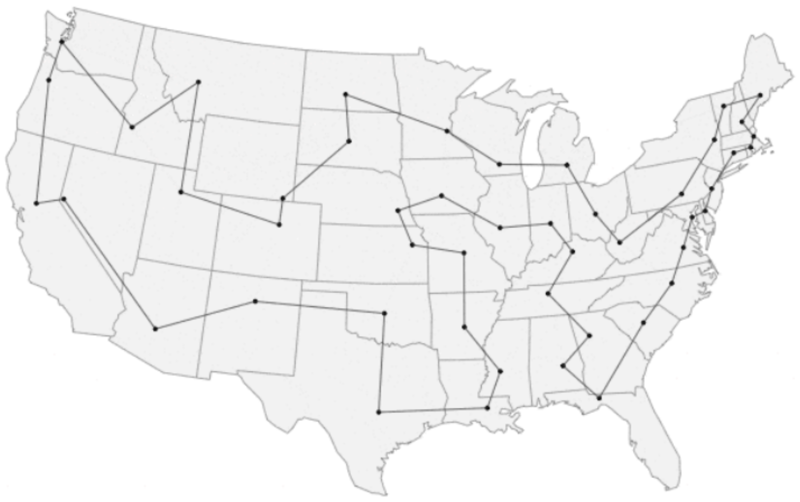
Graph Theory
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/mpp_2023/graph_theory/graph_theory.pdf



Seeking the shortest round trip connecting state capitals (lower 48 only).

Graph Theory

- *Mathematical graphs capture simple geometric notions of connection.*
- *The adjacency matrix or a Python dictionary can represent a graph computationally.*
- *We can compute the distance (in edges) between any two nodes of a graph.*
- *Breadth-first and depth-first searches are used to "read" the information in a graph.*
- *A weighted graph assigns a length to each edge.*
- *We can compute the shortest distance between two nodes in a weighted graph.*
- *The traveling salesperson problem is easily posed on a weighted graph, but very difficult to solve;*

1 The Bare Bones of Geometry

Mathematics is about numbers, geometry about (idealized) physical objects. You don't think of a triangle appearing in an equation, or an exponential function applied to a circle. However, the field of graph theory shows that the two subjects can be combined, creating an abstract world in which shapes and numbers both play a part.

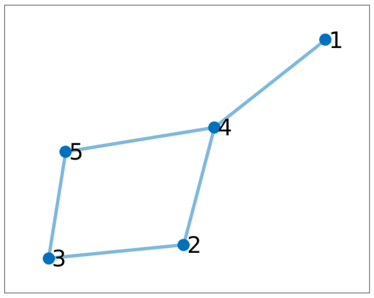
In contrast to geometry, graph theory takes as its subject a collection of finitely many *nodes*, also called *vertices*. Between any pair of nodes, there might be a connection, called an *edge*, or sometimes a *link*. The connection created by an edge normally goes both ways. If we wish, we can specify that a connection is a `directed edge`, like a one-way street. If we think of the nodes as cities, and the edges as roads, we may wish to include information about the mileage of each intercity road. In that case, we say we are using *weighted edges*.

This simple framework is enough for us to model a number of interesting real world problems involving this simplified geometry, to propose algorithms for solving them, and computer programs for efficiently finding solutions.

2 The Adjacency Matrix

A graph represents a simple discrete geometry, defined by a set of n points or nodes, some of which are connected by edges. For convenience, the nodes may be labeled numerically, 0 through $n-1$, or with letters or other titles.

Here is a typical graph:



A graph, with 5 nodes and 5 edges.

In order to represent a graph computationally, we need a way to describe the connection structure. One way that is suitable for small graphs is called the *adjacency matrix*. For a graph of n nodes, we construct an $n \times n$ matrix A such that $A_{i,j} = 1$ if nodes i and j are connected. Obviously, this matrix must be symmetric. Our sample graph would have the following adjacency matrix:

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

An alternative way to describe the adjacency matrix is that $A(i,j)$ is the number of paths from node i to node j using a single edge. This may sound awfully pointless, but let's compute the matrix $A^2 = A * A$:

$$A^2 = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 2 & 0 & 0 & 2 \\ 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & 2 & 3 & 0 \\ 1 & 2 & 0 & 0 & 2 \end{pmatrix}$$

If you look at the diagram of the graph, you should convince yourself that the fact that there are indeed exactly 2 distinct paths, involving two consecutive edges, to get from the node labeled 3 to the node labeled 4. Can you see what it means to say there are 3 distinct paths of length 2 to get from node 4 back to node 4?

We say that a graph is *connected* if for any starting node i there is always at least one path, of some length, to any final node j. If the graph has n nodes, then this path must be of length $n-1$ or less. Given the adjacency matrix A for a graph, consider, for $0 \leq \ell < n$, the matrices $P(\ell)$ defined by:

$$P(\ell) = \sum_{k=0}^{k \leq \ell} A^k$$

$P(\ell)$ records the number of paths of length ℓ or less, between any pair of nodes. For our example graph, we have

$$P(4) = A^0 + A^1 + A^2 + A^3 + A^4 = \begin{pmatrix} 5 & 6 & 12 & 17 & 6 \\ 6 & 12 & 23 & 29 & 11 \\ 12 & 23 & 11 & 12 & 23 \\ 17 & 29 & 12 & 17 & 29 \\ 6 & 11 & 23 & 29 & 12 \end{pmatrix}$$

For a given graph of n nodes, the path matrix $P(n-1)$ indicates the number of paths (of any length 0 through $n-1$) available to reach any node i from any node j. For our example graph, why does $P(4)$ by itself tell you that the graph is connected?

3 Edge Distance

From a given node i in a graph, we may ask how far away another node j is; that is, the length of the shortest path from i to j, if any. We may symbolize this edge distance as $d(j)$. Of course, $d(i)$ is 0, and if there is no path from i to j, we may set $d(j) = \infty$.

If we see a plot of a small graph, we can easily work out the values of d from a given starting point. But how could this be done computationally, even for a rather large and unseen graph?

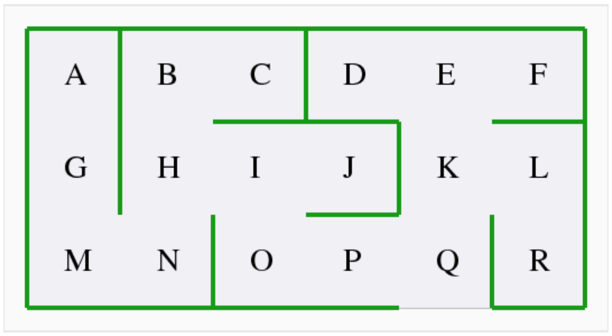
A simple algorithm is as follows:

```
The graph G has n nodes
Initialize all n values of d to oo.
Set d to zero for the starting node.
found = 1
while ( 0 < found )
  found = 0
  for every node i
    if d(i) < oo
      for every node j
        if A(i,j) == 1
          if ( d(j) == oo )
            d(j) = d(i) + 1
            found = found + 1
```

Using the adjacency matrix `A[,]` and a numpy array `d[]`, it is not too hard to implement this algorithm in Python. The exercises will ask you to try to do this, and apply it to the next example.

4 Edge Distance for a Museum Map

Consider the following map of a museum, with rooms labeleled `A` through `R`. If we enter the museum at room `Q`, it is natural to ask how far away the other rooms are.



From the previous algorithm, we can imagine starting at node `Q` with distance 0, then marking all the neighbors of `Q` as having distance 1, then marking all the neighbors of neighbors of ttQ as having distance 2, and so on. You can see that the search algorithm proceeds by creating levels of nodes. A new level is created from all the unvisited neighbors of the previous level.

d(node)	node			
0	Q			
1	K	P		
2	E	L	O	
3	D	F	R	I
4	H	J		
5	B	N		
6	C	M		
7	G			
8	A			

After there are no more levels to consider, the distance array is

```
d = [8,5,6,3,2,3,7,4,3,4,1,2,6,5,2,1,0,3]
```

Note that, if the graph had actually been disconnected, then at the end of our search, some entries of `d` would still be ∞, which is appropriate.

5 Dictionary Representation of a Graph

The adjacency matrix seems a natural representation of a graph. However, there is a more concise representation which can save a lot of space by omitting a recording of all the edges that don't exist.

We can replace our adjacency matrix `A[i,j]` by a Python dictionary, (technically called a *dict*). A dictionary is a collection of *keys* and *values*. We represent the graph as a set of nodes (the keys) and the list of their neighbor nodes (the values). You can think of this as a compressed version of the adjacency matrix which drops all the '0' entries, and replaces the '1' values by the identifier for the node neighbor.

Thus, for our simple 5 node graph, the dictionary format would be

```
G = {
0 : [ 3 ],
1 : [ 2, 3 ],
2 : [ 1, 4 ],
3 : [ 0, 1, 4 ],
4 : [ 2, 3 ]
}
```

Since our museum nodes are labeled with letters, we could describe the museum graph using the following dictionary:

```
museum = {
   'A' : [ 'G' ],
   'B' : [ 'C', 'H' ],
   'C' : [ 'B' ],
   'D' : [ 'E' ],
   'E' : [ 'D', 'F', 'K' ],
   'F' : [ 'E' ],
   'G' : [ 'A', 'M' ],
   'H' : [ 'B', 'I', 'N' ],
   'I' : [ 'H', 'J', 'O' ],
   'J' : [ 'I' ],
   'K' : [ 'E', 'L', 'Q' ],
   'L' : [ 'K', 'R' ],
   'M' : [ 'G', 'N' ],
   'N' : [ 'H', 'M' ],
   'O' : [ 'I', 'P' ],
   'P' : [ 'O', 'Q' ],
   'Q' : [ 'K', 'P' ],
   'R' : [ 'L' ]
}
```

One advantage of this format is that, if we have defined a dictionary `G` for our graph, then for any node `s`, we can generate a `for` loop that examines every neighbor node of `s` using the command:

```
for neighbor in G[s]:
```

We will take advantage of the concise format of the dictionary representation, and the convenience of referencing all the neighbor nodes, in our implementation of the breadth first search algorithm.

6 Breadth First Search

The search style we have used for the edge distance calculation is an example of *breadth first search*. Its structure is something like the gradual spread of a disease or a rumor or a drop of water wetting a cloth, involving the gradual spread from an initial point to nearby neighbors, gradually working its way in all directions.

We have already suggested a simple approach to this problem. However, we will now consider an alternative approach, which uses a `set` and a `queue`, which are Python structures which can be more efficient in processing speed, and allow a more concise programming style.

To carry out a breadth first search in Python, we need a kind of memory that keeps track of rooms we are just about to visit, and rooms that we need to plan to visit later. We will add newly encountered rooms to the end of the list, and choose our next room from the beginning of that list. Technically, this list is known as a *queue*; it's just like an customer service line, where new customers are added at the end, and the next customer to be served is the one at the front.

If our list is called `queue[]`, then

- To choose the next node to visit, we want to "pop" the first item off the list: `node = queue.pop(0)`
- As we visit the next node, we want to "append" its unvisited neighbors to the end of the list: `queue.append(neighbor);`

We actually need a second object, `visited[]`, which we treat as a set. The set includes every node that we have ever visited. We add a node using the command `visited.add(node)`, and we check whether or not a node has been visited with the formula `if ( node not in visited ):`.

The following code performs a breadth first search of a graph. It reports each node that it visits by printing its label. In other situations, we might have more complicated things to do at each node that we visit.

```
def bfs ( G, node ):
#
#  Create empty visited and queue objects.
#
  visited = []
  queue = []
#
#  Add starting node to both lists.
#
  visited.append ( node )
  queue.append ( node )
#
#  Is there a node we haven't checked?
#
  while next:

    node = queue.pop ( 0 )
    print ( node, end = " " )
#
#  Are there neighbors that haven't been visited?
#
    for neighbor in G[node]:
      if ( neighbor not in visited ):
        visited.append ( neighbor )
        next.append ( neighbor )

  print ( "" )

  return
```

7 Depth First Search

Another way we might search a graph corresponds more to how we would wander through a maze. Beginning at the starting node, we observe all the immediately accessible unexplored rooms. We choose one to explore, and "save" the others in some kind of list, planning to visit them later. We repeat this strategy, always moving to an unexplored room, until we reach a dead end. At that point, we reverse our path until we encounter an unvisited room, where we pick up our forward search strategy. By repeatedly traveling all the way to dead ends, and then backing up to the most recent unexplored room, we eventually visit every room. This procedure is known as *depth first search*.

During our search, we will need to choose the next node to visit. To do so, we need to keep track of which nodes have already been visited. We could create a logical array `visited` to do this, with all values initially `False`. However, since in this example we are using letters to label our nodes, we will instead use a *set*, that is, an unordered list of unique values.

Useful things about a Python set:

- The elements don't have to be numeric;
- We can start out with an empty set: `visited = set()`
- We can ask if a node is not already in the set: `if node not in visited`
- We can add a node to the set: `visited.add ( node )`

Now suppose we consider the following recursive depth-first-search algorithm:

```
def dfs ( G, node, visited ):
  if ( node not in visited ):
    print ( node, end = '' )
    visited.add ( node )
    for neighbor in G[node]:
      dfs ( G, neighbor, visited )
  return
```

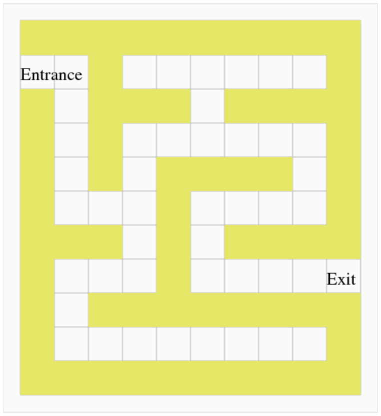
We can invoke a search of the museum map by initializing the `museum` dictionary definition, and `visited` and then calling as follows:

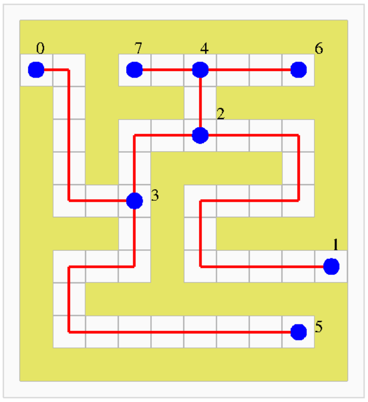
```
museum = [ ... ]
visited = set ( )
dfs ( museum, 'Q', visited )

Q K E D F L R P O I H B C N M G A J
```

Unlike the breadth first search, the depth first search has been implemented using recursion. Each recursive step explores a new room. If that room has unexplored neighbors, we take another recursive step; if it does not, then we back up one call; that is, we retreat to the previous room, and make the check again. Once all nodes have been visited, the code terminates.

8 Solving a Maze

A *maze* is an arrangement of rooms, connected in a complicated way. A maze might have an entrance and exit, or we might be required to go from one room to another.





A maze can be represented by a graph. The entrance, the exit, every dead-end, and every point where we have to make a choice should each be a node. The paths between these points become edges. The “right hand rule” suggests getting from one spot by walking with your right hand always touching the wall, turning whenever you have to to keep your hand on the wall.

In the figure above, we have a simple maze and a suggestion of its corresponding graph. We can easily verify that the “right hand rule” works for the yellow maze, by listing the sequence of nodes, starting at node 1, that lead to node 2 following this rule;

We can create an adjacency matrix for the maze:

$$A = \left[\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & . & . & . & 1 & . & . & . & . \\ 1 & . & . & 1 & . & . & . & . & . \\ 2 & . & 1 & . & 1 & 1 & . & . & . \\ 3 & 1 & . & 1 & . & . & 1 & . & . \\ 4 & . & . & 1 & . & . & . & 1 & 1 \\ 5 & . & . & . & 1 & . & . & . & . \\ 6 & . & . & . & . & 1 & . & . & . \\ 7 & . & . & . & . & 1 & . & . & . \end{array}\right]$$

The equivalent dictionary description is

```
M = {
0 : [3],
1 : [2],
2 : [1,3,4],
3 : [0,2,5],
4 : [2,6,7],
5 : [3],
6 : [4],
7 : [4]
}
```

Now we are going to use depth-first search on this maze, starting at node 0, and trying to reach a goal of node 5. Think of this procedure as something like this:

- Mark all nodes as unvisited.
- Visit 0, our current node;
- If the current node has no unvisited neighbors, back up to previous node;
- Otherwise, replace current node by an unvisited neighbor, and "remember" other nodes you didn't visit yet.

Here is how that procedure might be carried out for our simple maze:

```
node  edges
----  --------------
0
3     0->3
2        3->2
1           2->1     Dead end, back up from 1 to 2
2           2<-1
4           2->4
6              4->6  Dead end, back up from 6 to 4
4              4<-6
7              4->7  Dead end, back up from 7 to 4
4              4<-7
2           2<-4     No unvisited nodes, back up from 4 to 2
3        3<-2        No unvisited nodes, back up from 2 to 3
5        3->5        Success, we found the goal.
```

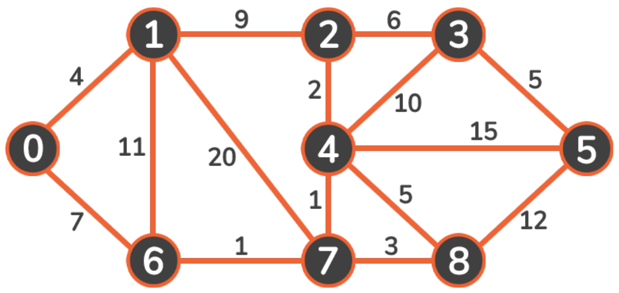
The exercises ask you to implement a depth-first search algorithm that solves this maze.

9 Shortest Path

Earlier, we computed the distance between a starting node i and every other node j in a graph G. We measured the distance as the number of edges we needed to travel across in our journey. But if our graph represented, say, a highway map, then an edge is a trip from one city to a neighboring city, and it is important to know the associated distance involved. It is natural to represent such a system using a "distance matrix" D, in which $D_{i,j}$ is the distance between nodes i and j. If there is no connection between the nodes, we set $D_{i,j} = \infty$. In Python, this would be done by

```
D[i,j] = np.Inf
```

A graph in which each edge is assigned a value is called a *weighted graph*. These weights are usually required to be positive. They might represent distances, but could instead represent other quantities, such as the flow capacity of a pipe. We are interested in solving the node-to-node distance problem again, but now with weights assigned to each edge. As an example, consider the following weighted graph, with 9 nodes and 15 edges.



What is the shortest distance from node 0 to all other nodes?

The distance matrix for this graph will be:

$$D = \left[\begin{array}{c|ccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
0 & 0 & 4 & . & . & . & . & 7 & . & . \\
1 & 4 & 0 & 9 & . & . & . & 11 & 20 & . \\
2 & . & 9 & 0 & 6 & 2 & . & . & . & . \\
3 & . & . & 6 & 0 & 10 & 5 & . & . & . \\
4 & . & . & 2 & 10 & 0 & 15 & . & 1 & 5 \\
5 & . & . & . & 5 & 15 & 0 & . & . & 12 \\
6 & 7 & 11 & . & . & . & . & 0 & 1 & . \\
7 & . & 20 & . & . & 1 & . & 1 & 0 & 3 \\
8 & . & . & . & . & 5 & 12 & . & 3 & 0
\end{array}\right]$$

where, for ease of viewing, we have written periods instead of ∞.

Now suppose that we want to find the shortest distance from a particular node (let's assume this is node 0) to any other node. We can do this by defining an array s such that s_j is the length of the shortest path from our starting node to node j.

The approach to filling in the array s is known as *Dijkstra's algorithm.* It goes as follows.

1. Initialize the array `s` to `np.Inf`.
2. Initialize the array `visited` to `False`.
3. Set `s[start] = 0`.
   (a) Set the current node to the unvisited node with smallest value of s.

(b) For every unvisited neighbor j of the current node, set `s[j] = min ( s[j], s[current] + D[current,j] ) ;`

(c) Set `visited[current]=True`.

(d) If there are no more unvisited nodes, terminate.

Once the algorithm has completed, the `s` array contains the shortest distance from the starting node to each node in the graph. If the graph is not connected, then some distances will be infinite.

10 Shortest Path Example

We can apply Dijkstra's algorithm to our example weighted graph. Each step of the algorithm chooses one node having the minimum distance, and then updates the distances of any unvisited neighbors. Here is how the entries of the array s are updated over the the sequence of steps of Dijkstra's algorithm.

0	1	2	3	4	5	6	7	8	base node
0	∞	∞	∞	∞	∞	∞	∞	∞	initialize
0	4	∞	∞	∞	∞	7	∞	∞	node 0, distance 0
0	4	13	∞	∞	∞	7	24	∞	node 1, distance 4
0	4	13	∞	∞	∞	7	8	∞	node 6, distance 7
0	4	13	∞	9	∞	7	8	11	node 7, distance 8
0	4	11	19	9	24	7	8	11	node 4, distance 9
0	4	11	17	9	24	7	8	11	node 2, distance 11
0	4	11	17	9	23	7	8	11	node 8, distance 11
0	4	11	17	9	22	7	8	11	node 3, distance 17
0	4	11	17	9	22	7	8	11	node 5, distance 22

11 The Traveler's Problem

A traveler wants to make a visit to each of n cities, during a round trip that begins and ends at city 0. For every pair of cities (i, j), the value $D_{i,j}$ records the distance in miles, or is ∞ if there is no direct connection possible. The traveler wishes to minimize the total mileage involved. This abstract problem has many real-world applications in business and industry, and is known as the *traveling salesperson's problem* or TSP.

It's clear that a solution must exist. There are only a finite number of possible round trips, and so we could imagine listing them all, and computing the associated mileage, and choosing one that minimizes the cost. A trip schedule would simply list the cities in order. If the cities are labeled with letters, a sample trip around $n = 5$ cities might look like `OCADBEO`. It should be clear that this suggests that there are $n!$ possible itineraries, a number which grows extraordinarily rapidly. In a real example, many city-to-city links might not exist, which reduces this complexity somewhat. Nonetheless, the TSP is a classic example of a problem which rapidly becomes too difficult for an exact solution.

For our example weighted graph, we can try a brute force approach to determine the shortest round trip. In roughly 0.12 seconds, we can determine that the minimal path has length 50, and proceeds as follows:

```
   4    9    6    5   12    5    1    1    7
 0-->1-->2-->3-->5-->8-->4-->7-->6-->0
```

The exercises ask you to try to reproduce this calculation.

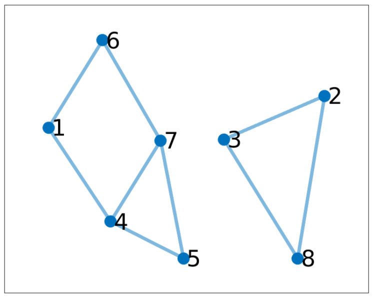
12 Traveling Salesperson Heuristics

Given that larger TSP problems are extremely difficult to solve, many suggestions have been made for coming up with approximate solutions. In each case we start with a basic itinerary `0-->?-->0` and try to fill in the blanks.

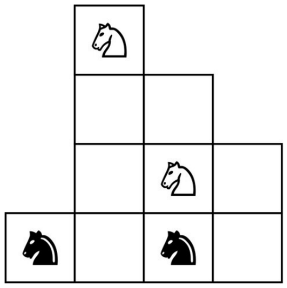
- *Nearest neighbor*: From your current location, add a trip to the nearest city that has not been visited yet. Repeat until all cities have been reached, and go home.
- *Greedy algorithm*: Find the shortest unused city-to-city trip which does not create a "short circuit" and which does not involve a city which already has a trip in and a trip out; Add this trip to your itinerary. Keep doing this until your itinerary includes every city.
- *Insertion*: Choose an unvisited city k at random. For every pair of consecutive cities i and j already in your itinerary, compute the length of the revised itinerary $i- > k- > j$. Choose to insert k between the cities i and j which minimize this change in trip length.
- *Transpose*: Start with an itinerary chosen at random. Pick two distinct non-neighboring cities i and j. Reverse the order of all the cities in the itinerary between i and j. If this reduces the total length, keep the change. Keep trying.

13 Exercises

1. Consider the graph in the following diagram. Compute the appropriate path matrix that can check whether the graph is connected or not. Explain what the path matrix says about this graph.

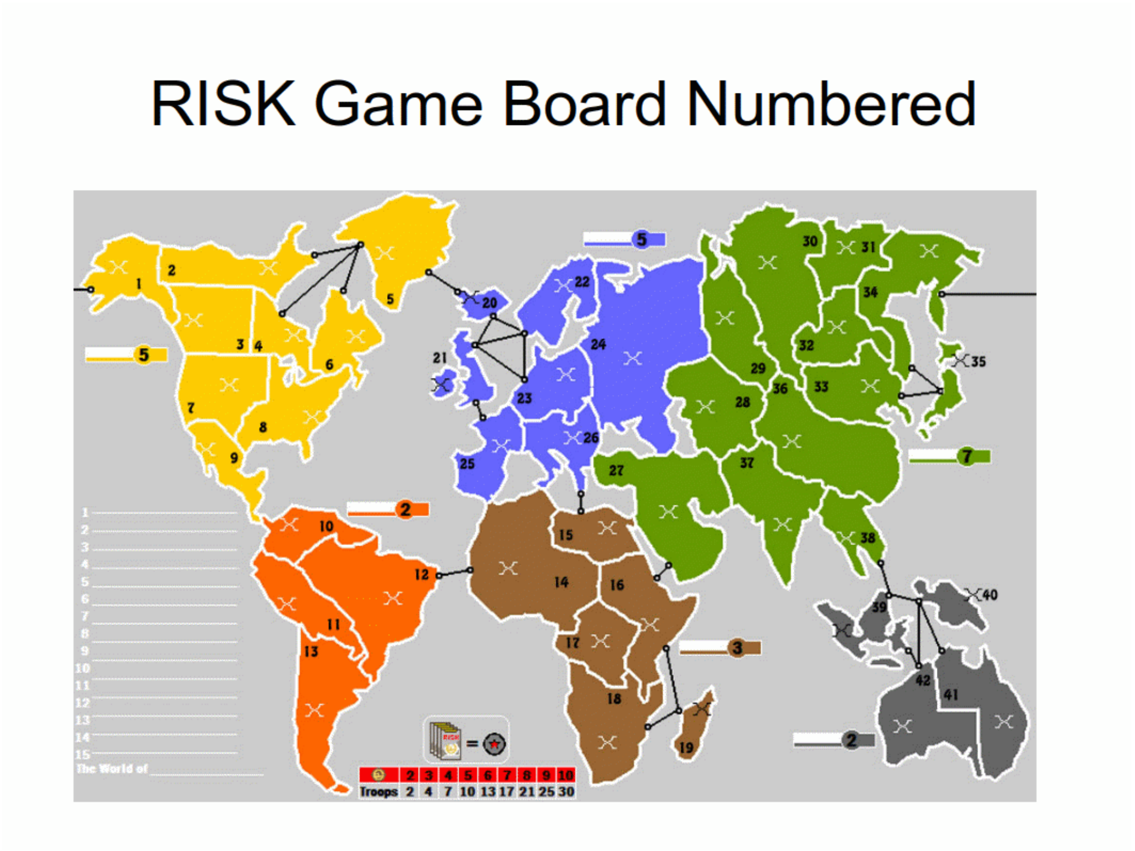


2. In the following puzzle, it is your task to interchange the black and white knights, using only knight moves, remaining on the fragmentary board that is shown, using no captures. You may find it very useful to construct the adjacency matrix in which two squares are adjacent if there is a legal knight move from one to the other.

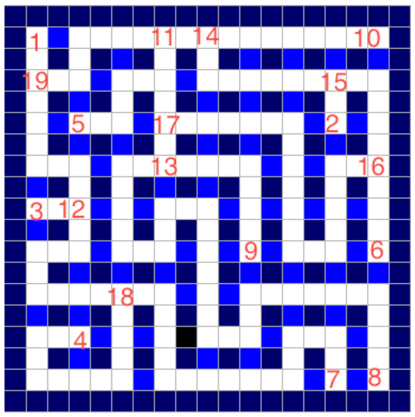


3. For the museum problem, write a Python program which computes the distance array d given any starting room. If your starting room is Q you should match the results reported above.

4. The game of Risk involves a map of 42 countries. Squinting at the map as best you can, determine the adjacency matrix. Then work out the distance to every country if you start from Mexico (country #9). Since Risk numbers starting at 1, you will have to make the usual Python index adjustment. What is the furthest country from Mexico?



5. Assume that you are given the blue maze in the figure below, and that you are to find a sequence of nodes that takes you from node 1 to node 2. Verify that the "right hand rule" does **not** work for this maze, by listing the sequence of nodes, starting at 1, until you come back to node 1. A depth first search will correctly find a path from node 1 to node 2. Give the steps that such a search might use. You can stop when you reach node 2, and you may allow your example to be "lucky". Your report will indicate, at each step of the algorithm, where you are (what node), what nodes you have seen but not explored, and what node you are about to explore next.
Step 0: I am at node 1, I see node 19, and I am about to move to node 19.



6. Repeat the depth-first search of the museum, but instead of using a set to store the list of visited nodes, use a logical array `visited`. To do this, you will need to redefine the museum dictionary, replacing the alphabetic node labels by numeric values.
7. Implement the depth-first search algorithm for the maze example; print out the sequence of nodes you visit, and compare it to the results reported in the text.
8. Write a Python code that implements Dijkstra's algorithm. Apply it to the 9-node example graph. Verify that your results agree with the text.
9. Dijkstra's algorithm doesn't actually remember the shortest path it used to reach each node. Modify the algorithm so that, every time that the distance d_j is adjusted using an edge from node i to node j, a corresponding value p_j is set to i. Show that this is enough information to work your way back from any node j to the starting node. In the 9-node example graph, compute the shortest path back from node 5 to node 0.
10. Using brute force, enumerate every possible round trip starting at node 0 on the weighted graph. For each round trip, evaluate the total distance. Report the length and itinerary of the shortest trip you found, the number of possible itineraries, and the time required for your computation.
11. Try to solve the TSP for the following distance matrix, using any of the heuristics given above, or any other method you can think up. The shortest trip I found had length 2090. The minimal tour length is actually 2085.

```
    0 633 257  91 412 150  80 134 259 505 353 324  70 211 268 246 121
633   0 390 661 227 488 572 530 555 289 282 638 567 466 420 745 518
257 390   0 228 169 112 196 154 372 262 110 437 191  74  53 472 142
 91 661 228   0 383 120  77 105 175 476 324 240  27 182 239 237  84
412 227 169 383   0 267 351 309 338 196  61 421 346 243 199 528 297
150 488 112 120 267   0  63  34 264 360 208 329  83 105 123 364  35
 80 572 196  77 351  63   0  29 232 444 292 297  47 150 207 332  29
134 530 154 105 309  34  29   0 249 402 250 314  68 108 165 349  36
259 555 372 175 338 264 232 249   0 495 352  95 189 326 383 202 236
505 289 262 476 196 360 444 402 495   0 154 578 439 336 240 685 390
353 282 110 324  61 208 292 250 352 154   0 435 287 184 140 542 238
324 638 437 240 421 329 297 314  95 578 435   0 254 391 448 157 301
 70 567 191  27 346  83  47  68 189 439 287 254   0 145 202 289  55
211 466  74 182 243 105 150 108 326 336 184 391 145   0  57 426  96
268 420  53 239 199 123 207 165 383 240 140 448 202  57   0 483 153
246 745 472 237 528 364 332 349 202 685 542 157 289 426 483   0 336
121 518 142  84 297  35  29  36 236 390 238 301  55  96 153 336   0
```