Assignment\_1 Report

1. **코드 설명**

* 코드 실행방법
* maze\_solving() 함수 내부에 있는 file\_names 이름의 list에 실행하고자 하는 미로의 file 이름을 추가한 뒤 코드를 실행하면 됩니다.
* 코드 동작
* 미로의 최단 경로와 탐색 노드 개수는 별도의 output file에 출력됩니다

1. **함수 설명 ( g(n) : 실제 사용 비용, h(n) : heuristic 값, f(n) = g(n) + h(n) )**
   1. bfs, ids, gbfs, a\_star 모두 “시작점에서 열쇠, 열쇠에서 다음 열쇠, 열쇠에서 다음 열쇠, … , 열쇠에서 도착점”들의 각각의 최단 경로를 구하여 최종적으로 합산하는 방법을 사용하였습니다. 이때 최종 합산된 값은 모든 열쇠의 순열을 탐색하여 그 중 최소값을 사용하였습니다.
2. bfs(m, n, maze)

* queue를 이용하여 구성하였습니다. 출발점과 가까운 depth의 노드부터 queue에 삽입해 탐색하며 depth가 멀어질 때마다 이동거리를 +1 하였습니다. 탐색은 같은 level의 depth끼리 진행하였으며 목표 노드를 발견하지 못하면 다음 depth로 넘어갑니다. 최단 거리는 목표 노드의 depth가 되며 목표 노드 발견 시 trace 배열에 자신이 진행한 경로를 저장하고 실행을 종료합니다.

1. ids(m, n, maze)

* 기본적으로 dfs의 개념을 이용하기 때문에 stack을 사용하여 구성하였습니다. d\_limit 값은 최대 탐색 깊이를 나타내며 만약 정해진 d\_limit 내에서 값을 찾지 못 할 경우 d\_limit을 1씩 증가시켜주며 진행하였습니다. 현재 노드에서 갈 수 있는 노드 중 방문하지 않은 노드들의 “위치, 자신의 depth”를 stack에 삽입하며 진행합니다. 여기서 자신의 depth는 dfs이기 때문에 부모 노드 depth에 +1을 하여 구합니다. 만약 stack에서 꺼낸 노드의 depth가 d\_limit과 크거나 같다면 해당 노드는 탐색하지 않습니다. 실행 중 찾아야 하는 노드가 발견되면 trace 배열에 자신이 진행한 경로를 저장하고 실행을 종료합니다. 최단 거리는 dfs이기 때문에 찾은 노드의 depth가 됩니다.

1. gbfs(m, n, maze)

* priority queue를 사용하여 제일 낮은 h(n) 값을 갖는 노드를 우선순위로 탐색하게 구성하였습니다. 현재 노드에서 갈 수 있는 노드 중 방문하지 않은 노드들의 “h(n) 값, 위치, g(n) 값”을 priority queue에 삽입하며 진행합니다. gbfs에서 우선순위 큐에 삽입되는 g(n) 값은 자신이 움직인 거리를 확인하기 위해 사용됩니다. 실행 중 찾아야 하는 노드가 발견되면 trace 배열에 자신이 진행한 경로를 저장하고 실행을 종료합니다. 최단 거리는 g(n) 값이 됩니다.

1. a\_star(m, n, maze)

* priority queue를 사용하여 제일 낮은 f(n) 값을 갖는 노드를 우선순위로 탐색하게 구성하였습니다. 현재 노드에서 갈 수 있는 노드 중 방문하지 않은 노드들의 “f(n) 값, 위치, g(n) 값”을 priority queue에 삽입하며 진행합니다. a\_star에서 우선순위 큐에 삽입되는 g(n) 값은 자신이 움직인 거리 확인과 f(n) 값 생성을 위해 사용됩니다. 실행 중 찾아야 하는 노드가 발견되면 trace 배열에 자신이 진행한 경로를 저장하고 실행을 종료합니다. 최단 거리는 g(n) 값이 됩니다.

1. print\_func(maze, trace, algorithm\_name, ans,visit\_node\_cnt)

* 결과를 output file로 만드는 함수, 파라미터로 받은 trace를 통해 최단 경로를 output 변수에 저장한다 (output 변수는 maze의 복사본이다. trace는 maze와 동일한 크기의 list이며 최단 경로 상에 존재하는 index들은 1, 나머지는 0으로 구성되어있다. 따라서 maze와 trace를 비교하며 output 변수에 5로 최단 경로 표시를 해준다)

1. find\_info(maze)

* 입력으로 들어온 미로에서 시작점, 출구, 열쇠의 위치를 찾아 저장하는 함수, 찾아진 값들은 전역 변수인 Start, End, Key\_list에 각각 저장된다 (Key\_list 안의 값들은 열쇠를 찾는 순서이며 이는 순열로 모든 경우가 저장되어 있다 – 1, 2, 3번 위치에 열쇠가 존재하는 경우 Key\_list 안에는 [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]의 값들이 들어있다)

1. heuristic(m, n, maze)

* heuristic 값을 return하는 함수, 시작점과 도착점의 x, y 값 차이의 절댓값을 반환한다

1. maze\_solving(m, n, maze)

* main 함수에서 최초 1회 실행되는 함수, 내부의 file\_names라는 이름의 리스트에 들어있는 이름의 파일에 대해 미로 찾기를 진행하는 함수

1. **실험 결과**

* 최단경로 (length)

1. Maze\_1
   1. BFS - 4134
   2. IDS - 4134
   3. GBFS - 4134
   4. A\_star - 4134
2. Maze\_2
   1. BFS - 640
   2. IDS - 640
   3. GBFS - 640
   4. A\_star - 640
3. Maze\_3
   1. BFS - 574
   2. IDS - 574
   3. GBFS - 574
   4. A\_star - 574
4. Maze\_4
   1. BFS - 2838
   2. IDS - 2838
   3. GBFS - 2838
   4. A\_star - 2838

* 탐색한 노드의 개수 (time)

1. Maze\_1
   1. BFS - 11234
   2. IDS - 10722396
   3. GBFS - 6640
   4. A\_star - 10848
2. Maze\_2
   1. BFS - 3706
   2. IDS - 744790
   3. GBFS - 2895
   4. A\_star - 3517
3. Maze\_3
   1. BFS - 2487
   2. IDS - 289107
   3. GBFS - 1658
   4. A\_star - 2191
4. Maze\_4
   1. BFS - 17033
   2. IDS - 15004921
   3. GBFS - 12368
   4. A\_star - 16681