ITEC

CS-300 Artificial Intelligence

Midterm Project

Dr. Nguyen Ngoc Thao

Msc. Nguyen Hai Dang - Msc. Do Trong Le - Nguyen Quang Thuc

Group info:

Member:

- Le Anh Dung - 2059006
- Nguyen Hoang Truong Thinh – 2059042

Grade: 20BIT1

Assignment of tasks:

- Problem modeling (#TODO 1 - #TODO 10, problems.py): Anh Dũng
- SearchAgent corresponds to BFS, DFS, UCS, and A* search strategies (#TODO 11 - #TODO 16, searchAgents.py): Anh Dũng
- Detailed implementation of four search algorithms (#TODO 17 - #TODO 22, search.py): Trường Thịnh
- Data structure serving the algorithms (#TODO 23– #TODO 25; util.py): Trường Thịnh
- Report: Trường Thịnh, Anh Dũng

Overview: This is a project that implements different search algorithms to solve the classic Pacman game. The project uses the A* search algorithm, Breadth-First Search (BFS), Depth-First Search (DFS), and Uniform Cost Search (UCS) to help Pacman find the shortest path to eat one or multiple food in the provided map without using the keyboard to control him and with no ghost in the game.

Modeling problem:
State: The state in this problem consists of a tuple of two elements. The first element is a tuple representing the current position of Pacman on the grid, and the second element is a Grid object representing the current state of the food on the grid.

Move state (edge): The move state is represented by a tuple containing three elements: the next state (as described above), the direction of the move taken to get to that state, and the cost of that move (which is always 1 in this case).

Cost: The cost of each move is 1 in this problem. The cost of a sequence of moves is calculated as the sum of the costs of each individual move.

Start state: The start state of the problem is determined by the starting game state provided as input to the constructor of the SingleFoodSearchProblem class. It consists of a tuple containing the current position of Pacman and the current state of the food on the grid.

Goal state: The goal state of the problem is reached when there is no more food left on the grid. This is determined by checking if the count of remaining food elements in the state (the second element of the state tuple) is equal to 0.

How to find adjacent states: To find adjacent states, the getSuccessors method is used. This method takes a state as input and returns a list of successor states, along with the move taken to reach each successor state and the cost of that move. To generate successor states, the method considers all possible moves from the current state in each of the four cardinal directions (north, south, east, west). For each valid move, it creates a new state tuple by updating the position of Pacman and the state of the food grid, and adds this as a successor state to the list.

Interface structure and operation of search agents:

1. BFS (Breadth-First Search)

The structure of BFS includes a queue data structure to keep track of the nodes that have been visited and the nodes that are still to be visited.

The BFS algorithm works as follows:

+ Initialize a queue with the root node.
+ Dequeue the first node in the queue and mark it as visited.
+ Enqueue all of the neighboring nodes that have not been visited and mark them as visited.
+ Repeat steps 2-3 for each node in the queue until the queue is empty.

2. DFS (Depth-First Search)

The structure of DFS includes a stack data structure to keep track of the nodes that have been visited and the nodes that are still to be visited.

The DFS algorithm works as follows:

+ Initialize a stack with the root node.
+ Pop the top node from the stack and mark it as visited.
+ Push all of the neighboring nodes that have not been visited onto the stack.
+ Repeat steps 2-3 for each node in the stack until the stack is empty.

3. A (A-Star)*

The structure of A* includes a priority queue and a set of visited states.

The A* algorithm works as follows:

+ Initialize a priority queue with the starting state.
+ Calculate the heuristic value for the current state.
+ Check the first state in the priority queue. If this state is the goal state, the solution has been found and the algorithm terminates. If not, create new states from this state and add them to the priority queue. These new states are prioritized based on the sum of their heuristic value and the cost to get from the starting state to the new state.
+ Mark the current state as visited.
+ Repeat the above steps until a solution is found or the priority queue is empty.

4. UCS (Uniform-Cost Search)

The structure of UCS includes a priority queue data structure to keep track of the nodes that have been visited and the nodes that are still to be visited, sorted by their path cost.

The UCS algorithm works as follows:

+ Initialize a priority queue with the root node, with a path cost of 0.
+ Dequeue the node with the lowest path cost from the priority queue and mark it as visited. If the current node is the goal node, the solution has been found and the algorithm terminates.
+ Enqueue all of the neighboring nodes that have not been visited, with their path cost updated to the sum of the path cost to the current node and the cost to move from the current node to the neighboring node.
+ If a neighboring node has already been visited, update its path cost if the new path cost is lower than its current path cost.
+ Repeat steps 2-5 for each node in the priority queue until the queue is empty or the goal node is found.

How to choose two heuristic functions for A* algorithm, explaining the consistent and admissible properties simply:

Discover functions that are acceptable and consistent when selecting heuristics for A*. The condition that the estimated cost from any state s and its successor state s' to the goal state, plus the estimated cost from s' to the goal state, is never less than the actual cost from s to s' plus the estimated cost from s' to the goal state is satisfied by a consistent heuristic function. An admissible heuristic function never overestimates the actual cost to reach the goal state from the current state.

Two ways heuristic function for the Pac-man project:

+ Manhattan distance: This algorithm calculates the total of the absolute differences in the x and y values of the current state and the target state to determine the distance between the two states. This algorithm is acceptable and reliable, and it frequently performs well in maze-like settings like those found in the Pac-Man video game.
+ Euclidean distance: This heuristic calculates the Euclidean distance between the coordinates of the present state and the target state to determine the separation between them. This heuristic is equally admissible and reliable, but in maze-like

settings with walls and obstructions, it might not perform as well as the Manhattan distance heuristic.

Characteristics of data structure settings:
The three data structures implemented in the code are Stack, Queue, and PriorityQueue.

Stack is a Last-In-First-Out (LIFO) data structure where elements are added to the top and removed from the top.

Queue is a First-In-First-Out (FIFO) data structure where elements are added to the back and removed from the front.

PriorityQueue is a queue where each element has a priority associated with it. The elements are dequeued according to their priority, with the highest priority element dequeued first.

When implementing these data structures, it is important to implement the appropriate methods to add and remove elements, and to keep track of the size of the data structure. For PriorityQueue, it is also important to maintain the heap property, where the parent node is always greater than or equal to its children nodes.

Additionally, the implementation of these data structures can vary depending on the programming language and the underlying data structures used. For example, the Queue implementation in the code uses a list and inserts elements at the front of the list, while in other implementations, a linked list or a circular buffer may be used.

We are also use some python library for this project: heapq and numpy
1. Heapq
- Heapq is a Python module that offers a heap queue algorithm application.
- It gives users access to and methods for working with heap data structures, which are basically priority lists.
- A collection is used to symbolize the heap data structure, with each entry standing for a node in the heap tree.
- The list's components are arranged so that the top item is always the lowest (or highest, based on the implementation) element in the heap.
- Applications that need quick access to the lowest (or biggest) part in a list of values can benefit from the heap module.

2. Numpy
- It offers a strong collection object that can be applied to mathematical calculations.
- Although the array object is identical to a list and allows numerous mathematical functions, it is more effective for big arrays.
- Arrays can depict vectors and other multi-dimensional data because they can have any number of dimensions.
- Numpy offers methods for accessing and modifying arrays, such as transposing, reshaping, and slicing.

Advantages and disadvantages in the project:

Advantages

1. Hands-on experience: We have the chance to gain practical experience creating a fundamental AI system or AI game thanks to the Pacman AI initiative. Since we're interested in pursuing a job in AI or machine learning, this exposure could be extremely useful.
2. Understanding Search Algorithm: Various AI methods, including BFS search, DFS search, A* search, UCS search, and Search Agent, are being implemented as part of the initiative. Working on the project will help me better grasp these AI algorithms, how they operate, and how to use them in real-world situations.
3. Improving problem-solving skills: The Pacman AI project involves solving complex problems, which can help me improve our problem-solving skills. These skills can be beneficial in various fields, including computer science, engineering, and mathematics

Disadvantages

1. Time Complexity: Using any of these algorithms to determine the best route for Pacman to approach the food could take a significant amount of time, depending on how complicated the game is. The program may take longer to run the more difficult the game is.

2. Memory Complexity: The methods may need a lot of memory to keep the search tree and the game state, depending on the size of the game board. This might cause speed problems and make the game lag.

3. Ineffective Pathfinding: Although the UCS, BFS, DFS, and A* algorithms are good at finding paths, they may not always be the most direct routes for Pacman to get to the food. Sometimes, they may omit shorter pathways in favor of lengthier ones.

4. Reliance on Heuristics: The efficacy of the heuristic function used is crucial to the performance of the A* algorithm. Pacman might follow less-than-ideal routes if the heuristic function is poorly constructed.

5. Restricted Exploration: DFS might not investigate every route before coming up with the answer, which could result in less-than-ideal routes.

References:

https://github.com/zhangjiedev/pacman

https://github.com/asqiriba/cs188-pacman/tree/master/Project01

https://www.youtube.com/watch?v=ataGotQ7ir8

https://www.youtube.com/watch?v=NKKfW8X9uYk

https://courses.edx.org/c4x/BerkeleyX/CS188.1x-4/asset/lec4-lg.pdf

https://www.youtube.com/watch?v=JtiK0DOeI4A

https://www.youtube.com/watch?v=W9zSr9jnoqY&list=PLWF9TXck7O_zsqnufs62t26_LJnLo4VRA&index=5&ab_channel=LearningOrbis

https://viblo.asia/p/a-search-algorithm-aWj53BN1l6m

https://www.geeksforgeeks.org/a-search-algorithm/

https://favtutor.com/blogs/breadth-first-search-python

https://plainenglish.io/blog/uniform-cost-search-ucs-algorithm-in-python-ec3ee03fca9f

https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/