

Assignment01 - Maze Game

20160768 김홍엽

1. 사용한 라이브러리 및 알고리즘 설명

라이브러리	설명
collections.deque	deque(큐)를 이용하여 BFS의 frontier를 구현함.
heapq	A* 알고리즘에 사용되는 우선순위 큐를 구현하기 위해 사용됨.
copy.deepcopy	mutable한 객체를 복사하기 위해 사용됨.

알고리즘	설명
<pre>def bfs(maze): """ [문제 01] 제시된 stage1의 맵 세가지를 BFS Algorithm을 통해 최단 경로를 return하시오. (20점) """ start_point=maze.startPoint() path=[] ##### Write Your Code Here ##### # BFS 구현을 위한 frontier(큐) 초기화 (point, path_to_point) frontier = deque([(start_point, [start_point])]) # 방문 체크를 위한 변수 초기화 visited = [] while frontier: current_point, current_path = frontier.popleft() cur_x, cur_y = current_point visited.append(current_point) if maze.isObjective(cur_x, cur_y): # GOAL TEST return current_path for neighbor in maze.neighborPoints(cur_x, cur_y): if neighbor not in visited: # 방문 여부 체크 frontier.append((neighbor, current_path + [neighbor])) return path #####</pre>	<ol style="list-style-type: none">1. BFS 구현을 위한 frontier(큐)를 생성한다.<ol style="list-style-type: none">1. frontier에 들어가는 데이터 형식은 ‘좌표, [현재 좌표까지의 경로]’이다.2. 방문 체크를 위한 변수 visited(closed list)를 생성한다.3. frontier에 원소가 존재하지 않을 때까지 다음을 반복한다.<ol style="list-style-type: none">1. frontier에서 원소를 pop한다.2. 방문 체크를 한다.3. 해당 원소의 좌표가 목적지인지 확인한다 (Goal Test). 목적지라면 해당 좌표까지의 경로를 반환하며 프로그램은 종료된다.4. 해당 좌표에서 이동할 수 있고, 아직 방문하지 않은 다음 좌표들을 frontier에 넣는다.<ol style="list-style-type: none">1. 경로를 업데이트 해준다.


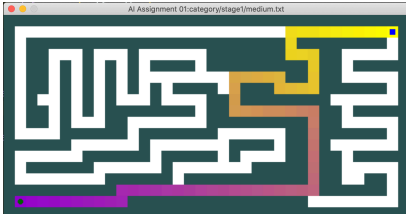
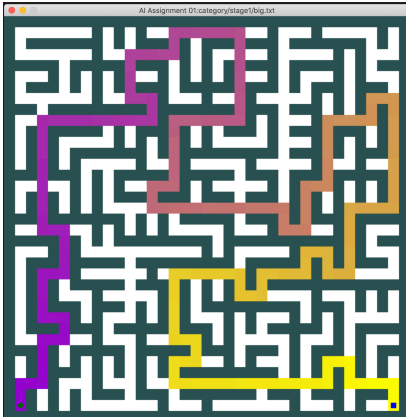
알고리즘	설명
<pre> def astar(maze): """ [문제 02] 제시된 stage1의 맵 세가지를 A* Algorithm을 통해 최단경로를 return하시오.(20점) (Heuristic Function은 위에서 정의한 manhattan_dist function을 사용할 것.) """ start_point=maze.startPoint() end_point=maze.circlePoints()[0] path=[] ##### Write Your Code Here ##### # start_node 초기화 start_node = Node(None, start_point) start_node.obj = end_point # 방문해야 할 목적지 # frontier 초기화 (node, path_to_node) frontier = [] heapq.heappush(frontier, \ (manhattan_dist(start_point, end_point), (start_node, [start_point]))) # 방문 체크를 위한 변수 초기화 visited = dict() while frontier: # f값이 가장 적은 node 선택 및 방문 처리 current_node, current_path = heapq.heappop(frontier)[1] cur_x, cur_y = current_node.location visited[current_node.location] = current_node.g if maze.isObjective(cur_x, cur_y): # GOAL TEST return current_path for neighbor in maze.neighborPoints(cur_x, cur_y): if neighbor not in visited: # 방문 여부 체크 new_node = Node(current_node, neighbor) new_node.g = current_node.g + 1 new_node.h = manhattan_dist(new_node.location, end_point) new_node.f = new_node.g + new_node.h updated_path = current_path + [neighbor] heapq.heappush(frontier, (new_node.f, (new_node, updated_path))) return path </pre>	<ol style="list-style-type: none"> Node class를 통해 start_node를 초기화한다. <ol style="list-style-type: none"> start_node.obj에는 방문해야 할 목적지가 들어간다. heapq를 사용하여 frontier(우선순위 큐)를 생성한다. <ol style="list-style-type: none"> frontier에 들어가는 데이터 형식은 '(노드의 f값, (현재 노드, [현재 노드까지의 경로]))' 노드의 f값이 우선순위의 기준이 된다. 방문 체크를 위한 변수 visited(closed list)를 dictionary형으로 생성한다. <ol style="list-style-type: none"> visited의 key는 노드의 좌표이며, value는 노드까지의 거리이다. frontier에 원소가 존재하지 않을 때까지 다음을 반복한다. <ol style="list-style-type: none"> frontier에서 원소를 pop한다. (f값이 가장 적은 원소) 방문 체크를 한다. 해당 원소의 좌표가 목적지인지 확인한다 (Goal Test). 목적지라면 해당 좌표까지의 경로를 반환하며 프로그램은 종료된다. 해당 좌표에서 이동할 수 있고, 아직 방문하지 않은 다음 좌표들을 가진 노드를 새롭게 frontier에 넣는다. <ol style="list-style-type: none"> 새로운 노드의 g값은 현재 노드의 g값 + 1이다. 새로운 노드의 h(heuristic)값은 현재 노드와 새로운 노드와의 맨해튼 거리이다. 새로운 노드의 f 값은 g + h이다. 경로를 업데이트 해준다.

알고리즘	설명
<pre> def astar_four_circles(maze): """ [문제 03] 제시된 stage2의 맵 세가지를 A* Algorithm을 통해 최단 경로를 return하시오.(30점) (단 Heuristic Function은 위의 stage2_heuristic function을 직접 정의하여 사용해야 한다.) """ end_points=maze.circlePoints() end_points.sort() path=[] ##### Write Your Code Here ##### # start_node 초기화 start_point = maze.startPoint() start_node = Node(None, start_point) start_node.obj = end_points # 방문하지 않은 목적지 # frontier 초기화 (node, path_to_node) frontier = [] heapq.heappush(frontier, (stage2_heuristic(start_node), (start_node, [start_point]))) # 방문 체크를 위한 변수 초기화 visited = dict() while frontier: # f값이 가장 작은 node 선택 및 방문 처리 current_node, current_path = heapq.heappop(frontier)[1] cur_x, cur_y = current_node.location visited[(current_node.location, tuple(current_node.obj))] = current_node.g if goal_test(current_node): # GOAL TEST return current_path for neighbor in maze.neighborPoints(cur_x, cur_y): if neighbor not in current_node.obj: # 다음 노드가 목적지가 아닐 때 next_node = Node(current_node, neighbor) next_node.obj = current_node.obj else: # 다음 노드가 목적지일 때 (방문하지 않은 목적지 수정) new_obj = [] for obj in current_node.obj: if obj != neighbor: new_obj.append(obj) new_obj.sort() next_node = Node(current_node, neighbor) next_node.obj = new_obj if (next_node.location, tuple(next_node.obj)) not in visited: # 방문 여부 체크 next_node.g = current_node.g + 1 next_node.h = stage2_heuristic(next_node) next_node.f = next_node.g + next_node.h updated_path = current_path + [neighbor] heapq.heappush(frontier, (next_node.f, (next_node, updated_path))) return path </pre>	<p>위 astar와 동일하나 다음과 같은 차이가 있다.</p> <ol style="list-style-type: none"> 1. Heuristic function으로 맨해튼 거리가 아닌 직접 정의한 stage2_heuristic을 사용한다. 2. visited의 key는 ‘(노드의 좌표, (현재 노드가 아직 방문하지 않은 목적지))’이다. 3. Goal test는 해당 노드가 모든 목적지를 방문했는지를 확인한다. 4. 다음 노드를 탐색할 때, 다음 노드가 목적지가 아니라면 다음 노드의 obj는 현재 노드의 obj를 그대로 갖는다. 목적지라면 현재 노드의 obj에서 해당 목적지를 뺀 값을 갖는다.


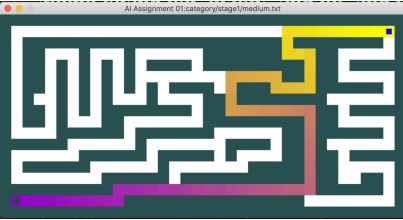
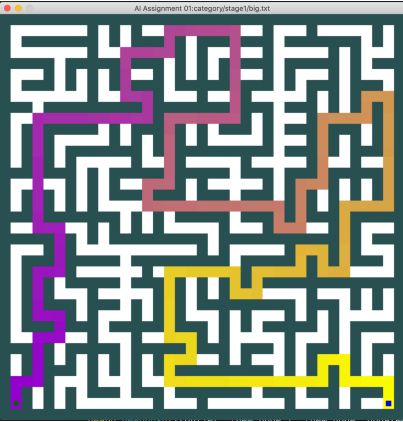
알고리즘	설명
<pre> def astar_many_circles(maze): """ [문제 04] 제시된 stage3의 맵 세가지를 A* Algorithm을 통해 최단 경로를 return하시오.(30점) (단 Heuristic Function은 위의 stage3_heuristic function을 직접 정의하여 사용해야 하고, minimum spanning tree 알고리즘을 활용한 heuristic function이아 한다.) """ end_points= maze.circlePoints() end_points.sort() path=[] ##### Write Your Code Here ##### # start_node 초기화 start_point = maze.startPoint() start_node = Node(None, start_point) start_node.obj = end_points # 방문하지 않은 목적지 # point들의 거리를 저장(cached) cached_distances = [[0 for _ in range(len(maze.mazeRaw[0]))] for _ in range(len(maze.mazeRaw))] # frontier 초기화 (node, path_to_node) frontier = [] heapq.heappush(frontier, (stage3_heuristic(maze, start_node, cached_distances), (start_node, [start_point]))) # 방문 체크를 위한 변수 초기화 visited = dict() while frontier: # frontier의 가장 작은 node 선택 및 방문 처리 current_node, current_path = heapq.heappop(frontier)[1] cur_x, cur_y = current_node.location visited[(current_node.location, tuple(current_node.obj))] = current_node.g if goal_test(current_node): # GOAL TEST return current_path for neighbor in maze.neighborPoints(cur_x, cur_y): if neighbor not in current_node.obj: # 다음 노드가 목적지가 아닐 때 next_node = Node(current_node, neighbor) next_node.obj = current_node.obj else: # 다음 노드가 목적지일 때 (방문하지 않은 목적지 수정) new_obj = [] for obj in current_node.obj: if obj != neighbor: new_obj.append(obj) new_obj.sort() next_node = Node(current_node, neighbor) next_node.obj = new_obj if (next_node.location, tuple(next_node.obj)) not in visited: # 방문 여부 체크 next_node.g = current_node.g + 1 next_node.h = stage3_heuristic(maze, next_node, cached_distances) next_node.f = next_node.g + next_node.h updated_path = current_path + [neighbor] heapq.heappush(frontier, (next_node.f, (next_node, updated_path))) return path </pre>	<p>위 astar_four_circles와 동일하나 다음과 같은 차이가 있다.</p> <ol style="list-style-type: none"> 1. Heuristic function으로 맨해튼 거리가 아닌 직접 정의한 stage3_heuristic을 사용한다. 2. stage3_heuristic에서 사용될 cached_distances가 사용된다. (각 point들의 거리를 저장하는 자료구조)

2. 각 stage별 프로그램 실행 화면


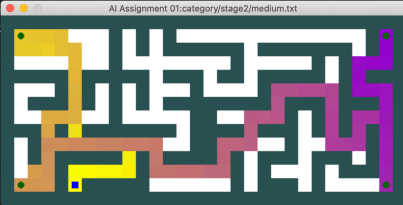
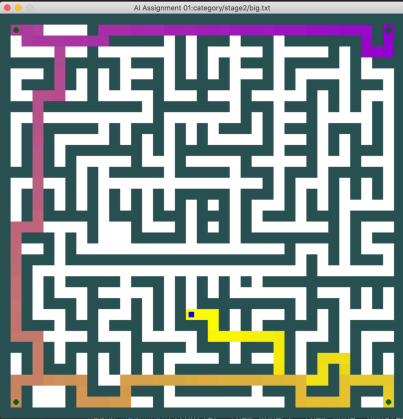
1. bfs - stage1

small	medium	big
 <pre> ===== [bfs results] (1) Path Length: 9 (2) Search States: 16 (3) Execute Time 0.0000648499 seconds ===== </pre>	 <pre> ===== [bfs results] (1) Path Length: 69 (2) Search States: 275 (3) Execute Time 0.0023720264 seconds ===== </pre>	 <pre> ===== [bfs results] (1) Path Length: 211 (2) Search States: 619 (3) Execute Time 0.0096271038 seconds ===== </pre>

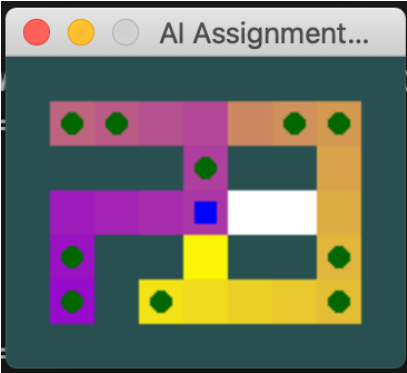
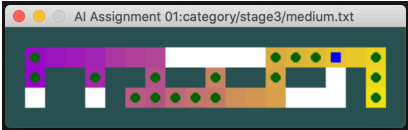

2. astar - stage1

small	medium	Big
 <p>[astar results] (1) Path Length: 9 (2) Search States: 14 (3) Execute Time 0.0001180172 seconds</p>	 <p>[astar results] (1) Path Length: 69 (2) Search States: 224 (3) Execute Time 0.0012431145 seconds</p>	 <p>[astar results] (1) Path Length: 211 (2) Search States: 549 (3) Execute Time 0.0030009747 seconds</p>

3. astar_four_circles - stage2

small	medium	Big
 <p>[astar_four_circles results] (1) Path Length: 29 (2) Search States: 310 (3) Execute Time 0.0061779022 seconds</p>	 <p>[astar_four_circles results] (1) Path Length: 107 (2) Search States: 1740 (3) Execute Time 0.0382390022 seconds</p>	 <p>[astar four circles results] (1) Path Length: 163 (2) Search States: 5434 (3) Execute Time 0.1084930897 seconds</p>

4. astar_many_circles - stage3

small	medium	Big
 <pre> [astar_many_circles results] (1) Path Length: 28 (2) Search States: 31 (3) Execute Time 0.0110988617 seconds </pre>	 <pre> [astar_many_circles results] (1) Path Length: 35 (2) Search States: 37 (3) Execute Time 0.0284531116 seconds </pre>	 <pre> [astar_many_circles results] (1) Path Length: 203 (2) Search States: 8879 (3) Execute Time 9.0336210728 seconds </pre>

3. stage2, stage3의 heuristic function 설명

stage2_heuristic	설명
<pre> def stage2_heuristic(node): left_end_points = deepcopy(node.obj) cur_point = node.location if len(left_end_points) == 0: return 0 max_distance = 0 for end_point in left_end_points: distance = manhattan_dist(cur_point, end_point) if max_distance < distance: max_distance = distance return max_distance </pre>	<ol style="list-style-type: none"> 1. stage2_heuristic은 parameter로 node를 받는다. 2. 현재 노드가 방문하지 않은 목적지가 없다면, heuristic 값으로 0을 반환한다. 3. 아직 방문하지 않은 목적지들과 현재 노드의 거리를 맨 하튼 거리를 통해 각각 구한다. 4. 구한 거리들 중에서 가장 큰 값을 heuristic 값으로 반환한다. <p>해당 heuristic function은 admissible하다. 현재 노드에서 가장 멀리 떨어진 목적지와의 거리는 남은 목적지들을 모두 방문하는 거리보다 절대 크지 않기 때문이다. 가장 멀리 떨어진 목적지를 가는 경로안에 남은 목적지들이 모두 존재한다해도 실제 거리보다 크지 않다.</p> <p>또한 consistent하다. 가장 멀리 떨어진 목적지가 중간에 바뀐다하더라도, 바로 이전 heuristic값이 가장 멀리 떨어진 목적지가 바뀐 현재 노드의 heuristic값에 기존 노드에서 현재 노드까지의 거리를 더한 값보다 크지 않기 때문이다.</p>

stage3_heuristic	설명
<pre>def stage3_heuristic(maze, node, cached_distances): total_cost = 0 cur_x, cur_y = node.location left_end_points = deepcopy(node.obj) if len(node.obj) == 0: return total_cost # 현재 위치부터 가장 가까운 end_point까지의 거리 계산 distances = [] for point in left_end_points: cached = 0 for edge in cached_distances[cur_x][cur_y]: if edge[1] == point: distances.append(edge) cached = 1 break if cached == 0: distance = astar_distance(maze, (cur_x, cur_y), point) distances.append((distance, point)) # cached_distances 업데이트 cached_distances[cur_x][cur_y].append((distance, point)) cached_distances[point[0]][point[1]].append((distance, (cur_x, cur_y))) cost, nearest_end_point = min(distances) total_cost += cost # 남은 end_point들의 mst total_cost += mst(maze, nearest_end_point, left_end_points, cached_distances) return total_cost</pre>	<p>stage3_heuristic은 parameter로 maze, node, cached_distances를 받는다.</p> <p>maze는 astar를 이용해 거리를 계산할 때 사용되며, cached_distances는 코드의 효율성을 위해 계산한 거리를 저장하는 자료구조이다. 거리를 계산하기 전에 이미 계산한 거리가 존재하면 계산하지 않고 cached_distances에서 그 값을 사용한다.</p> <ol style="list-style-type: none"> 1. 아직 방문하지 않은 목적지들 중, 현재 노드에서 가장 가까운 목적지와의 거리를 계산한다. 거리를 계산할 때 astar 알고리즘이 사용되며, 이 과정에서 탐색한 state는 search states에 포함되지 않는다. 2. 가장 가까운 목적지를 시작 포인트로하여 남은 목적지들과의 mst를 구한다. 3. 1, 2의 값을 모두 더하여 heuristic 값으로 반환한다.
<pre>def mst(maze, nearest_end_point, left_end_points, cached_distances): cost_sum=0 ##### Write Your Code Here ##### # Prim Algorithm cur_point = nearest_end_point # 남은 left_end_points들의 거리 계산 distances = calculate_distances(maze, left_end_points, cached_distances) visited = [] visited.append(cur_point) heap = distances[cur_point[0]][cur_point[1]] heapq.heapify(heap) while heap: cost, point = heapq.heappop(heap) if point not in visited: visited.append(point) cost_sum += cost for edge in distances[point[0]][point[1]]: if edge[1] not in visited: heapq.heappush(heap, edge) return cost_sum #####</pre>	<p>mst는 maze, nearest_end_point, left_end_points, cached_distances를 parameter로 받는다.</p> <ol style="list-style-type: none"> 1. calculate_distances 함수를 통해 남은 목적지들간의 거리를 계산한 뒤 distances에 저장한다. 2. nearest_end_point를 시작 포인트로 설정한 뒤, 그 점과 연결된 다른 목적지와의 거리 정보를 담은 리스트를 우선순위 큐로 만들어준다. 3. 우선순위 큐에 원소가 존재하지 않을 때까지 다음을 반복한다. <ol style="list-style-type: none"> 1. 우선순위 큐를 사용하여 거리가 가장 짧은 목적지부터 pop한다. 2. 해당 목적지를 방문하지 않았다면, 방문 체크를 해준 뒤 목적지와의 거리를 더해준다. 3. 해당 목적지와 연결된 목적지들 중 방문하지 않는 목적지를 큐에 넣어준다. 4. 모든 목적지를 연결한 거리의 합을 반환한다.
<pre>def calculate_distances(maze, end_points, cached_distances): vertices = deepcopy(end_points) distances = [[[] for _ in range(len(maze.mazeRow))] for _ in range(len(maze.mazeRow))] for point in end_points: vertices.remove(point) for vertex in vertices: cached = 0 for edge in cached_distances[point[0]][point[1]]: # cached_distances에 존재하는지 확인 if edge[1] == vertex: distances[point[0]][point[1]].append(edge) distances[vertex[0]][vertex[1]].append((edge[0], point)) cached = 1 break if cached == 0: distance = astar_distance(maze, point, vertex) distances[point[0]][point[1]].append((distance, vertex)) distances[vertex[0]][vertex[1]].append((distance, point)) # cached_distances 업데이트 cached_distances[point[0]][point[1]].append((distance, vertex)) cached_distances[vertex[0]][vertex[1]].append((distance, point)) return distances</pre>	<ol style="list-style-type: none"> 1. calculate_distances는 parameter로 받은 end_points간의 거리 정보를 담은 자료구조를 반환한다. 2. 거리를 계산할 때 astar 알고리즘이 사용되었으며, 이 과정에서 탐색한 state는 search states에 포함되지 않는다. 3. cached_distances를 사용하여 이미 계산된 거리는 다시 계산하지 않고 그 값을 사용한다. 새롭게 계산된 거리는 cached_distances에 업데이트한다.

stage3_heuristic	설명
<pre>def astar_distance(maze, start_point, end_point): path=[] start_node = Node(None, start_point) frontier = [] heapq.heappush(frontier, (manhattan_dist(start_point, end_point), (start_node, [start_point]))) visited = dict() while frontier: current_node, current_path = heapq.heappop(frontier)[1] cur_x, cur_y = current_node.location visited[current_node.location] = current_node.g if current_node.location == end_point: return len(current_path) for neighbor in neighborPoints(maze, cur_x, cur_y): if neighbor not in visited: new_node = Node(current_node, neighbor) new_node.g = current_node.g + 1 new_node.h = manhattan_dist(new_node.location, end_point) new_node.f = new_node.g + new_node.h updated_path = current_path + [neighbor] heapq.heappush(frontier, (new_node.f, (new_node, updated_path))) return len(path)</pre>	<p>astar_distance는 기존 astar 알고리즘과 동일하나, 여기서 탐색한 state는 search states에 포함되지 않는다.</p>