A* 알고리즘에 대해서

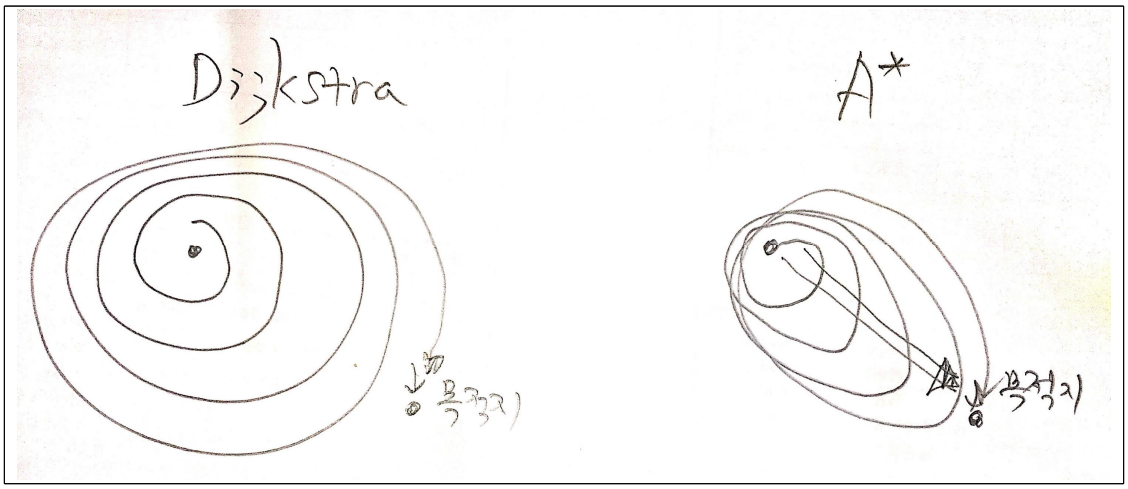
A* 알고리즘이란 게 있다. 우리말로 "에이 스타 알고리즘"이라고 읽는다. 이것은 Dijkstra 알고리즘의 성능을 개선한 것이다. Dijkstra 알고리즘은 잘 알다시피 그래프 상에서 최단경로를 찾는 알고리즘으로 유명하다. 1959년에 Dijkstra 알고리즘이 나왔고, 9년 뒤인 1968년에 A* Dijkstra 알고리즘이 개발되었다.

오해하지 말아야 할 것이 있다. A*가 더 좋다고 해서 Dijkstra보다 더 나은 경로를 찾는다는 뜻은 아니다. 두 알고리즘 모두 찾게 되는 경로는 똑 같다. 그러면 A*가 좋다는 것은 무슨 뜻인가? Dijkstra보다 더 빨리 찾을 수 있다는 뜻이다. 소프트웨어는 정확한 결과를 내는 것만큼이나 빠른 시간 내에 결과를 내는 것도 동작하는 것도 중요하다.

A*는 어떻게 Dijkstra보다 더 빨리 최단경로를 찾을 수 있을까? 간단히 얘기하자면 최단경로를 찾기 위한 방향을 지정할 수 있어서 Dijkstra보다 더 빨리 경로를 찾을 수 있다. 무슨 얘기인가? 예를 들어, 서울에서 부산까지 가는 최단경로를 찾는다고 하자. 부산은 서울의 남동쪽에 있기 때문에 수원이나 대전 쪽 방향의 경로를 먼저 탐색하는 것이 상식적으로 맞다. 서울의 서쪽에 있는 인천에서 부산가는 최단 경로를 찾느라 시간 낭비를 할 필요는 없는 것이다. Dijkstra는 순진무구해서 인천에서 시간을 낭비하는 반면, 영악한 A*는 과감히 인천을 포기하고, 그 시간에 가능성이 더 높은 곳을 먼저 탐색한다. 그렇다고 해서 인천을 완전히 제외시키는 것은 아니다. 다만 우선순위에서 밀리는 것뿐이다. 수원이나 대전에서 부산가는 경로를 못 찾으면, 결국 인천을 뒤늦게 찾게 된다.

그렇다면 이런 의문이 생길 수 있다. 만약 수원, 대전을 경유해서 부산가는 길이 없다고 하자. 그러면 결국 인천으로 돌아서 가야한다. 이 경우에는 잔머리 굴렸던 A*보다 우직했던 Dijkstra가 더 빨리 경로를 찾을 수 있는 것 아닌가? 맞다. A*가 Dijkstra보다 더 시간이 오래 걸릴 수도 있다. 이건 사실이다. 하지만 그럴 확률은 매우 낮기 때문에, 일반적으로 A*는 Dijkstra보다 성능이 좋은 것으로 알려져 있다.

Dijkstra와 A* 간의 최단경로 탐색 방법에 대해 좀 더 자세히 설명해 보자. Dijkstra는 현재 위치로부터 모든 방향을 대상으로 최단 경로를 탐색한다. 아래 그림의 왼쪽은 Dijkstra의 이러한 동작방식을 보여준다. 가운데 점이 출발지이고 목적지는 오른쪽 하단에 있다. 출발지를 중심으로 원형을 그려가면서 최단 경로를 찾아나간다. 반면 A*는 특정방향을 먼저 시도한다. 아래 오른쪽 그림과 같이 목적지 방향을 우선적으로 탐색한다. 따라서 가능성이 없는 곳을 탐색하는 시간을 줄일 수 있다. 이 경우 가능성이 없는 곳은 출발지의 왼쪽 윗방향이다. 다시 한 번 강조하지만 결과적으로 찾게 되는 최단경로는 두 알고리즘이 동일하다. 다만 소요시간에서 차이가 날 뿐이다.



이제 A* 알고리즘을 구현해 보도록 하자. 이것의 구현은 Dijkstra와 거의 동일하므로 먼저 Dijkstra 알고리즘의 pseudo code를 다시 한 번 생각해보자. 이를 위해 출발지가 있고, 4개의 vertex B, C, D, E가 있다고 가정하자. Dijkstra에서는 출발지가 B, C, D, E에 대해 아래와 같은 정보를 기록한다.

장소	B	C	D	E
종료				
최단거리				
직전노드				

위의 표에서 '최단거리'는 출발지에서 각 장소까지의 최단경로 거리를 기록한다. '직전노드'는 그 장소에 도달하기 직전에 거쳐야 하는 곳을 말한다. 출발지와 직접적으로 연결된 장소들은 직전노드에 출발지의 이름이 기입된다. '종료'는 최단경로 결정 여부를 기록하는 플래그이다. 이 플래그가 found이면 해당 장소까지의 최단경로가 결정된 것이다.

1. 각 장소에 해당하는 값을 초기화. 종료는 not-found, 최단거리는 무한대, 직전노드는 없음으로 한다.
2. 출발지에서 직접적으로 연결된 장소들에 대해서는 최단거리를 기입하고, 직전노드에 출발지를 기입한다.
3. 최단경로가 not-found이면서 최단거리가 가장 짧은 장소를 선택하여 새경로노드로 설정하고, 그 장소의 종료를 found로 바꾼다.
4. 최단경로가 not-found인 모든 장소들에 대해서, 아래 조건이 만족하는지 판단한다.
if (장소의 '최단거리' > 새경로노드의 '최단거리' + 새경로노드에서 그 장소까지 거리) 이면,
장소까지의 새로운 경로를 찾은 것이다. 그 장소에 대해서
4.1 '최단거리' = 새경로노드의 '최단거리' + 새경로노드에서 그 장소까지 거리
4.2 직전노드 = 새경로노드
5. 목적지 장소의 종료가 not-found라면 3단계로 복귀한다.

위는 Dijkstra알고리즘의 pseudo code이다. 최단경로를 찾은 장소가 생길 때마다, 아직까지 최단경로가 결정되지 않은 장소들에 대해 더 빠른 길이 없는가를 반복적으로 탐색한다. 즉, 아래 수식이 이러한 역할을 한다.

if (장소의 '최단거리' > 새경로노드의 '최단거리' + 새경로노드에서 그 장소까지 거리)

이제 A* 알고리즘를 구현해 보자. Dijkstra 알고리즘에서 사용되었던 표에 아래와 같이 한 가지 정보가 더 기록된다. 각 장소마다 '가중최단거리'가 추가된다. Dijkstra알고리즘의 3단계에서 '최단거리'가 최소인 장소를 찾았다면, A* 알고리즘에서는 '가중최단거리'가 최소인 장소를 찾는다. '가중최단거리'는 모든 장소를 고려하는 대신에 특정 방향에 있는 장소를 먼저 고려하도록 하는 역할을 한다. 이에 대해서는 뒤에서 더 자세히 설명한다.

장소	B	C	D	E
종료				
최단거리				
직전노드				
가중최단거리				

A* 알고리즘의 pseudo code는 다음과 같다.

1. 각 장소에 해당하는 값을 초기화. 종료는 not-found, 최단거리는 무한대, 직전노드는 없음으로 한다.
2. 출발지에서 직접적으로 연결된 장소들에 대해서는 최단거리를 기입하고, 직전노드에 출발지를 기입한다.
3. 각 장소의 가중최단거리를 f(장소)로 계산해서 넣고, 최단경로가 not-found이면서 가

중최단거리가 가장 짧은 장소를 선택하여 새경로노드로 설정하고, 그 장소의 종료를 found로 바꾼다.
4. 최단경로가 not-found인 모든 장소들에 대해서, 아래 조건이 만족하는지 판단한다.
if (장소의 '최단거리' > 새경로노드의 '최단거리' + 새경로노드에서 그 장소까지 거리)이면 장소까지의 새로운 경로를 찾은 것이다. 그 장소에 대해서
4.1 '최단거리' = 새경로노드의 '최단거리' + 새경로노드에서 그 장소
4.2 직전노드 = 새경로노드
5. 목적지 장소의 종료가 not-found라면 3단계로 복귀한다.

A* 알고리즘은 가중최단거리 = $f(\text{장소})$가 핵심이다. 여기에는 탐색방향성이 들어간다. 탐색방향성은 여러 가지로 할 수 있고, 딱히 정해진 것은 없다. 여기서는 가장 쉬운 걸로 한 번 해보자. 예를 들어 아래와 같이 현재 장소에서 목적지까지의 거리를 계산해서 넣을 수도 있다. 즉 Dijkstra와 같이 최단경로에다가, 새로운 가중치를 더하는 것이다.

$$f(\text{장소}) = '\text{장소}'의\ '\text{최단거리}' + \lambda(\text{장소에서 목적지까지 거리})$$

여기에서 λ는 임의로 설정가능한 상수이다. 거리는 Euclidean 거리로 할 수도 있지만, 거리에 비례하는 어느 것으로 하든 상관없다. $f(\text{장소})$를 사용하기 때문에 같은 최단경로를 가진 장소가 여러 개 있다면, 목적지 쪽에 더 가까운 장소를 먼저 선택하게 하는 효과를 가진다. 즉, 목적지가 현재 위치에서 남동쪽에 있다면, 굳이 북서쪽에 있는 장소를 먼저 탐색하기 보다는 그 쪽 방향에 있는 장소를 먼저 탐색하게 된다.

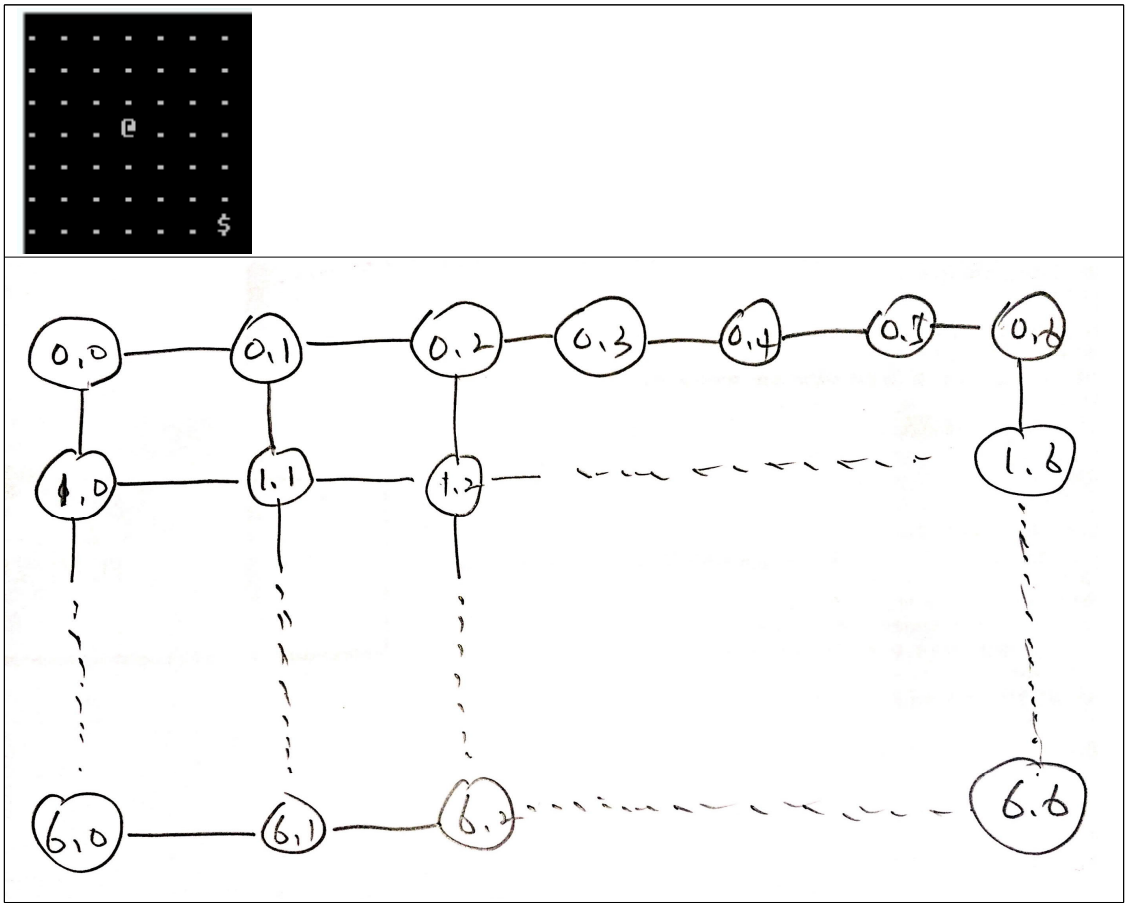
만약 $f(\text{장소})$를 아래와 같이 설정하면 Dijkstra 알고리즘과 동일하다. 따라서 A* 알고리즘은 보다 일반화된 (generalized) 최단경로 찾기 알고리즘인 것이다.

$$f(\text{장소}) = '\text{장소}'의\ '\text{최단거리}'$$

A*를 직관적으로 이해하자면 다음과 같다. Dijkstra는 최단거리를 가진 장소를 계속적으로 선택해가면서 진행한다. A*는 최단거리이면서, 목적지가 있는 방향에 쪽의 장소를 선택한다. 그래서 Dijkstra보다 빠른 것이다.

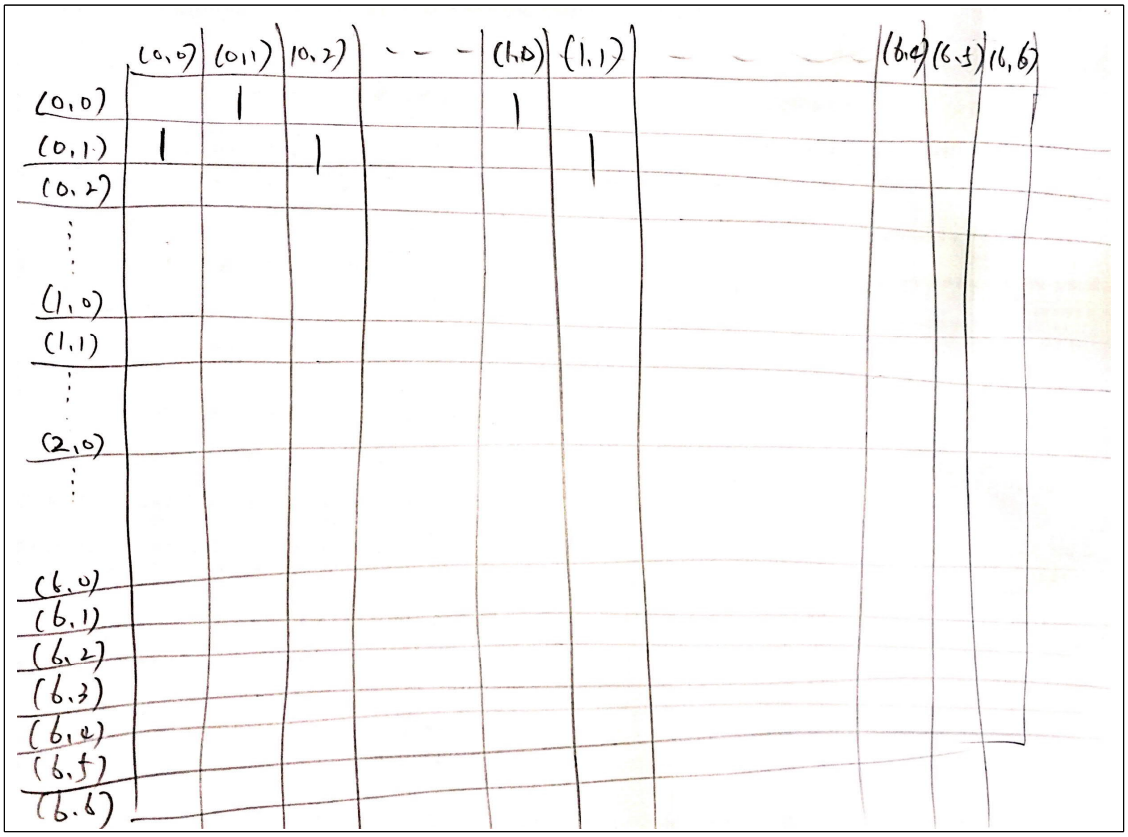
이제 실제로 C 코드로 구현해 보자. 아래와 같은 2차원 지도에서 최단경로를 찾는 것을 목표로 한다. 지도크기는 7x7이고, '.'으로 표시된 곳이 장소들

을 나타낸다. 특히 기호 '@'으로 표시된 곳이 출발장소이고, '$'는 도착지를 나타낸다. 장소에서는 상,하,좌,우에 있는 이웃장소로 이동이 가능하고, 대각선 방향으로의 직접이동은 안 된다.



2차원 지도를 그래프로 나타내면 위와 같다. Vertex안의 (행,열)은 지도에서 좌표를 나타내고, edge로 이어진 vertex들은 상,하,좌,우 이웃 장소들이다. 이 그래프는 undirected graph이므로 이웃끼리 양방향 이동이 가능하다.

이제 위 그래프를 프로그래밍 언어로 표현해보자. 그래프를 구현하는 방법은 두 가지가 있다. 하나는 2차원 배열을 이용하는 것이고, 다른 하나는 singly linked list를 사용하는 것이다. 여기서는 구현이 직관적인 2차원배열을 이용한다. 아래 그림은 위 그래프를 2차원 배열로 표현한 것이다.



vertex간의 연결관계는 49x49 2차원 배열로 표현할 수 있다. 가로와 세로는 각각 vertex에 해당한다. 두 vertex가 서로 인접하다면 해당하는 칸은 1이 기입하고, 그렇지 않은 경우에는 0이 된다. 예를 들어 vertex (0,0)은 vertex (0,1)과 (1,0)과 연결되어 있으므로 해당하는 칸에 1이 기입된다.

여기부터는 A* 알고리즘을 구현한 코드를 순차적으로 설명한다.

```
 7  #include <stdio.h>
 8  #include <limits.h>
 9  #include <cstdlib>
10  #include <Windows.h>
```

◦ 구현에 필요한 헤더파일들이다.

```
12  #define TRUE                1
13  #define FALSE               0
14  #define DELAY_MILISEC       1000
15  #define ASTAR
16  #define GRID_LEN            7
17  #define SZ                  GRID_LEN*GRID_LEN
```

15라인: 이 프로그램은 A*는 물론 Dijkstra 알고리즘도 동시에 구현한 것이다. #define

ASTAR를 comment-out하면 Dijkstra로 동작한다.
16라인: 2차원지도의 x축과 y축 길이
17라인: 그래프를 2차원 배열로 구현할 때 필요한 각 차원의 크기

```
19  int graph[SZ][SZ];
20  int startLocSeq;
21  int destLocSeq;
22  char map[GRID_LEN][GRID_LEN];
```

19라인: 지도에 해당하는 그래프를 2차원 배열로 표현한 것
20라인: 지도상에서 출발지 좌표. 2차원 좌표에 대한 1차원 일련번호 (뒤에서 설명)
21라인: 지도상에서 목적지 좌표 2차원 좌표에 대한 1차원 일련번호 (뒤에서 설명)
22라인: 지도를 화면에 표시하기 위한 2차원 배열

```
24  struct astar_info
25  {
26      int found;
27      int shortestDist;
28      int previousLoc;
29  };
30  struct astar_info astar_table[SZ];
```

24~29라인: A* 알고리즘 수행하면서 최단경로 등을 저장하기 위한 구조체.
◦ found: 해당 노드까지의 최단경로를 찾았는지를 표시. 찾았을 경우에는 1, 아닌 경우에는 1로 표시한다.
◦ shortestDist: 출발지 노드에서 해당 노드까지 최단거리. 이 최단거리는 found==0인 동안 계속적으로 업데이트된다.
◦ previousLoc: 출발지에서 해당노드에 이르는 경로에서, 해당노드에 이르기 직전의 노드. 노드의 일련번호로 저장된다. 이 값 역시 found==0인 동안 계속적으로 업데이트된다.
30라인: A* 알고리즘 수행하면서 최단경로 등을 저장하기 위한 구조체 배열
◦ 배열의 크기는 SZ이다. 이는 2차원 지도 상의 모든 위치들에 대한 최단경로관련 정보를 저장하기 위해서 이다.

2차원 지도에 해당하는 그래프를 표현하는 배열변수 graph에 vertex간의 edge정보를 입력하기 위해서 함수 init_graph( )를 이용한다.

```
void init_graph()
{
    int i, j;
    for (i = 0; i < SZ; i++)
    {
        for (j = 0; j < SZ; j++)
        {
            graph[i][j] = isNeighbor(i, j);
        }
    }
}
```

이 함수는 vertex i와 vertex j가 서로 이웃했으면 graph[i][j]에 1을 입력하고, 그렇지 아닌 경우는 0을 할당한다. 사용된 함수 isNeighbor(int x, int y)는 이웃 인접관계를 판단하여 위치 x와 위치 y가 이웃한 경우 1을 반환, 그렇지 않은 경우에는 0을 반환하는 함수이다. 이 함수는 인수로 x와 y를 받는데, 이것은 장소들에 0부터 일련번호를 매긴 값이다. 예를 들어 일련번호 0은 (0,0)에 해당하고, 7은 (0,6) 그리고 48은 (6,6)이 된다. 이렇게 2차원 벡터로 변환된 좌표를 이용하면, vertex가 서로 이웃한 것들인지 쉽게 판별이 가능하다. 구현된 소스코드는 아래와 같다.

```
int isNeighbor(int x, int y)
{
    int x1 = x / GRID_LEN;
    int x2 = x % GRID_LEN;

    int y1 = y / GRID_LEN;
    int y2 = y % GRID_LEN;

    // y가 x의 위에 있는지 확인
    if (x2 == y2 && (y1-1)==x1)
    {
        return 1;
    }

    // y가 x의 아래에 있는지 확인
    if (x2 == y2 && (y1+1)==x1)
    {
        return 1;
    }

    // y가 x의 좌측에 있는지 확인
    if (x1 == y1 && (y2-1)==x2)
    {
        return 1;
    }

    // y가 x의 우측에 있는지 확인
    if (x1 == y1 && (y2+1)==x2)
    {
        return 1;
    }
    return 0;
}
```

알고리즘 내에서 위치들은 2차원 좌표가 아닌 1차원의 일련번호로 구별이 된다. 따라서 2차원에서 1차원으로 변환하기 위한 함수를 만들어서 사용하면 편리하다. 이를 위해 함수 calcSeqNum( )을 만든다.

```
/*
 *
 * (x,y)좌표를 받아서 일련번호를 계산
 *
 * (0,0) -> GRID_LEN*0+0 = 0
 * (6,6) -> GRID_LEN*6+6 = 48
 */
int calcSeqNum(int x, int y)
{
    return ( GRID_LEN*x + y );
}
```

◦ 인수 (x,y): 2차원 좌표로 1차원으로 변환되어야 한다.
◦ 반환값형 int: 1차원값으로 변환된 값
◦ 변환공식: x행에 열의 전체개수(GRID_LEN)을 곱하고, 현재 열의 값을 더한다. 주석에 나온 것처럼 2차원 (6,6)은 일련번호 48로 변환된다.

또한 출발지와 목적지의 일련번호를 전역변수로 저장한다 .이를 위해 다음과 같이 두 개의 변수를 선언하고 , 각각 출발지와 목적지를 저장한다 .

```
int startLocSeq;
int destLocSeq;
```

```
startLocSeq = calcSeqNum(3,3);
destLocSeq = calcSeqNum(GRID_LEN-1,GRID_LEN-1);
```

◦ startLocSeq: 출발지 위치의 일련번호를 저장하는 변수로 2차원 좌표 (3,3)으로 설정한다. (3,3)을 일련번호로 변환하기 위해 함수 calcSeqNum( )을 이용한다.
◦ detLocSeq: 목적지 위치에 대한 변수이다. 우하단 끝 위치로 설정한다.

이제 앞서 설명한 최단경로관련 정보를 저장하는 배열 astar_table을 초기화하는 함수 init_astar_table ( )을 설명해보자. astar_table는 출발지에서 다른 모든 위치들로 가는 최단경로에 관한 정보를 저장하기 위한 것이다.

```
61 void init_astar_table(void)
62 {
63     for (int i = 0; i < SZ; i++)
64     {
65         if (i == startLocSeq)
66         {
67             astar_table[i].found = TRUE;            //
68             continue;
69         }
70
71         astar_table[i].found = FALSE;               //
72
73         if (isNeighbor(startLocSeq, i) == TRUE)
74         {
75             astar_table[i].shortestDist = 1;        //
76             astar_table[i].previousLoc = startLocSeq;
77         }
78         else
79         {
80             astar_table[i].shortestDist = -999;     //
81             astar_table[i].previousLoc = -1;        //
82         }
83     }
84 }
```

63: 배열 astar_table의 모든 요소들에 대해 수행한다.
65~69: 출발지 자신에 대한 것은 처리할 필요가 없다. 따라서 최단경로를 이미 찾았다는 의미로 found=1로 설정하고, 나머지 과정은 생략하고 다음 위치로 넘어간다 .
71: 초기화 과정이므로 최단경로를 찾지 못했다고 found = FALSE로 설정한다 .
73~76: 출발지의 이웃위치들에 대해서는 한칸 이동으로 갈 수 있으므로 shortestDist=1로 설정한다. 이 경우 목적지에 다다르기 전의 직전 위치는 출발지이므로 previousLoc을 출발지로 설정한다. (previousLoc = startLocSeq)
78~81: 인접하지 않은 위치들에 대해서는 shortestDist와 previousLoc을 설정하지 않는다. 그러한 의미로 -999와 -1을 넣는다.

잠깐! 최단경로를 찾아가는 과정을 시각적으로 확인하면 알고리즘을 쉽게 이해할 수 있다. 지도를 화면에 표시하기 위해서는 다음과 같은 것들이 필요하다.

```
char map[GRID_LEN][GRID_LEN];
```

◦ 지도를 저장하기 위한 2차원배열이다. 지도상에 출발지, 목적지 등을 문자로 표시하기 위해 char형이다.

```
196 void init_map()
197 {
198     for (int i = 0; i < GRID_LEN; i++)
199     {
200         for (int j = 0; j < GRID_LEN; j++)
201         {
202             map[i][j] = '.';
203         }
204     }
205
206     map[startLocSeq/GRID_LEN][startLocSeq%GRID_LEN] = '@';
207     map[destLocSeq/GRID_LEN][destLocSeq%GRID_LEN] = '$';
208 }
```

196: 지도를 초기화하기 위한 함수이다.

198~204: 지도상의 모든 위치에 문자 “.”로 초기화한다.

206: 출발지는 문자 “@”으로 표시한다. 출발지의 2차원 좌표는 1차원 일련번호를 변환해서 사용한다.

207: 목적지는 문자 “$”로 표시한다.

```
210 void display_map()
211 {
212     for (int i = 0; i < GRID_LEN; i++)
213     {
214         for (int j = 0; j < GRID_LEN; j++)
215         {
216             printf("%c ", map[i][j]);
217         }
218         printf ("₩n");
219     }
220     printf("-----------------------------------₩n");
221 }
```

210: 지도를 화면 상에 출력하기 위한 함수 display_map()

212~219: 2차원 지도를 1개 문자 단위로 화면에 출력한다.

218: 행단위로 줄바꿈을 한다.

함수 findSmallestWeightedDist( )는 astar_table에서 가장 짧은 최단경로를 갖는 장소들을 찾아 반환한다. A* 알고리즘의 pseudo code 3단계의 역할을 한다. 특히 매크로 ASTAR가 정의되었으면, A*로 동작하고, 그렇지 않은 경우에는 Dijkstra로 동작한다.

```
84 /*
85  * 가중치가 고려된 최단경로를 선정한다.
86  */
87 int findSmallestWeightedDist(void)
88 {
89     int smallestIdx = -1;
90     int smallestDist = INT_MAX;
91
92     for (int i = 0; i < SZ; i++)
93     {
94         if (astar_table[i].found == FALSE && astar_table[i].shortestDist > 0)
95             // 아직까지 최단경로가 발견되지 않은 좌표들에 대해서만 계산한다.
96         {
97 #ifdef ASTAR
98             int weightedDist = astar_table[i].shortestDist + calcEuclideanDistance(destLocSeq, i)*10;
99 #else
100             int weightedDist = astar_table[i].shortestDist;
101 #endif
102             if (weightedDist < smallestDist)
103             {
104                 smallestIdx = i;
105                 smallestDist = weightedDist;
106             }
107         }
108     }
109     return smallestIdx;
110 }
```

89라인: 가장 짧은 거리를 가진 장소에 대한 일련번호를 저장한다.
90라인: 가장 짧은 거리를 저장한다.
92~108라인: 모든 장소들에 대해서 검사한다.
94라인: 최단경로가 발견되지 않았고, 경로가 존재하는 장소들에 대해서만 검사한다.
98라인: A* 알고리즘으로 동작할 때 장소에 대한 가중최단거리를 계산한다. 여기서는 목적지와의 거리가 가까울수록 가중최단거리가 작아지기 때문에, 목적지 방향의 장소가 선정될 가능성이 높아진다.
100라인: Dijkstra 알고리즘으로 동작할 때 장소에 대한 최단거리를 계산한다.
102~109라인: 가장 짧은 거리를 가진 장소의 일련번호를 반환한다.

위 함수 findSmallestWeightedDist( )에서 사용된 또 다른 함수 calcEuclidean Distance ( )는 두 장소 간의 거리를 계산하는 함수이다. 즉, 목적지와 얼마나 가까운가를 계산한다.

```
112 /*
113  *  a=(x1, y1)과 b=(x2, y2)간의 거리를 계산한다.
114  *  계산속도를 위해서 squared root를 계산하지 않는다.
115  *
116  */
117 int calcEuclideanDistance (int a, int b)
118 {
119     int x1 = a / GRID_LEN;
120     int y1 = a % GRID_LEN;
121     int x2 = b / GRID_LEN;
122     int y2 = b % GRID_LEN;
123     return ((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
124 }
```

◦ 2차원의 두 점 사이의 거리를 구하는 수학공식을 떠올리면 된다 .

이제 main함수이다.

```
223 int main (void)
224 {
225     startLocSeq = calcSeqNum(3,3);
226     destLocSeq = calcSeqNum(GRID_LEN-1,GRID_LEN-1);
227
228     init_map();
229     display_map();
230
231     Sleep(2000);
232
233     init_graph();
234     //display_graph();
235     init_astar_table();
```

225라인: 출발지 좌표를 (3,3)으로 설정한다.

226라인: 목적지 좌표를 (6,6)으로 설정한다.

231라인: 시작하기 전에 2초간 잠시 멈추고, 지도를 보여준다 .

233~235라인: 그래프를 2차원 배열로 구성하고, 각종 정보들을 초기화한다 .

다음은 A*알고리즘의 핵심적인 부분이다 .

```
237     int loc;
238     while ((loc = findSmallestWeightedDist()) != -1)
239     {
240         //
241         // 새로 최단경로를 찾은 위치에 대해서 update한다.
242         //
243         astar_table[loc].found = TRUE;
244
245         map[loc/GRID_LEN][loc%GRID_LEN] = 'X';
246
247         system("cls");              // 화면을 지운다
248         display_map();
249         Sleep(DELAY_MILISEC);       // delay milisec동안 잠시 정지
250
251         if (loc == destLocSeq)  // 목적지에 도달했으면 중지
252         {
253             break;
254         }
255
256         // update astar_table
257         for (int i = 0; i < SZ; i++)
258         {
259             // 최단경로가 알려지지 않은 위치들에 대해서,
260             // 그리고 새로 발견된 경로와 연결되는 위치들에 대해서
261             // 더 빠르게 갈 수 있는 경로를 확인해 본다.
262             if (astar_table[i].found == FALSE && graph[loc][i] == 1)
263             {
264                 // 새로 찾은 경로를 거쳐가는 것이 빠르거나
265                 // 아직까지 경로가 알려져 있지 않은 경우에
266                 if (astar_table[loc].shortestDist + 1 < astar_table[i].shortestDist || astar_table[i].shortestDist < 0)
267                 {
268                     astar_table[i].shortestDist = astar_table[loc].shortestDist + 1;
269                     astar_table[i].previousLoc = loc;
270                 }
271             }
272         }
273     }
```

238라인: 함수 findSmallestWeightedDist( )는 모든 장소들의 최단경로가 결정되면 -1을 반환하므로, 그 때까지 반복해서 실행한다. 그렇지 않은 경우 변수 loc은 최단경로가 결정된 장소의 일련번호를 저장한다.

243라인: 변수 loc에 해당하는 장소에 대해 최단경로가 결정되었음을 표시

245라인: 화면에 표시할 지도에 최단경로가 결정된 곳의 위치를 'X'로 표시

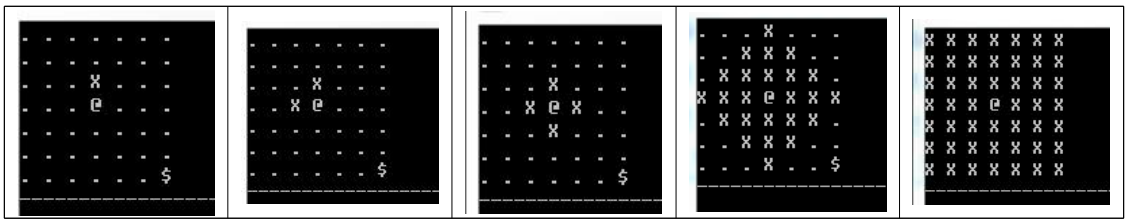
247~249라인: 알고리즘의 진행을 지도상에 표시하여 보여주기 위해, 화면을 지우고, 현재 업데이트된 지도를 다시 그리고, DELAY_MILISEC만큰 중지하여, 결과가 너무 빨리 지워지지 않도록 한다.

251~254: loc이 최종목적지이면 중지한다.

257 ~ 262라인: 최단경로 정보를 가진 배열의 모든 요소들에 대해서, 아직까지 최단경로가 결정되지 않았고, loc과 이웃한 장소들에 대해서 ...

266~269라인: loc을 통해서 가는 경로가 더 빠른 경로이거나, 기존에 경로가 없었다면 loc을 통해 가는 경로로 업데이트. 이 부분은 Dijkstra알고리즘과 동일하다.

이제 실행결과를 살펴보자. 우선 Dijkstra 알고리즘이 실행되는 과정을 보자. 소스코드에서 #define ASTAR를 comment-out하면, Dijkstra로 동작한다. 아래 그림과 같이, 출발지를 중심으로 'X'표시가 동심원을 그리며 퍼져나가는 것을 볼 수 있다. 'X'표시는 최단경로가 결정된 장소를 의미한다. 따라서 Dijkstra알고리즘은 출발지에서 가까운 곳부터 차례대로 경로를 검색하는 것을 알 수 있다.



A*의 동작은 아래와 같다. 출발지에서 목적지 방향으로의 장소들만 검색해 나가는 것을 알 수 있다. 특히 목적지가 발견되었을 때 ,아직까지 검색하지 않은 장소들이 많음을 알 수 있다. 이러한 수행과정상의 차이를 통해 A*알고리즘이 Dijkstra보다 효율적으로 최단경로를 검색하기 때문에 ,성능이 더 좋다는 것을 알 수 있다.