단순한 휴리스틱을 사용하여 무리 짓기에서 이웃 에이전트 탐색방법의 성능 개선※

장자순, 이재문
한성대학교 멀티미디어공학과
jzstammy@hanmail.net, jmlee@hansung.ac.kr

An Improvement of Finding Neighbors in Flocking Behaviors by Using a Simple Heuristic

Jiang Zi Shun, Jae Moon Lee
Dept. of Multimedia Engineering, Hansung University

요 약

무리 짓기는 대규모 무리의 사실적인 시뮬레이션으로 게임이나 컴퓨터 그래픽에서 자주 사용된다. 이러한 대규모 무리의 실시간 시뮬레이션은 계산 집약적 작업이기 때문에 효율적인 알고리즘에 대한 많은 연구들이 있었다. 본 논문에서는 기존의 효율적인 무리 짓기 알고리즘이 불필요한 계산을 포함하고 있다는 사실은 실험적으로 찾아내고, 간단한 휴리스틱으로 이러한 단점을 개선하는 새로운 알고리즘을 제안한다. 제안된 방법의 성능을 평가하기 위하여 많은 실험을 수행하였다. 실험의 결과는 제안하는 알고리즘이 기존의 효율적인 알고리즘에 비하여 평균 약 21%정도 성능을 개선한다는 것을 보였다.

ABSTRACT

Flocking behaviors are frequently used in games and computer graphics for realistic simulation of massive crowds. Since simulation of massive crowds in real time is a computationally intensive task, there were many researches on efficient algorithm. In this paper, we find experimentally the fact that there are unnecessary computations in the previous efficient flocking algorithm, and propose a noble algorithm that overcomes the weakness of the previous algorithm with a simple heuristic. 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 efficient algorithm by about 21% on average.

Keywords : Flocking, k-Nearest Neighbor Agent, Spatial Subdivision Method(무리 짓기, k개의 가장 가까운 이웃 에이전트, 공간 분할 방법)

접수일자 : 2011년 07월 25일 심사완료 : 2011년 08월 16일
교신저자(Corresponding Author) : 이재문
※ 본 연구는 한성대학교 교내연구비 지원과제임.

1. 서 론

무리 짓기란 자연에서 관찰되는 많은 현상으로 떼를 지어 다니는 새들, 어군속의 물고기들, 목장에서 양들, 음식물을 찾아다니는 개미들 등의 움직임이 모두 무리 짓기에 속한다. 이러한 동물들의 그룹은 무리속의 각각이 지속적으로 그들의 방향과 형태를 바꿀지라도 전체적으로 하나의 응집된 형태를 가지는 특성을 보인다. 무리속의 움직이는 하나의 개체를 보이드 또는 에이전트라고 한다[1,2,3,4,5,8]. 레이놀드 모델에서는 무리를 구성하는 각 에이전트들은 분리힘, 응집힘, 정렬힘을 바탕으로 조종에 대한 결정을 한다[1]. 이후 대부분의 모델들에서도 구체적인 조종력 계산 방법은 다소의 차이가 있으나 레이놀드 모델에 따라 무리 짓기를 시뮬레이션 한다[3,5].

분리힘은 주변의 무리들과 충돌을 피하는 힘이며, 응집힘은 무리로부터 홀로 떨어지지 않으려는 힘이고, 정렬힘은 무리와 동일한 방향으로 이동하려는 힘이다. 따라서 이러한 힘은 단순히 하나의 에이전트에 대하여 독립적으로 계산할 수 있는 것이 아니다. 근본적으로는 무리에 속한 모든 에이전트들을 고려하여 이러한 힘을 계산하는 것이 원칙이지만 [1,3,5,6,9]등 대부분의 연구에서는 영향을 주는 이웃 에이전트들만 고려하여 이러한 힘을 계산한다.

무리짓기에서 이웃 에이전트들을 정의하는 두 가지 방법이 있다[3,5,7,9,10]. 하나는 일정 반경 r을 정하여 대상 에이전트를 중심으로 반경 r내에 존재하는 모든 에이전트들을 이웃 에이전트로 정의하는 방법이며, 다른 하나는 임의의 상수 k를 정하여 대상 에이전트를 중심으로 가장 가까이 존재하는 k개의 에이전트들을 이웃으로 정의하는 것이다. 두 방법 사이에는 trade-off가 있다[7,9]. 전자는 일정한 범위 내에 존재하는 모든 에이전트들이 이웃 에이전트가 되므로 이웃 에이전트를 찾는 것이 비교적 쉬운 장점이 있으나, 최악의 경우 모든 에이전트가 이러한 반경 내에 존재하는 경우 모든 에이전트가 모든 이웃 에이전트로 고려되어야 하는 단점이 있다. 이것은 상기 세 가지 힘을 계산할 때 속도 면에서 심각한 문제를 야기한다. 후자의 경우 주변에 존재하는 에이전트가 적은 경우 매우 먼 거리의 에이전트도 포함하여야 하므로 이웃 에이전트를 찾는 비용이 클 수도 있는 단점이 있는 반면, 항상 이웃의 수가 k개로 고정되어 있기 때문에 세 가지 힘을 계산하는 속도가 일정하다는 장점이 있다. 따라서 무리 짓기의 시뮬레이션 환경에 따라 두 방법 중의 하나가 선택될 수 있다. [7,10]에서는 무리 짓기의 최종 목표를 생존을 위한 욕구로 정의하고 상기 두 방법중 어느 방법이 더욱 이러한 목표를 유지하는데 적절한지 실험적으로 연구하였고, 결과적으로 후자의 방법이 더욱 적절하다는 결론을 내렸다. 본 논문에서도 후자의 방법을 사용하는 것으로 한다.

무리 속에 n개의 에이전트들이 존재할 때 매 프레임마다 각각의 에이전트에 대하여 k개의 이웃 에이전트를 탐색하여야 한다. KD-트리 등과 같은 특별한 보조 자료구조가 없는 경우 하나의 에이전트에 대한 k개의 가장 가까운 이웃을 찾는 비용은 $O(n)$이다. 이것은 하나의 에이전트에 대하여 n-1개의 다른 모든 에이전트와 거리를 계산하고 그 중에서 가장 가까운 거리에 있는 k개의 에이전트를 탐색하여야 하기 때문이다. 따라서 모든 에이전트에 대하여 이를 계산하여야 하므로 그 비용은 $O(n^2)$이 된다[1,5]. 이러한 비용은 매 프레임마다 계산되어야 한다. 즉, 게임이 최소한 초당 30프레임의 화면을 보인다면 초당 30번씩 $O(n^2)$ 만큼 계산되어야 한다. 따라서 대규모 무리 짓기와 같이 n이 큰 경우 이 비용은 매우 크다. 이러한 비용을 줄이기 위하여 [1,5,9]에서는 공간분할 방법을 사용해 왔다. 이 방법은 매 프레임마다 에이전트들을 분할된 공간에 매핑 함으로서 이웃 에이전트들을 효율적으로 탐색하도록 한다. 이 경우의 비용은 $O(kn)$이 된다. 대부분의 경우 k가 n에 비하여 매우 작으므로 이 방법은 무리 짓기의 성능을 크게 향상시킨다.

무리 짓기에서 무리들의 다양한 특성들이 있다. 그러한 특성중의 하나가 공간적으로 가까이 있는 에이전트들은 그들의 이웃들을 공유한다는 것이다. 예를 들어 공간상에서 에이전트 p와 q는 가장 가까이 있는 에이전트라고 하자. 이 경우 q는 p의 가장 가까운 이웃인 동시에 p는 q의 가까운 이웃이다. 또한 p와 q가 가까이 있다면 p의 k개의 가장 가까운 이웃들의 대부분이 q의 k개의 가장 가까운 이웃에 속하게 될 것이다. [9]에서는 실험적으로 이러한 특성을 분석하였고, 이 특성을 적용하여 기존의 공간분할 방법의 성능을 약 2배 정도 개선 시켰다. 본 논문은 [9]에서 제안된 알고리즘이 기존의 공간분할 방법의 성능을 크게 개선시켰음에도 불구하고 불필요한 계산을 많이 포함하고 있다는 사실을 실험적으로 분석하였다. 또한 간단한 휴리스틱을 적용하여 이러한 불필요한 계산을 줄임으로써 [9]에서 제안된 알고리즘의 성능을 개선한다. 다음 기호들은 논문의 효율적인 기술을 위하여 사용된다.

$|p-q|$: 두 에이전트 p, q 사이의 거리

$|Q-q|$: 에이전트의 q를 중심으로 에이전트의 집합 Q의 모든 소속 에이전트를 포함하는 가장 작은 구(求)의 반경

$||Q||$: 에이전트의 집합 Q에서 포함된 에이전트의 수

2장은 기존의 무리 짓기 알고리즘들을 소개하며, 3장에서는 본 논문에서 제안하는 단순한 휴리스틱과 이를 이용한 알고리즘을 설명한다. 4장에서는 실험을 통하여 제안하는 알고리즘이 기존의 알고리즘에 비하여 얼마나 많은 성능개선 효과가 있는지를 비교 설명하며 5장에서 결론을 논한다.

2. 관련 연구

2.1 기존의 공간분할 알고리즘

무리 짓기에서 에이전트들은 지속적으로 움직이므로 매 프레임마다 그들의 위치가 변한다. 따라서 이웃 에이전트들을 찾기 위하여 KD-트리 등과 같은 인덱스를 사용하는 경우 삽입/삭제 비용이 너무 크다. [1,5,9]에서 제시된 공간분할 방법은 무리 짓기의 이러한 특성을 반영한 가장 단순한 자료 구조이다. 먼저 무리 짓기가 일어나는 공간을 하나의 큰 큐빅(2D의 경우 그리드)으로 나타내고, 하나의 큰 큐빅을 여러 개의 작은 셀로 분할한다. 다음 모든 에이전트에 대하여 그들의 위치에 따라 이들을 큐빅내 해당 셀에 할당한다. 이렇게 하면 게임 공간상의 모든 에이전트들은 그들의 위치에 따라 큐빅내 하나의 셀에 할당되게 된다. 분할된 셀의 수나 특정한 위치로 에이전트들의 쏠림 현상 등으로 하나의 셀에 여러 에이전트가 할당될 수도 있다. 이때 하나의 에이전트에 대하여 이웃 에이전트들을 찾는 경우 전체 에이전트를 대상으로 찾는 것이 아니라 현재의 에이전트가 속한 셀에서부터 시작하여 인접한 셀들만 탐색하여 이웃 에이전트를 구한다.

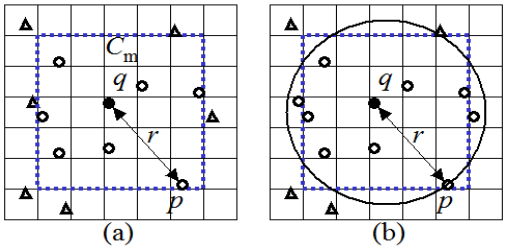
알고리즘1: Inputs: *agents, k,* Outputs: *None*

```
    // 공간 재분할
01: foreach q in agents {
02:     if(Cubic(q.oldLoc)!=Cubic(q.newLoc){
03:         Cubic(q.oldLoc).Delete(q)
04:         Cubic(q.newLoc).Insert(q)
05:     }
06:     q.oldLoc= q.newLoc
07: }
    // 이웃 에이전트 탐색 및 조종힘 계산
08: foreach q in agents {
09:     Q= FindNearKnn(q,k,Cubic)
10:     P= FindKnn(q,k,Q)
11:     q.force= GetSteeringForce(q,P)
12: }
    // 이동행동: 각 에이전트에 대한 위치 계산
```

[그림 1] 공간분할 무리 짓기 알고리즘

[그림 1]은 공간분할 무리 짓기에 대한 의사코드이다. 알고리즘의 입력은 에이전트의 집합, 이웃 에이전트들의 수이다. 알고리즘은 크게 두개의 반

복문으로 구성된다. 첫 번째 반복문은 공간분할 자료구조(*Cubic*)에 각 에이전트들의 새로운 위치정보를 반영하여 공간분할 데이터를 생성하는 것이다. 에이전트들이 매 프레임마다 위치를 변경하기 때문에 매 프레임마다 이들을 계산을 하여야 한다. 두 번째 반복문은 각 에이전트에 대하여 그들의 이웃 에이전트들을 찾고, 새로운 조종힘을 계산하는 것이다.



[그림 2] k=3일 때 이웃 에이전트를 찾는 예: (a) C_m과 p(C_m내에 7개의 에이전트 포함) (b) 알고리즘1에서 라인 9의 Q에 포함될 9개의 에이전트들

[그림 1]에서 두 번째 반복문이 공간분할 무리짓기의 핵심이다. 특히 라인 9에서 함수 *FindNearKnn*이 공간분할 자료구조인 *Cubic*을 이용하여 근사적인 이웃 에이전트들을 찾는 것이다. 이의 설명을 쉽게 하기 위하여 [그림 2]와 같이 2D 공간에서 알고리즘이 어떻게 동작하는지를 설명한다. 먼저 [그림 2](a)와 같이 q를 중심으로 최소한 k개의 에이전트를 포함하는 사각형 C_m을 찾는다. [그림 2](a)에서 k는 3이고 따라서 점선으로 표시된 사각형이 C_m이 된다. [그림 2](a)에서는 C_m내에 7개의 에이전트가 포함되어 있다. 다음 C_m내의 에이전트들 중 q와 가장 멀리 떨어진 에이전트([그림 2](a)에서 p)를 찾는다. [그림 2](a)에서처럼 q와 p사이의 거리를 r이라 하자. 마지막으로 [그림 2](b)와 같이 q를 중심으로 반경 r내에 존재하는 모든 에이전트들을 찾아 리턴한다. 이 경우 찾아진 에이전트는 k개보다 많을 수 있다. [그림 1]의 라인 10에서 함수 *FindKnn*은 단순히 Q 내의 에이전트들 중 q와 가장 가까운 k개의 에이전트만 선택하여 P에 저장한다. 3D 공간에서는 [그림 2](a)의 C_m은 직육면체가 되며, [그림 2](b)의 원은 구가 될 것이다. 라인 11에 있는 함수 *GetSteeringForce*는 에이전트 q와 이 에이전트의 k개의 가장 가까운 이웃 에이전트들을 이용하여 q의 분리힘, 응집힘, 정렬힘을 계산하는 함수이다. 이의 자세한 내용은 [5]에서 참조하였다.

2.2 이전 무리짓기 알고리즘

[9]에서는 공간적으로 인접한 에이전트끼리는 많은 이웃에이전트들을 공유한다는 특징을 실험적으로 분석하였고 이를 활용하여 기존의 공간 분할 알고리즘의 성능을 약 2배 정도로 개선하였다. 이 논문에서의 핵심 개념은 [그림 1]의 라인 9에서 *FindNearKnn*에서 찾아진 q의 이웃 에이전트 후보들(Q)에는 q와 매우 가까운 p의 이웃 에이전트들도 포함되어 있을 가능성이 높다는 것이며 이런 경우 별도의 *FindNearKnn* 호출 없이 p의 이웃 에이전트들을 찾는 것이 핵심 개념이다.

```
알고리즘2: Inputs: agents, k, Outputs: None
1-7  // 알고리즘1과 동일
     // 이웃 에이전트 탐색 및 조종힘 계산
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={φ}
16:          foreach t in Q {
17:              if(|Q-q|-|p-q|>=|t-p|) P.add(t)
18:          }
19:          if(||P|| < k) continue;
20:          P= FindKnn(q,k,Q)
21:          q.force= GetSteeringForce(q,P)
22:      }
23:  }
     // 이동행동: 각 에이전트에 대한 위치 계산
```

[그림 3] 개선된 공간분할 알고리즘

[그림 3]은 [9]에서 제안한 알고리즘이다. 라인 1~7까지의 공간 재분할 코드는 [그림 1]의 알고리즘1과 동일하므로 생략하였다. 라인 9의 코드를 제외하면 라인 8~12 사이의 내용도 [그림 1]의 알고리즘1과 동일하다. 따라서 [그림 3]은 [그림 1]의 알고리즘1에 라인 13~22사이의 코드가 추가된 것이다. 즉, q와 매우 인접한 에이전트 p에 대하여 p의 이웃 에이전트들 구하는 것이다. [9]에서는 q의 k개의 가장 가까운 이웃 에이전트 중에서 임의의 에이전트 p가 라인 17의 조건을 만족하는 에이전트의 수가 k개 이상이면 Q내에 p의 k개의 가장 가까운 에이전트가 존재한다는 것을 정리를 사용하여 증명하였다.

[그림 3]에서 볼 수 있듯이 라인 13~22 사이에서 에이전트 p가 라인 19의 조건을 적용할 때 라인 20에 도달하는 경우 이 에이전트는 함수 *FindNearKnn*의 호출없이 이웃 에이전트를 구하였으므로 성능을 개선하는 것이다.

3. 개선된 공간분할 알고리즘

3.1 단순 휴리스틱

[그림 3]의 알고리즘2가 [그림 1]의 알고리즘1에 비하여 우수하나, 성능 개선의 여지가 있다. 알고리즘2에서 하나의 취약점은 라인 19의 조건 때문에 라인 20에 도달하지 못하는 경우 라인 16~18에서 실행한 결과가 아무런 의미가 없어지는 것이다. 이것은 라인 16~18을 실행한 후에야 에이전트 p의 이웃 에이전트가 Q의 내부에 있는지 여부를 알 수 있기 때문이다. 본 논문에서는 실험을 통하여 알고리즘2에서 얼마나 많은 에이전트들이 라인 16~18을 실행하고도 라인 20에 도달하지 못하는지 분석하였다. [표 1]은 이에 대한 분석 결과이다. 여기서 비율 *ratio*은 다음과 같이 계산하였다.

$$ratio = \frac{\text{라인 20에 도달한 에이전트의 수}}{\text{라인 15를 통과한 에이전트의 수}} \times 100$$

[표 1] n, k에 따른 비율 *ratio*

k	n				
	128	256	512	1024	2048
0.1n	39%	36%	34%	31%	27%
0.2n	27%	27%	27%	27%	26%
0.3n	26%	27%	27%	27%	26%

[표 1]에서 n의 값은 128, 256, 512, 1024, 2048로 하였고, 각 n의 값에 대하여 이웃 에이전트의 수 k는 10%, 20%, 30%로 변하도록 한 후 [그림 3]의 라인 15를 통과한 에이전트의 수와 라인 20에 도달한 에이전트의 수에 대한 비율을 표시한 것이다. n과 k의 값에 따라 다소의 차이는 있으나 평균적으로 30%의 에이전트만이 라인 20에 도달함을 알 수 있다. 이것은 라인 16~18 사이의 계산량이 많음에도 불구하고 그 계산의 약 70%는 소용없게 된다는 것이다.

본 논문은 알고리즘2의 이러한 약점을 단순한 휴리스틱을 사용하여 개선하는 것이다. 알고리즘2의 약점을 극복하는 하나의 방법은 라인 16~18을 계산하기 이전에 임의의 에이전트 p에 대하여 이 에이전트가 라인 20에 도달할 가능성을 판단하여 그 가능성이 낮은 경우 라인 15 이전에 이 에이전트를 제외시키는 것이다. 예를 들어 q의 이웃 에이전트 중 p_1, p_2 두 에이전트를 고려하자. 여기서 p_1은 q와 매우 가까이에 있고, p_2는 q와 매우 먼 거리에 있다고 가정하자. 즉, $|p_2-q| \gg |p_1-q|$인 경우이다. 따라서 라인 17의 조건에서 $|Q-q|-|p_1-q| \gg |Q-q|-|p_2-q|$가 되므로 p_2보다는 p_1일 때 라인 17의 조건을 만족하는 에이전트 t가 더 많게 된다. 이것은 p_1이 p_2보다 라인 20에 도달할 가능성이 높다는 의미한다. 본 논문에서는 p_2와 같은 에이전트들을 15라인 이전에 제거함으로써 알고리즘2의 성능을 개선한다. 이것이 본 논문에서 제안하는 단순한 휴리스틱이다.

3.2 제안된 알고리즘

제안된 알고리즘은 매우 단순하다. [그림 4]에서 볼 수 있듯이 [그림 3]의 알고리즘2에서 단순히 라인 14′만 추가하였다. 라인 14′에서 α의 값은 [0, 1] 사이이다. 또한 $|p-q|$는 에이전트 p와 q사이의 거리를 나타낸다. 따라서 라인 14′는 에이전트 p가 q로부터 멀리 떨어져 있어 그 거리가 $\alpha|Q-q|$보다 큰 경우, p는 라인 20에 도달할 가능성이 매우 적다고 판단하여 미리 제거하는 것이다.

알고리즘3의 라인 14′에서 α값의 선택은 매우 중요하다. α값이 0인 경우 알고리즘3은 어떠한 에이전트도 라인 15를 통과할 수 없으므로 알고리즘1과 동일하게 되며, α값이 1인 경우 모든 에이전트가 라인 15를 통과할 수 있으므로 알고리즘2와 동일하게 된다. 따라서 본 논문에서 제안된 알고리즘은 α값의 선택에 따라 기존의 알고리즘인 알고리즘2보다 성능이 저하될 가능성도 있다. 본 논문에서는 이러한 α값을 실험을 통하여 결정한다.

```
알고리즘3: Inputs: agents, k, Outputs: None
1-12 // 알고리즘2와 동일

13:     foreach p in k-nearest neighbors of q {
14:         if(p.force != 0) continue
14':        if(α|Q-q|<|p-q|) continue
15:         P={φ}
16:         foreach t in Q {
17:             if(|Q-q|-|p-q|>=|t-p|) P.add(t)
18:         }
19:         if(||P|| < k) continue
20:         P= FindKnn(q,k,Q)
21:         q.force= GetSteeringForce(q,P)
22:     }
23: }
    // 이동행동: 각 에이전트에 대한 위치 계산
```

[그림 4] 제안된 알고리즘

4. 실험에 의한 성능 비교

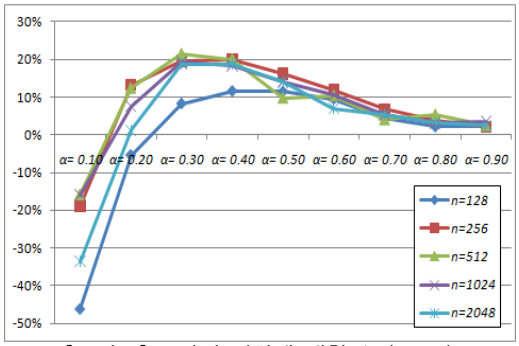
적절한 α값을 찾고, 기존의 알고리즘과 제안된 알고리즘의 성능 비교를 하기 위하여, 알고리즘1, 알고리즘2 및 알고리즘3을 구현하였다. 모든 알고리즘들은 윈도우즈상에서 Visual Studio 2005를 사용하여 구현하였다. 구현 언어로는 C++와 STL(standard template library)을 사용하였으며 에이전트에 대한 렌더링을 위하여 OpenGL을 사용하였다. 실험은 펜티움4 2.5GHZ CPU와 2GB 메모리를 가진 개인용 컴퓨터에서 실행하였다.

성능 측정의 변수로 에이전트의 수 n과 이웃 에이전트의 수 k를 사용하였다. 이것은 에이전트의 수는 게임 내용이나 게임 규모에 따라 정해지며, 이웃 에이전트의 수는 에이전트들의 행동을 제어하는데 사용될 수 있기 때문이다[3,7]. 모든 실험에서 성능 측정의 요소로 1,000 프레임을 연속적으로 실행할 때 소요되는 시간을 측정하였다. t_i는 알고리즘i의 실행에 대한 측정된 시간을 의미한다. 알고리즘i에 대한 알고리즘j의 성능 개선 비율(ξ_{ij})은 다음과 같이 정의한다.

$$\xi_{ij} = \frac{(t_i - t_j)}{t_i} \times 100(\%)$$

첫 번째 실험은 [그림 4]에 있는 라인 14'의 α 값의 적절한 선택을 위한 것이다. 본 논문의 주된 목적이 알고리즘2의 성능 개선이므로 주어진 환경에서 가장 큰 성능 개선 비율(ξ_{23})을 주는 α 값을 선택한다. 실험을 간단히 하기 위하여 k의 값은 50으로 고정하였고, n의 값은 128에서 2,048까지 값을 변경하였다. 이러한 환경에서 α 값을 0.1에서 0.9까지 변경하면서 ξ_{23}를 측정하였다. 측정 결과는 [그림 5]에서 나타나 있다. [그림 5]에서 x축은 α의 값을 나타내며 y축은 ξ_{23}의 값을 나타낸다. [그림 5]에서 볼 수 있듯이 α값이 작은 경우 ξ_{23}값이 음수이므로 제안된 알고리즘의 성능이 오히려 나쁘게 나타나는데 이것은 α 값이 작은 경우 제안된 알고리즘이 알고리즘1과 같게 되기 때문이다. 반대로 α 값이 증가하여 0.9인 경우에는 ξ_{23}의 값이 거의 0으로 수렴하는 것으로 나타난다. 이것에 대한

이유는 α 값이 큰 경우 제안된 알고리즘이 알고리즘2와 같게 되기 때문이다. [그림 5]에서 알 수 있듯이 α의 값이 0.3일 때 제안된 알고리즘이 가장 좋은 성능을 나타낸다는 것을 알 수 있다. 따라서 본 논문은 α의 값을 0.3으로 선택한다.



[그림 5] α값의 변화에 대한 ξ_{23}(k=50)

[표 2] n, k에 따른 성능 개선 비율 ξ_{13} 및 ξ_{23}

n	$k=0.1n$		$k=0.2n$		$k=0.3n$	
	ξ_{13}	ξ_{23}	ξ_{13}	ξ_{23}	ξ_{13}	ξ_{23}
128	29%	12%	44%	16%	49%	16%
256	39%	15%	47%	21%	59%	19%
512	46%	22%	54%	23%	60%	23%
1024	54%	25%	62%	25%	69%	24%
2048	59%	26%	68%	26%	75%	22%

두 번째 실험은 α의 값을 0.3으로 고정한 상태에서 제안된 알고리즘이 기존의 알고리즘1, 알고리즘2에 대하여 얼마나 많은 성능을 개선하는지를 측정하는 것이다. 실험에서 에이전트의 수 n은 128에서 2,048사이의 값으로 변경시켰고, 각 선택된 n 값에 대하여 이웃 에이전트의 수 k의 값은 n 값의 10%, 20%, 30%가 되도록 하였다. 실험의 결과는 [표 2]에 나타나 있다. [표 2]는 n 값이 증가함에 따라 성능 개선 비율도 증가함을 보이고, k 값에는 성능 개선 비율이 거의 일정함을 보인다. 이것은 n 값이 증가할수록 [그림 4]의 라인 13~22를 실행하는 에이전트의 대상이 증가하게 되는데 알고리즘2에 비하여 알고리즘3은 [그림 4]의 라인 20에 도달할 가능성이 적은 에이전트들을 라인 14′에서 제거하기 때문이다. [표 2]에서 ξ_{13} 및 ξ_{23} 값들을 평균하면 각각 55%, 21%이다. 즉, 이것은 제안하는 알고리즘이 기존의 공간 분할 알고리즘과 [9]에서 제안한 알고리즘에 비하여 평균적으로 각각 약 55%, 21%의 성능 개선이 있음을 의미한다.

5. 결 론

무리 짓기는 대규모 에이전트들의 사실적인 시뮬레이션이 적용되는 게임이나 영화 등에서 많이 사용되는 기술이다. 이러한 대규모 에이전트들의 시뮬레이션은 많은 컴퓨팅 시간을 요구하기 때문에 효율적인 알고리즘이 많이 연구되고 있다. 기존의 잘 알려진 알고리즘은 무리 짓기의 특성을 이용하여 크게 성능 개선을 시켰으나 불필요한 계산을 많이 하는 단점이 있었다. 본 논문은 이러한 단점을 분석하고, 이를 줄이는 간단한 휴리스틱을 적용하여 성능을 개선하는 알고리즘을 제안하였다.

제안된 방법의 성능은 실험적으로 기존의 방법과 비교되었다. 다양한 에이전트의 수와 이웃 에이전트의 수에 대하여 동일한 환경에서 기존의 알고리즘들과 제안하는 알고리즘을 실험하였으며, 그 결과 제안하는 알고리즘이 기존의 알고리즘들에 비하여 상대적으로 평균적으로 약 55%, 21%정도 성능을 개선한다는 사실을 알 수 있었다.

참고문헌

[1] Reynolds, C. W., Flocks, Herds, and Schools: A Distributed Behavioral Model, In Proceedings of SIGGRAPH 87, 21(4), 1987, pp. 25-34.

[2] Reynolds, C. W., Interaction with Groups of Autonomous Characters. In Proceedings of the Game Developers Conference, San Francisco, California, 2000, pp. 449 - 460.

[3] Iain D. Couzin, Jens Krause, Richard James,

Graeme D. Ruxton and Nigel R. Franks., Collective Memory and Spatial Sorting in Animal Groups. J. theory Biol., 2002, pp. 1-11.

[4] Jadbabaie, J. Lin, A.S. Morese, Coordination of groups of mobile autonomous agents using nearest neighbor rules. IEEE Transactions on Automatic Control, 48(6), 2003, pp. 988 - 1001.

[5] Buckland Mat., Programming Game AI by Example. Wordware Publications, ISBN 1556220782, 2005.

[6] G.Paulikas, D.Rubliauskas. Movement of Flocked Subpopulations in Distributed Genetic Programming. Information Technology And Control, Vol. 34, No. 4, 2005, pp 338 - 344.

[7] Ballerini M., N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, V. Zdravkovic, Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study. In Proceedings of the National Academy of Sciences, Vol. 105, No. 4, 2008, pp. 1232-1237.

[8] Seongdong Kim, Christoph Hoffmann and Jae Moon Lee, An Experiment in Rule-based Crowd Behavior for Intelligent Games. In Proceedings of Fourth International Conference on Computer Sciences and Convergence Information Technology, 2009, pp. 410-415.

[9] Jae Moon Lee, An Efficient Algorithm to Find k-Nearest Neighbors in Flocking Behavior. Information Processing Letters, 110, 2010, pp. 576-579.

[10] Jian Ma, Wei-guo Song, Jun Zhang, Siu-ming Lo, Guang-xuan Liao, “k-Nearest-Neighbor interaction induced self-organized pedestrian counter flow”, Physica A 389, 2010, pp. 2101~2117.

장 자 순 (Jiang Zi Shun)

2010년 한서대학교 컴퓨터공학과(학사)
2011년 한성대학교 멀티미디어공학과(석사과정)

관심분야 : 게임프로그래밍, 음성인식

이 재 문 (Lee, Jae Moon)

1986년 한양대학교 전자공학과(학사)
1988년 한국과학기술원 전기및전자공학과(석사)
1992년 한국과학기술원 전기및전자공학과(박사)
1994년-현재 한성대학교 멀티미디어공학과 교수

관심분야 : 기계학습, 게임프로그래밍, 감성컴퓨팅