Cardinal Neighbor Quadtree: a New Quadtree-based Structure for Constant-Time Neighbor Finding

Safwan W. Qasem
Computer Science Department, King Saud University
P.O. Box 51178
Riyadh 11543, Kingdom of Saudi Arabia

Ameur A. Touir
Computer Science Department, King Saud University
P.O. Box 51178
Riyadh 11543, Kingdom of Saudi Arabia

ABSTRACT
This paper presents a new quadtree structure: Cardinal Neighbor Quadtrees (CN-Quadtree), that allows finding neighbor quadrants in constant time regardless of their sizes. Gunter Schrack's solution [1] was able to compute the location code of equal size neighbors in constant-time without guaranteeing their existence. The structure proposed by Aizawa [3][2][3]was able to determine the existence of equal or greater size neighbors and compute their location in constant time, to which the access-time complexity should be added. The proposed structure, the Cardinal Neighbor Quadtree, a pointer based data structure, can determine the existence, and access a smaller, equal or greater size neighbor in constant-time O(1). The time complexity reduction is obtained through the addition of only four pointers per leaf node in the quadtree.

General Terms
Spatial data structures; Image coding; spatial access methods; quadtree;

Keywords
CN-Quadrees; Image coding , neighbor finding.

1. INTRODUCTION
Quadtrees are hierarchical spatial data structures widely used for spatial representation, originally introduced by Finkel et al. in [4]. It is a quaternary tree where each node represents a quadrant in the space. The root represents the whole space, which is the quadrant that encloses the space to be processed. The root represents a quadrant of $2^n x 2^n$ pixels. Several variants of quadtrees have been proposed in the literature to handle different types of data, such as PM Quadtrees, used to manipulate lines and polylines [5], PR Quadtrees [4][2], used to manipulate points, and region Quadtrees [6] or simply quadtrees used to manipulate 2D areas, etc. Since then numerous research focused on Quadtree manipulation and neighbor finding were published [7], [8], [9].

Our work will focus on region Quadtrees, named quadtrees hereafter.

A quadtree represents squared 2D area, which may be an image, an area where a robot can navigate, a map, etc. A quadrant is said to be homogeneous if it contains only one type of information, in which case it is represented as a leaf node in the quadtree. If it contains more than one type of information, the quadrant is said to be heterogeneous and assigned the gray color. It is then subdivided into four equal sized sub-quadrants. The subdivision will stop when all leaves of the quadtree represent homogeneous quadrants. For binary images a leaf node represents a quadrant that is either black or white (

Fig. **1**).

Originally, quadtree was defined as a pointer-based structure. Gargantini [10], proposed another way of coding the quadrants, allowing the structure to be pointerless. Each quadrant has its own location code. It is called talk about a linear quadtree. The location code of a quadrant is built during the subdivision of the quadtree. It consisted on adding to each new sub-quadrant 2 digits in base 2 that represented its relative position in its immediate parent quadrant. 00, 01, 10 and 11, were used to code the positions NW, NE, SW, and SE.

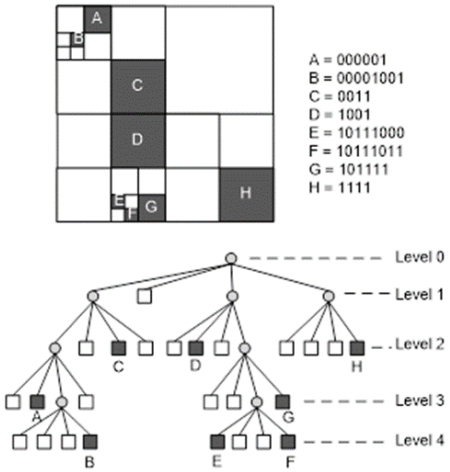


Fig. 1: A space decomposition and its quadtree representation and location codes of black leaf nodes

At the end of the subdivision, each leaf node has a sequence of digits that represent its position by a succession of subdivisions of the original root quadrant. The resulting code was an interleaved coordinate where the bits successively denoted the y and x coordinates. In a pair of bits, the first one indicated the northern child (0) or the southern child (1). Similarly, the second bit denoted the western child (0) or the eastern child (1). Using this coding 00 indicated the Northwestern child, 01, Northeastern child, 10, the Southwestern child and 11 the Southeastern child (Fig. 2).

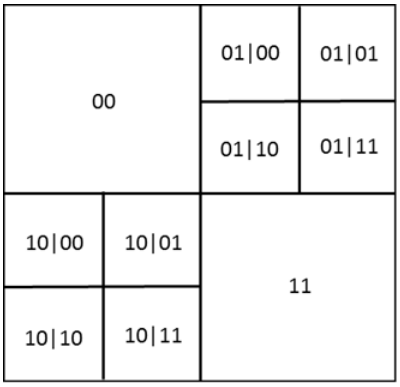


Fig. 2: Binary location code

Quadtrees are used in various applications such as the spatial indexing, GIS, robot path finding, collision detection, and gaming. In all those applications, neighbor finding is one of the fundamental operations Samet [14]. Extensive neighbor finding is required for surface area computation and region filling which are used in computer graphics, boundary determination which is essential for obstacle avoidance in robot navigation.

Klinger [15] proposed a method of moving between adjacent quadrants. Their method requires complex computation rather than following links especially when quadrant are not sibling as well as when they are of different sizes. In the contrary, Hunter et al. [18]0 used explicit links between the node and its adjacent equal size neighbors in the four directions through the use of adjacent trees and the chasing ropes. A rope is a link between adjacent nodes of equal size where at least one the nodes is a leaf node. Thus, not all adjacent quadrants have ropes. In case of absence of a rope for a given node, the method requires ascending the tree until finding a parent having a rope which helps chasing the desired neighbor. The authors improved the neighbor finding algorithm by introducing a 'net' which links all leaf nodes to their neighbors regardless of their size, which adds a huge storage requirement and increases update complexity for such a structure.

Samet [11] proposed a neighbor finding technique in quadtrees. To find the neighbor in a certain direction, the algorithm ascends in the tree until finding a common parent between the node at hand and its neighbors in the given direction, then navigating down the tree to find the adjacent node. Using this algorithm, finding neighbors in a quadtree takes O(n) computational time for the worst case, where n is the number of subdivision of the quadtree (the height of the quadtree).

Using the location code principle, Schrack [1] proposed a constant-time algorithm to find neighbors of equal size in linear quadtrees, using algebraic operations. He introduced an addition operator to add two binary interleaved location codes, and defined translation vector patterns to move from a quadrant to any of its equal-sized neighbors.

For a given direction i, to determine the equal-sized neighbor quadrant m of a quadrant n, Schrack proposed the following equation:

$m_q = n_q \oplus_q \left(\Delta n_i \ll \left(2(r-l)\right)\right),\ i = 0, 1, \ldots, 7$ Where:

r is the resolution of the space in question which is $2^r \times 2^r$ pixels

m_q is the location code of the quadrant m of size : $2^{r-l} \times 2^{r-l}$ pixels where l<r

n_q is the location code of the quadrant n: $2^{r-l} \times 2^{r-l}$ pixels

- Δn_i is a predefined translation vector in the direction i. For r = 3, the translation vectors are defined as follows:

$\Delta n_0 = 000001,$

$\Delta n_1 = 000011,$

$\Delta n_2 = 000010,$

$\Delta n_3 = 000111, etc.$

- $\ll$ is the shift left bits operator
- $\oplus_q$ is the addition operator of binary quadrant location code

This finds the code of equal size neighbors in constant time O(1), but requires O(r) computational time to find a different size neighbor's location-code, where r is the level difference between the two neighbors. To access the data structure corresponding to the computed location code, the algorithm requires the tree traversal cost. Since the neighbor location code is computed based on a translation vector, the algorithm is unable to determine the existence of an equal-size neighbor.

Vörös [8] used the same principle to find smaller sized neighbors by geometric translations in the four directions. His proposed algorithm while able to compute the location-codes of smaller sized neighbors is unable to tell if these neighbors existed or not. An extra tree traversal is still needed to check the existence of the computed nodes.

Yoder [16] extended the computation of same size neighbors location codes from quadtrees (2D) to octrees (3D), and hyper octrees structures.

Aizawa et al. [2][3] proposed an improvement to Schrack's algorithm to solve the existence problem of an equal-size neighbor. They used the translation vectors and the operator proposed by Schrack in addition to a new data structure that keeps the level differences between each quadrant and its adjacent neighbors in the four directions.

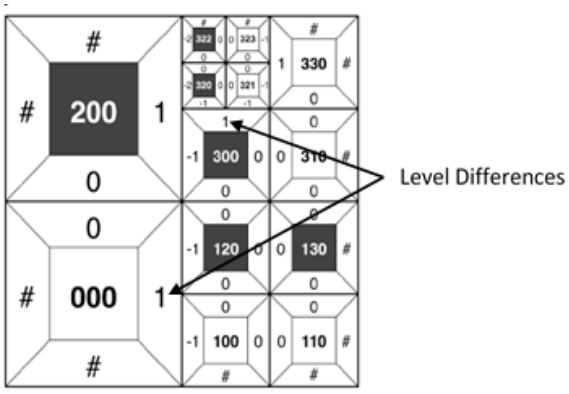


Fig. 3: Example of Level difference between adjacent quadrants [1]

The proposed algorithm builds the data-structure and computes the level differences during the quadtree subdivision (Fig. 3). Using this structure, Aizawa et al. [3] algorithm is able to compute for a quadrant, the geometric location of adjacent neighbors of equal or greater size in constant time, but its access requires a traversal the tree.

Kadowaki et al. [12], proposed a graph based quadtree structure that shows a better efficiency in neighbor finding compared to classical Samet [13] neighbor finding algorithm. This algorithm does not present a predictable behavior as, in some cases, it provides constant-time access to some smaller size neighbors, but needs further processing in other cases. In addition, unlike Schrack [1] and Aizawa [3], the proposed approach does not guarantee a constant-time access to equal or greater size neighbors.

Moreover, once the location code of a neighbor quadrant is computed using Schrack's formula, this algorithm is still unable to access the neighbors' data structure and needs to traverse the tree using this location code to find the neighbor quadrant's data structure.

This paper proposes a new pointer-based quadtree data structure, named CN-Quadtree, and a neighbor-finding algorithm that allow retrieving neighbor quadrants in constant time, whether they are of greater, equal or smaller size. The proposed data structure will hold a minimal set of data that allows a direct access to all neighbors of any given quadrant in the four cardinal directions, North, East, South, and West.

In section 2, the new data structure is presented; the algorithm of building the new data structure during the quadtree subdivision is detailed. In section 3, the neighbor retrieval algorithm is presented. The last section is dedicated to a discussion of the proposed structures and future work.

2. NEIGHBOR-FINDING STRATEGY

The sub-section2 2.1 below, starts by defining the Cardinal Neighbor Quadtree structure; some definitions are formalized and notations introduced in sub-section 2.2. The last sub-section 2.3 explains the different steps of building the Cardinal Neighbor Quadtree.

2.1. CN-Quadtree structure

The new data structure called Cardinal Neighbor Quadtree (CN-Quadtree), is a pointer-based quadtree structure. Each node N of the CN-Quadtree holds four references CNi to an adjacent neighbor quadrant. Each CNi is located in the side i of the quadrant N and is able to identify all the other neighbors located in the same side, where $i \in \{0,1,2,3\}$, and 0, 1, 2, 3 represent respectively the directions West, North, East and South.

These particular neighbors are called Cardinal Neighbors of the quadrant. Fig. 4 shows a representation of the CN quadrant.

Naturally if the neighbor CNi of a quadrant is of greater or equal size, then CNi will be the unique neighbor in the side i. Otherwise, it will be the first neighbor, that will be used to determine all the quadrant's remaining neighbors in that side. The first neighbor (called Cardinal neighbor) will be defined as follows:

- The Western cardinal neighbor is the top-most neighbor node among the western neighbors, noted CN_0.
- The North cardinal neighbor is the left-most neighbor node among the northern neighbors, noted CN_1.
- The Eastern cardinal neighbor is the bottom-most neighbor node among the eastern neighbors, noted CN_2.
- The Southern cardinal neighbor is the right-most neighbor node among the southern neighbors, noted CN_3.

Each node is represented using the data structure in (Fig. 5)

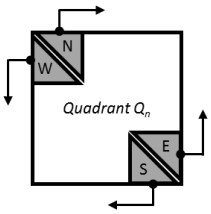


Fig. 4: Each Cardinal Neighbor is used to access all the neighbors in its side

Attribute LocationCode is a quaternary code computed during subdivision where NW, NE, SW, SE sub-quadrants are labeled 0, 1, 2, and 3 respectively. Two binary digits are added at each level of subdivision.

$$\begin{cases} \text{Location\_Code} \\ Size \\ Type = WHITE \mid BLACK \mid GRAY \\ Parent\_Node \\ Children\_Nodes = \{Ch[NW], Ch[NE], Ch[SW], Ch[SE]\} \\ Cardinal\_Neighbors = \{CN_0, CN_1, CN_2, CN_3, \} \end{cases}$$

Fig. 5: CN-Quadtree node data structure

The attribute Size represents the length of one side of the quadrant in pixels, the attribute Type can be one of BLACK, WHITE or GRAY; the attribute Parent_Node is a reference to the quadrant's parent node. If the node is of type GRAY, then the attribute Children_Nodes holds the four quadrant's children. The attribute Cardinal_Neighbors holds references to the quadrant's four Cardinal Neighbors.

2.2. Definitions and notations

This section presents the definition of some functions and operators on the CN-Quadtree that will be used later to compute the cardinal neighbors of each node.

The following functions are defined on the CN- Quadtree data structure:

- $\rho(D)$ returns the immediate parent of the node D. The notation $\rho^2(D)$ denotes the parent of the parent of D. $\rho^0(D) = D$.
- $Size(D)$ returns the side length of node N in pixels.
- $\varphi_i(D)$ returns the cardinal Neighbor of node D in direction i, for $i \in \{0,1,2,3\}$ where 0,1,2,3 represent respectively the directions West, North, East and South. For example, in Fig. 8.d, $\varphi_0(\#12)$ gives the node #031.
- $\varphi_{ij}(D)$ represents the Cardinal Neighbor in the direction i of the Cardinal Neighbor in direction j of

the Node D. $\varphi_{ij}(D) = \varphi_i\left(\varphi_j(D)\right)$. For example, in Fig. 8.d, $\varphi_3\left(\varphi_0(\#12)\right) = \varphi_{30}(\#12) = \#033$.

- $\varphi_i\left(\varphi_i(D)\right)$ will be noted as $\varphi_i^2(D)$. This represents the Cardinal Neighbor in the direction i of the Cardinal Neighbor in direction i of the Node D for $i \in \{0, 1, 2, 3\}$ where 0,1,2,3 represent respectively the directions West, North, East and South and where $\varphi_i^0(D) = D$. $\varphi_i^2(D) = \varphi_i\left(\varphi_i(D)\right)$

In Fig. 8.d, $\varphi_3\left(\varphi_3(\#011)\right) = \varphi_3^2(\#011) = \#031$, and $\varphi_3^0(\#011) = \#011$.

2.3. Building the CN-Quadtree structure

To represent an area, the CN-Quadtree is progressively populated during the subdivision process of the represented environment. CN-Quadtree is built by applying recursively the following three major steps:

1. Decomposing the gray quadrant and updating the parent node following the Z-order traversal.
2. Updating each new four children with their respective Cardinal Neighbors
3. Updating all neighbors accordingly.

2.3.1. Step1: Decomposing the gray quadrant and updating the parent node

The decomposition starts by identifying the first quadrant of type GRAY, decomposing it into four sub-quadrants. The sub-quadrants are added immediately after their parent. Each sub-quadrant first inherits its parent external neighbors then updates the internal neighbors, represented with arrows as shown in Fig. 6.

For example, the neighbor nodes of the NW sub-quadrant are updated as follows:

CN0 = inherited; CN1= inherited; CN2 = NE sub-quadrant; and for CN3 = SW sub-quadrant.

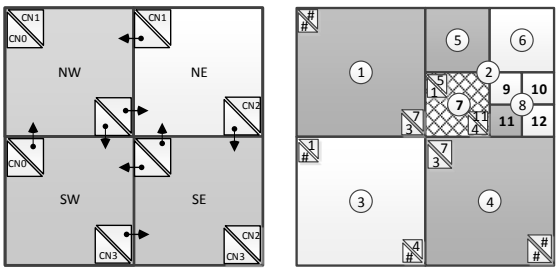


Fig. 6: Updating Cardinal Neighbors on subdivision

Considering the root quadrant R of size 64x64, it will be represented as follows: {#, 64, GRAY, #, {#, #, #, #}, {#, #, #, #}}

On subdivision of the root quadrant R, the four sub-quadrants are added to the quadtree as follows:

- R.Ch[NW] ={0, 32, WHITE, R, {#, #, #, #}, {#, #, 1, 2}}
- R.Ch[NE] ={1, 32, GRAY, R, {#, #, #, #}, {0, #, #, 3}}
- R.Ch[SW] ={2, 32, GRAY, R, {#, #, #, #}, {#, 0, 3, #}}
- R.Ch[SE]={3, 32, GRAY, R, {#, #, #, #}, {2, 1, #, #}}

The sub-quadrant's size is computed by diving the parent node's size by 2.

After finishing decomposition, the parent node is updated to refer to its sub-quadrants:

- R= {#, 64, GRAY, #, {0, 1, 2, 3}, {#, #, #, #}}

2.3.2. Step2: Updating the new children Cardinal Neighbors

The structure of the Quadtree offers interesting characteristics that make easy the use of parents' cardinal neighbors to identify the children's.

On decomposition, the sub-quadrants have been assigned their parent's Cardinal Neighbors on their external borders. Some of the sub-quadrants cardinal neighbors have to be updated to reflect the effective neighbor of the child quadrant rather than its parents'.

The western and northern cardinal neighbors of the parent remain valid for the NW child, while the eastern and southern cardinal neighbors of the parent remain valid for the SE child (Fig. 6). Thus, no change needs to be carried out in these cases. This is not the case for the NE and SW sub-quadrants as their Cardinal Neighbors may be different.

Distinction is made between two cases:

- Parent Cardinal Neighbor size is greater or equal to parent quadrant's size
- Parent Cardinal Neighbor size is smaller than parent quadrant's size

In the first case, no update is needed as the same Cardinal Neighbor applies for the parent and the child quadrants. In the second case, the child's Cardinal Neighbor has to be updated. The case of NE and SW children are explained respectively in the following sub-sections. The section 2.3.2.2, explains how to use the parent Cardinal Neighbors to identify the children ones.

2.3.2.1. Updating Cardinal Neighbors of NE sub-Quadrant

The Fig. 7 shows step by step how the NE sub-Quadrant's Northern Cardinal Neighbor is updated. The NE new sub-Quadrant [#21] first inherits its parent cardinal neighbors during decomposition (Fig. 7.b), which is [#022] while the correct cardinal neighbor is [#03]. Thus, the new north cardinal neighbor [#21].CN1 has to be updated. This is done through a horizontal traversal of the northern neighbors from west to east (Fig. 7.c), by repeating the operation[#21]. $CN_1 \leftarrow \varphi_2\left(\varphi_1(\#21)\right)$ until reaching the first quadrant that is a direct neighbor of the considered quadrant [#21]. In this case [#03] (Fig. 7.d). The value of $\varphi_1(\#21)$ has changed progressively as follows:

Iteration	$\boldsymbol{\varphi_1(\#21)}$
0 (initial value)	[#022]
1	[#023]
2	[#03]

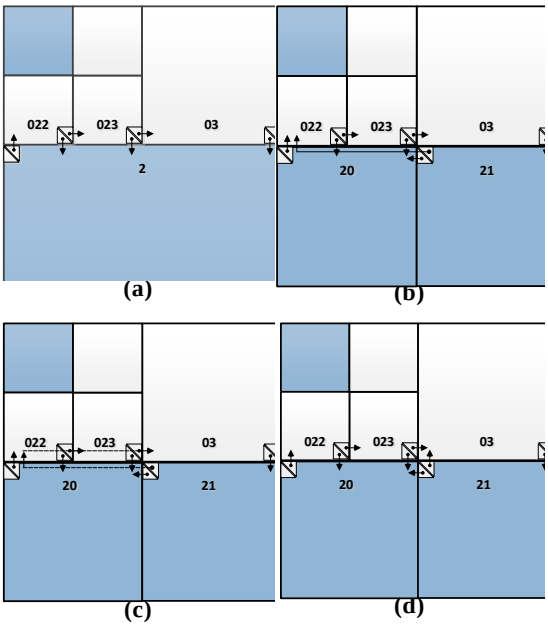


Fig. 7: Updating the NE sub-quadrant's Cardinal Neighbors

2.3.2.2. Methodology of identifying the new Cardinal Neighbors

A very simple method is used to identify the new cardinal neighbor and stop the horizontal traversal. One can notice that after a node decomposition, the two northern child-nodes have the same northern cardinal neighbor if their parent's northern cardinal neighbor is of equal or greater size than the parent node (eq. 1).

Considering a node Q and its 4 children Ci for $i \in \{0,1,2,3\}$ where 0,1,2,3 represent respectively the directions in Z-order, it is obvious that: $Size(Q) = Size(C_0) + Size(C_1)$

If $Size(\varphi_1(Q)) \geq Size(Q)$ then

$$\begin{cases} \varphi_1(C_0) = \varphi_1(Q) \\ \varphi_1(C_1) = \varphi_1(Q) \end{cases} \quad \text{(eq. 1)}$$

In case the parent's northern cardinal neighbor is of smaller size than the parent node (eq. 2), the northern cardinal neighbor is aligned on the left edge of its quadrant. As the distance between the left edge of C_0 and the left edge of C_1 is equal to $Size(C_0)$, the distance between the left edge of $\varphi_1(C_0)$ and $\varphi_1(C_1)$ is also equal to $Size(C_0)$. Thus starting from $\varphi_1(C_0)$, it is sufficient to move to the right for a distance equal to $Size(C_0)$ to reach $\varphi_1(C_1)$.

If $Size(\varphi_1(Q)) < Size(Q)$ then

$$\begin{cases} \varphi_1(C_0) = \varphi_1(Q) \\ \varphi_1(C_1) = \varphi_2^{k+1}(\varphi_1(C_0)) \end{cases} \quad \text{(eq. 2)}$$

Where k is the number of northern neighbor quadrants n_i of C_0, such that $\sum_{i=0}^{k} Size(n_i) = Size(C_0)$.

Notice that if k= 0, $n_0 = \varphi_1(C_0) \wedge Size(n_0) = Size(C_0)$.

Lemma 1:
In the process of west-east traversal, $\varphi_1(C_1)$ is reached when the sum of the sizes of all the traversed neighbors becomes greater than $Size(C_0)$.

In the example mentioned above, starting from [#022], the condition applied once [#03] is reached and

$Size([\#022]) + Size([\#023]) + Size([\#03]) > Size[\#20]$

Thus, [#03] becomes the new North Cardinal Neighbor of C_1.

As the quadrants subdivision is done in Z order, there is no need to update the Eastern cardinal neighbor (CN2). In fact, because of the decomposition order, the eastern neighbor of a quadrant will always be of equal or bigger size, since it is not yet decomposed and thus the parents' eastern cardinal neighbor is itself the Eastern CN of the NE and SE child quadrants. It will need to be updated in case the eastern neighbor is decomposed, as described in the 3rd step of the process.

2.3.2.3. Updating Cardinal Neighbors of SW sub-Quadrant

Fig. 8 shows the process of subdivision of the sub quadrant [#1] into its 4 children Ci and how the West Cardinal Neighbor is updated.

Applying the same principle shown in the previous sections, one can notice that:

$\varphi_0(C_2) = \varphi_3^2(\varphi_0(C_2))=\varphi_3^2(\varphi_0(\#10))= \varphi_3^2(\#011) = \varphi_3(\varphi_3(\#011)) = \varphi_3(\#013) = [\#031]$.

Given a node Q and its 4 children Ci for $i \in \{0,1,2,3\}$ as previously described, it can be noticed that: $Size(Q) = Size(C_0) + Size(C_2)$ and

If $Size(\varphi_0(Q)) \geq Size(Q)$ then

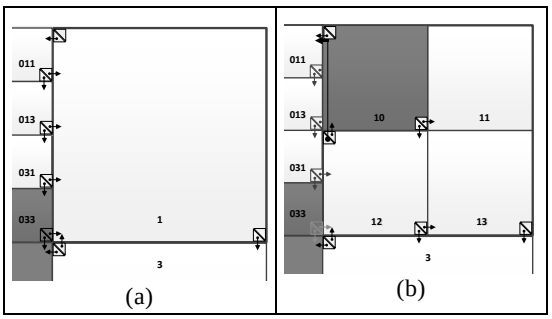
$$\begin{cases} \varphi_0(C_0) = \varphi_0(Q) \\ \varphi_0(C_2) = \varphi_0(Q) \end{cases} \quad \text{(eq. 3)}$$

If $Size(\varphi_0(Q)) < Size(Q)$ then

$$\begin{cases} \varphi_0(C_0) = \varphi_0(Q) \\ \varphi_0(C_2) = \varphi_3^{k+1}(\varphi_0(C_0)) \end{cases} \quad \text{(eq. 4)}$$

Where k is the number of western neighbor quadrants n_i of C_0, such that $\sum_{i=0}^{k} Size(n_i) = Size(C_0)$. Notice that if k= 0, $n_0 = \varphi_0(C_0)$ and $Size(n_0) = Size(C_0)$.

Lemma 2:
In the process of north-south traversal, $\varphi_0(C_2)$ is reached when the sum of the sizes of all the traversed neighbors becomes greater than $Size(C_0)$.



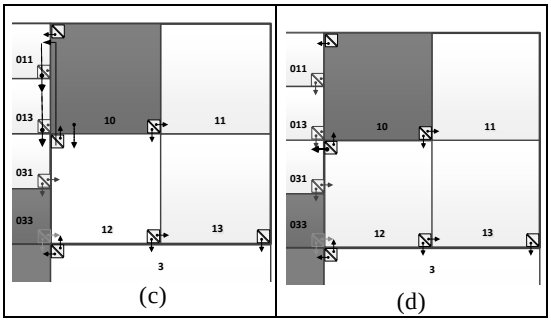


Fig. 8: Updating the SW sub-quadrant's Cardinal Neighbors

As the quadrants subdivision is done in Z order, there is no need to update the Southern cardinal neighbor (CN3). In fact, because of the decomposition order, the southern neighbor of a quadrant will always be of equal or bigger size, since it is not yet decomposed and thus the parents' southern cardinal neighbor is itself the Southern CN of the SW and SE child quadrants. It will be updated in case the southern neighbor is decomposed as described in the following section.

2.3.3 Step3: Updating all neighbors accordingly

After the decomposition of a quadrant, all its neighbors in the four directions must be informed of the change so that they can update their own cardinal neighbors accordingly.

On each direction, a full traversal of the neighbors should be performed. In every quadrant where a reference to the parent quadrant is stored as the Cardinal Neighbor, it should be replaced by one of its children created after the decomposition.To minimize the effort, the step 3 and step 2 will be performed in a single traversal on each side.

On the northern side, during the traversal described in section Step 2.a (cf. 2.3.2.1. Updating Cardinal Neighbors of NE sub-Quadrant) and represented in Fig. 7, each visited neighbor that had quadrant [#2] as its southern Cardinal neighbor must update it to one of its closest children [#20] and [#21]. The traversal will stop when a quadrant is encountered that has a southern CN different from the parent; in this case [#2].

The same process is applied to western neighbors during the same traversal described in section Step 2.c.

Considering the example represented in Fig. 8, all western neighbors of [#1] will update their eastern cardinal neighbors from [#1] to either [#10] or [#12].

As the quadrant decomposition is performed in Z order, a quadrant is always decomposed before its eastern and southern neighbors. Thus, at the moment of decomposition, any quadrant will have only one neighbor in the eastern and southern directions which will be itself the cardinal neighbor. Therefore, one assignment is sufficient to update each of the eastern and southern side neighborhoods of a quadrant.

To update the eastern CN of a quadrant Q that is being decomposed: Q.CN2.CN0=Q.Ch[NE]

To update the southern CN of a quadrant Q that is being decomposed: Q.CN3.CN1=Q.Ch[SE]

This neighbor will be of equal or greater size. So, only one assignment will be enough.

3. NEIGHBOR-FINDING IN CN-QUADTREE

Once the CN-Quadtree completely built, finding a neighbor of a specific quadrant is straightforward using its cardinal neighbors.

3.1. Finding north neighbors

To find the north neighbor of a quadrant Q, it is sufficient to access its north cardinal neighbor ($\varphi_1(Q)$) as it is the first north neighbor. From this initial point, the kth north neighbor of Q can be retrieved by a simple traversal from west to east using the function $\varphi_2^{k-1}(\varphi_1(Q))$.

All north neighbors (Say N) of a quadrant Q can be retrieved using the following algorithm:

$N = \varphi_1(Q)$

If $(Size(N) < Size(Q))$

While $(\varphi_3(N) == Q)$

$N = \varphi_2(N)$;

If the north neighbor of Q is of equal or greater size than Q, it will retrieved in 1 step. Otherwise, it will be retrieved in as many steps as its rank among of neighbors starting from the most western one.

3.2. General neighbor finding

In all the other directions, neighbor finding follows the same principle.

On the western side, the neighbors are found starting from the western CN and moving to the south. For the eastern side, the neighbors are identified starting from the Eastern CN and moving north, and last for the southern side, the neighbors are identified starting from the southern CN and moving to the west.

After defining the indices 0, 1, 2 and 3 to represent respectively the direction west, north, east and south, the algorithm can be rewritten in a general case as follows:

$N = \varphi_i(Q)$

If $(Size(N) < Size(Q))$

While $(\varphi_{((i+2)\ mod\ 4)}(N) == Q)$

$N = \varphi_{(3-i)}(N)$;

4. EXPERIMENTAL RESULTS

The major benefit of the proposed algorithm is to provide a direct access to the first neighbor in each direction and accessing the remaining ones sequentially. This advantage is not reached without memory overhead. It is then necessary to compare it with the other similar algorithms on the two aspects of performance and memory requirements.

For experimental evaluation, it is proposed to compare the new method (CN-Quadtree) with Aizawa algorithm, as it is the most recent method of quadtree neighbor finding that can compute the location code of the neighbors at constant time. It was demonstrated in [3], that the new method outperformed the classical Samet algorithm [11]. Aizawa's method will be referred to as AZW hereafter.

To perform the experimentation, a quadtree-based image coding application using C++ in a Linux environment has

been implemented. This application was used as a common basis for the two algorithms. The node structure of the quadtree was modified in each one of the two applications to include the additional fields required by each specific algorithms, and to include the necessary behavior for each method. The experiments were executed one at a time, on the same computing environment.

4.1. Methodology

The experimental evaluation aims to compare the time required to access neighbors of a pre-determined set of nodes. Because of the techniques used by the different algorithms, The cases where the nodes have only one neighbor on each side will be differentiated from the general case, because when a node has only one neighbor on each side, CNQT will plainly benefit from the direct access to that neighbor, while AZW algorithm will benefit from the constant time computation of the neighbor's location code. Both algorithms will have to use additional processing in the case of multiple neighbors.

The experiment consisted on selecting randomly a set of nodes and accessing all its neighbors using the two methods. The two algorithms accessed exactly the same set of nodes and logged only the time necessary to access the neighbors. These experiments were repeated using several images selected from Signal and Image Processing Institute's image database of the University of Southern California (USC-SIPI). For each considered image, the methodology used in this experiment, is as follows:

- Build the quadtrees of the image.
- Randomly select one thousand leaf nodes
- Using both considered methods:
- For each of the selected nodes, compute the time necessary to access all its neighbors.
- Compute the average access time for the nodes with a single neighbor per side.
- Compute the average access time for the nodes with a multiple neighbors per side.

4.2. Results and discussion

The test base images that was used is the SIPI image database of the University of Southern California [17] . The selected images were of 512x512 and 1024x1024 pixels resolution. Figure lists a sample of these images.

Figure 9: Samples of images used to obtain experimental results

These images have been converted to black and white using the threshold method.

In the first experiment, only the nodes having 4 neighbors, one on each side, have been selected. The average time to access the 4 neighbors using the three selected methods were computed and compared. As it is clearly shown in Figure 10, CNQT method largely outperforms the reference methods. Table 1 shows that CNQT provides about 75% gain compared to AZW method. Figure 10 illustrates this performance gain for each one of the five considered images.

Table 1: Performance gain of CNQT compared to AZW methods

Image	CNQT vs. AZW
Satellite	78%
Native-American	74%
Roof	78%
Airport	78%
Pentagon	79%

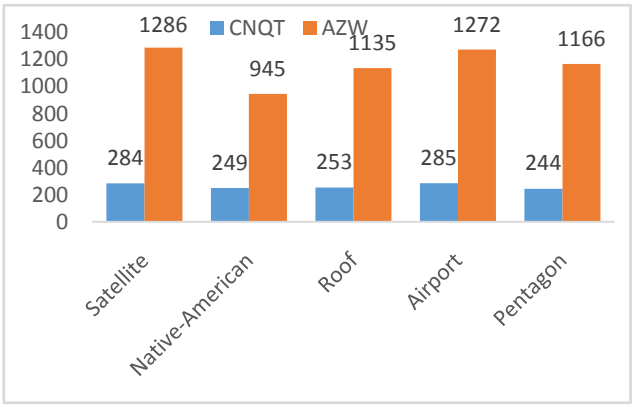


Figure 10: Comparison of Average Access time (in nanoseconds) for 4 neighbors

The second experiment of performance analysis considered the average access time to all neighbors of all selected nodes. As the selected nodes, had a number of neighbors varying from 2 to 70 per node, the average time to access each neighbor node individually was computed. The results are reported in Figure 11 and the gain percentages Table 2.

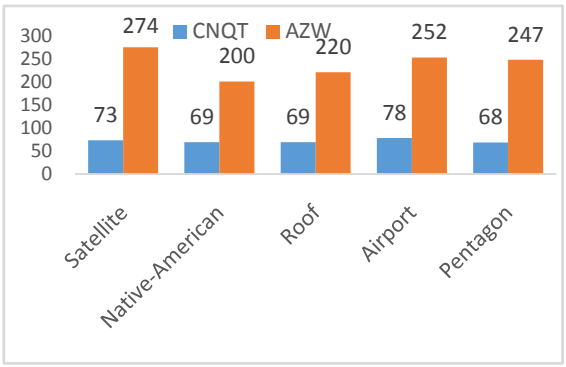


Figure 11: Comparison of Average Access time for a single neighbor node

Table 2: Performance gain of CNQT compared to AZW

Image	CNQT vs. AZW
Satellite	73%
Native-American	65%
Roof	68%
Airport	69%
Pentagon	73%

The experiments were conducted using a set of images of different contents. Some of them show some repetitive patens, whereas others have a random distribution of pixels. Some images resulted in a large set of small quadrants due to a deep subdivision level while others decomposed into bigger size quadrants.

Figure 11 shows that CNQT gives almost constant access time in all type of images, wich clearly reflects the fact that each neighbor is retrieved in constant time. AZW method present some variation of average time from an image to another. This can be explained by the specific images decomposition and the nature of the method. AZW method has to compute the location code of the neighbor and launch the search process from the root of the tree. If the selected nodes are in majority not deep in the image quadtree, the average access time using AZW method may be shorter.

4.3. Memory space impact

In this paragraph we'll propose a comparison of the two methods with regards to memory occupation. Both algorithms have some common fields in their data structure. They are namely: the location code, the node's level, the nodes color and pointer references to the parent node and to the four children nodes.

Both methods have the same structure for the gray / intermediary node structure but for the leaf node structure, each method has its own particularity. The common part of the node structure can be represented using 26 bytes in a 32 bit machine.

In addition to the common structure, the AZW algorithm adds 4 fields to represent the level differences with the neighboring nodes in each of the four directions. This can be represented using four additional bytes. The CNQT algorithm uses the common structure for the gray quadrants and adds 4 pointers in the leaf nodes to access the cardinal neighbors. This requires four additional pointers.

In a 32 bit machine, the CNQT method will require in average 38 bytes per node compared to 29 bytes needed by AZW. This leads to memory increase of 31%.

5. DISCUSSION AND CONCLUSION

In this paper, a new data structure, the CN-Quadtree was presented. In the building phase of the quadtree, four cardinal neighbors are determined for each quadrant. These particular neighbors will allow access to all the neighbors in each side in the retrieval phase. This access is performed in constant time for the cardinal neighbors regardless of their size.

If the neighbor is of equal or greater size, it will be itself stored in the cardinal neighbor and accessed immediately. For smaller size neighbors, as all the neighbors are solutions, they can also be accessed by traversing the list of neighbors whose head is the cardinal neighbor. The experimental results demonstrated that the proposed data structure and algorithm clearly outperforms the algorithm proposed by Aizawa et al. [2][3].

The proposed method required an increase of 31% in the quadtree memory occupation, and allowed 69.7% average reduction in the neighbor access time as highlighted by the experimental results.

The proposed solution, the CN-Quadtree, will present opportunities for new optimizations in the field of image analysis and processing and the field of robots path planning and navigation.

6. ACKNOWLEDGMENTS

This research has been supported by the research center of the college of Computer and Information Sciences at King Saud University. Project reference RC140204.

7. REFERENCES

[1] Schrack G 1992 Finding Neighbors of Equal Size in Linear Quadtrees and Octrees in Constant Time, CVGIP: Image Understanding, 55: 221-230.

[2] Aizawa K, Motomura K, Kimura S, Kadowaki R and Fan J 2008 Constant Time Neighbor Finding in Quadtrees: An Experimental Result, in: Proc. 3rd International Symposium on Communications, Control and Signal Processing, Malta.

[3] Aizawa K and Tanaka S 2009 A Constant-Time Algorithm for Finding Neighbors in Quadtrees, IEEE Trans. Pattern Analysis and Machine Intelligence, 31(7), 1178-1183.

[4] Finkel R A and Bentley J L 1974 Quad Trees: A Data Structure for Retrieval on Composite Keys, Acta Informatica, 4: 1-9.

[5] Samet H and Webber R E 1985 Storing a Collection of Polygons Using Quadtrees, ACM Transactions on Graphics 4(3): 182-222.

[6] Samet H 1985 A Top-Down Quadtree Traversal Algorithm, IEEE Trans. Pattern Analysis and Machine Intelligence 7: 94-98.

[7] Fuhrmann D R 1988 Quadtree Traversal Algorithms for Pointer-Based and Depth-First Representations, IEEE Trans. Pattern Analysis and Machine Intelligence, 10: 955-960.

[8] Vörös J 1997 A Strategy for Repetitive Neighbor Finding in Images Represented by Quadtrees, Pattern Recognition Letters, 18:955-962.

[9] Frisken S F and Perry R N 2002 Simple and Efficient Traversal Methods for Quadtrees and Octrees, The Journal of Graphics Tools, 7(3): 1-11.

[10] Gargantini I 1982 An Effective Way to Represent Quadtrees, Comm. ACM, 25: 905-910.

[11] Samet H 1982 Neighbor finding techniques for images represented by quadtrees, Computer Graphics and Image Processing, 18: 35-57.

[12] Kadowaki R, Motomura K, Ohkura S and Aizawa K 2010 Graphs Representing Quadtree Structures using Eight Edges, Proc. Int. Symposium on Communications, Control and Signal Processing, Cyprus.

[13] Samet H 1984 The quadtree and related hierarchical data structures, ACM Computing Surveys. 16(2): 187-260.

[14] Samet H 1990 Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Addison-Wesley, Boston.

[15] Klinger, A., and M.L. Rhodes, 1979. Organization and access of image data by areas, LEEE Trans. Pattern Anal. Mach. Lntell., PAMI-1:5& 60.

[16] Yoder R and Bloniarz P 2006 A Practical Algorithm for Computing Neighbors in Quadtrees, Octrees, and Hyperoctrees, Proc. Int. Conf. on Modeling, Simulation, and Visualization Methods, Las Vegas, USA.

[17] USC-SIPI Image Database, last accessed March 2015, http://sipi.usc.edu/database/,

[18] Hunter, G.M., and K. Steiglitz, 1979a. Operations on images using quad trees. IEEE Transactions on Pattern Analysis and Machine Intelligence,. 1, 2 (Apr.), 145-153.

[19] Hunter, G.M., and K. Steiglitz, 1979b. Linear transformation of pictures represented by quadtrees. Comput. Gr. Image Process. 10, 3 (July), 289-296.