A Brief Introduction to Quadtrees and Their Applications

Anthony D'Angelo[*]

Abstract

We briefly introduce the *quadtree* data structure, some of its variants (region, point, point-region, edge, polygonal map, and compressed), as well as some applications to problems in computer science including image processing, spatial queries, and mesh generation. This presentation is mainly a collection of selected materials from surveys by Aluru [1] and Samet [13, 14], as well as a textbook by Har-Peled [5].

1 Introduction

The *quadtree* is a hierarchical spatial data structure. It is a tree in which each level corresponds to a further refinement of the space under consideration. Though there are many types of quadtrees and quadtrees can be generalized to any dimension[1], the idea is always a recursive decomposition of space that helps us store only the important or interesting information about the space.

In this discussion we will focus on 2-Dimensional quadtrees. Typically, the space under consideration is a square normalized to be the unit square and it lies in the 2-Dimensional Euclidean plane. We start by creating a root node for the tree. The root represents the unit square. Like the root node, each node in the tree corresponds to an area of the plane. We say that a node in the tree represents a cell. Each internal node is further subdivided into four quadrants (usually squares) as long as there is interesting data in the cell for which more refinement is desired, but potentially limited to a pre-defined resolution size. Each leaf node in the tree corresponds to a cell that is not subdivided further. The data associated with a leaf cell varies by application, but the leaf cell represents a "unit of interesting spatial information". The height of quadtrees that follow this decomposition strategy is sensitive to and dependent on the spatial distribution of interesting cells.

1.1 Space-Filling Curves

Informally, a space-filling curve is one whose range is every point in the space being considered. For example, if we divide a unit square into a grid whose cells are all equal-sized squares, a space-filling curve passes through all the cells once. Of particular interest for us is the Morton curve, also known as the z-order curve [8]. There are a few variations of this curve, but we will be using it as outlined by Morton. For a 2x2 square, the z-order is South-West (SW), North-West (NW), South-East (SE), and North-East (NE).[2] The 4x4 square's z-order curve has the 2x2 square's curve copied into each of its 2x2 quadrants, and they are connected in the same z-order. The $2^k \times 2^k$ square's curve is similarly created by four copies of the $2^{k-1} \times 2^{k-1}$ curve (see Figure 1).

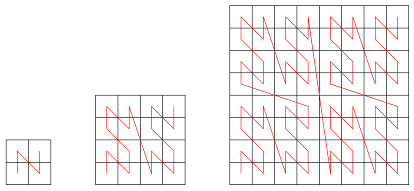
Figure 1: Z-curves for 2x2, 4x4, and 8x8 grids (image from [1])

The z-order defines a total order on the cells of any subdivision of a quadtree. Furthermore, mapping from cell coordinates to the cell's rank in the z-order is rather simple and relatively quick. Consider a grid where the origin of each cell is the bottom-left corner. to convert from the coordinate of the cell to the rank, we take the binary representation of the x and y coordinates ($x_1x_2 \dots x_k$ and $y_1y_2 \dots y_k$ respectively), then interleave the binary numbers ($x_1y_1x_2y_2 \dots x_ky_k$). The decimal representation of that number is the cell's rank. Mapping back to coordinates is done similarly.

2 Image Processing

Quadtrees have lent themselves well to 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*.

[*]School of Computer Science, Carleton University, anthony.dangelo@carleton.ca

[1]A 3-D quadtree is an *octree* and higher dimensional versions are called *hyperoctrees* [1].

[2]The curve starts in the bottom-left corner of the square and finishes in the top-right.

2.1 Region Quadtrees

Region quadtrees follow the general scheme outlined in the introduction (recursive subdivisions into squares). For region quadtrees, the data stored in a leaf node is some information about the space of the cell it represents (e.g. average height value for the cell).

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 [1, 14]. 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. Rather than store a big 2-D array of every pixel in the image, a quadtree can capture the same information potentially many divisive levels higher than the pixel-resolution sized cells that we would otherwise require. The tree resolution and overall size is bounded by the pixel and image sizes.

2.2 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 [1, 6, 7, 16].

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. Informally, the algorithm is as follows. Consider the nodes $v_1 \in T_1$ and $v_2 \in T_2$ corresponding to the same region in the images.

- If v_1 is *black*, we make the corresponding position in the output quadtree a *black leaf*.
- If v_1 is *white*, we copy the *subtree rooted at* v_2 into the corresponding position in the output quadtree.
- If v_1 and v_2 are both *grey*, we insert a *grey* node in the corresponding position of the output quadtree and fill in its subtree by considering the corresponding children of v_1 and v_2.

While this algorithm works, it does not by itself guarantee a minimally sized quadtree. For example, consider the result if we were to union a checkerboard (where every tile is a pixel) of size $2^k \times 2^k$ with its complement. The result is a giant black square which should be represented by a quadtree with just the root node (coloured black), but instead the algorithm produces a full 4-ary tree of depth k. To fix this, we perform a bottom-up traversal of the resulting quadtree where we check if the four children nodes have the same colour, in which case we replace their parent with a leaf of the same colour [1].

The intersection of two images is almost the same algorithm. One way to think about the intersection of the two images is that we are doing a union with respect to the *white* pixels. As such, to perform the intersection we swap the mentions of black and white in the union algorithm.

2.3 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*. Using the quadtree-representation of images, Samet [11] showed we can find and label these connected components in time proportional to the size of the quadtree [1, 13]. This algorithm can also be used for polygon colouring [13].

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.

To simplify the discussion, let us assume the children of a node in the quadtree follow the z-order (SW, NW, SE, NE). Since we can count on this structure, for any cell we know how to navigate the quadtree to find the adjacent cells in the different levels of the hierarchy. Step one is accomplished with a post-order traversal of the quadtree. For each black leaf v we look at the node or nodes representing cells that are Northern neighbours and Eastern neighbours (i.e. the Northern and Eastern cells that share edges with the cell of v). Since the tree is organized in z-order, we have the invariant that the Southern and Western neighbours have already been taken care of and accounted for. Let the Northern or Eastern neighbour currently under consideration be u. If u represents black pixels:

- If only one of u or v has a label, assign that label to the other cell.
- If neither of them have labels, create one and assign it to both of them.
- If u and v have different labels, record this label equivalence and move on.

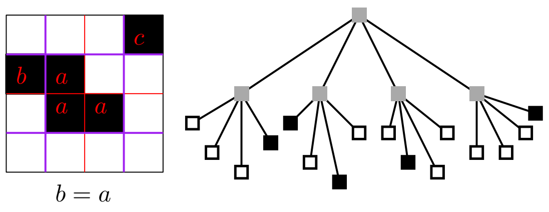

Figure 2: A binary image and the region quadtree representing it. A potential outcome of the first step of the connected component labelling algorithm (if the cell of b has further subdivisions). There has been a noted equivalence between a and b.

Figure 2 shows a hypothetical result at the end of the first step assuming the node labelled b has further subdivisions.

Step two can be accomplished using the *union-find* data structure [18]. We start with each unique label as a separate set. For every equivalence relation noted in the first step, we union the corresponding sets. Afterwards, each distinct remaining set will be associated with a distinct connected component in the image.

Step three performs another post-order traversal. This time, for each black node v we use the union-find's *find* operation (with the old label of v) to find and assign v its new label (associated with the connected component of which v is part).

3 Spatial Queries

Being a data structure that subdivides spaces and represents spatial points, it is no surprise that quadtrees can answer spatial queries efficiently. We begin by introducing the *point* and *point-region* quadtrees used for point data as well as some queries we can perform on them, followed by the *edge* and *polygonal map* quadtrees used for representing polygons and their application in point location queries.

3.1 Point and Point-Region (PR) Quadtrees

Point quadtrees are a generalization of the binary tree for multi-dimensional point data [4]. The manner in which the plane is divided up for use with the point quadtree is markedly different than the most commonly used method of dividing each cell into four equal-sized squares whose side length is a power of two. Point quadtrees are worth mentioning for completeness, but they have been surpassed by k-d trees [2] as tools for generalized binary search [1].

Point quadtrees are constructed as follows. Given the next point to insert, we find the cell in which it lies and add it to the tree. The new point is added such that the cell that contains it is divided into quadrants by the vertical and horizontal lines that run through the point. Consequently, cells are rectangular but not necessarily square. In these trees, each node contains one of the input points.

Since the division of the plane is decided by the order of point-insertion, the tree's height is sensitive to and dependent on insertion order. Inserting in a "bad" order can lead to a tree of height linear in the number of input points (at which point it becomes a linked-list). If the point-set is static, pre-processing can be done to create a tree of balanced height.

Point-region (PR) quadtrees [10, 12] are very similar to region quadtrees. The difference is the type of information stored about the cells. In a region quadtree, a uniform value was stored that applied to the entire area of the cell of a leaf. The cells of a PR quadtree, however, store a list of points that exist within the cell of a leaf.[3] As mentioned previously, for trees following this decomposition strategy the height depends on the spatial distribution of the points. Like the point quadtree, the PR quadtree may also have a linear height when given a "bad" set.

3.2 Range Query

Given a PR quadtree and a query region R (which is possibly open-ended), the result of a range query is all of the stored points in the PR quadtree that lie within R. This operation may correspond to finding records in a database that meet a certain criteria. To do the range query, first we find the smallest node/cell u in the tree that completely contains R. For each child cell v of u:

- If v lies completely within R, add the points of v's subtree to the query result and continue the query.
- If v and R do not intersect, discard v and continue with the other children.
- If v and R intersect, process the children of v similarly in turn (brute-force check leaves).

3.3 Spherical Region Query

Given a PR quadtree, a query point q, and a distance $r > 0$, the result of a spherical region query is all of the stored points in the tree that lie in the circle centred at q with radius r. An example query may be a request for all hospitals within 10 km of someone's current location. The query is answered by walking down the tree from the root to the cell containing q while exploring and ruling out cells on the way. For each cell v we have not yet ruled out:

[3]The number of points used as a stopping criterion for further subdivision varies by application, but let us assume a leaf stores at most one point.

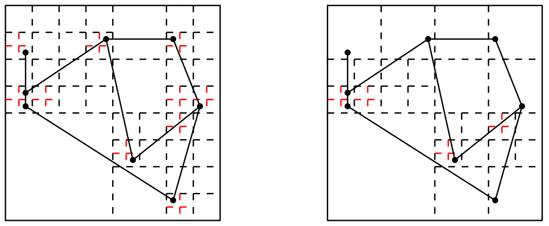
Figure 3: On the left is the subdivided plane for an edge quadtree, whereas on the right we have the subdivision for a PM quadtree. Note the PM quadtree saves space by not always subdividing areas with corners. (Images from [14])

- If the farthest spatial point[4] in v is less than r from q, add all stored points in v to the query result, remove v from consideration, and continue the query.

- If the closest spatial point in v is more than r from q, remove v from consideration, and continue the query.

- Otherwise, remove v and add its children for consideration (brute-force check the data points of a leaf's cell).

3.4 Edge and Polygonal Map (PM) Quadtrees

Quadtrees can also be used to capture polygonal data in the form of linear features. The *edge* [17, 19] and *polygonal map (PM)* [9, 15] quadtrees work almost the same way, approximating polygons by straight line segments. Both trees generally work by subdividing the space until there is a single line segment per cell. Near corners/vertices, edge quadtrees will continue dividing until they reach their maximum level of decomposition. The big difference with PM quadtrees is that the cell under consideration is not subdivided if the segments meet at a vertex in the cell (see Figure 3 to see the difference in subdivision).

3.5 Point Location

Given a quadtree for a polygonal map and a query point q, the result of a point location query is the polygon in which q lies. An example of such a query is a traveller asking a map service which city they are currently in. The quadtree we will consider here is a slight variation of the ones seen above. The difference is that the stopping criterion for the subdivision is when a cell is intersected by a constant number of polygons [5]. With such a quadtree, the point location query is rather simple: find the leaf in the quadtree that contains q, then check which of the $O(1)$ polygons contain q (in other words, brute-force check against the polygons in the leaf of q).

[4]Not the location of a data point from our point set that lies in v, rather the farthest point in the cell to q.

4 Compressed Quadtrees

In this section we will look at the use of compressed quadtrees [5]. If we were to store every node corresponding to a subdivided cell, we may end up storing a lot of empty nodes. We can cut down on the size of such sparse trees by only storing subtrees whose leaves have interesting data (i.e. "important subtrees"). We can actually cut down on the size even further. When we only keep important subtrees, the pruning process may leave long paths in the tree where the intermediate nodes have degree two (a link to one parent and one child). It turns out that we only need to store the node u at the beginning of this path (and associate some meta-data with it to represent the removed nodes) and attach the subtree rooted at its end to u. It is still possible for these compressed trees to have a linear height when given "bad" input points.

Although we trim a lot of the tree when we perform this compression, it is still possible to achieve logarithmic-time search, insert, and delete. Recall the previously-mentioned z-order. It maps in $O(1)$ time each cell of the full quadtree (and hence even the compressed quadtree) to a one-dimensional line (and maps it back in $O(1)$ time too), creating a total order on the elements. Therefore, we can store the quadtree in a data structure for ordered sets (in which we store the nodes of the tree). We must state a reasonable assumption [5] before we continue: we assume that given two real numbers $\alpha, \beta \in [0, 1)$ expressed as binary, we can compute in $O(1)$ time the index of the first bit in which they differ. We also assume that we can compute in $O(1)$ time the lowest common ancestor of two points/cells in the quadtree and establish their relative z-ordering, and we can compute the floor function in $O(1)$ time. With these assumptions, point location of a given point q (i.e. determining the cell that would contain q), insertion, and deletion operations can all be performed in $O(\log n)$ time (i.e. the time it takes to do a search in the underlying ordered set data structure).

To perform a point location for q (i.e. find its cell in the compressed tree):

1. Find the existing cell in the compressed tree that comes before q in the z-order. Call this cell v.

2. If $q \in v$, return v.

3. Else, find what would have been the lowest common ancestor of the point q and the cell v in an uncompressed quadtree. Call this ancestor cell u.

4. Find the existing cell in the compressed tree that comes before u in the z-order and return it.

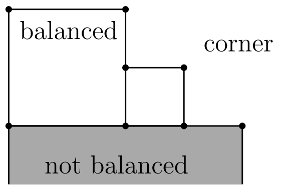

Figure 4: A balanced leaf has at most one corner in a side

Without going into detail, to perform insertions and deletions we first do a point location for the thing we want to insert/delete, and then insert/delete it. Care must be taken to reshape the tree as appropriate, creating and removing nodes as needed.

5 Mesh Generation

Mesh generation is essentially the triangulation of a point set for which further processing may be performed. As such, it is desirable for the resulting triangulation to have certain properties (like non-uniformity, triangles that are not "too skinny", large triangles in sparse areas and small triangles in dense ones, etc.) to make further processing quicker and less error-prone [3, 5]. Quadtrees built on the point set can be used to create meshes with these desired properties.

Consider a leaf of the quadtree and its corresponding cell v. We say v is *balanced* if the cell's sides are intersected by the corner points of neighbouring cells at most once on each side (see Figure 4). This means that the quadtree levels of leaves adjacent to v differ by at most one from the level of v. When this is true for all leaves, we say the quadtree is balanced.

Consider the cell v and the 5×5 neighbourhood of same-sized cells centred at v. We call this neighbourhood the *extended cluster*. We say the quadtree is *well-balanced* if it is balanced, and for every leaf u that contains a point of the point set, its extended cluster is also in the quadtree and the extended cluster contains no other point of the point set.

Creating the mesh is done as follows:

1. Build a quadtree on the input points.

2. Ensure the quadtree is balanced. For every leaf, if there is a neighbour that is too large, subdivide the neighbour. This is repeated until the tree is balanced. We also make sure that for a leaf with a point in it, the nodes for each leaf's extended cluster are in the tree.

3. For every leaf node v that contains a point, if the extended cluster contains another point, we further subdivide the tree and rebalance as necessary. If we needed to subdivide, for each child u of v we ensure the nodes of u's extended cluster are in the tree (and re-balance as required).

4. Repeat the previous step until the tree is well-balanced.

5. Transform the quadtree into a triangulation.

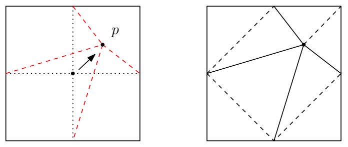

Figure 5: Warp the closest corner of p's cell and triangulate (image from [5])

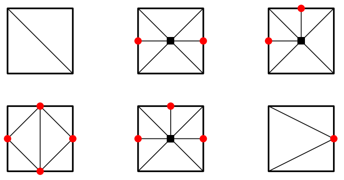

Figure 6: Other squares triangulated depending on their intersected sides (image from [5])

We consider the corner points of the tree cells as vertices in our triangulation. Before the transformation step we have a bunch of boxes with points in some of them. The transformation step is done in the following manner: for each point, warp the closest corner of its cell to meet it and triangulate the resulting four quadrangles to make "nice" triangles[5] (see Figure 5).

Figure 6 illustrates the manner in which the remaining squares are triangulated. For each regular square (no points within and no corner points in its sides), introduce the diagonal. Note that due to the way in which we separated points with the well-balancing property, no square with a corner intersecting a side is one that was warped. As such, we can triangulate them as follows. If there is one intersected side, the square becomes three triangles by adding the long diagonals connecting the intersection with opposite corners. If there are four intersected sides, we split the square in half by adding an edge between two of the four intersections, and then connect these two endpoints to the remaining two intersection points. For the other squares, we introduce a point in the middle and connect it to all four corners of the square and each intersection point.

At the end of it all, we have a nice triangulated mesh of our point set built from a quadtree.

[5]The interested reader is referred to chapter 12 of [5] for more details on "nice" triangles.

References

[1] S. Aluru. Quadtrees and octrees. *Handbook of Data Structures and Applications*, chapter 19 *Chapman & Hall/CRC*, 2005.

[2] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] M. de Berg, O. Cheong, M. van Kreveld, M.H. Overmars. Computational Geometry Algorithms and Applications. 3^{rd} edition, *Springer-Verlag*, 2008.

[4] R.A. Finkel and J.L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[5] S. Har-Peled. Geometric approximation algorithms. *Mathematical Surveys and Monographs Vol. 173, American mathematical society*, 2011.

[6] G.M. Hunter. Efficient Computation and Data Structures for Graphics. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, 1978.

[7] G.M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):145–153, 1979.

[8] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *International Business Machines Company*, 1966.

[9] R.C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *ACM SIGGRAPH Computer Graphics*, 20(4):197–206, 1986.

[10] J.A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, 1982.

[11] H. Samet. Connected component labeling using quadtrees. *Journal of the ACM (JACM)*, 28(3):487–501, 1981.

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

[13] H. Samet. An overview of quadtrees, octrees, and related hierarchical data structures. *Theoretical Foundations of Computer Graphics and CAD*, (pp. 51-68). *Springer-Verlag Berlin Heidelberg*, 1988.

[14] H. Samet. Hierarchical spatial data structures. *Symposium on Large Spatial Databases*, (pp. 191-212) *Springer-Verlag Berlin Heidelberg*, 1989.

[15] H. Samet and R.E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics (TOG)*, 4(3):182–222, 1985.

[16] M. Shneier. Calculations of geometric properties using quadtrees. *Computer Graphics and Image Processing*, 16(3):296–302, 1981.

[17] M. Shneier. Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing*, 17(3):211–224, 1981.

[18] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[19] J.E. Warnock. A hidden surface algorithm for computer generated halftone pictures. Computer Science Department TR 4-15, University of Utah, 1969.