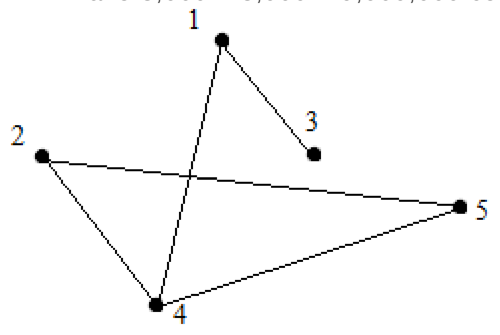
Data Structure for Graphs

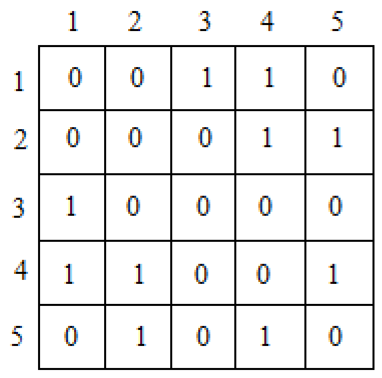
There are several possible ways to represent graphs. We discuss four useful representations below. We assume the graph G = (V,E) contains n vertices and m edges.

1. Adjacency Matrix — We can represent G using an n×n matrix M, where element M[i, j] is, say, 1, if (i, j) is an edge of G, and 0 if it isn't. This allows fast answers to the question "is (i, j) in G?", and rapid updates for edge insertion and deletion. It may use excessive space for graphs with many vertices and relatively few edges,however.

Consider a graph which represents the street map of Manhattan in New York City. Every junction of two streets will be a vertex of the graph, with neighboring junctions connected by edges. How big is this graph? Manhattan is basically a grid of 15 avenues, each crossing roughly 200 streets. This gives us about 3,000 vertices and 6,000 edges, since each vertex neighbors four other vertices and each edge is shared between two vertices. Such a small amount of data should easily and efficiently stored, but the adjacency matrix will have 3,000 × 3,000 = 9,000,000 cells, almost all of them empty!



Adjacency matrix for above graph will be represented as follow :

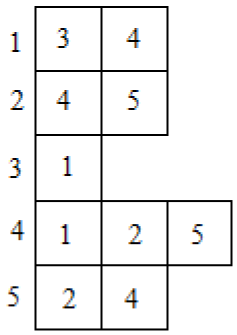


	1	2	3	4	5
1	0	0	1	1	0
2	0	0	0	1	1
3	1	0	0	0	0
4	1	1	0	0	1
5	0	1	0	1	0

2. Adjacency Lists in Lists — We can more efficiently represent sparse graphs by using linked lists to store the neighbors adjacent to each vertex. Adjacency lists require pointers but are not frightening once you have experience with linked structures.

Adjacency lists make it harder to ask whether a given edge (i, j) is in G, since we have to search through the appropriate list to find the edge. However, it is often surprisingly easy to design graph algorithms which avoid any need for such queries. Typically, we sweep through all the edges of the graph in one pass via a breadth-first or depths-first traversal, and update the implications of the current edge as we visit it.

3. Adjacency Lists in Matrices — Adjacency lists can also embedded in matrices, thus eliminating the need for pointers. We can represent a list in an array (or equivalently, a row of a matrix) by keeping a count k of how many elements there are, and packing them into the first k elements of the array. Now we can visit successive the elements from the first to last just like a list, but by incrementing an index in a loop instead of cruising through pointers.
Adjacency Lists in matrix for above graph is :



1	3	4	
2	4	5	
3	1		
4	1	2	5
5	2	4	

This data structure looks like it combines the worst properties of adjacency matrices (large space) with the worst properties of adjacency lists (the need to search for edges). However, there is a method to its madness. First, it is the simplest data structure to program, particularly for static graphs which do not change after they are built. Second, the space problem can in principle be eliminated by allocating the rows for each vertex dynamically, and making them exactly the right size.

To prove our point, we will use this representation in all our examples below.

4. Table of Edges — An even simpler data structure is just to maintain an array or linked list of the edges. This is not as flexible as the other data structures at answering "who is adjacent to vertex x?" but it works just fine for certain simple procedures like Kruskal's minimum spanning tree algorithm.

As stated above, we will use adjacency lists in matrices as our basic data structure to represent graphs. It is not complicated to convert these routines to honest pointer based adjacency lists.

We represent a graph using the following data type. For each graph, we keep count of the number of vertices, and assign each vertex a unique number from 1 to n vertices. We represent the edges in an

MAXV × MAXDEGREE array, so each vertex can be adjacent to MAXDEGREE others. By defining MAXDEGREE to be MAXV, we can represent any simple graph, but this is wasteful of space for low-degree graphs:

```
#define MAXV 100     /* maximum number of vertices */
#define MAXDEGREE 50     /* maximum vertex outdegree */
typedef struct {
int edges[MAXV+1][MAXDEGREE];     /* adjacency info */
int degree[MAXV+1];     /* outdegree of each vertex */
int nvertices;     /* number of vertices in graph */
int nedges;     /* number of edges in graph */
} graph;
```

We represent a directed edge (x, y) by the integer y in x’s adjacency list, which is located in the sub-array graph → edges[x]. The degree field counts the number of meaningful entries for the given vertex. An undirected edge (x, y) appears twice in any adjacency-based graph structure, once as y in x’s list, and once as x in y’s list.

To demonstrate the use of this data structure, we show how to read in a graph from a file. A typical graph format consists of an initial line featuring the number of vertices and edges in the graph, followed by a listing of the edges at one vertex pair per line.

```
read_graph(graph *g, bool directed)
{
int i; /* counter */
int m; /* number of edges */
int x, y; /* vertices in edge (x,y) */
initialize_graph(g);
scanf("%d %d",&(g→nvertices),&m);
for (i=1; i i ≤ m; i++) {
scanf("%d %d",&x,&y);
insert_edge(g,x,y,directed);
}
}
initialize_graph(graph *g)
{
int i; /* counter */
g → nvertices = 0;
g → nedges = 0;
for (i=1; i ≤ MAXV; i++) g→degree[i] = 0;
}
```

The critical routine is insert edge.We parameterize it with a Boolean flag directed to identify whether we need to insert two copies of each edge or only one. Note the use of recursion to solve the problem:

```
insert_edge(graph *g, int x, int y, bool directed)
{
if (g→degree[x] > MAXDEGREE)
printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);
g→edges[x][g→degree[x]] = y;
g→degree[x] ++;
if (directed == FALSE)
insert_edge(g,y,x,TRUE);
else
g→nedges ++;
}
```

Printing the associated graph is now simply a matter of nested loops:

```
print_graph(graph *g)
{
int i,j; /* counters */
for (i=1; i≤ g→nvertices; i++) {
printf("%d: ",i);
for (j=0; j < g→degree[i]; j++)
printf(" %d",g→edges[i][j]);
printf("\n");
}
}
```

Source:

http://www.learnalgorithms.in/#