韓國 컴퓨터情報學會 論文誌
第10卷 第4號, 2005. 9.

2005-10-4-4-23

게임 적용을 위한 Dynamic Programming 알고리즘 길찾기

이 세 일*

Dynamic Programming Algorithm Path-finding for Applying Game

Se-Il Lee*

요 약

게임 맵에서 NPC들이 목표 위치로 이동하기 위하여 A* 알고리즘을 비롯한 다양한 알고리즘들을 이용하여 탐색해 왔다. 그 중 가장 많이 사용하는 알고리즘은 탐색속도가 빠른 A*이다. 그러나 A*에는 다음과 같은 문제점들을 가지고 있다. 첫 번째는 무작위로 변화하는 맵에서는 변할 때마다 모든 계산을 다시 해야 하며 잘못될 경우에는 목표를 찾지 못할 경우도 있다. 두 번째는 장애물 같이 NPC에 피해를 입히는 위험 요소들이 위치하는 곳을 피하여 이동하기가 힘이 든다. 물론 위험 요소들에게 가중치를 부여하여 가중치가 높은 곳은 이동하지 않게도 할 수 있지만 위험 요소들의 근처에 이동하는 경우에는 제어하기 힘이 든다.

이러한 문제점을 해결하기 위하여 본 논문에서는 Dynamic Programming을 이용하여 길찾기 알고리즘에 적용하였다. 적용한 결과 무작위로 변화하는 맵 상의 변화에 잘 적응하였으며, NPC들이 자신의 위험 요소들을 멀리 피해가는 모습을 볼 수 있었다. 또한 A*와 비교에서도 좋은 결과가 나왔다.

Abstract

In order to move NPC's to target location at game maps, various algorithm including A* has been used. The most frequently used algorithm among them is A* with fast finding speed. But A* has the following problems. The first problem is that at randomly changing map, it is necessary to calculate all things again whenever there are any changes. And when calculation is wrong, it is not possible to search for target. The second problem is that it is difficult to move avoiding dangerous locations damaging NPC such as an obstruction. Although it is possible to avoid moving to locations with high weight by giving weight to dangerous factors, it is difficult to control in case NPC moves nearby dangerous factors.

In order to solve such problems, in this thesis, the researcher applied Dynamic Programming to path-finding algorithm. As the result of its application, the researcher could confirm that the programming was suitable for changes at the map with random change and NPC's avoided the factors being dangerous to them far away. In addition, when compared to A*, there were good results.

▸ Keyword : Dynamic Programming, A*, Artificial Intelligence Game, Tile Map

---

• 제1저자 : 이세일
• 접수일 : 2005.07.13, 심사완료일 : 2005.09.10
* 공주대학교 컴퓨터공학과 박사과정

Ⅰ. 서 론

온라인 게임의 등장과 발전으로 몇몇 사람들은 인간 플레이어를 상대할 지능적인 게임 제작에 열중하지 않아도 충분히 가능할 것이라고 생각하였다[1, 2]. 왜냐하면, 플레이어와 플레이어의 대결만으로도 충분히 재미를 느낄 수 있을 것이라고 생각했기 때문이다. 그러나 게임의 규모가 방대해지면서 인공지능은 예전에 사용했던 기술 이상으로 인공지능 기술이 필요하게 되었다. 예를 들어 NPC(Non Player Character)들이 원하는 목적지를 향하여 이동하려 하지만 잘못된 길로 이동하거나 목적지를 찾지 못하는 상황이 늘어나게 되면 플레이어들은 그 게임에 흥미를 조금씩 잃어 갈 것이다.

길찾기 알고리즘은 RPG(Roll Playing Game) 게임에서 가장 많이 사용하는 방법 중에 하나이다. 그러나 대부분의 길찾기 알고리즘은 다음과 같은 문제점을 가지고 있다. 첫 번째는 현재의 상황만을 고려하여, 즉 정적인 상황을 전제로 알고리즘 적용한다는 것이다. 그러므로 상황이 변화하는 시점에서 다시 알고리즘을 계산하게 되어 속도가 느려지거나 최악의 경우에는 잘못된 결과를 초래하게 된다. 두 번째는 최단 경로의 이동에만 집중되어 있다. 특별한 위험성이 없으면 좋은 결과가 나오지만 이동하는 경로에 위험한 장애물이나 함정 등이 도사리고 있다면 최단의 경로가 반드시 좋은 결과가 될 수 없다. A* 알고리즘도 예외는 아니다. 게임 중 자주 이러한 문제들이 일어나게 되면 플레이어들은 게임에 흥미를 잃게 될 것이다.

본 논문에서는 게임 상의 마법사들이 마술을 이용하여 갑작스러운 장애물의 출현으로 인한 맵(Map) 상의 환경이 변화여도, 이들의 변화에 대처할 수 있는 길찾기 알고리즘에 적용하기 위하여 Dynamic Programming 알고리즘을 사용한다. Dynamic Programming 알고리즘을 정사각형 타일맵(Tile Map)에서 적용한 결과 A* 알고리즘 보다 위험성이 적은 최적의 길을 찾아가며, 동적인 이동 상황에도 잘 적응하는 것을 알 수 있었다.

2장에서는 관련연구에 대하여 기술하고, 3장은 최적의 길찾기에 Dynamic Programming 알고리즘 적용하는 과정을 보이며, 4장에서는 실험과 결과를 보이고, 5장은 결론과 향후 적용 분야를 제시한다.

Ⅱ. 관련연구

2.1 인공지능 게임

게임에서 인공지능 분야의 역할은 NPC의 행동을 보다 지능적인 행동으로 보이도록 한다. 인공지능 기술로는 길찾기, 유한상태 기계, 의사결정 트리, 강화 학습, 다층 퍼셉트론, 인공 신경망 등을 이용하여 NPC의 사실성과 지적 능력을 한 단계 높이는데 많은 도움을 주었다. 특히 제일 간단할 것 같으면서 무시할 수 없는 길찾기 알고리즘은 인공 지능 문제해결에서 중요한 수단이 되었다[3, 4].

2.1.1 A* 알고리즘

A* 알고리즘은 g(Goal, 골), h(Heuristic, 휴리스틱), f(Fitness, 적합도)의 3가지 속성을 가지고 있다. g, h, f의 목적은 현재 노드까지의 경로가 얼마만큼이나 가능성을 가지고 있는지를 평가하기 위한 것이다. f를 구하기 위해서는 g의 값과 h의 값을 구해야 한다. g는 현재 노드까지 온 거리를 계산하고, h는 정해진 값이 아니라 추정치를 계산하여야 한다. A*가 길찾기에 많이 쓰이는 이유는 깊이 우선 탐색이나 너비 우선 탐색보다 훨씬 빠르며, 이 둘의 단점도 가지고 있지 않기 때문이다. 실제 게임에서 원하는 목적 노드를 빨리 찾을 뿐만 아니라 패스도 최단 경로가 된다. 그러나 A*는 무작위로 변경되는 지형에서나 위험을 감지하고 피해갈 수 있는 탐색 경로를 찾기는 쉽지 않다[5-7].

2.1.2 깊이 우선 탐색

깊이 우선 탐색(Depth First Search)은 출발 노드로부터 시작하여 노드를 계속적으로 확장하여 가장 최근에 생성된 노드로 먼저 확장시키는 탐색기법이다. 깊이 우선 노드는 목표 노드가 없는 곳을 계속 따라갈 수 있으므로 상황에 따라 상위 노드로 되돌아가는 백트래킹(Backtracking)이 필요하다[8-10]. 저장 공간이 작은 곳에 목표 노드가 깊은 단계에 있을 경우 목표를 빨리 구할 수 있는 장점을 가지고 있으며, 단점으로는 목표가 없는 경로에 깊이 빠질 우려가 있고 목표에 이르는 경로가 다수인 경우 얻어진 해가 최단 경로가 아닐 수도 있다.

2.1.3 너비 우선 탐색

너비 우선 탐색(Breadth First Search)은 맵 상에서 사용하는 알고리즘 중에서 가장 기본적인 알고리즘이다. 너비 우선 탐색은 현재 노드에서 한 칸 거리에 있는 모든 노드들을 먼저 처리하고[8-10], 다음에 두 칸, 세 칸 이런 식으로 길찾기에 성공하여 목표 노드에 도착할 때까지 실행된다. 장점은 목표에 이르는 경로가 다수인 경우에도 최단경로를 보장할 수 있고 목표가 존재하면 반드시 찾을 수 있다. 단점으로는 노드의 수가 늘어나면 탐색시간이 비현실적이고 기억공간에 대한 요구가 과중된다는 것이다.

2.2 타일맵

캐릭터들이 지면에 밟고 위치하기 위하여 기본적으로 필요한 것이 맵니다. 타일맵은 정사각형, 직사각형, 육각형, 마름모(그림 1) 등이 존재한다.

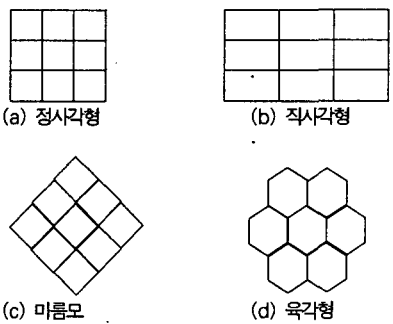


그림 1. 타일맵의 종류
Fig 1. Types of tile map

타일들은 겹쳐지지 않기 위하여 일정한 크기를 가지게 된다. 일정한 규격을 가진 타일은 규칙 적인 격자 단위를 가지며 바닥 레이어에 위치되며, 모든 바닥 레이어에 빠짐없이 위치하여야 한다[11].

2.3 Dynamic Programming 알고리즘

Dynamic Programming(동적 계획법)은 해결하고자 하는 큰 문제를 독립된 작은 문제로 나눈 다음 이 작은 문제들 사이의 반복되는 관계를 찾아내어 메모리를 이용하여 순환적으로 큰 문제를 해결하는 알고리즘이다. 모든 방법이 이 알고리즘에 적용되는 것이 아니라 어떤 문제가 최적이면 그 부분 문제도 최적이 되어야 하는 최적화의 원리가 성립되어야 한다. 이 알고리즘을 개발하기 위해서는 먼저 최적해의 구조를 찾은 후 최적해의 값을 재귀적으로 정의하여야 한다. 다음으로 최적해의 값을 상향식(Bottom-up)으로 계산하여 최적의 해를 구할 수 있다[7, 12-15]. Dynamic Programming이 많이 사용하는 분야는 행렬-체인 곱의 문제, 재귀적 해를 구하는 문제, 중복되는 반복 등의 문제에 적용하고 있다.

III. 최적의 길찾기에 Dynamic Programming 알고리즘 적용

길찾기 알고리즘을 게임에 적용하기 위해서 가장 중요한 것은 무엇보다 빨리 원하는 목적지를 찾아내는 것이다. 그러나 가장 빠른 길을 찾아내어도 NPC들의 이동시 장애물이나 못된 마법사를 만나서 에너지를 잃게 된다면 빠른 길을 찾는 것이 별로 도움이 되지 않는다. 또한 맵 상에서 NPC에 방해되는 요소들이 무작위로 변경되어도 길찾기에 무척 힘이 든다. 위와 같은 문제들을 해결하여 게임에 적용하면 좀 더 흥미로운 게임을 할 수 있다.

3.1 탐색을 위한 제반 사항

우선 정사각형 타일맵(그림 2)을 이용하기 위하여 배열을 사용한다. 목표점은 'T'로 정했지만 출발점은 어느 곳에서 시작해도 아무 문제가 없다.

1	1	1	1	1	1	1	1	1	T
1	30	30	1	1	1	10	10	1	1
1	30	30	1	1	1	10	10	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	50	50	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	20	20	1	1	1
1	1	1	1	1	20	20	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

그림 2. 비용을 이용한 위치 지정
Fig 2. Decision of location using costs

(그림 2)는 NPC의 이동 경로에 비용을 주어 탐색에 이용하였다. 탐색 방법은 현재 위치에서 가장 적은 비용이 계산된 곳으로 NPC를 움직여 목표에 도착하는 것이다. 산이나 장애물들에게는 높은 숫자를 적용하였고, 포악한 장애물에게는 비용을 더 주었으며, 비용이 낮을수록 NPC에게 적은 에너지를 빼앗는다. <표 1>은 각 비용에 대한 NPC의 피해 정도이다.

표 1. 비용에 대한 피해 정도
Table 1. Degree of damage for costs

비용	피해	비고
50	매우 위험	사망
30	중상	50% 소비
20	경상	20% 소비
10	경미한 상처	10% 소비
1	평범한 길	없음
T	목표	

또한 NPC가 이동하기 위해서는 이동 경로에 대한 제약 조건이 필요하다. 제약 조건이 없다면 NPC는 원하는 목표 위치에 도달하지 않고 잘못된 길을 헤매고 다닐 수도 있기 때문이다. 올바른 길을 유도하기 위해 제약 조건을 <표 2>과 같이 적용하여 NPC의 이동을 제한하였다.

표 2. NPC 이동 확률값
Table 2. Probability for NPC moving

NPC의 이동	이동 위치	확률값
오른쪽	오른쪽	0.5
	위쪽	0.25
	아래쪽	0.25
왼쪽	왼쪽	0.5
	위쪽	0.25
	아래쪽	0.25
아래쪽	아래쪽	0.5
	오른쪽	0.25
	왼쪽	0.25
위쪽	위쪽	0.5
	오른쪽	0.25
	왼쪽	0.25

3.2 Dynamic Programming 알고리즘의 길찾기 흐름도

(그림 3)은 최적의 방법으로 목표점을 찾기 위한 길찾기 흐름도이다.

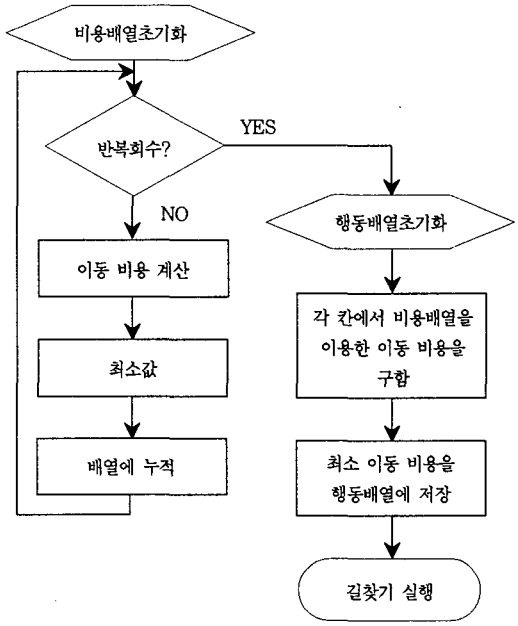


그림 3. 길찾기 흐름도
Fig 3. Flow chart for path-finding

각 칸들의 값은 배열에 저장하고 알고리즘을 이용하여 배열의 내용을 계속 변화하여 추적하며, 그 이동 비용을 계산한 후 최소값을 구하고 비용배열에 누적하고, 반복회수가 종료되면 행동배렬 값을 구하게 된다. 적절한 반복회수는 몇 번이라고 정하지는 않지만, 최소한 NPC가 목표물을 찾아 이동하는 경로의 수보다는 움직임이 많아야 한다. 또한 반복회수가 많으면 많을수록 정확한 결과를 얻을 수 있다. 모든 반복이 끝나게 되면 행동배열의 각 칸에서 이동할 수 있는 비용배열의 값을 계산한다. 계산된 값 중에서 최소값을 선택하여 행동배열에 저장하며, 최소값이 여러 개인 경우에는 정해진 순서대로 선택한다. 마지막으로 행동배열에 저장된 최소값을 따라 이동하게 되면, 최적의 길을 따라 목표점에 도달한다.

Ⅳ. 실험 및 결과

본 논문에서 실험은 펜티엄 IV, 2.8Ghz, 512MB의 환경에서 Windows XP 버전의 운영체제에서 Visual C++ 6.0을 이용하여 설계하고 실험하였다. (그림 4)는 길찾기 프로그램의 가장 중요한 부분인 looping() 함수의 일부분이다. looping() 함수는 Dynamic Programming 알고리즘을 반복 수행하는 부분이다.

```
void Cdp::looping()
{
   .
   .
  for (int i=0; i<10; i++)  //행 반복
  {
     for (int j=0; j<10; j++) // 열 반복
     {
        if ((i==m_row) && (j==m_col)) // 목표
            bi = 0,   m_best[i][j] = 'T';
           .
           .
        else if (bi_U <= _min(bi_D, bi_L, bi_R))
            bi = bi_U, m_best[i][j] = '^';
        // 위쪽 화살표 지정
        else if (bi_D <= _min(bi_U, bi_L, bi_R))
            bi = bi_D, m_best[i][j] = 'v';
        // 아래쪽 화살표 지정
        else if (bi_L <= _min(bi_R, bi_D, bi_U))
        bi = bi_L, m_best[i][j] = '<';
        // 왼쪽 화살표 지정
        else if(bi_R <= _min(bi_U, bi_D, bi_L))
        bi = bi_R, m_best[i][j] = '>';
        // 오른쪽 화살표 지정
        m_value.setValue(i, j, bi + m_map.Tage(i, j));
        // 배열에 값을 저장 함
     }
  }
}
```

그림 4. 알고리즘
Fig 4. Algorithm

길찾기 알고리즘을 이용하여 1회전 하였을 경우 (그림 5)와 같은데, 이것은 (그림 2)의 초기값과 같다.

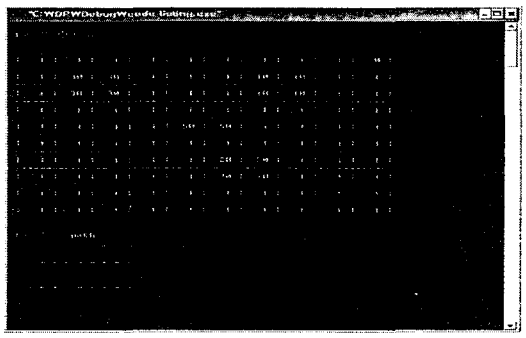


그림 5. 초기값 지정
Fig 5. Decision of initial value

2회전 한 후의 결과는 (그림 6)과 같으며, 6행 6열의 값이 11이 나오는 이유는 다음과 같다. 위쪽으로 이동하기 위해서는 <표 2>에서 지정된 확률값 0.5에 1회 반복한 테이블의 위쪽에 위치하는 값 50을 곱하고, 왼쪽의 확률값 0.25와 1을 곱하고, 오른쪽의 확률값 0.25와 1을 곱하여 모두 더하게 된다. 그러면 위쪽으로 이동해야할 비용이 결정된다. 이와 같은 방법으로 오른쪽, 왼쪽 그리고 아래의 이동 비용을 모두 계산하면 <표 3>의 결과가 나온다. 결과 중 가장 작은 비용을 선택한 아래쪽 값 10과 1회 반복한 테이블의 비용1을 더하여 11이 된다.

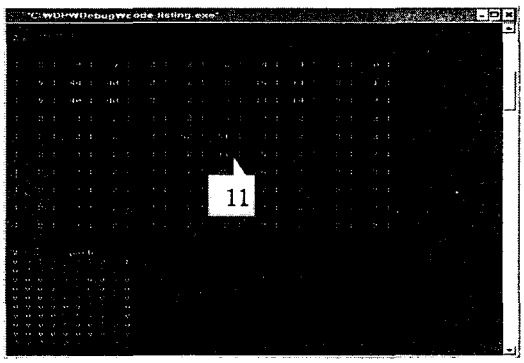
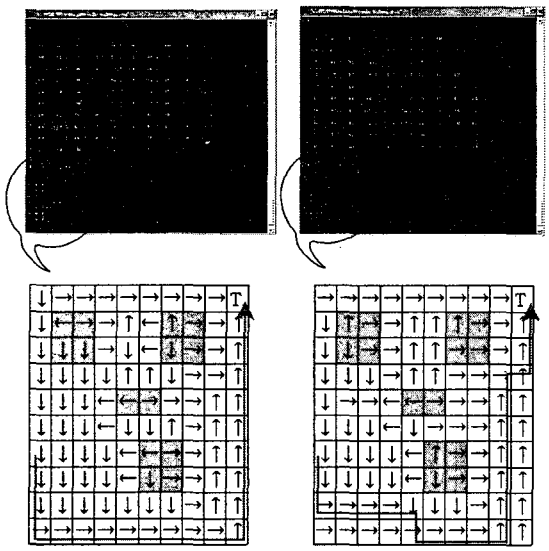


그림 6. 2회전 반복 결과
Fig 6. Repeated result at second

표 3. 이동 계산식
Table 3. Move calculate

위치	식	값
위쪽	0.5*50+0.25*1+0.25*1	≒25
왼쪽	0.5*1+0.25*50+0.25*20	≒18
오른쪽	0.5*1+0.25*50+0.25*20	≒18
아래쪽	0.5*20+0.25*1+0.25*1	≒10

이 알고리즘은 반복할 때마다 좀 더 나은 결과가 나오는 것을 (그림 7)로 알 수 있다. (그림 7)의 (a)는 장애물을 피해서 이동하지만 아주 멀리 이동하여 목표점를 찾아가는 것을 볼 수 있다. 범위가 넓어지면 더 넓게 돌게 된다. 그러나 44회 반복한 (b)는 장애물을 피해 아주 그럴듯하게 길을 찾아가는 것을 볼 수가 있다.



a. 15회 반복 b. 44회 반복

그림 7. 15회와 44회 반복 결과
Fig 7. Repeated result at 15th and 44th time

그러나 어느 정도 반복하면 더 이상 좋은 결과가 나오지 않는다는 것을 (그림 7)의 (b) 그림과 (그림 8) 그림을 비교 하여 알 수 있다.

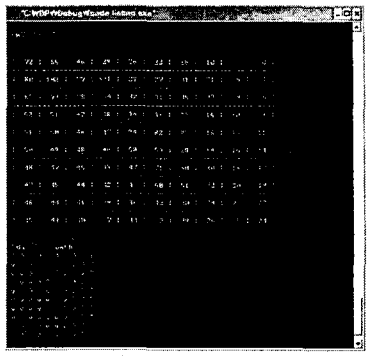
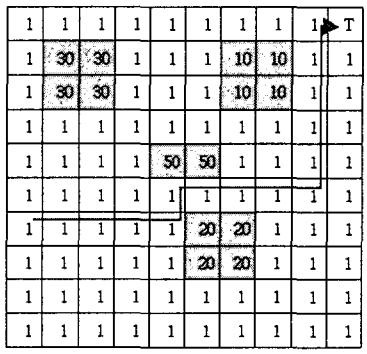
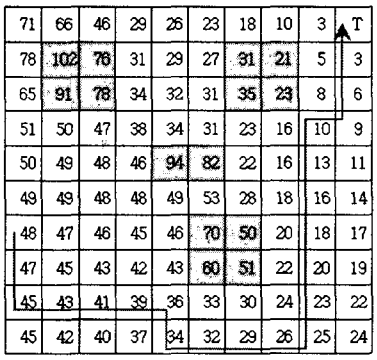


그림 8. 50회 반복 결과
Fig 8. Repeated result at 50th

또한 (그림 9)에서는 44회 반복한 Dynamic Programming 알고리즘과 A* 알고리즘을 비교했다. A* 알고리즘과 Dynamic Programming 알고리즘은 비용이 낮은 곳을 선택하여 이동하는 것을 알 수 있다. A*는 최고의 빠른 길을 선택하였지만 NPC에게 피해를 입힐 장애물 옆을 이동하기 때문에 위험성을 감수해야 한다. 그러나 Dynamic Programming 알고리즘은 위험성을 피하여 최적의 길을 찾아가는 것을 알 수 있다. 또한 맵의 위치가 무작위로 변화하게 된다면 A* 알고리즘은 다시 처음부터 이동 경로를 계산해야 하지만 Dynamic Programming 알고리즘은 위에서 계산한 것과 같이 변화된 상황에서 그 비용을 직접 계산함으로써 A* 알고리즘 보다 더 좋은 결과를 얻을 수 있다.



a. A* 알고리즘으로 이동



b. Dynamic Programming으로 이동

그림 9. A*와 비교 결과
Fig 9. Results compared to those of A*

V. 결론

A* 알고리즘은 무작위로 변화하는 맵 상에서 생각보다 최선의 길을 찾아 이동하기 힘이 들었으며, 또한 NPC에 피해를 입힐 수 있는 장애물이나 함정들을 피해서 이동하기도 힘이 들었다. 그러나 Dynamic Programming 알고리즘을 길찾기에 적용하면 무작위로 변경하는 맵 상에서도 자연스럽게 동작하는 것을 볼 수 있었고, 자신에게 해를 입히는 장애물들을 피해 최적의 길을 찾을 수 있었다.

Dynamic Programming 알고리즘은 마법의 능력을 가지고 있는 마법사나 접근하기에 위험성이 있는 몬스터들과 싸우는 게임에 적용하기 좋으며, 테트리스와 같이 변화가 자주 발생하는 게임에 적용할 수 있다.

그러나 Dynamic Programming 알고리즘은 반복적인 작업으로 속도가 떨어지는 문제점을 가지고 있지만, 좀 더 사실적이고 동적인 면을 강조하는 RPG 게임이나 어드벤처 게임에 적용하면, 문제점보다는 장점이 부각되어 사실감 있는 게임이 될 것이다.

참고문헌

[1] 서정호, 정광호, 게임 인공지능의 형태와 앞으로의 발전 방향, 한국게임학회, 2002

[2] 권일경, 이상용, 퍼지 플로킹 기반의 보이드 행동 모델링, 퍼지 및 지능시스템학회, 2004

[3] Daniel Fu, Ryan Houlette, Stottler Henke, "Putting AI In Entertainment : An AI Authoring Tool for Simulation and Games", IEEE Intelligent and System July/August, Vol.17, No.4, 2002

[4] Daniel Johnson, Janet Wiles, "Computer Games With Intelligence", IEEE International Fuzzy Systems Conference, 2001

[5] 이세일, 타일맵에서 A* 알고리즘을 이용한 유닛들의 길찾기 방법 제안, 한국컴퓨터정보학회, 논문지, 제9권, 제3호, pp.71-77, 2004

[6] www.policyalmanac.org/games

[7] Bertsekas, Dimitri, Neuro-Dynamic Programming, Athena Scientific, 1996

[8] 이상용, 인공지능, 상조사, pp.77-87, 2003

[9] http://www.ics.uci.edu/~eppstein/161/960215.html

[10] Empty space BSP트리를 이용한 3D 게임 렌더링 엔진 설계, 학국컴퓨터정보학회, 논문지, 제10권, 제3호, pp.345-352, 2005

[11] 정운철, 오상숙, Game Programming, 가남사, pp.175-178, 2001

[12] 한학용, 패턴인식 개론, 한빛미디어, pp.438-453, 2005

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliffford Stein, Introduce to Algorithms second editon, 문병로, 심규석, 이충세 역, pp.343-393, 한빛미디어

[14] S. E. Dreyfus and A. M. Law. The Art and Theory of Dynamic Programming, Academic Press, 1977

[15] Bertsekas, Dimitri, Dynamic Programming and Optimal Control: Second Edition, Athena Scientific, 2001

저 자 소 개

이 세 일

1993년 대전공업대학교 전자계산학과 졸업(공학사)

2001년 청운대학교 대학원 전산전자정보공학과(공학석사)

2004~현재 공주대학교 대학원 컴퓨터공학과(박사과정)

〈관심분야〉 Ubiquitous Computing, Context Awareness, Collaborative Filtering, 게임 알고리즘