Role of Adjacency Matrix & Adjacency List in Graph Theory

Harmanjit Singh
Assistant Professor
PCTE Baddowal
Ludhiana,Punjab,India

Richa Sharma
Assistant Professor
PCTE Baddowal
Ludhiana,Punjab,India

ABSTRACT
Today, graph theory has become major instrument that is used in an array of fields. Some of these include electrical engineering, mathematical research, sociology, economics, computer programming/networking, business administration and marketing. Indeed, many problems can be modeled with paths formed by traveling along the edges of a certain graph. Frequently referenced problems are efficiently planning routes for mail delivery, garbage pickup and snow removal, which can be solved using models that involve paths in graphs. Given these kinds of problems, graphs can become extremely complex, and a more efficient way of representing them is needed in practice. This is where the concept of the adjacency matrix & adjacency list comes into play.

Keywords
Adjacency matrix, adjacency list, Path matrix, Edge list, Node list

1. INTRODUCTION

There are two standard ways of representing or maintaining a graph G in memory of a computer.

1. Sequential representation of G using adjacent matrix.
2. Linked representation of G using adjacent list.

Regardless of the way one maintains a graph G in the memory of the computer, the graph G is normally input into the computer by using its formal definition: a collection of nodes and a collection of edges.

Sequential representation of a graph

Adjacency Matrix---- In mathematics and computer science, an adjacency matrix is a means of representing which vertices of a graph are adjacent to which other vertices. Specifically, the adjacency matrix of a finite graph G on n vertices is the n × n matrix where the non diagonal entry aij is the number of edges from vertex i to vertex j, and the diagonal entry aii, depending on the convention, is either once or twice the number of edges (loops) from vertex i to itself. Undirected graphs often use the former convention of counting loops twice, whereas directed graphs typically use the latter convention. There exists a unique adjacency matrix for each graph (up to permuting rows and columns), and it is not the adjacency matrix of any other graph. In the special case of a finite simple graph, the adjacency matrix is a (0, 1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric.

Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called v1, v2…, vm. Then the adjacency matrix A = (aij) of the graph G is the m × m matrix defined as follows:

aij = { 1 if vi is adjacent to vj, i.e.; if there is an edge (vi, vj).
0 otherwise}

Such a matrix A, which contains entries of only 0 and 1, is called a **Bit matrix or a Boolean matrix**.

The adjacency matrix A of graph G does depend on the ordering of the nodes of G, that is, a different ordering of nodes may result in a different adjacency matrix. However, the matrices resulting from two different orderings are closely related in that one can be obtained from the other by simply interchanging rows and columns. Unless otherwise stated, we will assume that the nodes of our graph G have a fixed ordering.

Suppose G is an undirected graph. Then the adjacency matrix A of G will be a symmetric matrix, i.e; one in which aij = aji for every i and j. This follows from the fact that each undirected edge [u, v] corresponds to the two directed edges (u, v) and (v, u).

The above matrix representation of a graph may be extended to multigraphs. Specifically, if G is a multigraph, then the adjacency matrix of G is the m × m matrix A = (aij) defined by setting aij, equal to the number of edges from vi to vj.

Consider the graph G in Fig. 1. Suppose the nodes are stored in memory in a linear array DATA as follows:

DATA: X, Y, Z, W

Then we assume that the ordering of the nodes in G is as follows: v_1=X, v_2=Y, v_3=Z and v4=W. The adjacency matrix A of G is as follows:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Note that the number of 1's in A is equal to the number of edges in G.

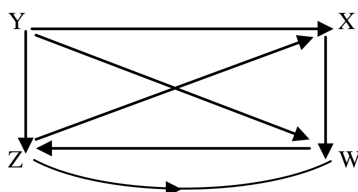


Fig. 1

Consider the powers A, A2, A3… of the adjacency matrix A of a graph G. Let ak (i,j) = the ij entry in the matrix Ak.

Observe that a1 (i,j) = aij gives the number of paths of length 1 from node vi to node vj. One can show that a_2 (i,j) gives the number of paths of length 2 from v_i to v_j.

Adjacency matrix of a bipartite graph-----

The adjacency matrix A of a bipartite graph whose parts have r and s vertices has the form:

$$A = \begin{pmatrix} O & B \\ B^T & O \end{pmatrix},$$

where B is an r × s matrix and O is an all-zero matrix. Clearly, the matrix B uniquely represents the bipartite graphs, and it is commonly called its biadjacency matrix. Formally, let G = (U, V, E) be a bipartite graph or bigraph with parts U = u1... ur and V = v1,...,vs.

An *r* x *s* 0-1 matrix *B* is called the **biadjacency matrix** if $B_{i,j}$

= 1 iff .

If G is a bipartite multigraph or weighted graph then the elements Bi,j are taken to be the number of edges between or the weight of (ui,vj) respectively.

Path Matrix----

Let G be a simple directed graph with m nodes, v1, v2……, vm. The path matrix or reach ability matrix of G is the m-square matrix P= (pij) defined as follows:

P_{ij}= {1 if there is a path from v_i to v_j

{0 otherwise

Suppose there is a path from vi to vj. Then there must be a simple path from vi to vj when vi ≠ vj, or there must be a cycle from vi to vj when vi = vj. Since G has only m nodes, such a simple path must have length m – 1 or less, or such a cycle must have length m or less. This means that there is a non- zero ij entry in the matrix Bm.

Variations---

The Seidel adjacency matrix or (0,−1,1)-adjacency matrix of a simple graph has zero on the diagonal and entry aij = − 1 if ij is an edge and +1 if it is not. This matrix is used in studying strongly regular graphs and two-graphs. A distance matrix is like a higher-level adjacency matrix. Instead of only providing information about whether or not two vertices are connected, also tells the distances between them. This assumes the length of every edge is 1. A variation is where the length of an edge is not necessarily 1.

Linked representation of graph

Adjacency list-----

In graph theory, an adjacency list is the representation of all edges or arcs in a graph as a list. If the graph is undirected, every entry is a set (or multiset) of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc. Typically, adjacency lists are unordered. Let G be a directed graph with m nodes. The sequential representation of G in memory---- i.e; the representation of G by its adjacency matrix A--- has a number of major drawbacks. First of all, it may be difficult to insert and delete nodes in G. This is because the size of A may need to be changed and the nodes may need to be reordered, so there may be many changes in the matrix A. Furthermore, if the number of edges is O(m) or O(m log2 m), then the matrix A will be a sparse(will contain many zeros); hence a great deal of space will be wasted. Accordingly, G is usually represented in memory by a linked representation, also called an **adjacency structure**.

Consider the graph G in Fig. 2(a). The table in Fig. 2(b) shows each node in G followed by its **adjacency list**, which is its list of adjacent nodes, also called its successors or neighbors. Fig. 2(c) shows a schematic diagram of a linked representation of G in memory.

Specifically, the linked representation will contain two lists (or files), a node list NODE and an edge list EDGE, as follows:

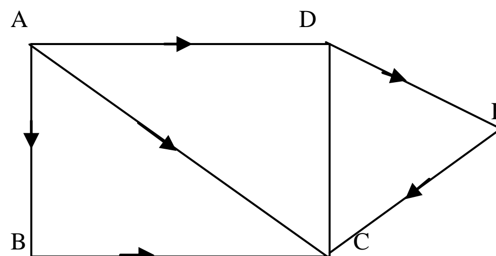


Fig. 2(a) Graph G

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

Fig. 2(b) Adjacency lists of G

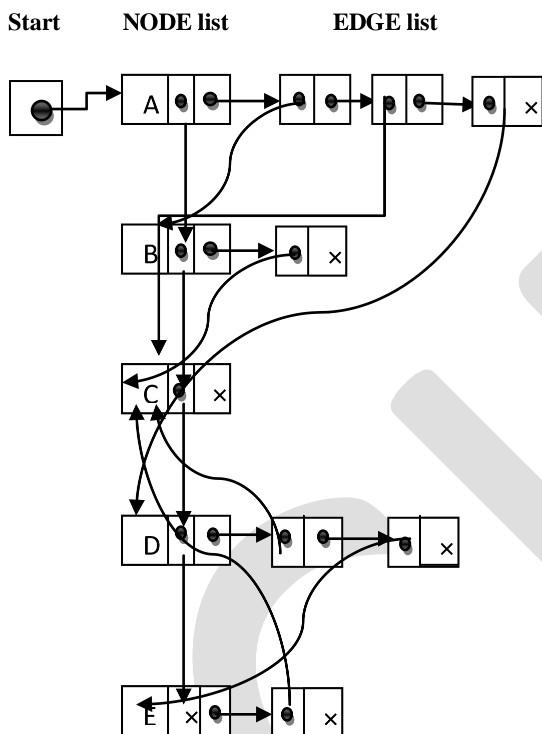


Fig. 2(c)

(a) Node list--- Each element in the list NODE will correspond to a node in G, and it will be a record of the form:

NODE	NEXT	ADJ	

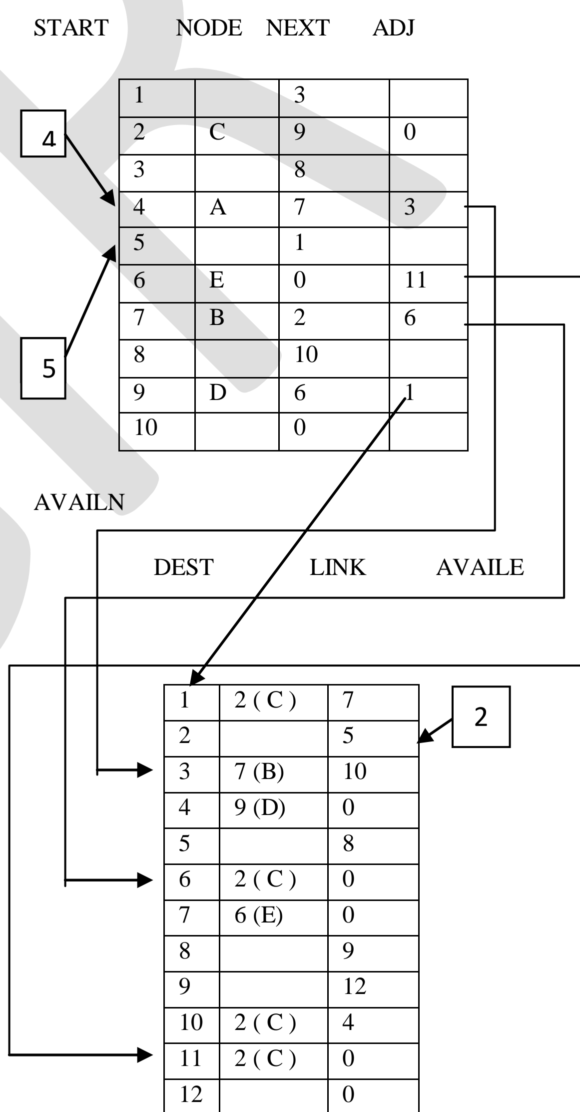
Here NODE will be the name or key value of the node, NEXT will be a pointer to the next node in the list NODE and ADJ will be a pointer to the first element in the adjacency list of the node, which is maintained in the list EDGE. The shaded area indicates that there may be other information in the record, such as the indegree INDEG of the node, the outdegree OUTDEG of the node, the STATUS of the node during the execution of an algorithm, and so on. (Alternatively, one may assume that NODE is an array of records containing fields such as NAME, INDEG, OUTDEG, STATUS…) The nodes themselves, as pictured in Fig. 2(a) and Fig. 2(b) will be organized as a linked list and hence will have a pointer variable START for the beginning of the list and a pointer variable AVAILN for the list of available space. Sometimes, depending on the application, the nodes may be organized as a sorted array or a binary search tree instead of a linked list.

(b) Edge list--- Each element in the list EDGE will correspond to an edge of G and will be a record of the form:

DEST	LINK	

The field DEST will point to the location in the list NODE of the destination or terminal node of the edge. The field LINK will link together the edges with the same initial node, that is, the nodes in the same adjacency list. The shaded area indicates that there may be other information in the record corresponding to the edge, such as a field EDGE containing the labeled data of the edge when G is a labeled graph, a field WEIGHT containing the weight of the edge when G is a weighted graph, and so on. We also need a pointer variable AVAILE for the list of available space in the list EDGE.

Fig. 3 shows how the graph G in Fig. 2(a) and Fig. 2(b) may appear in memory. The choice of 10 locations for the list NODE and 12 locations for the list EDGE is arbitrary.



	NODE	NEXT	ADJ
1		3	
2	C	9	0
3		8	
4	A	7	3
5		1	
6	E	0	11
7	B	2	6
8		10	
9	D	6	1
10		0	

	DEST	LINK
1	2 (C)	7
2		5
3	7 (B)	10
4	9 (D)	0
5		8
6	2 (C)	0
7	6 (E)	0
8		9
9		12
10	2 (C)	4
11	2 (C)	0
12		0

Fig. 3

The linked representation of a graph G that we have been discussing may be denoted by

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

The representation may also include an array WEIGHT when G is weighted or may include an array EDGE when G is a labeled graph.

Data Structures---

When used as a data structure, the main alternative for the adjacency matrix is the adjacency list. Because each entry in the adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only n2 / 8 bytes of contiguous space, where n is the number of vertices. Besides just avoiding wasted space, this compactness encourages locality of reference.

On the other hand, for a sparse graph, adjacency lists win out, because they do not use any space to represent edges which are not present. Using a naïve array implementation on a 32-bit computer, an adjacency list for an undirected graph requires about 8e bytes of storage, where e is the number of edges.

Noting that a simple graph can have at most n2 edges, allowing loops, we can let d = e / n2 denote the density of the graph. Then, 8e > n2 / 8, or the adjacency list representation occupies more space, precisely when d > 1 / 64. Thus a graph must be sparse indeed to justify an adjacency list representation.

Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list.

With an adjacency matrix, an entire row must instead be scanned, which takes O(n) time. Whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

Pros & cons of adjacency matrix-----

Pros----

Adjacency matrix is very convenient to work with. Add (remove) an edge can be done in O(1) time, the same time is required to check, if there is an edge between two vertices. Also it is very simple to program and in all our graph tutorials we are going to work with this kind of representation.

Cons---

1. Adjacency matrix consumes huge amount of memory for storing big graphs. All graphs can be divided into two categories, sparse and dense graphs. Sparse ones contain not much edges (number of edges is much less, that square of number of vertices, |E| << |V|2). On the other hand, dense graphs contain number of edges comparable with square of number of vertices. Adjacency matrix is optimal for dense graphs, but for sparse ones it is superfluous.

2. Next drawback of the adjacency matrix is that in many algorithms you need to know the edges, adjacent to the current vertex. To draw out such an information from the adjacency matrix you have to scan over the corresponding row, which results in O(|V|) complexity. For the algorithms like DFS or based on it, use of the adjacency matrix results in overall complexity of O(|V|2), while it can be reduced to O(|V| + |E|), when using adjacency list.

3. The last disadvantage, we want to draw you attention to, is that adjacency matrix requires huge efforts for adding/removing a vertex. In case, a graph is used for analysis only, it is not necessary, but if you want to construct fully dynamic structure, using of adjacency matrix make it quite slow for big graphs.

Pros & cons of adjacency list----

Pros----

Adjacent list allows us to store graph in more compact form, than adjacency matrix, but the difference decreasing as a graph becomes denser. Next advantage is that adjacent list allows to get the list of adjacent vertices in O(1) time, which is a big advantage for some algorithms.

Cons----

1. Adding/removing an edge to/from adjacent list is not as easy as for adjacency matrix. It requires, on the average, O(|E| / |V|) time, which may result in cubical complexity for dense graphs to add all edges.

2. Check, if there is an edge between two vertices can be done in O(|E| / |V|) when list of adjacent vertices is unordered or O(log2(|E| / |V|)) when it is sorted. This operation stays quite cheap.

3. Adjacent list doesn't allow us to make an efficient implementation, if dynamically change of vertices number is required. Adding new vertex can be done in O(V), but removal results in O(E) complexity.

Applications in computer science----

In computer science, an adjacency list is a data structure for representing graphs. In an adjacency list representation, we keep, for each vertex in the graph, a list of all other vertices which it has an edge to (that vertex's "adjacency list"). For instance, the representation suggested by van Rossum, in which a hash table is used to associate each vertex with an

array of adjacent vertices, can be seen as an example of this type of representation. Another example is the representation in Cormen et al. in which an array indexed by vertex numbers points to a singly-linked list of the neighbors of each vertex.

One difficulty with the adjacency list structure is that it has no obvious place to store data associated with the edges of a graph, such as the lengths or costs of the edges. To remedy this, some texts, such as that of Goodrich and Tamassia, advocate a more object oriented variant of the adjacency list structure, sometimes called an incidence list, which stores for each vertex a list of objects representing the edges incident to that vertex.

To complete the structure, each edge must point back to the two vertices forming its endpoints. The extra edge objects in this version of the adjacency list cause it to use more memory than the version in which adjacent vertices are listed directly, but these extra edges are also a convenient location to store additional information about each edge (e.g. their length).

Conclusion------

The main alternative to the adjacency list is the adjacency matrix. For a graph with a sparse adjacency matrix an adjacency list representation of the graph occupies less space, because it does not use any space to represent edges that are not present. On the other hand, because each entry in an adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only n2/8 bytes of contiguous space, where n is the number of vertices.

Besides the space trade-off, the different data structures also facilitate different operations. It is easy to find all vertices adjacent to a given vertex in an adjacency list representation. Adjacency lists use memory in proportion to the number edges, which might save a lot of memory if the adjacency matrix is sparse. It is fast to iterate over all edges, but finding the presence or absence specific edge is slightly slower than with the matrix.

To sum up, adjacency matrix is a good solution for dense graphs, which implies having constant number of vertices, but on the other hand, the adjacency list is a good solution for sparse graphs and lets us changing number of vertices more efficiently, than if using an adjacent matrix. But still there are better solutions to store fully dynamic graphs.

References

1. www.en.wikipedia.org/wiki/Adjacency_matrix
2. www.papers.ssrn.com/sol3/papers.cfm?abstract_id=1694711
3. www.datastructures.itgo.com/graphs/adjmat.htm
4. www.wapedia.mobi/en/Adjacency_matrix#2
5. www.encyclopedia.com/doc/1O11-adjacencymatrix.html
6. Data structures by schanum series.
7. Data structures by R.S Salaria