Image comparison using Quad-Trees

Todorica Matei Thira-Scheitzner Eduard-Paul

June 24, 2022

Abstract

Your abstract.

1 Summary of the documentation

This Document contains the process of developing this app, the algorithm behind the comparison of the images and the necessary functions, the complexity of said algorithm, how the UI works and how to use the app.

2 Some examples to get started

We must find a way to compare images in a efficient and easy manner without consuming too many memory resources and to accesses the differences by grouping together the identical pixels. The main purpose of this Project is to use the concept of Quad-Trees in order to find the differences by dividing the image into Quadrants, finding the groups of identical pixels grouping them together and comparing larger chunks, the algorithm does this process in a swifter way.

3 The objectives of this documentation

Some of the scopes of this document are:

- to explain the logic behind our algorithm
- to elaborate on the programming languages used to create the front-end as well as well as the back-end and the libraries that were necessary
- to explain how the concept of Quad-Trees was implemented in our app
- to elucidate the process of turning an image into a Quad-Tree
- to go in detail about the functions used to manipulate image and generate the differences
- to provide an user manual that illustrates how to install the app and use it
- to elaborate on potential real life applications for this program

4 Requirements for the user

In order for an user to run our App they will need and also the OpenCv library for the functions for visualising and processing the images. The program is compatible with Linux, Windows and macOS.

5 How the Algorithm works

5.1 How we turn an image into a Quad-tree

- The app takes an image from the disk by using its adress to convert it into a Quad-Tree. This process is done by taking the coordinates of the corners of the images and using them to split the image into 4 equal sectors and each sector is split until we reach the level of the pixel and insert each node into the Quad-Tree.
- We initialise the root node with the image converted into a Mat format and initialise the two extremes corners with 0,0 and max.size-1,max.size-1, where max.size is obtained by taking the number of total pixels in the matrix, making its logarithm to base 4 and raising two to the power of it.
- We use the generate function with the following parameters the pairs of int which represent the coordinates of the extremes and the root node. This function splits the image and initialises the nodes with the branch type. In order to find the extremes for each quadrant,we have two pairs A1 and A2, that are modified for each section of the image.
- After we have reached the smallest level possible for a node the algorithm checks with a bool if it has children that are not leaves or children with different contents. If the bool becomes false the node type turns into a leaf and if we are within the borders of the image the content becomes initialised with the content of the quadrant in the top right corner, otherwise it is initialised with a null value. For the bool true case the node, becomes a leaf, takes the value of one of its children and the rest are deleted.

5.2 How we compare two Quad-Trees

- In order to compare two images in the Quad-tree format we need two queues for each of the Trees,to keep track to where we are and two nodes N1 and N2 in order to go through them level by level. We initialise the nodes with the root of each the tree and put them in the front of their respective list. As long as N1 and N2's children are not null the algorithm compares these two, goes through the children pushes them in the back of the queue and after that we extract the items that we compared from their queues. This function return an vector that contains all of the differences.

6 How does the user interface works

The User interface is basically a class that has at its core an Qobject, which is a very powerful tool that helps with the communication between signals from the computer and slots in the program. In order to save the images selected by the user we have used QStringList, which puts the two images in a list to be easily processed, but also in the comparison process. QStringListModel helped us display in a widget format the addresses of the pictures. QImage got us access at the images in a pixel level. We have also used QLabel for displaying the images.

- We created the filepath attribute to set the seletion model for how the UI views the images and how it compares them, after they are selected.
- The function setPixmap obtains the pixel map of the image and there are set two widgets for previewing and comparing the image.
- The function setImage turns the image from a address format to a Pixel Map.

```
void MainWindow::setImage(QString newimg)
{

    delete internalImage;
    internalImage = new QImage(newimg);
    previewLabel->setPixmap(QPixmap::fromImage(*internalImage));
    compareLabel->setPixmap(QPixmap::fromImage(*internalImage));
}
```

- The function SaveSelectedFiles saves the images selected and the function onListViewClicked makes the user capable of previewing the images before comparing them.

```
void MainWindow::on_SaveSelectedFiles_clicked() {
    qDebug()<< filepaths->rowCount();
}

void MainWindow::on_listView_clicked(const QModelIndex &index)
{
    this->setImage(index.data().toString());
    lastSelection=index;
}
```

- Another useful function is On.Compare.View.Clicked which activates the process of comparison between the images.

```
void MainWindow::on_compareView_clicked(const QModelIndex &index)
{
    this->setImage(index.data().toString());
    if(toCompare->contains(index.data().toString()))
    {
        toCompare->removeOne(index.data().toString());
    }
    else
        *toCompare<<index.data().toString();
    qDebug()<<*toCompare;

}
```

- The last function is on.removeSelected.clicked which removes an image from the selection.

```
void MainWindow::on_removeSelected_clicked()
{
    filepaths->removeRow(lastSelection.row());
}
```

7 Programming languages and libraries used

For the back-end part of our program we have used C++ and "opencv2/core.hpp" and "opencv2/imgcodecs.hpp" for manipulating the images, turning them into Quad-Trees, there were also needed the "vector" and "queue" libraries for going through the nodes and comparing them in efficient way and inserting the different ones in a vector. The Qt library more specifically the QT widget helped to provide the signals and slots mechanism for inter-object communication.

8 Functions used to manipulate the Image and the quadTree

8.1 Generate function

The function generate has as parameters two variables of cv::Point type which represent the coordinates top left corner pixel and the bottom right pixel and the TreeNode pointer(cNode) which represents the node from where we start to split . We initialise the children of cNode with an initial life. We use idLeaf to check if a node is a leaf .After that we check if one of the children of node is a leaf and if so we stop and make idLeaf false. Then we compare the content of the children and if the content is different we make IdLeaf false. If after these operations the IdLeaf remains true, we put the value of the first child of cNode into cNode and delete its children, otherwise we calculate the average temperature of the children and we add the differences to the temprature of the node. If the difference is smaller than one, that means we have two cases, one where the quadrant has reached the size of a pixel and we make the cordinates the same with the nw point or two where we reach a null pixel.

```
template<typename dataType>
void ImToQuadTree<dataType>::generate(cv::Point nw, cv::Point se, TreeNode<dataType> *cNode)
    {
    cv::Point A2{0, 0};
    cv::Point A1{0, 0};
    int diff = (se.x - nw.x + 1) / 2;
    if (diff >= 1) {

        for (auto &i: cNode->children) {
            i = new TreeNode<dataType>({0, 0, 0}, nw, se, branch, cNode);
        }
        A1 = nw;
        A2.x = se.x - diff;
        A2.y = se.y - diff;
        generate(A1, A2, cNode->children[0]);

        A1.x = nw.x + diff;
        A1.y = nw.y;
        A2.x = se.x;
        A2.y = se.y - diff;
        generate(A1, A2, cNode->children[1]);


        A1.x = nw.x + diff;
        A1.y = nw.y + diff;
        A2 = se;
        generate(A1, A2, cNode->children[2]);

        A1.x = nw.x;
        A1.y = nw.y + diff;
        A2.x = se.x - diff;
        A2.y = se.y;
        generate(A1, A2, cNode->children[3]);

        bool idLeaf = true;
        for (auto const &i: cNode->children) {
            if (i->type != leaf) {
                idLeaf = false;
                break;
            }
        }

        for (int i = 0; i < 3 && idLeaf; i++) {
            if (cNode->children[i]->content != cNode->children[i + 1]->content)
                idLeaf = false;
        }
        if (idLeaf) {
            cNode->content = cNode->children[0]->content;
            cNode->delChildren();
            cNode->type = leaf;
        } else {
            cNode->propagateProp();
        }


    } else {
        cNode->type = leaf;
        if (nw.x < iMat.rows && nw.y < iMat.cols)
            cNode->content = iMat.at<cv::Vec3b>(nw.x, nw.y);
        else
            cNode->content = cv::Vec3b(0, 0, 0);
    }
```

8.2 The function ColorDistance(Node*)

This function calculates and returns the absolute difference of colors between two Nodes. The content of the first node is temp1 and the content from the second value is temp2. Firstly we add the cDif of both nodes to the total. After that we calculate the absolute difference between temp1 and temp2 and we add each of the values from the content to the total which we return.

```
 inline unsigned long long colorDistance(TreeNode <dataType> * rhs) {
        unsigned long long total = this->cDif;
        total+=rhs->cDif;
        auto temp1 = (cv::Vec3i) this->content;
        auto temp2 = (cv::Vec3i) rhs->content;
        temp1 -= temp2;
        for (int i = 0; i < 3; i++) {
            total += abs(temp1[i]);
        }
        return total;
    }
```

8.3 The function void propagateProp()

This function initialises an avg with 0,0,0 and if the node is not a leaf its content becomes the average of the children's content. Then we iterate through the children and substract it content from the avg and save it in temp. Also we add to cDif temp[0], temp[1], temp[2].

```
void TreeNode<dataType>::propagateProp() {
    cv::Vec3i avg{0, 0, 0};
    if (this->type != leaf) {
        for (const auto &it: this->children) {
            avg += it->content;
        }
        avg /= 4;
        this->content = (cv::Vec3i) avg;
        ///Propagate average
        for (const auto &it: this->children) {

            auto temp = avg - cv::Vec3i(it->content);
            this->cDif += it->cDif;
            this->cDif += abs(temp[0]);
            this->cDif += abs(temp[1]);
            this->cDif += abs(temp[2]);
        }

    }
}
```

8.4 The Function getDiff(ImToQuadTree¡dataType¿ *T1, unsigned long long threshold)

This function compares two QuadTress after as certain threshold we have a vector difValues which memorises the different points, two lists(Line1,Line2) that will help us get through the TreeNode. As long as both of the lists are not empty we find the color distance between the front of the lists(f1,f2). If the distance is greater than the threshold and both f1 and f2 are not leaves we put the children in L1 and L2. If f1 and f2 are both leaves we put f1's coordinates in difValues. In the case f1 is a leaf we make queue unEq for unique nodes we put f2 in front and we iterate through the children of f2 and put them in the front of unEq, else we put in the back of the difValuse the point that coresponds to the front of the unEq queue, pop the queue and make relDif the color distance between f2 and the front of the queue . We do a similar process if f2 is a leaf. We return DifValues

```
ImToQuadTree<dataType>::getDiff(ImToQuadTree<dataType> *T1, unsigned long long threshold) {
    std::vector<std::pair<cv::Point, cv::Point>> difValues;
    auto N1 = this->root;
    auto N2 = T1->root;

    int heightDif = this->maxHeight - T1->maxHeight;
    while (heightDif > 0 && N1->type != leaf) {
        N1 = N1->children[0];
    }
    while (heightDif < 0 && N2->type != leaf) {
        N2 = N2->children[0];
    }
    std::queue<TreeNode<dataType> *> Line1;
    std::queue<TreeNode<dataType> *> Line2;
    Line1.push(N1);
    Line2.push(N2);
    while ((!Line1.empty()) && (!Line2.empty())) {
        auto f1 = Line1.front();
        auto f2 = Line2.front();
        auto relDif = f1->colorDistance(f2);
        if (relDif > threshold) {
            if (f1->type != leaf && f2->type != leaf) {
                for (auto &iter: f1->children) {
                    Line1.push(iter);
                }
                for (auto &iter: f2->children) {
                    Line2.push(iter);
                }
            } else if (f2->type == leaf && f1->type == leaf) {
                difValues.push_back({f1->nw, f1->se});
                ///@warning possible bug here if f1-type is root, i'm assuming no image would
                    be a block of identical pixels however
            } else if (f1->type == leaf) {
                std::queue<TreeNode<dataType> *> unEq;
                unEq.push(f2);
                while (!unEq.empty()) {
                    if (unEq.front()->type != leaf && relDif > threshold) {
                        for (auto &i: unEq.front()->children) {
                            unEq.push(i);
                        }
                    } else {
                        difValues.push_back({unEq.front()->nw, unEq.front()->se});
                    }
                    unEq.pop();
                    ///Undefined behaviour when unEq is empty, however relDif WILL be
                        overwritten by the time it is used again
                    auto relDif = f1->colorDistance(unEq.front());
                }
            } else { //f2->type == leaf
                std::queue<TreeNode<dataType> *> unEq;
                unEq.push(f1);
                while (!unEq.empty()) {
                    if (unEq.front()->type != leaf && relDif > threshold) {
                        for (auto &i: unEq.front()->children) {
                            unEq.push(i);
                        }
                    } else {
                        difValues.push_back({unEq.front()->nw, unEq.front()->se});
                    }
                    unEq.pop();
                    ///Undefined behaviour when unEq is empty, however relDif WILL be
                        overwritten by the time it is used again
                    auto relDif = f2->colorDistance(unEq.front());
                }
            }
        }

        Line1.pop();
        Line2.pop();
    }
    return difValues;
}
```

9 How to install the app

In order to run the app there will be needed the Qt platform with the MSVC2019 compiler and the OpenCv library. You need to go the site https://www.qt.io/download-qt-installer download the installer qt-unified-windows-version.exe and open it. After agreeing with the terms and conditions, select the MSVC2019 component.Now that the components have been installed we need to download the project.We use Qt to open the project and MSVC2019 to compile the project

10 References

- https://doc.qt.io/qt-5/classesandfunctions.html
- https://doc.qt.io/qt-6/model-view-programming.html
- https://doc.qt.io/qt-6/moc.html
- https://docs.opencv.org/4.5.1/d3/d63/classcv$_{1 1}Mat.htmlaa11336b9ac538e0475d840657ce164behttps: //docs.opencv.org/4.5.1/dc/d84/group_{core_{basic.htmlga1e83eafb2d26b3c93f09e8338bcab192}}$

- ”https://docs.opencv.org/4.5.1/d3/dc0/group$_{imgproc_{shape.htmlgaaf0eb9e10bd5adcbd446cd31fea2db68”https://docs.opencv.org/4.x/d3/d52/tutori}}$

References