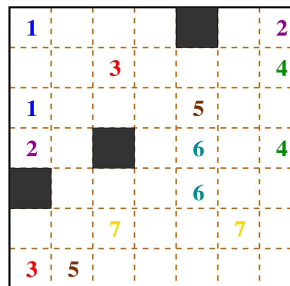
CS29003 ALGORITHMS LABORATORY
Assignment No: 7
Last Date of Submission: 09–September–2015

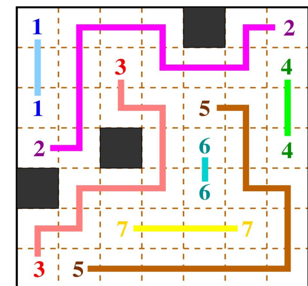
The origin of a puzzle called *Numberlink* (some people call it *Flow*) dates back to late nineteenth century. In this assignment, you implement an algorithm to prepare random puzzle instances for the user. Our variant of Numberlink is described in the next paragraph.

Consider an $n \times n$ array of squares. Some of the squares are empty, some are solid, and some non-solid squares are marked by integers 1, 2, 3, … Each integer occupies exactly two different squares on the board. The task of the player is to connect the two occurrences of each integer on the board by a <u>*simple*</u> path using horizontal and vertical movements alone. No two different paths are allowed to intersect one another. No path may include any solid square (solid squares are forbidden to appear on any path). Finally, <u>*all*</u> non-solid squares must be filled by the paths. Here follows an example of the puzzle, and its solution (the solution need not be unique, although in this example, it is).

To prepare a valid random puzzle with a given board size $n \times n$, you first generate random simple mutually non-intersecting paths on the board (see Part 3 for the details). If a few isolated squares remain outside all the generated paths, mark these isolated squares as solid (forbidden). You then supply the endpoints of the paths and the list of the solid squares as the puzzle. Thus you first generate a solution, and then work out the puzzle from the solution. The paths and the solid squares partition the $n \times n$ board. Let us use a union-find data structure to generate this partition. The data structure deals with the subsets of the set of n^2 squares on the board.



(a) A numberlink puzzle



(b) Its solution

Part 1: Write a data type to represent a node in the union-find tree. Each node represents a square on the board. It should have a parent pointer (but no child pointers), and the rank of the subtree rooted at that node. With the application in mind, we let each node store additionally the number of the path on which the node lies. For example, all squares connecting the two 1's have path number 1 (see the above picture). The path number is zero for an isolated node. Finally, we keep a flag to indicate whether the node is an endpoint of a path.

For quickly locating the node corresponding to any (i, j)-th square, we need an $n \times n$ array B of headers (node pointers). Write a function *initboard*() to initialize the array B. Initially, we have n^2 singleton trees in the union-find forest. For each (i, j), allocate memory to $B[i][j]$ to create a node for the (i, j)-th square. The rank of the node will be zero, and its parent pointer NULL. Initialize the path number and endpoint flag to zero. We will update these fields in Parts 3 and 4.

Part 2: Write the functions *findset*() and *setunion*() needed for the union-find data structure. The algorithm of Part 3 does not require path compression, so you do not need to implement it. It suffices that you make the root of the smaller-rank tree a child of the root of the larger-rank tree. If the two ranks are equal, break the tie arbitrarily.

Part 3: Write a function *addpath*() that attempts to add <u>one</u> new path to the board. The pseudocode of the algorithm is given below. Notice that a square is not already added to a path if and only if it resides in a singleton tree in the union-find forest, that is, if and only if the root of the union-find tree to which the node for the square belongs has rank zero. Each node (i, j) has four neighbors: $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$. If (i, j) is at the boundary of the board, some of these neighbors is/are not defined.

Locate squares (i, j) and (k, l) randomly on the board such that: /* see below for the details */
  (a) (i, j) and (k, l) are neighbors of one another, and
  (b) neither (i, j) nor (k, l) belongs to any path generated so far.
If no such pair of squares is found on the entire board, return FAILURE.
/* Here, (i, j) and (k, l) are the first two squares on the new path to be constructed. */
Make a union of the two union-find trees containing (i, j) and (k, l).
Repeat so long as the current path can be extended:
  Rename $(i, j) = (k, l)$.
  Locate a random neighboring square (k, l) of (i, j) such that: /* see below for the details */
    (a) (k, l) does not belong to any path generated so far (including the current one), and
    (b) the only neighbor (k, l) has on the partially constructed current path is (i, j).
  If no such neighbor (k, l) can be found, the path cannot be extended further, so break the loop.
  Otherwise, make the union of the two union-find trees to which (i, j) and (k, l) belong.
Set the endpoint flags of the two squares that are at the beginning and at the end of the new path.
Return SUCCESS.

To generate a random starting square (i, j) for a new path, we make an exhaustive search on the entire board. The search starts at a random index (i, j), and then visits the board in a row-major wrap-around fashion. That is, the search follows the sequence (i, j), $(i, j+1)$, $(i, j+2)$, …, $(i, n-1)$, $(i+1, 0)$, $(i+1, 1)$, …, $(i+1, n-1)$, …, $(n-1, n-1)$, $(0,0)$, $(0,1)$, …, $(i, j-1)$. A random neighbor (k, l) of (i, j) can be searched in one of the sequences NEWS, EWSN, WSNE, and SNEW.

Part 4: After the path-finding attempts, you have a union-find forest containing (a) trees with multiple nodes (each such tree represents a path on the board), and (b) some singleton trees (standing for the isolated solid squares). Write a function *addpathnum*() to assign a unique path number to the paths (in the sequence 1, 2, 3, …). All nodes residing in a single union-find tree must receive the same path number. Nodes residing in singleton trees are numbered zero (indeed, these are isolated nodes and their path numbers are initialized to zero in Part 1). Part 4 could have been clubbed together with Part 3. But as a separate part, this is fine too.

Part 5: Write a function *printboard*() to print the puzzle and the solution of the puzzle. The sample output demonstrates a sophisticated way of doing it. You may use a simpler format, like X for a solid square, . for a blank square (only in the puzzle), and an integer for an endpoint (puzzle) or a filled square (solution). This output does not explicitly mention the flows of the paths. Our path-generation algorithm guarantees that all paths are simple, so it is fairly straightforward to trace each path uniquely in the solution from its two endpoints supplied in the puzzle.

Part 6: Write a main() function to do the following:

- Read n from the user.
- Initialize an $n \times n$ board by invoking the function of Part 2.
- Add new paths to the board using the function of Part 3 until an attempt returns FAILURE.
- Call the function of Part 4 to assign path numbers.
- Print the puzzle and its solution by invoking the function of Part 5.

Submit a single C/C++ source file. Do <u>not</u> use any global/static variable/array.

Sample Output

```
n = 7

+++ Initializing board...

+++ Attempting to add new paths...
    New path: (4,5) (4,4) (4,3) (4,2) (4,1) (4,0) (5,0) (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (5,6)
    New path: (3,3) (2,3) (1,3) (0,3) (0,2) (0,1) (0,0) (1,0) (2,0) (2,1) (3,1)
    New path: (2,2) (3,2)
    New path: (0,4) (1,4) (1,5) (1,6) (2,6) (3,6) (3,5) (3,4)
    New path: (2,4) (2,5)
    New path: (5,1) (5,2) (5,3) (5,4) (5,5)
    New path: (0,5) (0,6)
    New path: (1,1) (1,2)

+++ Assigning path numbers...

+++ The puzzle
    +----+----+----+----+----+----+----+
    |    |    |    |    |  2 |  3 |  3 |
    +----+----+----+----+----+----+----+
    |    |  4 |  4 |    |    |    |    |
    +----+----+----+----+----+----+----+
    |    |    |  5 |    |  6 |  6 |    |
    +----+----+----+----+----+----+----+
    |XXXX|  1 |  5 |  1 |  2 |    |    |
    +----+----+----+----+----+----+----+
    |    |    |    |    |    |  7 |XXXX|
    +----+----+----+----+----+----+----+
    |    |  8 |    |    |    |  8 |  7 |
    +----+----+----+----+----+----+----+
    |    |    |    |    |    |    |    |
    +----+----+----+----+----+----+----+

+++ The solution
    +----+----+----+----+----+----+----+
    |  1 |  1 |  1 |  1 |  2 |  3 |  3 |
    +----+----+----+----+----+----+----+
    |  1 |  4 |  4 |  1 |  2 |  2 |  2 |
    +----+----+----+----+----+----+----+
    |  1 |  1 |  5 |  1 |  6 |  6 |  2 |
    +----+----+----+----+----+----+----+
    |XXXX|  1 |  5 |  1 |  2 |  2 |  2 |
    +----+----+----+----+----+----+----+
    |  7 |  7 |  7 |  7 |  7 |  7 |XXXX|
    +----+----+----+----+----+----+----+
    |  7 |  8 |  8 |  8 |  8 |  8 |  7 |
    +----+----+----+----+----+----+----+
    |  7 |  7 |  7 |  7 |  7 |  7 |  7 |
    +----+----+----+----+----+----+----+
```