Advanced Algorithmics and Graph Theory with Python

IMTx: NET04x

Graphs and Paths

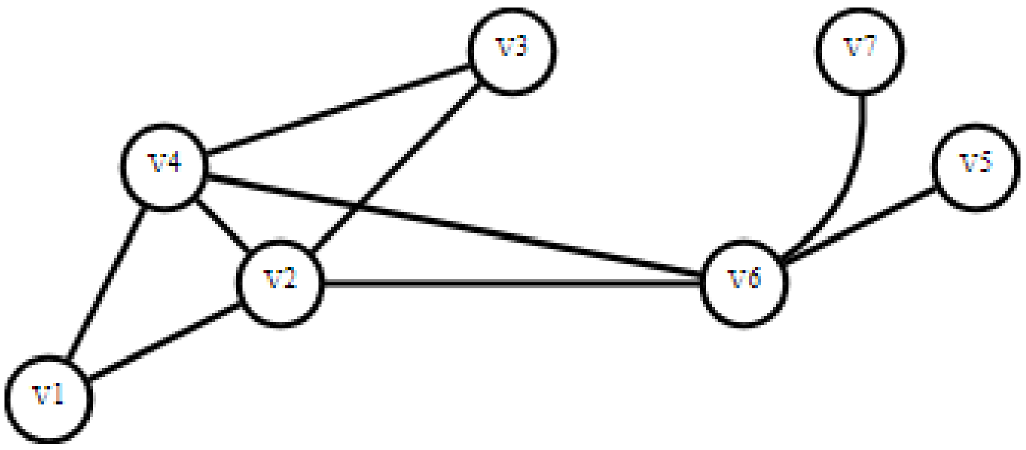
Basics Definitions

Graph theory is probably one of the most common sub-fields of discrete mathematics. Graphs are mathematical structures that are used to represent the relationships between objects.

Mathematically, a graph is composed of a finite set of vertices V, and a set of edges E. We can therefore refer to a graph as a couple G=(V,E) composed of V and E. A graph is often depicted using circles for vertices and lines for edges.

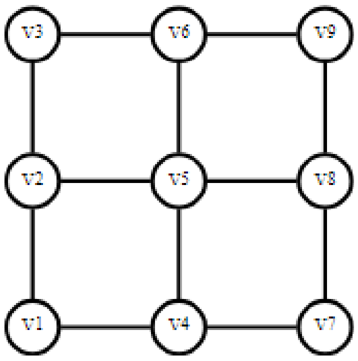
SOME EXAMPLES: GRAPHS CAN BE USED TO REPRESENT RELATIONSHIPS BETWEEN FAMILY MEMBERS IN A FAMILY TREE, IN WHICH CASE MEMBERS WOULD BE REPRESENTED AS VERTICES AND THEIR RELATIONSHIPS AS EDGES, OR, SIMILARLY, TO REPRESENT MOLECULE INTERACTIONS IN A BIOLOGICAL ORGANISM, NEURAL NETWORKS IN THE BRAIN, HYPERLINKS BETWEEN WEBSITES, OR, SIMPLY, A MAZE, IN WHICH CELLS CAN BE REPRESENTED AS VERTICES AND ADJACENT CELLS NOT SEPARATED BY A WALL CAN BE LINKED WITH EDGES.

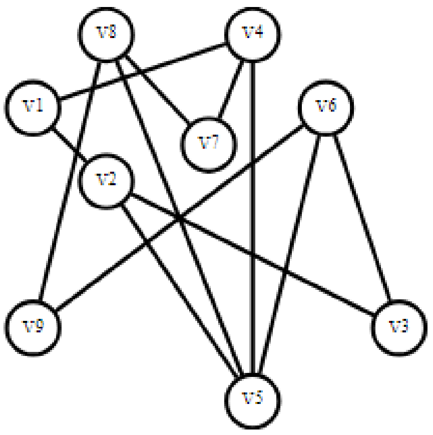
There are multiple ways to define graphs. In the simplest case, edges convey information about which pairs of vertices are connected and which are not. This means that E becomes a set of pairs of vertices. A pair {u,v} is an unordered set that contains two distinct elements, u and v. Considering that they're unordered, the pairs {u,v} and {v,u} are identical.
Here is an example of a graph:



ere, the set of vertices is V={v1,v2,…,v7} and the set of edges is E ={{v1,v2},{v1,v4},{v2,v3},{v2,v4},{v2,v6},{v3,v4},{v4,v6},{v5,v6},{v6,v7}}.

The order of a graph is the number of vertices it contains, which is 7 here in the example. The size of the graph is the number of edges, which is 9 in this example.
As you can see, this is a simple, effective way to visualize a graph. But it might also be misleading, because the creation of a figure like this requires you to make arbitrary decisions about the coordinates of vertices and the forms of lines.This can be seen in the following example figure, which shows two graphs that appear to be different. However, these two graphs are identical in the sense that they have exactly the same set of vertice and the same set of edges.





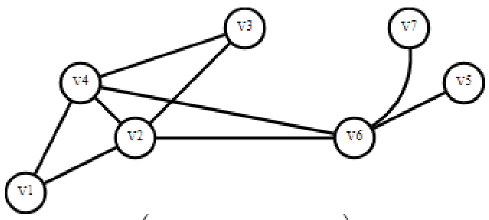
Graphs were first introduced by Euler in 1736. He was interested in formally proving that no one could walk around the city of Konigsberg and cross each of its bridges exactly one time. To prove this, he showed that starting and finishing at one point would require the graph to contain 0 or 2 land masses with an odd number of bridges. Since Konigsberg land masses all contained an odd number of bridges, this route was impossible.

In some cases, edges can be directed, which means that a vertex u can be connected to a vertex v, even if v is not connected to u. In such cases, edges are made up of *couples* of vertices, such as, for example, (u,v). Couples are more informative than pairs because the order of vertices in the couple is meaningful. For example, the couple (u,v) is distinct from the couple (v,u) if u and v are distinct themselves, whereas the pair {u,v} and the pair {v,u} are identical. When using couples, graphs are known as *digraphs* (which stands for directed graphs).

We will only consider graphs whose edges are pairs in this course, but it's worth pointing out that the algorithms presented here *do* also apply to digraphs.

The adjacency matrix and weighted graphs

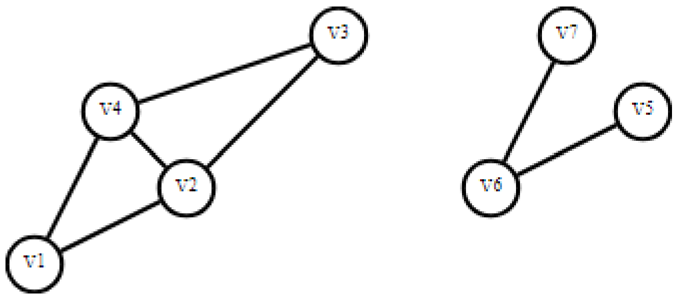
As in the previous examples, it's preferable to index vertices from 1 to n, which is the order of the graph. When vertices of the graph are indexed appropriately, an alternative way to represent a graph is to use a matrix.
The *adjacency matrix* of a graph is a matrix with as many rows and columns as the order of the graph.
The adjacency matrix is built by putting a 1 at line i and column j if {i,j} is an edge, and putting a 0 otherwise.
Here's an example of a graph and its corresponding adjacency matrix:



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

Paths and geodesic distances

A path is an ordered sequence of edges that are distinct from one other, and is obtained from a sequence of vertices by joining any two consecutive vertices in the corresponding edge. The two extreme vertices of the sequence are called the extremities of the path.
Take a look at this graph:



In this example, {v1,v2},{v2,v6},{v4,v6},{v3,v4} is a path, and we've obtained it from the vertex sequence v1,v2,v6,v4,v3.

Paths are often confused with *walks*. A walk is a sequence of vertices, such that any two consecutive vertices form an edge in the graph. The difference is that, in a path, an edge cannot appear twice. For example, v1,v2,v4,v2 in the previous graph is a walk, but the corresponding sequence {v1,v2},{v2,v4},{v2,v4} is not a path, because of the repetition of the edge {v2,v4}.

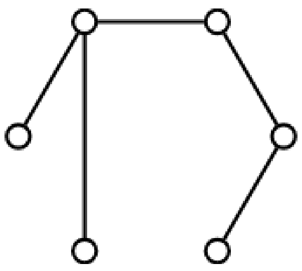
A *cycle* in a graph is a path in which the extremities are identical. For example, in the previous graph {v2,v6},{v4,v6},{v3,v4},{v2,v3} is a cycle obtained from the sequence of vertices v2,v6,v4,v3,v2.

The *length* of a path is the length of the sequence of edges. For weighted graphs, the length corresponds to the sum of the weights. If there exists at least one path linking two vertices u and v, there exists a shortest one. Most of the following lessons will be spent learning how to find the shortest paths between two vertices in a graph.

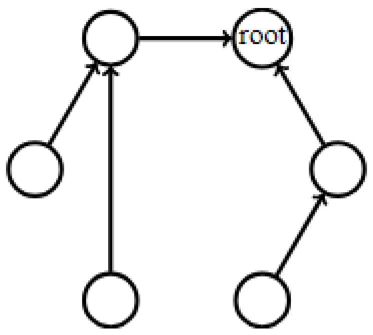
Finally, a graph is said to be connected if, for any two vertices there exists a path having these vertices as extremities. All the example graphs we've considered so far are connected. Here's one that's *not* connected (for example, observe that there is no path for which v3 and v7 are extremities):

Standard graphs

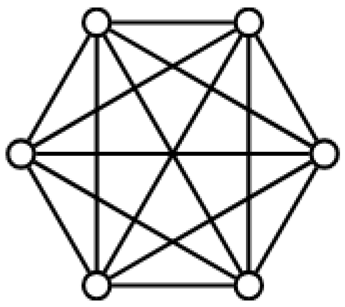
Some graphs are very useful because they can appear in many situations. For example, we encounter trees all the time: in file systems, in genealogical graphs, in sport competitions where tournaments are often depicted as trees and so on.
Trees are connected graphs that are cycle-free.



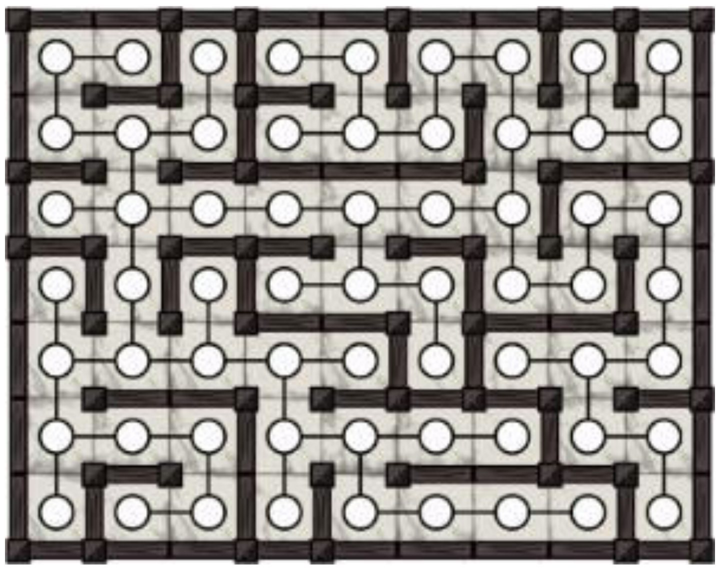
Note that trees are often confused with *rooted* trees, in which connected vertices have a specific relationship with one another. Rooted trees can be defined by choosing one arbitrary vertex to be the root of the tree. In the following example, we have depicted a rooted tree as a digraph, where edges points towards the root of the tree:



Another example of a standard graph is the complete graph. A complete graph includes all possible edges, and is often a good choice to test the abilities or the computation time of an algorithm that operates on graphs.



Mazes can also be represented as graphs. In this case, vertices can be used to represent the cells of the maze, and edges can be defined as neighboring cells that are not separated by a wall.



Representing Graphs

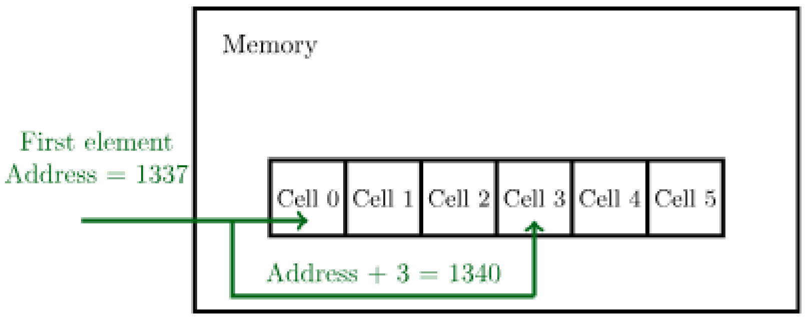
Representing data in computer memory

As I'm sure you'll recall, computer memory is organized into locations in which data are stored. Accessing data therefore requires us to know at which location, or, address, they've been stored.
This organization is convenient when you know where to look, but it can be tricky if you don't know the exact address of what you're looking for.
In the same way that it's easier to find a house by its address rather than by its description, or a library book by its call number, it's easier to find data when you know the specific address.

Basic data structures to represent graphs

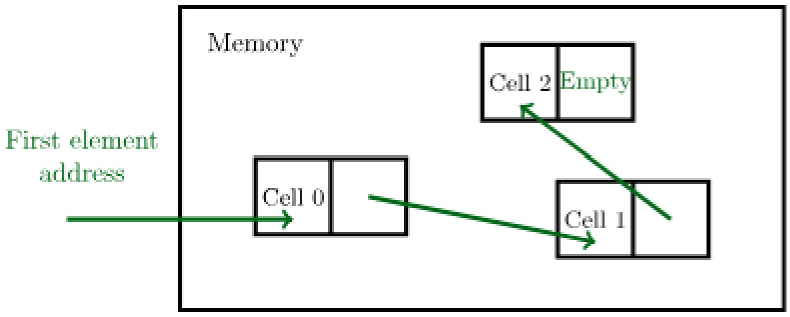
Let's imagine that we want to store the adjacency matrix of a graph.

- Using Arrays
  One way to store data in memory is to use an array. Arrays are adjacent data structures that are used to represent sequences of data, where each piece of data uses the same size in memory.



  So if you know the address of the first piece of data, the address of any other piece of data can easily be computed. For example, if the first element address is 1337, and if each piece of data is stored using one address, then the fourth element address is 1340. So, this means that you can rapidly access the n-th piece of data of an array. On the other hand, accessing the n-th non-zero element can be time-consuming, because all cells need to be checked one by one to see whether or not they contain a zero. In terms of graph adjacency matrices, arrays are effecient to check if two vertices u and v are connected by an edge. On the other hand they are not that efficient if you want to retrieve all the neighbors of u, because you need to test each vertex v one by one.

- Using Lists
  Another common data structure is a list. Lists are not adjacent in memory, and to find a piece of data, you need to find all the previous ones first. The principle of lists is that each address not only contains data, but also the address of the next piece of data to look for. So if you want to access the third piece of data in a list, you first must access the first, look at the address of the second, then look at the second to find the address of the third.



Accessing the n-th piece of data in a list can be time-consuming. However, it IS possible to overcome the time-consuming process of accessing the n-th piece of data by only storing pieces of data of interest, which considerably reduces memory usage compared to an array. In terms of graphs, lists can be used to store only information about the neighbors of vertices for example, since nonneighbors are often irrelevant. If each vertex has only a few neighbors, representing only neighbors with lists can significantly save memory. On the other hand, checking if u and v are connected by an edge may require you to search the full list of u's neighbors.

- Using Dictionnaries
  A final example of data structures is dictionaries. Dictionaries are elaborate structures that aim to combine the advantages of both arrays and lists, in particular, fast access and efficient memory usage, respectively. Dictionaries make use of hash functions, which are basically mechanisms to transform contents in addresses. This is the optimal choice for prototyping, because it allows programmers to benefit from both speed of access and low memory usage. As a rule of thumb, dictionaries should always be used if you don't know exactly what you're doing.