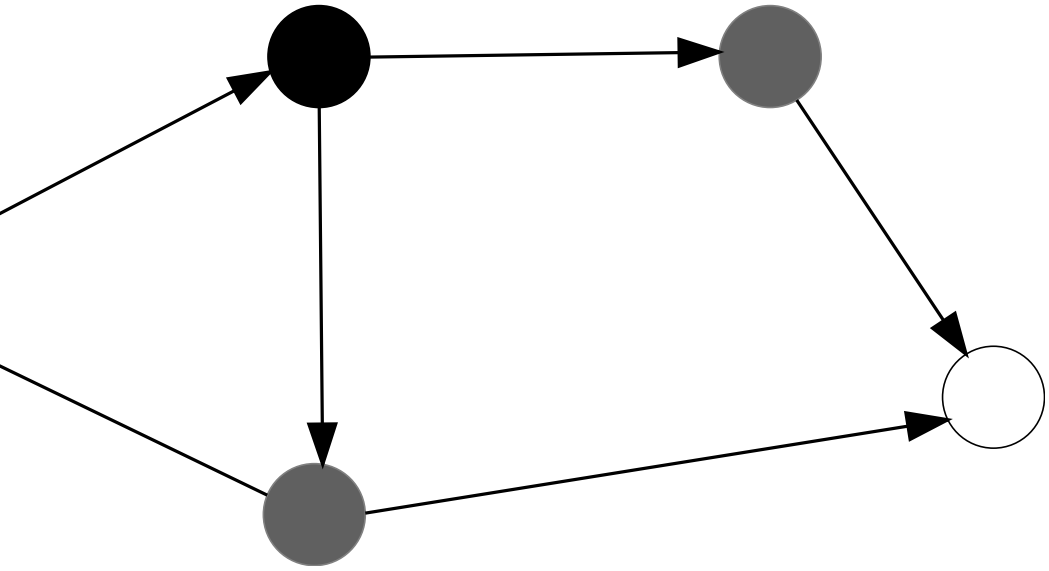
MICHAEL J. DINNEEN GEORGY GIMEL'FARB MARK C. WILSON



# Part I

# The Graph Abstract Data Type

# Chapter 1

# Lecture: Graph definitions

Graphs are important and general mathematical objects that are widely used in theory and practice.

Graphs distil the basic idea of a relationship among a set of objects.

Informally we can think of a graph as a collection of dots (the set of objects) with lines connecting them (describing the relationship). The lines can be either directed (arrows) or undirected.

We are interested in the algorithmic aspects of graph theory ("how can we do it efficiently and systematically?"). To talk about this precisely, we must start with precise definitions.

We start with the concept a **di**rected **graph**, or digraph.

**Definition 1.1.** A ***digraph*** $G = (V, E)$ is a finite nonempty set $V$ of ***nodes*** together with a (possibly empty) set $E$ of ordered pairs of nodes of $G$ called ***arcs***. Digraph stands for **di**rected **graph**).

**Example 1.2.** For the graph shown in Figure 1.1, write down the set $V$ and the set $E$

**Definition 1.3.** A ***graph*** $G = (V, E)$ is a finite nonempty set $V$ of ***vertices*** together with a (possibly empty) set $E$ of unordered pairs of vertices of $G$ called ***edges***. Note that the singular of vertices is ***vertex***.

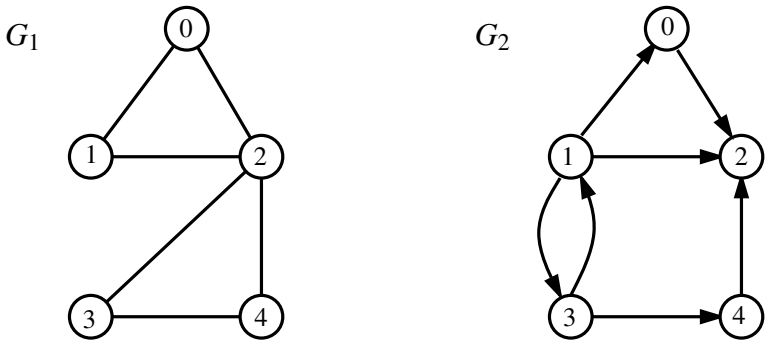



Figure 1.1: A graph $G_1$ and a digraph $G_2$.

**Example 1.4.** For the digraph shown in Figure 1.1, write down the set $V$ and the set $E$

*Note.* Graphs vs Digraphs

- Since we defined $E$ to be a set, there are no multiple arcs/edges between a given pair of nodes/vertices.
- For a digraph $G$ we sometimes denote the set of nodes by $V(G)$ and the set of arcs by $E(G)$ for clarity.

- A graph can be viewed as a digraph where every unordered edge $\{u,v\}$ is replaced by two directed arcs $(u,v)$ and $(v,u)$. This works in most instances and has the advantage of allowing us to consider only digraphs.
- Sometimes we must know whether our object is really a graph or just a symmetric digraph. Whenever there is a potential ambiguity, we shall point it out.
- An arc that begins and ends at the same node is called a ***loop***. We make the convention that *loops are not allowed in our digraphs*.
- Some authors use "undirected graph" to mean graph and use the term "graph" to mean what we call a directed graph. We will always use digraph and graph.
- In order to save writing "(di)graph" too many times, we treat the digraph as the fundamental concept.
- When we say something about digraphs, nodes and arcs, it is understood to also hold for graphs, nodes and edges unless explicitly stated otherwise.
- However, if we talk about graphs, edges, and vertices, our statement is not necessarily true for digraphs.

**Definition 1.5.** If $(u,v) \in E$ (that is, if there is an arc going from $u$ to $v$ we say that $v$ is ***adjacent*** to $u$, that $v$ is an ***out-neighbour*** of $u$, and that $u$ is an ***in-neighbour*** of $v$. If an (undirected) graph $G$, if $\{u,v\} \in E$, then $u$ is a ***neighbour*** of $v$ and $v$ is a neighbour of $u$.

**Example 1.6.** In the digraph $G_2$ in Figure 1.1, find all in-neighbours of node 2

**Definition 1.7.** The ***order*** of a digraph $G = (V,E)$ is $|V|$, the number of nodes. The ***size*** of $G$ is $|E|$, the number of arcs. We usually use $n$ to denote $|V|$ and $m$ to denote $|E|$.

For a given order $n$, the size $m$ can be as low as 0 (a digraph consisting of $n$ nodes and no arcs) and as high as $n(n-1)$ (each node can point to each other node; recall that we do not allow loops).

**Definition 1.8.** If $m$ is toward the low end, the digraph is called ***sparse***, and if $m$ is toward the high end, then the digraph is called ***dense***. These terms are obviously very informal. For our purposes we will call a class of digraphs sparse if $m$ is $O(n)$ and dense if $m$ is $\Omega(n^2)$.

**Definition 1.9.** A ***walk*** in a digraph $G$ is a sequence of nodes $v_0 v_1 \dots v_l$ such that, for each $i$ with $0 \leq i < l$, $(v_i, v_{i+1})$ is an arc in $G$.

The ***length*** of the walk $v_0 v_1 \dots v_l$ is the number $l$ (that is, the number of arcs involved).

A ***path*** is a walk in which no node is repeated.

A ***cycle*** is a walk in which $v_0 = v_l$ and no other nodes are repeated.

Note: in a graph, a walk of the form $uvu$ is not considered a cycle (going back and forth along the same edge should not count as a cycle). A cycle in a graph must be of length at least 3.

**Example 1.10.** For the graph $G_1$ of Figure 1.1 the following sequences of vertices are classified as being walks, paths, or cycles.

| vertex sequence | walk? | path? | cycle? |
|---|---|---|---|
| 032 | no | no | no |
| 01234 | yes | yes | no |
| 0120 | yes | no | yes |
| 123420 | yes | no | no |
| 010 | yes | no | no |

**Example 1.11.** Show that if there is a walk from $u$ to $v$, then we can find a path from $u$ to $v$.

**Definition 1.12.** In a graph, the ***degree*** of a vertex $v$ is the number of edges meeting $v$.

In a digraph, the ***outdegree*** of a node $v$ is the number of out-neighbours of $v$, and the ***indegree*** of $v$ is the number of in-neighbours of $v$.

A node of indegree 0 is called a ***source*** and a node of outdegree 0 is called a ***sink***.

**Definition 1.13.** The ***distance*** from $u$ to $v$ in $G$, denoted by $d(u,v)$, is the minimum length of a path from $u$ to $v$. If no path exists, the distance is undefined (or $+\infty$).

For graphs, we have $d(u,v) = d(v,u)$ for all vertices $u, v$.

**Example 1.14.** In graph $G_1$ of Figure 1.1, we can see by considering all possibilities that $d(0,1) = 1$, $d(0,2) = 1$, $d(0,3) = 2$, $d(0,4) = 2$, $d(1,2) = 1$, $d(1,3) = 2$, $d(1,4) = 2$, $d(2,3) = 1$, $d(2,4) = 1$ and $d(3,4) = 1$.

In digraph $G_2$, we have, for example, $d(0,2) = 1, d(3,2) = 2$. Since node 2 is a sink, $d(2,v)$ is not defined unless $v = 2$, in which case the value is 0.

## 1.1 Creating new digraphs from old ones

There are several ways to create new digraphs from old ones.

One way is to delete (possibly zero) nodes and arcs in such a way that the resulting object is still a digraph (there are no arcs missing any endpoints!).

**Definition 1.15.** A ***subdigraph*** of a digraph $G = (V,E)$ is a digraph $G' = (V',E')$ where $V' \subseteq V$ and $E' \subseteq E$. A ***spanning*** subdigraph is one with $V' = V$; that is, it contains all nodes.

**Example 1.16.** Figure 1.2 shows (on the left) a subdigraph and (on the right) a spanning subdigraph of the digraph $G_2$ of Figure 1.1.


Figure 1.2: A subdigraph and a spanning subdigraph of $G_2$.

**Definition 1.17.** The subdigraph ***induced*** by a subset $V'$ of $V$ is the digraph $G' = (V',E')$ where $E' = \{(u,v) \in E \mid u \in V' \text{ and } v \in V'\}$.

**Example 1.18.** Figure 1.3 shows the subdigraph of the digraph $G_2$ of Figure 1.1 induced by $\{1,2,3\}$.

It is sometimes useful to "reverse all the arrows". This produces the "reverse digraph".

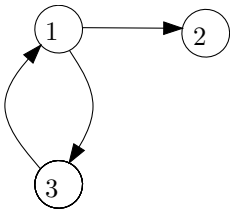



Figure 1.3: The subdigraph of $G_2$ induced by $\{1,2,3\}$.

**Definition 1.19.** The ***reverse digraph*** of the digraph $G = (V,E)$, is the digraph $G_r = (V,E')$ where $(u,v) \in E'$ if and only if $(v,u) \in E$.

**Example 1.20.** Figure 1.4 shows the reverse of the digraph $G_2$ of Figure 1.1.

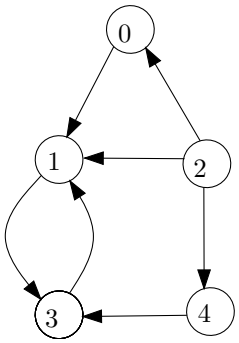



Figure 1.4: The reverse of digraph $G_2$.

It is sometimes useful to ignore the direction of arcs in a digraph to find the associated 'underlying graph".

**Definition 1.21.** The ***underlying graph*** of a digraph $G = (V,E)$ is the graph $G' = (V,E')$ where $E' = \{\{u,v\} \mid (u,v) \in E\}$.

Note: the underlying graph does not have multiple edges even when there are arcs $(u,v)$ and $(v,u)$. In that case, only one edge joins $u$ and $v$ in the underlying graph $G'$. This is because $\{u,v\}$ and $\{v,u\}$ are equal as sets, so appear only once in the set $E'$.

**Example 1.22.** Figure 1.5 shows the underlying graph of the digraph $G_2$ of Figure 1.1.

**Definition 1.23.** We can combine two or more digraphs $G_1, G_2, \ldots G_k$ into a single graph where the vertices of each $G_i$ are completely disjoint from each other and no arc goes between the different $G_i$. The constructed graph $G$ is called the ***graph union***, where $V(G) = V(G_1) \cup V(G_2) \cup \ldots \cup V(G_k)$ and $E(G) = E(G_1) \cup E(G_2) \cup \ldots \cup E(G_k)$.

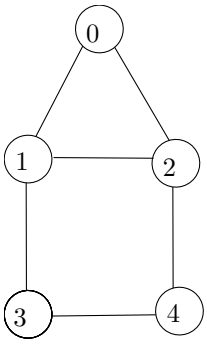



Figure 1.5: The underlying graph of $G_2$. Note that there is only a single edge between vertices 1 and 3.

# Chapter 2

# Lecture: Graph data structures

There are two common data structures used to representations for digraph in computers. For both structures, we assume that the digraph has nodes given in a fixed order with the *convention* that the nodes are labelled $0, 1, \ldots, n-1$.

**Definition 2.1.** 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 2.2.** 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 adjacent to node $i$.

In the adjacency lists representation, only the out-neighbours of node $i$ are listed in the sequence $L_i$. $L_i$ may or may not be sorted in order of increasing node number. Our *convention* is to sort them whenever it is convenient. Many implementations do *not* enforce this convention.

**Example 2.3.** For the graph $G_1$ and digraph $G_2$ of Example **??**, the adjacency matrices are

$$G_1: \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \qquad G_2: \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Notice that the number of 1's in a row is the out-degree of the corresponding node, while the number of 1's in a column is the in-degree.

**Example 2.4.** For the graph $G_1$ and digraph $G_2$ of Example **??**, the adjacency lists are

$G_1$ :

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

$G_2$ :

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

An empty sequence occurs where a node has no out-neighbours (for example, sequence 2 of the digraph $G_2$).

We may include node labels in the adjacency lists for clarity or where nodes are not numbered in the usual way, for example

$G_2$ :

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

## 2.1 Representing multiple graphs in a single file

It is often useful to input several digraphs from a single file. Our standard format is as follows. The file consists of several digraphs one after the other. To distinguish the beginning of one and the end of the other we have a single line giving the order at the beginning of each graph. If the order is $n$ then the next $n$ lines give the adjacency matrix or adjacency lists representation of the digraph. The end of the file is marked with a line denoting a digraph of order 0.

**Example 2.5.** Here is a file with multiple digraphs in the adjacency lists format. Draw the corresponding digraphs.

```
4
1  2
3

0  1  2
3
2
0
1
0
```

**Example 2.6.** Here is a file with multiple digraphs in the adjacency matrix format. Draw the corresponding digraphs.

TODO

## 2.2 Using other structures to represent graphs

There are also other specialized (di)graph representations besides the two mentioned in this section. These data structures take advantage of special structure for improved storage or access time, often for families of graphs sharing a common property or to optimise certain operations. For such specialized purposes they may be better than either the adjacency matrix or lists representations.

For example, trees can be stored more efficiently. We have already seen in Section **??** how a complete binary tree can be stored in an array. A general rooted tree of $n$ nodes can be stored in an array *pred* of size $n$. The value $pred[i]$ gives the parent of node $i$. The root is a special case and can be given value $-1$ (representing a NULL pointer), for example, if we number nodes from $0$ to $n-1$ in the usual way. This of course is a form of adjacency lists representation, where we use in-neighbours instead of out-neighbours.

**Example 2.7.** Draw the tree represented by the array $pred = [-,0,0,1,2,2,2,3]$

## 2.3 Implementation of digraph ADT operations

In this section we discuss implementing basic operations on digraphs such as checking the existence of an arc or deleting a node and compare the performance of different data structures.

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

Adjacency lists are a list of lists, and there are several ways in which a list can be implemented. For example by an array or singly- or doubly-linked list using pointers which have different properties: e.g., accessing the middle element is $\Theta(1)$ for an array but $\Theta(n)$ for a linked list.

Table 2.1: Digraph operations in terms of data structures.

| **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 $j$ |

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.

More complex data structures such as binary trees or heaps can be used to represent lists often with some benefits but we do not consider them here.

## 2.4 Complexity of basic digraph operations

**Example 2.8.** 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 (the $j$th element of the $i$th array). Each array access is a $\Theta(1)$ operation so overall it is $\Theta(1)$.

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 out-degree of node $i$ so searching for $j$ is $\Theta(d)$. 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 2.9.** Deleting a node (which perhaps necessitates deleting some arcs) is trickier. In the matrix case, we must delete a row and column, and move up some elements so there are no gaps in the matrix. In the lists case, we must remove a list and also all references to the deleted node in other lists. This requires scanning each list for the offending entry and deleting it.

Table 2.1 summarizes how the basic graph operations can be described in terms of the adjacency matrix or lists representations while Table 2.2 compares the performance of two different data structures. Performance for the adjaceny list representation is based on using doubly linked lists.

Finding outdegree with the lists representation merely requires accessing the correct list (constant time) plus finding the size of that list (constant time). Find-

Table 2.2: Comparative worst-case performance of adjacency lists and matrices.

| **Operation** | **matrix** | **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 $i$ | $\Theta(n^2)$ | $\Theta(n+m)$ |

ing indegree with the lists representation requires scanning all lists except one, and this requires us to look at every arc in the worst case, taking time $\Theta(n+m)$ (the $n$ is because we must consider every node's list even if it is empty). If we wish to compute just one indegree, this might be acceptable, but if all indegrees are required, this will be inefficient. It is better to compute the reverse digraph once and then read off the outdegrees, this last step taking time $\Theta(n)$ (see Exercise **??**).

One way around all this work is to use in our definition of adjacency lists representation, instead of just the out-neighbours, a list of in-neighbours also. This may be useful in some contexts but in general requires more space than is needed.

## 2.5 Space requirements

We also need to consider the space requirements for each implementation.

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

At first guess we might say adjacency lists representation requires $\Theta(n+m)$ storage, since we need $n$ lists and the combined length of all the lists is $m$ (each arc appears exactly once). However this is not strictly true as node numbers require more than one bit of storage each. The number $k$ requires on average $\Theta(\log k)$ bits so the space requirement is more like $\Theta(n+m\log(n))$

**Example 2.10.** What is the storage requirement for a complete digraph on $n$ nodes, that is, a digraph where every possible arc occurs?

For small, sparse digraphs, it is true that lists use less space than a matrix, whereas for small dense digraphs the space requirements are comparable. For large sparse digraphs, a matrix can still be more efficient, but this happens rarely.

As with most implementation considerations, the representation that is best will depend on the context and we cannot make general rules. We will mostly use

adjacency lists, which are clearly superior for many common tasks (such as graph traversals, covered in Chapter **??**) and generally better for sparse digraphs.