Flood Fill

Sample Problem: Connected Fields

Farmer John's fields are broken into fields, with paths between some of them. Unfortunately, some fields are not reachable from other fields via the paths.

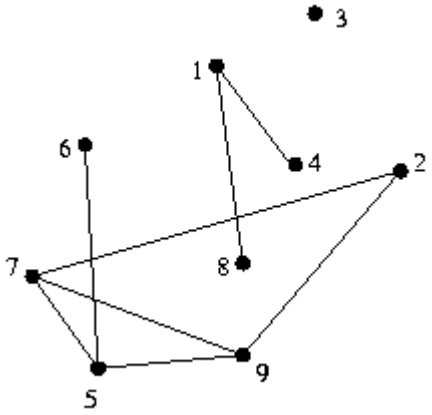
Define a *superfield* is a collection of fields that are all reachable from each other. Calculate the number of superfields.

The Abstraction

Given: a undirected graph

The *component* of a graph is a maximal-sized (though not necessarily maximum) subgraph which is connected.

Calculate the component of the graph.



This graph has three components: {1,4,8}, {2,5,6,7,9}, and {3}.

The Algorithm: Flood Fill

Flood fill can be performed three basic ways: depth-first, breadth-first, and breadth-first scanning. The basic idea is to find some node which has not been assigned to a component and to calculate the component which contains. The question is how to calculate the component.

In the depth-first formulation, the algorithm looks at each step through all of the neighbors of the current node, and, for those that

have not been assigned to a component yet, assigns them to this component and recurses on them.

In the breadth-first formulation, instead of recursing on the newly assigned nodes, they are added to a queue.

In the breadth-first scanning formulation, every node has two values: component and visited. When calculating the component, the algorithm goes through all of the nodes that have been assigned to that component but not visited yet, and assigns their neighbors to the current component.

The depth-first formulation is the easiest to code and debug, but can require a stack as big as the original graph. For explicit graphs, this is not so bad, but for implicit graphs, such as the problem presented has, the numbers of nodes can be very large.

The breadth-formulation does a little better, as the queue is much more efficient than the run-time stack is, but can still run into the same problem. Both the depth-first and breadth-first formulations run in N + M time, where N is the number of vertices and M is the number of edges.

The breadth-first scanning formulation, however, requires very little extra space. In fact, being a little tricky, it requires no extra space. However, it is slower, requiring up to N*N + M time, where N is the number of vertices in the graph.

Pseudocode for Breadth-First Scanning

This code uses a trick to not use extra space, marking nodes to be visited as in component -2 and actually assigning them to the current component when they are actually visited.

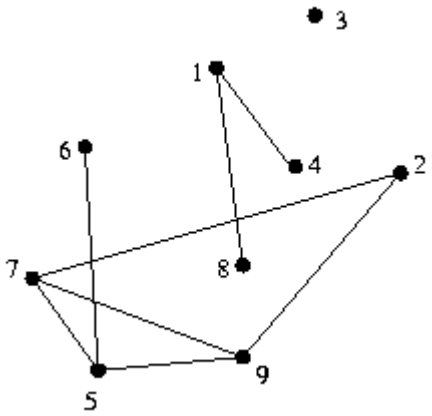
```
#  component(i)  denotes  the
#  component  that  node  i  is  in
  1  function  flood_fill(new_component)

  2  do
  3      num_visited  =  0
  4      for  all  nodes  i
  5          if  component(i)  =  -2
  6              num_visited  =  num_visited  +  1
  7              component(i)  =  new_component
  8              for  all  neighbors  j  of  node  i
```

```
 9                if  component(j)  =  nil
10                   component(j)  =  -2
11  until  num_visited  =  0

12  function  find_components

13    num_components  =  0
14    for  all  nodes  i
15        component(node  i)  =  nil
16    for  all  nodes  i
17        if  component(node  i)  is  nil
18            num_components  =
                                num_components  +  1
19            component(i)  =  -2
20            flood_fill(component
                                              num_components)
```

Running time of this algorithm is O(N^2), where N is the numbers of nodes. Every edge is traversed twice (once for each end-point), and each node is only marked once.

Execution Example

Consider the graph from above.



The algorithm starts with all nodes assigned to no component.

Going through the nodes in order first node not assigned to any component yet is vertex 1. Start a new component (component 1) for that node, and set the component of node 1 to -2 (any nodes not shown are unassigned).

Node	Component
1	-2

Now, in the `flood_fill` code, the first time through the `do` loop, it finds the node 1 is assigned to component -2. Thus, it reassigns it to component 1, signifying that it has been visited, and then assigns its neighbors (node 4) to component -2.

Node	Component
1	1
4	-2

As the loop through all the nodes continues, it finds that node 4 is also assigned to component -2, and processes it appropriately as well.

Node	Component
1	1
4	1
8	-2

Node 8 is the next to be processed.

Node	Component
1	1
4	1
8	1

Now, the `for` loop continues, and finds no more nodes that have not been assigned yet. Since the `until` clause is not satisfied (`num_visited` = 3), it tries again. This time, no nodes are found, so the function exits and component 1 is complete.

The search for unassigned nodes continues, finding node 2. A new component (component 2) is allocated, node 2 is marked as in component -2, and `flood_fill` is called.

Node	Component
1	1
2	-2

4	1
8	1

Node 2 is found as marked in component -2, and is processed.

Node	Component
1	1
2	2
4	1
7	**-2**
8	1
9	**-2**

Next, node 7 is processed.

Node	Component
1	1
2	2
4	1
5	**-2**
7	**2**
8	1
9	-2

Then node 9 is processed.

Node	Component
1	1
2	2
4	1
5	-2
7	2
8	1
9	**2**

The terminating condition does not hold (`num_visited` = 3), so the search through for nodes assigned to component -2 starts again. Node 5 is the first one found.

Node	Component
1	1
2	2
4	1
5	2
6	**-2**
7	2
8	1
9	2

Node 6 is the next node found to be in component -2.

Node	Component
1	1
2	2
4	1
5	2
6	**2**
7	2
8	1
9	2

No more nodes are found assigned to component -2, but the terminating condition does not hold, so one more pass through the nodes is performed, finding no nodes assigned to component -2. Thus, the search for unassigned nodes continue from node 2, finding node 3 unassigned.

Node	Component
1	1
2	2
3	**-2**
4	1

5	2
6	2
7	2
8	1
9	2

Node 3 is processed.

Node	Component
1	1
2	2
3	3
4	1
5	2
6	2
7	2
8	1
9	2

From here, the algorithm eventually terminates, as there are no more nodes assigned to component -2 and no unassigned nodes. The three components of the graph have been determined, along with the component to which each node belongs.

Problem Cues

Generally, these types of problem are fairly clear. If it asks for sets of "connected" things, it's probably asking for components, in which case flood fill works very well. Often, this is a step in solving the complete problem.

Extensions

The notion of ``components'' becomes muddied when you go to directed graphs.

However, the same flooding idea can be used to determine the points which are reachable from any given point even in a directed graph. At each recursive step, if the point isn't marked already, mark the point as reachable and recurse on all of its neighbors.

Note that to determine which points can reach a given point in a directed graph can be solved the same, by looking at every arc backwards.

Sample Problems

Company Ownership [abridged, IOI 93]

Given: A weighted directed graph, with weights between 0 and 100.

Some vertex A ``owns'' another vertex B if:

- A = B
- There is an arc from A to B with weight more than 50.
- There exists some set of vertices C_1 through C_k such that A owns C_1 through C_k, and each vertex has an arc of weight x_1 through x_k to vertex B, and $x_1 + x_2 + ... + x_k > 50$.

Find all (a,b) pairs such that a owns b.

Analysis: This can be solved via an adaptation of the calculating the vertices reachable from a vertex in a directed graph. To calculate which vertices vertex A owns, keep track of the ``ownership percentage'' for each node. Initialize them all to zero. Now, at each recursive step, mark the node as owned by vertex A and add the weight of all outgoing arcs to the ``ownership percentages.'' For all percentages that go above 50, recurse into those vertices.

Street Race [IOI 95]

Given: a directed graph, and a start point and an end point.

Find all points p that any path from the start point to the end must travel through p.

Analysis: The easiest algorithm is to remove each point in turn, and check to see if the end point is reachable from the start point. This runs in O(N (M + N)) time. Since the original problem stated that M <= 100, and N <= 50, this will run in time easily.

Cow Tours [1999 USACO National Championship, abridged]

The diameter of a connected graph is defined as the maximum distance between any two nodes of the graph, where the distance between two nodes is defined as the length of the shortest path.

Given a set of points in the plane, and the connections between those points, find the two points which are currently not in the same component, such that the diameter of the resulting component is minimized.

Analysis: Find the components of the original graph, using the method described above. Then, for each pair of points not in the same component, try placing a connection between them. Find the pair that minimizes the diameter.

Connected Fields

Farmer John contracted out the building of a new barn. Unfortunately, the builder mixed up the plans of Farmer John's barn with another set of plans. Farmer John's plans called for a barn that only had one room, but the building he got might have many rooms. Given a grid of the layout of the barn, tell Farmer John how many rooms it has.

Analysis: The graph here is on the non-wall grid locations, with edge between adjacent non-wall locations, although the graph should be stored as the grid, and not transformed into some other form, as the grid is so compact and easy to work with.

Flood Fill

种子染色法

译 by Lucky Crazy & Felicia Crazy

（Flood Fill 按原意应翻译成“水流式填充法”（如果我没译错），有些中文书籍上将它称作“种子染色法”，然而大部分的书籍（包括中文书籍）都直接引用其英文原名：Flood Fill。介于此，下文所有涉及到 Flood Fill 的都直接引用英文 ——译者）

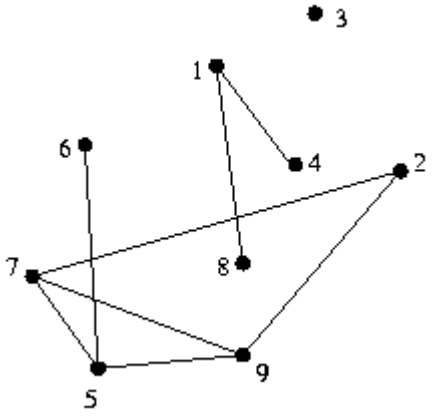
样例： 相连的农场

Farmer John 的农场被一次意外事故破坏了，有一些农场与其他的农场之间有道路相连，而有些道路却已被破坏。这使得 Farmer John 无法了解到从一个农场能否到达另一个农场。你的任务就是帮助 Farmer John 来了解哪些农场是连通的。

给出：

上题实际上就是要求寻找一张无向图的所有极大连通子图。

给出一张未知连通性的图，如下图：



可知，该图的极大连通子图是：{1,4,8}， {2,5,6,7,9} 和 {3}。

算法：Flood Fill

Flood Fill 可以用深度优先搜索，广度优先搜索或广度优先扫描来实现。他的实现方式是寻找到一个未被标记的结点对它标记后扩展，将所有由它扩展出的结点标上与它相同的标号，然后再找另一个未被标号的 结点重复该过程。这样，标号相同的结点就属于同一个连通子图。

深搜：取一个结点，对其标记，然后标记它所有的邻结点。对它的每一个邻结点这么一直递归下去完成搜索。

广搜：与深搜不同的是，广搜把结点加入队列中。

广度扫描(不常见)：每个结点有两个值，一个用来记录它属于哪个连通子图(c)，一个用来标记是否已经访问（v）。算法对每一个未访问而在某个连通子图当中的结点扫描，将其标记访问，然后把它的邻结点的（c）值改为当前结点的（c）值。

深搜最容易写，但它需要一个栈。搜索显式图没问题，而对于隐式图，栈可能就存不下了。

广搜稍微好一点，不过也存在问题。搜索大的图它的队列有可能存不下。深搜和广搜时间复杂度均为 O(N+M)。其中，N 为结点数，M 为边数。

广度扫描需要的额外空间很少，或者可以说根本不要额外空间，但是它很慢。时间复杂度是 O(N^2+M)。

（实际应用中，我们一般写的是 DFS，因为快。空间不是问题，DFS 可改用非递归的栈操作完成。但为了尊重原文，我们还是译出了广度扫描的全过程。——译者）

广度扫描的伪代码

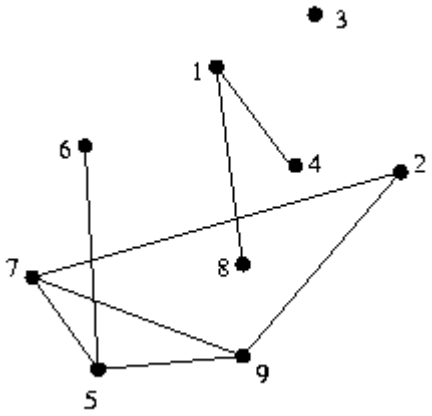
代码中用了一个小技巧，因此无须额外空间。结点若未访问，将其归入连通子图（-2），就是代码里的 component -2。这样无须额外空间来记录结点是否访问，请读者用心体会。

```
#  component(i)  denotes  the
#  component  that  node  i  is  in
  1  function  flood_fill(new_component)

  2  do
  3      num_visited  =  0
  4      for  all  nodes  i
  5          if  component(i)  =  -2
  6              num_visited  =  num_visited  +  1
  7              component(i)  =  new_component
  8              for  all  neighbors  j  of  node  i
  9                  if  component(j)  =  nil
10                      component(j)  =  -2
11  until  num_visited  =  0

12  function  find_components

13    num_components  =  0
14    for  all  nodes  i
15        component(node  i)  =  nil
16    for  all  nodes  i
17        if  component(node  i)  is  nil
18            num_components  =
                                  num_components  +  1
19            component(i)  =  -2
20            flood_fill(component
                                              num_components)
```

算法的时间复杂度是 $O(N^2)$，每个结点访问一次，每条边经过两次。

实例

考虑刚才的那张图：



开始时，所有的结点都没有访问。（下例中未访问被表示为 -2）

首先从结点 1 开始，结点 1 未访问，那么先处理结点 1，将它归入连通子图 1。

结点	连通子图
1	-2

标记完成后，对它进行第一步的扩展，由结点 4 和结点 8 与结点 1 连通，故它们被扩展出来。

结点	连通子图
1	1
4	-2
8	-2

之后，先处理结点 4，将它与结点 1 归入相同的连通子图。现在它没有可扩展的结点了（结点 1 已被扩展过）

结点	连通子图
1	1
4	1
8	-2

接着处理结点 8。结果与结点 4 一样。

结点	连通子图
1	1
4	1
8	**1**

现在，所有与结点 1 连通的结点都已扩展，标号为 1 的连通子图产生了。那么我们将跳出扩展步骤，寻找下一个连通子图，标号为 2。

与上一步相同的顺序，找到结点 2。

结点	连通子图
1	1
2	**-2**
4	1
8	1

扩展结点 2，结点 7 与结点 9 出现。

结点	连通子图
1	1
2	2
4	1
7	**-2**
8	1
9	**-2**

下一步，扩展结点 7，结点 5 出现。

结点	连通子图
1	1
2	2
4	1
5	**-2**
7	**2**
8	1
9	-2

然后是结点 9。

结点	连通子图
1	1
2	2
4	1
5	-2
7	2
8	1
9	**2**

扩展结点 5。结点 6 出现。

结点	连通子图
1	1
2	2
4	1
5	2
6	**-2**
7	2
8	1
9	2

很遗憾，结点 6 没有可供扩展的结点。至此连通子图 2 产生。

结点	连通子图
1	1
2	2
4	1
5	2
6	**2**
7	2
8	1
9	2

之后寻找连通子图 3，至此，仅有结点 3 未被扩展。

结点	连通子图
1	1
2	2
3	-2
4	1
5	2
6	2
7	2
8	1
9	2

结点 3 没有可供扩展的结点,这样,结点 3 就构成了仅有一个结点的连通子图 3。

结点	连通子图
1	1
2	2
3	3
4	1
5	2
6	2
7	2
8	1
9	2

结点 3 处理结束后，整个图的所有 9 个结点就都被归入相应的 3 个连通子图。Flood Fill 结束。

问题提示

这类问题一般很清晰，求解关于“连通”的问题会用到 Flood Fill。它也很经常用作某些算法的预处理。

扩展与延伸

有向图的连通性比较复杂。

同样的填充算法可以找出从一个结点能够到达的所有结点。每一层递归时，若一个结点未访问，就将其标记为已访问（表示他可以从源结点到达)，然后对它所有能到达且为访问的结点进行下一层递归。

若要求出可以到达某个结点的所有结点，你可以对后向弧做相同的操作。

例题

控制公司 [有删节, IOI 93]

已知一个带权有向图，权值在 0-100 之间。

如果满足下列条件，那么结点 A“拥有”结点 B：

- A = B
- 从 A 到 B 有一条权值大于 50 的有向弧。
- 存在一系列结点 C_1 到 C_k 满足 A 拥有 C_1 到 C_k, 每个节点都有一条弧到 B，记作 x_1，x_2...x_k, 并且 $x_1 + x_2 + ... + x_k > 50$。

找出所有的（A，B）对，满足 A 拥有 B。

分析：这题可以用上面提到的“给出一个源，在有向图中找出它能够到达的结点”算法的改进版解决。要计算 A 拥有的结点，要对每个结点计算其“控股百分比”。把它们全部设为 0。现在，在递归的每一步中，将其标记为属于 A 并把它所有出弧的权加到“控股百分比”中。对于每个“控股百分比”超过 50 的结点，进行同样的操作（递归）。

街道赛跑 [IOI 95]

已知一个有向图，一个起点和一个终点。

找出所有的 p，使得从起点到终点的任何路径都必须经过 p。

分析：最简单的算法是枚举 p，然后把 p 删除，看看是否存在从起点到终点的通路。时间复杂度为 O(N (M + N))。题目的数据范围是 M <= 100, N <= 50，不会超时。

牛路 [1999 USACO 国家锦标赛, 有删节]

连通图的直径定义为图中任意两点间距离的最大值，两点间距离定义为最短路的长。

已知平面上一个点集，和这些点之间的连通关系，找出不在同一个连通子图中的两个点，使得连接这两个点后产生的新图有最小的直径。

分析：找出原图的所有连通子图，然后枚举不在同一个连通子图内的每个点对，将其连接，然后找出最小直径。

笨蛋建筑公司

Farmer John 计划建造一个新谷仓。不幸的是，建筑公司把他的建造计划和其他人的建造计划混淆了。Farmer John 要求谷仓只有一个房间，但是建筑公司为他建好的是有许多房间的谷仓。已知一个谷仓平面图，告诉 Farmer John 它一共有多少个房间。

分析：随便找一个格子，遍历所有与它连通的格子，得到一个房间。然后再找一个未访问的格子，做同样的工作，直到所有的格子均已访问。虽然题目给你的不是直接的图，但你也可以很容易地对其进行 Flood Fill。