Implementation of Deletion Algorithm in Quad-Trees

Sergio Mosquera Dopico
sergio.mosquera@est.fib.upc.edu
Universitat Politècnica de Catalunya

***Abstract*—This paper presents an implementation of the tree data structure known as Quad-tree. Quad-trees are the two-dimensional analog of Octrees and are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The paper will cover the explanation of the main methods of this data structure, mainly focusing on the deletion and insertion algorithms, such that their execution may generate a change in the layout of the different quadrants. For the study of this data structure, we will evaluate the computational complexity of some of their operations, validate the correctness of the implementation carrying out some experiments, which will be presented along with the results obtained, and analyze some of the common applications that are widely used in the real world taking advantage of this data structure.**

***Index Terms*—Quadtree, deletion, quadrants, subdivision.**

I. Introduction

Trees are well-known as a non-linear data structure, because they don't store data in a linear way and the information is organized hierarchically. An example of this is the DOM (Document Object Model), an HTML API to organize the elements of an HTML page. For this case, the hierarchical ordering is done by means of tags that identify each element.

The first node of the tree is called the root, and if this root node is connected to another node, the root is then a parent node and the connected node is a child of this node. This applies to any further node with children, but when a node has no children, it is considered a leaf node.

We can find several classification for tree data structures, but the most common one is taking into account for what it would be useful, and the maximum number of children a node can have at most. In this paper we will focus on trees whose nodes can have at most four children, those are the quad-trees.

Each node in the tree represents a quadrant of a given size (quadrants are squares so the width and the length are the same) such that, their size is always smaller than its parent size (if they have parent). The tree itself represents a table (or a map) where a set of points with two coordinates (x,y) need to be distributed, so every time a new point needs to be inserted, we look for the quadrant whose coordinates match the point. Each quadrant can store at most one point itself. Depending on the coordinates of a given point, if a new point lies in that quadrant, we need to subdivide the quadrant into 4 children (also quadrants), each of them representing a cardinal position (NE, SE, SW, NW).

There is a wide variety of applications of this data structure, but the most important and common is for image compression. Each point can have a third value associated, an RGB code, that is, every point represent a pixel with a given colour. What makes this approach interesting is that every time a point lies in a region that already has a point with the same RGB code, we can omit it because the color will be the same and we are storing a pixel less, so following this process for every pixel in the original picture, the size of the new image will be reduced considerably.

The rest of the paper will cover a more detailed explanation of this data structure, the experiments proposed, an evaluation of the results and also more information about different applications.

II. Quad Trees

A quad-tree is a tree data structure composed of a parent node (root) that represents a whole map that is divided into different subsections successively. Each of the nodes in the tree (and therefore, the sections it is divided into) are called quadrants. There a set of restrictions regarding the a quad-tree and its subsequent subdivisions:

1) Initially, the quad-tree consists of an empty quad without points, that is the root node.
2) Every quad holds a set of 4 children, each of them can be either null or a new quad, also it may hold or not a given point.
3) Each child on any quad represents a cardinal position inside the given quad. This is useful to decide in which quad a new point will be stored.
4) If any quad holds a point, it is called a leaf node, and all its children must be null. Is the same in the other way around, if a quad has children, it cannot hold any point, as it is not a leaf node.
5) Whenever a leaf quad (already holds a point) receives a new point to be inserted, it needs to subdivide and move both points (the one it was already holding and the new one) into their corresponding children quads. Depending on the cardinal position that fits better the given points, they will be inserted in one child or another. This process is recursive (i.e. we may need to subdivide also the children) and is performed as many times as needed until we can separate the points into two different quads.
6) If a leaf quad is the only children of its parent quad, the child is deleted and the point it was storing is moved upwards to its parent. This process is recursive (i.e. we may need to move upwards the point several time and remove several quads) and is performed until we reach a quad whose parent has more than one non-null child or either we reach the root.

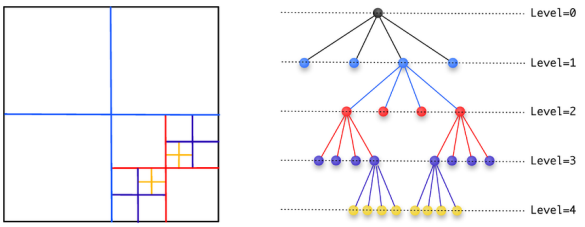

Fig. 1. Quad Tree Representation

A. Representation

In Figure 1 the upper node is the root of the tree, which, initially, has no points on it (i.e. is a leaf node) and represents an empty table. The rest of the tree is organized in levels, such that each level represents a subdivision of a given quadrant.

Note that the nodes without children are the leaf ones and they can be either null or not. If they are not, for sure they hold a point, and if a new point lies on their position, they will need to subdivide so that at least one of their children will become non-null (maybe leaves or maybe not as the may subdivide too).

The traversal in these kind of trees is very similar to the one we can perform on a binary search tree [5], such that the tree is divided into hierarchical levels [6]. To find a node it would be enough to take advantage of Depth First Search, as we can traverse thoroughly the successive nodes knowing the exact path we must follow. Depending on the value we are looking for, it is easy to decide to which quadrant we have to go, as we can see in Figure 2

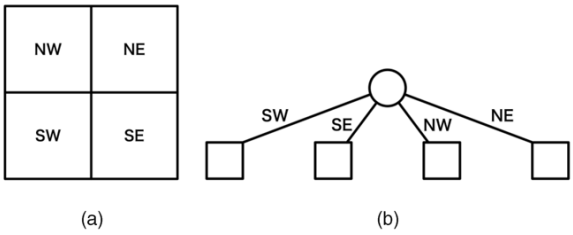

Fig. 2. Cardinality representation in Quad-trees

B. Basic Operations

There are a set of basic operations that every Quad-Tree implementation must have. These operations are the following:

- **Search**. Looks for an element in the tree with a given value. In this case, the value are the coordinates (x,y) of some point. Depending on the approach used to implement the tree, the efficiency could vary (further information about this can be found in Subsection II-C.
- **Insertion**. The main operation to build a Quad-Tree. When we want to insert a new point in the tree, this method will be in charge of traversing the tree comparing the new value to be inserted with all the already existing nodes until we find either a null quad to place the point, or a leaf node that has to be subdivided.
- **Deletion**. The key point of this paper was to implement the deletion algorithm. This operation is very similar to the insertion one, as we need to do the same process but the other way around. In this case, the first thing we need to do is to look for the point we want to delete over the tree (maybe the point doesn't exist). Once we find the point, we need to delete the reference to it on the quadrant it is stored, and also the quadrant. After deleting the point and the containing quadrant, we need to check some things regarding the parent quadrant to check whether we have to join the subdivisions of the parent or keep it as it is. If after removing the quad the parent has only one single non-null children and it is a leaf, the point of that non-null children is transferred to its parent that becomes itself a leaf node, and the children quad is removed. We need to propagate this process upwards.

C. Encoding approaches

When we decide to implement this data structure, we have to make a choice on how to implement it. As we have mentioned in Subsection II-B, depending on the selected approach, the efficiency of the operations may be different. In general, when talking about the complexity of Quad-tree operations we can find something similar as in Table I[1]. This will hold independently in any implementation we choose, but anyway, there exist some improvements to allow the operations to perform faster (caching the information, storing the information as a list, ...).

The implementation we have chosen, although it is not the most efficient one , lets us storing as a cache some important information, as it is the points that have been already inserted. We have decided to store for children inside each node, (NE, SE, SW, NW), such that we store only the reference of a parent node to their children. This representation is similar to a linked list data structure. Also, we have some auxiliary values as the quadrant middle point coordinates such that the task of deciding the children on which to insert a point is easier, a truth value to determine whether the node is removable or not and the size of the current quad.

TABLE I
COMPLEXITIES

Algorithm	Best case	Average case	Worst case
Search	$O(1)$	$O(\log\ n)$	$O(n)$
Insertion	$O(1)$	$O(n\log\ n)$	$O(n\log\ n)$
Deletion	$O(1)$	$O(n\log\ n)$	$O(n\log\ n)$

The following section explains the parts of the implementation that require special attention due to their importance on the project. The different algorithms (search, deletion and insertion) are explained using pseudocode.

[1]Note that n in this case is the height of the tree. The operations depend on the height instead of in the number of nodes.

D. Pseudocode

Algorithm 1 Insertion

```
1: function INSERT(point)
2:     if current_quad is a leaf then
3:         subdivide()
4:     position = get_cardinal_position(point)
5:     if exists children[position] then
6:         children[position].insert(point)
7:     else
8:         children[position] = new Quad(point)
```

As we have explained in Section II-B the insertion algorithm has to decide the most suitable quadrant to insert a point. Once this quadrant is found, we need to check if it is a leaf (so it holds a point). If so, we subdivide it and move the points to their corresponding cardinal positions. This method does not return anything because we keep a record of the already inserted points in a parent class, so this operation can only lead to correct insertion as the incompatibilities are checked out of this scope.

Algorithm 2 Subdividing

```
1: function SUBDIVIDE( )
2:     point = current_quad.point
3:     position = get_cardinality(point)
4:     children[position] = new Quad(point)
```

This auxiliary function is used in the insertion method every time we find a suitable quadrant to store the node. It is only executed if the node is a leaf and what we do is move the point in the current quad downwards to the corresponding child.

Algorithm 3 Deletion

```
1: function DELETE(point)
2:     if current_quad.point == point then
3:         current_quad.point = ∅
4:         Set quad as removable
5:         return True
6:     else
7:         position = get_cardinality(point)
8:         if exists children[position] then
9:             deleted = children[position].delete(point)
10:            if deleted == True then
11:                If children is removable, remove it
12:                If only one non empty quad in children
13:                and the quad not having children,
14:                move its point upwards to current quad
15:            return deleted
16:        return False
```

The same explanation we gave in previous sections for this algorithm are formalized in the above pseudocode.

III. Application of Quad-Trees

There are many applications of this data structure, but the most common one is the image processing and compression. Quadtrees are a very good option when dealing with image processing applications. In this section we will introduce the region quadtree, followed by descriptions of its use in the operations of image union and intersection, and connected component labelling.

A. Region Quadtrees

Region quadtrees follow the general scheme outlined in the previous sections (recursive subdivisions into smaller quads). For region quadtrees, the data stored in a leaf node is some information about the space of the cell it represents (we are already doing that on our implementation which stores the quadrant middle point's coordinates). We will limit our discussion of region quadtrees to binary image data, though region quadtrees and the image processing operations performed on them are just as suitable for colour images. Note the potential savings in terms of space when these trees are used for storing images; images often have many regions of considerable size that have the same colour/value throughout. We cut out several pixels of the same colour that lie in the same region in order to store less information. Therefore, the size of the image being stored is reduced considerably.

B. Image Union/Intersection

One of the advantages of using quadtrees for image manipulation is that the set operations of union and intersection can be done simply and quickly. Given two binary images, the image union (also called overlay) produces an image wherein a pixel is black if either of the input images has a black pixel in the same location. That is, a pixel in the output image is white only when the corresponding pixel in both input images is white, otherwise the output pixel is black.

Rather than do the operation pixel by pixel, we can compute the union more efficiently by leveraging the quadtree's ability to represent multiple pixels with a single node. The algorithm works by traversing the two input quadtrees (T_1 and T_2) while building the output quadtree.

C. Connected Component Labelling

Consider two neighbouring black pixels in a binary image. They are adjacent if they share a bounding horizontal or vertical edge. In general, two black pixels are connected if one can be reached from the other by moving only to adjacent pixels (i.e. there is a path of black pixels between them where each consecutive pair is adjacent). Each maximal set of connected black pixels is a connected component.

The algorithm works in three steps: establish the adjacency relationships between black pixels; process the equivalence relations from the first step to obtain one unique label for each connected component; label the black pixels with the label associated with their connected component.

IV. Work done

In this section I define the steps followed to finish the implementation as well as the experiments that were ran to ensure its correctness. The whole implementation was done in Python3 and no framework was used to carry out this task. In fact, no framework is really needed as we are doing any strong mathematical operation or we need to improve the efficiency of any data structure.

The content of further sections explains the implementation of the system, that is the defining the classes created to model the problem, the experiments that were ran over the implementation, how they were carried out and the specifications of the test environment, and finally, the results derived from the execution of the experiments and how they can be a prove that the data structure built is correct.

A. Implementation

As we have explained in Subsection II-C, there is more than one approach to implement a Quad-tree. We have decided to do it following a Composite design pattern [4], that is, creating a $Quad$ class, such that every quad would hold a reference to four quads (the children) that could be either null or non-null as we have explained before (all belong to the same class).

Above this abstraction on quads, there will be a main class called $Tree$ that holds a reference to a single quad (the root of the tree) and is an interface between the internal behavior of the tree (the logic implemented in the $Quad$ class) and the outer world. Apart from the reference to the root of the tree, this class will hold a list of coordinate points that have already been added to each instance of the quad-tree. This is done for two reasons, the first one is to improve the speed of the searching, insertion and deletion methods. Keeping a record of every point belonging to the tree, the algorithm will not waste time searching for a non-existing point, or adding a point that has been already added.

1) Quad Logic: Inside the $Quad$ class, there are several things to be explained. The first one is the way we store the information. We have mentioned before that each quad stores a reference to four children quads that represent cardinal positions in the quad space. When we are looking for a point or we need to insert one, we need to know in which quadrant is that point lying. How can we do this? Using as a reference the middle point of the quadrant. Whenever a point with coordinates (x_p, y_p) gets to a given quad and we need to decide to which position we should send it, there are four possibilities, taking into account the position of the square center (x_c, y_c):

- $x_p \geq x_c$ and $y_p \geq x_c$, so the point lies in NE quad.[2]
- $x_p \geq x_c$ and $y_p < x_c$, so the point lies in SE quad.
- $x_p < x_c$ and $y_p \geq x_c$, so the point lies in NW quad.
- $x_p < x_c$ and $y_p < x_c$, so the point lies in SW quad.

Another important point is the subdivision of the quadrants, which is very easy to apply taking into account the current implementation. As we already have a list with the children of the quad, we just need to assign a new object to the children whose cardinality corresponds to the one of the point that we want to insert.

For a deletion operation, I used a trick to confirm that, indeed we need to delete some quad. Once we find the point to be deleted, we set the reference to that point to null, mark the current quad as $removable$ and return $True$. As this process is recursive, we are reaching the parent quad, which upon receiving a $True$ returned in the method invocation, checks for removable quads among their children and finds the one we want to remove. This process keeps executing if we need to do a cascading removal on different quads. Applying recursion for this purpose (and also for searching and insertion operations) was strictly necessary taking into account our implementation.

B. Experiments

We propose two experiments. The first one consist on checking the validity of the implementation and the second one consists in checking the complexity of the deletion operation within a quad-tree. The following paragraphs cover the explanation of both experiments as well as the test environment where this experiments were executed.

1) First experiment: For this experiment we are testing our implementation, more concretely we are checking the behavior of the insertion and deletion, simultaneously. The experiment consists in inserting a set of points with coordinates x and y ranging from 0 to 100 (i.e. 100^2 points) and then performing the delete operation of the same values but in different order and store the output of the program. In this way we are able to check two things, the first one is the robustness of the code, that is if it is able to handle huge instances. The second thing, and the most important, is the performance of the code, we are able to check whether the code is working fine or not. The first one is checked immediately (as soon as the program finishes it execution we can say that it is robust enough), but for the second one we need to tune things a bit.

After the execution we will have an output file containing a set of paragraphs. Each paragraph corresponds to the execution of a given operation (first come the insertion ones and then the deletion). These paragraphs will indicate the current situation of the tree, that is, the value of the node and the middle point of the quadrant it belongs to. As we are stating a fixed sized initially for the table represented by the tree (it is size 100×100) the points will be uniformly distributed (as there are also 100×100 points).

The way to check whether the output is correct is splitting the output file when the deletion operation begins and reverse the lines of one of the two files, recalling that deletion operation is executed in reverse order, they will be identical. It is not easy to check the outputs manually as the files had many lines. Instead, we checked this using an online $diff$ tool[3] that shows in seconds the difference (if there was any) between the two files. The results for this experiment will be explained in Section IV-C.

[2] I assumed that the correct way to do the comparisons is assigning also the equals to the rightmost quadrant.

[3] https://www.diffchecker.com

2) Second experiment: The second experiment wants to test the complexity of deletion operation implemented for a quad-tree. The theoretical results are shown in Table I and we want to check whether our implementation fulfills this hypothesis. We will be checking just the average case as the best case is very easy to demonstrate. It happens whenever there are no points yet in the tree, assigning a point to the root node is done in constant time, as it is deleting or finding it.

We need to execute this experiment in second place, as we need to ensure first of all that the behavior of the algorithms is the expected one. Once we are able to verify the implementation with the first experiment, we can start taking measurements for further studies.

To evaluate this experiment we are generating a list with 10.000 random points (without repetition) that will be inserted in a map with size 1000×1000. We will measure the time taken for the deletion for each of the points, and then we will plot the results to check whether we were right or not on our hypothesis. If we are right, the plots should be very similar to $n \log(n)$.

C. Results

The results for the **first experiment** were successful, as the comparison gave no difference between both files, they have exactly the same amount of lines and the output was also the same. A sample output for a given operation (e.g. deletion) is like:

After deleting (12,43):
.....
Point: (1, 1): (x: 0.625000, y: 0.625000)
Point: (2, 2): (x: 1.875000, y: 1.875000)
Point: (4, 4): (x: 3.750000, y: 3.750000)
.....

The output corresponds to the current state of the tree after the execution of the operation. The information printed for each point are the coordinates of each point, along with the coordinates of the quadrant center where the point is stored.

For the **second experiment** we show in Figure 3 the deletion operation complexity and we do not have enough confidence to confirm the hypothesis. Also, there are many outliers that make this task even harder. We can see that there is a growth in the plot but the rate is very slow, although it still may be following an $O(n \log(n))$ behavior, but may be as well following a $O(n^2)$ behavior. I was not able to test in a different situation with bigger instances because a single execution took a lot of time and the amount of outliers became unmanageable.

V. Related work

Although the implementation is completed and the purpose of the experiments are fulfilled, I have just scratched the surface of the quad-trees. As we have presented in Section III, we can take advantage of quad-trees on many different situations. This could be an interesting starting point for future work.

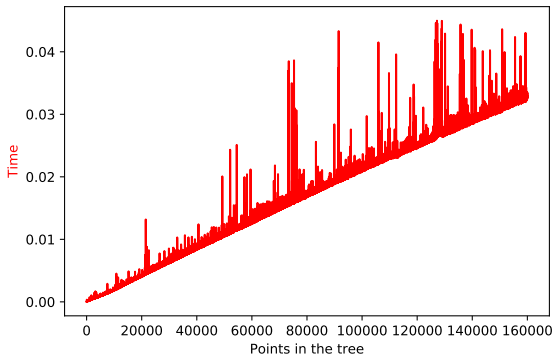

Fig. 3. Deletion Operation

Some examples of related work in this topic [1] [3], can be found in many computer science subjects in the US universities. I am also curious about the application of the current implementation to a real world scenario. These course works I have mentioned include real applications of these data structures, so it would not be hard to follow their approach and get some interesting results.

Another possibility would be looking for an alternative implementation of this data structure. There are several approaches on how to implement this, each with its strengths and weaknesses against the other. Once I would have tried a significant number of different approaches, I would be able to decide which one could be more accurate depending on the context of the problem.

VI. Conclusions

The contents of the paper aim to find an implementation for quad-trees with all its operations, specially the deletion one as it is the most complex to be implemented. The result of this task was successful, and I was able to perform some experiments to verify and validate the correctness of the implementation. Also, these experiments were helpful to check some theoretical aspects like the complexity of the different methods. The experiments and previous studies used in the different papers about this topic were very useful to make some decisions on my way to implement the algorithms or to run the experiments. Also, I found very interesting applications related to this topic that I find very interesting to be implemented in the future.

References

[1] Steve Outdot. Quadtrees: Hierarchical Grids. Stanford. CS468, Mon. Oct. 2nd, 2006

[2] Matthew Shelley. Examining Quadtrees, k-d Trees, and Tile Arrays. School of Computer Science, Carleton University.

[3] Anthony D'Angelo. A Brief Introduction to Quadtrees and Their Applications. School of Computer Science, Carleton University.

[4] Erich Gamma , Richard Helm , Ralph Johnson , John Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995

[5] Sarah F. Frisken, Ronald N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. Mitsubishi Electric Research Laboratories.

[6] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. Computer Science Department, University of Maryland