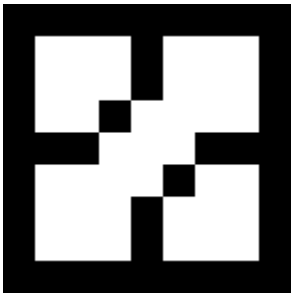
[Software Engineer] Flood Fill

Introduction

Flood fill is an algorithm that determines the area connected to a given node in a multi-dimensional array.

It is used in the "bucket" tool of paint programs to fill connected, similarly-coloured areas with a different colour.



We want to write the function that implements this algorithm with a couple of restrictions:

- we consider only 4 directions to connect pixels (up, down, left, right)
- we assume an already existing pseudo API like this one:

```
class Pixel { x, y: integer }
class Color { c: integer }
class Image { width, height: integer; getColor(Pixel) -> Color; setColor(Pixel, Color) }

FloodFill(Image origImage, Pixel startPixel, Color newColor)
```

Drill Downs

Testing

How to test your solution in the most automated way?

What about corner cases?

Multi-threading

How to use multiple threads to process the image?

What has to be protected by critical sections?

Very large images

How to deal with very large images that won't fit in memory at once?

How to interrupt the process and resume later?

Distributed system

With a large set of images, how to distribute the work among several servers?

How to balance the load? One worker per image or a more sophisticated strategy?

Fuzzy flood fill

How to deal with ranges of colour instead of exact ones?

How to adapt the API/UI/UX for this new feature?

Skills

Basic flow control

Check if-then-else and loop constructions, exit conditions, boundary checks, infinite loops, etc.

Data structures

Check correct usage of stack/queue and the `Image` API.

Check for useless temporary storage (e.g. the image is already a 2D array, do we really need to remember previously visited pixels?).

Ask about complexity (time, space), performance, optimisation.

Recursion vs Iteration

Check recursion shortcomings (e.g. stack overflows), recursion to iteration refactoring, BFS/DFS.

Multi-processing

Explore multi-threading pitfalls (race conditions, reentrant data structures, etc.).

Explore distributed systems pitfalls (message bus/queue, resilience, load balancing/sharding, etc.).

Solution Examples

Pseudo Code

Recursive

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

Iterative

```
FloodFillImpl (node, target-color, replacement-color):
    If target-color is equal to replacement-color, return.
    If color of node is not equal to target-color, return.
    Set Q to the empty queue.
    Set the color of node to replacement-color.
    Add node to the end of Q.
    While Q is not empty:
        Set n equal to the first element of Q.
        Remove first element from Q.
        If the color of the node to the west of n is target-color,
            set the color of that node to replacement-color and add that node to the end of Q.
        If the color of the node to the east of n is target-color,
            set the color of that node to replacement-color and add that node to the end of Q.
        If the color of the node to the north of n is target-color,
            set the color of that node to replacement-color and add that node to the end of Q.
        If the color of the node to the south of n is target-color,
            set the color of that node to replacement-color and add that node to the end of Q.
    Continue looping until Q is exhausted.
    Return.
```

Testing

One would prepare a set of original and filled images then compare automatically each pixel against the function output.

Beware of the infinite loop when target colour is the same as original colour.

Generating random images will only test if the algorithm finishes but is practically useless.

Multi-Threading / Multi-Processing / Distributed System

One possible idea is to cut the image in several chunks. Each chunk has its own concurrent queue.

Workers will push and pop pixels on their own queues but will have to push out-of-bounds pixels to the corresponding chunks' queues.

This is trivially translated to a multiple servers architecture by using a distributed message queue on top of the workers' ones.

Very large images

One possible idea is to treat the image in several memory-fitting chunks and save to disk new starting points each time we cross a chunk boundary, then repeat.

Fuzzy flood fill

One possible idea is to define a `distance(Color, Color) → float` function and refactor the code to use it against a user-specified threshold.