QuadTrees: Implementation, Analysis and Application

November 7, 2022

ANSHIKA (2021CSB1069) ,
AJAYBEER SINGH (2021CSB1063) ,
SHAIK DARAKSHINDA (2021CSB1130)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Ms.Monisha Singh

Summary: In this universe of ever-growing data, we are always looking for Data Structures that are compact and depending on the nature of the data they save space as well as time while facilitating operations such as Search. Quad-tree is a data structure that allows partitioning of space that is simple and efficient to navigate and search.
Our project brings about how Quadtrees can be implemented and how they significantly reduce the space and time requirements in practical situations. This data structure is represented as a tree structure, which has at most four children and multiple levels.They are based on the principle of recursive decomposition.
One of the many applications on which our project is focused is Collision Detection. This has a significant use in the development of Autonomous Vehicles.

1. Introduction

Quadtree is a heirarchal data structure based on the principle of recursive decomposition (similar to divide and conquer method). Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and improved execution times and is thus particularly useful for performing set operations.
However, hierarchical data structures are also attractive because of their conceptual clarity and ease of implementation.
This project focuses primarily on the Implementation of Quadtrees, and its application in Colllision Detection and related areas.
There are tremendous practical applications of collision detection of Quadtrees, some of them being the following queries:
1. We can determine all cities within 500 km of New Delhi that have a population in excess of 8000 people. Instead of checking each city individually, using a representation that decomposes India into quadrants having sides of length 100 km would mean that at most four squares need to be examined. Thus Mumbai and its adjacent states can be safely ignored.

2. An adaptive-grid risk-field model based on Quadtree grid-dividing is proposed for Path-Planning Strategy for Lane Changing Based on Adaptive-Grid Risk-Fields of Autonomous Vehicles. The method of Quadtree dividing is combined with risk-field theory. The grid division is performed for medium-risk grids, while high-risk and low-risk grids maintain a larger grid area, which improves the applicability and accuracy of the risk-field model in autonomous vehicle systems and provides more accurate traffic environment risk information for autonomous vehicle decision making and planning.

3. Another application on which we will focus in this project is Collision Detection using Quad-Tree.The data of all spatial information, such as the navigation area, the position of milestones and vehicles, forbidden areas can received. This spatial information about the mission is essential for creating path-plan for autonomous vehicles.
Decomposition of received spatial information is the first step that is necessary for the path-finding algorithm.

Having a geographical coordinate system is not ideal, as the navigation area of the mission may have a considerable number of geographical coordinates that are visited but not be in our scope of interest. Thus, we are consuming more computing resources and make path-finding improbable. We are using a Quad-tree data structure that divides our navigation area into smaller regions, quadrants.This is basically collision detection, but in advance and changing the path accordingly.[2]

2. Theorems and Lemmas

Lemma 1:Given a quadtree T of size n and of height h, one can preprocess it (using hashing), in linear time, such that one can perform a point-location query in T in O(log h) time. In particular, if the Quadtree has height O(log n) (i.e., it is "balanced"), then one can perform a point-location query in T in O(log log n) time [1].

Lemma 2 :Let P be a set of n points contained in the unit square, such that diam(P) = max||p-q||>= 1/2. Let T be a Quadtree of P constructed over the unit square, where no leaf contains more than one point of P. Then, the depth of T is bounded by O(log d), it can be constructed in O(n log d) time, and the total size of T is O (n log d) , where d = d(P).[1]
Lemma 3 :Two compressed Quadtrees with a common root cell can be merged in time proportional to the sum of their sizes.
Proof:
The merging algorithm presented above performs a preorder traversal of each compressed Quadtree.
The whole tree may not need to be traversed because in merging a node, it may be determined that the whole subtree under the node directly becomes a subtree of the resulting tree.
In every step of the merging algorithm, we advance on one of the trees after performing at most O(d) work.
Thus, the run time is proportional to the sum of the sizes of the trees to be merged.
To construct a compressed Quadtree for n points, scan the points and find the smallest and largest coordinate along each dimension. Find a region that contains all the points and use this as the root cell of every compressed Quadtree constructed in the process.
Recursively construct compressed Quadtrees for floor(n/2) points and the remaining ceil(n/2) points and merge them in O(dn) time.
The compressed Quadtree for a single point is a single node v with the root cell as L(v). The run time satisfies the recurrence.
T (n) = T(floor(n/2))+T(ceil(n/2))+O(dn) resulting in O(dn log n) run time.

Theorem 1: A Quadtree of depth 'd' storing a set of 'n' points has O((d + 1)n) nodes and can be constructed in O((d + 1)n) time.

Theorem 2: Let T be a Quadtree of depth 'd'. The neighbor of a given node 'v' in T in a given direction can be found in O(d + 1) time.

In layman's terms:
For collision detection, as an example, if I were to have 100 objects I need to check, I would normally have to compare each object to one another. This would be O(nxn) meaning that there would be 10000 check.
A Quadtree enables us to check our direct neighbors only. If it had a maximum depth of 4 created, worst case it would be O((d + 1)n) time, or O(5*100). So 500 comparisons. That's much better than 10000.
So a guaranteed 10,000 checks or a worst case 500 checks?[5]

3. Figures, Tables and Algorithms

3.1. Figures

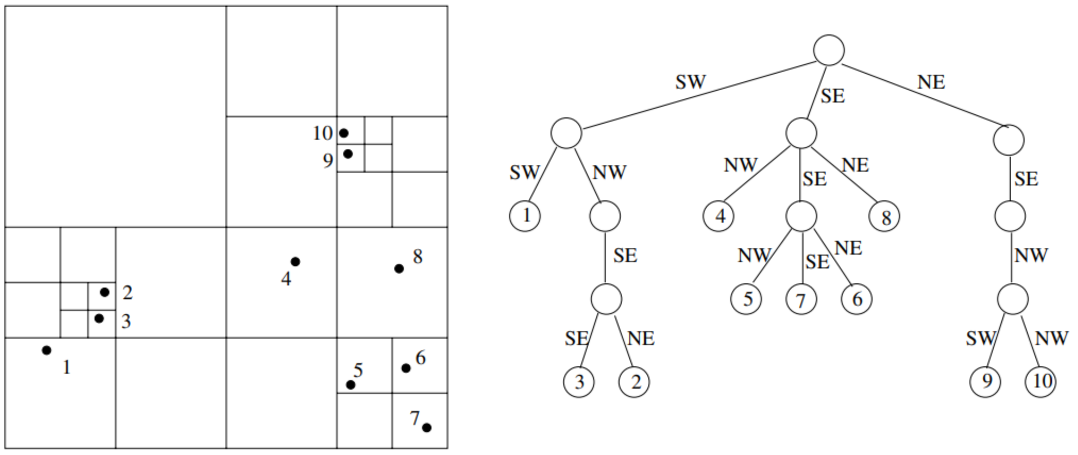
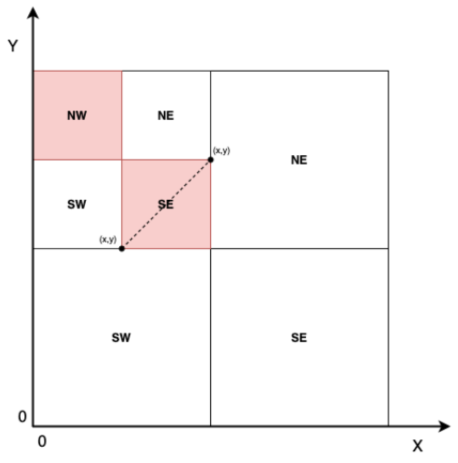
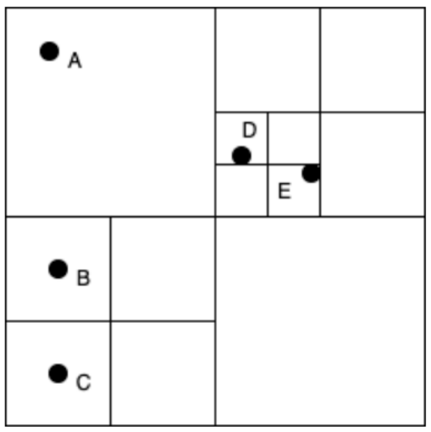


Figure 1: Structure of the Quadtree.

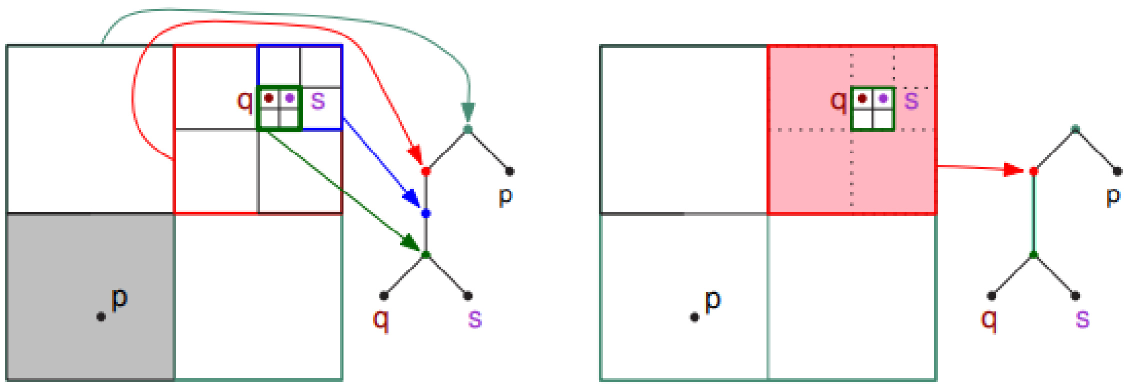


(a) Visual Representation of Quadtree on a grid.



(b) 2D space with data points.

Figure 2: Quadtree graph with Obstacles.



(a) 2D space with data points.

Figure 3: A point set, its quadtree and its compressewd Quadtree.

3.2. Tables

Space complexity: O(klog N)
Where k is count of points in the space and space is of dimension N x M, N >= M.

Time Complexity Table

	Expected Complexity	**Worst Complexity**
Find	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Table 1: Time Complexity (log base is 2)

3.3. Algorithms

The following algorithms are used in the implementation of Quadtrees in this project.

Algorithm 1 Pseudo code for splitting surface

```
Input: QuadNode parent; Identifies parent node
Input: Point topRight, bottomLeft; Node position in the surface
Input: Status status; Node stauts (Obstructed, Free or Split
Input: int resolution; maximum size of quadrant splitting
Input: Array[Obstacle] obstacles; All surface obstacles
1) for obstacle : obstacles do
2)     if obstacle.intresect(topRight, bottomLeft) then
3)         if obstacle.block(topRight bottomLeft) or QuadNode.width <=
               resolution or QuadNode.hight <= resolution then
4)             status = OBSTRUCTED;
5)         else
6)             subdevide(obstacles, resolution);
7)         end
8)     end
9)end
```

Algorithm 2 Pseudo code for finding a certain point in quad-tree

```
Input: Point locationToFind; Node position in the surface
1) while node is not leafNode do
2)     if locationToFind.x < MeanX then
3)         if locationToFind.y < MeanY then
4)             node = getSouthwest();
5)         else
6)             node = getNorthwest();
7)         end
8)     else
9)         if locationToFind.y < MeanYthen
10)            node = getSoutheast();
11)        else
12)            node = getNortheast();
13)        end
14)    end
15)end
[3]
```

4. Some further useful details

Space Requirements:

- Reducing the amount of space necessary to store data is one of the prime motivations to use Quadtrees. This is done by the aggregation of homogeneous blocks. The result of this aggregation is ultimately the reduction of execution time of various operations
- Quadtree is not, although, an ideal representation.The worst case for a Quadtree of a given depth in terms of storage requirements occurs when the region corresponds to a checkerboard pattern, as in Figure A1. The amount of space required is a function of the number of levels in the Quadtree.

How does the program work?

1. Our focus is on dynamic files (i.e., the amount of data can grow and shrink at will) and on applications involving search.
2. It is an adaptation of the region Quadtree to point data which associates data points (that need not be discrete) with quadrant.
3. The shape of the PR Quadtree is independent of the order in which data points are inserted into it
4. The program detects collision in Autonomous Vehicles. Firstly, the initial position of the center of the car is specified. Then the range where the car is to move is specified.
   Then the dimensions of the car (including safety margin is taken as input) and the obstacles (their coordinates) are taken as input.
   All of this is done by Artificial Intelligence in practical situations. After that, the task of Quadtree comes into play. It is to detect an obstacle and tell if there is going to be a collision or not.

5. Conclusions

Quadtree Data Structure effectively reduces time complexity by using recurvise decomposition while working on high frequency data. Quadtree can be efficiently used in Autonomous Vehicles to detect if the vehicle is safe or will collide. This project specifies the collision detection using Quadtree which is in two dimensions. The obstacles (which have to be specified by AI) that we have considered are static in nature according to our program. A future scope is the dynamic obstacle implementation of Quadtree for collision detection.

6. Bibliography and Citations

Acknowledgements

We take this opportunity to express profound gratitude and deep regards to our Course instructor Dr Anil Shukla for his exemplary guidance and encouragement throughout the course.
We also want to thank our mentor Ms. Monisha Singh for her constant support and help for the project.

References

[1] Har-Peled S. Geometric approximation algorithms. American Mathematical Soc.; 2011.

[2] Samet H. The quadtree and related hierarchical data structures. ACM Computing Surveys (CSUR). 1984 Jun 1;16(2):187-260.

[3] Rokicki TG. An algorithm for compressing space and time-Amazing performance improvements can result. DR DOBBS JOURNAL. 2006 Apr 1;31(4):12-+.

[4] Kalezic, Dorde. "Safety-Guaranteed Mission Planner for Autonomous Vehicles." (2020).

[5] https://www.jordansavant.com/book/algorithms/quadtree.md: :text=Complexity,

A. Appendix A

Theorem A: Given a set P of n points in the plane, one can compute a compressed Quadtree of P in O(n log n) deterministic time.
Theorem B: Given a Quadtree T of size n, with its leaves stored in an ordered-set data-structure D according to the Q-order, then one can perform a point-location query in O(Q(n)) time, where Q(n) is the time to perform a search query in D.
Theorem C: Assuming one can compute the Q-order in O(d) time for two points IRd, then one can maintain a compressed Quadtree of a set of points in IRd, in O(d log n) time per operation. The operations of insertion, deletion, and point-location query are supported. Furthermore, this can be implemented using any data-structure for ordered-set that supports insertion/deletion/predecessor operations in logarithmic time. In particular, one can construct a compressed Quadtree of a set of n points in IRd in O(dn log n) time.[4]