ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΜΩΝ

Αποδοτικοί Αλγόριθμοι Για Τον Υπολογισμό Του Μέγιστου Άδειου Κύβου Σε Χώρους Με Εμπόδια

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνα Μέλλου

Επιβλέπων: Δημήτρης Φωτάκης
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΜΩΝ

Αποδοτικοί Αλγόριθμοι Για Τον Υπολογισμό Του Μέγιστου Άδειου Κύβου Σε Χώρους Με Εμπόδια

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνα Μέλλου

Επιβλέπων: Δημήτρης Φωτάκης
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $21^{η}$ Ιουλίου 2014.

.....	.....	.....
Δημήτρης Φωτάκης	Στάθης Ζάχος	Άρης Παγουρτζής
Επ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2014

. . . . . .

Κωνσταντίνα Μέλλου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

Computational geometry is a branch of computer science, in particular algorithms design, that studies problems of geometric nature. This area has gained enormous attention over the years due to its various applications in computer graphics, robotics, geographic information systems and other important fields.

The application that motivated the study of this thesis lies in the field of electronic design automation. In order to achieve an accurate simulation of an integrated circuit, some parasitic components of the circuit need to be modeled. An important step of this procedure involves the calculation of the largest empty cube that can be placed among the conductors considering a given center. This calculation must be repeated numerous times and, therefore, an efficient algorithm is required.

This problem is closely related to fundamental problems of computational geometry, such as the search of the largest empty rectangle among a set of points or the nearest neighbor problem, both of which apply in many areas, like pattern recognition, data mining and material manufacturing. In this thesis, we describe these and other related problems and outline some of their proposed solutions.

After a general overview is provided to the reader, the main subject of this thesis is introduced. The first part focuses on Manhattan geometries, where all obstacles as well as the requested cube are axis aligned. The data structure that is used for their representation is called octree. The algorithm focuses on inserting the obstacles in the proper tree nodes, such that each search query examines only the obstacles that are located in the neighborhood of the query point. When the nearest obstacle to the query point is identified, the largest empty cube that abuts this obstacle is calculated.

Afterwards, this algorithm is generalized for the case of non-Manhattan geometries. Now, the obstacles can be polygonal or rotated rectangular cuboids and the requested cube is no more axis aligned. It can be rotated by an angle ϕ around the z axis, where ϕ is the direction of one of the obstacles' edges.

Finally, some accelerating techniques are proposed and an experimental study is conducted in order to evaluate the performance of the algorithm. More specifically, the algorithm is implemented in C++ and various input sets are used to assess the time performance and the memory usage of the program during the creation of the octree and the execution of the queries.

Keywords

computational geometry, cube, largest empty cube, 3D, Manhattan geometries, non-Manhattan geometries, obstacles, data structures, octree

Περίληψη

Η υπολογιστική γεωμετρία είναι ένας κλάδος της επιστήμης υπολογιστών, συγκεκριμένα του σχεδιασμού αλγορίθμων, που μελετάει προβλήματα γεωμετρικής φύσης. Αυτή η περιοχή έχει προσελκύσει το ενδιαφέρον στο πέρασμα των χρόνων λόγω των ποικίλων εφαρμογών της στη γραφική υπολογιστών, τη ρομποτική και άλλους τομείς.

Η εφαρμογή που ενέπνευσε τη μελέτη αυτής της εργασίας αφορά την περιοχή της σχεδίασης ηλεκτρονικών κυκλωμάτων. Προκειμένου να επιτευχθεί μια ακριβής προσομοίωση ενός ολοκληρωμένου κυκλώματος, κάποιες παρασιτικές συνιστώσες του πρέπει να μοντελοποιηθούν. Ένα σημαντικό βήμα της διαδικασίας αυτής περιλαμάνει τον υπολογισμό του μέγιστου άδειου κύβου με δεδομένο κέντρο που μπορεί να τοποθετηθεί ανάμεσα στους αγωγούς. Αυτός ο υπολογισμός πρέπει να επαναληφθεί πολλές φορές και, επομένως, ένας αποδοτικός αλγόριθμος είναι απαραίτητος.

Αυτό το πρόβλημα είναι στενά συνδεδεμένο με θεμελιώδη προβλήματα της υπολογιστικής γεωμετρίας, όπως η αναζήτηση του μέγιστου άδειου ορθογωνίου ανάμεσα σε σημεία και το πρόβλημα του πλησιέστερου γείτονα, τα οποία βρίσκουν εφαρμογή σε πολλούς τομείς, όπως η αναγνώριση προτύπων, η εξόρυξη δεδομένων και η κατασκευή υλικών. Σε αυτήν την εργασία, περιγράφουμε αυτά και άλλα σχετικά προβλήματα και σκιαγραφούμε κάποιες από τις προτεινόμενες λύσεις τους.

Στη συνέχεια, παρουσιάζεται το κυρίως μέρος της εργασίας. Αρχικά, επικεντρωνόμαστε στις γεωμετρίες Manhattan, όπου ο ζητούμενος κύβος και όλα τα εμπόδια είναι ευθυγραμμισμένα με τους άξονες. Η δομή δεδομένων που χρησιμοποίειται ονομάζεται octree. Ο αλγόριθμος στηρίζεται στην εισαγωγή των εμποδίων στους κατάλληλους κόμβους του δέντρου, έτσι ώστε κάθε αναζήτηση να εξετάζει μόνο τα γειτονικά εμπόδια του δοσμένου κέντρου. Όταν το πλησιέστερο εμπόδιο εντοπιστεί, υπολογίζεται ο μέγιστος άδειος κύβος που εφάπτεται σε αυτό. Έπειτα, ο αλγοριθμος γενικεύεται για non-Manhattan γεωμετρίες. Τα εμπόδια μπορεί να είναι πολυγωνικά ή περιστραμμένα ορθογώνια παραλληλεπίπεδα γύρω από τον άξονα z κατά γωνία ϕ, όπου ϕ ο προσανατολισμός της ακμής κάποιου εμποδίου.

Τέλος, προτείνονται κάποιες τεχνικές βελτιστοποίησης και ακολουθεί μια πειραματική μελέτη για την αξιολόγηση της επίδοσης του αλγορίθμου. Συγκεκριμένα, υλοποιούμε τον αλγόριθμο σε C++ και χρησιμοποιούμε διάφορα σετ εισόδου για να αξιολογήσουμε τη χρονική επίδοση και τη χρήση της μνήμης κατά την κατασκευή του δέντρου και την πραγματοποίηση των αναζητήσεων.

Λέξεις Κλειδιά

υπολογιστική γεωμετρία, κύβος, μέγιστος άδειος κύβος, 3D, Manhattan γεωμετρίες, non-Manhattan γεωμετρίες, εμπόδια, δομές δεδομένων, octree

Ευχαριστίες

Με την ολοκλήρωση αυτής της διπλωματικής, θα ήθελα να ευχαριστήσω τους καθηγητές της τριμελούς επιτροπής: κ. Στάθη Ζάχο, κ. Άρη Παγουρτζή και κ. Δημήτρη Φωτάκη. Ιδιαίτερα, θα ήθελα να ευχαριστήσω τον κ. Φωτάκη, ο οποίος ήταν και ο επιβλέπων αυτής της εργασίας, τόσο για τη στήριξη και τη βοήθεια του, αλλά και γιατί χάρη σε αυτόν γνώρισα και αγάπησα τους αλγορίθμους.

Ακόμα, ένα μεγάλο ευχαριστώ οφείλω στην εταιρεία Helic, Inc. και στον κ. Γιώργο Κουτσογιαννόπουλο που μου έδωσε την ευκαιρία να δουλέψω πάνω στο θέμα αυτό, αλλά και να αποκτήσω μια πρώτη επαφή με το εργασιακό περιβάλλον μιας πρωτοπόρας εταιρείας. Επίσης, θα ήθελα να ευχαριστήσω τους Παντελή Παπαδόπουλο και Μάριο Βισβάρδη για τη βοήθεια που μου παρείχαν απλόχερα οποιαδήποτε στιγμή τη χρειάστηκα.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαίτερα την οικογένεια μου για τη διαρκή στήριξη τους και τους φίλους μου για όσα περάσαμε μαζί κατά τη διάρκεια της φοιτητικής μας ζωής και όχι μόνο.

Contents

Chapter 1

Introduction

1.1 Problem Statement

Computational geometry is a branch of computer science, which emerged from the field of algorithms design and analysis. It is devoted to the study of problems of geometric nature and its goal is to offer efficient algorithmic solutions. Its applications extend across numerous domains, such as computer graphics, robotics, geographic information systems and others.

One of the fundamental problems in computational geometry concerns the computation of the largest area rectangle that can fit among obstacles in the plane. This problem applies in the fields of electronic design automation, geographic information systems, etc. and has received a lot of attention over the years. In the case where the obstacles are two-dimensional points and the requested rectangle axis aligned, Chazelle et al. [9] and Aggarwal et al. [6] have proposed interesting solutions with $O(nlog^3n)$ and $O(nlog^2n)$ time performance respectively.

There are many variants of this problem that may concern the dimensions of the space and the nature of the obstacles, which can be points, segments, polygons, cuboids and others. Moreover, another variation concerns the orientation of the requested rectangle, which can be axis aligned or not. Other problems include the search of the largest empty square among obstacles or the largest empty cube in the case of the three-dimensional space.

This thesis focuses on the study of one such variant: Imagine we have a number of obstacles in the three-dimensional space. Given a query point, the goal is to find the largest empty cube, whose center is the query point and which is empty, i.e. does not contain any part of the obstacles in its interior. This thesis covers

both the cases where the obstacles are axis aligned cuboids (chapter 3), as well as polygonal or rotated cuboids (chapter 4).

It is obvious that there must be at least one point of contact between the largest empty cube and an obstacle. Otherwise, the cube could be enlarged until it touched one and, thus, it would not be the largest possible empty cube. This observation demonstrates the equivalence of this problem to finding the nearest obstacle to a query point. This problem, also known as the nearest neighbor problem, is also very important in the field of computational geometry and has many applications in the area of pattern recognition, computer vision and databases.

A simple solution to this problem could be to compute the distances between the query point and all obstacles and identify the nearest one. This algorithm is linear to the number of cuboids, but it is not efficient when we deal with geometries with millions of cuboids, where the process of finding the largest empty cube must be realized for millions of different centers. Therefore, we need a more efficient algorithm. One of the main characteristics of this algorithm must be the restriction of the search for the nearest obstacle only in the neighborhood of the query point. This technique can significantly reduce the query time and achieve a much better performance. In order to achieve that, we need a space management technique for a convenient storage of the obstacles. The data structure that we selected is the octree, since it offers a simple yet efficient division of the space.

1.2 Motivation

The problem that motivated the study of this thesis lies in the field of electronic design automation. Multilayered integrated circuits present an increased density and the electromagnetic coupling effect among conductors influences a lot the performance of the circuit. Furthermore, in very large scale integrated (VLSI) circuits, there is important coupling among interconnects. Ideally, wire only connects the circuit elements without affecting their performance. However, a real wire has resistance, capacitance and inductance, which introduce parasitic effects. In order to achieve an accurate circuit simulation, these parasitic components must be modeled. Among these three parameters, capacitance has received the most attention, because of its major influence on time delay, power consumption and others [1]. The need for fast computational methods that simultaneously achieve great accuracy has led to advanced research in the field of VLSI circuits.

One method that can be used for modeling the parasitic capacitances, a procedure also known as capacitance extraction, is called floating random walk (FRW) algo-

rithm. This algorithm presents many advantages, such as good scalability, tunable accuracy and low memory usage [2]. In order to compute the capacitances related to each conductor, a Gaussian surface is constructed to enclose it and, based on the Gauss theorem and other methods, the charge of the conductor is computed. This computation involves sampling the gaussian surface and using each sample point as starting point for a walk. Each walk may include many hops. More specifically, the sample point serves as the center for the construction of a conductor free cube. That cube's surface is then sampled and another sample point is selected and used for the construction of another empty cube, etc. The charge of the conductor is computed as the statistical mean of the sampled values on the Gaussian surface, which are produced by the mean of the sampled values of the first cube's surface etc. The hops stop when the under examination sample point belongs to a conductor's surface. If sufficient walks start from a conductor's Gaussian surface then the charge of the conductor is accurately computed and the coupling capacitance between conductors is given from the statistical mean of some weight values that are calculated during the walks [2].

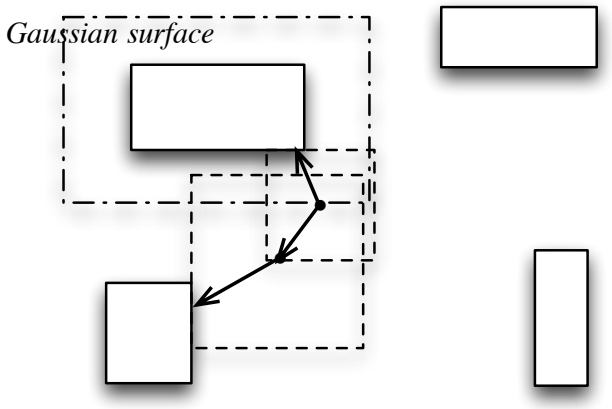

Figure 1.1: Example of two random walks [2]

It is obvious that an important step of this algorithm includes the computation of a cube that has the sample point as its center and which does not contain any conductors. In order to achieve satisfying efficiency, this cube should be as large as possible. Therefore, if we imagine the conductors as obstacles and the sample point as a query point, the problem is translated into finding the largest empty cube with a specific center, which is exactly the topic of this thesis.

1.3 Overview

The structure of this thesis is the following:

Chapter 2. Spatial Data Structures and Related Problems

Chapter 2 consists of two main sections. The first section presents the basic data structures for spatial representation. More specifically, we offer a description of quadtree, octree, R-tree and Voronoi diagram, which are all widely used in numerous applications in the field of databases, geographic information systems, etc. The second section reviews a variety of problems that are related to the topic of this thesis. These problems are fundamental in the area of computational geometry.

Chapter 3. Finding the Largest Empty Cube in Manhattan Geometries

This chapter introduces the main topic of this thesis. First of all, a description of Manhattan geometry is given and the problem statement is presented in details. The data structure that we selected to use for the spatial representation is called octree. The first step of the algorithm is the creation of the octree by inserting the obstacles in the tree and placing them in the proper leaf nodes. The tree changes dynamically during the insertion and leaf nodes may obtain children and become internal, if they correspond to a lot of obstacles. Afterward, the tree is traversed in order to identify the nearest obstacle to a query point and then compute the largest empty cube that can fit among the obstacles. After the reader is acquainted with the basic steps of the algorithm, some techniques are proposed in order to accelerate its performance. Finally, a simple example illustrates the steps of the algorithm in practice.

Chapter 4. Finding the Largest Empty Cube in Non-Manhattan Geometries

Chapter 4 covers the cases of non-Manhattan geometries. We first present some useful background, such as how to distinguish the visible and non-visible edges of a polygon or how to rotate points around another point or an axis. Then, we present the reviewed steps of the algorithm of chapter 3, which can now apply to non-Manhattan geometries as well. The requested cube may not be axis aligned,

but rotated by an angle ϕ around the z axis. The orientation ϕ of the cube is one of the directions of the obstacles. This restriction may not always lead to the largest cube of any direction. In the last section of this chapter, we present a short analysis to demonstrate the maximum possible difference between the cube produced by the algorithm and the actually largest cube of any direction.

Chapter 5. Practical Implementation and Experimental Results

After the theoretical analysis of the algorithm in the preceding chapters, we implement it in $C++$ and perform certain experiments to evaluate its performance. First of all, we describe the environment and the input sets we used for our experiments. Afterwards, we conduct the experiments and compare the execution time for various input sets and number of queries. Moreover, we experiment with data sets that contain obstacles of multiple directions and, finally, evaluate the importance of an algorithm parameter (the number of the obstacles that can correspond to each leaf node) in its performance.

Chapter 2

Spatial Data Structures and Related Problems

2.1 Spatial Data Structures

Spatial data include roads, cities, parts of CAD systems, images etc. It is obvious that there are many applications that require storage, retrieval and processing of this type of data. VLSI design, geographic information systems (GIS), urban planning, environmental monitoring, resource management are only some of the numerous examples [16]. The need for an efficient representation and storage of spatial data quickly led to the development of data structures that use the properties of the data, in order to achieve efficient memory management and good query performance.

Some of the main spatial data structures are presented in the following section. In particular, we describe the quadtree, the octree, the R-tree and the Voronoi diagram.

2.1.1 Quadtree and Octree

Quadrees are one of the first data structures that were used for higher-dimensional data. They were introduced by R. Finkel and J. L. Bentley in 1974 [3]. A quadtree is a rooted tree, where each node has either exactly four children (internal node) or no children at all (leaf node). The root corresponds to a square or a rectangle and each one of its children corresponds to one of its quadrants. Therefore, the

squares or rectangles of the leaves form a subdivision of the square or rectangle of the root.

The decomposition of space takes place recursively and the parent node may be divided in subspaces of equal or different sizes, according to the input data. The subdivision occurs when the capacity of each cell surpasses a predefined maximum capacity. For example, if we have a point set as an input, we may define a limit of one point per cell and, as a result, if there are two or more points in a cell, it is divided in four subcells. We can also define a number for the times the decomposition process is applied. In that case, the decomposition stops after a number of steps, otherwise its resolution depends on the properties of the input data [16].

In the following figure, we can see the correspondence between the spatial partitioning and the nodes of a quadtree.

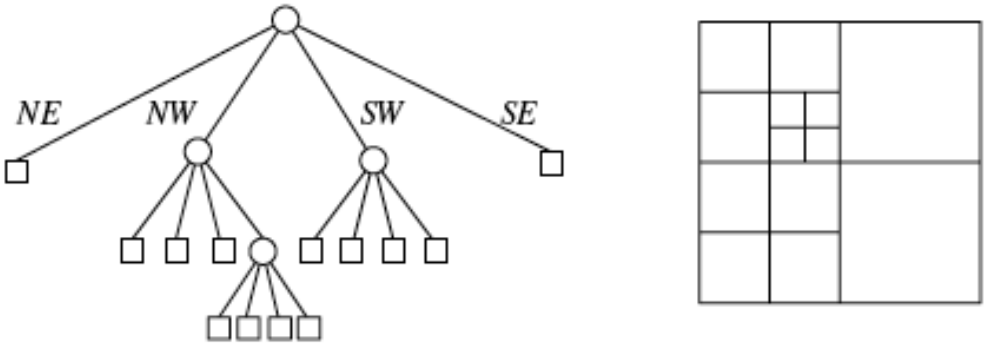

Figure 2.1: Example of quadtree [3]

Quadtrees can store a wide variety of data, such as points, polygons, edges and regions. Thus, they are useful in various fields of applications. First of all, quadtrees can be used in image processing and image analysis. The root of the tree represents the whole image and the subdivision of each parent node continues until each leaf contains blocks of pixels that have the same value, either all have value 0 or value 1. Another application includes geographic information systems, where quadtrees store the information about the geographic characteristics of an area of interest. Quadtrees can also be used for answering range queries, as they perform well in practice, although they are not the best solution from a theoretical point of view. Additional examples include among others nearest neighbor search and mesh generation for the simulation of the function of circuits designs [3].

It is obvious that this data structure offers a simple spatial representation that is appreciated in multiple applications. Its main disadvantage is that the result-

ing tree is not always balanced. There may be cases of point sets, for example, where the points are unevenly distributed and the recursive subdivision produces a quadtree that is quite unbalanced. However, its simplicity in terms of insertion and search queries and the fact that, in many cases, extremely unequally distributed input data are rare, renders it a satisfactory solution for many problems.

The three dimensional analog of quadtree is called octree. In this case, each internal node has exactly eight children and the rest of the nodes do not have any children (leaves). The three-dimensional object that corresponds to the root of the tree is recursively subdivided into octants, suboctants etc, until a condition for the termination of the subdivision is met.

Octrees are mainly used for the representation of 3D space and 3D objects. They can be applied to computer graphics and color quantization. More specifically, the color quantization algorithm of Gervautz and Purgathofer (1988) is based on a predetermined subdivision of the RGB (red, green, blue) space into levels of octants [18] Other examples include spatial indexing and nearest neighbor searching in three dimensions.

2.1.2 R-Tree

R-trees were introduced by Guttman in 1984 in order to handle geometrical data, such as points, surfaces, line segments, volumes, etc. They are an hierarchical data structure based on $B+$ trees. Their main characteristic is that they group nearby objects and use bounding rectangles for their representation. More specifically, each internal node corresponds to the minimum bounding rectangle (MBR) that bounds its children and the leaves have pointers directly to the object of the database. R-trees are a dynamic data structure, which implies that global reorganization is not required for insertions or deletions [4].

One of the main advantages of R-trees is that they are balanced, i.e. all their leaves are at the same height. In order to achieve the best performance, R-trees have a predefined maximum limit M and a minimum limit m on the number of entries of each node.

Searching in R-trees is simple. We begin from the root node and each time we examine only the children of the current node, whose MBR overlaps with the given query object (rectangle, box, etc). In that way, some subtrees are never examined. In order to achieve a good searching performance, it is important to ensure that there is not much overlap between MBRs and that they cover the least possible empty space. If the opposite happens, then, during the searching, only a few subtrees are pruned and the rest of them need to be examined. Therefore, the

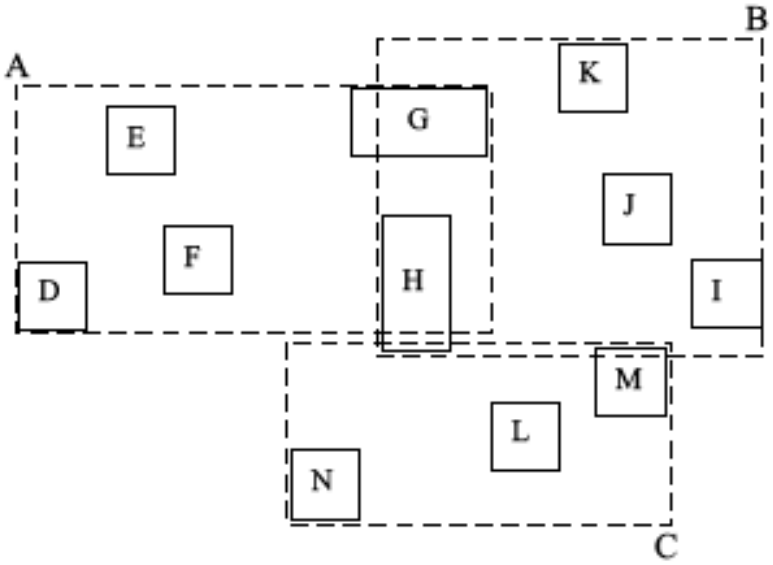

Figure 2.2: Example of data MBRs and their MBRs [4]

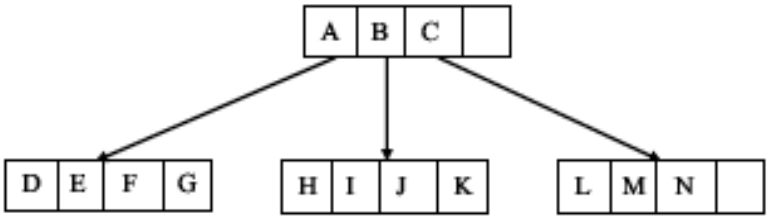

Figure 2.3: The corresponding R-tree [4]

performance is not satisfying and that is one of the main disadvantages of this data structure.

Insertions in R-trees start by examining the root node and then traversing the tree, searching for the appropriate leaf that can host the new entry. When the leaf is found, the new entry is inserted and all internal nodes that belong to the path from the leaf to the root are updated accordingly. In case that the leaf is already full, it is split into two new nodes and its entries are distributed in these nodes. The split procedure must be realized carefully in order to avoid "bad splits" (see figure below). Guttman proposed several algorithms for the insertion in R-trees. For example, linear split algorithm selects two rectangles of the node that are far away from each other and then assigns the rest of them to the node that requires

the least enlargement.

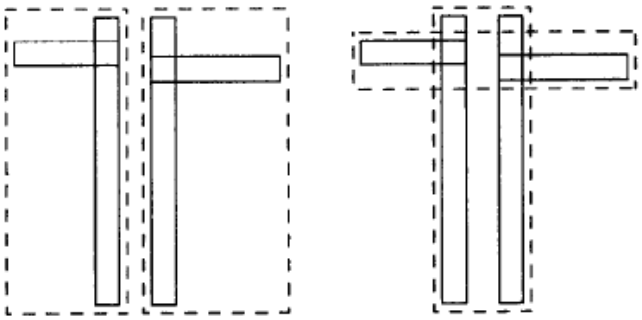

Figure 2.4: An example of a bad split (left) and a good split (right) [5]

In the following years, different variants of R-trees were introduced: R^+-tree, R^*-tree, Hilbert R-tree, X-tree, etc. R-trees have been used for the organization of multi-dimensional data in a wide range of applications. Besides VLSI design, which was their initial application, R-trees also contribute to geographic information systems for the representation of cities, buildings, etc. Many kinds of databases, such as image, music or video databases, rely on R-trees for the storage and retrieval of data. Finally, R-trees have proven to be very efficient for manipulating moving points and trajectories [4].

2.1.3 Voronoi Diagram

Let's assume we have n sites, which can be points, line segments, etc, and we assign every point to its closest site, with closeness being defined by a metric distance, e.g. Euclidean distance. In that way, we create a subdivision, which is called Voronoi diagram. A schematic example is given at the following figure: We have n points (sites) and the Voronoi diagram that results from the use of the Euclidean distance is shown in the following figure.

A simple algorithm can compute the Voronoi diagram of n sites in $O(n^2 logn)$ time. At first, it computes the bisector of each pair of points (sites), i.e. the perpendicular bisector of the line segment that connects the two points. In that way, the space is divided into half planes. If we take the intersection of all the half-planes that resulted from the bisector of a site s and any other point and contain the site s, we find the Voronoi cell that corresponds to that site. We can compute each Voronoi cell in $O(nlogn)$ time (see also [3]) and, thus, the total complexity is $O(n^2 logn)$.

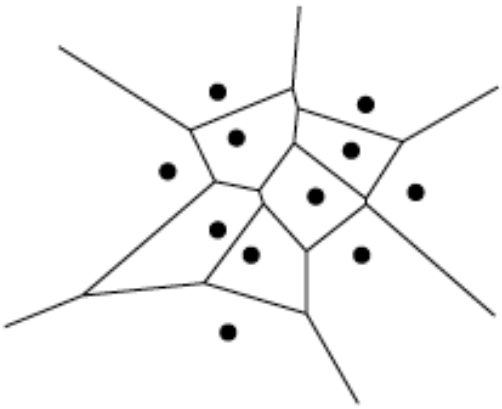

Figure 2.5: Voronoi Diagram [3]

Of course, there are faster algorithms, such as Fortune's algorithm, which has $O(nlogn)$ complexity. Fortune's algorithm is optimal for the problem, since the problem of sorting n numbers is reducible to the computation of the Voronoi diagram. Thus, a $\Omega(nlogn)$ limit is imposed. The main idea of the algorithm involves a horizontal line, which starts from the top of the plane and is swept towards the bottom. During sweeping, the algorithm maintains the information concerning the part of the Voronoi diagram above the sweep line that cannot be influenced by sites below the sweeping line. The reader can also see [3] Chapter 7 for a detailed presentation of the algorithm.

There are multiple variations of Voronoi diagrams. One of them computes the division of the space, where all points of each cell have the same site that is farther than any other. In another example, higher-order Voronoi diagrams have n-th order cells which contain points that have the same n nearest neighbor. Moreover, Voronoi diagram can be generalized for higher-dimensions. The definition remains the same: Each site has a corresponding cell, which contains the points that are closer to that site than any other. However, the complexity of its computation can be as high as $O(n^{\lceil d/2 \rceil})$ for d dimensions and its construction can be complicated because of degeneracies and numerical inaccuracies [19].

One of the initial problems for which Voronoi diagram was used is the following: Imagine the map of a city which depicts the n post offices of the area as n points. If we assume that the cost and the quality of the service of each post office are the same and that people reasonably select the post office that is the nearest to their house, Voronoi diagram offers a subdivision of the city, such that all people in the same region would select the same post office to post their letters. Similar questions

arise when we need to estimate the number of clients a new shop would attract, if the sites represent the competitor shops of the city, and in social geography, when we want to study the economic activities in a country, e.g. what is the trading area of certain cities [3]. Of course, Voronoi diagrams are not only used in geography. They are applicable in robotics, machine learning, geometry and many other fields.

2.2 Related Problems

There are many problems of computational geometry that deal with questions closely related to the topic of this thesis. Of course, they are not only useful in mathematics, but arise in many real-world situations, too. In the following sections, we present some of these problems. We give a short description of the problem, present some techniques and methods that are used for its solution, which often involve some of the data structures we analyzed above, and, finally, mention some of its applications.

2.2.1 Largest Empty Rectangle Among Points

Description of the Problem

Given a rectangle containing n points, we want to find the largest - area subrectangle with sides parallel to those of the original rectangle that can be placed among the n points.

Solution

This problem has received a lot of attention in computational geometry and there are various proposed solutions. We will present the general idea of the solution of Chazelle et al. [9], which achieves $O(nlog^3n)$ time and $O(nlogn)$ space.

First of all, we notice that the largest area subrectangle must have all its sides restricted by either at least one of the n points or the bounding rectangle. In the opposite case, the edge that does not contain at least one point or part of a bounding rectangle's edge can be shifted such that the subrectangle is enlarged.

This algorithm is based on a divide and conquer technique. The n points are sorted by their x-coordinate and then they are divided into two halves by x-

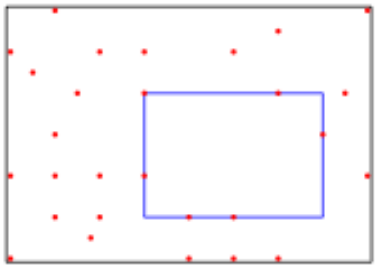

Figure 2.6: Example of the largest empty rectangle problem [20]

coordinate. The algorithm solves the problem recursively for each set of points: $S_1 = \{p_1, \ldots, p_{\lfloor n/2 \rfloor}\}$ and $S_2 = \{p_{\lfloor n/2 \rfloor + 1}, \ldots, p_n\}$. The bounding rectangles for each recursive call are adjusted as follows: Each call that corresponds to the first set (smaller x values) has $x_{\lfloor n/2 \rfloor}$ as the right edge of the bounding rectangle, while each call that corresponds to the second set has $x_{\lfloor n/2 \rfloor + 1}$ as the left edge of the bounding rectangle. These calls determine the largest empty rectangle with all four supporting points or edges in one half or the other.

We now need to examine the remaining cases: The rectangles that have three supporting points or edges in one half and one point in the other and the rectangles that have two supporting points or edges in each half. We then compare the largest empty rectangle of each category and we find the largest empty rectangle among the n points. Chazelle et al. proved in [9] that the largest rectangle with three supporting points or edges in one half and one in the other can be computed in $O(n)$, while the largest rectangle with two supports in each half needs $O(nlog^2 n)$ time.

The recursive solution gives [9]:

$$T(n) \leq 2T(n/2) + C(n) + D(n)$$

where $C(n) = O(n)$ is the required time for rectangles with three supports in one half and one in the other and $D(n) = O(nlog^2 n)$ is the required time for the rectangles with two supports in each half. As a result, $T(n) = O(nlog^3 n)$ is the total time complexity of the algorithm.

Applications

This problem is applicable in many areas. For example, in material manufacturing we can consider a piece of sheet metal, fabric or plastic, as the rectangle and

some faulty points on its surface as the n points. In that case, the solution to the largest empty rectangle problem reveals the largest-area rectangular piece of material that can be salvaged. Another application concerns floor space planning. The rectangle is a floor or a room of the house, the n points are permanent objects, such as pillars, or forbidden areas and the goal is to find the largest available space for placing equipment, etc.

2.2.2 Largest Empty Rectangle Among Rectangular Obstacles

Description of the Problem

Given a rectangle containing smaller rectangular obstacles, the goal is to find the largest empty subrectangle that can be placed among them. All rectangles have the orientation of the original rectangle.

Solution

Handa et al. [11] propose a solution for this problem considering that the rectangle is an FPGA (Field - Programmable Gate Array) surface and the task is to find the largest rectangular empty area on that surface. The rectangular surface has x columns and y rows and each cell of the array is a cell of the FPGA. They use a positive number to represent each empty cell and a negative for each occupied cell of the matrix. The positive number equals the total number of empty contiguous cells above and including the current one in that column, while the negative number equals the total number of contiguous occupied cells in the right and including the current cell. An example of the representation of cells with positive and negative numbers is given in the following figure.

An empty rectangle is called maximal if it cannot be fully covered by any other empty rectangle. Handa et al. [11] use a staircase data structure for the representation of maximal empty rectangles. An example of this data structure is presented in the following figure.

Each staircase contains a collection of overlapping maximal empty rectangles. The staircase is constructed row by row from top to bottom and each row is scanned from left to right direction. A staircase is maximal if it cannot be extended down or to its right. Otherwise, the rectangles of the extended staircase cover the rectangles of the current staircase and, thus, it is not maximal. Once a maximal staircase is constructed, we can check all of its rectangles to see if they are maximal and

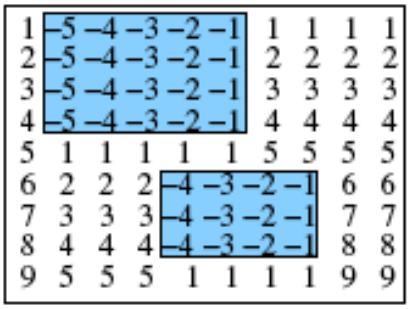

Figure 2.7: Representation of obstacles with negative numbers and empty space with positive numbers. (figure taken from [11])

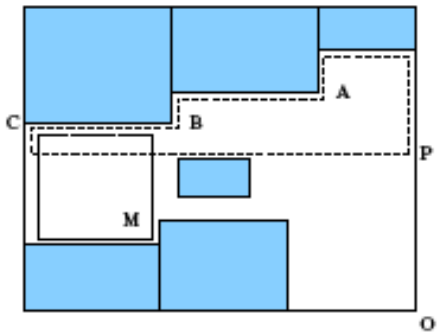

Figure 2.8: Example of staircases of maximal empty rectangles. (figure taken from [11])

all maximal rectangles are reported. This procedure is realized for each maximal staircase. This algorithm is generally fast and its worst performance occurs when a staircase needs to be made at every column in every row. In that case, we have the worst case performance of $O(xy)$. The space complexity of the algorithm is $O(min(x, y))$.

Applications

As we already mentioned, one application concerns the FPGA. If the original rectangle is an FPGA surface, we can think of the rectangular obstacles as the tasks that run on the FPGA at a specific moment. A task remains on the FPGA until it finishes its execution and it is then removed and the space it occupied can be used by a next task. The solution of the largest empty rectangle problem can quickly reveal an available space large enough for the placement of the following task.

2.2.3 Largest Rectangle Containing a Query Point

Description of the Problem

Given a rectangle that contains n points and a query point, the goal is to find the largest subrectangle that contains the query point and none of the n points. The orientation of the subrectangle is the same as the original rectangle's.

Solution

An algorithm for this problem is introduced by Gutierrez et al. [20]. This algorithm uses an R-tree for the storage of the points and uses the properties of the minimum bounding rectangles (MBRs) of the R-tree to avoid inspecting as many nodes as possible. The algorithm consists of two main steps.

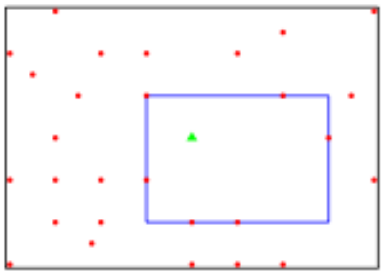

Figure 2.9: Example of the largest rectangle containing a query point [20]

Step 1. First of all, a set of points C is created. After the examination of the parent nodes of R-tree's leaves, we include in C the most distant points of each node's MBR to the query point. An example is illustrated in the following figure.

The points of C most probably are not part of the set of the actual n points. Using C as input, the algorithm computes the largest rectangles that contain only the query point and none of the points of C. These rectangles are called CERs (candidate maximum empty rectangles). As illustrated in the corresponding figure, not all CERs are the largest empty rectangles that contain the query point. More specifically, the figure shows two CERs, A and B. It is obvious that both rectangles cannot be enlarged and yet B is not the largest rectangle containing q, since A is larger.

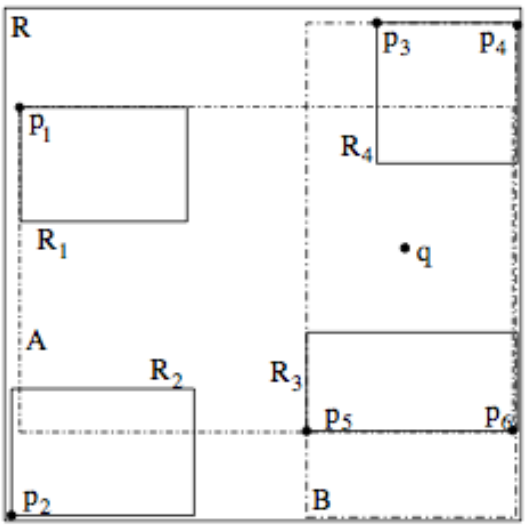

Figure 2.10: The query point q, the MBRs of the R-tree: R_1, R_2, R_3 and R_4 and the most distant points for each one. The set C that corresponds to this figure is the $C = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ [20]

Step 2. Having computed all the CERs, the algorithm sorts them according to their area, form largest to smallest. Then, for each CER, it accesses the leaves of the R-tree that contain the real points (the points that belong to the initial set of n points) that intersect with that CER. These real points form a set C'. Using this set as input, the algorithm computes the largest rectangles that contain only the query point. These rectangles are called MERs (maximum empty rectangles) and are similar to CERs except from the fact that MERs are produced using real points. MERs are always smaller or equal than the CER from the processing of which they were produced. The largest rectangle among these MERs is considered a candidate solution. As more CERs are processed, we update the candidate solution, keeping each time the one with the largest area. When the CER processing terminates, the candidate solution is the largest rectangle containing only the query point.

Applications

We can consider many applications of this problem in a wide range of areas. First of all, in data mining, we often need to check if certain ranges of values appear together. One representative example described in [20] concerns the database of a bank, where the amounts of money and the dates of each deposit are stored. We can consider a graph for the data, which represents the amount of money on the

x axis and the dates on the y axis. By selecting a certain amount of money and a specific date as a query point and finding the largest empty rectangle around that point, we can discover if there is a period of time without any deposits of a specific range of money. If, for example, for a period of time there are no deposits of more than one million dollar, we could assume that this may indicate the beginning of an economic crisis.

Another example is related to GIS (Geografic Information Systems) applications. Assume we want to construct a new building in a city at a specific location. If the city is the original rectangle and the already existing constructions are the n points, then the computation of the largest empty rectangle containing the location in which we are interested can reveal the limits that are imposed on the size of the new building. Finally, other applications include electronic design automation, such as the design of physical layout of integrated circuits, etc.

2.2.4 Largest Empty Maximal Hyper-Rectangle in Multi-Dimensional Space

Description of the Problem

This problem constitutes the generalization of the largest empty rectangle problem in many dimensions. The formulation of the problem is the following: Consider a d-dimensional space with n points contained in a d-dimensional box. The goal is to find the largest hyper-rectangle that is located inside the box and does not contain any of the n points. The bounding box and the requested hyper-rectangle are both axis-aligned.

Solution

There are several variations of this problem. In particular, the goal may be to find the largest empty hyper-rectangle or the maximal empty hyper-rectangles that are large enough or even all maximal empty hyper-rectangles. A hyper-rectangle R is considered maximal, if there does not exit another empty hyper-rectangle R' such that the entire R falls within the interior of R'. A maximal empty hyper-rectangle (MHR) has all its $2d$ surfaces bounded, which means that each surface contains at least one point or part of the boundary.

The main idea of the algorithm proposed by Ku et al. [10] is as follows. We begin with an empty set of points. That means that the whole bounding rectangle constitutes one and only MHR. Then, we add incrementally the n points, one point

at a time, and maintain a set with the existing MHRs. Every time a new point is to be added, we identify all the MHRs that contain the point. If the point is located on the surface of a MHR, then the MHR is just updated with the new bounding point. However, if the point is in the interior of the MHR, the latter has to be deleted since it is no longer empty. For each such deleted MHR, we can create new empty hyper-rectangles by splitting it at the reference point. More specifically, if we split each such MHR at the reference point along each dimension, we get $2d$ new empty hyper-rectangles. If these hyper-rectangles are maximal, then they are inserted in the set of the existing MHRs. In case that the goal is to find the MHRs that are large enough, then MHRs that are not sufficiently large are not even inserted in the MHRs set.

Applications

This problem arises in data mining, where there are datasets with k attributes. These form a k-dimensional space and the largest empty hyper-rectangle denotes the range of attributes where there are no data tuples. This can be useful, as described in [10], in the example of medical databases, where the attributes of the database are symptoms of diseases. In that case, the empty hyper-rectangle indicates the absence of the corresponding symptoms for a certain disease and that could be used in studying or finding a cure for that disease.

Moreover, in a multi-processor machine, processors are connected in a hyper-cube mesh. We can consider the processors as a multi-dimensional space, where the n points represent n busy processors. For each requested job in the system, it is necessary to find a mesh of connected processors of a certain size that can satisfy the job. Therefore, the problem is equivalent to computing the largest empty hyper-rectangle in this multi-dimensional space of the mesh of processors.

2.2.5 Nearest Neighbor Problem

Description of the Problem

Given n points in the space, find the point that is the closest to a specific query point. Closeness can be defined with the use of a distance metric, such as Euclidian distance, Manhattan distance etc, or any other dissimilarity function, according to which the less similar the points are, the larger the returned value of the function is.

Solution

One approach presented in [23] uses the Voronoi diagram in order to solve the closest point problem. Assume we have n points. First of all, we construct the Voronoi diagram. Each cell of the diagram contains all points that have the same nearest neighbor. Thus, the only task is to find to which Voronoi cell the query point belongs. If that cell is identified, the problem is solved.

In order to find the correct cell, first of all, a horizontal line is drawn through each vertex of the diagram and the plane is partitioned into slabs (see the following figure).

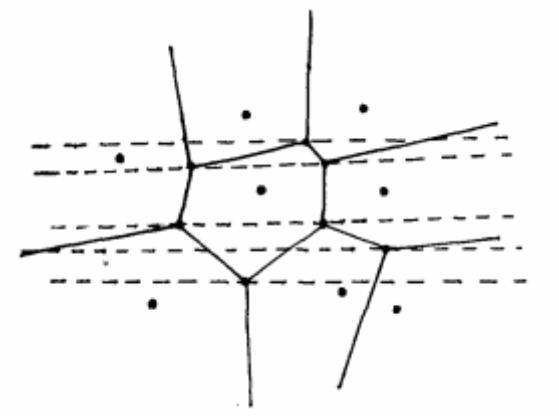

Figure 2.11: Example of slabs in a Voronoi diagram [23]

A preprocessing that sorts the Voronoi points according to their y coordinates allows the identification of the correct slab in $O(logn)$ time using binary search, since there are at most $2n - 3$ slabs [23]. Each slab can contain up to $O(n)$ line segments, because the whole Voronoi diagram has $O(n)$ segments. Therefore, the polygon to which a point belongs can be identified in $O(logn)$ time.

Figure 2.12: Example of one slab [23]

A different approach uses R-Trees for the solution of this problem. The algorithm presented in [24] introduces two useful distance metrics: MINDIST and MINMAXDIST. The first one is defined as the minimum possible distance of a query

point Q and an object O, while the second one corresponds to the minimum of the maximum possible distances between Q and O. These metric distances impose a lower and upper bound on the actual distance of O from Q [4].

A more formal definition of these metrics follows [4]: Imagine a rectangle R in the d-dimensional Euclidean space, whose major diagonal endpoints are the points $S(s_1, ..., s_d)$ and $T(t_1, ..., t_d)$ and let's denote R as $R = (S, T)$. If the Euclidean distances are squared for simplicity, we get the following definitions:

Definition 2.1. MINDIST: The MINDIST of a query point Q and a rectangle $R = (S, T)$ is defined as:

$$\text{MINDIST}(Q, R) = \sum_{i=1}^{d} |q_i - r_i|^2$$

where:

$$r_i = \begin{cases} s_i & \text{if } q_i < s_i \\ t_i & \text{if } q_i > t_i \\ q_i & \text{otherwise} \end{cases}$$

Definition 2.2. MINMAXDIST: The MINMAXDIST of a query point Q and a rectangle $R = (S, T)$ is defined as:

$$\text{MINMAXDIST}(Q, R) = \min_{1 \le k \le d} \left(|q_k - rm_k|^2 + \sum_{\substack{i \neq k \\ 1 \le i \le d}} |q_i - rM_i|^2 \right)$$

where:

$$rm_k = \begin{cases} s_k & \text{if } q_k \le \frac{s_k + t_k}{2} \\ t_k & \text{otherwise} \end{cases}$$

and:

$$rM_i = \begin{cases} s_i & \text{if } q_i \ge \frac{s_i + t_i}{2} \\ t_i & \text{otherwise} \end{cases}$$

The algorithm includes pruning of the tree which is based on the following criteria.

1. An object O is pruned if there is an object O' such that MINDIST$(Q, mbr(O))$ is greater than MINMAXDIST$(Q, mbr(O'))$, where $mbr(O)$ is the minimum bounding rectangle of an object O.

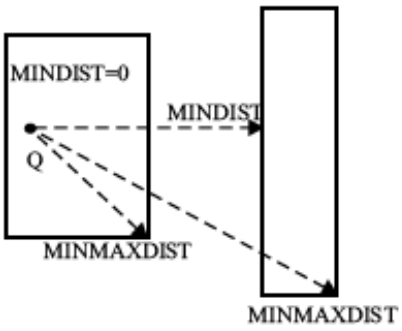

Figure 2.13: Example of MINDIST and MINMAXDIST [4]. If the query point is inside a rectangle, then its MINDIST from that rectangle is 0.

2. An object O is discarded, if the actual distance from the query point Q to O is greater than the than MINMAXDIST$(Q, mbr(O'))$, where O' is another object.
3. An object O is discarded, if MIDIST$(Q, mbr(O))$ is greater than the actual distance of Q to another obstacle O'.

The branch-and-bound traversal algorithm [24] starts from the root of the tree and for each node: If the node is internal, the node's children minimum bounding rectangles are sorted according to their MINDIST into an active branch list. That listed is pruned based on criteria 1 and 2 and this procedure is performed recursively for each member of the active branch. When we reach a leaf node, we compute the actual distance between the query point and each obstacle and we update a variable that maintains the nearest object so far. Finally, we apply an upward pruning according to the third criterion. For a more detailed description of the algorithm, the reader can refer to [4] or [24].

Applications

This problem has numerous applications. It arises, for example, in the area of computational geometry when we need to find the closest pair of points on a plane. Other areas include pattern recognition, e.g. recognition of written or printed characters in order to convert them into computer - readable text, as well as computer vision and databases, e.g. content-based search of images in large databases. Another example is the use of nearest neighbor search for plagiarism detection and suggesting the correct spelling in spell checking applications.

2.2.6 k-Nearest Neighbors Problem

Description of the Problem

This problem is similar to the Nearest Neighbor problem, except that we are not looking for the nearest neighbor of a query point, but for its $k > 1$ nearest neighbors. Its description is the following: Given a set of n points in the space and a query point, we wish to find the k points that are closest to the query point.

Solution

Shamos [23] introduces a solution for the k-Nearest Neighbors problem using a k-th order Voronoi diagram. As already mentioned, the order k Voronoi diagram consists of cells that contain all points that have the same k nearest neighbors. Assume we have n points. First of all, we construct the k-th order Voronoi diagram, which has $m = O(k(n-k))$ vertices. We then need to find the cell to which the query point belongs. According to the analysis of the previous section, this can be done in $O(logm) = O(logn)$ time. The complexity of displaying the result is $O(k)$, therefore, the total complexity of the algorithm is $O(max(k, logn))$. The required storage is $O(k^2(n-k)^2)$ [23].

Another approach presented in [4] uses R-Trees for the solution of the k-Nearest Neighbors problem. This solution is similar to the procedure that was analyzed for the Nearest Neighbor problem, with the modification that each time we maintain the k nearest obstacles and use the furthest one among them in order to perform the pruning.

Applications

The k-Nearest Neighbors problem arises in many cases, such as the classification of objects in pattern recognition. In that case, every object is classified into groups according to the voting of its k neighbors and the class in which the majority of these neighbors belong. Therefore, it is obvious that there is a need for fast computation of the nearest neighbors of each object. Another application, also in the area of pattern recognition, is regression. In that example, every object is assigned a specific value according to the average of the values of its k-nearest neighbors.

2.2.7 All Nearest Neighbors Problem

Description of the Problem

Given n points, we need to find the point that is closest to each one. The closeness is again defined with the use of a distance metric, such as Euclidian or Manhattan distance.

Solution

One naive solution of this problem includes computing the distance of each point to all other points and keeping the shorter distance among them, which reveals the nearest neighbor. This approach requires the computation of n^2 distances and, therefore, has a complexity of $O(n^2)$, which is not satisfying as it is proven that the lower bound for this problem is $O(nlogn)$ [23].

Shamos [23] introduced the use of Voronoi diagram for the solution of the problem in $O(nlogn)$. More specifically, based on the definition of Voronoi diagram, the edges of each cell are part of the perpendicular bisector of the internal point of the cell and its nearest neighbors. That means that the perpendicular bisector of a point p_i and its nearest neighbor p_j will also partially coincide with an edge of p_i's Voronoi cell. Therefore, for each point p_i, we can examine the edges of its cell and find the nearest one, which corresponds to its nearest neighbor p_j. In that way, the nearest neighbor of a point requires only the examination of all edges of its cell. It is known that Voronoi diagram has n cells - one for each point. The total number of the diagram edges is $O(n)$ and, since each edge belongs to two cells, it is examined twice. So, the total complexity of the search is $O(n)$, if we have already constructed the diagram. Voronoi diagram can be constructed in $O(nlogn)$ time (see 2.1.3) and, thus, the total complexity of this approach is $O(nlogn)$, which renders it optimal.

Applications

As mentioned in [15], we can use the all nearest neighbors problem for the estimation of the entropy of a system or process based on certain observations. The entropy observation is very useful in areas such as image analysis and speech recognition. In order to estimate the entropy, we firstly need to find the nearest neighbor of each point of our dataset. Then, the estimation is made based on the distribution of the distances between each point and its nearest neighbor.

Chapter 3

Finding the Largest Empty Cube in Manhattan Geometries

3.1 Manhattan Geometry

According to [21], a pattern has a Manhattan geometry if the following conditions are satisfied: First, all objects are rectangles and if two rectangles share an entire edge, they are merged into one. Second, all edges of rectangles have either horizontal or vertical orientation. Third, all edges have length $\geq d$, where d is the minimum size of the pattern. An example of a Manhattan geometry is presented in the following figure. This definition is easily extended in the 3D space.

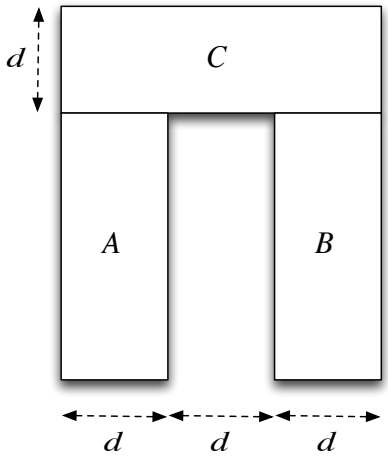

Figure 3.1: Example of Manhattan geometry [21]

3.2 Description of the Problem

Our input is a query point, some rectangular cuboids as obstacles and a bounding box of the cuboids. The goal is to find the largest empty cube that has the given point as center and is located entirely inside the bounding box.

It is obvious that there must be at least one point of contact between the cube and an obstacle or the bounding box. In the opposite case, the cube could be enlarged until it touched an obstacle or the bounding box and, thus, it would not be the largest cube. This observation leads to the conclusion that the problem of finding the largest empty cube with a given center is equivalent to finding the nearest obstacle (could be the bounding box, too) to the query point.

There are many possible solutions to this problem. The simplest one is to find the distances between the query point and all obstacles (including the bounding box) and then take the smallest one, which corresponds to the nearest obstacle. This algorithm is linear to the number of cuboids, but it is not efficient enough when we deal with geometries with millions of cuboids, where the process of finding the largest empty cube must be realized for millions of different centers.

Therefore, we need a more efficient algorithm. One of the main characteristics of this algorithm must be the restriction of the search for the nearest obstacle only in the neighborhood of the query point. This technique can significantly reduce the query time and achieve a much better performance. In order to achieve that, we need a space management technique for a convenient storage of the obstacles. The data structure that we selected is the octree, since it offers a simple yet efficient division of the space. In the following sections, we present some basic definitions and a detailed description of the proposed algorithm.

3.3 Main Idea of the Algorithm

The algorithm uses an octree for the representation of the three-dimensional space. The root of the tree represents the cell of the space which contains the obstacles. In our case, we can select the root to be the bounding box that is given as an input. Each node of the tree maintains a list, which is called candidate list and contains all the obstacles, such that given any query point inside the cell its nearest obstacle is in the list.

Each query begins with the traversal of the tree, in order to find the right leaf. Then, the distance of the query point and every obstacle of the leaf's candidate list is computed and the nearest obstacle is obtained. The distance of that obstacle and

the query point determines the largest possible side of the empty cube and along with the center of the cube, which is the query point, we have all the required information. We notice here that, since we work in Manhattan geometries, the requested cube will always have its sides parallel to the bounding box, i.e. parallel to the three cartesian planes.

3.4 Definitions

In this section, we are going to introduce some definitions, which are necessary for the comprehension of the algorithm.

Definition 3.1. Given a rectangular cuboid A, we call $x_{min}(A)$ and $x_{max}(A)$ the minimum and the maximum x coordinates of A, $y_{min}(A)$ and $y_{max}(A)$ the minimum and the maximum y coordinates and, finally, $z_{min}(A)$ and $z_{max}(A)$ the minimum and the maximum z coordinates.

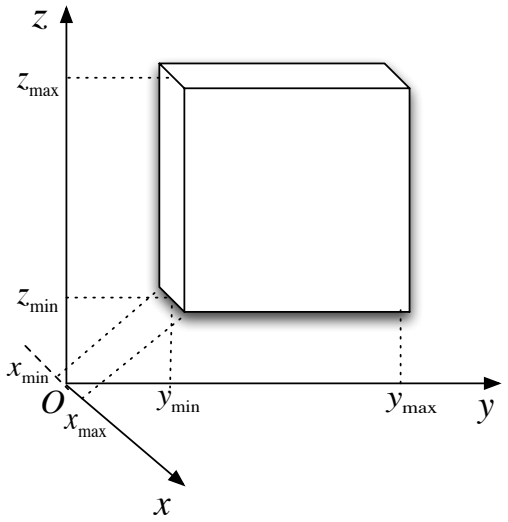

Figure 3.2: Rectangular cuboid

Definition 3.2. The $x - distance$ between two rectangular cuboids A and B is defined as the maximum of the values $x_{min}(B) - x_{max}(A)$ and $x_{min}(A) - x_{max}(B)$. The $y - distance$ and the $z - distance$ are defined similarly.

Definition 3.3. The $x - distance$ between a point P and a rectangular cuboid A is defined as the maximum of the values $x_{min}(A) - x(P)$ and $x(P) - x_{max}(A)$. The $y - distance$ and the $z - distance$ are defined similarly.

Definition 3.4. The distance d between two rectangular cuboids is defined as the maximum of the $x - distance$, $y - distance$ and $z - distance$ of the cuboids.

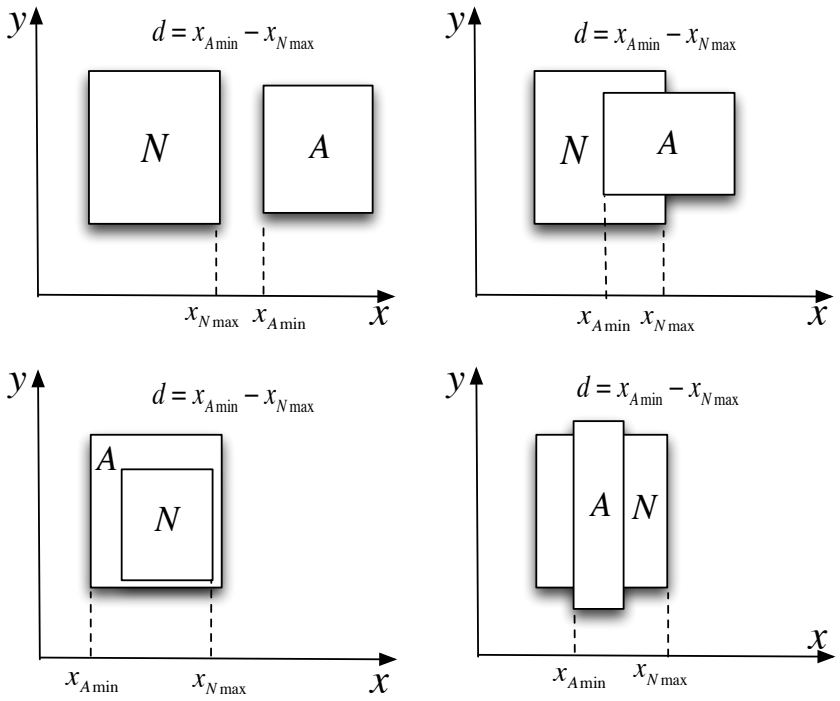

Figure 3.3: Some simple examples for the calculation of the distance between an obstacle A and a cell N (also see [2]). Same applies to 3D geometries.

Definition 3.5. The distance d between a point P and a rectangular cuboid A is defined as the maximum of the $x - distance$, $y - distance$ and $z - distance$ between the point and the cuboid.

Definition 3.6. The size of a spatial cell is defined as the maximum of its length, width and height.

Definition 3.7. The domination relationship between two cuboids is defined as follows: A cuboid A dominates a cuboid B regarding a spatial cell, if each point of the cell is closer to cuboid A than B [2]. It is obvious that, in this case, B can never be the nearest neighbor of any point in the cell.

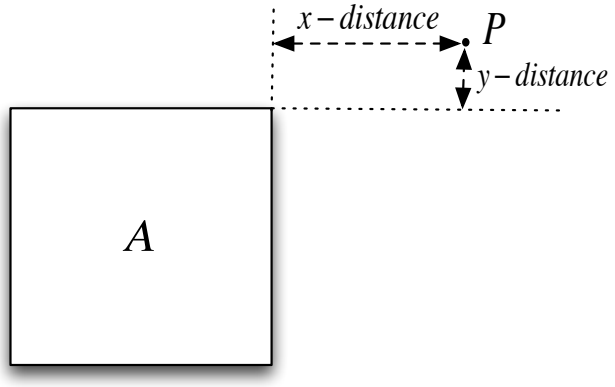

Figure 3.4: Calculation of the distance between a point P and an obstacle A. The distance here equals to the $x - distance$. Same applies to 3D geometries.

3.5 Detailed Description of the Algorithm

The algorithm consists of two parts: The first part contains the necessary methods for the creation of the octree. The required input includes the geometry of the obstacles and the bounding box. The resulting octree is then available to the second part, which is responsible for the execution of the queries. Therefore, we create the tree once and, then, each time we need to find the largest empty cube with a specific center, we can directly use the methods of the second part without needing to reconstruct the tree.

3.5.1 Creating the octree

Inserting obstacles into the tree

As we have already mentioned, each node of an octree corresponds to a subdivision of space, also called spatial cell. In the beginning of the algorithm, the tree consists only of a root node, which corresponds to the bounding box that is given as an input. The candidate list of the root node is initially empty.

We examine successively all the obstacles and insert them into the tree. Each obstacle is first inserted into the root node. In order to determine if the obstacle belongs to the cell in which it is being inserted, we need to perform some checks. We analyze the necessary checks in the following section. In case that the obstacle belongs to the cell, if the latter has children, it is inserted recursively into each

child node. If the cell is a leaf node, the obstacle is inserted in its candidate list. When the length of the candidate list exceeds a predetermined limit, and given that the height of the tree is less than a threshold, the leaf node is divided into 8 octants, which become its children, and all the obstacles of its candidate list are inserted into the child nodes.

Algorithm 1 Inserting a cuboid into a tree node

```
1: procedure INSERT(cuboid A, node N)
2:     d ← distance(A, N)                          ▷ Distance according to Def. 3.3
3:     if d ≥ DistanceThreshold(N) then return
4:     if N is a leaf node then
5:         CHECK(A, N)
6:         if (tree height < HeightThreshold) and (N's candidate list's length
7:             > CandidateListThreshold) then
8:             DIVIDE(N)                    ▷ Divides the cell N into 8 equal octants
9:             for each cuboid a of N's candidate list do
10:                for each child N_i of N do
11:                    INSERT(a, N_i)
12:    else
13:        for each child N_i of N do
14:            INSERT(A, N_i)
15:    return
```

Checking the obstacles

First of all, for each node, we maintain a value, such that if any obstacle's distance to the node is larger than that value, it cannot belong to the candidate list, since there must be another obstacle that is located nearer. Therefore, we use this value as a threshold when inserting obstacles into nodes in order to reject some of them. If an obstacle does not belong to the cell, the recursion of the insertion terminates.

The calculation of this critical value is simple: It equals the distance to the nearest obstacle so far plus the size of the cell. This value expresses the maximum possible distance between the nearest cuboid to the cell and any point located inside the cell. It is obvious that if an obstacle is located further than this distance, then it can never be the nearest one, since, no matter which point we select, there is going to be at least this one obstacle that is nearer. Every time a new obstacle

is inserted into a candidate list, we update the value, if the sum of its distance to the cell and the cell's size is smaller than the current value.

However, even if a cuboid is not rejected in the previous step, i.e. it is located in the cell's neighborhood, there are cases where it can never be the nearest obstacle of any point in the cell. That happens if a cuboid is dominated (see Def. 3.7) by another cuboid regarding the cell and, thus, any empty cube contacts the second cuboid prior to the first.

In order to check if there is such a cuboid, we need to traverse the candidate list once, before inserting the under examination obstacle. For each cuboid of the list, we check if it dominates or is dominated by the current cuboid. If either one of these relationships holds true, the cuboid that is dominated by the other should not be in the candidate list. If this is the case for the current cuboid, then the check stops and it is not inserted in the candidate list. Otherwise, the cuboid that is already part of the list, gets removed from it. The procedure that realizes the domination checking is described in the following section.

Algorithm 2 Checking the cuboids

```
1:  procedure CHECK(cuboid A, node N)
2:      d ← distance(A, N)                          ▷ Distance according to Def. 3.3
3:      if d ≥ DistanceThreshold(N) then return
4:      for each cuboid a of N's candidate list do
5:          if DOMINATES(a,A,N) then return
6:          else if DOMINATES(A,a,N) then
7:              Remove a from N's candidate list
8:      Insert A at the end of N's candidate list
9:      if d + size(N) < DistanceThreshold(N) then
10:         DistanceThreshold(N) ← d + size(N)
11:     return
```

Domination Checking

Assume we want to check if the obstacle A dominates the obstacle B regarding the cell N, i.e. if every point of N is closer to A than B according to the metric distance defined in Def. 3.4. If B is entirely or partially inside the cell N, then it certainly cannot be dominated by another cuboid.

In the opposite case, the first step is to compute the minimum possible size of a cube whose center is located in N and which contacts the obstacle B. That simply

means that we need to find the points of N that are closest to B. Given the fact that we deal with Manhattan geometries, these points will either be part of a side, part of an edge or a vertex of N. The minimum possible size of the cube centered at these points equals the $distance(B, N)$, where the distance is calculated according to Def. 3.3. The next step is to see what is the distance of A from all these points of N that are the closest to B (Again the distance is calculated according to Def. 3.3). If the distance of A from all these points is smaller than the one of B, then any cube centered at any of these points would contact A before B.

In order to examine if this holds true, we can "inflate" B by $d = distance(B, N)$ and create in that way the cuboid B'. This new cuboid will have $x_{min}(B') = x_{min}(B)-d$, $y_{min}(B') = y_{min}(B)-d$, $z_{min}(B') = z_{min}(B)-d$, $x_{max}(B') = x_{max}(B)+d$, $y_{max}(B') = y_{max}(B) + d$ and $z_{max}(B') = z_{max}(B) + d$. The contact area of B' and N reveals the closest points to B, i.e. the points such that a cube with a center at any of them and a side $distance(B, N)$ contacts B. We then "inflate" A by $distance(B, N)$ and call the new cuboid A'. An example of a two-dimensional case is given in the following figure.

If A' contains the entire contact area of B' and N, then all these points are closer to A than to B. Furthermore, we notice that, since these are the closest points to B, all the remaining points of N will also be closer to A than to B. Therefore, in that case, A dominates B. If A' does not contain the entire contact area of B' and N, then the points that are external to A' will be closer to B than to A and, thus, A does not dominate B.

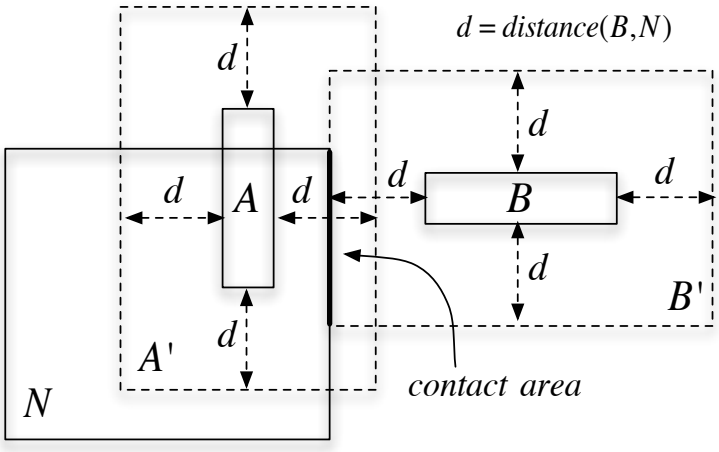

Figure 3.5: Checking if A dominates B regarding N: We can see the "inflated" obstacles A' and B' (dashed rectangles) and the contact area of B' and N. A' contains the contact area, therefore A dominates B. Similar applies to 3D geometries.

Algorithm 3 Checking if A dominates B

```
1: function DOMINATES(cuboid A, cuboid B, cell N)
2:     d ← distance(B, N)                          ▷ Distance according to Def. 3.3
3:     if d < 0 then return false    ▷ B is internal to N, cannot be dominated
4:     B' ← B inflated by d
5:     A' ← A inflated by d
6:     Find the contact area of B' and N
7:     if A' contains the contact area of B' and N then
8:         return true
9:     else
10:        return false
```

3.5.2 Executing queries

Tree traversal

The search for the largest empty cube with a given point as a center begins from the root of the tree. While the under examination node is not a leaf, we proceed to examining the child whose spatial cell contains the query point. Therefore, the traversal of the tree will not need to examine more nodes than the height of the tree. After the correct leaf has been located, we search its candidate list to find the nearest obstacle.

Algorithm 4 Tree traversal

```
1: function TRAVERSE(node N, point P)
2:     while N is not a leaf node do
3:         for each child N_i of N do
4:             if P belongs in N_i then
5:                 N ← N_i
6:     return N
```

Searching the candidate list

The searching of the candidate list is quite simple. The list is traversed linearly and the distance between each one of its members and the query point is computed according to the Def. 3.4. The smallest distance value corresponds to the nearest cuboid, which determines the size of the produced cube: Each side of the cube equals the double of that distance.

3.6 Techniques for better query performance

3.6.1 Sorting the cuboids of the candidate list

One technique that can significantly reduce the traversal of the candidate list is to keep its cuboids sorted based on their distance from the spatial cell. This sort takes place only once, when creating the octree. Of course, it slightly slows down the construction of the tree, but it offers an advantage during the search, which is crucial, since we may have numerous queries.

The main idea of this method is to prune some cuboids, which we know that do not need to be examined. We start traversing the list from the beginning and keep the distance between the point and its nearest cuboid so far. The pruning is based on the fact that if the distance of a cuboid from the cell d is larger than the nearest distance so far d_{min}, then its distance from any point inside the cell would be at least d. Therefore, this cuboid is certainly further than the nearest we have so far and there is no need to be examined. This is also true for all the obstacles that follow in the candidate list, since their distance from the cell is even larger. As a result, once the distance of the under examination cuboid from the cell becomes larger than the nearest distance so far, the list traversal terminates. One even stricter criterion is to examine if the distance of the cuboid from the cell d plus the minimum distance between the query point and the boundary d_{query_point} is larger than d_{min}. This criterion leads to an earlier pruning of the candidate list and, therefore, less obstacles are examined.

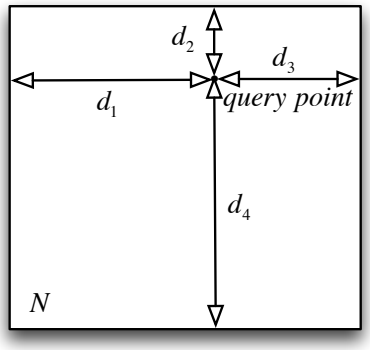

Figure 3.6: The least square size that is absolutely required in order to reach an obstacle that is located in the exterior of the cell equals the minimum of the distances d_1, d_2, d_3, d_4. Here, $d_{query\_point} = d_2$. Similarly, in three dimensions we consider the distances in the z dimension as well.

Algorithm 5 Searching the candidate list (sorted)

```
1: function SEARCH(node N, point P)
2:     DistanceToNearestObstacle ← ∞
3:     for each cuboid a of N's candidate list do
4:         DistanceToObstacle ← distance(a, P) ▷ Distance according to Def.3.4
5:         if DistanceToObstacle < DistanceToNearestObstacle then
6:             DistanceToNearestObstacle ← DistanceToObstacle
7:         if distance(a, N) + d_query_point > DistanceToNearestObstacle then
8:             break
9:     return 2 · DistanceToNearestObstacle                    ▷ Cube size
```

3.6.2 Quick access to the neighboring cells

Another technique that can accelerate the query time includes the use of pointers to the neighboring cells for each cell node. More specifically, for each cell we store all the other cells that share a common side, an edge or even a point. However, since these cells can be numerous, we select to keep pointers only to the cells of the same size as the current one and only if there is not such cell, we have pointers to neighboring cells of bigger size. This method creates a network that links all the neighboring cells.

This structure does not offer any advantages when we make the first query. However, the main application of our algorithm, which is described in the introductory chapter, ensures that each query point will not be very far from the previous one. Therefore, we can use the structure we created in order to move from cell to cell without traversing the whole tree from the root every time. In particular, the first search must be conducted by traversing the whole tree. Each following search starts from the previous leaf node and moves across the neighbors network towards the correct cell. If this cell is an internal tree node, we just traverse its subtree until a leaf node is found. As a result, the full traversal of the tree is avoided and the appropriate cell is quickly located.

3.6.3 Critical box

Finally, we notice that there are instances where the distance of some sides of the bounding box from their nearest obstacle is comparatively large. That means that there are many query points, such that their corresponding largest cube contacts the bounding box, before any other obstacles. For some of these points we can easily avoid the traversal of the octree and the search of the candidate list, based

on the following observation.

Assume we call mbr the minimum bounding box of the obstacles, i.e. the box that bounds all the cuboids and is the minimum volume box that has that property. If a point's distance from the initial bounding box is smaller than its distance from the mbr, then the derived cube will certainly contact the initial bounding box before any obstacle. We call critical box, the box whose sides are equidistant from the corresponding sides of the initial bounding box and the mbr. It is obvious that if a point is located outside the critical box, then it is certainly nearer to the initial bounding box and there is no need of examining the obstacles.

Therefore, before the creation of the tree, we can simply compute the critical box and then use it as a reference to determine if a query point requires a tree traversal or not. In fact, we can also set the root of the tree to correspond to the critical box, since, according to this technique, all tree queries will concern internal points of the critical box. As a result, this method avoids tree search for some points, as well as decreases the size of the root and, as a consequence, the size of the nodes. This may lead to smaller leaves with shorter candidate lists and, thus, accelerate the query time.

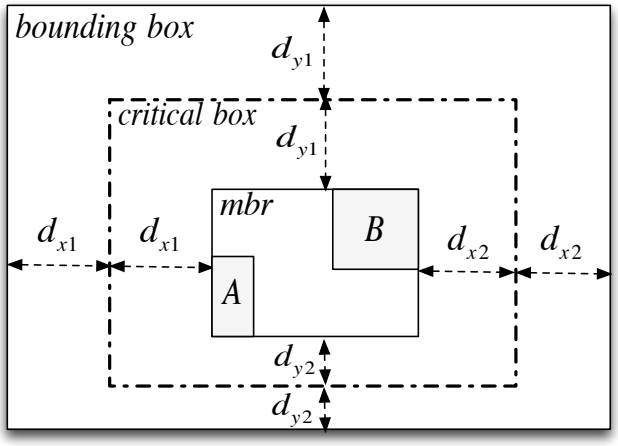

Figure 3.7: A simple example with two obstacles (A and B), their minimum bounding box (mbr), the initial bounding box and the critical box. Same applies to 3D geometries.

3.7 Example

3.7.1 Creation of the tree

In this section, we present a simple example that illustrates the main steps of the algorithm. The example is in two dimensions, but the same applies in three dimensions as well. We have five obstacles, A, B, C, D and E. In the following figure, we can see their topology.

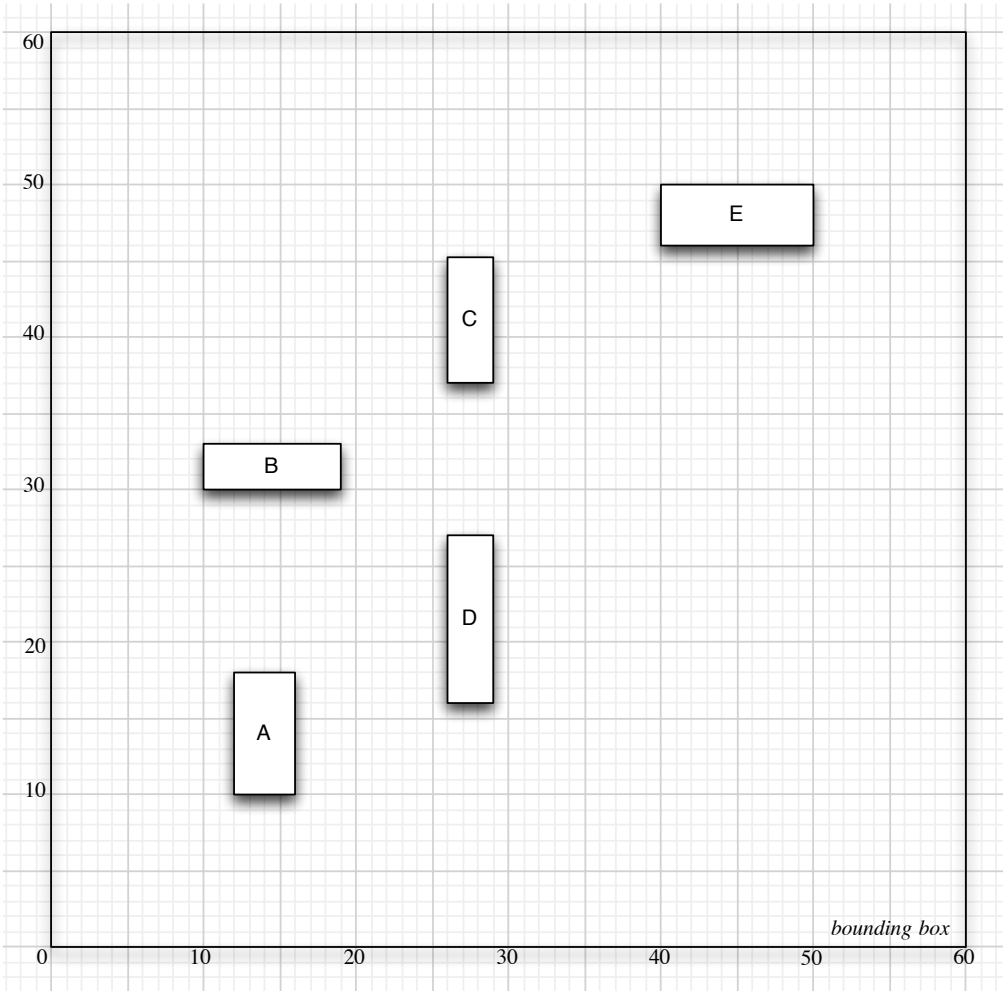

Figure 3.8: Topology of the five obstacles

Step 1

First of all, we compute the minimum bounding rectangle (mbr) of the obstacles, which is the smallest rectangle that contains all obstacles. In our case, the mbr is defined by the inequalities $10 \leq x \leq 50$ and $10 \leq y \leq 50$.

Step 2

The critical box is the rectangle whose sides are equidistant from the corresponding sides of the initial bounding box and the mbr. Therefore, it is defined by the inequalities $5 \leq x \leq 55$ and $5 \leq y \leq 55$.

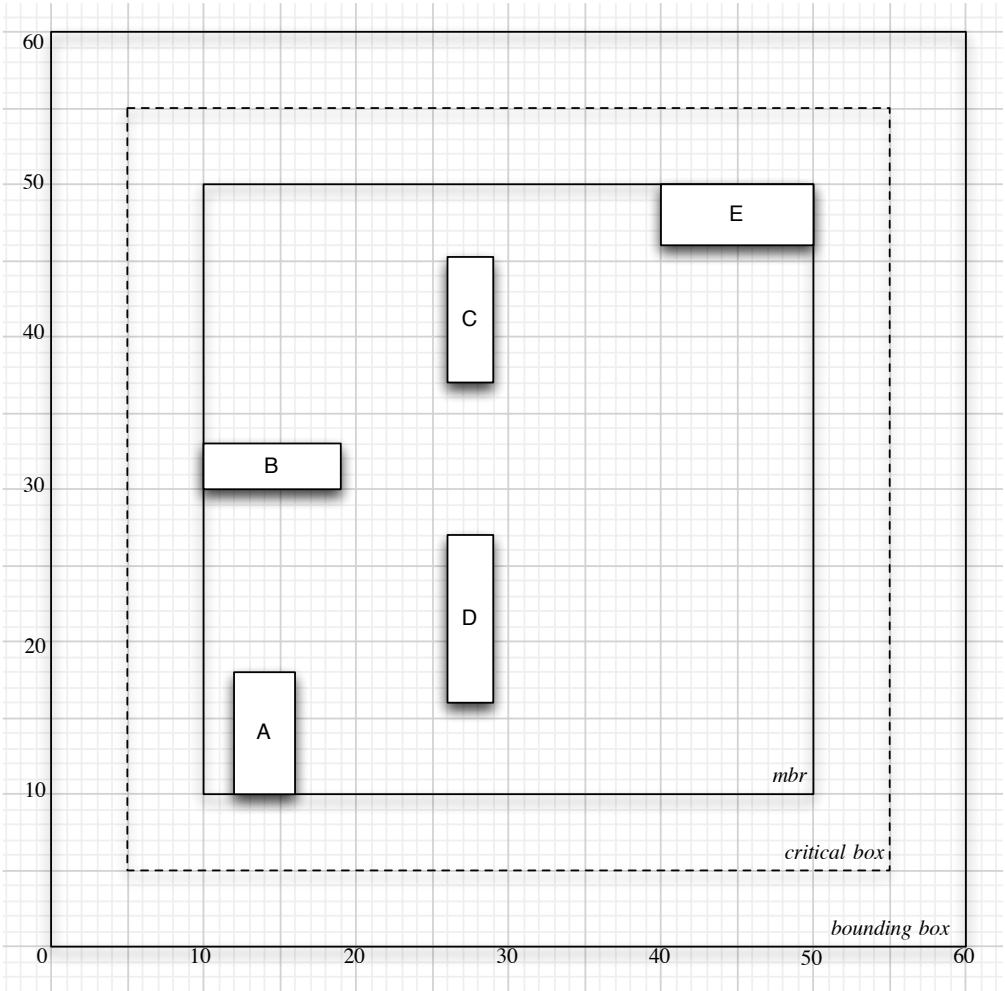

Figure 3.9: The minimum bounding rectangle and the critical box of the obstacles

Step 3

We create the root of the octree, which corresponds to the critical box $\{5 \leq x \leq 55, 5 \leq y \leq 55\}$. The candidate list is initially empty.

Step 4

We insert the obstacles in the octree. Assume that for our example we have selected the size limit of each candidate list to be 3 obstacles.

Step 4.1 We insert the obstacle A. The candidate list of the root is empty and A is located inside the root cell. Therefore, it is inserted in its candidate list.

Step 4.2 We insert the obstacle B. It is located inside the root cell and, thus, it belongs in its candidate list.

Step 4.3 We insert the obstacle C. It is located inside the root cell and, thus, it belongs in its candidate list.

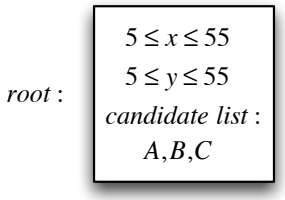

Figure 3.10: Tree after Step 4.3

Step 4.4 We insert the obstacle D. It is also located inside the root cell and, thus, it belongs in its candidate list. However, if we insert it in the root's candidate list, its size will surpass the threshold of 3 obstacles per list. Therefore, the root cell needs to be divided.

Step 4.4.1 Since we work in two dimensions, we divide the root cell into four equal parts in order to create a quadtree. Its analog in three dimensions is the octree. The four subrectangles are the following: $LL\ (LowerLeft) : \{5 \leq x \leq 35, 5 \leq y \leq 35\}$, $LR\ (LowerRight) : \{35 \leq x \leq 55, 5 \leq y \leq 35\}$, $UL\ (UpperLeft) : \{5 \leq x \leq 35, 35 \leq y \leq 55\}$and $UR\ (UpperRight) : \{35 \leq x \leq 55, 35 \leq y \leq 55\}$.

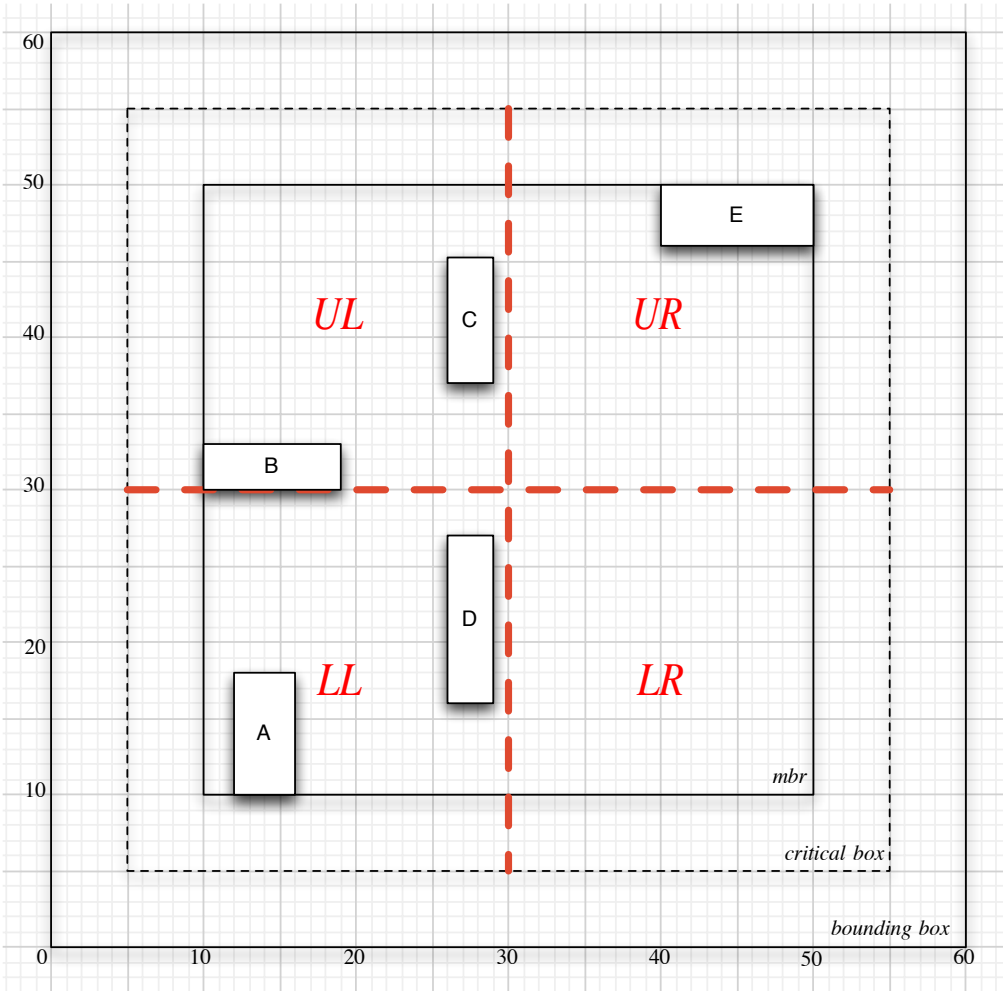

Figure 3.11: Division of the root cell into four equal subrectangles: LL $(LowerLeft)$, LR $(LowerRight)$, UL $(UpperLeft)$, UR $(UpperRight)$.

Step 4.4.2 We insert each obstacle of the root's candidate list into its child nodes.

Obstacle A:

It is located inside the LL subrectangle, thus it belongs to its candidate list. It is also inserted in the candidate list of all other three subrectangles, since they are all empty.

Obstacle B:

It belongs to the UL subrectangle and, as a result, we insert it into its candidate list. However, we can easily see that B dominates A regarding UL (also see Algorithm 3), since any square centered in UL will contact B before A, and, therefore, we remove A from the candidate list. The candidate list of the other subrectangles contains only the obstacle A so far. We need to check the domination relationship between the obstacles A and B in respect of the remaining subrectangles LL, LR, UR, using the Algorithm 3. As far as the LL rectangle is concerned, A does not dominate B, neither B dominates A, therefore they both belong to its candidate list. Similarly for the LR rectangle. However, we can see that B dominates A regarding UR. More specifically, if we "inflate" both A and B by $distance(A, UR)$, the inflated B' covers the whole common area of the inflated A' and the UR rectangle (See also the following figure for a schematic illustration). Therefore, we insert B into UR's candidate list and we remove A.

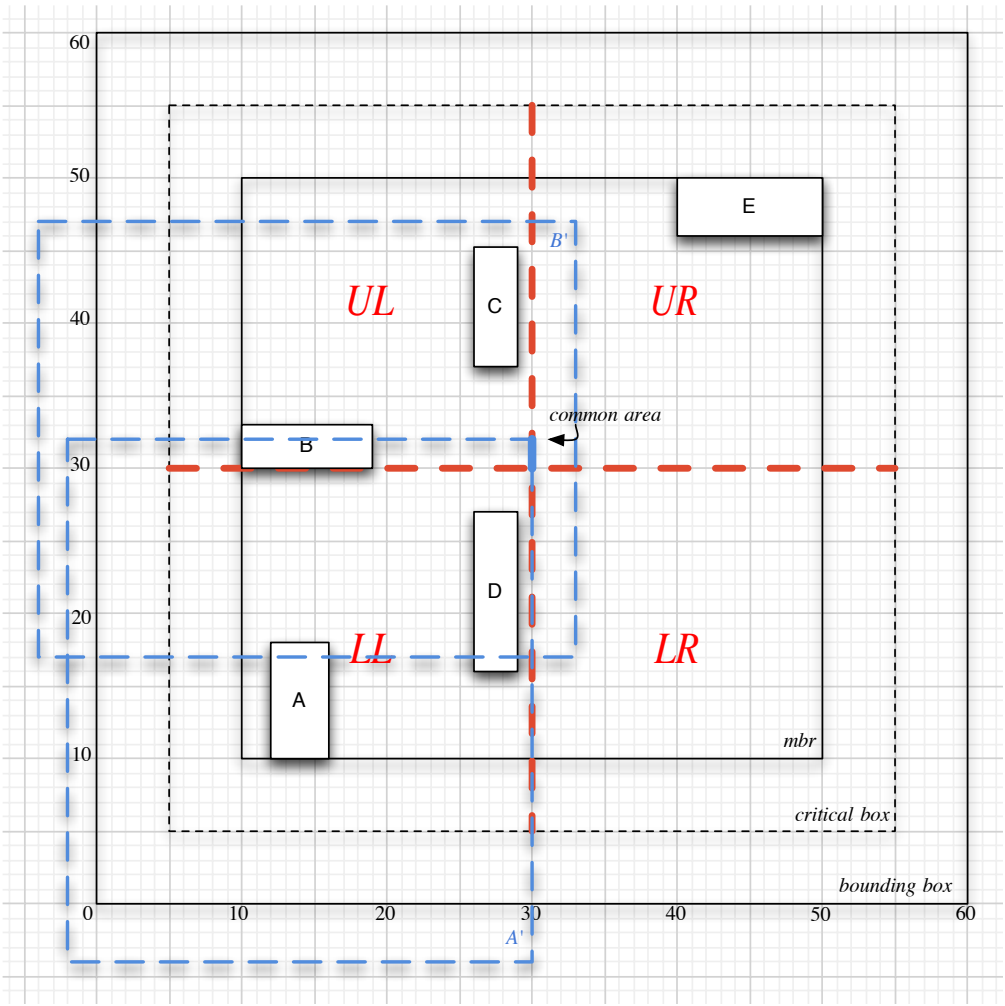

Figure 3.12: Domination checking: B dominates A regarding UR (See also algorithm 3)

Obstacle C:

The obstacle C belongs to UL and, thus, we insert it in its candidate list. As far as UR is concerned C dominates B, and, as a result, we remove B and insert C. Their is no domination relationship between the conductors regarding cells LL and LR, thus they all belong to their candidate list.

Obstacle D:

It belongs to LL. It dominates C regarding LL and so C is removed from its list. Moreover, we observe that it dominates A, B and C regarding LR as well. Thus, we remove A, B and C from the candidate list and insert D. D is also inserted

in the candidate list of the remaining rectangles, since there is no domination relationship with the rest of the obstacles.

The tree, as it is formed so far, is illustrated in a following figure.

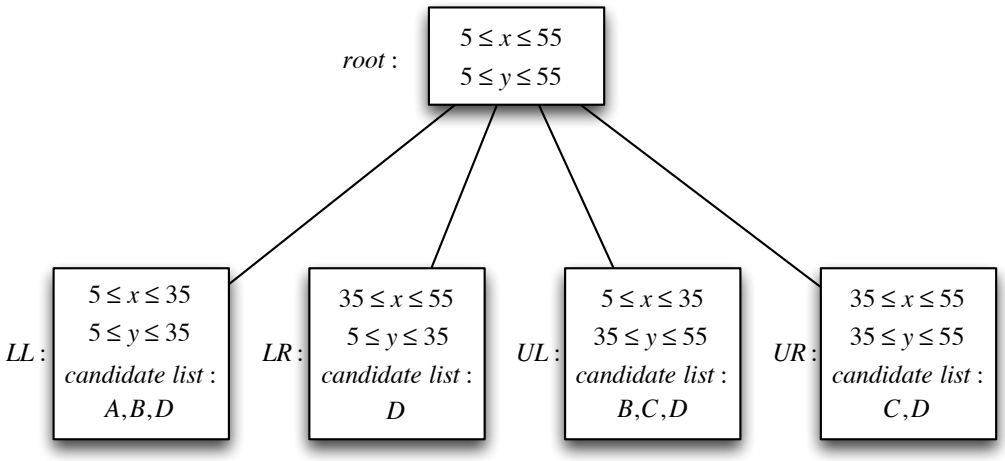

Figure 3.13: Tree after Step 4.4.2

Step 4.5 We insert the obstacle E. The root is no more a leaf node, thus we insert E into all of its children. E is located inside the UR rectangle and, thus, it belongs to its candidate list. We observe that E also belongs to the candidate list of LR as it is not dominated by any obstacle regarding that cell. However, C dominates E regarding UL and LL and, as a result, E is not inserted in their candidate lists.

Step 5

We sort the obstacles of every candidate list according to their distance from the corresponding cell. More specifically, the obstacles that are located inside the cell are in the beginning of the list and are followed by the remaining obstacles sorted by increasing distance.

The final form of the tree after the insertion of all conductors and the sorting of the candidate lists is presented in the following figure.

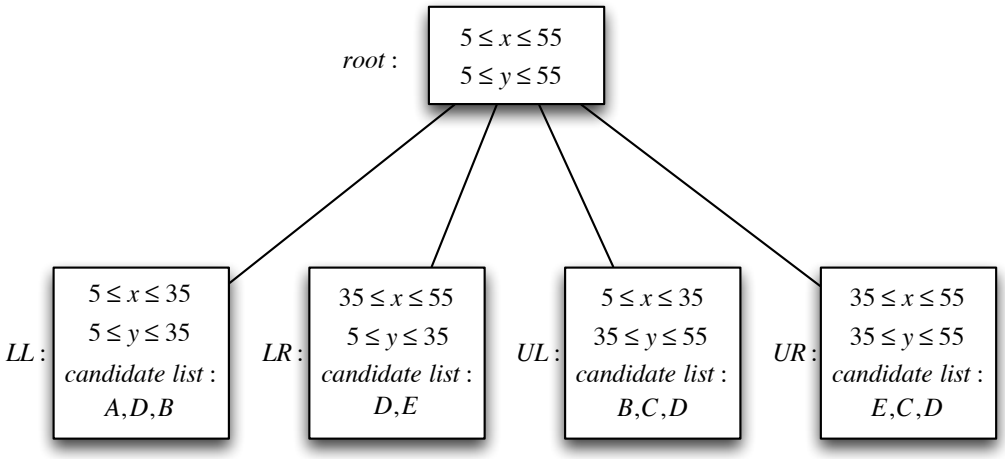

Figure 3.14: The tree that corresponds to the topology of the example

3.7.2 Search of the largest square

Assume we are given the point $(35, 38)$ and we need to calculate the largest empty square with that center.

Step 1

We check if the point is located outside of the critical box. In that case, the point is certainly nearer to the bounding box than to any obstacle and we do not need to examine the tree. The side of the largest square is given as the double of the distance between the point and the boundary. However, in our case, the point is located inside the critical box.

Step 2

We need to traverse the tree, until the leaf node that contains the point is identified. The point is located inside the root node and, thus, we examine all of its children. It is easily observed that $(35, 38)$ belongs to UR rectangle.

Step 3

The distance between the point and the obstacles of the candidate list is computed.

Obstacle E:

$x - distance = max(x_{Emin} - x_{point}, x_{point} - x_{Emax}) = max(40 - 35, 35 - 50) = 5$
$y - distance = max(y_{Emin} - y_{point}, y_{point} - y_{Emax}) = max(46 - 38, 38 - 50) = 8$
$distance(point, E) = max(x - distance, y - distance) = max(5, 8) = 8$

Obstacle C:

$x - distance = max(x_{Cmin} - x_{point}, x_{point} - x_{Cmax}) = max(26 - 35, 35 - 29) = 6$
$y - distance = max(y_{Cmin} - y_{point}, y_{point} - y_{Cmax}) = max(37 - 38, 38 - 45) = -1$
$distance(point, C) = max(x - distance, y - distance) = max(6, -1) = 6$

Obstacle D:

$x - distance = max(x_{Dmin} - x_{point}, x_{point} - x_{Dmax}) = max(26 - 35, 35 - 29) = 6$
$y - distance = max(y_{Dmin} - y_{point}, y_{point} - y_{Dmax}) = max(16 - 38, 38 - 27) = 11$
$distance(point, D) = max(x - distance, y - distance) = max(6, 11) = 11$

Step 4

The side of the largest square is defined by the closest obstacle to the point. In our case, the closest obstacle is C and its distance from the center is 6. Therefore, the largest empty square with that point as center has a side of $2 * 6 = 12$.

3.8 Comments

The example of the previous section, although simple, illustrates the main steps of the algorithm. It is two-dimensional, but the differences with the three-dimensional case are minor. It is obvious that this algorithm offers a method to limit the number of obstacles that need to be examined when searching for the largest empty cube. For example, in our case, we examined only three of the five obstacles. Of course, here the difference is not important, but in cases of thousands of obstacles, there is a major improvement in the running time of the algorithm.

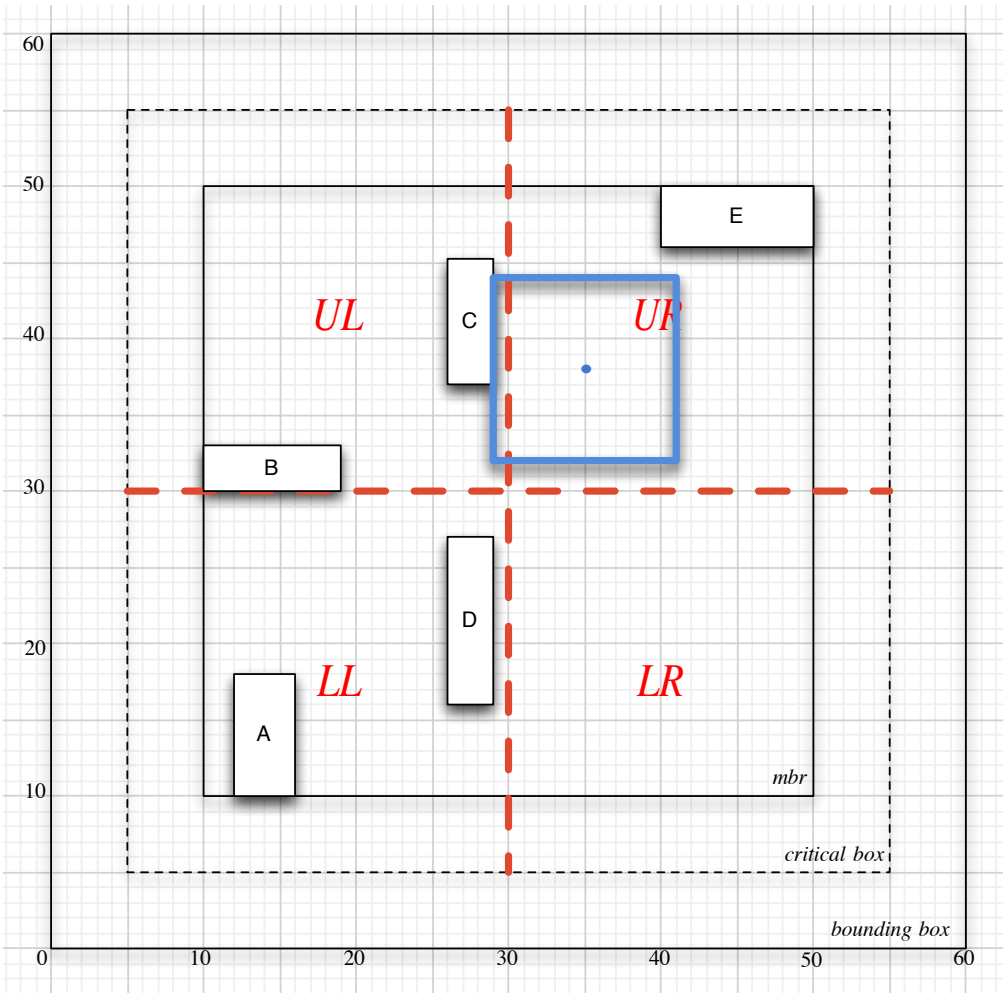

Figure 3.15: The largest empty square centered at $(35, 38)$

Chapter 4

Finding the Largest Empty Cube in Non-Manhattan Geometries

4.1 Non-Manhattan Geometry

In the previous chapter, all geometries were Manhattan, which means that the obstacles were rectangular cuboids with axis parallel edges. This chapter examines non-Manhattan geometries. We can describe this type of obstacles as follows:

Imagine a two-dimensional polygon on a plane parallel to the xy plane, i.e. whose points have constant z value. This polygon, also called *ring*, accompanied by two values called *height* and *thickness* can fully define our obstacles. More specifically, each obstacle consists of the points whose x and y coordinates are inside the polygon and whose z coordinates satisfy the relationship:

$$height \leq z \leq height + thickness$$

The polygons that we examine are always considered convex.

For the purposes of the following analysis, we can consider the simplest case of these polygons, which are rectangles rotated by an angle ϕ around the z axis. We choose this type of polygons in order to offer simplicity to our description and given that it leads to no variation of the main ideas of the algorithm.

4.2 Description of the problem

The problem in non-Manhattan geometries is the following: Given a set of obstacles, like the ones described in the previous section, and a boundary box, the goal is to find the largest empty cube with a given center that can be placed among them. The main difference with Manhattan geometries is that, in this case, the cube is not necessarily axis aligned.

The application that motivated our study, as described in the introductory chapter, requires the calculation of the largest cube that has its top and bottom sides parallel to the xy plane (constant z), property that holds true for all the obstacles, and is rotated by an angle ϕ around the z axis. This angle is the direction of one of our obstacles, or, in the general case where we do not have rectangular obstacles, the direction of an edge of one obstacle. This practically means that we need to limit our search only on those cubes that have the direction of one of the obstacles and ignore the rest. In that way, there is increased chance that the cube and its closest obstacle share a common side, instead of an edge or a vertex. It is true that this cube will not necessarily be the largest one possible, but it is the appropriate one for our application.

4.3 Useful Background

4.3.1 Visible edges

Imagine we have a two-dimensional polygon A and a point P, regarding to which we need to find all of A's visible edges. We will examine each edge separately. For each edge, we call $V_1(x_1, y_1, z_1)$ the first vertex, according to a clockwise traversal of the polygon, and $V_2(x_2, y_2, z_2)$ the second vertex with $z_1 = z_2$. We call (x, y, z) the coordinates of point P.

Assume $\vec{M}$ is the vector that starts from a point of the edge and ends at the observation point P. We can choose, without loss of generality, the start of $\vec{M}$ to be the vertex V_1.

Therefore, we can easily compute $\vec{M}$:

$$\vec{M} = (x - x_1, y - y_1, z - z_1)$$

We define $\vec{N}$ as the vector that is perpendicular to the edge and lies on a plane

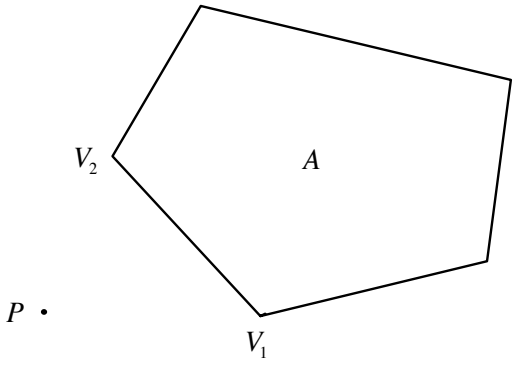

Figure 4.1: Polygon A and point P

parallel to the xy plane. Therefore, it is obvious that $\vec{N}$ can be computed as the cross product of the unit vector $\hat{e_z} = (0, 0, 1)^T$ and $V_1\vec{V}_2$.

$$\vec{N} = \hat{e_z} \times V_1\vec{V}_2 = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ 0 & 0 & 1 \\ x_2 - x_1 & y_2 - y_1 & 0 \end{vmatrix} = (-y_2 + y_1, x_2 - x_1, 0)^T$$

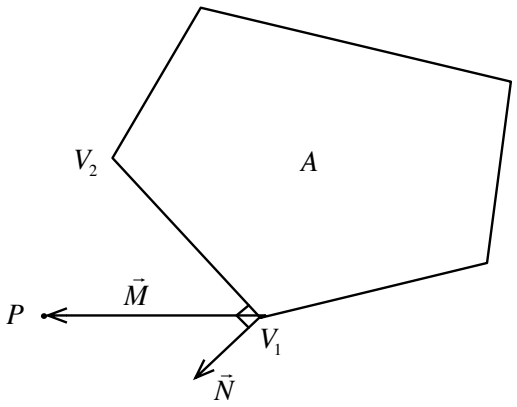

Figure 4.2: Vectors $\vec{M}$ and $\vec{N}$

In order to examine if the edge $V_1\vec{V}_2$ is visible from P, we need to calculate the dot product of vectors $\vec{M}$ and $\vec{N}$.

$$\begin{aligned} \vec{M} \cdot \vec{N} &= (x - x_1)(-y_2 + y_1) + (y - y_1)(x_2 - x_1) \\ &= -x(y_2 - y_1) + y(x_2 - x_1) + x_1y_2 - x_2y_1 \end{aligned}$$

According to [22], an edge V_1V_2 is visible from a point P if and only if the above dot product is positive: $\vec{M} \cdot \vec{N} > 0$.

4.3.2 Rotation around a point

A transformation that will be useful in the following study is the rotation of a vertex $V(v_x, v_y)$ around a point $P(p_x, p_y)$ by an angle ϕ. Let's call this point *origin*. This rotation is composed by three simpler transformations:

Translation of the vertex V to the origin. The resulting point $V'(v'_x, v'_y)$ is the following:

$$v'_x = v_x - p_x$$
$$v'_y = v_y - p_y$$

Rotation of the point V' around the origin by an angle ϕ. We called $V''(v''_x, v''_y)$ the point that results from the rotation of V' around P.

$$v''_x = v'_x \cdot \cos\phi - v'_y \cdot \sin\phi$$
$$v''_y = v'_x \cdot \sin\phi + v'_y \cdot \cos\phi$$

Translation of V'' back to its original position. Now, we need to translate V'' back to its original position. We call V_R the rotated vertex V around P.

$$v_{R_x} = v''_x + p_x$$
$$v_{R_y} = v''_y + p_y$$

In the case of a three dimensional point $V(v_x, v_y, v_z)$, if we keep unchanged the third coordinate v_z and rotate the point $V_{proj}(v_x, x_y)$ around the origin $(0, 0)$, the resulting point $V_R(v_{R_x}, v_{R_y}, v_z)$ is equal to the rotation of point V around the z axis.

4.3.3 Largest axis aligned empty square adjoining a line segment

Assume we have a point P and a line segment AB on the xy plane and we need to find the largest axis aligned square that does not contain any point of AB. Of course, the square contacts AB. In the opposite case, it could be enlarged until it contacted AB and, therefore, would not be the largest. AB can have any direction. An example is given in the following figure.

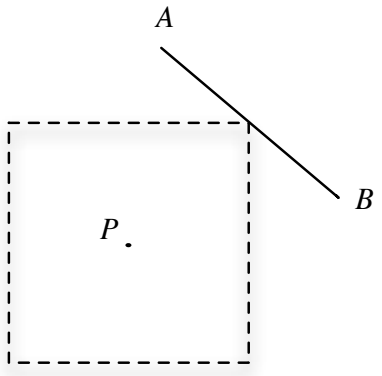

Figure 4.3: Example of a line segment AB, a point P and the largest axis aligned square

Depending on the relative position of P and AB, the possible cases are the following: Either one vertex of the square contacts AB, or a vertex of AB contacts an edge of the square. Of course, AB may share more than one point with an edge of the square.

In order to deal with the first case illustrated in the Figure 4.4, we need to examine if there is an intersection of AB and the lines with slope 1 and -1, which are the diagonals of the square.

We call (p_x, p_y) the coordinates of P, (a_x, a_y) the coordinates of A and (b_x, b_y) the coordinates of B.

The equation that describes AB can be written in a parametric form as follows:

$$x = a_x + k(b_x - a_x) \tag{4.1}$$

$$y = a_y + k(b_y - a_y) \tag{4.2}$$

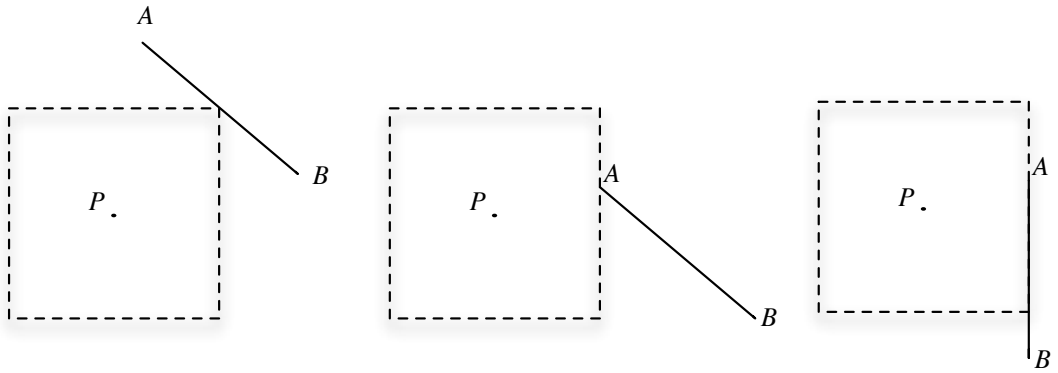

Figure 4.4: Three possible topologies of the line segment AB and the point P and the corresponding largest squares

If $0 \leq k \leq 1$, (x, y) is located on the line segment AB. If $k < 0$, the point belongs to the extension of the line segment from point A, else if $k > 1$, it belongs to the extension from B.

The line that has slope 1 (l_1) and contains the point $P(p_x, p_y)$ is described by the following parametric equation:

$$x = p_x + t \tag{4.3}$$

$$y = p_y + t \tag{4.4}$$

Similarly, the line with slope -1 (l_2) that contains the point $P(p_x, p_y)$ is described by the following parametric equation:

$$x = p_x + t \tag{4.5}$$

$$y = p_y - t \tag{4.6}$$

The next step is to compute the intersection points of AB with lines l_1 and l_2, if there are any.

Intersection of AB and l_1. When inserting the x value from (4.3) in (4.1) and the y value from (4.4) in (4.2), we get:

$$p_x + t = a_x + k(b_x - a_x) \tag{4.7}$$

$$p_y + t = a_y + k(b_y - a_y) \tag{4.8}$$

(4.8) − (4.7):

$$p_y - p_x = a_y + k(b_y - a_y) - a_x - k(b_x - a_x) \Rightarrow k = \frac{(p_y - p_x) - (a_y - a_x)}{(b_y - b_x) - (a_y - a_x)} \quad (4.9)$$

given that $(b_y - b_x) - (a_y - a_x) \neq 0$. If $(b_y - b_x) - (a_y - a_x) = 0$ there is no intersection point.

If $0 \leq k \leq 1$, l_1 and AB have an intersection point, which is given from equations (4.1) and (4.2) if we replace the value of k from (4.9). That means that this point could be a possible contact point between the largest empty square and the line segment. If that holds true, the largest empty square will have center at P and side equal to $2 \cdot |x_{intersection} - p_x|$.

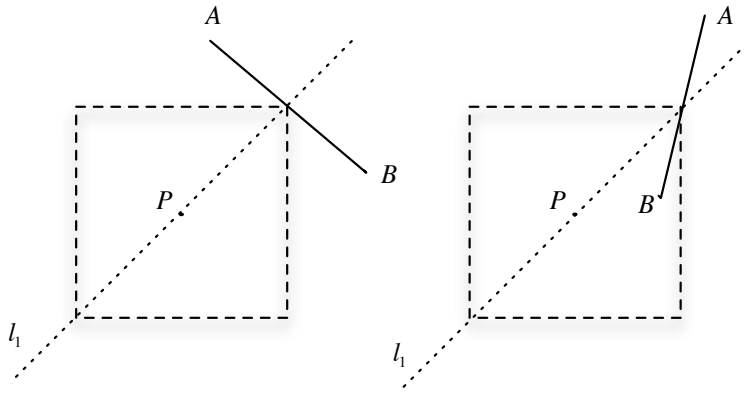

Figure 4.5: In the first case, the square that is defined by the intersection point is indeed the largest possible one. In the second case, the square is not valid, since it contains points of AB in its interior. The correct square is calculated after the examination of the line with slope -1 as well.

If $k < 0$, l_1 intersects the extension of AB from the part of A and the closest point of AB to that intersection point is A (see also the following figure). Therefore, the length of the diagonal with slope 1 of the square is not restricted by the line segment AB. In that case, we examine the restrictions imposed to the square size by point A. A permits a square side less or equal to $2 \cdot max\left(|a_x - p_x|, |a_y - p_y|\right)$.

Similarly, we deal with the cases where $k > 1$. We now examine the restrictions imposed by point B and compute a side limit of $2 \cdot max\left(|b_x - p_x|, |b_y - p_y|\right)$

Intersection of AB and l_2. The intersection point of AB and l_2, if any, is computed similarly to the previous section. We insert the x value from (4.5) in

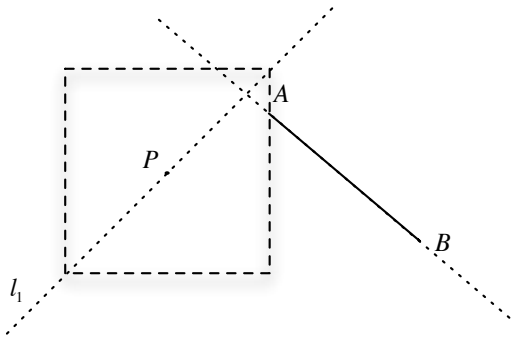

Figure 4.6: Example where the $k < 0$ and the intersection point lies on the extension of AB nearer to A.

(4.1) and the y value from (4.6) in (4.2), we get:

$$p_x + t = a_x + k(b_x - a_x) \tag{4.10}$$

$$p_y - t = a_y + k(b_y - a_y) \tag{4.11}$$

(4.10) + (4.11):

$$p_x + p_y = a_x + k(b_x - a_x) + a_y + k(b_y - a_y) \Rightarrow k = \frac{(p_x + p_y) - (a_x + a_y)}{(b_x + b_y) - (a_x + a_y)} \tag{4.12}$$

given that $(b_x + b_y) - (a_x + a_y) \neq 0$. If $(b_x + b_y) - (a_x + a_y) = 0$ there is no intersection point.

If $0 \leq k \leq 1$, l_2 and AB have an intersection point, which is given from equations (4.1) and (4.2) if we replace the value of k from (4.12). This intersection point limits the side of the square to the value $2 \cdot |x_{intersection} - p_x|$.

If $k < 0$, l_2 intersects the extension of AB from the part of A and the closest point of AB to that intersection point is A. Therefore, the length of the diagonal with slope -1 of the square is not restricted by the line segment AB. However, the square side may be restricted by point A: A permits a square side less or equal to $2 \cdot max\left(|a_x - p_x|, |a_y - p_y|\right)$, otherwise A is located inside the square.

Similarly, we deal with the cases where $k > 1$. We now examine the restrictions imposed by point B and compute a side limit of $2 \cdot max\left(|b_x - p_x|, |b_y - p_y|\right)$

The examination of the intersection of l_1 and AB provides an upper limit on the size of the square. Similarly, the examination of the intersection of l_2 and AB

provides another upper limit. The size of the largest empty square must satisfy both; it is provided, therefore, by the lower value of the two limits.

Although in the previous paragraphs we referred to the cases of the first type of Figure 4.4, we actually covered all possible instances. Indeed, the other cases correspond to values of k less than 0 or greater than 1, where the size of the square is bounded by the endpoints of the line segment.

4.3.4 Axis aligned cube with given center C that contacts an obstacle A

In this section, we will calculate the size of the empty cube that has C as its center and contacts the obstacle A. First of all, we will compute the size of the cube if it contacts the upper or lower side of the obstacle. As we mentioned, each obstacle has a bottom side with constant z value: $z_{min}(A) = height$ and a top side with constant z value: $z_{max}(A) = height + thickness$. Similarly to the Def. 3.3, we compute the value of $z - distance$:

$$z - distance = max(z_{min}(A) - z(C), z(C) - z_{max}(A))$$

This value expresses the distance of the center C of the cube from the obstacle, if the cube adjoins the lower or upper bound of the obstacle. It is obvious that its side will, in this case, equal twice the $z - distance$.

We now need to check the size of the cube if we assume that it adjoins any of the remaining sides of the obstacles. In that case, the z dimension does not influence the result and we can, therefore, work in two dimensions in order to simplify the problem. First of all, we take the projection of the obstacle on the xy plane and the two-dimensional point C_{proj} that corresponds to C if we ignore the z coordinate. Then, we search for the square with center C_{proj} that contacts the polygon.

For that purpose, we examine each edge of the obstacle's polygon: Based on the analysis of section 4.3.3, the square with center C_{proj} that adjoins each edge of the obstacle's polygon is calculated. In that way, each edge imposes an upper limit on the size of the empty square, since it already contacts the edge and, therefore, cannot be enlarged anymore. The restrictions of all edges must be satisfied and, thus, the smallest square side is required.

So far, we have computed two values for the size of the requested cube: one value in case it contacts the obstacle on its upper or lower side and one value if it contacts any of its remaining sides. The value that ensures that the cube contacts the obstacle is the largest between the two. Therefore, this is the requested size of the cube.

4.3.5 Smallest axis aligned empty cube with center inside a given cell N that contacts an obstacle A

Given a rectangular axis aligned cuboid (cell) N and an obstacle A outside of N, we want to find the size of the cube that has its center inside N and contacts A. In other words, if we imagine all the largest empty cubes with center inside N, we need to find the smallest one among them. The size of this cube, as will be shown in a following section, will be very useful for the elimination of certain obstacles from the insertion in some tree nodes, which will accelerate the performance of the algorithm.

To begin with, we will compute the size of the cube if it adjoins the upper or lower side of the obstacle. As we mentioned in the previous section, each obstacle has a bottom side with constant z value: $z_{min}(A) = height$ and a top side with constant z value: $z_{max}(A) = height + thickness$. Likewise, the cell has a bottom side $z_{min}(N)$ and a top side $z_{max}(N)$. Similarly to the Def. 3.2, we compute the value of $z - distance$:

$$z - distance = max(z_{min}(A) - z_{max}(N), z_{min}(N) - z_{max}(A))$$

This value expresses the distance of the center of the cube from the obstacle, if the cube contacts the lower or upper bound of the obstacle, and corresponds to a cube size of twice the $z - distance$.

We now need to compute the cube that could adjoin the remaining sides of the obstacle. The following figure illustrates a simple example of a cell N and an obstacle A. These are their projections on the xy plane, since the third dimension is not important for now.

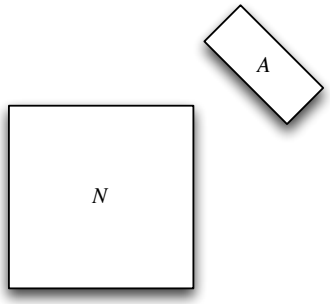

Figure 4.7: Example of a cell N and an obstacle A

The requested square may not be unique. We claim that, in all cases, there will be at least one such square that satisfies one of the following:

a) It will have its center on a vertex of the cell N and will contact a point of an obstacle's edge, or

b) It will have its center on a cell's edge and will contact a vertex of the obstacle A.

Indeed, if both the center and the contact point are internal edge points, we can move toward the vertices of the edges and, if these edges are not parallel, at least one of these moves will lead to a smaller square, since the edges approach each other (See the following figure for an illustration). If the edges are parallel, moving toward the vertices will result in a square of same size, that will also satisfy either (a) or (b). Thus, in all cases, the requested square satisfies one of (a) or (b) sentences.

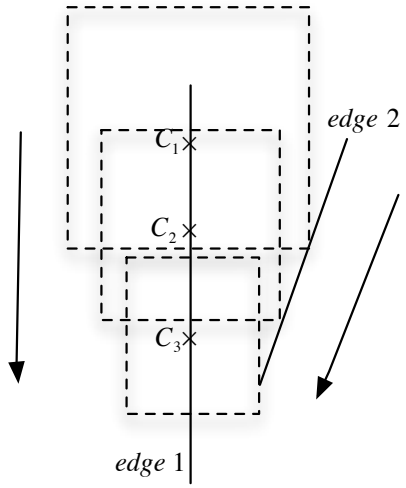

Figure 4.8: The arrows show a move that leads to smaller squares: The square centered at C_1 has both its center and contact point in the internal of the edges and it is larger than the one centered at C_3 whose contact point is an end point.

As a consequence, it suffices to examine only the squares that are defined by all possible pairs edge-vertex, where each pair consists of a cell's edge and an obstacle's vertex or an obstacle's edge and a cell's vertex. For each such pair, we compute the empty square that is centered at the pair's point and contacts the pair's edge (the contact point can be internal or end point). Each pair imposes in that way an upper limit to the square size. Since all restrictions must be satisfied, we keep the smallest value among them, which corresponds to a cube side twice that value.

Finally, we compare the aforementioned value with the $z - distance$. The size

of the axis aligned cube with center C that contacts A is equal to the largest value.

4.3.6 Largest axis aligned empty cube with center inside a given cell N that contacts an obstacle A

In this section, we will compute the maximum cube with center inside a cell N that contacts an obstacle A. This computation of the cube is very similar to the procedure that was analyzed in the previous section for finding the minimum empty cube with center inside N that contacts A.

First of all, we compute the $z - distance = max(z_{min}(A) - z_{max}(N), z_{min}(N) - z_{max}(A))$, which is equal to half the side of the cube, if the cube contacts the obstacle on its upper or lower side. Then, we find the maximum cube if it contacts one of the remaining sides of A.

For reasons similar to the ones analyzed in the previous section, the cube will satisfy one of the following sentences:

a) It will have its center on a vertex of the cell N and will contact a point of an obstacle's edge, or

b) It will have its center on a cell's edge and will contact a vertex of the obstacle A.

Therefore, we construct all edge - vertex pairs like before, but we consider only the edges and the vertices of A that are visible from the cell. The remaining edges and vertices, since they are not visible, can lead to a cube which is not feasible, because it will definitely contain some points of the obstacle in its interior. Moreover, it will likely be larger than the one defined by the visible edges and points and it will alter our result. Thus, we find all the visible edges using the method described in section 4.3.1. Then, we find the visible points. These are the vertices that belong to at least one visible edge.

The following steps are approximately the same as the previous section's: For each pair, the corresponding cube is constructed based on section 4.3.3. The appropriate cube is the largest one among them. Finally, this value is compared with the $z - distance$ and the largest one determines the size of the largest empty cube that contacts A and has center in N.

4.4 Detailed Description of the Algorithm

In the case of non-Manhattan geometries, we create multiple octrees, one for each possible orientation of the cube. The application that motivated this algorithm guarantees that there will be only a few different directions and, thus, the number of octrees will be small. Of course this approach causes a memory overhead. However, given that there are only a few octrees and that this method offers simplicity to the solution and a good performance, it can be neglected.

4.4.1 Creating the octrees

First of all, we need to identify all possible cube orientations. This can be easily realized by examining linearly all the obstacles and finding the direction of each one, since the orientation of the cube can only be one of the directions of the obstacles or the obstacles' edges. Then, for each direction, an octree must be created.

Each octree represents the whole space that contains the obstacles rotated by an angle $-\phi$ around the z axis, where ϕ is the direction that corresponds to the specific octree. Thereby, when the largest axis aligned empty cube is identified in the rotated geometry of an octree, it actually corresponds to a cube of direction ϕ in the original geometry. In that way, we can always perform the searching for the largest axis aligned empty cube, which is simple and efficient, and actually find cubes of any of the possible directions.

The construction of each octree requires a preprocessing of the obstacles. Before their insertion in the root of the tree, i.e. before the first call of the function insert, each obstacle must be rotated by an angle $-\phi$ around the z axis, where ϕ is the angle that corresponds to the current tree.

The critical box is rotated be $-\phi$ as well. Since the root of the octree must always correspond to an axis aligned spatial cell, we cannot use the rotated critical box as the root. One solution is to create its minimum bounding rectangle and consider that to be the tree's root. However, our experiments showed that this approach is not that effective. A different approach with better results is presented in the following figure, where we can see the projection of a rotated critical box on the xy plane. First of all, we compute an axis aligned rectangle that is located totally in the inside of the critical box (see T_1). The extension of its sides creates a space partition. Each rectangle that intersects with the rotated critical box defines a root for an octree. In that way, more octrees are created, but each obstacle belongs to

only a few of them. As a result, there is not important memory overhead, since these multiple octrees are comparatively small.

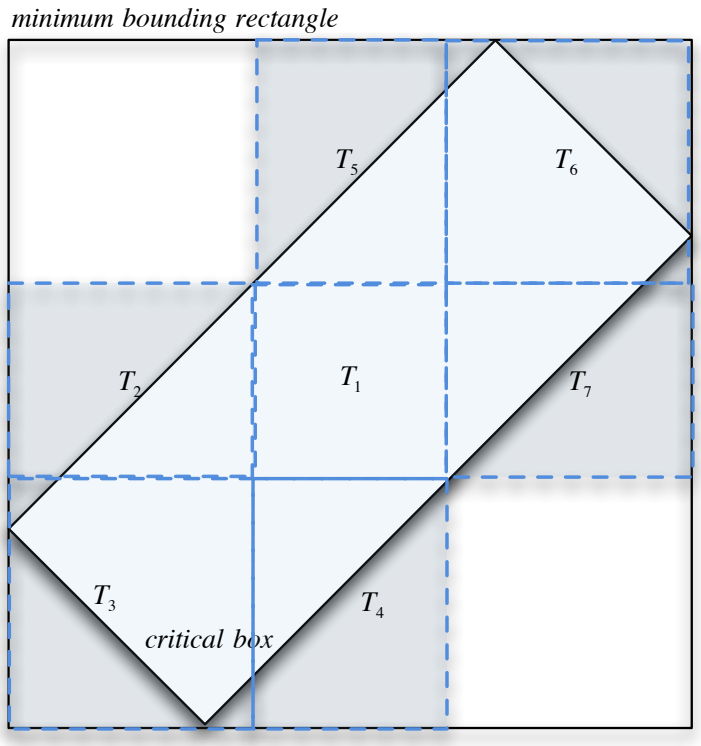

Figure 4.9: The axis aligned spatial cells T_1, T_2, T_3, T_4, T_5, T_6 and T_7 are the roots of the octrees that are created for this rotated critical box.

After that, the creation of each octree presents an important resemblance with the case of the Manhattan geometries. More specifically, the algorithm for the insertion of an obstacle into a tree node (Algorithm 1 of section 3.5) and the algorithm that performs the checking of the cuboids before the insertion (Algorithm 2 of section 3.5) are almost the same. For completeness, we cite them again, after performing some minor changes. On the other hand, the algorithm for the domination checking presents certain differences.

Inserting obstacles into the tree

The only difference of this function with the corresponding one of chapter 3 is that we first compute the minimum bounding box of the obstacle A. We then use the distance between that bounding box and the cell as a first indication whether A

should be examined or not. If that distance is greater than a distance threshold, then the distance of the cell from the actual obstacle A would be at least the same and the obstacle does not belong to this cell, as it is not in its neighborhood.

Algorithm 6 Inserting a cuboid into a tree node

```
1: procedure INSERT(cuboid A, node N)
2:     d ← distance(mbr(A), N)                    ▷ Distance according to Def. 3.3
3:     if d ≥ DistanceThreshold(N) then return
4:     if N is a leaf node then
5:         CHECK(A, N)
6:         if (tree height < HeightThreshold) and (N's candidate list's length
7:             > CandidateListThreshold) then
8:             DIVIDE(N)                          ▷ Divides the cell N into 8 equal octants
9:             for each cuboid a of N's candidate list do
10:                for each child N_i of N do
11:                    INSERT(a, N_i)
12:    else
13:        for each child N_i of N do
14:            INSERT(A, N_i)
15:    return
```

Checking the obstacles

The goal of these checks is to determine if an obstacle A belongs to cell N's candidate list. The difference between the following algorithm and the one of Manhattan geometries (see also section 3.5.1) concerns the update of the variable *DistanceThreshold*. The purpose of this value is the following: If any obstacle A's distance from the cell is greater than *DistanceThreshold*, then the largest empty cube will certainly touch another obstacle before A.

More specifically, the distance of A from the cell N in non-Manhattan geometries actually refers to the side d of the smallest cube that touches A and has its center inside N. That value expresses the least size of the cube that is required in order to approach and contact A.

As far as the computation of *DistanceThreshold* is concerned, let's imagine an obstacle B and call D the size of the largest cube that contacts B with center inside N. D can be computed based on the detailed analysis of section 4.3.6.

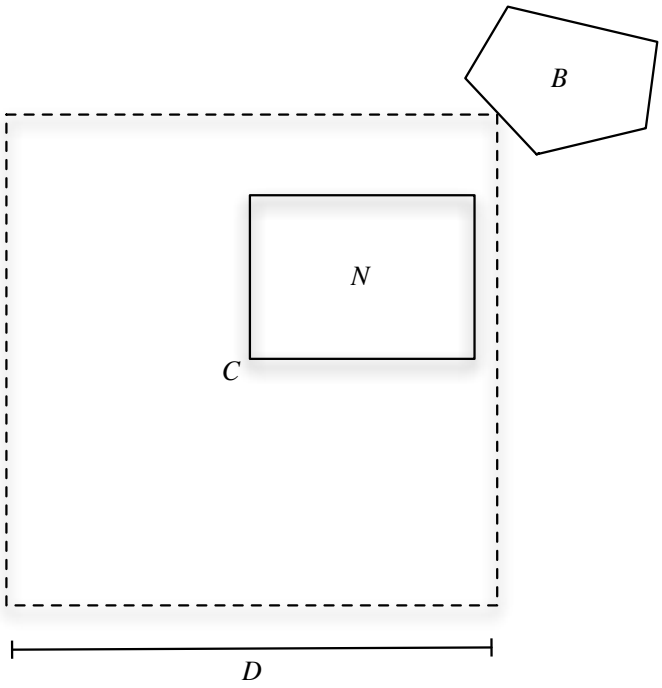

Figure 4.10: Two dimensional projection of cell N, obstacle B and the largest empty cube that contacts B with center C inside N.

In order to ensure that any axis aligned cube contacts B before A, we just need to check if $d > D$. If that inequality holds true, A is definitely outside of any cube that touches B and, therefore, in no case A is the nearest neighbor of a cell's point. As demonstrated in Algorithm 7, $DistanceThreshold$ is updated in order to store the minimum of all possible values D that correspond to the obstacles. Thereby, more obstacles will be rejected after the inequality checking $d > DistanceThreshold$.

In the following algorithm, we prefer to use half the values $DistanceThreshold$ and d that were described in the previous paragraphs, since they express better the notion of the distance between the center of the cube and the contact point. Moreover, that offers better consistency with the algorithm of the previous chapter.

Algorithm 7 Checking the cuboids

```
1: procedure CHECK(cuboid A, node N)
2:     d ← distance(A, N)                          ▷ Half the cube size of section 4.3.5
3:     if d ≥ DistanceThreshold(N) then return
4:     for each cuboid a of N's candidate list do
5:         if DOMINATES(a,A,N) then return
6:         else if DOMINATES(A,a,N) then
7:             Remove a from N's candidate list
8:     Insert A at the end of N's candidate list
9:     D ← maxDistance(A, N)                       ▷ Half the cube size of section 4.3.6
10:    if D < DistanceThreshold(N) then
11:        DistanceThreshold(N) ← D
12:    return
```

Domination Checking

Before the insertion in the candidate list, each obstacle must be checked in order to see if it dominates or is dominated by another obstacle. Assume we want to check if obstacle A dominates obstacle B regarding cell N, i.e. if every cube with center inside N contacts A before B. If B is entirely or partially inside the cell N, then it certainly cannot be dominated by another cuboid.

In the opposite case, we check if each cube contacts A before B. Of course, it is not feasible to examine all possible cubes. Fortunately, it suffices to examine only some specific cases:

Case a**.** If the smallest empty cube that touches B, as calculated from section 4.3.5, contacts the upper or lower side of B:

Similarly to Manhattan geometries, we "inflate" B by $z - distance$. Since in non-Manhattan geometries this is not as easy to calculate as in Manhattan geometries, we follow a different approach. We will only consider the points of B, which inflated by $z - distance$ contact the cell N. That means that we do not need to examine the points of B that are located further than $z - distance$ from the cell. These points form a polygon. We take its vertices and "inflate" each one of them by $z - distance$. The cube that is created for each vertex has a common area with cell N.

Because obstacle A may be not-axis aligned and we cannot "inflate" it that effectively in order to check if the "inflated" A contains the common areas, a different

method is applied. First of all, we notice that if we select any point inside the common area and form a cube of side twice the distance of the obstacle B, it will definitely contact B. In order to be sure that B is dominated by A, we must check all these possible cubes. Fortunately, it is enough to check the cubes with center the vertices of the common areas, since these are the extreme cases and if this holds true for them, it will also hold true for all their remaining points.

As a result, we consider the vertices of each common area and, for each one, we create a cube, with center that vertex, that adjoins A, according to section 4.3.4. If the side of that cube is smaller than twice the $z - distance$, then A dominates B regarding that point. If this holds true regarding all these extreme points, then it will also hold true for all points of the common area. Moreover, due to convexity and the nature of the obstacles, B is dominated by A regarding all the remaining points of cell N as well.

Case *b***.** If the smallest empty cube that touches B, as calculated from section 4.3.5, contacts any of the remaining sides of B: In that case, we only need to examine the cubes whose two dimensional projection on the xy plane is a square that belongs to one of the following categories:

b.1) It has its center C on a vertex of the 2D projection of the cell and their contact point within a visible from C edge of the projection of B.

In that case, we work on the cell's edge that contains C. We need to examine all the points of the edge, whose z coordinates are between B's $height - distance$ and $height + thickness + distance$, where $distance$ equals half the side of the cube that adjoins B. Of course, we only examine the points that are located on the edge of the cell and not the ones on its extension. Then, we take the two points with the minimum and the maximum z coordinates. These are the extreme points. For each point, we create a cube that adjoins the obstacle A, according to 4.3.4. If the size of both cubes is less than the size of the corresponding cube that is required in order to reach B, i.e. $2 \cdot distance$, then A dominates B regarding those points. Due to convexity and to the nature of obstacles, it will also dominate it regarding the remaining points of the edge that we had to examine. If for one of these two points, the cube that adjoins B is smaller than the one that adjoins A, then B is not dominated by A.

b.2) It has its center C on the projection of a cell's edge and its contact point with the projection of B is one of its visible from C vertices.

That vertex of B's 2D projection corresponds to a vertical edge of B in the three dimensions. Similarly to Manhattan geometries, we "inflate" that edge of B by

distance along the three axes in order to create an axis-aligned cuboid, where $2 \cdot distance$ is the least side of the cube that is required to reach that edge. Then, we compute the common area of this inflated edge and the surface of the cell.

Afterwards, we take the vertices of this common area and for each one of them we compute the cube that is required to reach A considering this vertex as its center. This cube can be calculated according to section 4.3.4. If all these cubes have a side less than $2 \cdot distance$, which means they reach A before B, B is dominated by A regarding these points. In the opposite case, if for one of these points the cube that corresponds to A is larger than $2 \cdot distance$, B cannot be dominated by A. If B is dominated by A regarding all these extreme points, due to the convexity and the nature of the obstacles, it will also be dominated regarding the remaining points of the common area.

To sum up, if A dominates B regarding these points (the vertices of the common areas), it will also dominate it for all points inside the common areas, since these are the extreme cases. Furthermore, it will dominate it regarding all the remaining points of the cell due to convexity of the obstacles. Indeed, if A is closer than B to all the vertices we examined then, due to the nature of the obstacles, it will be closer to the remaining points as well.

Algorithm 8 Checking if A dominates B

```
1: function DOMINATES(cuboid A, cuboid B, cell N)
2:     d ← distance(B, N)                          ▷ Half the size of the cube of 4.3.5
3:     if d < 0 then return false    ▷ B is internal to N, cannot be dominated
4:     if d equals z − distance then                                        ▷ Case a
5:         Find the points of B, which inflated by d contact N
6:         These points form a polygon, take its vertices
7:         Inflate each vertex by d
8:         Find the common areas of the inflated vertices and cell N
9:         for each common area do
10:            Take all vertices
11:            for each vertex do
12:                Find the cube with center the vertex that reaches A      ▷ 4.3.4
13:                if the cube's side is larger than 2d then
14:                    return false
15:        return true
16:    else          ▷ Case b. We examine the 2D projections on the xy plane
17:        Find all visible edges of B                          ▷ According to 4.3.1
18:        Find all visible vertices of B
19:        for all pairs visible edge(B)-vertex(N) do                       ▷ Case b.1
20:            Find the vertical edge of N in 3 dimensions that contains the vertex.
21:            Create the line segment, part of the edge, with z > height − d and
   z < height + thickness + d, where height and thickness are B's parameters
22:            Take the points with the minimum and maximum z coordinates
23:            Create the cubes that adjoin A with these points as centers
24:            if a cube's side is larger than 2d then
25:                return false                              ▷ A does not dominate B
26:        for all pairs visible point(B)-edge(N) do                        ▷ Case b.2
27:            Find the vertical edge of B that contains the point
28:            Inflate the edge by d along the three axes
29:            Find the common area of the inflated edge and cell N
30:            for each vertex of the common area do
31:                Find cube with that vertex as center that adjoins A ▷ See 4.3.4
32:                if the side of the cube is larger than 2d then
33:                    return false                          ▷ A does not dominate B
34:        return true
```

4.4.2 Executing queries

The search for the largest empty cube with a given point C as a center requires the examination of one octree per possible orientation. First of all, for each orientation, we find the tree whose root contains the query point. The, for each tree, a traversal is performed until the leaf node that contains the point is identified.

Algorithm 9 Tree traversal

```
1: function TRAVERSE(node N, point P)
2:     while N is not a leaf node do
3:         for each child N_i of N do
4:             if P belongs in N_i then
5:                 N ← N_i
6:     return N
```

Then, we search its candidate list to find the nearest obstacle. More specifically, we traverse the candidate list and compute the largest empty cube with center C that adjoins each obstacle, based on section 4.3.4. That cube is the largest empty cube of direction ϕ in the original geometry, where ϕ is the corresponding angle of the specific octree. In that way, we find the largest empty cube for each possible direction ϕ. The largest one among them is the requested cube with center C.

Algorithm 10 Searching the candidate list (sorted)

```
1: function SEARCH(node N, point P)
2:     DistanceToNearestObstacle ← ∞
3:     for each cuboid a of N's candidate list do
4:         DistanceToObstacle ← distance(a, P)    ▷ Distance according to 4.3.4
5:         if DistanceToObstacle < DistanceToNearestObstacle then
6:             DistanceToNearestObstacle ← DistanceToObstacle
7:         if a outside of N then
8:             if distance(a, N)+d_query_point > DistanceToNearestObstacle then
9:                 break                          ▷ Distance according to 4.3.5
10:    return 2 · DistanceToNearestObstacle                          ▷ Cube size
```

4.5 Techniques for better query performance

The same techniques that were described in the case of Manhattan geometries apply in non-Manhattan geometries as well. More specifically, a sorting of the obstacles of each candidate list can be performed based on the least cube size with center inside the cell that is required in order to make contact with the obstacle. As a result, the traversal of the candidate list can terminate earlier if the remaining obstacles are located far from the query point. Moreover, the use of pointers for quick access of the neighboring cells of each node and the use of the critical box that can avoid the whole tree traversal for some query points offer a performance acceleration. More details for these methods can be found in chapter 3.6.

4.6 Maximum difference between the produced cube and the largest cube of any direction

The empty cube that is created by this algorithm has the orientation of one of the obstacles' edge. This restriction was imposed by the application that motivated our study. However, from a theoretical point of view, this cube may not actually be the largest one possible. We will provide a short analysis that demonstrates the greatest possible deviation between the cube that is produced by our algorithm and the actually largest cube of any direction.

Let's work in two dimensions in order to have a more comprehensive view via figures. First of all, imagine a geometry of rectangular obstacles in two dimensions that are all axis aligned. This is the case of Manhattan geometry that was analyzed in chapter 3. However, now we will not limit the largest empty square to be axis aligned, but, instead, it can have any direction.

In the following figure, we present the example of one obstacle and a query point. If we construct an axis aligned square, then its contact point with the obstacle is one of its vertices and the side of the square will be $d\sqrt{2}$.

In figure 4.12, we can see the same obstacle and query point as before, but now we can create a square of any direction. Let's connect the query point with the closest vertex of the obstacle and call the length of this line segment d. It is obvious that d somehow determines the size of the square. Any square, regardless its direction, cannot have a side greater than $2d$. If that were true, the square would certainly contain vertex V in its interior. Thus, the largest square is achieved when its side equals $2d$. That practically means that V equals the middle of a square's side, since, in that case, half the square size is d, thus the square has a side of $2d$.

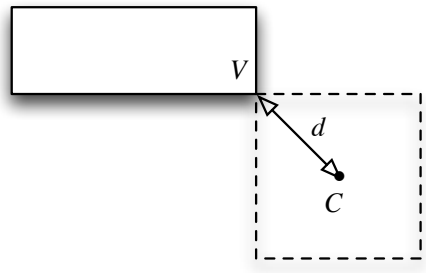

Figure 4.11: Example of an obstacle and the largest axis aligned square

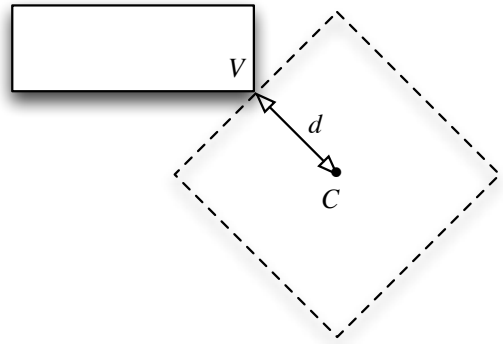

Figure 4.12: Example of an obstacle and the largest square of any direction

It can be proven that if V equaled any other point of the square's side, its size would be smaller. More specifically, we select a random point on the square's size. Its distance from the query point is d. Given that the hypotenuse of a triangle is larger than any of its other sides, we easily realize that this square's half side is smaller than d. Therefore the square's size is less than $2d$ and it is not optimal.

To sum up, the largest square is obtained if the contact point V equals the middle of the square's side. In that case, the side of the square has length $2d$, where d is the distance between V and the query point. By the same logic, the minimum square is obtained when V equals a vertex of the square. Now, half the diagonal equals d, therefore the side of the square is $d\sqrt{2}$. As a result, in the worst case scenario (figure 4.11), i.e. when the deviation between the produced square and the largest possible one is the greatest, the relation between the square of our algorithm and the actual largest square would be $1 : \sqrt{2}$. In terms of squares' area, this difference translates to a square of double area.

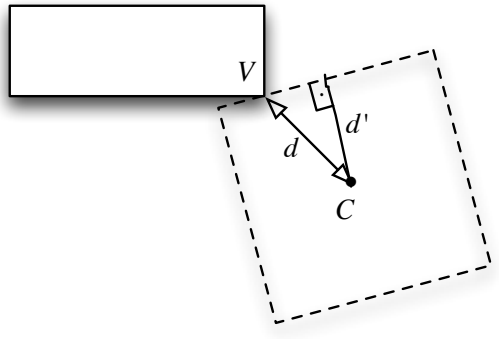

Figure 4.13: $d' < d \Leftrightarrow 2d' < 2d \Leftrightarrow$ the square is not optimal

Similarly, in the three dimensions, given a query point and an obstacle, the cube that adjoins the obstacle is the smallest one if its contact area with the obstacle is only a point and in particular one of the cube's vertices. This case is analogous to figure 4.11 in two dimensions. In that case, if we call the distance between the query point and the contact point d, i.e. the cube diagonal is $2d$, the cube has side of length $2d/\sqrt{3}$, since the diagonal of a cube equals $\sqrt{3}$ times its side. The largest possible cube is the one whose half side equals d (its contact point with the obstacle is the middle point of one of its sides), and, thus, its side is $2d$. The relation between the worst case cube size and the largest one is $1 : \sqrt{3}$.

The above analysis applies when the obstacle is a polygon with edges of any direction, as well. In this case, again the largest square could have up to $\sqrt{2}$ times the side length of our algorithm's square and the largest cube up to $\sqrt{3}$ times.

If there are more obstacles in the space, the described scenario is still the one that produces the largest difference between the two results. However, in this case, the largest possible cube that is permitted by one obstacle may be restricted by another, and, thus, the difference between the largest cube of any direction and the actually constructed one may be smaller.

Chapter 5

Practical Implementation and Experimental Results

5.1 Implementation

The language that was used for the implementation of the algorithm is C++.

5.1.1 Data Structures

The main data structures of our program concern the description of the obstacles and the tree nodes. The data structure for the obstacles is the following:

Obstacles

```
struct Obstacle {
    Ring polygon;
    double distance_to_cell;
    Id id;
    Box<Point_3> mbr;
    double direction;
};
```

Details

polygon: Ring is a data structure that is used to describe a polygon. It stores the vertices of the polygon, considering a clockwise traversal.

distance_to_cell: The value distance_to_cell is used to represent the half side of the smallest axis aligned empty cube that can abut this obstacle. For its calculation, the methodology of section 4.3.5 is applied.

id: Each obstacle is characterized by a distinct integer. If we have n obstacles, their ids range from 0 to $n-1$. This number is used to identify which is the nearest obstacle to the query point, i.e. which obstacle the largest empty cube abuts. More specifically, for each query, our program returns a pair with the largest empty cube (size, center, angle) as the first element and the id of the nearest obstacle as the second. Thereby, not only the largest empty cube is calculated, but the obstacle that restricts its size is identified as well.

mbr: This field stores the minimum bounding rectangle for each obstacle. If the obstacle is axis aligned, then its *mbr* equals the obstacle itself. Otherwise, the *mbr* is the smallest axis aligned rectangle that contains the obstacle in its interior. Box<Point_3> represents a three-dimensional axis aligned rectangle. In particular, it stores the minimum values of x, y and z and the maximum values of x, y, z. In that way, the 3D rectangle is fully defined.

direction: This field stores the direction of the obstacle. This variable is valid only if the obstacles are rectangular cuboids of any direction. In case of polygonal cuboids, the edges are not necessarily parallel and each one should be examined separately.

Nodes of the octree

```
struct Octree_node{
        Box<Point_3> cell;
        double size;
        double distance_threshold;
        unsigned int height;
        std::vector<Octree_node> children;
```

```
        Octree_node * neighbors[3][3][3];
        std::vector<Obstacle> candidate_list;
};
```

Details

cell: This field contains the coordinates of the corresponding spatial cell of the node. Specifically, each node corresponds to an axis aligned rectangular cuboid in the three-dimensional space. If we store its minimum and maximum x, y and z coordinates, then all the information that is needed for its description is known.

size: This field stores the value of the largest side of the spatial cell. It equals the greatest value among its length, width and height. Although the computation of this value is easy, we chose to store it in the data structure, because this value is used very often in the code. In that way, the code is simpler and redundant calculations are avoided.

distance_threshold: This value is used during the decision making whether an obstacle belongs to this cell's candidate list or not. If the smallest axis aligned cube with center inside the cell and which adjoins the obstacle has a half side that is larger than this value, then it definitely cannot belong to the cell's candidate list, since this value guarantees that there is an obstacle closer than that.

height: This value expresses the height of the node in the octree. The root corresponds to height 0, its children to height 1 and so on. In order to avoid a very tall tree, which slows both its creation and the execution of the queries, we selected to impose a limit on the height of the tree. This limit depends on the input size: There is a higher limit for larger inputs and the other way around. When the tree reaches that height, the nodes of that height are no more divided into subcells, independently of the candidate list length.

children: Each node has eight pointers, one for each of its child nodes.

neighbors: If we select to use the neighbors network in order to move faster from point to point, this structure is necessary. The array contains pointers to the cells that share a side, an edge or a vertex with the current cell and are of the same size. If a cell of the same size does not exist then the next larger one is used.

The pointer $neighbors[1][1][1]$ corresponds to the tree node itself, while the rest to its neighbors.

candidate_list: This vector contains the obstacles that belong to the node. The elements of this vector are all the candidate nearest obstacles of any query point that is located inside the spatial cell.

5.1.2 Libraries

Our code makes use of the library boost.geometry for the representation of some geometric elements and in order to perform certain actions, such as check whether two polygons intersect, etc.

5.2 Experiments

5.2.1 Experimental Enviroment

The experiments are executed on a Pentium(R) 2.93GHz Dual-Core processor, with 4GB RAM. The operating system is CentOS (6.4), equipped with GNU C++ 4.7.2 compiler. The compilation uses optimization level 3. All of our programs are single threaded, however, the construction of multiple octrees in the case of non-Manhattan geometries facilitates a possible multithreaded execution.

5.2.2 Input

Our program will be executed with inputs of two categories. The first one contains only obstacles of Manhattan geometry. The second one contains obstacles of two directions: axis aligned and rotated by 45° around the z axis rectangular cuboids. All cuboids are of random dimensions and randomly distributed in space. Each of these categories includes many input sets, each of which has a different number of obstacles. The experiments examine the performance of the algorithm regarding the execution time and the memory usage of the program.

5.2.3 Measurements

This chapter presents the performance of our program. Specifically, we study separately the two main parts of the algorithm: the creation of the tree and the execution of the queries. As far as the creation of the tree is concerned, for each input category, we have created input sets of various sizes. The smallest ones have less than 100 obstacles, while the largest have hundred thousands or more than a million obstacles. Our measurements include the required time for the execution of the program as well as some representative examples of the memory usage.

Then, we study the time needed for the execution of the queries. We perform queries with random query points. The first experiment includes executing the same number of queries on input sets of different sizes, in order to study the variations depending on the input size. Afterwards, we select one input set and perform different numbers of queries. Thereby, we can observe the scaling of the execution time, when the number of queries increases.

Finally, we study the influence of the parameter $CandidateListThreshold$, which is the limit of the length of each tree node's candidate list, on the creation of the tree and the execution of the queries. More specifically, we execute the program with a specific input set and a specific number of queries. We select various values of the parameter and for each one we measure the execution time that is needed for the creation of the tree and the search of the largest empty cube for each query point and, then, we compare the results.

5.3 Results

5.3.1 Time performance

Creation the octree

First of all, we examine the time performance of the algorithm regarding the construction of the octree. Table 5.1 presents the results for Manhattan geometries for various input sizes. The $CandidateListThreshold$ equals 100 in all cases.

Table 5.1: Time Performance (Manhattan Geometries)

Number of Obstacles	38	501	2449	16663	99936	665509	1251627
Execution Time (sec)	0.003	0.139	1.117	11.077	96.162	869.301	2205.090

Remarks. We notice that the time needed for the construction of the octree increases with the size of the input, as expected.

The second experiment examines the time performance in non-Manhattan geometries. We have rectangular obstacles with random size and one of the two possible directions: axis aligned or rotated by 45° around the z axis. The value of the variable $CandidateListThreshold$ equals 100.

Table 5.2: Time Performance (Non-Manhattan Geometries)

Number of Obstacles	30	522	1401	7557	16779	100241	333688
Execution Time (sec)	0.005	0.350	1.179	11.102	29.272	499.07	4197.72

Remarks. In the case of non-Manhattan geometries, we observe that more time is needed for the construction of a tree with almost the same number of obstacles. This is mainly due to the fact that we need to construct multiple octrees, in order to cover each existing direction. Moreover, if the obstacles are not axis aligned, more computations are generally required. For example, a non-axis aligned obstacle has in general more visible edges if we observe it from the interior of a spatial cell than the axis-aligned.

Execution of queries

The following tables represent the time performance of the algorithm when numerous queries are executed. More specifically, as far as the first two tables are concerned, we used the input sets from the tables 5.1 and 5.2, after creating the corresponding octree, we executed the same number of queries for all inputs. The number of queries we selected is 10 million in order to be able to observe any performance differences in details.

Table 5.3: Queries Execution (Manhattan Geometries) - 10 million queries

Number of Obstacles	38	501	2449	16663	99936	665509	1251627
Execution Time (sec)	18.777	14.363	15.871	22.190	21.122	33.599	51.141

Remarks. We notice that there is not important difference between the execution times for different input sets. Generally, the execution time tends to increase as the input size increases. This behavior is due to the bigger height of the tree,

Table 5.4: Queries Execution (Non-Manhattan Geometries) - 10 million queries

Number of Obstacles	30	522	1401	7557	16779	100241	333688
Execution Time (sec)	24.707	25.171	30.174	34.464	42.112	43.881	58.747

which may require longer traversal. Moreover, in geometries with many obstacles, for example in the case of 1251627 obstacles, there is larger probability their space density to be higher. Thus, more obstacles correspond to each octree node and the search of the largest cube requires the traversal of a longer candidate list.

We will also study the performance of our program when different numbers of queries are executed. We will use the input set of 16663 obstacles in the case of Manhattan geometries and we will perform various queries and observe the results. Then, we will perform the same study in non-Manhattan geometries using the input set of 16779 obstacles, almost the same as in Manhattan geometry.

Table 5.5: Execution of queries

Number of queries	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Manhattan (sec)	0.001	0.015	0.146	1.998	21.312	215.539	2662.320
Non-Manhattan (sec)	0.002	0.035	0.422	4.252	42.134	412.740	4348.341

Remarks. In the above table, we can observe the expected results. First of all, we notice that the execution of queries in non-Manhattan geometries requires more time than the execution of the same number of queries in Manhattan geometries. This result is anticipated, since in non-Manhattan geometries, multiple trees are created and in order to find the largest empty cube, a traversal of one tree per angle must be realized. Therefore, here, we have two possible orientations and, thus, two trees are examined for each query. We can see that this leads to double execution time compared to Manhattan geometries, where the tree is unique.

Furthermore, we notice that the relation between different numbers of queries and the execution times for the same geometry is almost linear. For example, 10 million queries of the first input set need 21.312 sec, while ten times this number (100 million queries) require almost ten times this execution time (215.539 sec).

Influence of the parameter *CandidateListThreshold*

An important parameter of the algorithm is the limit of the length of each cell's candidate list. We experiment with various values of this parameter and present the results in the following table for both the creation of the tree and the execution of the queries. In all executions, we use the same input and, more specifically, the Manhattan geometry of 16663 obstacles. The number of queries that we execute equals 10 million. Finally, we impose a limit on the height of the tree, which, for this input size, equals 4, since it is enough for an efficient storage of the obstacles in most cases.

Table 5.6: Influence of the parameter *CandidateListThreshold*

Parameter Value	5	20	50	100	200	500	1000
Creation of tree (s)	63.900	14.592	12.1213	11.105	12.076	15.033	20.014
Queries execution (s)	15.904	16.393	17.066	21.980	26.084	61.847	118.448

Remarks. First of all, let's observe the time needed for the execution of the queries. As expected, for the same number of queries, the less the length of the candidate list, the faster the queries execution. This is easy to understand. Although smaller candidate lists lead to taller trees, the traversal of each candidate list is realized significantly faster and, therefore, much better results are obtained.

As far as the time that is required for the construction of the octree is concerned, the values may not seem reasonable when examined for the first time. However, the following remarks help understand their behavior: When the length of the candidates' list is small, the nodes of the tree divide many times in order to achieve the desired candidates list's length or to reach the imposed height limit of the tree. So, it is reasonable to notice a decreasing time when starting from small values of *CandidateListThreshold* and increasing them. However, after the parameter gets larger than a value, the time that is needed for each node division is that long that causes a delay of the construction of the tree. In particular, each division leads to reinserting all the members of the candidate list to each of the node's children. When the candidate list has length 500 or 1000, it is obvious that this procedure causes an important time delay. This also holds true for the insertion of any new obstacle, since a domination checking must be performed for each existing element of the candidate list.

5.3.2 Memory Usage

Our final experiment studies the memory usage of the program. The following table presents the memory usage in the case of Manhattan obstacles for various input sizes. The memory usage is expressed in MB.

Table 5.7: Memory Usage

Number of Obstacles	38	501	2449	16663	99936	665509	1251627
Memory Usage (MB)	0.048	1.599	9.708	65.79	374.8	981.2	1549.5

Remarks. We observe that as the input size increases the program needs more memory during its execution. This memory is used for the representation of the octree, which has more nodes as the number of obstacles gets larger. In all the above cases, we consider the value of $CandidateListThreshold$ to be 100.

Bibliography

[1] Wenjian Yu and Zeyi Wang, *Capacitance extraction*, in Encyclopedia of RF and Microwave Engineering , K. Chang [Eds.] , John Wiley and Sons Inc., 2005

[2] Chao Zhang, Wenjian Yu, *Efficient Space Management Techniques for Large-Scale Interconnect Capacitance Extraction With Floating Random Walks*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 10, Oct. 2013

[3] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars, *Computational Geometry*, Springer, Third Edition, 2008.

[4] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, Yannis Theodoridis, *R-Trees: Theory and Applications*, Springer, 2006.

[5] Antonin Guttman, *R-Trees: A Dynamic Index Structure for Spatial Searching*, Proceeding SIGMOD '84 Proceedings of the 1984 ACM SIGMOD international conference on Management of data, Pages 47-57, 1984

[6] Alok Aggarwal, Subhash Suri, *Fast Algorithms for Computing the Largest Empty Rectangle*, Proceeding SCG '87 Proceedings of the third annual symposium on Computational geometry, Pages 278-290, ACM, 1987

[7] Michael Ian Shamos, Dan Hoey, *Closest-point problems*, Proceeding SFCS '75 Proceedings of the 16th Annual Symposium on Foundations of Computer Science Pages 151-162, IEEE Computer Society Washington, DC, USA

[8] A. Naamad, D.T. Lee, W.L. Hsu, *On the Maximum Empty Rectangle Problem*, Discrete Applied Mathematics 8, Pages 267-277, North-Holland, 1984

[9] B.Chazelle, R. L. Drysdale, D. T. Lee, *Computing the Largest Empty Rectangle*, Journal SIAM Journal on Computing archive Volume 15 Issue 1, Pages 300 - 315, Feb. 1986

[10] Liang-Ping Ku, Bing Liu, Wynne Hsu *Discovering Large Empty Maximal Hyper-Rectangle in Multi-Dimensional Space*, National University of Singapore, Department of Information Systems and Computer Science, 1997

[11] Manish Handa, Ranga Vemuri, *A Fast Algorithm for Finding Maximal Empty Rectangles for Dynamic FPGA Placement*, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, IEEE, 2004

[12] Haim Kaplan, Micha Sharir, *Finding the Maximal Empty Rectangle Containing a Query Point*, June 2011, http://arxiv.org/abs/1106.3628

[13] Adrian Dumitrescu, Minghui Jiang, *On the Largest Empty Axis-Parallel Box Amidst n Points*, Nov. 2009, http://arxiv.org/abs/0909.3127

[14] N. S. Altman, *An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression*, The American Statistician 46 (3), Pages 175P185, 1992

[15] J. Beirlant, E. J. Dudewicz, L. Gyorfi, E. C. van der Meulen *Nonparametric entropy estimation: An overview*, In International Journal of Mathematical and Statistical Sciences, Volume 6, Pages 17P 39, 1997

[16] Hanan Samet, *Spatial Data Structures*, Modern Database Systems: The Object Model, Interoperability, and Beyond, W. Kim, ed., Addison Wesley / ACM Press, Reading, MA, Pages 361-385, 1995

[17] Hanan Samet, *The Quadtree and Related Hierarchical Data Structures*, ACM, ComputingSurveys, Vol.16, No.2, June 1984

[18] Zhigang Xiang, *Color Image Quantization by Minimizing the Maximum Intercluster Distance*, ACM Transactions on Graphics, Vol. 16, No. 3, Pages 260P276, July 1997

[19] Sunil Arya, Theocharis Malamatos, David M. Mount, *Space-Efficient Approximate Voronoi Diagrams*, ACM STOCU02, Montreal, Quebec, Canada, May 19-21, 2002

[20] Gilberto Gutierrez, Jose R. Parama, *Finding the largest empty rectangle containing only a query point in Large Multidimensional Databases*, Proceeding SSDBM'12 Proceedings of the 24th international conference on Scientific and Statistical Database Management, Pages 316-333, 2012

[21] Xu Ma, Gonzalo R. Arce, *Computational Lithography*, Glenn Boreman, John Wiley and Sons, Inc, 2010

[22] Th. Theocharis, A. Mpem, *Computer Graphics (in Greek)*, Symmetria, Athens, 1999

[23] Shamos, M.I. *Geometric Complexity*, Proceedings of the Seventh ACM Symposium on the Theory of Computing, May, 1975

[24] N. Roussopoulos, S. Kelley and F. Vincent, *Nearest Neighbor Queries*, Proceedings ACM SIGMOD Conference on Management of Data, pp.71-79, San Jose, CA, 1995.