ITEC

CS-300 Artificial Intelligence

midterm project

PACMAN

by

1959002 1959024 1959035 1959040

TABLE OF CONTENTS

I. GROUP INFORMATION

For this Midterm project, our group consists of 4 people. All of us are members of Cycle 13, ITEC program, and class 19BIT. Members of our group include:

- **1959002** – **Phạm Đình Chương**
- **1959024** – **Nguyễn Cao Nhân** (Group Leader)
- **1959035** – **Lê Trần Bá Tân**
- **1959040** – **Hồ Ngọc Thảo Trang**

II. JOB DIVISION AND WORKING PHASES

1. JOB DIVISION

ID	Name	Job	Contributed
1959002	**Pham Dinh Chuong**	Breadth-First Search	100%
1959024	**Nguyen Cao Nhan**	A* Search	100%
1959035	**Le Tran Ba Tan**	Depth-First Search	100%
1959040	**Ho Ngoc Thao Trang**	Uniform Cost Search	100%

2. WORKING PHASES

The working course of our group is divided into main phases:

- **Phase 1:** An online meeting was organized and our team worked together to come up with an agreement on the way we use and structure the shared utilities (the data structures and helper functions), as well as the way we organized the problems.
- **Phase 2:** We divided 4 search algorithms for 4 members, each of the members is responsible for programming and structuring a search algorithm that is capable of solving both single food and multiple food problems.
- **Phase 3:** The leader of the group merged the works of all members and tested the search algorithms on different layouts (the provided ones by the lecturer and some additional layouts found from the Internet or self-made).
- **Phase 4:** Our team completed the report for the project.

III. PROBLEM FORMULATING

The problem can be viewed as a searching problem, where the implementation of the searching algorithms is finding the set of movements (along the path) to get from the initial state of the problem to the goal state.

In detail, the problem can be described with the following element:

- **State:** *(node in search tree)* each state of the problem includes information about the current position of Pacman (a tuple of x and y-coordinates), the current food grid (a 2-dimensional list of boolean values showing if a position includes food or not), the layout of the maze (a 2-dimensional list of boolean values showing if a position is a wall or a valid path to travel). The state of the game is pre-designed in the source code and our team implemented the class GameState to use as a single state of the game. For easy implementation, our team added to the pre-defined GameState class an additional array to store the path traveled to get from the initial state (or a defined starting position – in case of the multiple-food problems) to the current position.

- **Action:** *(edge in search tree)* an action is a move of the Pacman from one position to another position, and the move must be among the valid ones (in one of the directions: North, South, East, West or Stop and must not go over the wall). When Pacman reaches a position with food, it automatically eats the food at that location without requiring additional action. Each action will update the state's information to a newer state, changing the location of Pacman, the status of the food grid (if changed).
- **Path cost:** Pacman can perform 1 move for a single unit time, during which the score is decreased by 1. So the path cost for the problem is 1.
- **Initial state:** The initial state of the problem is a state object with information about the layout of the maze, the food grid and the initial position of Pacman.
- **Goal state:** The goal state of the problem is when the number of food in the food grid decreased by 1 compared to the initial state (for single-food problem) or when the number of food in the food grid is 0 (for multiple-food problems)
- **Finding successors:** To find successors, the current state is used to find out the list of legal moves for Pacman (except the Stop move), using a pre-defined function of GameState class. Each move will correspond to a valid successor with new information about the new position of Pacman, a new state of the food grid (if changed), and an updated list of paths traveled. We use the pre-defined function of class GameState to get a set of new states from legal moves and return through the problem class a tuple illustrating the action and the state after performing the action.

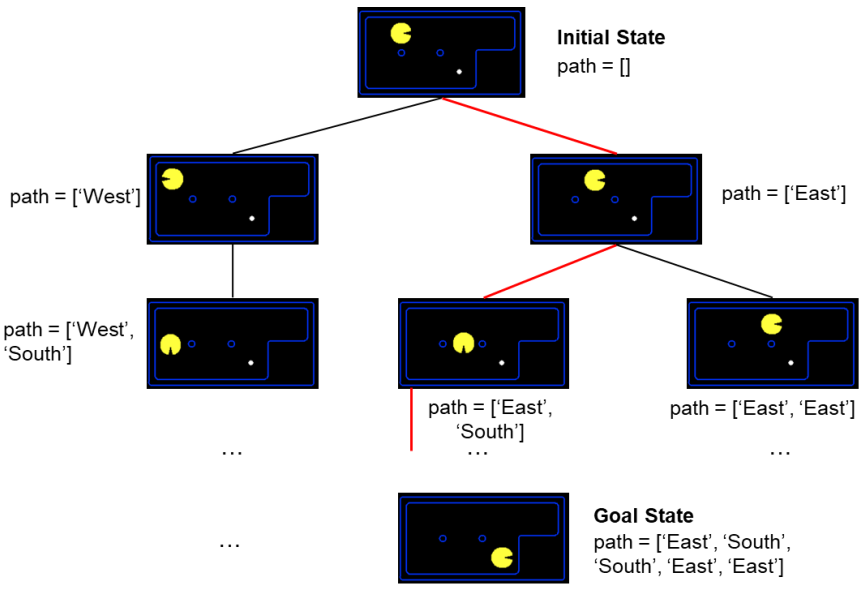

Figure 1 - Search tree for BFS Single Food Problem on minimaxClassic maze

IV. SEARCH AGENTS

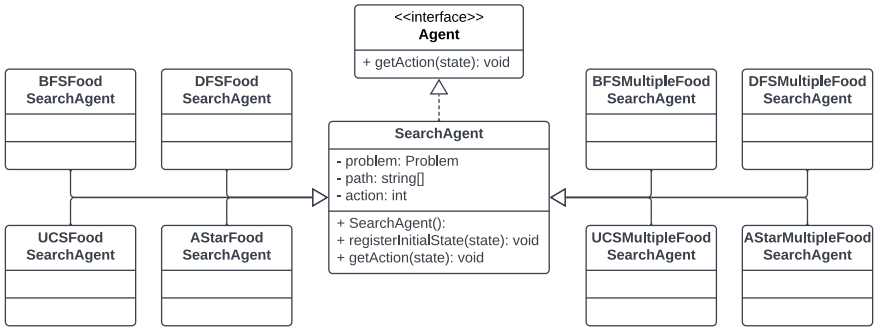

Figure 2 - UML Class Diagram of Agent interface and related classes

- **Agent interface:** We use the pre-defined Agent interface in the source code including a method *getAction()*, that will return the next action to go for the Pacman from the set of actions defined by the searching algorithm.
- The **SearchAgent** class implements the Agent interface:
  - Attributes:
    - ***problem***: the current solving problem's type, which is an object of type Problem defined based on whether the problem is single-food search or multiple-food search.
    - ***path***: a set of actions in form of a Direction object, that is solved and returned by the search functions.
    - ***action***: an attribute works as the iterator over the traversing path.
  - Methods
    - ***registerInitialState()*** that is called when the problem is first started, to provide initial information about the problem and in this function, the finding of the path to the goal by the search functions is called.
    - ***getAction()*** that will return the next action in move.
- We created 8 subclasses that inherited the SearchAgent interface, each can be used to call in the start command for the problem to identify the type of problem (single-food or multiple-food search) used and the type of searching algorithms (BFS, DFS, UCS, or A*) used.

V. CODING IMPLEMENTATION AND CHOICE OF HEURISTIC FUNCTIONS

1. BFS, DFS, AND UCS

For the multiple-food search problems, our team's first attempt was to break the problem into smaller single-food search problems and perform the single problem one by one until all of the food in the food grid is eaten. However, our team figured out that there is one difficulty with this approach: deciding the order of the food to eat is a challenging task.

After researching for a while over the problem and how each algorithm works, our team figured out that for BFS, DFS, and UCS, the iteration of the algorithms doesn't depend on the goal, which is the position of the food. So suppose there is no food in the food grid, the algorithms just find the way to travel the maze until reaching a defined goal state based on the property of the algorithm.

Because of that, our team's implementation of BFS, DFS, and UCS for multiple-food search problems is not very different from the single-food problems. We added the same implementation as the single-food problem, and loop until there is no food in the food grid. For each loop, the Pacman just travels the maze until reaching a position with food, when the code broke out of the loop, add up the traveled path to the overall path, reset the starting position for the next loop and then restart the search algorithms.

2. A*

For the A* algorithm, we chose Manhattan distance as the heuristic function for the algorithm.

A. SINGLE-FOOD PROBLEM

For the single-food problems, the heuristic value of each position is calculated based on the Manhattan distance between that position and the location of the food.

$$h(n) = |pos(0) - food(0)| + |pos(1) - food(1)|$$

Admissibility: Manhattan distance is ensured to be admissible because there are 4 possible movements for the Pacman in 4 directions and the calculatation of Manhattan distance, based on the difference between x and y-coordinates, is the same as the optimized path available for Pacman to travel the food in case there are no walls in the path. Therefore, the Manhattan distance doesn't overestimate the real cost from a position to the food (as in the real path, for each wall met, the Pacman has to take a detour and it costs at least 2 more steps).

Consistency: Each move of Pacman costs a unit cost and for each of the available new position, since Pacman can only moves in 4 directions (not diagonal), the value of heuristic for the next position can only be 1 value larger or smaller than the current position's heuristic. We can then assure that $h(n) \leq 1 + h(n') \Leftrightarrow h(n) \leq c(n, a, n') + h(n')$. Therefore, Manhattan distance is also a consistent heuristic.

B. MULTIPLE-FOOD PROBLEM

For multiple-food problems, our team designed our heuristic function based on Manhattan distance and the real observation on how the Pacman will travel to eat all the food. Similar to the remaining 3 algorithms, we assumed the A* multiple-food search problem will look at each food as a goal for a single-food problem and will execute all until the food grid is empty. So our calculation for the heuristic function will be: only add up the Manhattan distance between the current position and the closest food. Then loop until all the food is considered, each iteration adds up the Manhattan distance between the last eaten food and the nearest food to that. This can estimate very close to the optimized path the Pacman will travel to eat all the food in case there are no walls.

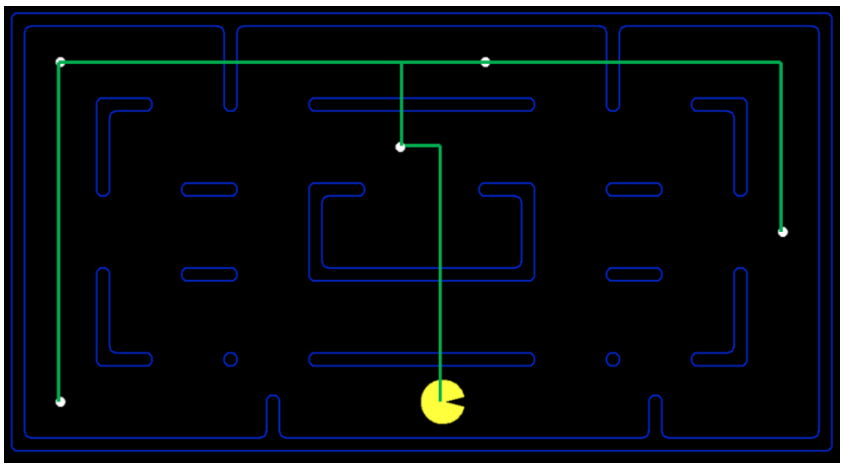

Figure 3 - Heuristic value for initial Pacman position based on our team's heuristic function

Comparison between the costs of paths returned by the A* algorithm with the remaining algorithms on 15 different layouts is shown in the table below. For multiple-food layouts, the path costs of A* are not larger than those of BFS and UCS. Our team predicts that the heuristic function chosen is admissible and consistent as the BFS and UCS results are likely optimized, since each movement costs the same for this problem.

Group	Layout	BFS	DFS	UCS	A*
SINGLE FOOD	bigMaze	210	210	210	210
	mediumClassic	14	38	14	14
	mediumMaze	68	76	68	68
	minimaxClassic	5	7	5	5
	openMaze	54	146	54	54
	noWallsMaze	54	54	54	54
	smallMaze	19	29	19	19
MULTIPLE FOOD	bigMazeMultipleFood	304	556	304	304
	bigMazeSelfMade	334	852	334	334

	bigSearch	336	459	336	336
	mediumClassicMultipleFood	49	158	49	49
	mediumMazeMultipleFood	180	326	180	180
	mediumSearch	170	190	170	170
	tinySearch	31	41	31	31
	trickySearch	68	82	68	68

Table 1 - Comparison between costs of returned paths between the algorithms on different layouts

VI. DATA STRUCTURES

For this project, Stack (for DFS), Queue (for BFS), and Priority Queue (for UCS and A*) are the data structures required. Our team used the simple implementation with basic helper functions for Stack, Queue, and Priority Queue based on the code of members during the lab assignments.

However, for easy implementation, we also used the pre-defined structure for the Priority Queue with function to be used with A*, where the data structure can be used the same way as priority queue but will automatically call the function for calculating heuristic value and add up before putting into the priority queue.

```
def push(self, item, cost):
    super().push(item, cost + self.function(item[0], self.problem))
    pass
```

Figure 4 - Implementation of push function inherits original push function of PriorityQueue class

VII. CONVENIENCE AND DIFFICULTIES

1. CONVENIENCE

- Our team can reuse the code from week lab assignments and only modified some parts to use for this project. Compared to having to code everything from scratch, this saved up a lot of time for our team and we could spend more time on deciding the heuristic function to be used for multiple food searches with A*.
- Although the task requires designing the problem structure, functions for use, there are multiple pre-defined helper functions inside the class GameState. So our team can use those functions instead of having to write from scratch all helper functions required.

2. DIFFICULTIES

- There are many new concepts our team had to deal with, especially we are all quite new to Python and have not yet been experienced in coding in a complete project with many files. Because of that, it cost quite a lot of time for our team to understand and can start with the project.
- The current schedule of this semester is very busy, so our team had a limited amount of time to do all the assignments. It was quite stressful but we managed to complete all of the tasks to the best of our ability.
- During the given time for completing this assignment, 2 of 4 members in our team were infected by the Covid virus. So we lost quite an amount of time for recovery (to good enough health condition) before we could catch up with the assignment. This was also a challenge for our team.

VIII. REFERENCES

[1] S. Ran, "Search in Pacman Project Report," n.d. [Online]. Available: https://rshcaroline.github.io/research/resources/pacman-report.pdf.

[2] R. Hu, "Good A* heuristics for the first pacman project?," 15 October 2011. [Online]. Available: https://www.reddit.com/r/aiclass/comments/ld0y7/good_a_heuristics_for_the_first_pacman_project/.

[3] Wikipedia, "Taxicab geometry," n.d. [Online]. Available: https://en.wikipedia.org/wiki/Taxicab_geometry.

[4] GeeksForGeeks, "Heap and Priority Queue using heapq module in Python," 1 Oct 2020. [Online]. Available: https://www.geeksforgeeks.org/heap-and-priority-queue-using-heapq-module-in-python/.

[5] Stuart Russell, Peter Norvig, "Conditions for optimality: Admissibility and consistency," in *Artificial Intelligence: A Modern Approach (3rd Edition)*, New Jersey, Pearson Longman, 2010, pp. 94-99.

[6] Nguyễn Ngọc Thảo, Nguyễn Hải Minh, "CS300 Lecture Slide - Informed Search Strategies," Ho Chi Minh City, n.d.

[7] Nguyễn Ngọc Thảo, Nguyễn Hải Minh, "CS300 Lecture Slide - Uninformed Search Strategies," Ho Chi Minh City, n.d.