CSCE 4613 Artificial Intelligence Assignment 2: Pac-Man Finds Food

Levi Crider and Ethan Weems
lccrider@uark.edu —Student ID: 010914822
emweems@uark.edu — Student ID: 010903516
Due: February 28, 2023

Abstract

This assignment aimed to familiarize students with several fundamental searching algorithms. Having been given the code necessary to run a Pac-Man game, our task was to provide a means by which a Pac-Man agent can go from its initial position to a location containing its 'food'. This was completed for multiple maps containing a classic two-dimensional Pac-Man maze. We implemented two uninformed search algorithms: depth-first search and breadth-first search. We also implemented two informed search algorithms: uniform-cost search and A* search.

Environment Setup and Program Structure

This project necessitates the usage of Python 2.7. So, we are using a virtual environment with that version installed. All of our changes are made in the file 'search.py'. Our functions containing our algorithms are to receive one parameter: 'problem'. This is given. It contains the information including the position of the Pac-Man agent and the food. It also has the methods getStartState(), isGoalState(), and getSuccessors(current), which are used throughout our code. Our functions then need to return a list of directions to the goal in the following format: ['East', South', 'West', 'West']. These are the instructions "given" to the agent to reach the food from its starting position. Refer to the image provided at the end of the report, which demonstrates the testing of an algorithm within our virtual environment.

Depth-First Search

Our first task was to implement a Depth-First searching (DFS) algorithm. In DFS, the goal is to fully explore a single path of a search before continuing to the next. For example, if the Pac-Man agent can initially move west, the algorithm will proceed from the initial location to the destination fully prior to considering directions other than west. We chose to use the stack data structure, which was provided in the file named 'util'. This is a first-in-last-out (FILO) data structure. FILO is useful here because it allows us to add all children nodes for each node, but proceed by only considering the most recent child added.

DFS Results

Metric	Tiny Maze	Medium Maze	Big Maze
Total Cost	10	130	210
Nodes Expanded	15	146	390
Score	500	380	300

Breadth-First Search

Next, we were to implement a Breadth-First searching (BFS) algorithm. BFS, unlike DFS, will consider all nodes at a given depth before any of their children. Hence, we chose to use the queue data structure, which was also provided in 'util'. Queues are a first-in, first-out (FIFO) data structure. We selected this data structure so that during the iteration of our loop, the algorithm will always revert to the oldest node stored for the next evaluation. For instance, if some node has two children and two siblings, the sibling nodes will have their children added to the queue before the original node's children are evaluated.

BFS Results

Metric	Tiny Maze	Medium Maze	Big Maze
Total Cost	8	68	210
Nodes Expanded	15	269	620
Score	502	442	300

Uniform-Cost Search

Then, we needed to implement a Uniform-Cost searching (UCS) algorithm. This algorithm evaluates all possible paths to the goal and then chooses the one with the lowest cost. So, we chose to use the priority queue provided in 'util'. This data structure will keep all of the paths to the food sorted by its total cost. Fundamentally, it will perform similarly to BFS, except that it is less concerned with reaching the goal but rather reaching the goal with the lowest cost. For each path, it will compare the total cost to the cheapest path currently being tracked.

UCS Results

Metric	Tiny Maze	Medium Maze	Big Maze
Total Cost	8	68	210
Nodes Expanded	16	275	620
Score	502	442	300

A* Search

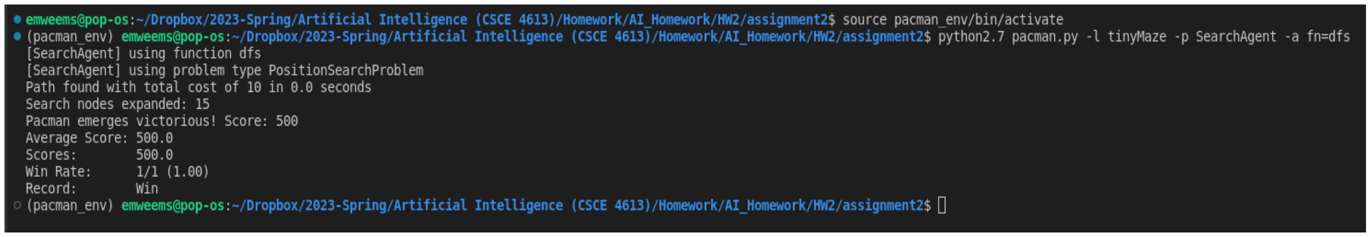
Lastly, we needed to implement the A* search algorithm. This algorithm in this context is similar to the UCS algorithm implemented above. We still use the priority queue data structure, but instead of choosing the lowest cost as our priority, we choose the option with the lowest cost and the lowest remaining cost to the destination. To do this, we used the ’Manhatten distance’ function and passed in $previousCost + totalRemainCost$ as our priority into the priority queue. This will choose the option with the least total remaining cost at each choice that the algorithm has to make. This algorithm found the best solution and always should because it is complete and optimal.

A* Results

Metric	Tiny Maze	Medium Maze	Big Maze
Total Cost	8	68	210
Nodes Expanded	14	221	549
Score	502	442	300

Acknowledgments

This assignment was completed as a part of the CSCE 4613: Artificial Intelligence course at the University of Arkansas during the Spring 2023 semester. The class is taught by Dr. Khoa Luu, and additional information is provided by teaching assistant Thanh-Dat Truong. The authors would like to thank them both for their contributions to their education.



```
● emweems@pop-os:~/Dropbox/2023-Spring/Artificial Intelligence (CSCE 4613)/Homework/AI_Homework/HW2/assignment2$ source pacman_env/bin/activate
● (pacman_env) emweems@pop-os:~/Dropbox/2023-Spring/Artificial Intelligence (CSCE 4613)/Homework/AI_Homework/HW2/assignment2$ python2.7 pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
  [SearchAgent] using function dfs
  [SearchAgent] using problem type PositionSearchProblem
  Path found with total cost of 10 in 0.0 seconds
  Search nodes expanded: 15
  Pacman emerges victorious! Score: 500
  Average Score: 500.0
  Scores:        500.0
  Win Rate:      1/1 (1.00)
  Record:        Win
○ (pacman_env) emweems@pop-os:~/Dropbox/2023-Spring/Artificial Intelligence (CSCE 4613)/Homework/AI_Homework/HW2/assignment2$ 
```

Example command line usage of our algorithms.