AI Assignment01 보고서

```
AI Assignment01 보고서
학번 : 20171612
학과 : 컴퓨터공학과
이름 : 김성일
```

사용한 라이브러리

```
from collections import defaultdict, deque
import heapq as hq
```

- `defaultdict`
  - MST를 위한 그래프를 만들 때 사용하였습니다.
- `deque`
  - `BFS`에서 사용 될 큐를 대신하기 위해 사용하였습니다.
- `heapq`
  - `A*` 알고리즘에서 `open` 리스트에서 제일 작은 값을 가진 `Node` 를 가져오기위한, min heap으로 사용하였습니다.
  - Prim 알고리즘을 이용하여 `MST` 를 구축할 때 사용하였습니다.

각 라이브러리에 관한 자세한 설명은 각 문제를 설명할 때 추가로 적어놓았습니다.

Problem 01

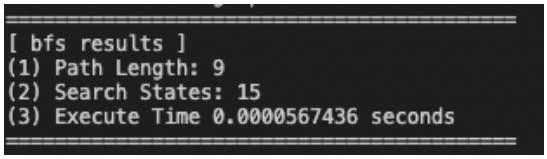
Stage 1의 최단 경로 탐색을위한 BFS 구현문제.

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

    # End point
    end_point = maze.circlePoints()[0]

    # Tracing을 위한 dictionary
    prev = {}

    queue = deque()
    queue.append(start_point)
    visit = [start_point]

    while queue:
        y,x = queue.popleft()
```

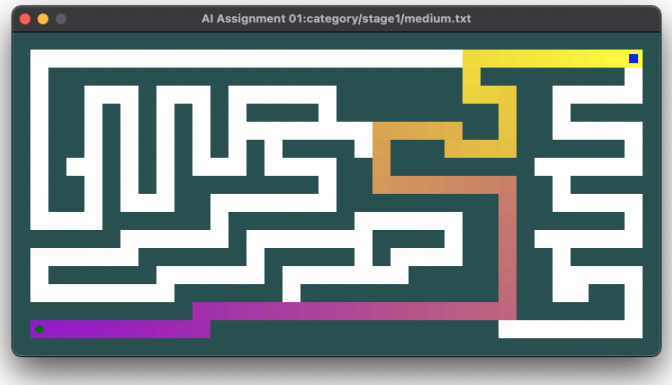
```
        if (y,x) == end_point:
            break
        for dy,dx in maze.neighborPoints(y,x):
            if (dy,dx) not in visit:
                visit.append((dy,dx))
                queue.append((dy,dx))
                prev[(dy,dx)] = (y,x)

    # Trace path
    node = (y,x)
    path = [node]

    while node != start_point:
        node = prev[node]
        path.append(node)

    return path[::-1]
```

출력 경로 및 결과

Stage1 small

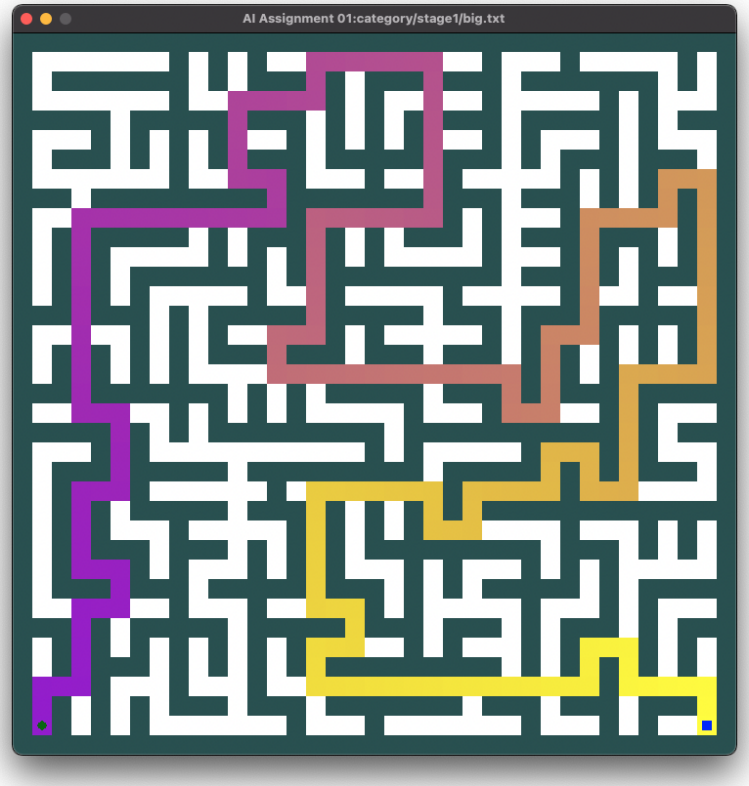


Stage1 medium




```
==========================================
[ bfs results ]
(1) Path Length: 69
(2) Search States: 269
(3) Execute Time 0.0016620159 seconds
==========================================
```

Stage1 big




```
==========================================
[ bfs results ]
(1) Path Length: 211
(2) Search States: 619
(3) Execute Time 0.0065488815 seconds
==========================================
```

사용한 라이브러리 및 자료구조

- `deque` 라이브러리
  - `Breadth-First-Search`를 구현하기위해 사용하였습니다.
  - BFS에서 사용 될 `queue` 를 대신하기 위해 사용하였습니다.
- `visit` 리스트
  - 이미 방문했는지 체크를하기위해 사용하였습니다.
  - 방문하지 않은 좌표라면 방문하였다는 것을 기록하고 그 좌표를 queue에 삽입하여 다음에 방문하도록 하였습니다.
  - 다음에 이동할 수 있는 좌표는 이미 `maze.py`에 `neigborPoints` 함수로 구현되어 있어 그것을 사용했습니다.
- `prev` 딕셔너리
  - 여태까지 방문했던 좌표들의 경로를 추적하기위해 사용한 dictionary입니다.
  - 현재 좌표를 key로하여 부모 좌표 (이전 좌표) 를 value 리스트에 삽입하여, BFS가 끝나면 `prev` 를 타고 올라가며 `path` 를 기록하였습니다.
  - 이렇게 생성 된 `path`는 구하려는 path의 역순이기 때문에 `path[::-1]` 를 return하도록 하였습니다.

위에서 설명과 같이 기본적인 BFS를 구현하여 prev를 통해 최종적인 path를 구했습니다.

Problem 02

Stage 1의 최단 경로 탐색을위한 A* 알고리즘 구현문제.

```
def astar(maze):

    """
    [문제 02] 제시된 stage1의 맵 세가지를 A* Algorithm을 통해 최단경로를 return하시오.(20점)
    (Heuristic Function은 위에서 정의한 manhatten_dist function을 사용할 것.)
    """

    start_point=maze.startPoint()

    end_point=maze.circlePoints()[0]

    path=[]

    start = Node(None, start_point)
    end = Node(None, end_point)

    open = []
    close = []
    hq.heappush(open,start)

    while open:
        cur_node = hq.heappop(open)
        close.append(cur_node)
```

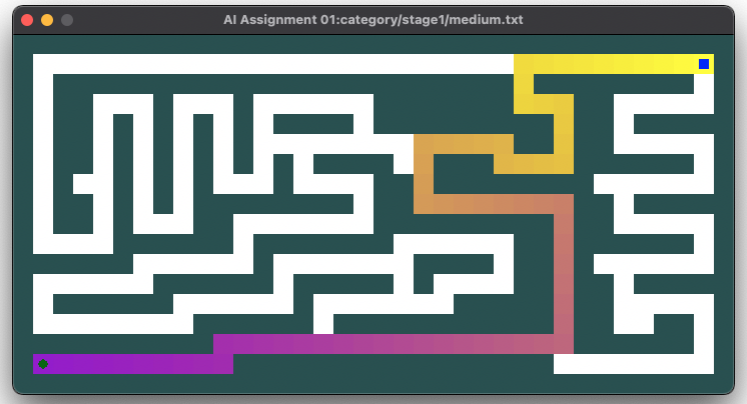
```
        if cur_node == end:
            cur = cur_node
            while cur:
                path.append(cur.location)
                cur = cur.parent
            break


        for dy,dx in maze.neighborPoints(cur_node.location[0],cur_node.location[1]):
            new_node = Node(cur_node,(dy,dx))
            if new_node in close:
                continue
            new_node.g = cur_node.g + 1
            new_node.h = manhatten_dist(new_node.location, end.location)
            new_node.f = new_node.g + new_node.h

            for value in open:
                if new_node == value and new_node > value:
                    break
            else:
                hq.heappush(open,new_node)


    path = path[::-1]
    return path
```

출력 경로 및 결과

Stage1 small

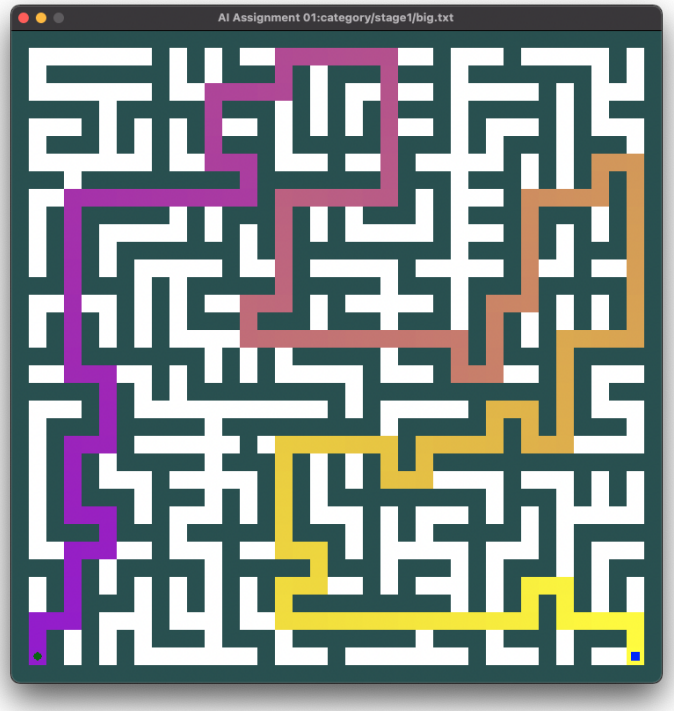

```
==========================================
[ astar results ]
(1) Path Length: 9
(2) Search States: 14
(3) Execute Time 0.0001170635 seconds
==========================================
```

Stage1 medium




```
==========================================
[ astar results ]
(1) Path Length: 69
(2) Search States: 224
(3) Execute Time 0.0058901310 seconds
==========================================
```

Stage1 big




```
==========================================
[ astar results ]
(1) Path Length: 211
(2) Search States: 549
(3) Execute Time 0.0291631222 seconds
==========================================
```

사용한 라이브러리 및 자료구조

- `heapq` 라이브러리
  - `open` 리스트에서 제일 작은 값을 가진 `Node` 를 가져오기위한, `min heap`으로 사용하였습니다.
  - `Node`의 값 비교는 이미 `Node` 클래스에서 h값을 비교하도록 오버라이드 되어있어 그것을 통해 비교합니다.
- `open` 리스트
  - heapq 라이브러리를 통해 구성되어있는 Node들의 open 리스트(min heap)입니다.
  - 방문하지 않은 상태를 가진 좌표들이 담겨있습니다.
- `close` 리스트
  - 이미 방문한 상태를 가진 좌표들이 담겨있습니다.
  - BFS에서 사용한 visit과 같은 역할을합니다.

후보가 될 수 있는 좌표 (open에 삽입할 Node) 구하기

다음에 이동할 좌표의 후보는 `maze.py`에 `neigborPoints` 함수로 구현되어 있어 그것을 사용했습니다.

그 후보들의 좌표(튜플)를 `Node` 객체로 만들어, 그 Node의 g, h, f 값을 계산하고 open 리스트에 같은 location을 가지며 자신보다 작은 f 값을 가진 Node가 존재하면 삽입하지 않고 그렇지 않다면 삽입하도록 하였습니다.

즉, 아래와 같은 경우가 존재하게 됩니다.

- 같은 location을 가지며 f 값이 자신보다 큰 Node가 있으면 삽입
  - min-heap과 close 리스트를 사용하기 때문에 사실상 교체와 같은 동작을하게 됩니다.
- 같은 location을 가진 Node가 없으면 삽입

G, H, F 값의 계산

- G
  - 현재 Node에서 출발 지점까지의 코스트이기 때문에, 부모 좌표(이전 좌표)의 G값에서 1을 더한 값을 사용했습니다.
- H
  - 문제에 주어진대로 `manhatten_dist` 를 통해 도착지와의 맨해튼 거리를 H값으로 사용했습니다.
- F
  - 정의와 같이 G와 H 값을 더한 값을 F로 사용했습니다.

경로 구하기

현재 Node가 도착점이라면, 현재 Node부터 시작 Node까지 부모를 타고 올라가며 path를 구했습니다.

이렇게 구한 path는 BFS와 마찬가지로 역순으로 구해지기 때문에 `path[::-1]` 을 return하게 됩니다.

Problem 03

Stage 2의 최단 경로 탐색을위한 A* 알고리즘 구현문제.

```
def stage2_heuristic(cur, end, visit):
    min_value = float("inf")
    for idx,end_point in enumerate(end):
        if not visit[idx]:
            min_value = min(min_value,manhatten_dist(cur,end_point.location))
    return min_value

def astar_four_circles(maze):
    """
    [문제 03] 제시된 stage2의 맵 세가지를 A* Algorithm을 통해 최단 경로를 return하시오.(30점)
    (단 Heurstic Function은 위의 stage2_heuristic function을 직접 정의하여 사용해야 한다.)
    """

    end_points=maze.circlePoints()
    end_points.sort()

    path=[]

    ######################## Write Your Code Here ################################
    start_point = maze.startPoint()

    start = Node(None, start_point)
    end = [Node(None, end_point) for end_point in end_points]
    visit = [False for _ in range(len(end_points))]

    for _ in range(4):
        open = []
        close = []
        if len(path) != 0:
            hq.heappush(open,Node(None,path[-1]))
        else:
            hq.heappush(open,start)
        while open:
            cur_node = hq.heappop(open)
            close.append(cur_node)

            if cur_node in end:
                if not visit[end.index(cur_node)]:
                    visit[end.index(cur_node)] = True
                    cur = cur_node
                    tmp = []
                    while cur:
                        tmp.append(cur.location)
                        cur = cur.parent
                    path = path[:-1] + tmp[::-1]
                    break


            for dy,dx in maze.neighborPoints(cur_node.location[0],cur_node.location[1]):
                new_node = Node(cur_node,(dy,dx))
                if new_node in close:
                    continue
                new_node.g = cur_node.g + 1
                new_node.h = manhatten_dist(new_node.location, end.location)
                new_node.f = new_node.g + new_node.h
```

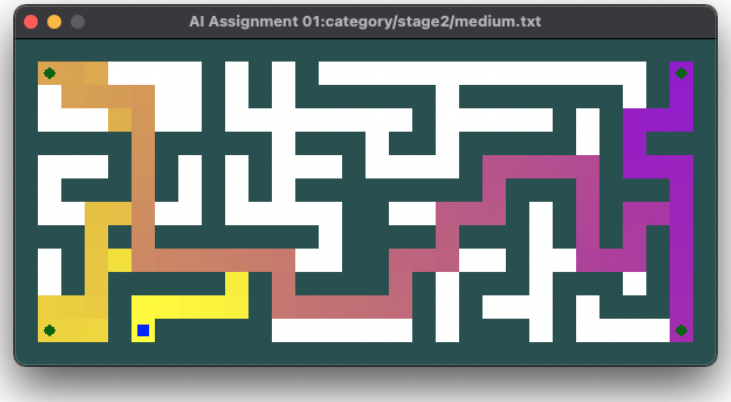
```
                for value in open:
                    if new_node == value and new_node > value:
                        break
                else:
                    hq.heappush(open,new_node)

    return path
```

출력 경로 및 결과

Stage2 small

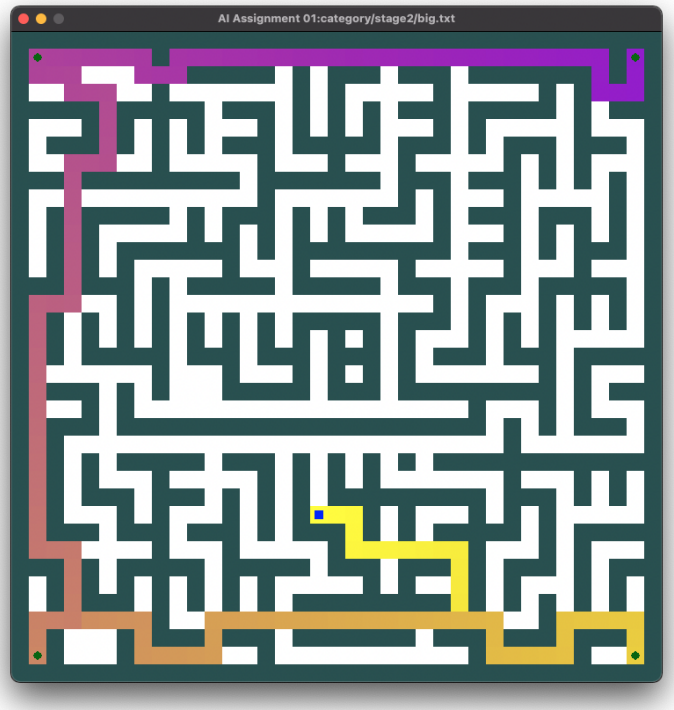

```
==========================================
[ astar_four_circles results ]
(1) Path Length: 33
(2) Search States: 51
(3) Execute Time 0.0004348755 seconds
==========================================
```

Stage2 medium



```
[ astar_four_circles results ]
(1) Path Length: 107
(2) Search States: 327
(3) Execute Time 0.0082409382 seconds
```

Stage2 big




```
[ astar_four_circles results ]
(1) Path Length: 163
(2) Search States: 401
(3) Execute Time 0.0069530010 seconds
```

사용한 라이브러리 및 자료구조

- `heapq` 라이브러리
  - `open` 리스트에서 제일 작은 값을 가진 `Node` 를 가져오기위한, `min heap`으로 사용하였습니다.
  - `Node`의 값 비교는 이미 `Node` 클래스에서 h값을 비교하도록 오버라이드 되어있어 그것을 통해 비교합니다.
- `open` 리스트
  - heapq 라이브러리를 통해 구성되어있는 Node들의 open 리스트(min heap)입니다.
  - 방문하지 않은 상태를 가진 좌표들이 담겨있습니다.
  - for 문의 iteration이 한 번 끝날 때마다 초기화됩니다.
- `close` 리스트

- 이미 방문한 상태를 가진 좌표들이 담겨있습니다.
- BFS에서 사용한 visit과 같은 역할을합니다.
- for 문의 iteration이 한 번 끝날 때마다 초기화됩니다.
    - 도착지점이 여러개이기 때문에 이미 지났던 곳을 지날 수 있어, 초기화하게 됩니다.

후보가 될 수 있는 좌표 (open에 삽입할 Node) 구하기

Problem 02와 동일합니다.

다음에 이동할 좌표의 후보는 `maze.py`에 `neigborPoints` 함수로 구현되어 있어 그것을 사용했습니다.

그 후보들의 좌표(튜플)를 `Node` 객체로 만들어, 그 Node의 g, h, f 값을 계산하고 open 리스트에 같은 location을 가지며 자신보다 작은 f 값을 가진 Node가 존재하면 삽입하지 않고 그렇지 않다면 삽입하도록 하였습니다.

즉, 아래와 같은 경우가 존재하게 됩니다.

- 같은 location을 가지며 f 값이 자신보다 큰 Node가 있으면 삽입
    - min-heap과 close 리스트를 사용하기 때문에 사실상 교체와 같은 동작을하게 됩니다.
- 같은 location을 가진 Node가 없으면 삽입

G, H, F 값의 계산

- G
    - 현재 Node에서 출발 지점까지의 코스트이기 때문에, 부모 좌표(이전 좌표)의 G값에서 1을 더한 값을 사용했습니다.
- H
    - `stage2_heuristic` 을 통해 구한 값을 H값으로 사용했습니다.
- F
    - 정의와 같이 G와 H 값을 더한 값을 F로 사용했습니다.

stage2_heuristic

`stage2_heuristic(cur:tuple, end:list[tuple], visit:list[bool]) -> int`

방문하지 않은 도착 좌표 중 현재 좌표와의 맨해튼 거리 중 가장 작은 값을 반환하는 휴리스틱 함수입니다.

- `cur`
    - 현재 좌표
- `end`
    - 도착 좌표들을 가지고 있는 리스트

- `visit`
  - 도착 좌표들을 이미 방문했는지 체크하기 위한 리스트

경로 구하기

이 문제는 **도착해야하는 좌표가 4개**라는 점에 기인하여, **Problem 02**에서 진행했던 작업을 for문을 통해 4번 반복하였습니다.

각 iteration마다 `open`과 `close`는 빈 배열로 초기화되며, 이전 단계가 존재했다면 (이미 다른 도착점에 도착한적이 있다면) `open` 에 `path`의 마지막 값(마지막으로 방문했던 좌표)을 삽입하여 경로 탐색을 시작하도록 했습니다. 그것이 아니라면 첫 시작점이 `open` 에 삽입되게 됩니다.

그리고 경로 탐색을 하던 중 현재 Node가 도착점이라면, 현재 Node부터 시작 Node까지 부모를 타고 올라가며 임시 path를 구했습니다.

이렇게 구한 임시 path는 BFS와 마찬가지로 역순으로 구해지기 때문에 `tmp[::-1]`을 하게 되는데, 이전에 구한 경로의 마지막 지점과 `tmp[::-1]`의 시작지점은 겹치게 되기 때문에 `path = path[:-1] + tmp[::-1]` 을 사용하여 최종 경로를 각 iteration 동안 쌓게 됩니다.

그렇게 구한 최종적인 path는 시작점부터 각 4개의 지점을 통과한 경로가 됩니다.

Problem 04

Stage 3의 최단 경로 탐색을위한 A* 알고리즘 및 이를 위한 MST를 이용한 휴리스틱 함수 구현문제.

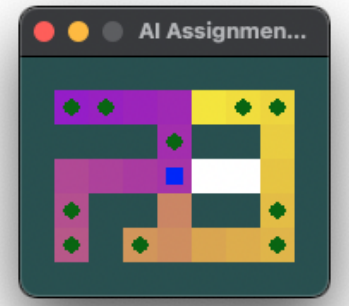
```
def mst(start, objectives):
    '''
    방문하지 않은 objectives와 현재 노드를 합쳐 mst를 만든다
    각각의 edge들의 weight는 각 edge에서 다른 edge까지의 mahatten 거리를 사용
    mst는 prim 알고리즘을 통해 구현
    '''

    cost_sum=0
    ######################## Write Your Code Here ################################
    graph = defaultdict(list)

    for y1,x1 in objectives:
        for y2,x2 in objectives:
            if y1 == y2 and x1 == x2:
                continue
            weight = manhatten_dist((y1,x1),(y2,x2))
            graph[(y1,x1)].append((weight, (y2,x2)))
            graph[(y2,x2)].append((weight, (y1,x1)))

    for y,x in objectives:
        weight = manhatten_dist(start,(y,x))
        graph[(y,x)].append((weight,start))
```

```
            graph[start].append((weight,(y,x)))

        connected = set([start])
        candidate_edge = graph[start]
        hq.heapify(candidate_edge)

        while candidate_edge:
            w, v = hq.heappop(candidate_edge)
            if v not in connected:
                connected.add(v)
                cost_sum += w

                for edge in graph[v]:
                    if edge[1] not in connected:
                        hq.heappush(candidate_edge,edge)

        return cost_sum

        ###################################################################################


    def stage3_heuristic(cur, objectives, visit):
        # cur_node에서 end_node까지 가는 mst를 구축하여, cost_sum을 반환
        not_visit_objectives = []

        for idx, objective in enumerate(objectives):
            if not visit[idx]:
                not_visit_objectives.append(objective.location)

        return mst(cur,not_visit_objectives)

    def astar_many_circles(maze):
        """
        [문제 04] 제시된 stage3의 맵 세가지를 A* Algorithm을 통해 최단 경로를 return하시오.(30점)
        (단 Heurstic Function은 위의 stage3_heuristic function을 직접 정의하여 사용해야 하고, minimum spanning tree
        알고리즘을 활용한 heuristic function이어야 한다.)
        """

        end_points= maze.circlePoints()
        end_points.sort()

        path=[]

        ######################## Write Your Code Here ################################
        start_point = maze.startPoint()

        start = Node(None, start_point)
        end = [Node(None, end_point) for end_point in end_points]
        visit = [False for _ in range(len(end_points))]

        while visit.count(True) < len(visit):
            open = []
            close = []
            if len(path) != 0:
                hq.heappush(open,Node(None,path[-1]))
            else:
                hq.heappush(open,start)
            while open:
                cur_node = hq.heappop(open)
                close.append(cur_node)

                if cur_node in end:
                    if not visit[end.index(cur_node)]:
                        visit[end.index(cur_node)] = True
                        cur = cur_node
                        tmp = []
                        while cur:
```

```
                    tmp.append(cur.location)
                    cur = cur.parent
                path = path[:-1] + tmp[::-1]
                break


        for dy,dx in maze.neighborPoints(cur_node.location[0],cur_node.location[1]):
            new_node = Node(cur_node,(dy,dx))
            if new_node in close:
                continue
            new_node.g = cur_node.g + 1
            new_node.h = stage3_heuristic(new_node.location, end ,visit)
            new_node.f = new_node.g + new_node.h

            for value in open:
                if new_node == value and new_node > value:
                    break
            else:
                hq.heappush(open,new_node)
    return path
```

출력 경로 및 결과

Stage3 small

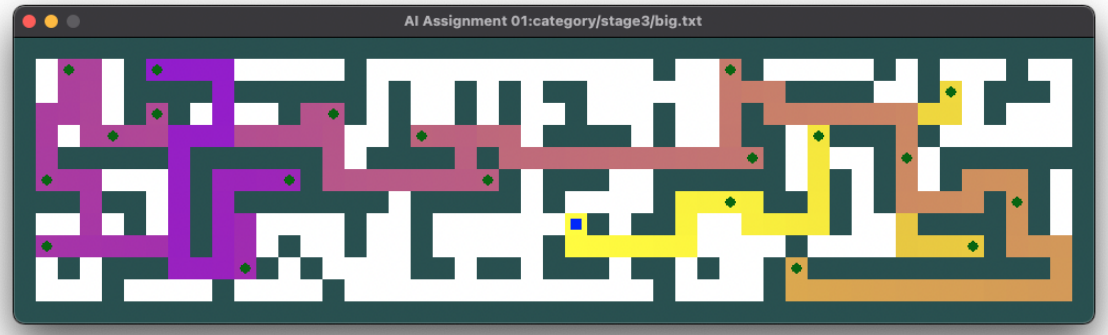



```
[ astar_many_circles results ]
(1) Path Length: 32
(2) Search States: 35
(3) Execute Time 0.0020701885 seconds
```

Stage3 medium


```
==========================================
[ astar_many_circles results ]
(1) Path Length: 49
(2) Search States: 76
(3) Execute Time 0.0099098682 seconds
==========================================
```

Stage3 big




```
==========================================
[ astar_many_circles results ]
(1) Path Length: 223
(2) Search States: 745
(3) Execute Time 0.1534631252 seconds
==========================================
```

사용한 라이브러리 및 자료구조

- `heapq` 라이브러리
  - `open` 리스트에서 제일 작은 값을 가진 `Node` 를 가져오기위한, `min heap` 으로 사용하였습니다.
    - `Node` 의 값 비교는 이미 `Node` 클래스에서 h값을 비교하도록 오버라이드 되어있어 그것을 통해 비교합니다.
  - MST를 구축할 때 candidate edge 중 가장 작은 weight를 가진 edge를 가져오기 위해 사용하였습니다.

- `defaultdict` 라이브러리
  - MST를 위한 그래프를 만들 때 사용하였습니다.
  - 그래프는 `{u1 : [(weight,v), (weigh,v)...], u2: [(weight,v)...]}` 와 같은 방식으로 이루어져있으며, 일반적인 dictionary를 통해 구현하게 되면 dictionary에 key에 해당하는 value가 없을 때 초기화하는 과정을 거쳐야하므로 그것을 방지하기 위해 defaultdict를 사용하였습니다.
- `open` 리스트
  - heapq 라이브러리를 통해 구성되어있는 Node들의 open 리스트(min heap)입니다.
  - 방문하지 않은 상태를 가진 좌표들이 담겨있습니다.
  - 가장 바깥 while문의 iteration이 한 번 끝날 때마다 초기화됩니다.
- `close` 리스트
  - 이미 방문한 상태를 가진 좌표들이 담겨있습니다.
  - BFS에서 사용한 visit과 같은 역할을합니다.
  - 가장 바깥 while문의 iteration이 한 번 끝날 때마다 초기화됩니다.
    - 도착지점이 여러개이기 때문에 이미 지났던 곳을 지날 수 있어, 초기화하게 됩니다.

후보가 될 수 있는 좌표 (open에 삽입할 Node) 구하기

Problem 02와 동일합니다.

다음에 이동할 좌표의 후보는 `maze.py`에 `neigborPoints` 함수로 구현되어 있어 그것을 사용했습니다.

그 후보들의 좌표(튜플)를 `Node` 객체로 만들어, 그 Node의 g, h, f 값을 계산하고 open 리스트에 같은 location을 가지며 자신보다 작은 f 값을 가진 Node가 존재하면 삽입하지 않고 그렇지 않다면 삽입하도록 하였습니다.

즉, 아래와 같은 경우가 존재하게 됩니다.

- 같은 location을 가지며 f 값이 자신보다 큰 Node가 있으면 삽입
  - min-heap과 close 리스트를 사용하기 때문에 사실상 교체와 같은 동작을하게 됩니다.
- 같은 location을 가진 Node가 없으면 삽입

G, H, F 값의 계산

- G
  - 현재 Node에서 출발 지점까지의 코스트이기 때문에, 부모 좌표(이전 좌표)의 G값에서 1을 더한 값을 사용했습니다.
- H
  - `stage3_heuristic` 을 통해 구한 값을 H값으로 사용했습니다.
- F

- 정의와 같이 G와 H 값을 더한 값을 F로 사용했습니다.

mst

`mst(start:tuple, objectives:list[tuple]) -> int`

현재 좌표와 방문하지않은 도착 좌표들을 이용하여 MST를 만들고, 그 MST의 weight 합을 반환하는 함수입니다.

각 좌표 사이의 weight는 `manhatten_distance` 를 이용하여 구현했습니다.

- `start`
  - 현재 좌표 (MST에서의 시작 좌표)
- `objectives`
  - `astar_many_circles` 에서 주어진 도착 좌표 중 방문하지 않은 좌표

MST를 구현할 때는 prim 알고리즘을 사용하여 구현하였고, 현재 edge에서 갈 수 있는 candidate_edge 중 가장 작은 가중치를 가진 edge를 가져오기 위해 heapq 즉, min-heap을 사용하여 구현하였습니다.

stage3_heuristic

`stage3_heuristic(cur:tuple, objectives:list[tuple], visit:list[bool]) -> int`

방문하지 않은 도착 좌표들과 현재 좌표를 `mst` 함수에게 전달하여 구축한 MST의 weight의 합을 반환받아, 그것을 반환하는 휴리스틱 함수입니다.

- `cur`
  - 현재 좌표
- `objectives`
  - 도착 좌표들을 가지고 있는 리스트
- `visit`
  - 도착 좌표들을 이미 방문했는지 체크하기 위한 리스트

경로 구하기

이 문제는 **도착해야하는 좌표가 n개** 였습니다.

Problem 03에서 진행했던 작업을 확장하여, visit 배열을 이용하여 모든 도착 좌표를 방문하는 동안 while iteration을 반복하도록 했습니다. (`while visit.count(True) < len(visit):`)

각 iteration마다 `open` 과 `close` 는 빈 배열로 초기화되며, 이전 단계가 존재했다면 (이미 다른 도착점에 도착한적이 있다면) `open` 에 `path` 의 마지막 값(마지막으로 방문했던 좌표)을 삽입하여 경로 탐색을 시작하도록 했습니다. 그것이 아니라면 첫 시작점이 `open` 에 삽입되게 됩니다.

그리고 경로 탐색을 하던 중 현재 Node가 도착점이라면, 현재 Node부터 시작 Node까지 부모를 타고 올라가며 임시 path를 구했습니다.

이렇게 구한 임시 path는 BFS와 마찬가지로 역순으로 구해지기 때문에 `tmp[::-1]`을 하게 되는데, 이전에 구한 경로의 마지막 지점과 `tmp[::-1]`의 시작지점은 겹치게 되기 때문에 `path = path[:-1] + tmp[::-1]` 을 사용하여 최종 경로를 각 iteration 동안 쌓게 됩니다.

그렇게 구한 최종적인 path는 시작점부터 주어진 모든 도착 지점을 통과한 경로가 됩니다.