Journal of Korea Game Society 2015 Aug; 15(4): 179-188
http://dx.doi.org/10.7583/JKGS.2015.15.4.179

게임 프로그래밍

장애물 정보를 이용한 A* 알고리즘의 탐색 공간의 감소

조성현
홍익대학교 게임학부
scho@hongik.ac.kr

Reducing Search Space of A* Algorithm Using Obstacle Information

Sung Hyun Cho
School of Games, Hongik University

요 약

A* 알고리즘은 잘 알려진 길찾기 알고리즘이다. 그러나 장애물 정보를 이용하지 않을 경우에는 장애물을 만날 때 까지 탐색을 진행하여 장애물과 충돌하거나 늪(swamp)에 들어갈 수 있다. 본 논문에서는 장애물을 회피하고 늪에 들어가지 않도록 장애물 정보를 사용하고, 장애물 정보를 이용한 휴리스틱 함수도 제안한다. 장애물 정보를 사전에 처리하는데 시간이 걸리지만 실시간에 처리하는 것이 아니기 때문에 대부분의 경우에 문제가 되지 않는다. 실험을 통하여 제안한 방법들이 검색 공간을 효과적으로 줄일 수 있음을 보여주었다. 더불어 장애물 정보를 이용한 휴리스틱 함수는 사전에 장애물 정보를 처리할 필요 없이 효과적으로 검색 공간을 줄일 수 있다는 것을 보여주었다.

ABSTRACT

The A* algorithm is a well-known pathfinding algorithm. However, if the information about obstacles is not exploited, the algorithm may collide with obstacles or lead into swamp areas unnecessarily. In this paper, we propose new heuristic functions using the information of obstacles to avoid them or swamp areas. It takes time to process the information of obstacles before starting pathfinding, but it may not cause any problems most of cases because it is not processed in real time. We showed that the proposed methods could reduce the search space effectively through experiments. Furthermore, we showed that heuristic functions using obstacle information could reduce the search space effectively without processing obstacle information at all.

Keywords : A* Algorithm, Path Finding, Heuristic Function, Search Space Reduction

Received: Jul, 10, 2015 Accepted: Aug, 10, 2015
Corresponding Author: Sung Hyuun Cho(Hongik University)
E-mail: scho@hongik.ac.kr

ISSN: 1598-4540 / eISSN: 2287-8211

Ⓒ The Korea Game Society. All rights reserved. This is an open-access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.otg/licenses/by-nc/3.0), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. 서 론

대부분 컴퓨터 게임에서 길찾기 알고리즘을 사용하고 있으며[1], A* 알고리즘은 게임과 같이 상호작용하는 가상세계에서 길찾기를 위하여 가장 널리 사용되고 알고리즘이다[2].

A* 알고리즘은 휴리스틱(heuristic) 함수가 동일할 때 A* 알고리즘보다 더 적은 수의 노드들을 탐색해서 최적의 경로를 찾아낼 수 있는 검색 알고리듬은 없다고 알려져 있다[2,3].

지금까지 A* 알고리즘에 관한 많은 연구가 진행되었다[1~5]. 기존의 대부분 연구들은 길찾기 과정에서 시작노드에서 현재 노드까지의 비용은 알지만, 현재 노드에서 목표 노드까지의 비용은 모른다는 가정을 한다. 그러나 게임에서는 장애물의 위치를 사전에 알 수 있기 때문에 장애물 정보를 이용하여 탐색할 필요가 없는 늪(swamp) 지역을 회피할 수 있다[6].

본 논문에서는 장애물 정보를 이용하여 장애물을 회피하고 늪에 들어가지 않는 방법을 제시하고, 장애물 정보를 이용한 휴리스틱 함수도 제안한다. 장애물 정보를 실시간에 처리하는 것이 아니기 때문에 대부분의 경우에 문제가 되지 않는다. 실험을 통하여 제안한 방법들이 검색 공간을 효과적으로 줄일 수 있음을 보이고자 한다. 더불어 장애물 정보를 이용한 휴리스틱 함수는 사전에 장애물 정보를 처리하지 않고도 효과적으로 검색 공간을 줄여 준다는 것을 보여준다.

그리드 맵에서 실험을 통하여 본 논문에서 제안한 방법의 유용성을 보이고자 한다. 본 논문의 구성은 다음과 같다. 2절은 관련 연구, 3절은 본 논문에서 제안하는 장애물 및 습지 회피 방법을 제시하고 장애물 정보를 사용하는 새로운 휴리스틱 함수를 제시하며, 4절은 실험 환경을 설명하고 실험 결과를 분석하며, 5절은 결론과 미래의 연구 방향을 논한다.

2. 관련 연구

길찾기는 출발노드에서 목표노드까지 장애물을 피하고, 적군을 피하고, 이동하는 비용 (연료, 시간, 거리, 비용 등)을 최소화하는 좋은 경로를 찾는 문제이다[1,7]. A* 알고리즘은 다익스트라 알고리즘에서 사용하는 정보 (출발노드에 가까운 노드를 선호함)와 최적노드선탐색(Best-First-Search) 방법이 사용하는 정보(목표노드에 가까운 노드를 선호함)를 결합하여 사용한다[1,2,7].

A* 알고리즘에서 g(n)는 출발노드에서 임의의 노드 n까지 경로의 비용이고, h(n)는 노드 n에서 목표 노드까지 경로의 휴리스틱 추정 비용으로 총 비용은 f(n) = g(n) + h(n)이다. A* 알고리즘은 출발노드에서 목표노드로 이동하면서 g(n)와 h(n)를 이용한다. A* 알고리즘은 루프가 반복될 때마다 f(n)값이 최소인 노드를 찾아서 먼저 그 노드를 먼저 탐색한다. A* 알고리즘은 현재 노드에 인접한 노드들 중에서 f(n) 값이 최소인 노드를 찾아서 그 노드로 이동하는 과정을 반복함으로써 가장 적은 비용의 최종 경로를 찾는다. 그 노드는 자신의 부모 노드, 즉 가장 적은 비용으로 자신에 이르도록 한 노드를 가리키는 포인터를 가진다. 목표노드에 도달하면 그 포인터를 사용하여 목표까지 이르는 노드들을 역추적하여 출발노드까지 가게 되며, 출발노드에서 목표노드까지 가장 적은 비용의 경로를 찾는다[1,8].

A* 알고리즘의 속도는 검색 공간의 크기에 크게 영향을 받기 때문에 검색공간의 최소화를 위한 연구가 진행되었으며, 또한 알고리즘 자체의 최적화를 위하여 메모리 할당과 해제가 자주 발생하기 때문에 메모리 할당과 관리 문제, 그리고 정렬을 빨리 하기 위한 기법들이 연구되었다[9,10].

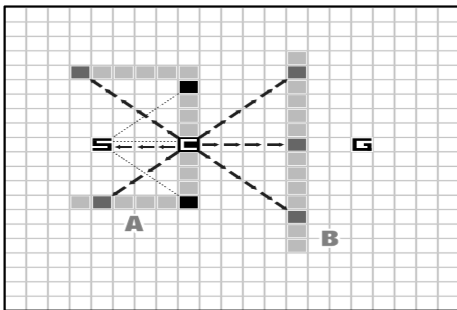
최근에 A* 알고리즘의 성능을 대단히 증가시킬 수 있는 점프 포인트 방법이 제시되었다[11]. 이 방법은 사전처리도 필요 없고, 메모리 부담도 없지만 대단히 빠르다는 장점을 갖고 있다. 그러나 모든 노드를 통과하는 비용이 같을 때에만 적용된다

는 단점이 있다. 게임에서는 재미를 더하기 위하여 노드의 통과비용이 다른 경우가 많기 때문에, 본 논문에서는 노드들의 통과비용이 다를 수 있는 일반적인 상황을 가정한다.

3. 장애물 정보를 이용한 A*

3.1 습지(swamp) 및 장애물 피하기

본 논문에서는 [Fig. 1]에서와 같이 장애물에 속한 노드들이 8방향으로 다른 장애물을 만날 때까지 장애물의 존재를 알려준다고 가정한다. 예를 들면 [Fig. 1]에서 장애물 A에 속한 노드 C는 6방향(위와 아래 방향은 장애물이 위치하므로 자신의 존재를 전달할 필요가 없음)으로 자신의 존재를 전달한다. 장애물에 속한 모든 노드들이 자신의 존재를 전달하고 나면, 일반 노드들은 장애물이 어떤 방향으로 얼마나 떨어져 있는지 알게 된다. 장애물의 정보는 8방향으로만 전달되기 때문에 일반 노드가 장애물에 대한 완전한 정보를 가지는 것은 아니다.



[Fig. 1] Propagation of Obstacle Information

[Fig. 1]에서 노드 S는 장애물 A의 정보는 가지게 되지만 장애물 B의 정보는 전달 받지 못한다. 즉 일반 노드는 자신의 8방향 시야에 있는 장애물만 인식하게 된다. 즉, 일반 노드들은 각 방향으로 자신에게서 가장 가까운 장애물의 존재와 거리를 기억한다. [Fig. 1]에서 장애물 A는 없고, 장애물 B만 있다고 가정하자. 노드 S는 장애물 B 때문에 동쪽 방향으로 목표 노드 G에 도달할 수 없다는 것을 알 수 있기 때문에 S의 동쪽에 이웃한 노드를 열린 목록에 삽입하지 않음으로써 길찾기 과정에서 장애물 정보를 이용하면 탐색 노드의 수를 줄일 수 있게 된다.

[Fig. 1]에서 노드 S는 북쪽, 동쪽, 남쪽 3방향에 장애물이 있다는 것과 그 거리를 알고 있기 때문에 길찾기 탐색과정에서 북쪽, 북동쪽, 동쪽, 남동쪽, 남쪽의 이웃 노드는 탐색에서 제외시킬 수 있다. 그래서 습지를 불필요하게 탐색하지 않게 된다.

장애물 및 습지를 피하기 위하여 길찾기를 시작하기 전에 전처리를 한다고 가정한다. 게임 진행 중에 장애물이 동적으로 발생하더라고 그리드 맵에 있는 모든 노드가 아니라 동적으로 발생한 장애물 정보는 8방향으로 시야가 트여 있는 일반 노드에만 전달되기 때문에 장애물 정보 수정이 일부 지역에만 국한되는 경우에는 동적으로도 장애물 정보 수정이 가능할 것으로 기대한다.

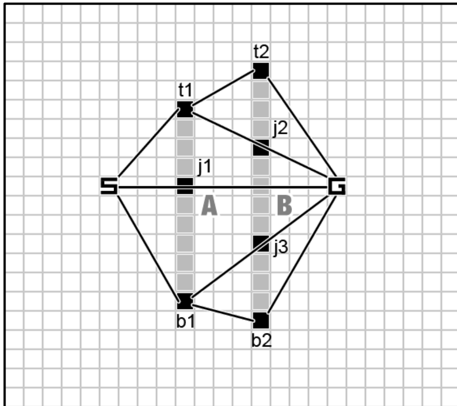
3.2 휴리스틱 함수

A* 알고리즘에서 f(n) = g(n) + h(n)를 사용하는데, g(n)는 출발노드에서 임의의 노드 n까지 경로의 비용이며, h(n)는 노드 n에서 목표 노드까지 경로의 추정 비용으로 총비용은 f(n) = g(n) + h(n)이 된다. A* 알고리즘은 출발노드에서 목표노드로 이동하면서 g(n)와 h(n)를 이용한다. 현재 노드로부터 목표노드까지 경로 비용은 추정하는 수밖에 없다. 그러나 최적의 경로를 보장하기 위해서는 비용을 과대평가해선 안 된다. 과대평가하지 않고 추정치를 구하는 일반적인 방법 중 하나는 주어진 노드로부터 목표까지의 맵 상에서 최단거리에 단위거리 당 통과하는데 필요한 최소 비용을 곱하는 것이다.

A* 알고리즘에서 휴리스틱 함수 h(n)는 다음과 같은 의미가 있다. h(n)의 값이 실제 비용보다 작다면 A* 알고리즘은 항상 최단경로를 찾지만 수행속도가 늦어지고, 반대로 h(n)의 값이 실제 비용보

다 크다면 A* 알고리즘은 항상 최단경로를 얻을 수 있는 것은 아니지만 수행속도는 빨라진다[1,2].

[Fig. 2]에서 최단 경로를 보장하기 위하여 h(S) = distance(S, G)를 사용할 수 있다. 즉, 대부분의 경우에 S와 목표노드 G의 직선거리를 사용한다.



[Fig. 2] Heuristic Functions h1() and h2()

그러나 노드 S가 장애물 A의 존재를 알고 있다면 h1(S) = distance(S, t1) + distance(t1, G)가 h(S) 보다 좀 더 정확한 거리를 제시해 준다. 노드 S는 장애물 B의 8방향 시야에서 벗어나 있기 때문에 장애물 B의 존재를 직접적으로 알 수 없다. 그러나 노드 S가 노드 t1과 b1을 찾을 수 있다면, 노드 t1은 장애물 B의 존재를 알고 있기 때문에 노드 t1은 h(t1) = distance (t1, G) 대신에 t1에서 노드 G까지 좀 더 정확한 거리인 h(t1) = distance(t1, t2) + distance(t2, G)를 사용할 수 있다. 그래서 본 논문에서 아래와 같은 2개의 휴리스틱 함수 h1(S)과 h2(S)를 사용한다.

h1(S) = min {distance(S, t1) + distance(t1,G),
distance(S, b1) + distance(b1, G)}, (eq. 1)

h2(S) = min {distance (S, t1) + h1(t1),
distance(S, b1) + h1(b1)}, (eq. 2)

일반적으로 h(S) <= h1(S) <= h2(S) 관계가 있으나, h1(S)와 h2(S)는 노드 S에서 node G까지 실제 비용보다는 같거나 작기 때문에 A* 알고리즘이 항상 최단경로 찾는 것을 보장해 준다.

노드 S가 노드 t1과 b1을 실시간에 찾아내는 과정은 아래와 같다.

1. 노드 S와 노드 G를 연결하는 직선의 기울기를 계산한다.
2. 노드 S에서 노드 G 방향으로 기울기에 따라서 한 노드씩 움직이면서 장애물과 만나는 노드를 찾는다([Fig. 2]에서 노드 j1).
3. 노드 j1의 위쪽 방향으로 장애물을 벗어나는 일반노드 t1을 찾는다. (장애물이 맵의 상단에 접해 있는 경우에는 t1은 존재하지 않고, 그쪽으로 돌아가는 길이 존재하지 않기 때문에 distance(S, t1) = ∞ 이 된다.).
4. 노드 j1의 아래쪽 방향으로 장애물을 벗어나는 일반노드 b1을 찾는다. (장애물이 맵의 하단에 접해있는 경우에는 b1은 존재하지 않고, 그쪽으로 돌아가는 길이 존재하지 않기 때문에 distance(S, b1) = ∞ 된다).

4. 실 험

4.1 실험 환경

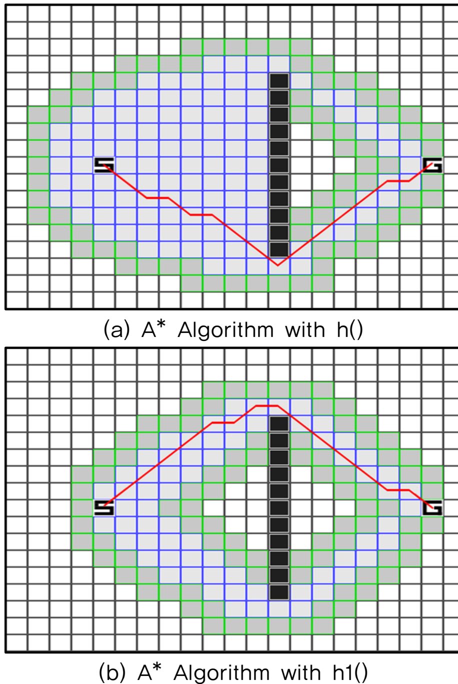
A* 알고리즘은 매 루프마다 열린 목록에서 현재 노드를 제거하고, 현재 노드의 8개 이웃 노드를 탐색을 한다. 이 때 목표 노드의 탐색에 도움이 되는 노드는 열린 목록에 삽입하고, 도움이 되지 않는 노드는 삽입하지 않는다. A* 알고리즘이 끝났을 때 열린 목록에 남아 있는 노드는 닫힌 목록에 있는 노드에 비하여 처리 비용이 훨씬 적기 때문에 닫힌 목록에 있는 노드의 수가 A* 알고리즘의 성능을 좌우한다. 그래서 본 논문에서는 닫힌 목록에 있는 노드의 수를 성능의 판단 기준으로 사용한다.

실험결과를 나타내는 그림에서 실험 환경은 다음과 같다. 시작노드는 왼편에 S로 표시하고, 목표

노드는 오른편에 G로 표시하며, 검정색 노드들은 장애물을 의미한다. 실험결과에서 선은 찾아낸 경로를 의미하고, 옅은 회색 노드는 닫힌 목록에 있는 노드, 회색 노드는 열린 목록에 남아있는 노드를 의미한다.

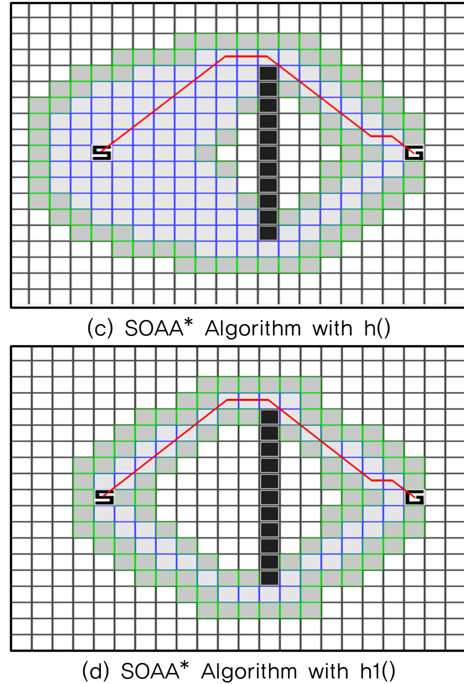
본 논문에서는 임의의 노드 n에서 3가지 휴리스틱 함수 h(n), h1(n), h2(n)를 사용한다. h(n)는 목표노드 G까지 유크리드 거리를 계산하는 함수, h1(n)은 (eq. 1)로 정의되며 장애물 1개만 고려한 함수이며, h2(n)는 (eq. 2)로 장애물 2개를 고려한 함수이다. 장애물이 없는 경우에는 h(n), h1(n), h2(n)이 같은 값을 갖게 된다. 본 논문에서 2개의 A* 알고리즘을 사용하는데, 기존과 같이 장애물이나 늪 회피를 고려한지 않는 A* 알고리즘과 장애물이나 늪을 회피하는 SOAA* (Swamp and Obstacle Avoiding A*) 알고리즘이다.

4.2 실험결과 및 분석



(a) A* Algorithm with h()

(b) A* Algorithm with h1()



(c) SOAA* Algorithm with h()

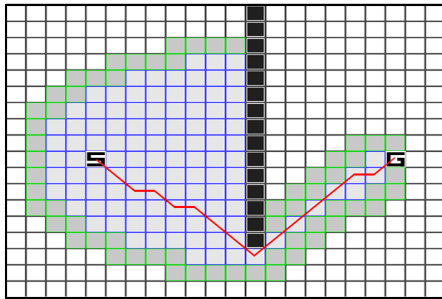
(d) SOAA* Algorithm with h1()

[Fig. 3] A Simple Bar Obstacle

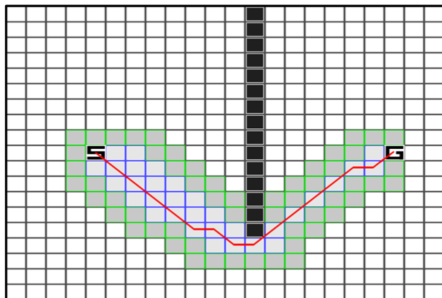
단순한 막대형 장애물 1개가 있는 경우로 휴리스틱 함수 h(), h1()과 A*와 SOAA* 알고리즘을 실험에 사용하였다. [Fig. 3] (b)와 (d)에서 장애물 앞에 삼각형 모양에 속한 노드들은 휴리스틱 함수 h1()로 인하여 검색에서 제외되었음을 알 수 있다. [Table 1]에서 A*(h1)와 SOAA*(h)를 비교해 보면 SOAA* 자체보다도 휴리스틱 함수 h1()도 장애물 정보를 활용하고 있으며 h1()의 영향이 더 크다는 것을 알 수 있다.

[Table 1] A Simple Bar Obstacle

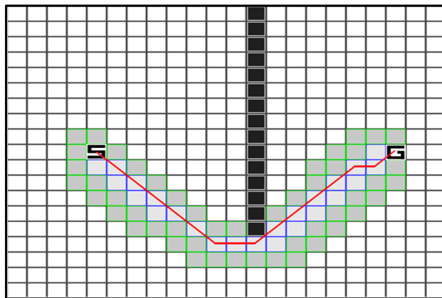
Algorithm	No. of Search	Performance
A*(h)	127	1
A*(h1)	70	1.81
SOAA*(h)	111	1.44
SOAA*(h1)	46	2.76



(a) A* Algorithm with h()



(b) A* Algorithm with h1()



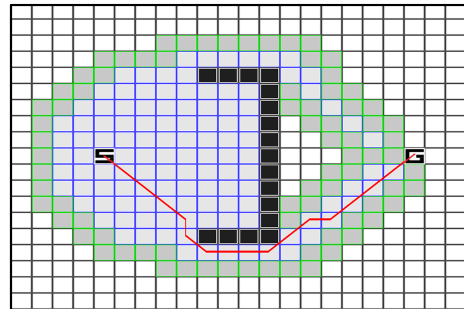
(c) SOAA* Algorithm with h1()

[Fig. 4] A Simple Bar Obstacle on the Boundary

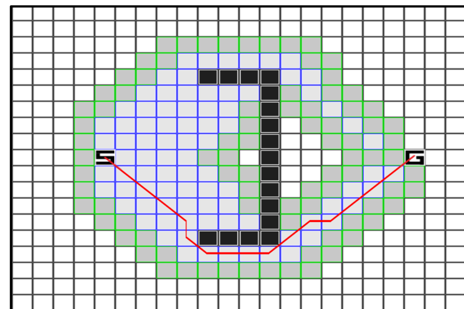
[Fig. 4]에서처럼 장애물이 경계에 접해 있는 경우에는 [Table 2]에서 보는 바와 같이 접하지 않은 [Table 1]의 경우보다 성능에 엄청난 차이를 보이고 있다. 이는 h1()과 SOAA* 모두 경계 쪽을 회피하도록 탐색을 진행하기 때문이다.

[Table 2] A Simple Bar Obstacle on the Boundary

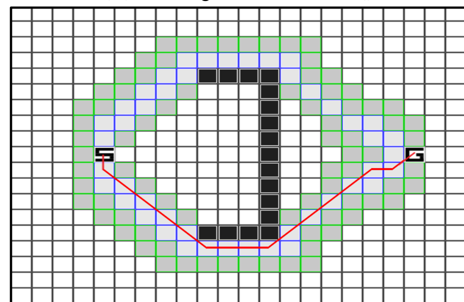
Algorithm	No. of Search	Performance
A*(h)	112	1
A*(h1)	34	3.29
SOAA*(h1)	26	4.31



(a) A* Algorithm with h()



(b) A* Algorithm with h1()



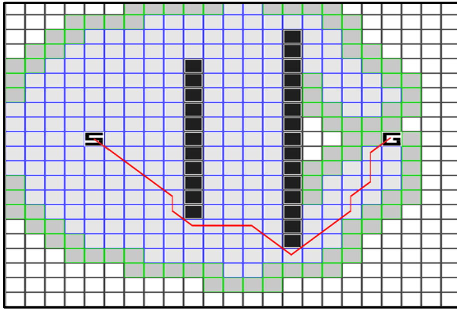
(c) SOAA* Algorithm with h1()

[Fig. 5] A Swamp Obstacle

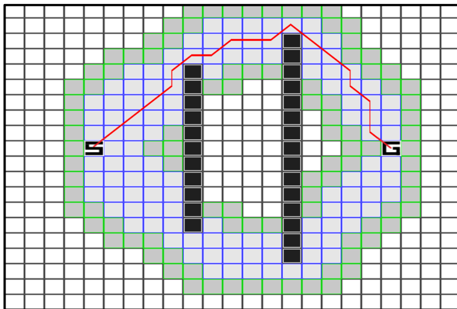
[Fig. 5]에서처럼 늪지 장애물이 있는 경우에는 [Table 3]에서 보는 바와 같이 SOAA*(h1)의 경우에 성능이 우수함을 알 수 있다. 이는 SOAA* 알고리즘과 h1() 모두 습지와 전면 장애물 회피 기능에 있기 때문이다.

[Table 3] A Swamp Obstacle

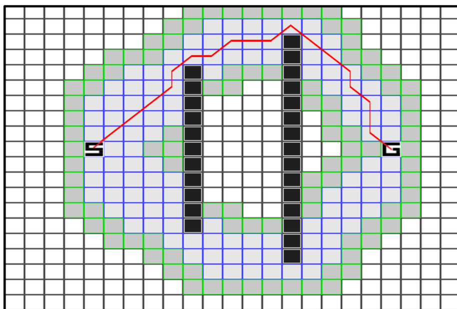
Algorithm	No. of Search	Performance
A*(h)	131	1
A*(h1)	82	1.60
SOAA*(h1)	47	2.79



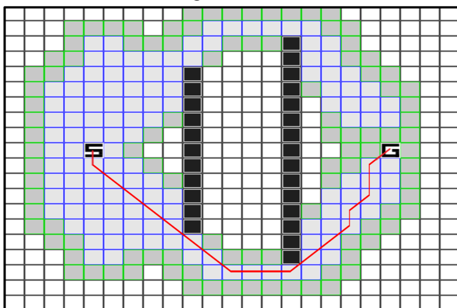
(a) A* Algorithm with h()



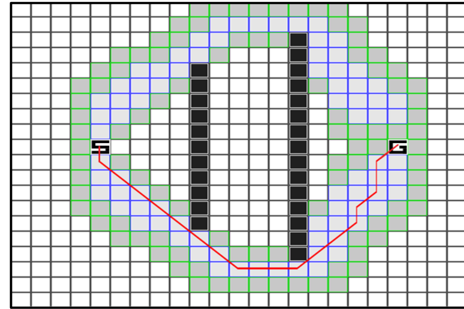
(b) A* Algorithm with h1()



(c) A* Algorithm with h2()



(d) SOAA* Algorithm with h1()



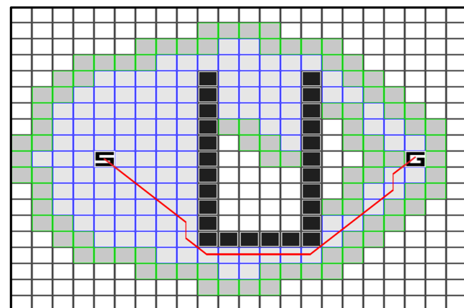
(e) SOAA* Algorithm with h2()

[Fig. 6] Two Bar Obstacles

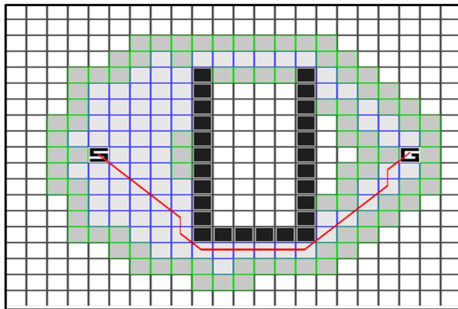
2개의 장애물까지 고려할 때 효과를 검증하기 위하여 [Fig. 6]과 같이 2개의 막대 장애물이 있는 상황을 설정하였다. [Table 4]는 A*(h1) 보다 A*(h2), SOAA*(h1) 보다 SOAA*(h2)의 성능이 우수함을 보여 주고, A*(h2)가 SOAA*(h1)의 성능보다 우수함을 보여주고 있다. 종합적으로 판단했을 때 h2()가 SOAA* 알고리즘보다 성능에 더 많이 영향을 미치고 있는 것으로 보인다.

[Table 4] Two Bar Obstacles

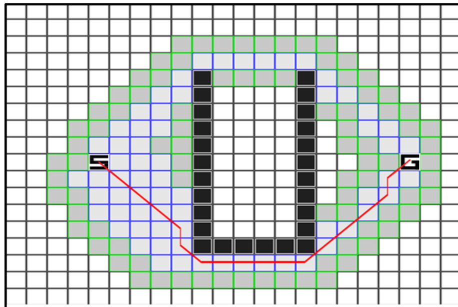
Algorithm	No. of Search	Performance
A*(h)	235	1
A*(h1)	160	1.47
A*(h2)	119	1.97
SOAA*(h1)	135	1.74
SOAA*(h2)	82	2.87



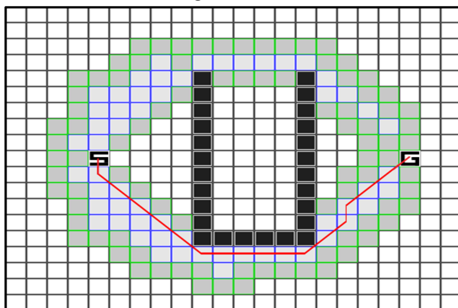
(a) A* Algorithm with h()



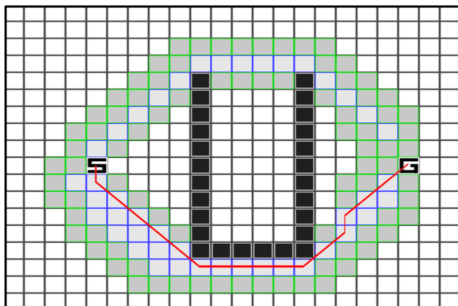
(b) A* Algorithm with h1()



(c) A* Algorithm with h2()



(d) SOAA* Algorithm with h1()



(e) SOAA* Algorithm with h2()

[Fig. 7] An Upward Swamp Obstacle

[Fig. 7]은 2개의 막대 장애물과 늪지가 동시에 존재하는 상황이다. [Fig. 7] (a)을 제외하고 모든 경우에 늪지 처리가 잘 되고 있다. 특히 [Fig. 7] (b)는 A*(h1)의 경우로 늪지 회피 기능이 없음에도 불구하고, h1() 단독으로도 늪지에 잘 대응하고 있음을 보여준다.

[Table 5] An Upward Swamp Obstacle

Algorithm	No. of Search	Performance
A*(h)	124	1
A*(h1)	83	1.49
A*(h2)	68	1.82
SOAA*(h1)	59	2.10
SOAA*(h2)	45	2.76

5. 결 론

A* 알고리즘에서 좋은 휴리스틱 함수를 찾을 수 없기 때문에 일반적으로 유크리드 거리 공식을 사용하고 있다. 그러나 본 논문에서는 완전한 휴리스틱 함수가 아닐지라도 h1()와 h2()와 같은 간단한 휴리스틱 함수가 길찾기 성능에 많은 영향을 미친다는 것을 실험결과를 통하여 확인하였다.

또한 장애물과 늪지 처리를 길찾기 시작하기 전에 처리한다면 h1()와 h2()와 같은 간단한 휴리스틱 함수와 SOAA* 알고리즘이 상호 상승작용으로 탐색하는 노드 수를 많이 줄일 수 있으며, 이것은 곧 길찾기 알고리즘의 성능 향상으로 이어진다는 것을 보여 주었다.

특별히 h1() 휴리스틱 함수는 간단하면서도 길찾기 시작 전에 추가적으로 처리할 것도 없다. 그러나 A* 알고리즘만을 사용하는 A*(h)와 A*(h1)과 성능을 비교하면, [Table 1]~[Table 5]에서 보는바와 같이 대부분의 경우에 47% 또는 그 이상의 성능 향상을 보여 주었다. 앞으로 실제적인 게임 상황에서 좀 더 심도 있게 실험 및 분석하는 후속 연구가 필요하다.

ACKNOWLEDGMENTS

This work was supported by 2014 Hongik University Research Fund.

REFERENCES

[1] Bryan Stout, "The Basics of A* for Path Planning", Game Programming Gems, pp.254-263, Charles River Media, 2000.
[2] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", IEEE Trans. Syst. Sci. Cybernet, 4(2): pp. 100 - 107, 1968.
[3] Nils J. Nilsson, Artificial intelligence: A New Synthesis, Morgan Kafmann, 1998.
[4] R. Korf, "Depth-first Iterative Deepening: An Optimal Admissible Tree Search", Artificial Intelligence, 27: pp. 97 - 109, 1985.
[5] Robert Kirk and DeLisle, "Beyond A*: IDA* and Fringe Search", Game Programming Gems 7, pp. 289-294, 2008.
[6] Nir Pochter, Aviv Zohar, Jeffrey S. Rosenschein, Ariel Felnner, "Search Space Reduction Using Swamp Hierarchies", Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, pp. 155-160, 2010.
[7] Ron Penton, Data Structures for Game Programmers, Premier Press Game Development, pp. 715-767, 2002.
[8] Sung Hyun Cho, "A Pathfinding Algorithm Using Path Information", Korea Game Society, Vol. 13, No. 1, pp. 31-40, 2013.
[9] Dan Higgins, "Fast A* implementation", AI game Programming Wisdom, pp. 223-238, 2003.
[10] Steve Ravin, "A* Speed Optimizations", Game programming Gems, pp. 363-381, Charles River Media, 2000.
[11] Daniel Harabor and Alban Grastein, "Online Graph Pruning for Pathfinding on Grid Maps", Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, pp. 1114-1119, 2011.

조 성 현(Cho, Sung Hyun)

약 력 : 1978 서울대학교 계산통계학과 이학사
1980 서울대학교 계산통계학과 이학석사
1995 UCLA 컴퓨터과학과 이학박사
1996-현재 홍익대학교 게임학부 교수

관심분야 : 게임 프로그래밍, 게임 그래픽스, 게임 물리, 분산 시스템