

기초 인공지능

AI_assignment01 보고서

20191621 이민영

1. 사용한 라이브러리와 각 알고리즘마다 구현한 방법

1) 사용한 라이브러리

import itertools : permutations를 사용하기 위해서 import 하였다.

from collections import defaultdict : prim알고리즘에서 사용하기 위해서 import하였다.

from heapq import * : prim알고리즘에서 heap을 사용하기 위해서 import 하였다.

2) 각 알고리즘마다 구현한 방법

-BFS

시작지점을 우선 집어넣고, 인접한 길을 탐색하면서 BFS 알고리즘을 수행한다. 이미 갔던 지점을 다시 방문하지 않기 위해서 visited 를 사용하였고 처음 방문할 때 visited함수에 넣어두었다. bfs이기 때문에 인접한 정점을 우선적으로 탐색하는데, 인접한 정점을 모두 검사한 후에는 지금까지 저장한 목록에서 다음 정점을 꺼내서 방문하였다. 즉, 정점의 목록에서 어떤 정점을 먼저 꺼내는지에 따라서 방문 순서를 정할 수 있도록 구현하였다.

마지막에 완성된 길을 출력하기 위해서 trace_path에 node를 저장해두었다.

BFS를 완료한 후, tracing 함수를 통해서 최종 가장 짧은 길을 이미 저장해두었던 trace_path를 이용해서 구한다.

```
def bfs(maze):
    """
    [문제 01] 제시된 stage1의 맵 세가지를 BFS Algorithm을 통해 최단 경로를 return하시오. (20점)
    """
    start_point=maze.startPoint()

    visited = []
    fringe = []
    path=[]
    # 값을 초기화하기 빈 리스트

    ##### Write Your Code Here #####

    fringe.append(start_point)
    def bf_Search():
        closed = []
        a = 1

        #fringe안에 있는 node를 살펴보
        while fringe:
            node = fringe.pop(0)

            visited.append(node)
            for i in maze.neighborPoints(node[0],node[1]):
                if i not in visited:
                    #이웃노드들 중 아직 방문하지 않은 노드가 있으면 추가하
                    fringe.append(i)

            trace_path[i] = node

            if maze.circlePoints == node:
                break

    '''최종 경로를 출력하기 위한 tracing함수'''
    def tracing(node,trace_path):
        if(node == maze.startPoint()):
            return
        index = node[0]*10+node[1]
        next_node = trace_path[node]
        tracing(next_node,trace_path)
        path.append(next_node)

    trace_path = dict()
    bf_Search()

    tracing(maze.circlePoints()[0],trace_path)
    path.append(maze.circlePoints()[0])

    return path

#####
```

-A*

시작노드를 받고, 살펴볼 노드들은 openlist에 저장한다. 이 openlist를 통해서 while문을 반복한다. 살펴볼 노드를 openlist에서 꺼내고, 이미 살펴본 노드는 closedlist에 저장한다. 살펴볼 노드의 이웃 노드들을 살펴보면서 child에 넣고, 이 안에서 현재까지의 상황을 저장하는 g와 앞으로의 상황을 예측한 h를 더해서 f값에 저장한다. 이 f 값이 가장 적은 순서대로 살펴보게 된다. 살펴볼 노드가 마지막 목표지점과 동일할 경우에 path를 저장하고 return하게 된다. 이때 path는 따라온 부모노드 즉 now_N.parent를 따라서 저장하였다. stage1,2,3모두 동일하게 위와 같은 방식으로 A* 알고리즘을 구현하였고 heuristic에 변화를 주었다.

```
def A_star():
    #초기화하기

    openlist = []
    closedlist=[]
    index = 0

    startN = Node(None,start_point)
    endN = Node(None,end_point)

    openlist.append(startN)
    while openlist:
        node = openlist[0]
        index = 0

        for i,j in enumerate(openlist):
            if j.f < node.f:
                index = i
                node = j

        openlist.pop(index)
        closedlist.append(node)

        if node == endN:

            path = []
            now_N = node
            while now_N is not None:
                path.append(now_N.location)
                now_N = now_N.parent
            return path[::-1]

        child = []

        for i in maze.neighborPoints(node.location[0],node.location[1]):
            node_append = Node(node,i)
            child.append(node_append)

        for now_child in child:
            if now_child in closedlist:
                continue

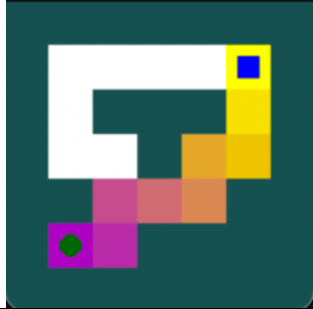
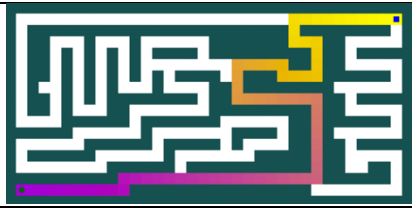
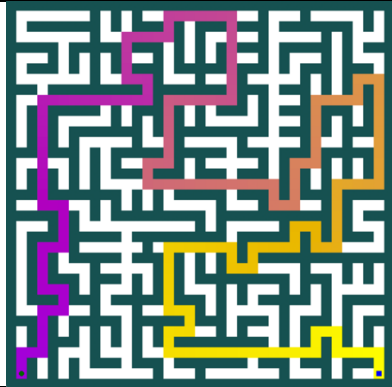
            now_child.g = node.g + 1
            now_child.h = manhattan_dist(now_child.location,endN.location)
            now_child.f = now_child.g + now_child.h

            if len([openNode for openNode in openlist
                    if now_child == openNode and now_child.g > openNode.g]) > 0:
                continue

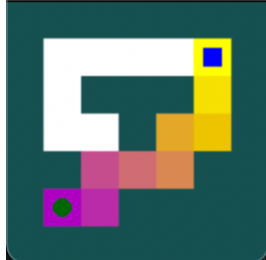
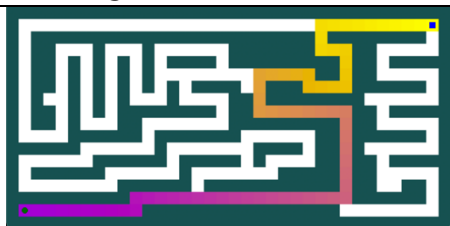
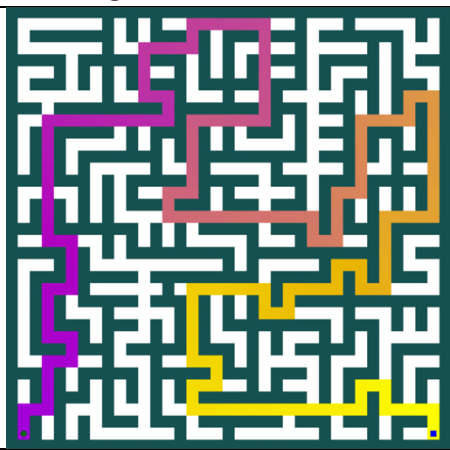
            openlist.append(now_child)
```

2. 캡처 화면


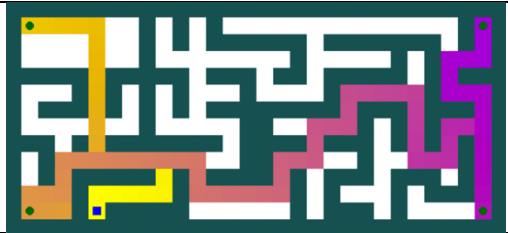
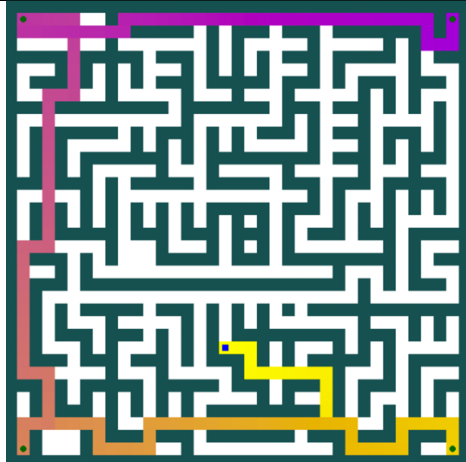
1) bfs method로 실행한 stage1의 small.txt, medium.txt, big.txt 경로 그림과 출력 화면

	<pre>===== [bfs results] (1) Path Length: 9 (2) Search States: 18 (3) Execute Time 0.0001711845 seconds =====</pre>
bfs-stage1-small.txt 경로그림	bfs-stage1-small.txt 출력 화면
	<pre>===== [bfs results] (1) Path Length: 69 (2) Search States: 282 (3) Execute Time 0.0032868385 seconds =====</pre>
bfs-stage1-medium.txt 경로그림	bfs-stage1-medium.txt 출력 화면
	<pre>===== [bfs results] (1) Path Length: 211 (2) Search States: 648 (3) Execute Time 0.0099709034 seconds =====</pre>
bfs-stage1-big.txt 경로그림	bfs-stage1-big.txt 출력 화면

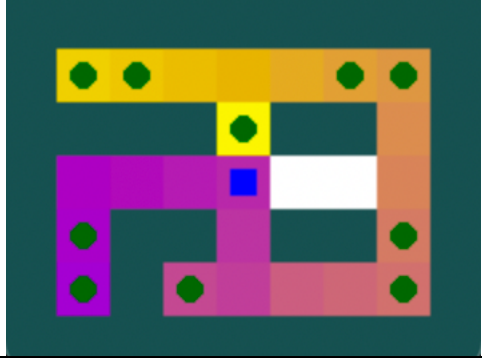
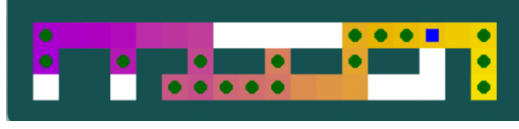
2) astar method로 실행한 stage1의 small.txt, medium.txt, big.txt 경로 그림과 출력 화면

	<pre>===== [astar results] (1) Path Length: 9 (2) Search States: 14 (3) Execute Time 0.0003890991 seconds =====</pre>
<p>astar-stage1-small.txt 경로그림</p>	<p>astar-stage1-small.txt 출력 화면</p>
	<pre>===== [astar results] (1) Path Length: 69 (2) Search States: 224 (3) Execute Time 0.0132939816 seconds =====</pre>
<p>astar-stage1-medium.txt 경로그림</p>	<p>astar-stage1-medium.txt 출력 화면</p>
	<pre>===== [astar results] (1) Path Length: 211 (2) Search States: 549 (3) Execute Time 0.0515820980 seconds =====</pre>
<p>astar-stage1-big.txt 경로그림</p>	<p>astar-stage1-big.txt 출력 화면</p>

3) astar_four_circles method로 실행한 stage2의 small.txt, medium.txt, big.txt 경로 그림과 출력 화면

	<pre>===== [astar_four_circles results] (1) Path Length: 29 (2) Search States: 922 (3) Execute Time 0.0081279278 seconds =====</pre>
<p>astar_four-stage2-small.txt 경로그림</p> 	<p>astar_four-stage2-small.txt 출력 화면</p> <pre>===== [astar_four_circles results] (1) Path Length: 107 (2) Search States: 15468 (3) Execute Time 1.4493260384 seconds =====</pre>
<p>astar_four-stage2-medium.txt 경로그림</p> 	<p>astar_four-stage2-medium.txt 출력 화면</p> <pre>===== [astar_four_circles results] (1) Path Length: 163 (2) Search States: 6236 (3) Execute Time 1.3822510242 seconds =====</pre>
<p>astar_four-stage2-big.txt 경로그림</p>	<p>astar_four-stage2-big.txt 출력 화면</p>

- 4) astar_many_circles method로 실행한 stage3의 small.txt, medium.txt, big.txt 경로 그림과 출력 화면

	<pre>===== [astar_many_circles results] (1) Path Length: 28 (2) Search States: 7429 (3) Execute Time 0.0430552959 seconds =====</pre>
astar_many-stage3-small.txt 경로그림	astar_many-stage3-small.txt 출력 화면
	<pre>===== [astar_many_circles results] (1) Path Length: 35 (2) Search States: 17240 (3) Execute Time 0.1029281616 seconds =====</pre>
astar_many-stage3-medium.txt 경로그림	astar_many-stage3-medium.txt 출력 화면
astar_many-stage3-big.txt 경로그림	astar_many-stage3-big.txt 출력 화면

3. Stage2와 Stage3에서 직접 정의한 Heuristic Function에 대한 설명

1) Stage2에서 정의한 Heuristic Function

Stage2에서는 우선 4개의 circle들이 다 돌면서 발생할 수 있는 경우의 수 중 가장 최소의 경우를 선택을 하였다. 그 후에 앞으로 이동해야 할 위치와 최소의 경우 시작점의 거리가 가장 작은 이동할 위치를 골랐다.

가장 최소의 경우는 astar_four_circles 함수 첫 부분에서 구현하였고, heuristic function은 stage2_heuristic함수를 이용하였다. 이 때 start와 현재 위치의 거리를 manhattan_dist함수를 이용해서 구하고 이와 astar_four_circles 함수 첫 부분에 구현한 최솟값과 더해서 return해주었다.

```
def stage2_heuristic(maze,p1,start,min_dis):
    return manhattan_dist(p1.location,start) + min_dis
```

2) Stage3에서 정의한 Heuristic Function

Stage3에서는 MST를 이용해서 Heuristic 함수를 짰다. circle들을 이용해서 MST를 만들었다. 이 때 MST를 만들기 위해서 Prim 알고리즘을 이용하였다. 이 후에 이에 해당하는 최솟값을 가지는 지점으로 이동할 수 있도록 구현하였다.

MST는 prim알고리즘을 이용하여 mst 함수에 구현하였다. 지금까지 방문하지 않은 circle 들을 입력받아서 MST를 구현한다. 이 후에 최종 숫자를 더해서 return 한다.

```
def mst(objectives, edges):
    cost_sum=0
    ##### Write Your Code Here #####
    cost_list = []
    #print(objectives)
    #print(edges)
    mstt = list()
    adj = defaultdict(list)
    start = objectives[0]
    for a,b in edges:
        adj[a].append((edges[(a,b)],a,b))
        adj[b].append((edges[(a,b)],b,a))
        #인접하는 edge값 저장해
    connect = set([start])
    fringe = adj[start]
    heapify(fringe)
    temp=start
    k=0
    while fringe:
        w,a1,b1 = heappop(fringe)
        if (b1 not in connect) :
            connect.add(b1)
            mstt.append((w,a1,b1))
            #print(w,cost_sum)
            cost_sum = cost_sum+w
            temp = b1
            for jj in adj[b1]:
                if jj[2] not in connect:
                    heappush(fringe,jj)
    cost_list.append(0)
    |
    return cost_sum,mstt
#####
```

heuristic function은 stage3_heuristic함수에 구현하였다. mst함수를 불러서 그 return값과 A_star_circle함수를 이용해서 현재 node와 시작 노드의 거리를 더해서 child의 h 값으로 반환하였다.