Table of Contents

[Graph Representations 1](#_Toc107595414)

[Adjacency Matrix 1](#_Toc107595415)

[Representation 1](#_Toc107595416)

[**Weighted vs. Unweighted Adjacency Matrix** 2](#_Toc107595417)

[**Directed vs. Undirected Adjacency Matrix** 2](#_Toc107595418)

[**Examples** 2](#_Toc107595419)

[**Vertex (Node) Data** 4](#_Toc107595420)

[Adjacency List 4](#_Toc107595421)

[**Directed vs. Undirected Adjacency List** 5](#_Toc107595422)

[**Weighted vs. Unweighted Adjacency List** 5](#_Toc107595423)

[**Examples** 5](#_Toc107595424)

[**Vertex (Node) Data** 6](#_Toc107595425)

[Which Representation is Better? 7](#_Toc107595426)

[**Performance of Common Operations** 7](#_Toc107595427)

[**Space Requirements** 8](#_Toc107595428)

[Trade-Offs 8](#_Toc107595429)

# **Graph Representations**

* We want to be able to represent a graph as a data structure in a computer.
* There are two common representations:

1. **Adjacency Matrix**

2. **Adjacency List**

* Notice the **adjacency** term. What does it mean for two vertices to be adjacent?
  + Two vertices are said to be ***adjacent*** if there is simply an ***edge*** between them.
  + That is, vertex ***j***is ***adjacent*** to ***i***if and only if (*i*, *j*) ∈ *E*.

# **Adjacency Matrix**

## **Representation**

* An **adjacency matrix** for a graph with *n* vertices numbered 0, 1, . . ., *n* – 1 is an *n* x *n* array **two-dimensional array** (matrix).
  + The matrix holds all possible edges in the set *E*.
* If a graph has ***V*** vertices, how many distinct edges can it have?
  + The first vertex ***V1*** can have an edge to every vertex (including itself): ***V*** edges
  + The second vertex ***V2*** can have an edge to every vertex (including itself): ***V*** edges
  + ... and so on for each of the ***Vi*** vertices
* Therefore, a graph that has ***V*** vertices can have a maximum ***V*** 2 edges (*E* = *V x V* = *V2*)
  + A graph with “all” *V2* edges is considered ***complete*** (the matrix is completely filled)
  + A graph with “close to” *V2* edges is considered ***dense***
  + A graph with “closer to” *V*edges is considered ***sparse***
* Example: A graph has ***5*** vertices, how many distinct edges can each node have?
  + Each node can have up to ***5*** edges
  + The graph can have up to ***25*** edges

* An adjacency matrix is an appropriate representation if the graph is **complete** or **dense**:
  + |*E*| = O(V2).

### **Weighted vs. Unweighted Adjacency Matrix**

* In an **unweighted graph**, we can let matrix[i][j] be the **number of edges** joining vertex ***i*** and ***j***. (usually is 1 or 0, but there can be more than one edge between any two nodes)
  + If there is single edge between vertices *i* and *j*, the value will be 1.
  + If there is no edge between vertices *i* and *j*, the value will be 0.
* In a **weighted graph**, we can let matrix[i][j] be the **weight** that is between vertices ***i*** and ***j***.
  + If there is an edge between vertices *i* and *j*, the value will be the ***weight*** value that labels the edge.
  + If there is no edge between vertices *i* and *j*, the value stored will be ***infinity*** (or a very large number, in practice).
* The diagonal entry matrix[i][i] corresponds to the number of loops (self-connecting edges) at vertex ***i*** (often disallowed).

A picture containing text

Description automatically generated

### **Directed vs. Undirected Adjacency Matrix**

* Remember that in an **undirected graph**, each edge can be traversed in either direction
  + An undirected graph with edge (*v*, *w*) also has edge (*w*, *v*)
  + *v* 🡪 *w* and *w* 🡪 *v*
* An adjacency matrix for an **undirected graph is symmetrical**.
* This means that every index in the matrix, matrix[i][j] has a symmetrical pair at matrix[j][i].
  + That is, matrix[i][j] equals matrix[j][i].
* In a **directed graph**, we simply list all of the edges between nodes, without the symmetry. The edge value between two nodes will depend on whether the graph is weighted vs. unweighted.

### **Examples**

* The following figure is an example of a ***weighted*** ***undirected*** graph and adjacency matrix.
  + For nodes that don’t have an edge, their values are infinity.
  + Notice the symmetry in the graph

A screenshot of a computer

Description automatically generated with low confidence

* The following figures are examples of ***unweighted*** ***undirected*** graph and adjacency matrix.
  + For vertices that don’t have an edge between them, the weight is 0

**Example 1**

Shape

Description automatically generatedA picture containing text, electronics

Description automatically generated

**Example 2**

A picture containing text, antenna

Description automatically generatedCalendar

Description automatically generated

* The following figure is an example of a ***weighted* *directed*** graph and adjacency matrix

Diagram

Description automatically generatedTable

Description automatically generated

* The following figure is an example of an ***unweighted* *directed*** graph and adjacency matrix.
  + 0 represents false, and 1 represents true.
  + Notice that the diagonal entries matrix[i][i] are always false (0). If a node were to point to itself, the associated diagonal entry would be true (1).

A picture containing chart

Description automatically generated

### **Vertex (Node) Data**

* Our definition of an adjacency matrix does not mention the value, if any, **in a *vertex*** (we have so far been talking about edge values, not node values).
  + For example, what if node A holds some value?
* If you need to associate values with vertices, you can use a separate second array named ***values***to represent the ***n* vertex values**.
* The array is one-dimensional, and values[i] represents the value in vertex *i*.

# **Adjacency List**

* An **adjacency list** for a graph with ***n***vertices numbered 0, 1, . . ., *n* – 1 is a one-dimensional array of size ***n***.
* For each vertex, we keep a ***linked list*** of all adjacent vertices.
  + ***n*** vertices
  + array of ***n*** linked lists, one per vertex
  + each list contains a list of all vertices adjacent to the head of the list

### **Directed vs. Undirected Adjacency List**

* The adjacency list for an **undirected graph** treats each edge as if it were ***two directed edges* *in opposite directions***.
  + If vertex A has an undirected edge with vertex B, then vertex A will have a node in its linked list pointing to B, and vertex B will have a node pointing to vertex A.
* The adjacency list for a **directed graph** simply has one pointer to each adjacent vertex *j* that has an edge with vertex *i*.

### **Weighted vs. Unweighted Adjacency List**

* The adjacency list for a **weighted** graph stores the ***identity*** and ***edge*** values all adjacent vertices with node *i*.
  + If the edge is a dynamically allocated node, we can simply have a pointer pointing to adjacent vertex *j*.
  + If the edge is not yet dynamically allocated, we need to create a new node containing the identity of the adjacent vertex, as well as the edge’s value.
* The adjacency list for an **unweighted** graph does not need to store any extra information other than the ***identity*** of the adjacent vertex.

### **Examples**

* The following figure shows a ***weighted* *undirected*** graph and its adjacency list.
  + The adjacency list for an undirected graph treats each edge as if it were two directed edges in opposite directions.
  + Thus, the edge between *A* and *B* in Figure 20-9a appears as edges from *A* to *B* and from *B* to *A* in Figure 20-9b.
  + The graph in 20-9a happens to be weighted; you can include the edge weights in the nodes of the adjacency list, as shown in Figure 20-9b.

Diagram

Description automatically generated with medium confidence

* The following figure shows an ***unweighted undirected*** graph and its adjacency list.

Chart, diagram

Description automatically generated

* The following figure shows an ***unweighted* *directed*** graph and its adjacency list.
  + You can see, that vertex 0 (*P*) has edges that point to vertex 2 (*R*) and vertex 5 (*W*).
  + Thus, the first linked chain in the adjacency chain contains nodes for *R* and *W*.
  + Each edge is unweighted, so there is no additional information that needs to stored other than a pointer to

Diagram

Description automatically generated

### **Vertex (Node) Data**

* If an adjacent vertex has no value, the node only needs to store some indication of the vertex’s identity.

## **Which Representation is Better?**

* Which of these two implementations of a graph is better?
* The answer depends on how your particular application uses the graph.

### **Performance of Common Operations**

* Two of the most commonly performed graph operations are:

1. **Determine whether there is an edge from vertex *i* to vertex *j***

***Adjacency Matrix*:**

*Advantage:*

Supports direct access in O(1) time.

To determine whether there is an edge from *I* to *j* by using an adjacency matrix, you need only index the value of matrix[i][j].

*Disadvantage:*

Can take up more and unnecessary space for sparse graphs.

***Adjacency List*:**

*Advantage:*

No need to extra space.

*Disadvantage:*

You must traverse the *i*th linked chain to determine whether a vertex corresponding to vertex *j* is present. This can take up to O(N) time.

1. **Find all vertices adjacent to a given vertex *i***

***Adjacency Matrix:***

If you use an *adjacency matrix*, you must either traverse the *i*th row of the array (where there may be empty indexes).

***Adjacency List:***

An *adjacency list* supports this operation more efficiently. You need only traverse the *i*th linked chain to find all vertices adjacent to a given vertex *i*.

For a graph with *n* vertices, the *i*th row of the adjacency matrix always has *n* entries, whereas the *i*th linked chain has only as many nodes as there are vertices adjacent to vertex *i*, a number typically far less than *n*.

### **Space Requirements**

**Adjacency Matrix**

* An **adjacency matrix** is an appropriate representation if the graph is **dense**: |*E*| = O(|*V*|2).
* The more complete a graph is, the denser it is (more edges).

**Adjacency List**

* If the graph is not dense, in other words, if the graph is **sparse**, a better solution is an **adjacency list** representation.
* For each vertex, we keep a list of all adjacent vertices.
* The space requirement is then *O*(|*E*| + |*V*|), which is linear in the size of the graph.
* The number of nodes in an adjacency list always equals the number of edges in a directed graph or twice that number for an undirected graph.
* Even though the adjacency list also has n head pointers, it often requires less storage than an adjacency matrix.

## **Trade-Offs**

* The **adjacency matrix**
  + allows direct access
  + can take more space (and time) for sparse graphs.
* The **adjacency-list**
  + is easier to manage when vertices are added to or removed from the graph
  + examining/traversing all edges starting at a vertex when we “visit” the vertex

**Which to Choose?**

When choosing a graph implementation for a particular application, you must consider such factors as **what operations you will perform most frequently** on the graph and the number of edges that the graph is likely to contain.

For example, if you are determining the sequence of flights from an origin city to a destination city. The flight map for this problem is a directed graph.

The most frequent operation would be to find all cities (vertices) adjacent to a given city (vertex). Therefore, the adjacency list would be the more efficient implementation of the flight map. The adjacency list also requires less storage than the adjacency matrix, which you can demonstrate as an exercise.

## **Runtime Comparison**

Table

Description automatically generated