Assignment1

인공지능

2014005114

컴퓨터 전공

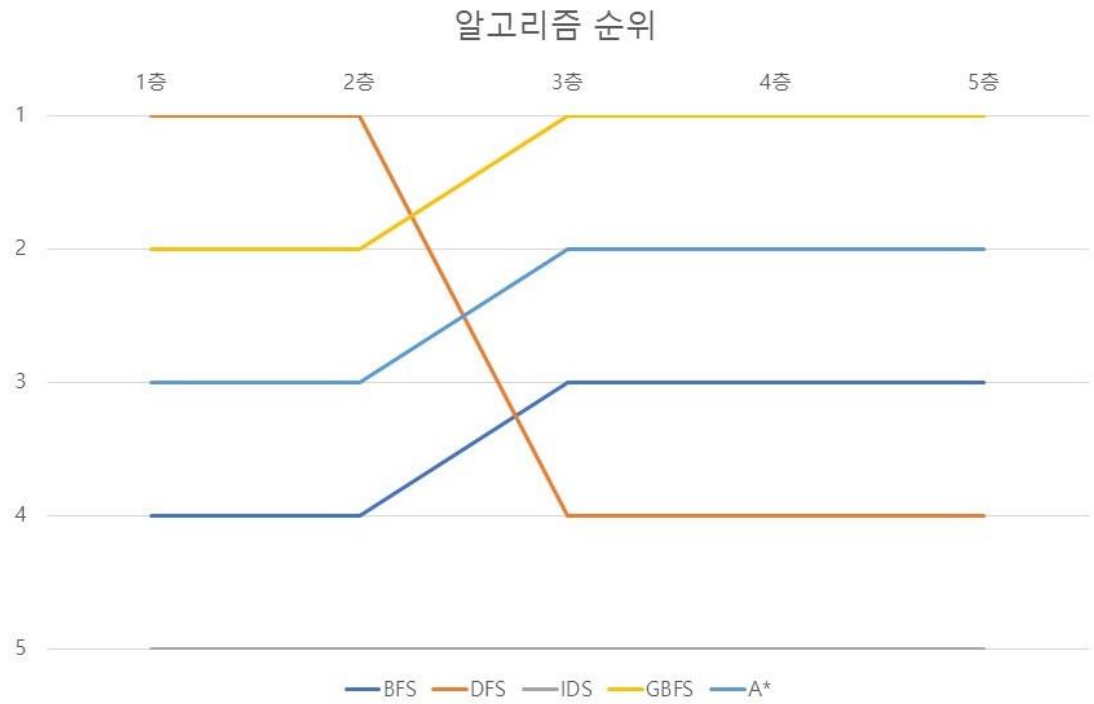
정근욱

1. 목표

- 미로의 최단 경로 찾기
- 적은 노드를 탐색해서 미로를 탈출하기

2. 결과

		1층	2층	3층	4층	5층
Breadth First Search	최단 경로	3850	758	554	334	106
	시간	6740	1716	996	589	224
Depth First Search	최단 경로	3850	758	554	334	106
	시간	5708	982	1330	688	248
Iterative Deepening Search	최단 경로	3850	758	554	334	106
	시간	7588267	306813	143691	54685	5003
Greedy Best First Search	최단 경로	3850	758	554	334	106
	시간	5834	991	651	444	119
A*	최단 경로	3850	758	554	334	106
	시간	6585	1541	742	542	129



Uniform Cost Search 는 step cost 가 상수일 경우 BFS 와 동일하기 때문에 제외하고, 나머지 5 가지 알고리즘에 대해 미로 문제를 풀어보았다. GBFS 가 3, 4, 5 층에서 가장 좋은 결과를 보여주었고 DFS 는 1 층, 2 층에서 좋은 모습을 보여주다가 나머지 층에서는 4 위로 떨어졌다. IDS 는 항상 꼴지를 했고 A*는 BFS 보다 좋은 결과가 나왔다.

특이하게 GBFS 가 A*보다 좋은 모습을 보여주었다. 둘의 차이점은 evaluation 함수를 계산할 때 GBFS 는 heuristic 함수만 사용하지만, A*는 현재 상태까지 오는 비용을 추가적으로 더한다는 것이다. 즉, 전자는 heuristic 만 보고 용감하게 goal 까지 달려가지만 후자는 heuristic 값이 좋더라도 현재 상태까지 오는데 너무 많은 비용이 소모되었다면 일단 heap 에 넣어두고 나중에 추가적으로 탐색한다. 그 결과 GBFS 는 운이 좋을 경우 빠르게 goal 을 찾을 수 있지만 그렇지 않다면 많은 시간이 걸리게 된다. 우리 경우에는 운이 좋았다.

또한 IDS 는 탐색 시간이 BFS 와 크게 차이가 나지 않는다고 배웠는데 결과는 다르게 나왔다. 이러한 현상은 네 방향으로 자유롭게 움직이기보다는 대부분의 경우에 한 방향으로 밖에 움직일 수밖에 없는 미로의 특성 때문이다.

Branch factor 가 b 이고 optimal 이 있는 depth 가 d 일 때 IDS 와 BFS 의 성능이 크게 차이 나지 않는다는 주장은 다음 식 때문이었다.

$$T(BFS) = b + b^2 + \cdots + b^d = \frac{b(b^d - 1)}{b - 1}$$

$$T(IDS) = db + (d-1)b^2 + \cdots + b^d$$

$$b \times T(IDS) = db^2 + (d-1)b^3 + \cdots + b^{d+1}$$

$$(1-b)T(IDS) = db - b^2 - \cdots - b^{d+1} = db - \frac{b^2(b^d-1)}{b-1} = db - b \times T(BFS)$$

$$T(IDS) = -\frac{db}{b-1} + \frac{b}{b-1}T(BFS) \cong \frac{b}{b-1}T(BFS)$$

예를 들어, b 가 10 이고 d 가 5 인경우 T(BFS)는 111,110 T(IDS)는 123,450 이 나와 별차이가 없다. 하지만 이런 예와 다르게 우리 미로는 상당히 적은 b 값을 가지고 있다. 다음 식을 가지고 평균 branch factor 를 측정해보자.

$$\text{avg branch factor} = \frac{\Sigma(\textit{이동 가능 방향} - 1)}{2\ \textit{의 개수}}$$

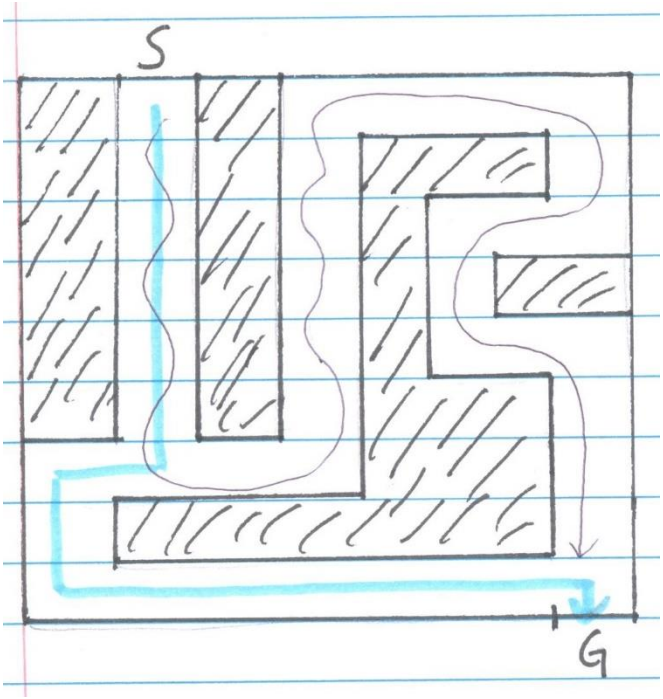
모든 상태는 그 전 상태로 갈 수가 없으므로 이동 가능 방향에서 1 을 빼서 평균 b 를 계산하면 1 층에서 5 층까지 모두 0.99 정도되는 값이 나온다. 이 말은 평균적으로 미로에서는 한 방향으로 밖에 움직일 수 없다는 뜻이다. 따라서 두 알고리즘의 시간은 오히려

$$T(BFS) = 1 + 1 + \cdots + 1 = d$$

$$T(IDS) = d + (d-1) + \cdots + 1 = \frac{\mathrm{d(d+1)}}{2}$$

이렇게 표현하는 것이 정확하고 위 결과에서 큰 시간 차이가 발상한 것이 설명된다.

1, 2 층에서는 DFS 가 3, 4, 5 층에서는 GBFS 가 가장 좋은 모습을 보여주었지만 이 알고리즘들을 사용하지 않을 것이다. 왜냐하면 이들은 optimal 이 보장되지 않기 때문이다. Optimal 을 보장하는 다른 알고리즘들과 같은 최단 거리를 보여주긴 했지만 이는 주어진 미로 특성에 잘 부합했거나 start 부터 goal 까지 가는 경로가 하나밖에 존재하지 않아서 일 것이다.



예를 들어 다음과 같이 goal 까지 갈 수 있는 경로가 2 개 있는 미로에서 DFS 는 optimal 경로를 찾지 못할 수도 있다. 특히 start 가 좌 상단, goal 이 우 하단에 있기 때문에 아래쪽 방향과 오른쪽 방향 움직임에 우선권을 주는 방법을 사용하는 경우에는 optimal 경로를 찾지 못하게 된다.

따라서 optimal 이 보장되는 알고리즘 BFS, IDS, A* 중에 선택해야 한다. 주어진 조건은 메모리 사용량은 고려하지 않고 시간만을 고려하기 때문에 IDS 는 BFS 에 비해 좋은 선택이 될 수 없다. 또한 A*가 BFS 보다 좋은 성능을 보여줬으므로 A*를 선택하자.

3. Heuristic

좋은 Heuristic 함수를 만들기 위해 manhattan distance(이하 MD)와 Euclidean distance(이하 ED)를 결합했다. MD 는 시작 지점과 목표 지점의 가로 세로 길이의 합이고 ED 는 둘 사이의 직선거리이다. 사용한 만든 공식은 다음과 같다.

$$h(n) = manhattandist + \left(\frac{euclideandist}{manhattandist} - 1\right)$$

두 상태가 동일한 MD 를 가지고 있더라도 목표까지의 ED 가 더 작은 상태에 낮은 h 값을 주기 위해 위 식을 만들었다. 동시에 A* 알고리즘이 optimal 하기 위해서는

$$h \le h^* \left(h^* \text{는} goal \text{까지의최단거리}\right)$$

를 만족해야 한다. 이미 $MD \le h^*$는 성립하고

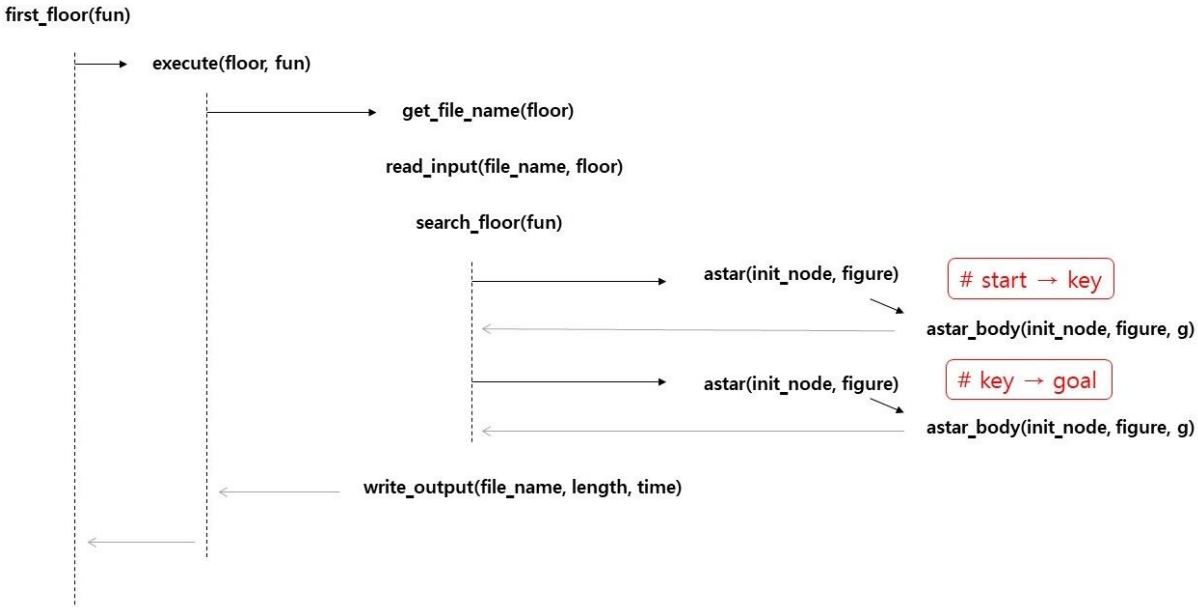
$$ED \le MD \Leftrightarrow 0 \le \frac{ED}{MD} \le 1 \Leftrightarrow -1 \le \frac{ED}{MD} - 1 \le 0$$

$$MD + \frac{ED}{MD} - 1 \le MD \le h^*$$

하기 때문에 optimal 을 만족한다. 이렇게 바뀐 heuristic 을 사용하였더니 모든 층에서 시간이 감소하였다.

	1층	2층	3층	4층	5층
MD	6604	1614	818	562	158
MD+(ED/MD-1)	6585	1541	742	542	129

4. 코드



- 전역 변수

```
max_m = 0
max_n = 0
data = []
Node = namedtuple("Node", "index prev depth")
```

input 파일에서 읽은 m 값과 n 값을 max_m, max_n 전역 변수에 저장하고 미로의 구조는 data 전역 변수에 저장한다. 탐색 과정에서 노드 정보를 저장하기 위해 index, prev, depth 속성을 가진 Node 네임드 튜플 객체를 정의한다.

- 함수 first_floor

```
def first_floor(fun):
    execute(1, fun)
```

✓ 인자: fun - 알고리즘 함수, 여기서는 astar 함수를 사용한다.

✓ 해당하는 층수를 인자로 넣어 execute 함수를 호출한다. 층 별로 함수가 존재한다.

- 함수 execute

```
def execute(floor, fun):
    """Read input and start searching. Write result to the output file"""
    in_file_name, out_file_name = get_file_name(floor)
    read_input(in_file_name, floor)
    length, time = search_floor(fun)
    write_output(out_file_name, length, time)
```

- ✓ 인자: floor – 탐색하려는 미로의 층수, fun - 알고리즘 함수
- ✓ 함수 get_file_name, read_input, search_floor, write_output 을 호출한다.

- 함수 get_file_name

```
def get_file_name(floor):
    """Get floor number and return input and output filename
    if floor == 1:
        in_file_name = "first_floor_input.txt"
    elif floor == 2:
        in_file_name = "second_floor_input.txt"
    elif floor == 3:
        in_file_name = "third_floor_input.txt"
    elif floor == 4:
        in_file_name = "fourth_floor_input.txt"
    elif floor == 5:
        in_file_name = "fifth_floor_input.txt"
    else:
        print("ERROR: wrong floor")
        sys.exit(-1)

    out_file_name = in_file_name.replace("input", "output")
    return in_file_name, out_file_name
```

- ✓ 인자: floor – 탐색하려는 미로의 층수
- ✓ 반환값: in_file_name – 층에 따른 input 파일 이름, out_file_name – 층에 따른 output 파일 이름
- ✓ 층에 따라 input, output 파일 이름을 결정하고 반환한다. 이때 floor 매개 변수가 1 부터 5 사이의 정수가 아니라면 에러를 출력한다.

- 함수 read_input

```
def read_input(file_name, floor):
    """Read input file and save meta data and dat in global variable."""
    global max_m, max_n, data

    with open(file_name, "r") as f:
        # Read first line and get metadata of maze.
        metadata = f.readline()
        floor_in_file, max_m, max_n =  metadata.split()
        floor_in_file, max_m, max_n = int(floor_in_file), int(max_m), int(max_n)

        # Check file has different floor number with input.
        if floor != int(floor_in_file):
            print("ERROR: floor is different with floor_in_file")
            sys.exit(-1)

        # Read rest of the file and get data of maze.
        data = []
        for line in f:
            data.append([int(i) for i in line.split()])

        # Check data have diffent m and n with metadata.
        if (len(data) != max_m or
            any([len(data[i]) != max_n for i in range(len(data))])):
            print("ERROR: wrong m and n")
            sys.exit(-1)

    print("floor:", floor, "m:",max_m, "n:", max_n, end=' ')
```

- ✓ 인자: file_name – input 파일 이름, floor – 탐색하려는 미로의 층수
- ✓ input 파일을 읽어서 미로의 가로 세로 길이를 전역 변수 max_m, max_n 에 저장하고 미로의 구조를 data 에 저장한다. 인자로 입력 받은 층수가 파일의 층수와 다르거나, 미로의 구조의 m, n 값이 파일과 다르면 에러를 출력한다.

- 함수 write_output

```
def write_output(file_name, length, time):
    """Write changed data to the file."""
    with open(file_name, "w") as f:
        for line in data:
            new_line = ' '.join([str(i) for i in line]) + '\n'
            f.write(new_line)

        f.write("---\n")
        f.write("length=" + str(length) + "\n")
        f.write("time=" + str(time) + "\n")

    print("length:", length, "time:", time)
```

- ✓ 인자: file_name – output 파일 이름, length – 탐색한 최단 거리, time – 탐색한 시간
- ✓ 미로 탐색 과정에서 바뀐 전역변수 data 와 최단 거리, 시간을 output 파일에 쓴다.

- 함수 search_floor

```
def search_floor(fun):
    """Search key first and goal next. Calculate length ar
    global data

    start_fig = 3
    key_fig = 6
    goal_fig = 4

    # Initialize start index
    start_idx = find_index(start_fig)
    start_node = Node(start_idx, None, 0)

    # Select algorithm and find key and goal
    key_time, key_node = fun(start_node, key_fig)
    new_start_node = Node(key_node.index, None, 0)
    goal_time, goal_node = fun(new_start_node, goal_fig)

    length = key_node.depth + goal_node.depth
    time = key_time + goal_time

    # Calculate optimal path and change data
    path = set()
    path |= (backtrace(key_node))
    path |= backtrace(goal_node)
    path -= set([start_node.index, goal_node.index])
    for a, b in path:
        data[a][b] = 5

    return length, time
```

  - ✓ 인자: fun – 알고리즘 함수
  - ✓ 반환값: length – 탐색한 최단 거리, time – 탐색한 시간
  - ✓ 인자로 받은 함수를 start 부터 key 까지, key 부터 goal 까지 두 번 호출해서 각각의 최단거리와 시간을 구한 뒤 합친다. 또 각각의 최단 경로를 5 로 바꿔줘야 하기 때문에 두 경로를 합 한 뒤에 data 변수의 값을 바꾼다.

- 함수 astar

```
def astar(init_node, figure):
    return astar_body(init_node, figure, lambda node : node.depth)
```

  - ✓ 인자: init_node – 탐색 시작 노드, figure – 탐색 목표
  - ✓ 반환값: astar_body 함수의 반환값
  - ✓ 현재 상태까지의 비용, 즉 노드의 depth 를 반환하는 함수를 인자로 넣어서 astar_body

함수를 호출한다.

- 함수 astar_body

```
def astar_body(init_node, figure, g):
    """A* algorithm. Return explored indices and goal index."""
    explored = set()
    fig_idx = find_index(figure)

    # Push tuple (f, idx) to the heap
    h = [(f(init_node, fig_idx, g), init_node)]
    while h:
        elm = heapq.heappop(h)
        explored.add(elm[1].index)

        # Goal test
        m, n = elm[1].index
        if data[m][n] == figure:
            return len(explored), elm[1]

        for c in find_successor(elm[1], figure):
            if c.index not in (explored or [x.index for _,x in h]):
                heapq.heappush(h, (f(c, fig_idx, g), c))
    else:
        print("FAILED: cannot found figure")
        sys.exit(-1)
```

  - ✓ 인자: init_node – 탐색 시작 노드, figure – 탐색 목표, g – g 함수
  - ✓ 반환값: len(explored) – 최단거리, elm[1] – 탐색 목표 노드의 네임드 튜플 객체
  - ✓ A* 알고리즘이 구현된 함수다. heapq 라이브러리를 사용해서 가장 좋은 f 함수 값을 가진 노드를 리스트에서 pop 한다. 초기에는 먼저 시작 노드를 push 한다. 힙에서 아이템을 pop 한 뒤 목표가 맞는지 확인하고 아니라면 다음 노드를 힙에 push 한다. 이 과정을 힙이 빌 때까지 반복한다.

- 함수 f

```
def f(node, fig, g):
    """Returns g+h value.

    g : cost from start to now(depth)
    h : heuristic function(MD+ED/MD-1)
    """
    m, n = node.index
    p, q = fig

    h = abs(m-p)+abs(n-q)
    h += ((m-p)**2+(n-q)**2)**0.5 - 1

    return g(node) + h
```

- ✓ 인자: node – evaluation 함수를 계산할 노드, fig – 목표가 위치한 인덱스, g – g 함수
- ✓ 반환값: g(node) + h
- ✓ 인자로 받은 함수를 이용해 구한 g 값과 heuristic 을 이용해 구한 h 값을 더해서 반환한다.

5. 실행

```
if __name__ == "__main__":
    algorithm =  astar
    first_floor(algorithm)
    second_floor(algorithm)
    third_floor(algorithm)
    fourth_floor(algorithm)
    fifth_floor(algorithm)
```

- 사용할 알고리즘인 astar 를 인자로 해서 각 층에 해당하는 함수를 호출한다.
- 각 함수는 함수 이름에 적힌 층에 해당하는 이름을 가진 input 파일만 읽을 수 있다.
- 각 함수 이름에 적힌 층과 파일 첫번째 줄에 있는 층수는 일치해야 한다.
- 파일 첫번째 줄의 m, n 값과 나머지 줄에 있는 미로 정보는 일치해야 한다.