**A\* Algorithm**

**Application Note.**

**Revision 1.00**

**15 Feb, 2025**

**Industrial Automation Division**

**Application Center**

**LSIS Co., Ltd.**

**Legal information**

**Use of application examples**

Application examples illustrate the solution of automation tasks through an interaction of several components in the form of text, graphics and/or software modules. The application examples are a free service by LSIS. They are non-binding and make no claim to completeness or functionality regarding configuration and equipment. The application examples merely offer help with typical tasks; they do not constitute customer-specific solutions. You yourself are responsible for the proper and safe operation of the products in accordance with applicable regulations and must also check the function of the respective application example and customize it for your system.

Siemens grants you the non-exclusive, non-sub licensable and non-transferable right to have the application examples used by technically trained personnel. Any change to the application examples is your responsibility. Sharing the application examples with third parties or copying the application examples or excerpts thereof is permitted only in combination with your own products. The application examples are not required to undergo the customary tests and quality inspections of a chargeable product; they may have functional and performance defects as well as errors. It is your responsibility to use them in such a manner that any malfunctions that may occur do not result in property damage or injury to persons.

**Disclaimer of liability**

LSIS shall not assume any liability, for any legal reason whatsoever, including, without limitation, liability for the usability, availability, completeness and freedom from defects of the application examples as well as for related information, configuration and performance data and any damage caused thereby. This shall not apply in cases of mandatory liability, for example under the German Product Liability Act, or in cases of intent, gross negligence, or culpable loss of life, bodily injury or damage to health, non-compliance with a guarantee, fraudulent non-disclosure of a defect, or culpable breach of material contractual obligations. Claims for damages arising from a breach of material contractual obligations shall however be limited to the foreseeable damage typical of the type of agreement, unless liability arises from intent or gross negligence or is based on loss of life, bodily injury or damage to health. The foregoing provisions do not imply any change in the burden of proof to your detriment. You shall indemnify LSIS against existing or future claims of third parties in this connection except where LSIS is mandatorily liable.

By using the application examples you acknowledge that LSIS cannot be held liable for any damage beyond the liability provisions described.

**Other information**

LSIS reserves the right to make changes to the application examples at any time without notice. In case of discrepancies between the suggestions in the application examples and other LSIS publications such as catalogs, the content of the other documentation shall have precedence.

The LSIS terms of use (https://www.lsis.com) shall also apply.

**Security information**

LSIS provides products and solutions with industrial security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. LSIS’ products and solutions constitute one element of such a concept.

Customers are responsible for preventing unauthorized access to their plants, systems, machines and networks. Such systems, machines and components should only be connected to an enterprise network or the Internet if and to the extent such a connection is necessary and only when appropriate security measures (e.g. firewalls and/or network segmentation) are in place.

For additional information on industrial security measures that may be implemented, please visit https://www.lsis.com.

LSIS’ products and solutions undergo continuous development to make them more secure. LSIS strongly recommends that product updates are applied as soon as they are available and that the latest product versions are used. Use of product versions that are no longer supported, and failure to apply the latest updates may increase customer’s exposure to cyber threats.

Contents

[1. Path-finding in 2d maze 4](#_Toc190536002)

[1.1 Path-finding in 2d maze for two mobile robots using A-Star algorithm 4](#_Toc190536003)

[1.2 Assumptions 4](#_Toc190536004)

[1.3 Build and Run 4](#_Toc190536005)

[1.4 Results 4](#_Toc190536006)

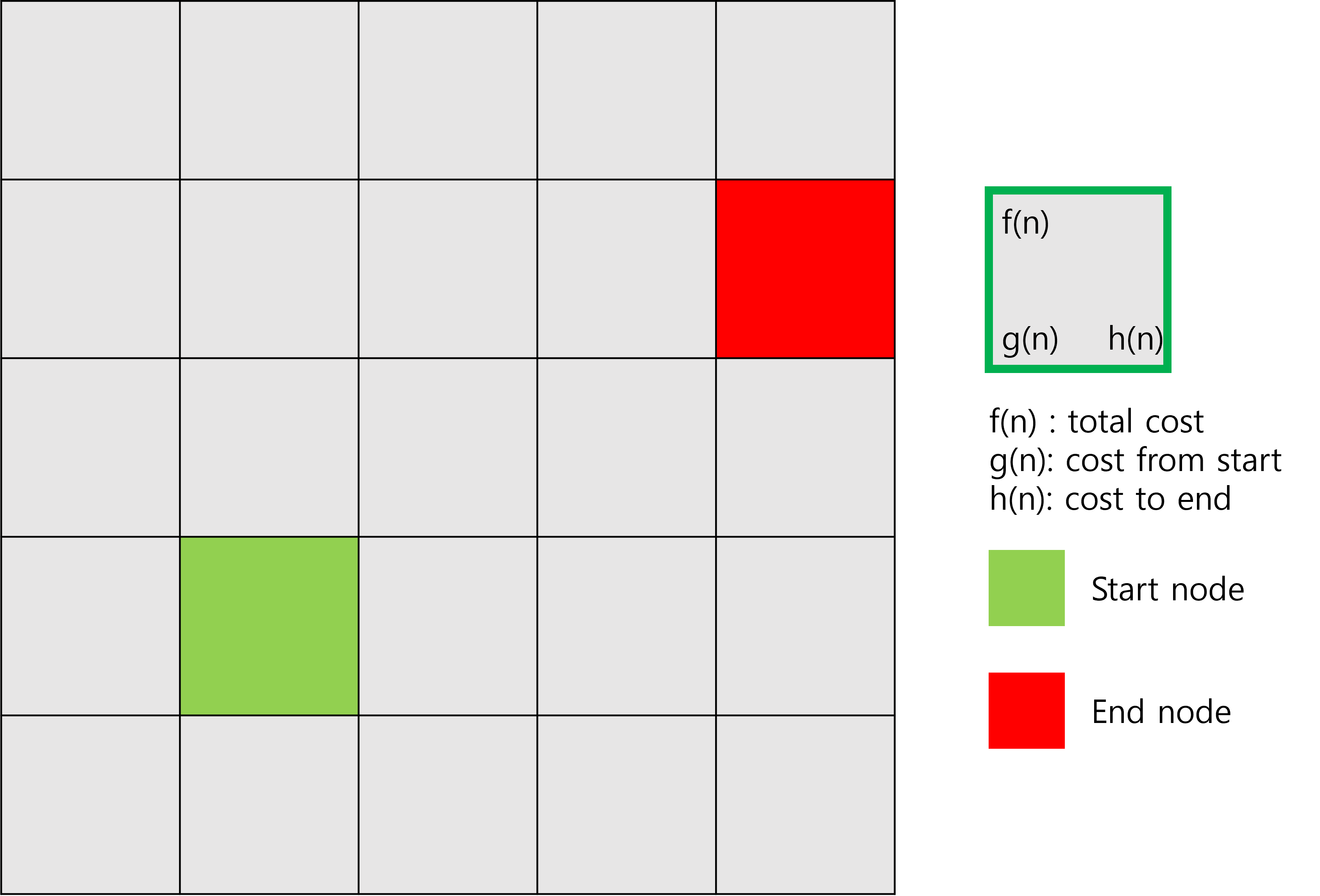
[1.5 UML Class Diagram 4](#_Toc190536007)

[2. C++ Project Source 9](#_Toc190536008)

[3. a\* 길찾기 알고리즘 : C++ 11](#_Toc190536009)

# A\* algorithm 구현

 단도직입적으로 A\* 알고리즘이 어떤 식으로 구현되는지 알아보자. 일단 아래와 같이 격자 지도(grid map)를 생성하여 지도(map)를 단순화한다. 이때 오각형, 육각형, 직사각형 등의 격자를 사용하지 않고 정사각형 격자를 쓴 이유는 가장 단순한 모양이기 때문에 성능 저하를 막고 빠르게 처리할 수 있기 때문이다.  그리고 생성된 격자 지도에서 출발 지점(초록 사각형)과 목표 지점(빨강 사각형)을 임의로 지정하였다. 여기서 각 사각형의 중심점을 노드(node)라고 부르기로 한다. 각 사각형은 빈 공간이여서 '이동 가능'하거나 또는 장애물이 있어서 '이동 불가능'한 2가지 상태 중 하나를 갖는다. 이때 A\* 알고리즘의 목적은 주어진 아래 지도에서 출발 지점(node)과 목표 지점(node) 사이의 최단 거리를 찾는 것이다.



grid map에 생성된 출발 노드(Start node)와 목표 노드(End node)

## 비용 계산

 위 그림에서 맨 오른쪽 바깥 격자 보면 f(n), g(n), h(n)이라는 표시가 있는데 이는 비용(cost)를 뜻한다. 경로를 탐색하기 위해서는 각 사각형(node)마다 이동에 소요되는 비용을 계산이 필요한데, 이중 가장 작은 비용을 가진 사각형들을 서로 이으면 시작 지점과 목표 지점을 잇는 경로가 생성이 된다. 각 비용이 적게 드는 node끼리 연결했으므로 찾아낸 경로가 최단 경로라고 짐작할 수 있다. 각 비용의 의미는 다음과 같다.

**Heuristic cost function: f(n) = g(n) + h(n)**

 g(n): 출발 노드에서 현재 노드 n까지 도달하기 위한 최단 비용

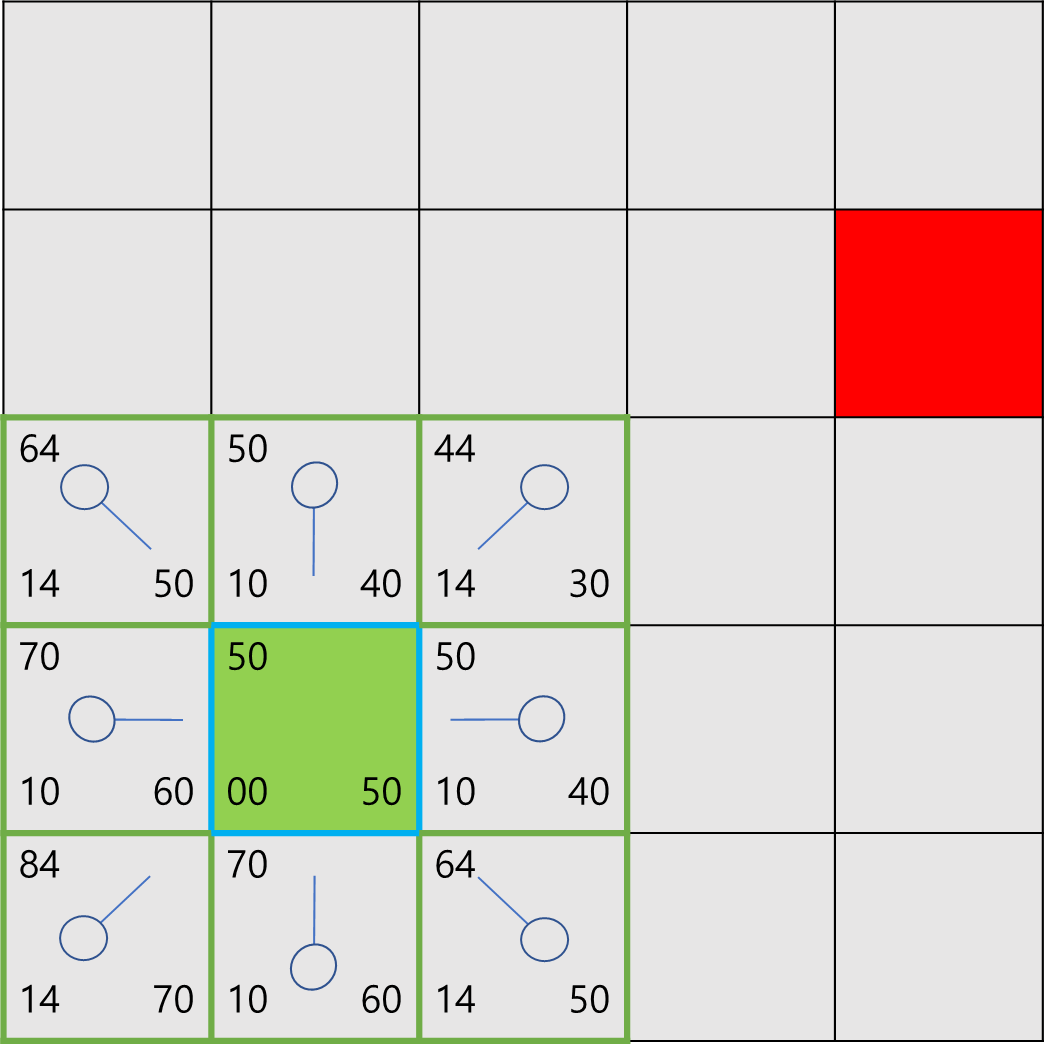
 h(n): 현재 노드 n에서 목표 노드까지의 예상 이동 비용으로, 휴리스틱(Heuristic) 거리 측정값이라고 한다.

 f(n): 현재 노드 n까지의 최단 비용 g(n)과 목표 노드까지의 예상 이동 비용 h(n)을 더한 총 비용이다. Heuristic cost function이라고 한다.

 자세하게 설명하자면, g(n)은 출발 노드에서 현재 노드까지의 이동하는 데 드는 최소 이동 비용을 일컫는다. 여기서는 g(n)을 유클리디안 거리(Euclidean distance)를 사용하여 나타내기로 한다.  가로/세로 한 칸 이동당 10을, 대각선 한 칸 이동은 14로  설정하기로 한다. 대각선 한 칸 이동이 14인 이유는 피타고라스 정리에 따라 대각선의 이동 거리는 가로/세로 이동 거리의 1.414배이기 때문이다. 이때 정확하게 계산하면 14.14가 되는데 소수보다는 정수가 컴퓨터가 처리하는 데 더 빠르므로 소수점을 제외하여 14라고 설정한다. 만약 출발 노드에서 현재 노드까지 오는데 가로 한 칸 이동, 세로 한 칸 이동, 대각선 한 칸 이동이라면 g(n)=10+10+14=34가 되는 것이다. 여기서는 g(n)은 사각형의 왼쪽 아래에 표기하기로 한다.

 그 다음으로 h(n)은 다양한 측정 방식을 사용할 수 있는데 여기서는 맨하탄 거리(Manhattan distance)를 사용하기로 한다. 해당 방식의 이름의 유래는 미국의 도시 중 하나인 맨하탄(Manhattan)으로부터 유래됐다. 맨하탄에는 블록마다 건물이 빽빽하게 들어가 있기 때문에 두 지점 사이를 오갈 때 대각선으로는 갈 수 없고 오직 가로/세로로만 이동할 수가 있는데, 이와 같은 방식으로 거리를 계산하기 때문에 맨하탄 거리라고 명명되었다. 이름의 유래처럼, 이 방식은 현재 노드에서 목표 노드까지 도달하기 위한 이동에서 대각선 이동을 제외하고 오직 가로/수직 이동만의 비용을 계산한다. 가로/세로 이동 한 칸당 10이 더해진다. 이를테면, 만약 가로 2칸, 세로 3칸을 이동한 경우 총 5칸을 이동했으므로 h(n)=50이 되는 것이다. h(n)은 오른쪽 아래에 표기하기로 한다.

 마지막으로 f(n)은 g(n)와 h(n)를 더한 총 비용 값이다. 이 값을 통해 경로 탐색을 할 때 어느 노드가 가장 작은 비용을 가지는지 판단하게 된다. f(n)은 사각형 왼쪽 위에 표기하기로 한다.



 위 그림에서 **초록색 출발 사각형**의 오른쪽 사각형의 비용을 살펴보자. 출발 사각형으로부터 오른쪽 사각형까지 가로로 한 칸 이동해야 하기 때문에 해당 사각형 왼쪽 아래에 **g(n)=10**이라고 표기되어 있다. 그리고 오른쪽 사각형에서 빨간색 목표 사각형까지 가로 2칸, 세로 2칸 이동해야 하므로 사각형 오른쪽 아래에 h(n)=20+20=40 이라고 표기되어 있다. 마지막으로 사각형의 왼쪽 위는 50이라고 써 있는데, 이는 총 비용인 f(n)=g(n)+h(n)=10+40=50을 표기한 것이다. 나머지 노드들도 비슷하게 계산할 수 있다.

이제 본격적으로 경로 탐색 방법에 대해 알아보도록 하자.

## 경로 탐색1

 최단 경로 탐색을 위해 아래와 같은 순서로 탐색을 시작한다. 출발 노드부터 목표 노드까지 인접한 사각형을 확인해 가면서 경로를 만들어 나가게 된다.

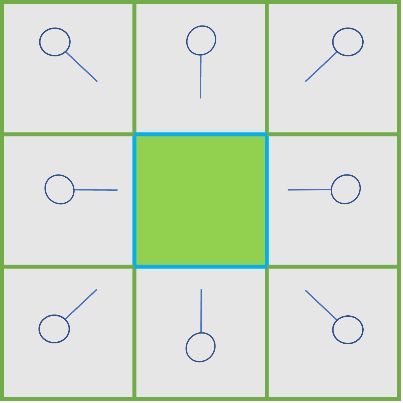
**(1) 출발 사각형(노드)를 '열린 목록(open list)'에 넣어준다.** 이 목록은 일종의 장바구니와 같다. 지금은 출발 노드만 있지만 탐색하면서 점차 늘어날 것이다. 이때 열린 목록에 있는 노드는 초록 윤곽선으로 둘러싸서 표현할 것이다.

**(2)** 출발 사각형에 인접한 장애물은 무시하고 **지나갈 수 있는 사각형을 '열린 목록'에 넣어 준다. 이 사각형들은 출발 사각형를 부모로 지정한다. (이떄 '부모 노드(parent node)'는 경로를 다 찾고 거슬로 올라갈 때 사용된다.)** 출발 노드를 비롯한 열린 목록에 있는 사각형들의 f(n), g(n), h(n) 비용들을 계산하여 각각의 노드에 기입해 준다.

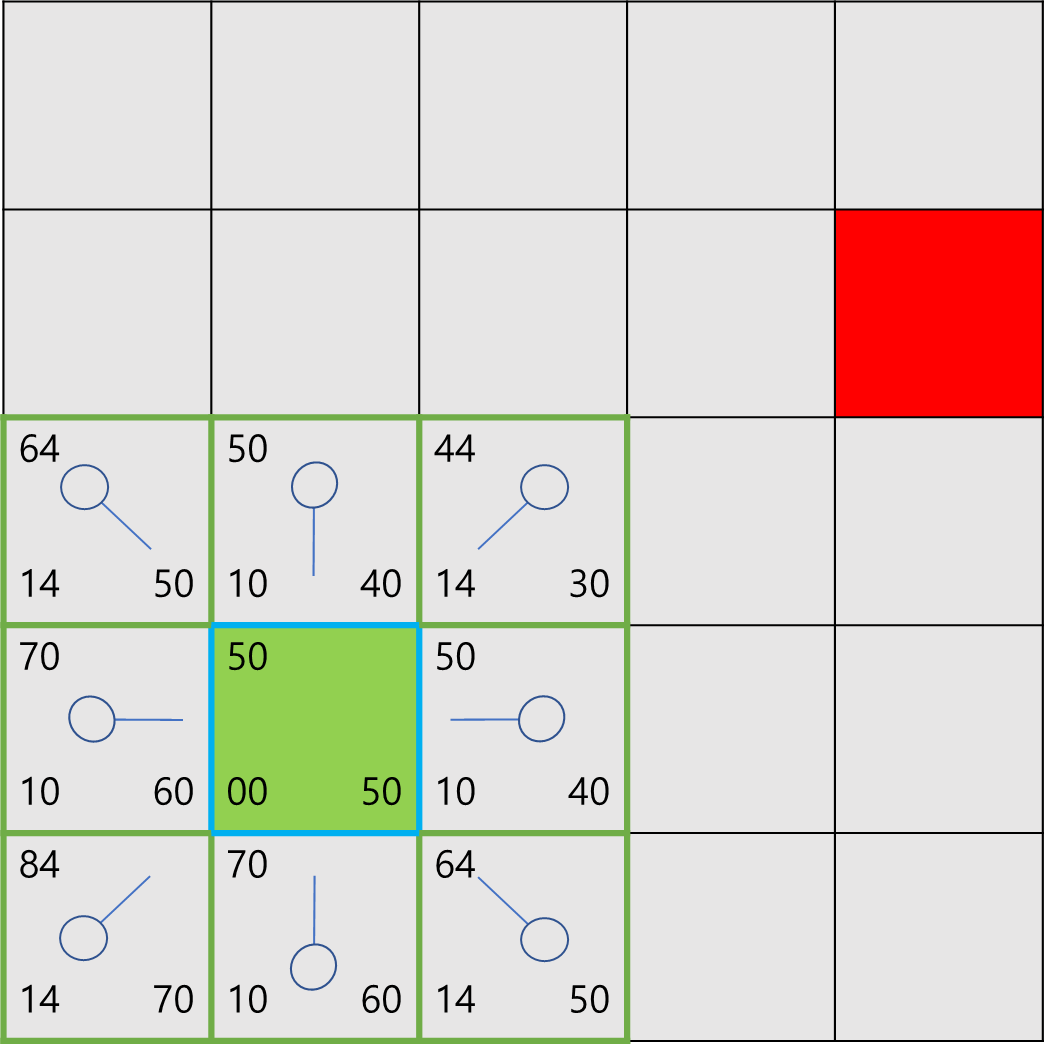
**(3) '열린 목록'에서 출발 사각형을 없애고 다시 볼 필요 없는 '닫힌 목록'(closed list)에 추가해 준다.** 이때 닫힌 목록에 있는 노드는 하늘색 윤곽선으로 둘러싸서 표현할 것이다.

위 과정을 순서대로 수행하면 아래와 같은 그림을 얻게 된다.

**부모 노드와 화살표(포인터)**



 위 그림에서 가운데 있는 초록 사각형은 출발 사각형 (노드) 이다. 하늘색 윤곽선 테두리는 출발 노드가 '닫힌 목록'에 추가가 되어 더 이상 볼 필요가 없다는 것을 의미한다. 그리고 출발 노드에 인접한 8개의 사각형(노드)은 초록 윤곽선으로 둘러싸여 있다. 이는 '열린 목록'에 들어가 있다는 것을 뜻한다. 이때 회색 화살표 모양의 포인터(pointer)가 보이는데, 화살표는 해당 노드의 부모 노드(parent node)를 가리킨다. 인접한 8개의 사각형(노드)이 부모 노드인 초록 사각형을 가리키고 있는 것이다. 인접 노드들의 f(n), g(n), h(n) 비용을 계산해주면 아래와 같다.



## 경로 탐색2

 계속 탐색하기 위해, **'열린 목록'에 있는 사각형들 중에서 가장 작은 f(n) 비용을 가지고 있는 사각형(노드)을 선택한다.** 그리고 아래의 순서에 따라 진행하면 된다.

(4) **선택한 사각형(노드)**을 '열린 목록'에서 빼고 '닫힌 목록'에 넣어 준다. 이는 선택된 사각형이 '부모 노드'가 될 준비가 되었음을 의미한다.

(5) **인접한 사각형을 확인한다.** 인접 사각형 중에 '닫힌 목록'에 있거나 장애물인 것들을 제외하고, 나머지 '열린 목록'에  사각형이 없다면 '열린 목록'에 추가한다. 그리고 현재 사각형을 '열린 목록'에 새롭게 추가된 사각형들의 '부모'로 만든다. (여기서 현재 사각형이란 (4)에서의 선택된 사각형을 말한다.) 이때 열린 목록에 있는 사각형들의 f(n), g(n), h(n) 비용들을 계산하여 각각의 노드에 기입해 준다.

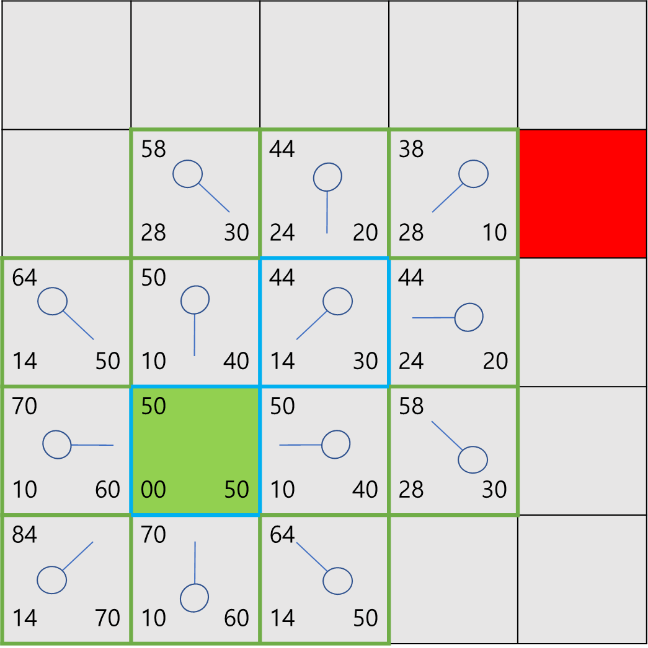
**※인접한 사각형의 g(n) 비용은 그 부모로부터 g(n) 비용을 얻어와서 부모로부터 가로/세로로 이동하면 10,  대각선으로 움직이면 14를 추가한다.**

(6) **인접한 사각형이 이미 기존의 '열린 목록'에 있다면 현재 사각형을 기준으로 해당 인접한 사각형까지 이동할 때 g(n) 비용이 낮아지는 지 확인한다.** 비용이 더 낮아지지 않으면 아무것도 하지 않는다. 만약 현재 사각형을 통해 해당 인접 사각형까지 이동하는 데 g(n) 비용이 더 낮게 나온다면, 해당 인접 사각형의 부모 노드를 현재 사각형으로 바꾼다. 그리고 해당 인접 사각형의 f(n), g(n)을 다시 계산한다. (부모 노드가 되면 해당 인접 사각형의 화살표가 부모 노드 쪽으로 향하게 된다.)

(7) 위 과정이 끝나면 '열린 목록'에 있는 사각형들 중에서 가장 작은 f(n) 비용을 가지고 있는 사각형(노드)을 선택한다. 그리고 다시 경로 탐색 중 빨간색 목표 노드가 '열린 목록'에 추가되거나 '열린 목록'이 비어 있게 될 때까지 (4)~(6) 과정을 계속 반복한다.

(8) 빨간색 목표 노드가 '열린 목록'에 추가되면, 목표 사각형으로부터 각각의 사각형의 부모 사각형을 향하여 시작 사각형에 도착할 때까지 거슬러 올라갑니다. 부모 사각형이 어디인지는 화살표의 방향을 보면 알 수 있다. 화살표가 가리키는 곳이 부모 노드이다.

 위 (4) ~ (8) 작업들을 직접 해보자.



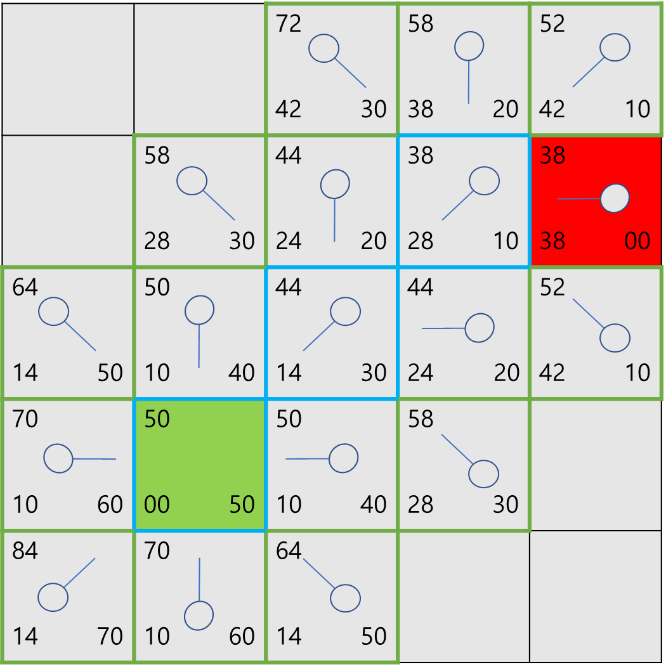
 처음에 9개 사각형 중에 초록색 출발 사각형을 '닫힌 목록'에 넣은 뒤, 인접한 8개의 사각형을 '열린 목록'에 추가해줬다. 이들중 f(n) 비용이 가장 작은 것은 출발 사각형의 오른쪽 대각선쪽에 있는 f(n)=44 사각형이다.

(4) 따라서 이 사각형을 닫힌 목록에 추가하여 준다. 닫힌 목록에 들어 갔으므로 하늘색 윤곽선으로 표시해준다. 이는 선택된 사각형이 '부모 노드'가 될 준비가 되었음을 의미한다.

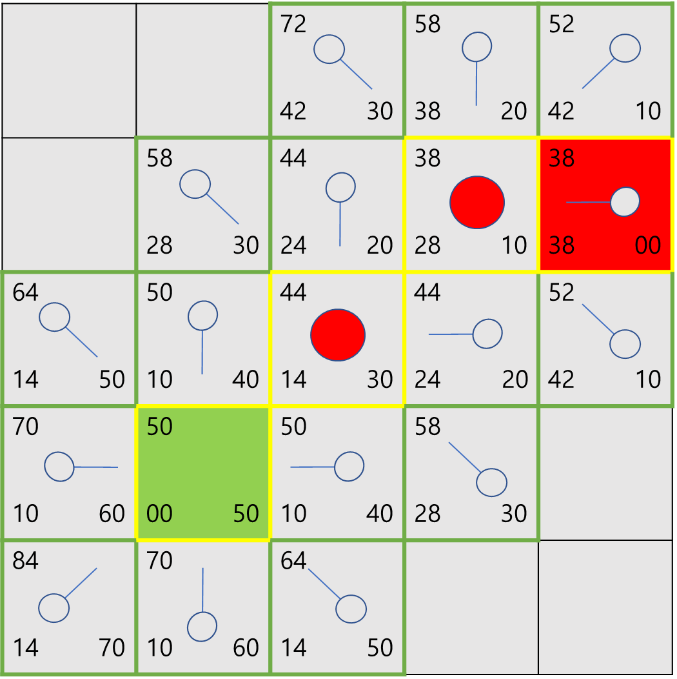
(5) 그런 다음 인접한 사각형을 확인한다. 선택된 현재 사각형의 왼쪽 위 대각선, 위쪽, 오른쪽 위 대각선, 오른쪽, 오른쪽 아래 대각선 모두 열린 목록에 없었으므로 '열린 목록'에 새로 추가해준다. 현재 사각형을 '열린 목록'에 새롭게 추가된 사각형들의 '부모 노드'로 만든다. 새롭게 추가된 사각형들의 화살표는 부모 노드를 향하게 된다. 더불어 열린 목록에 있는 사각형들의 f(n), g(n), h(n) 비용들을 계산하여 각각의 노드에 기입해 준다.

 ※인접한 사각형의 g(n) 비용은 그 부모로부터 g(n) 비용을 얻어와서 부모로부터 가로/세로로 이동하면 10,  대각선으로 움직이면 14를 추가한다.

(6) 현재 사각형의 왼쪽, 아래에 있는 사각형은 이미 '열린 목록'에 있는 사각형이다. 왼쪽, 아래쪽 사각형 모두 g(n)=10인데, 만약 현재 사각형을 거쳐서 이동하면 가로 또는 세로로 한 칸씩 이동해야 하므로 g(n) 비용은 10씩 추가되어 g(n)=24으로 늘어나게 된다. 따라서 현재 사각형을 통해 가도 g(n) 비용이 개선되지 않으므로 아무것도 하지 않는다.



(7) 선택된 현재 사각형의 인접 사각형을 모두 확인했으므로 '열린 목록'에 있는 사각형들 중에서 가장 작은 f(n) 비용을 가지고 있는 사각형을 선택한다. f(n)=38인 사각형이 가장 작은 f(n) 비용을 가지므로 이 사각형을 선택하여 닫힌 목록에 추가해 준다. 닫힌 목록에 들어 갔으므로 하늘색 윤곽선으로 표시해준다. 이때 인접한 노드 중 빨간색 목표 노드가 열린 목록에 추가되므로 더 이상의 탐색을 중단하고, 인접한 노드들의 f(n), g(n), h(n) 비용들을 계산해준다.



(8) 빨간색 목표 노드가 '열린 목록'에 추가되었으므로,  목표 사각형으로부터 각각의 사각형의 부모 사각형을 향하여 시작 사각형에 도착할 때까지 거슬러 올라간다. 부모 사각형이 어디인지는 화살표의 방향을 보면 알 수 있다. 화살표가 가리키는 곳이 부모 노드이다. 찾아낸 최단 경로가 위 그림에서 노란색 테두리로 표시되어 있다.

## A\* 알고리즘 요약(Summary of the A\* Method)

**1. 출발 사각형에 인접한 지나갈 수 있는 사각형을 '열린 목록'에 넣어 준다.**

**2. 다음의 과정들을 반복한다.**

(1) 열린 목록에서 가장 작은 f(n) 비용을 가진 사각형을 찾아 선택한다.

(2) 이것을 열린목록에서 꺼내 닫힌 목록에 추가한다.  
  
(3) 선택한 사각형에 인접한 8 개의 사각형에 대해 탐색한다.

**a. 인접 사각형 중에 '닫힌 목록'에 있거나 장애물인 것은 제외하고, 나머지 '열린 목록'에  사각형이 없다면 '열린 목록'에 추가한다.** 그리고 추가된 사각형의 부모 노드를 현재 사각형으로 만든다. 추가된 사각형의 f(n), g(n), h(n) 비용을 기입한다.

**b.** **인접한 사각형이 이미 기존의 '열린 목록'에 있다면 현재 사각형을 기준으로 해당 인접한 사각형까지 이동할 때 g(n) 비용이 낮아지는 지 확인한다.** 비용이 더 낮아지지 않으면 아무것도 하지 않는다. 만약 현재 사각형을 통해 해당 인접 사각형까지 이동하는 데 g(n) 비용이 더 낮게 나온다면, 해당 인접 사각형의 부모 노드를 현재 사각형으로 바꾼다(부모 노드가 되면 해당 인접 사각형의 화살표가 부모 노드쪽으로 향하게 된다). 그리고 해당 인접 사각형의 f(n), g(n)을 다시 계산한다.

**3. 탐색 중단 조건**

(1) 경로 탐색 중 목표 사각형을 '열린 목록'에 추가했을 경우

(2) 열린 목록이 비게 되는 경우

(목표사각형을 찾는데 실패한 경우이다. 이 경우에는 길이 없는 경우다)

**4. 최단 경로 도출**

목표 사각형으로부터 각각의 사각형의 부모사각형을 향하여 시작사각형에 도착할 때까지 거슬러 올라간다. 부모 사각형이 어디인지는 화살표의 방향을 보면 알 수 있다. 화살표가 가리키는 곳이 부모 노드이다.

# Wiki의 A\* 설명

[다익스트라 알고리즘](https://namu.wiki/w/%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98" \o "다익스트라 알고리즘)을 확장하여 만들어진 경로 탐색 알고리즘이다. 드론이나 로봇 차량의 인공지능 주행을 구현하기 위해 개발되었다.  
에이스타 알고리즘 이라고 읽는다. 지금까지 가장 최소의 비용으로 도달한 지점부터 탐색하는 다익스트라 알고리즘의 원리를 차용한 것으로, A\* 알고리즘은

현재 상태의 비용을 *g*(*x*),

현재 상태에서 다음 상태로 이동할 때의 [휴리스틱](https://namu.wiki/w/%ED%9C%B4%EB%A6%AC%EC%8A%A4%ED%8B%B1" \o "휴리스틱) 함수를 *h*(*x*)라고 할 때,

둘을 더한 *f*(*x*)=*g*(*x*)+*h*(*x*) 가 최소가 되는 지점을 우선적으로 탐색하는 방법이다.

이 *f*(*x*)가 작은 값부터 탐색하는 특성상 [우선순위 큐](https://namu.wiki/w/%ED%81%90(%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0)" \l "s-3.2" \o "큐(자료구조))가 사용된다.

휴리스틱 함수 *h*(*x*)에 따라 성능이 극명하게 갈리며,

*f*(*x*)=*g*(*x*)일 때는 다익스트라 알고리즘과 동일하다.  
  
A\*를 사용하는 이유는 다익스트라를 직접 현실 문제에 적용하기가 매우 부담되기 때문이다. 당장에 네트워크 같은 디지털적인 공간이 아닌, 현실의, 사람이 사는 공간을 생각해보자. 사람이 다닐 수 있는 "거리"는 명백히 아날로그하다. 이것들을 전부 노드화 시키기에는 그 수가 엄청나게 많아질 수 있다. 그렇다면 탐색해야 하는 공간도 그만큼 커지게 되고, 시간 복잡도 역시 아득히 커질 것이다. 또한 어찌어찌 잘 노드화시켜서 다익스트라를 사용할 수 있는 상황을 만들어서 경로를 발견했다고 치자. 그렇게 탐색한 경로가 자동차 정체 구간, 출근길 등 다양한 변수로 인해 오히려 더 느려질 수 있는 경우도 발생하기 마련이다. 이러한 변수 때문에 A\* 알고리즘을 사용하는 것이다.

가장 기본이 되는 식은 다음과 같다.  
*f*(*x*)=*g*(*x*)+*h*(*x*)  
동작은 다음과 같이 된다.

1. *f*(*x*)를 오름차순 우선순위 큐에 노드로 삽입한다.
2. 우선순위 큐에서 최우선 노드를 pop한다.
3. 해당 노드에서 이동할 수 있는 노드를 찾는다.
4. 그 노드들의 *f*(*x*)를 구한다.
5. 그 노드들을 우선순위 큐에 삽입한다.
6. 목표 노드에 도달할 때까지 반복한다.

의사코드는 다음과 같다.

PQ.push(start\_node, g(start\_node) + h(start\_node)) //우선순위 큐에 시작 노드를 삽입한다.

**while** PQ is not empty //우선순위 큐가 비어있지 않은 동안

node = PQ.pop //우선순위 큐를 pop한다.

**if** node == goal\_node //만일 해당 노드가 목표 노드이면 반복문을 빠져나온다.

**break**

**for** next\_node in (next\_node\_begin...next\_node\_end)//해당노드에서 이동할수있는 다음노드들을 보는동안

PQ.push(next\_node, g(node) + cost + h(next\_node)) //우선순위 큐에 다음 노드를 삽입한다.

**print** goal\_node\_dist //시작 노드에서 목표 노드까지의 거리를 출력한다.

## 응용

다만 사용하는 휴리스틱에 따라서 최단거리를 확실하게 보장하는 대신 속도를 더 높이는 방법 등이 여럿 연구되어, 목적에 따라 개량하여 사용하는 경우가 많다. 따라서 Weighted A\*, LPA\*, IDA\*, D\*, D\* Lite[[1]](https://namu.wiki/w/A*%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98" \l "fn-1), **JPS** 등등 A\*의 개념에 기본을 두고 발달한 무수히 많은 알고리즘들이 있다. [학문적 길찾기 알고리즘 경진대회](https://movingai.com/GPPC/detail.html" \t "_blank" \o "https://movingai.com/GPPC/detail.html)[[2]](https://namu.wiki/w/A*%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98" \l "fn-2)를 보면 전처리를 하거나 메모리를 사용하거나 추가적인 전제[[3]](https://namu.wiki/w/A*%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98" \l "fn-3)를 넣는 등의 여러 접근 방식이 있음을 알 수 있다. 동일한 공간을 탐색하는 데 어느 알고리즘은 추가 메모리를 아예 소모하지 않는 반면 어느 알고리즘은 수십, 수백 기가바이트씩 차지하는 것을 볼 수 있다.

처음부터 길과 장벽을 모두 알고 시작하는 결정론적인 공간이 아니라 점진적으로 주위 환경을 탐색하면서 길을 탐색해 나가는 목적으로 개발된 변형이 D\*(Dynamic A\*)알고리즘인데 A\*보다 메모리를 대량으로 소비하는 탓에 여러 개의 오브젝트들이 동시에 빠르게 길을 찾아야 하는 게임 등에는 잘 쓰이지 않지만 단일 물체를 확실하게 조향해야 하는 자율주행차나 행성 탐사로봇 등에 사용되기 적합하다. 현세대의 대부분의 차량 [내비게이션](https://namu.wiki/w/%EB%82%B4%EB%B9%84%EA%B2%8C%EC%9D%B4%EC%85%98" \o "내비게이션)은 이를 기반으로 만들어져 있다고 보면 된다.

# A\* algorithm

## 다익스트라의 확장판 알고리즘

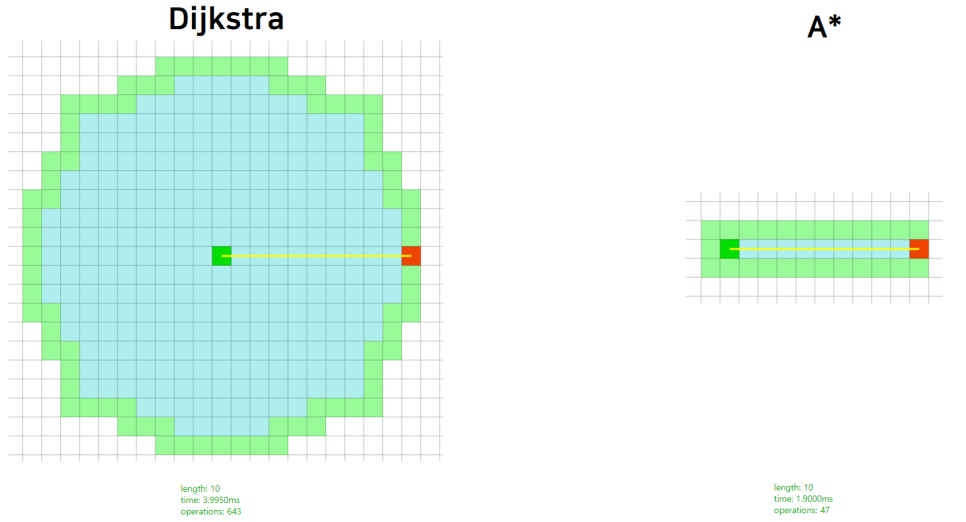
**A\* 알고리즘은 다익스트라를 확장해 만든 경로 탐색 알고리즘이다.**

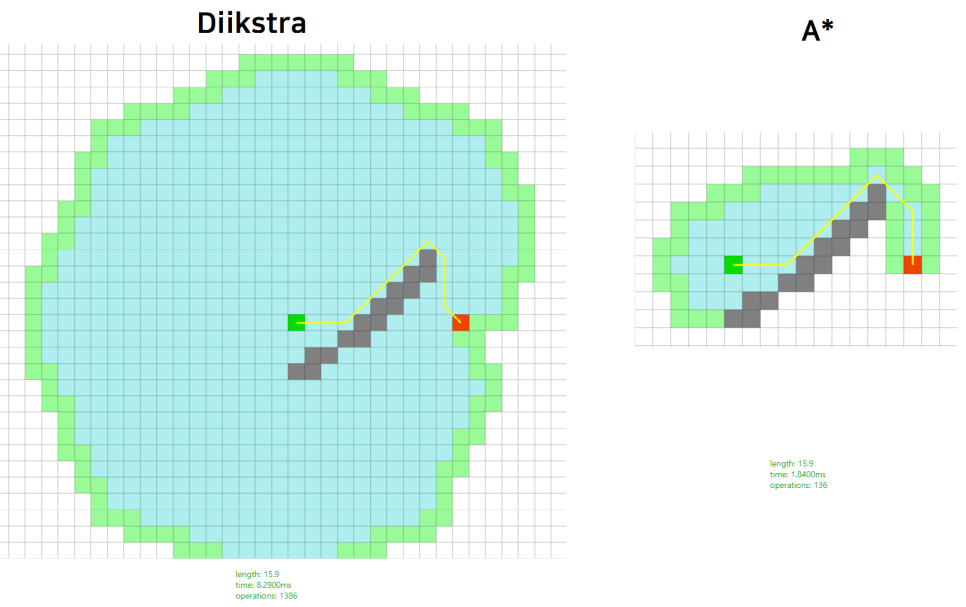
A\*가 나타나게 된 것은 다익스트라의 '현실 적용'의 어려움이다. 현실은 아날로그이기 때문에 다익스트라처럼 "거리"같은 개념들을 노드화 할 수 없기 때문이다. 그렇게 등장한 A\* 알고리즘의 파생형으로 IDA\*, JPS 등이 또 등장했으며, A\*를 업그레이드한 버전이 D\*(Dynamic A\*) 알고리즘이 있다. 현세대 네비게이션들은 대부분 D\* 알고리즘을 사용하고 있다고 한다.

**A\*의 장점 중 하나는 탐색 시간에 있다.** 휴리스틱에 따라 조금씩 다르긴 하지만 휴리스틱으로 '맨해튼 거리' 계산을 한다면 **다익스트라에 비해 빠른 탐색이 가능하다.**

<http://qiao.github.io/PathFinding.js/visual/>

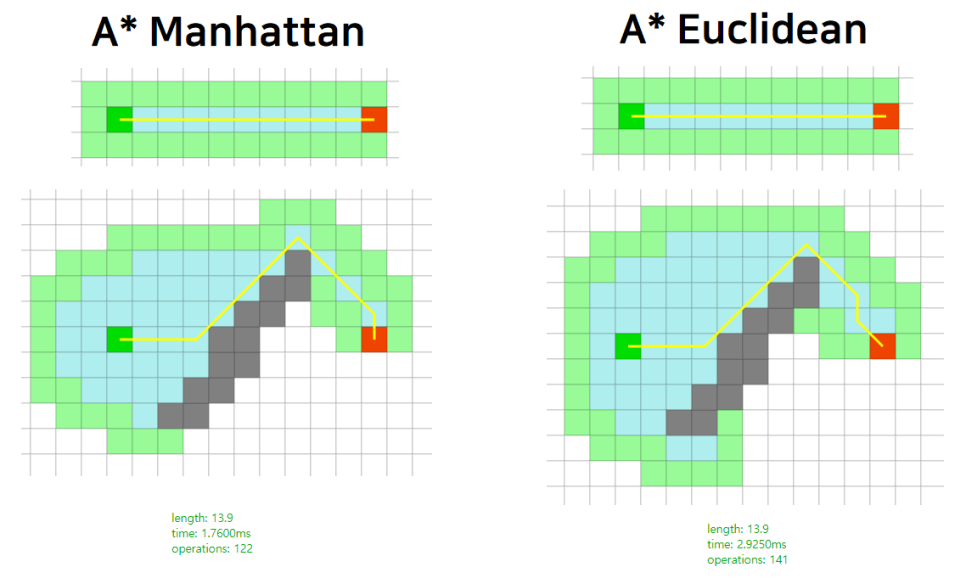
위 사이트에서 직접 여러 길찾기 알고리즘을 테스트 해볼 수 있다.





위 사이트에서 **시뮬레이션을 비교해보면 다익스트라와 A\*의 성능차이를 크게 느낄 수 있다.** 시간과 연산 횟수가 모두 눈에 띄게 차이난다. A\*가 목표를 향해 이동하는 것이라면 다익스트라는 BFS처럼 반경을 늘려가며 주변을 계속 돌다가 목표가 나오길 바라기 것과 같기 때문이다.

물론 **같은 A\*라도 앞서 말했던 것처럼 휴리스틱 함수에 따라 성능 차이가 난다.** 대부분 맨해튼 거리를 휴리스틱으로 사용하고 위의 시뮬레이션 역시 맨해튼 거리를 이용했다. 다른 휴리스틱 함수 중 하나로는 유클리드 거리가 있는데 둘을 비교하면 다음과 같다.



두 지점사이에 장애물이 없는 경우에는 똑같이 나오지만, 장애물이 나온다면 맨해튼에 비해 유클리드가 더 많은 영역을 탐색한 것을 볼 수 있다. 시간과 연산 횟수 역시 유클리드가 더 크게 나왔다. **그냥 편하게 생각하면 휴리스틱은 '맨해튼 거리'를 쓴다 생각하는게 좋다.**

​

A\* 알고리즘의 전반적인 내용과 휴리스틱 관련된 부분은

<https://www.redblobgames.com/pathfinding/a-star/introduction.html>

[Red Blob Games: Introduction to A\*](http://Red Blob Games: Introduction to A*Interactive tutorial for A*, Dijkstra's Algorithm, and other pathfinding algorithmswww.redblobgames.com)

[Interactive tutorial for A\*, Dijkstra's Algorithm, and other pathfinding algorithms](http://Red Blob Games: Introduction to A*Interactive tutorial for A*, Dijkstra's Algorithm, and other pathfinding algorithmswww.redblobgames.com)

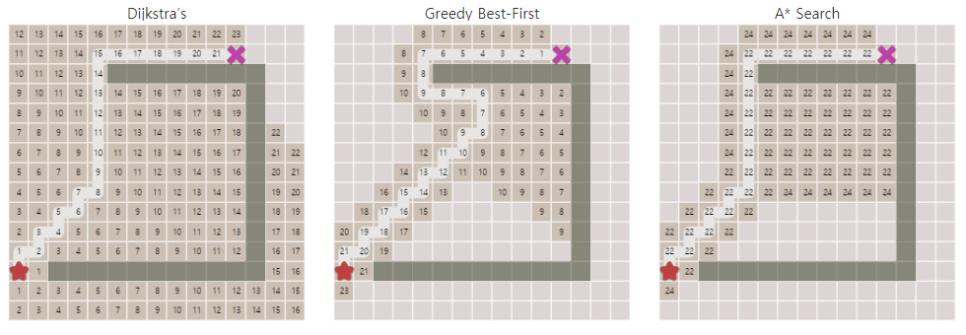
[www.redblobgames.com](http://Red Blob Games: Introduction to A*Interactive tutorial for A*, Dijkstra's Algorithm, and other pathfinding algorithmswww.redblobgames.com)

<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>

[theory.stanford.edu](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html" \t "_blank)

위 사이트에서도 확인이 가능하다. 매우 자세한 설명과, 고퀄 시뮬레이션 등으로 아주 보기 좋게 되어 있다. 아주 강추하는 사이트로 게임과 관련된 알고리즘들이 많이 올라와있다. 게임 알고리즘 사이트에 올라올 정도로 A\* 알고리즘은 게임에서도 많이 사용되며, 타워 디펜스류 게임에서 타워를 향해 몬스터가 최단 거리로 이동해야 된다거나, 스타크래프트같은 RTS에서 랠리 포인트를 찍는 등의 경우에 사용된다. 위 사이트에서 BFS, 다익스트라, A\*의 특징을 표현한 그림은 아래와 같다.

출처 : redblobgames



또한 여기서도 다익스트라와 A\*의 어마어마한 차이를 느낄 수 있다.

​

## 알고리즘 설명

**A\*는 출발부터 현재 꼭지점 x까지의 비용을 g(x), 현재 꼭지점 x에서 목표 지점까지의 비용 h(x)라고 할 때 이 둘을 더한 f(x) = g(x) + h(x)가 최소가 되는 지점을 우선 탐색하여 경로를 찾는 방법이다.** 작은 f(x)를 찾기 위해 여기서도 우선순위 큐가 사용되고, h(x)에 따라 성능의 차이가 발생한다. h(x)로는 맨해튼 거리, 유클리드 거리, Diagonal distance 등이 있고 **여기서는 '맨해튼 거리'를 기준으로 설명하도록 하겠다.**

기본적인 알고리즘 과정은 다음과 같다.

1. f(x)를 기준으로 오름차순 정렬되는 우선순위 큐를 생성한다.

2. 시작 노드 S를 우선순위 큐에 삽입한다.

3. 우선순우 큐에서 노드를 pop하고 해당 노드에서 이동할 수 있는 노드를 찾는다.

3-1. pop된 노드는 폐쇄 노드 리스트에 들어간다.

4. 그 노드들의 f(x)를 구해 우선순위 큐에 삽입한다.

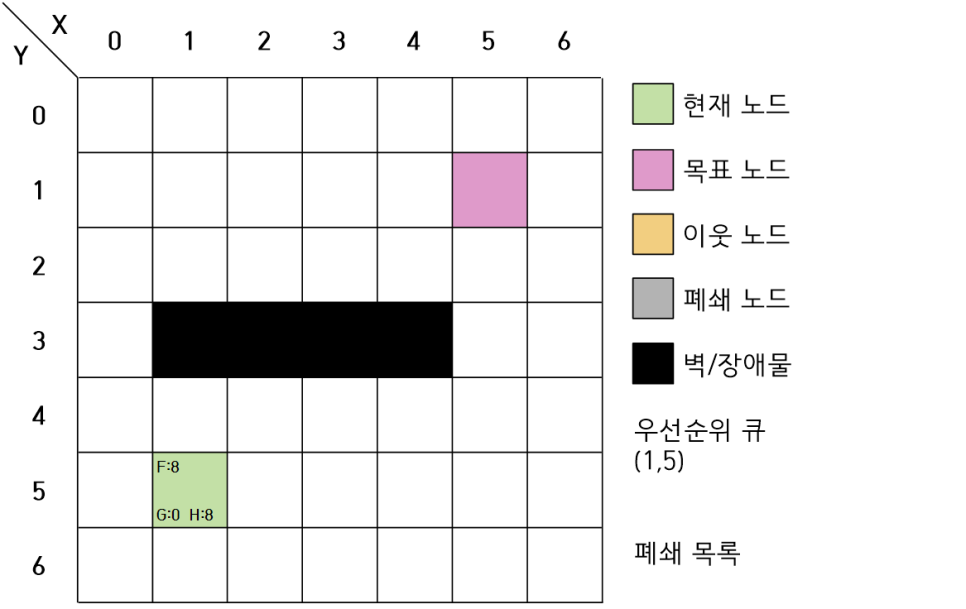
5. 목표 노드에 도착할 때 까지 3~4를 반복한다.

​

이동할 수 있는 노드를 찾을 때는, 상하좌우의 4방향인지 대각선까지 이동하는 8방향인지에 따라 다르다. 4방향으로만 가능하다면 한 칸마다의 가중치를 1로 잡아도 10으로 잡아도 상관없지만, 대각선이라면 직각삼각형을 생각해서 보통 상하좌우를 10, 대각선을 14정도로 많이 잡는다.

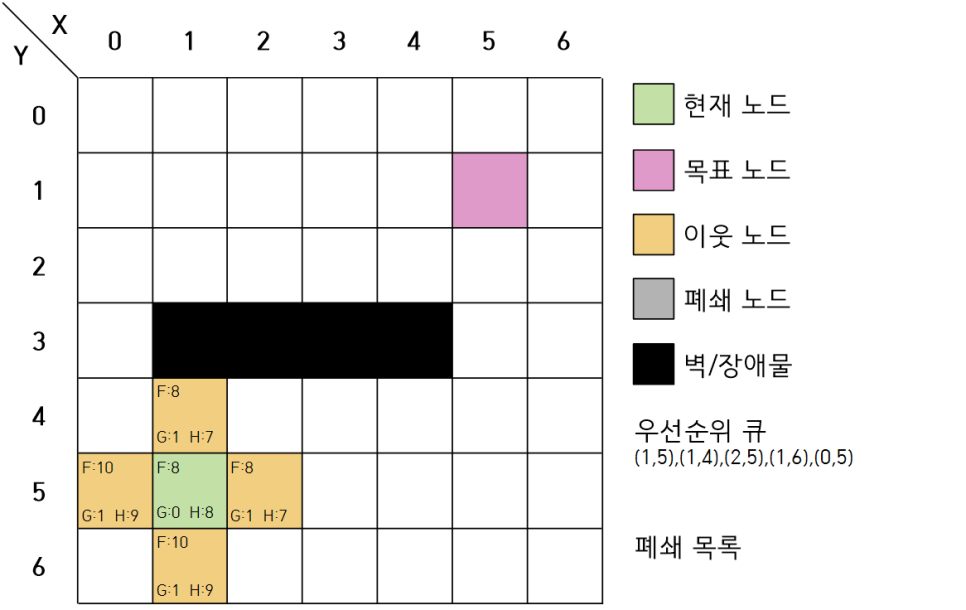
위 내용을 그림으로 보면 다음과 같다.

**0.(1,5)에서 (5,1)로 가는 맵을 생성한다. (1,3) ~ (4,3)은 이동할 수 없는 장애물이다.**



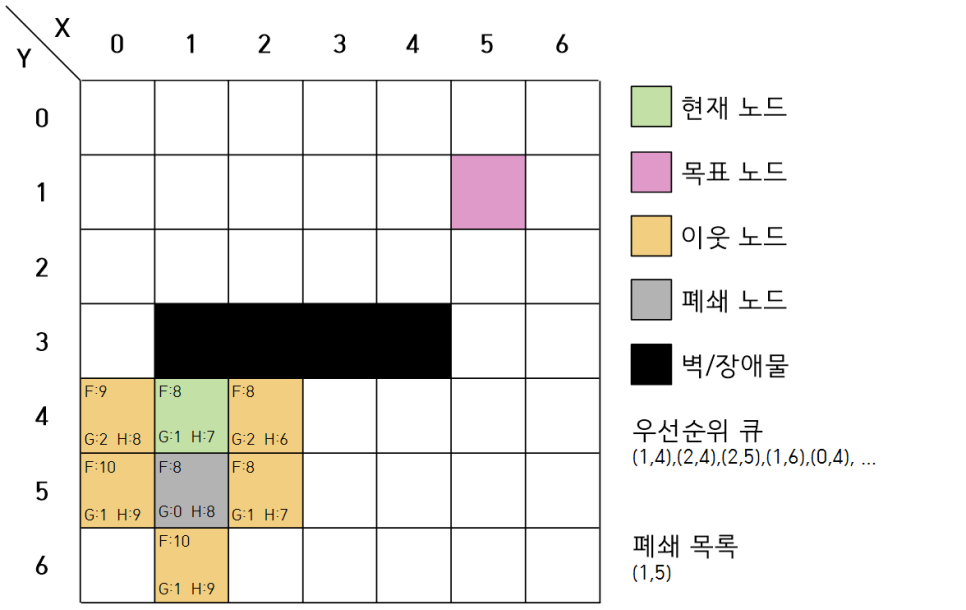
한 칸 당 이동 거리는 1로 잡았고 상하좌우의 4방향만 고려하도록 하겠다. (1,5)에서 (5,1)까지의 맨해튼 거리는 8이고, 시작지점이므로 G는 0이다.

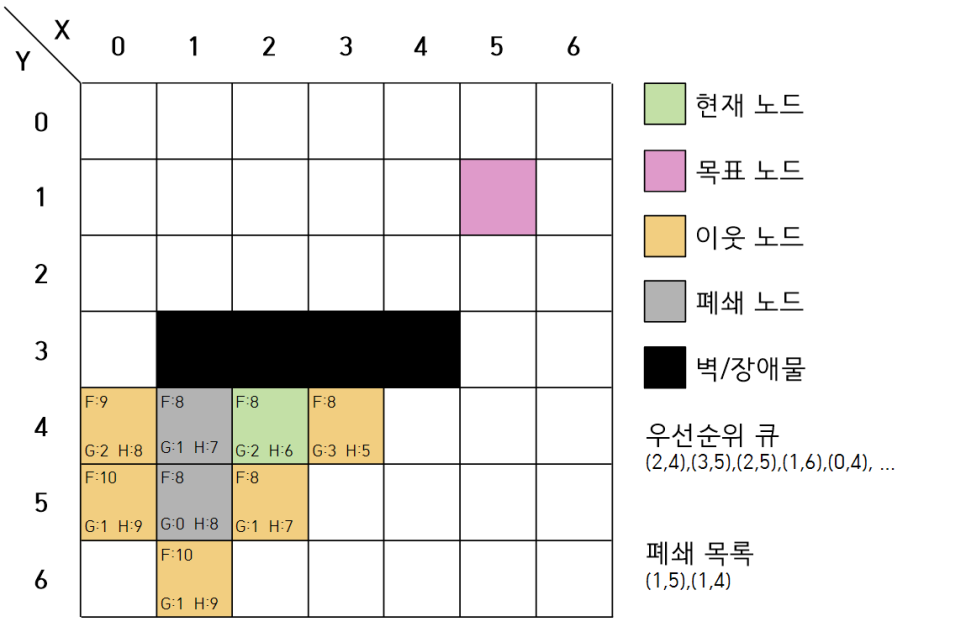
**1. (1,5)의 주변 노드를 넣는다. 탐색 순서는 위쪽부터 시계 방향이다.**



h(x)를 계산할 때는 벽과 상관없이 좌표로만 맨해튼 거리를 구한다.

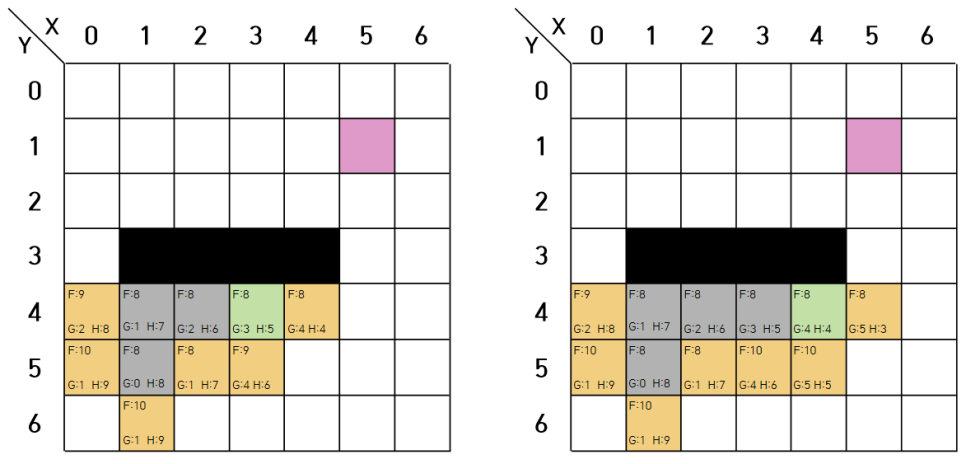
**2. 사용된 (1,5)는 폐쇄 목록으로 들어가고 F가 가장 작은 노드를 선택한다. 값이 같다면 G가 큰 순서로, G가 같다면 더 먼저 삽입된 노드를 우선으로 한다.**

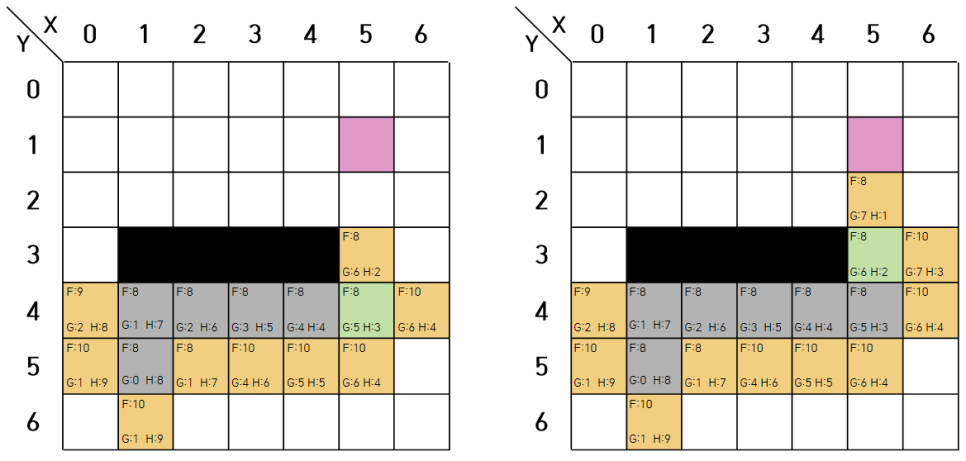


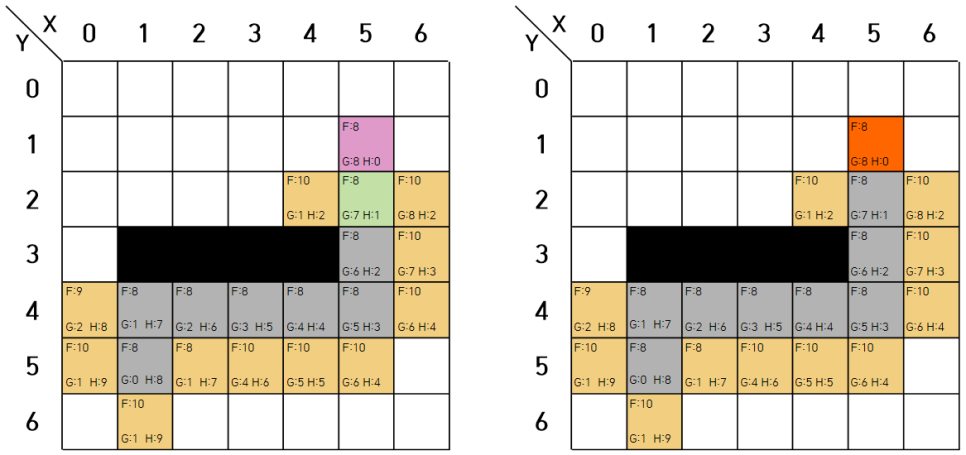


큐에서 F가 가장 작고 먼저들어온 (1,4)가 선택된다. 그 다음엔 (2,4)와 (2,5)중 더 늦게 삽입된 (2,4)가 선택 된다.

**3. 이런식으로 반복하면서 목표 노드를 향한다.**



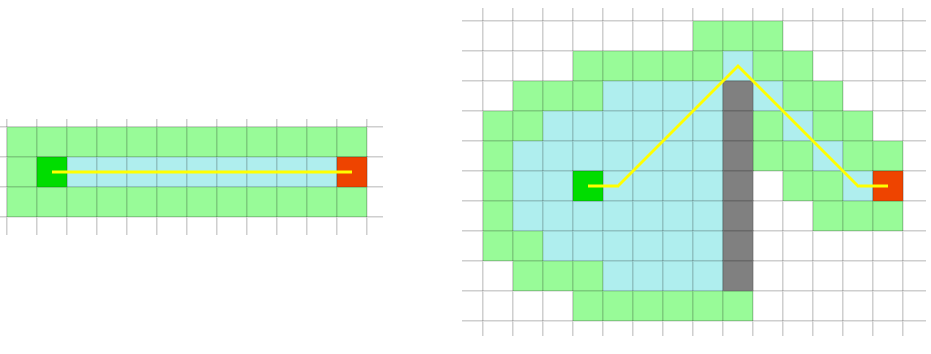




**4. 과정이 완료되면 폐쇄 리스트를 통해 경로를 알아낼 수 있다.**



실제 구현에서는 폐쇄 목록을 그대로 사용하지 않고 뒤집어서 역추적을 한다. 그 이유는 폐쇄 목록에 들어가는 노드가 정확하게 경로에 해당하는 노드만 들어가는 것이 아니기 때문이다. 앞서 봤던 시뮬레이션을 보면



연두색 노드가 우선순위 큐에 들어간 노드들, 하늘색 노드가 폐쇄 목록에 들어간 노드에 해당한다. 장애물이 없는 왼쪽의 경우 폐쇄 목록이 정확하게 경로와 일치하지만 중간에 벽이 있는 오른쪽은 그렇지 않다. 이런 경우 폐쇄 목록에서 정확하게 경로만 뽑아내기 위해 역추적이 필요하다.

## 코드 설명

​

pq.enqueue(start\_node, g(start\_node) + h(start\_node)) // 우선순위 큐에 시작 노드를 삽입한다.

while pq is not empty // 우선순위 큐가 비어있지 않은 동안

node = pq.dequeue // 우선순위 큐에서 pop한다.

if node == goal\_node // 만약 해당 노드가 목표 노드이면 반복문을 빠져나온다.

break

for next\_node in (next\_node\_begin...next\_node\_end) // 해당 노드에서 이동할 수 있는 다음 노드들을 보는 동안

pq.enqueue(next\_node, g(node) + cost + h(next\_node)) // 우선순위 큐에 다음 노드를 삽입한다.

return goal\_node\_dist // 시작 노드에서 목표 노드까지의 거리를 출력한다.

위 수도 코드도 위키백과에서 참조했다.

C++로 구현한 코드는 다음과 같다.

#include <iostream>

#include <vector>

#include <math.h>

#include <queue>

using namespace std;

typedef struct node {//노드

int x, y;//좌표

int G, H;//g(node), h(node)

pair<int, int> parent;//역추적에 쓰일 이전 노드

}Node;

struct cmp {//우선순위 큐 비교 함수

bool operator()(const node u, const node v) {

if (u.G + u.H > v.G + v.H) return true;//F는 작은게 위로 오게

else if (u.G + u.H == v.G + v.H) {//F가 같다면 G가 큰게 위로 오게

if (u.G < v.G) return true;

else return false;

}

else return false;

}

};

void print\_map(vector<vector<int>> map) { // 맵 출력 함수

for (int i = 0; i < map.size(); i++) {

for (int j = 0; j < map.size(); j++) cout << map[i][j] << " ";

cout << '\n';

}

}

int Astar(vector<vector<int>> map, pair<int, int> start, pair<int, int> goal) {

priority\_queue<Node, vector<Node>, cmp> open; // 우선순위 큐

bool close[10][10] = { 0, }; // 폐쇄 리스트(리스트&visit)

vector<Node> close\_list;

Node s\_node; // 시작 노드

int cx[8] = { 0,1,0,-1,1,1,-1,-1 }; // 방향 좌표 ↑→↓←↘↗↙↖

int cy[8] = { -1,0,1,0,1,-1,1,-1 };

// 시작지점 초기화

s\_node.x = start.second;

s\_node.y = start.first;

s\_node.G = 0;

s\_node.H = (abs(goal.second - s\_node.x) + abs(goal.first - s\_node.y)) \* 10;

s\_node.parent = make\_pair(-1, -1); // 시작 노드의 부모 노드는 -1,-1

open.push(s\_node);

close[s\_node.y][s\_node.x] = true; // 폐쇄 노드

vector<vector<int>> result = map;

while (open.size()) {

int x = open.top().x; // 우선순위 큐에서 top 정보 추출

int y = open.top().y;

int G = open.top().G;

close\_list.push\_back(open.top());

result[y][x] = 8;

open.pop();

if (x == goal.second && y == goal.first) break; // 도착 지점이 나오면 끝

Node add;

for (int i = 0; i < 4; i++) {// top 노드에서 상하좌우 4방향으로 탐색(i<8이면 8방향)

int nextX = x + cx[i];

int nextY = y + cy[i];

if (nextX >= 0 && nextX < map.size() && nextY >= 0 && nextY < map.size()) {

if (map[nextY][nextX] != 1 && close[nextY][nextX] == false) {

add.x = nextX;

add.y = nextY;

add.G = i < 4 ? G + 10 : G + 14;// 상하좌우면 10, 대각선이면 14(√200)

add.H = (abs(goal.second - add.x) + abs(goal.first - add.y)) \* 10;

add.parent = make\_pair(y, x); // 기존 top노드를 부모 노드로 설정

close[nextY][nextX] = true;

result[nextY][nextX] = 9;

open.push(add); // 우선순위 큐에 삽입

system("cls");

print\_map(result);

}

}

}

}

int px = close\_list.back().x;

int py = close\_list.back().y;

while (close\_list.size()) { // close\_list를 역추적해 경로 탐색

if (px == close\_list.back().x && py == close\_list.back().y) {// 목표 노드부터 부모 노드를 탐색해 역추적

result[py][px] = 4;

px = close\_list.back().parent.second;

py = close\_list.back().parent.first;

system("cls");

print\_map(result);

}

close\_list.pop\_back();

}

return 0;

}

int main() {

int size = 7;

/\*

cout << "맵 사이즈 : ";

cin >> size;

cout << "맵 입력(0:길, 1:벽, 5: 출발점, 6 : 도착점)\n";

\*/

vector<vector<int>> map(size, vector<int>(size, 0));

pair<int, int> start, goal;

start = { 5, 1 };

goal = { 1, 5 };

map = { {0,0,0,0,0,0,0},

{0,0,0,0,0,6,0},

{0,0,0,0,0,0,0},

{0,1,1,1,1,0,0},

{0,0,0,0,0,0,0},

{0,5,0,0,0,0,0},

{0,0,0,0,0,0,0},

};

print\_map(map);

Astar(map, start, goal);

start = { 4, 4 };

goal = { 8, 6 };

map = { {0,0,0,0,0,0,0,0,0,0},

{0,0,0,0,0,0,0,0,0,0},

{0,0,1,1,1,1,1,1,0,0},

{0,0,0,0,0,0,0,1,0,0},

{0,0,0,0,5,0,0,1,0,0},

{0,0,0,0,0,0,0,1,0,0},

{0,0,1,1,1,1,1,1,0,0},

{0,0,0,0,1,0,0,0,0,0},

{0,0,0,0,1,0,6,0,0,0},

{0,0,0,0,1,0,0,0,0,0},

};

print\_map(map);

Astar(map, start, goal);

return 0;

}

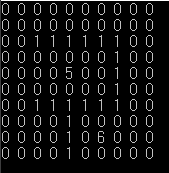
쉬운 관리를 위해 node 구조체를 만들어 해당 노드의 좌표, g(x), h(x)의 값, 부모 노드의 좌표를 관리하도록 했다. 메인 함수에서 2차원 배열로 그리드 맵을 만들어 계산하도록 했다. 여기서 0은 그냥 길, 1은 벽, 5는 출발점, 6은 도착지점을 의미한다. 첫 번째 테스트는 위의 그림 설명과 동일하다.



(좌) 탐색 (우) 역추적

9는 우선순위 큐에 들어간 노드들, 8은 폐쇄 리스트에 들어간 노드에 해당한다. 4는 역추적으로 찾은 경로를 뜻한다.

두번째로 길이 막혀 돌아가는 경우



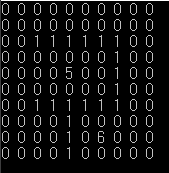
(좌) 탐색 (우) 역추적

기본적인 맨하튼 거리 휴리스틱 특성상 벽을 고려하지 않고 F가 작은 곳을 쫓기 때문에 직선거리가 가까운 오른쪽, 그 다음 아래쪽으로 향하지만 모두 길이 없어서 돌아가는 모습이다.

위 두가지 케이스를 8방향으로 탐색하면 아래와 같다.

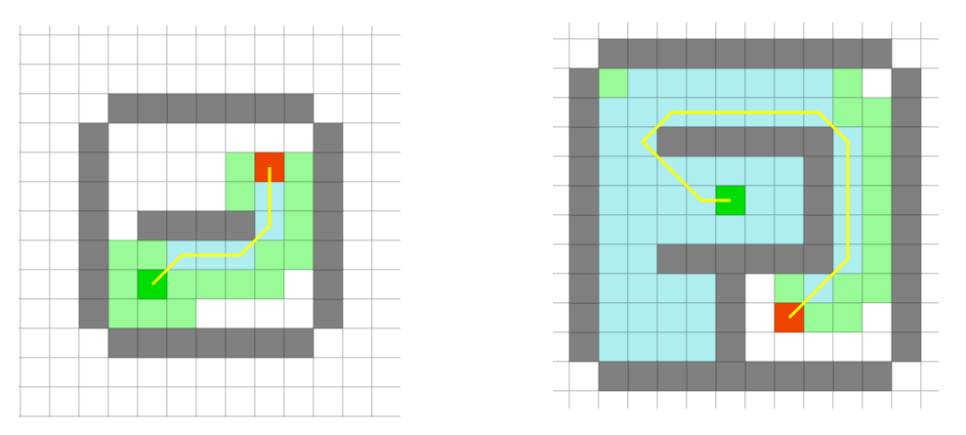


(좌) 탐색 (우) 역추적



(좌) 탐색 (우) 역추적

조금이지만 탐색 범위와 경로의 길이가 8방향이 4방향보다 짧은 것을 볼 수 있다.



시뮬레이션에서도 같은 결과를 확인 할 수 있다.

​

# Path-finding in 2d maze

## Path-finding in 2d maze for two mobile robots using A-Star algorithm

This repository contains source code for ENPM809Y final project on search algorithm for autonomous robots moving on a 2d maze. The source code is in C++ 11. The maze will printed on the terminal and the user shall be prompted to enter the locations of two robots and also two targets. Later, the user will be asked to assign task for one of the robots. The other robot automatically assigned the unassigned task. The two robots complete the navigation task using an A\* algorithm.

## Assumptions

* You have a C++ 11 compatible compiler
* You have a Mac/ Windows/ Ubuntu operating system
* You have CMake version 3.13 (minimum)
* You have git installed on your machine

## Build and Run

In order to run the project, do the following in a new terminal :

git clone https://github.com/arunumd/A-Star-Algorithm-for-path-finding-in-2d-maze-Cpp.git

cd A-Star-Algorithm-for-path-finding-in-2d-maze-Cpp

mkdir build

cd build

cmake.. && make

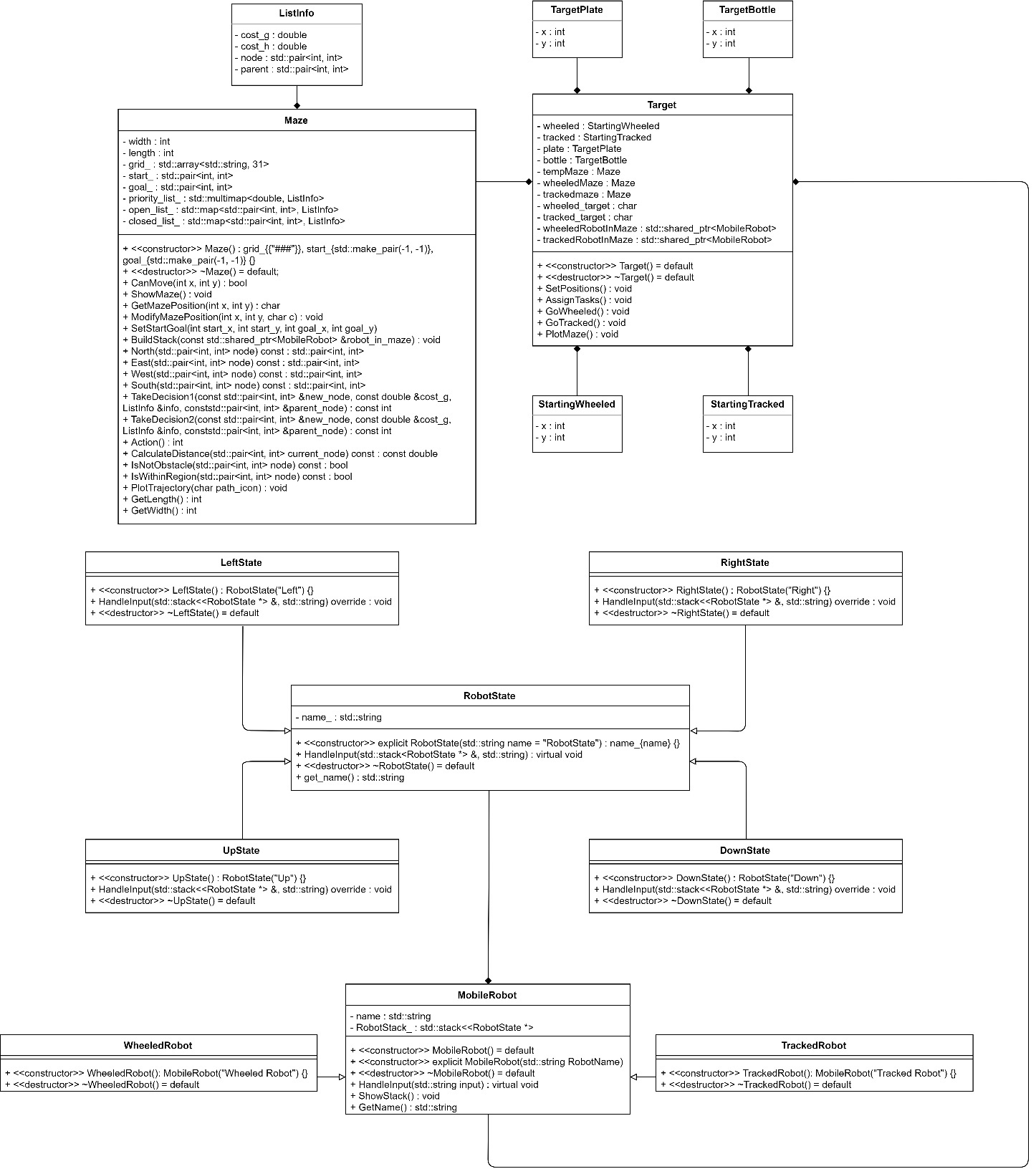
./Final\_Version\_809Y\_Project

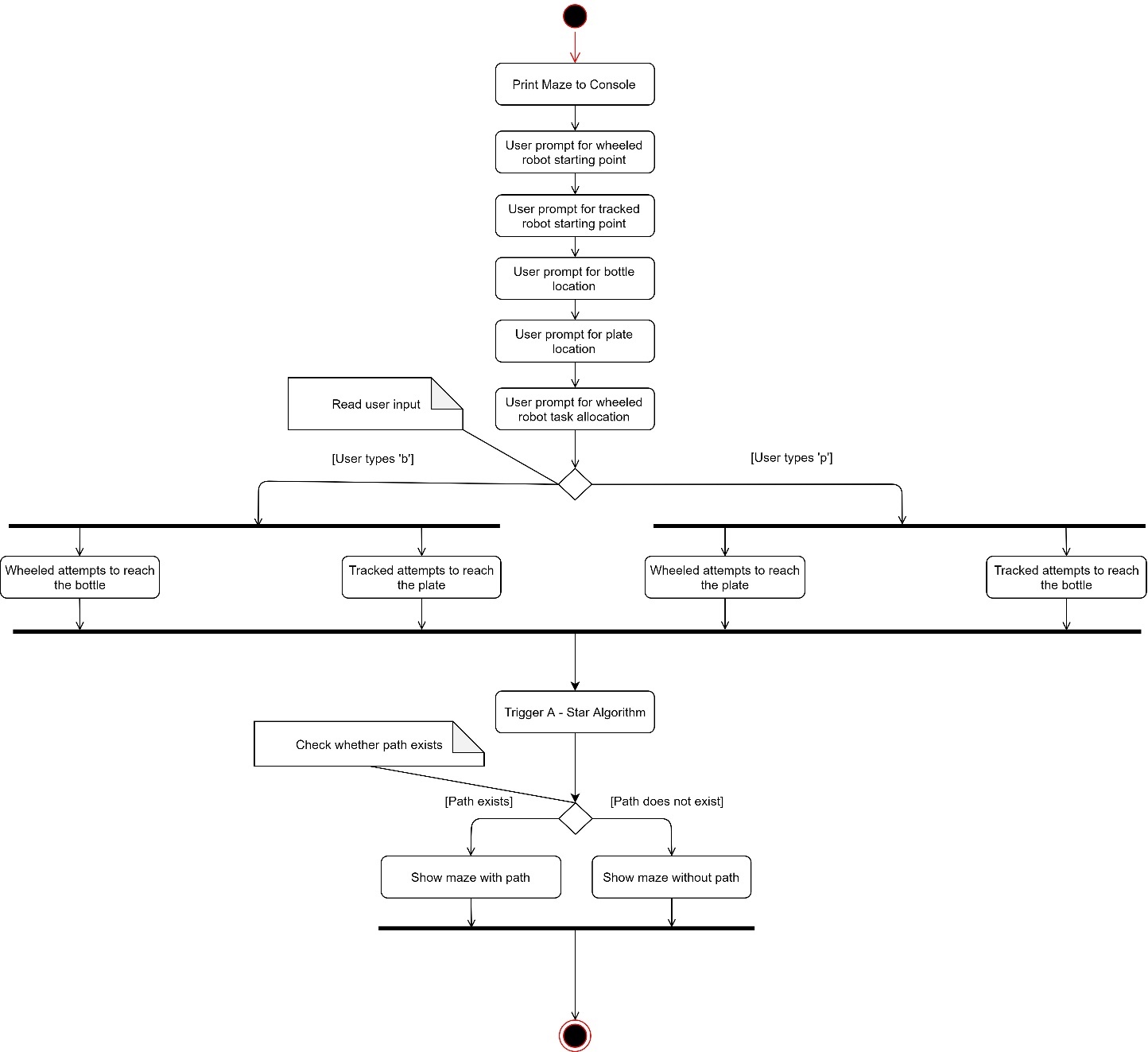
After these please follow on-screen instructions

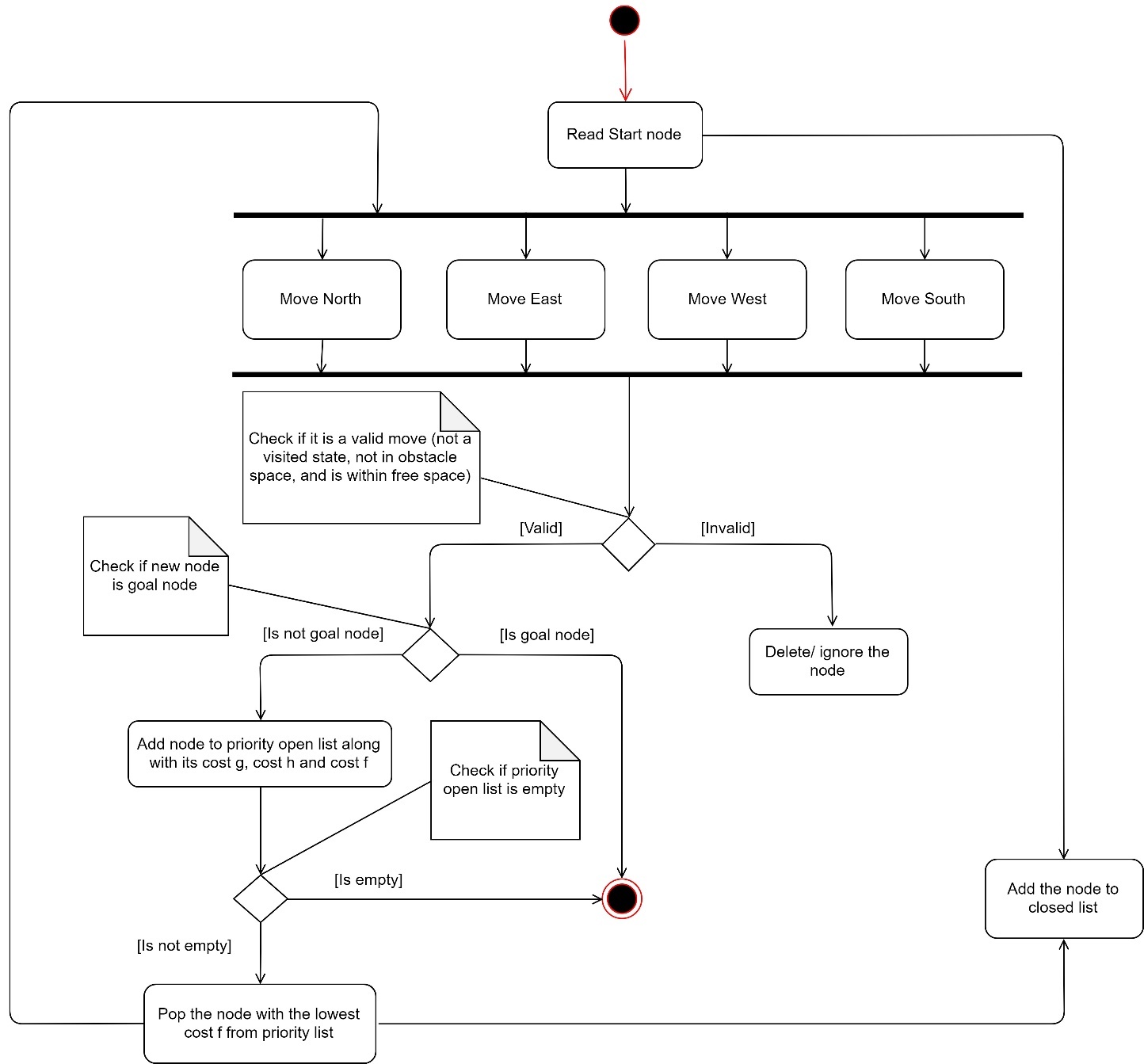
## Results



## UML Class Diagram







# A\* 알고리즘 이란?

다익스트라에서 조금 더 추가된 내용인데 그 조금이 바로 출구를 알고 있다는 점이다.

즉 출구(목적지)에 가까워질수록 가산점이 있다는 점이있다.

결국에는 입구에서 부터 얼마나 이동을 하는지도 중요하지만, 출구에서 부터 얼마나 떨어져있는지도 같이 생각을 하여서 가산점을 줘서 점수를 줘서 정한다는 점이 추가 된것이고, 그 외에 점은 다익스트라랑 비슷하다고 볼 수 있다.

A\* 알고리즘을 사용하는 이유는 BFS와 다익스트라에는 목적지의 개념이 없기떄문이다.

점수를 줄 때 우리는

입구에서부터 얼마나 떨어져 있는지?

출구에서부터 얼마나 떨어져 있는지?

그래서 점수를 매기는 공식을 보자면  
**F = G + H  
F = 최종 점수(작을 수록 좋음)  
G = 시작점에서 해당 좌표까지 이동하는데 드는 비용  
H = 목적지에서 해당 좌표까지 이동하는데 드는 비용**  
이런식으로 점수를 매긴다. 하지만 공식은 유동적으로 바꿀 수 있기 때문에 공식에 목을 매달 필요는 없다고 본다. 그럼 구현을 해보자.

A\* 알고리즘 구현

일단 우선순위 큐에 넣을 PQNode라는 것을 만들어준다.

struct PQNode

{

PQNode(int32 f, int32 g, Pos pos) : f(f), g(g), pos(pos) { }

bool operator<(const PQNode& other) const { return f < other.f; }

bool operator>(const PQNode& other) const { return f > other.f; }

int32 f; // f = g + h

int32 g;

Pos pos;

};

그리고 일단 기본적인 코드를 만들어준다.

// F = G + H

// F = 최종 점수(작을 수록 좋음)

// G = 시작점에서 해당 좌표까지 이동하는데 드는 비용

// H = 목적지에서 해당 좌표까지 이동하는데 드는 비용

Pos start = \_pos;

Pos dest = \_board->GetExitPos(); // 목적지

Pos front[] =

{

Pos(-1, 0), // UP

Pos(0, -1), // LEFT

Pos(1, 0), // DOWN

Pos(0, 1), // RIGHT

Pos(-1, -1), // UP\_LEFT

Pos(1, -1), // DOWN\_LEFT

Pos(1, 1), // DOWN\_RIGHT

Pos(-1, 1), // UP\_RIGHT

};

여기서 cost같은 것을 추가해서 이동할 때 코스트를 넣어줄 수 있다.

int32 cost[] = {10, 10, 10, 10, 14, 14, 14, 14 };

이런식으로 만들어주는데 1로 만들면 대각선 거리가 1.4가 되어버리기 때문에 10으로 만들어서 14로 만들어 주는게 더 빠르다. (정수 연산이 실수 연산보다 빠름)

그다음으로 맵의 사이즈를 받아와준다.

const int32 size = \_board->GetSize();

또 다익스트라에서 했었던 best(베스트 케이스)를 y,x 기준으로 만들어야 하기 때문에

// best[y][x] -> 지금까지 (y,x)에 댛나 가장 좋은 비용 (작을 수록 좋음)

vector<vector<int32>> best(size, vector<int32>(size, INT32\_MAX));

이 부분은 나중에 map을 배운다면 map으로 구현가능하다.

그리고 CloseList라는 것을

// ClosedList -> closed[y][x] -> (y, x)에 방문을 했는지 여부

// Option) 사실 best만으로 판별 가능

vector<vector<bool>> closed(size, vector<bool>(size, false));

이런식으로 만들어서 closed[y][x]가 true라면 그 지역에 방문을 했었다는 것을 알고 방문을 하지 않게 만들어 준다.

그리고 우리가 항상 만들었던 것 처럼 parent로 추적할 수 있게 만들어준다.

// 부모 추적 용도

vector<vector<Pos>> parent(size, vector<Pos>(size, Pos(-1, -1)));

그리고 우리는 예약시스템으로 구현을 하고 뒤늦게 더 좋은 경로가 발견될 수 있기 때문에 예외 처리를 해줘야 한다. 그럼 OpenList 즉 우선순위 큐를 만들어준다.

// OpenList : 지금까지 발견된 목록

priority\_queue<PQNode, vector<PQNode>, greater<PQNode>> pq;

여기서 하나 헷갈리는 것이 있을 수 있는데  
발견과 방문의 차이이다.  
발견은 발견은 하였지만 나중에 더 좋은 경로가 나올 가능성을 염두해두고 있는 것이고  
방문은 이미 완벽한 베스트 케이스여서 실행을 하는 상태이다.

그리고 초기값을 이런식으로 지정을 해준다.

// 초기값

{

int32 g = 0;

int32 h = 10 \* (abs(dest.y - start.y) + abs(dest.x - start.x));

pq.push(PQNode(g + h, g, start));

best[start.y][start.x] = g + h;

parent[start.y][start.x] = start;

}

일단 g는 시작점부터 현재 위치인데 시작점이니 0을 넣어준다.  
그리고 h는 목적지에서 현재 위치까지 이동하는데 얼마나 노력을 하는가? 이기 때문에  
우리는 h를 목적지의 y,x의 값에서 현재 위치의 y,x를 빼주고 둘을 더하고 10을 곱해주는 공식을 만들 것이다.

그리고 F는 G+H이기 때문에 실질적으로 우선순위 큐에 넣어줄 때는 g+h를 넣어주면 된다.  
그리고 best[start.y][start.x]에는 g + h를 넣어준다.  
또 parent[start.y][start.x]에는 start를 넣어준다.

while (pq.empty() == false)

{

// 제일 좋은 후보를 찾는다

PQNode node = pq.top();

pq.pop();

// 동일한 좌표를 여러 경로로 찾아서

// 더 빠른 경로로 인해서 이미 방문(closed)된 경우 스킵

if (closed[node.pos.y][node.pos.x])

continue;

// 기껏 등록했더니만. 나보다 더 우수한 후보가 있다?

if (best[node.pos.y][node.pos.x] < node.f)

continue;

// 방문

closed[node.pos.y][node.pos.x] = true;

// 목적지에 도착했으면 바로 종료

if (node.pos == dest)

break;

for (int32 dir = 0; dir < 8; dir++)

{

Pos nextPos = node.pos + front[dir];

// 갈 수 있는 지역은 맞는지 확인

if (CanGo(nextPos) == false)

continue;

if (closed[nextPos.y][nextPos.x])

continue;

int32 g = node.g + cost[dir];

int32 h = 10 \* (abs(dest.y - nextPos.y) + abs(dest.x - nextPos.x));

// 다른 경로에서 더 빠른 길을 찾았으면 스킵

if (best[nextPos.y][nextPos.x] <= g + h)

continue;

// 예약 진행

best[nextPos.y][nextPos.x] = g + h;

pq.push(PQNode(g + h, g, nextPos));

parent[nextPos.y][nextPos.x] = node.pos;

}

}

그리고 항상 하였던 것처럼 우선순위 큐가 빌때까지 작동하는 while문을 만들어주고 그리고 여기서 제일 좋은 후보를 찾아주면 된다. 일단 pq.top()으로 값을 뽑아주고 pop으로 내보낸다.

그 다음에 동일한 좌표를 여려 경로로 찾아서 더 빠른 경로로 인해서 이미 방문(closed)된 경우 스킵을 해준다. 그 후에 나보다 더 우수한 후보가 있다면 스킵을 해준다.

다음에는 방문을 해준다. 즉, closed를 true로 해준다.

그리고 목적지에 도착했으면 종료하는 코드를 만든다.

그리고 for문으로 상하좌우를 움직여 준다.  
nextPos에 node의 위치와 방향에 따른 위치의 좌표를 더해줘서 다음 좌표를 구한다.

그리고 만들어둔 CanGo함수를 통하여 간선이 연결되있는가?를 구해 갈 수 있는 지역이 맞는지 확인을 한다. 그리고 다시 한 번 방문이 된 곳인지 체크를 한다.

그 다음에는 다음 좌표까지 가는 비용을 계산해보고 체크를 해서 이전에 더 빠른 경로가 있었다면 스킵을 하고 그게 아니라면 그곳으로 가능 기능을 만들어준다.

일단 g를 계산하는데 node.g즉 이전에 좌표까지의 이동값에다가 cost[dir] 즉 위에서 cost로 방향에 따른 비용을 관리하고 있기 때문에 그것을 이용해서 새로운 g를 계산해준다. 그 다음에는 h를 구해주는데 위에서 말했던 공식을 사용하여서 목적지의 y,x에서 다음 좌표의 y,x를 빼주고 더한후에 10을 곱해서 h를 구해준다.

그리고 우리는 다른 경로에서 더 빠른 길을 찾았으면 스킵을 해줘야 한다.  
best[nextPos.y][nextPos.x] <= g + h이렇다면 스킵을 해준다.

여기까지 통과를 했다면 예약을 해주면 된다.  
하지만 이것은 최종적인 예약은 아니고 갱신이 될 수 있다.

그 후에 BFS와 시작점 구하고 뒤바꾸는 것은 똑같기 때문에 이런식으로 코드를 짜준다.

\_path.clear();

Pos pos = dest;

while (true)

{

\_path.push\_back(pos);

// 시작점

if (pos == parent[pos.y][pos.x])

break;

pos = parent[pos.y][pos.x];

}

/\*vector<Pos> temp(\_path.size());

for (int i = 0; i < \_path.size(); i++)

temp[i] = \_path[\_path.size() - 1 - i];

\_path = temp;\*/

std::reverse(\_path.begin(), \_path.end());

그러면 이런식으로 길찾기를 하는 것을 볼 수 있다.

정리

일단 길찾기를 하기 위한 긴 여정이 끝났다.

여기서 우리가 알고 가야할 것은 적어도 개념을 알고가자! 이다.  
코드는 구현을 못해도 우리가 언제가 이 개념을 기억하고 가져간다면 분명 나중에 도움되는 일이 많을 것이다.

결국 A\* -> Dijkstra -> BFS -> Graph (PQ)  
라고 볼 수 있는 것이다.

이걸 전체적으로 설명을 해보자면  
일단 그래프는 정점과 간선이 연결되어 있는 것을 그래프라고 하고  
그 그래프를 탐색(서치)하는 방법이 여러가지가 있는데  
전체 순회를 할 때  
우리는 깊이를 우선으로 할 것인가 너비를 우선으로 할 것인가에 따라서  
DFS와 BFS로 나뉘는데  
여기서 BFS는 너비 우선 탐색이다.  
그리고 BFS를 구현할 때는 큐로 구현하는 것이 일반적이다.  
왜냐하면 BFS는 발견한 순서대로 방문하는 것이기 때문에 큐를 사용하는 것이 합리적이고  
내가 발견한 순서대로 방문을 하기 때문에 길찾기와 밀접하게 관련 있는 것을 확인 할 수 있다.  
즉, 내가 목적지를 발견했다고 한다면 그 경로를 추적해서 어떻게 왔는지 알 수 있을 것이고 그게 바로 최단거리 일 것이다.

다만 단점으로는 애가 목적지라는 개념이 없이 아무곳이나 스캔하다 보니 필요없는 곳을 스캔하는 것이 마음에 안들 수가 있다.  
그리고 가중치라는 개념이 없어서 어디길이 더 좋은지 판별을 할 수가 없었다.

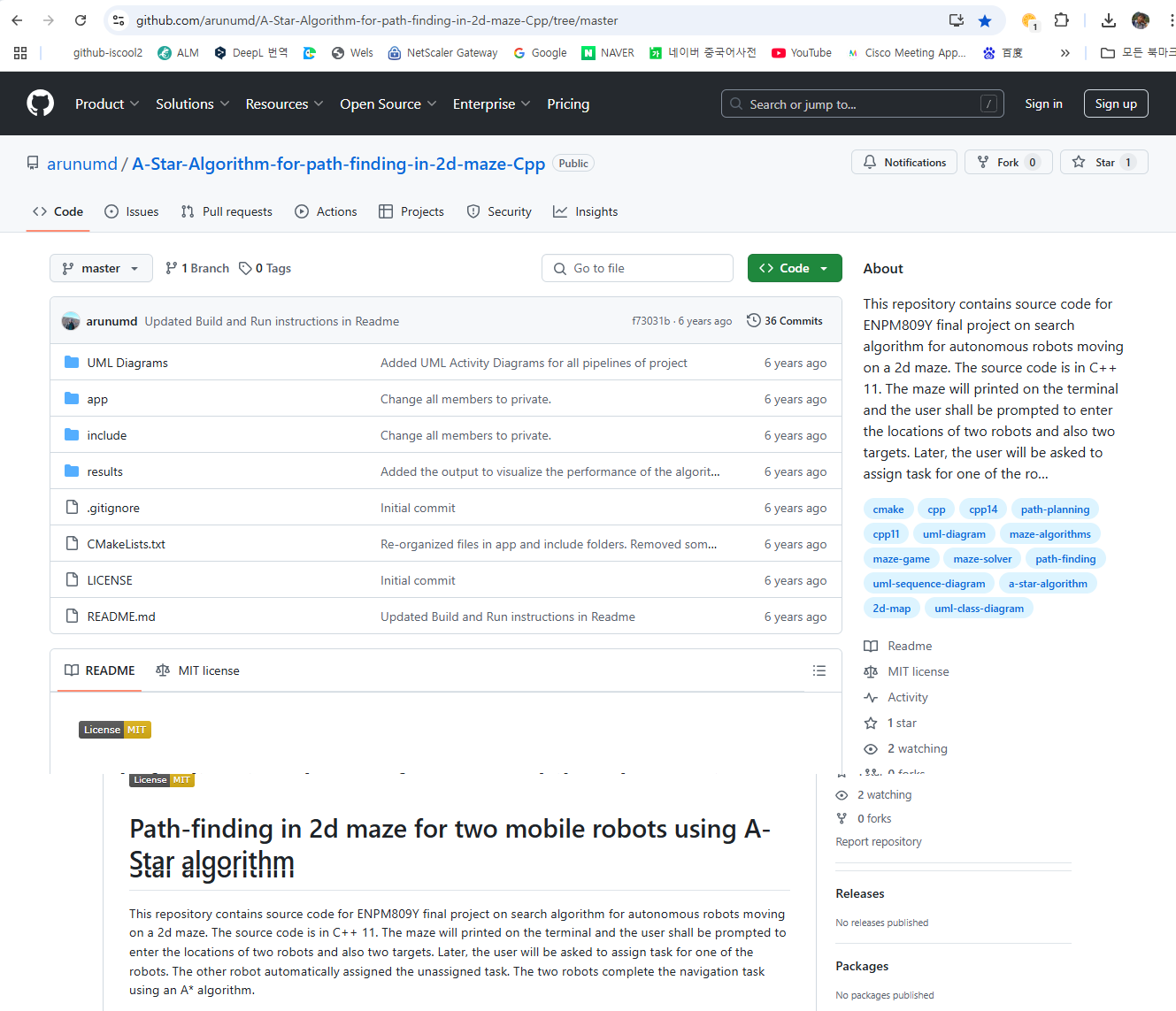
여기서 BFS에 가중치라는 개념을 집어넣은 것이 다익스트라이다.  
다익스트라는 가중치라는 개념을 가지고 가산점을 줘서 나중에 라도 더 좋은 길을 선택할 수 있도록 유도를 해줄 수 있고  
가중치가 생김으로써 게임에서 온갖 상황을 다 묘사할 수 있게된다.

하지만 다익스트라도 목적지라는 개념을 두고 특정 방향으로 가는것이 아니기 때문에  
A\*로 업그레이드가 된다.

A\*같은 경우는 현재 내 좌표를 기준으로 목적지를 바라보고 가려는 시도를 하기 때문에 BFS나 다익스트라보다는 확률적으로 좀 더 옳은 방향으로 갈 확율이 높을 것이다.

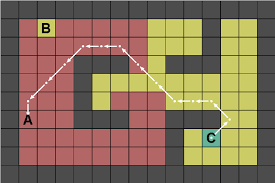
왜냐하면 A\*는 목적지 기준으로 서칭을 하기위해 노력할 것이기 때문에 결과는 비슷하게 보여도 불필요한 움직임이 덜 할 것이다.

# Test Project 1





# Test Project 2



개요및 소개

🔥개요

**리그오브레전드**같은 탑다운게임에서는 플레이어가 마우스를 클릭한 곳으로 캐릭터가 이동합니다.본 포스팅에서는 2차원배열내에서 출발점에서 도착점까지 최단경로로 이동하는 \*알고리즘을 구현하겠습니다.***이 글은 어디까지나 저의 개인적인 노트, 정리같은것이기 때문에 이론들을 깊게 설명하는것은 하지 않겠습니다.***

🔥개발환경

**2021-08-08**기준 **Windows10 Home** 사용했으며, 컴파일러는 **GCC**를 사용했습니다.  
**c++버전은 17**사용했습니다.***일반적으로 소스코드를 복붙하는것보다는 소스코드에대한 전반적인 이해를 하시는것을 추천드립니다.***

✨A\*알고리즘

🔥정의

A\*알고리즘은 어떠한 2차원배열에서 출발점과 도착점이 주어졌을때, 최단경로를 탐색하는 알고리즘입니다.

🔥소스코드

" MAP1.txt "

4 8

1 0 0 1 0 0 0 0

0 1 0 1 0 1 0 0

0 1 0 0 0 1 0 0

2 0 0 0 0 1 0 3

" MAP2.txt " ( 위에 썸네일의 맵과 동일합니다. )

10 15

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 0 0 0 1 0 0 1

1 0 0 0 0 0 0 0 0 0 0 1 0 0 1

1 0 0 0 1 1 1 0 0 1 1 1 0 0 1

1 0 0 0 1 0 0 0 0 0 0 0 0 0 1

1 0 0 0 1 0 0 0 0 0 0 0 0 0 1

1 2 0 0 1 1 1 0 0 1 1 1 0 0 1

1 0 0 0 0 0 0 0 0 1 0 3 0 0 1

1 0 0 0 0 0 0 0 0 1 0 0 0 0 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

C++17

#include <bits/stdc++.h>

constexpr int MAX = 101;

constexpr double INF = 1e9+7;

// 직선

const int dx1[4] = { 0, 0, 1, -1 };

const int dy1[4] = { -1, 1, 0, 0 };

// 대각선

const int dx2[4] = { 1, -1, -1, 1 };

const int dy2[4] = { -1, 1, -1, 1 };

using Pair = std::pair<int, int>;

using pPair = std::pair<double, Pair>;

struct Cell {

int parent\_x, parent\_y;

double f, g, h;

};

char zmap[MAX][MAX];

int ROW = 0, COL = 0;

bool isDestination(int row, int col, Pair dst) {

if (row == dst.first && col == dst.second) return true;

return false;

}

bool isInRange(int row, int col) {

return (row >= 0 && row < ROW && col >= 0 && col < COL);

}

bool isUnBlocked(std::vector<std::vector<int>>& map, int row, int col) {

return (map[row][col] == 0);

}

double GethValue(int row, int col, Pair dst) {

return (double)std::sqrt(std::pow(row - dst.first, 2) + std::pow(col - dst.second, 2));

}

void tracePath(Cell cellDetails[MAX][MAX], Pair dst) {

std::stack<Pair> s;

int y = dst.first;

int x = dst.second;

s.push({ y, x });

// cellDetails의 x, y의 부모좌표가 모두 현재좌표와 동일할때까지 반복

while (!(cellDetails[y][x].parent\_x == x && cellDetails[y][x].parent\_y == y)) {

int tempy = cellDetails[y][x].parent\_y;

int tempx = cellDetails[y][x].parent\_x;

y = tempy;

x = tempx;

s.push({ y, x });

}

while (!s.empty()) {

zmap[s.top().first][s.top().second] = '\*';

s.pop();

}

}

bool aStarSearch(std::vector<std::vector<int>>& map, Pair src, Pair dst) {

if (!isInRange(src.first, src.second) || !isInRange(dst.first, dst.second)) return false;

if (!isUnBlocked(map, src.first, src.second) || !isUnBlocked(map, dst.first, dst.second)) return false;

if (isDestination(src.first, src.second, dst)) return false;

bool closedList[MAX][MAX];

std::memset(closedList, false, sizeof(closedList));

Cell cellDetails[MAX][MAX];

// 내용초기화

// 이 구조 많이 보셨을겁니다. (최대유량알고리즘과 유사)

// 계산해야할 값부분은 INF로하고, 계산할 경로는 -1로 초기화

for (int i = 0; i < ROW; ++i) {

for (int j = 0; j < COL; ++j) {

cellDetails[i][j].f = cellDetails[i][j].g = cellDetails[i][j].h = INF;

cellDetails[i][j].parent\_x = cellDetails[i][j].parent\_y = -1;

}

}

// src의 좌표가 첫좌표가 된다.

int sy = src.first;

int sx = src.second;

cellDetails[sy][sx].f = cellDetails[sy][sx].g = cellDetails[sy][sx].h = 0.0;

cellDetails[sy][sx].parent\_x = sx;

cellDetails[sy][sx].parent\_y = sy;

std::set<pPair> openList;

openList.insert({ 0.0, { sy, sx } });

// 이 반복구조 bfs와 엄청 똑같습니다.

while (!openList.empty()) {

pPair p = \*openList.begin();

openList.erase(openList.begin());

int y = p.second.first;

int x = p.second.second;

closedList[y][x] = true;

double ng, nf, nh;

// 직선

for (int i = 0; i < 4; ++i) {

int ny = y + dy1[i];

int nx = x + dx1[i];

if (isInRange(ny, nx)) {

if (isDestination(ny, nx, dst)) {

cellDetails[ny][nx].parent\_y = y;

cellDetails[ny][nx].parent\_x = x;

tracePath(cellDetails, dst);

return true;

}

// bfs와 굳이 비교하자면, closedList를 방문여부라고 생각하시면 됩니다.

else if (!closedList[ny][nx] && isUnBlocked(map, ny, nx)) {

// 이부분 y x, ny nx 헷갈리는거 조심

ng = cellDetails[y][x].g + 1.0;

nh = GethValue(ny, nx, dst);

nf = ng + nh;

// 만약 한번도 갱신이 안된f거나, 새로갱신될 f가 기존f보다 작을시 참

if (cellDetails[ny][nx].f == INF || cellDetails[ny][nx].f > nf) {

cellDetails[ny][nx].f = nf;

cellDetails[ny][nx].g = ng;

cellDetails[ny][nx].h = nh;

cellDetails[ny][nx].parent\_x = x;

cellDetails[ny][nx].parent\_y = y;

openList.insert({ nf, { ny, nx } });

}

}

}

}

// 대각선

for (int i = 0; i < 4; ++i) {

int ny = y + dy2[i];

int nx = x + dx2[i];

if (isInRange(ny, nx)) {

if (isDestination(ny, nx, dst)) {

cellDetails[ny][nx].parent\_y = y;

cellDetails[ny][nx].parent\_x = x;

tracePath(cellDetails, dst);

return true;

}

else if (!closedList[ny][nx] && isUnBlocked(map, ny, nx)) {

ng = cellDetails[y][x].g + 1.414;

nh = GethValue(ny, nx, dst);

nf = ng + nh;

if (cellDetails[ny][nx].f == INF || cellDetails[ny][nx].f > nf) {

cellDetails[ny][nx].f = nf;

cellDetails[ny][nx].g = ng;

cellDetails[ny][nx].h = nh;

cellDetails[ny][nx].parent\_x = x;

cellDetails[ny][nx].parent\_y = y;

openList.insert({ nf, { ny, nx } });

}

}

}

}

}

return false;

}

void PrintMap() {

for (int i = 0; i < ROW; ++i) {

for (int j = 0; j < COL; ++j) {

std::cout << zmap[i][j];

}

std::cout << '\n';

}

}

std::vector<std::vector<int>> fileload(std::string filepath) {

std::ifstream ifs(filepath);

int col, row, cur = 0;

if (ifs.is\_open()) {

ifs >> ROW >> COL;

std::vector<std::vector<int>> result(ROW, std::vector<int>(COL));

for (int i = 0; i < ROW; ++i) {

for (int j = 0; j < COL; ++j) {

ifs >> result[i][j];

}

}

return result;

}

return std::vector<std::vector<int>>();

}

int main() { // 0: 빈 공간, 1: 벽, 2: 출발지점, 3: 도착지점

Pair src, dst;

int row, col;

/// 방법1 - 맵정보 직접 입력하기

/\*

std::cin >> row >> col;

ROW = row;

COL = col;

std::vector<std::vector<int>> grid(row, std::vector<int>(col));

for(int i=0;i<row;++i){

for(int j=0;j<col;++j){

std::cin >> grid[i][j];

}

}\*/

/// 방법2 - 파일로 부터 맵정보 불러오기

std::vector<std::vector<int>> grid = fileload("W:\\MAP.txt");

if (grid.empty()) return -1;

for (int i = 0; i < ROW; ++i) {

for (int j = 0; j < COL; ++j) {

if (grid[i][j] == 2) {

src = { i, j };

grid[i][j] = 0;

}

if (grid[i][j] == 3) {

dst = { i, j };

grid[i][j] = 0;

}

}

}

for (int i = 0; i < ROW; ++i) {

for (int j = 0; j < COL; ++j) {

zmap[i][j] = grid[i][j] + '0';

}

}

if (aStarSearch(grid, src, dst)) PrintMap();

else std::cout << "실패.";

return 0;

}

🔥소스코드 설명

💎함수 - isDestination

현재좌표가 도착지점과 일치하다면 참, 아니면 거짓을 반환하는 함수.

💎함수 - isInRange

현재좌표가 전체 맵안에 존재하면 , 아니면 거짓을 반환하는 함수.

💎함수 - isUnBlocked

현재좌표가 벽이아니라면 참, 아니면 거짓을 반환하는 함수.

💎함수 - GethValue

현재좌표로부터 도착지점까지의 거리를 계산하는 함수.

💎함수 - tracePath

backtracking을 이용하여 최단경로를 탐색하는 함수.

💎함수 - aStarSearch

a\*알고리즘을 실행하는 함수.

💎함수 - PrintMap

현재 맵의 상태를 출력하는 함수.

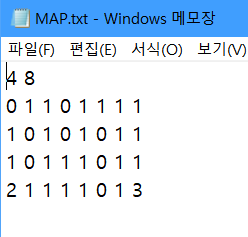
💎함수 - fileload

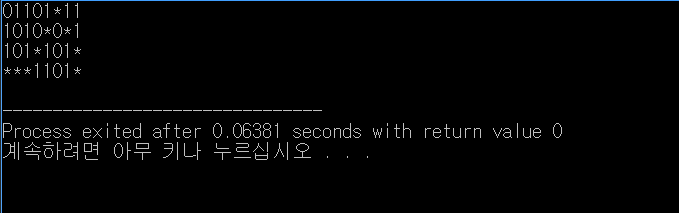
텍스트파일로부터 맵정보를 불러오는 함수.

🔥실행

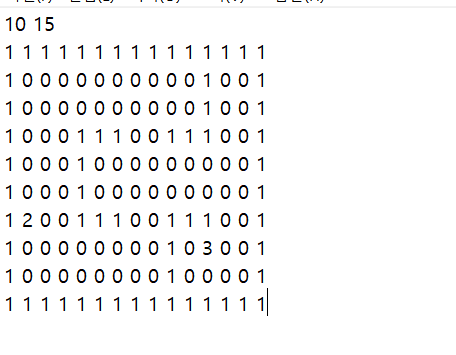
**(0: 빈 공간, 1: 벽, 2: 출발지점, 3: 도착지점)**

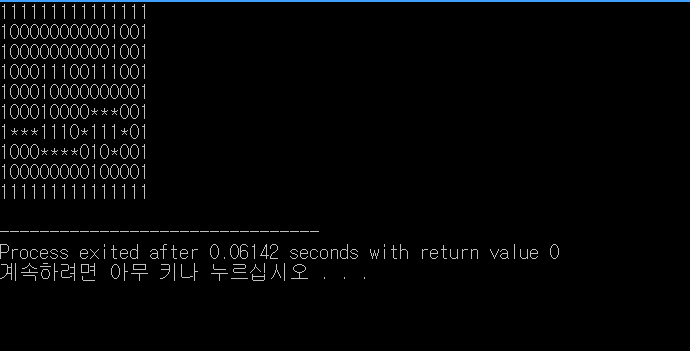
" MAP1.txt "

(밑에 사진 0과 1위치가 반전됬습니다. 참고바랍니다.)  




" MAP2.txt "  
MAP2는 썸네일의 맵입니다.  
실제 출력해보니까, 저의 코드에서는 썸네일의 경로와는 다른 최단경로를 찾아냈습니다.





**와우~!!! 아주 정상적으로 잘됩니다!!**

🔥다운로드

[깃허브 다운로드](https://github.com/Mawi1e/AstarImpl)