Finding the Loneliest Point

by

Ka Yee Yeung

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 1996

Abstract

We study the following problem which finds application in solving equations. Given a Euclidean domain, there are two operations, *insert* and *isolate*. *Insert* adds points to the domain. *Isolate* returns a point in the domain which is satisfactorily far from the points already inserted in the domain.

The objective is to perform *insert* and *isolate* in constant amortized time per operation, linear space and a constant worst case ratio. Worst case ratio measures how far the answer from *isolate* differs from the best possible answer. We concentrate on the case with a two dimensional domain.

The basic approach is to divide the domain into geometric shapes and build a hierarchy of that shape. We have studied the three regular polygons that tile a given two dimensional domain: squares, hexagons and triangles. It turns out that the worst case ratio is smaller when the domain is divided into hexagons than when the domain is divided into squares, the worst case ratio for squares is in turn smaller than that of the triangles.

We have also explored other techniques to improve the worst case ratio while keeping constant amortized running time per operation and linear space. We have explored overlapping, embedding (or diamonds), and multiple grids. The worst case ratio can be improved further when these techniques are combined.

We have also shown that the square technique can be extended to cubes in a three dimensional domain.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The problem of finding roots of equations in several variables arises in many scientific computations. If nothing is known about the equations, except perhaps that the functions involved are continuous almost everywhere, then one is forced into some sort of iterative method of refining solutions and then looking for other solutions in other regions, presumably far from those explored earlier. Motivated by the goal of including such techniques in the Maple Symbolic Algebra System [1], Gaston Gonnet [5] abstracted the problem of finding new starting points by proposing the operations *insert* and *isolate*. *Insert* adds points which represent evaluations already made into the domain and *isolate* returns a point which is satisfactorily far (details later) from the points already evaluated. He asked how efficiently this data type could be supported. In this thesis, we provide constant time solutions for several forms of the problem, though we concentrate on the two-variable case for the sake of simplicity.

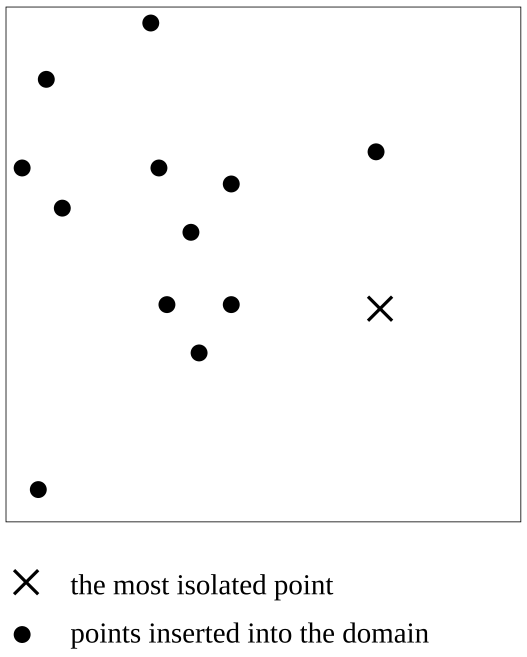
1.2 The Problem In More Detail

More precisely, we define the problem as follows: given a bounded domain (Euclidean space), there are two operations, *insert* and *isolate*. *Insert* adds a point to the domain. *Isolate* returns a point which is satisfactorily far (as defined below) from the points already inserted in the domain, and inserts the returned point. The insertion of the returned point guarantees that multiple *isolate*s return different points. We focus primarily on the two-dimensional case and, without loss of generality, assume the domain is a square. By distance, we mean Euclidean distance.

While the point in the domain maximally distant from any of the inserted points should ideally be found, we will see that this is substantially more difficult than obtaining an approximate solution. In the context of our application, it is satisfactory to return a point guaranteed to be at least a constant factor times the distance from the most isolated point to its nearest neighbour.

If only one point is inserted and it is in the lower right corner of the square domain, then there are two obvious candidates for the most isolated point: the upper left corner of the domain and the middle of the domain. In this thesis, we choose to return the latter as the most isolated point. Effectively, the boundary of the domain is assumed to be occupied. As a consequence, if there are no points in the domain, *isolate* returns the midpoint of the domain. See Figure 1.1. However, our techniques are easily modified to adopt the convention that the boundary is not occupied.

Figure 1.1: Illustration of the most isolated point



1.3 An Exact Solution to the Problem

Finding the most isolated point in a given domain is known as the *largest empty circle* problem in the Computational Geometry literature. Given n points in the plane, the largest empty circle problem is to find the largest circle the centre of which lies in the convex hull of the n given points such that none of the n given points lies in the interior of the circle. Thus, the centre of the largest empty circle is the most isolated point in the given domain.

Shamos [8] outlined an $O(n \log n)$ algorithm for solving this problem using the

Voronoi Diagram. Details of the Voronoi Diagram can be found in any standard Computational Geometry text, such as [7]. Toussaint [10] generalized the problem such that the centre of the largest empty circle does not have to lie in the convex hull of the n given points, but in an arbitrary convex n-gon. He gave an $O(n \log n)$ algorithm for the generalized version of the problem. Toussaint's algorithm also makes use of the Voronoi Diagram of the n given points.

It is known that $\Theta(n \log n)$ is a lower bound (under the usual comparison model) on the time required to solve the largest empty circle problem [9]. In one dimension, the problem reduces to finding the two adjacent points on a line that are farthest apart. This is related to the *element uniqueness* question for which $\Theta(n \log n)$ is a lower bound [4]. Therefore, the algorithms given by Shamos and Toussaint are optimal for the static (n insertions and one *isolate*) version of our problem.

We can find the exact solution of our iterative problem as stated in Section 1.2 using Toussaint's algorithm [10]. In the *insert* operation, points are placed in a queue and the Voronoi Diagram is not updated or constructed, thus an *insert* operation takes constant time. When an *isolate* operation is encountered, points on the queue are used to construct the Voronoi Diagram or are inserted into the existing Voronoi Diagram to find the centre of the largest empty circle. Constructing the Voronoi Diagram takes $\Theta(n \log n)$ time. So, there exists a sequence of n operations ($n - 1$ *insert*s and then an *isolate*) such that the total running time is $\Theta(n \log n)$. Hence, the worst case running time of any sequence of n *insert*s and *isolate*s is $\Omega(n \log n)$, giving an amortized running time of $\Omega(\log n)$ per operation for the exact solution.

1.4 Our Approach

A rough approximate solution was discovered by Norbert Blum, Martin Dietzfelbinger and Ian Munro shortly after the problem was posed by Gonnet. The idea is to divide the domain into squares. A hierarchy of squares is built by grouping four smaller squares into a bigger square. To perform an *insert*, the point is interpolated into the smallest square containing it, and then up to parent squares in the hierarchy. *Isolate* returns the midpoint of one of the largest unoccupied squares in the hierarchy and inserts the returned point. Chapter 2 covers the details.

It is also reasonable to consider dividing the domain into other geometric shapes, and using other tricks like overlapping and multiple hierarchies in the hope of achieving a more efficient method or a better solution. A *hierarchy* of a particular geometric shape is a classification of that shape based on size. A hierarchy consists of *levels* in which the geometric shapes of the same size are grouped into the same level. Levels are indexed, starting with the smallest geometric shape from level 0. Sizes of geometric shapes in consecutive levels differ by a constant factor. Chapter 3 discusses the division of the plane into hexagons, while Chapter 4 considers division of the plane into triangles.

1.5 Model of Computation

In the following discussions, we assume a Random Access Machine with the following constant time operations :

- Arithmetic operations (addition, subtraction, multiplication and division of both integer and reals)
- Bitwise Boolean operations (AND, OR)
- Shift operations by arbitrary numbers of bits
- Comparisons of both integers and real numbers
- Finding the number of zero bits before the first one bit in an integer

One major advantage of assuming a Random Access Machine is that we can do table lookups in constant time.

It will be assumed that the integers stored are of length at most $\log_2 n + c$ bits, where n is the number of points inserted into the domain and c is a constant. The precision of real numbers will not be an important issue.

In the following discussion, the "size" of a geometric shape refers to the length of a side of that shape. For instance, the size of a square is the length of a side of the square.

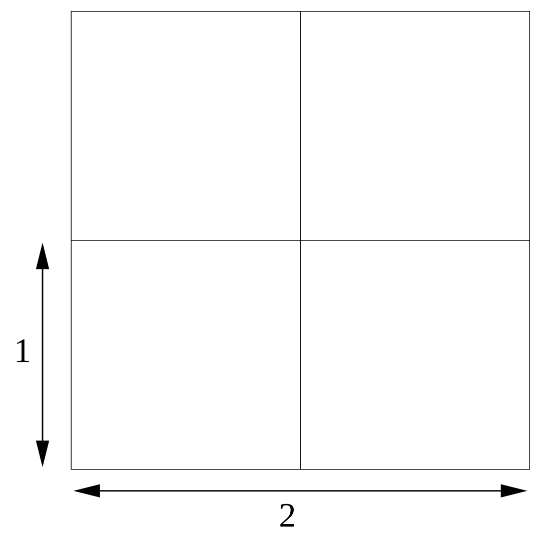
Chapter 2

Dividing the Domain into Squares

The basic idea is to divide the domain into squares and then to build a hierarchy of squares of decreasing sizes. The square hierarchy is formed by decomposing a square into four equal-sized smaller squares in the next lower level in the hierarchy as shown in Figure 2.1. Define the *grid* to be the set of squares making up the bottommost level in the square hierarchy. We keep track of the occupied squares, i.e., those containing at least one inserted point. The isolate operation returns the centre of one of the largest unoccupied squares in the hierarchy and the returned point is then inserted. The absence of any unoccupied squares forces decomposition into smaller squares in the hierarchy. An insertion into a square occupies it, thus removing it from consideration for subsequent isolate operations.

In Section 2.1, we consider a naive approach to the problem which uses linear space but takes $\Theta(\log n)$ amortized running time per operation. In Section 2.2, a more sophisticated method due to Norbert Blum, Martin Dietzfelbinger and Ian Munro is described. It solves the problem in linear space and constant amortized

Figure 2.1: FOUR UNIT-SIZED SQUARES MAKE UP A DOUBLE-SIZED SQUARE



running time per operation, and achieves a constant worst case ratio. Although the naive approach does not guarantee a constant amortized running time per operation, the quality of the solution is the same as that of the refined approach.

2.1 The Naive Approach

2.1.1 Data Structures

Using our strategy, *isolate* returns the midpoint of one of the largest empty squares in the domain. Hence, we have to keep track of whether a particular square is occupied or not. A bit map is the obvious choice: each square in the hierarchy is represented by a bit. The bit is turned on if the corresponding square is occupied, otherwise it is turned off. The bits are organized into a hierarchy representing squares of decreasing sizes. Bits representing squares of the same size are grouped into the same level in the hierarchy. The levels are indexed, with the smallest squares at level 0. A linear scan of the bit vector, starting at the top level in the hierarchy, is performed in *isolate*, and the midpoint of the first unoccupied square encountered, which is represented by a zero bit, is returned. However, if all the squares are occupied, decomposition into smaller squares is required in order to generate empty squares. Therefore, the isolate algorithm consists of a loop in which the bit vector is scanned and the square hierarchy is decomposed one level below the current level until an empty square (a zero bit) is found. Since the first zero bit must be at the lowest level of the current square hierarchy, we need a bit vector representing the squares in the bottommost level of the hierarchy.

Decomposition of the inserted points into a finer grid of squares requires the coordinates of the inserted points be known. Therefore, the coordinates of the inserted points are stored in a linked list or an array. Order is not important in the linked list or the array. Alternatively, we can use an array of linked lists to store the inserted points. The array is indexed by the squares in the bottommost level in the square hierarchy. The list for each array position stores points inserted in the corresponding square. Using an array of linked lists has the advantage of easier interpolation into a finer grid by looking at the trailing bits only (details later).

Moreover, we need a counter to keep track of the number of points already inserted into the domain.

With the above data structures, the insert operation consists of determining the smallest square in which the new point lies (interpolation), inserting the point into the linked list and updating the bit vector if necessary. The case that the inserted point lies on the boundary of the squares will be discussed in Section 2.2.6.

There is an alternative lazy approach to the naive insert and isolate algorithms. In the lazy approach, points are put on a queue in the insert operation. The real work of fixing the bit vector in the insert algorithm is postponed until an isolate operation is encountered. In the isolate algorithm of the lazy approach, all the points on the queue have to be inserted using the insert algorithm before the isolate algorithm is applied. The lazy approach has the advantage that the bit hierarchy is fixed only when an isolate operation is encountered. Any insertions after the last *isolate* need not be performed. The analysis of the running time for the lazy approach is the same.

2.1.2 Naive Insert

- Insert the point into the linked list or the array.
- Interpolate the point into the proper square at the bottom level and change the bit representing that square in the bit vector to 1.
- Increment n.

2.1.3 Naive Isolate

- Repeat {
    - Pick up the scan of the bit vector from where it was terminated on the last isolate operation and continue until the first zero bit (unoccupied square) is encountered.

```
        - If an unoccupied square is found then
              return the midpoint of the square corresponding to the zero bit.
        - If all squares are occupied then
              * expand the hierarchy 1 level below
              * interpolate the points into the finer grid
              * update the bit vector to represent the current bottommost level
    } until an empty square is found.
```

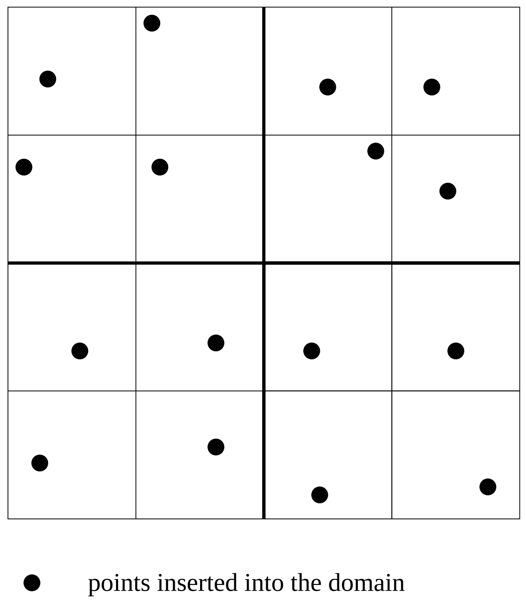
- Insert the point returned using the algorithm in Section 2.1.2.

2.1.4 Analysis of the Naive Algorithms

The naive *insert*, naive *isolate*, and the naive lazy *insert* and the naive lazy *isolate* do not guarantee constant amortized time per operation. Suppose n points have been inserted before the first *isolate*. Without loss of generality, assume n is an exact power of four. In the worst case, $\log_4 n + 2$ levels are needed to guarantee a free square. In Figure 2.2, only three levels are shown; the squares need to be decomposed into an additional level in order to have free squares in the domain.

In order to find an empty square in the domain, the *repeat* loop has to be executed $\Theta(\log n)$ times since there are $\Theta(\log n)$ levels in the hierarchy. There are n points to be interpolated each time through the loop and $n+1$ operations (n *inserts* and 1 *isolate*), thus giving a $\Theta(\log n)$ amortized running time. Note that postponing the work of fixing the bit hierarchy in the lazy approach has no effect on the running time. Also observe that even if the inserted points are uniformly distributed, $\Theta(\log n)$ average case cost per operation is to be expected for a sequence

Figure 2.2: Scenario where $\log_4 n + 2$ levels are needed (with $n = 16$)



of insertions followed by one isolate operation. On the other hand, if *isolate*s occur reasonably frequently, constant amortized running time seems a reasonable hope.

2.2 A Refined Approach

In order to guarantee constant amortized running time, we decompose the domain into squares in one global refinement so that at least one empty square is guaranteed to exist. We also need to keep track of the entire square hierarchy, instead of only the bottommost level. The inserted points are interpolated only once on each *isolate*. The smallest squares in which the inserted points lie are found. The bit hierarchy can then be built by checking whether any of the four smaller squares making up the current square is occupied or not, without interpolating the inserted points. Therefore, we start from the lowest level in the square hierarchy and build the hierarchy upwards.

As before, assume there are n points in the domain when an isolate operation is encountered. When there are no empty squares, the domain is decomposed into 4^k equal-sized squares, where k is equal to $\lceil \log_4(n+1) \rceil$. With this k, an unoccupied square is guaranteed to exist. The n points already in the domain are interpolated into this finer square grid, containing $\Theta(n)$ squares. The bit hierarchy is then built by repeatedly doing an OR operation on the four bits corresponding to the four smaller squares in the level below the current level, until the level before the decomposition is reached. A linked list (or a simple array) is required to store all the points inserted. The hierarchy is built by looking at the bits corresponding to the squares in the level below, not at the points inserted. Finally, the bits in the hierarchy are scanned until the first zero bit which represents one of the largest free squares is encountered. The midpoint of the square which corresponds to this zero bit is inserted into the domain.

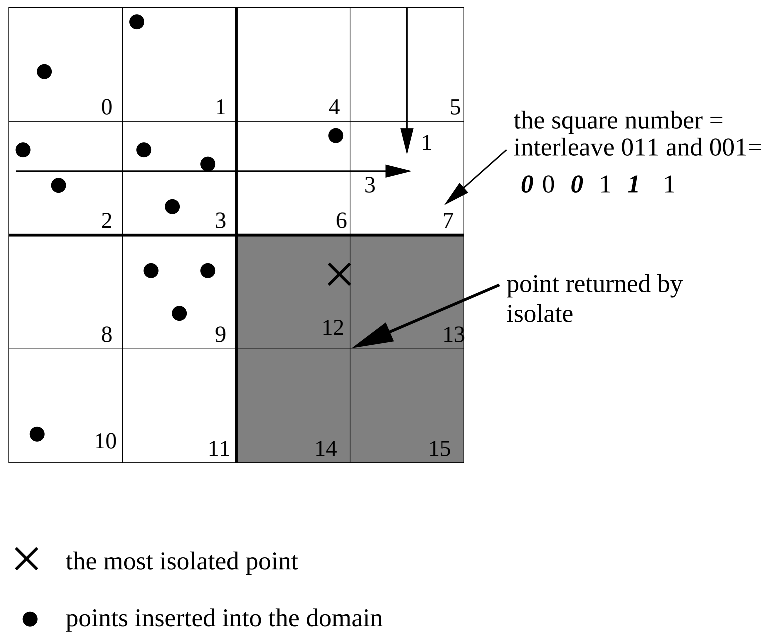
As in the naive algorithm, a lazy approach in which points are put on a queue during *inserts* and the bit hierarchy is fixed only when an isolate operation is encountered can also be used. The lazy approach has no effect on the analysis of the running time.

As an example, consider the scenario in Chapter 1 again after the domain is divided in squares. In Figure 2.3, the largest unoccupied square is shaded.

The number of squares or bits in the deepest level is linear in n. With k equal to $\lceil \log_4(n+1) \rceil$, 4^k lies between $n+1$ and $4n$ (inclusive).

The total number of bits in the hierarchy is also linear in n. Since four smaller squares are grouped into a bigger one in the next level up in the hierarchy, the number of bits in each level in the hierarchy is reduced by a factor of four. Assume the bottommost level in the hierarchy is level 0 with 4^k squares. At level i, there are 4^{k-i} squares. The total number of squares in the hierarchy, and hence bits required in the representation vector, is $\sum_{i=0}^{k} 4^{k-i} < \frac{4^{k+1}}{3} \leq \frac{16}{3}n$.

Figure 2.3: EXAMPLE WITH $n = 12$ AND $k = 2$

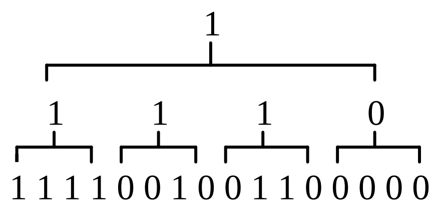

0
1
4
5
2
3
6
7
8
9
12
13
10
11
14
15
1
3
the square number = interleave 011 and 001=
0 0 0 1 1 1
point returned by isolate
the most isolated point
points inserted into the domain

2.2.1 Details of Insert

- Interpolate the point into the proper square at the bottom level. This square is the current square.

- Fix the bit hierarchy :

Figure 2.4: The bit vector hierarchy



The bit vector :
1 1 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 0 0 0

```
While the current square is not marked (current bit is 0) do {
    - mark the current square as occupied (change the current bit to 1)
    - current square := parent of current square
}
```

- Increment n

2.2.2 Details of Isolate

- Pick up the scan of the bit hierarchy from where it was terminated on the last isolate operation and continue until the first zero bit is encountered.
- If an unoccupied square is found, return the midpoint of the square corresponding to the zero bit.
- If all squares are occupied then
  - Compute $k := \lceil \log_4(n+1) \rceil$
  - Decompose the domain into 4^k squares

  - Interpolate all n points into the grid of 4^k squares
  - Extend the bit hierarchy to levels below the original hierarchy
  - Scan the bit hierarchy, starting at the first square of the level below that in which all squares were occupied, until the first zero bit is found which corresponds to one of the largest empty squares and return the midpoint of the square found.

- Insert the point returned as in Section 2.2.1.

2.2.3 Determining the Square in Which a Given Point Lies

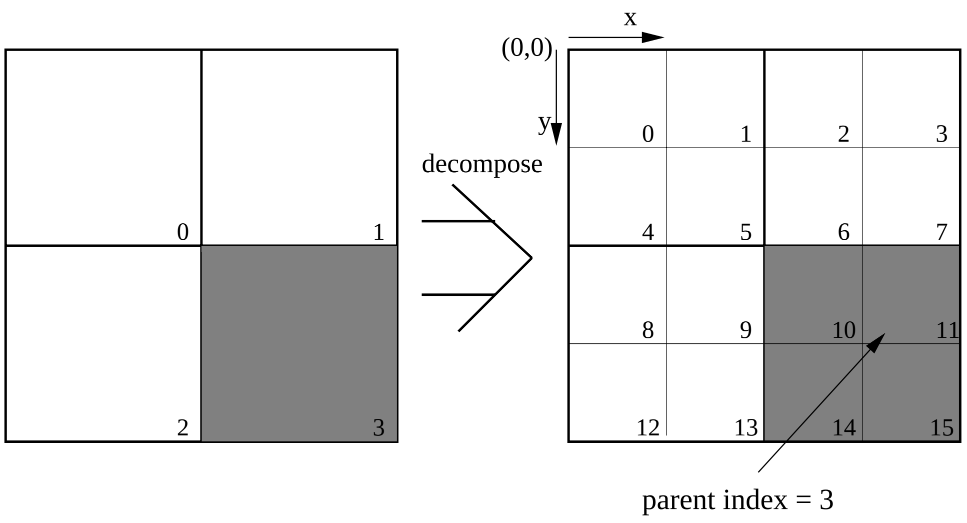
In both the insert and isolate algorithms, the square in which a given point lies has to be determined in order to interpolate the point into the square grid. In this section, we are going to discuss two different conventions of numbering the squares and show that we can interpolate a point in constant time and $o(n)$ space using either approach.

The Concatenating Approach

The first approach is to number the squares across the rows and then down the columns, starting from zero. The index of the square at row r and column c is $X * r + c$ for a grid of X^2 squares assuming the columns and rows start from zero. In Figure 2.5, square 11 has row 2 and column 3, and has index equal to $2 * 4 + 3$. Since the row number and column number can be determined in constant time, the square index can be determined in constant time. Hence, a point can be interpolated into the square grid in constant time using this approach. This is called the concatenating approach because the square index is determined by concatenating two bit patterns: the row number and the column number.

It can be verified that the index of the parent of a square indexed j is $\lfloor \lfloor j/X \rfloor /2 \rfloor * (X/2) + \lfloor (j \bmod X)/2 \rfloor$. Since the number of squares in the parent grid is reduced by a factor of four, i.e., $\frac{X^2}{4}$, the number of rows and columns in the parent grid is $\sqrt{\frac{X^2}{4}} = \frac{X}{2}$. $\lfloor j/X \rfloor$ gives the row number in the current grid, and $j \bmod X$ gives the column number. For example, the parent of square 11 is $\lfloor \lfloor 11/4 \rfloor /2 \rfloor * (4/2) + \lfloor (11 \bmod 4)/2 \rfloor = 3$ in Figure 2.5. Therefore, we can find the index of the parent square in constant time.

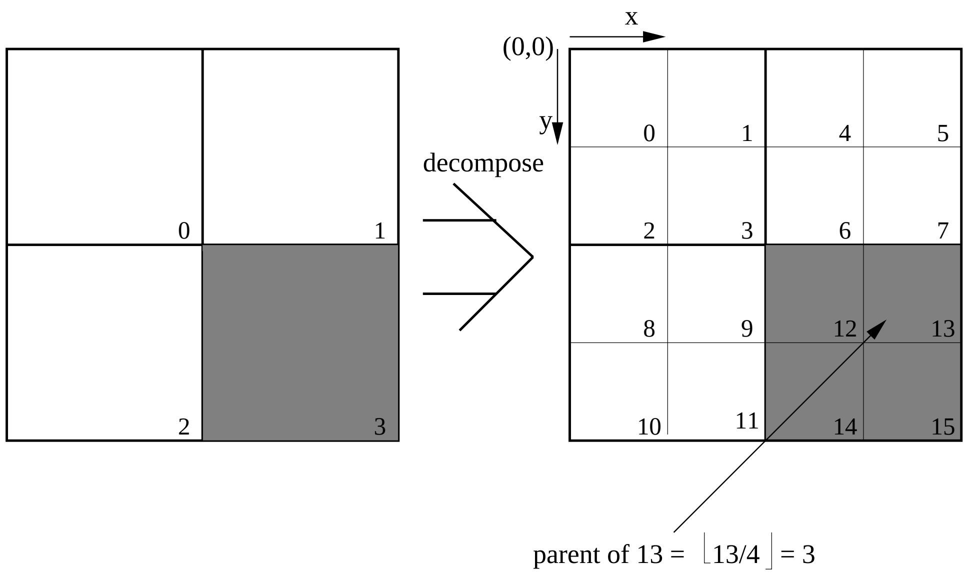
Figure 2.5: The concatenating numbering convention



The Interleaving Approach

The second approach of numbering the squares is conceptually more complicated and more interesting, but the implementation is simpler. Assume the following convention of numbering the squares (and hence the bits) in the hierarchy: the numbering starts from zero at each level and starts from the top level in the square hierarchy. Thus, the 1 bit representing the square domain in the topmost level has index 0. A parent square is decomposed into 4 children squares when going down one level in the square hierarchy. The children of a square indexed j have indices $4*j$ (upper left), $4*j+1$ (upper right), $4*j+2$ (lower left) and $4*j+3$ (lower right) as illustrated in Figure 2.3. Hence, the parent of the square indexed j has index $\lfloor \frac{j}{4} \rfloor$. See Figure 2.6.

Figure 2.6: The interleaving numbering convention



The square's index can be determined in constant time by interleaving two bit patterns, as will be proved in Theorem 2.1, and hence this is called the interleaving approach. The two bit patterns are the binary representations of the numbers of base squares vertically above and horizontally to the left of the current square. For example, the square numbered 7 in Figure 2.3 has one base square above it and three base squares to its left. Interleaving 001 (= 1) and 011 (= 3) gives 0 **0** 0 **1** 1 **1**, which is the binary representation of the index 7.

Lemma 2.1 *The index of the square in which a point lies can be determined by interleaving two bit patterns which represent the number of base squares vertically above and horizontally to the left of the given point.*

Proof of Lemma 2.1:

The proof is by induction on the number of levels in the square hierarchy. The induction hypothesis is as follows: the index of the square in which a point lies can be determined by interleaving two bit patterns when there are k levels in the square hierarchy.

Basis: The induction hypothesis is trivially true when there is only one level in the hierarchy, i.e., when there is only one square and that square represents the entire domain in the hierarchy.

When there are two levels in the hierarchy, square 0 has zero squares above and zero squares to the left. Interleaving 0 0 and 0 0 gives 0 **0** 0 **0** which is equal to the index 0. Square 1 has zero squares above and one square to the left. Interleaving 0 0 and 0 1 gives 0 **0** 0 **1** which is 1. Square 2 has one square above and zero squares to the left. Interleaving 0 1 and 0 0 gives 0 **0** 1 **0** which is 2. Similarly, square 3 has one square above and one square to the left. Interleaving 0 1 and 0 1 gives 0 **0** 1 **1** which is 3.

The induction step: Assume the induction hypothesis is true with k levels in the hierarchy, i.e., the square index can be determined by interleaving two bit patterns when there are k levels in the hierarchy. Consider a particular square at level k with index j which has y squares above and x squares to the left. By the induction hypothesis, j is equal to interleaving y and x. Suppose there are now $k + 1$ levels in the hierarchy. When there is an additional level in the hierarchy, each of the original base squares is decomposed into 4 equal sized squares. Suppose the square indexed j is decomposed into j_0 (upper left), j_1 (upper right), j_2 (lower left) and j_3 (lower right). Square j_0 has $2y$ base squares at level $k + 1$ above and $2x$ base squares to its left. The binary representation of $2y$ is that of y shifted one bit left, and $2x$ is that of x shifted one bit left, both shifts with a zero bit appended to the right. Hence, interleaving $2y$ and $2x$ is the same as interleaving y and x with two zero bits appended to the right. Interleaving y and x gives j by the induction hypothesis. Thus, $j_0 = 4 * j$. Similarly, j_1 has $2y$ squares above and $2x + 1$ squares to its left. The binary representation of $2x + 1$ is the same as shifting x left by one bit and appending a 1 bit to the right. So, interleaving $2y$ and $2x + 1$ gives $j_1 = 4 * j + 1$. Similarly, j_2 has $2y + 1$ squares above and $2x$ to its left and interleaving $2y + 1$ and $2x$ gives $j_2 = 4 * j + 2$. Finally, j_2 has $2y + 1$ squares above and $2x + 1$ to its left and interleaving $2y + 1$ and $2x + 1$ gives $j_3 = 4 * j + 3$. Therefore, the induction hypothesis is also true when there are $k + 1$ levels in the hierarchy.

■

Having established that the index of the square in which a point lies can be determined by interleaving two bit patterns in Lemma 2.1, we have to be able to perform the interleaving step in constant time and $o(n)$ space such that the interleaving step would not add to the space and time requirement of the insert and isolate algorithms. It is obvious that creating a table of all possible bit patterns interleaved and performing a table lookup takes constant time. However, the space required for the table is $\Theta(n)$, thus adding to the space requirement of *insert*s and *isolate*s. The following lemma gives a constant time and $o(n)$ space interleaving algorithm. The basic idea is to split the bit patterns into halves, create a table of all the half bit patterns, perform table lookups for the half bit patterns and combine the bit patterns. The table size required for the half bit patterns is $o(n)$, so we do not have to split again.

Lemma 2.2 *Interleaving two given bit patterns which represent the number of base squares vertically above (y) and horizontally to the left (x) of a base square takes constant time and $o(n)$ space with the following algorithm:*

<u>INTERLEAVE</u> *(y,x):*

1. *Let x_1 = the first half of the bits in x, and x_2 = the second half of the bits in x. Similarly, let y_1 = the first half of the bits in y, and y_2 = the second half of the bits in y.*
2. *yx_1 = result of table lookup of interleaving y_1 and x_1.*
3. *yx_2 = result of table lookup of interleaving y_2 and x_2.*
4. *Let yx be the result of interleaving y and x. yx = concatenate yx_1 and yx_2.*

Proof of Lemma 2.2:

It is obvious that splitting two bit patterns into halves, interleaving each pair and then concatenating the two pairs is equivalent to interleaving the original two bit patterns. Thus, the above algorithm interleaves two given bit patterns.

There are $4^{\lceil \log_4(n+1) \rceil} \leq 4n$ base squares. So there can be at most $\sqrt{4n} = 2\sqrt{n}$ base squares vertically above and horizontally to the left of any base square. Hence, the number of bits required to represent the number of squares above (y) and to the left of (x) any base square is at most $\lceil \log_2(2\sqrt{n}) \rceil = \lceil 1 + \frac{1}{2}\log_2 n \rceil$. The resulting number ($yx$) of interleaving the two bit patterns has at most $2\lceil 1 + \frac{1}{2}\log_2 n \rceil$ bits. We will represent x and y with $\lceil \frac{1}{2}\log_2 n + 1 \rceil$ bits and x_1, x_2, y_1 and y_2 with $\lceil \frac{1}{4}\log_2 n \rceil + 1$ bits. Thus, yx_1 and yx_2 have $2\lceil \frac{1}{4}\log_2 n \rceil + 2$ bits. Hence, the table has size $2^{2\lceil \frac{1}{4}\log_2 n \rceil + 2} \leq 2^{\frac{1}{2}\log_2 n + 3} = 8\sqrt{n} = o(n)$ bits.

Splitting x and y into two halves can be done by a mask. Specifically, $x_2 = x$ AND $(2^{\lceil \frac{1}{4}\log_2 n \rceil + 1} - 1)$ and $x_1 = (x - x_2)$ SHIFT RIGHT $(\lceil \frac{1}{4}\log_2 n \rceil + 1)$ bits. Table lookups are constant time operations using our RAM model of computation. Concatenating two bit patterns can be done by a SHIFT and an OR operation, i.e., $yx = (yx_1$ SHIFT LEFT $2\lceil \frac{1}{4}\log_2 n \rceil + 2$ bits) OR yx_2. Hence, the above algorithm takes constant time. ■

Theorem 2.1 *Determining the square index in which a given point with coordinates (X, Y) lies takes constant time and $o(n)$ space of overhead.*

Proof of Theorem 2.1:

The number of base squares above (y) and to the left of (x) a given point with coordinates (X, Y) can be determined in constant time. Assume the origin of the square domain is at the top left corner, the x-coordinates increase towards the right and the y-coordinates increase towards the bottom of the domain. See Figure 2.6. Suppose the size of a base square is s. Then, the number of base squares to the left

(x) of a given point with coordinates (X, Y) is $\lfloor \frac{X}{s} \rfloor$. Similarly, the number of base squares vertically above (y) the point (X, Y) is $\lfloor \frac{Y}{s} \rfloor$.

We have proven in Lemma 2.1 that the index of the square in which a point lies can be determined by interleaving two bit patterns that represent the number of base squares vertically above and horizontally to the left of a base square respectively. We have also given an algorithm in Lemma 2.2 that interleaves the two bit patterns in constant time and $o(n)$ space of overhead. Therefore, the index of the square in which a given point with coordinates (X, Y) lies can be determined in constant time and $o(n)$ space. ■

2.2.4 Analysis of the Runtime

In this section, we shall show that the insert and isolate operations, taken together have constant amortized time per operation. Let n be the total number of points inserted so far.

Most *inserts* only involve interpolating a point into the square grid and would not go very far up in the hierarchy because the number of bits in the hierarchy decreases by a factor of four with each level upward. However, there are at most $\lceil \log_4(n+1) \rceil$ levels in the hierarchy. The running time of an *insert* is constant per level and so can be $\Theta(\log n)$ in the worst case. On the other hand, a single *isolate* can take $\Theta(n)$ time because it takes $\Theta(n)$ work to decompose the domain into at most $4n$ squares, build the bit hierarchy, scan the bits in the hierarchy and insert the point.

However, the running time is constant amortized per operation when *insert* and *isolate* are taken together. Divide any sequence of insert and isolate operations into chunks of operations in which the number of points inserted is quadrupled. Note that a point is inserted in both the insert and isolate operations.

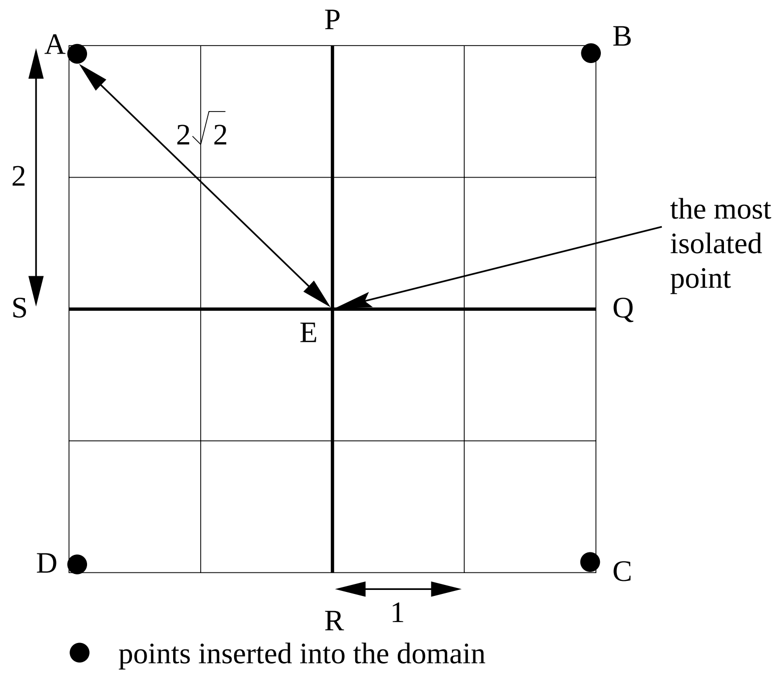
Suppose there are 4^m points at the beginning of the current block of operations, where $m \geq 0$. At the end of the current block, there are $4^{m+1}-1$ points. The number of levels is at most $m+1$ in the current block and there are $3(4^m)$ operations in the current block. In an insert operation, a point has to be interpolated into the current grid and the current square hierarchy has to be fixed. Interpolating $3(4^m)$ points takes $\Theta(4^m)$ operations. There are at most $m+1$ levels in the current hierarchy, and so the number of bits in the bit hierarchy is at most $\sum_{i=0}^{m+1} 4^i = \frac{4^{m+2}-1}{3}$. Hence, the number of operations to fix the hierarchy in the current block of operations is at most the number of bits in the hierarchy, $\Theta(4^{m+2})$. There can be multiple *isolate*s in the current block, but there is at most one decomposition into a finer grid in each block of operations where the number of operations is quadrupled. There are at most $4^{m+1}-1$ points in the current block, thus decomposition and interpolating the points into a finer grid takes $\Theta(4^{m+1})$ operations. In the bit scan, we pick up from where we left off from the last *isolate*. So, the total number of bits scanned in the current block is at most the number of bits in the current bit hierarchy which is $\Theta(4^{m+2})$. Thus, the worst case running time of the current block of operations is at most the total cost of one decomposition and interpolation into a finer grid, scanning the bits in the hierarchy, interpolating the new points and fixing the bit hierarchy which is $\Theta(4^{m+2})$. There are a total of $\Theta(4^m)$ operations. Therefore, the amortized running time of insert and isolate operations taken together is equal to the worst case running time of the current block of operations divided by the total number of operations in the current block, and so is $\Theta(4^{m+2})/\Theta(4^m) = \Theta(1)$.

2.2.5 The Quality of Isolate

The worst case ratio measures the quality of the isolate operation. It is the ratio of the maximum distance between any point in the domain and the closest inserted point to it among those actually present to the distance between the returned isolate point and its closest possible inserted point in the domain.

It is desirable to have the worst case ratio as small as possible. We will show that the strategy as described in the preceding sections (both the naive algorithms and the refined algorithms) gives a worst case ratio of $4\sqrt{2}$.

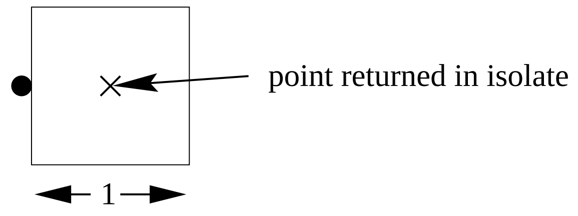
Figure 2.7: ILLUSTRATION OF THE $4\sqrt{2}$ WORST CASE RATIO



In order to compute the worst case ratio, we have to determine: (a) the maximum distance between any point in the domain and its closest inserted point, and (b) the minimum distance between the returned isolate point and its closest possible inserted neighbour in the domain. The worst case ratio is defined to be $(a)/(b)$.

To get the maximum distance between any point in the domain and its closest inserted point, we have the following scenario in which the inserted points are as far away from each other as possible: all the squares of size 2 or greater are occupied; somewhere in the domain, there are four points occupying the corners of four size 2 squares, as shown in Figure 2.7, and there are no other points in the circle through the four points. The circle passing through points A, B, C and D is the circle with maximum radius containing the entire size 2 squares and any four inserted points in the domain. The most isolated point in the domain should be point E which is equidistant from A, B, C and D. Therefore, the distance between the most isolated point E and an inserted point in the domain is $2\sqrt{2}$ units. However, using our strategy, the midpoint of a unit size square is returned in *isolate*. In the worst case, the midpoint of a unit size square is returned which has a point 1/2 unit away. See Figure 2.8.

Figure 2.8: A SIZE 1 SQUARE WITH THE CLOSEST POINT BEING $\frac{1}{2}$ UNITS AWAY



The distance between A and E is $2\sqrt{2}$ units, and the minimum possible distance between the returned point and an inserted point is 1/2 units. Therefore, the worst case ratio is $\frac{2\sqrt{2}}{1/2} = 4\sqrt{2} \approx 5.66$.

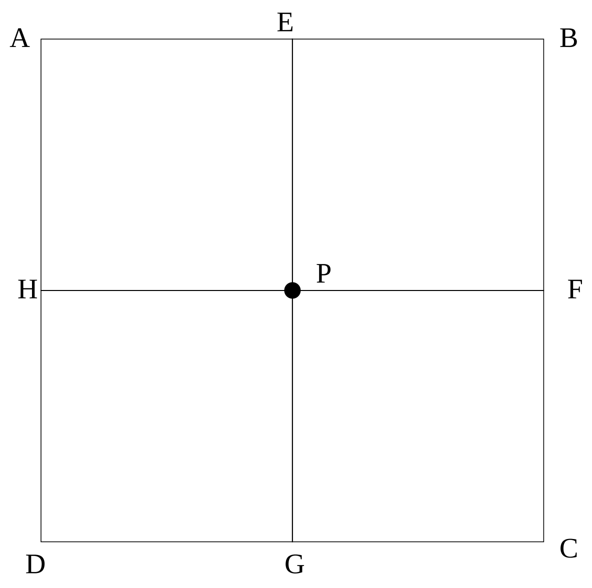
Theorem 2.2 *Using the square approach as described in the preceding sections,* insert *and* isolate *can be performed in constant amortized time per operation, linear space and* $4\sqrt{2}$ (≈ 5.66) *worst case ratio.*

2.2.6 A Correction for the Boundaries

Points lying on the boundaries of the squares may be inserted. Consider the case in which the first two operations are consecutive *isolate*s. The point inserted by the first *isolate* lies on the boundary of the four squares after decomposition in the second *isolate*. See Figure 2.9. To solve this problem, we assume that a square includes both the upper and left boundary edges and the top upper corner point, but excludes the lower and right boundary edges. Furthermore, if the square has boundary edges on the domain boundary, the boundary edges do not belong to the square. Therefore, the square is assumed to be unoccupied if there are no other points inserted in the square even though the domain boundary is occupied. In our example in Figure 2.9, square EBFP consists of edges EB and EP and corner point E, point P belongs to square PFCG, and the bottom level of the square hierarchy in Figure 2.9 consists of bits 0 0 0 1.

Note that with our boundary corrections, we cannot have points B, C and D lying on the boundary in Figure 2.7. In order for all the squares of size 2 to be occupied to compute the worst case ratio, we need a point in each of squares PBQE, EQCR and SERD. We can get points as close to the boundary as possible, hence, the worst case ratio is very close to and is slightly less than $4\sqrt{2}$.

Figure 2.9: Illustration of the boundary cases



The boundary correction convention discussed in this section is not unique. Other correction conventions may be used.

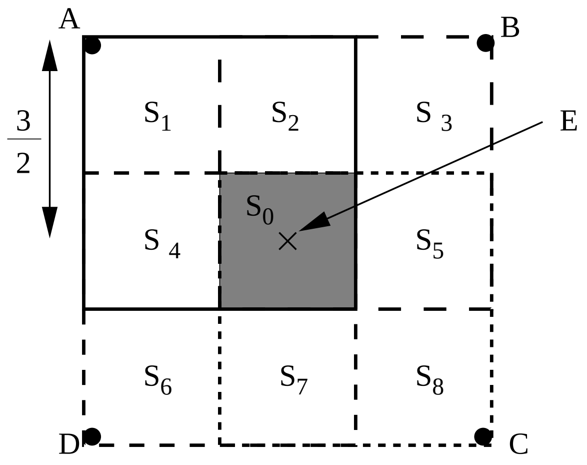
2.3 Improving the Worst Case Ratio

The technique described in Section 2.2 can be modified to improve the worst case ratio. We propose two different approaches, both of which introduce another set of squares to reduce the maximum distance between any point in the domain and its closest inserted point. The first idea achieves the improvement by having an additional set of squares overlapping the first. The second technique uses a rotation so that the second set is at a forty-five degree angle to the first. Note that the actual worst case ratios are slightly smaller than those stated below with the boundary correction convention discussed in Section 2.2.6.

2.3.1 Overlapping

The maximum distance between any point in the domain and its closest inserted point can be reduced by an overlapping technique. There are four possible double-sized squares containing a unit sized square. In Figure 2.10, four distinct squares of size 2 (in solid and dashed lines) cover the size one square which is shaded in the middle.

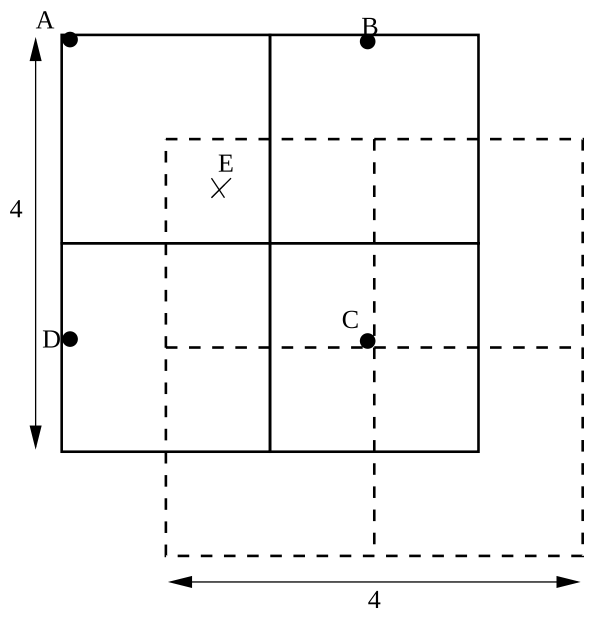
Figure 2.10: IMPROVING THE WORST CASE RATIO WITH OVERLAPPING



Note that the three extra levels of overlapping are just lying around and they do not form additional hierarchies. See Figure 2.11. The intersection of the quadruple sized squares in dashed lines and the quadruple sized squares in solid lines is not a double sized square.

There are two possible ways to represent the square hierarchy in this case, exhibiting space and time tradeoffs, respectively. The first representation uses a bit to represent each of the three additional overlapping squares in each level. The second representation determines whether an overlapping square is occupied on the

Figure 2.11: Overlapping squares do not form hierarchies



fly by doing an OR operation on the four bits representing the squares in the level below that form the overlapping square.

With the first representation, the number of bits needed to represent the hierarchy of squares is four times the number without overlapping, starting from level one. Since the overlapping squares cannot improve the worst case ratio even if they are present in the bottommost level of the square hierarchy, we do not have the three extra overlapping levels at the bottommost level of the bit vector hierarchy. At most $4n$ bits are needed for level 0. Hence, the total number of bits is at most the number of bits in level 0 plus four times the maximum possible number of bits needed starting from level 1 in the hierarchy, which is at most $4n + 4(\frac{4}{3}n)$. Therefore, at most $\frac{28}{3}n$ bits are needed for the bit vector hierarchy using the first representation.

The number of bits used to represent the square hierarchy is not increased with the second representation with care in coding. With the implicit bit representation, the insert algorithm is exactly the same as that in Section 2.2.1. The isolate algorithm is the same as that in Section 2.2.2, except that the simple bit scan is replaced by the following MODIFIED BIT SCAN, which makes use of the INTERLEAVE algorithm in Lemma 2.2, to find an empty square (including the overlapped squares) at level $i \geq 1$ in the square hierarchy:

<u>MODIFIED BIT SCAN (i,k):</u>

```
1. If an unoccupied square (not the overlapping ones) is found by doing a linear
   scan on the bit vector representing the current level $i$ in the hierarchy, return
   the unoccupied square found.

2. If the bit vector representing the current level $i$ in the hierarchy consists of
   only 1 bits,
   (a) Number of squares in the level below (level $i-1$), $N_{i-1} = 4^{k-i+1}$
   (b) For $r = 1$ to $\sqrt{N_{i-1}} - 1$ and $c = 1$ to $\sqrt{N_{i-1}} - 1$ (for each square in level
       $i-1$) do
       i. $S_0$ = INTERLEAVE (c, r)
       ii. $S_1$ = INTERLEAVE (c-1, r-1)
       iii. $S_2$ = INTERLEAVE (c, r-1)
       iv. $S_3$ = INTERLEAVE (c+1, r-1)
       v. $S_4$ = INTERLEAVE (c-1, r)
       vi. $S_5$ = INTERLEAVE (c+1, r)
```

```
vii. $S_6$ = INTERLEAVE (c-1, r+1)
viii. $S_7$ = INTERLEAVE (c, r+1)
ix. $S_8$ = INTERLEAVE (c+1, r+1)
x. $Square_0 = S_0$ OR $S_1$ OR $S_2$ OR $S_4$
xi. $Square_1 = S_0$ OR $S_2$ OR $S_3$ OR $S_5$
xii. $Square_2 = S_0$ OR $S_4$ OR $S_6$ OR $S_7$
xiii. $Square_3 = S_0$ OR $S_5$ OR $S_7$ OR $S_8$
xiv. If any of $Square_0$, $Square_1$, $Square_2$ or $Square_3$ is empty, return
     the empty square.
```

Step 1 in algorithm Modified Bit Scan is the same as the simple bit scan which tries to find an empty non-overlapping square in the current level i. Step 2 tries to find an empty square (both overlapping and non-overlapping) by looking at the appropriate bits in the level below (i.e., level $i-1$). Step 2(b) looks at all the squares at level $i-1$ that are at least one square away from the domain boundary. The current square is S_0 and the relative positions of S_j where $j = 1, \ldots, 8$ are illustrated in Figure 2.10. Recall that Interleave (y, x) is the constant-time algorithm given in Lemma 2.2 which gives the square index of a particular square, with y squares above and x squares to the left. $Square_j$ where $j = 0, \ldots, 3$ are the four overlapping squares at level i that overlap the current square S_0.

Modified Bit Scan runs in time proportional to the number of squares in level $i-1$. Modified Bit Scan is called at each level starting from the top level k until an empty square is found or until level 0 is reached. So, the total running time of scanning for overlapping and non-overlapping squares by calling

MODIFIED BIT SCAN is proportional to the total number of squares in the square hierarchy which is $\Theta(n)$, and hence is of the same order as the simple bit scan in Section 2.2.2. Therefore, the amortized running time per operation is not changed by using MODIFIED BIT SCAN instead of the simple bit scan.

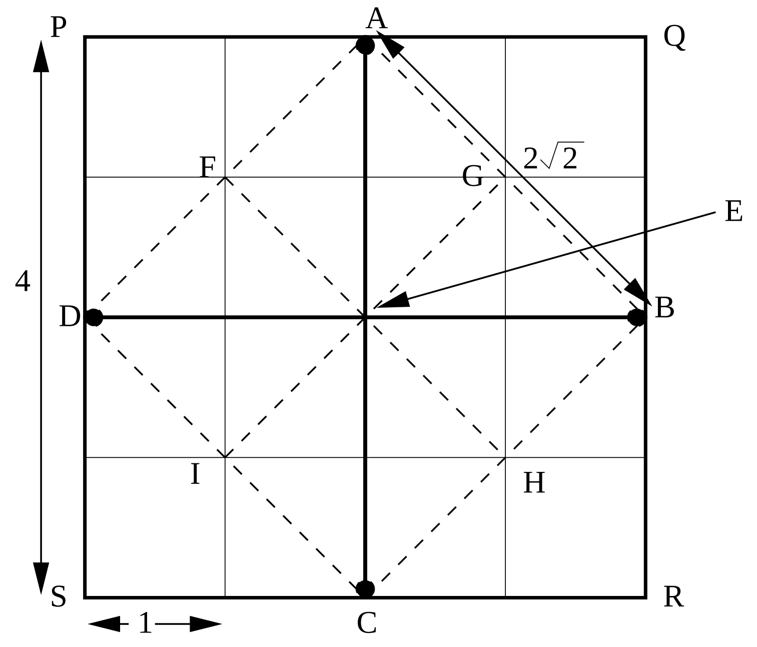
The worst case ratio is $3\sqrt{2}$ in this case. Consider the situation given in Figure 2.10. The four points, A, B, C, and D occupy the corners of four size 2 squares which overlap the middle shaded unit sized square, and there are no other points lying in the circle passing through points A, B, C and D. All squares with size at least 2 are occupied. Point E is the centre of this circle, equidistant from points A, B, C and D. The distance from A to E is $\frac{3}{2}\sqrt{2}$ units. It is easily seen that this is the maximum distance that a point may be from its nearest neighbour if all four overlapping size 2 squares are occupied. On the other hand, the midpoint of a size 1 square with a point 1/2 unit away could be returned by *isolate*. Therefore, the worst case ratio with overlapping is $\frac{3}{2}\sqrt{2}/\frac{1}{2}$ which is $3\sqrt{2} \approx 4.24$.

Theorem 2.3 *The worst case ratio of the square approach with overlapping is* $3\sqrt{2}$ (≈ 4.24).

2.3.2 Diamonds

The maximum distance between a point in the domain and the closest inserted point to it can also be reduced by having another smaller square rotated and embedded in each square in the hierarchy. We call the additional rotated squares *diamonds*. A diamond is a 45-degree-rotated square of reduced size. In Figure 2.12, the dotted square ABCD is a diamond of the size 4 square, PQRS. The diamond has sides of length $2\sqrt{2}$.

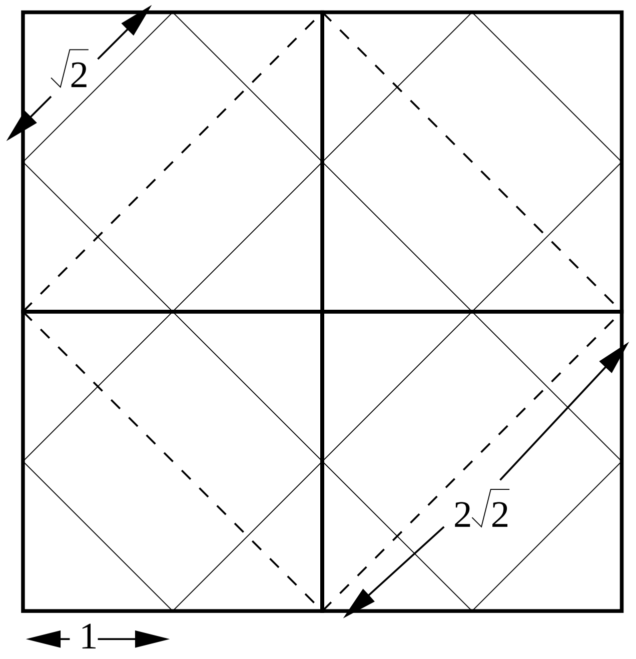
Figure 2.12: Improving the worst case ratio with diamonds



As in the overlapping approach in Section 2.3.1, the diamonds do not form another hierarchy. Observe, for instance, the four diamonds of size $\sqrt{2}$, each corresponding to a square of size 2, shown in solid lines in Figure 2.13. The diamonds cannot be grouped together to form a diamond of size $2\sqrt{2}$ as shown in dashed lines. Nevertheless, diamonds do provide useful information. If a square contains a point, but the diamond with the same centre is empty, it may give a good solution to an isolate query.

In order to preserve the constant amortized running time per operation, we do not want to look at all the points already inserted to determine whether a diamond is occupied or not. For this reason, we seek a more complex solution. A natural

Figure 2.13: DIAMONDS DO NOT FORM A HIERARCHY

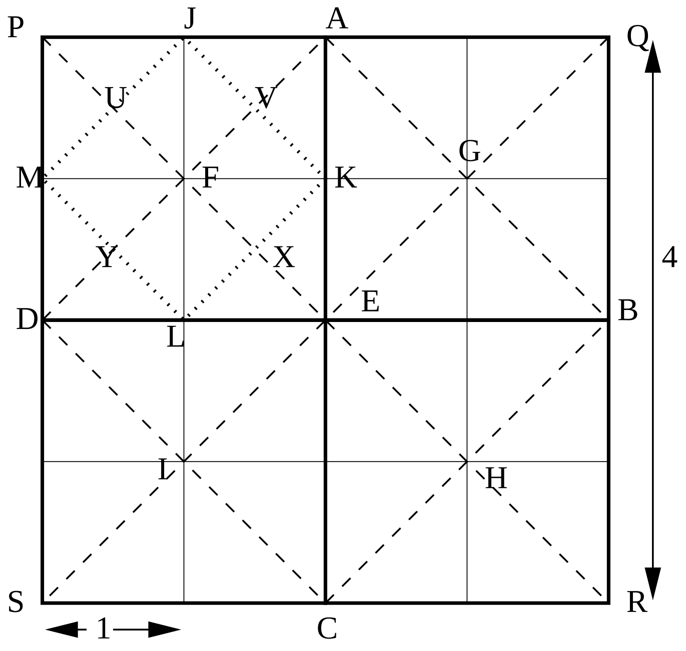


approach is to have a separate bit for each diamond. The diamond can be occupied only if its enclosing square is occupied. Moreover, given a point in a hierarchy of squares, its containment or non-containment in the corresponding diamonds may be complex. The obvious approach of checking each possible diamond in the associated square hierarchy will lead to a $\Theta(\log n)$ solution even on an amortized basis.

Observe that each diamond is made up of eight triangles which are quarters of the squares in the level below in the hierarchy. In Figure 2.12, diamond ABCD is made up of triangles AFE, FED, DEI, CIE, CHE, HBE, BEG and AGE. A diamond is occupied if any of the eight triangles making up the diamond is occupied. Similarly, a square is made up of four triangles. For example, in Figure 2.14, square AEDP is made up of triangles AFE, FED, PFD and APF. So, a square is occupied if any of the four triangles making up the square is occupied. By breaking up a square into four triangles, we have a clean hierarchy. The triangle in the current level is made up of four smaller triangles in the level below. For instance, triangle AFE is made up of triangles AVK, VKF, KXF and KXE in Figure 2.14. The smaller triangles are needed to determine whether the diamond for the square in the next level up is occupied or not. In Figure 2.14, diamond JKLM is made up of triangles JVF, VKF, KXF, FXL, FLY, FYM, FMU and JUF. Hence, whether a square is occupied or not can be determined by an OR operation on the 4 bits representing the four triangles making up the square and whether a diamond is occupied or not can be determined by an OR operation on the 8 bits representing the eight triangles making up the diamond. Both of these are constant time operations, therefore, constant amortized running time can be preserved. However, a more complex numbering convention similar to that in Section 2.2.3 is needed to determine the triangle index.

As in the overlapping technique in Section 2.3.1, we can represent the square hierarchy with diamonds using either of two approaches. The first approach is to use a bit to represent each diamond explicitly. Each square in the hierarchy is represented by 4 bits, each of which represents a quarter of the square. There are at most $\frac{16}{3}n$ squares in the hierarchy, and so we need at most $4(\frac{16}{3}n) = \frac{64}{3}n$ bits to represent the triangles. There is one diamond for each square. We also use one bit to represent each diamond in the hierarchy. Hence, a total of at most $\frac{64}{3}n + \frac{16}{3}n = \frac{80}{3}n$ bits are required to represent the diamonds explicitly. In the second approach, we can determine whether a diamond is occupied or not on the

Figure 2.14: Diamonds are made up of 8 triangles



fly by doing an OR operation on the 8 bits representing the eight triangles making up the diamond. No bits are used to represent the diamonds explicitly. Thus, the space requirement is at most $\frac{64}{3}n$ bits for the implicit representation.

The algorithm for the insert operation is the same as in Section 2.2.1 except that points are inserted into triangles instead of into squares, and one bit is used to represent each triangle. When determining whether all squares at a particular level are occupied in the isolate operation, if all the squares are occupied, we have to check whether all the diamonds corresponding to the squares at that level are occupied. If there are empty diamonds, the centre of an empty diamond is returned for *isolate*. Therefore, we only go down one level in the hierarchy when there are neither free squares nor diamonds.

To obtain the worst case ratio, imagine that all the size 2 squares and $\sqrt{2}$ and $2\sqrt{2}$ diamonds are occupied. Somewhere in the domain, we have four points A, B, C, D occupying the corners of a size $2\sqrt{2}$ diamond such that the four size 2 squares covered by the diamonds are also occupied and there are no other points in the circle passing through points A, B, C and D. See Figure 2.12. The most isolated point should be E which is 2 units away from point A. It is obvious that the circle passing through points A, B, C and D is the circle with maximum radius containing the size $2\sqrt{2}$ diamond such that all the size 2 squares and size $\sqrt{2}$ and size $2\sqrt{2}$ diamonds are occupied. On the other hand, the midpoint of a size 1 square which has a point 1/2 unit away is returned in *isolate*. Hence the worst case ratio is $\frac{2}{1/2}$ which is 4.

Theorem 2.4 *The worst case ratio of the square approach with diamonds is* 4.

2.3.3 Overlapping With Diamonds

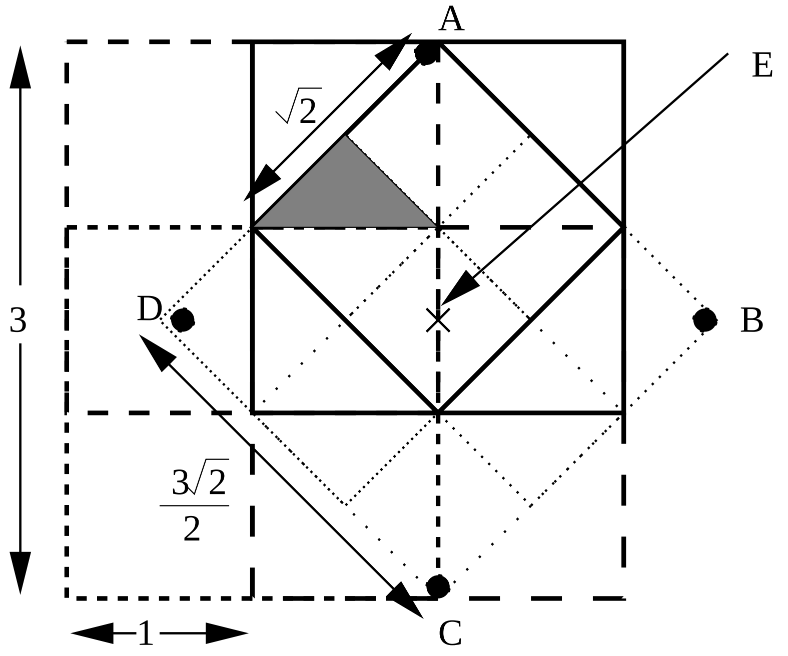
The worst case ratio can be further reduced by combining the techniques of overlapping and diamonds.

In Figure 2.15, the solid lines represent the base square and diamond and the dashed lines represent the overlapping squares and diamonds.

In order to preserve constant amortized running time, we have to cut a square into four triangles as in the diamond technique in Section 2.3.2. Each diamond is made up of eight such triangles. Thus, 4 bits are needed to represent each square in the square hierarchy.

As in the overlapping and diamond techniques, we can either represent the

Figure 2.15: IMPROVING THE WORST CASE RATIO WITH BOTH OVERLAPPING AND DIAMONDS



overlapping squares and diamonds explicitly with bits or implicitly by doing additional OR operations on the fly. With the explicit representation, 4 bits are needed to represent the triangles making up a square in the hierarchy. Thus, at most $\frac{16}{3}(4n) = \frac{64}{3}n$ bits are needed. Another 3 bits are needed for each of the additional overlapping squares starting from level 1, which is at most $3(\frac{4}{3}n) = 4n$ bits. One bit is needed for each square in the hierarchy, which is $\frac{16}{3}n$ bits. As well, 1 bit is needed for each of the four diamonds in each square at each level of the hierarchy, thus adding $4(\frac{16}{3})n$ bits. Therefore, the total number of bits needed in the explicit representation is at most $\frac{64}{3}n + 4n + \frac{16}{3}n + \frac{64}{3}n = 52n$. Alternatively, the number of bits required in the implicit approach is at most $\frac{16}{3}(4n) = 21\frac{1}{3}n$, because 4 bits are needed to represent each triangle making up a square in the hierarchy and no additional bits are required to represent the diamonds or overlapping squares.

Consider the following scenario: all the size 2 squares and size $2\sqrt{2}$ and size $\sqrt{2}$ diamonds are occupied (including the overlapping ones). The worst case scenario is captured by Figure 2.15, where four points occupy the corners of the overlapped size $\frac{3}{2}\sqrt{2}$ diamond and there are no other points in the circle containing points A, B, C and D. The distance between point A and the centre point E is $\frac{3}{2}$ units which is easily seen to be the maximum possible radius of a circle passing through points A, B, C and D. So, the maximum possible distance between the most isolated point and an inserted point is $\frac{3}{2}$ units in the scenario. On the other hand, the minimum possible distance between the returned isolate point and an inserted point is 1/2 unit. Therefore, the worst case ratio is $\frac{3}{2}/\frac{1}{2} = 3$.

Theorem 2.5 *Using the square approach together with overlapping and diamonds,* insert *and* isolate *can be performed in constant amortized time per operation, linear space and worst case ratio of 3.*

2.4 Summary of the Square Techniques

In the following table, the *implicit* approach which does not use any bits to represent the overlapping squares and diamonds is assumed.

2.5 Multiple Grids

The worst case ratio can be reduced by having squares of different sizes. For instance, when all the squares of size at least 2 are occupied, the midpoint of a

Table 2.1: Summary of the square techniques

technique	*max # of bits needed*	*worst case ratio*
squares only	$\frac{16}{3}n$	$4\sqrt{2} \approx 5.66$
squares with overlapping	$\frac{16}{3}n$	$3\sqrt{2} \approx 4.24$
squares with diamonds	$\frac{64}{3}n$	4
squares with overlapping and diamonds	$\frac{64}{3}n$	3

square of size less than 2 but greater than 1 is returned, which results in a smaller worst case ratio. The above idea is captured in the multiple grids technique. Instead of having a single grid, we have several grids which do not necessarily align. The idea is to have multiple independent hierarchies, thus having squares of multiple precisions. When there are no free squares at a particular level in the square hierarchy, instead of going down one level, we go to the next smaller hierarchy at the same level to check whether there are free squares. So, we only go down one level when all the hierarchies at the current level do not have free squares.

The insert algorithm is similar to that in Section 2.2.1, except a point has to be inserted into all of the base squares in all the grids and all the hierarchies have to be fixed when a new point is inserted. The isolate algorithm is the same as that in Section 2.2.2 except the simple bit scan is replaced by the following Multiple Hierarchy Bit Scan algorithm (assume there are m grids and the base square of grid u is greater than that of grid v if and only if $u > v$):

Multiple Hierarchy Bit Scan (k):

```
• For $i = k$ down to 0 (for each level in the hierarchies) do
    – For $j = m$ down to 1 (for each hierarchy) do
```

1. Simple bit scan on the bit vector at level i in the hierarchy representing Grid j.
2. If an empty square is found, return the midpoint of the empty square and terminate the bit scan.

The running time of the MULTIPLE HIERARCHY BIT SCAN algorithm is obviously m times the total number of bits in a square hierarchy which is $\Theta(n)$ when m is a constant. Therefore, the amortized running time per operation remains constant with multiple grids.

In this section, we are going to study the special case of having two grids and four grids, then the general case of having m grids will be discussed.

2.5.1 Double Grids

In the double grids technique, there are two independent grids. Let Grid 1 be a grid with the size of the smallest square being 1, and Grid 2 be another grid with size of the smallest square being α, where $\alpha > 1$.

There are two independent square hierarchies in this case. The goal is to determine α such that the worst case ratio is minimized. Two cases arise: (a) when all the squares at a particular level in Grid 2 are occupied, and the midpoint of a square at the same level in Grid 1 is returned; (b) when all the squares at a particular level in Grid 1 are occupied, and the midpoint of a square at the next lower level in Grid 2 is returned. The worst case ratio is minimized when the worst case ratios in case (a) and (b) are equal.

Squares Only Approach

Without loss of generality, assume all the squares with size greater than or equal to 2 are occupied. However, there exists an empty square of size α, so the midpoint of a size α square is returned by *isolate*. Using the same technique as in Section 2.2.5, the worst case ratio is equal to $2\sqrt{2}/\frac{\alpha}{2} = \frac{4\sqrt{2}}{\alpha}$. On the other hand, if all the squares with size greater than or equal to α are occupied, the midpoint of a square of size 1 would be returned. So, the worst case ratio is $\alpha\sqrt{2}/\frac{1}{2} = 2\alpha\sqrt{2}$ instead. With $\alpha = \sqrt{2}$, the two worst case ratios are equal to 4, which is an improvement over the corresponding single grid approach.

However, the number of bits required to represent the square hierarchies is increased. If there are n squares of size 1 in Grid 1, we will have $\frac{n}{2}$ squares of size $\sqrt{2}$ in Grid 2. Therefore, the total number of bits needed is at most $\frac{16}{3}n + \frac{16}{3} * \frac{n}{2} = 8n$.

Squares With Overlapping

Using the square approach with overlapping, when all the squares with size greater than or equal to 2 are occupied and the midpoint of a size α square is being returned, the worst case ratio is equal to $\frac{3}{2}\sqrt{2}/\frac{\alpha}{2} = \frac{3\sqrt{2}}{\alpha}$. See Figure 2.10. The worst case ratio, when all the squares with size greater than or equal to α are occupied and the midpoint of a size 1 square being returned, is $\frac{3}{4}\alpha\sqrt{2}/\frac{1}{2} = \frac{3\alpha\sqrt{2}}{2}$. With $\alpha = \sqrt{2}$, the two worst case ratios are both equal to 3.

With the implicit bit representation, the number of bits required to represent the square hierarchies is the same as in the case without overlapping which is $8n$.

Squares With Diamonds

Using the square approach with diamonds, the worst case ratio when all the squares with size greater than or equal to 2 and the associated diamonds are occupied, and the midpoint of a size α square is returned, is equal to $2/\frac{\alpha}{2} = \frac{4}{\alpha}$. See Figure 2.12. The worst case ratio, when all the squares with size greater than or equal to α and their associated diamonds are occupied and the midpoint of a size 1 square is returned, is $\alpha/\frac{1}{2} = 2\alpha$. With $\alpha = \sqrt{2}$, the two worst case ratios are equal to $2\sqrt{2}$, which improves over the corresponding single grid approach.

The number of bits required to represent the square hierarchies with diamonds is four times that of the squares only approach and is at most $4(8n) = 32n$ with the implicit representation.

Diamonds With Overlapping

Using the square approach with diamonds, the worst case ratio when all the squares (including the overlapping ones) with size greater than or equal to 2 and their associated diamonds are occupied, and the midpoint of a size α square is returned, is equal to $\frac{3}{2}/\frac{\alpha}{2} = \frac{3}{\alpha}$. See Figure 2.15. The worst case ratio, when all the squares with size greater than or equal to α and their diamonds are occupied, and the midpoint of a size 1 square is returned, is equal to $\frac{3}{4}\alpha/\frac{1}{2} = \frac{3}{2}\alpha$. With $\alpha = \sqrt{2}$, the two worst case ratios are both equal to $\frac{3}{2}\sqrt{2}$.

With the implicit bit representation, the number of bits required to represent the square hierarchies is the same as that for the diamond approach and is at most $32n$.

2.5.2 Quadruple Grids

Let Grid 1 be a grid with the size of the smallest square being 1, Grid 2 with size of the smallest square being α_1, Grid 3 with size of the smallest square being α_2, and Grid 4 being α_3, where $2 > \alpha_3 > \alpha_2 > \alpha_1 > 1$.

There are 4 independent square hierarchies in this case. The goal is to determine $\alpha_1, \alpha_2, \alpha_3$ such that the worst case ratios do not change when we move between the grids.

Using the same technique as in Section 2.5.1, it can be shown that $\alpha_1 = 2^{\frac{1}{4}}$, $\alpha_2 = 2^{\frac{1}{2}}$, and $\alpha_3 = 2^{\frac{3}{4}}$ for the four approaches considered. There are n squares of size 1, $\frac{n}{\sqrt{2}}$ squares of size α_1, $\frac{n}{2}$ squares of size α_2 and $\frac{n}{2\sqrt{2}}$ squares of size α_3.

The worst case ratios and the maximum number of bits needed to represent the hierarchies with the *implicit* approach for each of the four approaches are listed in the following table.

Table 2.2: SPACE REQUIREMENT AND THE WORST CASE RATIO WITH 4 GRIDS

technique	*maximum # of bits*	*worst case ratio*
squares only	$\approx 13.66n$	$2^{\frac{7}{4}} \approx 3.36$
squares with overlapping	$\approx 13.66n$	$\frac{3}{2} * 2^{\frac{3}{4}} \approx 2.52$
squares with diamonds	$\approx 54.63n$	$2 * 2^{\frac{1}{4}} \approx 2.38$
squares with overlapping and diamonds	$\approx 54.63n$	$\frac{3}{2} * 2^{\frac{1}{4}} \approx 1.78$

2.5.3 m Grids

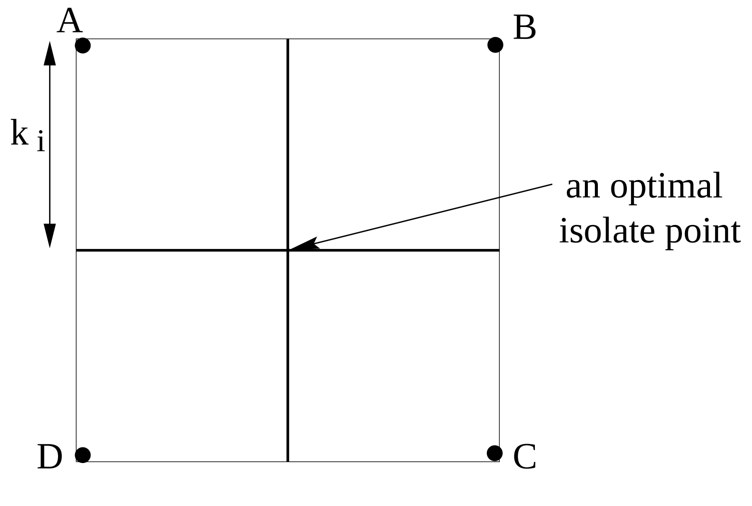
Let Grid 1 be a grid with the size of the smallest square being 1, and Grid i be a grid with size of the smallest square being α_{i-1}, where $i = 2, 3, \ldots, m$, $2 > \alpha_{m-1}$;

$\alpha_i > \alpha_{i-1}$, and $\alpha_1 > 1$.

Squares Only Approach

Without loss of generality, assume all the squares with size greater than or equal to 2 are occupied. The midpoint in a square of size α_{m-1} is returned by *isolate* with the squares only approach. Using the same technique as in Section 2.2.5, the worst case ratio is equal to $2\sqrt{2}/\frac{\alpha_{m-1}}{2} = \frac{4\sqrt{2}}{\alpha_{m-1}}$. For $i = 2, 3, \ldots, m-1$, if all the squares with size greater than or equal to α_i are occupied, the midpoint of a square of size α_{i-1} will be returned. So, the worst case ratio is $\alpha_i\sqrt{2}/\frac{\alpha_{i-1}}{2} = \frac{2\sqrt{2}\alpha_i}{\alpha_{i-1}}$. See Figure 2.16. Finally, if all the size α_1 squares are occupied, the midpoint of a size 1 square will be returned and the worst case ratio becomes $\alpha_1\sqrt{2}/\frac{1}{2} = 2\sqrt{2}\alpha_1$. Equating the above worst case ratios, it can be verified that $\alpha_i = 2^{i/m}$. Hence the worst case ratio is $2\sqrt{2} * 2^{1/m}$.

Figure 2.16: Multiple grids with the squares only approach



As the number of grids tends to infinity, the lower bound of the worst case ratio for this approach is $2\sqrt{2}$. However, the number of bits needed to represent the hierarchies would also increase to infinity in order to achieve the $2\sqrt{2}$ lower bound.

Theorem 2.6 *The lower bound of the worst case ratio of the squares only approach with multiple grids is* $2\sqrt{2}$.

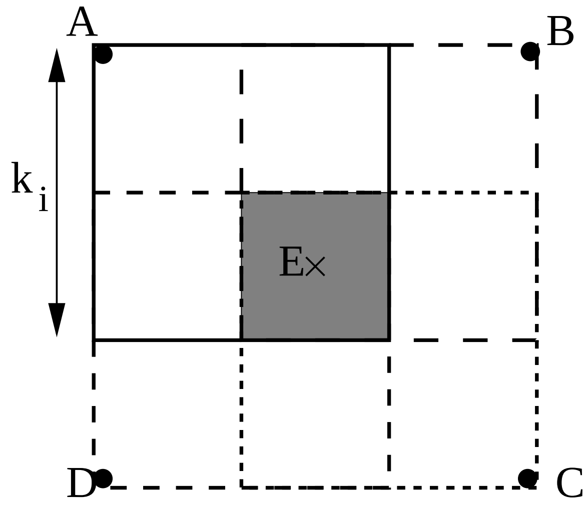
Squares Approach With Overlapping

Similar to the approach in the preceding section, the worst case ratio, when all the overlapping and non-overlapping squares with size greater than or equal to 2 are occupied and the midpoint in a size α_{m-1} square is returned, is equal to $\frac{3}{2}\sqrt{2}/\frac{\alpha_{m-1}}{2} = \frac{3\sqrt{2}}{\alpha_{m-1}}$. For $i = 2, 3, \ldots, m-1$, the worst case ratio, when all the squares with size greater than or equal to α_i are occupied and the midpoint of a square of size α_{i-1} is returned, is $\frac{3}{4}\alpha_i\sqrt{2}/\frac{\alpha_{i-1}}{2} = \frac{3\sqrt{2}\alpha_i}{2\alpha_{i-1}}$. See Figure 2.17. Finally, if all the size α_1 squares are occupied, the midpoint of a size 1 square will be returned and the worst case ratio becomes $\frac{3}{4}\alpha_1\sqrt{2}/\frac{1}{2} = \frac{3}{2}\sqrt{2}\alpha_1$. Equating the above worst case ratios, it can be verified that $\alpha_i = 2^{i/m}$. Hence the worst case ratio is $\frac{3}{2}\sqrt{2} * 2^{1/m}$.

Therefore, the worst case ratio approaches $\frac{3}{2}\sqrt{2}$ as m tends to infinity and the space required would also tend to infinity.

Theorem 2.7 *The lower bound of the worst case ratio of the square approach with overlapping and multiple grids is* $\frac{3}{2}\sqrt{2}$.

Figure 2.17: Multiple grids with overlapping

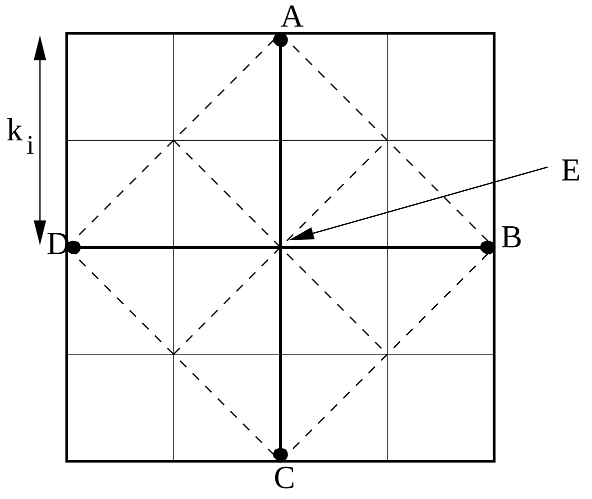


Squares With Diamonds

Similar to the preceding sections, the worst case ratio, when all the squares with size greater than or equal to 2 and their associated diamonds are occupied and the midpoint of a size α_{m-1} square is returned, is equal to $2/\frac{\alpha_{m-1}}{2} = \frac{4}{\alpha_{m-1}}$. For $i = 2, 3, \ldots, m-1$, the worst case ratio, when all the squares with size greater than or equal to α_i and their associated diamonds are occupied and the midpoint of a square of size α_{i-1} is returned, is $\alpha_i/\frac{\alpha_{i-1}}{2} = \frac{2\alpha_i}{\alpha_{i-1}}$. See Figure 2.18. Finally, if all the size α_1 squares and the associated diamonds are occupied, the midpoint of a size x square will be returned and the worst case ratio becomes $\alpha_1/\frac{x}{2} = 2\alpha_1$. Equating the above worst case ratios, it can be verified that $\alpha_i = 2^{i/m}$. Hence the worst case ratio is $2 * 2^{1/m}$.

Therefore, the worst case ratio approaches 2 as m tends to infinity and the number of bits required to represent an infinite number of hierarchies would also

Figure 2.18: Multiple grids with diamonds



.

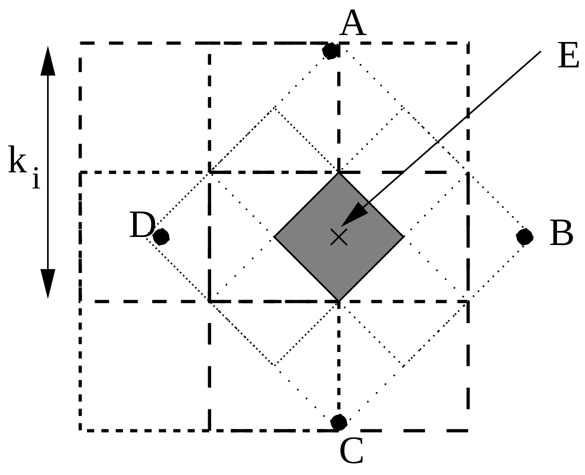
tend to infinity.

Theorem 2.8 *The lower bound of the worst case ratio of the square approach with diamonds and multiple grids is 2.*

Squares With Overlapping and Diamonds

Similarly, the worst case ratio, when all the squares (including the overlapping ones) with size greater than or equal to 2 and their associated diamonds are occupied and the midpoint of a size α_{m-1} square is returned, is equal to $\frac{3}{2}/\frac{\alpha_{m-1}}{2} = \frac{3}{\alpha_{m-1}}$. For $i = 2, 3, \ldots, m-1$, the worst case ratio, when all the squares with size greater than or equal to α_i are occupied and the midpoint of a square of size α_{i-1} is returned, is $\frac{3}{4}\alpha_i/\frac{\alpha_{i-1}}{2} = \frac{3\alpha_i}{2\alpha_{i-1}}$. See Figure 2.19. Finally, if all the size α_1 squares and the associated diamonds are occupied, the midpoint of a size 1 square will be returned and the worst case ratio becomes $\frac{3}{4}\alpha_1/\frac{1}{2} = \frac{3}{2}\alpha_1$. Equating the above worst case ratios, it can be verified that $\alpha_i = 2^{i/m}$. Hence the worst case ratio is $\frac{3}{2}2^{1/m}$.

Figure 2.19: MULTIPLE GRIDS WITH OVERLAPPING AND DIAMONDS



Hence, the worst case ratio approaches $\frac{3}{2}$ as m tends towards infinity and the number of bits required to represent an infinite number of grids would also tend to infinity.

Theorem 2.9 *The lower bound of the worst case ratio of the square approach with overlapping, diamonds and multiple grids is* $\frac{3}{2}$.

Chapter 3

Dividing the Domain into Hexagons

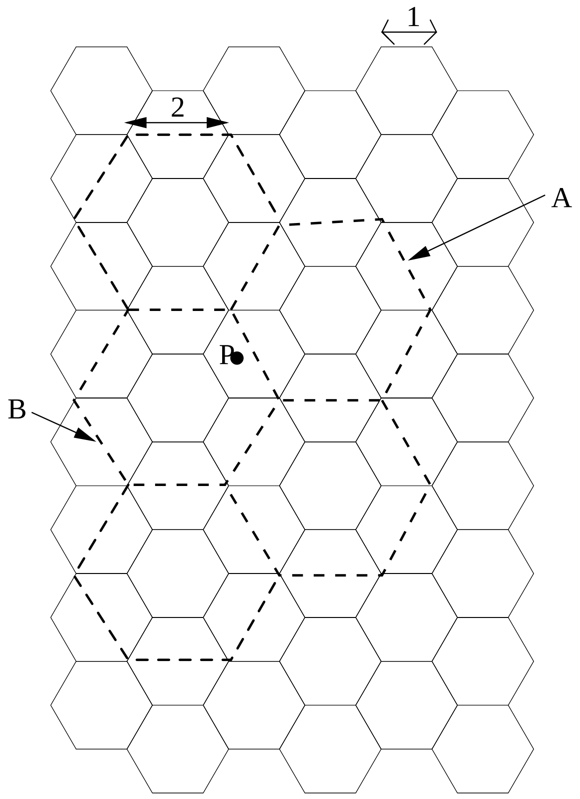
The approach of dividing the domain into a hierarchy of squares and returning the midpoint of one of the largest unoccupied squares using *isolate* can be modified to improve the worst case ratio while attaining our goal of constant amortized running time per operation and linear space.

An alternative to dividing the domain into squares is to tile the plane with regular hexagons, similar to a honeycomb. Regular hexagons are promising because they are closer to circles than are squares, but they still tile the plane. In Section 3.1, we look at a naive approach which gives a worst case and amortized running time of $\Theta(\log n)$ per operation. In Section 3.2, the algorithm giving constant amortized running time per operation will be discussed.

3.1 The Naive Approach

The problem of tiling the plane with regular hexagons lies in grouping several hexagons to form a larger hexagon in the hierarchy. A hexagon in the middle with its six neighbours do not form a larger hexagon. Instead, a double edge-lengthed regular hexagon can be formed by "cutting" the neighbouring six unit sized hexagons into halves as illustrated in Figure 3.1. These half hexagons are *isosceles trapezoids.*

Figure 3.1: GROUPING OF HEXAGONS TO FORM A DOUBLE EDGE-LENGTHED HEXAGON



Each side of the dotted-line regular hexagons in Figure 3.1 has length double that of the small solid-line hexagons. The area of a dotted-line hexagon is four times that of a solid-line small hexagon. Therefore the number of hexagons, and so bits in the hierarchy is reduced by a factor of four each level up the hierarchy.

For simplicity we assume that the smallest hexagons at the bottommost level of the hexagon hierarchy have unit size, and we call them *base hexagons*. The lowest level in the hierarchy is called level 0.

The major difference between the grouping of hexagons and that of squares to form a hierarchy is that squares are "cleaner" in the sense that the boundary of four unit-sized squares forms a double-sized square. But we have to "cut" the neighbouring unit-sized hexagons into halves to form a double edge-lengthed hexagon. Therefore, when we go up one level in the hexagon hierarchy, a naive approach requires inspecting all of the inserted points to decide in which double edge-lengthed hexagon the point lies. For example, in Figure 3.1, we have to consider point P at level 1 again to decide whether it lies in the double edge-lengthed hexagons A or B. If we have to look at each of the n points already inserted in each of the $\lceil \log_4(n+1) \rceil$ levels to build the hexagon hierarchy, the worst case running time for n insert and isolate operations is $n\lceil \log_4(n+1) \rceil$. Thus the amortized running time per operation is $\Theta(\log n)$ using this naive approach.

3.2 A Refined Approach

In order to guarantee constant amortized running time, we need the following lemmas.

Lemma 3.1 *In constructing a hierarchy of hexagons, as outlined in Figure 3.1, each hexagon of the base size is bisected at most once by sides of larger hexagons. Indeed if the entire plane is tiled, rather than only a finite region, each base hexagon would be bisected precisely once.*

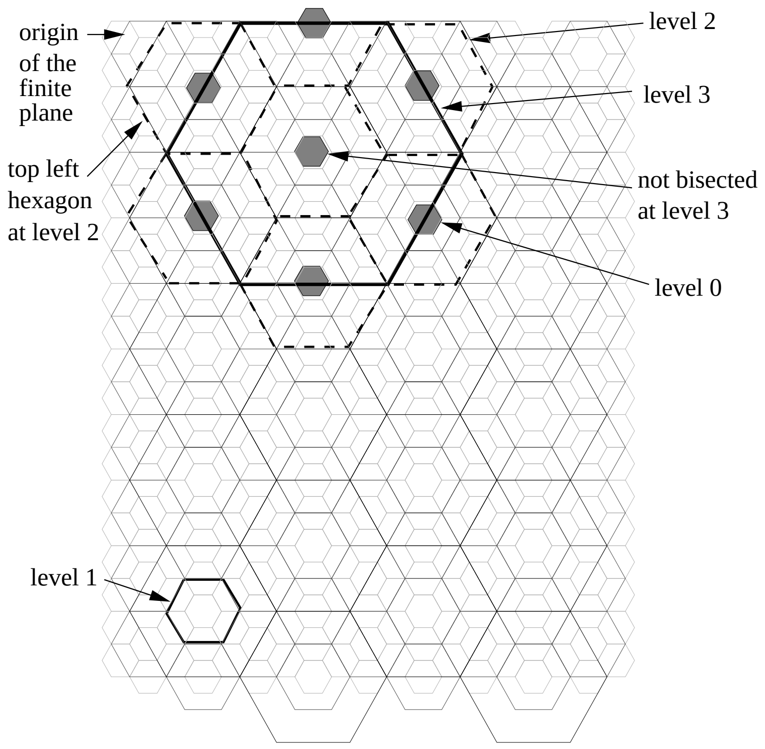
Proof of Lemma 3.1: The proof is by induction on the number of levels that have already been built in the hierarchy. The inductive hypothesis is as follows: at level k, all base hexagons except the centres of level k hexagons have been bisected once and the centres of level k hexagons have not been bisected. Furthermore, all three pairs of opposite vertices of the hexagons up to level k are connected by a (straight) sequence of base-hexagon bisectors, broken only by the lack of any bisector in the middle base hexagon. We call these sequences of base-hexagon bisectors *incomplete diagonals.*

The Basis Case: At level 0, we have only the base hexagons, so the inductive hypothesis is vacuously true. The situation at level 1 is illustrated in Figure 3.1, so the inductive hypothesis is seen to be true at level 1 as well.

The Induction Step: Assume that the inductive hypothesis is true up to and including level k. In the case of a finite plane, take the top left base hexagon as the origin of the plane and start building hexagons from the top left corner of the plane, so that the top leftmost hexagon at the current level is at the upper left position of the hexagon in the next level, as in Figure 3.2. When level $k+1$ is being built, six level k hexagons together with a middle one at the same level will form a hexagon at level $k+1$ by bisecting the middle base hexagons of the six level k hexagons. These bisectors complete the sides of the level $k+1$ hexagon. The portions of edges of the outer level k hexagons which adjoin each other and are now inside the level $k+1$ hexagon, together with the three incomplete diagonals of the inner level k hexagon, produce the three incomplete diagonals of the level $k+1$ hexagon. See Figure 3.2. The base hexagon in the centre of the middle level k hexagon will become the centre of the level $k+1$ hexagon and is not bisected.

If the plane is infinite, there will be an infinite number of levels of hexagons, and so all base hexagons will be bisected exactly once. ■

Figure 3.2: Illustration of the inductive step with $k = 2$



Before stating further lemmas, some numbering conventions and terminology will be defined.

Without loss of generality, assume the given finite domain is a square. The given domain is tiled with regular hexagons as in honeycombs. We are going to look at the base hexagons in the hexagon hierarchy in terms of columns and rows. Two adjacent columns in the honeycomb tiling do not lie side by side; they are shifted vertically by half a hexagon. See Figure 3.3. Without loss of generality, we are going to assume the following boundary conditions: the first column starts with the first column of intact base hexagons and the first row starts with a complete base hexagon and alternates with half hexagons as in Figure 3.3. Since we are dealing with a finite domain, we shall assume the base hexagon at position column 1 and row 1 to be the origin of the domain. As in the proof of Lemma 3.1, we start building hexagons from the origin of the domain, and the top left hexagon at the current level is at the upper left position of the hexagon in the next level up. See Figure 3.3.

Define the orientation of hexagons such that there are two horizontal sides. Thus, there are three distinct directions in which a hexagon can be bisected. We refer to these three bisectors as *horizontal cut*, *right slash* and *left slash* as shown in Figure 3.4. Two consecutive slashes form a pair of *matching* slashes in a column of base hexagons if one of them is left and the other one is right.

The *vertical gap* of a hexagon at a particular level is defined to be the difference in row numbers of base hexagons in the same column between corresponding positions in consecutive hexagons at that level. For example, the vertical gap of a level 1 hexagon is 2 as shown in Figure 3.3.

Figure 3.3: COLUMNS AND ROWS IN THE HONEYCOMB TILING

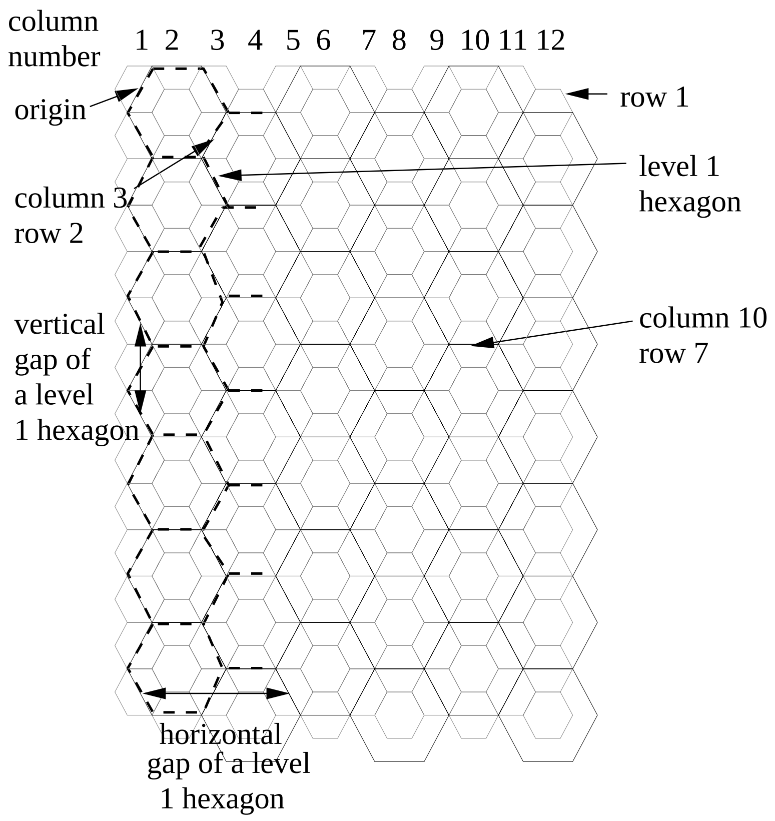
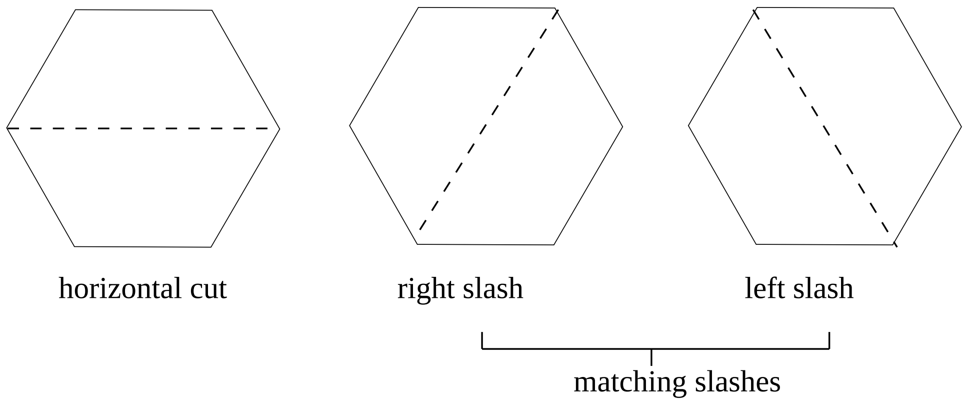


Figure 3.4: THE THREE DIRECTIONS OF CUTS



Similarly, the *horizontal gap* of a hexagon at a particular level is defined to be the difference in column numbers of base hexagons in the same row between corresponding positions in consecutive hexagons at that level. For example, the horizontal gap of a level 1 hexagon is 4 as shown in Figure 3.3.

Lemma 3.2 *The vertical gap for level k hexagons is 2^k and the horizontal gap for level k hexagons is 2^{k+1}, for $k = 1, 2, \ldots, \lceil \log_4(n+1) \rceil$, where n is the number of points already inserted in the domain.*

Proof of Lemma 3.2: The proof is by induction on the number of levels in the hexagon hierarchy.

The Basis Case: The vertical gap and horizontal gap for a level 1 hexagon are 2 and 4 respectively. See Figure 3.3. Hence, the lemma is true at level 1.

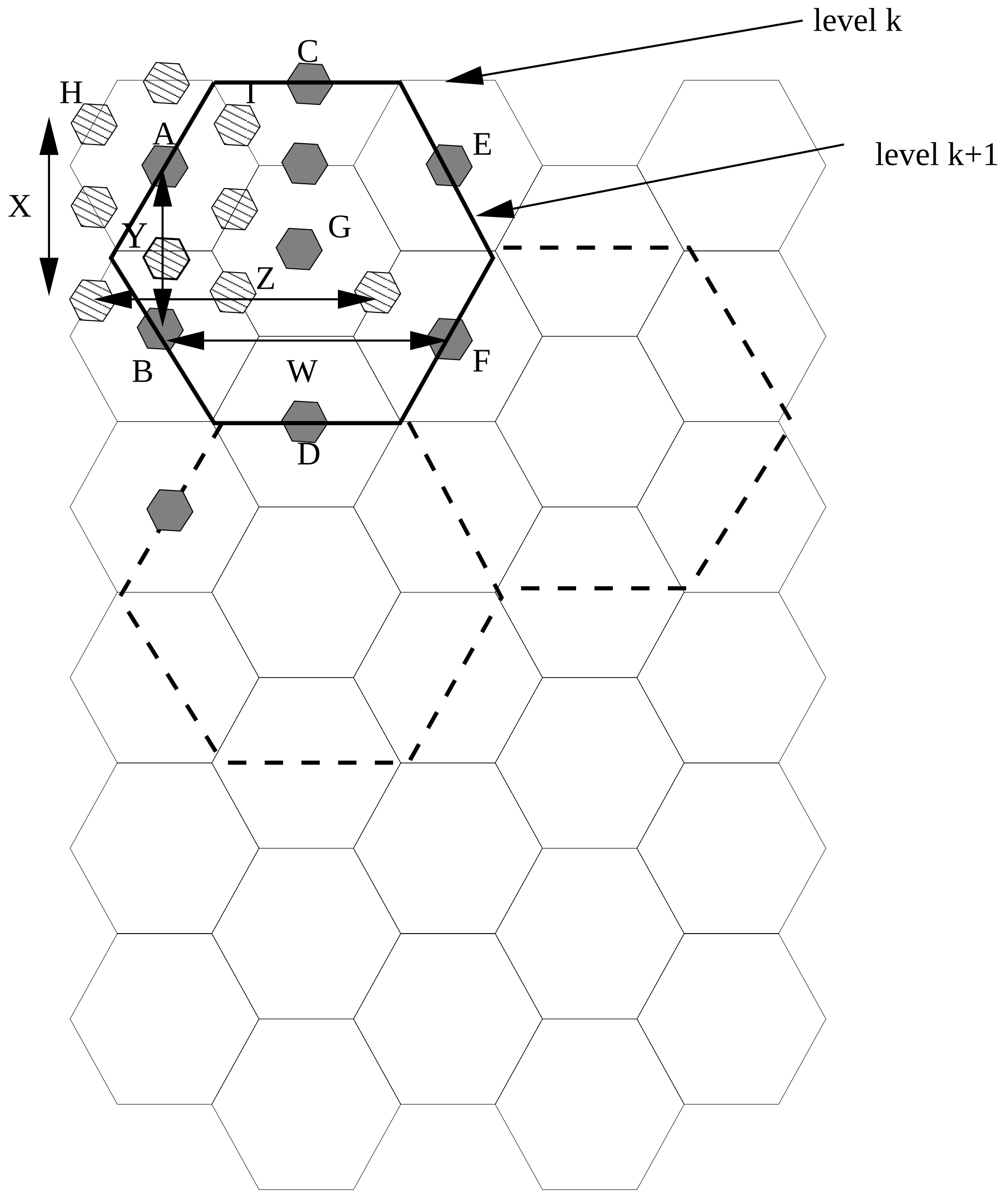
The Induction Step: Assume the vertical gap is 2^k and the horizontal gap is 2^{k+1} for level k hexagons. Now let us consider hexagons at level $k+1$. By definition, X is the vertical gap of level k hexagons and X is equal to 2^k by the induction hypothesis. See Figure 3.5. Similarly, Z is the horizontal gap of level k hexagons by definition and is equal to 2^{k+1}. By construction, vertical gaps X and Y are equal and horizontal gaps Z and W are equal. The vertical gap of level $k+1$ hexagons is equal to twice Y by definition, and so is $2(2^k) = 2^{k+1}$. Similarly, the horizontal gap of level $k+1$ hexagons is equal to twice W, and so is equal to $2(2^{k+1}) = 2^{k+2}$. Therefore, the induction hypothesis is also true at level $k+1$. ■

Lemma 3.1 proved that each base hexagon is to be cut into two halves, and Lemma 3.2 showed that there is a regular pattern for how the base hexagons are

Figure 3.5: Illustration of the inductive step in the proof of Lemma 3.2

▨ is a base hexagon in the centre of a level k-1 hexagon

⬢ is a base hexagon in the centre of a level k hexagon



bisected. The following lemmas are going to show there exists a linear time algorithm that determines how the base hexagons are bisected. The algorithm is needed to interpolate points into the proper half of the base hexagon so that the hexagon hierarchy can be built without looking at the inserted points.

Lemma 3.3 *The following algorithm gives the directions of bisectors of base hexagons in the centres of level $k-1$ hexagons when level k hexagons are being built. Suppose the current base hexagon has column number* column *and row number* row.
For each level k, where $k = 1, 2, \ldots, \lceil \log_4(n+1) \rceil$:

1. *offset $= 2^{k-2}$ if $k \geq 2$. Otherwise, offset $= 0$.*

2. *If* column $\bmod 2^{k+1} = 2^{k-1}$

    (a) *if (*row *- offset - 1)* $\bmod 2^k = 0$ *then assign a right slash to the base hexagon*

    (b) *else if (row - offset - 1)* $\bmod 2^k = 2^{k-1}$ *then assign a left slash to the base hexagon*

3. *else if column* $\bmod 2^{k+1} = 3 * 2^{k-1}$

    (a) *if (row - offset - 1)* $\bmod 2^k = 0$ *then assign a left slash to the base hexagon*

    (b) *else if (row - offset - 1)* $\bmod 2^k = 2^{k-1}$ *then assign a right slash to the base hexagon*

4. *else if column* $\bmod 2^{k+1} = 2^k$

```
    • if (row - 1) $\bmod 2^k = 0$ then assign a horizontal slash to the base hexagon
5. else if column $\bmod 2^{k+1} = 0$
    (a) if $k \geq 2$ then
        • if (row - 1) $\bmod 2^k = 2^{k-1}$ then assign a horizontal slash to the base hexagon
    (b) else ($k = 1$)
        • if row $\bmod 2^k = 0$ then assign a horizontal slash to the base hexagon
```

Proof of Lemma 3.3: In Lemma 3.2, we have shown that the horizontal gap for level k hexagons is 2^{k+1}. Hence, the pattern of bisectors repeats every 2^{k+1} columns. This explains why the column number is taken modulo 2^{k+1} in the above algorithm. The proof is by induction on the number of levels that have been built. The induction hypothesis is as follows: when level k hexagons are being built, the above algorithm gives the directions of bisectors of the base hexagons in the centres of the level $k-1$ hexagons, except for those which are also in the centres of the level k hexagons.

The Basis Case: At level 1, we have the situation as shown in Figure 3.3. The offset is 0. The dotted lines in Figure 3.3 gives the directions of bisectors of the base hexagons in columns 1, 2, 3 and 4. Hence, the above algorithm works for level 1.

The Induction Step: Assume hexagons up to level k have been built. Let us consider the scenario when level $k+1$ hexagons are being built. Seven level k

hexagons with one of them at the centre form a level $k+1$ hexagon. See Figure 3.5. Let base hexagons A, B, C, D, E and F be the centres of the six outer level k hexagons and let G be that of the middle level k hexagon. By construction, base hexagon G is also the centre of the level $k+1$ hexagon. The stripped base hexagons are in the centres of level $k-1$ hexagons. By the induction hypothesis, the algorithm gives the directions of bisectors of the stripped base hexagons when level k hexagons are built. By the algorithm, stripped base hexagon H has column number 2^{k-1} and row number 2^{k-2} and stripped base hexagon I has column number $3(2^{k-1})$ and row number 2^{k-2}. Hence, one of the centre base hexagons of level k hexagons should be in column 2^k. By construction, the offset of base hexagon A should be double that of the stripped base hexagons H and I. Hence, base hexagon A in Figure 3.5 is covered by case 2(a) in the above algorithm. By Lemma 3.2, the vertical gap of a level k hexagon is 2^k. By construction, the number of rows between base hexagon B and base hexagon A is equal to the vertical gap of a level k hexagon which is covered in case 2(b) in the algorithm. By Lemma 3.2, the horizontal gap of a level $k+1$ hexagon is 2^{k+2}. The number of columns between base hexagons A and E is half of the horizontal gap which is 2^{k+1}. Similarly, base hexagons E and F are covered by case 3(a) and 3(b) respectively. By Lemma 3.2, the vertical gap of a level $k+1$ hexagon is 2^{k+1}. Hence, base hexagons C and D are assigned a horizontal slash in case 4 in the algorithm. Therefore, the induction hypothesis is also true at level $k+1$. ■

Lemma 3.4 *The following algorithm is equivalent to Lemma 3.3 and gives the direction of bisector of a base hexagon with column number* column *and row number* row*. Assume the least significant bit of any integer is at position zero.*

```
1. k = (number of 0 bits before the first 1 bit in the binary representation of column) +1
2. Extract bits in positions $k$ and $k-1$ in column and denote the two bits by $c$.
3. If $k \geq 2$, offset = $2^{k-2}$. Otherwise, offset = 0.
4. If row $\leq$ offset then
      • assign a horizontal slash to the base hexagon
5. else
      (a) Extract bits in positions $k-1$ and $k-2$ in (row - offset -1) and denote the two bits by $r$. If $k = 1$ and row is odd, set $r = 0$. Otherwise, if $k = 1$ and row is even, set $r = 2$.
      (b) If $k \geq 3$ and bits in positions $0, 1, \ldots, k-3$ in the binary representation of (row - offset -1) are not all zeros then
            • assign a horizontal slash to the base hexagon
      (c) else if $(c + r) \bmod 4 = 1$ then
            • assign a right slash to the base hexagon
      (d) else if $(c + r) \bmod 4 = 3$ then
            • assign a left slash to the base hexagon
      (e) else
            • assign a horizontal slash to the base hexagon
```

Proof of Lemma 3.4: In the algorithm of Lemma 3.3, a base hexagon is assigned a left or right slash or a horizontal cut if it is the centre of a hexagon at a particular level below.

The conditions in Lemma 3.3 can be rewritten as the following :

2. (column mod $2^{k+1})/2^{k-1} = 1$
3. (column mod $2^{k+1})/2^{k-1} = 3$
4. (column mod $2^{k+1})/2^{k-1} = 2$
5. (column mod $2^{k+1})/2^{k-1} = 0$

The operation (column mod 2^{k+1}) / 2^{k-1} is essentially extracting bits in positions k and $k-1$ in column. So, c in the current algorithm corresponds to the quantity (column mod 2^{k+1}) / 2^{k-1} in Lemma 3.3. Left and right slashes are only assigned in Steps 2 and 3 of the algorithm in Lemma 3.3 such that the quantity c in the current algorithm is odd, hence the bit at position $k-1$ is a 1-bit. In the algorithm in Lemma 3.3, k increases from 1 to $\lceil \log_4(n+1) \rceil$, and so k is equal to the number of 0 bits before the first 1 bit in *column* plus one. Therefore, k can be uniquely determined from a given column number.

Similarly, extracting bits in positions $k-1$ and $k-2$ from (row - offset - 1) is the same as (row - offset - 1) mod 2^k / 2^{k-2}. So, r in the current algorithm corresponds to the quantity (row -offset - 1) mod 2^k / 2^{k-2} in Lemma 3.3. In Step 2 of Lemma 3.3, a right slash is assigned if (row - offset - 1) mod 2^k / $2^{k-2} = 0$. Observe that a right slash is assigned in both Steps 2(a) and 3(b). In Step 2(a), $c = 1$ and $r = 0$ and in Step 3(b), $c = 3$ and $r = 2$ in terms of the current algorithm.

Hence, $c + r \bmod 4 = 1$ in both cases. Similarly, a left slash is assigned in both Steps 2(b) and 3(a). In Step 2(b), $c = 1$ and $r = 2$, and in Step 3(a), $c = 3$ and $r = 0$. Hence, $c + r \bmod 4 = 3$ in both cases. Therefore, in the current algorithm, a right slash is assigned in Step 5(c) and a left slash is assigned in Step 5(d).

The idea is then to assign horizontal slashes to all other base hexagons between the right and left slashes in a column. Step 5(e) covers the case when $c = 1$ or $c = 3$, but r is neither 0 nor 2.

Step 5(b) covers the case when $c = 0$ or $c = 2$ which is covered in Steps 4 and 5 in Lemma 3.3. A horizontal slash is assigned when the integer (row - offset - 1) is a multiple of twice the vertical gap of level k. ■

Lemma 3.5 *The running time of the algorithm in Lemma 3.4 is $O(1)$ per base hexagon.*

Proof of Lemma 3.5: Bit extractions can be done in constant time by using a mask. Constants in the form 2^m can be obtained by shifting the constant 1 by m bits to the left. Since we have assumed arbitrary bit shifts to be constant time operations in our model of computation, constants in the form 2^m can be obtained in constant time. To extract bits in positions k and $k-1$ in *column* (Step 2 of the algorithm in Lemma 3.4), a mask $M = 2^k + 2^{k-1}$, can be used. Shifting the number (column AND M) right by $(k-1)$ positions extracts the required bits. Similarly, extracting bits in positions $k-1$ and $k-2$ in *(row - offset -1)* in Step 5(a) can be done by shifting the number ((row - offset -1) AND $(2^{k-1} + 2^{k-2})$) right by $(k-2)$ bits. Checking whether the bits in positions $0, 1, \ldots, k-3$ in *(row - offset -1)* are not all zeros in Step 5(b) is equivalent to checking ((row - offset -1) AND $(2^{k-2}-1)$) $\neq 0$.

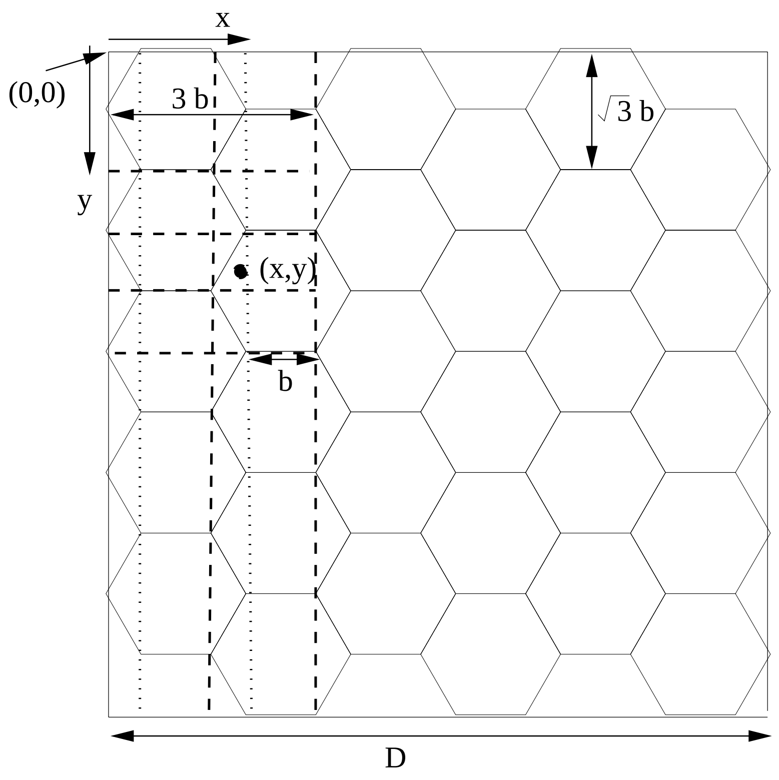
Since we have assumed that finding the number of 0 bits before the first 1 bit in an integer is a constant time operation, all the operations in the algorithm in Lemma 3.4 are constant time operations. Therefore, the algorithm has a constant running time. ■

Lemma 3.6 *Tiling a square domain with* m *regular hexagons requires approximately* $p\sqrt{m}$ *columns and* $q\sqrt{m}$ *rows of base hexagons, where* $p = 1.075\ldots$, $q = .931\ldots$.

Proof of Lemma 3.6: Let D be the dimension of the square domain and b be the length of a side of a base hexagon. Suppose we require R rows and C columns of base hexagons to tile the square domain. For example, in Figure 3.6, $R = 5.5$ and $C = 6$. The height of a size b hexagon is $\sqrt{3}b$. Solving $\sqrt{3}bR = D$ and $\frac{3}{2}bC = D$ with $RC = m$ gives $C = \frac{\sqrt{2}}{3^{1/4}}\sqrt{m} \approx 1.075\sqrt{m}$ and $R = \frac{3^{1/4}}{\sqrt{2}}\sqrt{m} \approx 0.931\sqrt{m}$. The size of the base hexagons is $D/(\sqrt{3}\frac{3^{1/4}}{\sqrt{2}}\sqrt{m}) = \sqrt{2}D/(3^{3/4}\sqrt{m}) \approx 0.620\frac{D}{\sqrt{m}}$. ■

Lemma 3.7 *Given a point with coordinates (x,y), the base hexagon in which the point lies can be determined in constant time. Assume the upper left corner of the square domain has coordinates (0,0), and the x-coordinates increase from left to right and the y-coordinates increase from top to bottom. In the following algorithm, assume* b *is the size of the base hexagon;* b *can be determined by the method in the proof of Lemma 3.6. The base hexagon in which (x,y) lies has column number* column *and row number* row.

Figure 3.6: Illustration of Lemma 3.6



```
1. $c = \lceil x/\frac{3}{2}b \rceil$
2. $extra\_x = x - (c-1) * \frac{3}{2}b$
3. $half\_row = \lceil y/\frac{\sqrt{3}}{2}b \rceil$
4. If $extra\_x \geq \frac{b}{2}$ then
     • column = c
5. else
   (a) $extra\_y = y - (half\_row - 1) * \frac{\sqrt{3}}{2}b$
   (b) If c is odd and half_row is odd then
         i. if $(\frac{\sqrt{3}}{2}b - extra\_y)/extra\_x > \sqrt{3}$ then
              • column = c - 1
        ii. else
              • column = c
   (c) else if c is odd and half_row is even then
         i. if $extra\_y/extra\_x > -\sqrt{3}$ then
              • column = c
        ii. else
              • column = c - 1
   (d) else if c is even and half_row is odd then
         i. if $(\frac{\sqrt{3}}{2}b - extra\_y)/(\frac{b}{2} - extra\_x) > -\sqrt{3}$ then
```

```
            • column = c - 1
        ii. else
            • column = c
    (e) else if c is even and half_row is even then
        i. if $extra\_y/(\frac{b}{2} - extra\_x) > \sqrt{3}$ then
            • column = c
        ii. else
            • column = c - 1
6. If c is odd then
    • row = $\lceil y/\sqrt{3}b \rceil$
7. else if c is even then
    • row = $\lfloor half\_row/2 \rfloor + 1$
```

Proof of Lemma 3.7: The idea of the algorithm is to divide the domain into columns separated by the vertical dashed lines and into half-rows separated by the horizontal dashed lines in Figure 3.6. *Extra_y* denotes the vertical distance of the point from the closest horizontal dashed line above and *extra_x* is the horizontal distance from the vertical dashed line to the left of the point. See Figure 3.7.

The distance between two vertical dashed lines is $\frac{3}{2}b$ and Step 1 computes the column number c of the point. Note that the column number and the row number start with the first intact column and first intact row of base hexagons being column 1 and row 1, respectively. The vertical distance in a half-row is half of the height of the base hexagon which is $\frac{\sqrt{3}}{2}b$, and Step 3 computes the number of half-rows up to the current point.

Step 4 covers the case when the point is inside the rectangular region of the base hexagon. In Figure 3.7, the rectangular region of the base hexagon in column 1 and row 1 is ECEF.

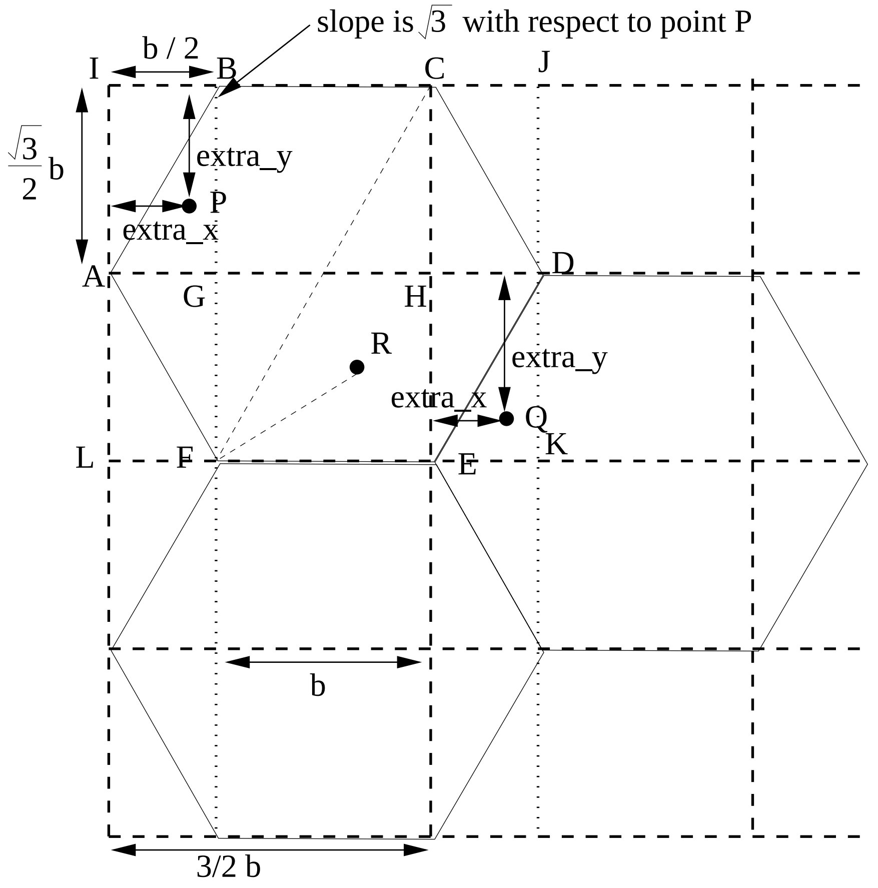
Without loss of generality, the following discussion assumes a point lying in region IJKL in Figure 3.7. Note that $c = 1$ if the point lies in region ICEL, while $c = 2$ if it lies in region CJKE. Step 5(b) covers the case when the point P lies in region IBGA with $c = 1$. The idea is to compute the slope of the line PA which is equal to $(\frac{\sqrt{3}}{2}b - extra\_y)/extra\_x$. If the slope of line PA is greater than that of line AB which is $\sqrt{3}$, the point P must lie in the triangular region IBA and hence the column number of the base hexagon in which P lies should be $c - 1$. On the other hand, if the slope line PA is less than that of line BA, the point P must lie in the triangular region BAG. Similarly, Step 5(c) covers the case when the point lies in region AGFL with $c = 1$, Step 5(d) covers the case when it lies in region CJDH with $c = 2$, and Step 5(e) when the point Q lies in region HDKE with $c = 2$.

All the operations in the algorithm are constant time operations, and so the base hexagon in which a given point lies can be determined in constant time. ■

3.2.1 Putting Things Together

Lemma 3.1 proves that each base hexagon is cut into halves as the hexagon hierarchy is built. Given the number of base hexagons in the bottommost level of the hierarchy, the number of columns and rows of base hexagons in the square domain

Figure 3.7: Illustration of Lemma 3.7

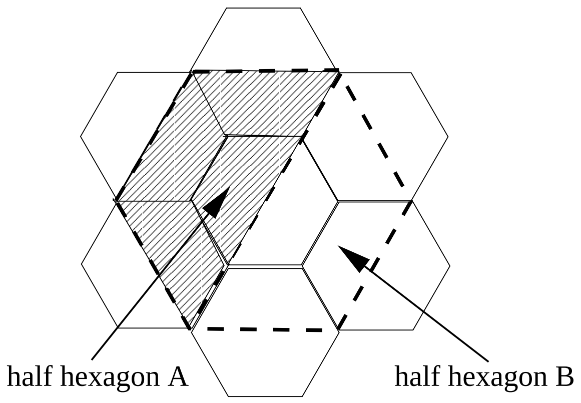


and the size of the base hexagon can be determined by Lemma 3.6 in constant time. The algorithm in Lemma 3.4 can be used as a preprocessing step after decomposition into a finer grid of base hexagons to give the direction of the bisector in each base hexagon. After running the algorithm, each base hexagon is assigned a horizontal slash, right or left slash which represents the way that each hexagon should be divided into isosceles trapezoids. There are $\Theta(n)$ base hexagons and the algorithm in Lemma 3.4 runs in constant time per base hexagon, therefore, the total running time of preprocessing the base hexagons after decomposition is $\Theta(n)$.

Recall the insert algorithm using the square approach in Section 2.2.1. When points are inserted, the unit-size square in which the point lies is determined and then the square hierarchy has to be fixed.

The insert operation for hexagons is similar to that of squares except that squares are replaced by isosceles trapezoids (half-hexagons). Using the algorithm in Lemma 3.7, the base hexagon in which a particular point lies can be determined in constant time. However, in the hexagon approach, we have to keep track of isosceles trapezoids so that we only have to do an OR on the four bits representing the four isosceles trapezoids in the level below and not on the points already inserted when building the hexagon hierarchy. For example, in Figure 3.8, to find out whether isosceles trapezoid A is occupied, we only have to check the four bits representing the shaded isosceles trapezoids in the level below. Therefore, before points can be inserted into the proper unit-size isosceles trapezoid, the algorithm in Lemma 3.4 must be run as a preprocessing step to determine how each base hexagon is to be bisected into base isosceles trapezoids.

Figure 3.8: Illustration of how hexagons are divided into halves



Using a similar approach to that of Lemma 3.7, the base isosceles trapezoid in which a point lies can also be determined in constant time. First of all, the algorithm in Lemma 3.7 is used to determine in which base hexagon the point lies. Without loss of generality, assume the point falls in base hexagon ABCDEF in Figure 3.7. If the point lies in the "narrow" regions IBGA, AGFL, CJDH or HDKE, it is trivial to determine in which base isosceles trapezoid it lies. Now, let us consider the case when the point R lies in the rectangular region BCEF in Figure 3.7. If the base hexagon ABCDEF has a horizontal slash, the point R lies in the upper half of the base hexagon (i.e., the upper isosceles trapezoid ABCD) if the *half_row* number is odd. Otherwise, it lies in the lower half of the base hexagon (i.e., the lower isosceles trapezoid ADEF). If the base hexagon has a right slash, we can compute the slope of line RF. If the slope of line RF is greater than that of line CF, then the point R must lie in the isosceles trapezoid ABCF. Otherwise, it lies in isosceles trapezoid CDEF. Similarly, we can also determine in which isosceles trapezoid the point R lies if the base hexagon has a left slash.

For the isolate operation in the case of hexagons, an additional preprocessing step has to be run after decomposing the domain into 4^k hexagons in the algorithm in Section 2.2.2. The running time of the preprocessing step is $\Theta(n)$ as discussed earlier and interpolating inserted points into a finer grid of hexagons takes $\Theta(n)$ time. Hence, the additional preprocessing does not change the order of the running time of the decomposition and interpolation step. By similar reasoning as in Section 2.2.4, the amortized running time per operation when *inserts* and *isolates* are taken together is constant.

However, the maximum number of bits needed to maintain the bit hierarchy is increased to $\frac{32}{3}n$. The area of a hexagon in the next higher level in the hexagon hierarchy is four times that of the level below, which is the same as that for squares. But we have to keep track of isosceles trapezoids (half-hexagons) rather than hexagons in order to guarantee constant amortized running time, and so each hexagon is represented by two bits. Hence, the number of bits required is double that of the squares only approach.

The worst case ratio using this approach is $8/\sqrt{3}$ which is approximately 4.62, hence is an improvement over the squares only approach. In Figure 3.9, the edge-length of a dashed-line hexagon is twice that of a solid-line hexagon. Let the length of an edge of a solid-line hexagon be 1. Consider the following scenario in which the points are put as far away from each other as possible to determine the maximum distance between any point in the domain and the closest inserted point: all the hexagons with edge-lengths at least 2 are occupied and somewhere in the domain there are three points A, B and C at the corners of 3 dashed-line hexagons with edge-lengths 2. The three points are at a distance of 4 units from point D. Besides, there are no other points inside the circle centred at point D passing through points A, B and C. It can be seen that the circle centred at D with radius 4 is the circle with maximum radius passing through three inserted points and containing hexagons with edge-lengths 2. The maximum possible distance between the most isolated point and an inserted point is 4 units. On the other hand, the midpoint of a size 1 hexagon is returned by *isolate*. The closest possible point to the midpoint of a size 1 hexagon is at a distance $\frac{\sqrt{3}}{2}$ units away. See Figure 3.10. Hence, the worst case ratio is equal to $4/\frac{\sqrt{3}}{2}$ which is $8/\sqrt{3} \approx 4.62$.

Figure 3.9: ILLUSTRATION OF THE HEXAGONS ONLY APPROACH

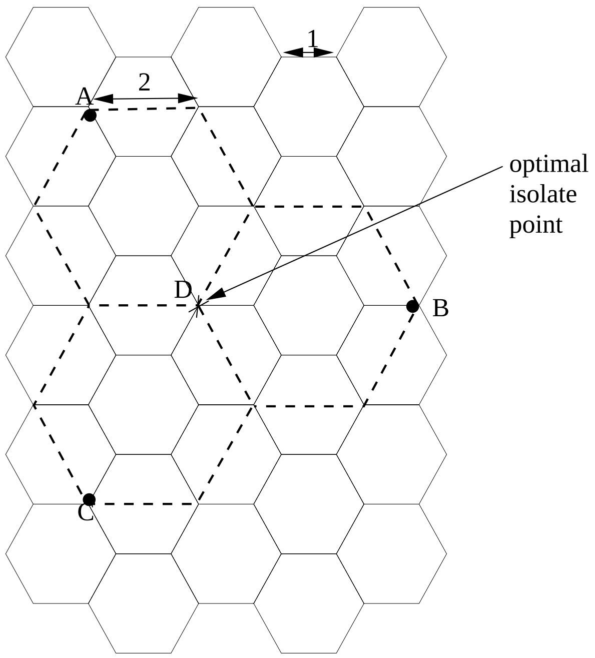
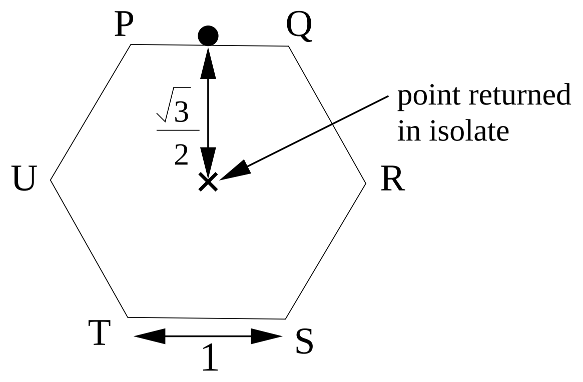


Figure 3.10: A HEXAGON WITH EDGE-LENGTH 1 WITH THE CLOSEST POINT $\frac{\sqrt{3}}{2}$ UNITS AWAY



Theorem 3.1 *Using the half-hexagon approach as described in the preceding sections,* insert *and* isolate *can be performed in constant amortized time per operation, linear space and a worst case ratio of* $8/\sqrt{3}$ (≈ 4.62).

3.2.2 A Correction for the Boundaries

As in the case of squares, a boundary correction convention has to be adopted to decide in which hexagon a point on the boundary of more than one hexagon lies. Assume the upper edges and the two top corner points belong to the hexagon, for example, in Figure 3.10, hexagon PQRSTU consists of edges UP, PQ, and QR, and corner points P and Q. If the side of a hexagon lies on boundary of the domain, the side do not belong to the hexagon so that the hexagon is not occupied unless there are other inserted points in the hexagon. If a hexagon is not complete and is cut by a boundary edge, the hexagon is assumed to be occupied such that it would not be returned by *isolate.*

With the above correction convention, points B and C in Figure 3.9 do not lie on the boundary, but is inside the double edge-lengthed hexagon. Therefore, the actual worst case ratio is slightly less than $\frac{8}{\sqrt{3}}$, but it can be made arbitrarily close to $\frac{8}{\sqrt{3}}$.

3.3 Improving the Worst Case Ratio

As in the case of squares in Section 2.3, the technique described in Section 3.2 can be modified to improve the quality of *isolate*s. There are two general techniques for improving the worst case ratio: overlapping and embedding. The overlapping technique for hexagons is similar to that for squares; additional sets of hexagons are added to the hexagon hierarchy. The embedding technique is similar to the diamond technique for squares; hexagons rotated and of slightly larger size which circumscribes hexagons in the hierarchy are added. With the boundary correction convention in Section 3.2.2, the actual worst case ratios are slightly smaller than those stated in the following lemmas, but they can be made arbitrarily close to the upper bound stated.

3.3.1 Overlapping

Two possible sets of additional overlapping hexagons which do not form hierarchies will be considered. In the *quadruple overlapping* technique, three additional hexagons overlap with each hexagon in each level of the hierarchy, while in the *triple overlapping* approach, there are only two additional overlapping hexagons for each hexagon. It will be shown that quadruple overlapping achieves a better worst case ratio but its space requirement is higher than that of triple overlapping.

In both overlapping techniques, we have to divide each hexagon into six equilateral triangles in the obvious way and to represent each triangle with one bit in order to have constant amortized running time per operation. See Figure 3.11 and Figure 3.13. The reason for keeping track of whether each of the six triangles in a hexagon is occupied or not is the same as in Section 3.2.1. A triangle in a hexagon in the level above is occupied if any of the four smaller triangles making it up are occupied. This can be achieved in constant time by an OR operation on four bits. A hexagon is occupied if any of the six triangles making up the hexagon is occupied. The triangles are only used to determine whether the hexagons are occupied or not. Chapter 4 discusses the case when the triangles are the actual search structures. All the overlapping hexagons in both overlapping techniques can be represented by triangles making up the hexagons one level below in the hierarchy. Therefore, whether an overlapping hexagon is occupied can be determined in constant time with the triangle representation of the hexagons.

The algorithm for *inserts* is the same as that described in Section 3.2.1 except that we keep track of triangles instead of isosceles trapezoids. Using a similar approach in Lemma 3.7, the triangle in which a point lies can be determined in constant time.

For the isolate operation, the algorithm is the same as that in Section 3.2.1 except in determining whether free hexagons exist and in the decomposition step. A free hexagon exists if any of the hexagons (including the overlapping ones) are unoccupied. When there are no free hexagons, the domain is decomposed to 4^k hexagons, that is $6 * 4^k$ triangles. The preprocessing step which determines the way each base hexagon is bisected is *not* needed in the overlapping techniques because all base hexagons are divided in the same way.

The amortized running time per operation, with which *insert*s and *isolate*s are taken together, is constant for the 6-triangle representation of the overlapping approach since in determining whether an overlapping hexagon is occupied, points inserted do not have to be checked.

Quadruple Overlapping

The idea of the quadruple overlapping is that each of the six neighbouring hexagons cut by the next higher level one can become the middle hexagon.

In Figure 3.11, only three of the possible overlappings of next level hexagons are shown because the other three overlappings would not help to improve the worst case ratio. The three additional overlapping hexagons do not form hierarchies. See Figure 3.12. The large dashed hexagon at the next level does not intersect the large solid hexagon in the same way as the small dashed hexagons intersect the small solid hexagons.

As in the square approach with overlapping, there are two alternatives in representing the bit hierarchy. One approach is to use a bit to represent each hexagon including the overlapping ones, and we determine whether an overlapping hexagon is occupied when we build the hierarchy. The second approach is to determine if an overlapping hexagon is occupied on the fly; no extra bits are used to represent the overlapping hexagons.

With the first approach, the number of bits required to represent the hexagon hierarchy is three times that without overlapping since six bits are needed to rep-

Figure 3.11: Improving the worst case ratio with quadruple overlapping

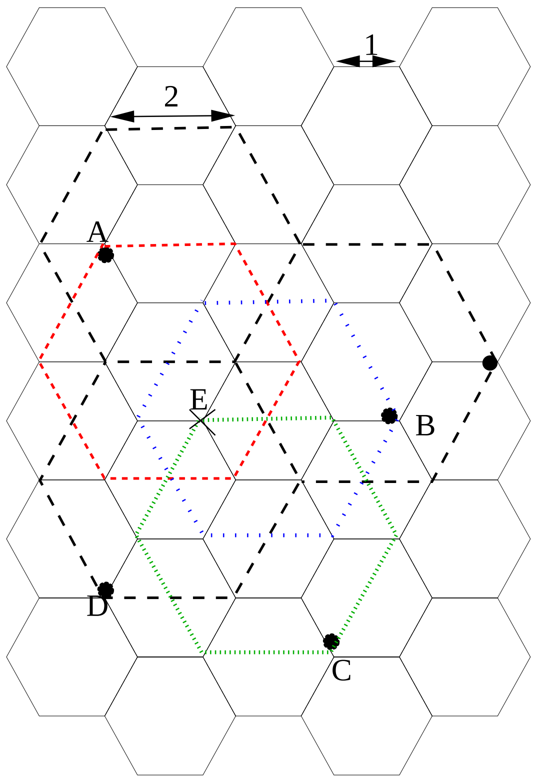
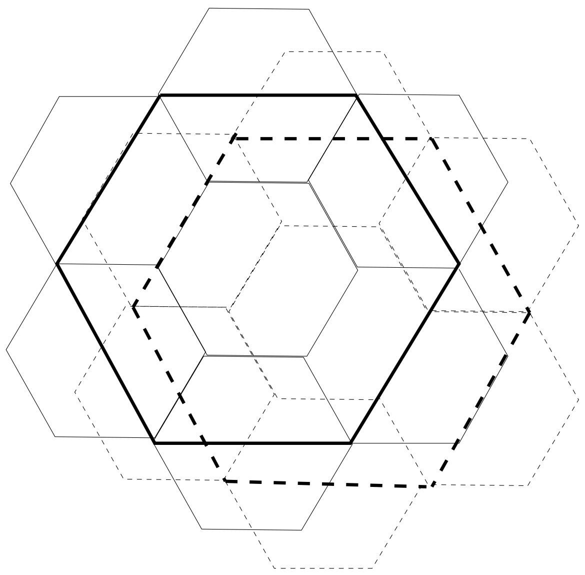


Figure 3.12: Overlapping hexagons do not form hierarchies



resent each hexagon instead of the two bits in Theorem 3.1. So, $\frac{32}{3}n * 3 = 32n$ bits are needed to represent each of the six triangles of a hexagon with one bit. In addition, four bits are required to represent each hexagon in the hierarchy starting at level 1 because of the three extra overlapping hexagons. This adds another $4 * \frac{4}{3}n = \frac{16}{3}n$ bits to the explicit representation. One bit is needed to represent each of the hexagons at level 0. Hence, at most $32n + \frac{16}{3}n + 4n = 41\frac{1}{3}n$ bits are needed for the bit vector hierarchy with the first approach. With the second approach, the maximum number of bits needed is just that required to represent each of the six triangles for a hexagon which is $\frac{32}{3}n * 3 = 32n$.

The worst case ratio is $2\sqrt{3}$ with quadruple overlapping. All the double sized hexagons are represented by non-solid lines in Figure 3.11. Suppose all hexagons (including the overlapping ones) with edge-lengths at least 2 are occupied. Somewhere in the domain, inserted points A, B, C and D occupy the corners of four size 2 hexagons. The circle passing through points A, B and C is centred at point E and the radius of the circle is 3. There are no other points inside the circle. It can be seen that the circle passing through points A, B and C is the circle with maximum radius passing through four hexagons with edge-length 2. So, the most isolated point is point E, and is 3 units away from inserted point A. However, *isolate* returns a size 1 square which has a point $\frac{\sqrt{3}}{2}$ units away from the midpoint. Hence, the worst case ratio is $3/\frac{\sqrt{3}}{2} = 6/\sqrt{3} = 2\sqrt{3}$ and is approximately 3.47.

Theorem 3.2 *The worst case ratio of the hexagon approach with quadruple overlapping is* $2\sqrt{3}$ (≈ 3.47).

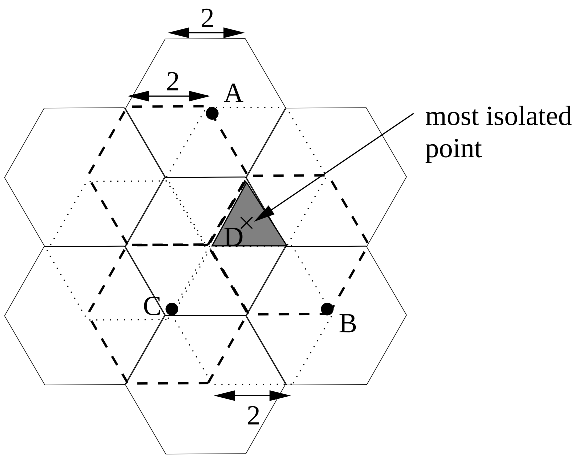
Triple Overlapping

The following overlapping technique saves space but gives a slightly higher worst case ratio than quadruple overlapping. Again, the two additional overlapping hexagons do not form hierarchies.

As in Section 3.3.1, there are two approaches to represent the bit vector hierarchy. One approach is to use a bit to represent each of the two overlapping hexagons. The alternative approach is to find out if an overlapping hexagon is occupied on the fly. As in the quadruple overlapping approach, we need six bits to represent each of the six triangles making up a hexagon in order to preserve constant amortized running time per operation.

With the explicit representation, the number of bits required to represent the hexagon hierarchy is three times that without overlapping, in addition to three

Figure 3.13: IMPROVING THE WORST CASE RATIO WITH TRIPLE OVERLAPPING



extra bits per overlapping hexagon starting from level 1 and one bit per hexagon at level 0. Hence, at most $\frac{32}{3}n * 3 + \frac{4}{3}n * 3 + 4n = 40n$ bits are needed for the bit vector hierarchy. On the other hand, only at most $\frac{32}{3}n * 3 = 32n$ bits are needed with the implicit representation of hexagons.

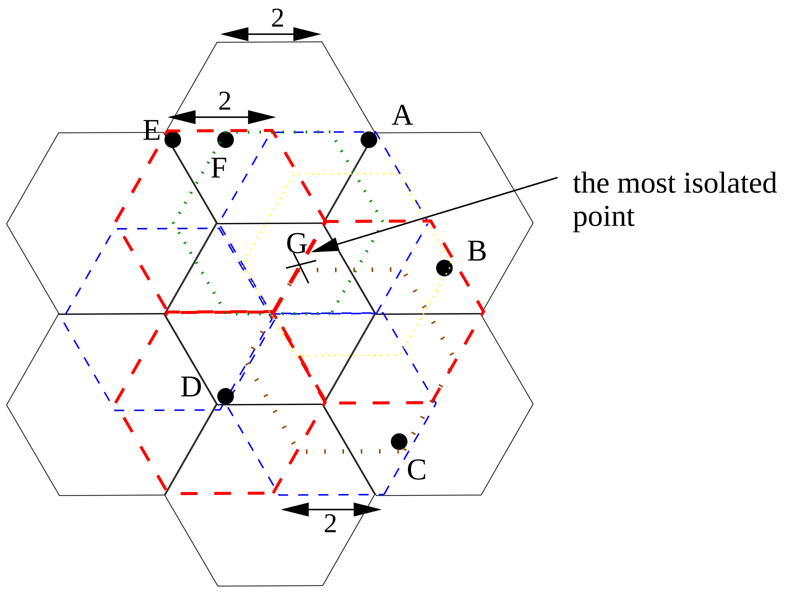
To compute the worst case ratio, imagine all the size 2 non-solid-line hexagons are occupied and there are no points lying in the circle centred at point D with radius DA. The circle centred at point D passing through points A, B and C is the circle with maximum radius containing three size 2 hexagons. Point D is in the middle of the shaded equilateral triangle such that it is equidistant from all three vertices of the triangle. It can be verified that the distance between A and D is $\sqrt{\frac{28}{3}} \approx 3.055$ units. So, the worst case ratio is $\sqrt{\frac{28}{3}}/\frac{\sqrt{3}}{2} \approx 3.53$.

Theorem 3.3 *The worst case ratio of the hexagon approach with triple overlapping is* $\frac{2\sqrt{28}}{3}$ *(*≈ 3.53*).*

Combined Overlapping

The obvious approach of combining the quadruple and triple overlapping does not improve the worst case ratio further than that of quadruple overlapping. Since the combined overlapping would require more space than quadruple overlapping, the combined approach is not worthwhile.

Figure 3.14: Illustration of the worst case ratio with combined overlapping



In Figure 3.14, the distance between A and G, B and G, D and G, and F and G is 3 units. The distance between C and G is 4 units and the distance between E and G is $\sqrt{13}$ units which is greater than 3. The circle centred at G passing through points A, B, D and F is the circle with maximum radius passing through seven size 2 hexagons. Hence, the worst case ratio is $3/\frac{\sqrt{3}}{2} = 2\sqrt{3}$, which is the same as that for quadruple overlapping.

3.3.2 Embedding

Two embedding techniques, in which a rotated hexagon circumscribes another hexagon, will be discussed. One additional level of embedding is used in *double embedding* while four additional levels of embedding are employed in *quintuple embedding*. It will be shown that quintuple embedding achieves a better solution at the expense of a higher space requirement.

Double Embedding

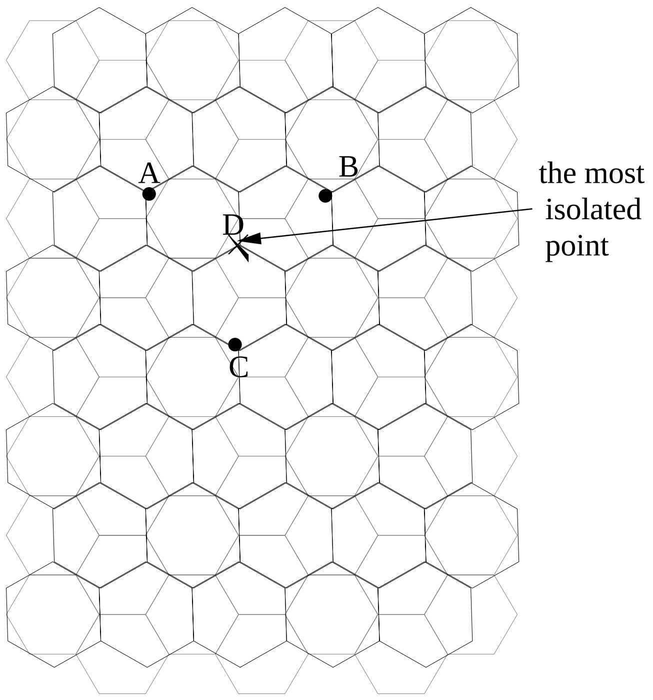
In the double embedding technique, a rotated hexagon of slightly larger size circumscribes each hexagon in the hierarchy. There is a regular pattern for this technique in which hexagons are embedded in other hexagons. See Figure 3.15.

The rotated hexagons form a hierarchy which is independent of the base hexagon hierarchy. Thus, two bit vectors are maintained, one representing the base hexagon hierarchy and the other representing the rotated hexagon hierarchy. To guarantee constant amortized running time per operation, one bit is used to represent each half of the rotated hexagon (rotated isosceles trapezoid) as described in Section 3.2.1.

The insert operation is slightly different from the case without embeddings. Points are inserted into *both* the base hexagons and the rotated base hexagons. Only four isosceles trapezoids of the level below in the respective hierarchy have to be checked to determine whether an isosceles trapezoid at a particular level is occupied or not. In addition, there are two independent hierarchies, both of which have to be updated when a point is inserted.

In the isolate operation, the scan of the bit vectors starts at the topmost level of

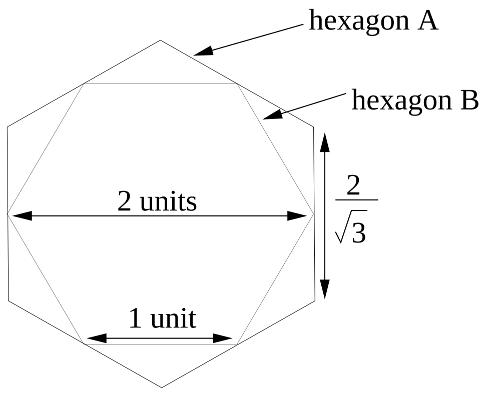
Figure 3.15: Tiling of the domain with the circumscribing hexagons



the rotated hexagon hierarchy. If the larger rotated hexagon is occupied, the scan continues with the topmost level of the base hexagon hierarchy. If it is occupied, the scan goes down one level to the rotated hexagon hierarchy. Thus, the bit scan goes from the rotated hexagon hierarchy to the base hexagon hierarchy at the same level, and goes down one level from the base hexagon hierarchy to the rotated hexagon hierarchy if there are neither free rotated hexagons nor unrotated hexagons at the current level. The midpoint of the first free hexagon encountered, either rotated or not, is returned and inserted. Furthermore, in the decomposition step, both hierarchies have to be decomposed and interpolated.

Suppose the edge-length of a base hexagon is 1 unit (a unit-size hexagon). The edge-length of the circumscribing hexagon is $2/\sqrt{3}$ units and the distance between parallel sides (height) of the circumscribing hexagon is 2 units. See Figure 3.16.

Figure 3.16: HEXAGON A CIRCUMSCRIBING THE BASE HEXAGON B



The worst case ratio can be improved with the additional hexagon hierarchy, having rotated base hexagons of edge-length $2/\sqrt{3}$. If all the hexagons with edge-length at least 2 are occupied and there exists an empty circumscribing hexagon with edge-length $2/\sqrt{3}$. Instead of returning the midpoint of a unit edge-lengthed hexagon, the midpoint of a hexagon of edge-length $2/\sqrt{3}$ is returned. The closest possible point to the hexagon of edge-length $2/\sqrt{3}$ is one unit away. Hence the worst case ratio is 4 using the argument in Section 3.2.1.

On the other hand, if the largest unoccupied hexagon has edge-length 1, the maximum possible distance between the most isolated point and an inserted point is $\frac{4}{\sqrt{3}}$. For example, the distance between A and D in Figure 3.15 is such a case. Using the argument in Section 3.2.1, the worst case ratio is $\frac{4}{\sqrt{3}}/\frac{\sqrt{3}}{2} = \frac{8}{3}$. Hence, there is a large jump in the worst case ratio depending on the scenario.

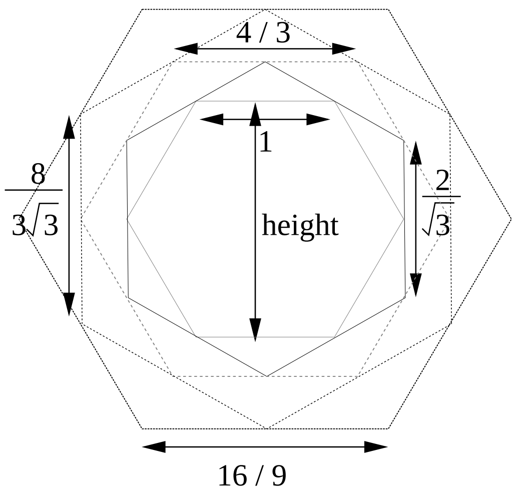
The number of bits required to represent the two hexagon hierarchies will be double that of the hexagons only approach since there is a rotated hexagon for each unrotated hexagon at all levels. Therefore, the space requirement is at most $\frac{32}{3}n * 2 = \frac{64}{3}n$ bits.

Theorem 3.4 *The worst case ratio of the hexagon approach with double embedding without overlapping is* 4.

Quintuple Embedding

The worst case ratio with the double embedding technique can be further improved if five levels of embeddings are used instead of two. The reason for using five levels of embedding is that the edge-length of the sixth level is larger than twice the edge-length of the base hexagon. Hexagons at the level above in the base hexagon hierarchy have edge-lengths doubled.

Figure 3.17: FIVE LEVELS OF EMBEDDING



With quintuple embedding, 5 independent hexagon hierarchies are built. For each hexagon at all levels of the base hierarchy, there are four additional rotated hexagons with different edge-lengths. See Figure 3.17. Therefore, the number of levels in each of the five hexagon hierarchies is the same. The edges lengths of the hexagons in the bottommost level of the hierarchies are 1, $\frac{2}{\sqrt{3}}, \frac{4}{3}, \frac{8}{3\sqrt{3}}, and \frac{16}{9}$, respectively. The edge-lengths and heights of each of the five levels are shown in Table 3.1.

The algorithms for *insert* and *isolate* are similar to those of the double embedding technique, except that there are four additional levels of embedding instead of one. Five bit vectors are required to represent the hexagon hierarchies. When a new point is inserted, the point has to be inserted in the appropriate isosceles

Table 3.1: The edge-lengths and heights of the quintuple embedding technique

edge-length	$\frac{1}{2}$ *height*	*worst case ratio*
1	$\frac{\sqrt{3}}{2}$	
$\frac{2}{\sqrt{3}}$	1	$2 * \frac{2}{\sqrt{3}} / \frac{\sqrt{3}}{2} = \frac{8}{3}$
$\frac{4}{3}$	$\frac{2}{\sqrt{3}}$	$2 * \frac{4}{3} / 1 = \frac{8}{3}$
$\frac{8}{3\sqrt{3}}$	$\frac{4}{3}$	$2 * \frac{8}{3\sqrt{3}} / \frac{2}{\sqrt{3}} = \frac{8}{3}$
$\frac{16}{9}$	$\frac{8}{3\sqrt{3}}$	$2 * \frac{16}{9} / \frac{4}{3} = \frac{8}{3}$
2	$\sqrt{3}$	$2 * 2 / \frac{8}{3\sqrt{3}} = \frac{3\sqrt{3}}{2} \approx 2.60$

trapezoids at the bottommost level in each of the five hexagon hierarchies, and the bit vector for each of the five hexagon hierarchies is updated. Since the five hexagon hierarchies are independent, they can be built by looking at the four bits representing the appropriate isosceles trapezoids in the level below in each of the hierarchies. However, when looking for the largest unoccupied hexagon, we start with the topmost level of the hexagon hierarchy with base hexagon size $\frac{16}{9}$. If it is occupied, we go to the same level of the hexagon hierarchy with base hexagon size $\frac{8}{3\sqrt{3}}$ and so on. So, we only go down one level in the bit scan after looking at all the bits representing the five hierarchies at the same level in the order of decreasing edge-lengths. Constant amortized running time per operation is guaranteed as in the double embedding technique.

The number of bits needed to represent the five hexagon hierarchies is five times that in Section 3.2.1, which is $5(\frac{32}{3}n) = \frac{160}{3}n$ bits.

The worst case ratios in Table 3.1 are computed using the argument in Section 3.2.1. Note that the worst case ratio when we jump from one level to the next in the hexagon hierarchies is approximately 2.60, which is less than $\frac{8}{3}$. Therefore, the worst case ratio with the quintuple embedding technique is $\frac{8}{3}$.

Theorem 3.5 *The worst case ratio of the hexagon approach with quintuple embedding without overlapping is* $\frac{8}{3}$.

3.3.3 Quintuple Embedding With Quadruple Overlapping

In this section, it will be shown that combining the quadruple overlapping technique with the quintuple embedding approach gives a worst case ratio of 2.0 at the expense of at most $160n$ bits of space.

In the combined technique, quadruple overlapping is used for each of the five hexagon hierarchies. Thus, the number of bits required to represent the hexagon hierarchies is five times that of the pure quadruple overlapping technique, which is at most $5(32n) = 160n$ bits. Here, we are using the implicit representation in which whether a hexagon is occupied or not is determined on the fly by doing an OR operation on the bits representing the six triangles making up the hexagon.

When the worst case ratio was computed in Section 3.2.1, the maximum possible distance between the most isolated point and an inserted point is 4 units. With quadruple space overlapping in Section 3.3.1, the maximum distance is 3 units. So, the quadruple overlapping technique reduces the maximum possible distance between the most isolated point and an inserted point by a factor of $\frac{3}{4}$. Since the closest possible point to the midpoint of the hexagon returned by *isolate* with quintuple embedding is not changed by the additional overlapping hexagons, the worst case ratio is reduced by a factor of $\frac{3}{4}$. Hence the worst case ratio of the quintuple embedding technique with quadruple overlapping is $\frac{8}{3} * \frac{3}{4} = 2$.

Theorem 3.6 *The worst case ratio of the hexagon approach with quintuple embedding and quadruple overlapping is 2.*

3.4 Summary of the Hexagon Techniques

In the following table, the implicit approach of determining whether an overlapping hexagon is occupied by doing OR operations on the fly is assumed.

Table 3.2: SUMMARY OF THE HEXAGON TECHNIQUES

technique	*max # of bits*	*worst case ratio*
hexagons only	$\frac{32}{3}n$	$\frac{8}{\sqrt{3}} \approx 4.62$
quadruple overlapping	$32n$	$\frac{6}{\sqrt{3}} \approx 3.47$
triple overlapping	$32n$	$\frac{2}{3}\sqrt{28} \approx 3.53$
quintuple embedding	$\frac{160}{3}n$	$\frac{8}{3} \approx 2.67$
quintuple embedding & quadruple overlapping	$160n$	2.0

3.5 Multiple Grids

As in the case of squares in Section 2.5, we can improve the quality of the solution by having multiple independent grids.

3.5.1 Double Grids

Let Grid 1 be the base grid with the edge-length of the smallest hexagon being 1, and Grid 2 be another grid with edge-length of the smallest hexagon being α, where $\alpha > 1$.

Hexagons Only Approach

Without loss of generality, assume all the hexagons with edge-lengths greater than or equal to 2 are occupied. So, the midpoint in a hexagon of edge-length α is returned by *isolate*. Using the same argument as in Section 3.2.1, the worst case ratio is equal to $2(2)/\frac{\sqrt{3}\alpha}{2} = \frac{8}{\sqrt{3}\alpha}$. On the other hand, if all the hexagons with edge-lengths greater than or equal to α are occupied, the midpoint of a unit edge-length hexagon is returned. And the worst case ratio is $2\alpha/\frac{\sqrt{3}}{2} = \frac{4\alpha}{\sqrt{3}}$ instead. With $\alpha = \sqrt{2}$, the two worst case ratios are equal to $4\sqrt{\frac{2}{3}} \approx 3.27$, which is an improvement over the corresponding single grid approach.

However, the number of bits required to represent the hexagon hierarchies is increased. If there are n hexagons of size 1 in Grid 1, we will have approximately $\frac{n}{2}$ hexagons of size $\sqrt{2}$ in Grid 2. Therefore, the total number of bits needed is at most $\frac{32}{3}n + \frac{32}{3} * \frac{n}{2} = 16n$.

Hexagons With Quadruple Overlapping

The quadruple overlapping technique discussed in Section 3.3.1 reduces the worst case ratio of the hexagons only approach by a factor of $\frac{3}{4}$. It can be verified that with $\alpha = \sqrt{2}$, the worst case ratio of the quadruple overlapping technique is $4\sqrt{\frac{2}{3}}(\frac{3}{4}) = 3\sqrt{\frac{2}{3}} \approx 2.45$.

With the implicit bit representation, the number of bits required to represent the hexagon hierarchies would be $32(n + \frac{n}{2}) = 48n$.

Hexagons With Quintuple Embedding

In the quintuple embedding technique with multiple grids, when all the hexagons with edge-lengths greater than or equal to 2 are occupied, the midpoint of a hexagon with edge-length $\frac{16}{9}\alpha$ is returned in *isolate.* The worst case ratio is $2(2)/\frac{8\alpha}{3\sqrt{3}} = \frac{3\sqrt{3}}{2\alpha}$. If all the hexagons of edge-lengths $\frac{16}{9}\alpha$ are occupied, the midpoint of a hexagon of edge-length $\frac{16}{9}$ is returned and the worst case ratio is $2(\frac{16}{9}\alpha)/\frac{8}{3\sqrt{3}} = \frac{4\alpha}{\sqrt{3}}$. Similarly, when all hexagons with edge-lengths $\frac{16}{9}$ are occupied, the midpoint of a hexagon with edge-length $\frac{8}{3\sqrt{3}}\alpha$ is returned and the worst case ratio is $2(\frac{16}{9})/\frac{4\alpha}{3} = \frac{8}{3\alpha}$. The same reasoning applies to all the other levels of embedding. It can be verified that with $\alpha = \sqrt{\frac{2}{\sqrt{3}}}$, the worst case ratios are equal to $\frac{4}{\sqrt{3}}(\frac{2}{\sqrt{3}})^{1/2}$.

The number of bits required to represent the hexagon hierarchies with quintuple embedding is at most $\frac{160}{3}(n + \frac{n}{2}) = 80n$.

3.5.2 m Grids

Let Grid 1 be the base grid with the edge-length of the smallest hexagons being 1, and Grid i be a grid with the edge-length of the smallest hexagons being α_{i-1}, where $i = 2, 3, \ldots, m$, $2 > \alpha_{m-1}$, $\alpha_i > \alpha_{i-1}$, and $\alpha_1 > 1$.

Hexagons Only Approach

It can be shown that with $\alpha_i = 2^{i/m}$, the worst case ratio is $\frac{4}{\sqrt{3}} * 2^{1/m}$. As the number of grids tends to infinity, the lower bound of the worst case ratio for the hexagons only approach is $\frac{4}{\sqrt{3}}$. However, the number of bits needed to represent the hierarchies would also increase to infinity in order to achieve the lower bound.

Theorem 3.7 *The lower bound of the worst case ratio of the hexagons only approach with multiple grids is* $\frac{4}{\sqrt{3}} \approx 2.31$.

Hexagons With Quadruple Overlapping

Since the worst case ratio is reduced by a factor of $\frac{3}{4}$ from the hexagons only approach, the worst case ratio with m grids is $\frac{4}{\sqrt{3}} * 2^{1/m} * \frac{3}{4} = \sqrt{3} * 2^{1/m}$. Hence, the lower bound of the worst case ratio is $\sqrt{3}$ when the number of grids tends to infinity.

Theorem 3.8 *The lower bound of the worst case ratio of the hexagons approach with quadruple overlapping and multiple grids is* $\sqrt{3} \approx 1.73$.

Hexagons With Quintuple Embedding

It can be shown that with $\alpha_i = (\frac{2}{\sqrt{3}})^{i/m}$, the worst case ratio is $\frac{4}{\sqrt{3}} * (\frac{2}{\sqrt{3}})^{1/m}$. As the number of grids tends to infinity, the lower bound of the worst case ratio is $\frac{4}{\sqrt{3}}$.

Theorem 3.9 *The lower bound of the worst case ratio of the hexagons approach with quintuple embedding and multiple grids is* $\frac{4}{\sqrt{3}} \approx 2.31$.

Chapter 4

Dividing the Domain into Triangles

Besides squares and hexagons, triangles can also tile a plane. In this chapter, equilateral triangles and right-angled triangles are considered in the hope of achieving a simple and high quality solution with low space requirements.

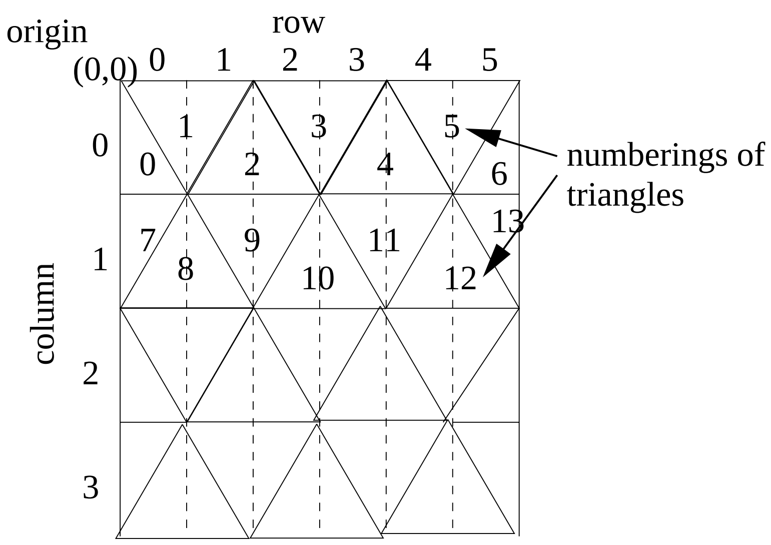
In the square approach with diamonds in Chapter 2 and the hexagon approach with overlapping in Chapter 3, we have to keep track of triangles making up a square or a hexagon instead of the actual squares or hexagons so as to achieve constant amortized running time per operation. Recall that in the square approach with diamonds a square is made up of four isosceles triangles, while in the hexagon approach with overlapping a hexagon is made up of six equilateral triangles. In fact, the grouping of four equilateral triangles to form a larger equilateral triangle in the hexagon approach with overlapping is the same as that in the equilateral triangle approach to be discussed in Section 4.1. However, the basic shape by which the containment relationship is divided differs in each case: the basic shape is the triangle, the square and the hexagon in the triangle approach, square approach and the hexagon approach respectively. The basic shape is the shape with which the worst case ratio is computed. For example, in the hexagon approach, all the hexagons of a certain size have to be occupied such that the midpoint of a hexagon one level down in the hierarchy is returned by *isolate* and a hexagon is occupied if any of the six equilateral triangles making up the hexagon is occupied. On the other hand, all the equilateral triangles of a certain size have to be occupied in order for the midpoint of a triangle one level down in the hierarchy is returned by *isolate* in the equilateral triangle approach which will be discussed in this chapter. Moreover, there are at most $4n$ base equilateral triangles in order to guarantee at least one empty triangle in the equilateral triangle approach while there are at most $4n$ base hexagons which is equivalent to $24n$ triangles to guarantee at least one empty hexagon in the hexagon approach. Furthermore, the overlapping techniques are different in the equilateral triangle approach and in the hexagon approach, as will be discussed in Section 4.1.2.

4.1 Equilateral Triangles

4.1.1 Basic Approach

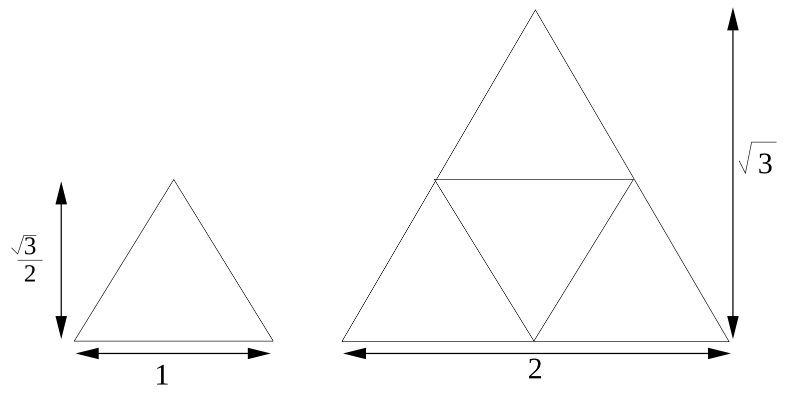
Four smaller equilateral triangles make up a double sized equilateral triangle. See Figure 4.1. Equilateral triangles are simpler than hexagons because the boundaries of four equilateral triangles form the boundary of a larger equilateral triangle.

Figure 4.1: FOUR SMALLER EQUILATERAL TRIANGLES MAKE UP A BIGGER ONE



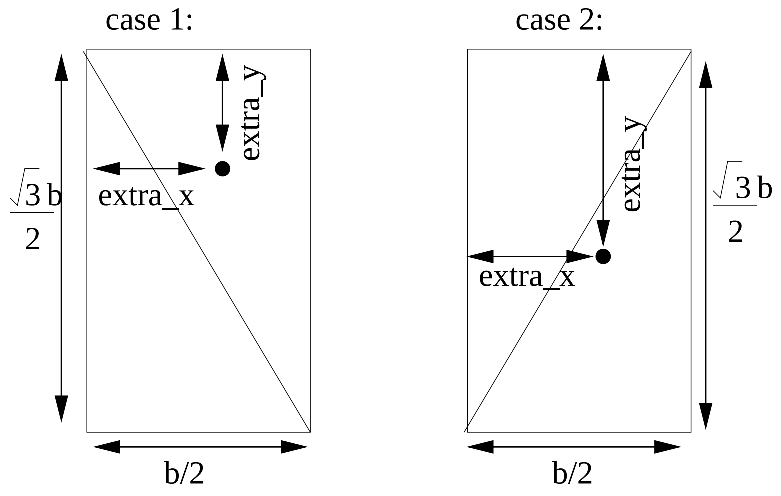
The equilateral triangle in which a given point with cooridinates (X,Y) lies can be determined in constant time using algorithm WHICHTRIANGLE. Assume the triangle numbering starts from zero and across the rows and then down the columns, and the origin is in the top left hand corner of the domain, as illustrated in Figure 4.2.

Figure 4.2: TO DETERMINE THE TRIANGLE A GIVEN POINT LIES IN



The idea of the algorithm WHICHTRIANGLE is very similar to that for hexagons in Lemma 3.7 in Chapter 3. First, find the rectangle in which the point lies, and then determine whether it lies in the upper or lower half of the rectangle by comparing the slope. Case 1 is the case in which the point lies in a rectangle cut by a line with negative slope, and case 2 is that with a positive slope. See Figure 4.3. Let b be the edge length of the base equilateral triangle.

Figure 4.3: ILLUSTRATION OF CASE 1 AND CASE 2



<u>WHICHTRIANGLE (X,Y):</u>

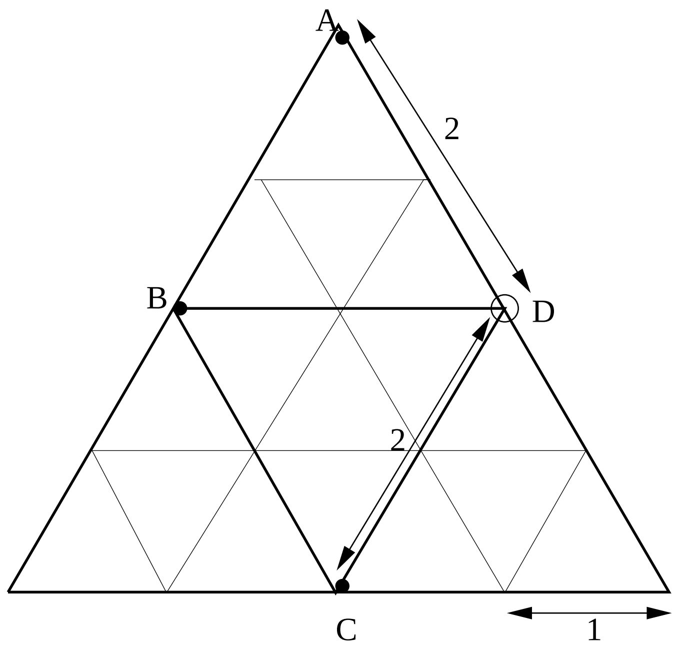
```
1. $col = \lfloor \frac{X}{b/2} \rfloor$
2. $row = \lfloor \frac{Y}{\sqrt{3}b/2} \rfloor$
3. $extra\_x = x - col * \frac{b}{2}$
4. $extra\_y = y - row * \frac{\sqrt{3}b}{2}$
5. if $row + col \bmod 2 = 0$ then
```

```
    • if $\frac{extra\_y}{extra\_x} > -\sqrt{3}$ then
        – $col = col + 1$
6. else (i.e., $row + col \bmod 2 = 1$)
    • if $\frac{\frac{\sqrt{3}b}{2} - extra\_y}{extra\_x} < \sqrt{3}$ then
        – $col = col + 1$
7. RETURN $(row, col)$
```

As in the square approach, we need a linked list (or an array), a bit vector representing the triangle hierarchy and a counter of the number of points inserted. The linked list (or array) stores the points inserted in the domain. The number of bits in the bit hierarchy is reduced by a factor of 4 each level up the hierarchy. The bit vector hierarchy can be built by doing an OR of the four bits representing the four smaller triangles making up the larger one. Therefore, as in the squares only approach, the number of bits required is at most $\frac{16}{3}n$ where n is the number of points already inserted into the domain, and the triangle hierarchy can be built in linear time.

The worst case ratio is approximately 6.93 which is worse than the squares only approach. Suppose all equilateral triangles of size at least 2 are occupied. Somewhere in the domain, there are three points A, B and C at the corners of three size 2 triangles and they are at a distance of 2 units from the centre point D, as illustrated in Figure 4.4. There are no other points inside the circle centred at point D and passing through points A, B and C. It can be verified that the circle centred at D with radius 2 units passing through points A, B and C is the circle with maximum radius passing through any six size 2 equilateral triangles. The distance between the most isolated point and an inserted point is 2. On the other hand, the closest point possible to a size 1 equilateral triangle is $\frac{1}{2\sqrt{3}}$ units away from the midpoint of the triangle as shown in Figure 4.5. Hence, the worst case ratio is $2/\frac{1}{2\sqrt{3}}$ which is $4\sqrt{3} \approx 6.93$.

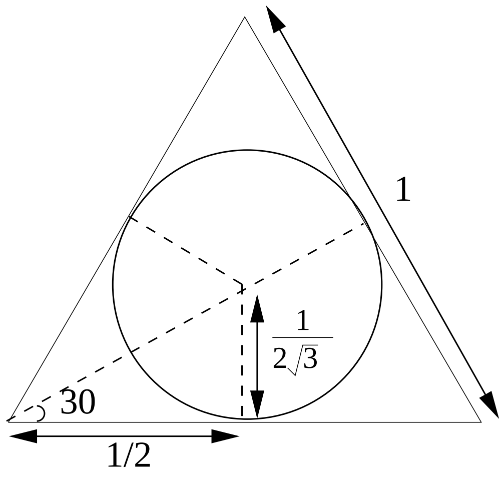
Figure 4.4: Illustration of the worst case ratio of the equilateral triangle technique



Note that a boundary correction convention has to be adopted to solve the problem of having a point lying on the boundaries of more than one triangle. Therefore, the actual worst case ratio is slightly less than $4\sqrt{3}$.

Theorem 4.1 *Using the equilateral triangle approach as described in the preceding sections,* insert*s and* isolate*s can be performed in constant amortized time per operation, linear space and constant worst case ratio of* $4\sqrt{3}$ (≈ 6.93).

Figure 4.5: ILLUSTRATION OF THE CLOSEST POINT TO AN EQUILATERAL TRIANGLE



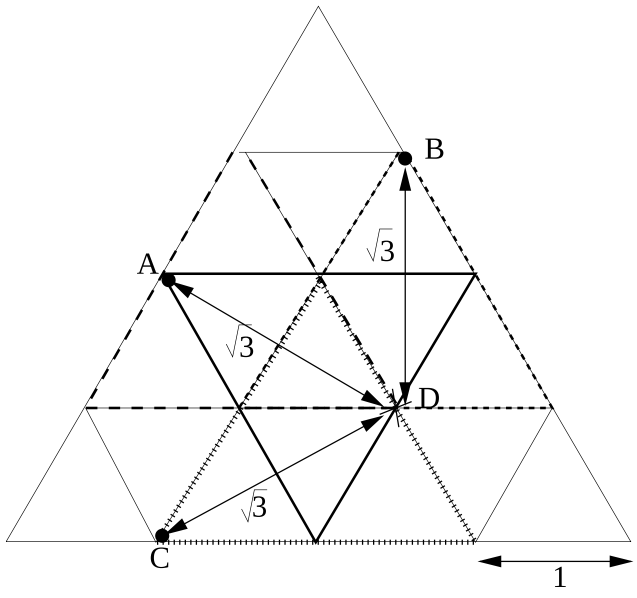
4.1.2 With Overlapping

The worst case ratio can be improved to 6 by using overlapping triangles. See Figure 4.6. Suppose all equilateral triangles (including the overlapping ones) with size at least 2 are occupied. There are three points A, B and C somewhere in the domain, occupying four equilateral triangles and there are no other points in the circle passing through these three points. The distance between D and A is $\sqrt{3}$. The circle centred at D passing through points A, B and C is the circle with maximum radius passing through four overlapping triangles. As in Figure 4.5, an inserted point $\frac{1}{2\sqrt{3}}$ units away from the midpoint of a size 1 triangle is returned by *isolate*. Hence, the worst case ratio is $\sqrt{3}/\frac{1}{2\sqrt{3}} = 6$.

Due to the boundary correction convention, the actual worst case ratio is slightly less than 6.

The maximum number of bits needed to represent the overlapping triangles

Figure 4.6: Improving the worst case ratio with the overlapping technique



explicitly is increased to $\frac{28}{3}n$ as in the square approach with overlapping. With the implicit representation of the overlapping triangles, the number of bits remains the same as in the case without overlapping.

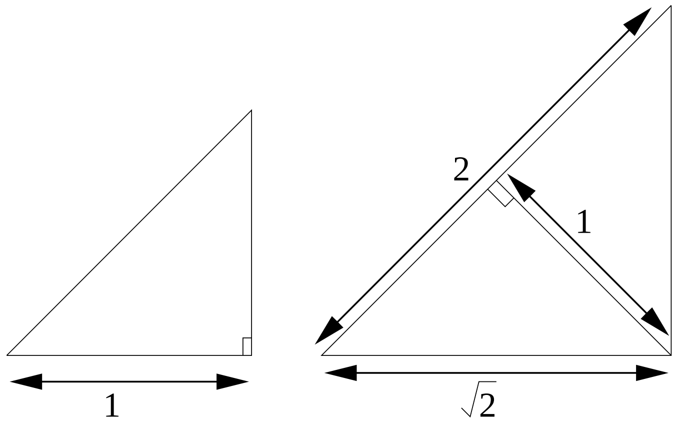
Theorem 4.2 *The worst case ratio of the equilateral triangle approach with overlapping is 6.*

4.2 Right-Angled Triangles

4.2.1 Basic Approach

Two unit size right-angled triangles combine to form a right-angled triangle of size $\sqrt{2}$ as in Figure 4.7. As in the case of equilateral triangles, the boundary of the unit-size triangles form the boundary of the larger triangle.

Figure 4.7: TWO SIZE 1 RIGHT-ANGLED TRIANGLES MAKE UP ONE OF SIZE $\sqrt{2}$

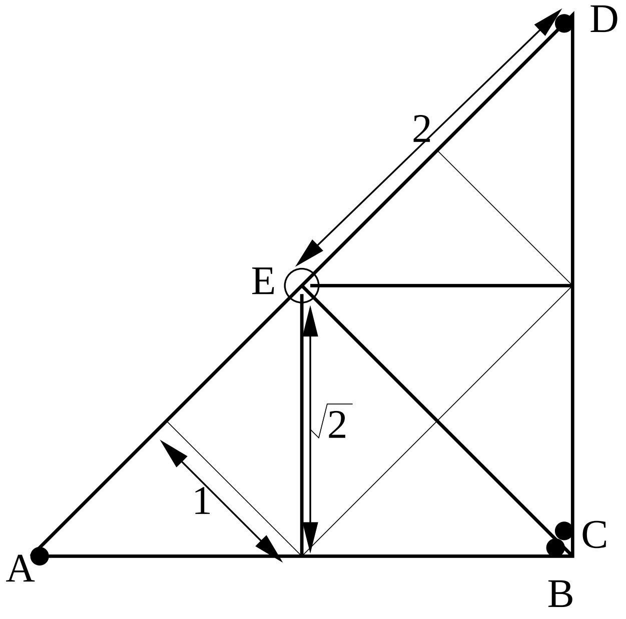


Since two unit-size right-angled triangles make up a size $\sqrt{2}$ right-angled triangle, the number of bits in each level up the bit hierarchy is reduced by a factor of 2. To guarantee at least one unoccupied right-angled triangle, we need $\lceil \log_2(n+1) \rceil$ levels in the triangle hierarchy. Hence, the number of triangles at the bottommost level is $2^{\lceil \log_2(n+1) \rceil}$, which is at most $2n$. With a square domain, the top level consists of two right-angled triangles. Therefore, the number of bits required to represent the bit hierarchy is at most $\sum_{i=1}^{\log_2 n+1} 2^i = 4n - 2 < 4n$, assuming a square domain.

The right-angled triangle in which a point lies can be determined in constant time by first determining the square in which the point lies and then comparing the slope as in algorithm WHICHTRIANGLE (Section 4.1.1).

Consider Figure 4.8: all the triangles with size at least $\sqrt{2}$ are occupied, and somewhere in the domain there are four points A, B, C and D occupying four size $\sqrt{2}$ triangles. There are no other points inside the circle passing through points A, B, C and D. The distance between A and E is 2 units. It can be seen that this circle, centred at point E, is the circle with maximum radius containing exactly eight size $\sqrt{2}$ right-angled triangles. On the other hand, the closest point possible to a right-angled triangle of size 1 is $\frac{2-\sqrt{2}}{2} \approx 0.293$ units away as illustrated in Figure 4.9. Therefore, the worst case ratio is $2/\frac{2-\sqrt{2}}{2} = \frac{4}{2-\sqrt{2}} \approx 6.83$.

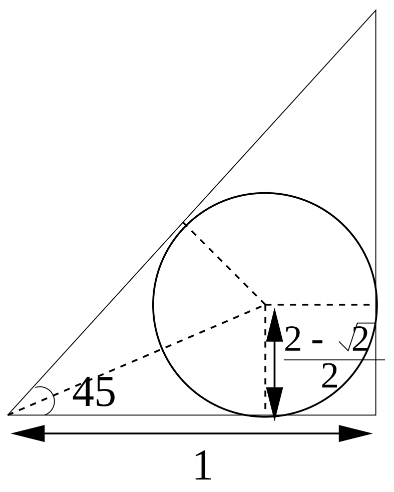
Figure 4.8: Illustration of the worst case ratio of the right-angled triangle technique



The actual worst case ratio is slight less than $\frac{4}{2-\sqrt{2}}$ with the adoption of a boundary correction convention.

Theorem 4.3 *Using the right-angled triangle approach as described in the preceding sections,* inserts *and* isolates *can be performed in constant amortized time per operation, linear space and constant worst case ratio of* $\frac{4}{2-\sqrt{2}}$ (≈ 6.83).

Figure 4.9: Illustration of the closest point to a right-angled triangle



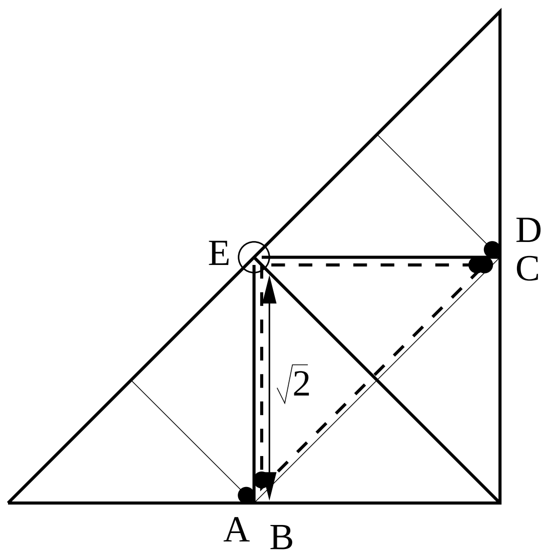
4.2.2 With Overlapping

With overlapping, the worst case ratio can be reduced to approximately 4.83. See Figure 4.10. All the triangles (including the overlapping ones) with size at least $\sqrt{2}$ are occupied. The distance between A and E is $\sqrt{2}$ units. The circle centred at point E passing through points A, B, C and D is the circle with maximum radius passing through the overlapping and non-overlapping size $\sqrt{2}$ triangles. As in Figure 4.9, a point $\frac{2-\sqrt{2}}{2} \approx 0.293$ units from the midpoint of a size 1 triangle is returned by *isolate*. Therefore, the worst case ratio is $\sqrt{2}/\frac{2-\sqrt{2}}{2} = \frac{2\sqrt{2}}{2-\sqrt{2}} \approx 4.83$.

Again, the actual worst case ratio is slightly less than $\frac{2\sqrt{2}}{2-\sqrt{2}}$ with a boundary correction convention.

With the explicit representation of overlapping triangles, the number of bits needed to represent the triangle hierarchy is doubled with overlapping, and at most $8n$ bits are needed. Alternatively, at most $4n$ bits are needed with the implicit

Figure 4.10: Improving the worst case ratio for right-angled triangles with overlapping



representation.

Theorem 4.4 *The worst case ratio of the right-angled triangle approach with overlapping is* $\frac{2\sqrt{2}}{2-\sqrt{2}}$ *(*≈ 4.83*).*

4.3 Summary of the Triangle Techniques

In the following table, the *implicit* representations of overlapping triangles are assumed.

4.4 Multiple Grids

As in the cases of squares and hexagons, we can improve the worst case ratio at the expense of space by having multiple independent grids.

Table 4.1: Summary of the triangle techniques

technique	*maximum # of bits*	*worst case ratio*
equilateral triangles only	$\frac{16}{3}n$	$4\sqrt{3} \approx 6.9$
equilateral triangles with overlapping	$\frac{16}{3}n$	6
right-angled triangles only	$4n$	$\frac{4}{2-\sqrt{2}} \approx 6.83$
right-angled triangles with overlapping	$4n$	$\frac{2\sqrt{2}}{2-\sqrt{2}} \approx 4.83$

4.4.1 Double Grids

Let Grid 1 be the base grid with the size of the smallest equilateral triangles being 1, and Grid 2 be another grid with size of the smallest equilateral triangles being α, where $\alpha > 1$.

Equilateral Triangles Only Approach

Without loss of generality, assume all the triangles with size greater than or equal to 2 are occupied. So, the midpoint in a triangle of size α is returned by *isolate*. Using the same argument as in Section 4.1.1, the worst case ratio is equal to $2/\frac{\alpha}{2\sqrt{3}} = \frac{4\sqrt{3}}{\alpha}$. On the other hand, if all the triangles with size greater than or equal to α are occupied, the midpoint of a triangle of size 1 is returned, and the worst case ratio is $\alpha/\frac{1}{2\sqrt{3}} = 2\sqrt{3}\alpha$ instead. With $\alpha = \sqrt{2}$, the two worst case ratios are equal to $2\sqrt{6} \approx 4.90$.

However, the number of bits required to represent the triangle hierarchies is increased. If there are n triangles of size 1 in Grid 1, we will have approximately $\frac{n}{2}$ triangles of size $\sqrt{2}$ in Grid 2. Therefore, the total number of bits needed is at most $\frac{16}{3}n + \frac{16}{3} * \frac{n}{2} = 8n$.

Equilateral Triangles With Overlapping

Using the argument in Section 4.1.2, when all the triangles with size greater than or equal to 2 are occupied, and the midpoint in a triangle of size α is returned by *isolate*. It can be verified that the worst case ratio is equal to $\sqrt{3}/\frac{\alpha}{2\sqrt{3}} = \frac{6}{\alpha}$ with overlapping. On the other hand, if all the triangles with size greater than or equal to α are occupied, the midpoint of a triangle of size 1 is returned, and the worst case ratio is $\frac{\sqrt{3}\alpha}{2}/\frac{1}{2\sqrt{3}} = 3\alpha$ instead. With $\alpha = \sqrt{2}$, the two worst case ratios are equal to $3\sqrt{2} \approx 4.24$.

The number of bits required to represent the triangle hierarchies with overlapping remains $8n$ with the implicit representation of overlapping triangles.

4.4.2 m Grids

Let Grid 1 be the base grid with the size of the smallest triangles being 1, and Grid i be a grid with size of the smallest triangles being α_{i-1}, where $i = 2, 3, \ldots, m$, $2 > \alpha_{m-1}$, $\alpha_i > \alpha_{i-1}$, and $\alpha_1 > 1$.

Equilateral Triangles Only Approach

It can be shown that with $\alpha_i = 2^{i/m}$, the worst case ratio is $2\sqrt{3} * 2^{1/m}$. As the number of grids tends to infinity, the lower bound of the worst case ratio for the equilateral triangles only approach is $2\sqrt{3}$. However, the number of bits needed to represent the hierarchies would also increase to infinity in order to achieve the lower bound.

Theorem 4.5 *The lower bound of the worst case ratio of the equilateral triangles only approach with multiple grids is* $2\sqrt{3}$ (≈ 3.46).

Equilateral Triangles With Overlapping

Similarly, it can be shown that with $\alpha_i = 2^{i/m}$, the worst case ratio is $3 * 2^{1/m}$. Therefore, as the number of grids tends to infinity, the lower bound of the worst case ratio is 3.

Theorem 4.6 *The lower bound of the worst case ratio of the equilateral triangles approach with overlapping and multiple grids is* 3.

Chapter 5

Extending to Three Dimensions

Chapters 2, 3 and 4 handle the case when the given domain is two dimensional. In this chapter, we briefly discuss the case when the given domain is three dimensional. In terms of the application of the root-finding problem in *Maple* [1], the three dimensional domain corresponds to solving an equation with three variables.

The techniques of dividing the given domain into regular shapes in Chapters 2, 3 and 4 will be extended to three dimensions. Section 5.1 considers the case of extending the square approach to cubes and Section 5.2 discusses the possibility of dividing the domain into other geometric shapes.

5.1 Dividing the Domain into Cubes

The idea of dividing the given two dimensional domain into squares and building a hierarchy of squares can be extended to the three dimensional domain using cubes. In this section, the possibility of improving the quality of the solution using other techniques in the two dimensional case, such as overlapping and diamonds, are discussed.

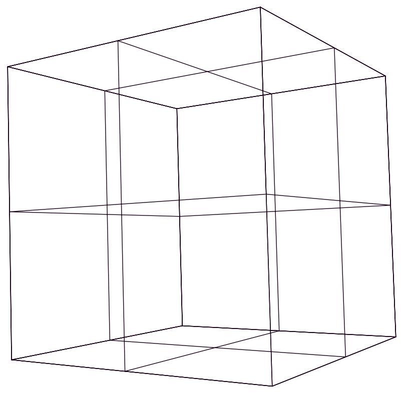
5.1.1 The General Approach

Without loss of generality, assume the given three dimensional domain is a cube. The idea is to divide this domain into smaller cubes and to build a hierarchy of these cubes, represented by a bit vector. The bit corresponding to a particular cube in the hierarchy is turned on if that cube is occupied and is turned off if that cube is empty. As in the case of squares, a linked list (or a simple array) is used to store points inserted in the domain.

In the case of squares, four unit-size squares combine to form a double-size square. However, in the case of cubes, eight unit-size cubes combine to form a double sized cube as shown in Figure 5.1. Therefore, the number of bits in each level is reduced by a factor of eight instead of four in going up a level.

The algorithm for the insert operation is the same as in Section 2.2.1 except that squares are replaced by cubes. The cube in which a point lies can be determined in constant time using a similar technique to that described in Section 2.2.3. The isolate operation is also similar to that described in Section 2.2.2 except in the decomposition step. We need $8^{\lceil \log_8(n+1) \rceil}$ cubes in the bottommost level in the hierarchy in order to guarantee at least one free cube in the domain. Hence, k is set to be $\lceil \log_8(n+1) \rceil$ in the isolate algorithm. Then, the domain is decomposed into 8^k cubes which is at most $8n$. All the n points inserted are interpolated into the grid of 8^k cubes. Then, the bit hierarchy is built and the bit scan continues.

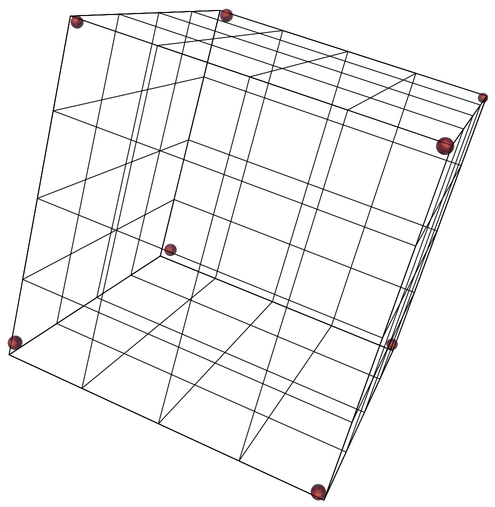
Figure 5.1: Eight unit-size cubes make up a double-size cube



The amortized running time per operation is constant when *inserts* and *isolates* are taken together. The total number of bits in the cube hierarchy is at most $\sum_{i=0}^{\log_8 n+1} 8^i \leq \frac{64}{7}n$, which is $\Theta(n)$. With reasoning similar to that in Section 2.2.4, the worst case running time of any sequence of n operations is linear.

The worst case ratio is $4\sqrt{3}$ with the cubes only approach. Consider the following scenario: all cubes with size at least 2 are occupied in the domain. Somewhere in the domain, there are eight points at the corners of eight size 2 cubes which form a size 4 cube and there are no other points in the sphere containing the eight corner points. See Figure 5.2. The sphere passing through the eight points is the sphere with maximum radius containing eight unit-size cubes. Hence, the maximum distance between any point in the domain and an inserted point is between the centre of the size 4 cube and one of the eight points at the corners of a size 2 cube, or a distance of $2\sqrt{3}$. The midpoint of a unit-size cube is returned in *isolate*. The closest possible point to the returned isolate point is $\frac{1}{2}$ units away. Therefore, the worst case ratio is $2\sqrt{3}/\frac{1}{2} = 4\sqrt{3} \approx 6.93$.

Figure 5.2: Illustration of the worst case ratio



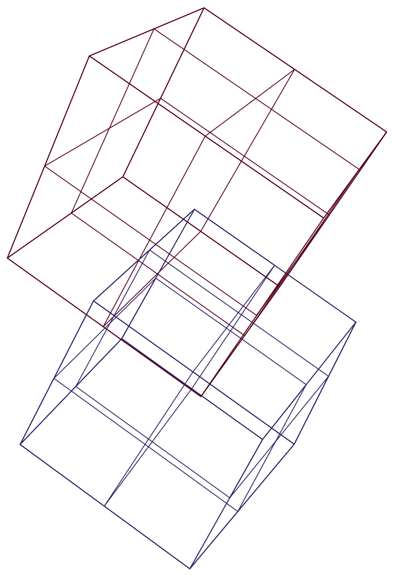
A boundary correction convention similar to that described in Section 2.2.6 can be adopted. Hence, the actual worst case ratio is slightly less than but can made arbitrarily close to $4\sqrt{3}$.

Theorem 5.1 *Using the cubes only approach as described in the preceding sections,* insert*s and* isolate*s can be performed in constant amortized time per operation, linear space and constant worst case ratio of* $4\sqrt{3}(\approx 6.93)$.

5.1.2 The Overlapping Technique

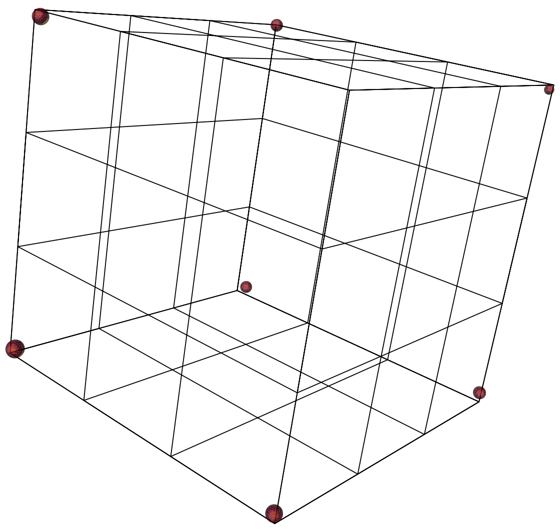
The overlapping technique in Section 2.3.1 can be extended to cubes to improve the worst case ratio. Observe that a size 2 cube is made up of eight unit-size cubes, and so a unit-size cube has eight possible overlapping size 2 cubes. The eight overlapping size 2 cubes form a size 3 cube with a common middle unit-size cube, as in Figure 5.3.

Figure 5.3: Illustration of the overlapping approach



In determining the worst case ratio, imagine all the size 2 cubes are occupied and there are eight points in the domain occupying the corners of the size 3 cube described above with no other points in the sphere containing the eight points. The distance between the middle point in the size 3 cube, which is the most isolated point, and one of the eight corner points is $\frac{3}{2}\sqrt{3}$. See Figure 5.4. Hence, the worst case ratio is $\frac{3}{2}\sqrt{3}/\frac{1}{2} = 3\sqrt{3} \approx 5.20$.

Figure 5.4: Worst case ratio of the overlapping approach



As in the case without overlapping, the actual worst case ratio is slightly less than $3\sqrt{3}$ with boundary correction.

As in the case of squares, we do not require overlapping in the bottommost level. In the explicit approach, one bit is used to represent each of the eight overlapping cubes. So, the total number of bits required to represent the cube hierarchy is $8\sum_{i=0}^{\log_8 n} 8^i + 8^{\log_8 n+1} \leq \frac{120}{7}n$. In the implicit approach, we determine whether a cube is occupied or not on the fly by doing additional OR operations on the bits representing smaller cubes making up the current cube. So, the number of bits required for the implicit approach is the same as in the cubes only approach which is at most $\frac{64}{7}n$.

Theorem 5.2 *The worst case ratio of the cubes approach with overlapping is* $3\sqrt{3}(\approx 5.20)$

5.1.3 The Diamond Technique

The obvious approach of rotating a reduced size cube and embedding the rotated cube in a unit-size cube does not work. A cube has eight corners but only six faces. It is impossible to have all the corners of the reduced size cube touching all the faces of a cube.

5.2 Dividing the Domain into Other Geometric Shapes

In the two dimensional case, we considered dividing the given domain into squares, hexagons and triangles. In fact, triangles, squares and hexagons are the only regular polygons that can tile a plane [6]. On the other hand, in the three dimensional case, there are only five regular polyhedra, the *Platonic solids*. The five Platonic solids are: tetrahedron, cube, octahedron, dodecahedron and icosahedron [2]. Tetrahedrons, octahedrons and icosahedrons are made up of triangular faces. Cubes are made up of square faces and dodecahedrons are made up of regular pentagonal faces. So, there is no natural correspondence between any of the five platonic solids in three dimensions and the regular hexagons in two dimensions. However, there is a "semi-regular" polyhedra, called the truncated cuboctahedron, which has two regular hexagons and one square joined at each vertex, that tiles a three dimensional space [3]. The possibility of extending our results in Chapter 3 using hexagons in a two dimensional domain to using truncated cuboctahedron in a three dimensional domain would be an interesting problem for future work.

Chapter 6

Conclusion

We have solved the problem such that the amortized running time when *insert*s and *isolate*s are taken together is constant with a constant worst case ratio. The quality of the solution is measured by the worst case ratio. The smaller the worst case ratio, the better the solution. We have considered three different approaches in order to improve the quality of the solution, namely: dividing the domain into different geometric shapes; using techniques such as overlapping, diamonds or embedding; and, having multiple grids of the same geometric shape.

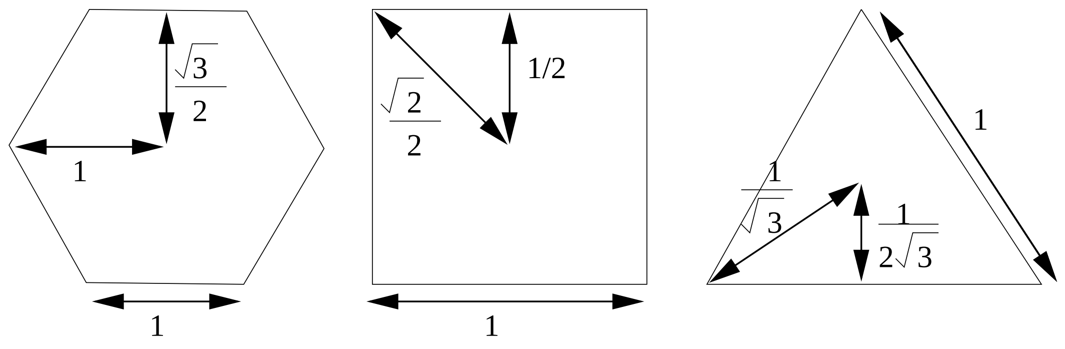
6.1 Dividing the Domain into Different Shapes

The general idea of our solution is to divide the domain into a particular geometric shape and build a hierarchy of the geometric shape. We have to keep track of whether a particular geometric shape is occupied or not, and so we would like to be able to find in which geometric shape a particular point lies easily and efficiently.

We have considered dividing the two dimensional domain into the three regular polygons which tile a plane, namely squares, hexagons and triangles [6]. Only regular polygons are discussed because of the relative ease of determining in which polygon in the given domain a particular point lies. It turns out that the worst case ratio is smaller with the hexagons only approach than with the squares only approach, which is in turn smaller than that with equilateral triangles only. The result is not surprising since hexagons are closer to circles than are squares and triangles. Being closer to circles is desirable because the ratio of the distance between the centre of the polygon and the farthest point on the polygon to the distance between the centre and the closest point on the polygon is smaller. The ratio is 1 for circles. The ratio is $1/\frac{\sqrt{3}}{2} = \frac{2}{\sqrt{3}} \approx 1.15$ for hexagons, $\frac{\sqrt{2}}{2}/\frac{1}{2} = \sqrt{2} \approx 1.41$ for squares and $\frac{1}{\sqrt{3}}/\frac{1}{2\sqrt{3}} = 2$ for triangles as illustrated in Figure 6.1. There are many other geometric shapes which are very close to being circular which can also tile a plane [6]. For instance, regular octagons together with squares tile a plane and octagons are closer to circles than are hexagons. However, building a hierarchy with two regular polygons is not trivial and would be an interesting problem for future work.

In the case of a three dimensional domain, only cubes have been considered. There are only five platonic solids in three dimensions, namely tetrahedron, cube, octahedron, dodecahedron and icosahedron [2], but not all of them can tile a three dimensional space. There is no natural correspondence of any of the five platonic solids to regular hexagons in two dimensions, but there is a “semi-regular” polyhedron, the truncated cuboctahedron, which has regular hexagonal faces. It would be

Figure 6.1: Illustration of how hexagons are closer to circles than are squares and triangles



an interesting problem to try to solve our problem with “semi-regular” polyhedra and build a hierarchy using “semi-regular” polyhedra.

6.2 Other Techniques

It has been shown that the quality of the solution can be improved using techniques such as overlapping, embedding or diamonds at the expense of either more operations or more bits required. The following table shows the worst case ratios, with the number of bits required for the implicit representation in brackets, using various techniques. Note that quadruple overlapping and quintuple embedding are assumed for hexagons.

6.3 With Multiple Grids

Furthermore, the worst case ratio can be improved by having multiple grids of different sizes of the same geometric shape which form independent hierarchies.

Table 6.1: SUMMARY OF THE SQUARE, HEXAGONS AND TRIANGLES TECHNIQUES

technique	*squares*	*hexagons*	*equilateral triangles*
no special technique	$4\sqrt{2}\ (\frac{16}{3}n)$	$\frac{8}{\sqrt{3}}\ (\frac{32}{3}n)$	$4\sqrt{3}\ (\frac{16}{3}n)$
with overlapping	$3\sqrt{2}\ (\frac{16}{3}n)$	$2\sqrt{3}\ (32n)$	$6\ (\frac{16}{3}n)$
with diamonds or embedding	$4\ (\frac{64}{3}n)$	$\frac{8}{3}\ (\frac{160}{3}n)$	
with overlapping and embedding	$3\ (\frac{64}{3}n)$	$2\ (160n)$	

When the number of grids tends to infinity, the worst case ratio will tend to a lower bound. The following table shows the lower bounds of the worst case ratio with various techniques.

Table 6.2: SUMMARY OF THE SQUARE, HEXAGONS AND TRIANGLES LOWER BOUNDS

technique	*squares*	*hexagons*	*equilateral triangles*
no special technique	$2\sqrt{2} \approx 2.83$	$\frac{4}{\sqrt{3}} \approx 2.31$	$2\sqrt{3} \approx 3.46$
with overlapping	$\frac{3}{2}\sqrt{2} \approx 2.12$	$\sqrt{3} \approx 1.73$	3
with diamonds or embedding	2	$\frac{4}{\sqrt{3}} \approx 2.31$	
with overlapping and embedding	$\frac{3}{2} = 1.50$		

As shown by the first row in Table 6.2, the lower bound for the hexagons only approach is smaller than that for the squares only approach, which is in turn smaller than that for the equilateral triangles only approach, reinforcing what we have discussed in Section 6.1. However, the trend may not be true when combined with other techniques such as overlapping and embedding (diamonds) because the other techniques that can be applied depends on the specific geometric shape. For instance, the lower bound for the hexagon approach with quintuple embedding is larger than that for the square approach with diamonds. In fact, the lower bound for the hexagon approach with quintuple embedding is the *same* as that for the hexagons only approach. The worst case ratio for the hexagons only approach with m grids is $\frac{4}{\sqrt{3}} * 2^{1/m}$ and that for the hexagon approach with quintuple embedding is $\frac{4}{\sqrt{3}} * (\frac{2}{\sqrt{3}})^{1/m}$. Since $\frac{2}{\sqrt{3}} \approx 1.15 < 2.0$, the rate of convergence to the lower bound is much faster with the quintuple embedding.

6.4 Implementation Issues

We have concluded that dividing the given two dimensional domain into regular hexagons gives a more favourable worst case ratio since hexagons are closer to circles than squares and triangles in Section 6.1. However, in practice, determining in which square a point lies requires fewer operations than determining in which hexagon it lies. Furthermore, in Section 2.5.2, we have shown that with the square approach with overlapping and diamonds and 4 grids, the worst case ratio is approximately 1.78 and the number of bits required is at most $55n$. On the other hand, the hexagon approach with overlapping and quintuple embedding gives a worst case ratio of 2.0 with at most $160n$ bits of space. Therefore, the square approach with overlapping, diamonds and 4 grids is highly favorable compared to the hexagons approach both in terms of the worst case ratio and the space requirement. We recommend this technique for the actual implementation of the root-finding problem in Maple [1].

6.5 Future Work

There are many possible interesting problems for future work related to this problem.

We have only considered dividing the given domain into regular polygons and regular polyhedra. It would be interesting to divide the domain into "semi-regular" polygons and polyhedra that can tile the given domain. In two dimensions, regular octagons and squares tile a plane and octagons are promising since they are closer to circles than are hexagons. In three dimensions, the truncated cuboctahedron corresponds naturally to the regular hexagon approach in two dimensions. However, building a hierarchy and computing the worst case ratio with "semi-regular" geometric shapes may not be trivial.

Moreover, there are many other non-regular shapes that can tile a plane or space. Although it requires a lot of computational effort to determine in which shape a given point lies, exploring the possibility of getting a better solution with non-regular shapes would be of theoretic interest.

We have also attempted to divide the domain into circles in order to improve the solution. Since circles do not tile a plane, we did not have any success in attaining a lower worst case ratio with constant amortized running time per operation. This is certainly another possibility for improving the quality of the solution.

Gaston Gonnet has conjectured a lower bound of $\sqrt{2}$ for the two dimensional case with the square approach. We have proved a lower bound of 1.50 for the square approach with overlapping and diamonds. It is possible that overloading the square approach with additional techniques may give a smaller lower bound.

We concentrated on the Euclidean (i.e., L_2) metric in this thesis. Other metrics could have been used instead, for example, the L_1 and L_∞ metrics. In the Euclidean metric, the closer the geometric shape is to a circle, the better is the quality of the solution. In L_1 and L_∞ metrics, squares should give very good solutions.

Finally, extending our current approach to k dimensions would be another interesting problem with practical applications of solving an equation with k variables in Maple [1].

Bibliography

[1] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan and S. M. Watt. *Maple V Library Reference Manual.* Springer-Verlag, 1991.

[2] H. S. M. Coxeter. *Introduction to Geometry.* John Wiley & Sons, Inc. Chapter 10, 1969.

[3] H. S. M. Coxeter. *Regular Complex Polytopes.* Cambridge University Press. Chapter 2, 1991.

[4] D. Dobkin and R. Lipton. *On the Complexity of Computations under Varying Sets of Primitives.* Yale University, Department of Computer Science Technical Report, # 42, 1975.

[5] G. H. Gonnet. *Open Problem Session.* Schloss Dagstuhl Colloquium on Data Structures, Feb 1996.

[6] B. Grünbaum and G. C. Shephard. *Tilings and Patterns: An Introduction.* Freeman, pp. 20, 21, 1989.

[7] F. P. Preparata and M. I. Shamos. *Computational Geometry.* New York: Springer-Verlag, 1985.

[8] M. I. Shamos. *Computational Geometry.* Ph.D. Thesis, Yale University, 1978.

[9] M. I. Shamos and D. Hoey. *Closest-point Problems.* Sixteenth Annual IEEE Symposium on Foundations of Computer Science, pp. 151-162, Oct 1975

[10] G. T. Toussaint. *Computing Largest Empty Circles with Location Constraints.* International Journal of Computer and Information Sciences, pp. 347-558, Volume 12, Number 5, 1983.