`

**자료구조**

미로탐색

2018-1

11조

201411802 응용통계학과 전민규

201310672 컴퓨터공학과 한석호

201712845 류한길

201611748이예은

목차

1. 개요
   1. 주제
   2. 구현 방향
   3. 구현 목표
2. 구현 상세
   1. 미로 입력 방법 – 한석호

- 트리

- 그래프

* 1. DFS알고리즘을 사용한 미로탐색 방법 – 이예은

- DFS란?

- 코드 설명

* 1. BFS알고리즘을 사용한 미로탐색 방법 – 류한길

- BFS란?

- 코드 설명

* 1. Dijkstra 알고리즘을 사용한 미로탐색 방법 – 전민규

- Dijkstra란?

- 코드 설명

1. 결과 및 동작
   1. Main함수
   2. 실행 화면
2. 후기 및 기록
   1. 공부한 부분
   2. 아쉬운 점
   3. 회의 기록

**미로탐색 알고리즘**

1. 개요
   1. 주제

트리와 그래프 자료구조를 기반으로 하는 다양한 알고리즘을 사용해 미로를 탐색하는 방법을 구현

* 1. 구현 방향

초기 구현은 트리 자료구조를 기반으로 하는 미로 탐색 알고리즘의 구현이었고, 트리에 대한 이해가 완료된 후, 이를 더욱 확장하여 그래프로 표현된 미로에 대한 알고리즘까지 공부하고 구현해보기로 하였음

* 1. 구현 목표

트리 자료 구조에 대한 이해와 구현

트리를 활용한 알고리즘의 이해와 구현

트리의 개념을 그래프까지 확장하여 미로 찾기 알고리즘을 적용

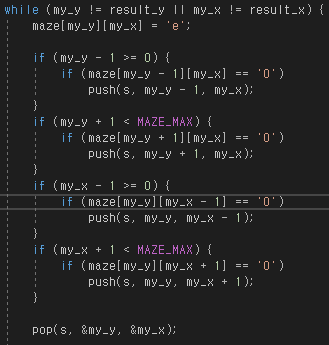
1. 구현 목표
   1. 미로 입력 방법

* 트리

트리는 사이클이 존재하지 않는 자료구조이다. 그렇기 때문에 미로 내에서 사이클이 존재할 경우 트리 자료구조를 활용할 수 없다는 단점이 있다.

미로 찾기에서 가장 중요한 것은 갈림길에서 어느 방향으로 갈 것인지 선택하는 것이라고 생각했고, 각 갈림길을 노드로 하여 트리를 만들 수 있다고 판단했다. 미로를 읽어 트리 자료구조에 담아낼 때에는 스택 자료구조를 함께 활용했다.

미로의 출발점부터 갈 수 있는 길을 스택에 담아 도착지까지 도달하는 간단한 미로 찾기 알고리즘이 있는데, 이를 활용했다.



미로 찾기의 코드는 위와 같다.

현재 좌표가 목적지의 좌표와 같지 않다면 while문을 실행한다.

현재 좌표에서 상하좌우 4방향을 탐색하여 갈 수 있는 길(‘0’)이면 스택 자료구조에 push한다. 현재 좌표에서 갈 수 있는 길을 모두 스택에 담는다고 생각할 수 있다.

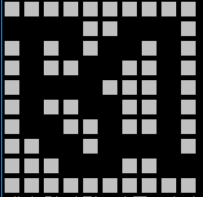
이후 스택의 값을 pop하고 그 위치로 이동한다.

위와 같은 알고리즘을 사용할 경우, 출발점부터 갈 수 있는 길을 스택에 담아 가며 이동하게 되고, 막힌 길에서는 이전 갈림길에서 저장했던 다른 길로 이동하여 다시 길을 진행하게 된다.

위와 같은 알고리즘으로 진행하다가, 갈 수 있는 방향이 2군데 이상 존재하는 갈림길이라면 트리 노드에 추가하는 방식으로, 트리 자료구조를 구현할 수 있었다.

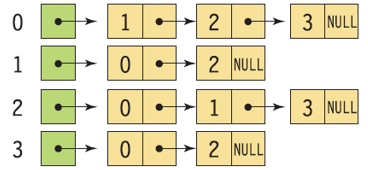
* 그래프

그래프와 트리의 가장 큰 다른 점은 사이클이 존재한다는 점이라고 생각했다. 그래서 다음과 같이 내부에 사이클이 존재하는 미로를 만들었다.



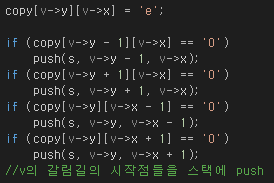
그래프 자료구조에 미로를 읽어 저장할 때에도 위에서 활용한 스택 미로 찾기 알고리즘을 응용했다.

먼저 (0,0)부터 (10,10)까지 모든 칸을 탐색하여 출발점, 도착점, 막힌 길, 갈림길을 판별하고 이를 그래프 자료구조에 담았다.

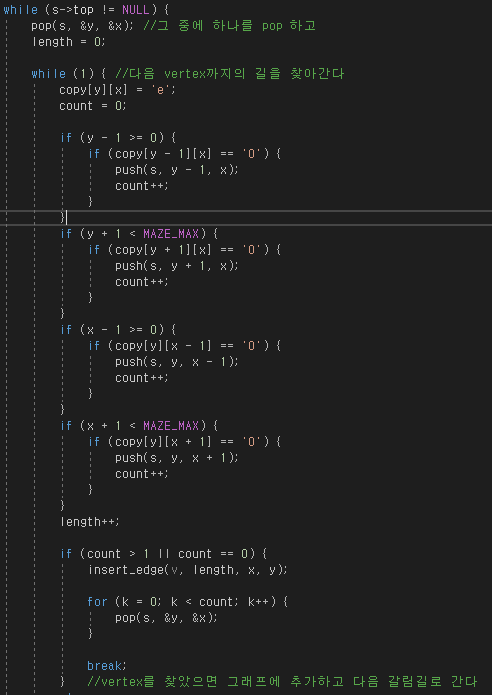


그래프 자료구조는 위와 같은 형태로, 각 vertex들을 연결리스트로 구현하였다.

각각의 vertex로부터 연결된 vertex들을 리스트에 담아 주어야 하기 때문에, 각각의 모든 vertex를 출발점으로 연결된 모든 vertex까지 길을 찾아가는 작은 미로 찾기를 구현했다.



매개변수로 받은 vertex부터 시작하여

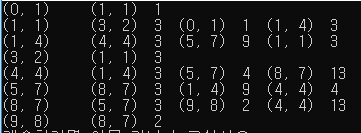


연결된 vertex를 스택에 push하고 pop하면서 찾아간 후

이를 그래프 자료구조에 추가하는 형식으로 구현하였다.

또한 가는 거리를 계산하여 후에 이루어질 Dijkstra 알고리즘에 활용할 수 있도록 하였다.

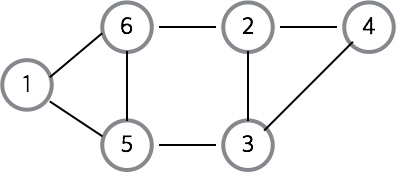
구현하여 그래프 자료구조에 담긴 내용을 출력하여 보았고 그 내용은 아래와 같다.



* 1. DFS알고리즘을 사용한 미로탐색 방법
* DFS란?

Depth First Search의 약자로 깊이 우선 탐색을 뜻한다. 더 이상 갈 곳이 없는 막힌 정점에 도달할 경우, 그 전 정점으로 돌아가게 된다.

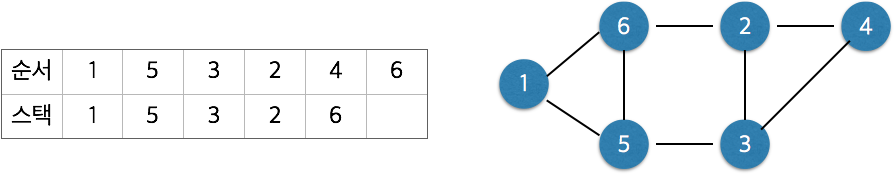
예를 들어



그림을 1번부터 DFS알고리즘의 순서대로 출력한다면,

1 – 5 – 3 – 2 – 4 – 6 순서대로 출력하게 될 것 이다.

이를 스택을 사용해서 길이 없을 경우 pop을 통해 이전 노드로 되돌아 가는 방식을 이용하면



스택에는 이렇게 쌓이게 될 것이다.

트리의 깊이가 목적 노드 깊이와 같을 때 최악의 경우 모든 노드를 다 검사하게 되고 이때 시간 복잡도는O(b­­d)이다. 공간복잡도는 트리에서 루트 노드로부터 어떠한 노드까지의 경로상에 있는 각 노드의 자식들 중 해당 노드를 제외한 것을 저장하고, 이때 가장 긴 경로의 길이가 d이면 공간복잡도는 O(bd)로 나타나게 된다.

\*장점 : 공간복잡도가 비교적 작다. 목표 노드가 깊은 단계에 있을 경우 빠른 탐색이 가능하다.

\*단점 : 해가 속하지 않은 경로의 끝까지 탐색하므로 비효율적이다. 최단경로가 보장되지 않는다.

Pseudocode

DFS(G,v) ( v is the vertex where the search starts )

Stack S := {}; ( start with an empty stack )

for each vertex u, set visited[u] := false;

push S, v;

while (S is not empty) do

u := pop S;

if (not visited[u]) then

visited[u] := true;

for each unvisited neighbour w of u

push S, w;

end if

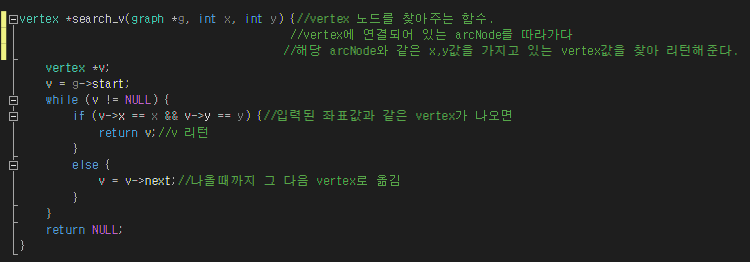
end while

END DFS()

* 코드 설명

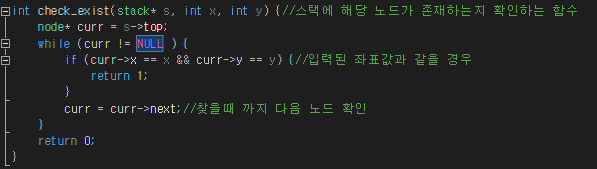
\*스택은 미로를 읽는 부분에서 같이 사용

1)정점을 찾는 함수



스택에 push된 노드와 같은 좌표값을 가지고 있는 vertex노드를 찾아 리턴해주는 함수이다. 매개변수로 그래프와 좌표값을 받아 그래프의 vertex들을 같은 좌표값을 가진 vertex가 나올 때까지 다음 vertex를 찾는 함수이다. 만약 해당 좌표값과 같은 값을 가진 vertex가 없을 경우, NULL을 리턴한다.

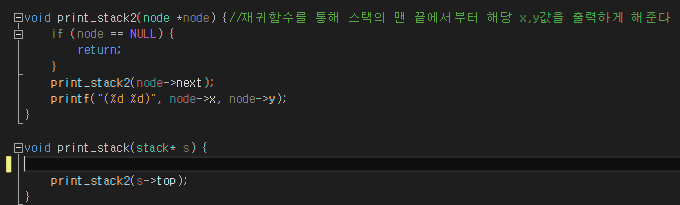
2)스택에 해당 노드가 존재하는지 확인하는 함수



DFS탐색을 할 때 해당 노드가 이미 스택에 존재한다면 굳이 다시 탐색할 필요가 없음! 스택과 좌표를 매개변수로 받아, 스택에 있는 노드들을 돌며 같은 좌표값을 가진 노드가 있을 경우 return 1 없으면 return 0

3)방문한 좌표값들을 출력하는 함수

처음 만든 출력함수는 맨 마지막에 들어간 top 즉, 탈출구부터 미로의 시작점까지를 출력해주는 함수였지만 미로의 시작부터 보여주는 것이 더 탐색의 의미에 맞는 것 같아서 이를 고민해 보았다.



print\_stack로 출력할 스택을 받는다. 그러면 print\_stack은 노드를 매개변수로 하는 print\_stack2을 부른다(print\_stack2는 입력 받은 스택의 탑 노드를 가르킨다 ). 그럼 print\_stack2는 탑 노드를 입력 받아 재귀함수를 이용해 스택의 끝까지 들어간 후, NULL을 만나게 되면 맨 끝 스택의 좌표부터 찍으면서 나오게 된다. 이렇게 하면 방문한 노드들의 좌표값을 순차적으로 화면에 찍을 수 있다.

4)DFS함수

void DFS(graph \*g) {

int x, y; //pop(stack\*, int\*, int\*)을 위한 더미 변수.

stack\* s; //노드를 넣을 스택 선언

s = malloc(sizeof(stack)); //스택 할당

s->top = NULL;

vertex\* init\_node = g->start; //그래프의 시작노드와 init\_node를 같다고 설정

push(s, init\_node->x, init\_node->y); //처음 노드는 미로의 입구이므로 무조건 스택에 push

node\* current\_node = s->top; //스택의 현재 노드를 가르키는 current node

while (current\_node != NULL) { // 현재 노드가 0이 아닐때

vertex\* find\_degree\_from = search\_v(g, current\_node->x, current\_node->y);

//search\_v를 통해 찾은 current 와 같은 좌표를 가진 vertex를 저장

arcNode\* next\_edge = find\_degree\_from->degree;

//find\_degree\_from에 연결된 arcNode를 next\_edge에 저장

while (next\_edge != NULL && check\_exist(s, next\_edge->x, next\_edge->y)) {

//next\_edge가 null이 아니고, 스택에 존재한다면,(check\_exist -> (x, y)를 방문했는지 확인하기 위함. stack에는 방문한 위치가 저장되어 있음)

next\_edge = next\_edge->nextArc; //다음 arcNode로 넘어감

}

if (next\_edge == NULL) { //arc Node 노드가 더 이상 없다면 (다음 경로X)

pop(s, &x, &y);

current\_node = s->top;

}

else { //arcNode 노드가 있을 경우(다음 경로O)

find\_degree\_from->degree = next\_edge->nextArc;

// degree의 시작점부터 next\_edge까지는 다시 갈 일이 없음. (이미 갔던 곳이기 때문)

//degree -> ㅁ-ㅁ-ㅁ-ㅁ next\_edge가 세 번째 ㅁ이라면 첫 번째, 두 번째, 세 번째 ㅁ은 이미 방문한 노드를 뜻하므로 더 이상 필요가 없습니다.

push(s, next\_edge->x, next\_edge->y); //stack에 방문한 위치 추가.

current\_node = s->top;

if (next\_edge->x == 9 && next\_edge->y == 8) { //미로 탈출에 성공 printf("DFS result : ");

print\_stack(s); //경로 출력

break;

}

}

}

}

미로의 처음 좌표를 스택에 저장한 후, 저장된 스택과 같은 좌표를 가진 vertex를 찾는다(find\_degree\_from). 해당 vertex에 연결된 arcNode(next\_edge)들을 순회하며 갈 수 있는 길인지 확인한다. 1)next\_edge의 값이 NULL이 아니면서 스택에 존재할 경우, 이미 탐색했기 때문에 다시 탐색을 진행하지 않고 연결된 다음 arcNode로 넘어간다. ⁴)만약 next\_edge의 값이 스택에 존재하지 않고 NULL이 아니라면, 갈 수 있는 길이기 때문에 해당 좌표값을 스택에 push해준다. 3)next\_edge가 NULL일 경우는 다음 경로가 없다는 뜻이므로 pop을 해준다. 이 과정을 반복하다 미로의 탈출 좌표인 (9,8)에 도달할 경우, 미로탈출에 성공한 것이므로 print\_stack를 호출해 지나온 경로를 출력해주고 함수를 빠져나온다.

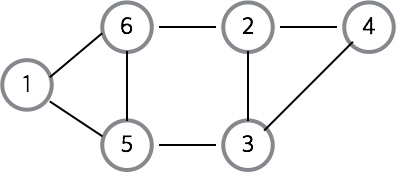
* 구현 시 애로사항

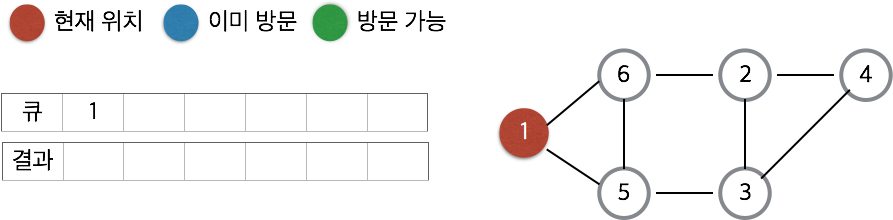
처음 DFS를 만들 때 DFS함수를 따로 만들고 재귀를 통해 DFS를 불러오는 방식을 사용하였습니다. 하지만 스택에서 재귀의 구조를 다시 사용하는 것은 비효율적이기 때문에 이를 해결하기 위해 고민하는 과정이 좀 오래 걸렸습니다. 또한 미로를 탐색하는 부분에서는 길이 있을 때 스택에 있는지 없는지 까지 고려해야 하는데 그 부분을 고려하지 못해 구현이 어려웠습니다. 그리고 방문한 위치를 추가하는 push함수에서 이미 갔던 곳은 지나치게 하는 방식을 어떻게 구현할까 고민하는데 오랜 시간이 걸렸습니다.

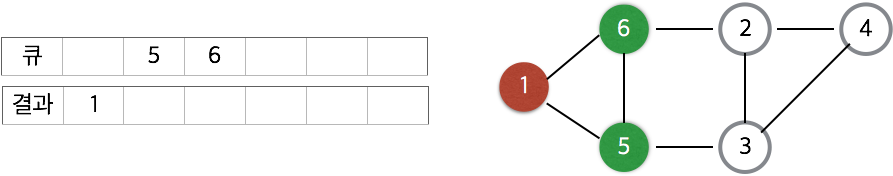
* 1. BFS알고리즘을 사용한 미로탐색 방법
* BFS란?

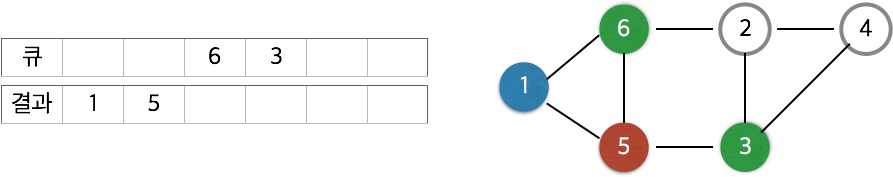
너비 우선 탐색으로, DFS와 달리 시작 노드에서 출발해 그 노드의 모든 인접 노드를 방문한 후 다음 노드에 대한 탐색(그 노드의 모든 인접 노드 방문)을 하는 알고리즘이다. 방문한 인접 노드들에 대해 그 다음에 그 노드들의 인접 노드들을 방문하며 탐색이 이루어지므로, 즉 순서대로 노드를 저장한 후 순서대로 노드 정보를 출력해 사용하므로 큐 자료구조를 이용해 구현한다.

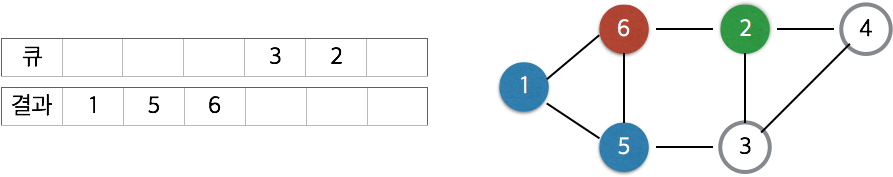
예로 밑의 그림과 같은 그래프가 있다고 하면 아래와 같이 진행된다.

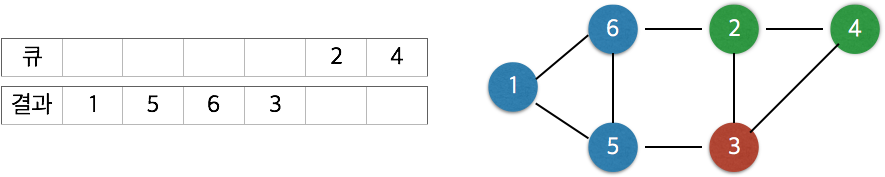


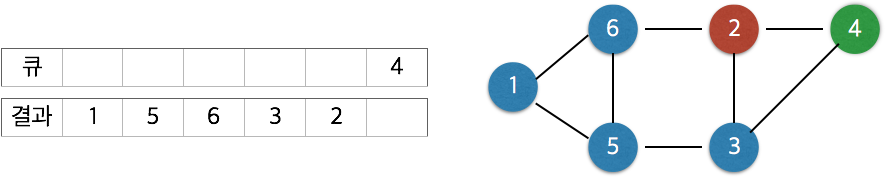


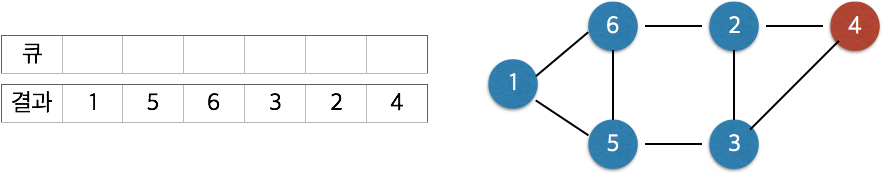












Bfs는 매우 간단한 알고리즘으로, 노드간 거리가 없을 때(즉 가중치가 없을 때) 최단 경로를 보장한다는 장점이 있지만 경로가 길 경우 탐색 가지가 급격히 증가해 많은 기억 공간을 필요로 하고, 해가 존재하지 않는 유한 그래프의 경우 모든 그래프 탐색 후 실패로 끝난다는 점, 무한 그래프의 경우 끝낼 수 없다는 단점이 있다.

* 코드 설명

1)자료구조 큐를 사용하기 위한 구조체 및 함수

typedef struct queuenode {

int x, y;

struct queuenode \*next;

}QueueNode;

typedef struct queue{

QueueNode \*front;

QueueNode \*rear;

}Queue;

typedef struct point {

int x;

int y;

}Point;

위와 같은 구조체와 아래와 같은 함수로 큐를 구현하였다.

void init\_q(Queue \*q) {

q->front = q->rear = NULL;

}

int q\_is\_empty(Queue \*q) {

return (q->front == NULL);

}

void enqueue(Queue \*q, int x, int y) {

QueueNode \*node = (QueueNode\*)malloc(sizeof(Queue));

if (node == NULL) printf("메모리 할당 실패\n");

else {

node->next = NULL;

node->x = x;

node->y = y;

if (q\_is\_empty(q)) {

q->front = node;

q->rear = node;

}

else {

q->rear->next = node;

q->rear = node;

}

}

}

Point dequeue(Queue \*q) {

QueueNode \*tmp = q->front;

Point item;

if (q\_is\_empty(q)) printf("큐가 비어 있습니다.\n");

else {

item.x = tmp->x;

item.y = tmp->y;

q->front = q->front->next;

if (q->front == NULL)

q->rear = NULL;

return item;

}

}

디큐함수에서 두 개의 정수를 반환해야 하기 때문에 point 구조체를 정의하였다.

2)bfs함수(최단 경로 탐색)

#define MAZE\_MAX 10

int visited[MAZE\_MAX][MAZE\_MAX] = { 0 };

int dis[MAZE\_MAX][MAZE\_MAX] = { 0 };

Point pre[MAZE\_MAX][MAZE\_MAX] = { -1, -1 };

void bfs(graph \*g) {

vertex\* v;

arcNode \*w;

Point item;

int x, y;

Queue q;

init\_q(&q);

v = g->start;

x = v->x;

y = v->y;

visited[x][y] = 1;

pre[x][y].x = -2;

pre[x][y].y = -2;

//printf("%d, %d\n", x, y);

enqueue(&q, x, y);

while (!q\_is\_empty(&q)) {

item = dequeue(&q);

x = item.x;

y = item.y;

for (v = g->start; v->x != x||v->y != y; v = v->next);

for (w = v->degree; w != NULL; w = w->nextArc) {

if (visited[w->x][w->y] == 0) {

visited[w->x][w->y] = 1;

//printf("%d, %d\n", w->x, w->y);

enqueue(&q, w->x, w->y);

}

if(pre[x][y].x!=w->x&&pre[x][y].y != w->y)

if (dis[w->x][w->y] == 0 || dis[w->x][w->y] > dis[x][y] + w->arc) {

dis[w->x][w->y] = dis[x][y] + w->arc;

pre[w->x][w->y].x = x;

pre[w->x][w->y].y = y;

}

}

}

}

먼저, bfs함수를 구현할 때 필요한 전역변수들을 정의하였다. 노드의 정보가 좌표로 표현되고 있기 때문에 2차배열들을 사용했다. 방문여부를 저장할 visited와 시작노드(좌표)에서 각 노드(좌표)까지의 거리를 저장할 dis, 각 노드(좌표)의 이전노드(좌표) (즉 경로에서 거쳐온 이전 좌표)를 저장할 pre. 각 인덱스가 좌표를 나타내는 것으로 생각하였다.

bfs의 원형은 반환값이 없으므로 void형이고, 미로의 정보를 받아올 그래프가 필요하므로 매개변수로 그래프를 받는다. 사실 정보만 받아오면 되므로 굳이 포인터로 그래프를 받을 필요는 없지만, 별 차이 없다고 생각해 그냥 포인터로 받았다.

코드에 대해 설명하자면, 우선 그래프의 정보를 받아서 쓸 vertex와 arcNode의 포인터 변수를 선언하였고, 큐의 정보를 받아오기 위한 Point변수, 큐를 이용하기 위한 Queue변수를 선언하였다. 함수를 시작한 시점에서 시작노드는 방문한 것이기에 visited의 해당 좌표를 1로 바꾸어 주고, 시작노드라는 것을 알려주기 위해 이전노드를 가리키는 배열 pre에 -2,-2를 대입하였다. 이것은 이후 print\_path 최단 경로를 출력하는 함수에서 쓰임을 위한 것이다. v에 그래프의 시작노드를 넣고 시작좌표를 큐에 넣은 후, 본격적으로 미로를 탐색한다. Bfs로 미로를 탐색하되 방문하면 visited의 해당 좌표를 1로 바꾸고 큐에 해당 좌표를 넣는다. 노드를 방문시(탐색시)에는 해당arc노드가 가진 정점과의 거리값arc와 시작노드에서 정점까지의 거리dis를 더한 값이 시작노드에서 해당arc노드까지의 거리값보다 크거나 0이면(즉 거리를 재는 게 처음이면) dis를 갱신해주고 이전 노드를 갱신해준다. 여기서는 다익스트라 알고리즘의 개념이 들어갔다고 볼 수 있다.

위의 코드는 문제점을 모두 고친 코드이지만, 가중치가 없는 그래프에 대해 bfs를 구현해보고 가중치가 있는 그래프에 대해 구현해보고, 최단 거리를 찾는 단계까지 발전시키면서 코드를 짰는데, 너비우선탐색은 시작 노드에 대한 레벨별로 탐색이 이루어지기 때문에 같은 레벨에 있는 노드까지는 최단 거리를 잘 찾고 잘 갱신되지만, 그래프 내에 순환이 있고, 가중치에 의해 거쳐가는 노드의 수는 많지만 거리는 더 짧은 경로를 찾지 못하는 문제점이 있었다. 방문한 노드에 대해서도 계속 갱신하되, 이전 레벨의 노드는 탐색하면 안 된다는 것을 깨닫고, 문제점을 고친 위의 코드를 완성하였다.

3)최단 경로 출력 함수

void print\_path(int x, int y) {//도착 좌표 입력

if (pre[x][y].x == -1 && pre[x][y].y == -1) {

printf("도착지점까지 경로가 존재하지 않습니다.\n");

return;

}

if (pre[x][y].x == -2 && pre[x][y].y == -2) {

printf("BFS 탐색 최단 경로 : (%d, %d)", x, y);

return;

}

print\_path(pre[x][y].x, pre[x][y].y);

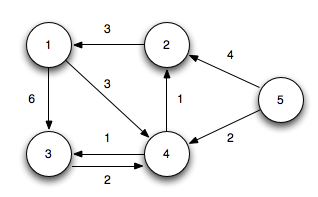
printf(" (%d, %d)", x, y);

}도착좌표를 입력받아 pre배열에 저장된 값을 따라가며 출력하는 함수이다. 재귀함수 호출을 통해 거꾸로 쭉 올라가서 순서대로 경로를 출력(반환)한다. Base case는 시작좌표, 혹은 도착좌표와 이어진 길이 없을 때이다.

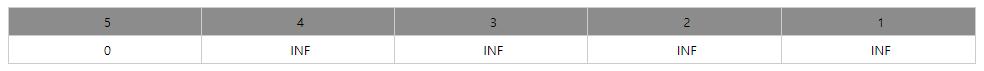
* 1. Dijkstra 알고리즘을 사용한 미로탐색 방법
* Dijkstra란?

하나의 정점(vertex)에서 다른 모든 정점까지의 최단 경로를 구하는 알고리즘으로 간선들은 모두 양의 간선(arc)들을 가져야 한다. 기본적인 로직은 첫 vertex를 기준으로 연결되어 있는 arc node들을 추가해가며 최단 거리를 갱신하는 것이다. 정점을 잇기 전까지는 시작점을 제외한 모든 정점들은 모두 무한대의 값을 갖는다.

정점 A에서 정점 B로 이어지면 정점 B가 갖는 최단 거리는 시작점부터 정점 A까지의 최단거리 + 점A와 점B 간선(arc)의 가중치와, 기존에 있던 정점 B의 거리 중 작은 값이 B의 최단 거리가 된다.



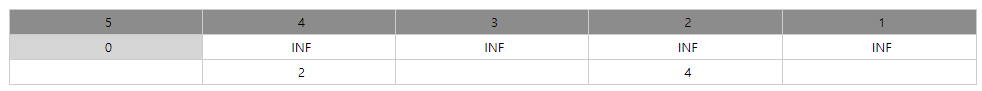
이 그림에서 시작점이 5번 정점이라 하면 일단 5번 노드를 제외한 나머지 정점들이 갖는 최단경로는 아직 연결되어 있지 않으므로 무한대이다.



이제 여기서 경로가 가장 짧은 정점을 고른다. 여기서 5번 노드이고 5번 노드와 연결되어 있는 노드는 2,4번 노드이다.

먼저 2번 노드부터 보면, 2번 노드의 최단 거리를 갖고 있는 현재 최단거리(무한)와 5번 노드의 최단 거리(0)+2번~5번의 가중치(4) 값 중 작은 값으로 갱신한다

이 때 무한대와 0+4 중 4가 더 작으므로 2번 테이블의 거리를 4로 갱신하고 4번 노드 또한 현재 4번 노드의 최단거리(무한)와 5번 노드의 최단거리(0)+4~5번의 가중치(2)값 중에 더 작은 0+2=2의 값으로 update한다



업데이트 한 결과이다. 이런식의 과정을 더 이상 최단거리를 업데이트 할 수 없을 때까지 하면 결국 하나의 테이블이 완성된다.

즉 이런 방법을 사용하여 미로찾기에서 출발점이라는 정점에서 다른 모든 정점까지의 최단 경로를 구할 수 있으며 테이블로 표현 가능하다.

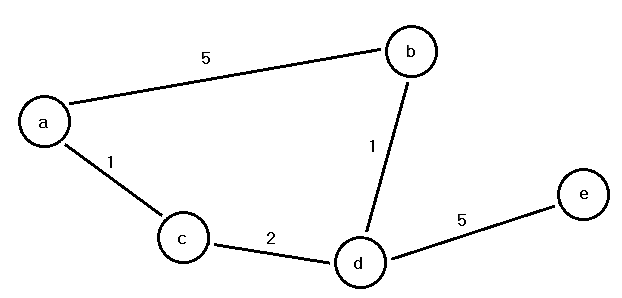
* 우선순위 큐를 활용한 Dijkstra

일반 큐를 써도 우선순위 큐를 쓴 것과 결과는 같지만 속도에 더 이점이 있다.

기존 일반 다익스트라에서는 가장 최단 거리를 가진 정점을 찾기위해 선형 탐색을 이용하지만 MinHeap을 이용하면 최솟값이 가장 위로 올라오도록 빠르게 정렬되기 떄문이다.

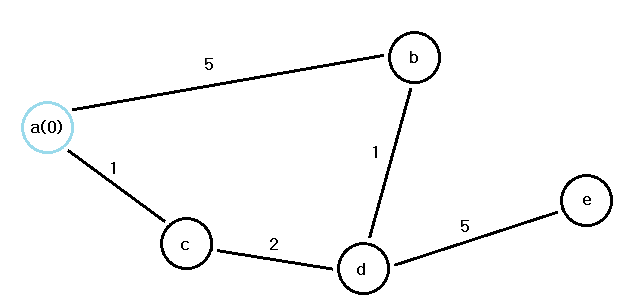
예를들어 A정점에서 B정점으로 가는 값이 이미 있는데 더 작은 값이 생긴다면 그냥 push한 후에, 나중에 가중치를 비교하여 가중치가 더 큰 것이었다면 무시하도록 한다.

1) a번 정점에서 시작한다고 가정

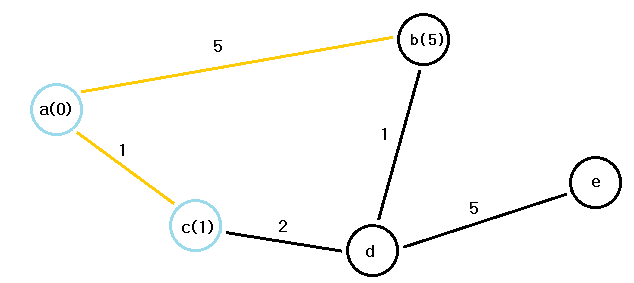


2) a번 정점은 시작점이므로 0의 최단 거리 값을 가짐

현재 dist 값 – dist[0]=0



3) a번 정점에서 c번 정점으로의 최단거리는 1, a번 정점에서 b번 정점으로 가는 최단거리는 5이고 a번 정점에서 가장 최단거리는 c 이므로 c부터 방문함

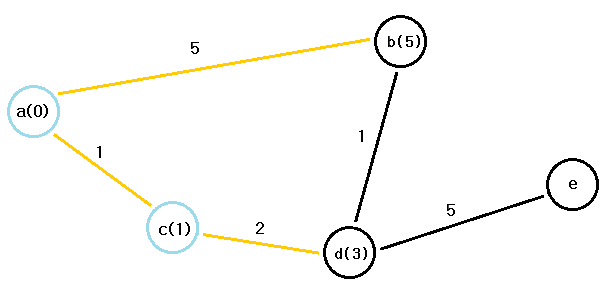


현재 dist값 – dist[0]=0

dist[1]=5

dist[2]=1

4) c번 정점에서 d까지의 최단거리는 3이다



현재 dist 값 – dist[0]=0

dist[1]=5

dist[2]=1

dist[3]=3

5) c번 정점에서의 방문이 끝났으니 그 다음 가장 최단 거리를 가진 d번 정점을 방문함

이와 같은 방식으로 더 이상 갱신할 수 없을 때 까지 반복한다.

* 코드 설명

우선 크게 5가지 함수가 필요합니다.

1) doDijkstra : Dijkstra를 실행하는 함수

2) Dijkstra 테이블을 업데이트 하는 함수

3) Dijkstra를 실행하는 도중에 최단 거리가 밝혀지지 않은 vertex가운데 가장 작은 distance를 가진 vertex를 반환하는 함수

4) Dijkstra 테이블을 초기화 하는 함수

5) 테이블을 출력하는 함수

typedef struct dijkstra{ //다익스트라 테이블 구조체

int found;//(k)최단거리를 찾았는가? 1=yes, 0=no

int distance;//최단거리(미로의 한 갈림길에서 다음 갈림길까지의 거리)

Point prev\_vertex;//바로 직전에 어느 vertex를 거쳤는가(x,y좌표)

}Dijkstra;

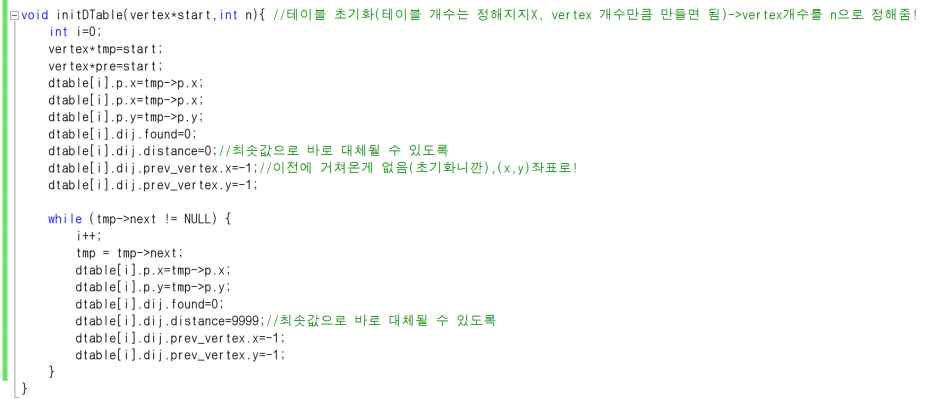
typedef struct \_dtable{

Point p;//x,y좌표

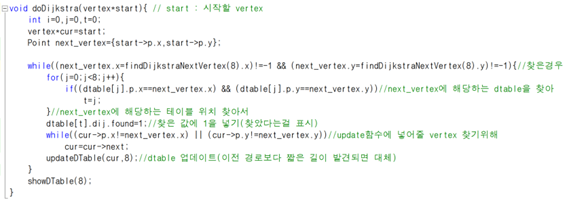
Dijkstra dij;//다익스트라 구조체

}Dtable;

기존에서 추가된 구조체이며 table은 x, y좌표, 최단거리를 찾았는지 아닌지의 여부, 최단거리, 이전 vertex의 x, y 좌표로 구성됩니다. Dtable은 전역변수로 선언하였습니다.



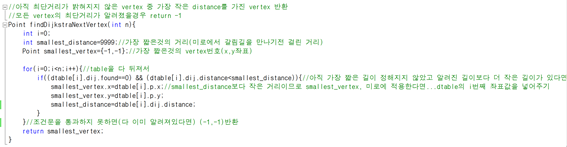
우선 initDTable로 table을 초기화 시켰습니다. 매개변수로 start vertex와 총 vertex의 개수를 입력으로 넣고 Table은 (x,y)좌표, found(찾았는지 여부), distance(최단 거리), prev vertex(이전 vertex의 좌표)로 초기화할때 현재 있는 vertex의 x,y는 각각 넣어주고 found는 0으로, distance는 출발하는 vertex에 해당하는 표만 0, 나머지와의 거리는 9999로 설정하였습니다.(계속 최단거리가 업데이트 되도록 하는 알고리즘이라 처음의 거리를 거의 무한으로 설정)



doDijkstra로 실제로 main에서 실행시킬 함수입니다. 현재 dtable은 위에서 보다시피 전역변수로 Dtable dtable[NUM\_VERTEX]; 이렇게 선언해 놓은 상태고 매개변수로는 출발 vertex를 넣어주었고, Point next\_vertex에 출발 x,y좌표를 넣어주었습니다. 그 다음 while((next\_vertex.x=findDijkstraNextVertex(8).x)!=-1 && (next\_vertex.y=findDijkstraNextVertex(8).y)!=-1) 이것을 보면 지금 findDijkstraNextVertex(8)이 vertex 테이블에서 최단거리가 밝혀지지 않은것의 vertex를 찾아서 그 중 가장 작은 vertex의 x,y좌표를 반환하는 함수인데 (아래에서 다시 설명하겠습니다! )

그렇게 찾은 후 조건문에 들어가서 그 아래의 for문은 그 x,y 좌표에 해당하는 dtable의 위치를 찾아가서 t에 넣고 가장 짧은 거리를 가진 vertex를 발견했으므로 dtable[t].dij.found=1;로 찾았음을 저장해준 후에 이제 그 아래 while문은 또 update함수에 넣어줄, 즉 t에 해당하는 vertex의 x,y좌표를 가진 실제 vertex를 찾는 과정입니다.

이렇게 찾은 ‘cur’ vertex를 updateDTable함수에 넣어서 update를 해주는데 이는 아래서 다시 설명하겠습니다.

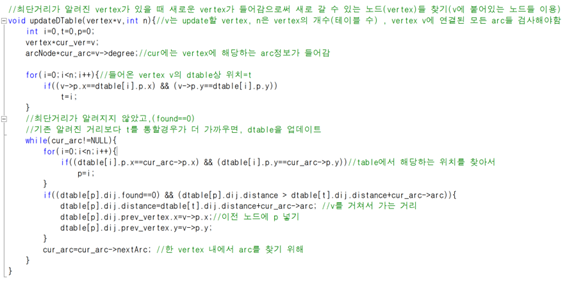


위에서 말한 findDijkstra함수입니다. 우선 smallest 거리를 9999로 해놨고 smallest vertex를9999로 설정해 두었습니다

그리고 나서 이제 table을 for문으로 뒤지는데, if문에는 found==0(아직 최단거리를 발견하지 못했고), dtable[i].dij.distance(i번째 dtable의 거리)가 smallest\_distance(처음엔 9999)보다 작으면 이제 조건문을 들어갑니다.

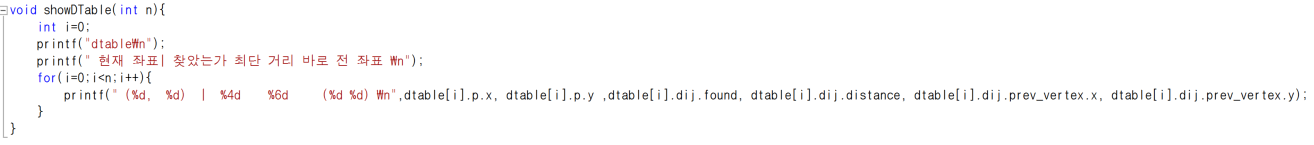
이렇게 들어온 뒤에는 smallest\_vertex의 x,y좌표를 조건문에 해당하는 dtable[i].p.x, p.y를 넣어주고 smallest\_distance에는 거리를 넣어줍니다. Smallest거리가 갱신됐으므로 반복문을 돌면서 if문안에 맞지 않는 경우가 생깁니다. 그러면 이제 반복문이 끝나고 smallest\_vertex를 출력합니다. 이 때 조건문을 통과하지 못한다면 smallest\_vertex는 -1,-1을 출력합니다(이래서 위에 doDijkstra함수에서 while((next\_vertex.x=findDijkstraNextVertex(8).x)!=-1 && next\_vertex.y=findDijkstraNextVertex(8).y)!=-1){//찾은경우

에 해당합니다.



다음은 table을 update하기 위한 함수입니다. 매개변수 vertex\*v는 doDijkstra함수에서 볼 수 있듯이 현재 업데이트 할 vertex를 넣어주고 arcNode\*cur\_arc=v->degree를 보면 vertex에서 arc들을 찾아 나갈 수 있게 cur\_arc를 선언해주었습니다.

그 다음에 for문으로 역시 매개변수로 들어온 update할 vertex가 dtable에서 어디에 해당하는지 찾아서 t에 그 인덱스를 저장했고, While(cur\_arc!=NULL)은 arc끝까지 갈 때까지 다시 for문에서 그 arc가 table에 어디에 해당하는지 위치를 찾은 후 그 인덱스를 p에 저장했습니다. 그 후 조건문을 들어가 dtable[p].dij.found(이 때 p는 들어온 vertex가 있는 table의 index)이 0으로 아직 최단거리를 찾지 못했고, dtable[i].dij.distance(그 p에 해당하는 vertex의 거리가) dtable[t].distance(t는 while문 안에서 cur\_arc의 dtable상 위치) + cur\_arc->arc(해당하는 cur\_arc의 arc)를 합한것보다 크다면, 즉 기존의 거리가 새로운 거리+아크(가중치)값보다 컸다면 조건문을 들어가 업데이틀 합니다. 이 때 업데이트는 Dtable[p].dij.distance=새로운 거리 + 아크(가중치)값으로 대체하고 이전 노드의 좌표를 넣어주었습니다.

Table을 출력하는 함수입니다.

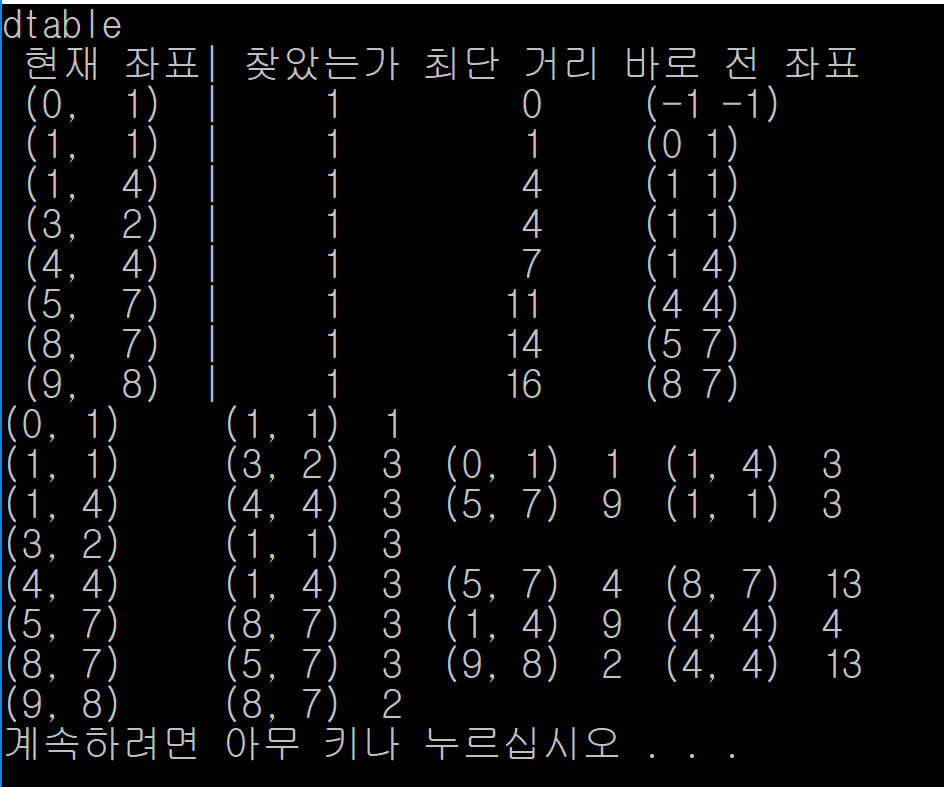
-구현 시 애로사항

우선 구현을 시작하였을 때 우선순위 큐에 대한 개념이 전혀 없었고 그래프와 다익스트라 알고리즘에 대한 공부만 한 후 간단한 다익스트라 알고리즘을 구현해 보았습니다. 그 후 미로에 적용하는 식으로 구현했습니다. 그러나 기존에는 vertex의 값이 하나로 이루어져 있었기에 그것을 index로 하면 자동으로 vertex 값이 작은 순으로 테이블이 만들어지고 index를 찾아 접근하기가 매우 편했지만 미로에서는 x,y좌표가 있기에 테이블을 찾아가는 것도 매우 어려웠습니다. 처음에 우선 table자체를 연결리스트로 구현하였지만 조금 생각해보니 굳이 그럴 필요는 없는 것 같아서 구조체 배열을 만들었습니다. 찾을 때는 테이블 크기만큼 반복문을 돌려 x,y좌표에 맞는 위치를 찾아내는 방식으로 구현하였습니다. 그러나 또 다른 문제점이 있었습니다. 첫 vertex에 연결된 다음 vertex로의 탐색과정, 즉 처음 과정은 잘 업데이트가 되었는데 그 다음에 제가 생각한 후 구현한 코드로는 반복문을 돌며 다른 vertex들도 업데이트가 되어야 하는데 update가 되지 않았습니다. 각 함수마다 printf함수로 찍어가며 디버깅을 해본 결과 문제는 doDijkstra함수에 있었습니다.

while((cur->p.x!=next\_vertex.x) || (cur->p.y!=next\_vertex.y))

cur=cur->next;

이 코드 부분에서 기존에 ||부분을 &&로 구현하였기 때문입니다. x좌표랑 y좌표 둘 다 달라야 cur->next로 넘어가는데 좌표는 x나 y중 하나만 달라도 다른 거고 다를 때 next로 넘어가는 게 의도이므로 &&을 ||로 고쳐야 했습니다. 그러나 이것을 찾아내는 디버깅 과정이 쉽지 않았습니다. 반복문이 돌지 않는 이유가 doDijkstra함수가 보시다시피 updatetable, findDijstra 등 여러 함수와 맞물려 진행되므로 어디가 잘못됐기에 반복문이 돌아가지 않는지를 찾는 과정이 매우 어려웠습니다. 이 때문에 몇일 밤을 세웠고 결국 팀원들에게 코드를 설명해 준 후 같이 고민해보고 나서야 원인을 찾을 수 있었습니다.



이것이 출력함수 입니다. 보시다시피 표가 위(dtable), 아래로 구성되어있는데 아래는 한석호 학생이 만든 각 vertex에 연결된 arc들을 나타낸 테이블입니다. (0,1) , (1,1) ,(1,4)…로 이루어진 첫 열이 vertex가 되며 그 오른쪽에 괄호로 이루어진 부분이 vertex에 연결된 arc node들, 그리고 괄호로 이루어지지 않고 숫자 하나가 거리(가중치) 입니다. 이를 참고하여 dtable을 본다면 현재 좌표가 첫 열에 그대로 나와있고 모두 최단거리를 찾았기에 0으로 초기화했던 found가 1로 바뀌었습니다. 또한 각 최단거리가 나와있고 바로 이전에 어느 좌표를 거쳤는지도 출력하도록 하였습니다.

1. 결과 & 동작
   1. Main 함수

int main() {

char maze[MAZE\_MAX][MAZE\_MAX] = {

{ '1', '1', '1', '1', '1', '1', '1', '1', '1', '1' },

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

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

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

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

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

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

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

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

{ '1', '1', '1', '1', '1', '1', '1', '1', '1', '1' },

};

int countZero;

int i, j;

graph\* g = create\_graph();

vertex\* tmp;

//maze[j][i]; i = x, j = y;

for (i = 0; i < MAZE\_MAX; i++) {

for (j = 0; j < MAZE\_MAX; j++) {

if (i == 0 || j == 0) {

if (maze[j][i] == '0')

insert\_vertex(g, i, j);

}

//start point, end point

else {

if (maze[j][i] == '0') {

countZero = 0;

if (maze[j - 1][i] == '0')

countZero++;

if (maze[j][i - 1] == '0')

countZero++;

if (maze[j + 1][i] == '0')

countZero++;

if (maze[j][i + 1] == '0')

countZero++;

if (countZero > 2 || countZero == 1) {

insert\_vertex(g, i, j);

}

}

}

//check 4-direction

}

}

//처음 vertex들을 그래프에 추가함

//print\_vertex(g);

tmp = g->start;

add\_edge(tmp, maze);

while (tmp->next != NULL) {

tmp = tmp->next;

add\_edge(tmp, maze);

}

//각 vertex를 돌면서 연결된 arcNode들을 추가함

/\*tmp = g->start;

print\_edge(tmp);

while (tmp->next != NULL) {

tmp = tmp->next;

print\_edge(tmp);

}\*/

print\_graph(g);

//print는 vertex edge 합쳐서 출력하도록 바꿀 수 있음

//다른 분이 올려주셨음

system("pause");

printMaze(maze);

bfs(g);

print\_path(9, 8);

printf("\n");

initDTable(g->start, 8);

doDijkstra(g->start);

printf("\n");

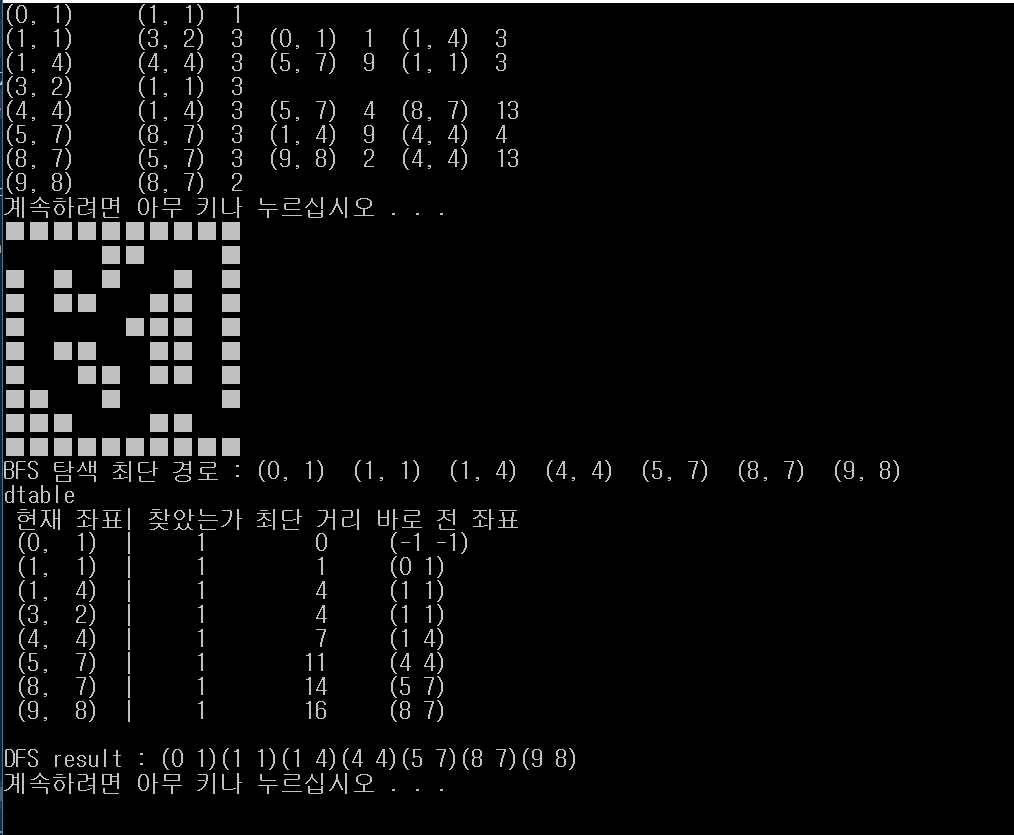
DFS(g);

printf("\n");

return 0;

}

* 1. 실행 화면



1. 후기 및 기록
   1. 공부한 부분

* 그래프

수업 진도가 그래프까지 나가지 않은 상황에서 미리 그래프를 공부하고 구현하고자 했습니다. 어려운 부분도 많고 잘못된 부분도 있을 수 있겠지만, 수업을 들으면서 혼자 공부했던 부분과 비교하고 더 발전시켜 나갈 수 있을 것으로 생각합니다.

* 1. 아쉬운 점
* Dijkstra 알고리즘

아쉬운 점이 있다면 우선순위 힙을 먼저 알았다면 좋았을 것이라는 생각입니다. 물론 다익스트라를 공부하면서 우선순위 힙을 공부하지 않은 제 잘못이지만 우선순위 힙을 사용했다면 아마 각 table에 해당하는 vertex를 찾기 위해 하나하나 반복문을 써서 찾을 필요는 없었을 것 같습니다. 또한 저렇게 반복문을 써서 하나하나 찾는 과정에서 매우 비효율적일 것 같습니다… 최단 거리를 찾기 위해 사용한 알고리즘인데 저런 반복문 하나하나의 과정 때문에 속도가 느려질 것 같습니다.

* A\*알고리즘

A\*알고리즘에 대해 학습하였으나, 프로젝트의 진행 상황과 발표 일정을 고려할 때, 구현하여 발표에 추가하는 것이 어렵다고 판단하여 보류하였습니다. 더 충분한 시간이 있다면 공부한 것을 바탕으로 구현해보고 싶습니다.

* 미로 생성 알고리즘

미로를 생성하는 방법에 대해서도 찾아보았고, Kruskal, prim 등의 알고리즘이 있다는 것을 확인하였습니다. 위와 같이 일정을 고려하여 발표 내용에는 추가하지 못하였습니다.

* 미로 출력

현재 미로를 출력하는 함수는 단순히 미로의 형태만 보여주는 함수입니다. 좀 더 가시적으로 보이기 위해 어떤 노드들을 방문했는지도 보여주었으면 하는데 그러지 못해 아쉽습니다.

* 1. 회의 기록

오프라인 : 상허관 세미나실

* 매 주 금요일 오후 3~5시

온라인 : 행아웃, 카카오톡

* 주말 오후 및 모두가 가능한 평일 시간

5/11 ~ 5/17 : 각자 미로탐색 알고리즘에 대한 조사, 과제 제출

5/18 ~ 5/22 : 간단한 미로탐색 알고리즘 구현,

어떤 알고리즘을 활용할지 정하고 그에 맞는 역할을 분담

5/23 ~ 5/30 : 각자 구현한 알고리즘의 확인, 구현 중 막힌 부분과 개선점에 대한 토의,

발표 준비 및 보고서 작성