A 알고리즘

태그 알고리즘

현재 위치 기준으로 이동할 수 있는 방향을 조사하고

```
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <climits>

using namespace std;

namespace Astar
{
    struct Node {
        int x, y;
        int g, h, f;
        Node* parent;

        Node(int x, int y, int g = 0, int h = 0, Node* parent = nullptr)
            : x(x), y(y), g(g), h(h), f(g + h), parent(parent) {}
    };

    // 휴리스틱(맨해튼 거리) 계산
    int heuristic(int x1, int y1, int x2, int y2) {
        return abs(x1 - x2) + abs(y1 - y2);
    }

    // Node 우선순위 비교를 위한 구조체
    struct CompareNode {
        bool operator()(Node* a, Node* b) {
            return a->f > b->f;
        }
    };

    vector<Node*> getNeighbors(Node* current, const vector<vector<int>>& graph) {
        vector<Node*> neighbors;
        int dx[] = { 0, 0, 1, -1 };
        int dy[] = { 1, -1, 0, 0 };
        int n = graph.size();

        for (int i = 0; i < 4; ++i) {
            int nx = current->x + dx[i];
            int ny = current->y + dy[i];
```

```
            if (nx >= 0 && nx < n && ny >= 0 && ny < n && graph[nx][ny] != INT_MAX
                neighbors.push_back(new Node(nx, ny));
            }
        }
        return neighbors;
    }

    vector<Node*> findPath(const vector<vector<int>>& graph, int startX, int start
        int n = graph.size();
        priority_queue<Node*, vector<Node*>, CompareNode> openList;
        vector<vector<bool>> closedList(n, vector<bool>(n, false));

        Node* startNode = new Node(startX, startY);
        startNode->h = heuristic(startX, startY, goalX, goalY);
        startNode->f = startNode->g + startNode->h;
        openList.push(startNode);

        while (!openList.empty()) {
            Node* current = openList.top();
            openList.pop();

            if (current->x == goalX && current->y == goalY) {
                vector<Node*> path;
                while (current) {
                    path.push_back(current);
                    current = current->parent;
                }
                reverse(path.begin(), path.end());
                return path;
            }

            closedList[current->x][current->y] = true;
            vector<Node*> neighbors = getNeighbors(current, graph);

            for (Node* neighbor : neighbors) {
                if (closedList[neighbor->x][neighbor->y]) continue;

                neighbor->g = current->g + graph[neighbor->x][neighbor->y];
                neighbor->h = heuristic(neighbor->x, neighbor->y, goalX, goalY);
                neighbor->f = neighbor->g + neighbor->h;
                neighbor->parent = current;

                openList.push(neighbor);
            }
        }

        // 경로가 없으면 빈 벡터 반환
        return vector<Node*>();
    }

    void GetAStar()
```

```
    {
        // 4x4 그래프 생성
        vector<vector<int>> graph = {
            {1, 1, 1, 1},
            {INT_MAX, INT_MAX, 1, INT_MAX},
            {1, 1, 1, INT_MAX},
            {1, INT_MAX, 1, 1}
        };

        int startX = 0, startY = 0;
        int goalX = 3, goalY = 3;

        vector<Node*> path = findPath(graph, startX, startY, goalX, goalY);

        if (path.empty()) {
            cout << "-1 (No path found)" << endl;
        }
        else {
            cout << "Shortest Path from (" << startX << "," << startY << ") to ("
            for (auto node : path) {
                cout << "(" << node->x << "," << node->y << ") ";
            }
            cout << endl;
        }
    }
}
```

게임에서 이용 하기

0,0에서 출발해서 5,5로 도착하는 미로를 자동으로 생성하는 코드를 구현하고 싶다.

보드 판의 0,0과 5,5는 각각의 출발지와 도착지로 결정되어 있고, 그 사이에는 랜덤하게 플레이어의 이동을 막는 벽이 생성될 수 있다. 그런데, 여기서 등장할 수 있는 맵이 도착지로 도달할 수 없으면, 게임이 진행이 안되므로 다시 실행하도록 코드를 보완하였다. 위 내용을 A* 알고리즘을 이용해서 구현할 수 있겠는가?

```
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <algorithm>

using namespace std;

const int N = 5;
```

```
const int WALL = -1;
const int START = 0;
const int GOAL = N * N - 1;

struct Node {
    int x, y;
    int f, g, h;
    Node* parent;

    Node(int x, int y, int g = 0, int h = 0, Node* parent = nullptr)
        : x(x), y(y), g(g), h(h), f(g + h), parent(parent) {}
};

struct CompareNode {
    bool operator()(Node* a, Node* b) {
        return a->f > b->f;
    }
};

int heuristic(int x1, int y1, int x2, int y2) {
    return abs(x1 - x2) + abs(y1 - y2); // Manhattan distance
}

vector<vector<int>> generateRandomGrid(int n) {
    vector<vector<int>> grid(n, vector<int>(n, 0));
    srand(time(0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if ((i != 0 || j != 0) && (i != n - 1 || j != n - 1)) {
                grid[i][j] = (rand() % 3 == 0) ? WALL : 0;
            }
        }
    }
    return grid;
}

vector<Node*> getNeighbors(Node* current, vector<vector<int>>& grid) {
    vector<Node*> neighbors;
    int dx[] = { 0, 0, 1, -1 };
    int dy[] = { 1, -1, 0, 0 };

    for (int i = 0; i < 4; ++i) {
        int nx = current->x + dx[i];
        int ny = current->y + dy[i];

        if (nx >= 0 && nx < N && ny >= 0 && ny < N && grid[nx][ny] != WALL) {
            neighbors.push_back(new Node(nx, ny));
        }
    }
    return neighbors;
}
```

```
void printPath(Node* node) {
    vector<int> path;
    while (node) {
        path.push_back(node->x * N + node->y);
        node = node->parent;
    }
    reverse(path.begin(), path.end());
    for (int p : path) {
        cout << p << " ";
    }
    cout << endl;
}

int aStar(vector<vector<int>>& grid) {
    priority_queue<Node*, vector<Node*>, CompareNode> openList;
    vector<vector<bool>> closedList(N, vector<bool>(N, false));

    Node* start = new Node(0, 0);
    start->h = heuristic(0, 0, N - 1, N - 1);
    start->f = start->g + start->h;
    openList.push(start);

    while (!openList.empty()) {
        Node* current = openList.top();
        openList.pop();

        if (current->x == N - 1 && current->y == N - 1) {
            printPath(current);
            return 0;
        }

        closedList[current->x][current->y] = true;
        vector<Node*> neighbors = getNeighbors(current, grid);

        for (Node* neighbor : neighbors) {
            if (closedList[neighbor->x][neighbor->y]) continue;

            neighbor->g = current->g + 1;
            neighbor->h = heuristic(neighbor->x, neighbor->y, N - 1, N - 1);
            neighbor->f = neighbor->g + neighbor->h;
            neighbor->parent = current;

            openList.push(neighbor);
        }
    }
    return -1;
}

int Print() {
    vector<vector<int>> grid = generateRandomGrid(N);
```

```
    cout << "Generated Grid (0: Free, -1: Wall):" << endl;
    for (const auto& row : grid) {
        for (int cell : row) {
            cout << (cell == WALL ? "#" : ".") << " ";
        }
        cout << endl;
    }

    cout << "Shortest Path from Start to Goal: ";
    if (aStar(grid) == -1) {
        cout << "-1 (No path found)" << endl;
        return -1;
    }
    return 0;

}

void GetBoard()
{
    while (Print() == -1)
    {
        Print();
    }
}
```