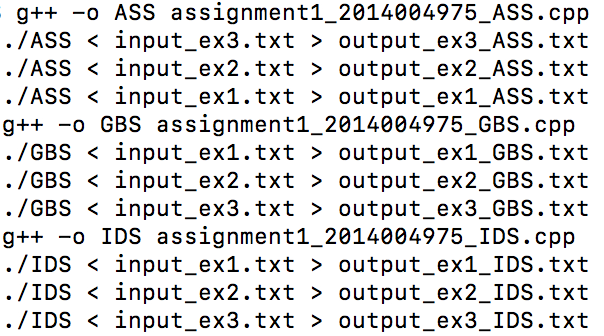
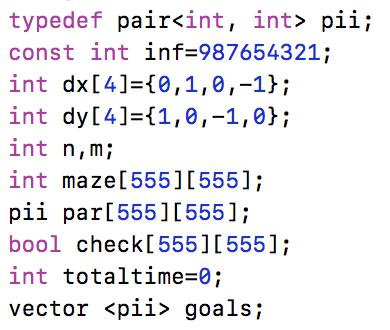
**AI Assignment #1 \_ Search Algorithms**

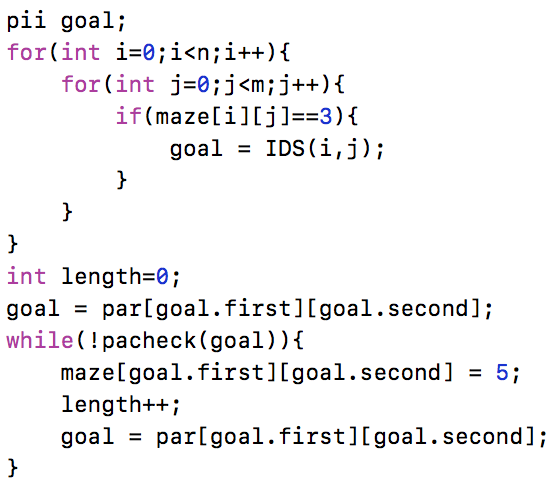
2014004975 이윤성

1. 컴파일 방법 및 사용 버전
   1. 컴파일 방법
      1. 
   2. 사용 버전
      1. OS
         1. macOS High Sierra 10.13.4
      2. g++ version
         1. Apple LLVM version 9.1.0 (clang-902.0.39.1)
         2. Target: x86\_64-apple-darwin17.5.0
         3. Thread model: posix
         4. gcc 4.2.1
2. 코드 설명
   1. 공통
      1. declaration



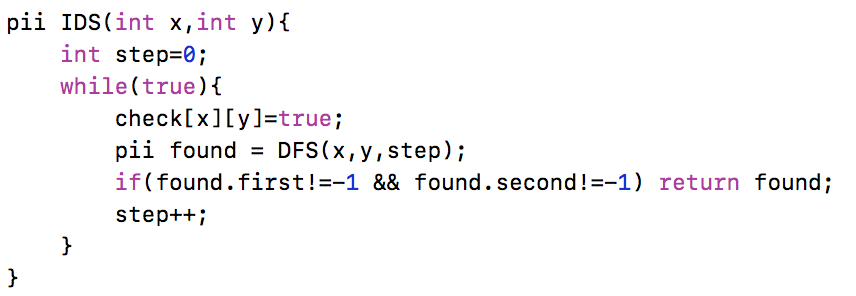
dx,dy배열은 현재 노드에서 다음노드로 방문하는 4방향을 나타내는 배열이다. 오른쪽부터 시계 방향으로 방문한다. inf값은 500\*500보다 큰 값으로 정한다.

* + 1. back tracking



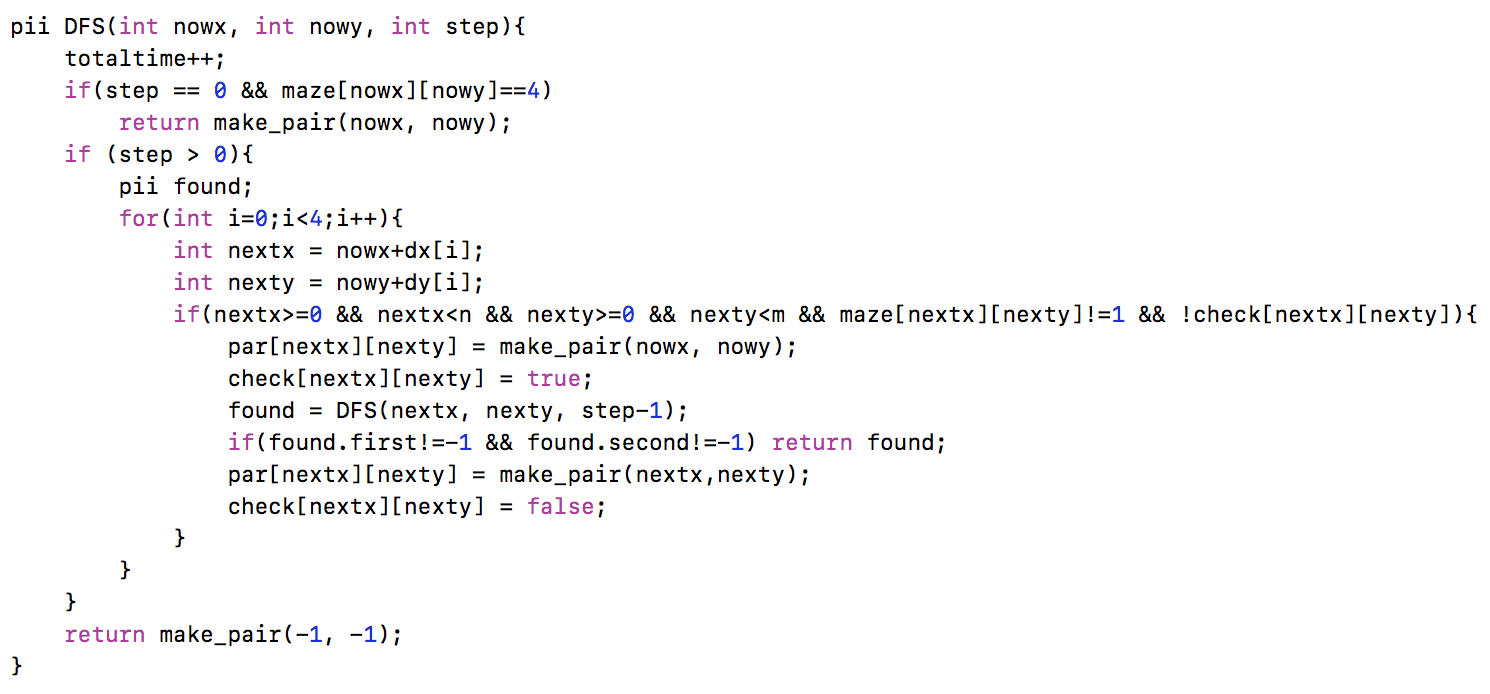
IDS를 통해 최종 도착지를 return받아 도착지에서 부터 back tracking을 하며 path들을 5로 바꿔준다. 이때, 출발점과 도착점은 포함하지 않도록 한다.

* 1. Iterative Deepening Search (IDS)
     1. IDS function



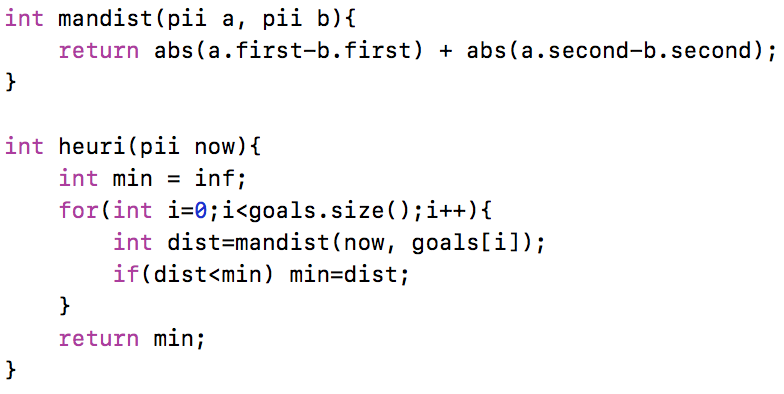
도착지에 도착 가능한 DFS를 완료할 때까지 DFS의 step수의 limit을 1씩 늘려주는 코드이다. pii 는 pair <int, int> 형으로 도착점에 도달하면 도착점의 좌표를 return한다. 도착점에 도달하지 못할 경우 (-1,-1)을 리턴한다.

* + 1. DFS function



도착점에 도달한경우 (step==0 && maze[nowx][nowy]==4)인 경우 그 때의 좌표를 리턴해 준다. totaltime은 노드 하나하나를 탐색할 때마다 1씩 증가시켜준다. par[i][j]는 (i,j)위치는 어떤 좌표를 부모로 갖는지를 나타내는 배열로 나중에 back tracking에 사용된다. check[i][j]는 (i,j) 위치를 이번 dfs에서 도달한 적이 있는지를 확인하는 배열이다.

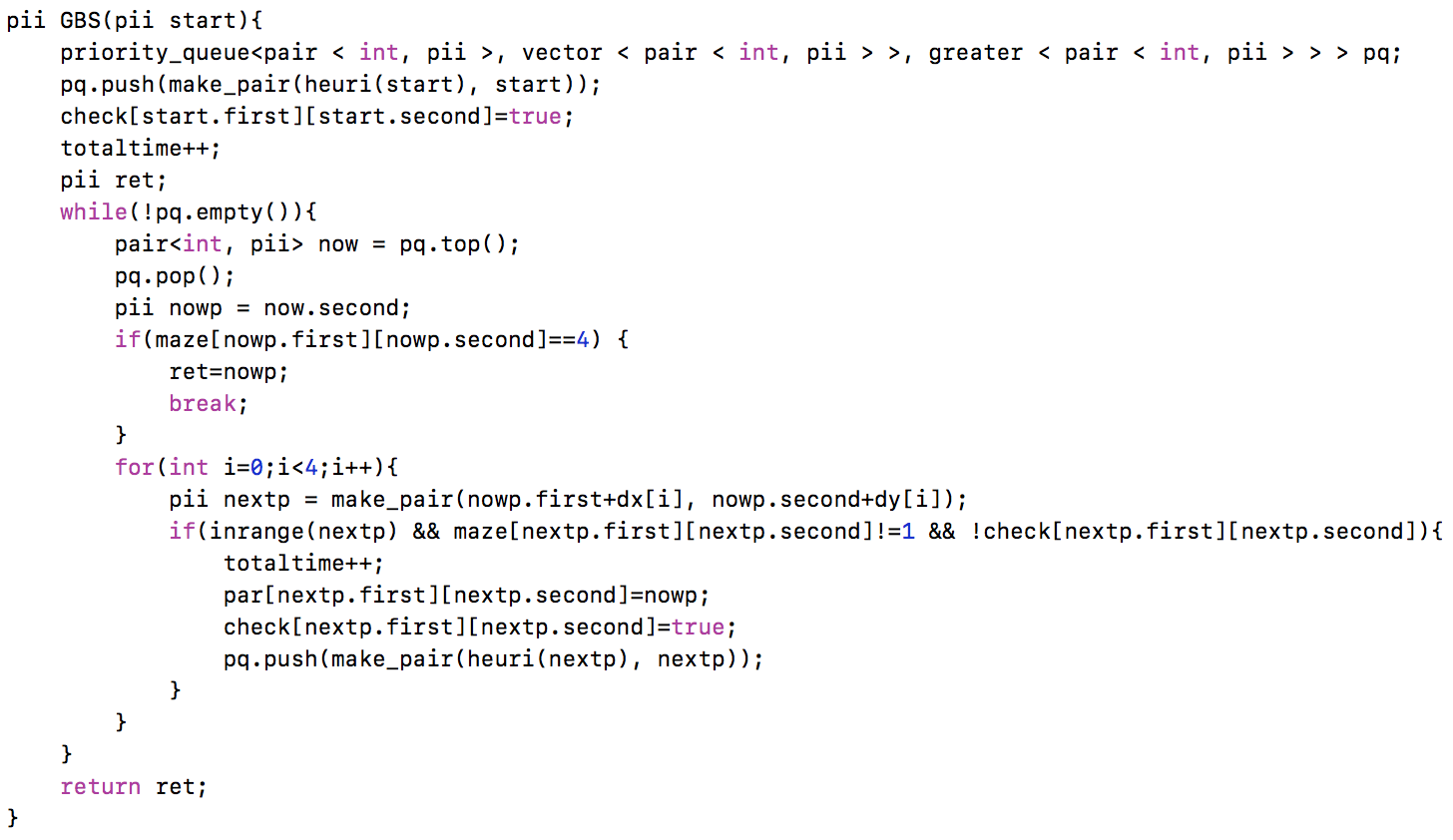
* 1. Greedy Best-first Search (GBS)
     1. heuristic function



현재 위치에서 도착점까지의 거리(manhattan distance)들 중 가장 작은 것을 heuristic function으로 선정한다. goals vector에는 값이 4인 지점, 즉 도착점인 지점들이 들어있다.

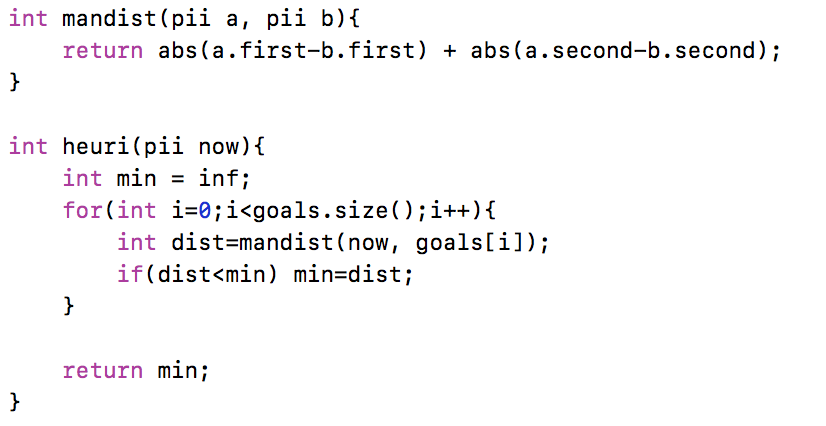
이렇게 heuristic function을 정한 이유는 이후에 사용할 A\*search 에서도 사용가능한 Admissible heuristic 이기 때문이다. (항상 h(n)<=h\*(n)이다.)

* + 1. GBS function



priority\_queue는 현재 선택가능한 노드들 중, heuristic 값이 가장 작은 노드를 방문할 수 있도록 해주는 min\_heap기반의 자료구조이다. priority\_queue에 heuristic값을 기준으로 값이 들어갈 수 있도록 넣어주며 gbs를 진행한다. pq에 새로운 노드를 넣을 때마다 totaltime을 증가시켜준다.

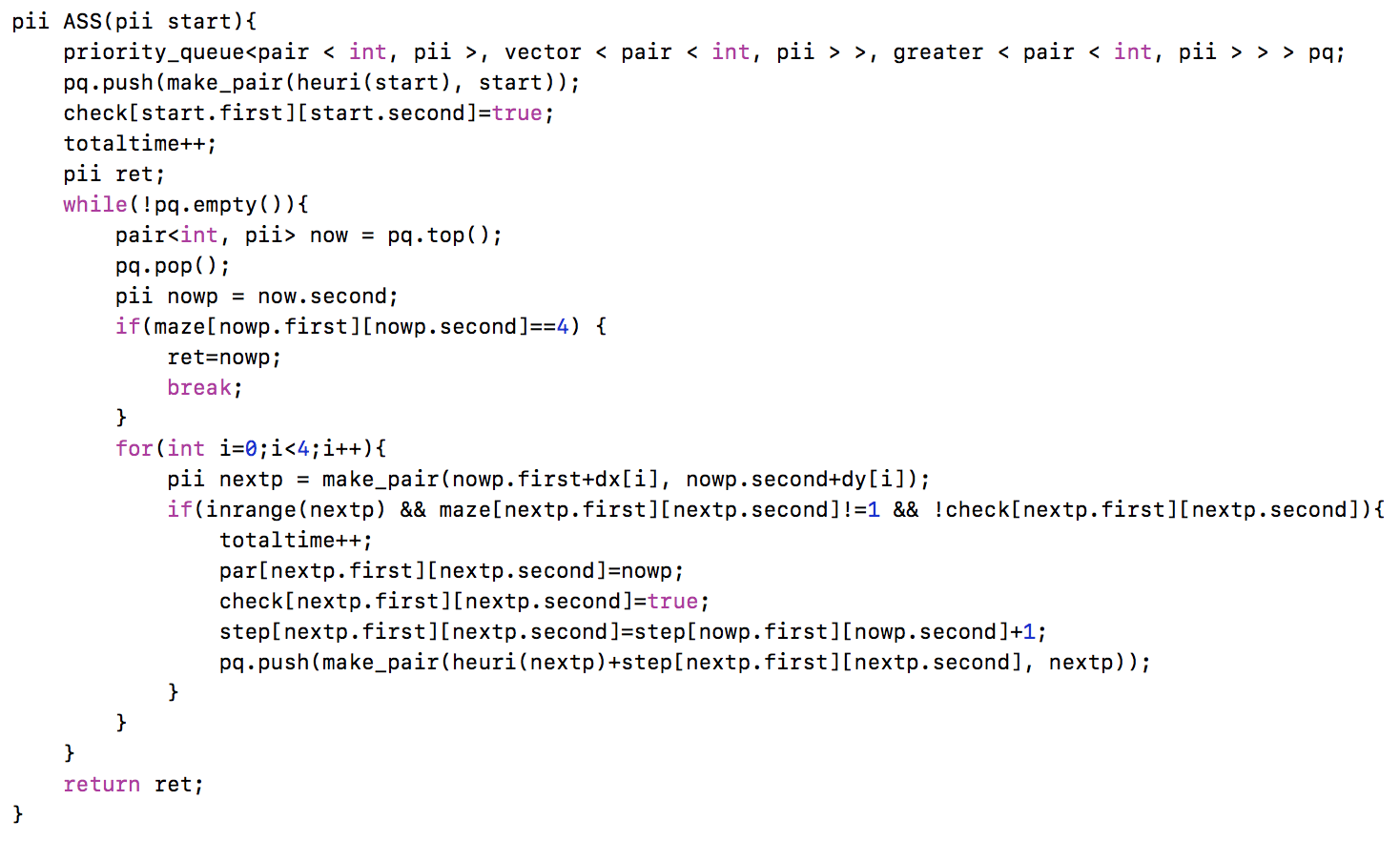
* 1. A\* Search (ASS)
     1. heuristic function



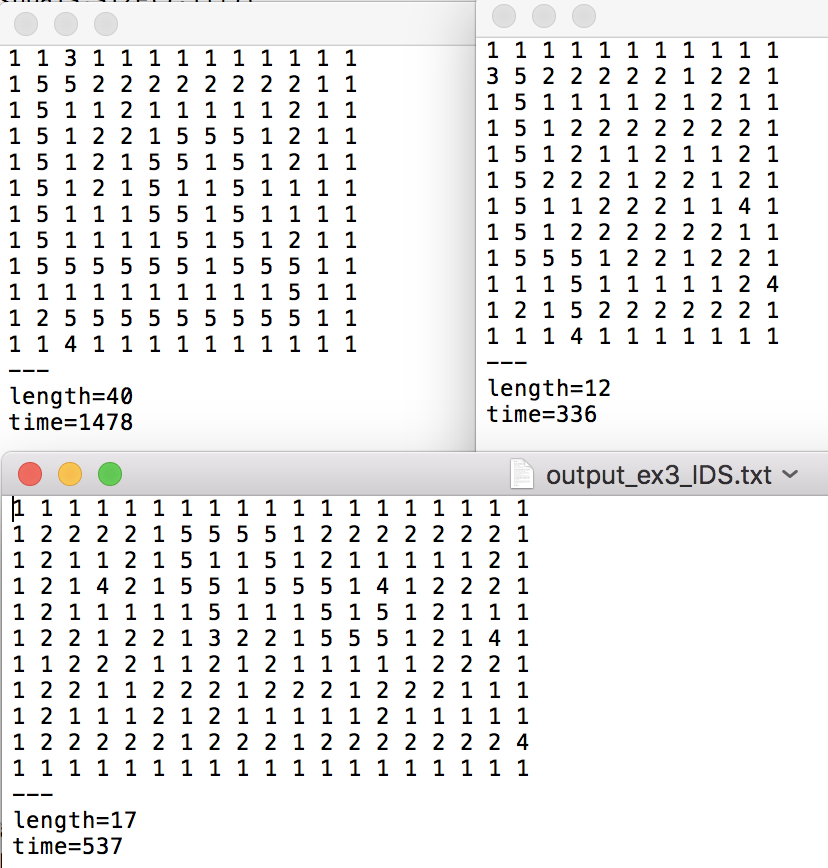
이번에도 현재 위치에서 도착점까지의 거리(manhattan distance)들 중 가장 작은 것을 heuristic function으로 선정한다.

이렇게 heuristic function을 정한 이유는 Admissible heuristic 이기 때문이다. 즉, 항상 h(n)<=h\*(n)이다. 그 이유는 미로의 형태가 어떻든지 간에 도착점들까지의 맨하탄 거리들 중 최소보다 더 짧은 경로로 도착점에 도달할 수 없기 때문이다.

* + 1. ASS function



이번에도 priority\_queue를 이용하는 것은 GBS와 같다. 허나 이번엔 heuristic 값으로만 우선순위를 정하는 것이아니라 현재까지 진행해온 step수를 합해서 (heuri(nexp)+step[nextp.first][nextp.secon]) 우선순위를 정한다. step[i][j]는 출발점에서부터 (i,j)에 도달하기까지 걸린 step의 수를 의미한다.

1. 실험 결과
   1. IDSㅂ

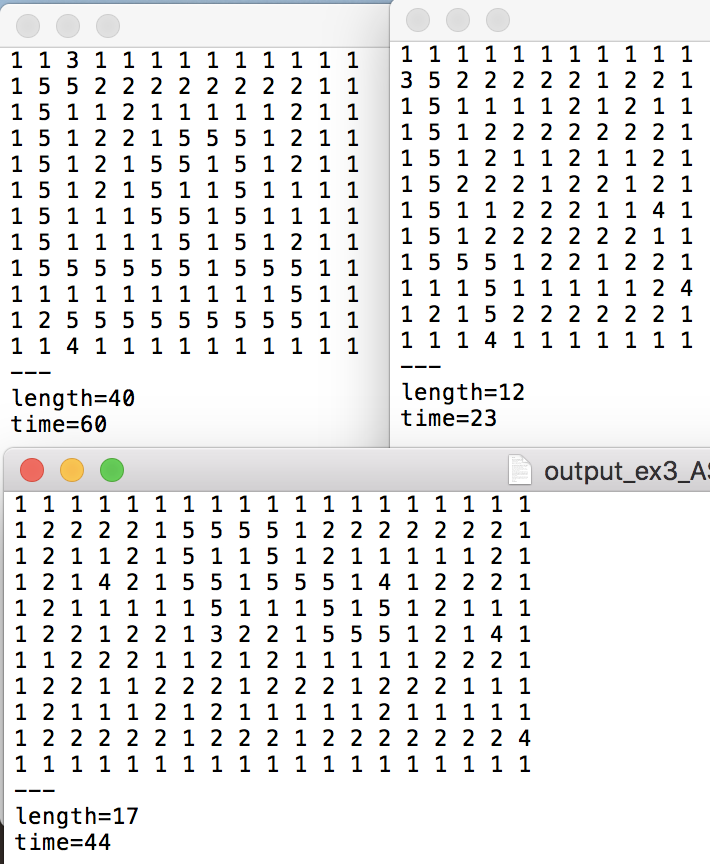
사진을 통해 볼 수 있듯이, path-cost가 non decreasing 하므로 optimal한 해를 항상 구한다. 하지만, 시간이 꽤 오래걸린 것을 볼 수 있다.

* 1. GBS



GBS의 time complexity도 worst case에서는 exponential 하지만, in practice에서는 IDS에 비해 크게 향상된 것을 볼 수 있다. 하지만 optimal한 해를 찾지 못하기도 한다.

* 1. ASS



time도 GBS에 근접할 정도로 우수하고 optimal한 해도 항상 구한다.

* 1. Conclusion

ASS가 in practice에서 optimal도 만족하면서 time역시도 우수하게 나왔다. 하지만, ASS는 space complexity 면에서 worst case에서 exponential 하므로, 큰 memory를 요구하는 문제상황에서는 optimal도 만족하면서, 선형크기의 memory만을 필요로하는 IDS를 사용해야할 수도 있다.

1. 소감

평소 algorithm PS를 자주하면서 search문제가 나오면 단순한 bfs만을 사용하면서 풀었는데, 적절한 heuristic을 구할 수 있는 상황이라면 A\* search를 이용해서도 풀어보고 싶어졌다. 또한 그간 많이 사용해왔던 dijkstra algorithm도 search algorithm들에서 응용한 부분들이 보였다. 최근 Deep Learning 학습에 신경쓰느라 많이 놓고 있었던 PS 공부도 되었던 것 같아 뜻깊은 시간이었다.