2016026026 컴퓨터 전공 최솔비

인공지능 과제 1

IT/BT 탈출하기

1. 코드 설명

1) 자료구조

```
12  # 각각의 node 를 나타내는 자료구조입니다. 미로의 node들은 이중 배열로 저장합니다.
13  class Node():
14
15      def __init__(self, x, y, value):
16
17          self.xpos = x
18          self.ypos = y
19          self.state = value
20
21          self.parentNode = None
22          self.childNodes =[]
23
24          #key를 찾고 난 다음에 아래의 값들은 초기화 되어야합니다.
25          self.heuristicValue = 1000
26          self.movedDistance = 0
27
```

미로에서의 각각의 위치는 Node 라는 커스텀 클래스로 관리합니다. 클래스의 인스턴스 멤버들은 다음을 포함합니다.

xpos, ypos: xy좌표

State: 노드가 가지고 있는 상태(1,2,3,4,5,6)

parentNode: 부모노드(이전에 위치해 있던 곳)

childNodes: 자식 노드 리스트(갈수 있는 노드)

멤버 함수 seekChildNodes(self, arrValue)로 부모-자식 관계를 등록합니다.

heuristicValue: 휴리스틱 함수의 값. 멤버 함수 Heuristic(self, goal) 으로 업데이트합니다.

movedDistance: 출발지점, 혹은 키를 발견한 지점(root Node)부터 움직인 거리.

멤버 함수 setMovedDistance(self)로 업데이트 합니다.

HeuristicValue 와 movedDistance 는 informed search 인 greedy best first search 알고리즘과 A* 알고리즘을 구현하기 위해 필요한 정보입니다. 각각의 노드들은 read_and_print_file() 함수에서 자료를 받아 올 때 초기화 되며 이중리스트의 형태로 관리합니다. 또한 내부적으로 부모-자식 관계를 형성하여 tree search를 할 수 있습니다.

read_and_print_file() 호출 다음에 find_points()함수를 불러오게 됩니다. 해당 함수에서는 미로의 출발점, 키의 위치, 도착점을 찾아 리턴합니다. Uninformed search에서는 출발점만, informed search 에서는 키의 위치와 도착점에 대한 정보를 이용하여 알고리즘을 수행하게 됩니다.

2) 미로 찾기 알고리즘

미로 찾기 알고리즘은 uninformed search 인 BFS, Iterative Deepening Search와 informed search 인 Greedy Best First Search , A* search로 총 4가지를 구현하였습니다. 내용은 다음과 같습니다.

A. Breadth First Search & Iterative Deepening Search

```
102  def BFS(arrInform, arrValue, start_point):
103
104      time = 0
105      dq = deque()
106      # 큐에 시작점을 넣어줍니다.
107      dq.append(start_point)
108
109      while dq:   # 큐가 비어있지 않은 동안
110          here = dq.popleft()     # pop은 방문하는 것
111          here.setMovedDistance()
112          # print(str(here.xpos)+" " + str(here.ypos)+" "+str(here.state) )
113
114          time += 1
115
116          # 만일 키 값이라면, 탐색을 종료하고 다시 시작합니다.
117          if here.state == 6:
118              break
119
120          # 갈수 있는 길을 탐색하여 자식 노드에 등록 후 큐에 넣도록 합니다.
121          here.seekChildNodes(arrValue)
122          for child in here.childNodes:
123              dq.append(child)
124
125      key_length = here.backtrackPath()
126      here.parentNode = None
127      dq = deque()
128
129      # key에서부터 목적지까지의 탐색을 다시 시작합니다.
130      dq.append(here)
131
132      while dq:
133          here = dq.popleft()
134          here.setMovedDistance()
135          time +=1
136
137          # 만일 goal 값이라면, 탐색을 종료합니다.
138          if here.state == 4:
139              break
140
141          here.seekChildNodes(arrValue)
142          for child in here.childNodes:
143              dq.append(child)
144
145      goal_length = here.parentNode.backtrackPath() + 1
146      whole_length = key_length + goal_length
147
148      print("length :" + str(whole_length))
149      print("time: " + str(time))
150
151      return arrValue, whole_length, time
152
```

파이썬 collections 라이브러리의 deque(double ended queue)자료구조를 이용하여 구현하였습니다. BFS 와 IDS의 구현 방법은 유사하며, BFS 는 deque 를 큐로, IDS는 스택으로 사용하게 됩니다. 시작점을 큐에 push한 다음 큐가 비어있지 않은 동안 자식노드를 등록하여 큐에 넣는 방식 입니다. key를 찾아야 하기 때문에 같은 알고리즘을 처음에는 key를 target으로 실행한 다음, 다음은 goal 을 target 으로 실행하게 됩니다. 키값을 찾은 후는 클래스 멤버함수인 backtrackPath()를 이용하여 경로를 5로 바꾼 다음, 현재 위치(키값)의 부모 노드를 다시 none 으로 설정하고 큐를 비워주어야 합니다. 이후 같은 알고리즘을 goal 값에 대해 시행합니다. 경로가 5로 바뀐 이중 리스트와 경로 길이, 탐색시간을 리턴하게 됩니다.

IDS 함수에서는 큐대신 스택을 사용하며, depth_limit 이라는 변수를 증가시키며 반복적으로 DFS 를 수행하게 됩니다. 나머지는 BFS 와 동일합니다.

B. Greedy Best First Search & A* Algorithm

```
# Greedy best first search 는 휴리스틱 값만 고려합니다.
def greedyBestFirst(arrInform, arrValue, start_point, key_point, goal_point):

    time = 0
    pq = []

    # 우선순위 큐에 시작점을 넣어줍니다.
    # 우선순위는 휴리스틱 값으로 설정합니다.
    start_point.heuristic(key_point)
    heapq.heappush(pq, (start_point.heuristicValue, start_point))

    while pq:   #priority queue 가 비어있지 않은 동안

        # 휴리스틱 값이 가장 작은 node 부터 pop 하게 됩니다.
        idle, here = heapq.heappop(pq)
        time += 1

        # 만일 키 값이라면, 탐색을 종료하고 다시 시작합니다.
        if here.state == 6:
            break

        # 키 값이 아니라면, 갈수 있는 길을 탐색하여 자식 노드에 등록 후 큐에 넣도록 합니다.
        here.seekChildNodes(arrValue)
        for child in here.childNodes:
            # 모든 갈 수 있는 child node 에 대해서 휴리스틱 함수를 계산한 후에 우선순위 큐에 넣도록 합니다.
            child.setMovedDistance()
            child.heuristic(key_point)
            heapq.heappush(pq, (child.heuristicValue, child))

    # 백트래킹하여 경로를 표시합니다.
    key_length = here.backtrackPath()
    here.parentNode = None
    pq = []


    # key에서부터 목적지까지의 탐색을 다시 시작합니다.
    here.heuristic(goal_point)
    heapq.heappush(pq, (here.heuristicValue, here))

    while pq:
        idle, here = heapq.heappop(pq)
        time += 1

        # 만일 goal 값이라면, 탐색을 종료합니다.
        if here.state == 4:
            break

        here.seekChildNodes(arrValue)
        for child in here.childNodes:
            child.setMovedDistance()
            child.heuristic(goal_point)
            heapq.heappush(pq, (child.heuristicValue, child))

    goal_length = here.parentNode.backtrackPath() + 1
    whole_length = key_length + goal_length


    print("length :" + str(whole_length))
    print("time: " + str(time))

    return arrValue, whole_length, time
```

파이썬 내장라이브러리 가운데 heapq 를 이용한 priority queue 를 사용하여 구현하였습니다. Greedy Best First Search와 A* Algorithm의 구현은 유사하지만 전자는 휴리스틱 함수만을 고려하며, 후자는 지금까지 이동한 거리 또한 고려하게 됩니다. Greedy Best First Search는 시작점을 우선순위 큐에 넣은 후, while 문 안에서 현재 노드에서 갈수 있는 곳(프론티어)의 휴리스틱 값을 계산 하여 이를 우선순위로 큐에 push 하게 됩니다. 다음 방문 노드는 우선순위 큐에서 pop한 (함수값이 가장 작은) 노드가 됩니다. 반면, A* 알고리즘은 child.heuristicValue + child.movedDistance를 우선순위로 지정합니다.

3) 각 층 함수

```
453  def first_floor():
454
455      #arrInform은 각 층의 미로에 대한 행과 열의 정보,
456      #arrValue 는 각 층의 미로 구조에 대한 정보를 담고 있습니다.
457
458      arrInform, arrValue = read_and_print_file("first_floor_input.txt")
459      start, key, goal = find_points(arrInform, arrValue)
460
461      changedArr, length, time = greedyBestFirst(arrInform, arrValue, start, key, goal)
462
463      f = open(os.path.expanduser("first_floor_output.txt"), 'w', encoding='utf8')
464
465      f.write( arrToString(arrInform, changedArr))
466      f.write("---\nlength="+str(length)+"\ntime="+str(time))
467
468      f.close()
```

각 층에서 미로찾기 알고리즘을 수행하는 함수 입니다. 인자와 리턴 값은 없기에 메인함수에서 first_floor() 과 같이 호출합니다. 위에서 설명한 바와 같이 read_and_print_file()함수를 통해 미로에 대한 정보를 받아온 후 find_points() 함수를 이용하여 시작점, 키의 위치, 도착점을 찾습니다. 이후 미로찾기 알고리즘을 적용하여 결과 값을 output 파일로 출력합니다. 5개의 층 모두 greedy best first search 알고리즘이 최단시간 결과값이 나왔기에 이를 적용하였습니다.

2. 출력 결과 분석 및 알고리즘 적용

5개의 층에 대해서 Breadth First Search, Iterative Deepening Search, Greedy Best First Search, A* Algorithm을 각각 적용해 보았습니다. 크기가 가장 작은 미로인 5층의 출력 결과 예시는 다음과 같습니다.

```
1. Answered by BFS:
length :106
time: 236
1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 5 1 2 2 5 5 5 1 5 5 5 5 5 1 2 2 2 1 2 1
1 5 1 1 1 5 1 5 1 5 1 1 1 5 1 2 1 2 1 2 1
1 5 5 5 5 5 1 5 1 5 5 5 1 5 1 2 1 2 2 2 1
1 1 1 1 1 1 1 5 1 1 1 5 1 5 1 2 1 1 1 2 1
1 2 2 2 2 2 1 5 5 5 1 5 1 5 1 2 2 2 1 2 1
1 2 1 1 1 2 1 1 1 5 1 5 1 5 1 1 1 2 1 2 1
1 2 2 2 1 2 2 2 1 5 5 5 1 5 5 5 1 2 1 2 1
1 2 1 2 1 1 1 2 1 1 1 1 1 1 1 5 1 2 1 1 1
1 2 1 2 1 2 2 2 1 5 5 5 5 5 1 5 1 2 2 2 1
1 2 1 2 1 1 1 2 1 5 1 5 1 1 1 5 1 2 1 2 1
1 2 1 2 2 2 1 2 1 5 1 5 5 5 5 5 1 2 1 2 1
1 2 1 1 1 2 1 2 1 5 1 1 1 1 1 1 1 1 2 1 1
1 2 1 2 1 2 1 2 1 5 5 5 1 2 2 5 5 5 5 5 1
1 2 1 2 1 2 1 1 1 1 1 5 1 2 1 5 1 1 1 5 1
1 2 1 2 1 2 1 5 5 5 5 5 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 1 1 1 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 5 5 5 1 2 1 5 1 5 5 5 1
1 2 1 2 1 2 1 5 1 5 1 5 1 1 1 5 1 5 1 1 1
1 2 2 2 1 2 2 5 5 5 1 5 5 5 5 5 1 5 5 5 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 1

2. Answered by IDS:
length :106
time: 5118
1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 5 1 2 2 5 5 5 1 5 5 5 5 5 1 2 2 2 1 2 1
1 5 1 1 1 5 1 5 1 5 1 1 1 5 1 2 1 2 1 2 1
1 5 5 5 5 5 1 5 1 5 5 5 1 5 1 2 1 2 2 2 1
1 1 1 1 1 1 1 5 1 1 1 5 1 5 1 2 1 1 1 2 1
1 2 2 2 2 2 1 5 5 5 1 5 1 5 1 2 2 2 1 2 1
1 2 1 1 1 2 1 1 1 5 1 5 1 5 1 1 1 2 1 2 1
1 2 2 2 1 2 2 2 1 5 5 5 1 5 5 5 1 2 1 2 1
1 2 1 2 1 1 1 2 1 1 1 1 1 1 1 5 1 2 1 1 1
1 2 1 2 1 2 2 2 1 5 5 5 5 5 1 5 1 2 2 2 1
1 2 1 2 1 1 1 2 1 5 1 5 1 1 1 5 1 2 1 2 1
1 2 1 2 2 2 1 2 1 5 1 5 5 5 5 5 1 2 1 2 1
1 2 1 1 1 2 1 2 1 5 1 1 1 1 1 1 1 1 2 1 1
1 2 1 2 1 2 1 2 1 5 5 5 1 2 2 5 5 5 5 5 1
1 2 1 2 1 2 1 1 1 1 1 5 1 2 1 5 1 1 1 5 1
1 2 1 2 1 2 1 5 5 5 5 5 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 1 1 1 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 5 5 5 1 2 1 5 1 5 5 5 1
1 2 1 2 1 2 1 5 1 5 1 5 1 1 1 5 1 5 1 1 1
1 2 2 2 1 2 2 5 5 5 1 5 5 5 5 5 1 5 5 5 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 1
```

```
3. Answered by Greedy Best First search:
length :106
time: 121
1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 5 1 2 2 5 5 5 1 5 5 5 5 5 1 2 2 2 1 2 1
1 5 1 1 1 5 1 5 1 5 1 1 1 5 1 2 1 2 1 2 1
1 5 5 5 5 5 1 5 1 5 5 5 1 5 1 2 1 2 2 2 1
1 1 1 1 1 1 1 5 1 1 1 5 1 5 1 2 1 1 1 2 1
1 2 2 2 2 2 1 5 5 5 1 5 1 5 1 2 2 2 1 2 1
1 2 1 1 1 2 1 1 1 5 1 5 1 5 1 1 1 2 1 2 1
1 2 2 2 1 2 2 2 1 5 5 5 1 5 5 5 1 2 1 2 1
1 2 1 2 1 1 1 2 1 1 1 1 1 1 1 5 1 2 1 1 1
1 2 1 2 1 2 2 2 1 5 5 5 5 5 1 5 1 2 2 2 1
1 2 1 2 1 1 1 2 1 5 1 5 1 1 1 5 1 2 1 2 1
1 2 1 2 2 2 1 2 1 5 1 5 5 5 5 5 1 2 1 2 1
1 2 1 1 1 2 1 2 1 5 1 1 1 1 1 1 1 1 2 1
1 2 1 2 1 2 1 2 1 5 5 5 1 2 2 5 5 5 5 5 1
1 2 1 2 1 2 1 1 1 1 1 5 1 2 1 5 1 1 1 5 1
1 2 1 2 1 2 1 5 5 5 5 5 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 1 1 1 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 5 5 5 1 2 1 5 1 5 5 5 1
1 2 1 2 1 2 1 5 1 5 1 5 1 1 1 5 1 5 1 1 1
1 2 2 2 1 2 2 5 5 5 1 5 5 5 5 5 1 5 5 5 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 1
```

```
4. Answered by A* search:
length :106
time: 151
1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 5 1 2 2 5 5 5 1 5 5 5 5 5 1 2 2 2 1 2 1
1 5 1 1 1 5 1 5 1 5 1 1 1 5 1 2 1 2 1 2 1
1 5 5 5 5 5 1 5 1 5 5 5 1 5 1 2 1 2 2 2 1
1 1 1 1 1 1 1 5 1 1 1 5 1 5 1 2 1 1 1 2 1
1 2 2 2 2 2 1 5 5 5 1 5 1 5 1 2 2 2 1 2 1
1 2 1 1 1 2 1 1 1 5 1 5 1 5 1 1 1 2 1 2 1
1 2 2 2 1 2 2 2 1 5 5 5 1 5 5 5 1 2 1 2 1
1 2 1 2 1 1 1 2 1 1 1 1 1 1 1 5 1 2 1 1 1
1 2 1 2 1 2 2 2 1 5 5 5 5 5 1 5 1 2 2 2 1
1 2 1 2 1 1 1 2 1 5 1 5 1 1 1 5 1 2 1 2 1
1 2 1 2 2 2 1 2 1 5 1 5 5 5 5 5 1 2 1 2 1
1 2 1 1 1 2 1 2 1 5 1 1 1 1 1 1 1 1 2 1
1 2 1 2 1 2 1 2 1 5 5 5 1 2 2 5 5 5 5 5 1
1 2 1 2 1 2 1 1 1 1 1 5 1 2 1 5 1 1 1 5 1
1 2 1 2 1 2 1 5 5 5 5 5 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 1 1 1 1 2 1 5 1 2 1 5 1
1 2 1 2 1 2 1 5 1 5 5 5 1 2 1 5 1 5 5 5 1
1 2 1 2 1 2 1 5 1 5 1 5 1 1 1 5 1 5 1 1 1
1 2 2 2 1 2 2 5 5 5 1 5 5 5 5 5 1 5 5 5 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 1
```

경로는 5로 바뀌었으며 4개의 알고리즘 모두 같은 경로와 length를 출력하는 것을 확인 할 수 있습니다. 탐색을 위해 사용된 time은 각 알고리즘마다 차이가 있었으며 이를 표로 정리하였습니다.

	BFS	IDS	Greedy	A*	LENGTH
1층	6747	7595488	5831	6609	3850
2층	1723	308396	1008	1614	758
3층	1006	144839	659	822	554
4층	694	60025	433	618	334
5층	236	5118	121	151	106

5개의 층 모두에서 Greedy Best First Search 알고리즘이 가장 적게 탐색하는 것을 확인할 수 있습니다. Greedy Best First Search는 모든 경우에 최적해를 찾지는 못하지만(Optimality를 보장하지 않습니다), 제시된 미로 5개에서는 가장 좋은 성능을 보였기에 해당 알고리즘을 *_floor() 함수에 이식하였습니다. 이 외에도 A* search의 성능이 두번째, BFS 가 세번째로 뛰어난 것을 확인 할 수 있었습니다.