OPTIMIZING ALGORITHM USING BREADTH FIRST SEARCH MANNER

Shauma Hayyu Syakura

Program Studi Informatika
Sekolah Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
e-mail : if17025@students.if.itb.ac.id

ABSTRACT

Flood fill, also called seed fill, is an algorithm that determines the area connected to a given node in a multi-dimensional array. Many problems have been solved using this algorithm, one of the examples is the using of this algorithm in the "bucket" fill tool of paint programs to determine which parts of a bitmap to fill with color, and in puzzle games such as Minesweeper, Puyo Puyo, Lumines, Samegame and Magical Drop for determining which pieces are cleared. When applied on an image to fill a particular bounded area with color, it is also known as boundary fill. In this paper, a more opti-mized, enhanced, and memory preserving flood fill algorithm using Breadth First Search (BFS) Algorithm will be shown instead of using ordinary recursive flood fill algorithm.

Keywords : node, flood fill, stack overflow, depth first search, breadth first search

1. Prefaces

The flood fill algorithm takes three parameters: a start node, a target color, and a replacement color. The algorithm looks for all nodes in the array which are connected to the start node by a path of the target color, and changes them to the replacement color. There are many ways in which the flood-fill algorithm can be structured, but they all make use of a queue or stack data structure, explicitly or implicitly. One implicitly stack-based (recursive) flood-fill imple-mentation (for a two-dimensional array) goes as follows:

```
Flood-fill (node, target-color,
replacement-color):
    1.  If the color of node is not equal
        to target-color, return.
    2.  Set the color of node to
        replacement-color.
    3.  Perform Flood-fill (one step to the
        west of node, target-color,
        replacement-color).
        Perform Flood-fill (one step to the
        east of node, target-color,
        replacement-color).
        Perform Flood-fill (one step to the
        north of node, target-color,
        replacement-color).
        Perform Flood-fill (one step to the
        south of node, target-color,
        replacement-color).
    4.  Return.
```

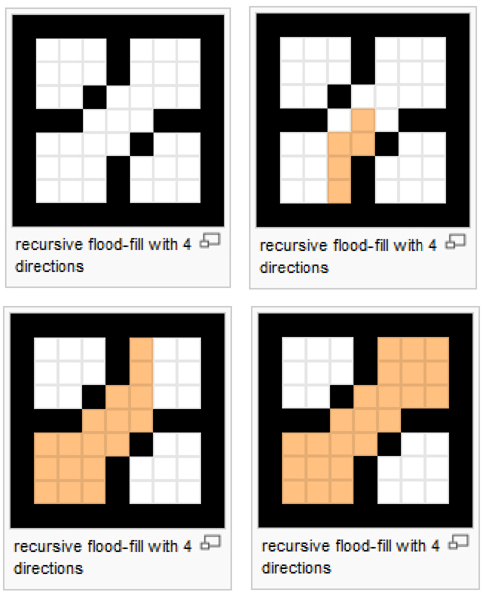
recursive flood-fill with 4 directions
recursive flood-fill with 4 directions
recursive flood-fill with 4 directions
recursive flood-fill with 4 directions

Figure 1. step by step of recursive flood fill with 4 directions

Though easy to understand, the implementation of the algorithm used above is impractical in languages and environments where stack space is severely constrained (e.g. Java applets), and in bitmap coloring, stack memory in internal memories are perserved, causing massages like "stack overflow" often encountered.

2. Method

The depth first search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph. A recent TopCoder problem was a classic application of the depth first search, the flood-fill. The flood-fill operation will be familiar to anyone who has used a graphic painting application. The concept is to fill a bounded region with a single color, without leaking outside the boundaries.

In Flood Fill Algorithm using Depth First Search method, stack is used for it’s data structure. A stack is one of the simplest data structures available. There are four main operations on a stack:

- Push - Adds an element to the top of the stack
- Pop - Removes the top element from the stack
- Top - Returns the top element on the stack
- Empty - Tests if the stack is empty or not

This concept maps extremely well to a Depth First search. The basic concept is to visit a node, then push all of the nodes to be visited onto the stack. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well and we continue doing this until all nodes are visited. It is a key property of the Depth First search that we not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it, then unmarking it after we have finished our recursions. This action allows us to visit all the paths that exist in a graph; however for large graphs this is mostly infeasible so we sometimes omit the marking the node as not visited step to just find one valid path through the graph (which is good enough most of the time).

So the basic structure will look something like this:

```
dfs(node start) {
 stack s;
 s.push(start);
 while (s.empty() == false) {
  top = s.top();
  s.pop();
  mark top as visited;

  check for termination condition

  add all of top's unvisited neighbors to
the stack.
  mark top as not visited;
 }
}
```

Alternatively we can define the function recursively as follows:

```
dfs(node current) {
 mark current as visited;
 visit all of current's unvisited neighbors
by calling dfs(neighbor)
 mark current as not visited;
}
```

The problem we will be discussing is grafixMask, a Division 1 500 point problem from SRM 211. This problem essentially asks us to find the number of discrete regions in a grid that has been filled in with some values already. Dealing with grids as graphs is a very powerful technique, and in this case makes the problem quite easy.

We will define a graph where each node has 4 connections, one each to the node above, left, right and below. However, we can represent these connections implicitly within the grid, we need not build out any new data structures. The structure we will use to represent the grid in grafixMask is a two dimensional array of booleans, where regions that we have already determined to be filled in will be set to true, and regions that are unfilled are set to false.

To set up this array given the data from the problem is very simple, and looks something like this:

```
bool fill[600][400];
initialize fills to false;

foreach rectangle in Rectangles
    set from (rectangle.left, rectangle.top)
to (rectangle.right, retangle.bottom) to
true
```

Now we have an initialized connectivity grid. When we want to move from grid position (x, y) we can either move up, down, left or right. When we want to move up for example, we simply check the grid position in (x, y-1) to see if it is true or false. If the grid position is false, we can move there, if it is true, we cannot.

Now we need to determine the area of each region that is left. We don't want to count regions twice, or pixels twice either, so what we will do is set fill[x][y] to true when we visit the node at (x, y). This will allow us to perform a Depth-First search to visit all of the nodes in a connected region and never visit any node twice, which is exactly what the problem wants us to do! So our loop after setting everything up will be:

```
int[] result;

for x = 0 to 599
 for y = 0 to 399
  if (fill[x][y] == false)
   result.addToBack(doFill(x,y));
```

All this code does is check if we have not already filled in the position at (x, y) and then calls doFill() to fill in that region. At this point we have a choice, we can define doFill recursively (which is usually the quickest and easiest way to do a depth first search), or we can define it explicitly using the built in stack classes. I will cover the recursive method first, but we will soon see for this problem there are some serious issues with the recursive method.

We will now define doFill to return the size of the connected area and the start position of the area:

```
int doFill(int x, int y) {
// Check to ensure that we are within the
bounds of the grid, if not, return 0
 if (x < 0 || x >= 600) return 0;
// Similar check for y
 if (y < 0 || y >= 400) return 0;
// Check that we haven't already visited
this position, as we don't want to count it
twice
 if (fill[x][y]) return 0;

// Record that we have visited this node
 fill[x][y] = true;

 // Now we know that we have at least one
empty square, then we will recursively
attempt to
 // visit every node adjacent to this node,
and add those results together to return.
 return 1 + doFill(x - 1, y) + doFill(x + 1,
y) + doFill(x, y + 1) + doFill(x, y - 1);
}
```

This solution should work fine, however there is a limitation due to the architecture of computer programs. Unfortunately, the memory for the implicit stack, which is what we are using for the recursion above is more limited than the general heap memory. In this instance, we will probably overflow the maximum size of our stack due to the way the recursion works, so we will next discuss the explicit method of solving this problem.

Stack memory is used whenever you call a function; the variables to the function are pushed onto the stack by the compiler for you. When using a recursive function, the variables keep getting pushed on until the function returns. Also any variables the compiler needs to save between function calls must be pushed onto the stack as well. This makes it somewhat difficult to predict if you will run into stack difficulties. I recommend using the explicit Depth First search for every situation you are at least somewhat concerned about recursion depth.

In this problem we may recurse a maximum of 600 * 400 times (consider the empty grid initially, and what the depth first search will do, it will first visit 0,0 then 1,0, then 2,0, then 3,0 ... until 599, 0. Then it will go to 599, 1 then 598, 1, then 597, 1, etc. until it reaches 599, 399. This will push 600 * 400 * 2 integers onto the stack in the best case, but depending on what your compiler does it may in fact be more information. Since an integer takes up 4 bytes we will be pushing 1,920,000 bytes of memory onto the stack, which is a good sign we may run into trouble.

We can use the same function definition, and the structure of the function will be quite similar, just we won't use any recursion any more:

```
class node { int x, y; }

int doFill(int x, int y) {
 int result = 0;

 // Declare our stack of nodes, and push our
starting node onto the stack
 stack s;
 s.push(node(x, y));

 while (s.empty() == false) {
  node top = s.top();
  s.pop();

// Check to ensure that we are within the
bounds of the grid, if not, continue
  if (top.x < 0 || top.x >= 600) continue;
// Similar check for y
  if (top.y < 0 || top.y >= 400) continue;
// Check that we haven't already visited
this position, as we don't want to count it
twice
  if (fill[top.x][top.y]) continue;

  fill[top.x][top.y] = true; // Record that
we have visited this node

  // We have found this node to be empty,
and part
  // of this connected area, so add 1 to the
result
  result++;

  // Now we know that we have at least one
empty square, then we will attempt to
  // visit every node adjacent to this node.
  s.push(node(top.x + 1, top.y));
  s.push(node(top.x - 1, top.y));
  s.push(node(top.x, top.y + 1));
  s.push(node(top.x, top.y - 1));
 }

 return result;
}
```

As you can see, this function has a bit more overhead to manage the stack structure explicitly, but the advantage is that we can use the entire memory space available to our program and in this case, it is necessary to use that much information. However, the structure is quite similar and if you compare the two implementations they are almost exactly equivalent.

3. A More Optimized Flood Fill Algorithm using Breadth First Search

A queue is a simple extension of the stack data type. Whereas the stack is a FILO (first-in last-out) data structure the queue is a FIFO (first-in first-out) data structure. What this means is the first thing that you add to a queue will be the first thing that you get when you perform a pop().

There are four main operations on a queue:

- Push - Adds an element to the back of the queue
- Pop - Removes the front element from the queue
- Front - Returns the front element on the queue
- Empty - Tests if the queue is empty or not

In C++, this is done with the STL class queue:

```
#include
queue myQueue;
```

In Java, we unfortunately don't have a Queue class, so we will approximate it with the LinkedList class. The operations on a linked list map well to a queue (and in fact, sometimes queues are implemented as linked lists), so this will not be too difficult.

The operations map to the LinkedList class as follows:

- Push - boolean LinkedList.add(Object o)
- Pop - Object LinkedList.removeFirst()
- Front - Object LinkedList.getFirst()
- Empty - int LinkedList.size()

```
import java.util.*;
LinkedList myQueue = new LinkedList();
```

In C#, we use Queue class:

The operations map to the Queue class as follows:

- Push - void Queue.Enqueue(Object o)
- Pop - Object Queue.Dequeue()
- Front - Object Queue.Peek()
- Empty - int Queue.Count

The Breadth First search is an extremely useful searching technique. It differs from the depth-first search in that it uses a queue to perform the search, so the order in which the nodes are visited is quite different. It has the extremely useful property that if all of the edges in a graph are unweighted (or the same weight) then the first time a node is visited is the shortest path to that node from the source node. You can verify this by thinking about what using a queue means to the search order. When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the FIFO property of the queue and ends up being an extremely useful property. One thing that we have to be careful about in a Breadth First search is that we do not want to visit the same node twice, or we will lose the property that when a node is visited it is the quickest path to that node from the source.

The basic structure of a breadth first search will look this:

```
void bfs(node start) {
 queue s;
 s.push(start);
 while (s.empty() == false) {
  top = s.front();
  s.pop();
  mark top as visited;
```

check for termination condition (have we reached the node we want to?) add all of top's unvisited neighbors to the stack.

```
 }
}
```

Notice the similarities between this and a depth-first search, we only differ in the data structure used and we don't mark top as unvisited again. In DepthFirstSearch (DFS), we explore a vertex's neighbours recursively, meaning that we reach as much depth as possible first, then go back and visit othe rneighbours (and hence the name DepthFirst). Another useful search algorithm is the BreadthFirst Search (BFS). In BFS, we start with one vertex in a visited set, the source vertex. Then, at each step, we visit the entire layer of unvisited vertices reachable by some vertex in the visited set, and add them to the visited set. Doing so, BFS visits vertices in order of their breadth, or simply the distance from that vertex to the source. BFS builds its own BreadthFirst tree, and is an iterative algorithm. A simple problem that BFS is good at is the floodfill problem, mentioned in the DFS section. A floodfill simply fills all vertices reachable by some source vertex with the same colour, much like the paint bucket tool in imageediting programs. The idea is to visit the vertices in a breadthfirst manner using BFS, and colour each vertex as we reach it.

```
bool M[128][128]; // adjacency matrix (can
have at most 128 vertices)
bool seen[128]; // which vertices have been
visited
int n; // number of vertices
// ... Initialize M to be the adjacency
matrix
queue<int> q; // The BFS queue to represent
the visited set
    int s = 0; // the source vertex
// BFS floodfill
for( int v = 0; v < n; v++ ) seen[v] =
false; // set all vertices to be "unvisited"
seen[s] = true;
DoColouring( s, some_color );
q.push( s );
```

```
while ( !q.empty() ) {
        int u = q.front(); // get first
untouched
        vertex
        q.pop();
        for( int v = 0; v < n; v++ ) if(
!seen[v] && M[u][v] ) {
                seen[v] = true;
                DoColouring( v, some_color );
                q.push( v );
                }
        }
```

Now, several things are going on in this example. We use a queue to represent the visited set because a queue will keep the vertices in order of when they were first visited. This means the queue will keep the vertices in a breadthfirst manner. We first start off colouring the source vertex, marking it as seen, and pushing it onto the queue. Now, for each vertex in the queue, we simply do the same for all of its neighbours – colour them, mark them as seen, and push them onto the queue. Note that BFS is an iterative algorithm, so we did not write it as a function.

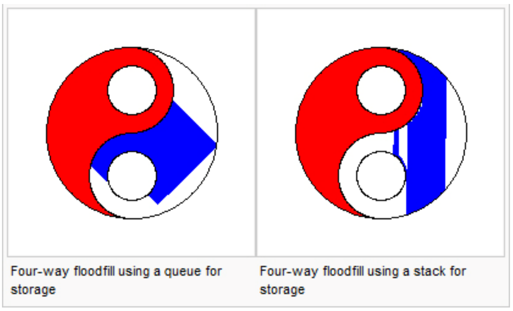
Four-way floodfill using a queue for storage
Four-way floodfill using a stack for storage

Figure 2. BFS Floodfill vs DFS Floodfill

Depth-first search, which is the natural algorithm, uses far more auxiliary space than a breadth-first search. I dare not try depth-first search by simple recursion; the depth of recursion is limited only by REDACTED, and experiments show that an PROBLEM REDACTED could nevertheless require a stack depth of over a million. So, put the stack in an auxiliary data structure. Using an explicit stack actually makes it easy to try breadth-first search as well, and it turns out that breadth-first search can use forty times less space than depth-first search.

4. Conclusion

The standard flood fill algorithm using DFS causing problems in internal memory size. Huge data will cause the stack to be overflowed. BFS method usage in floodfill algorithm has been proven to be more optimize, faster, and better in preserving internal memory.

REFERENCE

[1] http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=findSolution, 2-1-2010, 01.34 AM

[2] http://stackoverflow.com/questions/1257117/does-anyone-have-a-working-non-recursive-floodfill-algorithm-written-in-c, 2-1-2010, 01.43 AM

[3] http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part1.pdf, 2-1-2010, 01.47 AM

[4] http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=graphsDataStrucs2, 3-1-2010, 04.30 PM

[5] http://en.wikipedia.org/wiki/Flood_fill, 2-1-2010, 01.32 AM