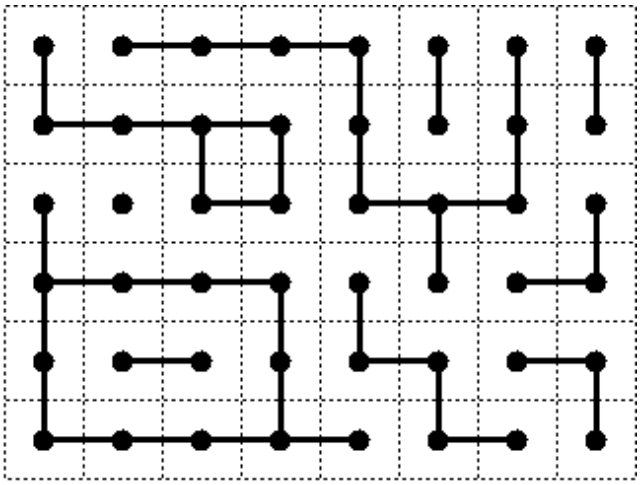
TopCoder Training Camp >> Tutorials >> Connected Components and FloodFill

A problem that appeared several times already is to identify regions of the same color in a matrix. Whether you have to count these regions, determine their areas or recolor them, the same basic algorithm can be used all the time.

There's a more general problem called "Connected Components". The input is an undirected graph and a connected component is a maximal subgraph in where every two vertices in the subgraph are connected by a path of edges in the original graph. Maximal means that we make each component as large as possible.

The matrix problem we have can be viewed as a special case of the connected components problem. To see this, look at the following example. On the left side we have a matrix that could've been the input to the recent "Farmers" problem of SRM 88 and on the right side we see the same input viewed as a graph.

A	C	C	C	C	A	C	B
A	A	A	A	C	A	C	B
B	C	A	A	C	C	C	A
B	B	B	B	A	C	A	A
B	C	C	B	A	A	C	C
B	B	B	B	B	A	A	C



If I counted it correctly, there are ten connected components here (one of them consists of only a single node).

Connected Components Algorithm

Let's first see how we can solve the general problem. We can use Depth First Search (DFS) to identify one connected component (CC) and when we apply this to every node in the graph, we can detect all CCs. Here is some pseudocode that could be used to count the number of CCs:

```
int numberOfComponents ( Graph G ) {
    for each node N of G
        seen[N] = false;
    answer = 0
    for each node N of G
        if not seen[N]
            ++answer
            DFS( G, N )
    return answer
}

void DFS ( Graph G, Node N ) {
    if seen[N]
        return
    seen[N] = true
    for each neighbour M of N in G
        DFS( G, M )
}
```

Here a call of DFS(G, N) marks all nodes in the CC that N belongs to as "seen", so that we count each component exactly once. Note that this is a very simple use of DFS. It's probably one of the most versatile and

useful algorithms out there.

FloodFill Algorithm

In our matrix problem, we don't yet have a nice graph representation. We could build a graph representation from a given matrix and then run the general CC algorithm on it, but it's much easier to directly hardcode the graph structure into the algorithm. That means, instead of having an explicit list of neighbours for each node, we look at the cells around a cell and view them as "neighbours" if they have the same color. Furthermore, our "numberOfComponents" will have two nested loops to walk over all "nodes". Usually this algorithm is called something like "FloodFill", since we somehow "flood" or "fill" the regions.

This time the code is in Java. The matrix is given as an array of Strings "land", just like in the Farmers problem. Height and width of the matrix are stored in "m" and "n", respectively.

We should only visit a neighbour cell if it has the same color as the current cell. But checking this directly would require much code, since we also have to check if the coordinates of each neighbour are inside the matrix before looking at its color. Instead, we defer this check and blindly visit all neighbour cells and we add the out-of-bounds check once and for all at the *entrance* of "flood". Since the visited node doesn't know from where it was visited, we have to give him at least the information what color we are searching for, here I call it "farmer".

```
String[] land;
int m, n;
boolean[][] seen;

void flood ( int i, int j, char farmer ) {
  if( i<0 || i>=m || j<0 || j>=n || seen[i][j] || land[i].charAt(j)!=farmer )
    return;
  seen[i][j] = true;
  flood( i-1, j, farmer );
  flood( i+1, j, farmer );
  flood( i, j-1, farmer );
  flood( i, j+1, farmer );
}

int numberOfRegions ( String[] land ) {
  this.land = land;
  m = land.length;
  n = land[0].length();
  seen = new boolean[m][n];

  int answer = 0;
  for( int i=0; i<m; ++i ){
    for( int j=0; j<n; ++j ){
      if( ! seen[i][j] ){
        ++answer;
        flood( i, j, land[i].charAt(j) );
      }
    }
  }

  return answer;
}
```

The variable "farmer" could also be global like land, m and n. On the other hand, these variables could also be passed as parameters to "flood", like "farmer". People tend to make things global, since that usually requires less typing and might run a bit faster

If you can write the cell contents (if you're using C++ strings or if you use two-dimensional arrays) then you can omit the "seen" matrix and instead overwrite the "land" matrix cells with something that doesn't appear in it otherwise, for example with '-'. This will also show that a cell has already been seen and will even be automatically tested when we test if the cell is of the color we're currently searching for.

References

You can find DFS and connected components discussed in many algorithm books, for example in Introduction to Algorithms by Cormen etc, and (I believe so, but will check it soon) in The Algorithm Design Manual by Steven Skiena.

Some problems where you can apply the FloodFill algorithm: Farmers of SRM 88, ... (more will follow, I'll just have to find them first).

Stefan Pochmann
Last modified: Sat May 25 16:00:59 PDT 2002