Project-1: Search

2015136107 이현진

컴퓨터공학부
2sguswls2s@koreatech.ac.kr

1 Introduction

1.1 A Pacman Search Problem

이번 프로젝트는 팩맨의 길을 개척하는 것이다. 특정한 장소를 도달하거나, 먹이를 잘 먹기 위해서 검색 알고리즘을 구축하여 팩맨 시나리오에 적용한다.

이번 프로젝트를 적용하기 위해서 8가지의 질문으로 구성되어 있는데,
Depth-First Search
Breadth-First Search
Varying the Cost Function
A* Search
Finding All the Corners
Corners Problem: Heuristic
Eating All The Dots
Suboptimal Search
로 구성되어 있다.

위의 문제들을 해결하기 위해서 어떠한 해결방안을 사용하였고, 수도코드 및 Grading 결과, 결과 및 분석, 느낀점을 기록한 보고서이다.

```
python autograder.py
```

해당 코드를 실행 할 경우 점수를 볼 수 있다.

팩맨 파이썬 파일 설명 및 주의사항

Table 1. 팩맨 파일 중 중요한 파이썬 파일 설명

파일명	설명
Search.py	탐색 알고리즘을 작성하는 곳
SearchAgents.py	탐색 알고리즘의 기반을 작성하는 곳
Pacman.py	팩맨 파일의 중요 파일. 팩맨의 GameState를 볼 수 있음.
Game.py	팩맨이 어떻게 움직이는지 볼 수 있는 곳. AgentState, Agent, Direction, Grid로 구성되어 있음.
Util.py	다양한 탐색 알고리즘 데이터구조가 정의되어있음
graphicsDisplay.py	팩맨의 그래픽 코드
graphicsUtils.py	팩맨의 그래픽 코드 지원
textDisplay.py	아스키 그래픽 코드
ghostAgents.py	유령 관련 코드
keyboardAgents.py	키보드 인터페이스
layout.py	레이아웃 코드
autograder.py	프로젝트 검사 코드
testParser.py	Autograder.py 솔루션 파일 구문 분석
testClasses.py	프로젝트 검사 코드
test_cases/	질문에 대한 테스트 사례가 포함된 디렉터리
searchTestClasses.py	코드 등급 클래스

주의사항으로는 과제를 하는 동안 클래스의 이름이나 함수의 이름을 바꾸는 것은 내부 구조가 꼬일 수 있으므로 하지 않도록 한다.

2 Questions

2.1 Depth-First Search

해결방법

Depth-First Search를 해결하는데 있어서 가장 중요한 점으로는 노드의 탐색에 있다. 노드를 탐색할 때, 일단 왼쪽이든 오른쪽이든 한노드로 깊이 들어가서 탐색하는데, 자식노드의 끝레벨까지 확인한다. 이후 마지막 자식노드를 검색하고 나면 자신의 형제노드를 검색하고, 형제 노드를 들린 후, 부모노드로 가서 부모노드의 형제노드를 탐색하는 것을 반복하여 문제를 해결한다. 이를 코드로 작성함에 있어서 가장 쉽게 해결 할 수 있는 방법으로는 스택을 사용하면 된다.

스택을 사용할 때 유의 해야할 점으로는 스택에는 부모와 인접한 노드를 집어 넣는다는 점이다. 루트노드에서의 인접된 자식 노드 중 오름차순 순으로 스택에 들어가고, 스택의 가장 위에 있는 노드를 뺀 후, 뺀 노드에 대해서 자식 노드를 탐색 후, 다시 스택에 넣고, 다시 스택에 있는 데이터를 뺀 후, 자식을 검색하는 방법을 이용한다. 만약 자식 노드가 발견되지 않으면 스택에 있는 데이터를 뽑음으로써 이 문제를 해결한다. 흔히 미로에서 사용하는 탐색방법 중 하나로 많이 이용하고 있다.

여기서는 스택에 집어 넣고, 스택에 빼면서 다음 노드를 찾고, 찾은 노드와 함께 다음에 수행할 액션을 전체 거리에 추가하여 스택에 집어 넣는 방법을 이용하였다.

수도코드

Depth-First-Search

```
Algorithm depthFirstSearch
Input : problem
Output : list

Stack stack
List visit, direction
Node node, successor
Action action
Int Item = 0

Stack.push(state, empty list)
While stack is not empty  do
   Node, direction = stack.pop()
   Increment visit (item, node)
   Item = Item + 1
   If node is GoalState then
      return direction
   else
      for successor, action in node do
         if successor not in visit then
            stack.push(successor, direction + [action])
         endif
      endfor
   endif
endwhile
```

수도코드 설명

1. 일단 스택에 현재 상태(state)와 이동한 위치(direction) (여기서는 아무것도 이동하지 않은 상태)를 넣는다.

2. while문
   1. 스택이 비어 있지 않을때까지 반복수행한다.

   2. 스택에 저장 되어 있는 내용을 하나 빼서 Node와 direction을 채워넣는다

   3. visit 리스트에 방문 횟수(item)와 현재 스택에서 뺀 노드(현재 방문한 노드, node)를 집어넣는다.

   4. 방문횟수를 하나 늘린다. (item)

   5. if –then – else 문

      1. 현재 위치(노드)가 도착점이면 여태까지 이동했던 경로(direction)를 반환한다

      2. Else, For문

         1. 그렇지 않으면 node에 저장된 다음 좌표(successor)와 이동하는 위치(action)를 반복한다.

         2. If-then 문

            1. 현재 위치가 방문된 위치가 아니라면 다음 좌표(successor)와 여태까지 이동 했던 경로(direction)과 다음에 이동할 방향(action)을 합쳐서 스택에 넣는다.

Grading code 결과

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python autograder.py -q q1 --no-graphics
Starting on 4-20 at 3:44:58

Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###


Finished at 3:44:58

Provisional grades
==================
Question q1: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 1. Depth-First Search Grading code 결과

결과 및 분석

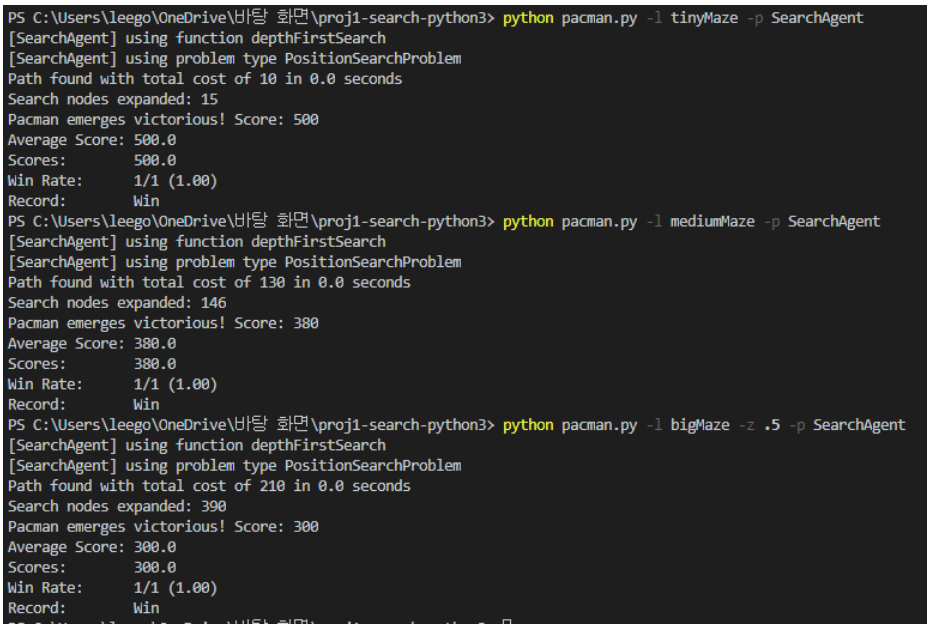

Fig. 2. Depth-First Search 결과

수행해본 결과 팩맨이 이동하는데 있어 왼쪽부터 검색을 하기 때문에 왼쪽으로 가는 모습을 확인 할 수 있었다. 왼쪽으로 가는 길 중 먹이로 갈 수 있는 방법을 찾는데, 중간 중간에 있는 노드들도 액션을 스택에 넣는 우선순위에 따라 탐색이 진행 되었음을 볼 수 있었다.

결론적으로 DFS는 어디를 먼저 방문할 것인지에 따라서 속도가 달라짐을 확인 할 수 있었다.

느낀 점

이 문제를 하면서 배울 수 있었던 점은 지난 3학년 때 알고리즘 시간에 배웠던 DFS를 다시 한번 구현 할 수 있어서 좋았다. 특히 알고리즘 시간에 미로찾기에 대한 코드를 작성해보았는데, 이 경험이 많은 도움이 되었던 것 같다.

2.2 Breadth-First Search

해결방법

Breadth-First-Search 의 경우는 Depth-First Search와 유사한 코드를 작성 할 수 있는데, 깊이 우선 탐색은 우선 깊게 들어가는 특징이 있지만, 너비 우선 탐색은 넓게 들어가는 특징이 존재한다. 따라서 Depth-First Search에서 사용된 스택 구조를 단순히 큐로 바꿔주기만 하면 된다. 왜냐하면 Depth-First Search은 깊게 들어가기 위해서는 자식 노드를 스택에 넣고, 위에 하나 있는 스택을 빼서 해당 노드의 자식노드들을 다시 스택에 집어넣음으로써 깊숙하게 들어갈 수 있었다면, Breadth-First-Search는 FIFO 구조를 이용하여 큐에서 뽑힌 노드에서 자식 노드를 뒤로 집어 넣고, 그 다음에 자신과 같이 들어갔던 형제 노드를 뽑아서 마치 옆으로 훑는 모습을 보여준다.

따라서 Breadth -First Search의 경우는 최단경로를 탐색하는데 적절한 알고리즘이고, Depth -First Search의 경우는 사이클 탐지같은 곳에서 사용하기 좋은 알고리즘이다.

수도코드

(해결한 알고리즘에 대한 설명 또는 수도코드 작성)

(코드를 캡쳐해서 첨부하지 말 것)

Breadth-First-Search

```
Algorithm BreadthFirstSearch
Input : problem
Output : list

Queue queue
List visit, direction
Node node, successor
Action action

Queue.push(state, empty list)
Increment visit (state)
While Queue is not empty  do
   Node, direction = Queue.pop()
   If node is GoalState then
       return direction
   else
      for successor, action in node do
         if successor not in visit then
            Queue.push(successor, direction + [action])
            Increment visit (successor)
         endif
      endfor
```

```
    endif
endwhile
```

수도코드 설명

1. 일단 큐에 현재 상태(state)와 이동한 위치(direction) (여기서는 아무것도 이동하지 않은 상태)를 넣는다.

2. 처음 큐에 넣었던 상태(state)를 방문한 리스트에(visit) 집어넣는다. 왜냐하면 루트 노드를 넣지 않으면 알고리즘 오류가 생김.

3. while문
   1. 큐가 비어 있지 않을때까지 반복수행한다.

   2. 큐에 저장 되어 있는 내용을 하나 빼서 Node와 direction을 채워넣는다

   3. if –then – else 문

      1. 현재 위치(노드)가 도착점이면 여태까지 이동했던 경로(direction)를 반환한다

      2. Else, For문

         1. 그렇지 않으면 node에 저장된 다음 좌표(successor)와 이동하는 위치(action)를 반복한다.

         2. If-then 문

            1. 현재 위치가 방문된 위치가 아니라면 다음 좌표(successor)와 여태까지 이동 했던 경로(direction)과 다음에 이동할 방향(action)을 합쳐서 큐에 넣는다.
            2. 그리고 해당 노드는 방문 했기 때문에 방문 리스트 visit에 삽입한다.

Grading code 결과

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python autograder.py -q q2 --no-graphics
Starting on 4-20 at 3:53:42

Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Finished at 3:53:42

Provisional grades
==================
Question q2: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 3. Breadth-First Search Grading code 결과

결과 및 분석

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Fig. 4. Breadth -First Search 결과

Fig. 5. Breadth -First Search 결과

Breadth -First Search를 수행한 결과 Depth-First Search에서 전혀 가지 않았던 아랫길을 탐색을 수행한 것을 알 수 있었다. 즉, 자식 노드를 탐색 한 후, 자식노드의 형제노드를 전부 탐색을 하는 것을 의미한다. 여기서는 탐색을 마친 후, 가장 짧은 길로 팩맨이 움직이지만, 실시간으로 수행한다면 팩맨이 순간이동하는 것을 볼 수 있을 것이다.

느낀 점

이 문제를 풀면서 알 수 있었던 점은 Depth-First Search에서 전혀 탐색하지 않던 길을 탐색하는 것을 볼 수 있었고, 이를 통해서 완벽한 알고리즘은 존재하지 않는다는 것을 알 수 있었다. 왜냐하면 Depth-First Search가 빠른 경우도 있고, Breadth -First Search가 빠른 경우도 존재하기 때문에 상황에 맞는 적절한 알고리즘을 사용해야겠다고 느꼈다.

2.3 Varying the Cost Function

해결방법

해당 파트에서는 Breadth -First Search에서 확장된 Uniform-Cost-Search를 사용하게 된다. Uniform-Cost-Search은 노드 간에 최소비용을 찾아주는 다익스트라 알고리즘을 사용하는데, Breadth -First Search에 비용이 추가되어

코드를 작성한다. 왜냐하면 Breadth -First Search에서 자식 노드를 큐에 넣고, 큐에서 자식노드를 빼어 손자노드를 큐로 넣는 방법을 수행하기 때문이다. 위 과정을 통해서 자식 노드와 그의 형제 노드들의 비용을 다른 알고리즘에 비해 계산 할 수 있고, 최종적으로 도착점에 도착한 노드들이 큐에 모여서 비교 한 후, 가장 비용이 적게 드는 노드를 이용하면 된다는 것을 알 수 있다. 만약 Depth -First Search로 수행하게 된다면, 최소신장트리, 최단경로를 탐색할 수 없기 때문에 Breadth -First Search로 수행했을 때 보다 많은 시간이 들 수 밖에 없다. 따라서 Uniform-Cost-Search은 Breadth -First Search 변형 알고리즘이라는 것을 알 수 있다. 따라서 Breadth -First Search와 유사한 코드를 보여준다는 특징을 가지고 있다. 여기서는 util.PriorityQueue를 이용하였는데, 일반 큐와 다른 점이 존재한다. 일반 큐는 update 부분이 존재하지 않지만, PriorityQueue는 update가 존재하는데, update의 코드는 해당 노드가 존재하면 수정하고, 그렇지 않으면 삽입하는 코드를 담고 있다.

수도코드

Uniform-Cost-Search

```
Algorithm UniformCostSearch
Input : problem
Output : list

PriorityQueue queue
List direction
Node node, successor
Action action
Set visit
Int cost, newcost

Queue.push(state, empty list)
While Queue is not empty  do
   Node, direction = Queue.pop()
   If node not in visit then
      Increment visit (state)
   else
      continue
   endif

   If node is GoalState then
      return direction
   else
```

```
        for successor, action, cost in node do
          if successor not in visit then
            newcost = problem.getCostofActions(direction + [action])
            Queue.update((successor, direction + [action]), cost + newcost)
          endif
        endfor
      endif
endwhile
```

수도코드 설명

1. 일단 큐에 현재 상태(state)와 이동한 위치(direction) (여기서는 아무것도 이동하지 않은 상태)를 넣는다.

2. while문
   1. 큐가 비어 있지 않을때까지 반복수행한다.

   2. 큐에 저장 되어 있는 내용을 하나 빼서 Node와 direction을 채워넣는다

   3. if-then-else 문
      1. 만약 현재 노드가 방문하지 않은 노드이면 방문 집합 visit에 추가함.

      2. 그렇지 않으면 아래 코드를 무시하고 건너 뜀. (마지막 노드에 대한 검출)

   4. if –then – else 문

      1. 현재 위치(노드)가 도착점이면 여태까지 이동했던 경로(direction)를 반환한다

      2. Else, For문

         1. 그렇지 않으면 node에 저장된 다음 좌표(successor)와 이동하는 위치(action), 여태까지 움직인 거리(cost)들을 반복한다.

         2. If-then 문

            1. 현재 위치가 방문된 위치가 아니라면 현재 이동했던 경로에서 이동할 방향을 추출하여 계산해주는 getCostOfActions 함수에 넣어 걸리는 비용을 계산하여 넣어준다.

2. 다음 좌표(successor)와 여태까지 이동 했던 경로(direction)과 다음에 이동할 방향(action), 이전 비용(cost)와 새로운 비용(newcost)을 합쳐서 큐에 업로드(수정) 해준다.

Grading code 결과

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python autograder.py -q q3 --no-graphics
Starting on 4-20 at 4:04:56

Question q3
===========
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         261
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         169
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###


Finished at 4:04:56

Provisional grades
==================
Question q3: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 6. Uniform-Cost-Search Grading code 결과

결과 및 분석

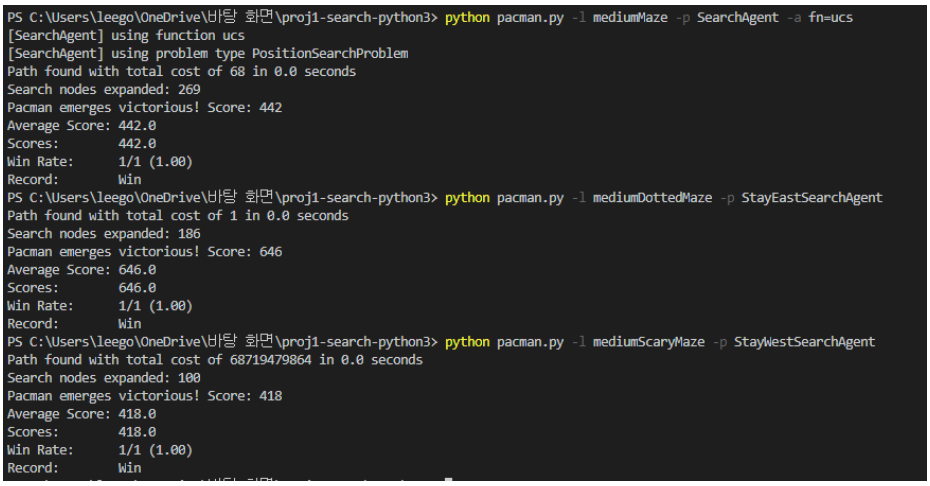

Fig. 7. Uniform-Cost-Search 결과

해당 코드를 실행 해본 결과 BFS였으면 단순하게 첫번째 예제처럼 가장 가까운 거리를 갔을 것이다. 하지만 두번째 실행코드와 세번째 실행코드를 실행해보았을 때 알 수 있던 것은 먹이가 있는 곳은 우선순위가 되어 해당 지역을 우선적으로 탐색하는 경향이 생기고, 몬스터가 있는 지역은 비용이 크다고 인지하여 가지않는 지역임을 보였다. 결과적으로 움직이는 것을 보았을 때, 최소비용에 따른 움직임을 수행함을 알 수 있었다. 하지만 태생이 BFS이다보니 전체적인 탐색을 하는 단점이 존재하였다.

느낀 점

해당 문제를 풀면서 알 수 있었던 점은 다익스트라 알고리즘을 기반으로 한 BFS의 확장판 알고리즘이라는 것을 공부 할 수 있었다. 특히 모든 루트가 비용이 같을 경우에는 BFS와 똑같이 돌아가는 점을 알 수 있었고, 비용이 존재한다면, 최소비용을 전제로 한 방향으로 이동하여 맵 전체를 탐색하기 이전에 탐색을 마칠 수 있다는 것을 배웠다. 이번의 균일비용탐색은 처음 공부를 해보았는데 ,BFS를 알고 있다보니 조금 유연하게 배울 수 있었다.

2.4 A* Search

해결방법

A* 알고리즘의 경우는 UCS 알고리즘과 유사하긴 하지만, 휴리스틱을 이용해서 계산한다는 차별점이 있다. 휴리스틱이란 내가 A에서 B까지 가는데 어떤 루트를 이용하면 어느정도로 갈 수 있다고 예측하여 예측한 방향으로 가는 것을 의미하는데, 여기서는 휴리스틱을 기본 휴리스틱을 이용하는 점이 있었다. 휴리스틱을 제외한다면, UCS와 유사하기 때문에 결국 해당 알고리즘도 큐를 이용하여 문제를 해결한다는 점을 알 수 있었다. 하지만 완전히 똑같다는 뜻은 아니므로 주의해야한다. 이 코드에서는 휴리스틱의 값이 0이기 때문에 UCS와 똑 같은 알고리즘임을 알 수 있다. 그리고 여기서는 방문 집합이 아닌 리스트를 사용했는데, 후에 6번에서 오류가 나서 리스트로 변경하였다.

수도코드

AStarSearch

```
Algorithm AStarSearch
Input : problem, heuristic
Output : list

PriorityQueue queue
List direction
Node node, successor
Action action
Set visit
Int cost

Queue.push(state, empty list)
While Queue is not empty  do
   Node, direction = Queue.pop()
   If node not in visit then
      Increment visit (state)
   else
      continue
   endif

   If node is GoalState then
      return direction
   else
     for successor, action, cost in node do
       if successor not in visit then
          newcost = problem.getCostofActions(direction) + heuristic(successor, problem)
          Queue.update((successor, direction + [action]), cost + newcost)
       endif
     endfor
```

```
    endif
endwhile
```

수도코드 설명

1. 일단 큐에 현재 상태(state)와 이동한 위치(direction) (여기서는 아무것도 이동하지 않은 상태)를 넣는다.

2. while문
   1. 큐가 비어 있지 않을때까지 반복수행한다.

   2. 큐에 저장 되어 있는 내용을 하나 빼서 Node와 direction을 채워넣는다

   3. if-then-else 문
      1. 만약 현재 노드가 방문하지 않은 노드이면 방문 리스트 visit에 추가함.

      2. 그렇지 않으면 해당 줄 아래 코드를 무시하고 다음 반복문으로 건너 뜀.

   4. if –then – else 문

      2. 현재 위치(노드)가 도착점이면 여태까지 이동했던 경로(direction)를 반환한다

      2. Else, For문

         3. 그렇지 않으면 node에 저장된 다음 좌표(successor)와 이동하는 위치(action), 여태까지 움직인 거리(cost)들을 반복한다.

         4. If-then 문

            3. 현재 위치가 방문된 위치가 아니라면 현재 이동했던 경로에서 heuristic 함수를 넣어주는데, heuristic함수에는 successor랑 problem을 넣어주어 계산함. 하지만 여기서는 0이 들어감.

            4. 다음 좌표(successor)와 여태까지 이동 했던 경로(direction)과 다음에 이동할 방향(action)을 합쳐서 큐에 넣는다.

Grading code 결과

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python autograder.py -q q4 --no-graphics
Starting on 4-20 at 4:08:05

Question q4
==========
*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###


Finished at 4:08:05

Provisional grades
==================
Question q4: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 8. AStarSearch Grading code 결과

결과 및 분석

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Fig. 9. AStarSearch 결과

결과 분석 결과 맨해튼 휴리스틱 알고리즘을 쓰기 때문에, UCS보다 좀 더 개선된 이동경로를 보임을 볼 수 있다. 맨해튼 알고리즘은 단순히

유클리드거리에서 응용한 것으로, 각 x좌표와 y좌표의 변위를 절대값 씌워서 더한 값을 의미한다. 이를 통해서 A* 또한 휴리스틱이 존재하지 않으면 UCS와 같은 경로를 안내해줌을 알 수 있었다.

느낀 점

A* 알고리즘을 수행하면서 느낄 수 있었던 점은 UCS 알고리즘과 유사하다는 점이다. UCS알고리즘은 다익스트라 알고리즘을 기반으로하는 최소비용을 찾아가는 알고리즘이라고 하면, A*는 휴리스틱을 기반으로 하여 문제를 해결하는 점에 있다. 휴리스틱을 어떻게 만드냐에 따라서 A*는 속도가 빠를수도, 느릴수도 있다는 점을 배울 수 있고, 별도로 설정하는게 없다면 UCS 알고리즘과 동일하다는 점을 배울 수 있었다.

2.5 Finding All the Corners

해결방법

코너를 해결하기 위해서 코너의 개수와 코너를 들렸는지에 대한 유무 판단이 필요하다고 느꼈고, 이에 코드로 따로 작성하여 문제를 풀었다. 이때, 위치를 가져오는 함수에 feed 라는 코너를 들렸는지 판단하는 변수를 두어 코너에 들리는 경우 갱신되는 코드를 작성하였고, 코너의 움직인 순서 또한 매개변수로 넘겨주었다. 그리고 도착했는지 확인하는 함수에서는 corner를 하나씩 빼는 코드로 작성하였기 때문에, 개수가 없고 feed에 안들린 곳이 없으면 ture를 반환하게 하였다. Feed의 처리의 경우 벽이 아니면서, 다음 갈 위치가 코너가 아니면 그대로 nextState에 넣는 코드를 하였고, 그게 아니라면 feed 부분을 확인 후, newfeed에 1로 채워서 feed와 newfeed를 합친 값을 nextState를 넣어서 나중에 successors에 집어 넣는 방법을 채용하였다.

수도코드

Finding All the Corners : getStartState

Algorithm getStartState
Input : self
Output : state

Tuple startingPosition
List corners

```
List temp
List feeds
Int i

feeds = [0,0,0,0]
For i = 0 to (corners) length do
   If self.startingPosition equal self .corners[i] then
      feeds[i] = 1
   Endif
Endfor

For i = 0 to (self .corners) length do
   If feeds[i] equal 0 then
      Increment temp (self .corners[i])
   Endif
Endfor

Return self .startingPosition, temp, feeds
```

수도코드 설명

1. feeds 배열(먹이를 먹었는지 확인하는 코드)의 초기 값을 0으로 지정한다. 이때 4인 이유는 코너는 4개가 존재하기 때문이다.

2. for문
   1. i가 0부터 시작해서 corners의 배열 개수 만큼 반복수행한다.

   2. if-then 문
      1. 만약 현재 위치가 코너이면, feeds[i]를 1로 바꿔준다.

3. for문
   1. i가 0부터 시작해서 corners의 배열 개수 만큼 반복수행한다.

   2. if-then 문
      1. 만약 feeds[i]가 0이면(방문하지 않은 코너), temp 배열(코너의 정보)에 넣어준다.

4. 현재위치, 남은 코너의 위치, 먹은 코너의 정보를 반환해준다.

Finding All the Corners : isGoalState

Algorithm isGoalState
Input : self, state
Output : Bool

Tuple position
List corners
List feeds

position, corners, feeds = state

If corners length equal 0 and 0 not in feeds **then**
  Return True
Endif

Return False

수도코드 설명

1. 현재 상태(state)에서 현재 위치(position), 코너 위치 리스트(corners), 코너를 방문했는지 알려주는 리스트(feeds)를 지정해준다.

2. if 문
   1. corners의 길이가 0이고, feeds 내부에 0이 없으면 True을 출력해준다.

3. False를 출력해준다

Finding All the Corners : getSuccessors

Algorithm getSuccessors
Input : self, state
Output : successors

List successors
Tuple position, nextPosition
List corners
List feeds, newfeed
Tuple nextState

If not hitsWall **then**

  If nextPosition not in corners **then**
    nextState = (nextPosition, corners, feeds)

```
  else
     if 0 in feeds then
       newfeed[index(feeds)] = 1
       for i = 0 to newfeed length do
          newfeed[i] = newfeed[i] + feeds[i]
       endfor
     endif
   endif

     for i = 0 to newfeed length do
       if newfeed[i] equal 0 then
          Increment tempcorners (self.corners[i])
       Endif
     Endfor

     nextState = (nextPosition, tempcorners, newfeed)
Endif

Increment successors (nextState, action, 1)
```

수도코드 설명

1. if-then-else 문
    1. 벽에 도달하지 않은 이동에 한해 결정한다.
    2. 만약 코너가 아니면 이전의 값 그대로 nextState에 집어넣는다.
    3. Else문
        1. If문
            1. Feeds 리스트에 0이 있다면, newfeed에 1을 넣는다.
            2. For 문
                1. Newfeed의 길이만큼 반복한다.
                2. Newfeed의 값을 이전의 feeds 값과 합친다.

        2. For문
            1. newfeed의 길이만큼 반복한다.
            2. 만약 newfeed[i] 값이 0이면, tempcorners에 코너를 집어넣는다. (방문하지 않은 노드)

        3. nextState에 최종적으로 나온 nextPosition, tempcorners, newfeed를

넣는다.

2. 마지막으로 nextState(다음 위치에 대한 정보), action (이동할 위치)를 집어넣는다.

Grading code 결과

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python autograder.py -q q5 --no-graphics
Note: due to dependencies, the following tests will be run: q2 q5
Starting on 4-20 at 4:08:27

Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Question q5
===========
*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 3/3 ###


Finished at 4:08:27

Provisional grades
==================
Question q2: 3/3
Question q5: 3/3
------------------
Total: 6/6

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 10. Finding All the Corners Grading code 결과

결과 및 분석

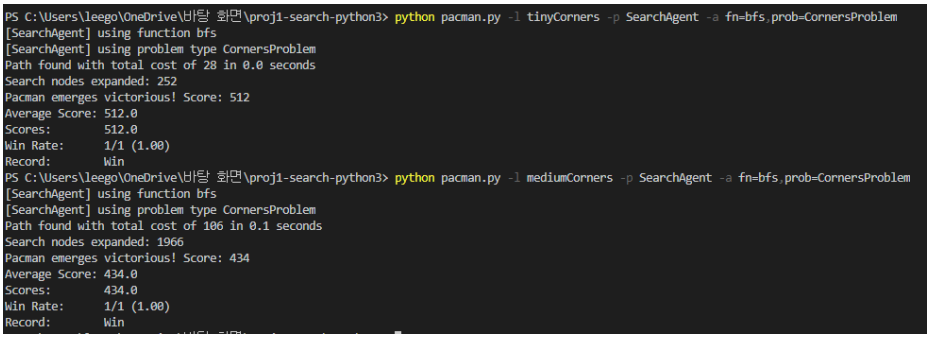

Fig. 11. Finding All the Corners 결과

결과를 분석한 결과, 왼쪽 하단 모서리부터 왼쪽 상단, 오른쪽 상단, 오른쪽 하단으로 가는 최단 경로를 가는 것을 확인 할 수 있었다. 만약에 corners 값의 갱신이 제대로 이뤄지지 않는다면, 기존의 길이보다 많이 걸리는 루트를 가서 문제를 해결한다는 것 또한 관찰할 수 있었다.

느낀 점

이번 문제를 해결하면서 알 수 있었던 점은 feed쪽을 직접 건들어서 문제를 해결하려고 했는데, 값이 연동되는 바람에 시간이 많이 걸렸던 구역이었다. 또한 코너에 들렸는지 확인하는 방법으로 코너의 정보로도 충분하였지만 feed를 둠으로써 확실하게 제어 할 수 있었던 것 같다.

2.6 Corners Problem: Heuristic

해결방법

코너문제를 해결하기 위한 휴리스틱 알고리즘을 짜는 문제임을 인지하고, 문제를 해결하였다. 이 때, 맨해튼 거리 알고리즘을 사용했는데, 맨해튼 거리는 절대값(현재 x위치 – 가고자 하는 x위치) + 절대값(현재 y위치 – 가고자 하는 y위치)을 의미한다. 즉, 유클리드거리를 이용한 휴리스틱 알고리즘이다. 이를 이용해서 나온 거리는 실제 거리보다는 크면 안되지만, 예상 된 거리중에서는 가장 커야 휴리스틱을 돌리는데 있어 보다 유사하게 값이 나오기 때문에 구했던 거리들 중 가장 큰 값으로 지속적으로 바꿔주는 코드를 작성하였다.

수도코드

cornersHeuristic

```
Algorithm cornersHeuristic
Input : state, problem
Output : Int

Tuple position
List corners
List feeds
Int x1, x2, y1, y2, i, j, newdistance, distance

position, corners, feed = state
x1, y1 = position
distance = 0
for i = 0 to corners length  do
   x2, y2 = corners[i]
   for j = 0 to feeds length do
      if 0 in feeds[j] then
         newdistance = abs(x2-x1) + abs(y2-y1)
         if Compute (distance < newdistance) then
            distance = newdistance
         endif
      endif
   endfor
endfor

if state is GoalState then
   return 0
endif
return distance
```

수도코드 설명

1. 상태의 값은 다양한 값으로 구성되어 있기 때문에, 각각의 필요한 값들로 나눠주는데, 현재위치(position), 들려야 할 코너 위치(corners), 코너를 들렸는지 확인해주는 변수(feed)를 채워준다.

2. position 또한 집합으로 구성되어 있기 때문에 x1과 y1 값을 따로 알기 위해서 따로 변수를 두어 저장한다. 그리고 초기 distance는 0으로 지정한다.

3. for 문
   1. i가 corners의 길이만큼 크기가 될 때까지 반복한다.

2. 코너를 위치를 알려주는 변수인 corners 리스트에 있는 x와 y 또한 추출

3. for 문

1. j가 feeds의 길이만큼 크기가 될 때까지 반복한다.

2. if-then 문

1. feed의 내부에 0이 있는지 검사하고(코너에 이미 들렸는지), 맨해튼 거리를 이용하여 거리를 구한다.

2. 이 후, 기존의 거리와 비교하여 기존의 거리보다 값이 크면 교체한다.

4. if –then 문

1. 현재 위치가 마지막 도착 위치라면, 0을 반환시킨다.

5. 최종적으로 구한 distance를 반환한다.

Grading code 결과

(Grading code 결과 캡처)

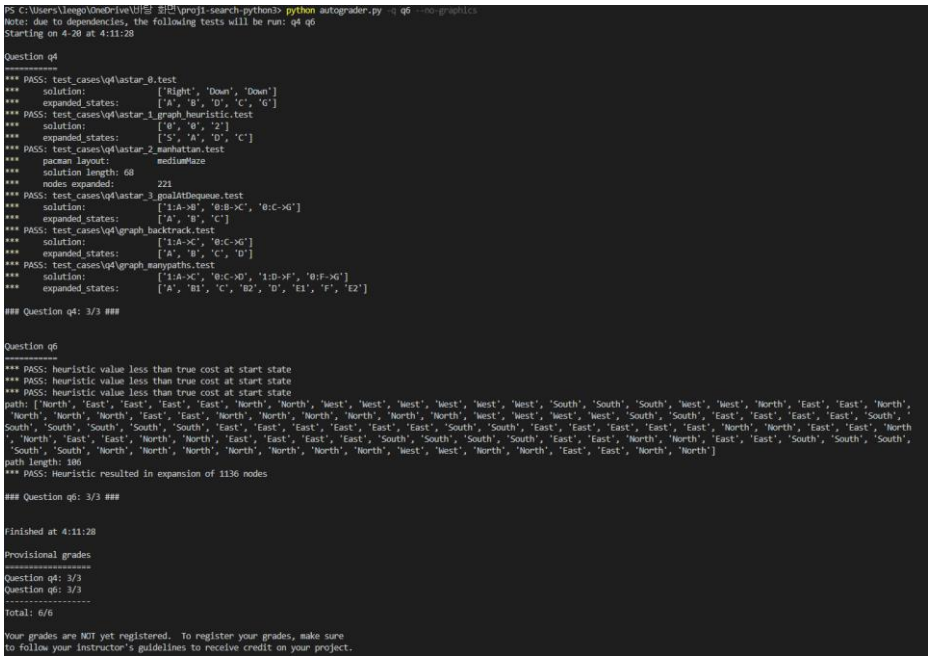

Fig. 12. Corners Problem : Heuristic Grading code 결과

결과 및 분석

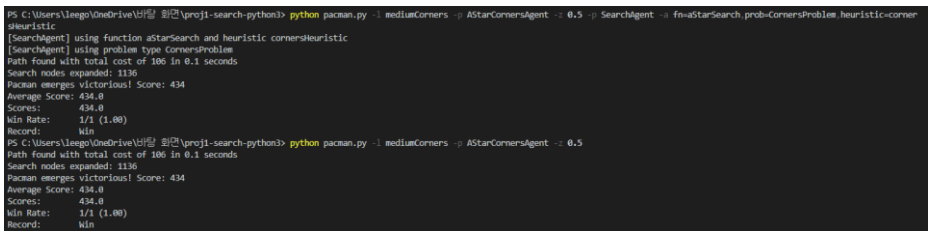

Fig. 13. Corners Problem : Heuristic 결과

결과 분석 결과 모서리에 있는 먹이를 잘 먹고 움직임을 알 수 있었다. 여기서 알 수 있었던 점은 코너 휴리스틱을 적용하기 전과 다르게 움직이는 팩맨을 관찰 할 수 있었다.

느낀 점

이번 휴리스틱코너를 하면서 느낀 점은 다양한 알고리즘 중 맨해튼 거리 알고리즘을 선택하였는데, 그 이유는 휴리스틱의 기초가 되는 알고리즘이라서 제일 이해하여 문제를 해결할 수 있던 알고리즘이었던 것 같다. 기회가 된다면 맨해튼 거리 뿐만 아니라 다른 알고리즘을 적용하여 실행하면 어떻게 될지 궁금하기도 하였다.

2.7 Eating All The Dots

해결방법

모든 점을 먹었는지 확인하는 함수를 작성하는 곳인데, 여기서 중요한 점은 state가 팩맨의 위치와 먹이의 위치를 알려주는 튜플형식으로 되어 있다는 점과 foodGrid.asList가 먹어야 할 점의 위치배열을 알려주는 것을 알 수 있었다. 또한 A*를 이용하면 UCS보다 더 안좋게 동작하기 때문에, 사용에 주의하라는 점도 존재하였고, 이를 위해 일관성 있는 휴리스틱을 작성하라고 하였다. 이 점을 유의하여 2.6에서 사용했던 코드와 비슷하게 코드를 작성하였고, searchAgents 파일 내에 있는 mazedistance 함수를 보고 코드를 작성하였는데, 해당 함수는 미로내에서 가까운 음식의 거리와 멀리 있는 음식 사이의 거리를 구하는 함수이다. 새로운 problem을 탐색을 한 후, BFS로 탐색을 수행하는 함수라고 말할 수 있다. 이를 통해 다양한 코너에 대한 거리를 구해 저장된 거리와 비교하여 큰 경우에 교체하는 방법을 이용하였다.

수도코드

foodHeuristic

```
Algorithm foodHeuristic
Input : state, problem
Output : int

Tuple position
List foodGrid
List foodlist
List food
Int newdistance, distance

position, foodGrid  = state
foodlist = foodGrid.asList()
distance = 0
if foodlist length is 0 then
   return 0
```

```
endif

for food in foolist  do
      newdistance = maze distance of A to B
      if Compute (distance < newdistance) then
         distance = newdistance
      endif
endfor

return distance
```

수도코드 설명

1. 현재 상태(state)에서 현재 위치(position)와 먹이 정보(foodGrid)를 가져온다.

2. 먹이 정보 중 먹이의 좌표 리스트(foodlist)를 가지고 온다.

3. if-then 문

   1. foodlist의 길이가 0이면 0을 반환한다.

   2. for 문

      1. foodlist 의 모든 좌표 정보를 읽을 때 까지 반복한다.

      2. mazeDistance (maze distance of A to B) 함수를 실행하여 먹이의 위치를 갱신하면서 새로운 미로의 거리를 가져온다.

      3. if-then 문

         1.만약 새로운 거리가 기존 거리보다 크면 거리를 갱신시킴.

4. 거리의 값을 반환한다.

Grading code 결과

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python autograder.py -q q7 --no-graphics
Note: due to dependencies, the following tests will be run: q4 q7
Starting on 4-20 at 4:12:07

Question q4
===========
*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###


Question q7
===========
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###


Finished at 4:12:21

Provisional grades
==================
Question q4: 3/3
Question q7: 5/4
------------------
Total: 8/7

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 14. Eating All the Dot Grading code 결과

결과 및 분석

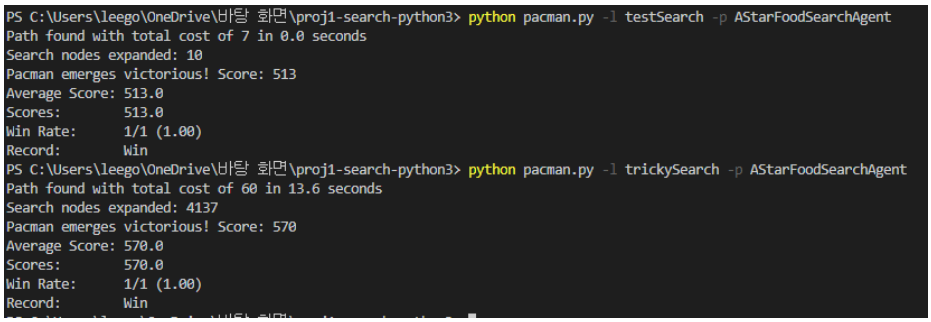

Fig. 15. Eating All the Dot 결과

분석 결과 기존의 실행의 경우 16000가지의 노드를 탐색하였지만, 4137개의 노드만 탐색하는 결과로 시간이 줄어듬을 알 수 있었다. 또한 다른 알고리즘을 적용하면 더욱 줄일 수 있을 것이라 생각이 들었다.

느낀 점

해당 문제를 해결하면서 알 수 있었던 점은 점을 먹는데 대한 휴리스틱을 처리하는데, trickySearch 맵의 경우 많은 점이 존재하였고, 이를 탐지하기 위해서 16000개의 노드를 탐색하는 것을 보고 상당히 놀랐다. 조금 더 좋은 알고리즘을 작성하고 싶었지만 제 머리로는 한계라고 느꼈던 과제였습니다.

2.8 Suboptimal Search

해결방법

이 문제를 해결하기 위해서는 다양한 알고리즘을 사용하여

수도코드

Suboptimal Search : findPathToClosestDot
Algorithm findPathToClosestDot **Input : self, gameState** **Output : direction**

Problem = AnyFoodSearchProblem(gameState)
Return search.aStarSearch(problem)

수도코드 설명

1. problem을 AnyFoodSearchProblem을 이용하여 미로 문제를 새로 정의한다.
2. 새로 정의 한 문제를 기반으로 팩맨 수행 알고리즘을 실행하는데, aStarSearch 알고리즘을 사용함.

Suboptimal Search : isGoalState

Algorithm isGoalState
Input : self, state
Output : Bool

List foodlist

If state in foodlist **then**
  Return True
endif
Return False

수도코드 설명

1. 현재 상태가 foodlist에 있으면, True를 출력하고, 아니면 False를 출력함.

Grading code 결과

```
### Question q8: 3/3 ###


Finished at 4:14:56

Provisional grades
==================
Question q8: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Fig. 16. Suboptimal Search Grading code 결과

결과 및 분석

```
PS C:\Users\leego\OneDrive\바탕 화면\proj1-search-python3> python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Fig. 17. Suboptimal Search 결과

분석 결과 많은 양의 점을 먹는 팩맨을 볼 수 있었다. A* 알고리즘 말고 다른 알고리즘으로도 수행해보았는데, Path cost 만 달라지고, 나머지는 동일한 모습을 보여주었다. 다만, DFS의 경우 무한히 깊이 빠지는 현상으로 인해서 다른 알고리즘과 다르게 엄청나게 많은 값을 보유하고 있음을 보여주었다.

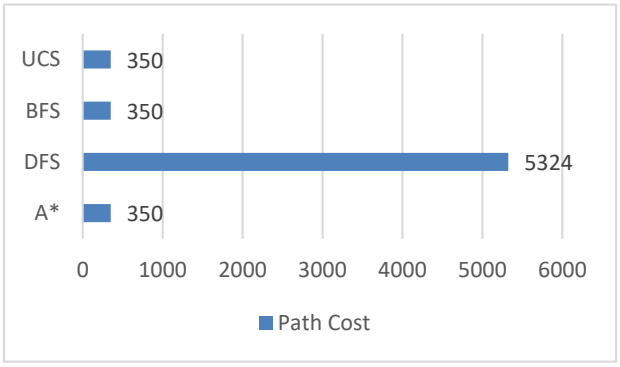

Fig. 18. 다양한 알고리즘의 결과

느낀 점

문제를 하면서 알 수 있었던 점은 위에서 짜왔던 코드들이 하나가 되어 돌아가는 모습을 볼 수 있었는데, 다양한 알고리즘 중 작년에 짜본 A*가 마음에 들어 A*알고리즘으로 실행하였다. A*의 문제점이 드러나는데, 이전의 휴리스틱이 0이기 때문에 UCS와 동일하게 굴러가는 점을 알 수 있었다.

3 Conclusion

느낀 점

이번 과제를 하면서 느낀점은 팩맨을 자동으로 돌아가게 하는데도 많은 어려움이 존재한다는 것을 느꼈다. 특히 휴리스틱의 설계가 매우 어려웠는데, 다양한 정보들을 수집하고 해도 이해하는데 시간이 너무 많이 걸렸다. 그래도 이번 과제를 하면서 뭔가 해냈다는 느낌을 받을 수 있어서 좋았다.

어려웠던 점

문제를 해결하면서 어려웠던 점은 역시 휴리스틱의 설계가 가장 어려웠다고 생각이 든다. 이러닝의 자료나 인터넷을 찾아봐도 이론으로만 설명되어있고, 이를 쉽게 설명하는 곳이 부재하여 이해하는데 많은 시간이 걸렸다. 정말 인공지능이라는 파트가 다른 곳에 비해 공부를 많이 해야한다는 것을 알았고, 영어공부 또한 열심히 해야겠다고 생각이 들었다.

제안점

과제가 너무 어려워서 하는데 너무 많은 시간이 걸렸습니다. 과제에 대한 피드백이나 과제하는데 있어서 필요한 설명하는 영상이나 자료가 있었으면 좋겠습니다.

References

1. CS 188, https://inst.eecs.berkeley.edu/~cs188/fa18/project1.html 2018/09/07.
2. KOREATECH E-Learning https://el.koreatech.ac.kr/ilos/main/main_form.acl
3. AISTUDY http://www.aistudy.com/heuristic/heuristic_definition.htm
4. AISTUDY http://www.aistudy.co.kr/heuristic/heuristic.htm