1. 연구제목 선정배경 및 이유

우연히 'Mindustry"라는 게임에 대해 알게 되어 재밌어 보여 시작했습니다. "Mindustry"는 타워 디펜스 게임인데 다른 일반적인 타워 디펜스 게임과는 다르게 적 들이 정해진 방향이 아닌 모든 방향에서 심지어 내가 '아 여기로 공격 들어오면 질 거 같은데' 하는 부분으로 적 들이 스스로 공격로를 정해서 공격하는 것이 정말 신선했습니다. 그렇게 이 게임을 하다가 문득 어떤 원리로 공격로를 설정하는 것인지 궁금해졌습니다. 그렇게 어떻게 공격로를 설정하는지 찾다가 길 찾기 알고리즘이라는 것을 알게 되었습니다. 그때 마침 미로를 알고리즘으로 풀 수 있는 방법은 없나 고민하면서 링크드 리스트를 이용해 미로를 해결하는 알고리즘을 고안하고 있었는데, 이 길 찾기 알고리즘 이란 것을 이용하면 내가 고안했던 것 보다 훨씬 정교하고, 빠르게 계산할 수 있을 것 같다는 생각이 들어서 이 연구를 하게 되었습니다.

2. 과제 후기

이번 과제를 하면서 처음으로 자료구조와 알고리즘에 대해 자세히 배웠습니다. 처음 다익스트라 알고리즘에 대해 조사했을 때는 정말 이것이 무슨말인지 이해하지 못해서 며칠 동안 이것만 붙잡고 고민했습니다. 그러다 마침내 이것이 무슨 말인지 깨닫고 어떻게 작동하는지 알아내서 직접 문제를 만들어 풀었을 때는 정말 좋았습니다. 그리고 그렇게 A* 알고리즘까지 해결하고 미로에 적용했을 때 또 새로운 생각이 났고, 그것을 적용하면 필요한 계산 량을 줄일 수 있다는 생각이 들어 정말 좋았습니다. 만약 다음에도 이런 활동을 할 기회가 있다면 재미있게 참여할 거 같습니다.

길 찾기 알고리즘과 미로에서 최적화 할 수 있는 방안

19011625 허진수

PathFinding Algorithms and A* Algorithms with C

19011625 HeoJinSu

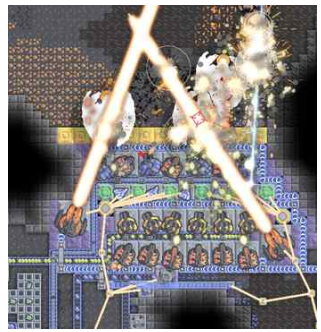
요 약

다양한 게임에서 플레이어 캐릭터 또는 적 NPC(Non Player Character)가 한 지점에서 다른 지점으로 가는 것은 매우 중요하다. 이러한 역할을 수행하는 것이 '길 찾기 알고리즘(PathFinding Algorithms)'이다. 본 연구에서는 이러한 길 찾기 알고리즘에 대하여 알고, 미로에서 어떤 길 찾기 알고리즘이 유리한지 분석하고, 해당 길 찾기 알고리즘을 좀 더 미로에 최적화 하여 풀 수 있는 방안을 연구한다.

1. 서 론

게임 'Mindustry'에서 적 NPC(Non Player Character)는 한 지점 또는 여러 지점에서 스폰 후 플레이어가 수비해야할 최종 지점인 코어를 향해 이동한다. 플레이어는 이동하는 적 들을 코어에 도달하기 전 까지 자원을 캐고, 그 자원으로 디펜스 건물을 지어 적이 코어에 도달하는 것을 막아야 한다. 여기서 적 NPC는 특이하게도 수비 측에서 취약한 부분을 찾아 공격로를 설정하고 공격한다. 기존의 정해진 한 방향으로 공격하는 일반적인 디펜스 게임과는 달리 'Mindustry'에서 적 NPC는 수비 측에서 취약한 부분을 찾고, 해당 부분을 공격하기 위한 공격로를 설정하는 등 기존의 디펜스와는 달리 다양한 공격로를 설정한다.

이처럼 'Mindustry' 뿐만 아니라 다양한 게임에서 적 NPC뿐만 아니라 주인공 캐릭터 까지 현재 지점에서 지정된 지점까지 이동하는 알고리즘은 필수적이다. 또한 단순히 이동하는 것만이 아니라 지정된 지점 까지 이동하는 중간에 방해물이 있으면 그것을 우회할 수도 있어야 하고, 가장 효율적으로, 최단 시간에 가는 방법 또한 있어야 한다. 이러한 역할을 수행해 주는 것이 바로 길 찾기 알고리즘(PathFinding Algorithms)이다. 본 연구에서는 이러한 길 찾기 알고리즘에 대하여 알고, 미로에서 어떤 길 찾기 알고리즘이 유리한지 분석하고, 해당 길 찾기 알고리즘을 좀 더 미로에 최적화 하여 풀 수 있는 방안을 연구해본다.



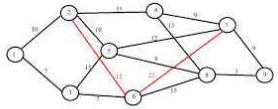
[그림 1]게임 'Mindustry'
(https://anuke.itch.io/mindustry)

2. 다익스트라 알고리즘

1. 가중치 그래프

최단 경로를 구할 때는 가중치 그래프를 사용한다. 가중치 그래프(Weight Graph)는 각 간선 e에 연관되는 숫자 레이블(예를 들어 정수)을 $w(e)$로 갖는 그래프이다. 여기서 $w(e)$는 간선 e의 가중치(weight)라고 한다.

G 를 가중치 그래프라고 할 때, 경로 P의 길이[length, 또는 가중치(weight)]는 P의 간선들의 가중치의 합이다.[1]



[그림 2] 가중치 그래프

2. 다익스트라 알고리즘

다익스트라 알고리즘(영어: Dijkstra Algorithm) 또는 데이스크라 알고리즘은 도로 교통망 같은 곳에서 나타 날 수 있는 그래프에서 꼭짓점 간의 최단 경로를 찾는 알고리즘 이다. 이 알고리즘은 컴퓨터 과학자 에츠허르 다익스트라(네덜란드어: Edsger Wybe Dijkstra)가 1965년에 고안했다. 단일-소스(single-source) 최단 경로 문제에 그리디 메소드를 적용한 결과가 다익스트라 알고리즘 이다.

다익스트라 알고리즘의 설명을 간단히 하기 위해 다음을 가정한다. 입력 그래프 G는 비방향성이며 단순하다. 즉, 어떠한 자기-반복(self-loop)과 병렬 간선(parallel deges)이 없다. 그러므로 G의 간선을 비순서화된 정점들의 쌍 (u,z)로 표시할 수 있다. 최단 경로를 찾기 위한 다익스트라 알고리즘에서, 그리디 메소드의 응용에서 최적화 하려는 비용 함수는 최단 경로 거리(shortest path distance)를 계산하는 함수이다.

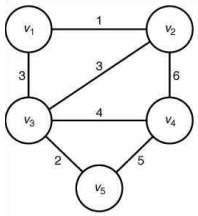
G 내에서 v에서 u로의 거리 근사(appriximate)에 사용하기 위해, V 내의 각 정점 u를 위한 레이블 $D[u]$를 정의한다. 이러한 레이블 $D[u]$는 항상 지금까지 발견한 v에서 u로의 최상 경로의 길이를 저장한다. 초기에 $D[v]=0$이고 각 $u \neq v$에 대해 $D[u]=+\infty$이다.

정점들의 선택 집합 C 를 초기에 공집합 $\varnothing$으로 정의한다. 알고리즘의 각 반복에서, C에 들어있지 않으면서 최소의 $D[u]$레이블을 갖는 정점 u를 선택하고, C에 u를 넣는다. 첫 번째 반복에서는 v를 C에 넣는다. C에 새로운 정점 u가 들어가면 u를 경유하여 z로 가는 더 좋은 새로운 방법이 있을 지도 모른다는 사실을 반영하기 위해 u에 인접하고 C에 들어있지 않은 각 정점 z의 레이블 $D[z]$를 갱신한다. 이 갱신 연산은 과거의 평가를 취하여 실제 값에 더 가깝도록 하기 위해 향상되어질 수 있는지 점검하기 때문에 경감(relaxation)절차로 알려져 있다. 다익스트라 알고리즘에 경우, 경감은 간선 (u,z)를 이용하여 $D[z]$을 위한 더 나은 값이 있는지를 알아보기 위해 $D[u]$의 새로운 값을 계산하도록 간선 (u,z)에서 실행된다. 간선 경감 연산은 다음과 같이 이루어진다.[1]

(식 1) $\text{if } D[u]+w((u,z)) < D[z] \text{ then}$
$D[z] \leftarrow D[u]+w((u,z))$

3. 다익스트라 알고리즘 예시

다음과 같은 가중치 그래프가 있을 때 V_1에서 V_5까지 갈 때 최단 거리를 구하는 방법은 다음과 같다.



[그림 3] 가중치 그래프

가중치 그래프를 인접 행렬로 나타내면 다음과 같다.

[표 1] 가중치 그래프의 인접 행렬

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	3	INF	INF
V_2	1	0	3	6	INF
V_3	3	3	0	4	2
V_4	INF	6	4	0	5
V_5	INF	INF	2	5	0

G 내에서 v에서 u로의 거리 근사에 사용하기 위해, V 내의 각 정점 u를 위한 레이블 $D[u]$를 정의하고, 지금까지 발견한 v에서 u로의 최상 경로의 길이를 저장하여 나타내면 다음과 같이 나타낼 수 있다. 표는 경감 연산을 하여 최단 경로를 저장하여 표시한 값이다.

[표 2] $V_1 \rightarrow V_5$ 최단 거리

	V_1	V_2	V_3	V_4	V_5
$D[V_1]$	0	1	3	INF	INF
$D[V_2]$	0	1	3	7	INF
$D[V_3]$	0	1	3	7	5
$D[V_4]$	0	1	3	7	5
$D[V_5]$	0	1	3	7	5

V_1에서 V_3로 가는 경로는 V_2를 경유하는 방법과 V_3로 바로 가는 방법 2가지가 있다. 이 두 방법의 경로를 비교했을 때 $V_1 \rightarrow V_3$의 경로가 더 짧기 때문에 $D[V_3]$에서 3으로 저장한다. 위와 같은 계산을 하였을 때 V_1에서 출발 할 경우 각 노드에 대한 최단 거리는 표의 최종 값인 {0, 1, 3, 7, 5} 이다. 또한 V_1부터 V_5까지의 최단 경로는 $V_1 \rightarrow V_3 \rightarrow V_5$이다.

4. 다익스트라 알고리즘의 시간복잡도

다익스트라 알고리즘을 우선순위 큐Q의 효율적인 구현인 힙(heap)으로 사용할 경우 실행 시간은 다음과 같다. 우선순위 큐 Q를 힙(heap)를 사용할 경우 $O(\log n)$시간 내에 가장 작은 D레이블을 가진 정점 u를 추출하는 것을 가능하게 한다. 레이블 $D[z]$를 갱신할 때 마다 우선순위 큐 안의 z의 키를 갱신하는 것을 필요로 한다. 예를 들어, 만약 Q가 힙으로 구현되었다면, 이 키 갱신은 기존 키를 먼저 제고하고 새로운 키를 가진 z를 삽입함으로써 이루어질 수 있다. 만약 우선순위 Q가 위치 지시자 패턴(locator pattern)을 지원한다면, $O(\log n)$시간의 키 갱신을 쉽게 구현할 수 있다. Q의 이러한 구현을 가정한다면 다익스트라 알고리즘은 $O((n+m)\log n)$시간 안에 실행된다.[1][3]

3. A* 알고리즘

1. A* 알고리즘

A* 알고리즘(영어: A* search algorithm, 발음 : A-star)은 최단 경로 탐색 알고리즘 중 하나로 시작 노드와 목적지 노드를 분명하게 지정하여 이 두 노드 간의 최단 경로를 파악한다. 다익스트라 알고리즘에 경우 시작 노드만을 지정해 다른 모든 노드에 대한 최단 경로를 파악하지만, A* 알고리즘의 경우 시작 노드와 목적지 노드를 분명하게 지정하여 이 두 노드 간의 최단 경로를 파악하기 때문에 다익스트라 알고리즘에 비해 실행시간이 짧다. A* 알고리즘은 휴리스틱 추정 값을 통해 알고리즘을 개선한다. 따라서 휴리스틱 추정 값을 어떤 방식으로 제공하느냐에 따라 얼마나 빨리 최단 경로를 파악할 수 있느냐가 결정된다.[4]

이 알고리즘은 1968년 피터 하트, 닐스 닐슨, 버트램 라펠이 처음 기술하였다. 그 3명의 논문에서, 이 알고리즘은 A알고리즘(algorithm A)이라고 불렸다. 적절한 휴리스틱을 가지고 이 알고리즘을 사용하면 최적(optimal)이 된다. 그러므로 A* 알고리즘이라고 불린다.

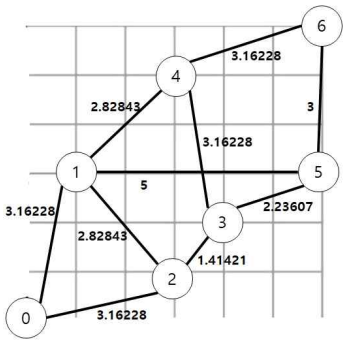
A* 알고리즘은 출발 꼭짓점으로부터 목표 꼭짓점까지 최적 경로를 탐색하기 위한 것이다. d를 위해서는 각각의 꼭짓점에 대한 평가 함수를 정의해야 한다. 이를 위한 평가 함수 $f(n)$은 다음과 같다.[5]

(식 2) $f(n)=g(n)+h(n)$

- $g(n)$: 출발 노트로부터 노드 n까지의 경로 가중치
- $h(n)$: 노드 n으로부터 목표 노드까지의 추정 경로 가중치(휴리스틱 추정치)

2. A* 알고리즘 예시

다음과 같은 가중치 그래프가 있을 때 노드 0에서 노드 6까지 가는 최단 경로를 구하는 방법은 다음과 같다. 표의 값은 정확도를 위해 계산 값을 소수점 6째 자리에서 반올림한 값이다.



[그림 4] 가중치 그래프

각 노드로부터 도착 노드까지의 유클리드 거리(Euclidean distance)를 휴리스틱 추정 값으로 잡았다. 먼저 열린 목록 저장소(Open List)와 닫힌 목록 저장소(Closed List)를 정의한다. 각각의 저장소에는 Node ID, $f(n)$, $g(n)$, $h(n)$, Parent Node ID가 들어간다. 먼저 닫힌 목록 저장소(이하 C)에는 시작 노드를 넣는다. 열린 목록 저장소(이하 O)에는 시작 노드와 연결된 노드를 넣는다.

[표 3] 저장소

O	ID	1	2
	$f(n)$	8.99323	8.99323
	$g(n)$	3.16228	3.16228
	$h(n)$	5.83095	5.83095
	P	0	0
C	ID	0	1
	$f(n)$	0	8.99323
	$g(n)$	0	3.16228
	$h(n)$	0	5.83095
	P	-	0

O 저장소에 들어간 노드 중 $f(n)$이 가장 작은 노드를 선택하여 C 저장소에 저장한다. 다만, 위의 경우처럼 가장 작은 $f(n)$의 값을 가지는 노드가 여러 개 일 경우 어느 노드를 선택하여 C에 저장해도 상관없다. 이 경우에는 노드 1을 선택하여 저장하였다.

$f(n)$이 가장 작은 노드를 C 저장소에 저장하였다면, C 저장소에 추가된 노드와 연결된 노드 중 C 저장소에 존재 하지 않는 노드를 O 저장소에 넣고 계산한다.

[표 4] 저장소

O	ID	2	4	5
	$f(n)$	8.99323	9.15298	11.16228
	$g(n)$	3.16228	5.99070	8.16228
	$h(n)$	5.83095	3.16228	3
	P	0	1	1
C	ID	0	1	2
	$f(n)$	0	8.99323	8.99323
	$g(n)$	0	3.16228	3.16228
	$h(n)$	0	5.83095	5.83095
	P	-	0	0

O 저장소에서 노드 2는 노드 0이 Parent Node일 때와 비교 했을 때 노드 1이 Parent Node일 때의 $f(n)$이 더 크므로 $f(n)$이 더 작은 Parent Node가 1인 노드로 유지한다. O 에 저장된 노드 2, 5, 4를 비교했을 때 노드 2의 $f(n)$이 가장 작으므로 C 저장소에 넣는다. 그리고 O 저장소에 노드 2와 연결된 노드 중 C 저장소에 있는 노드를 제외한 노드들을 넣고 계산한다.

[표 5] 저장소

O	ID	4	5		3
	$f(n)$	9.15298	11.16228		9.04863
	$g(n)$	5.99070	8.16228		4.57649
	$h(n)$	3.16228	3		4.47214
	P	1	1		2
C	ID	0	1	2	3
	$f(n)$	0	8.99323	8.99323	9.04863
	$g(n)$	0	3.16228	3.16228	4.57649
	$h(n)$	0	5.83095	5.83095	4.47214
	P	-	0	0	2

O 저장소에서 노드 3의 $f(n)$이 가장 작으므로 C 저장소에 넣은 후, O 저장소에 노드 3과 연결된 노드 중 C 저장소에 저장되지 않는 노드를 추가하고 계산한다.

[표 6] 저장소

O	ID	4			5	
	$f(n)$	9.15298			9.81256	
	$g(n)$	5.99070			6.81256	
	$h(n)$	3.16228			3	
	P	1			3	
C	ID	0	1	2	3	4
	$f(n)$	0	8.99323	8.99323	9.04863	9.15298
	$g(n)$	0	3.16228	3.16228	4.57649	5.99070
	$h(n)$	0	5.83095	5.83095	4.47214	3.16228
	P	-	0	0	2	1

노드 4의 경우 Parent Node가 1일 때가 Parent Node가 3일 때 보다 $f(n)$의 값이 더 작으므로 유지한다. O 저장소에 저장된 노드 중 $f(n)$ 크기가 가장 작은 노드 4를 C 저장소에 저장하고, 노드 4와 연결된 노드 중 C 저장소에 존재 하지 않는 노드를 O 저장소에 넣는다.

[표 7] 저장소

O	ID	5			6		
	$f(n)$	9.81256			9.15298		
	$g(n)$	6.81256			9.15298		
	$h(n)$	3			0		
	P	3			4		
C	ID	0	1	2	3	4	6
	$f(n)$	0	8.99323	8.99323	9.04863	9.15298	9.15298
	$g(n)$	0	3.16228	3.16228	4.57649	5.99070	9.15298
	$h(n)$	0	5.83095	5.83095	4.47214	3.16228	0
	P	-	0	0	2	1	4

노드 1과 노드 3은 이미 C 저장소에 저장되있으므로 O 저장소에 새로 추가하지 않는다. O 저장소에 저장된 노드 중 노드 6의 $f(n)$ 값이 가장 작으므로 C 저장소에 추가한다. 도착 노드인 노드 6이 C 저장소에 저장되었으므로 알고리즘은 종료된다. 이제 도착 노드의 Parent Node를 보며 시작 노드까지 거슬러 올라가면 6, 4, 1, 0임을 알 수 있다. 따라서 0 -> 1 -> 4 -> 6 순의 경로가 최단 경로이며, 최단 거리는 약 9.15298이다.

3. A* 알고리즘과 다익스트라 알고리즘의 차이

A* 알고리즘과 다익스트라 알고리즘의 가장 큰 차이는 도착 노드가 정해져 있냐 정해져 있지 않냐 이다. A* 알고리즘의 경우 시작 노드와 도착 노드 모두가 정해여 있기 때문에 도착 노드에 도달하는 최단 경로만을 구한다. 반면 다익스트라 알고리즘의 경우 도착 노드가 정해져 있지 않기 때문에 시작 노드로부터 모든 노드에 대해 최단 경로를 구하는 방법이다. 따라서 A* 알고리즘의 실행시간이 다익스트라 알고리즘의 실행시간보다 짧다. 그래서 일반적으로 A* 알고리즘을 게임 내에서 길 찾기 알고리즘으로 사용하는 경우가 많다.

A* 알고리즘의 경우 휴리스틱 추정 값에 많이 의존한다. 따라서 휴리스틱 추정 값에 따라서 최단 경로가 아닐 수도 있다. 예를 들어 위의 예시에서 휴리스틱 추정 값을 유클리드 거리(Euclidean distance)를 썼는데, 휴리스틱 추정 값으로 유클리드 거리가 아닌 맨해튼 거리(Manhattan distance)를 쓸 경우 다음과 같은 결과가 나온다.

[표 8] 최종 C 저장소 값 - 노드 1

ID	0	1	4	6
$f(n)$	0	11.16228	9.99070	9.15298
$g(n)$	0	3.16228	5.99070	9.15298
$h(n)$	0	8	4	0
P	-	0	1	4

위 표의 값은 첫 번째 O 저장소에서 노드 1과 노드 2에 값이 같았을 때 노드 1을 C 저장소에 넣은 결과 이다. 그러나 노드 1이 아닌 노드 2를 C 저장소에 넣을 경우 다음과 같은 결과가 나온다.

[표 9] 최종 C 저장소 값 - 노드 2

ID	0	2	3	5	6
$f(n)$	0	11.16228	10.57649	9.81256	9.81256
$g(n)$	0	3.16228	4.57649	6.81256	9.81256
$h(n)$	0	8	6	3	0
P	-	0	2	3	5

노드 1을 처음 C 저장소에 넣었을 때와 다르게 최종 경로가 달라진다. 이는 A* 알고리즘으로 최단 경로를 구하고 싶다면 위 가중치 그래프에서는 맨해튼 거리가 휴리스틱 추정 값으로 알맞지 않음을 알 수 있게 한다.

또한 A* 알고리즘에서 휴리스틱 추정 값을 적용하지 않을 경우, 즉 $f(n) = g(n)$일 경우 A* 알고리즘은 다익스트라 알고리즘과 동일하게 모든 노드에 대해 계산하게 된다.

[표 10] 최종 C 저장소($f(n) = g(n)$ 인 경우)

ID	$f(n)$	$g(n)$	$h(n)$	P
0	0	0	0	-
1	3.16228	3.16228	0	0
2	3.16228	3.16228	0	0
3	4.57649	4.57649	0	0
4	5.99070	5.99070	0	1
5	6.81256	6.81256	0	3
6	9.15298	9.15298	0	4

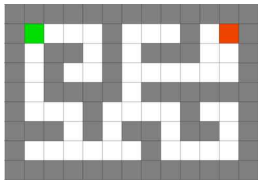
[표 11] 다익스트라 알고리즘

	0	1	2	3	4	5	6
D_0	0	3.16 228	3.16 228	INF	INF	INF	INF
D_1	0	3.16 228	3.16 228	INF	5.99 070	8.16 228	INF
D_2	0	3.16 228	3.16 228	4.57 649	5.99 070	8.16 228	INF
D_3	0	3.16 228	3.16 228	4.57 649	5.99 070	6.81 256	INF
D_4	0	3.16 228	3.16 228	4.57 649	5.99 070	6.81 256	9.15 298
D_5	0	3.16 228	3.16 228	4.57 649	5.99 070	6.81 256	9.15 298
D_6	0	3.16 228	3.16 228	4.57 649	5.99 070	6.81 256	9.15 298

4. 연구

1. A* 알고리즘의 미로 적용

다음과 같은 미로가 있다고 가정한다. 미로에서의 대각선 이동은 불가능 하고, 거리 측정법은 맨해튼 거리(Manhattan distance)를 사용하겠다.



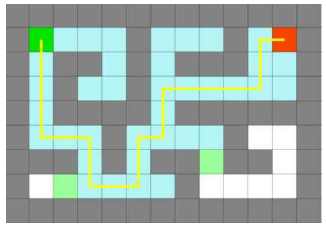
[그림 5] 미로[6]

일반적으로 A* 알고리즘에 경우 블록 하나하나를 노드로 본다. 그리고 왼쪽에서 오른쪽으로 1 ~ 11, 위에서 아래로 A ~ G라고 하였을 때 휴리스틱으로 맨해튼 거리를 사용할 경우 $f(n)$을 정리하면 다음과 같이 나타낼 수 있다. 'X'는 미로의 벽에 해당하고 '-'는 $f(n)$의 값이 상대적으로 작아 A* 알고리즘에 의해 접근하지 않아 계산하지 않는 부분이다.

[표 11] $f(n)$의 값

	1	2	3	4	5	6	7	8	9	10	11
A	S	10	10	10	X	22	22	22	X	22	D
B	12	X	X	12	X	22	X	X	X	22	22
C	14	X	16	14	X	22	22	22	22	22	22
D	16	X	X	X	X	22	X	X	X	X	X
E	18	18	18	X	22	22	22	22	X	-	-
F	X	X	20	X	22	X	X	24	X	X	-
G	-	24	22	22	22	22	X	-	-	-	-

위의 표에 의해 A*알고리즘으로 이 미로를 풀 경우 다음과 같이 최단 경로가 산출된다.



[그림 6] A* 알고리즘에 의한 미로의 풀이

파란색으로 칠해진 노드는 A* 알고리즘에 의해 C 저장소에 저장된 노드이고 초록색으로 칠해진 노드는 A* 알고리즘에 의해 C 저장소에 저장되지 않은 노드들이다. 미로에 3방향이 막혔을 때에 추가로 연결된 노드가 없으므로 이미 O 저장소에 저장된 다른 노드를 C 저장소에 저장하기 때문에 모든 블록을 노드로 보았을 때 A* 알고리즘으로 미로를 해결할 수 있다.

2. A* 알고리즘의 미로에 대한 최적화

위에서 보였든 일반적인 A* 알고리즘은 모든 블록을 노드로 보고 계산한다. 그러나 미로의 경우 계산을 할 필요가 없는 노드가 몇 가지 있다. 예를 들면 가던 방향으로만 갈 수 있는 경우 추가적인 선택지가 존재 하지 않고 오직 한 방향으로만 가게 된다. 이러한 경우에는 계산을 하지 않아도 다음 노드가 어디인지 예측 가능하다. 따라서 미로에서 실제로 이런 경우에는 노드로 만들 필요가 없다. 그러므로 다음의 경우에 해당하는 블록만 노드로 만든다.

1) 들어온 방향에서 다른 방향으로 바뀌는 블록

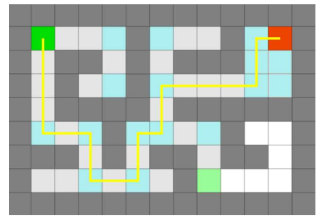
2) 갈 수 있는 방향이 들어온 방향을 제외하고 두 방향 이상인 블록

위의 정의의 따라 노드를 만들어 $f(n)$의 값을 구하면 다음과 같이 정리된다.

[표 12] $f(n)$의 값

	1	2	3	4	5	6	7	8	9	10	11
A	S			10	X	22			X	22	D
B		X	X		X		X	X	X	22	22
C		X		14	X	22				22	22
D		X	X	X	X		X	X	X	X	X
E	18		18	X	22	22		22	X		-
F	X	X		X		X	X		X	X	
G			22		22	22	X	26			-

위의 표에 의해 미로를 풀 경우 다음과 같이 최단 경로가 산출된다.



[그림 7] 미로에 최적화 된 A* 알고리즘에 의한 미로의 풀이

5. 결 론

길 찾기 알고리즘에는 많은 종류가 있지만 대표적으로 최단 경로 탐색 알고리즘에 해당되는 두 개의 알고리즘인 다익스트라 알고리즘과 A* 알고리즘이 있다. 다익스트라 알고리즘은 시작 노드에서 모든 노드에 대해 최단 경로를 탐색하기 때문에 계산 량이 매우 많고 오래 걸린다. 반면 A* 알고리즘은 시작 노드와 도착 노드가 정해져 있으므로 모든 노드에 대해 최단 경로를 탐색하지 않고 시작 노드와 도착 노드에 대해서만 최단 경로를 탐색한다. 따라서 다익스트라 알고리즘에 비해 계산 량이 적고 실행시간이 빠르다. 반면 A* 알고리즘은 휴리스틱 추정 값에 크게 의존하기 때문에 휴리스틱 추정 값이 잘못 될 경우 최단 경로가 나오지 않을 수도 있다.

미로의 경우 대각선 이동이 금지되기 때문에 거리 추산이 상대적으로 쉽다. 또한 대각선 이동이 금지되고 모든 거리가 직선으로 바로 다음 붙은 블록들로 이동하기 때문에 맨해튼 거리(Manhattan

distance)로 휴리스틱 추정 값을 구할 수 있다. 또한 미로의 특성 상 미로가 커질 경우 계산해야할 노드의 수가 매우 많아진다. 따라서 미로의 경우 다익스트라 알고리즘보다 A* 알고리즘이 더 적합하다.

한편 A* 알고리즘의 경우에도 미로가 클 경우 필요한 노드의 수가 매우 많다. 따라서 미로에 대해서 A* 알고리즘의 노드를 최적화할 필요가 있다. 그래서 본 연구에서는 노드가 생성될 때 일정 조건을 만족해야만 생성하도록 하여 A* 알고리즘의 노드 수를 최대한 줄이는 방안을 생각하였다. 이 방안으로 A* 알고리즘에서 노드 수가 기존의 수 보다 줄어 필요한 계산 량이 상당히 줄 것으로 예상되어 미로에서 더 적은 계산 량과 실행 시간으로 A* 알고리즘으로 미로를 해결할 수 있을 것으로 예상된다.

6. 참고문헌

[1] Michael T.Goodrich, Roberto Tamassia, David Mount. Data Structures and Algorithms in C++. Trans. 범한서적주식회사, 2004

[2] 이승찬. 컴퓨팅 사고를 위한 기초 알고리즘 모두의 알고리즘 with 파이썬, 길벗, 2017

[3] "데이크스트라 알고리즘", 위키백과, 2019년 10월 25일 수정, 2019년 12월 4일 접속, https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%81%AC%EC%8A%A4%ED%8A%B8%EB%9D%BC_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

[4] "최단 경로 탐색-A*알고리즘", GIS DEVELOPER, 2018년 6월 21일 수정, 2019년 12월 5일 접속, http://www.gisdeveloper.co.kr/?p=3897

[5] "A* 알고리즘", 위키백과, 2019년 12월 5일 접속, https://ko.wikipedia.org/wiki/A*_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

[6] "PathFinding", Github, 2019년 12월 6일 접속 https://qiao.github.io/PathFinding.js/visual/