Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

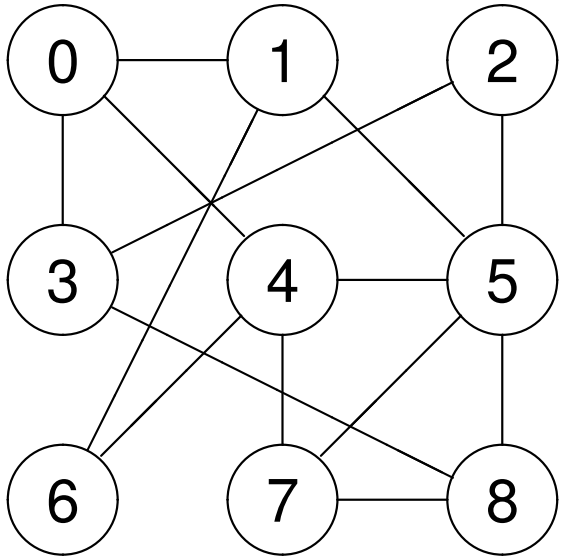
MCS 360 Lecture 39
Introduction to Data Structures
Jan Verschelde, 29 November 2017

Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

visiting each vertex in a graph

Consider the graph:



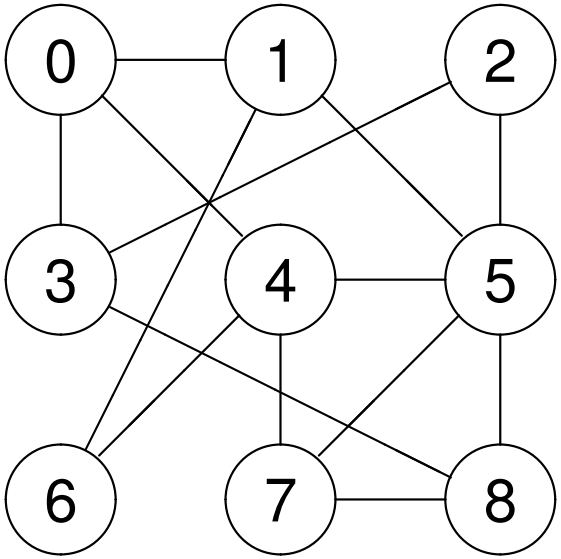
Problem: visit each vertex in a systematic order.

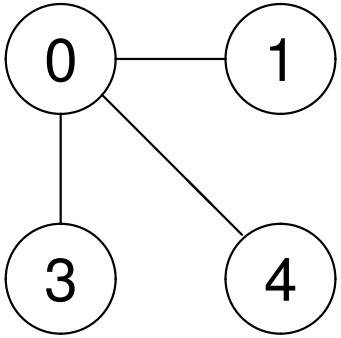
Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

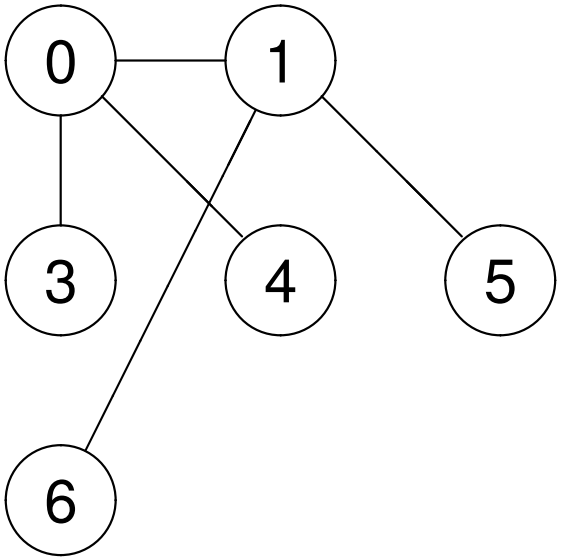
breadth-first search

We start at vertex 0.



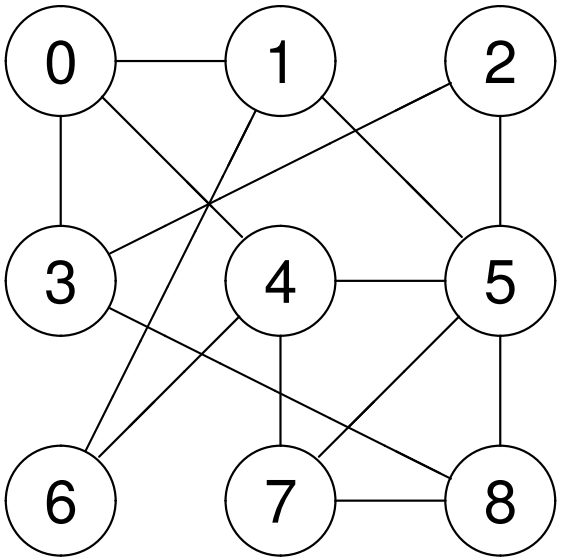


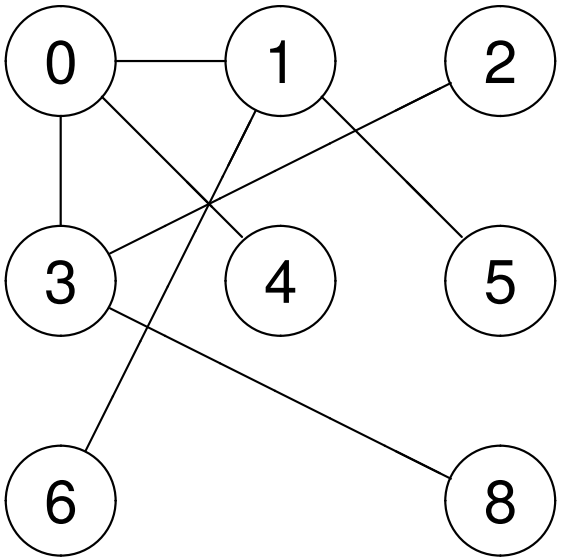
at 0: visit 1, 3, 4



at 1: visit 5 and 6

breadth-first search continued



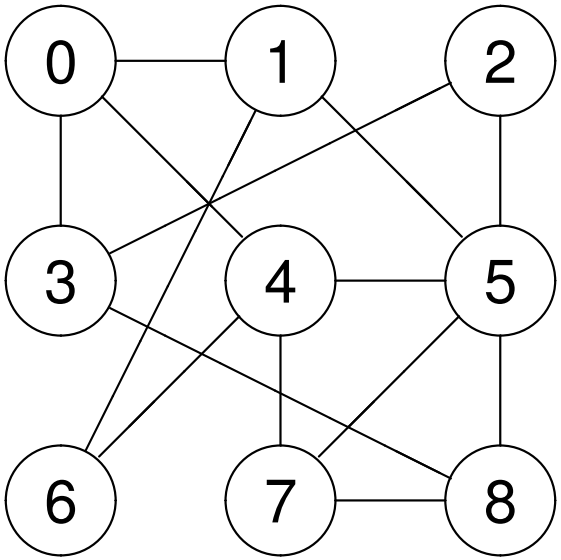


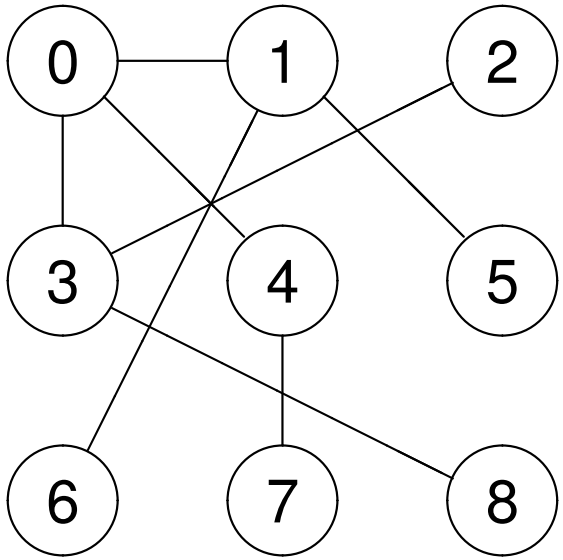
at 3: visit 2 and 8

Every vertex is visited only once.

Although there is an edge from 4 to 5, the breadth-first search will not visit this edge because 5 has already been visited.

breadth-first search continued





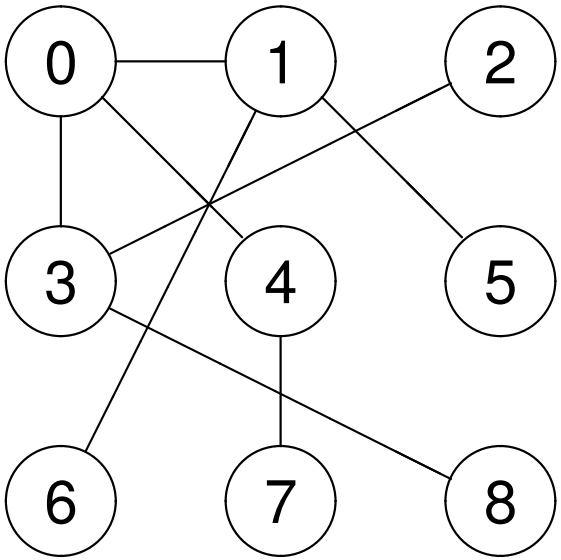
at 4: visit 7

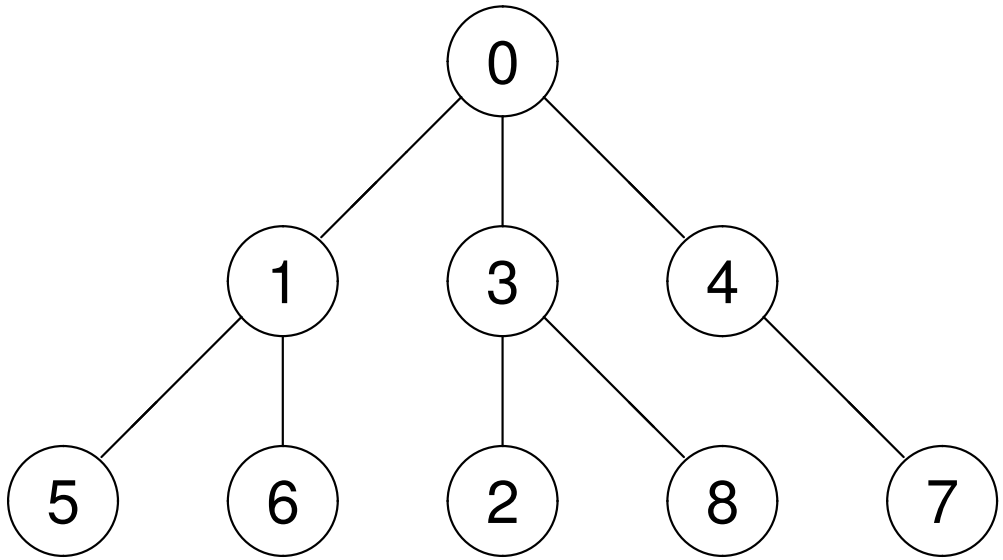
Observe that the graph on the right is a tree.

the breadth-first tree

At the left is the result of the breadth-first search.

At the right is the *breadth-first tree*.





The *breadth-first tree of a graph* contains all the vertices of the graph and the edges visited in the breadth-first search.

the breadth-first search algorithm

Input: $G = (V, E)$.

Select the start vertex $v \in V$.

Initialize the queue Q with v: $Q.\text{push}(v)$.

while not $Q.\text{empty}()$ do
    $u = Q.\text{pop}()$;
    visit u;
    for all v adjacent to u do
        if v is not visited then
            if $v \notin Q$ then
                $Q.\text{push}(v)$.

running the breadth-first search algorithm

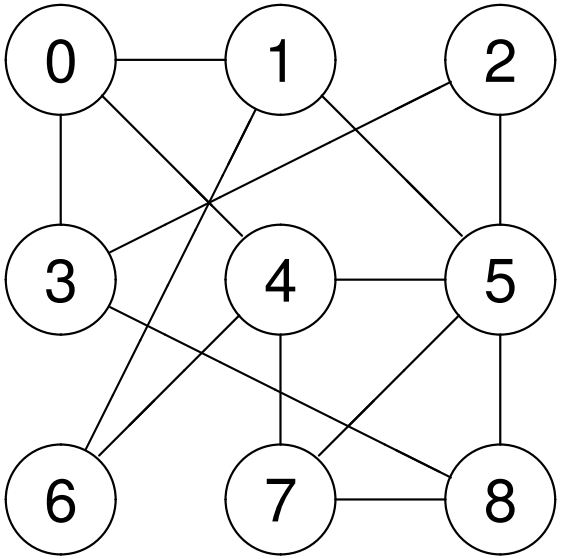
$$V = \{0,1,2,3,4,5,6,7,8\}$$
$$E = \{(0,1),(0,3),(0,4),(1,5),(1,6),(2,3),(2,5),(3,8),$$
$$(4,5),(4,6),(4,7),(5,7),(5,8),(7,8)\}$$

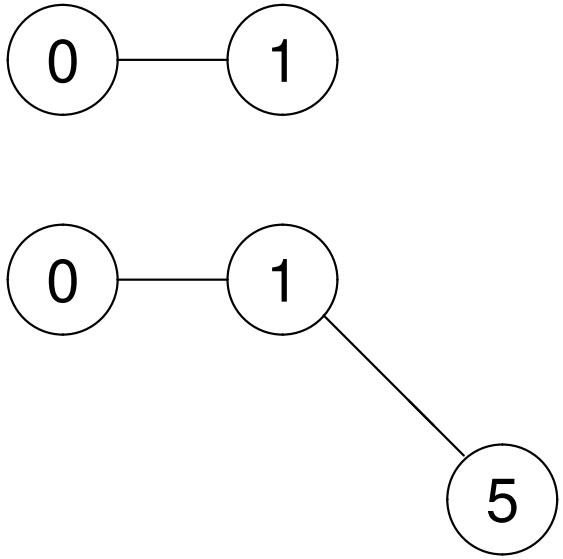
$Q = 0$
pop 0, visit 0, $Q = 1,3,4$
pop 1, visit 1, $Q = 3,4,5,6$
pop 3, visit 3, $Q = 4,5,6,2,8$
pop 4, visit 4, $Q = 5,6,2,8,7$
pop 5, visit 5, $Q = 6,2,8,7$
pop 6, visit 6, $Q = 2,8,7$
pop 2, visit 2, $Q = 8,7$
pop 8, visit 8, $Q = 8$
pop 7, visit 7, $Q = \emptyset$

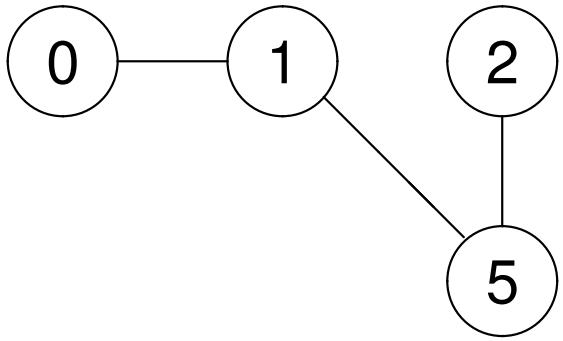
Graph Traversals

depth-first search

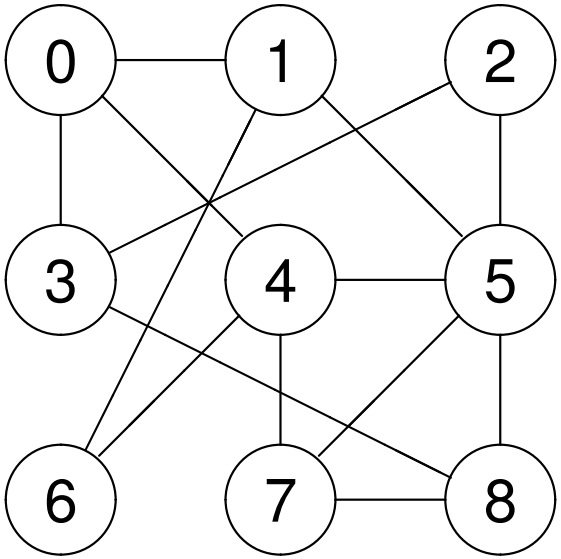
We start at vertex 0.

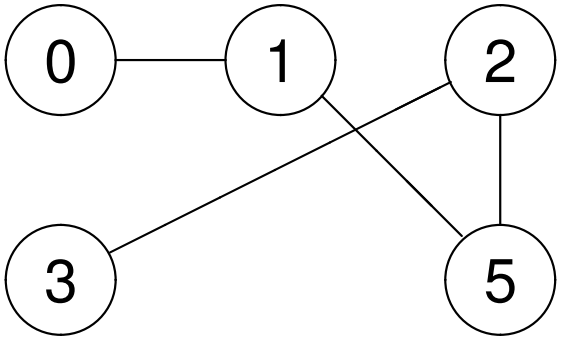


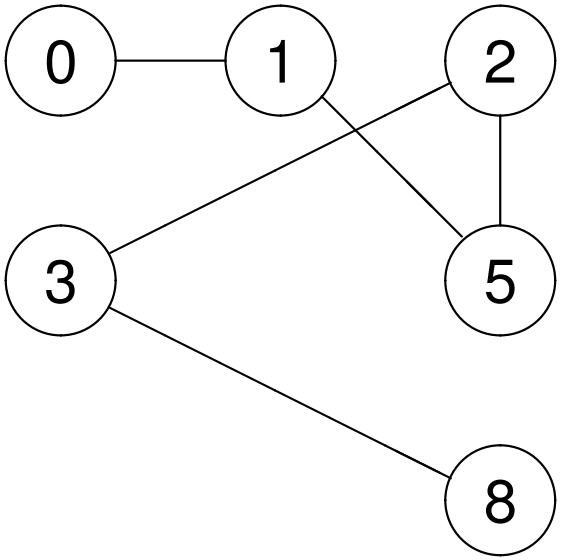




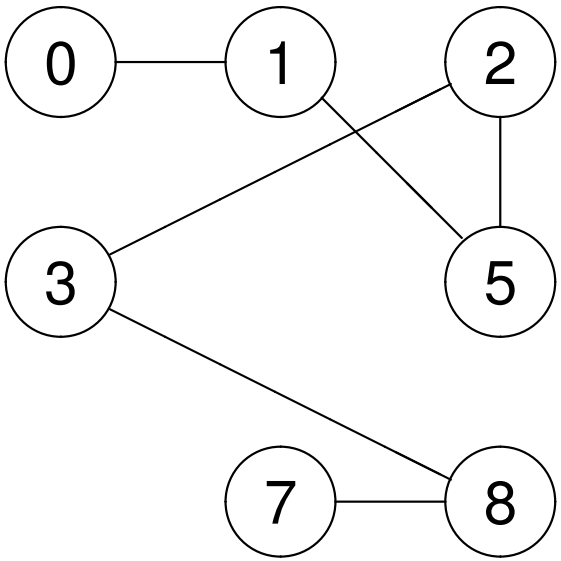
depth-first search continued

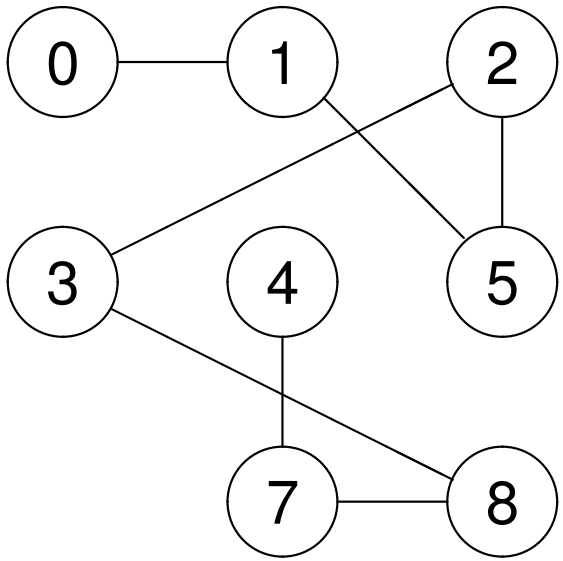




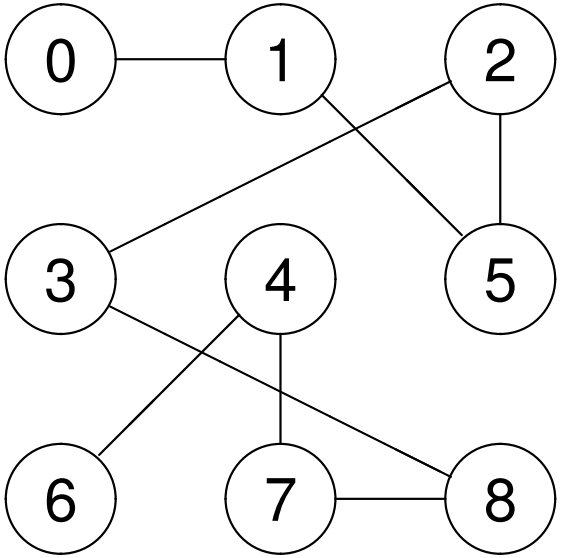


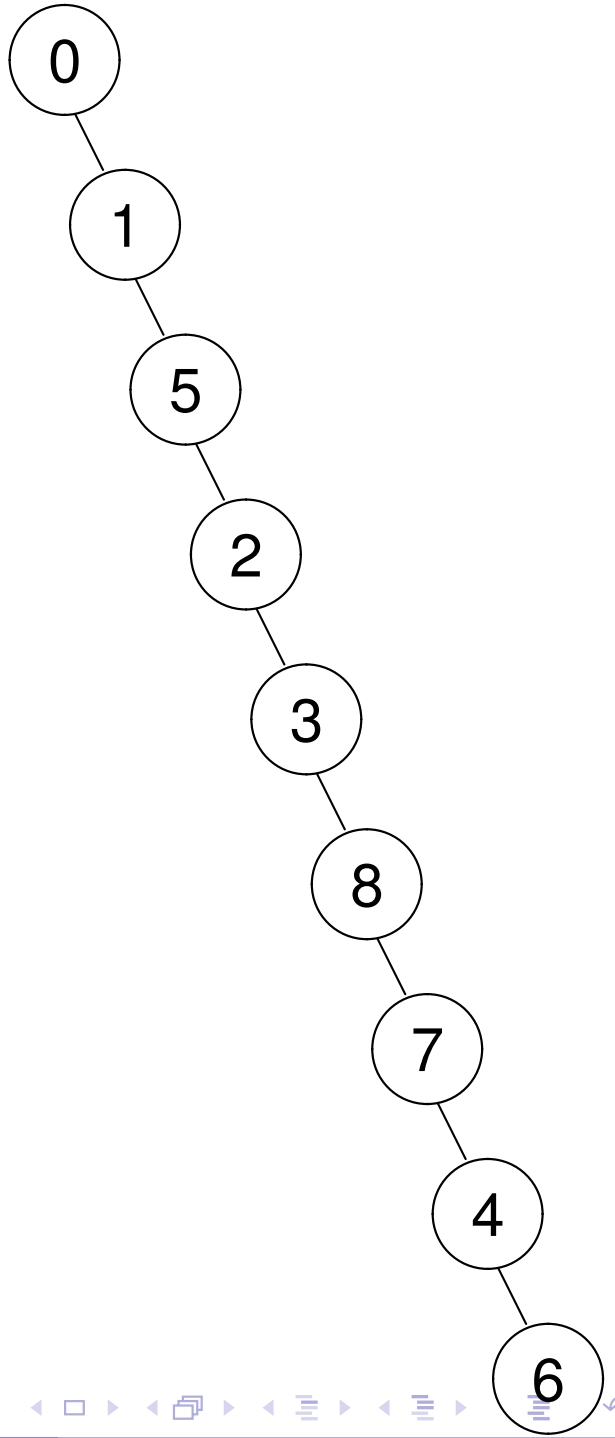
depth-first search continued





the depth-first search tree





The *depth-first search tree of a graph* contains all the vertices of the graph and the edges visited in the depth-first search.

the depth-first search algorithm

algorithm VISIT($V, E, u,$ Visited)
// Visits all vertices in the graph (V, E), starting at u.
// Visited collects the vertices visited.

visit u;
Visited.push(u);
for all $v \in V$, v is adjacent to u do
    if $v \notin$ Visited then
        VISIT($V, E, v,$ Visited).

We call algorithm VISIT with vertex 0 for u
and an empty list for Visited.

running the depth-first search algorithm

$$V = \{0,1,2,3,4,5,6,7,8\}$$
$$E = \{(0,1),(0,3),(0,4),(1,5),(1,6),(2,3),(2,5),(3,8),$$
$$(4,5),(4,6),(4,7),(5,7),(5,8),(7,8)\}$$

visit 0, VISIT($V, E, 1, \{0\}$)
  visit 1, VISIT($V, E, 5, \{0,1\}$)
    visit 5, VISIT($V, E, 2, \{0,1,5\}$)
      visit 2, VISIT($V, E, 3, \{0,1,5,2\}$)
        visit 3, VISIT($V, E, 8, \{0,1,5,2,3\}$)
          visit 8, VISIT($V, E, 7, \{0,1,5,2,3,8\}$)
            visit 7, VISIT($V, E, 4, \{0,1,5,2,3,8,7\}$)
              visit 4, Visited $= \{0,1,5,2,3,8,7,4\}$
                visit 6, Visited $= \{0,1,5,2,3,8,7,4,6\}$

The tree of recursive calls is the depth-first search tree.

Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

depth-first search on a random matrix

```
Give the number of vertices : 10
A random Boolean matrix :
 1 0 0 0 1 1 0 0 0 0
 1 1 0 1 0 0 1 0 0 0
 1 0 1 0 1 0 1 1 1 0
 1 1 0 0 0 0 1 0 0 0
 1 0 1 1 1 0 1 0 1 1
 0 1 1 0 1 1 0 0 0 0
 1 1 1 1 1 0 0 1 1 1
 1 1 0 0 0 0 1 1 1 1
 1 0 1 0 1 0 0 0 1 0
 1 0 0 1 0 0 1 0 0 1

The depth-first search tree :
-> 0
   -> 4
      -> 2
         -> 6
            -> 1
               -> 3
            -> 7
               -> 8
               -> 9
      -> 5

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

prototype of the depth-first search function

```
void depth_first_search
  ( vector< vector<bool> >& mat, size_t vertex,
    list<size_t>& visited, bool verbose );
/*
    Applies depth-first search to visit all vertices.
    The vertices are collected in the list visited.

    ON ENTRY :
      mat          adjacency matrix of the graph;
      vertex       start vertex;
      visited      should be an empty list;
      verbose      if true, vertices are written.

    ON RETURN :
      visited      list of visited vertices. */
```

definition of the depth-first search function

```
void depth_first_search
 ( vector< vector<bool> >& mat, size_t vertex,
   list<size_t>& visited, bool verbose )
{
   if(verbose) cout << " -> " << vertex;
   visited.push_back(vertex);

   for(size_t neighbor=0; neighbor<mat.size(); neighbor++)
      if(mat[vertex][neighbor])
      {
         list<size_t>::iterator it;
         it = find(visited.begin(), visited.end(), neighbor);

         if(it == visited.end())
            depth_first_search(mat,neighbor,visited,verbose);
      }
}
```

Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

breadth-first search on a random matrix

```
A random Boolean matrix :
 1 0 0 0 1 1 0 0 0 0
 1 1 0 1 0 0 1 0 0 0
 1 0 1 0 1 0 1 1 1 0
 1 1 0 0 0 0 1 0 0 0
 1 0 1 1 1 0 1 0 1 1
 0 1 1 0 1 1 0 0 0 0
 1 1 1 1 1 0 0 1 1 1
 1 1 0 0 0 0 1 1 1 1
 1 0 1 0 1 0 0 0 1 0
 1 0 0 1 0 0 1 0 0 1

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

```
The breadth-first search tree :
-> 0
   -> 4
      -> 2
         -> 7
      -> 3
      -> 6
      -> 8
      -> 9
   -> 5
      -> 1
```

prototype of the breadth-first search function

```
void breadth_first_search
 ( vector< vector<bool> >& mat, size_t vertex,
   bool verbose );
/*
   Applies depth-first search to visit all vertices.

   ON ENTRY :
     mat          adjacency matrix of the graph;
     vertex       start vertex;
     verbose      if true, vertices are written. */
```

definition of the breadth-first search function

```
void breadth_first_search
  ( vector< vector<bool> >& mat, size_t vertex,
    bool verbose )
{
   list<size_t> queue;
   queue.push_back(vertex);
   list<size_t> visited;

   while(queue.size() > 0)
   {
      size_t first = queue.front();
      queue.pop_front();
      if(verbose) cout << " -> " << first;
      visited.push_back(first);
```

definition continued

```
for(size_t neighbor=0; neighbor<mat.size(); neighbor++)
{
   if(mat[first][neighbor])
   {
      list<size_t>::iterator itv;
      itv = find(visited.begin(), visited.end(), neighbor);
      if(itv == visited.end())
      {
         list<size_t>::iterator itq;
         itq = find(queue.begin(), queue.end(), neighbor);
         if(itq == queue.end()) queue.push_back(neighbor);
      }
   }
}
```

Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

finding a path in a maze

Consider a grid of tiles, of two types:

1. a tile can be free; or
2. a tile can be blocked.

The grid represents a maze. Tiles are blocked by walls.
We hop from tile to tile horizontally or vertically, not diagonally.

Problem: in a grid with n rows and m columns,
find a path from tile $(0, 0)$ to $(n-1, m-1)$.

Solution:

- Apply the depth-first search algorithm to find a path.
- If blocked at a dead end, *backtrack* to the previous fork.

the definition of the grid

The grid is represented by an n-by-m integer matrix A.
There are three types of elements:

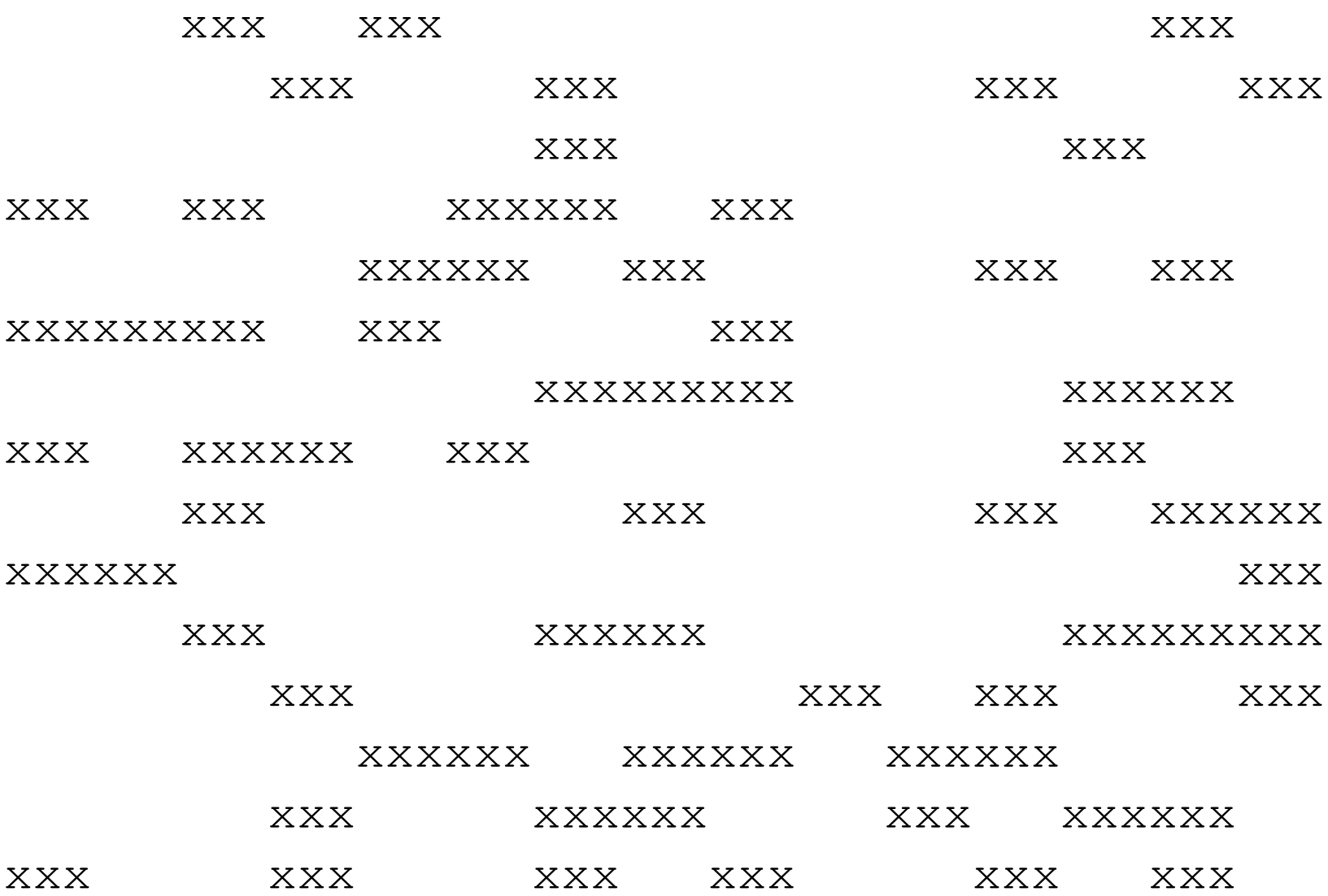
1. $A_{i,j} = -1$: tile (i, j) is blocked by a wall;
2. $A_{i,j} = 0$: tile (i, j) is free;
3. $A_{i,j} = k > 0$: tile (i, j) has been visited at step k.

Whether or not a tile is free is determined by a loaded coin:

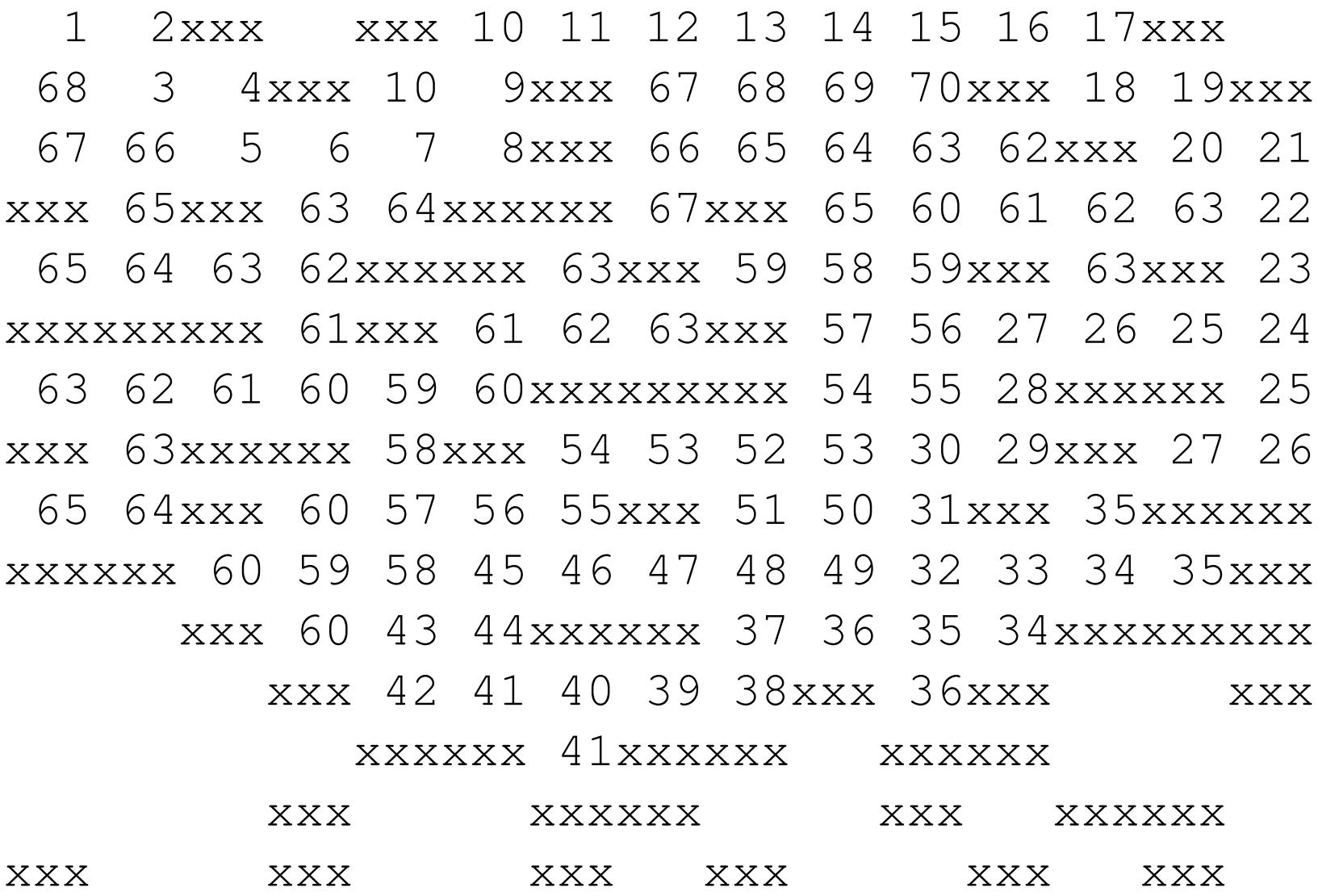
```
int loaded_coin ( double p )
{
   double r = ((double) rand())/RAND_MAX;

   return (r <= p ? 0 : 1);
}
```

where `p` is the probability of a zero.
If `p = 0.75`, then 75% of the tiles will be free.

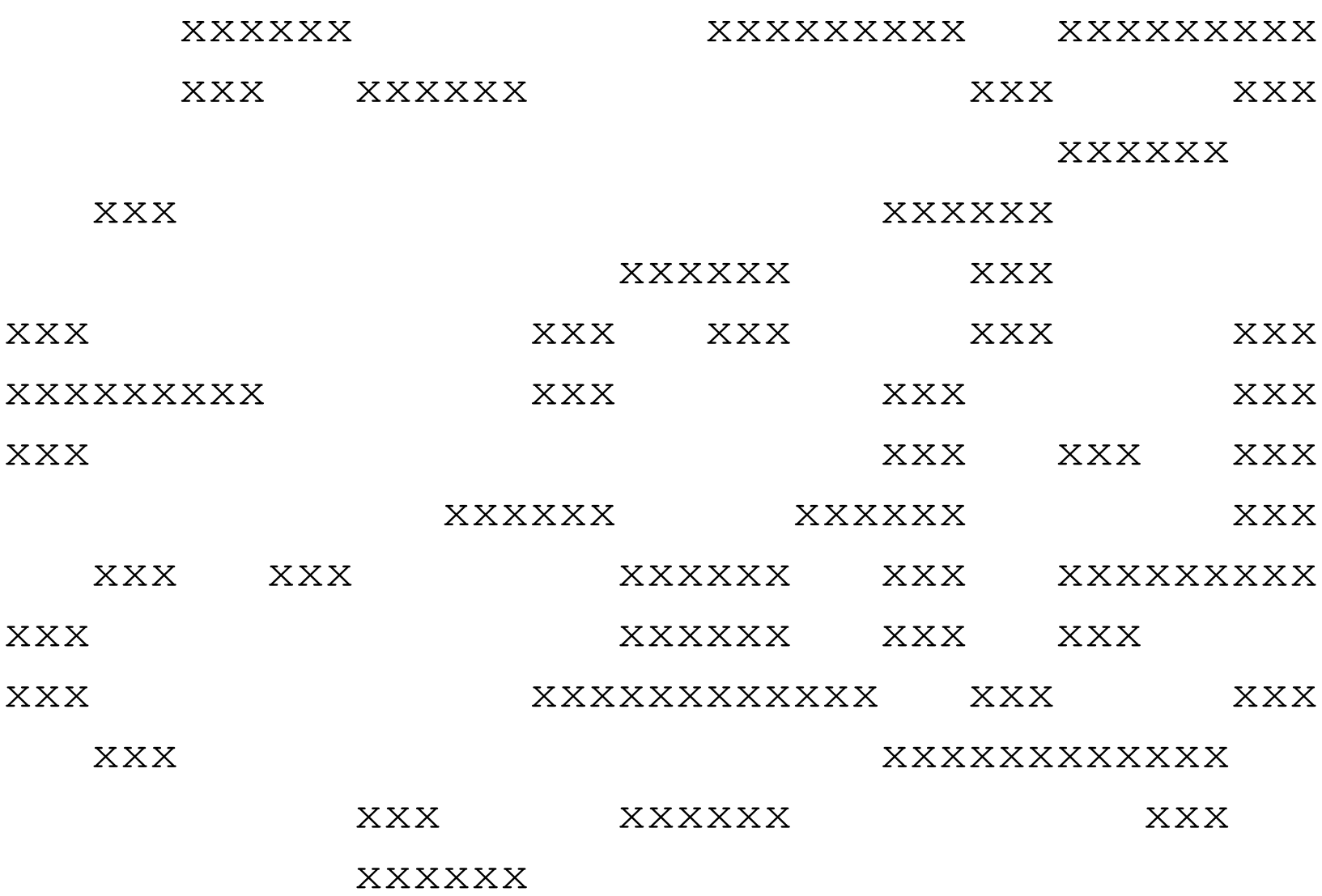
a random 15-by-15 maze, $p = 0.7$



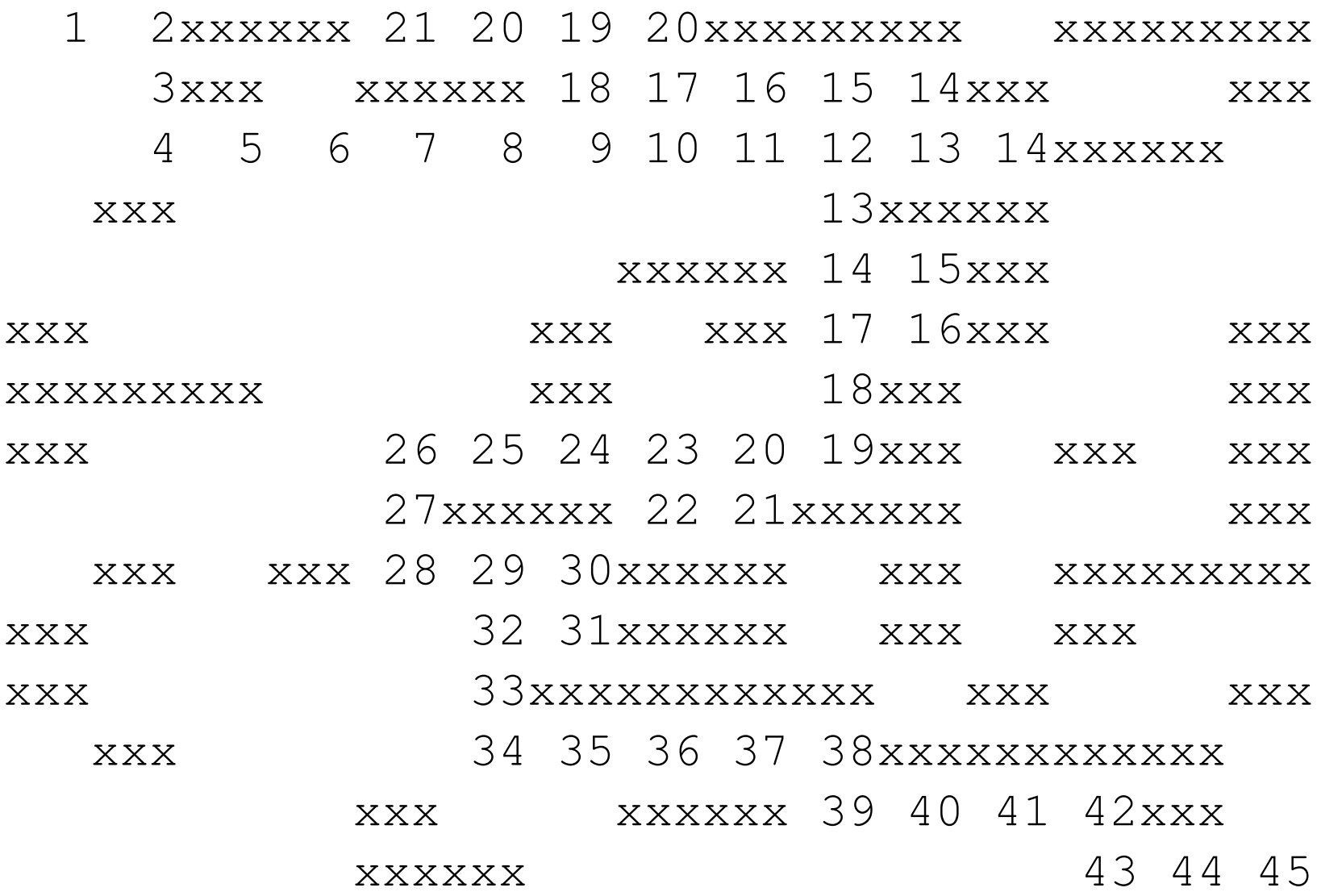
no path found



another random 15-by-15 maze, $p = 0.7$



found a path


1 2 21 20 19 20 14
3 18 17 16 15 14
4 5 6 7 8 9 10 11 12 13 14
13
14 15
17 16
18
26 25 24 23 20 19
27 22 21
28 29 30
32 31
33
34 35 36 37 38
39 40 41 42
43 44 45

Graph Traversals

1. Graph Traversals
   - visiting each vertex
   - breadth-first search
   - depth-first search
2. Traversing Random Graphs
   - code for depth-first search
   - code for breadth-first search
3. Application: Backtracking
   - finding a path in a maze
   - applying depth-first search

prototype of the search function

```
bool search
  ( vector< vector<int> >& mat, size_t count,
    size_t row, size_t col );
/*
    Applies depth-first search to find a path
    from the first row to the last column in the grid.
    The parameter have the following meaning:

    mat   : a grid of tiles, free, block, or visited;
    count : counts the number of tiles visited;
    row   : row of the next tile;
    col   : column of the next tile.

    Visited tiles are marked in mat by the counter.
    Returns true if a path is found, false otherwise.
 */
```

definition of the base cases

```
bool search
  ( vector< vector<int> >& mat,
    size_t count, size_t row, size_t col )
{
   if(mat[row][col] != 0) // tile is not free
      return false;
   else
   {
      mat[row][col] = count+1; // visit

      const int rows = mat.size();
      const int cols = mat[0].size();

      if((row == rows-1) and (col == cols-1))
         return true;  // reached destination
      else
```

depth-first search in the general case

```
{
   bool found = false;

   if(col < cols-1)                        // go down
   found = search(mat, count+1, row, col+1);
   if(not found and (row < rows-1)) // go right
      found = search(mat, count+1, row+1, col);
   if(not found and (col > 0))          // go left
      found = search(mat, count+1, row, col-1);
   if(not found and (row > 0))          // go up
      found = search(mat, count+1, row-1, col);

   return found;
}
```

summary + exercises

Breadth-first and depth-first search lead to spanning trees.
We first visit all adjacent vertices in a breadth-first search.
Depth-first is preferable when finding a path in a maze.

Exercises:

1. We implicitly assumed that in our graphs all vertices are connected. Describe the modifications to the search algorithms to visit all vertices in a graph which has several components.
2. Write code for the breadth-first search algorithm for a graph defined by an adjacency list.
3. Write code for the depth-first search algorithm for a graph defined by an adjacency list.
4. Write code to apply breadth-first search to solve the problem of finding a path in the maze.
5. Adjust the code to compute all solutions to the problem of finding a path in a maze. The best solution is the one which visits the fewest number of tiles. Report the best solution.