# Lecture 22

# Graph data structures

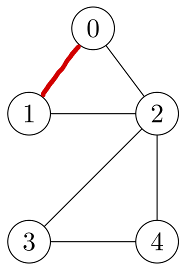
When representing a digraph in a computer, we assume that it has nodes given in a fixed order with the **convention that the nodes are labelled** $0, 1, \ldots, n-1$.

**Definition 22.1.** Let $G$ be a digraph of order $n$. The ***adjacency matrix*** of $G$ is the $n \times n$ Boolean matrix (often encoded with $0$'s and $1$'s) such that entry $(i, j)$ is true if and only if there is an arc from the node $i$ to node $j$.

**Definition 22.2.** For a digraph $G$ of order $n$, an ***adjacency lists*** representation is a sequence of $n$ sequences, $L_0, \ldots, L_{n-1}$. Sequence $L_i$ contains all nodes of $G$ that are out-neighbours of node $i$.

In the adjacency lists representation, $L_i$ may or may not be sorted in order of increasing node number. Our **convention is to sort them whenever convenient**. Many implementations do **not** enforce this convention.

**Example 22.3.** A graph and its adjacency matrix and adjacency list.

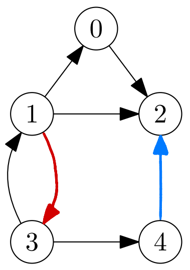


$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & \boxed{1} & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 1 & 0 \end{array}$$

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | | |
| 1 | 0 | 2 | | |
| 2 | 0 | 1 | 3 | 4 |
| 3 | 2 | 4 | | |
| 4 | 2 | 3 | | |

Notice that the number of $1$'s in a row in the adjacency matrix is the outdegree of the corresponding node, while the number of $1$'s in a column is the indegree.

**Example 22.4.** A digraph and its adjacency matrix and adjacency list.



$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & \boxed{1} & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 & 0 \end{array}$$

$$\begin{array}{c|ccc} 0 & 2 & & \\ 1 & 0 & 2 & 3 \\ 2 & & & \\ 3 & 1 & 4 & \\ \boxed{4} & \boxed{2} & & \end{array}$$

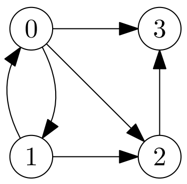
2 is a sink, therefore no out-neighbours

An empty sequence occurs in the adjacency list where a node has no out-neighbours (for example, sequence 2).

Sometimes the node labels in adjacency lists are omitted, so the digraph in Example 22.4 would be written as:

$$\begin{array}{ccc} 2 & & \\ 0 & 2 & 3 \\ & & \\ 1 & 4 & \\ 2 & & \end{array}$$

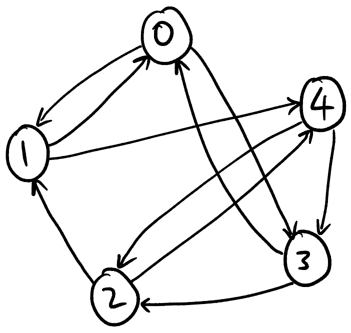
**Example 22.5.** Give the adjacency matrix of the digraph below.



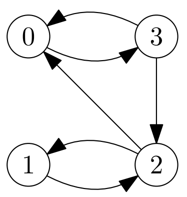
$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array} \text{(sink)}$$

Draw the digraph corresponding to the given adjacency matrix.

$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 2 & 0 & 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 \end{array}$$



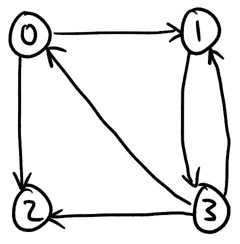
**Example 22.6.** Give the adjacency lists of the digraph below.



| | |
|---|---|
| 0 | 3 |
| 1 | 2 |
| 2 | 0 1 |
| 3 | 0 2 |

Draw the digraph corresponding to the given adjacency lists.

| | |
|---|---|
| 0 | 1 2 |
| 1 | 3 |
| 2 | |
| 3 | 0 1 2 |

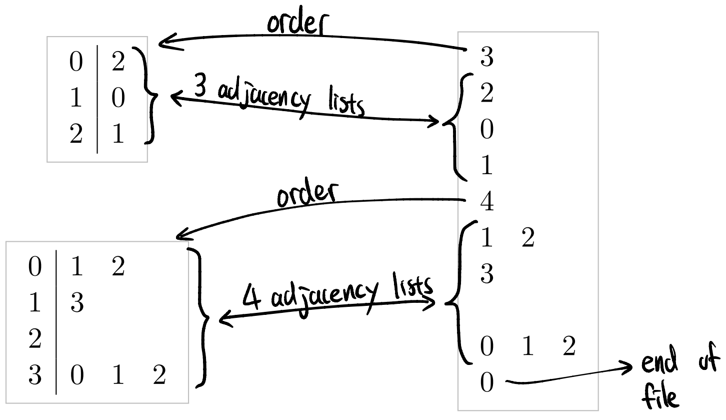


## 22.1 Representing multiple graphs in a single file

We can store several digraphs one after the other in a single file as follows:

- We have a single line giving the order at the beginning of each digraph.
- If the order is $n$ then the next $n$ lines give the adjacency matrix or adjacency lists representation of the digraph. Node labels are omitted.
- The end of the file is marked with a line denoting a digraph of order $0$.

**Example 22.7.** The two digraphs on the left could be put in a single file:



## 22.2 Using other structures to represent graphs

Other specialized digraph representations may be used to take advantage of special structure in a family of digraphs for improved storage or access time. For such specialized purposes they may be better than adjacency matrices or lists.

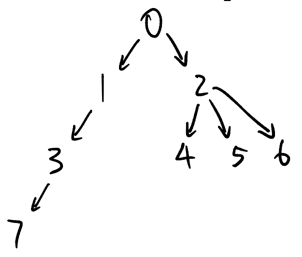
For example, trees can be stored more efficiently. When looking at heapsort we saw how to store a complete binary tree in an array. Here is way of storing a general tree in an array.

- A general rooted tree of $n$ nodes can be stored in array `pred` of size $n$.
- `pred`[$i$] is the parent of node $i$.
- The root has no parent, so assign it `null` or $-1$ if we number nodes from $0$ to $n-1$ in the usual way.
- This is a form of adjacency lists, using in-neighbours instead of out-neighbours.

**Example 22.8.** Draw the tree represented by the array

0 1 2 3 4 5 6 7

`pred` = $[-1, 0, 0, 1, 2, 2, 2, 3]$.



## 22.3 Implementation of digraph ADT

An adjacency matrix is simply a matrix which is an array of arrays.

Adjacency lists are a list of lists. There are several ways in which a list can be implemented, for example by an array, or singly- or doubly-linked lists using pointers. These have different properties, for example, accessing the middle element is $\Theta(1)$ for an array but $\Theta(n)$ for a linked list. Searching for a value that may or may not be in the list requires sequential search and takes $\Theta(n)$ time in the worst case. We do not consider other data structures (e.g. heaps) that can be used to represent lists.

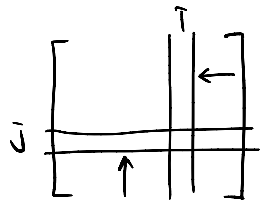
## 22.4 Complexity of basic digraph operations

The basic operations we consider are checking for the existence of an arc between two nodes, finding the outdegree of a node, finding the indegree of a node, adding an arc between two nodes, deleting an arc between two nodes, adding a node, and deleting a node.

For the two data structures, consider the steps we need to carry out various basic operations and the cost of all steps.

**Example 22.9.** Compare the matrix and lists data structures for checking whether arc $(i, j)$ exists.
**Adjacency matrix representation**: We need to check whether element $(i, j)$ is 1. This requires accessing an array element twice, to first find the $i$th array then its $j$th element. Each array access is in $\Theta(1)$ so overall it is in $\Theta(1)$.
**Adjacency lists representation**: We need to search for $j$ in list $i$. The complexity then depends on the length of list $i$. List $i$ is length $d$ where $d$ is the outdegree of node $i$ so searching for $j$ is in $\Theta(d)$. But how large is $d$? Even when the graph is sparse, it could still be the case that $d$ is $O(n)$, though typically in a sparse graph $d$ is $O(1)$. In a dense graph $d$ is $O(n)$.

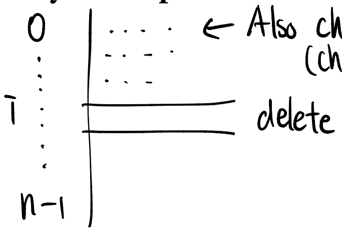
**Example 22.10.** Compare the matrix/lists data structures for deleting a node.
**Adjacency matrix representation**:



Need to shift rows below up, and columns right of (i) to left

$\therefore \Theta(n^2)$

**Adjacency lists representation**:



← Also check if (i) is in this list (check for indegrees of (i))

$\Theta(n+m+1) = O(n+m)$

Table 22.1 shows the steps required and Table 22.2 the time required for basic graph operations when using adjacency matrix or lists representations. Performance for the adjacency list representation is based on using doubly linked lists.

Table 22.1: Steps required to perform basic digraph operations by data structure.

| **Operation** | **Adjacency matrix** | **Adjacency lists** |
|---|---|---|
| arc $(i, j)$ exists? | is entry $(i, j)$ 0 or 1 | find $j$ in list $i$ |
| outdegree of $i$ | scan row, count 1's | size of list $i$ |
| indegree of $i$ | scan column, count 1's | for $j \neq i$, find $i$ in list $j$ |
| add arc $(i, j)$ | change entry $(i, j)$ | insert $j$ in list $i$ |
| delete arc $(i, j)$ | change entry $(i, j)$ | delete $j$ from list $i$ |
| add node | create new row and column | add new list at end |
| delete node $i$ | delete row/column $i$<br>shuffle other entries | delete list $i$<br>for $j \neq i$, delete $i$ from list $j$ |

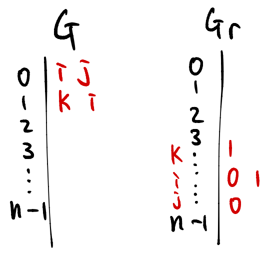
Table 22.2: Comparative worst-case performance of adjacency matrices and lists.

| **Operation** | **Adjacency matrix** | **Adjacency lists** |
|---|---|---|
| arc $(i, j)$ exists? | $\Theta(1)$ | $\Theta(d)$ |
| outdegree of $i$ | $\Theta(n)$ | $\Theta(1)$ |
| indegree of $i$ | $\Theta(n)$ | $\Theta(n + m)$ |
| add arc $(i, j)$ | $\Theta(1)$ | $\Theta(1)$ |
| delete arc $(i, j)$ | $\Theta(1)$ | $\Theta(d)$ |
| add node | $\Theta(n)$ | $\Theta(1)$ |
| delete node $i$ | $\Theta(n^2)$ | $\Theta(n + m)$ |

Using lists, apparently similar problems like finding the outdegree ($\Theta(1)$) and indegree ($\Theta(n + m)$) have very different time complexity.

Clearly finding all indegrees would be slow if we just used multiple calls to a method designed for getting the indegree of a single node.

**Example 22.11.** Show that the sorted adjacency lists representation of the reverse digraph of $G$ can be found in time $\Theta(n+m)$ given the sorted adjacency lists of $G$. Also show how this can be used to find indegrees for all nodes of $G$ in time $\Theta(n+m)$.



$u \rightarrow v = G$

$u \leftarrow v = Gr$

1) To find Gr, iterate through each list, in list i, if j is present add i to jth list of Gr

n list combined with m items, $\therefore n+m$, inserting node $= \Theta(1)$,

$\therefore$ Total time $= \Theta(n+m)$

2) Indegree of u in G is outdegree of u in Gr

outdegree $= \Theta(1)$, want to do n times $\therefore$ Can do so in $\Theta(n+m)$

(but requires extra space)

## 22.5 Space requirements

The adjacency matrix representation requires $\Theta(n^2)$ storage as we simply need a matrix of $n^2$ bits.

At first guess we might say adjacency lists require $\Theta(n+m)$ storage, since we need $n$ lists and the combined length of all the lists is $m$. But node numbers require more than one bit of storage each; the number $k$ uses about $\Theta(\log k)$ bits. The average entry in a list is $\frac{n}{2}$, so the total space requirement is more like $\Theta(n + m \log(n))$

**Example 22.12.** What is the storage requirement for a complete digraph on *n* nodes, that is, a digraph where every possible arc occurs?

Adjacency matrix $= n^2$

Adjacency list $= n + m \log n$

$= n + n(n-1) \log n$

$= n + n^2 \log n$

For sparse digraphs, lists use less space than a matrix, whereas for dense digraphs the space requirements are comparable.

The representation that is best will depend on the application and we cannot make general rules. We will mostly use adjacency lists, which are clearly superior for many common tasks and generally better for sparse digraphs.