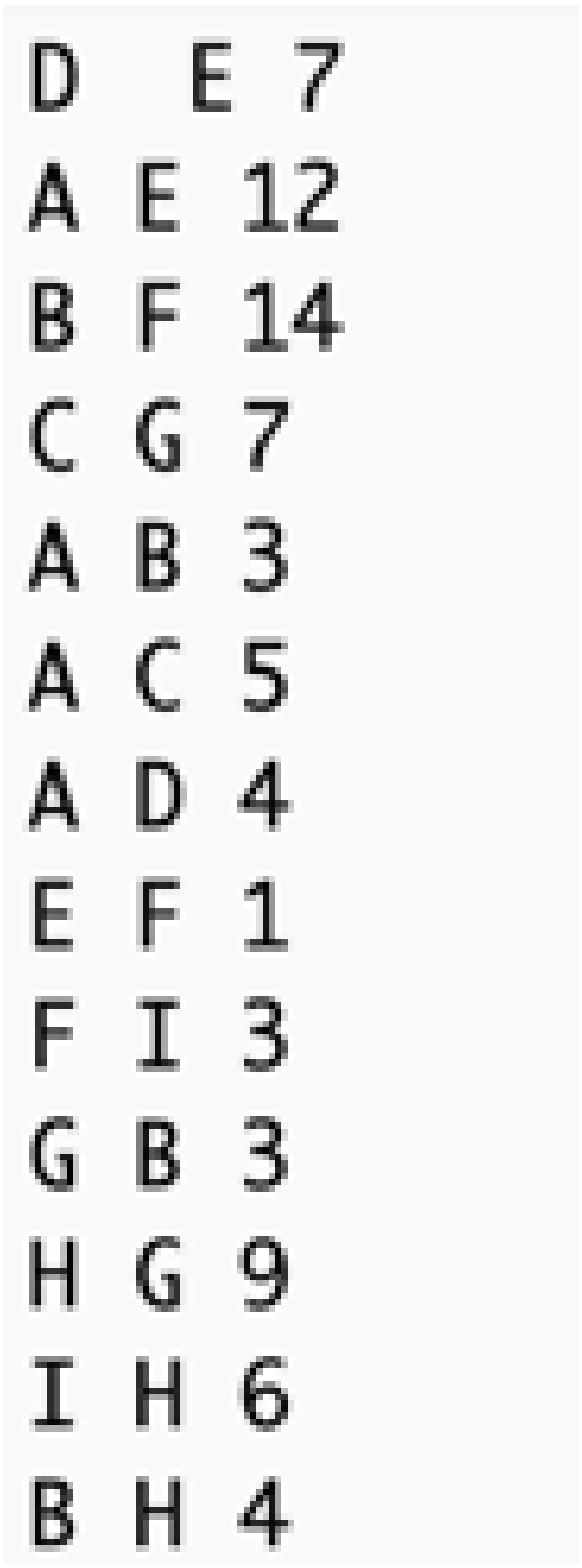
Essentially, Dijkstra's algorithm is a BFS combined with a greedy approach. Initially, all the nodes are added to the queue and in each iteration, the element at the front of the queue is popped out. This can be regarded as a BFS on the graph.

In addition, since we are popping out the node which has the shortest path distance from the source node. Similarly, this approach can be taken as a greed algorithm.

## Exercise

**Run a step-by-step Dijkstra's algorithm on the given graph below.**

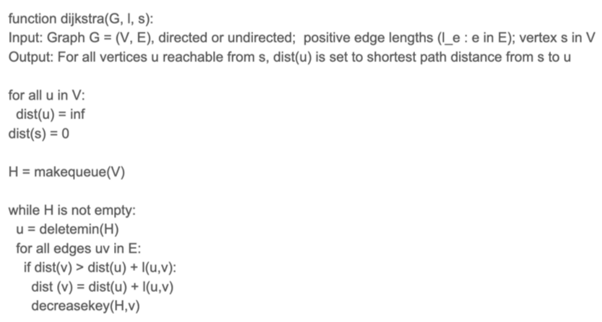

4
A
D
5
3
1
C
7
F
1
B
E
7
4
3
3
9
6
G
H
I



```
D   E 7
A E 12
B F 14
C G 7
A B 3
A C 5
A D 4
E F 1
F I 3
G B 3
H G 9
I H 6
B H 4
```

**Solution:**

Here's the Pseudocode for Dijkstra's algorithm from Dasgupta:



```
function dijkstra(G, l, s):
Input: Graph G = (V, E), directed or undirected;  positive edge lengths (l_e : e in E); vertex s in V
Output: For all vertices u reachable from s, dist(u) is set to shortest path distance from s to u

for all u in V:
  dist(u) = inf
dist(s) = 0

H = makequeue(V)

while H is not empty:
  u = deletemin(H)
  for all edges uv in E:
    if dist(v) > dist(u) + l(u,v):
      dist (v) = dist(u) + l(u,v)
      decreasekey(H,v)
```

1. enqueue all the nodes of the graph into the graph. H = {A, B, C, D, E, F, G, H, I}. <br > Then set the distance of the starting node to be 0, then H would be : H = {A = 0, B = INF, C = INF, D = INF, E = INF, F = INF, G = INF, G = INF, H = INF, I = INF}
2. Use a loop to pop out the front node in the queue in each iteration and update all the nodes in the queue.

- remove A from the H.

  H = {B = 3, D = 4, C = 5, E = 12, F = INF, G = INF, H = INF, I = INF}.
  A : 0

- remove B from the H.
  A : 0
  B : 3
  H = { D = 4,C = 5, G = 6, H = 7, E = 12, F = 17, I = INF}.

- remove D from the H.

  A : 0
  B : 3
  D : 4
  H = {C = 5, G = 6, H = 7, E = 11, F = 17, I = INF}.

- remove C from the H.

  A : 0
  B : 3
  D : 4
  C : 5
  H = {G = 6, H = 7, E = 11, F = 17, I = INF}.

- remove G from the H.

A : 0
B : 3
D : 4
C : 5
G : 6
H = {H = 7, E = 11, F = 17, I = INF}

- remove H from the H.

A : 0
B : 3
D : 4
C : 5
G : 6
H : 7
H = { E = 11, I = 13, F = 17}

- remove E from the H.

A : 0
B : 3
D : 4
C : 5
G : 6
H : 7
E : 11
H = { F = 12, I = 13}

- remove F from the H.

A : 0
B : 3
D : 4
C : 5
G : 6
H : 7
E : 11

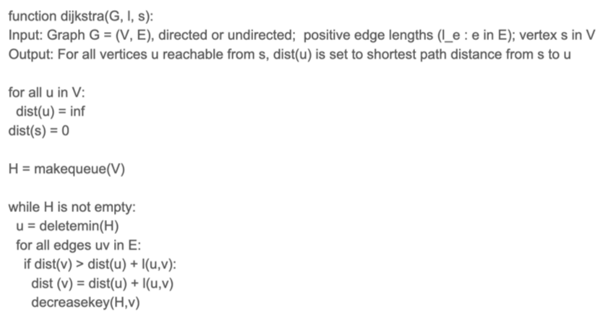
F : 12
H = {I = 13}

- remove I from the H.

A : 0
B : 3
D : 4
C : 5
G : 6
H : 7
E : 11
F : 12
I : 13
H = {}

## 5.2 Bellman-Ford Algorithm

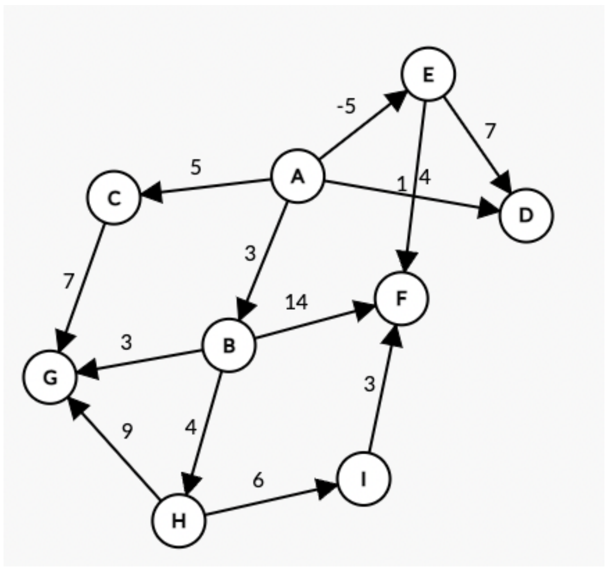
Similar to Dijkstra's algorithm, Bellman-Ford algorithm can also be used for finding the shortest path coming off from the source node in the graph. However, the Dijkstra's algorithm is subject to negative edges in the graph which indicates that if there's at least one negative edge, the Dijkstra's algorithm can no longer apply. Because when one element in the queue is popped out, it is possible that the distance from the source ndoe to the popped-out node can be smaller in the following iteration due to negative edges. And that's how Bellman-Ford kicks in ---- it can not only run on a positive weighted graph, but also on a weighted graph with negative edges. In addition, Bellman-Ford can also detect negative cycles in the graph. Here's the pseudocode for Bellman-Ford algorithm.



```
function dijkstra(G, l, s):
Input: Graph G = (V, E), directed or undirected;  positive edge lengths (l_e : e in E); vertex s in V
Output: For all vertices u reachable from s, dist(u) is set to shortest path distance from s to u

for all u in V:
  dist(u) = inf
dist(s) = 0

H = makequeue(V)

while H is not empty:
  u = deletemin(H)
  for all edges uv in E:
    if dist(v) > dist(u) + l(u,v):
      dist (v) = dist(u) + l(u,v)
      decreasekey(H,v)
```

## Exercise

Run a step-by-step execution of Bellman-Ford algorithm on the graph with negative edges given below.



Since there are 9 vertices in the graph, there would be 9 - 1 = 8 iterations for traversing all edges in the graph.

The 1st iteration

| Edge | Weight |
|---|---|
| A -> B | 3 |
| A -> C | 5 |
| A -> D | 4 |
| A -> E | -5 |
| B -> F | 14 |
| B -> H | 4 |
| B -> G | 3 |
| C -> G | 7 |
| E -> D | 7 |
| H -> G | 9 |
| H -> I | 6 |
| I -> F | 3 |
| E -> F | 1 |

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| d | 0 | 3 | 5 | 2 | -5 | -4 | 6 | 7 | 13 |

The 2nd iteration

| Edge | Weight |
|---|---|
| A -> B | 3 |
| A -> C | 5 |
| A -> D | 4 |
| A -> E | -5 |
| B -> F | 14 |
| B -> H | 4 |
| B -> G | 3 |
| C -> G | 7 |
| E -> D | 7 |
| H -> G | 9 |
| H -> I | 6 |
| I -> F | 3 |
| E -> F | 1 |

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| d | 0 | 3 | 5 | 2 | -5 | -4 | 6 | 7 | 13 |

Since all nodes are not uodated in the 2nd iteration, we can terminate the loop ahead of time. And the final answer is {B = 3, C = 5, D = 2, E = -5, F = -4, G = 6, H = 7, I = 13}

## 5.3 Subset Parameters (Matchings and Domination)