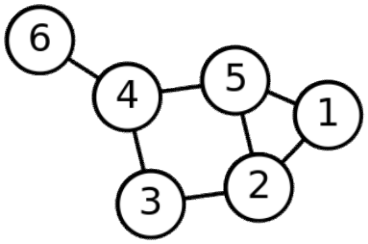
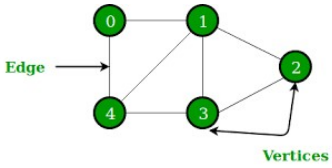
10. Graphs

Monday, May 23, 2022 10:21 PM

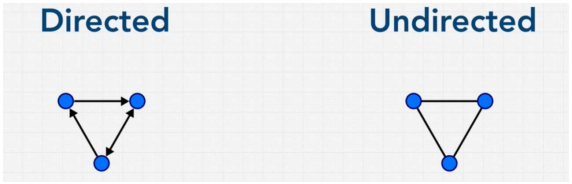


- nodes / vertices, interchangeable
- nodes are connected with edge

graphs are:
- set of values related in a pair-wise fashion



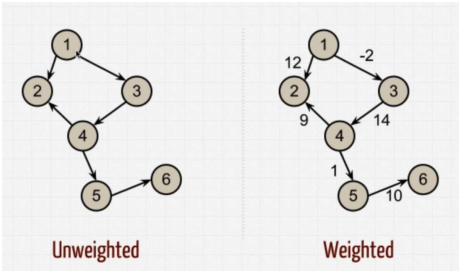
Types of graphs



Directed vs undirected

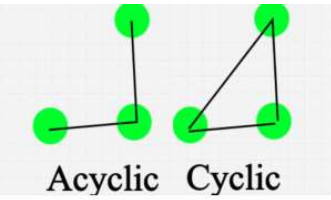
- movement is not bi-directional
- similar to one way street
- similar to twitter

- can go back & forth between nodes
- similar to highways
- similar to facebook, friends



unweighted vs weighted

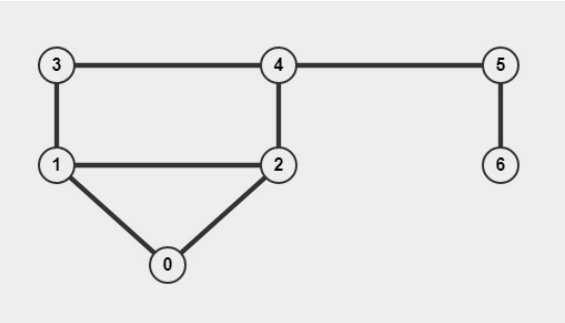
- google maps calculate optimal path from A to b



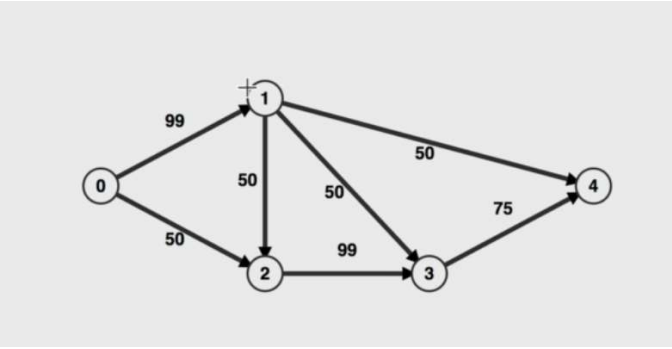
Acyclic vs Cyclic

- can circle back
  - like google maps
- only needs at least 1 cycle

e.g.



is undirected, unweighted, cyclic



is directed, weighted, acyclic

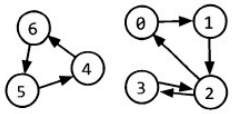
Graph Implementation (can be w/ array, hashmap, arraylist, LinkedList, etc)

3 ways to implement:

1) Edge List (u, v), shows connection from node u to v
   - if undirected, you need (u, v) and (v, u)

2) Adjacency List, hybrid of edge list & adjacency matrix (most common)
   - index of array is value of graph's node
   - value of index is the node's neighbor
   - if undirected, store twice

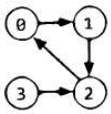
   represented w/ an array (or hashmap) of lists (arr, arrList, linked list, etc)




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

graph = [[1], [2], [0, 3], [2], [6], [4], [5]]

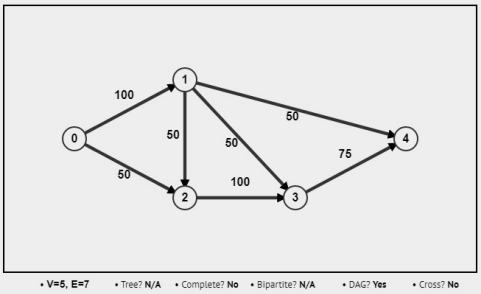
3) Adjacency Matrix, N x N boolean (where N is number of nodes)
   - 0s and 1s indicating if node x has a connection to node y
     (0 = false, 1 = true)
   - 0/1 can be replaced with weight



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

- 0 is connected to 1
- 1 to 2
- 2 to 0
- 3 to 2

graph = [
[0, 1, 0, 0],
[0, 0, 1, 0],
[1, 0, 0, 0],
[0, 0, 1, 0]
]



• V=5, E=7 • Tree? N/A • Complete? No • Bipartite? N/A • DAG? Yes • Cross? No

Adjacency Matrix

	0	1	2	3	4
0	0	100	50	0	0
1	0	0	50	50	50
2	0	0	0	100	0
3	0	0	0	0	75
4	0	0	0	0	0

Adjacency List

0:	(1, 100)	(2, 50)	
1:	(2, 50)	(3, 50)	(4, 50)
2:	(3, 100)		
3:	(4, 75)		
4:			

Edge List

0:	0	1	100
1:	0	2	50
2:	1	2	50
3:	1	3	50
4:	1	4	50
5:	2	3	100
6:	3	4	75

- can also hold weight in list
- lists can be objects to hold key, value

e.g. graph = [
0: [0, 1, 100],
1: [0, 2, 50]
...etc
]

Implementing Graph w/ Adjacency List

Graph

- addVertex(node)
- addEdge(node1, node2)

Main

C Test]

* Use ArrayList<LinkedList<Integer>>, because it is easier to add/remove new vertices & edges over array of arrays (b/c you have to shift elements)

* Time: addVertex O(1)
addEdge O(1)

Space: O(V+E)

```
import java.util.*;

public class Graph {
   // Adjacency List
   ArrayList<LinkedList<Integer>> adjacencyList = new
ArrayList<LinkedList<Integer>>();


   public void addVertex(int value) {
      LinkedList<Integer> newVertex = new LinkedList<Integer>();
      newVertex.add(value);

      // Move the LinkedList to ArrayList
      adjacencyList.add(newVertex);

   }

   public void addEdge(int source, int destination) {
     adjacencyList.get(source).add(destination);
 }

   public void printGraph() {
      for (LinkedList<Integer> inner : adjacencyList) {
         System.out.print(inner.getFirst() + " -> ");
         for (int i = 1; i < inner.size(); i++) {
            System.out.print(inner.get(i) + ", ");
         }
         System.out.println();
      }
   }
}
```

```
public class Main {

   public static void main(String[] args) {

      Graph graph = new Graph();

      graph.addVertex(0);
      graph.addVertex(1);
      graph.addVertex(2);
      graph.addVertex(3);
      graph.addVertex(4);
      graph.addEdge(0, 1);
      graph.addEdge(1, 0);
      graph.addEdge(1, 4);
      graph.addEdge(3, 1);
      graph.addEdge(4, 1);
      graph.addEdge(4, 2);
      graph.addEdge(4, 0);

        graph.printGraph();
   }
}
```

```
0 -> 1,
1 -> 0, 4,
2 ->
3 -> 1,
4 -> 1, 2, 0,
```

Summary

Pros

Good for relationships

(good for finding shortest paths)

Cons

Scaling is hard