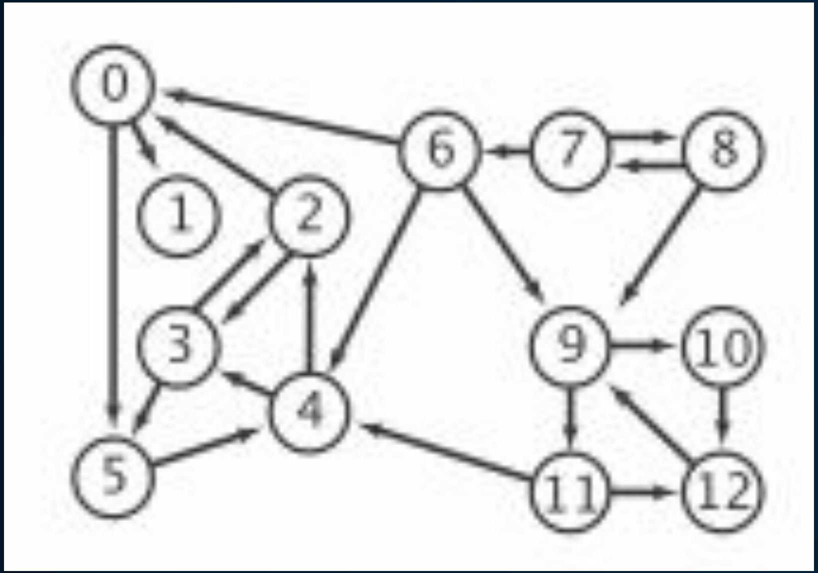
Directed graphs

DIRECTED GRAPHS

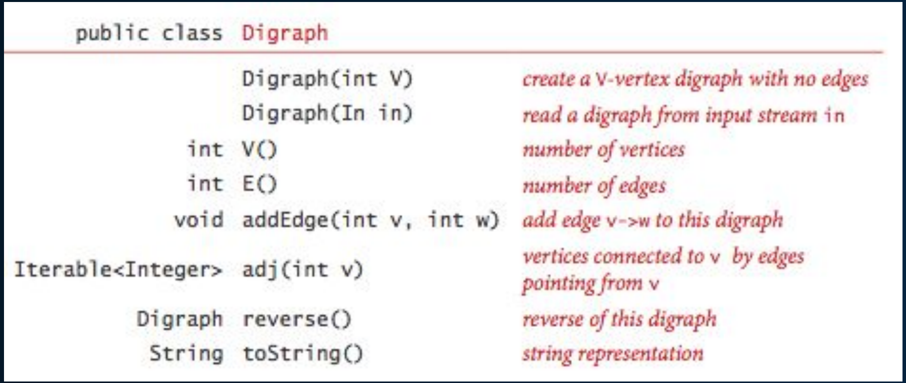
Introduction



A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. In contrast, a graph where the edges are bidirectional is called an undirected graph. When drawing a directed graph, the edges are typically drawn as arrows indicating the direction, as illustrated in the following figure.

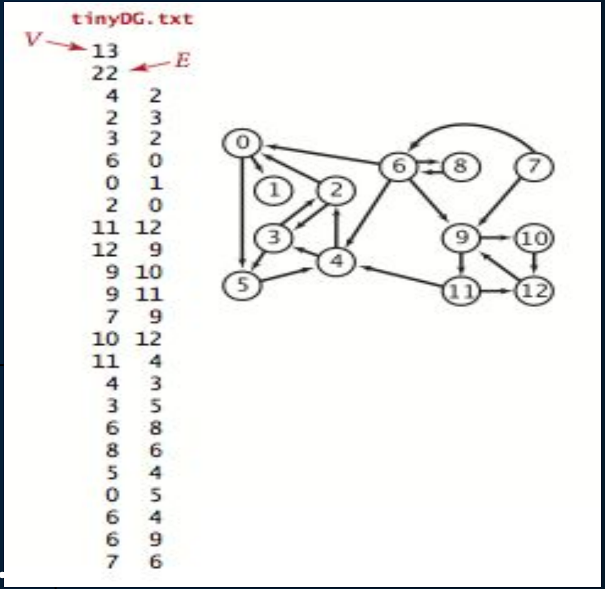
Digraph graph data type

We implement the following digraph API.

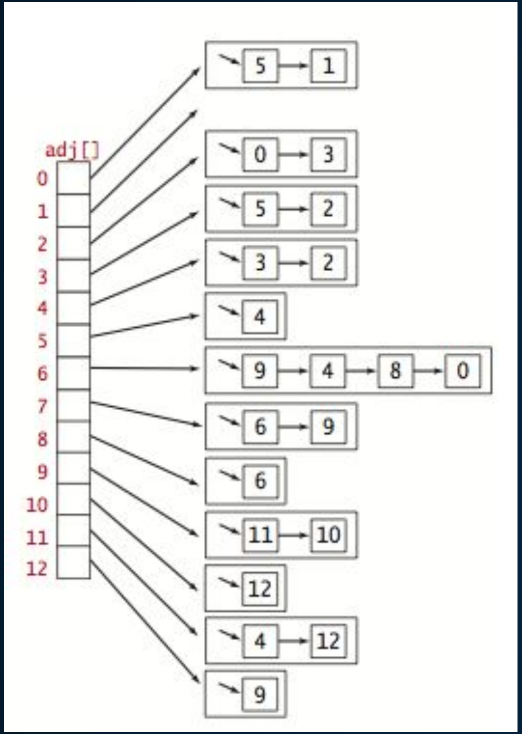


public class	Digraph	
	Digraph(int V)	*create a* V*-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v->w *to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to* v *by edges pointing from* v
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

The key method adj() allows client code to iterate through the vertices adjacent from a given vertex. We prepare the test data tinyDG.txt using the following input file format.



```
tinyDG.txt
13
22
 4  2
 2  3
 3  2
 6  0
 0  1
 2  0
11 12
12  9
 9 10
 9 11
 7  9
10 12
11  4
 4  3
 3  5
 6  8
 8  6
 5  4
 0  5
 6  4
 6  9
 7  6
```



Graph representation

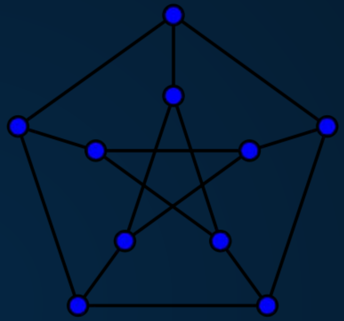
We use the *adjacency-lists representation*, where we maintain a vertex-indexed array of lists of the vertices connected by an edge to each vertex.

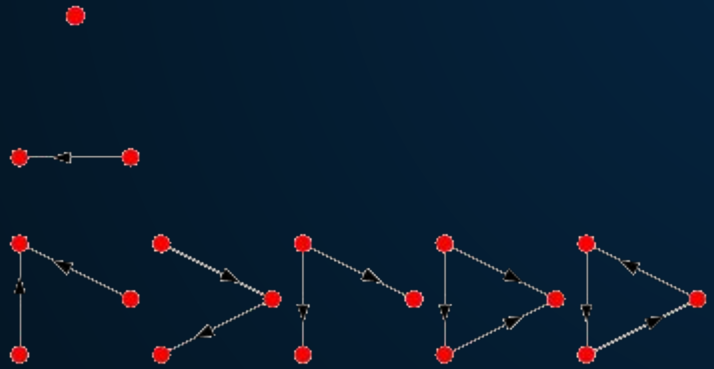
Digraph.java implements the digraph API using the adjacency-lists representation. AdjMatrixDigraph.java implements the same API using the adjacency-matrix representation.

Types of directed graphs

Subclasses

Symmetric directed graphs are directed graphs where all edges are bidirected (that is, for every arrow that belongs to the digraph, the corresponding inversed arrow also belongs to it).



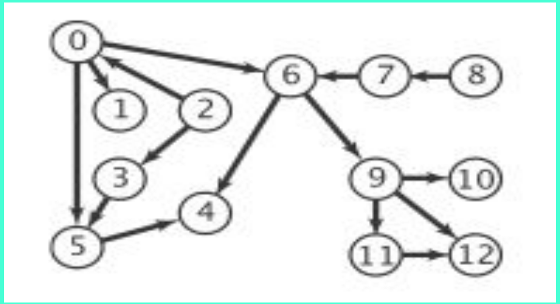


Simple directed graphs are directed graphs that have no loops (arrows that directly connect vertices to themselves) and no multiple arrows with same source and target nodes. As already introduced, in case of multiple arrows the entity is usually addressed as directed multigraph. Some authors describe digraphs with loops as loop-digraphs.

Reachability in digraphs

Depth-first search and breadth-first search are fundamentally digraph-processing algorithms.

Cycles and DAGs. Directed cycles are of particular importance in applications that involve processing digraphs.



Let's look at examples with code

Coding

-number of vertices in this digraph

-number of edges in this digraph

-adj[v] = adjacency list for vertex v

```
● package com.company;
● import java.util.LinkedList;
● import java.util.NoSuchElementException;
● import java.util.Scanner;
● public class Digraph {
●     private final int numberOfVertices;
●     private int numberOfEdges;
●     private LinkedList<Integer>[] adj;
●     private int[] degrees;
```

RESOURCES

```
lic Digraph(Scanner in) {
 if (in == null) throw new IllegalArgumentException("argument is null");
 try {
    this.numberOfVertices = in.nextInt();
    if (numberOfVertices < 0)
       throw new IllegalArgumentException("number of vertices in a Digraph must be non-negative");
    degrees = new int[numberOfVertices];
    adj = (LinkedList<Integer>[]) new LinkedList[numberOfVertices];
    for (int v = 0; v < numberOfVertices; v++) {
       adj[v] = new LinkedList<Integer>();
    }
    int numberOfEdges = in.nextInt();
    if (numberOfEdges < 0)
       throw new IllegalArgumentException("number of edges in a Digraph must be non-negative");
    for (int i = 0; i < numberOfEdges; i++) {
       int v = in.nextInt();
       int w = in.nextInt();
       addEdge(v, w);
    }
 } catch (NoSuchElementException e) {
    throw new IllegalArgumentException("invalid input format in Digraph constructor", e);
 }
```

In this class(Digraph), depending on each point, it is stored where it goes.

"addEdge()" - checks if there is such a point or not, will add points to the matrix, add +1 to umberOfEdges

"validateVertex()" - if the specified point is <0 or more "numberOfVertices" then it will give an error

```
public int V() {
        return numberOfVertices;
    }
    public int E() {
        return numberOfEdges;
    }
    private void validateVertex(int v) {
        if (v < 0 || v >= numberOfVertices)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (numberOf
    }
    public void addEdge(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        adj[v].add(w);
        degrees[w]++;
        numberOfEdges++;
    }
    public Iterable<Integer> adj(int v) {
        validateVertex(v);
        return adj[v];
    }
    public int outdegree(int v) {
        validateVertex(v);
        return adj[v].size();
    }
    public int indegree(int v) {
        validateVertex(v);
        return degrees[v];

    }
```

- the "toString" method is used to output to output
-in the main method we create a new object and add values through the scan

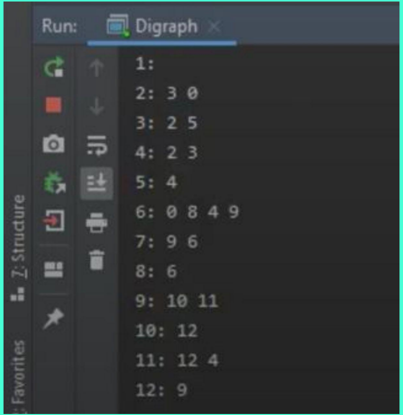
```java
public String toString() {
   StringBuilder s = new StringBuilder();
   s.append(numberOfVertices + " vertices, " + numberOfEdges + " edges " + "\n");
   for (int v = 0; v < numberOfVertices; v++) {
      s.append(String.format("%d: ", v));
      for (int w : adj[v]) {
         s.append(String.format("%d ", w));
      }
      s.append("\n");
   }
   return s.toString();
}

public static void main(String[] args) {
   Digraph G = new Digraph(new Scanner(System.in));
   System.out.println(G);
}
```

INPUT:

```
13
22
 4  2
 2  3
 3  2
 6  0
 0  1
 2  0
11 12
12  9
 9 10
 9 11
 7  9
10 12
11  4
 4  3
 3  5
 6  8
 8  6
 5  4
 0  5
 6  4
 6  9
```

OUTPUT



```
1:
2: 3 0
3: 2 5
4: 2 3
5: 4
6: 0 8 4 9
7: 9 6
8: 6
9: 10 11
10: 12
11: 12 4
12: 9
```

THE TEAM

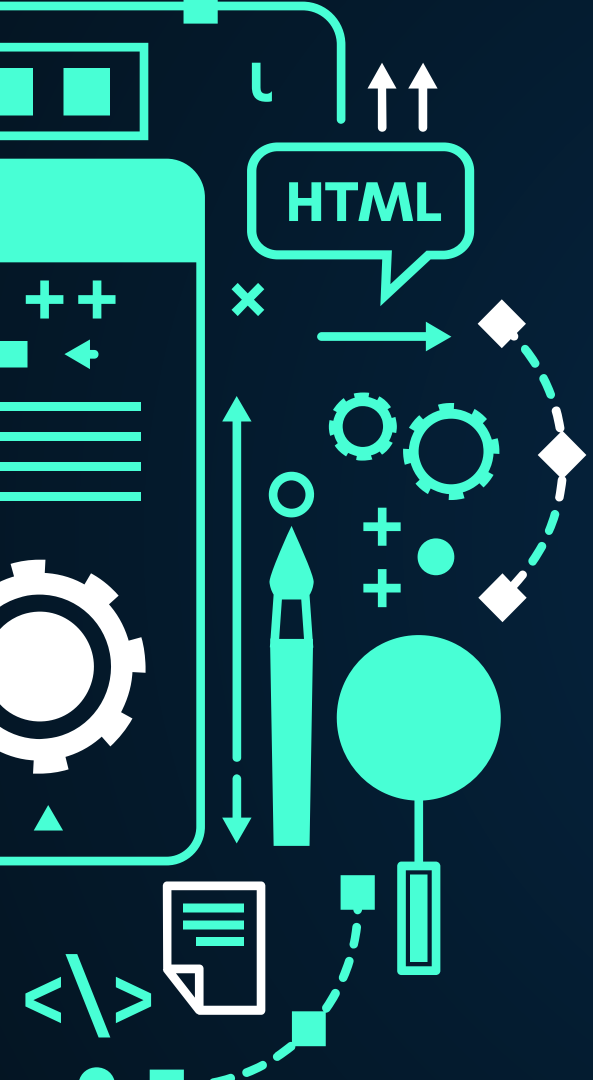
NURBOLAT

AYAULYM

SULTAN

GULSANA

DINMUKHAMMED



THANKS!