Data Structures and Algorithms
Introducing Graphs

Goodrich & Tamassia, Sections 13.1 & 13.2
Sahni, Sections 17.1 – 17.5

- Motivation
- Concepts
- Graph Abstract Data Type
- Graph Implementations

Motivation

Many problems can be formulated in terms of

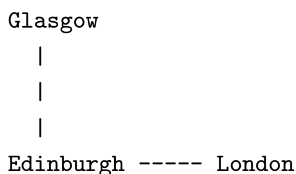
- a set of entities;
- relationships between them.

Examples:

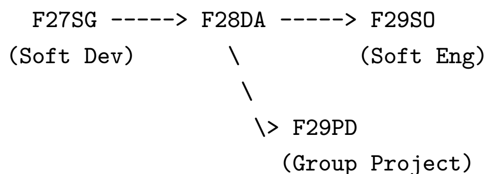
- Route finding:
  objects = towns
  relationships = road/rail links.
- Course planning:
  objects = courses
  relationships = prerequisites.
- Circuit analysis:
  objects = components
  relationships = wire connections.
- Game playing:
  objects = board state
  relationships = moves.

These can all be graphically represented:

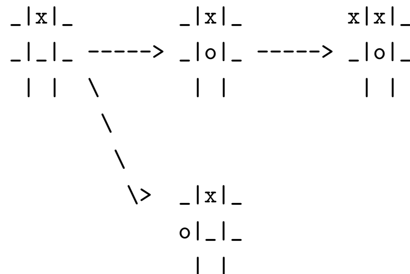
- Graph 1: Routes



- Graph 2: Course Precedences



- Graph 3: Game Moves



Each of these is a graph structure.

Definition

A graph is a data structure consisting of:

- a set of *vertices* (or nodes).
- a set of *edges* (or arcs) connecting the vertices.

i.e, G = (V, E) where V is a set of vertices, E is a set of edges, and each edge is formed from a pair of distinct vertices in V.

If we represent our problem data using a graph data structure, we can use standard graph algorithms (often available from code libraries) to solve it.

Example Graphs

- V_1 = {E,G,L}
  E_1 = {(E,G),(E,L)}
- V_2 = {F27SG,F28DA,F29SO,F29PD}
  E_2 = {(F27SG,F28DA), (F28DA,F29SO), (F28DA,F29PD)}

Graph Algorithms

Graph algorithms that we will look at include:

- Searching for a path between two nodes.
  - Can be used in game playing, AI, route finding, ..
- Finding shortest path between two nodes.
- Finding a possible *ordering* of nodes given some constraints.
  - e.g., finding order of modules to take; order of actions to complete a task.

Graph Concepts

Directed and Undirected

Vertices i and j are **adjacent** if (i, j) is an edge in the graph. The edge (i, j) is **incident** on vertices i and j.
E.g. in Graph 1 Glasgow is adjacent to Edinburgh, but not to London.
Exercise: Which nodes are adjacent in Graph 2?

If the edges are **undirected** (the order of the pair of vertices doesn't matter) the graph is **undirected** e.g. Graph 1 is undirected.

If the edges are **directed** (the order of the pair of vertices matters) the graph is **directed**, sometimes called a **digraph** e.g. Graph 2 is directed.
Exercise: Is graph 3 directed or undirected?

In a directed graph we say edge (i, j) is **incident from** i and **incident to** j; i is **adjacent to** j and j is **adjacent from** i.

In an undirected graph the **degree** of a vertex i is the number of edges incident on i. In a directed graph the **in-degree** of a vertex i is the number of edges incident to i, and the **out-degree** is the number of edges incident from i.
E.g. in Graph 1 the degree of Glasgow is 1
E.g. in Graph 2 the out-degree of F28DA is 2, and its in-degree is 1

Paths

A **path** $(i_1, i_2, ...i_k)$ is a sequence of vertices such that (i_j, i_{j+1}) is an edge for $1 \leq j \leq k$.
E.g. Paths from Glasgow (G) to London(L)

Exercise: How many paths are there from Glasgow (G) to London(L)?

In a simple path all vertices, except perhaps the first and last, are different

Exercise: Give a simple path from F27SG to F29PD in graph 2.

Cyclic and Connected Graphs

A directed graph is **cyclic** if there is a **path** from a vertex to itself, and **acyclic** otherwise.
E.g. graph 2 is acyclic.

Exercise: Is Graph 3 cyclic or acyclic?

A graph is **connected** if there is a path between every pair of vertices
E.g. Graph 1 is connected.

Exercise: Show that graph 2 is unconnected.

Weights, Labels and Trees

A **weighted** graph assigns costs to edges.
E.g. Costs are time, money, distance:

Similarly a **labelled** graph adds names to vertices.

Graphs are more general than trees. (Trees are a special kind of connected graph, with no cycles and a distinguished vertex(node), the "root").

A Graph Abstract Data Type

AbstractDataType Graph
{
instances
a set of vertices and a set E of edges

operations
$vertices()$: return the set of all vertices in graph
$edges()$: return the set of all edges in the graph
$existsEdge(v_i, v_j)$: return `true` if there is an edge from v_i to v_j; `false` otherwise
$putEdge(v_i, v_j)$: add edge e to the graph
$removeEdge(v_i, v_j)$: remove the edge from v_i to v_j
$degreeOf(v_i)$: return the degree of vertex v_i, defined only for undirected graphs
$inDegree(v_i)$: return in-degree of vertex v_i
$outDegree(v_i)$: return out-degree of vertex v_i
}

Graph ADT Exercises

Exercise: What is the result of each of the following abstract operations on Graph 1?

- $vertices()$
- $edges()$
- $existsEdge(E, L)$
- $existsEdge(G, L)$
- $putEdge(G, L)$
- $inDegree(E)$

Exercise: What is the result of each of the following abstract operations on Graph 2?

- $vertices()$
- $edges()$
- $existsEdge(F27SG, F29SO)$
- $putEdge(F29SO, F29PD)$
- $inDegree(F27SG)$
- $outDegree(F27SG)$

ADT Reflection

An ADT is an abstract specification that is independent of any

- any specific implementation, e.g. adjacency list/matrices
- any specific programming language, e.g. C#, SML, ...

Although you haven't seen any code specifying how the graph is implemented, the exercises show that you can reason about the expected program behaviour.

This is **crucial** for building software in large teams: you don't need to know details of how other components are implemented, just how they will work.

Implementing Graph Varieties

There are many varieties of graphs: un/directed, un/weighted etc.

Clearly we want to reduce implementation effort by **deriving** as much of each graph class as possible.

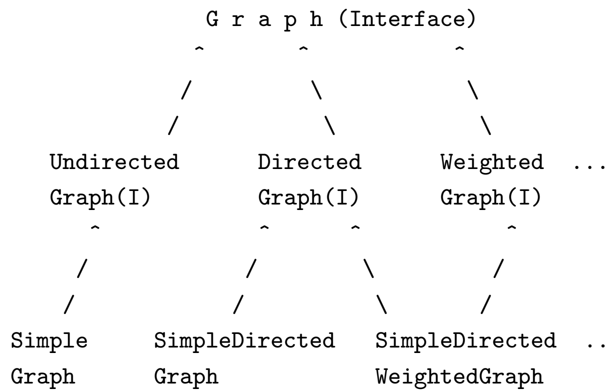
However there's no single obvious class structure to provide these varieties and this probably explains why there are no Graphs in the JDK Collections

We will use an Open Source Java Graph Library, `jgrapht` (googleable)

`jgrapht` uses both interfaces to specify functionality and abstract classes to ensure complete and consistent implementations.

As a production library the `jgrapht` is more complex than is ideal for teaching, but that's OK as it's abstract so you don't need to understand all the details.

jgrapht Interface Hierarchy



Abstract Class Refresher

`jgrapht` also uses abstract classes to ensure **complete** and **consistent** implementations.

Recall that Java has both **abstract** and **nonabstract**(default) classes.

Abstract classes

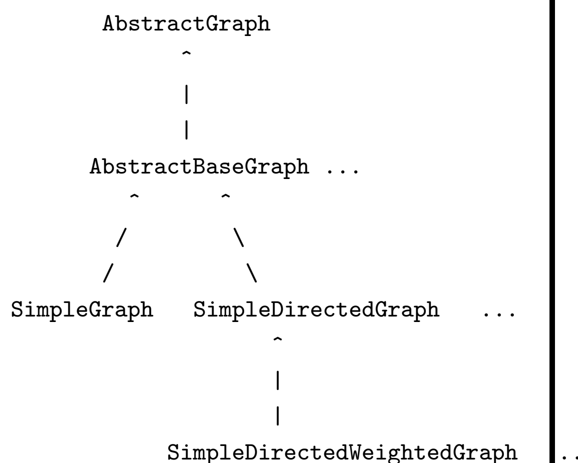
- contain abstract methods (no implementation provided)
- cannot be instantiated

Making every ADT implementation derive from an abstract class ensures a **complete** and **consistent** implementation of the ADT.

jgrapht Abstract Classes

As a production library `jgrapht` provides a complex hierarchy of abstract classes, rooted at `AbstractGraph`

Part of the hierarchy includes:



Graph ADT Implementations

What built in or user defined datatype can we use to represent a graph?

- Adjacency matrices
- Adjacency lists

An adjacency matrix is for a graph with N vertices is an N x N array of boolean values e.g. for Graph 2:

	F27SG	F28DA	F29SO	F29PD
F27SG	F	T	F	F
F28DA	F	F	T	T
F29SO	F	F	F	F
F29PD	F	F	F	F

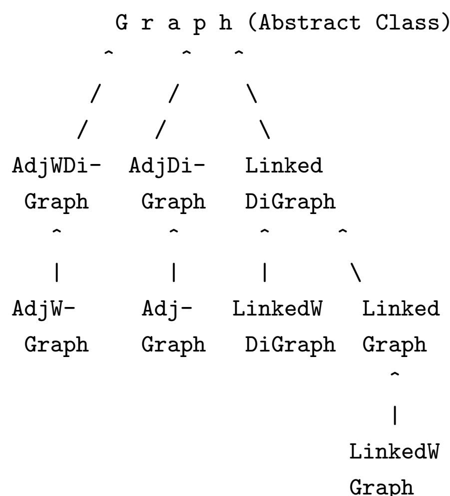
If array name is G, then G[n][m] = T if edge exists between node n and node m.

Exercise: Can you suggest a way to reduce the space requirements for undirected graphs?

Alternative Graph Class Structure

An alternative Graph class structure uses the notion that there are 4 possible types of graph: unweighted undirected, weighted undirected, unweighted digraphs, weighted digraphs;

Each can be implemented as an adjacency list or adjacency matrix.



Adjacency Matrix Digraph in Java

The relatively simple Java graph implementations and algorithms in the following lectures come from

> Sartaj Sahni, *Data Structures Algorithms and Applications in Java,* McGraw Hill, 2nd edition, 2004, ISBN: 0-92-930633-3.

By all means view the `jgrapht` implementation classes, but they are far more complex.

A simple edge class:

```
public class Edge
{
  int vertex1;  // one end point of the edge
  int vertex2;  // other end point of the edge

  public Edge(int theVertex1, int theVertex2)
  {
    vertex1 = theVertex1;
    vertex2 = theVertex2;
  }
}
```



```
public abstract class Graph
{
   // ADT methods
   public abstract int vertices();
   public abstract int edges();
   public abstract boolean existsEdge(int i, int j);
   public abstract void putEdge(Object theEdge);
   public abstract void removeEdge(int i, int j);
   public abstract int degree(int i);
   public abstract int inDegree(int i);
   public abstract int outDegree(int i);
...
}
```



```
public class AdjacencyDigraph extends Graph
{
  int n;              // number of vertices
  int e;              // number of edges
  boolean [][] a;    // adjacency array

  // constructors
  public AdjacencyDigraph(int theVertices)
  {
    // validate theVertices
    if (theVertices < 0)
      throw new IllegalArgumentException
                ("number of vertices must be >= 0");
    n = theVertices;
    a = new boolean [n + 1] [n + 1];
    // default values are e = 0 and a[i][j] = false
  }

  // default is a 0 vertex graph
  public AdjacencyDigraph()
    {this(0);}
```

```
public int vertices()
  {return n;}

public int edges()
  {return e;}

public void putEdge(Object theEdge)
{
  Edge edge =  (Edge) theEdge;
  int v1 = edge.vertex1;
  int v2 = edge.vertex2;
  if (v1 < 1 || v2 < 1 ||
      v1 > n || v2 > n || v1 == v2)
    throw new IllegalArgumentException
      ("(" + v1 + "," + v2 + ") not a valid edge");

  if (!a[v1][v2])  // new edge
    {a[v1][v2] = true;
     e++; }
}
```



```
public void removeEdge(int i, int j)
{
  if (i >= 1 && j >= 1 &&
      i <= n && j <= n && a[i][j])
    {a[i][j] = false;
     e--;}
}

/* undefined for directed graphs */
public int degree(int i)
  {throw new NoSuchMethodError
             ("AdjacencyDigraph:degree");}

public int outDegree(int i)
{
  if (i < 1 || i > n)
    throw new IllegalArgumentException
             ("no vertex " + i);

  // count out edges from vertex i
  int sum = 0;
  for (int j = 1; j <= n; j++)
    if (a[i][j]) sum++;
  return sum;
}
```


Exercise: Write the `boolean existsEdge(int i, int j)` method for this class. **Hint:** Check that `i` and `j` are valid vertices.

Exercise: Write the `int inDegree(int i)` method for this class.

Adjacency Lists

An **adjacency list** for a vertex i is a list of verticies adjacent to i.

An adjacency list representation of a graph is an array of adjacency lists, one for each vertex.

```
F27SG    [F28DA]
F28DA    [F29SO, F29PD]
F29SO     []
F29PD    []
```

Adjacency List Digraph in Java

The adjacency list may be implemented using a **list** or an **array**.

The following implementation uses a linked list class, `GraphChain`, with familiar methods:

- `add` an element at a specified position to the list
- `indexOf` search for an element and return its index
- `remove` the i-th element
- `removeElement` to remove a given element

The elements in the linked list are instances of `EdgeNode`, with only one field `int vertex`.


```
public class LinkedDigraph extends Graph
{
  int n;                 // number of vertices
  int e;                 // number of edges
  GraphChain [] aList;   // an array of adjacency lis

  public LinkedDigraph(int theVertices)
  {
    // validate theVertices
    if (theVertices < 0)
      throw new IllegalArgumentException
                ("number of vertices must be >= 0");
    n = theVertices;
    aList = new GraphChain [n + 1];
    for (int i = 1; i <= n; i++)
      aList[i] = new GraphChain();
    // default value of e is 0
  }

  public boolean existsEdge(int i, int j)
  {
    if (i < 1 || j < 1 || i > n || j > n
        || aList[i].indexOf(new EdgeNode(j)) == -1)
      return false;
    else
```



```
      return true;
  }

  public void putEdge(Object theEdge)
  {
    Edge edge =  (Edge) theEdge;
    int v1 = edge.vertex1;
    int v2 = edge.vertex2;
    if (v1 < 1 || v2 < 1 ||
        v1 > n || v2 > n || v1 == v2)
      throw new IllegalArgumentException
            ("(" + v1 + "," + v2 +
             ") is not a permissible edge");
                                    // new edge
    if (aList[v1].indexOf(new EdgeNode(v2)) == -1)
    {
      // put v2 at front of chain aList[v1]
      aList[v1].add(0, new EdgeNode(v2));
      e++;
    }
  }
```

Exercise: Write the `int outDegree(int i)` and `int inDegree(int i)` methods for this class.

Comparison

Adjacency matrix is:

- Easy to implement
- Most operations are efficient.

but it uses a large array, and inefficient for **sparse** graphs.

Edge lists are:

- A little harder to implement.
- Some operations are less efficient, as require list traversal.

but it is much more efficient in terms of space, especially for sparse graphs.

Summary

- Graphs are used for problems where data consists of objects and relationships between objects.
- Graph = set of vertices (nodes) and set of edges between vertices.
- There are many different types of graphs: e.g. digraphs, weighted graphs.
- ADT needs operations to modify and inspect edges.
- Two main implementations: adjacency lists and adjacency matrix.