Image Comparison Using Quadtree

Ram Keerthy Athinarayanan, Kavita Soni, Suman K Batra

Department of Physics and Computer Science,
Wilfrid Laurier University,
Waterloo, Ontario – N2M 5C8

{ athi8690@mylaurier.ca, soni1197@mylaurier.ca , batr3900@mylaurier.ca }

Motivation:
Most of the image processing logic do pixel-by-pixel comparison to generate the desired output as shown in Fig.1. Often these comparisons have cubic runtime complexity. For a general image of k discrete colours and of $w * h$ pixels it has runtime complexity of $\Theta(k * w * l)$. Through this project we strive to use quadtree decomposition to compare two images and achieve faster and more efficient image comparison method.

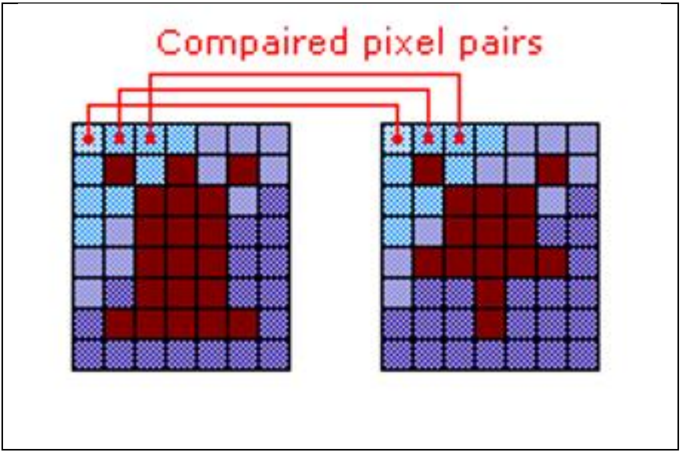


Fig 1: Pixel by Pixel Comparison

I. INTRODUCTION

Quad-tree has at most of four nodes in them and the tree is not necessarily to be balanced. Quad-tree grows up to a maximum height of $logN$. Where N is $max(w, h)$. In image manipulation we get a complete quad-tree with the leaf nodes consisting of individual pixels of the image. On an image, a quadtree recursively divides the image into four sub images and stops when some criteria are met or reaching a single pixel. Each node either stores the color of the pixel (if it's a single pixel), or the average of the colors of all pixels in the quadrant. A simple quad-tree for a really simple image will look like as shown in Fig.2.

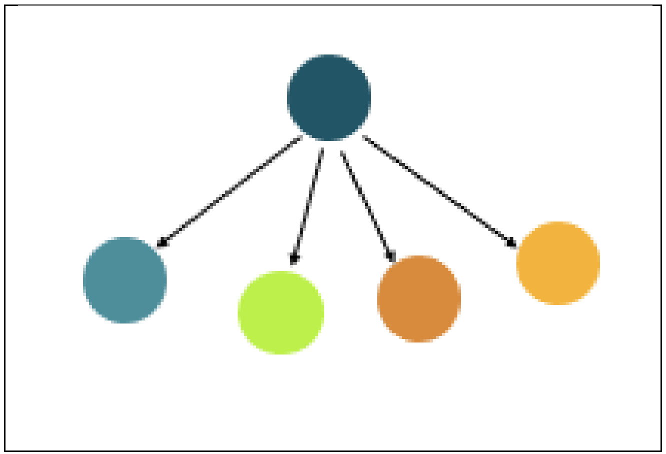


Fig 2: Sample Quadtree

For the comparison, first the images are compressed using the quadtree decomposition. Average of all the pixels would be stored in a single root node. The two images can be compared by descending the root node until a certain threshold level. Setting up a threshold value while traversing through the quad-tree will give faster results depending on the level specified. The more the level we traverse the better will be the image quality, $threshold\ level \propto \frac{1}{fastness\ of\ algorithm}$
If an image processing algorithm does not need the full clear image, then the threshold can be adjusted accordingly to meet the requirement of the image processing algorithm. This makes the quad-tree more adaptive. We can limit the resolution of the image which is given as input to the image processing logic by limiting the level of propagation in quad tree.

II. CONSTRUCTION OF QUADTREE

Divide the image/ sub-image into four equal sub-images. If a sub-image contains more than one colour in it, create a child node. The idea is to split only in those quadrants where the colors of the children differ greatly. This also helps to achieve space optimization. If a sub-image contains only one colour assign the colour to that node and do not create a child for it. Recurse for each of the children. If the recursion reaches the pixel level assign the colour value of the pixel to the node's colour. If all the child nodes have the same colour then they can be represented using a single node. From Fig.3 it can be seen how quadtree formation of a single image looks like.

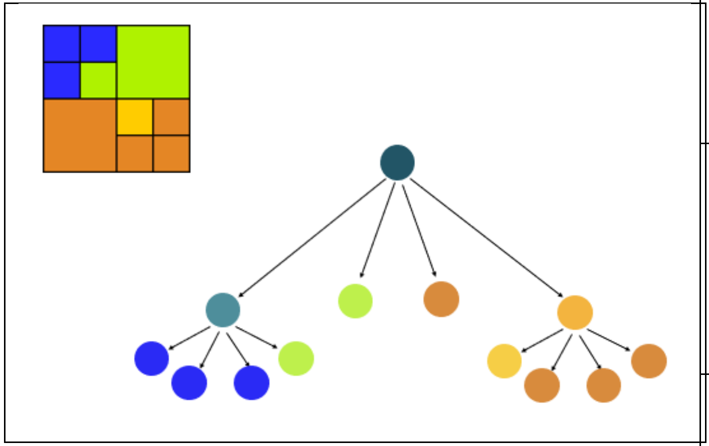


Fig 3: Construction of Quadtree

III. IMPLEMENTATION OF ALGORITHM

A. Classes used in implementation

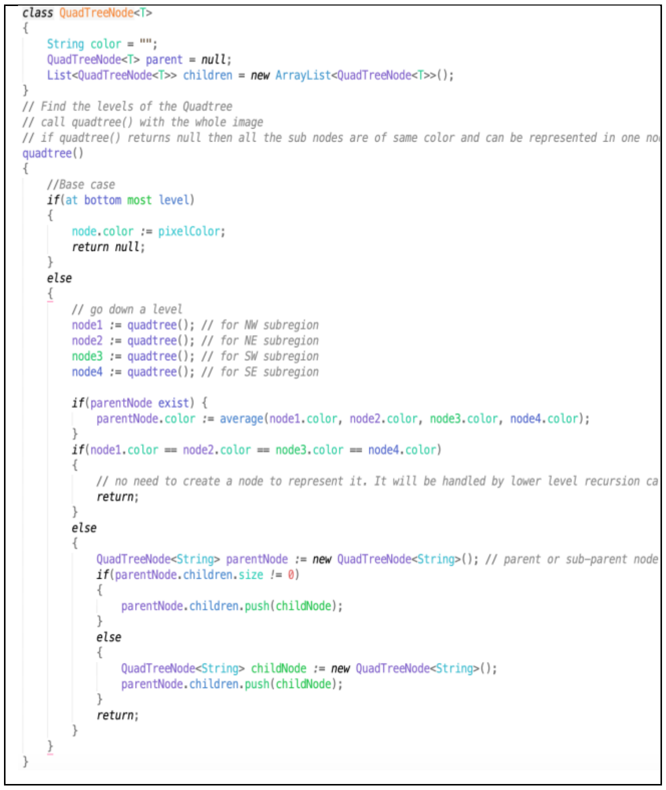
Below Table.1 illustrates the implementation of the classes used for quadtree algorithm implementation

Colour	This class is specifically used to store and get colours of each pixel present in image. The colour format used here is alpha, red, green and blue
Coordindate	Coordinate class represents a point in image which is represented as x and y.
MathematicalOperations	This specific class is used to compute log calculations to check if the image is in correct format for constructing quadtree.
ImageScaling	Image scaling is used for preprocessing the images into specific format of 1024x1024 resolution to form more accurate quadtree.
QuadtreeNode	QuadTreeNode is the basic building block of quadtree which stores the information of image in form of quadtree with parent and children nodes
Quadtree	This class is responsible for constructing quadtree from image by using recursion. Some other functionalities of this class are to compare two quadtrees and see if they are same to the given level.
ComapreImages	This class has a naïve implementation of how two images are compared to each other. The worst case of this comparison will be tested by passing same images for comparison.
QuadtreeMain	This is the driver class which has public static void main in it. This class get the image inputs, levels of quadtrees to be compared and calls in appropriate functions for classes to compare the images

Table 1: Classes used in implementation

B. Algorithm

Fig.4 shows the algorithm for quadtree implementation.



```
class QuadTreeNode<T>
{
    String color = "";
    QuadTreeNode<T> parent = null;
    List<QuadTreeNode<T>> children = new ArrayList<QuadTreeNode<T>>();
}
// Find the levels of the Quadtree
// call quadtree() with the whole image
// if quadtree() returns null then all the sub nodes are of same color and can be represented in one no
quadtree()
{
    //Base case
    if(at bottom most level)
    {
        node.color := pixelColor;
        return null;
    }
    else
    {
        // go down a level
        node1 := quadtree(); // for NW subregion
        node2 := quadtree(); // for NE subregion
        node3 := quadtree(); // for SW subregion
        node4 := quadtree(); // for SE subregion

        if(parentNode exist) {
            parentNode.color := average(node1.color, node2.color, node3.color, node4.color);
        }
        if(node1.color == node2.color == node3.color == node4.color)
        {
            // no need to create a node to represent it. It will be handled by lower level recursion ca
            return;
        }
        else
        {
            QuadTreeNode<String> parentNode := new QuadTreeNode<String>(); // parent or sub-parent node
            if(parentNode.children.size != 0)
            {
                parentNode.children.push(childNode);
            }
            else
            {
                QuadTreeNode<String> childNode := new QuadTreeNode<String>();
                parentNode.children.push(childNode);
            }
            return;
        }
    }
}
```

Fig 4: Algorithm

C. *Operations and Runtime of Quadtree*

Below Table.2 shows the runtime of various operations of quadtree.

Creation	The complexity of creating a full-grown quad-tree is $\Theta(w * h)$. Where w is width and h is height of the image.
Insertion	The complexity of inserting a new node to the existing quad-tree is $O(logN)$. Where N is $max(w, h)$.
Searching	The complexity of searching an element in a quad-tree is $O(logN)$. Where N is $max(w, h)$.
Deleting	The complexity of deleting an element in a quad-tree is $O(logN)$. Where N is $max(w, h)$.

Table 2: Runtime of various operations in Quadtree

IV. RESULTS

We tested our algorithm using two images as shown in Fig.5 and Fig.9 respectively. Firstly, the two images were compressed using quadtree compression, which resulted in obtaining the root node for both the images as shown in Fig.6 and Fig.10. To show the correctness of the root node that represents the average of the whole image by comparing with the online image average color convertor is shown in Fig.8 and Fig.12 . Fig.7 and Fig.11 show first level descendance of the root nodes of both the images.

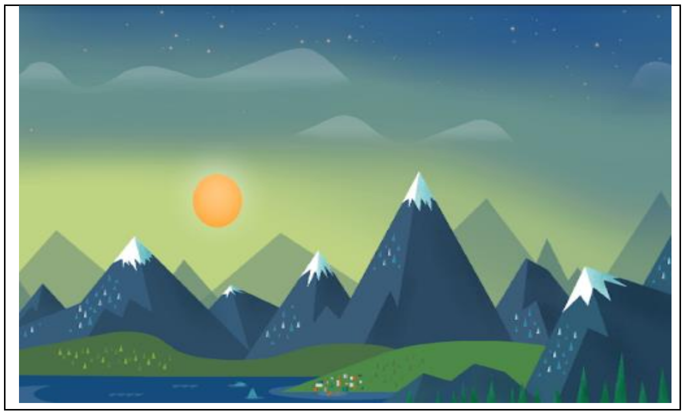


Fig 5: Image 1

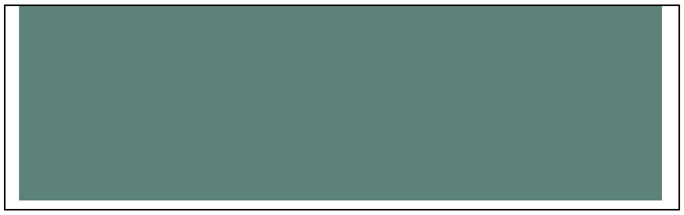


Fig 6: Root Node of Image1

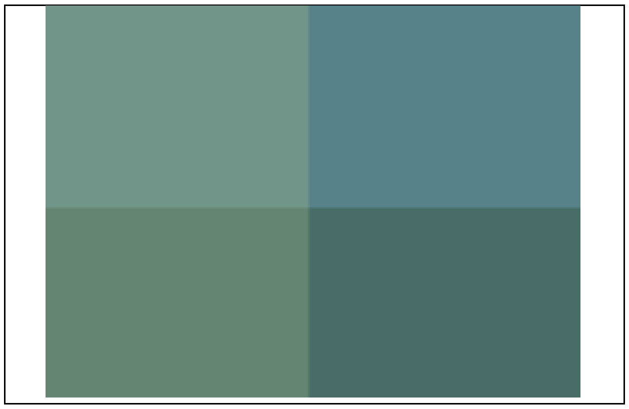
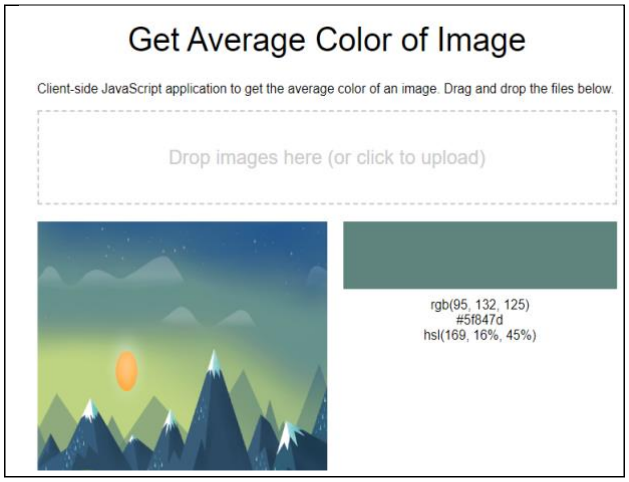


Fig 7: First Level Descendance of Image 1


Get Average Color of Image

Client-side JavaScript application to get the average color of an image. Drag and drop the files below.

rgb(95, 132, 125)
#5f847d
hsl(169, 16%, 45%)

Fig 8: Comparison with online average color tool for Image 1

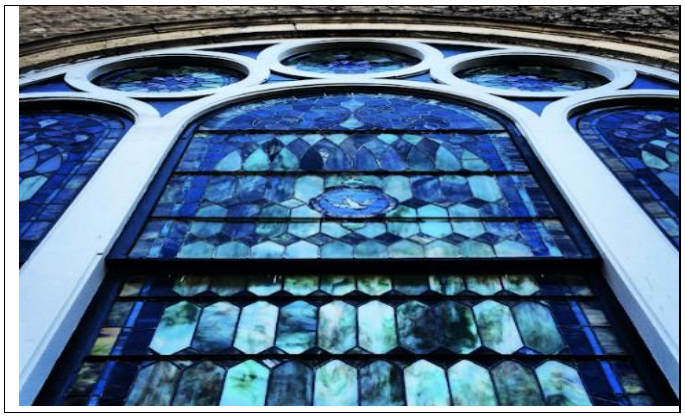


Fig 9: Image 1

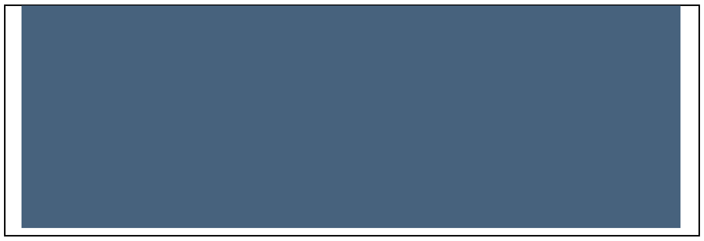


Fig 10: Root Node of Image2

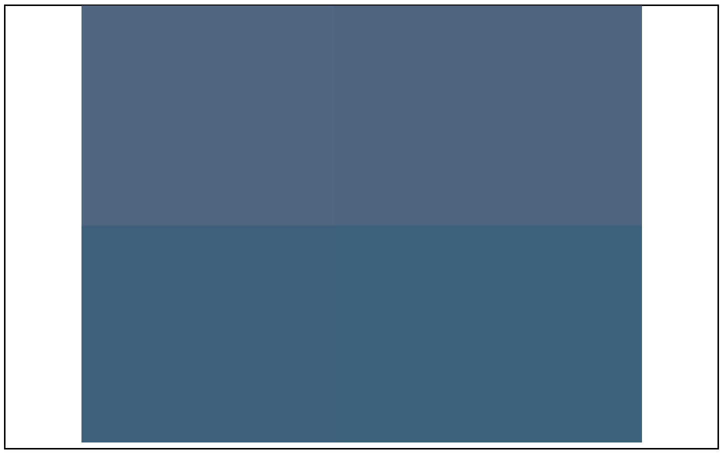
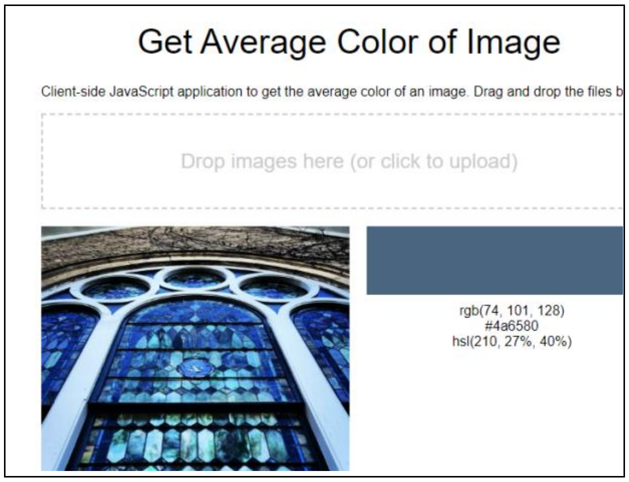


Fig 11: First Level Descendance of Image 2



Get Average Color of Image

Client-side JavaScript application to get the average color of an image. Drag and drop the files b

Drop images here (or click to upload)

rgb(74, 101, 128)
#4a6580
hsl(210, 27%, 40%)

Fig 12: Comparison with online average color tool for Image 2

V. LIMITATIONS

The quadtree construction is optimal when the input image resolution is $2^n * 2^n$. Therefore, scaling of image is done to match default 1024 * 1024 image resolution. This image resolution can be changed in code to any $2^n * 2^n$ combination. This evaluation between quadtree and naïve implementation does not take in quadtree construction time into account. For too large images the recursive call stack may exceed. We have not encountered any such behavior while developing and testing this project. Since we use recursion there is a possibility that the call stack may exhaust and cause runtime exception.

VI. REFERENCES

1. http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall16/Lectures/gis_qdt1.pdf
2. https://en.wikipedia.org/wiki/Quadtree
3. https://www.geeksforgeeks.org/quad-tree/
4. https://www.geeksforgeeks.org/image-manipulation-using-quadtrees/
5. https://support.smartbear.com/testcomplete/docs/_images/testing-with/checkpoints/regions/image-comparison-alg.gif
6. http://www.cs.unca.edu/~reiser/imaging/quadtree.html
7. https://matkl.github.io/average-color/