Homework#1 Report

Introduction to Artificial Intelligence
41 분반
2018312280 이상수
2023/3/16

Uniform Cost Search 구현

UCS2.py 라는 파일이 메인 실행 파일입니다.

가장 중요한 알고리즘이 들어간 파일 **solver.py** 안에 있으며, Class UCS2 안에 **def uniform_cost_search** 가 존재합니다.
이 때 인접 노드들을 찾아주는 함수는 maze.py 라는 파일 안에 존재합니다. 함수명은 find_neighbours3 입니다.

Current_level (현재 트리에 있는 노드), neighbour_coors (neighbour search 함수에서 받아온 인접 노드들) 처럼 원래 미로의 지점들의 x,y 좌표 값만 들어가 있던 list 들에 3 번째 cost 값을 추가하였다는 것입니다. (x, y, cost)
그래서 각 지점마다 시작점으로부터 거리인 cost 를 항상 값으로 가지고 있습니다.

이를 이용하여 매번 현재 지점에서 새로운 지점으로 이동할 때 current_level 리스트 안에 값들 중에 가장 cost 가 작은 값으로 먼저 이동합니다.
이 때 cost 가 같은 값이 있을 경우 breath_first_search 기준을 따라 먼저 이동합니다. 이는 항상 목표좌표에 도착하면 인접 노드들을 다 current_level 에 넣고 이동하기 때문입니다. 그리고 이렇게 이동할 때마다 path 리스트에 좌표와 cost 를 같이 추가합니다.

이동하면 이동한 지점에서 인접한 새로운 노드들의 위치를 find_neighbours3 함수에서 파악하고, 사이에 벽이 있는지, 이미 가본 지점은 아닌지 확인한 후, 이동한 지점의 cost 를 그 노드들에 +1 하여 neighbour_coors 에 추가합니다. 이렇게 하면 새로 찾은 노드들도 cost 가 1 씩 추가된 채로 생성되어 언제든지 cost 를 좌표만 알면 확인할 수 있습니다.

이 때 breath_first_search 를 기준으로 하다 보니 매번 분기점이 나올 때마다 양쪽 노드의 경로를 계속 왔다갔다 하며 cost 에 따라 이동을 합니다. 이 때 왔다갔다 할 때 back track 을 해야합니다.
Back track 의 조건은 다음 목표지점이 현재지점에서 1 보다 많이 떨어져 있거나, 둘 사이가 벽으로 막혀 있을 때 back track 을 합니다. 이외에는 그냥 가도 갈수 있는 1 거리의 막히지 않은 지점입니다.
Back track 은 우선 출발하는 곳을 a, 도착해야 하는 곳을 b 라고 하겠습니다.
아래 그림을 통해 설명하겠습니다.

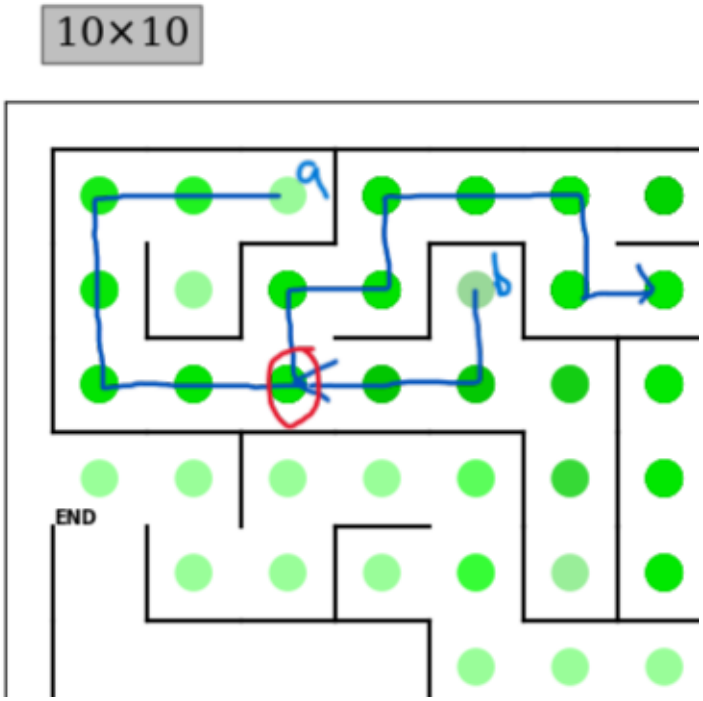


Figure 1. 미로 예시

a 에서 출발하여 start 지점까지 가는 경로를 list 로 만듭니다. 그 다음 b 에서 출발하여 start 까지 가는 경로를 list 로 만들기 시작합니다. 이 때 b 에서 시작해서 만들어지는 경로는 한점 한점 갈 때마다 a 의 list 와 겹치는지 확인합니다. 겹치는 지점 (그림에서 빨간지점)을 찾으면 거기서 멈추고, list 를 뒤집고 list[0]을 뺍니다. 그후 a 의 list 중 빨간지점 보다 start 에서 거리가 가까운 점들(a->빨간점 이외의 좌표)는 전부 뺍니다. 그 다음 a 의 경로를 path 에 추가하고, b 의 경로를 path 에 추가하면 a->b 까지 가는 경로가 path 에 저장됩니다. 그러다가 exit_coor 에 도달하면 path 를 return 합니다. 코드상에는 end 에서 start 까지 가는 경로도 저장해둘 가능성을 위해 opt_path 에 그 특정 경로만 저장하는 코드가 있긴 하지만, 마지막에 clear 시키고 다시 path 를 opt_path 에 그대로 넣어서 return 하며 동시에 len(path)로 코스트도 return 합니다.

이 외에도 maze_manager.py 에 UCS2 class 를 실행되게 하기 위해 UCS2 를 import 하고 solve_maze 함수에도 method 중 하나로 추가하였습니다.

A_Star_Search 구현

거의 UCS 알고리즘을 구현한 것을 그대로 복사해서 사용하였습니다.
실행 파일은 A_search.py 를 실행하면 됩니다.

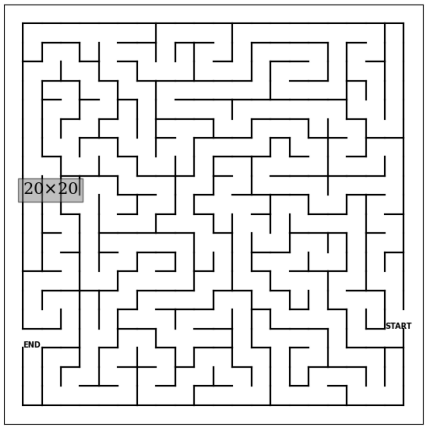
UCS 와 마찬가지로 solver.py 에 A_search class 밑에 a_star_search 함수로 구현되었고, 인근 좌표를 찾는 함수는 똑같이 find_neighbours3 를 이용하였습니다.

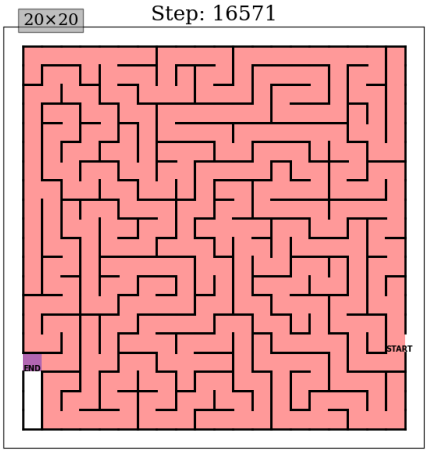
유클리드 함수는 maze_manager.py 파일에 solve_maze 위에 작성하였습니다. 이름은 euclid()입니다. 유클리드 함수 euclid(self, ax, ay, dx, dy)는 ax, ay 는 현재 좌표, dx, dy 는 도착지점 좌표를 받아서 둘의 직선 거리를 계산해 return 합니다.

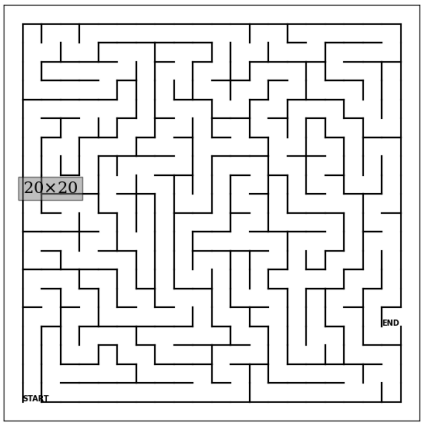
이미 시작점에서 좌표까지 거리는 cost 로 좌표마다 저장되기 때문에, 좌표를 받으면 앞서 말한 current_level 리스트에서 다음으로 갈 목적지 좌표를 정할 때, 각 좌표마다 heuristic 함수로(여기서는 유클리드) 계산하여 cost 끼리 비교할 때 더해서 비교합니다. 그렇게 총합 cost 가 작은 점이 목적지가 되고, current_level 리스트에 더 이상 확인할 좌표가 없을 때까지 반복하여 가장 cost 합이 적은 좌표로 이동합니다. 나머지 동작은 UCS 와 동일합니다.

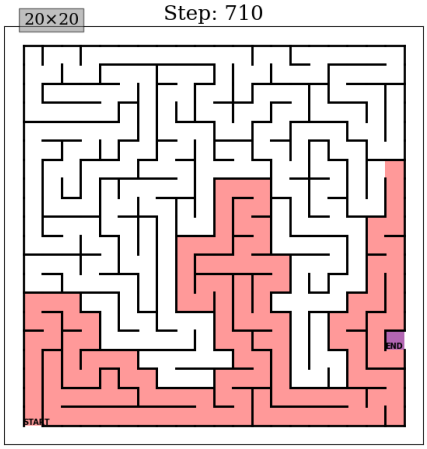
이 외에도 UCS 와 마찬가지로 maze_manager.py 에 A_search class 를 실행되게 하기 위해 A_search 를 import 하고 solve_maze 함수에도 method 중 하나로 추가하였습니다.

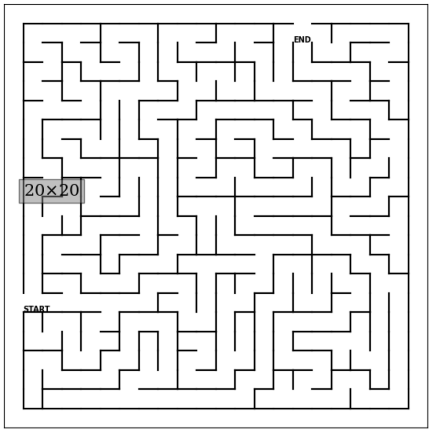
Uniform cost search 20x20 10 회 결과

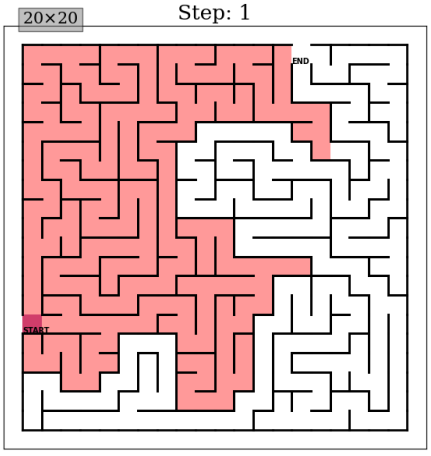


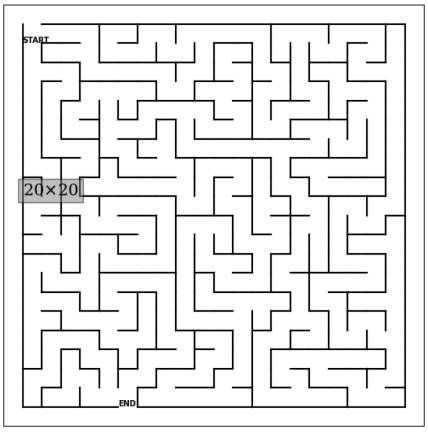


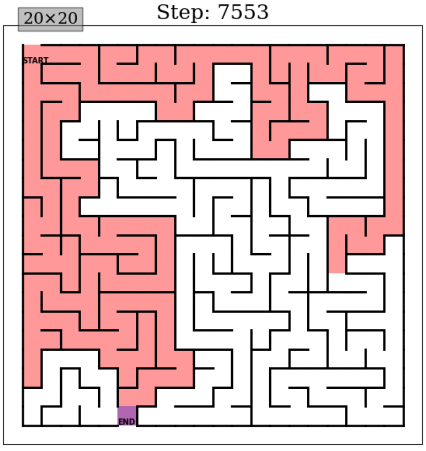


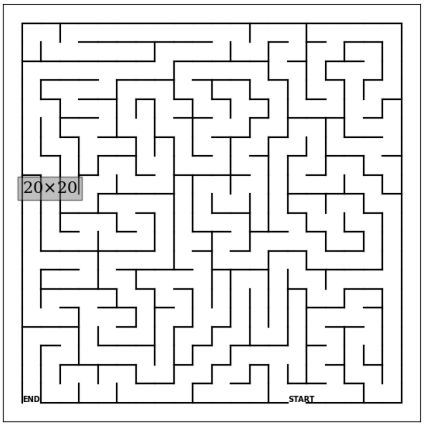


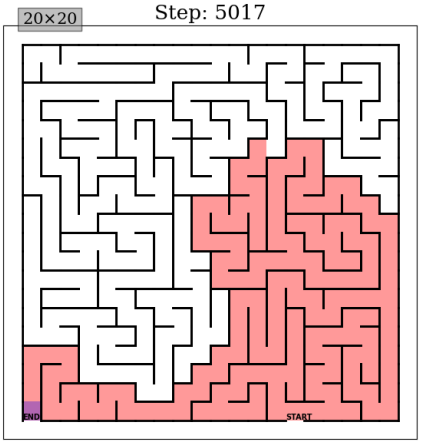


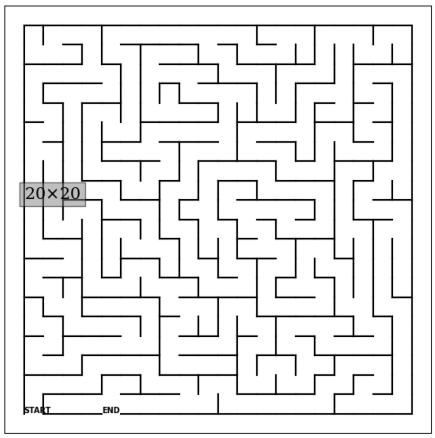


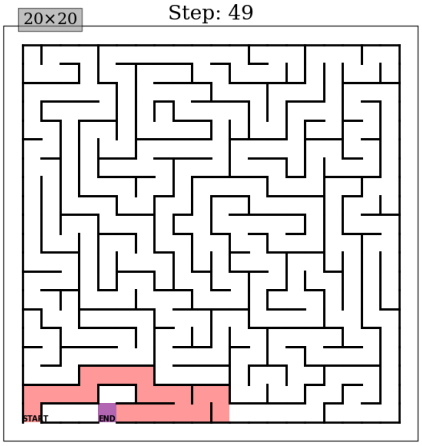

START
20×20
END

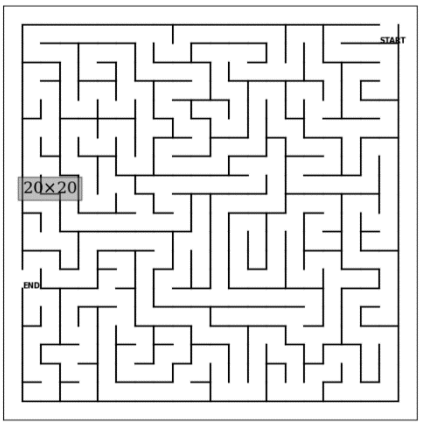

20×20
Step: 7553
START
END

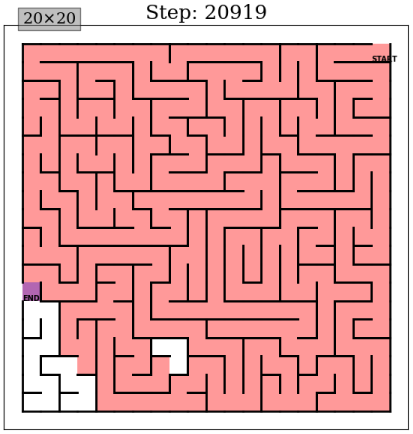

20×20
END
START

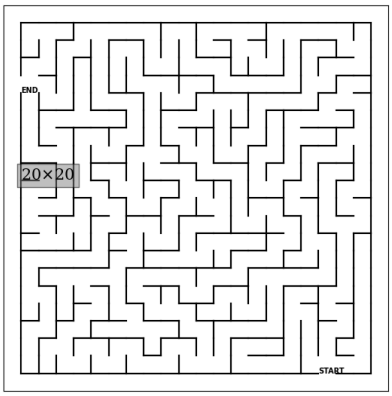

20×20
Step: 5017
END
START

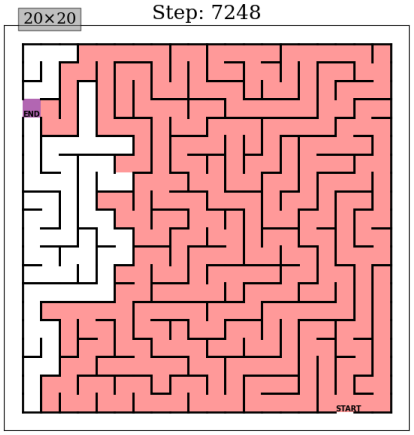

20×20
START
END

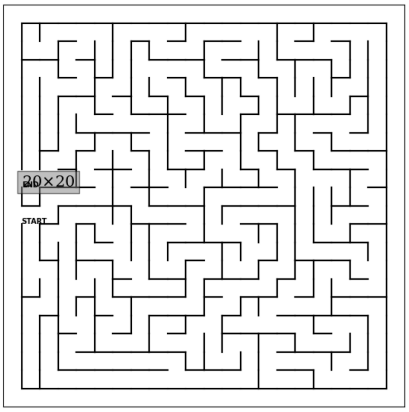

20×20
Step: 49
START
END

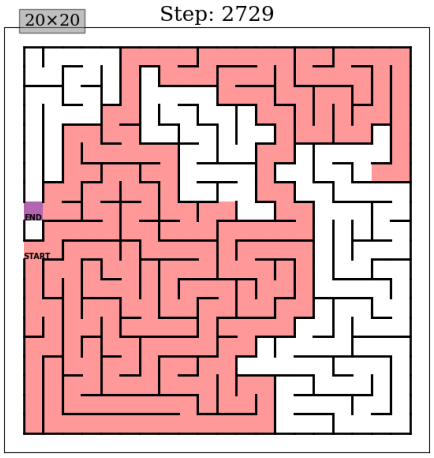

START
20×20
END

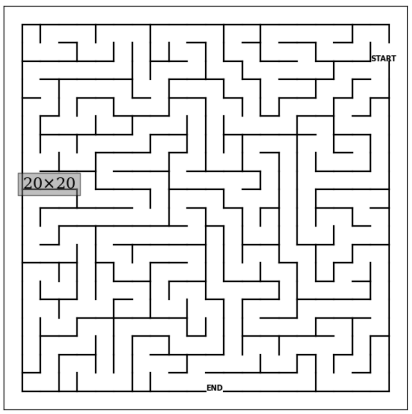

20×20
Step: 20919
START
END

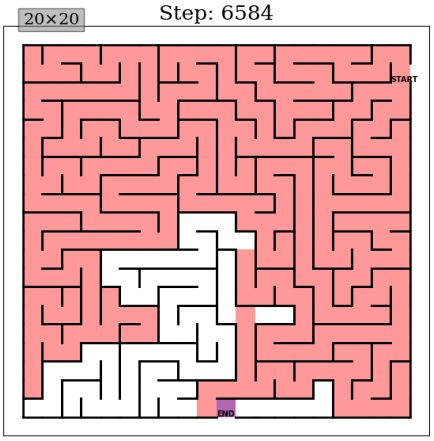

END
20×20
START


20×20
Step: 7248
END
START

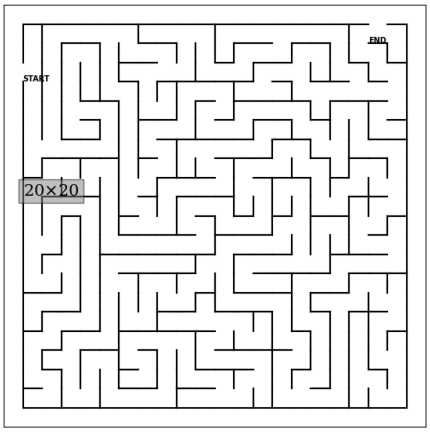

20×20
END
START

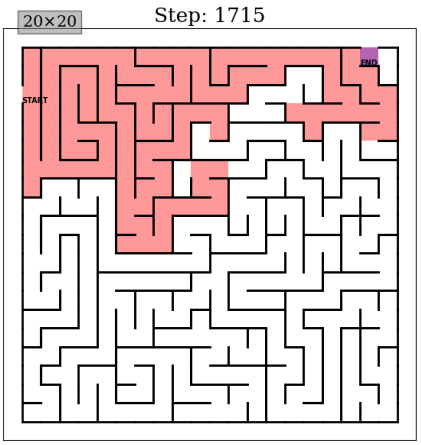

20×20
Step: 2729
END
START

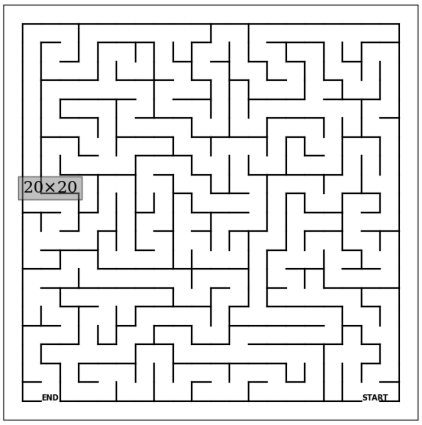

START
20×20
END

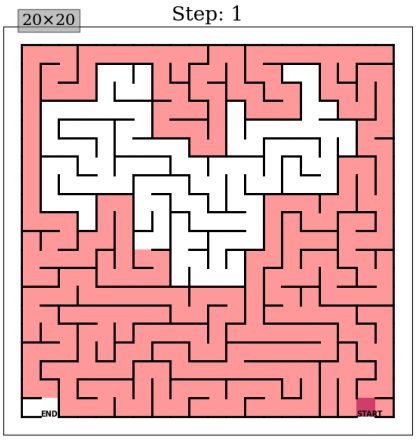

20×20
Step: 6584
START
END

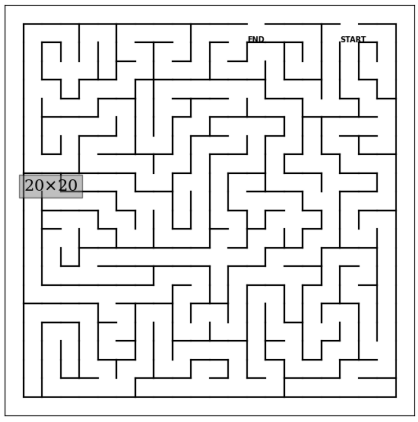
A* search 20x20 10 회 결과

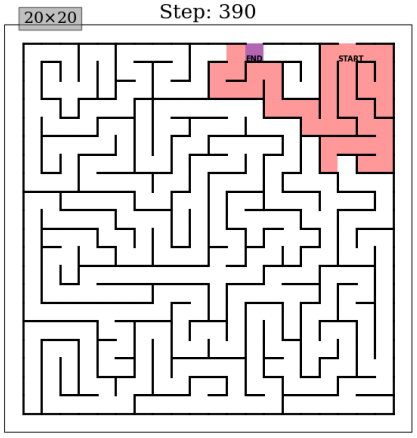


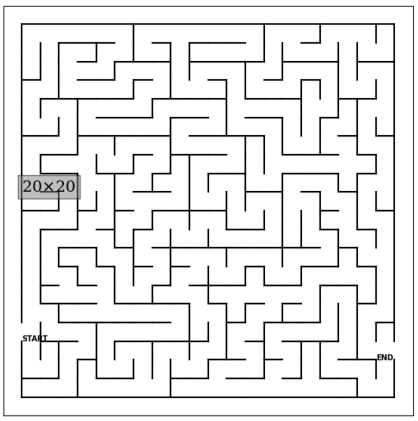


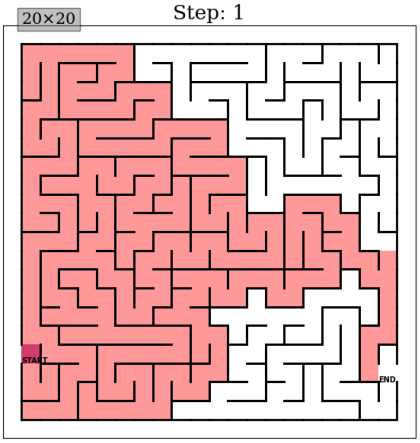


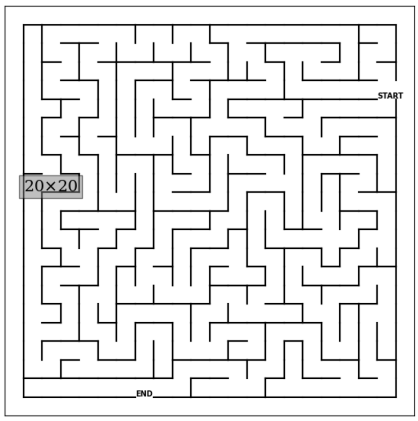


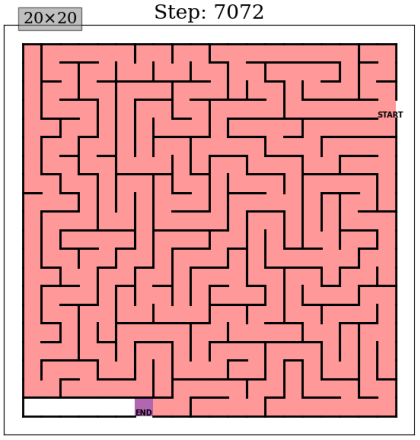


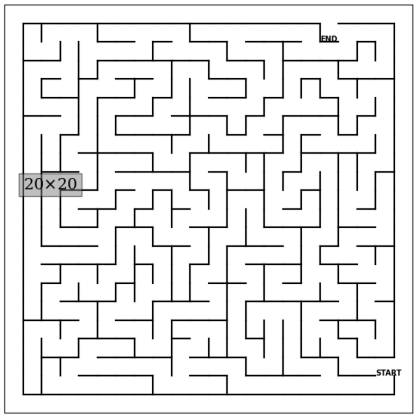


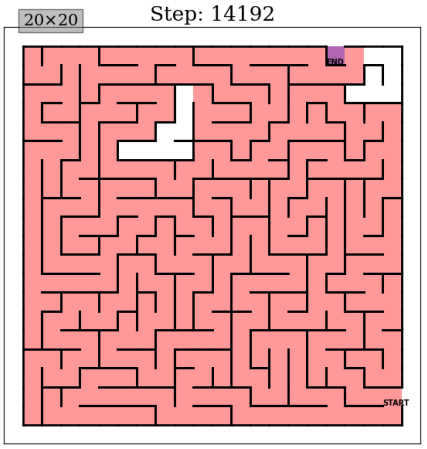

20×20
START
END

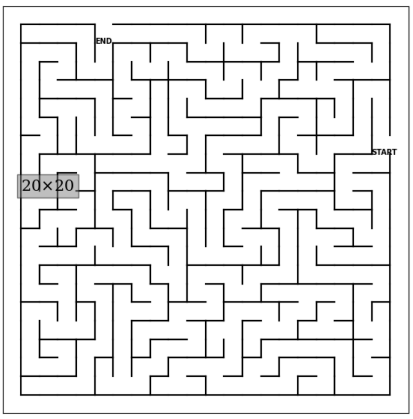

20×20
Step: 1
START
END

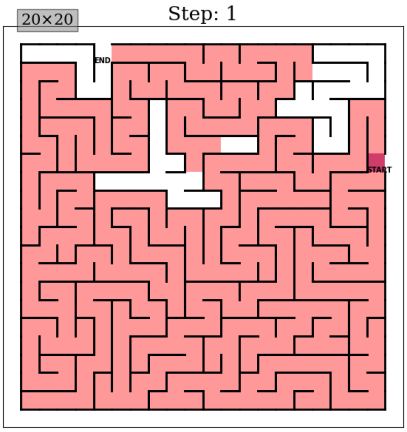

START
20×20
END

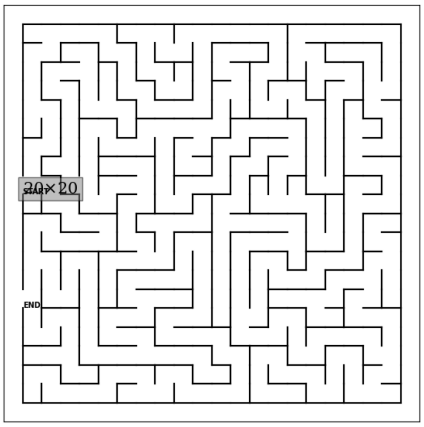

20×20
Step: 7072
START
END

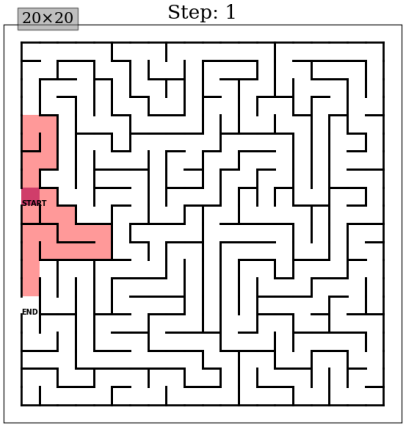

END
20×20
START

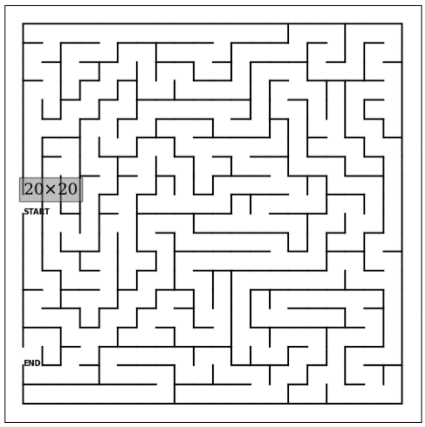

20×20
Step: 14192
END
START

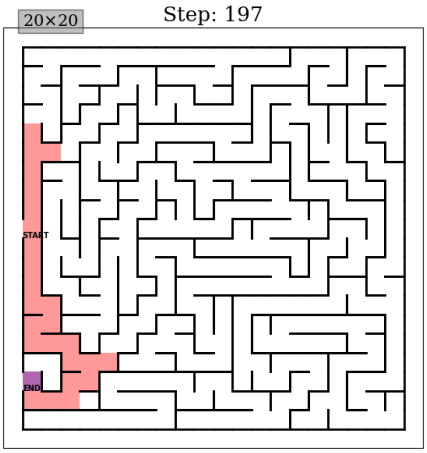

END
START
20×20

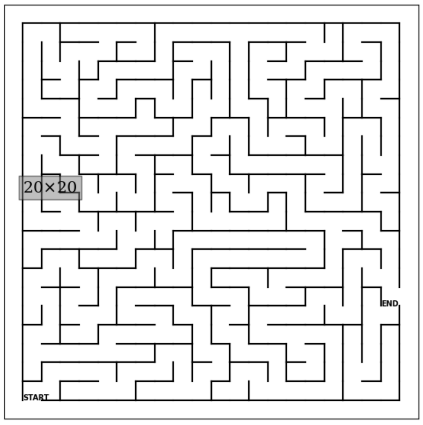

20×20
Step: 1
END
START

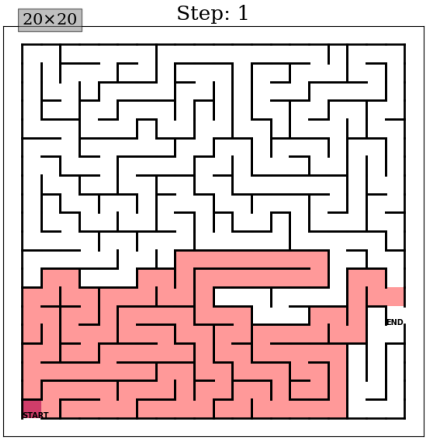

20×20
START
END


20×20
Step: 1
START
END









step 수는 사진에는 제대로 저장되지 않은 것 같습니다.
자세한 결과는 mp4 동영상 파일에서 확인하실 수 있습니다.
Mp4 영상들은 UCS_vid.zip, A_star_vid.zip 집파일에 있습니다.

단순히 뭐가 더 좋다 라는 결론은 내리기 어렵다는 결론이 나왔습니다. 이는 미로에 따라 각각 장점과 단점이 존재하기 때문입니다. A* search 가 더 좋을 것이라 예상했지만, 경우에 따라 비교가 어려울 정도로 오래 걸리는 경우들도 있었습니다.

Uniform cost search 의 장점은 항상 가장 작은 코스트의 길을 택하기에 빠르게 찾기만 한다면 가장 작은 코스트의 길을 찾을 수 있습니다.
대신 어떤 경로든지 거리만 가까우면 먼저 조사하기에 목적지까지 경로와 잘못된 길의 경로가 거리가 더 짧다면 그쪽으로 가려고 하다가 다시 올바른 길로 가려다 보니 왔다 갔다 하면서 활동을 많이 해야 합니다.

A* search 는 Uniform cost search 와는 조금 다르지만 가는 코스트와 도착지까지의 예측 코스트도 같이 계산하여 길을 찾기에 역시 가장 작은 코스트의 길을 찾기 좋습니다.
그러나 heuristic 함수의 정확성이 중요하고, 말 그대로 예측 코스트를 추가하기에 이 예측 코스트가 함수가 잘못되면 비효율적으로 경로를 탐색할 수 있습니다. 예를 들어 본 과제에서 사용한 goal 까지의 직선거리만 코스트로 추가하면, goal 로 가는 정답 경로가 goal 에서 멀어졌다가 돌아와야 하는 둘러가는 길 같은 경우에는 그쪽으로 가려는 경우가 줄어들어 더 오래 걸리게 됩니다. 대신 이 함수가 잘 맞춰졌을 경우 행동 횟수가 훨씬 줄어들 수 있습니다. 그러나 앞선 Uniform cost search 보다 말그대로 함수가 하나 더 들어가기에 계산하는 시간 같은 코스트가 더 많이 듭니다.