Intro Math Problem Solving
December 05

The Maze Problem

Is a Graph Connected?

Distance from a Node

Can You Get From A to B?

Search the Graph

Final Project Correction!

In problem 4, the original instructions told you to check whether one date was notequal to another by:

```
if ( D1~= D2 && M1 ~= M2 && Y1 ~= Y2 )...
```

but the correct statement uses OR's:

```
if ( D1~= D2 || M1 ~= M2 || Y1 ~= Y2 )...
```

Two dates are unequal if any of D, M or Y is different.

References

Nick Berry describes properties of mazes, the relation to graph theory, and how to create a "perfect" graph in an online article at: http://datagenetics.com/blog/november22015/index.html

He includes an interactive demo of a depth-first-search procedure that sets up a random maze.

Jamis Buck,

Mazes for Programmers:

Code Your Own Twisty Little Passages

Lewis Carroll Doublets: *https://en.wikipedia.org/wiki/Word_ladder*

Six Degrees of Kevin Bacon: *http://oracleofbacon.org/movielinks.php*

The Maze Problem

The Maze Problem

A maze is a puzzle involving a number of rooms or locations connected by doors or passageways. One location is the entrance or starting point, the other location is the exit.

The player seeks the exit location by moving from room to room.

The player has no idea where the exit is, there are many dead-ends, and when faced with a choice of several new rooms to explore, there is no obvious choice.

Searching the Labyrinth

Theseus in the Labyrinth

In mythology, the Greek hero Theseus entered the Labyrinth, an underground maze where the Minotaur monster lived.

He had to wander through the maze until he encountered and killed the Minotaur.

Once his mission was accomplished, he to escaped the Labyrinth using a spool of thread he had been unwinding since he entered the maze.

Deleting Dead Ends

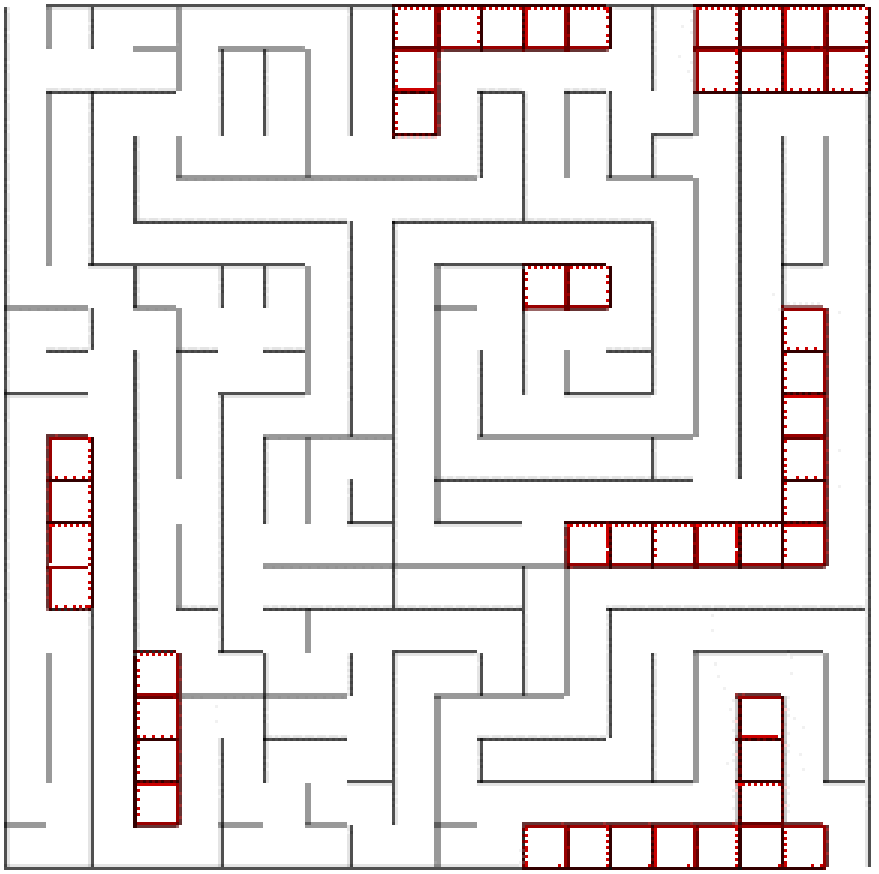
A spool of thread only helps you retrace your steps.

What if you want to solve a maze, that is, get from the start to the exit?

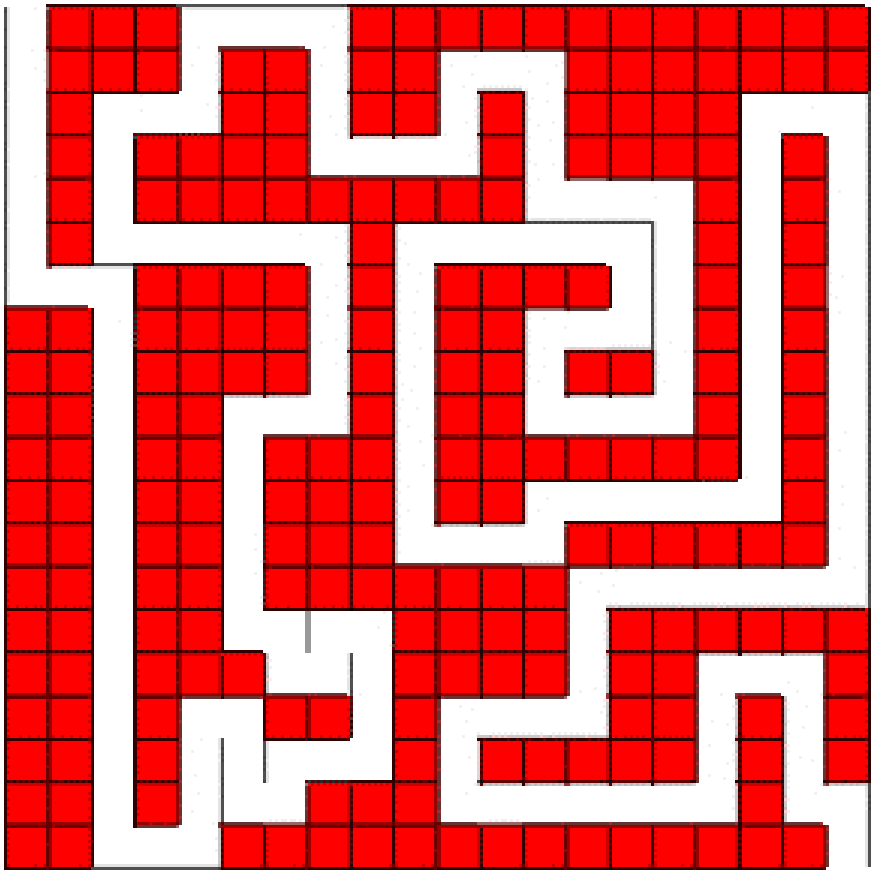
If you are lucky to have a map, you can try to simplify the puzzle by filling in all the dead ends. As you fill them in, you may see more dead ends that can be eliminated.

Dead End Elimination: *Given a map of the maze, eliminate all paths that you can see are dead ends. The remaining paths may form a direct route to the exit.*

Removing First Dead Ends



After All Dead Ends Removed



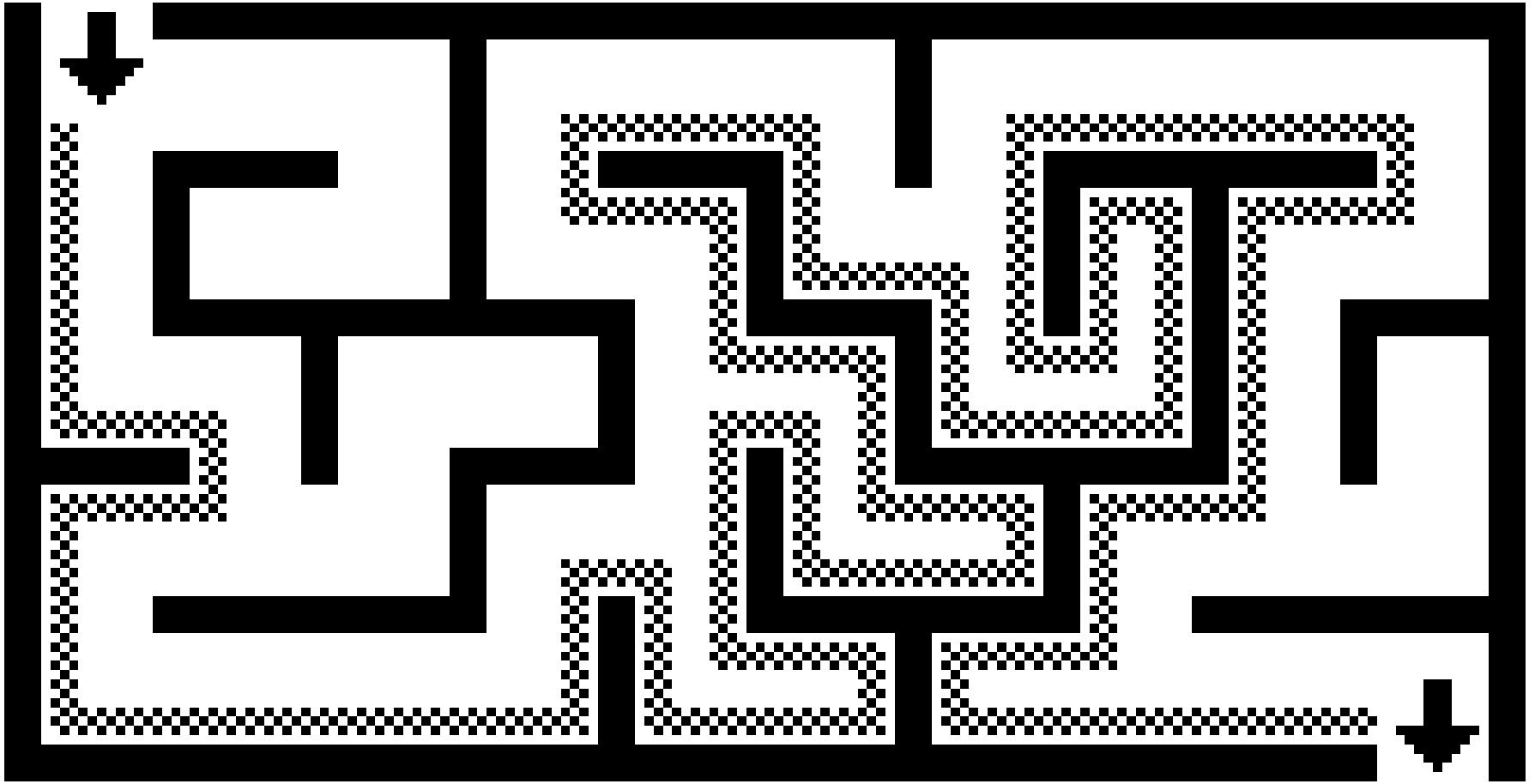
A Maze Algorithm

The dead-end rule can be useful if you have a map or a printed puzzle, as long as the puzzle has just a single path to the exit.

What if you simply walk into a maze and have to find your way through, using only local information?

A suggested method is the Right-Hand-Algorithm: *at the starting point, place your right hand on the maze wall. Then proceed in such a way that your right hand never leaves the wall. If the maze is "simple", you will reach the exit.*

Using the Right Hand Rule



Right Hand Rule Idea

The right hand rule works for simple mazes, in which, if we ignore the openings for the entrance and exit, the maze walls form a single object.

That's because the right hand rule is essentially tracing the entire maze wall, up to the point where the exit is encountered.

After all, if someone told you to **paint** the maze, you could do it this way, leaving a continuous trail of paint until you reached the exit.

In fact, if there was no exit, then we would exactly trace the entire maze and return to the entrance.

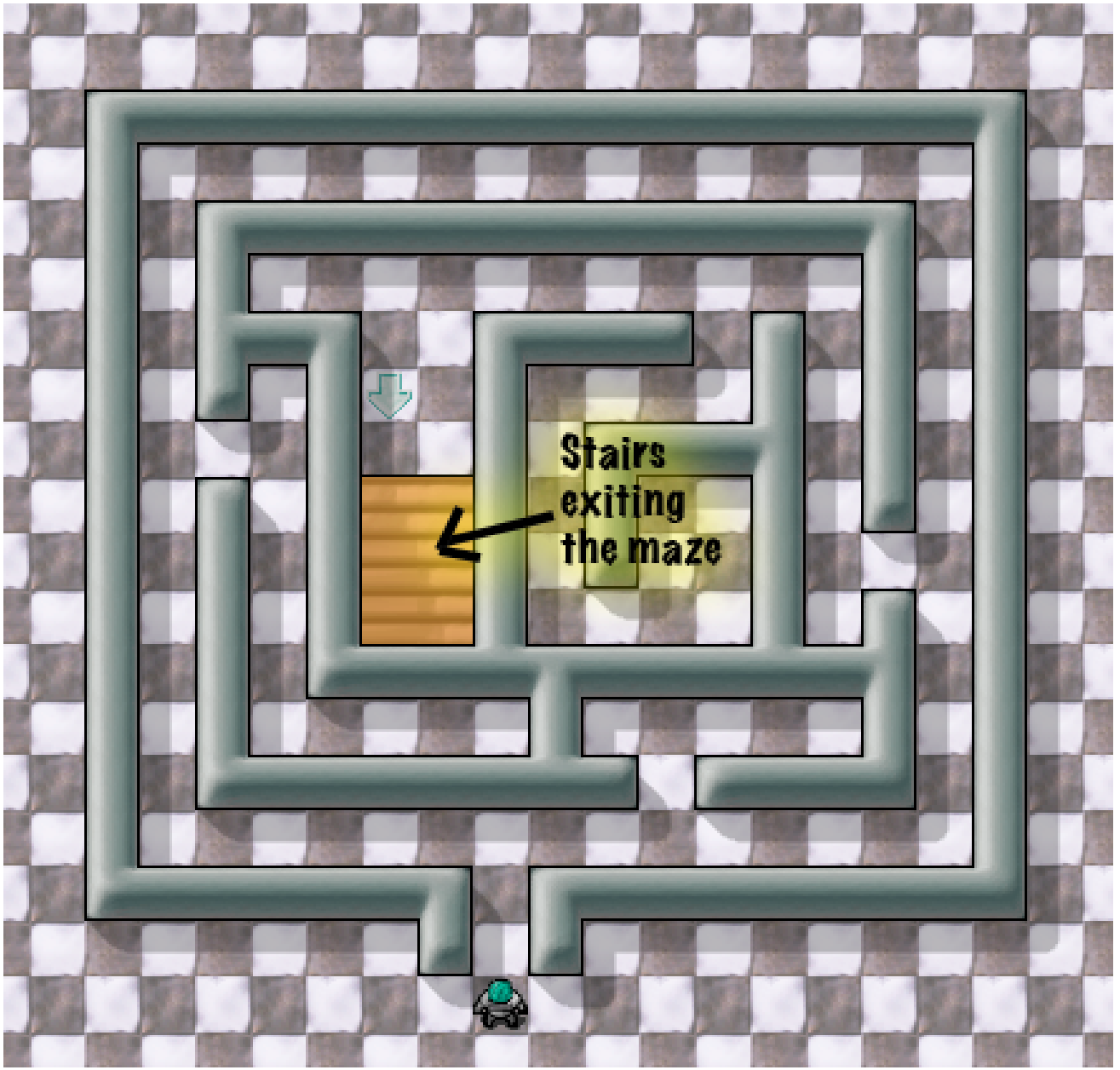
Right Hand Rule Can Fail

The Right Hand Rule can fail if the maze has a more complicated structure.

In particular, if the entrance or exit is in the interior of the maze, and the maze walls are not one connected piece, then the rule can fail.

Corn mazes often include overpasses or tunnels that make the right hand rule fail.

Right Hand Rule Can Fail



Tremaux: A Better Maze Rule

Mark each path as you follow it. The marks need to be visible at both ends of the path. You can imagine the marks as being your footprints, or as chalk marks you make at the beginning and end of each path.

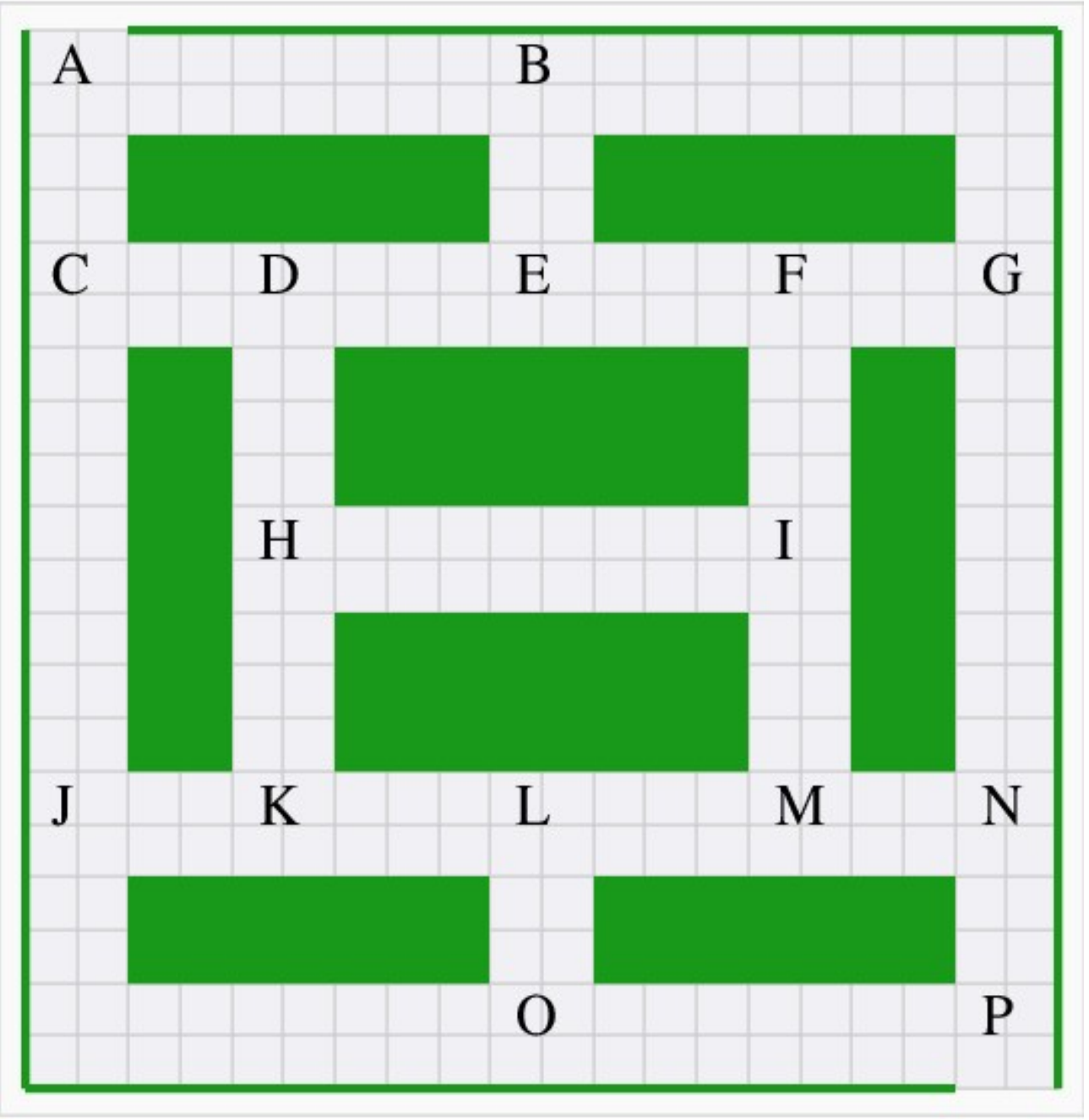
Never enter a path which has two marks on it (footprints in + footprints out).

If you arrive at a junction where you have to choose your next path:

* if all the new paths are unmarked, choose an arbitrary path and proceed;
* if your current path has only one mark, turn around.
* else your path has two marks. Choose the path with the fewest marks.

When you finally reach the solution, paths marked exactly once will indicate a way back to the start.

Example Maze for Tremaux



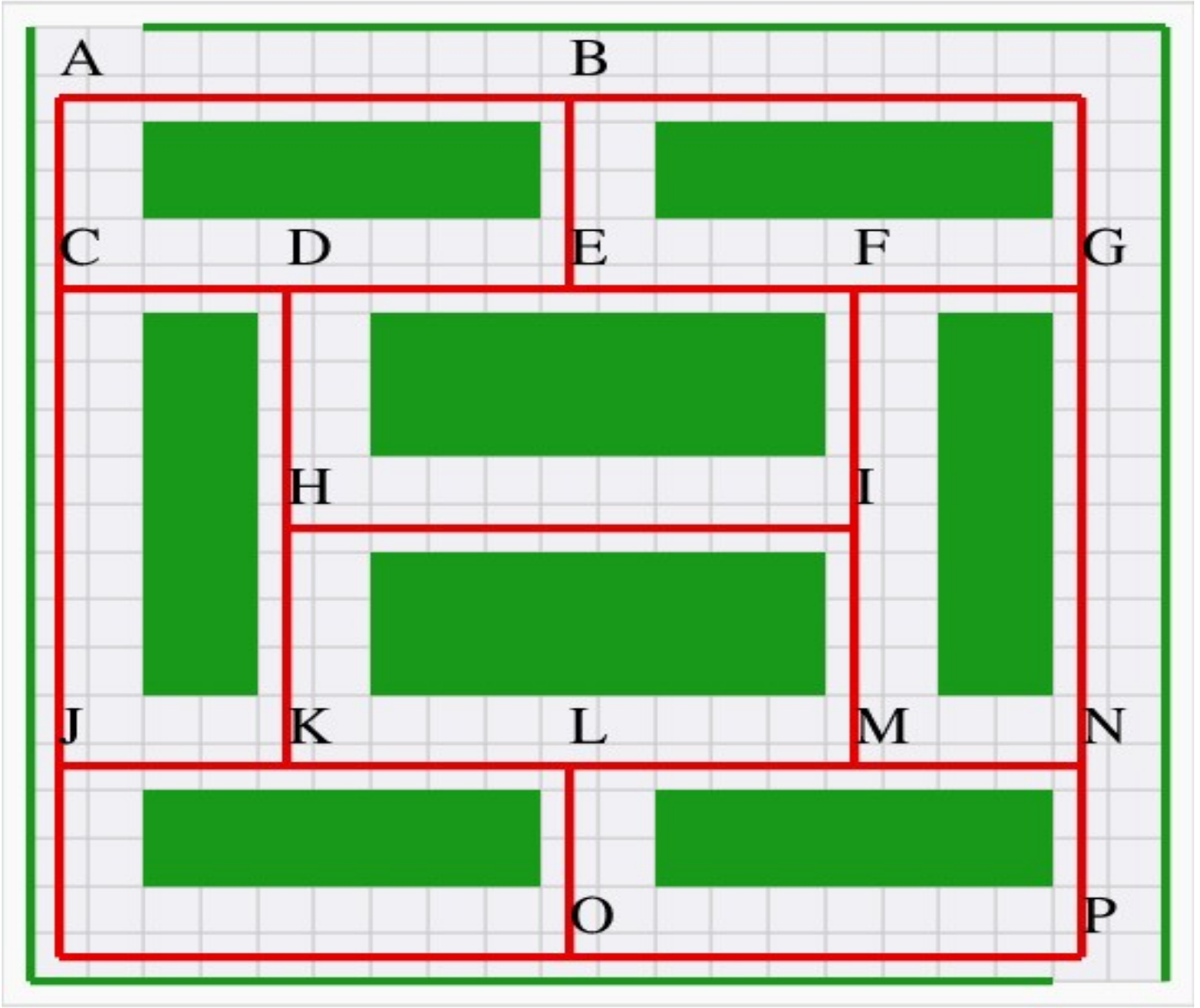
Maze -> Graph

If we think of the "rooms" as nodes, and the connections or pathways as "edges", then we can represent any maze as a mathematical graph.

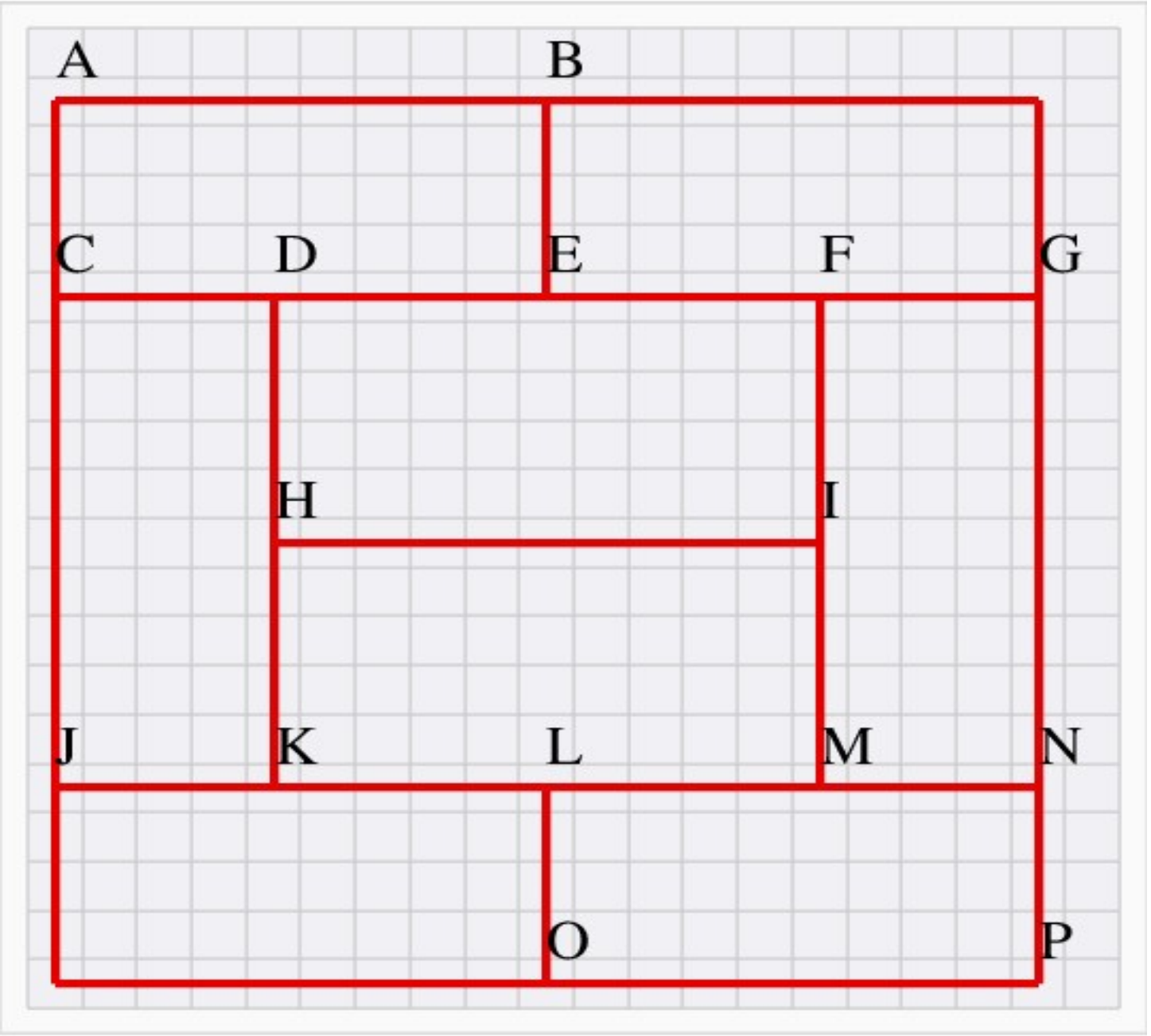
This means we can store the maze as an adjacency matrix (or edge list or other data structure).

It means we can apply mathematical algorithms to the graph, in order to answer questions about the maze.

Tremaux Maze -> Graph



Tremaux Graph

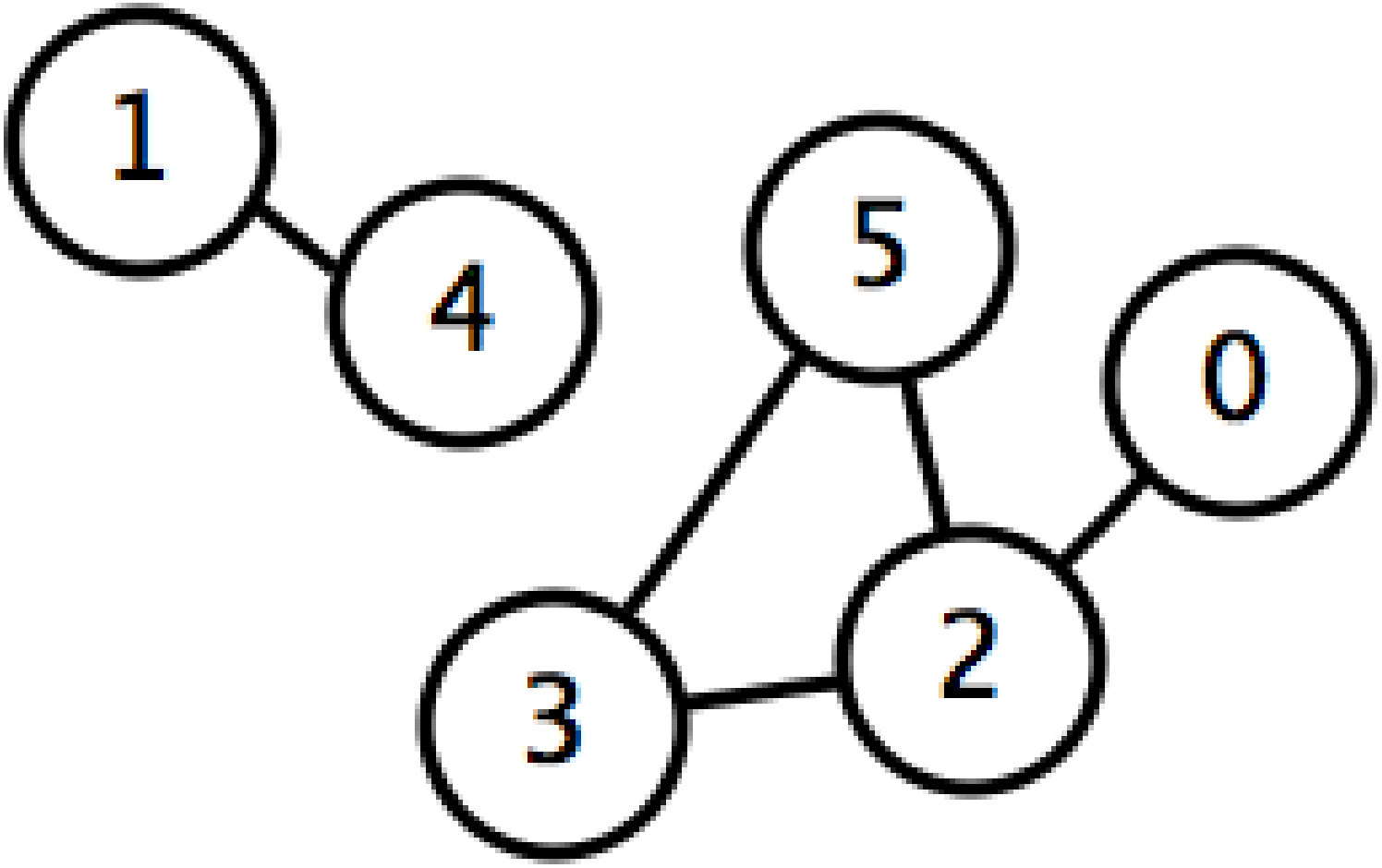


Questions

Questions we can ask about our maze, or any graph:

* Is the graph connected?
* How far away is each node from the entrance?
* Is there a path from A (entrance) to B (exit)?
* What is that path?
* How can we be sure to visit every node in the graph?
* How can we search for the Minotaur, and then find our way back to the starting position?

Is a Graph Connected?



Connection Detection

A graph is connected if it is possible to find a path (a sequence of edges) that lead from any node A to any other node B.

Our eye can easily decide whether a graph is connected, at least for small graphs.

For the maze puzzle, connectedness guarantees that no matter which nodes we choose as start and finish, there will be a path from one to the other.

Connected: Any A to Any B

From the definition, we know that a graph G is connected if we can get from any node A to any node B. This might suggest the following algorithm.:

```
value = true;
for a = 1 : n
  for b = 1 : n
    value = a_to_b ( adj, a, b );  % Is there a path from node A to node B?
    if ( value == false )
      return
    end
  end
end
```

A "Flooding" Solution

Here is a simpler and faster approach for connection.

Suppose our bathtub starts overflowing at node 1. What dry rooms will become wet? The rooms connected to room 1. Stop worrying about room 1, and start worrying about these newly flooded rooms (say rooms 5 and 7).

Any room connected to 5 or 7 that was dry will also become wet. List all those rooms, and forget about 5 and 7.

Keep doing this until no more rooms get flooded. Count the wet rooms. Does this match the number of rooms in the graph? Then the graph is connected.

GRAPH_CONNECTED in Words

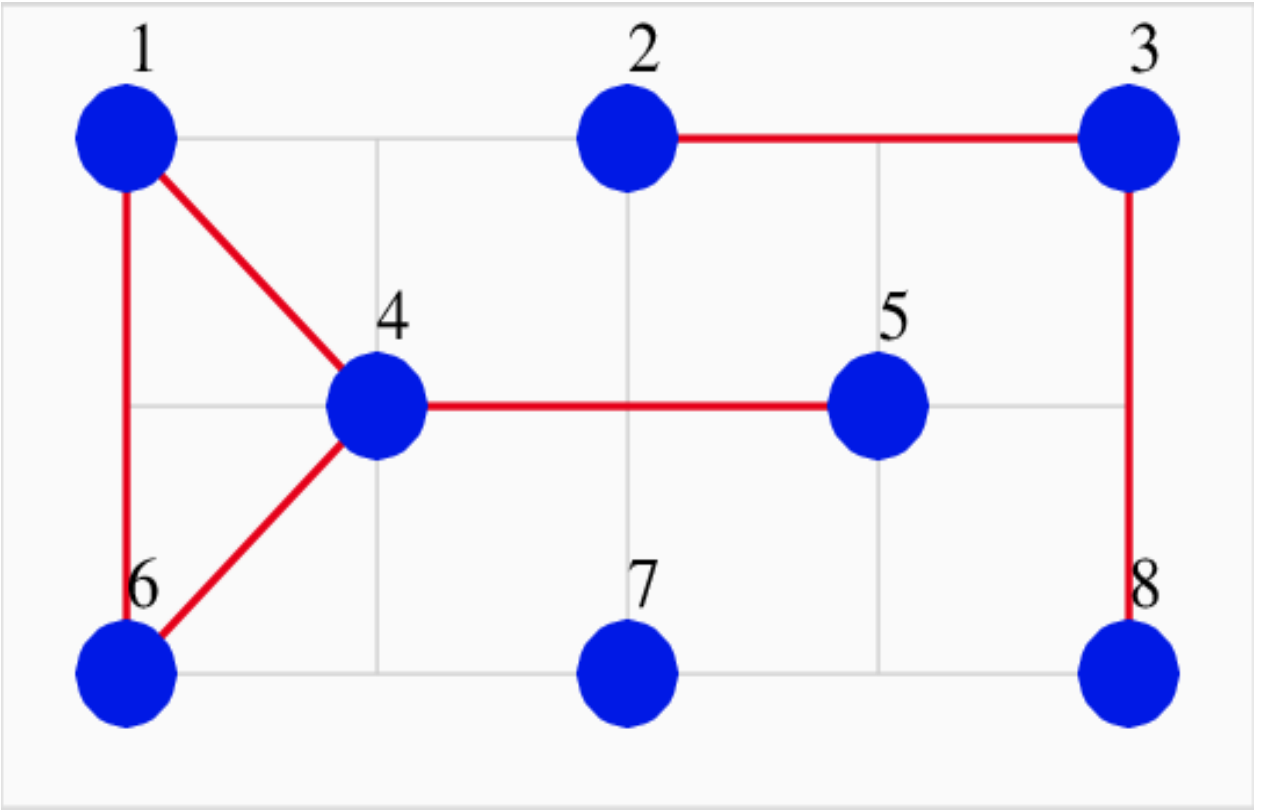
```
function value = graph_connected ( adj )
%
%% GRAPH_CONNECTED is true if the graph G is connected.
%
%  N is the number of nodes.
%
%  WET(I) will be TRUE if node I got wet.
%  WET_NUM counts how many nodes are wet.
%
%  OLD holds the nodes that got wet on the last step.
%  NEW lists the nodes that just got wet on this step.
%
%  If there were NEW nodes on the previous step, make them OLD,
%  set NEW to the empty list and take another step:
%
%     For each node J in the OLD list
%
%        if node I a neighbor node that is DRY?
%        then node I is WET, node I is added to the NEW list, and WET_NUM increases by 1.
%
%  If WET_NUM == N, then the graph is connected.
```

graph_connnected.m

```
function value = graph_connected ( adj )

  [ n, n ] = size ( adj );
  reach = zeros(1,n);
  reach(1) = true;
  new = [1];
  reached = 1;
  while ( 0 < length ( new ) )
    old = new;
    new = [];
    for k = 1 : length ( old )
      j = old(k);
      for i = 1 : n
        if ( adj(i,j) == 1 )
          if ( ~ reach(i) )
            reach(i) = true;
            new = [ new, i ];
            reached = reached + 1;
          end
        end
      end
    end
  end
  value = ( reached == n );
  return
end
```

Disconnected Graph

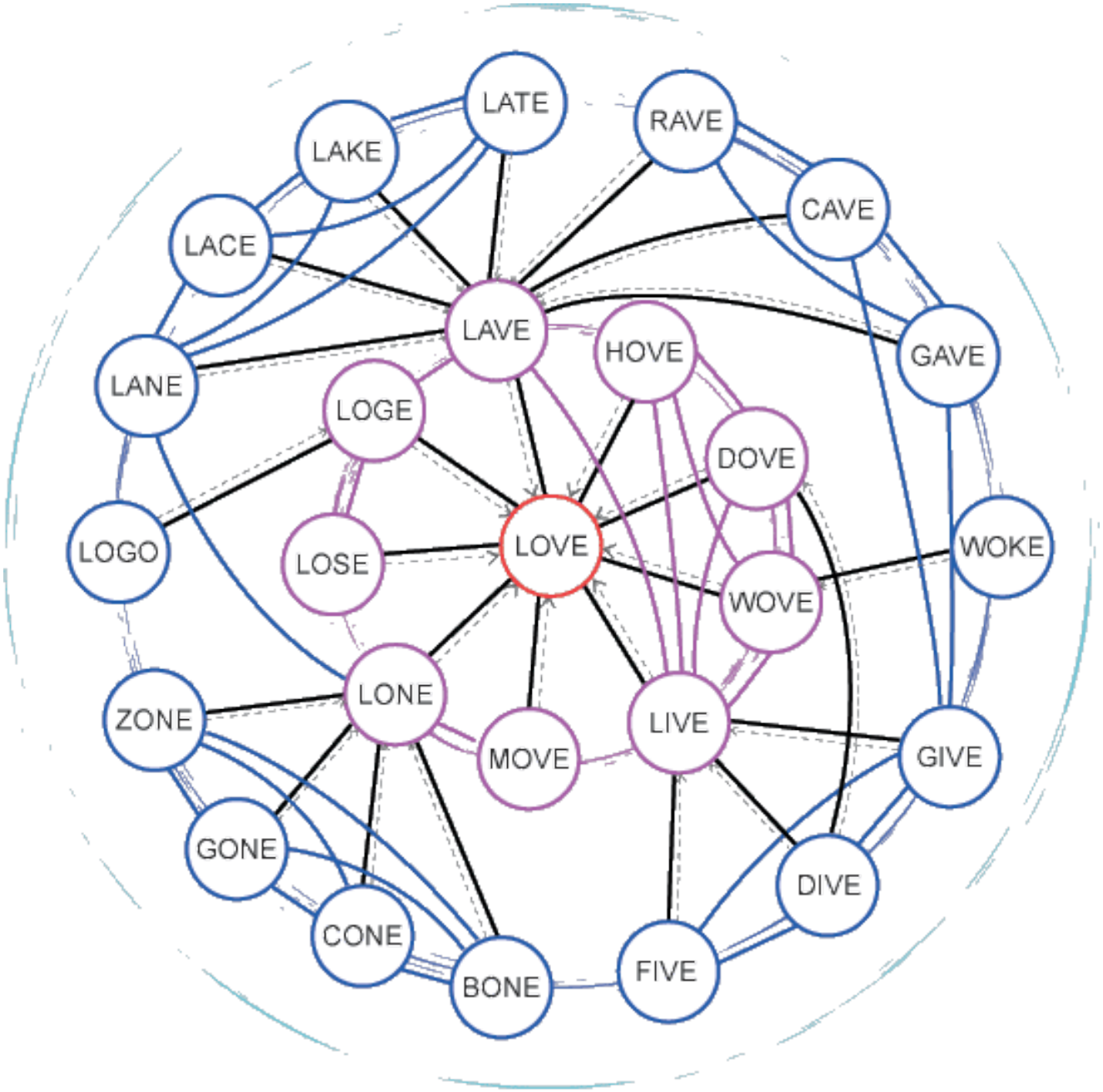
	1	2	3	4	5	6	7	8
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	0	0	0
3	0	1	0	0	0	0	0	1
4	1	0	0	0	1	1	0	0
5	0	0	0	1	0	0	0	0
6	1	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0



The row of zeros makes it easy to see that this graph is not connected.

Usually, it is not so easy!

Distance from a node



Lewis Carroll's "Doublets"

A word game invented by Lewis Carroll tries to transform one word into another by switching one letter at a time. Although Carroll called the game "Doublets", it is also known as "Word Ladders". The score is the number of steps needed.

Thus, FIND to LOSE:

FIND

FINE

LINE

LIVE

LOVE

LOSE

for a score of 5.

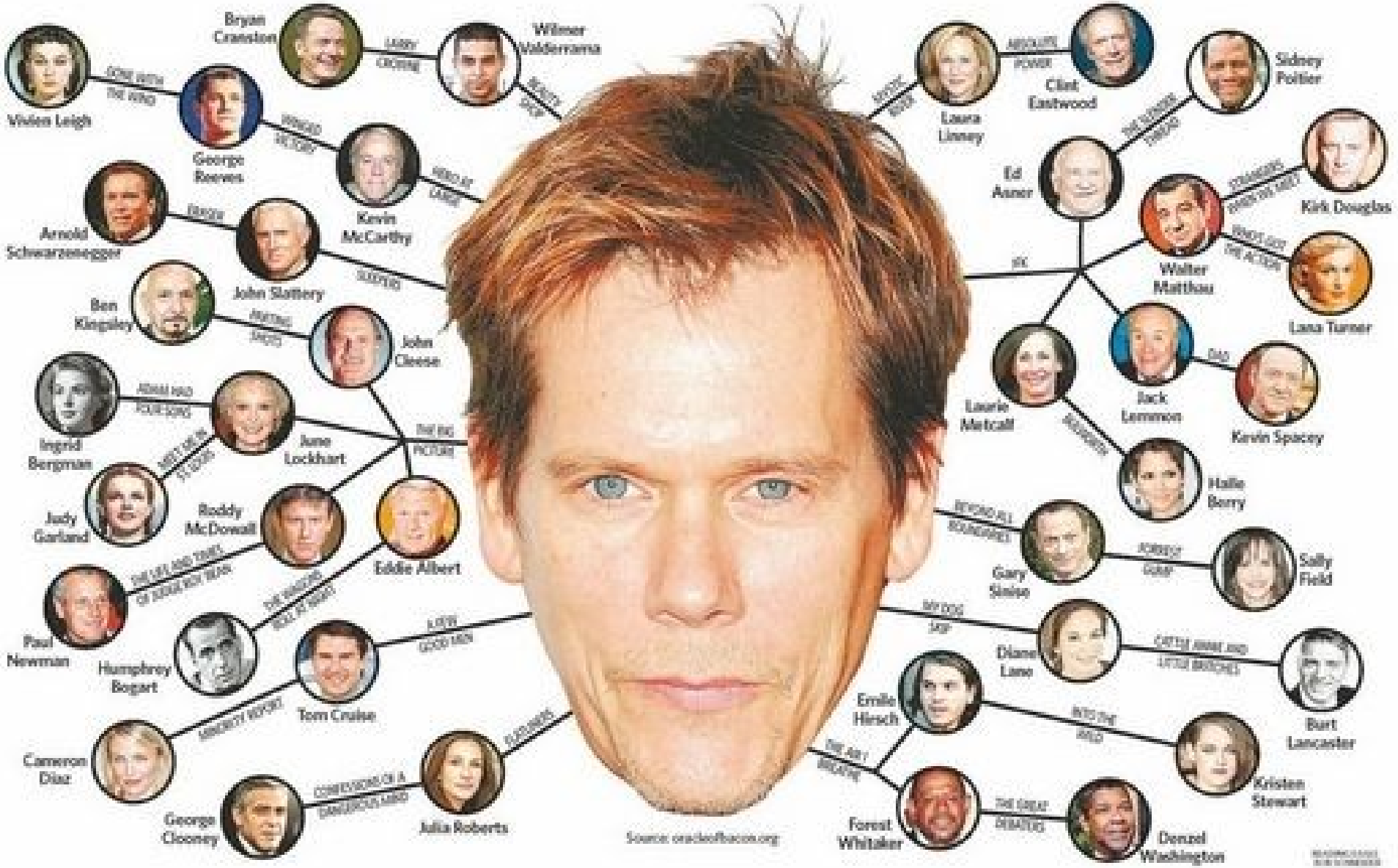
A Graph Distance Puzzle

We can think of the problem of changing “FIND” to “LOSE” as a graph puzzle. Imagine all four letter words as nodes. Connect any two words that differ in a single position.

Our task is to find a path from “FIND” to “LOSE” (if there is one) and to choose the path that is the shortest, that is, that requires the fewest number of edges to make the trip.

More puzzles: LOVE to HATE, WARM to COLD, WHEAT to BREAD, TEARS to SMILE...

Six Degrees of Kevin Bacon



Another Graph Puzzle

In the Kevin Bacon game, the challenger names an actor, and the player must find a sequence of movies that link that actor to Kevin Bacon. For instance, if the challenge is "Heath Ledger" then:

1: Heath Ledger

was in "I'm Not There" and so was...

2: Tyrone Benskin

was in "Criminal Law", and so was...

3: Kevin Bacon

So Heath Ledger is "two degrees" from Kevin Bacon.

Graph Distance

Lewis Carroll's Doublets and Six Degrees of Kevin Bacon are simple examples of the problem of determining the distance between two nodes on a graph.

We assume that all we have to work with is the adjacency information, which tells us about immediate neighbors.

What we want to know is the minimum number of steps we need in order to go from a given starting point to a given goal.

Distance from Node A to any Node

The “flooding” idea, which we used to determine if a graph is connected, can also help us to determine how many steps it takes to reach any node if we start at a given node A.

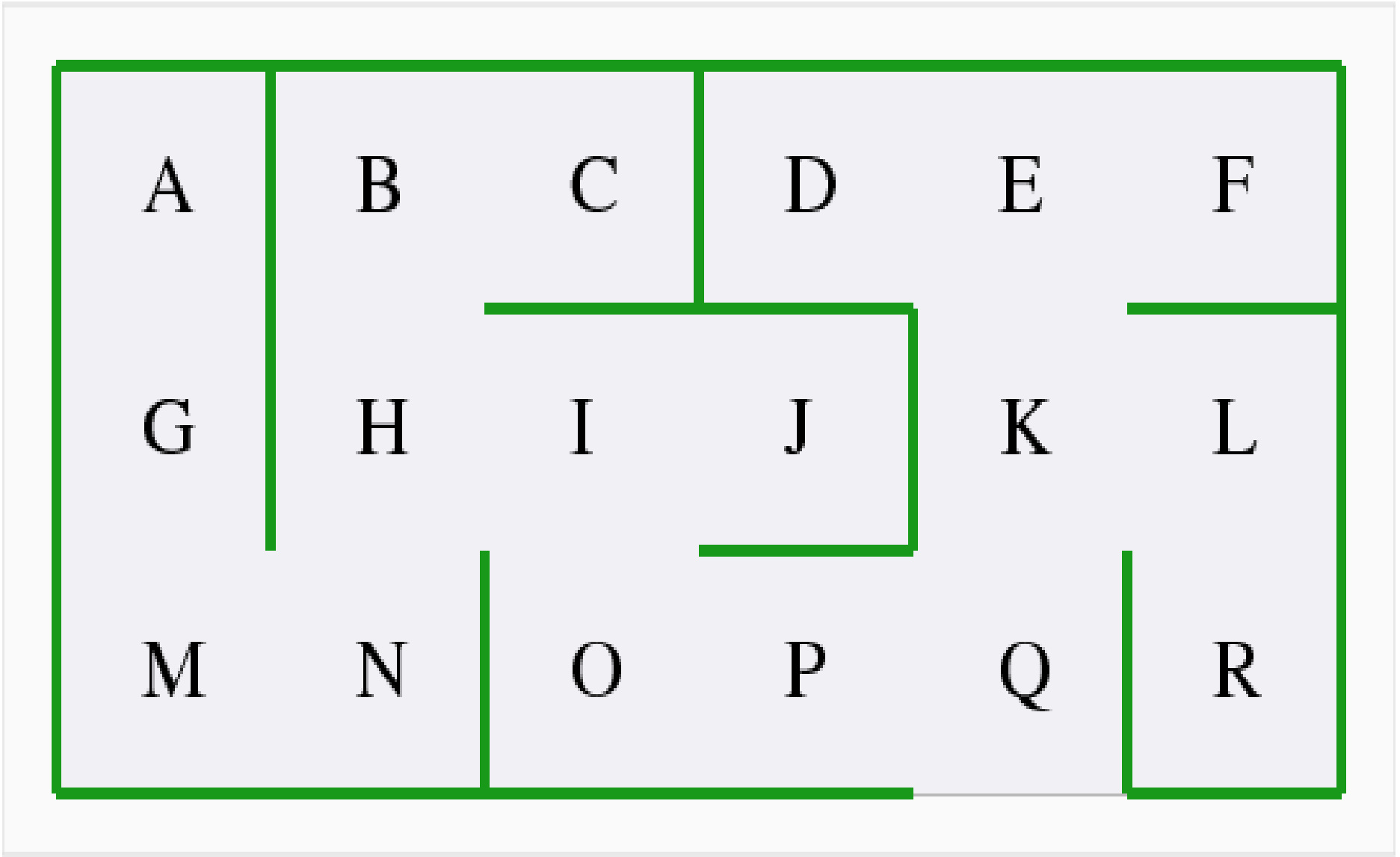
Node A will have a distance of 0.

All the immediate neighbors of A have a distance of 1.

Neighbors of neighbors of A, if not already encountered, have a distance of 2, and so on.

Nodes we never reach will have distance “Infinity”.

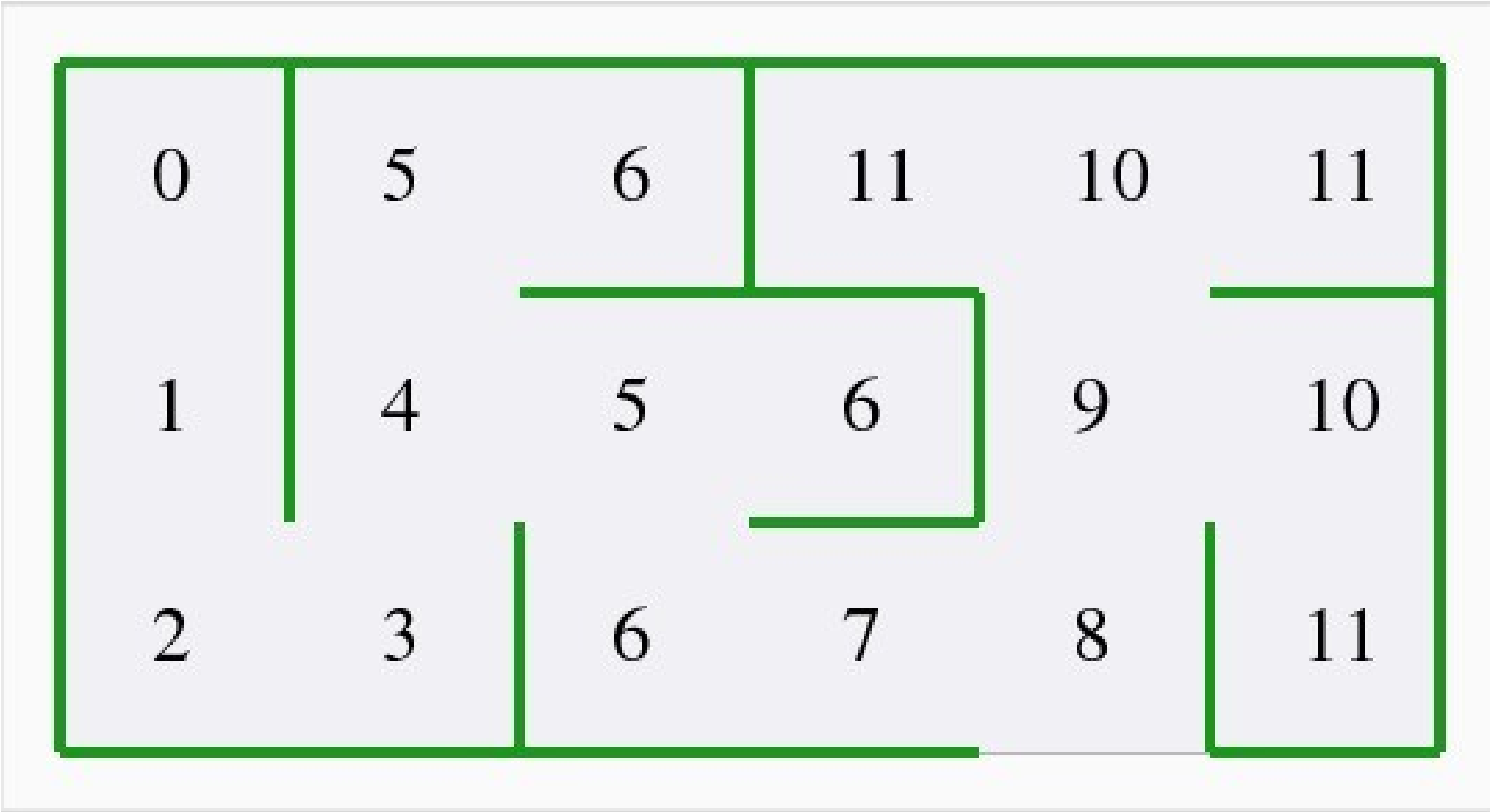
The Museum Map



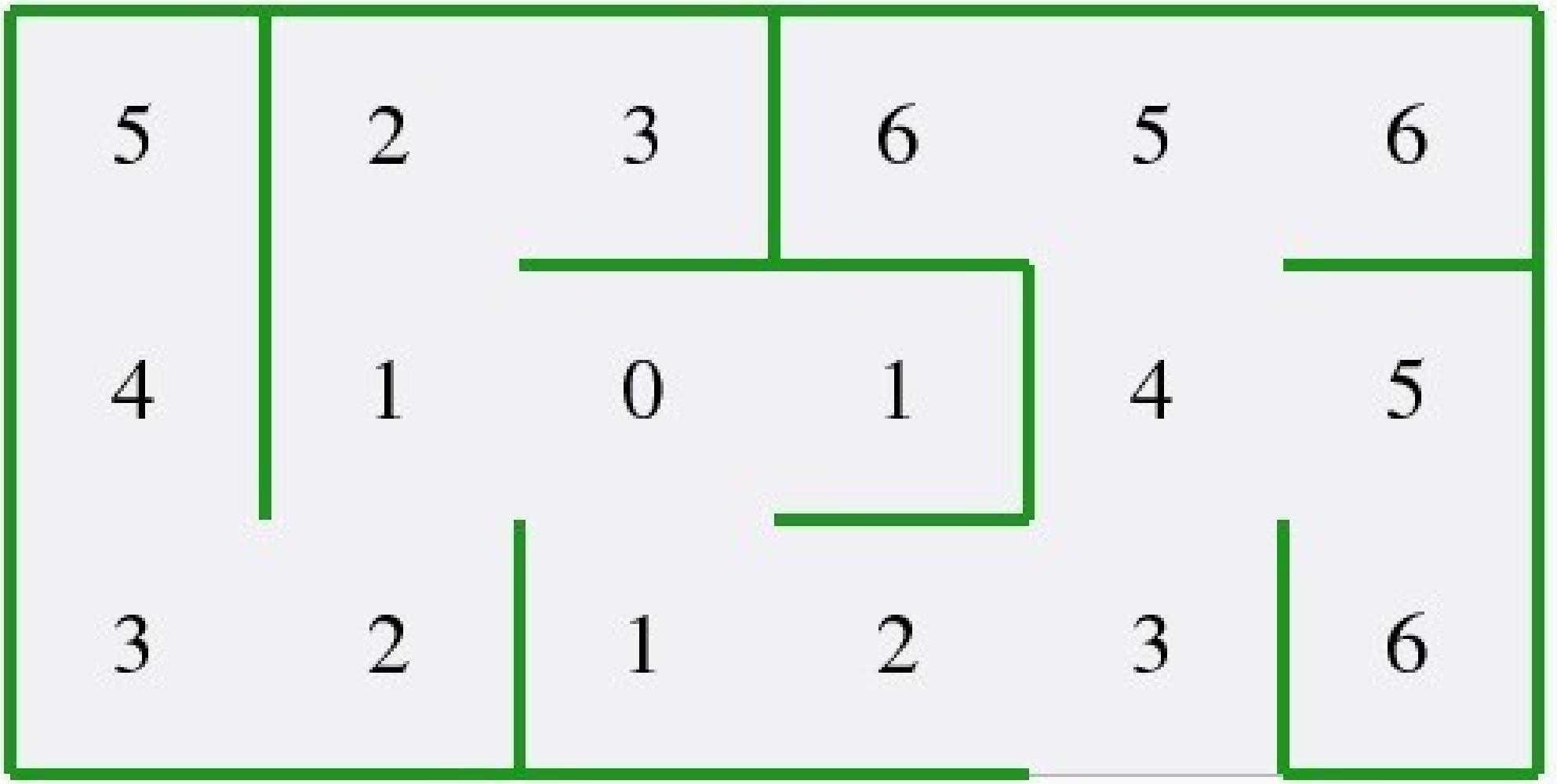
distance_from_a.m

```
function d = distance_from_a ( adj, a )

  [ n, n ] = size ( adj );
  d(1:n,1) = Inf;               %  <- Iniialize distance to “Infinity”
  distance = 0;
  d(a) = distance;
  new = [ a ];

  while ( 0 < length ( new ) )       %  <- Search for next layer of neighbors

    old = new;
    new = [];
    distance = distance + 1;

    for k = 1 : length ( old )
      j = old(k);
      for i = 1 : n

        if ( adj(i,j) == 1 && d(i) == Inf )
            d(i) = distance;
            new = [ new, i ];
          end

        end
      end
    end
  end
return
end
```

distance_from_a (Adj, 1)



distance_from_a (Adj, 9)

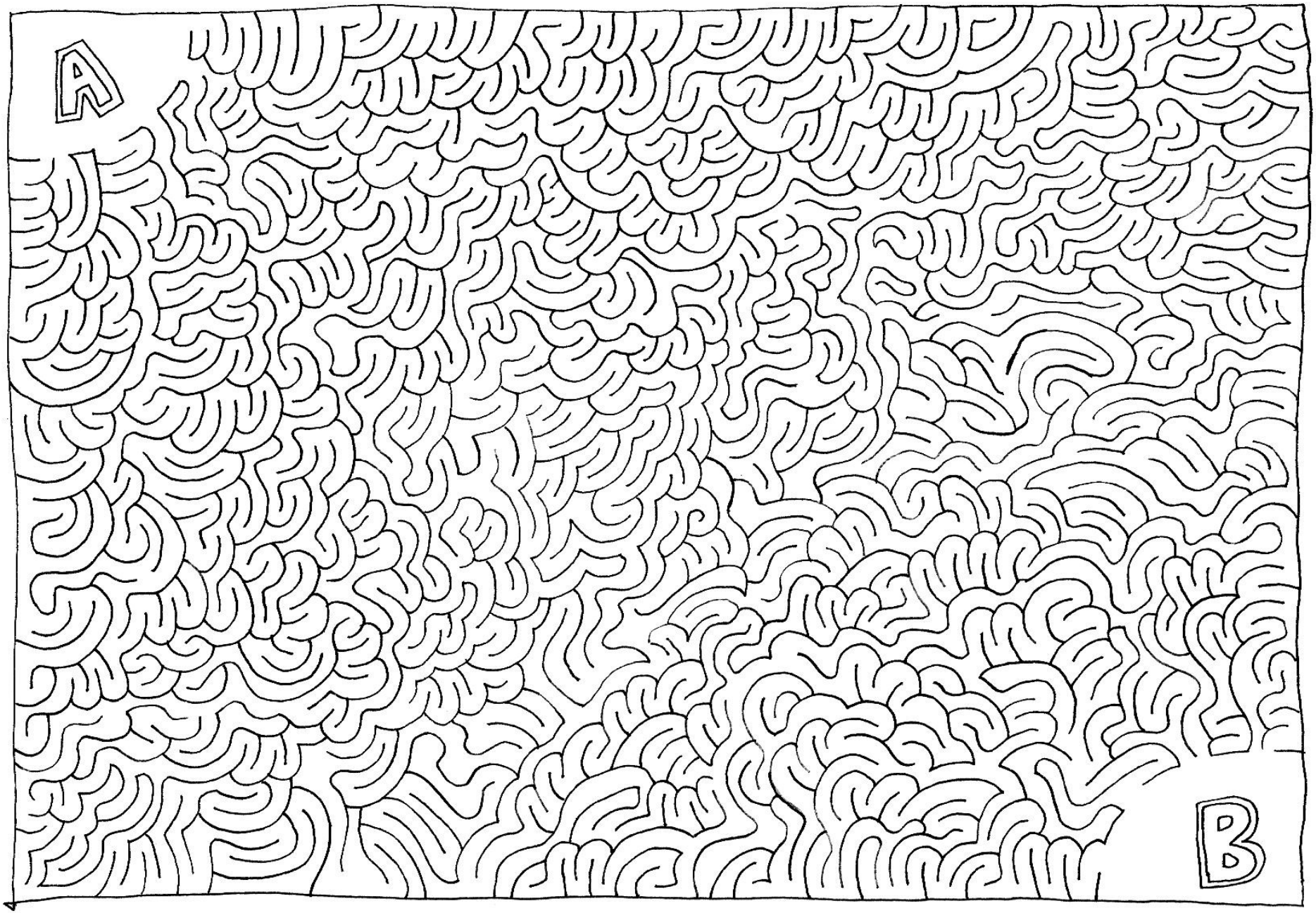


Breadth First Search

The “flooding” idea has now allowed us to determine whether a graph is connected, and how many steps it takes to reach any node from a given starting point.

This idea is very important in mathematical graph theory, and in computer science. It is known as “breadth first search”, (BFS) because it starts at a single point, and explores the graph by looking at successively further layers of nodes.

Can You Get From A to B?



A to B?

If a graph is connected, then we already know that we can get from any node to any other node, so we could consider running a code like graph_connected().

If this returns FALSE, however, we really only know that some nodes can't reach other nodes.

And if it returns TRUE, then although we know A and B have a connecting path, we don't know what it is.

So graph_connected() does not really answer our problem sufficiently.

A to B?

Again, let's consider using the "flood" method. We will start at node A, and find all the neighbors, then the neighbor's neighbors and so on.

We will stop if:

1) we encounter B, because now we know they are connected, or

2) if we never encounter B, running out of neighbors. So there is no path.

This method gives us a reliable TRUE/FALSE answer, although not an actual path.

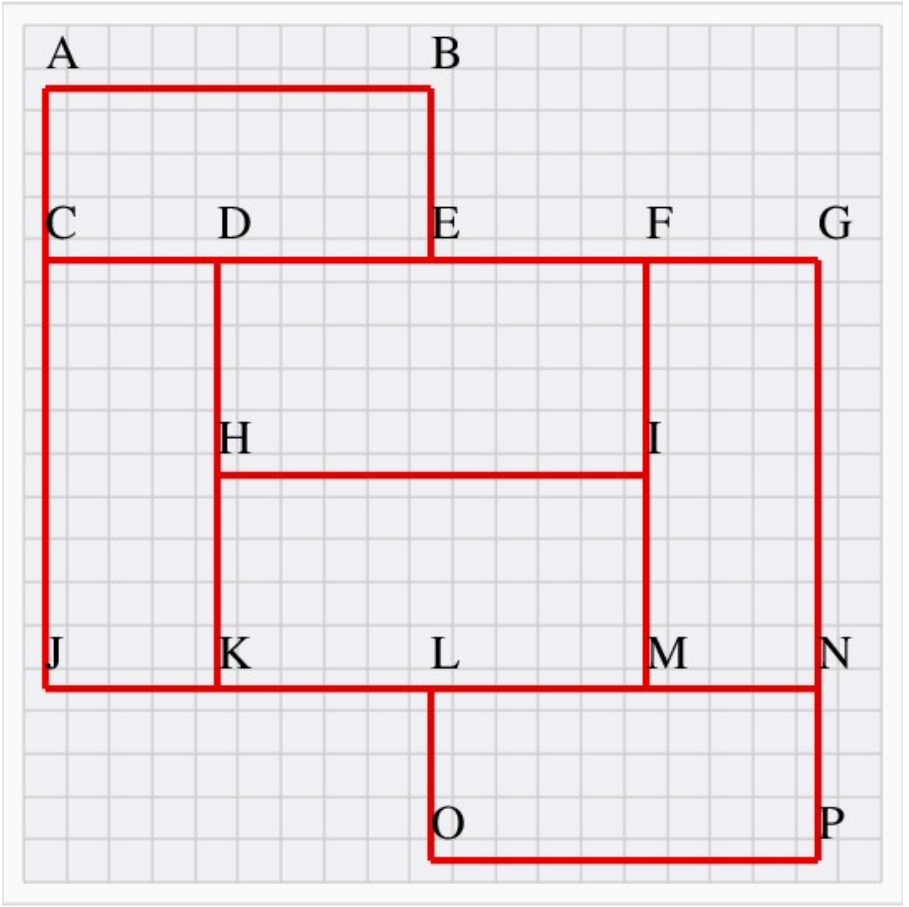
A_TO_B in Words

```
function value =  a_to_b ( adj, a, b )
%
%% A_TO_B is true if there is a path from node A to node B.
%
%  VALUE = FALSE;                                  % Assume there is no path
%  WET(I) will be TRUE if node I got wet.           %  WET(A) = TRUE.
%  WET_NUM counts how many nodes are wet.  %  WET_NUM = 1;
%
%  OLD holds the nodes that got wet on the last step.
%  NEW lists the nodes that just got wet on this step.    % NEW = [ A ];
%
%  If there were NEW nodes on the previous step, make them OLD,
%  set NEW to the empty list and take another step:
%
%     For each node J in the OLD list
%
%        if node B is a neighbor, VALUE = TRUE, and RETURN
%
%        if node I is a neighbor node that is DRY
%        then node I is WET, node I is added to the NEW list, and WET_NUM increases by 1.
%
```

a_to_b.m

```
function value = a_to_b ( adj, a, b )

  value = false;
  [ n, n ] = size ( adj );
  reach = zeros(1,n);
  reach(a) = true;
  new = [ a ];

  while ( 0 < length ( new ) )

    old = new;
    new = [];

    for k = 1 : length ( old )
      j = old(k);
      for i = 1 : n
        if ( adj(i,j) == 1 && ~ reach(i) )
          reach(i) = true;
          new = [ new, i ];

          if ( i == b )
            value = true;
            return
          end

        end; end; end; end; return; end
```

Tremaux Graph: Path from A to P?



a_to_b (adj, 1, 16) returns TRUE

Is There a Path from A to B?

```
adj = disconnected_adj ( );
value = a_to_b ( adj, 1, 5 );
value = a_to_b ( adj, 1, 8 );

adj = tremaux_adj ( );
value = a_to_b ( adj, 4, 12 );
value = a_to_b ( adj, 10, 1 );

adj = risk_adj ( );
value = a_to_b ( adj, 42, 10 );
```

List the path from A to B

It turns out to be easy to get the path from A to B. We are at node J, looking for a “dry” neighbor I and when we find one, we set REACH(I) to TRUE. Instead, we can set REACH(I) to J.

This tells us that if we are at node I, we can get closer to the starting node by moving to node J.

If we move to node J, we can look at REACH(J) and get one more step closer, and so on.

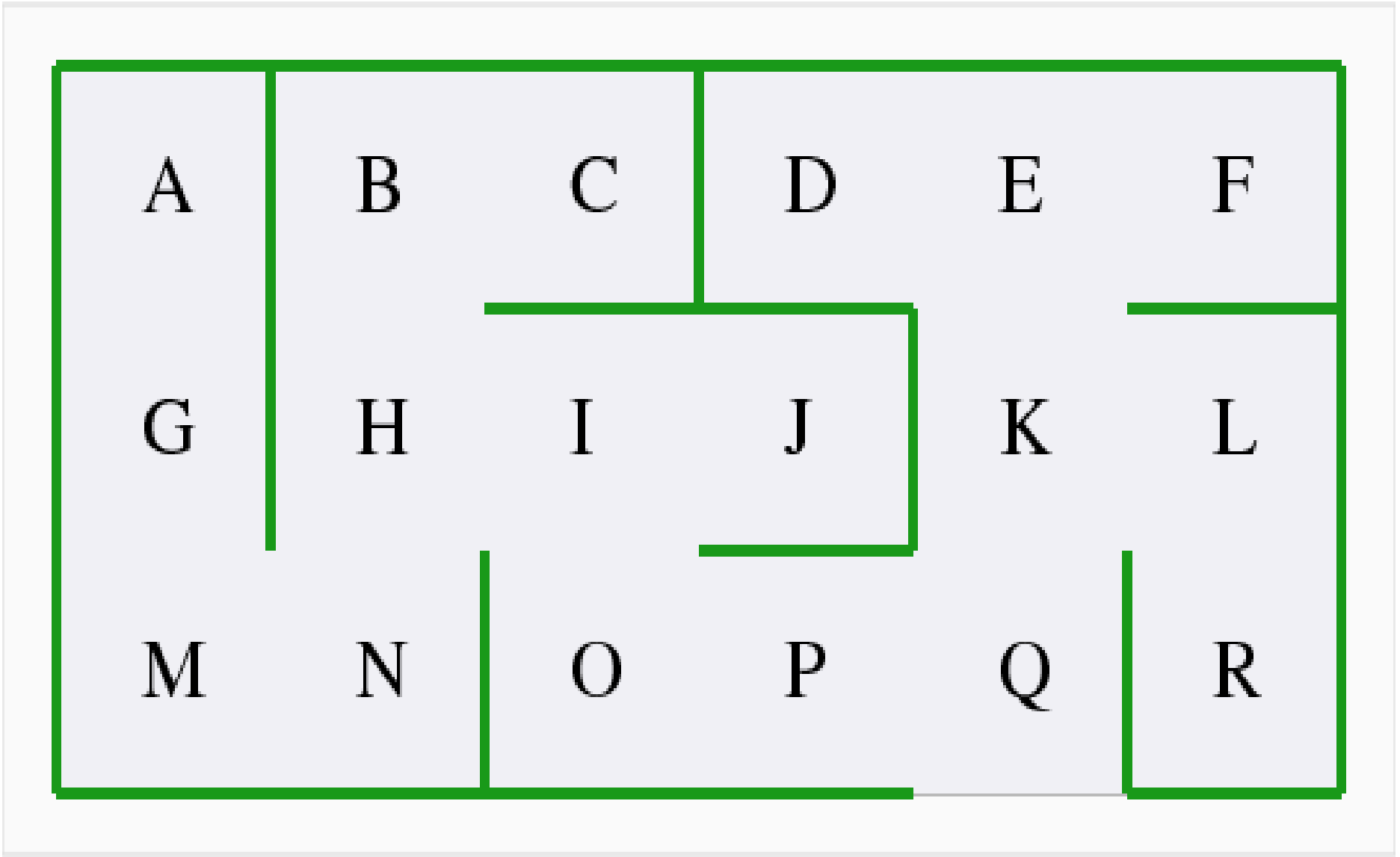
A_TO_B_PATH in Words

```
function path =  a_to_b_path ( adj, a, b )
%
%% A_TO_B_PATH lists nodes in a path from A to B.
%
%  BACK(I) points to a node closer to A.
%  BACK(A) = -1
%
%  OLD holds the nodes that we encountered on the last step.
%  NEW lists the nodes that we just touched on this step.         <- Initially NEW = [ A ];
%
%  If there were NEW nodes on the previous step, make them OLD,
%  set NEW to the empty list and take another step:
%
%     For each node J in the OLD list
%
%        if node I is a neighbor and BACK(I) = -1, then BACK(I) = J
%        if this is actually node B, we are done.
%        otherwise  add node I to the NEW list.
%
%  PATH = []
%  node = B;
%  while (true)
%    PATH = [ node, PATH ]
%   node = BACK(node)
```

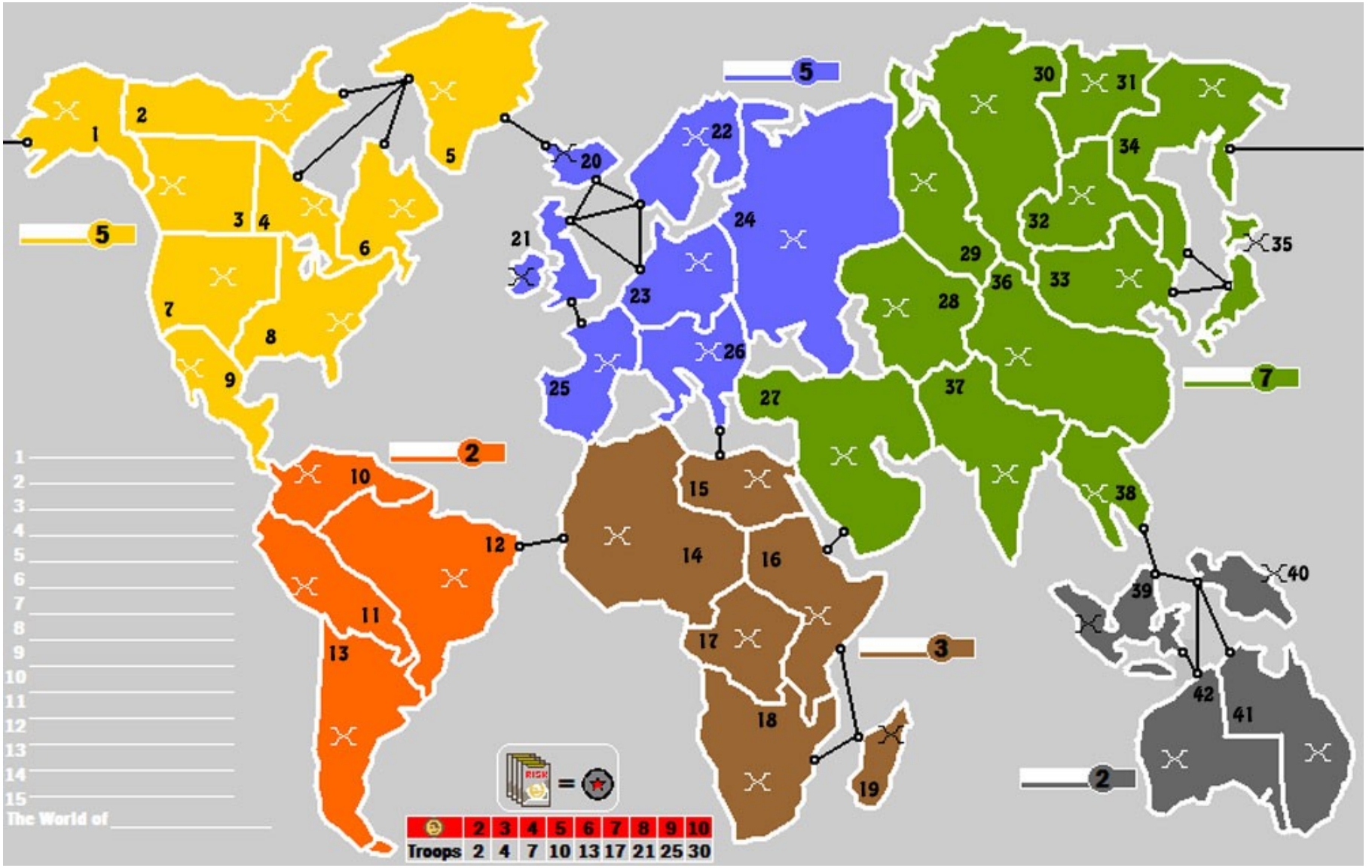
Path Examples

```
adj = museum_adj(); label = museum_label();
path = a_to_b_path ( adj, 17, 1 );
label(path);
  'QPOIHNMGA'


adj = risk_adj();  label = risk_label();
path = a_to_b_path(adj,42,9);
label(path);
    "Eastern Australia"
    "Indonesia"
    "Siam"
    "China"
    "Mongolia"
    "Kamchatka"
    "Alaska"
    "Alberta"
    "Western US"
    "Central America"
```

The Museum Map



RISK Game Board Numbered



Search a Graph

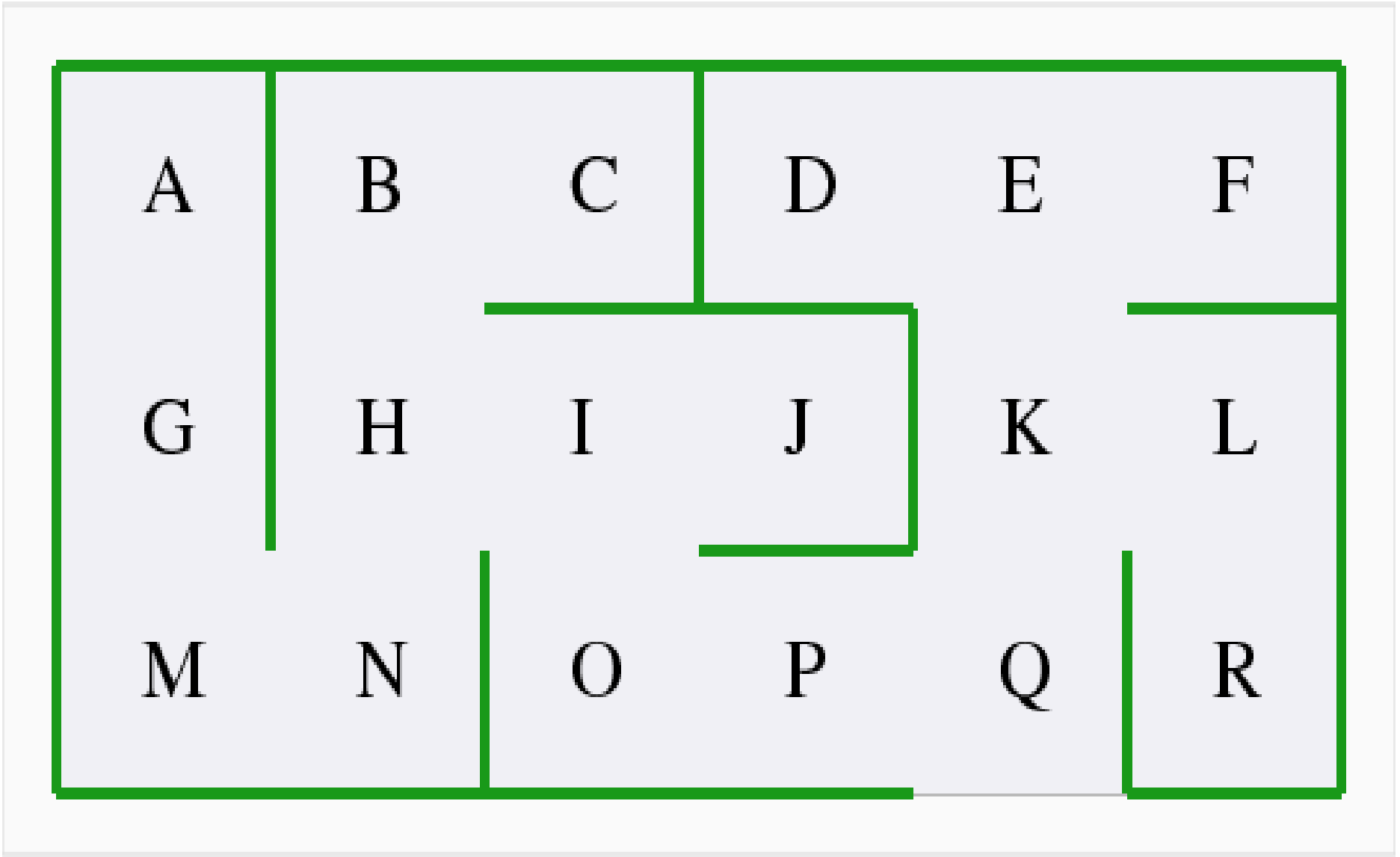
Search the Graph

A graph is a sort of simplified map, and a map can be used to search for things.

Suppose we have a map of the rooms of a museum, and we want to organize a search of the rooms until we find the room that contains a particular painting.

Instead of wandering, we want to plan a trip that is guaranteed to visit every room (if necessary) until we find the painting.

The Museum Map



Museum Adjacency

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
A	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.
B	.	.	1	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.
C	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
D	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.
E	.	.	.	1	.	1	.	.	.	.	1	.	.	.	.	.	.	.
F	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.
G	1	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.
H	.	1	.	.	.	.	.	.	1	.	.	.	.	1	.	.	.	.
I	.	.	.	.	.	.	.	1	.	i	.	.	.	.	1	.	.	.
J	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.
K	.	.	.	.	1	.	.	.	.	.	1	.	.	.	.	1	.	
L	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	1
M	.	.	.	.	.	.	1	.	.	.	.	.	.	1	.	.	.	.
N	.	.	.	.	.	.	.	1	.	.	.	.	1	.	.	.	.	.
O	.	.	.	.	.	.	.	.	1	.	.	.	.	.	1	.	.	
P	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	1	.
Q	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	1	.	.
R	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.

The Museum Problem

For the museum problem, suppose we start at node "Q", and we need a plan to visit every room in the museum until we find the room containing a specific painting.

We can assume that this is actually the room labeled "D", or numerically "4", but this fact is not known to us until we reach that room.

We know how to find a path from "Q" to every room, but that's not the best way to think about the problem.

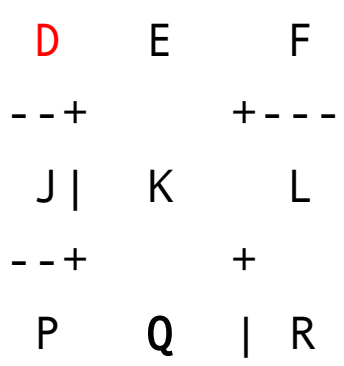
Instead, we imagine starting at Q and, as long as possible, always moving to a neighboring unvisited room, with three rules:

1) If you have a choice of rooms to visit, remember the ones you didn't choose so you can come back to them later;

2) If you reach a dead-end (no neighboring unvisited rooms), reverse your path until you reach a room that still has unvisited neighbors.

3) If you reach the room that contains the desired object (the painting) then you are done.

A Miniature Museum Search



Q: options P and K. Choose P, remember K.
P: no options, return to Q.
Q: option is K.
K: options L and E. Choose L, remember E.
L: only option is R:
R: no options. Return to L.
L: no options, return to K.
K: option E.
E: options D and F. Choose D.
D: found the painting!

Programming

Our program needs the adjacency matrix.

It needs to know the current location.

It needs to know the path from "Q" to the current location.

It needs to know if a room was already visited.