

Assignment #1

|  |  |
| --- | --- |
| 과목명. | 인공지능 |
| 담 당. | 최 용 석 교수님 |
| 제출일. | 2020년 10월 10일 |
| 공과대학 | 컴퓨터소프트웨어학부 |
| 학 번 | 2016025969 |
| 이 름. | 정지훈 |

****

**목차.**

**1. 코드 설명**

**- 코드 실행 방법**

**- 코드 동작**

**2. 함수 설명**

**- BFS**

**- IDS**

**- Point Class**

**- GBFS**

**- A\***

**- Backtracking**

**3. 실험 결과**

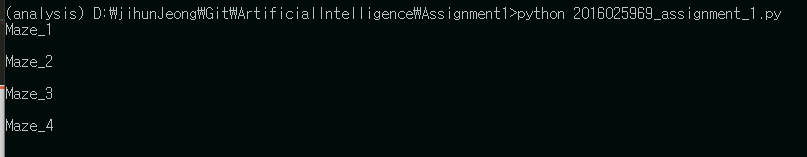
**- 최단 경로 (length)**

**- 탐색한 노드의 개수 (time)**

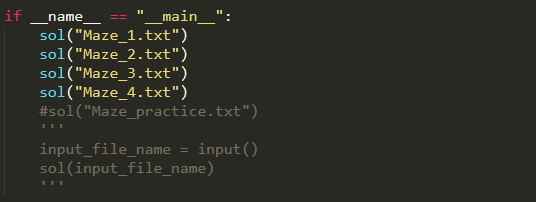
**4. 고찰**

**1. 코드 설명**

**- 코드 실행 방법**

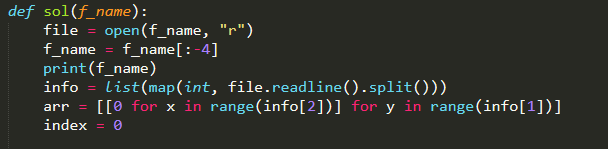
****

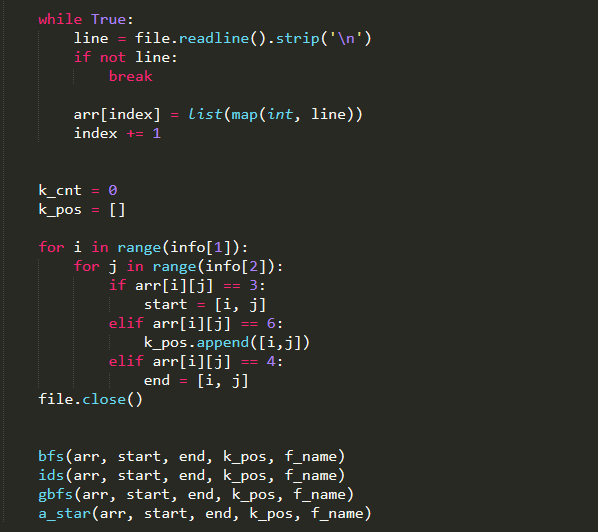
제출한 파일을 기존 compiler를 이용해서 실행시키면 기본적으로 주어진 Maze 1~4 까지의 탐색을 완료합니다.

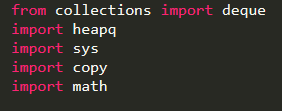


다른 미로를 탐색하고 싶을 때 주석처리한 input() 부분을 통해 원하는 미로를 탐색할 수 있습니다. 단 이때 정확한 File 명과 txt 확장자를 입력해야 합니다.

**- 코드 동작**



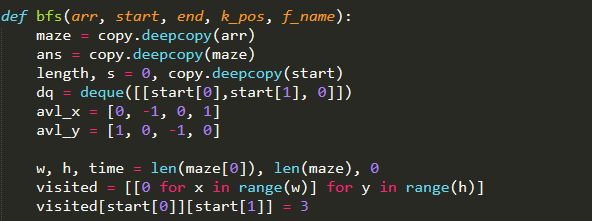
입력으로 통해 들어온 File 명을 가지고 File 입력을 진행합니다. Line으로 입력을 받으며 한 자리 숫자를 기준으로 Split 한 후 정수형으로 변환 arr 배열에 넣음으로써 미로의 정보를 받아옵니다. 미로의 정보를 한 번 훑어 시작 위치, 끝 위치, 열쇠의 위치, 미로의 이름을 4가지 미로 탐색 함수에 인자로 넘겨주어 미로 탐색을 시작합니다.



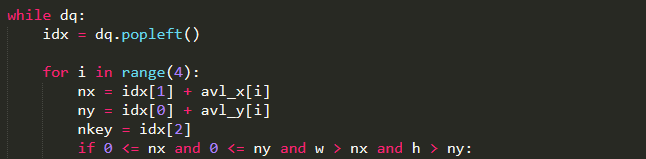
다음과 같은 내부 Python Library를 사용하였습니다

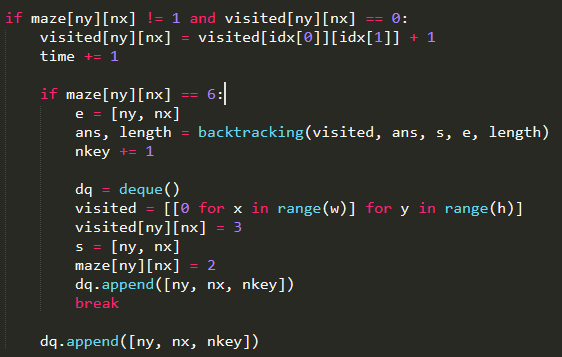
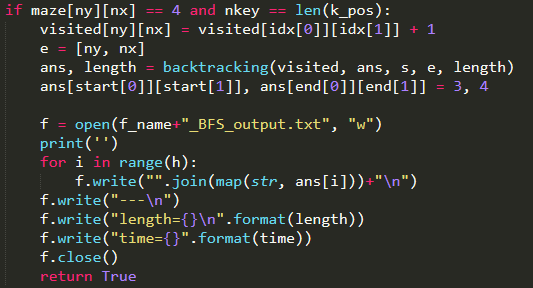
**2. 함수 설명**

**- BFS**

****

인자를 통해 들어온 미로 정보는 다음 탐색 알고리즘에서도 사용하므로 현재 쓸 미로 정보 maze와 답 ans 배열을 깊은 복사를 통해 만들어줍니다. BFS는 FIFO인 Queue를 이용하는 알고리즘이므로 Python의 Deque를 이용했습니다. 현재 위치에서 갈 수 있는 x, y 4가지의 방향을 avl\_x, avl\_y에 저장합니다. BFS 알고리즘이 돌 때 기존에 갔던 곳을 방문하지 않기 위해 방문한 정보를 담는 visited 배열을 만들어 관리합니다.

시작위치부터 dq에 정보가 있을 때까지 Bfs 탐색을 진행합니다. 새로운 ny, nx가 경계를 넘어서지 않는 다면 다음에 나오는 코드를 실행합니다.



1번 2번

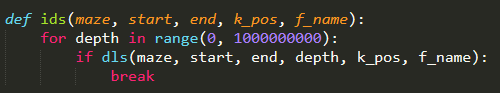
1. 모든 키를 찾고 도착점에 도착하는 경우

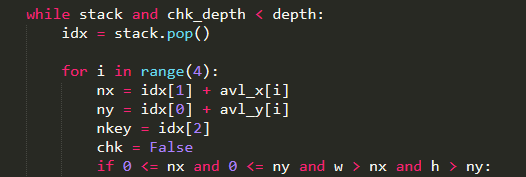
미로 탐색이 끝난 경우이므로 기존에 탐색해온 과정을 backtracking으로 최적 경로 추적, 나온 결과를 Output txt에 저장합니다.

2. 탐색하던 와중에 키를 만난 경우

BFS탐색을 하던 중 열쇠를 만난 경우입니다. Start에서 가장 가까운 열쇠를 찾게 된 것이며 이것이 최적 경로가 되고 Backtracking을 통해 해당 결과를 ans 배열에 넣어줍니다. 그 후 해당 열쇠부터 다음 열쇠 또는 도착점까지 새로운 탐색을 시작해야 하므로 열쇠의 개수를 1개 증가시킨 상태로 새로운 경로를 담기 위해 visited 배열과 dq 배열을 초기화한채 해당 열쇠 위치는 단순 경로 값을 가지며 새로운 start가 됩니다.

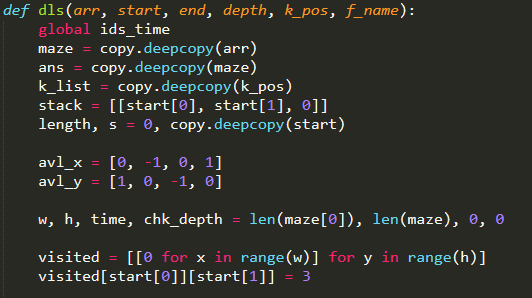
**- IDS**

Limit depth까지만 탐색하는 DLS 함수를 for 문을 통해 iterative 하게 depth 값을 넣어줍니다. 탐색 성공했을 시 DLS에서 True 값이 반환이 되며 실패 시 False 값이 반환이 됩니다.



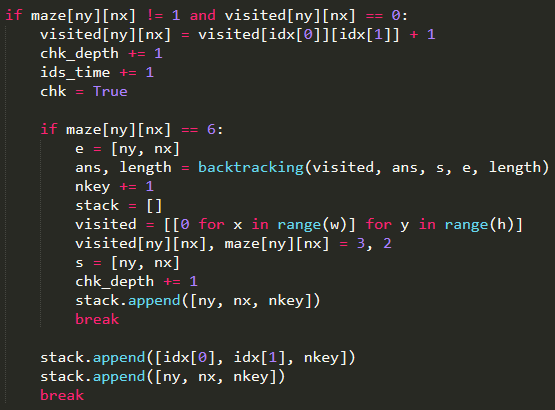
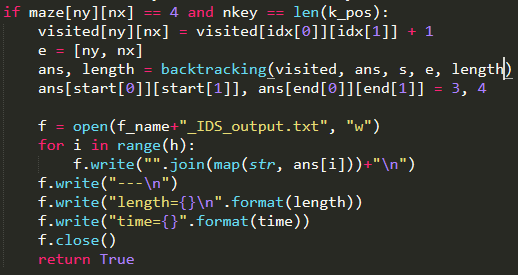
1번

2번



1. DLS는 기본적으로 depth가 limit이 있는 DFS입니다. 따라서 DFS를 사용하기 위해 LIFO인 stack을 이용했습니다.

2. Depth를 관리할 때 현재 노드에서 expand 되는 지 안 되는 지를 파악하는 bool 함수를 만들어줍니다. 인자로 받아온 depth 값을 넘지 않기 위해 조건문에 chk\_depth < depth를 넣어줍니다.



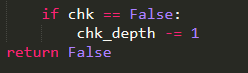
1번 2번

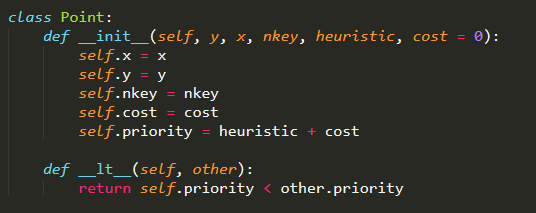
1. 모든 키를 찾고 도착점에 도착하는 경우

BFS와 마찬가지로 최종 결과를 출력합니다. 그후 True를 반환해 더 이상 DLS가 실행되지 않도록 합니다.

2. 탐색하던 와중에 키를 만난 경우

새로운 위치로 Expand 되는 경우 Chk는 True가 되며 depth가 1이 증가하게 됩니다. 만약 새로운 위치에 열쇠가 있는 경우 마찬가지로 그 위치에서부터 새로운 탐색을 시작한다고 생각해서 기존에 방문했던 경로를 Backtracking에 저장하고 새로운 저장 경로를 담을 visited를 만듭니다.

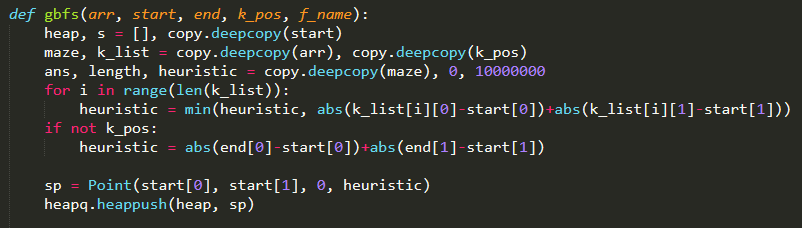
새로운 위치가 Expand 되지 않았다면 돌아오게 되므로 depth를 1 감소시켜 줍니다. Limit depth에 의해 성공하지 못한 탐색은 False를 반환해 현지 depth+1의 DLS가 실행될 수 있도록 합니다.

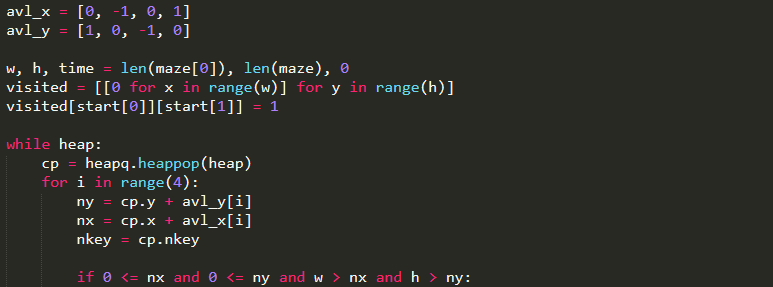
**- Point Class**

Heuristic 정보를 통해 탐색하는 GBFS와 A\*를 위해 해당 위치의 점이 가지게 되는 정보를 담는 Class를 만들었습니다.

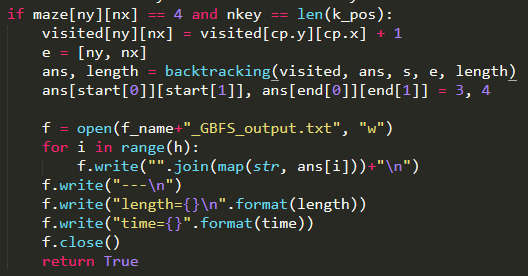
\_\_lt\_\_ Function을 통해 나중에 heap에서 우선순위를 비교할 때 위치의 priority를 가지고 비교할 수 있도록 하였습니다.

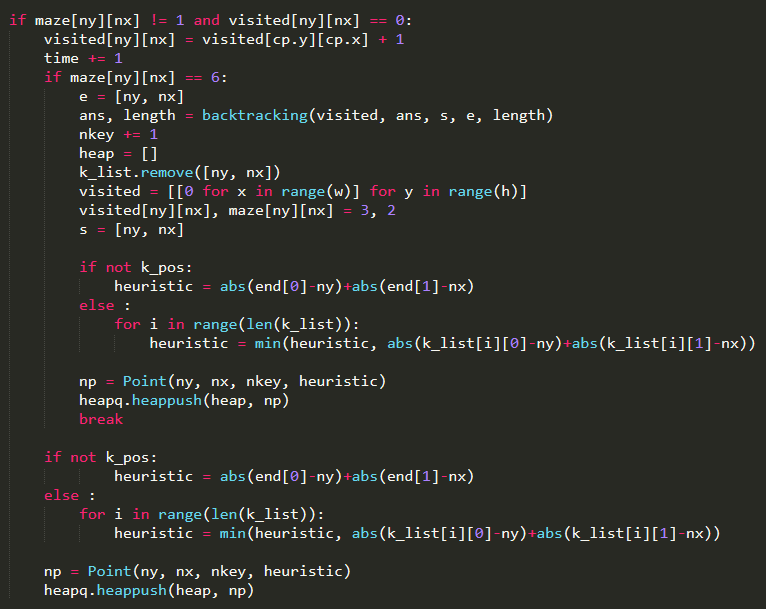
**- GBFS**



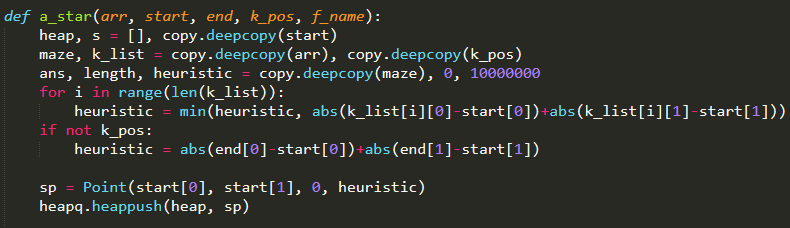
****현재 위치에서 다음 열쇠 까지의 거리를 정보를 가지는 GBFS 탐색입니다. 현재부터 모든 열쇠까지의 거리 중 가장 가까운 거리부터 탐색해 나가는 방법입니다. 가까운 거리부터 탐색해야 하기에 heapq 모듈에 있는 Min-heap을 사용해서 하였습니다. 만약 탐색할 열쇠가 없다면 End 좌표를 가지고 heuristic을 설정하였습니다. 이렇게 얻어진 정보를 가지고 Point 객체를 만들어 heap에 넣습니다. Heuristic을 비교할 때 Class에 cost 값을 주지 않아 cost는 초기값 0으로 설정이 되어 priority 비교에 영향을 받지 않게 됩니다.

Heap을 통해 heuristic 값이 가장 작은 좌표를 얻어 옵니다.

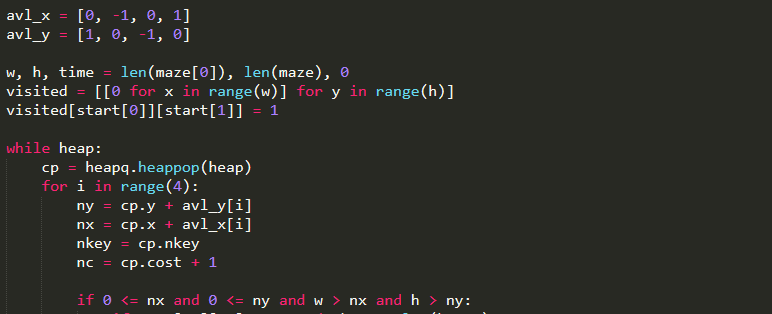
앞선 탐색과 마찬가지로 모든 열쇠를 찾고 도착점에 도착했다면 최적경로를 결과에 넣어주고 True를 반환합니다.

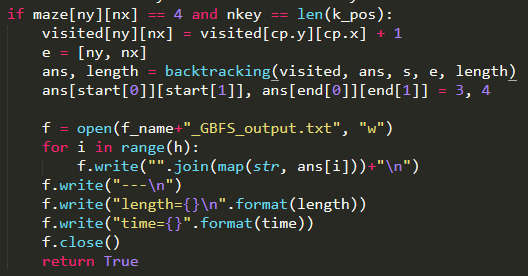
****heap에서 얻어진 좌표를 가지고 탐색을 합니다. 다음 위치에서 남아 있는 모든 열쇠와의 heuristic을 계산해 그 중 가장 값이 적은 것을 설정해 새로운 Point 객체를 만들고 heap에 넣어줍니다. 만약 새로운 위치에 열쇠가 있다면 마찬가지로 지금까지의 최적 경로를 Backtracking을 통해 해당 경로를 찾고 새로운 위치부터 다시 GBFS를 실행합니다.

**- A\***



GBFS와 heuristic 정보는 동일하지만 이후 몇 칸을 갔는가에 대한 정보도 함께 가지게 되어 두개의 정보 합의 최소인 좌표를 탐색하는 A\* 탐색입니다. 현재 시작 위치이기에 Point 객체를 만들 때 g(n) = 0 이므로 정보를 넘겨주지 않았습니다.

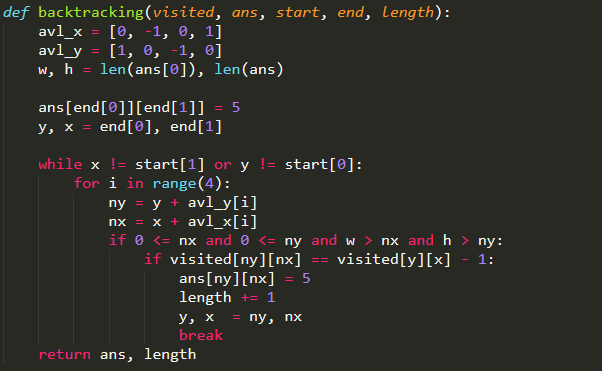
****

heap에서 좌표를 받은 뒤에 갈 수 있는 좌표에 대한 새로운 정보를 입력합니다. 이때 새로운 좌표는 기존 좌표에서 한 칸 이동한 것이기에 nc = cp.cost + 1을 통해 값이 증가되는 것을 볼 수 있습니다.

출력부분은 앞선 다른 탐색 방법과 동일합니다.

GBFS와 같이 heap에서 얻어진 좌표를 가지고 탐색을 합니다. 다음 위치에서 남아 있는 모든 열쇠와의 heuristic을 계산해 그 중 가장 값이 적은 것을 설정해 새로운 Point 객체를 만들고 heap에 넣어줍니다. 또한 기존에 있던 cost에서 새로운 위치로 왔기에 증가한 nc도 같이 객체에 넘겨줍니다. 만약 새로운 위치에 열쇠가 있다면 마찬가지로 지금까지의 최적 경로를 Backtracking을 통해 해당 경로를 찾고 새로운 위치부터 다시 GBFS를 실행합니다

**- Backtracking**



탐색 알고리즘을 통해 start에서 목표한 end 점을 갔다면 그 두 점사이의 최적의 경로를 찾아 ans 배열에 최적 경로 표시인 5의 값을 넣어줍니다. 경로를 찾는 방법은 시작점부터 끝점까지의 경로가 visited 배열에 값이 1씩 증가되면서 저장이 되던 것을 끝 점부터 while을 통해 현재 위치보다 다음 위치의 visited 값이 -1 되는 방향으로 찾아 시작점까지 역순으로 찾는 것입니다.

**3. 실험 결과**

**- 최단 경로 (length)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BFS | IDS | GBFS | A\* |
| Maze\_1 | 4134 | 4134 | 4134 | 4134 |
| Maze\_2 | 640 | 640 | 640 | 640 |
| Maze\_3 | 574 | 574 | 574 | 574 |
| Maze\_4 | 2838 | 2838 | 2838 | 2838 |

**- 탐색한 노드의 개수 (time)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BFS | IDS | GBFS | A\* |
| Maze\_1 | 11236 | 12972703 | 11714 | 11234 |
| Maze\_2 | 1200 | 758555 | 1177 | 1198 |
| Maze\_3 | 1023 | 234223 | 1001 | 1025 |
| Maze\_4 | 6631 | 6985800 | 6411 | 6631 |

**- 최단 경로 지도 (map)**

(첨부 파일 참조)

**4. 고찰**

미로 탐색을 BFS, IDS, GBFS 그리고 A\*를 통해 구현해보았습니다. 처음 열쇠를 찾는 것에 대해 시작점부터 시작해 가까운 열쇠를 찾아 나가는 것이 답이라고 생각하였으나 이는 열쇠가 한 개 있을 때의 경우이며 열쇠가 많은 경우 Greedy 적인 성격이 있기에 결국 열쇠 간의 모든 거리와 열쇠와 시작점, 끝 점 까지의 거리를 구해 시작부터 열쇠 그리고 도착점 까지의 모든 순열의 경우의 수를 가지는 경우 중에 가장 작은 값을 가지는 것이 최적이라는 결론에 도달하였습니다. 하지만 이는 시작 위치부터 시작해 나간다는 미로의 특성과 맞지 않는다고 판단하여 고려하지 않았습니다. 또한 실생활에서 거리가 크게 줄어드는 도시의 경우 BFS에 비해 A\*가 실용적으로 탐색의 경우가 줄어들 것이지만 미로에서는 모든 칸이 1이기에 이동한 거리의 g(n)과 heuristic value의 합이 크게 변하지 않아 유의미한 time의 감소를 보지 못했습니다. 탐색 알고리즘을 짜면서 수업에서 배운 내용을 더 잘 이해하는 계기가 되었습니다.