**과제1 – IT/BT 탈출하기**

컴퓨터전공

2014005014

이지민

1. 코드 설명

1) **global 변수**

・maze: 2차원 배열로 미로를 저장한다.

・m, n: 미로의 행과 열을 저장한다.

2) **State class**

미로의 위치를 가지고 tree구조로 관리한다.

・member 변수

- posRow, posCol: 미로에서 한 위치를 나타내는 행과 열을 의미한다.

- cost: root에서부터의 거리를 가리킨다. 즉 tree의 깊이를 의미한다.

- left, right, up, down: 각 방향을 가리키는 위치의 state를 나타낸다. 즉 tree구조에서의 자식 노드가 된다.

- parent: 부모 노드를 가리킨다.

・method 함수

- insert\_left, right, up, down(m,n): 인자로 받은 미로 위치를 바탕으로 state를 생성해 cost와 부모 노드 설정 후 해당 자식 노드에 삽입한다.

- get\_position(): 자신의 위치를 나타내는 행과 열을 return한다.

- get\_left, right, up, down(): 해당 자식 노드를 return한다.

- get\_cost(): cost값을 return한다.

- is\_parent(m,n): 인자로 받은 위치와 자신의 부모 노드의 위치를 비교해 똑같는지 확인.

3) **프로그램 함수**

・get\_key\_position(): maze에 저장되어 있는 미로에서 key의 위치를 return한다.

・get\_start\_position(): maze에 저장되어 있는 미로에서 start의 위치를 return한다.

・get\_goal\_position(): maze에 저장되어 있는 미로에서 goal의 위치를 return한다.

・read\_file(path): 해당 파일에서 floor, 미로의 사이즈, 미로를 읽어온다.

・write\_file(path): 해당 파일에 maze에 저장되어 있는 미로를 쓴다.

4) **search 알고리즘을 사용한 함수**

・search\_with\_BFS(start\_row,start\_col,goal\_row,goal\_col):

인자로 받은 start 위치에서 goal 위치까지 BFS를 사용해서 탐색하는 함수이다. 우선 시작 위치(root)를 queue에 놓는다. 먼저 queue에서 pop하고 그 state가 goal인지 확인한다. 다음으로 state에서 좌, 밑, 우, 위를 확인하고 벽이 아니면 queue에 push한다. 이것을 goal이 나올 때까지 반복한다. 그 후에 goal state에서부터 root까지 부모 노드로 올라가면서 해당 state의 가리키는 미로의 위치에 ‘5’를 쓴다. 마지막으로 탐색한 노드의 개수와 최단거리를 return한다.

・search\_with\_GBF(start\_row,start\_col,goal\_row,goal\_col):

인자로 받은 start 위치에서 goal 위치까지 Greedy-best-first를 사용해서 탐색하는 함수이다. Heuristic값은 벽 상관없는 start위치에서 goal위치까지의 최단거리로 지정한다. 자료구조로는 priority queue를 사용한다. 첫 key는heuristic값이고 다음 key는 FIFO로 한다. 우선 시작 위치(root)를 priority queue에 놓는다. 먼저 queue에서 pop하고 그 state가 goal인지 확인한다. 다음으로 state에서 좌, 밑, 우, 위를 확인하고 벽이 아니면 priority queue에 push한다. 이것을 goal이 나올 때까지 반복한다. 그 후에 goal state에서부터 root까지 부모 노드로 올라가면서 해당 state의 가리키는 미로의 위치에 ‘5’를 쓴다. 마지막으로 탐색한 노드의 개수와 최단거리를 return한다.

・search\_with\_Astar(start\_row,start\_col,goal\_row,goal\_col):

인자로 받은 start 위치에서 goal 위치까지 A\* 알고리즘을 사용해서 탐색하는 함수이다. Heuristic값은 GBF와 동일하고 자료구조로도 priority queue를 사용한다. 첫 key는 heuristic값과 cost의 합이고 다음 key는 FIFO로 한다. 우선 시작 위치(root)를 priority queue에 놓는다. 먼저 queue에서 pop하고 그 state가 goal인지 확인한다. 다음으로 state에서 좌, 밑, 우, 위를 확인하고 벽이 아니면 priority queue에 push한다. 이것을 goal이 나올 때까지 반복한다. 그 후에 goal state에서부터 root까지 부모 노드로 올라가면서 해당 state의 가리키는 미로의 위치에 ‘5’를 쓴다. 마지막으로 탐색한 노드의 개수와 최단거리를 return한다.

5) **미로 최단거리 탐색 알고리즘**

우선 start위치, key위치, goal위치를 얻는다. 먼저 start에서 key까지 search알고리즘을 사용해 탐색한다. 다음으로 같은 알고리즘으로 key에서 goal까지 탐색한다. start에서 key까지의 최단거리와 key에서 goal까지의 최단거리를 합해 전체 최단거리를 알아낸다.

2. 실험결과 및 알고리즘 선택

1) **first\_floor**

・BFS: length=3850, time=6750

・Greedy-best-first: length=3850, time=5831

・A\*: length=3850, time= 6609

-> 제일 빨리 탐색을 한 Greedy-best-first를 선택.

2) **second\_floor**

・BFS: length=758, time=1721

・Greedy-best-first: length=758, time=1005

・A\*: length=758, time= 1614

-> 제일 빨리 탐색을 한 Greedy-best-first를 선택.

3) **third\_floor**

・BFS: length=554, time=1007

・Greedy-best-first: length=554, time=659

・A\*: length=554, time= 822

-> 제일 빨리 탐색을 한 Greedy-best-first를 선택.

4) **fourth\_floor**

・BFS: length=334, time=594

・Greedy-best-first: length=334, time=451

・A\*: length=334, time= 563

-> 제일 빨리 탐색을 한 Greedy-best-first를 선택.

5) **fifth\_floor**

・BFS: length=106, time=234

・Greedy-best-first: length=106, time=121

・A\*: length=106, time= 155

-> 제일 빨리 탐색을 한 Greedy-best-first를 선택.

3. 결론

모든 층에서 greedy-best-first 알고리즘을 선택하게 된 결과가 나왔다. Greedy-best-first는 optimal을 보장하지 않지만 최단거리를 알맞게 찾으면 3개중에서는 제일 빨리 찾을 수 있다는 것을 알 수 있다.