인공지능 과제 #1 (2025년도 1학기)

이번 과제에서는 수업시간에 배운 깊이 우선 탐색과 넓이 우선 탐색, 균일 비용 그래프 탐색, A* 탐색 알고리즘들을 이용하여 팩맨(PacMan) 게임을 수행하는 agent를 구현합니다.

수업 홈페이지에 제공되는 압축파일(ai-sp25_hw1_code.zip)을 풀면, search라는 이름을 가진 디렉토리를 확인할 수 있습니다. 이 디렉토리에 대해서 아래의 문제에 맞는 요구사항을 구현하면 됩니다.

1. 구현해야 하는 파일과 코드가 아닌 게임 그래픽 관련된 코드나 다른 코드들은 수정하지 않습니다.
2. 모든 채점은 autograder.py 파일을 통해서 이루어집니다.
3. 과제는 8개의 세부 문항들로 구성되어 있으며, 각 세부 문항별 배점은 3점입니다.
4. 과제 제출
    A. 기한: **4월 15일(금) 23:59**까지
    B. 방법: 학교 LMS의 인공지능(7분반)의 레포트 제출에서 **구현이 포함되어 있는 파일들을 하나의 zip 파일로 압축해서 제출하시기 바랍니다.**
    (파일 이름 예시, 학번이 1234567인 홍길동 학생: 홍길동_`1234567.zip`)

제출시 주의 사항

1. zip 형식 이외의 다른 압축 형식으로 제출하거나, Windows 10이나 Windows 11 환경에서 압축이 풀리지 않은 zip 파일들은 0점 처리합니다.
2. 파일 이름이 예시 형식과 맞지 않을 경우 이 과제 만점의 10%를 감점합니다.
3. 다른 사람의 코드를 그대로 사용하거나 인터넷에 공개된 코드나 로직을 베껴서 사용한 것이 적발되면 해당 과제는 수업 첫 시간의 안내에 따라 처리합니다. 여러분들이 인터넷에서 찾을 수 있을만한 자료들은 이미 조교들이 모두 확보하고 있다고 생각하시면 됩니다. 스스로의 힘으로 과제를 풀어보세요.

1. 과제 소개

이 과제에서 팩맨 agent는 미로 세계에서 특정 위치에 도달하고 음식을 효율적으로 수집하기 위한 경로를 찾아야 합니다. 여러분들은 탐색 알고리즘을 구축하여 팩맨 시나리오에 적용하게 됩니다. 이 과제에는 컴퓨터에서 답을 채점할 수 있는 자동 채점기가 포함되어 있습니다. 이 자동 채점기는 명령어로 실행할 수 있습니다:

```
python autograder.py
```

이 과제의 코드는 여러 개의 Python 파일들로 구성되어 있으며, 과제를 완료하기 위해 일부는 읽고 이해해야 하며 일부는 무시해도 됩니다. 모든 코드와 지원 파일은 ai-sp25_hw1_code.zip에 포함되어 있습니다.

[구현해야 하는 파일들]

search.py	모든 탐색 알고리즘을 포함하는 파일

searchAgents.py	모든 탐색 기반 agent 들이 포함된 파일

[살펴볼 만한 파일]

pacman.py	팩맨 게임을 실행하는 메인 파일입니다. 이 파일은 이 과제에서 사용하는 팩맨 GameState type에 대해 설명합니다.
game.py	이 파일은 팩맨 세계를 구성하는 agent 상태, agent, 방향, 그리드 등 여러 지원 유형에 대해 설명합니다.
util.py	탐색 알고리즘 구현에 유용한 데이터 구조.

[변경하지 말아야 할 파일]

searchTestClasses.py	과제 1 전용 자동 채점기 테스트 클래스
graphicsDisplay.py	팩맨용 그래픽
graphicsUtils.py	팩맨 그래픽 유틸리티
textDisplay.py	팩맨의 ASCII 기반 그래픽
ghostAgents.py	고스트 제어용 agent
keyboardAgents.py	팩맨 제어를 위한 키보드 인터페이스
layout.py	레이아웃 파일을 읽고 내용을 저장하는 코드
autograder.py	과제 자동 채점기
testParser.py	자동 채점 테스트 및 솔루션 파일 파싱
testClasses.py	기본 자동채점 테스트 클래스
test_cases/	각 질문에 대한 테스트 케이스가 포함된 디렉터리

2. 팩맨에 오신 것을 환영합니다.

코드를 다운로드하고 압축을 푼 다음 디렉터리로 변경한 후 명령줄에 다음을 입력하면 팩맨 게임을 플레이할 수 있습니다:

```
python pacman.py
```

팩맨이 살고 있는 세계를 효율적으로 탐색하는 것이 첫 임무입니다.

searchAgents.py 에서 가장 단순한 agent 는 항상 서쪽으로 이동하는 GoWestAgent 입니다. 이 agent 는 단순하지만 가끔 이기는 경우도 있습니다:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

하지만 제대로 회전하지 못합니다:

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

팩맨이 막혀서 더 이상 움직일 수 없으면 터미널에 CTRL-c를 입력해 게임을 종료할 수 있습니다. 이번 과제를 통해 tinyMaze뿐만 아니라 원하는 모든 미로를 해결할 수 있는 팩맨 agent를 구현합니다.

pacman.py 는 각각 긴 방식(예: --layout) 또는 짧은 방식(예: -l)으로 표현할 수 있는 여러 가지 옵션을 지원합니다.
모든 옵션 목록과 기본값은 다음을 통해 확인할 수 있습니다:

```
python pacman.py -h
```

3. Q1: 깊이 우선 탐색을 이용한 고정된 과자 찾기

searchAgents.py는 완전히 구현된 SearchAgent를 포함하고 있는데, 미로를 통과하는 경로를 계획한 다음 그 경로를 단계별로 실행하는 agent입니다. 탐색 알고리즘은 구현되어 있지 않으며, 여러분이 직접 구현해야 합니다.

먼저 SearchAgent를 실행하여 제대로 작동하는지 테스트합니다:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

위의 명령은 탐색 agent가 search.py에서 구현된 탐색 알고리즘으로 tinyMazeSearch를 사용하도록 지시합니다. 팩맨이 미로를 성공적으로 탐색하는 것이 목적입니다.

이제 본격적인 탐색 함수를 작성하여 팩맨이 경로를 계획하도록 합니다. 작성할 탐색 알고리즘의 의사 코드는 강의 슬라이드에 있습니다. 탐색 노드에는 상태 뿐만 아니라 해당 상태에 도달하는 경로(계획)를 재구성하는 데 필요한 정보도 포함되어야 합니다.

[중요!] 모든 탐색 함수는 agent를 시작부터 목표까지 안내하는 동작들을 리턴하며, 동작들은 모두 실제로 가능한 이동이어야 합니다(유효한 방향, 벽을 통과하지 않는 이동).

[중요!] util.py에 제공된 Stack, Queue 및 PriorityQueue들을 사용해야 자동 채점기로 평가가 가능합니다. 제공된 자료구조 구현에 자동 채점기와의 호환성을 위해 필요한 특정 속성이 있습니다.

[힌트] 각 탐색 알고리즘들의 구현은 매우 유사합니다. 깊이 우선 탐색과 넓이 우선 탐색, 균일 비용 그래프 탐색 및 A*의 알고리즘은 프런티어를 관리하는 방법에 대한 세부 사항만 다릅니다. 따라서 깊이 우선 탐색을 올바르게 구현하면 나머지는 비교적 간단합니다. 한 가지 구현 가능한 방법은 알고리즘별 큐잉(queuing) 전략으로 구성된 하나의 일반 탐색 방법을 만드는 것입니다. (그러나 반드시 이런 방법으로만 구현할 필요는 없습니다)

search.py의 depthFirstSearch 함수에서 깊이 우선 탐색(DFS) 알고리즘을 구현합니다. 알고리즘을 완성하려면 이미 방문한 상태의 확장을 피하는 그래프 탐색 버전의 DFS를 작성하세요.

여러분이 작성한 코드는 아래 명령어들을 빠르게 수행할 수 있어야 합니다.

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

팩맨 보드에 탐색한 상태와 탐색한 순서가 GUI에 표시됩니다(빨간색이 더 밝을수록 더 일찍 탐색했음을 의미). 탐색 순서가 예상했던 것과 일치하는지, 팩맨이 실제로 목표를 향해 가는 도중에 탐색한 모든 사각형을 통과하는지 확인하세요.

[힌트] 스택을 자료구조로 사용하는 경우, mediumMaze에 대해 DFS 알고리즘이 찾은 해의 길이는 130이어야 합니다 (getSuccessors에서 제공하는 순서대로 successor를 프런티어에 추가하는 경우, 역순으로 추가하면 246이 될 수 있습

니다). 이것이 최소 비용 솔루션이어야 합니다. 그렇지 않다면 깊이 우선 탐색이 무엇을 잘못하고 있는지 생각해 보세요.

[채점] 아래 명령을 실행하여 작성한 코드가 자동 채점기의 모든 테스트 케이스를 통과하는지 확인하세요.

```
python autograder.py -q q1
```

4. Q2: 넓이 우선 탐색

search.py의 breadthFirstSearch 함수에서 넓이 우선 탐색(BFS) 알고리즘을 구현합니다. 이미 방문한 상태가 확장되지 않도록 그래프 탐색 알고리즘을 작성해야 합니다. 깊이 우선 탐색과 같이 아래 명령어를 통해서 여러분이 작성한 코드를 테스트하세요.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

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

BFS에서 최소 비용 솔루션을 찾았나요? 그렇지 않다면 작성한 코드를 확인하세요.

[힌트] 팩맨이 너무 느리게 움직인다면 -frameTime 0 옵션을 사용해 보세요.

[참고] 탐색 코드를 일반적으로 작성했다면 8개의 퍼즐 탐색 문제에서도 코드를 변경하지 않아도 똑같이 잘 작동해야 합니다.

```
python eightpuzzle.py
```

[채점] 아래 명령을 실행하여 작성한 코드가 자동 채점기의 모든 테스트 케이스를 통과하는지 확인하세요.

```
python autograder.py -q q2
```

5. Q3: 비용 함수 변경

BFS는 목표까지 가장 적은 노드들을 거치는 경로를 찾지만, 다른 의미에서 '최적'인 경로를 찾고 싶을 수도 있습니다. mediumDottedMaze와 mediumScaryMaze를 고려해 봅시다.

비용 함수를 변경하면 팩맨이 다른 경로를 찾도록 할 수 있습니다. 예를 들어 고스트가 있는 위험한 영역에 더 높은 비용을 할당하고 음식이 많은 영역에 더 적은 비용을 할당하면 제대로 구현된 팩맨 agent는 그에 따라 다르게 움직입니다.

search.py의 uniformCostSearch 함수에서 균일 비용 그래프 탐색(UCS) 알고리즘을 구현해야 합니다. util.py에서 제공하는 자료구조들이 도움이 될 수 있습니다. 구현 시, 다음 세 가지 레이아웃에서 모두 성공해야 하며, 아래 모든 agent들은 UCS agent로 사용하는 비용 함수만 다릅니다(agent와 비용 함수는 이미 작성되어 있음):

```
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 함수로부터 StayEastSearchAgent 및 StayWestSearchAgent에 대해서 각각 매우 낮은 경로 비용과 매우 높은 경로 비용을 얻어야 합니다(자세한 내용은 searchAgents.py 참조).

[채점] 아래 명령을 실행하여 작성한 코드가 자동 채점기의 모든 테스트 케이스를 통과하는지 확인하세요.

python autograder.py -q q3

6. Q4: A* 탐색

search.py의 aStarSearch 함수에 A* 그래프 탐색을 구현하세요. A*는 휴리스틱 함수를 인자로 받습니다. 휴리스틱은 탐색 문제의 상태(주 인자)와 문제 자체(참조 정보용)의 두 가지 인자를 받습니다. search.py의 nullHeuristic 휴리스틱 함수는 간단한 예입니다.

미로를 통과하여 고정된 위치로 가는 경로를 찾는 원래 문제에서 맨해튼 거리 휴리스틱을 사용하여 A* 구현을 테스트 할 수 있습니다(searchAgents.py에서 이미 manhattanHeuristic으로 구현됨).

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

A*가 균일 비용 탐색보다 약간 빠르게 최적의 솔루션을 찾는 것을 볼 수 있을 것입니다(구현에서 확장된 탐색 노드는 약 549개 대 620개이지만 우선 순위가 동점일 경우 수치가 약간 달라질 수 있습니다). 다양한 탐색 전략에 대해 openMaze에서는 어떤 결과가 나오는지 확인해 보세요.

[채점] 아래 명령을 실행하여 작성한 코드가 자동 채점기의 모든 테스트 케이스를 통과하는지 확인하세요.

python autograder.py -q q4

7. Q5: 모든 모서리 찾기

[참고] Q5는 Q2의 답을 기반으로 하기 때문에, Q5를 풀기 전에 Q2를 반드시 완료해야 합니다.

A*의 진가는 더 어려운 탐색 문제에서 드러납니다. 이제, 새로운 문제를 위한 휴리스틱을 설계해 봅시다.

주어진 미로의 각 모서리에 하나씩 4개의 점이 있다고 생각합니다. 새로운 탐색 문제는 (미로에 실제로 먹이가 있든 없든) 네 모서리에 모두 닿는 가장 짧은 경로를 찾는 것입니다. tinyCorners와 같은 일부 미로의 경우, 최단 경로가 항상 가장 가까운 점(음식)으로 먼저 가는 것이 아닙니다. (힌트: tinyCorners를 통과하는 가장 짧은 경로는 28단계입니다.)

searchAgents.py에서 CornersProblem 탐색 문제를 구현합니다. 네 모서리에 모두 도달했는지 여부를 감지하는 데 필요한 모든 정보를 나타내는 상태 표현을 적당하게 정의해야 합니다. 탐색 agent는 다음 문제들을 해결해야 합니다:

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

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

모든 점수를 받기 위해서는, 관련 없는 정보(예: 고스트의 위치, 여분의 음식이 있는 위치 등)를 포함하지 않는 상태 표현을 정의해야 합니다. 특히, 팩맨 GameState를 탐색 상태로 사용하지 않습니다. 코드가 매우 느려지거나, 오동작 할 수 있습니다.

CornersProblem 클래스의 인스턴스는 특정 상태가 아닌 전체 탐색 문제를 나타냅니다. 특정 상태는 여러분이 작성한 함수에 의해 반환되며, 함수는 여러분이 선택한 데이터 구조(예: 튜플, 집합 등)로 상태를 반환합니다.

또한, 프로그램이 실행되는 동안에는 탐색 알고리즘의 대기열에는 여러 상태가 동시에 존재하며, 이 상태들은 서로 독립적이어야 합니다. 즉, CornersProblem 객체 전체에 대해 하나의 상태만 가져서는 안 되며, 탐색 알고리즘에 제공할 다양한 상태를 클래스에서 생성할 수 있어야 합니다.

[힌트] 구현에서 참조해야 하는 게임 상태의 유일한 부분은 팩맨의 시작 위치와 네 모서리의 위치 뿐입니다.

[힌트] getSuccessors를 구현할 때 비용이 1인 자식을 successor 목록에 추가해야 합니다.

breadthFirstSearch를 구현하면 mediumCorners에서 탐색 노드가 2000개 미만으로 확장됩니다. 그러나 휴리스틱(A* 탐색과 함께 사용)을 사용하면 필요한 탐색량을 줄일 수 있습니다.

[채점] 아래 명령을 실행하여 작성한 코드가 모든 자동 채점기 테스트 케이스를 통과하는지 확인하세요.

```
python autograder.py -q q5
```

8. Q6: 모서리 문제-휴리스틱

[참고] Q6은 Q4의 답을 기반으로 하기 때문에, Q6을 풀기 전에 Q4를 반드시 완료해야 합니다.

cornersHeuristic의 CornersProblem에 대해 non-trivial하고 일관성 있는 휴리스틱을 구현합니다.

[참고] AStarCornersAgent는 다음 명령의 단축명령어입니다.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

허용 가능성 (Admissibility) vs 일관성 (Consistency): 휴리스틱은 탐색 상태를 입력 받아 가장 가까운 목표에 대한 비용을 추정한 숫자를 예측하는 함수입니다. 더 효과적인 휴리스틱일수록 실제 목표 비용에 더 가까운 값을 예측합니다. 허용 가능한, 즉 유효한 휴리스틱은 휴리스틱 값이 가장 가까운 목표까지의 실제 최단 경로 비용의 하한값이어야 하며 음수가 아니어야 합니다. 일관성을 유지하기 위해서는, 어떤 행동의 비용이 C인 경우 해당 행동을 취하면 휴리스틱이 최대 C만큼만 하락할 수 있다는 점도 추가로 유지해야 합니다.

그래프 탐색에서 정확성을 보장하기 위해서는 허용 가능성만으로는 충분하지 않으며, 일관성이라는 더 강력한 조건이 필요합니다. 최적화 문제에서 파생된 허용 가능한 휴리스틱은 일반적으로 일관성이 있습니다. 따라서 일반적으로 허용 가능한 휴리스틱을 먼저 고안하는 것이 쉽습니다. 그 후 휴리스틱이 실제로 일관성이 있는지 확인할 수 있습니다. 일관성을 보장하는 유일한 방법은 증명하는 것입니다. 그러나 확장된 노드의 f 값과 비교해 successor 노드들의 f 값들이 같거나 또는 심지어 더 높은 경우 일관성이 없는 것을 찾아낼 수 있습니다. 또한 UCS와 A*가 서로 다른 길이의 경로를 반환하는 경우에도 휴리스틱이 일관성이 없다고 할 수 있습니다. 어려운 문제이긴 합니다.

Non-Trivial 휴리스틱: Trivial 휴리스틱은 모든 곳에서 0을 반환하는 휴리스틱(UCS)과 실제 완료 비용을 계산하여 그 값을 취하는 휴리스틱이 있습니다. 전자는 시간을 절약할 수 없지만 후자는 자동 채점기에서 시간 초과로 판정할 수 있습니다. 총 계산 시간을 줄이는 것이 휴리스틱의 성능을 평가하는 정확한 방법이지만, 이 과제에서는 자동 채점기가 확장된 (즉 방문한) 노드들의 개수를 확인하는 것으로 평가합니다 (여기에 추가로 적당한 timeout이 적용됩니다).

[채점] non-trivial하고 음수가 아닌 일관된 휴리스틱이어야 점수를 받을 수 있습니다. 휴리스틱이 모든 목표 상태에서 0을 반환하고 음수 값을 반환하지 않는지 확인하세요. 휴리스틱이 확장하는 노드 수에 따라 점수가 매겨집니다:

휴리스틱이 확장하는 노드 수	점수
2,000개 이상	0/3
최대 2000개	1/3
최대 1600개	2/3
최대 1200개	3/3

휴리스틱이 일관되지 않으면 점수를 받을 수 없으니 주의하세요!

아래 명령을 실행하여 작성한 코드가 모든 자동 채점기 테스트 케이스를 통과하는지 확인하세요.

```
python autograder.py -q q6
```

9. Q7: 모든 점(과자) 먹기

[참고] Q7은 Q4의 답을 기반으로 하기 때문에, Q7을 풀기 전에 Q4를 반드시 완료해야 합니다.

이제 가능한 적은 단계로 팩맨이 음식을 모두 먹는 좀더 어려운 탐색 문제를 해결해 보겠습니다. 새로운 탐색 문제 정의가 필요합니다. (searchAgents.py의 FoodSearchProblem(이미 작성되어 있음)) 솔루션은 팩맨 세계의 모든 음식을 수집하는 경로로 정의됩니다. 현재 과제에서 솔루션은 고스트나 파워 알약을 고려하지 않고 벽, 일반 음식, 팩맨의 배치에만 의존합니다. (고스트를 피하는 것은 여기서는 고려하지 않습니다) 탐색 방법을 올바르게 작성했다면, null 휴리스틱(균일 비용 탐색과 동일)을 사용하는 A*로 코드 변경 없이(총 비용 7) testSearch에 대한 최적의 솔루션을 빠르게 찾을 수 있습니다.

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

[참고] AStarFoodSearchAgent 는 다음 명령의 단축 명령어입니다.

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

단순해 보이는 tinySearch에서도 UCS가 느려지기 시작한다는 것을 알 수 있습니다. 참고로, 제공되는 구현에서는 5057개의 탐색 노드를 확장한 후 길이 27의 경로를 찾는 데 2.5초 정도 걸립니다.

FoodSearchProblem를 위해 일관성 있는 휴리스틱을 구현해 searchAgents.py의 foodHeuristic에 작성하세요. trickySearch 보드에서 agent를 사용해 보세요:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

UCS agent는 16,000개 이상의 노드를 탐색하여 13초 정도의 시간으로 최적의 솔루션을 찾습니다.

음수가 아닌 모든 일관된 휴리스틱은 1점을 받습니다. 휴리스틱이 모든 목표 상태에서 0을 반환하고 음수 값을 반환하지 않는지 확인하세요. 휴리스틱이 확장하는 노드 수에 따라 추가 점수를 받을 수 있습니다:

휴리스틱이 확장하는 노드 수	점수
15,000개 이상	1/4
최대 15,000개	2/4

최대 12,000개	3/4
최대 9,000개	4/4 (전체 점수)
최대 7,000개	5/4 (추가 점수)

휴리스틱이 일관성이 없으면 점수를 받을 수 없으니 주의하세요. 짧은 시간 안에 mediumSearch를 풀 수 있나요? 그렇다면 매우 잘 작성되었거나 일관성 없는 휴리스틱을 작성한 것입니다.

[채점] 아래 명령을 실행하여 작성한 코드가 모든 자동 채점기 테스트 케이스를 통과하는지 확인하세요.

```
python autograder.py -q q7
```

10. Q8: 최적이 아닌 경로 탐색

A* 알고리즘과 좋은 휴리스틱을 사용하더라도 모든 점을 통과하는 최적의 경로를 찾는 것이 어려울 때가 있습니다. 이런 경우에도 합리적인 방법으로 좋은 경로를 빠르게 찾고 싶습니다. 이 문제에서는 항상 가장 가까운 점을 우선적으로 먹는 agent를 작성하게 됩니다. searchAgents.py에 가장 가까운 점으로의 경로를 찾는 핵심 기능이 누락된 ClosestDotSearchAgent가 구현되어 있습니다.

searchAgents.py의 findPathToClosestDot 함수를 구현하세요. Agent는 경로 비용이 350인 미로를 최적이 아닌 방식으로 1초 이내에 해결해야 합니다:

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

[힌트] findPathToClosestDot을 가장 빠르게 작성하는 방법은 목표 테스트가 누락된 AnyFoodSearchProblem을 채운 후 적절한 탐색 함수를 구현하세요. 솔루션은 매우 짧습니다.

여러분이 작성한 ClosestDotSearchAgent는 항상 미로를 통과하는 최단 경로를 찾지는 못할 것입니다. 왜 그런지 생각해 보고, 가장 가까운 점으로 계속 가는 것이 모든 점을 먹는 최단 경로를 찾지 못하는 작은 예제를 생각해보세요.

[채점] 아래 명령을 실행하여 작성한 코드가 모든 자동 채점기 테스트 케이스를 통과하는지 확인하세요.

```
python autograder.py -q q8
```