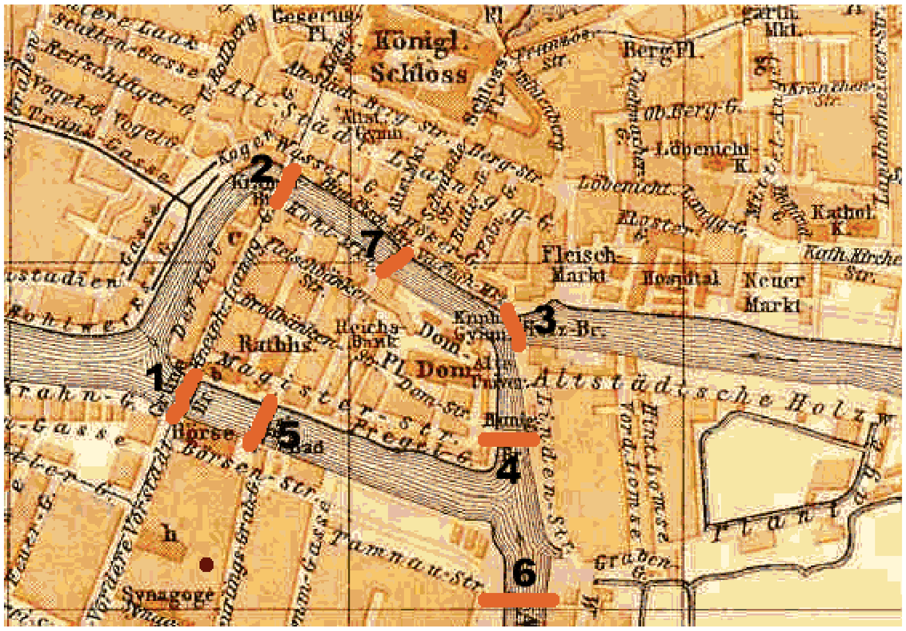
A Graph Toolkit
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/math1800_2023/graph_tools/graph_tools.pdf



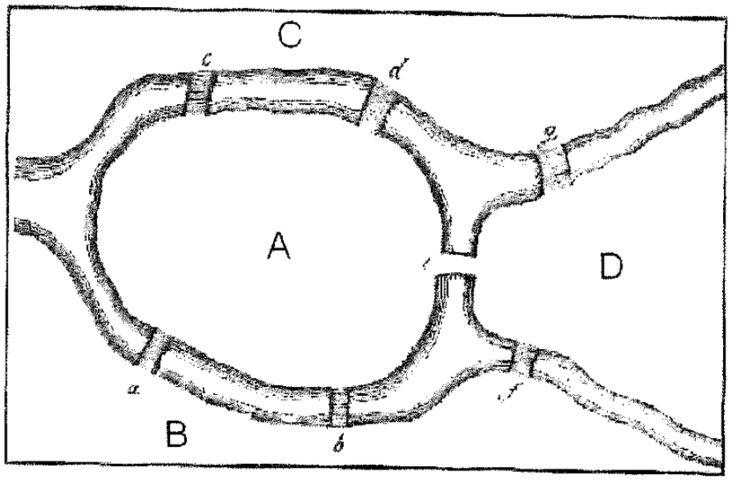
The Seven Bridges of Koenigsberg Problem.

"Graph Theory"

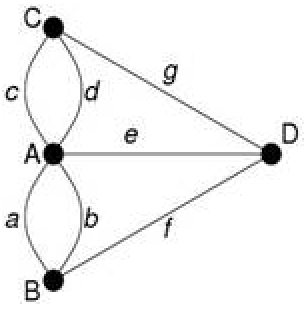
- *Leonhard Euler realized that the solution to a puzzle depended only on the pattern of connections between four areas of Koenigsberg;*
- *These patterns could be symbolized by points and lines, also called nodes and edges.*
- *This simple representation of mathematical graphs created a new language in which many new problems could be described and solved.*
- *Variations on the basic simple graph structure allow for one-way connection, multiple connections, "selfie" connections, and weighted connections.*
- *Graph theory problems include determining connectedness, finding the shortest path, exiting a maze, the traveling salesperson problem, the page rank algorithm for optimal search;*

1 Graph Theory is about connections

The rivers that joined at the town of Koenigsberg divided it into four areas, connected by seven bridges. It was a common challenge to try to start from a given point and return after crossing each bridge just once. No one had done it, but no one knew if it could be done. Leonhard Euler looked at the problem, saw a way to analyze it, published a paper, and invented the field of graph theory.



Euler realized that the important features of this problem were the bridges, and the areas that connected them, while everything else was a distraction. He represented the areas as dots, labeled with a capital letter, and the bridges as lines, labeled with lower case letters.



One huge advantage of this approach is that we can even replace this diagram by a table, called the *adjacency matrix*, which records how many bridges there are between any two areas. Although Euler did not use this matrix, this kind of numerical representation means we can easily transfer graph theory problems to a computational setting.

	A	B	C	D
A	0	2	2	1
B	2	0	0	1
C	2	0	0	1
D	1	1	1	0

After searching by hand for solutions, Euler changed his approach and reasoned as follows: Suppose I found a solution to this problem. I could use a red pen to retrace my path on the map. If I start from node A, then A will now have one red bridge extending from it. If I move from there to node C, I must enter and then leave, so two bridges connected to C will become red; as I visit each node, I must enter and leave, so the number of red bridges at each node will always be even. And when I finally return to node A, I add one more red bridge there. So if a round trip is possible, then all nodes must have been entered and exited an even number of times. That means, in the original diagram, every node must be an endpoint of an even number of bridges.

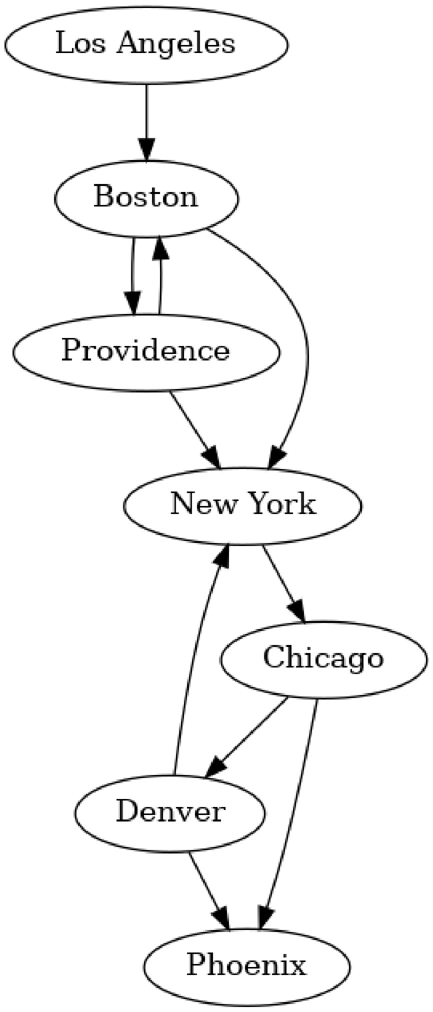
In honor of Euler, a round trip that uses every edge exactly once is known as an *Euler circuit.* We conclude that there is no Euler circuit for the Koenigsberg bridges.

Then we might ask, is it at least possible to use all the bridges once if we are allowed to stop at a different node from our starting node? Such a process is known as an *Euler path.* If an Euler path exists for a graph, you should see that there must be exactly two nodes that have an odd number of bridges, and your trip must start at one of those nodes and end at the other. But all four Koenigsberg areas have an odd number of bridges, so there is no Euler path either.

As usual, solving one mathematical problem created another one. Using Euler's insight, he could show that in some cases, a solution was impossible. But he had not have a general test that would say for any graph whether an Euler circuit or Euler path existed, and if so, how to compute it.

2 Our example digraph

Our discussion today will include an example directed graph. We will try to represent it, list the nodes and vertices, and answer a few simple questions about it. But we will do this by creating some simple tools that can also be used for similar problems.



3 The adjacency representation

There are a number of ways of representing a graph or digraph. The one that might seem most natural uses the *adjacency matrix*. If we have n nodes, then we define an $n \times n$ matrix A such that $A[i,j]$ is 1 if there is a direct link from node i to node j.

For our example digraph, A would be:

```
A = np.array ( [
   [0,1,1,0,0,0,0],
   [1,0,1,0,0,0,0],
   [0,0,0,1,0,0,0],
   [0,0,0,0,1,1,0],
   [0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0],
   [1,0,0,0,0,0,0] ] )
```

4 The dictionary representation

Another representation uses an adjacency list, or dictionary, which we might call G. Each node is associated with a list of all the directly linked nodes. In a Python dictionary, we could use the numeric index or the city name when we set up the dictionary:

For our example graph, the dictionary would be:

```
G = {
   'Boston'      : [ 'Providence', 'New York' ],
   'Providence'  : [ 'Boston', 'New York' ],
   'New York'    : [ 'Chicago' ],
   'Chicago'     : [ 'Denver', 'Phoenix' ],
   'Denver'      : [ 'Phoenix', 'New York' ],
   'Phoenix'     : [],
   'Los Angeles' : [ 'Boston' ] }
```

5 Working with a dictionary

Before we can be comfortable with this dictionary representation it is useful to recall the Python commands we need in order to work with the data we will encounter.

First, recall that a `dictionary` is a collection of *keys* and *values*, having the basic format

```
G = { key0 : value0, key1 : value1, ..., keyk, valuek }
```

In our case, each value item is a list, delimited by square brackets.

To add `key` to a dictionary `G`, with no value

```
G[key] = []
```

To add the pair `key : value` to a dictionary `G`, if `key` is already a key, and we just want to add `value` to the list of values associated:

```
if key in G:
  G[key].append ( value )
```

but if `key` is not in `G`, we need to say

```
  else:
    G[key] = [ value ]
```

To consider all the keys in the dictionary, one at a time

```
  for n in G,keys():
     ...actions involving n
```

To consider all the values for a key, for all the keys in the dictionary, one at a time

```
  for n in G.keys():
    for n2 in G[n]:
      ...actions involving n and n2
```

To determine whether the value of a key `n` is empty:

```
  if ( not G[n] ):
      ...actions because n has no values
```

To make a list of all the keys:

```
  list ( G.keys() )
```

6 Working with a list

The values in the dictionary are lists. For each key, the value is a list of the nodes that are directly reachable from the key node. If no nodes are reachable, we write '[]', if one, we write ['Boston'], and if several, we provide them in sequence, separated by commas.

A list is ordered. The first item in a list called *nodes* would be `node[0]` and the last is by convention `node[-1]]`.

To iterate through all the items in a list:

```
  for ( n in node ):
     ...actions involving n
```

To check whether item `n` is in the list, we write

```
  if ( n in node ):
     ...action...
```

To concatentate two lists

```
  newlist = list1 + list2
```

To add one more item to a list, we use the `append()` function. To add the node `n` to the end of the `node` list:

```
  node.append(n)
```

or

```
  node = node + [ n ]
```

7 List the nodes and edges

To verify that `G` has been properly set up, we can print out the nodes and edges. Let's make a function to collect the nodes:

```
def digraph_nodes_all ( G ):
  nodes = list ( G.keys ( ) )
  return nodes
```

The edges are a little trickier:

```
def digraph_edges_all ( G ):
  edges = []
  for node in G.keys ( ):
    for dest in G[node]:
      edges.append ( [ node, dest ] )
  return edges
```

Now we can check our data by

```
  nodes = digraph_nodes_all ( G )
  print ( nodes )
  edges = digraph_edges_all ( G )
  print ( edges )
```

8 Plot the digraph

We know that *graphviz* can be used to plot graphs and digraphs, but it requires issuing a `node()` command for every node, and an `edge()` command for every edge. Luckily, we can automate this:

```
def digraph_plot ( G ):
  from graphviz import Digraph
  edges = digraph_edges_all ( G )
  nodes = digraph_nodes_all ( G )
  dot = Digraph ( format = 'png' )
  for n in nodes:
    dot.node ( n )
  for e in edges:
    dot.edge ( e[0], e[1] )
  dot.render ( 'digraph.dot', view = True )
  return
```

9 Find isolated nodes

A peculiar feature of our graph is that one node is isolated, that is, there is no path from it to another node. We might be interested in being alerted about all such nodes. To do so, we simply have to open an empty list, run through the dictionary, and collect every key with no value.

```
def digraph_find_isolated_nodes ( G ):
  isolated = list()
  for node in digraph:
    if ( not G[node] ):
      isolated.append ( node )
  return isolated
```

10 Add a node

Adding a node to the graph is simple. This is simply a new key to be added to the dictionary, initialized with an empty value list:

```
def digraph_add_node ( G, n ):
  if ( n not in G ):
    G[n] = []
  else:
    print ( 'Node', n, 'is already in the digraph.' )
  return G
```

11 Add an edge

Adding an edge is a little more complicated. Our input `e` should be a list of the form `[n0,n1]`, where, as far as the dictionary is concerned, `n0` is going to be the "key" and `n1` the value. How we do this depends on whether the key is already in the dictionary, in which case we need to append `n1` to whatever value is already there. But if `n0` was not already in the dictionary, we simply initialize this dictionary entry:

```
def digraph_add_edge ( G, e ):
  n0 = e[0]
  n1 = e[1]
  if ( n0 in G ):
    G[n0].append ( n1 )
  else:
    G[n0] = [ n1 ]
  return G
```

12 The reach of a node

Given any particular node `n`, we think of its edges as connections to other nodes. If we start at `n`, then the dictionary tells us the immediate neighbor nodes. So, if we imagined walking along the graph, we can certainly "reach" node `n` because we start there, and we can reach all the immediate neighbors of `n`. But these neighbors might in turn have neighbors that we haven't visited, and there might be more neighbors beyond that. The "reach" of node `n` is simply the set of nodes that can be reached by starting at node `nn` and traversing a connected sequence of edges.

The code to do this is actually reasonably simple, except that we have to be able to stop searching if our current set of neighbors that we are examining doesn't connect to any new unvisited neighbors. This is not an efficient implementation, but it gets the right answer.

```
def digraph_reach_node ( G, node ):
#
## digraph_reach_node() returns the nodes reachable from a given node.
#
  reach = [ node ]
  more = True
  while ( more ):
    more = False
    for n in reach:
      for dest in G[n]:
        if ( not dest in reach ):
          reach = reach + [ dest ]
          more = True
  return reach
```

13 A path from one node to another

If we want to plan a trip from node `a` to node `b`, we can use a recursive approach. From our starting node, we consider all one-hop trips; but each one hop trip is allowed to grow to a two hop trip, and so on. We are constantly checking to see whether one of these tentative trips has reached our goal, and if so, we roll the recursion back up and return with a successful plan.

Again, this is not the most efficient approach, but is enough to get an idea of how these calculations are done.

```
def digraph_path_find ( G, this_node, end_node, path = None ):
  if ( path == None ):
    path = []
  path = path + [ this_node ]
  if ( this_node == end_node ):
    return path
  if ( this_node not in G ):
    return None
  for node in G[this_node]:
    if ( node not in path ):
      extended_path = digraph_path_find ( G, node, end_node, path )
      if ( extended_path ):
        return extended_path
  return None
```