FLOCKING OF BIRDS: A SIMULATION STUDY

A Project Report Submitted

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

in

Mathematics and Computing

by

Mudit Tiwari

(Roll No. 162123024)

to the

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

April 2018

CERTIFICATE

This is to certify that the work contained in this project report entitled **"FLOCKING OF BIRDS: A SIMULATION STUDY"** submitted by **Mudit Tiwari** (**Roll No: 162123024**) to Department of Mathematics, Indian Institute of Technology Guwahati towards the partial requirement of the course **MA699 Project** has been carried out by him under my supervision.

It is also certified that, along with literature survey, a few new results are established and simulation studies have been carried by the student under the project.

Turnitin Similarity: 15%

Guwahati - 781 039
April 2018

(Dr. Partha Sarthi Mandal)
Project Supervisor

ABSTRACT

Many things in nature sometimes leaves us astonished by it's beauty, one such beautiful phenomenon can be seen in birds and fishes who are able to organize themselves into larger group so easily. It's very fascinating to learn about the patterns made by these flocks of birds. One of the major achievement in learning this behavior is simulating at on computer. In 1986, Craig Reynolds first simulated flocking of birds. The time complexity of algorithm suggested by him is $O(n^2)$ time. The goal of this project is to simulate the flocking of birds and also to reduce the complexity of the algorithm proposed by Reynolds. We propose a modified algorithm, whose complexity is $O(kn \log n)$.

Contents

1 Introduction **1**

1.1 Goal of the Project 3

1.2 Overview of the Project 3

2 The Flocking Algorithm **4**

2.1 Flocking Rules 4

2.1.1 Avoidance 4

2.1.2 Flock Centering 5

2.1.3 Alignment 7

2.2 Finding Neighbourhood 8

2.3 Simulation of Boids 10

2.4 Limiting the Velocity 11

2.5 Review and Python Based Simulaation 12

2.5.1 Analysis of Algorithm 13

2.5.2 Simulation 13

3 Modified Flocking Algorithm **18**

3.1 Introduction to 2-D Tree 18

3.1.1 Construction of 2-D Tree 19
3.1.2 Median of Medians Algorithm 20
3.1.3 Analysis of Median of Medians Algorithm 21
3.2 Nearest Neighbour Search 22
3.3 Modified Structure of Boid 24
3.3.1 K-Nearest Neighbour Search 25
3.3.2 Updating 2-D Tree 26

4 Conclusion **28**

Bibliography **29**

Chapter 1

Introduction

Animal nature has been of interest to humans since ages. There are many animals like birds, fishes, and herd of sheep who formulate themselves in large group is of very high interest. The behavior shown by birds is called flocking of birds. A large class of birds fly in a certain pattern but Starlings makes the most beautiful patterns while flying (Figure 1.1).

Figure 1.1: Pattern made by typical flock of starlings

In this project, we want to simulate the pattern made by these birds on computer. The first work in this regard was done by Craig Reynolds [1], he coined the term *boids* for a group of bird and proposed few simple rules which were enough to define the motion of any bird. The three rules which were introduced by him are as follows:

Separation : Each bird should maintain some distance from each other so as to avoid a collision.

Cohesion : Keeps each bird close to each other. In other words, it tries to make a cluster of these birds.

Alignment : Keeps the flock moving in a particular direction.

These three rules constitute flocking algorithm. There are a lot of application of flocking algorithm, few are listed below:

- From Tim Burton's Batman Returns(Featured flocking bats) to Disney's The Lion King(Featured Wildebeest stampede) flocking has been used in many movies as well as games to simulate crowds.
- In games, flocking algorithm is widely used to control a mass of characters and their motion.
- Being in the 21st century it is not so strange to think of roads full of driver-less intelligent cars, and flocking can be really helpful in those times during traffic congestion, we can model each car on the road as an individual boid and apply flocking algorithm distributively so as to clear traffic as soon as possible.

- Flocking is also extensively used in screen savers and other graphics application. Flocking is also considered for controlling of drones and UAVs.

1.1 Goal of the Project

First objective of this project is to simulate the bird flocking algorithm [1]. Second objective is to reduce the complexity of algorithm from $O(n^2)$ to $O(kn \log n)$.

1.2 Overview of the Project

This projects simulates the boid model, using above three rules. In Chapter 2, we will describe the three rules in great details along with their implementation details, we will discuss the overall complexity of the algorithm and also demonstrate the simulation. Then in Chapter 3, we will learn some prerequisite for reducing the complexity of algorithm from Chapter 2, and will state the final result.

Chapter 2

The Flocking Algorithm

In the last chapter, we saw three rules to govern the motion of the birds. In this chapter, we will see the flocking algorithm which will use these three rules mainly. We will also see few other subroutines for flocking algorithm.

2.1 Flocking Rules

Here we will see three rules in details along with one algorithm for each rule.

2.1.1 Avoidance

Avoidance means that we want each of our boid to maintain a certain distance R from each other, and hence we are making sure that they do not collide with each other whatsoever. In figure 2.1 we can see that green boid is trying to maintain certain distance from 3 other boids in it's neighbourhood.
We will examine each boid and will make sure that none of other boid is having the euclidean distance less then R from the first boid. If it does

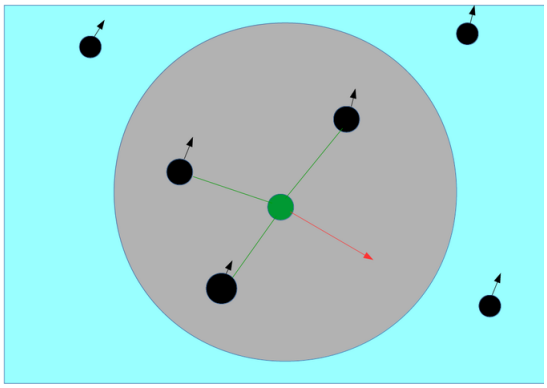


Figure 2.1: Avoidance

happen then we want to push it further away so as to keep our property. Here in Algorithm 1 we've taken a vector c the displacement of each boid

Algorithm 1

```
1: procedure AVOIDANCE(BOID b_j, NBD[ ])
2:     Vector c = 0;
3:     for EACH BOID b in NBD of b_j do
4:         if b ≠ b_j then
5:             if Distance of b and b_j is less than R then
6:                 c = c − (b.position − b_j.position)
7:     return c
```

which is near by. We initialise c to 0 as we want this rule to give us a vector which when added to the current position moves away from those near it.

2.1.2 Flock Centering

In this rule we want our boids to not scatter away from each other. To overcome this we want each boid to steer toward center of mass of it's neighbour. In figure 2.2 the green boid is trying to come at center of mass of it's neighbourhood.

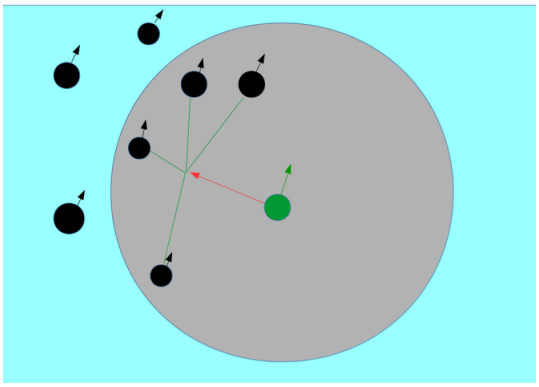


Figure 2.2: Flock Centering

The 'center of mass' is simply the average position of all the birds(Except itself) in neighbourhood. The center of mass is the property of all the birds in neighbourhood and not of the single bird, and hence we want each bird to move towards this center of mass.

Algorithm 2

```
1: procedure COHESION(BOID b_j,NBD[ ])
2:     Vector c;
3:     for EACH BOID b in NBD of b_j do
4:         if b ≠ b_j then
5:             c = c + b.position
6:     % N is the total number of boids in neighbourhood.
7:     c = c/(N − 1) ;
8:     return (c − b_j.position)/100
```

After calculating center of mass on line 7 of Algorithm 2 we want to work out on, how to move boid towards it? We move it 1% of the way towards the center hence we return $c - b_j.position/100$.

2.1.3 Alignment

All boids try to match velocity of neighbourhood boids. By this they are also trying to steer towards the average heading of it's neighbourhood, In figure 2.3 green bird is aligning it to the direction of boids around it. This is very much similar to last rule of *Flock Centering*. Instead of averaging the position of all the boids we are averaging their velocity, and then adding a small portion to boid's current velocity. Here in Algorithm 3 we are adding 1/8 of the current velocity.

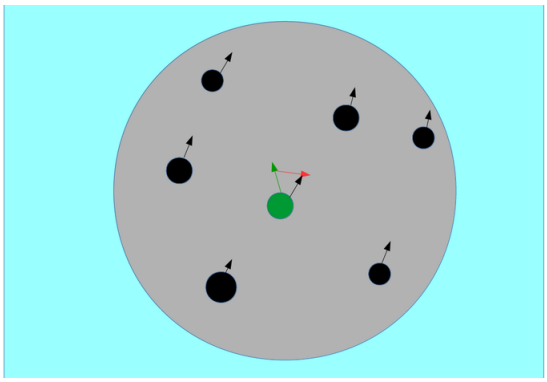


Figure 2.3: Alignment

Algorithm 3

```
1: procedure ALIGNMENT(BOID b_j, NBD[ ])
2:     Vector c;
3:     for EACH BOID b in NBD of b_j do
4:         if b ≠ b_j then
5:             c = c + b.velocity
6:     % N is the total number of boids in neighbourhood.
7:     c = c/(N − 1) ;
8:     return (c − b_j.velocity)/8
```

2.2 Finding Neighbourhood

In Algorithm 1, Algorithm 2 and Algorithm 3 from last section we saw that we need to find a suitable neighbourhood of each boid. Indeed it's a vary crucial step, otherwise each boid will take all other boids in consideration.

There could be (Mainly) 2 notion of neighbourhood of a boid :

1. Every boid whose euclidean distance is less than the specified treshold is considered in the neighbourhood, see Figure 2.4.
2. This is a more elaborate notion of neighbourhood [2] which only considers boids in the current forward direction of boid, see Figure 2.5.

Here is the pictorial explanation of the two types :

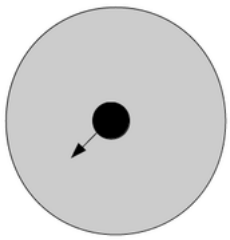


Figure 2.4: Normal Neighbourhood finding from 1st point. Every boid in grey area is a neighbour of that boid

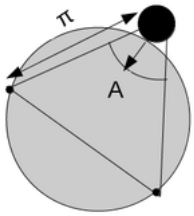


Figure 2.5: Normal Neighbourhood finding from 2nd point. Every boid in grey area is a neighbour of that boid

In Figure 2.5, we get a much more realistic vision of the bird, angle A is boid's angle of vision.

Here Algorithm 4 is to find the array of neighbourhood boids for both cases.

Algorithm 4

```
1: procedure NEIGHBOURHOOD(BOID b_j)
2:     Arr = [ ] % Empty array to put boids in neighbourhood.
3:     for EACH BOID b IN ENVIRONMENT do
4:         if b is in Grey Region then
5:             Append b in Arr.
6:     return Arr
```

Finding grey region in first case is quite trivial, we first define a range , and then simply draw a circle of that radius taking boid's position as center. On the other hand in second case we're dealing with quite natural vision of a boid. For this we're considering two things: (1) Angle of vision A and (2) Approximate range of vision π. Here are the steps to find grey region in second case:

1. Find the velocity vector of boid.

2. Find two new vectors by rotating the velocity vector by angle $A/2$ in counterclockwise direction and $A/2$ in clockwise direction.

3. Find points on those vectors at π distance from boid's position.

4. Find circumcirle passing through these three points, i.e. two found in last step and boid's position. This circle is required grey area.

2.3 Simulation of Boids

After learning three basic rules of simulation in previous section we are ready to learn about motion of each boid. In our simulation we are updating boids position frame wise (WLOG) we are updating at 40 frames per second. One of the basic question at this point is that how it is updating it's position after each frame?

We get the answer to this question in Reynolds [1] paper, he has updated position of each boid by simply adding to it's velocity vector, also he found velocity of each boid by simply adding the result of Algorithm 1, Algorithm 2, Algorithm 3 applied on itself. See Algorithm 5 which is used to move all boids to new position. And finally Algorithm 6 is to start the simulation.

The SIMULATE() procedure in Algorithm 6 puts all boids at a starting location. Purposely they are placed at random location on screen. In third line of Algorithm 6 we are just drawing the boids and then we call MOVE() which moves all boids continuously.

Algorithm 5

```
1: procedure MOVE()
2:     Vector v1, v2, v3;
3:     Boid b ;
4:     for Each Boid b do
5:         NBD =NEIGHBOURHOOD(b)
6:         v1 = AVOIDANCE(b, NBD)
7:         v2 = COHESION(b, NBD)
8:         v3 = ALIGNMENT(b, NBD)
9:         b.velocity = b.velocity + v1 + v2 + v3
10:        b.position = b.position + b.velocity
```

Algorithm 6

```
1: procedure SIMULATE()
2:     while TRUE do
3:         Draw boids 40 times in a second.
4:         MOVE();
```

2.4 Limiting the Velocity

It is a good idea to limit the magnitude of the boids velocities, this way they don't go too fast. Without such limitations, their speed might become arbitrarily large. We are here setting a velocity limit, $vlim$ which is the upper bound of velocities of each boid. Here is a procedure to limit the velocity:

Algorithm 7

```
1: procedure LIMITVELOCITY(BOID b_j)
2:     Integer vlim
3:     if |b_j.velocity| ≥ vlim then
4:         b_j.velocity = (b_j.velocity/|b.velocity|) * vlim
```

Algorithm 7 creates a unit vector by dividing $b_j.velocity$ by its magnitude,

then multiplies this vector by *vlim*. The resulting velocity vector has same direction as the original but it's magnitude has changed. Note that this procedure directly applies on velocity of a boid, so we simply limit velocity after calculating it in line 8 of Algorithm 5. Here is a modified version of Algorithm 5 after updating velocity :

Algorithm 8

```
1: procedure MOVE()
2:     Vector v1, v2, v3;
3:     Bird b ;
4:     for Each Bird b do
5:         NBD = NEIGHBOURHOOD(b)
6:         v1 = AVOIDANCE(b, NBD)
7:         v2 = COHESION(b, NBD)
8:         v3 = ALIGNMENT(b, NBD)
9:         b.velocity = b.velocity + v1 + v2 + v3
10:        LIMITVELOCITY(b)
11:        b.position = b.position + b.velocity
```

2.5 Review and Python Based Simulaation

In this chapter we learned three basic rules for our simulation. These three rules completely defined the motion of each boid. Seeing the natural motion of boid there can be many other tweak than these rules, like Goal seeking , perching etc, anyway it is of very much importance to analyze the algorithm and see the possibility of further improvement.

2.5.1 Analysis of Algorithm

In Algorithm 8 while finding neighbourhood of a boid b_j we were testing all other boids to include it as a nearby neighbour (Within a certain range of boid) or not include it in neighbour set. Which implies for every boid we are testing it n times, therefore the complexity of algorithm is $O(n^2)$ where n is the number of boids in our environment. This does not say that algorithm is slow or fast but as the number of boids increases the complexity increases even faster. Reynolds [2] in 2000 used a special approach using grids to speed up the process, however our approach to increase the efficiency is different. In Chapter 3 we'll introduce K Dimensional tree and we'll use different subroutines to speed up the flocking algorithm.

2.5.2 Simulation

Our simulation is done on TKinter module of python 2.7, each boid is circular with pointed head towards velocity's direction and has black color with vision of 140 degree. The value of R (see Algorithm 1) is 50 pixels. The value of π (see Section 2.2) is 100 pixels. Screen size is: 560 * 560, and initially we're providing just 10 boid and with a left click we can insert as many boids as we please. All these values are variables and can be changed anytime. Screen shots of the simulation in 4 different cases is given starting from next page.

Number of boids : 10 , Value of R : 10 pixels, FPS : 40

First Frame

Second Frame

Third Frame

Fourth Frame

As $R = 10$ in this case, the boids can be seen very close to each other and hence they make different different small groups which flocks independently. In first frame they look distorted but till fourth frame they tend to make patterns.

Number of boids : 20, Value of $R = 10$ pixels, FPS : 40

First Frame

Second Frame

Third Frame

Fourth Frame

This case is similar to last case, only difference is in number of boids.

Number of boids : 10 , Value of R : 40 pixels , FPS : 40

First Frame

Second Frame

Third Frame

Fourth Frame

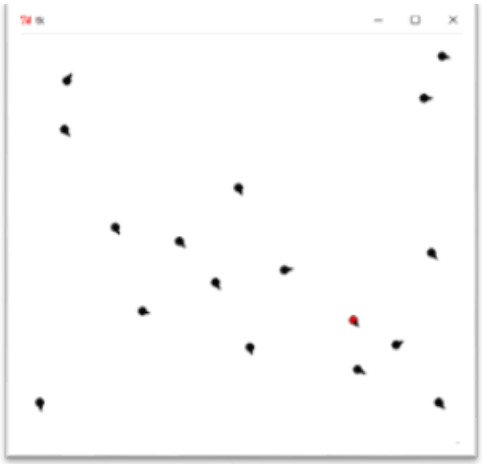
Here in this case value of R is 40, so each boid maintain a minimum distance of 40 from another. So, this time they are looking more apart from each other. In first frame they look distorted but till fourth frame they tend to make patterns.

Number of boids : 20, Value of $R = 40$ pixels, FPS : 40

First Frame

Second Frame

Third Frame



Fourth Frame

This case is similar to last case, only difference is in number of boids.

Chapter 3

Modified Flocking Algorithm

In this chapter we want to improve the complexity of flocking algorithm. For this we will use a data structure K-D Tree. A K-D Tree [3] is a space partitioning data structure and an extension of binary search tree also, every node is a point from space. In our case of applying it to *Flocking Algorithm* we will restrict ourselves to just 2 dimensional tree. Each point in the node will be the boid's current position. In this chapter we'll describe 2-D Trees, their construction and few various operations. Our goal from last chapter was to reduce complexity while finding neighbourhood of any boid, so here we will see algorithm to find neighbours in more efficient way.

3.1 Introduction to 2-D Tree

A 2-D tree is a binary tree in which every node is a point from $\mathbb{R}^2$. Every non leaf node can be thought of as implicitly generating a splitting hyper plane that divides the space into two parts, known as half-spaces. Points to

the left of the hyper plane are represented by the left sub-tree of that node and points right of the hyper plane are represented by the right sub-tree.

3.1.1 Construction of 2-D Tree

There are many different ways to construct a 2-D tree, the main method of 2-D tree construction has the following constraints :

1. As one moves down the tree, one cycles through the axes used to splitting planes. In a 2-D Tree, the root would have an x-aligned plane, the root's children would both have y-aligned plane and root's grand children will again have x-aligned plane and this cycle goes on.

2. Points are inserted by selecting the median of the points being put into the sub tree, with respect to their coordinates in the axis being used to create the splitting plane.

This method leads to a balanced 2-D Tree, in which leaf node is approximately the same distance from the root. Given n points (position of each boid), the Algorithm 9 constructs a 2-D Tree containing those points. Although there was no need to select the median points but it was chosen so as to ensure a balanced 2-D Tree. Finding median using Heap sort or merge sort takes $O(n \log n)$, and constructing the Tree is taking $O(\log n)$. So, overall complexity of constructing a 2-D Tree is $O(n \log^2 n)$.

Theorem 3.1.1. *Worst case complexity of constructing a 2-D Tree is $O(n \log^2 n)$ (if we use sorting to find median), where n is total number of points.*

Algorithm 9

```
1: procedure 2-D-TREE(ARRAY OF POINTS Arr, INT DEPTH)
2:     %Select axis based on depth so that axis cycles through all values.
3:     axis = depth mod 2
4:     Find median by key = axis from Arr
5:     %Create node and construct subtree
6:     node.position = median
7:     node.leftchild = 2-D-TREE(points in Arr before median, depth + 1)
8:     node.rightchild = 2-D-TREE(points in Arr after median, depth + 1)
9:     return node
```

If we find median in linear i.e. $O(n)$ time then construction cost will come down to $O(n \log n)$. In next subsection we present another algorithm to find median in worst case linear time.

3.1.2 Median of Medians Algorithm

The problem of median finding algorithm can be reduced to finding ith smallest element of array. Where i is $n/2$ is n(length of array) is even and $[n/2]+1$ if n is odd. The median of median algortithm [4] is a deterministic algorithm which finds median in linear time. Following are the steps of median of medians algorithm, The algorithm take in a list and an index (of median) - MOM(A,i):

1. Divide the array into sub arrays each of length five (if there are fewer than five elements available for the last array, that is fine)

2. Sort each sub array and determine the median. Sorting very small arrays takes linear time since these sub arrays have five elements, this takes $O(n)$ time.

3. Use the median-of-median algorithm to recursively determine the median of the set of all the medians.

4. Use this median as the pivot element,x. The pivot is an approximate median of the whole list and then each recursive step brings closer to the true median.

5. Reorder A such that all elements less x than are to the left of x , and all elements of A that are greater than x are to the right. This is called partitioning. The elements are in no particular order once they are placed on either side of x. Now let k is the index of x after partitioning.(See Figure 3.1)

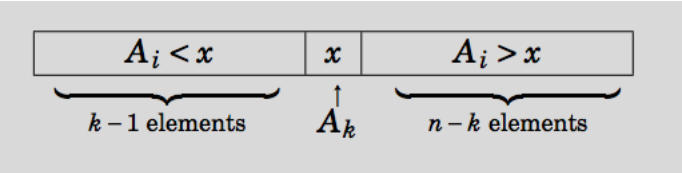


Figure 3.1: Partitioning of A

6. If $i = k$, then return x.

7. If $i < k$, then recur, MOM($A[1, ..., k-1]$,i).

8. If $i > k$, then recur, MOM($A[k+1, .., i]$,$i-k$).

3.1.3 Analysis of Median of Medians Algorithm

Let n be the total number of elements in array, initially in the first step n is divided into $n/5$ sub arrays of 5 elements each. If M is the array of all the

medians from these sub arrays, then M has $n/5$-one median for each $n/5$ sub lists. Let's call the median of the array M, p. So, p is median of medians. Half of the $n/5(=n/10)$ elements in M are less than p. For each these $n/10$ elements, there are two elements that are smaller than it (since these elements were medians in the array of five elements). Therefore $\frac{3n}{10} < p$ and in worst case, the algorithm may have to recur on remaining $7n/10$ elements. The time for dividing the arrays, finding the medians of the sub arrays, and partitioning takes $T(n) = T(\frac{n}{5}) + O(n)$ time, with the recursion factored in, the overall recurrence to describe the median of medians algorithm is $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$, which on applying masters theorem gives total complexity of $O(n)$, which leads to the following theorem.

Theorem 3.1.2. *The worst case complexity of median of medians algorithm is* $O(n)$.

Hence we can now say that to construct a 2-d Tree time required is $O(nlogn)$.
Our main problem which was increasing the time complexity of flocking algorithm was finding neighbours, which was taking $O(n^2)$ time. Here we are describing a algorithm to find nearest neighbour of any point from 2-D, we'll use it to reduce the overall time complexity of flocking algorithm.

3.2 Nearest Neighbour Search

The goal of this problem is to find the closest point from any given point. Finding nearest neighbour is quite simple, the idea is to traverse the whole Tree, but we make two modification to stop searching a space:

1. Let Q be the query point, keep variable of closest point C found so far. We want to stop looking in any sub tree if it is not possible to get any point closer than C.

2. Each node has a bounding box, and hence if distance between query point and bounding box of any node is greater than the current shortest distance than do not look further in sub tree of that node. In Figure 3.2 rectangle is bounding box of node T, d is the distance between rectangle and Q.

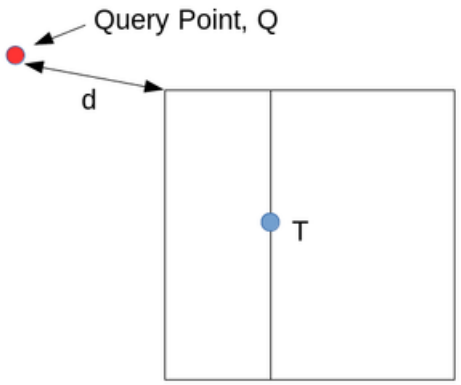


Figure 3.2: Demonstration of finding distance from query point to the bounding box

   If $d > dist(Q, C)$ then no point of subtree rooted at T can be closer to Q than C. So, no reason to search in this subtree.

Now we write the Algorithm 10 to find nearest neighbour of given query point. Although the worst case complexity of above algorithm can reach upto $O(n)$, as we might traverse whole tree in some cases, e.g. When all

Algorithm 10

```
1: procedure NN(POINT Q, KDTREE root, INT axis, RECT BB)
2:     if T==NULL or distance(Q, BB) > best_dist then
3:         return
4:     dist = distance(Q, root.position)
5:     if dist < best_dist then
6:         best = root
7:         best_dist = dist
8:     if Q[axis] < root.position[axis] then
9:         NN(Q, root.left, nextaxis, BB.trimleft(axis, root.position))
10:        NN(Q, root.right, nextaxis, BB.trimright(axis, root.position))
11:    else
12:        NN(Q, root.right, nextaxis, BB.trimright(axis, root.position))
13:        NN(Q, root.left, nextaxis, BB.trimleft(axis, root.position))
```

points are on a circle and query is center of circle. Also, the expected time required to search nearest neighbour is $O(\log n)$.[5]

So far we have learned about the working of flocking algorithm in $O(n^2)$, have seen the construction of 2-D Tree in $O(n \log n)$ and also we saw nearest neighbourhood finding algorithm which took $O(\log n)$ on average. In this chapter we'll see revised version of flocking algorithm.

3.3 Modified Structure of Boid

In previous chapter we defined a boid to have two attributes, position and velocity. Now since we want to put these boids in 2-D Tree we'll need revise the structure of a boid. Now boid also have two pointers to point at their children in Tree, So boid will have 4 attributes (1) It's current position ,(2) It's velocity, (3) left pointer, (4) right pointer. We'll built the whole 2-D Tree using position of all the boids and all the operations will be done with

respect to boids position. Given n boids we can easily build the 2-D Tree in $O(n \log n)$ time using techniques learned before. After constructing there are two problems we're facing, first is to update the 2-D Tree after each frame and other is to find neighbours of boid (Since we were finding just one nearest neighbour). The more important is the later one as it will help us reduce the overall complexity of flocking algorithm. We'll solve this problem using famous machine learning algorithm - *K nearest neighbour search.*

3.3.1 K-Nearest Neighbour Search

K nearest neighbour(kNN) search algorithm is used to find k nearest neighbours of any given query point. The idea is to maintain k closest point instead of just one in Nearest Neighbour Search. We'll use the priority queue of size k, with highest priority been given to farthest point. We'll simply en queue the points in queue and when it is full we'll de queue the farthest point, this way we will have k nearest neighbours in the queue. The Algorithm 11 is for finding k nearest neihgbours. Note that initially value of *best_dist* can be given very large. The expected time complexity of finding nearest neighbour was $O(\log n)$ here for finding k nearest neighbour, expected time complexity is $O(k \log n)$.

In Algorithm 4 we were using only one constraint(predefined distance) to define a neighbourhood of the boid, but if we are using kNN search than instead of using distance we'll give k, which will be the number of boids we want to keep in consideration of those three rules. Now we'll look at the other problem of updating the 2-D Tree.

Algorithm 11

```
1: procedure KNN(POINT Q, KDTREE root, INT axis, RECT BB, INT k)
2:     Priority queue P
3:     %Queue's size is 'k', highest priority is given to farthest point from query.
4:     if T==NULL or distance(Q, BB) > best_dist then
5:         return
6:     dist = distance(Q, root.position)
7:     if P is not full then
8:         P.enqueue(root.position)
9:     elsedist < best_dist
10:        P.dequeue()
11:        P.enqueue(root)
12:        best_dist = dist
13:    if Q[axis] < root.position[axis] then
14:        KNN(Q, root.left, n_axis, BB.trimleft(axis, root.position), k)
15:        KNN(Q, root.right, n_axis, BB.trimright(axis, root.position), k)
16:    else
17:        KNN(Q, root.right, n_axis, BB.trimright(axis, root.position), k)
18:        KNN(Q, root.left, n_axis, BB.trimleft(axis, root.position), k)
```

3.3.2 Updating 2-D Tree

As boids are changing position in each frame we need to update 2-D Tree after calculating new position for each boid. Updating boids is easier task than finding neighbours. Building the tree was taking $O(n \log n)$ which is less than finding neighbour, so we will simply rebuild the 2-D Tree after each frame giving us the cost of updating equal to building the tree. Algorithm 12 is for updating the 2-D Tree after each frame. And Algorithm 13 is the final version of the simulation algorithm which will use MOVE() procedure from Algorithm 14.

Algorithm 12

```
1: procedure UPDATE(Arr)
2:     TWODTREE(Arr, 0)
3:     %See ALgorithm 9
```

Algorithm 13

```
1: procedure SIMULATE()
2:     Array of Boids Arr
3:     while True do
4:         Draw boids 40 times in a second.
5:         T = TWODTREE(Arr, 0)
6:         MOVE(T)
```

Algorithm 14

```
1: procedure MOVE(T)
2:     int k, Vector v1, v2, v3
3:     Array of Boids Arr
4:     for EACH BIRD b do
5:         NBD = KNN(b.position, T,0,k)
6:         v1 = AVOIDANCE(b, NBD)
7:         v2 = COHESION(b, NBD)
8:         v3 = ALIGNMENT(b, NBD)
9:         b.velocity = b.velocity + v1 + v2 + v3
10:        LIMITVELOCITY(b)
11:        b.position = b.position + b.velocity
```

So, we have got a efficient algorithm for flocking and hence we've improved the complexity from $O(n^2)$ to $O(kn \log n)$. Where n is number of boids in environment and k is any small integers.

Chapter 4

Conclusion

In this chapter we will see the overall conclusion along with some interesting applications of flocking algorithm in many different areas.

This project report showed work done by Craig Reynolds, Bentley and others, and then combined them to reduce the overall complexity of flocking algorithm from $O(n^2)$ to $O(kn \log n)$ with n being the number of birds and k being any small integer.

Bibliography

[1] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioural model. SIGGRAPH'87, pages 25–34, ACM, New York, NY, USA, 1987. ACM.

[2] Craig W. Reynolds. Steering behaviours for autonomous characters. Gsme Developers Conference, 2000.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM*, pages 509–517, ACM, New York, NY, USA, 1975. ACM.

[4] Thomas H. Cormen. Probabilistic analysis and randomized algorithms. In *Introduction to Algorithms*, 2009.

[5] Jon Louis Bentley Jerome H Friedman. An algorithm for finding best matches in logarithmic expected time. In *ACM TOMS, Volume 3 Issue 2*, ACM, New York, NY, USA, 1977. ACM.