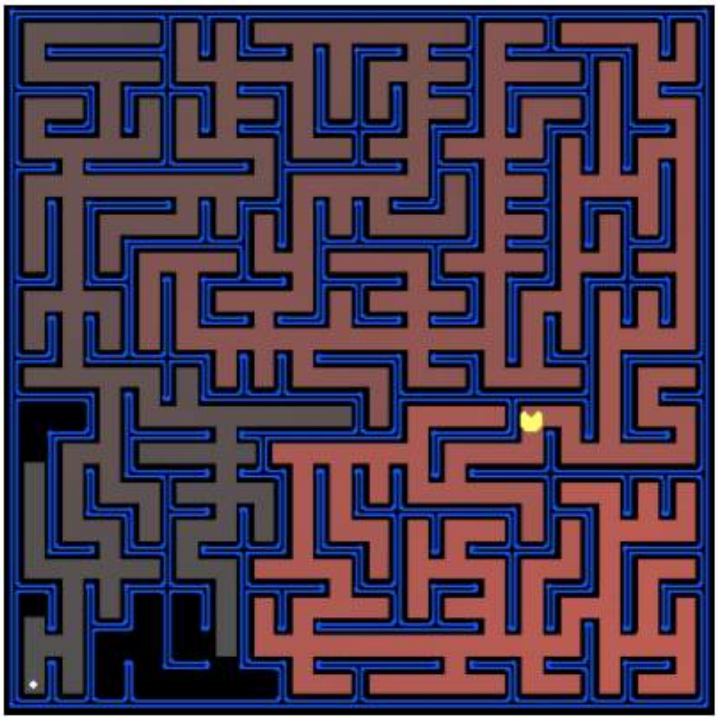
Project 1 : Search in Pacman



소개

(1) **P1 프로젝트는 Pacman(그림 참조)의 agent가 미로를 거쳐 특정 위치에 있는 음식을 수집할 수 있는 경로를 찾는 시나리오**입니다. 경로는 찾는 방법은 일반적인 검색 알고리즘(BFS, DFS, UCS 등)을 이용합니다.

(2) Project 1은 총 **8 문제**를 포함합니다.

3) P0과 마찬가지로, 자동 채점이 포함되어 있으며, 다음 명령어를 통해 실행시킬 수 있습니다.
python autograder.py

(4) 관련한 모든 코드는 zip archive에 포함되어있습니다.
https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/search.zip

(5) 본 p1 프로젝트는 수강생들을 위해 다음 주소의 사이트를 번역 하였습니다. (필히 참고)
https://edge.edx.org/courses/course-v1:BerkeleyX+CS188+2018_SP/courseware/eafff8d8427440069a749c1b825c0561/6171e5fdfd8645c98cdcdf53dfea1880/?activate_block_id=block-v1%3ABerkeleyX%2BCS188%2B2018_SP%2Btype%40sequential%2Bblock%406171e5fdfd8645c98cdcdf53dfea1880

파일에 대한 설명

수정하거나 실행하는 파일들

- search.py : Where all of your search algorithms will reside.
- searchAgents.py : Where all of your search-based agents will reside.

수정은 하면 안 되지만, 꼭 봐야하는 파일들

- pacman.py : The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
- game.py : The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
- util.py Useful data structures for implementing search algorithms.

무시해도 괜찮은 파일들

- graphicsDisplay.py : Graphics for Pacman
- graphicsUtils.py : Support for Pacman graphics
- textDisplay.py : ASCII graphics for Pacman
- ghostAgents.py : Agents to control ghosts
- keyboardAgents.py : Keyboard interfaces to control Pacman
- layout.py : Code for reading layout files and storing their contents
- autograder.py : Project autograder
- testParser.py : Parses autograder test and solution files
- testClasses.py : General autograding test classes
- test_cases/ : Directory containing the test cases for each question
- searchTestClasses.py : Project 1 specific autograding test classes

P1 기본 튜토리얼

1) 먼저 zip archive(search.zip)을 다운로드 하고 압축을 풀고 다음 명령어를 실행합니다.

   python pacman.py

   * 위 명령어를 실행하면, 실제 pacman 게임이 시작됩니다.

2) searchAgents.py의 가장 단순한 에이전트는 GoWestAgent이며, 이 에이전트는 다음 명령어로 실행할 수 있습니다.

   python pacman.py --layout testMaze —pacman GoWestAgent

   * searchAgents.py의 GoWestAgent class 파일을 보게 되면, 어떻게 서쪽으로 가는 행동을 게임에서 실행시켰는지 확인할 수 있습니다.

3) 다음 명령어를 실행시켜봅니다.

   python pacman.py --layout tinyMaze --pacman GoWestAgent

   * GoWestAgent의 행동은 tinyMaze 맵에서는 적합하지 않은 것을 확인할 수 있습니다.

4) pacman.py는 (예 : --layout, -l)로 옵션을 지원합니다. 다음을 통해 모든 옵션 및 기본 값의 목록을 볼 수 있습니다.

5) python pacman.py -h

   이 프로젝트에 나타나는 모든 명령은 쉽게 복사하여 붙여 넣을 수 있도록 commands.txt에 저장됩니다.

q1 문제 : 깊이 우선 검색(DFS)를 이용하여 음식 찾기

SearchAgents.py 파일에서는 Pacman게임 내에서 경로를 계획하고, 그 경로를 단계별로 실행시킬 수 있습니다. 경로에 대한 계획(검색 알고리즘)은 구현되어있지 않으며, 이 부분을 직접 구현해야합니다.

먼저 다음의 명령어를 실행시켜 봅시다.
python pacman.py -l tinyMaze -p SearchAgent –a fn=tinyMazeSearch

위 명령어는 searchAgent에게 tinyMaze라는 맵에서 tinyMazeSearch 검색알고리즘으로 경로를 찾아 가라고 지시 합니다. 이 알고리즘(tinyMazeSearch)는 search.py에 구현되어있습니다.

이제는 Pacman이 경로를 계획하는 일반 검색 알고리즘을 작성해야합니다. 일반적인 검색알고리즘은 다음과 같습니다.

```
function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

DFS, BFS, UCS 및 A* 에 대한 알고리즘은 위의 일반 검색 알고리즘 의사 코드와 매우 유사합니다. fringe가 어떻게 관리되는지에 대해서만 다릅니다. 따라서 DFS를 올바르게 잘 구현하면, 나머지는 비교적 간단하게 구현할 수 있습니다.

먼저 **search.py의 depthFirstSearch 함수에서 깊이 우선 탐색 (DFS) 알고리즘을 구현하십시오.** 주의할 점은 방문한 장소(state)를 다시 방문하지 않아야 한다는 점입니다. DFS 알고리즘을 구현했다면, 아래 명령어를 이용하여 테스트할 수 있습니다.

python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 –p SearchAgent

위 명령어를 실행할 때, pacman 지도에서의 밝은 빨간색은 탐사된 순서를 표시합니다.

q2 문제 : 너비 우선 검색(BFS)를 이용하여 음식 찾기

search.py의 breadthFirstSearch 함수에 너비 우선 탐색 (BFS) 알고리즘을 구현하십시오. 이미 방문한 상태를 다시 지나치지 않는 그래프 검색 알고리즘을 작성해야하며, 깊이 우선 탐색(DFS)과 같은 방식으로 코드를 테스트할 수 있습니다.

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

* Pacman이 너무 느리게 움직이는 경우 --frameTime 0 옵션을 시도하십시오.

* 참고 : 검색 코드를 일반적으로 작성한 경우 코드는 변경하지 않고 8가지 퍼즐 검색 문제에 대해 똑같이 잘 작동합니다.
python eightpuzzle.py

q3 문제 : **균일비용탐색(Uniform Cost Search) 이용하여 음식 찾기**

BFS는 목표까지 행동 경로를 찾을 수 있지만, 다양한 종류의 문제에 대해 최적화 되지는 못합니다. mediumDottedMaze와 mediumScaryMaze를 실행시켜보십시오.

python pacman.py -l mediumDottedMaze -p SearchAgent -a fn=bfs
python pacman.py –l mediumScaryMaze -p SearchAgent -a fn=bfs

위와 같은 사례들을 해결하기 위해, 비용 함수(Cost Function)를 변경함으로써 Pacman이 다른 경로를 찾도록 할 수 있습니다. 예를 들어 유령이 많은 지역에서 음식을 많이 먹어야 하는 지역의 위험한 상황에서는 더 많은 비용을 청구 할 수 있습니다. 이와 같이 합리적인 팩맨 에이전트는 상황에 맞춰 행동을 조정해야합니다.

search.py의 uniformCostSearch(UCS) 함수에서 uniform-cost 그래프 검색 알고리즘을 구현하십시오. 구현에 유용한 일부 데이터 구조는 **util.py**를 살펴보고 사용할 수 있습니다. 아래의 세 가지 레이아웃 모두에서 성공적인 동작을 보여야 합니다. 아래의 에이전트들은 사용하는 비용(cost function)만 다릅니다.(에이전트 및 비용 기능은 미리 작성되어 있습니다.)

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

* 참고 : 지수 비용 함수(exponential cost functions)로 인해 StayEastSearchAgent 및 StayWestSearchAgent에 대해 각각 매우 낮은 경로 비용과 매우 높은 경로 비용을 가져야합니다. (자세한 사항은 searchAgents.py을 살펴볼 것.)

q4번 문제 : A*를 이용하여 음식 찾기

search.py의 함수 aStarSearch에 A* 그래프 검색을 구현하십시오.
A* 그래픽 검색은 휴리스틱 함수를 사용합니다. 또한 휴리스틱 함수는 두 가지 인자를 사용합니다. (서치 문제의 state(메인 인자), 문제 자체(참조 정보를 위한))
search.py의 nullHeuristic 휴리스틱 함수는 간단한 예입니다.

Manhattan distance heuristic (이미 searchAgents.py에서 manhattanHeuristic으로 구현되어 있음.)을 사용하여 경로를 찾는 문제에 대해 A * 구현을 다음 명령어로 테스트 할 수 있습니다.

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

A* 는 균일한 비용 검색(uniform cost search)보다 조금 빠른 속도로 최적의 솔루션을 찾을 수 있습니다. (검색 노드 자체는 549에서 620개의 검색 노드로 확장되었음.)

q5번 문제 : 모든 코너(모퉁이)에서의 길 찾기(문제 정의)

*** 참고 : 질문 5는 질문 2에 대한 답을 바탕으로 작성되었습니다.**

A* 알고리즘으로 좀 더 어려운 문제를 통해 해결해봅시다. 이번 문제는 새로운 문제를 정의**(q5)**하고, 그것을 위한 휴리스틱**(q6)**을 디자인 할 것입니다.

모퉁이 미로에는 각 구석에 4개의 점이 있고, 이를 위한 최단 경로를 찾는 것입니다.
tinyCorners와 같은 일부 미로의 경우 최단 경로가 항상 가까운 구석으로 가는 것은 아닙니다.
* 힌트 : tinyCorners를 통과하는 최단 경로는 28걸음이 걸립니다.

searchAgents.py에서 CornersProblem 검색 문제를 구현하십시오. 네 모퉁이에 모두 도달했는지에 대한 상태를 확인하는 표현을 인코딩 해야 합니다. 검색 에이전트는 다음 명령어를 통해 테스트 할 수 있습니다.

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

전체적인 완성도를 높이기 위해서는 관련 없는 정보(예, 유령의 위치, 음식의 위치 등)를 인코딩 하지 않는 추상적인 상태 표현을 정의해야 합니다. 특히, pacman Gamestate를 검색 상태로 사용하지 마십시오.

*** 힌트 : 구현에서 참조해야 하는 게임 상태의 유일한 부분은 Pacman 시작위치와 네 모퉁이 위치입니다.**

breadtheFirstSearch를 구현하면, mediumConrners에서 2000개 미만의 검색 노드를 확장할 수 있습니다. 하지만 A* 알고리즘을 사용하면 검색 노드를 많이 줄일 수 있습니다.

q6번 문제 : 코너 문제 : 휴리스틱

*** 참고 : q6은 q4에 대한 답을 바탕으로 작성되었습니다.** (q6을 작성하기 전 q4를 작성해야 합니다.)

모서리에 있는 모서리 문제에 대해 간단하고 일관된 휴리스틱을 구현하십시오.

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Note: AStarCornersAgent is a shortcut for
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic.

허용성(Admissibility) vs 일관성(Consistency)
: 휴리스틱은 검색 상태를 확인하고, 가장 가까운 목표로 가는 비용을 추정하여 숫자를 반환하는 함수입니다. 효과적인 휴리스틱은 실제 목표 비용에 가까운 값을 반환합니다. 허용성을 보장하기 위해, 휴리스틱 값은 가장 가까운 목표에 대한 실제 최단 경로비용이 가장 작은 값이어야 합니다. 또한 일관성을 유지하기 위해, 어떤 행동에 비용 C가 있는 경우 그 행동을 하는 것이 옳은 일인지 확인해야 합니다.
허용성은 그래프 검색의 정확성을 보장하기에는 충분하지 않습니다.(강한 일관성의 조건이 필요함.) 하지만 허용 가능한 휴리스틱은 일반적으로 완화된 문제인 경우 일관성을 보장하기도 합니다. 따라서 일반적으로 허용 가능한 방법을 브레인스토밍을 통해 생각해봅시다. 일단 잘 작동하는 휴리스틱이 있다면, 그것이 실제로 일관성이 있는지에 대한 여부를 판단할 수 있습니다. (일관성 보장을 확인하는 실제 확인을 통해 증명하는 것입니다.)
하지만 불일치는 확장하는 각 노드를 확장할 때 후속 노드가 f-value 값이 이상인지 확인을 할 때 탐지 될 수 있습니다. 게다가 UCS와 A*의 길이가 다른 경로를 반환하게 되면 휴리스틱은 일관적이지 않은 것입니다.

Non-Trivial Heuristics : trivial 휴리스틱은 모든 곳에서 0을 반환하는 UCS와 실제 완료 비용을 계산하는 방법입니다. 전자는 시간을 절약하지 못하고(UCS), 후자는 자동채점 시간을 넘어갑니다.

채점 : 작성하는 휴리스틱은 어떠한 포인트에서도 일관성이 있는 방법이어야 합니다 (Non-Trivial, non-negative). 휴리스틱은 모든 목표에서 0을 반환하고, 음수 값을 반환하면 안 됩니다. 휴리스틱이 확장되는 노드의 수에 따라 채점이 됩니다.

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

주의사항 : 휴리스틱이 일관성이 없다면, 점수를 받을 수 없습니다.

q7번 문제 : **모든 음식들 찾기**

*** 참고 : q7를 완성하기 저 q4를 확실히 완료해야 합니다.**

이제부터는 어려운 검색 문제를 해결할 것입니다. 가능한 한 적은 스텝으로 모든 팩맨 음식을 먹도록 구현합시다. 이를 위해 음식 정리 문제를 공식화하는 새로운 검색 문제 정의가 필요합니다. (searchAgents.py의 FoodSearchProblem (사용자를 위해 구현되었음.))

이를 위한 솔루션은 Pacman 세계에서 모든 음식을 수집하는 걸로 정의합니다. 현재의 프로젝트(p1)에서의 솔루션은 유령이나 파워 알약을 고려하지 않습니다. 솔루션은 벽, 일반 식품 및 Pacman의 위치만 취급합니다. (유령은 솔루션의 실행을 망칠 수 있습니다!. 우리는 다음 프로젝트에서 그 점을 알게 될 것입니다.)
일반적인 검색 방법을 올바르게 작성했다면, null heuristic이 포함된 A* (uniform-cost search와 동일)는 코드 변경 없이 testSearch에 대한 최적의 솔루션을 찾아냅니다(총 비용: 7).

python pacman.py -l testSearch -p AStarFoodSearchAgent

*** 참고** : StarFoodSearchAgent는 가장 빠른 길입니다.(for -p SearchAgent –a fn=astar, prob=FoodSearchProblem, heuristic=foodHeuristic.)

UCS는 단순한 tinySearch처럼 느려지기도 합니다. 참고로 구현할 때 5057개의 노드를 확장하고, 길이가 27인 경로는 찾는 2.5초가 걸립니다.
SearchAgents,py의 FoodHeuristic을 작성하십시오. (FoodSearchProblem에 대한 일관성 있는 휴릭스틱을 가진) 완성했다면 다음 명령어로 테스트 합니다.

python pacman.py -l trickySearch -p AStarFoodSearchAgent

UCS 에이전트는 약 13초만에 최적의 솔루션을 찾고, 16,000개 이상의 노드를 탐색합니다.

Non-trival하고, 일관성이 없는 휴리스틱은 1점을 받습니다. 휴리스틱은 모든 목표 상태에서 0을 반환하고, 절대 음수 값을 반환하면 안됩니다. 휴리스틱이 확장되는 노드의 수에 따라 추가 점수를 얻을 수 있습니다.

Number of nodes expanded	Grade
more than 15000	1/4
at most 15000	2/4
at most 12000	3/4
at most 9000	4/4(full credit:medium)
at most 7000	5/4(optional extra credit:hard)

q8번 문제 : 차선의 검색(Suboptimal Search)

A* 와 휴리스틱을 잘 작성했다고 하더라도, 때로는 모든 점들에 대한 최적의 경로를 찾기 어렵습니다. 때문에 이번 문제(q8)에서는 가장 가까운 음식을 모으는 에이전트를 작성합니다. **ClosestDotSearchAgent는 SearchAgents.py 에서 구현되지만 가장 가까운 점에 대한 경로를 찾는 주요 기능이 없습니다.**

searchAgents.py에 findPathToClosestDot 함수를 구현하십시오. 이 에이전트는 이 미로를 350 경로 비용으로 해결해야 합니다.

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

*** 힌트 : findPathToClosestDot를 완성하는 가장 빠른 방법은 목표 테스트가 누락된 AnyFoodSearchProblem을 채우는 것입니다. 그런 다음 적절한 검색 기능으로 문제를 해결하십시오. 해결책은 매우 짧아야합니다!**

ClosestDotSearchAgent가 미로를 통해 가능한 가장 짧은 경로를 항상 찾지는 않습니다. 이유를 이해하고 가장 가까운 점으로 반복적으로 가는 것이 모든 점을 먹는 최단 경로를 찾지 못하는 작은 예를 생각해보십시오.