#Artificial Intelligence

- First assignment REPORT

9/30/2018

2014004411 컴퓨터 전공
김시완

1. 코드 설명

a. 1층 구현

```
def first_floor():
  file = open("first_floor_input.txt", 'r')
  length = 0
  line = file.readline()
  info = line.split()
  table = [[] for i in range(0,int(info[1]))]

  for i in range(0,int(info[1])):
    table[i] = file.readline()
    table[i] = table[i].strip("₩n")
    table[i] = table[i].split()

  file.close()

  start, end, key = find_index(info, table)

  #time, route = bfs(table, start, key, end)
  time, route = dfs(table, start, key, end)
  #time, route = greedy(table, start, key, end)
  #time, route = a_star(table, start, key, end)

  for index in route:
    length = length + 1
    if(int(table[index[0]][index[1]]) == 4):
      continue
    table[index[0]][index[1]] = 5

  file = open("first_floor_output.txt",'w')

  for i in range(int(info[1])):
    for j in range(int(info[2]) - 1):
      file.write(str(table[i][j]) + ' ')
    file.write(str(table[i][int(info[2]) - 1]) + '₩n')
  file.write('---')
  file.write("₩nlength=" + str(length))
  file.write("₩ntime=" + str(time))
  file.close()
```

open 함수와 readlines 함수를 통해 입력 파일인 first_floor_input.txt 파일을 읽은 다음 table 배열에 값을 저장한 다음, find_index 함수를 통해

시작 위치, key 위치, 종료 위치 값을 찾아 주었다.

find_index 함수는 다음과 같다.

```
def find_index(info, table):
 start = (-1,-1)
 end = (-1,-1)
 key = (-1, -1)
 for i in range(0, int(info[1])):
  for j in range(0, int(info[2])):
   if(table[i][j] == '3'):
    start = (i,j)
   if(table[i][j] == '4'):
    end = (i,j)
   if(table[i][j] == '6'):
    key = (i,j)
 return start, end, key
```

argument로 주어진 배열을 통해 2중 for문을 통해 search하면서 각 위치 값을 찾아주었다.

time값과 시작 위치에서 key 위치를 지나 종료 위치까지 도달하는 경로를 알기 위해 1층에서는 dfs 알고리즘을 사용하였다. dfs 함수를 통해 얻은 route(경로)의 값을 5로 변경해 주었으며, output file인 first_floor_output.txt 파일에 출력해 주기 위해서는 write 함수를 이용하여 작성해 주었다.

1층에서 사용한 dfs 함수는 다음과 같다.

```
def dfs(map, start, key, end):
 stack = queue.LifoQueue()
 time = 0
 route = []
 table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]
 current = start

 while current != key:
  if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) ==  6): #up
   stack.put([(current[0] - 1, current[1])] + route)
  if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 6): #left
   stack.put([(current[0], current[1] - 1)] + route)
  if current[0] != len(table) - 1 and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 6): #down
   stack.put([(current[0] + 1, current[1])] + route)
  if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) == 6): #right
   stack.put([(current[0], current[1] + 1)] + route)

  route = stack.get()
  current = route[0]
  table[current[0]][current[1]] = 1
  time += 1

 stack = queue.LifoQueue()
 table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]

 while current != end:
  if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) ==  4): #up
   stack.put([(current[0] - 1, current[1])] + route)
  if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 4): #left
   stack.put([(current[0], current[1] - 1)] + route)
  if current[0] != len(table) - 1 and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 4): #down
   stack.put([(current[0] + 1, current[1])] + route)
  if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) == 4): #right
   stack.put([(current[0], current[1] + 1)] + route)

  route = stack.get()
  current = route[0]
  table[current[0]][current[1]] = 1
  time += 1

 return time, route
```

argument로 미로 경로를 담고 있는 배열과 함께, 시작 위치, key 위치, 종료 위치를 받았으며 dfs로 경로를 탐색하기 위해 queue.LifoQueue() 함수를 사용하여 stack을 생성해주었다. 생성한 stack에 현재 위치에서 위, 왼, 아래, 오른쪽 순서로 탐색을 하여 길이 존재할 경우 지나온 경로를 담아주었으며 이러한 순서는 시작 위치에서 key가 존재하는 위치와 종료 위치가 오른쪽 아래 방향에 위치하여 다음과 같은 순서를 잡아주었다. 먼저 첫 번째 while문을 통해 시작 위치에서부터 key를 찾기 위해 탐색을 해주었으며 현재 위치에서 사방면으로 길이 있는지 탐색을 한 후(stack에 넣어준 후), 다음 경로로 이동을 하기 위해 get함수를 통해 stack에서 값을 꺼내어 현재 값을 변경시켜주었다. 이때, stack에는 경로가 저장되어 있으므로 stack에서 꺼낸 값은 route 값으로 설정을 해 주었으며, 경로 0번째 index에 최근

위치(현재의 값)이 저장 되어 있으므로 current 값으로 설정 해 주었다. 이렇게 계속 while문을 반복하다 보면 key 값에 위치하게 될 경우 while문이 종료 되고 key 값을 찾을 수 있게 된다. key 값에서 end 위치를 찾는 경우에도 위와 같은 방법으로 while문을 사용하였으며, 다만 2번째 while문을 시작하기 전에 변경된 지나온 거리를 표시해 뒀던 table 값과 stack을 다시 한 번 초기화 시켜준 후 진행하였다.

b. 2층 구현

```
def second_floor():
  file = open("second_floor_input.txt", 'r')
  length = 0
  line = file.readline()
  info = line.split()
  table = [[] for i in range(0,int(info[1]))]

  for i in range(0,int(info[1])):
    table[i] = file.readline()
    table[i] = table[i].strip("₩n")
    table[i] = table[i].split()

  file.close()

  start, end, key = find_index(info, table)

  #time, route = bfs(table, start, key, end)
  time, route = dfs(table, start, key, end)
  #time, route = greedy(table, start, key, end)
  #time, route = a_star(table, start, key, end)

  for index in route:
    length = length + 1
    if(int(table[index[0]][index[1]]) == 4):
      continue
    table[index[0]][index[1]] = 5

  file = open("second_floor_output.txt",'w')

  for i in range(int(info[1])):
    for j in range(int(info[2]) - 1):
      file.write(str(table[i][j]) + ' ')
    file.write(str(table[i][int(info[2]) - 1]) + '₩n')
  file.write('---')
  file.write("₩nlength=" + str(length))
  file.write("₩ntime=" + str(time))
  file.close()
```

2층은 1층과 동일한 알고리즘이 사용되었다. 다만, read file 이름과 write file 이름만 second_floor로 변경을 해주었다.

c. 3층 구현

```
def third_floor():
 file = open("third_floor_input.txt", 'r')
 length = 0
 line = file.readline()
 info = line.split()
 table = [[] for i in range(0,int(info[1]))]

 for i in range(0,int(info[1])):
  table[i] = file.readline()
  table[i] = table[i].strip("₩n")
  table[i] = table[i].split()

 file.close()

 start, end, key = find_index(info, table)

 #time, route = bfs(table, start, key, end)
 #time, route = dfs(table, start, key, end)
 time, route = greedy(table, start, key, end)
 #time, route = a_star(table, start, key, end)

 for index in route:
  length = length + 1
  if(int(table[index[0]][index[1]]) == 4):
   continue
  table[index[0]][index[1]] = 5

 file = open("third_floor_output.txt",'w')

 for i in range(int(info[1])):
  for j in range(int(info[2]) - 1):
   file.write(str(table[i][j]) + ' ')
  file.write(str(table[i][int(info[2]) - 1]) + '₩n')
 file.write('---')
 file.write("₩nlength=" + str(length))
 file.write("₩ntime=" + str(time))
 file.close()
```

3층도 기본적인 함수 구현은 1층, 2층과 동일하나 다만 경로를 탐색하는데 있어 greedy 알고리즘을 사용하였다.

구현한 greedy 함수는 다음과 같다.

```
def greedy(map, start, key, end):
 que = queue.PriorityQueue()
 time = 0
 route = []
 table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]
 que.put((abs_sign(key[0] - start[0]) + abs_sign(key[1] - start[1]), [start] + route))
 current = que.get()[1][0]

 while  current != key:
  if current[0] != (len(table) - 1) and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 6):
   que.put(( abs_sign(key[0] - (current[0] + 1)) + abs_sign(key[1] - current[1]), [(current[0] + 1, current[1])] + route))
  if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) == 6):
   que.put((abs_sign(key[0] - current[0]) + abs_sign(key[1] - (current[1] + 1)), [(current[0], current[1] + 1)] + route))
  if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 6):
   que.put((abs_sign(key[0] - current[0]) + abs_sign(key[1] - (current[1] - 1)), [(current[0], current[1] - 1)] + route))
  if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) ==  6):
   que.put((abs_sign(key[0] -  (current[0] - 1)) + abs_sign(key[1] - current[1]), [(current[0] - 1, current[1])] + route))

  route = que.get()[1]
  current = route[0]
  table[current[0]][current[1]] = 1
  time = time + 1

 que = queue.PriorityQueue()
 table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]

 while current != end:
  if current[0] != (len(table) - 1) and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 4):
   que.put(( abs_sign(end[0] - (current[0] + 1)) + abs_sign(end[1] - current[1]), [(current[0] + 1, current[1])] + route))
  if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) == 4):
   que.put((abs_sign(end[0] - current[0]) + abs_sign(end[1] - (current[1] + 1)), [(current[0], current[1] + 1)] + route))
  if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 4):
   que.put((abs_sign(end[0] - current[0]) + abs_sign(end[1] - (current[1] - 1)), [(current[0], current[1] - 1)] + route))
  if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) ==  4):
   que.put((abs_sign(end[0] -  (current[0] - 1)) + abs_sign(end[1] - current[1]), [(current[0] - 1, current[1])] + route))

  route = que.get()[1]
  current = route[0]
  table[current[0]][current[1]] = 1
  time += 1

 return time, route
```

greedy 함수는 기본적인 구현 방법은 dfs와 비슷하나 dfs는 넣는 순서에 따라 나오는 순서도 결정되는 반면에, queue.PriorityQueue() 함수를 이용하여 지정해둔 우선순위에 따라 경로를 설정해 주었다. 여기서는 목적지까지의 거리를 우선순위의 기준으로 두었으며, 현재 위치에서 목적지와 최대한 가까이 위치한 state로 이동하도록 구현을 하였다. priority queue에 값을 넣어줄 때 1번째 parameter로 abs_sign 함수를 통해 구한 거리 값을

넣어주었으며(우선순위 기준), 2번째 parameter로는 stack에서와 마찬가지로 해당 state까지 오는 경로 값을 넣어주었다. get 함수를 통해 queue에서 값을 꺼낼 때에는 우선순위 기준으로 나오기 때문에 목적지와 가까이 위치한 state로 이동을 하게 된다. 마찬가지로 첫 번째 while문이 종료가 되면 key 위치를 발견 할 수 있으며, 2번째 while문이 종료 될 때 종료 위치를 찾을 수 있게 된다.

목적지에서 현재 위치까지의 거리를 찾기 위해 구현한 abs_sign 함수는 argument로 주어진 값을 절대 값으로 변경해 주는 함수로써 다음과 같다.

```
def abs_sign(a):
 if a >=0:
  return a
 else:
  return -a
```

d. 4층, 5층 구현

```
def fourth_floor():
 file = open("fourth_floor_input.txt", 'r')
 length = 0
 line = file.readline()
 info = line.split()
 table = [[] for i in range(0,int(info[1]))]

 for i in range(0,int(info[1])):
  table[i] = file.readline()
  table[i] = table[i].strip("\n")
  table[i] = table[i].split()

 file.close()

 start, end, key = find_index(info, table)

 #time, route = bfs(table, start, key, end)
 #time, route = dfs(table, start, key, end)
 time, route = greedy(table, start, key, end)
 #time, route = a_star(table, start, key, end)

 for index in route:
  length = length + 1
  if(int(table[index[0]][index[1]]) == 4):
   continue
  table[index[0]][index[1]] = 5

 file = open("fourth_floor_output.txt",'w')

 for i in range(int(info[1])):
  for j in range(int(info[2]) - 1):
   file.write(str(table[i][j]) + ' ')
  file.write(str(table[i][int(info[2]) - 1]) + '\n')
 file.write('---')
 file.write("\nlength=" + str(length))
 file.write("\ntime=" + str(time))
 file.close()
```

```
def fifth_floor():
  file = open("fifth_floor_input.txt", 'r')
  length = 0
  line = file.readline()
  info = line.split()
  table = [[] for i in range(0,int(info[1]))]

  for i in range(0,int(info[1])):
    table[i] = file.readline()
    table[i] = table[i].strip("\n")
    table[i] = table[i].split()

  file.close()

  start, end, key = find_index(info, table)

  #time, route = bfs(table, start, key, end)
  #time, route = dfs(table, start, key, end)
  time, route = greedy(table, start, key, end)
  #time, route = a_star(table, start, key, end)

  for index in route:
    length = length + 1
    if(int(table[index[0]][index[1]]) == 4):
      continue
    table[index[0]][index[1]] = 5

  file = open("fifth_floor_output.txt",'w')

  for i in range(int(info[1])):
    for j in range(int(info[2]) - 1):
      file.write(str(table[i][j]) + ' ')
    file.write(str(table[i][int(info[2]) - 1]) + '\n')
  file.write('---')
  file.write("\nlength=" + str(length))
  file.write("\ntime=" + str(time))
  file.close()
```

4층과 5층에서도 3층과 마찬가지로 greedy 함수를 이용하여 경로를 탐색하였다.

e. bfs 알고리즘

```
def bfs(map, start, key, end):
  que = queue.Queue()
  time = 0
  route = []
  table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]
  current = start
  while current != key:
    if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) ==6): #right
      que.put([(current[0], current[1] + 1)] + route)
    if current[0] != len(table) - 1 and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 6): #down
      que.put([(current[0]+1, current[1])] + route)
    if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 6):  #left
      que.put([(current[0], current[1] - 1)] + route)
    if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) == 6): #up
      que.put([(current[0] - 1, current[1])] + route)

    time += 1
    route = que.get()
    current = route[0]
    table[current[0]][current[1]] = 1

  que = queue.Queue()
  table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]

  while current != end:
    if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) ==4): #right
      que.put([(current[0], current[1] + 1)] + route)
    if current[0] != len(table) - 1 and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 4): #down
      que.put([(current[0]+1, current[1])] + route)
    if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 4):  #left
      que.put([(current[0], current[1] - 1)] + route)
    if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) == 4): #up
      que.put([(current[0] - 1, current[1])] + route)

    time += 1
    route = que.get()
    current = route[0]
    table[current[0]][current[1]] = 1

  return time, route
```

각 층에서 경로를 탐색하는데 사용되지는 않았지만, bfs 함수는 dfs와 달리 stack 대신에 queue를 사용하여 경로를 탐색하기 때문에 위와 같이 구현을 해주었다. 기본적인 탐색 방법은 dfs와 동일한데 다만 state 값들을 queue.Queue() 함수를 통해 생성한 queue에 저장시켜 주었으며 먼저 들어간 값이 먼저 나오기 때문에 먼저 탐색 순위로 설정해주어야 하는 오른쪽 왼쪽에 위치한 state들이 먼저 queue에 들어갈 수 있도록 구현을 해주었다.

f. A* 알고리즘

```
def a_star(map, start, key, end):
  que = queue.PriorityQueue()
  time = 0
  route = []
  table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]
  distance = [[0 for i in range(len(map[0]))] for j in range(len(map))]
  que.put((abs_sign(key[0] - start[0]) + abs_sign(key[1] - start[1]), [start] + route))
  current = que.get()[1][0]

  while  current != key:
    if current[0] != (len(table) - 1) and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 6):
      distance[current[0] + 1][current[1]] = distance[current[0]][current[1]] + 1
      que.put(( abs_sign(key[0] - (current[0] + 1)) + abs_sign(key[1] - current[1]) + distance[current[0] + 1][current[1]], [(current[0] + 1, current[1])] + route))
    if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) == 6):
      distance[current[0]][current[1] + 1] = distance[current[0]][current[1]] + 1
      que.put((abs_sign(key[0] - current[0]) + abs_sign(key[1] - (current[1] + 1)) + distance[current[0]][current[1] + 1], [(current[0], current[1] + 1)] + route))
    if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 6):
      distance[current[0]][current[1] - 1] = distance[current[0]][current[1]] + 1
      que.put((abs_sign(key[0] - current[0]) + abs_sign(key[1] - (current[1] - 1)) + distance[current[0]][current[1] - 1],[(current[0], current[1] - 1)] + route))
    if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) ==  6):
      distance[current[0] - 1][current[1]] = distance[current[0]][current[1]] + 1
      que.put((abs_sign(key[0] -  (current[0] - 1)) + abs_sign(key[1] - current[1]) + distance[current[0] - 1][current[1]], [(current[0] - 1, current[1])] + route))


    route = que.get()[1]
    current = route[0]
    table[current[0]][current[1]] = 1
    time += 1
```

```
que = queue.PriorityQueue()
table = [[map[j][i] for i in range(len(map[0]))] for j in range(len(map))]
distance = [[0 for i in range(len(map[0]))] for j in range(len(map))]

while current != end:
  if current[0] != (len(table) - 1) and (int(table[current[0] + 1][current[1]]) == 2 or int(table[current[0] + 1][current[1]]) == 4):
    distance[current[0] + 1][current[1]] = distance[current[0]][current[1]] + 1
    que.put(( abs_sign(end[0] - (current[0] + 1)) + abs_sign(end[1] - current[1]) + distance[current[0] + 1][current[1]], [(current[0] + 1, current[1])] + route))

  if current[1] != len(table[0]) - 1 and (int(table[current[0]][current[1] + 1]) == 2 or int(table[current[0]][current[1] + 1]) == 4):
    distance[current[0]][current[1] + 1] = distance[current[0]][current[1]] + 1
    que.put((abs_sign(end[0] - current[0]) + abs_sign(end[1] - (current[1] + 1)) + distance[current[0]][current[1] + 1], [(current[0], current[1] + 1)] + route))
  if current[1] != 0 and (int(table[current[0]][current[1] - 1]) == 2 or int(table[current[0]][current[1] - 1]) == 4):
    distance[current[0]][current[1] - 1] = distance[current[0]][current[1]] + 1
    que.put((abs_sign(end[0] - current[0]) + abs_sign(end[1] - (current[1] - 1)) + distance[current[0]][current[1]- 1], [(current[0], current[1] - 1)] + route))
  if current[0] != 0 and (int(table[current[0] - 1][current[1]]) == 2 or int(table[current[0] - 1][current[1]]) == 4):
    distance[current[0] - 1][current[1]] = distance[current[0]][current[1]] + 1
    que.put((abs_sign(end[0] -  (current[0] - 1)) + abs_sign(end[1] - current[1]) + distance[current[0]-1][current[1]], [(current[0] - 1, current[1])] + route))


  route = que.get()[1]
  current = route[0]
  table[current[0]][current[1]] = 1
  time += 1

return time, route
```

A* 알고리즘의 경우도 위의 경로 탐색에서는 사용되지 않았지만, 기본적인 방법은 greedy 알고리즘과 매우 유사하다. 다만, 똑같이 우선순위 큐를 사용하지만 단순히 현재 위치에서 목적지까지의 거리만으로 우선순위를 잡아주었던 greedy와는 달리 현재 위치에서 목적지까지의 거리의 값 + 여태 지나온 거리 값을 더해준 값으로 우선순위를 매겨 탐색 순서를 결정 시켜준다는 점에서 차이점이 존재한다. 때문에 A* 알고리즘을 구현하기 위해서 따로 distance 배열을 만들어 여태 지나온 거리의 값을 따로 저장시켜주어 구현하였다.

2. 사용 알고리즘

1층과 2층을 탐색하는데는 dfs 알고리즘을 사용하였으며 나머지 3층. 4층, 5층을 탐색하는데 있어서는 greedy 알고리즘을 사용하였다.

먼저 IDS 같은 경우에는 depth 0부터 depth 값을 하나 하나 증가시키면서 계속 반복하여 dfs를 진행하기 때문에 반복하여 탐색하는 state가 너무 많이 존재하여 일반 dfs보다 성능이 떨어질 것이라고 판단하여 사용할 알고리즘에서 제외 시켰으며, UCS 같은 경우에는 해당 과제에서는 경로 weight 값이 모두 동일 하기 때문에 bfs와 동일한 성능을 지닐 것이라고 판단을 하여 제외시켰다. 따라서 bfs, dfs, greedy A* 총 4개의 알고리즘을 사용하여 1층에서부터 5층까지의 경로 탐색을 진행 시켜 보았는데 밑에 나와 있는 표와 마찬가지로 1층과 2층에서는 DFS가 가장 좋은 성능을 보였으며 3층에서부터 5층까지는 GREEDY 알고리즘이 가장 좋은 성능을 보였음을 알 수 있었다. 다만 1층에서 5층까지 모두 종료 위치와 key 위치가 시작 위치보다 오른쪽 아래에 존재하였기에 탐색 순서에 있어서는 오른쪽과 왼쪽 경로를 우선시 할 수 있도록 구현을 진행하였다.

3. 실험 결과

a. 각 층별 탐색 length

	TIME
1층	3850
2층	758
3층	554
4층	334
5층	106

b. 알고리즘에 따른 각 층별 탐색 time

	BFS	DFS	GREEDY	A*
1층	6746	5090	5813	6602
2층	1716	850	992	1612
3층	999	732	653	816
4층	591	460	430	560
5층	229	158	120	156