A Simple Heuristic to Find Efficiently $k$-Nearest Neighbors in Flocking Behaviors

Jae Moon Lee
Dept. of Multimedia Engineering
Hansung University
392 Samsundong 2Ga, Sungbukgu, Seoul
Korea
jmlee@hansung.ac.kr

Hye-Kyung Cho
Dept. of Multimedia Engineering
Hansung University
392 Samsundong 2Ga, Sungbukgu, Seoul
Korea
hkcho@hansung.ac.kr

Abstract
Flocking behaviors are used in games and computer graphics for realistic simulation of massive crowds. Simulation of massive crowds in real time is a computationally intensive task. This intensity mostly comes from the $O(n^2)$ complexity of the traversal algorithm. It is because each agent in crowd has to decide for itself which neighbors fall into its environment. There are several algorithms to enhance the simulation of flocking behaviors. One of the efficient algorithms adapted the characteristic of the flocking behaviors which two agents may share many common neighbors if they are spatially close to each other. It ran up to two times faster than the conventional flocking algorithm based on spatial subdivision method. In this paper we present a noble flocking algorithm that yields an improvement in the execution time over the previous algorithm based on the characteristic. To do this, we analyzed the weakness of the previous algorithm and proposed the simple heuristic to overcome the weakness. A number of experiments were conducted to evaluate the performance of the proposed algorithm. The experimental results showed that the proposed algorithm outperformed the previous method by about 20%.

Key-Words: Flocking behavior, Agent, Spatial subdivision, $k$-nearest neighbors

1 Introduction
Flocking behavior is widely found in nature: birds fly in flocks, fish swim in schools, sheep move as a herd steered by a dog. While engaging in this behavior, many animal groups display structural order, behaving in a way that they appear to move as a single coherent entity, while still changing their shape and direction [1, 2, 3, 4, 5, 9, 10]. In Reynolds’ model [1], each agent makes steering decisions based on the following behaviors: separation - avoid crowding neighbors, alignment - steer towards average heading of neighbors, and cohesion - steer towards average position of neighbors. The three simple behaviors are applied not to the entire flock, but only to local neighbors; each agent has to decide for itself which neighbors fall into its environment.

There are two methods to define the neighbors of an agent in flocking behavior. One defines all the agents within a radius in the space, while the other defines the $k$-nearest neighbors of an agent. The former has an advantage to efficiently find the neighbors rather than the latter, while the latter has an advantage which the number of agents in neighbors is always constant. The former has a disadvantage which does not know the number of agents in neighbors before finding the neighbors. [8] showed that interaction ruling animal flocking behaviors depends on the latter rather than the former. This paper also focuses on finding the neighbors defined like the latter, that is, $k$-nearest neighbors.

A basic implementation of finding neighbors in flocking behavior has the $O(n^2)$ complexity because every agent has to calculate its distance from every other agent in the entire flock in order to decide whether or not it falls into its environment [6, 7, 8, 11]. Even when interactions are limited to some $k$-nearest neighbors, it is still necessary to find the $k$-nearest neighbors out of the total population of $n$, and also has the $O(n^2)$ complexity. An accepted solution to this complexity problem is a bin-lattice spatial subdivision [2, 6, 11]. It reduced this complexity by considerably factors, using adequate data structures for spatial subdivision, achieving interactive frame rates in real-time simulations. If each agent only examines the $k$-nearest neighbors, the algorithm will run in the $O(kn)$ complexity in average.
There are several characteristics in flocking behavior. One of them is that two agents may share many common neighbors if they are spatially close to each other. [11] improved the conventional flocking algorithm by using the characteristic and showed that it ran up to two times faster than the conventional flocking algorithm based on spatial subdivision method. In this paper, we will improve the algorithm proposed in [11] with a simple heuristic. In order to explain the paper simply, the following symbols are used.

$p\cdot q$: Distance between two agents $q$ and $p$.
$|S\cdot q|$: Distance between the agent $q$ and the farthest agent in $S$ from $q$, where $S$ is a set of agents.
$|S|$: Number of agents in $S$, where $S$ is a set of agents.

2 Previous works

2.1 Conventional Flocking Algorithm

One way to find the $k$-nearest neighbors efficiently is to perform a spatial subdivision method. By using this technique, the entire area the flock can move in is divided into a large number of bins. These would be squares in 2D space and cubic cells in a 3D space. In this method, there is a global register which stores which bins contain which agents. Each time an agent moves from one bin to another, it updates its location in this global register. This decreases the complexity of the flocking algorithm enormously, since agents only have to look in the register to see which agents are close enough to possibly be in their local environment.

\[\text{Algorithm: (input: agents, } k)\]

\[
\text{// Construct global register: Cubic}
\]

01: foreach $q$ in agents \{
02: \hspace{0.5em} if (Cubic($q\cdot oldLoc$) != Cubic($q\cdot newLoc$)) \{
03: \hspace{1.0em} Cubic($q\cdot oldLoc$).Delete($q$)
04: \hspace{1.0em} Cubic($q\cdot newLoc$).Insert($q$)
05: \hspace{0.5em} \}
06: $q\cdot oldLoc$ = $q\cdot newLoc$
07: \}

\[
\text{// Find the } k\text{-nearest neighbors and compute}
\]

\[
\text{// separation, cohesion, and alignment forces}
\]

08: foreach $q$ in agents \{
09: \hspace{0.5em} $Q$ = FindNearKnn($q$, $k$, Cubic)
10: \hspace{1.0em} $P$ = FindKnn($q$, $Q$)
11: \hspace{1.0em} $q\cdot force$ = GetSteeringForce($q$, $P$
12: \}

\[
\text{// Compute the new location($q\cdot newLoc$) with } q\cdot force
\]

Fig.1 Algorithm: Conventional Flocking Algorithm

\[\text{Algorithm: based on Spatial Subdivision Method}\]

The pseudo-code for the conventional flocking algorithm based on the spatial subdivision method is given in Fig.1. In order to simulate flocking behavior, the conventional flocking algorithm must be executed at least 30 times a second. The inputs of the algorithm are a set of agents represented as $agents$, the value of $k$. The algorithm is largely divided into two for-loops. The first for-loop between the line 1 and 7 updates the global register shown as $Cubic$ in Fig.1. The second for-loop between the line 8 and 12 finds the $k$-nearest neighbors for each agent and computes the steering force. In particular, finding the $k$-nearest neighbors of an agent is divided into two procedures, $FindNearKnn$ and $FindKnn$ as shown at the line 9 and 10. When given the agent $q$, the value of $k$, the global register $Cubic$ as the inputs, the procedure $FindNearKnn$ finds a set of agents which contains at least the $k$-nearest neighbors of $q$. In order to do this, it first finds the rectangle $C_m$ shown in Fig.2-(a) which is centered at $q$ and contains at least $k$ agents. Let $r$ be the distance between $q$ and $p$, in which $p$ is the farthest agent in $C_m$ from $q$. Finally, it finds all the agents which are within the circle (sphere for 3D) of radius $r$ centered at $q$. These are represented as the small circles in Fig.2-(b).

At the line 10 in Fig.1, the procedure $FindKnn$ specifies the ascending order of the agents in $Q$ based on the distance from $q$ and then selects the $k$-nearest agents of $q$ in $Q$. At the line 11 in Fig.1, the procedure $GetSteeringForce$ computes the steering force which consists of separation, cohesion and alignment forces.

Its details are referenced from [3, 6].
spatially close to each other. It also showed experimentally that the closest two agents share 89% of their neighbors in average. [11] proposed the condition which can check whether two agents share their neighbors for a given set of agents or not.

Fig. 3 is the pseudo-code of the flocking algorithm proposed in [11]. If the lines between the line 13 and 22 are deleted in Algorithm 2, it is exactly the same to Algorithm 1. The main idea of Algorithm 2 is represented in the lines between the line 13 and 22. Without loss of generality, the k-nearest neighbors of q are considered as the candidate agents close to q as shown in the line 13. The condition of the line 19 was proved by Theorem in [11]. To check this condition, P is first computed at the lines between the line 16 and 18, which contains the k-nearest neighbors of p for the agents in Q. If P passes the condition of the line 19, the k-nearest neighbors of p are directly computed from P as shown in the line 20.

Algorithm 2 (input: agents, k)
01-07: same to those of Algorithm 1

// Find the k-nearest neighbors and compute separation, cohesion, and alignment forces
08: foreach q in agents {
09: if(q.force != 0) continue
10: Q = FindNearKnn(q, k, Cubic)
11: P = FindKnn(q, k, Q)
12: q.force = GetSteeringForce(q, P)
13: foreach p in k-nearest neighbors of q {
14: if(p.force != 0) continue
15: P = {\Phi}
16: foreach t in Q {
17: if(|Q-q|-|p-q|\geq|t-p|) P.add(t)
18: }
19: if(|P| < k) continue
20: P = FindKnn(p, k, P)
21: p.force = GetSteeringForce(p, P)
22: }
23: }

Algorithm 3 (input: agents, k)
01-12: same to those of Algorithm 2

// Find the k-nearest neighbors and compute separation, cohesion, and alignment forces
13: foreach p in k-nearest neighbors of q {
14: if(p.force != 0) continue
14-1: if(q.at(Q-q)|\leq|p-q|) continue
15: P = {\Phi}
16: foreach t in Q {
17: if(|Q-q|-|p-q|\geq|t-p|) P.add(t)
18: }
19: if(|P| < k) continue
20: P = FindKnn(p, k, P)
21: p.force = GetSteeringForce(p, P)
22: }
23: }

3 Proposed Flocking Algorithm

In this section, we propose the algorithm to improve Algorithm 1 by applying a simple heuristic. One of the weaknesses of Algorithm 2 is to discard the agent p which does not satisfy the condition of the line 19 even if the cost of executing the lines between the line 16 and 18 is so expensive. We analyzed the ratio of the number of agents to pass the line 19 to the number of agents to pass the line 15, which is defined as follows:

\[ r = \frac{\text{number of agents to pass line 19}}{\text{number of agents to pass line 15}} \times 100. \]

Table 1. Ratio r with various values of n and k

<table>
<thead>
<tr>
<th>k</th>
<th>n</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1n</td>
<td>39.8%</td>
<td>36.1%</td>
<td>34.1%</td>
<td>31.6%</td>
<td>27.6%</td>
<td></td>
</tr>
<tr>
<td>0.2n</td>
<td>27.3%</td>
<td>27.5%</td>
<td>27.6%</td>
<td>27.3%</td>
<td>26.9%</td>
<td></td>
</tr>
<tr>
<td>0.3n</td>
<td>26.9%</td>
<td>27.0%</td>
<td>27.2%</td>
<td>27.1%</td>
<td>26.9%</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 shows the measured values of the ratio r with various values of n and k. Even if the values of r were measured with the various values of n and k, they appear almost identical values. The average value in Table 1 is about 29.4%. That is, 70.6% of agents to pass the line 15 are discarded at the line 19 even if P was computed at the line 16 and 18. In order to improve Algorithm 2, it implies that it is necessary to reduce agents to be discarded at the line 19.

In this paper, we propose the simple heuristic which discards the agent before the line 15 which has the high possibility to be discarded at the line 19. Let consider \( p_1 \) and \( p_2 \) which are elements of the k-nearest neighbors of q and \( p_1 \) is much nearer to q than \( p_2 \), that is, \( |p_1-q| < |p_2-q| \). It is obvious that \( p_1 \) has higher possibility to pass the line 19 than \( p_2 \). It is because agent t at the line 17 has higher possibility to be added to P for \( p_1 \) rather than \( p_2 \). It is the heuristic to be adapted in this paper. That is, if an agent p sufficiently far from q, it is discarded before the line 15.

Algorithm 1 (input: agents, k)
01-07: same to those of Algorithm 1

// Find the k-nearest neighbors and compute separation, cohesion, and alignment forces
08: foreach q in agents {
09: if(q.force != 0) continue
10: Q = FindNearKnn(q, k, Cubic)
11: P = FindKnn(q, k, Q)
12: q.force = GetSteeringForce(q, P)
13: foreach p in k-nearest neighbors of q {
14: if(p.force != 0) continue
15: P = {\Phi}
16: foreach t in Q {
17: if(|Q-q|-|p-q|\geq|t-p|) P.add(t)
18: }
19: if(|P| < k) continue
20: P = FindKnn(p, k, P)
21: p.force = GetSteeringForce(p, P)
22: }
23: }

Fig. 3 Algorithm 2: Flocking Algorithm in [11]

Fig. 4 Algorithm 3: Proposed Flocking Algorithm
The proposed algorithm is shown in Fig.4. It is the same to Algorithm2 except for the line 14-1 in Fig.4. In the line 14-1, the symbol \( \alpha \) has the value between 0 and 1. If it has 0, the proposed algorithm is the same to Algorithm3, because all the agents can pass the line 14-1, while if it has 1, the proposed algorithm is the same to Algorithm1, because any agents never pass the line 14-1. As the performance of the proposed algorithm is greatly dependent on the value of \( \alpha \), the value of \( \alpha \) must be carefully chosen by the experiments.

4 Performance Comparisons

A number of experiments were conducted to evaluate the performance of the proposed algorithm. The experiments were performed on the personal computer with Intel(R) Core (TM2) Duo CPU and 4GB RAM, running Windows 7 operating system. The three algorithms introduced in the paper were implemented on windows XP using visual Studio 2005. C++ and STL (standard template library) were used. OpenGL was used for rendering of the agents and terrains.

In order to simplify the description of performance comparison, we use the following notations.

\[
T_X : \text{Execution time of Algorithm}_X \text{ for 1,000 frames, which unit is second}
\]
\[
\xi_{XY} : \text{Performance improvement ratio of Algorithm}_X \text{ to Algorithm}_Y \text{ which is defined as } \frac{T_X - T_Y}{T_X} \times 100(\%)
\]

In simulating an animation, it is desirable to measure the performance of the entire or partial animation rather than that of a shot in the animation. We also measured the execution time while each algorithm is consecutively executed by 1,000 frames.

The first experiment was executed to find the proper value of \( \alpha \) in the line 14-1. For the value of \( k=50 \), the experiments were executed when the number of agents, \( n \) was varied between 128 and 2,048 and the value of \( \alpha \) was varied between 0 and 1. The experimental results are shown in Fig.5. X-axis represents the values of \( \alpha \) and y-axis represents the performance improvement ratios, \( \xi_{23} \). For the low values of \( \alpha \), most of agents cannot pass the line 14-1 and so the performance of Algorithm3 is nearly the same to that of Algorithm1. In this case, Algorithm1 has worse performance than Algorithm3. This situation is shown at Fig.5 when the value of \( \alpha \) is 0.1. For the high values of \( \alpha \), most of agents can pass the line 14-1 and so the performance of Algorithm3 is nearly the same as that of Algorithm2. It is appeared in Fig.5 when the value of \( \alpha \) is 0.9. From Fig.5, we can easily choose 0.3 as the proper value of \( \alpha \), because the performance of Algorithm3 is generally best at \( \alpha=0.3 \).

Fig.5 \( \xi_{23} \) for variances of \( \alpha \)

According to the values of \( k \) and \( n \), the various performance improvement ratios were measured as shown in Fig.6. X-axis represents the values of \( k \) and y-axis represents the values of the performance improvement ratio, \( \xi_{23} \). Fig.6 shows that \( \xi_{23} \) is slightly increased when the values of \( n \) are increased, but its effect may be ignorable specially when the values of \( k \) are large. Fig.6 also shows that \( \xi_{23} \) is slightly increased as \( k \) is increased. It is because the number of agents is large at the line 13 in Fig.4 so that more agents can pass the condition of the line 14-1 in Fig.4. However, Fig.6 shows that \( \xi_{23} \) is not seriously dependent on the values of \( k \) and \( n \). From results of experiments in Fig.6, we can know that the proposed
algorithm enhances the performance of Algorithm$_2$ by about 20% on average.

5 Conclusions

Flocking behaviors are used in games and computer graphics for realistic simulation of massive crowds. Simulation of massive crowds in real time is a computationally intensive task. One of the efficient algorithms adapted the characteristic of the flocking behaviors which two agents may share many common neighbors if they are spatially close to each other. In this paper we proposed a noble flocking algorithm that yields an improvement in the execution time over the previous algorithm based on the characteristic. To do this, we analyzed the weakness of the previous algorithm and proposed the simple heuristic to overcome the weakness.

A number of experiments were conducted to evaluate the performance of the proposed algorithm. First from the experiments, we found the proper value of parameter used in the proposed heuristic. Then, by using its value, the proposed algorithm was experimentally compared to the previous algorithm. The experimental results showed that the proposed algorithm outperformed the previous method by about 20%.

Acknowledgements

This work was supported by the National Research Foundation of Korea grant funded by the Korean Government (MEST) (NRF-2009-0076003) and the grant from the research program (10033663) of the Ministry of Knowledge Economy (MKE) of Korea.

References


