**VIETNAM NATIONAL UNIVERSITY**

**UNIVERSITY OF SCIENCE**

**FACULTY OF INFORMATION TECHNOLOGY**



**PROJECT 01**

**ARTIFICIAL INTELLIGENCE**

VÕ VĂN QUỐC HUY 18127113

VŨ CÔNG MINH 18127155

TỪ KIẾN VINH 18127255

TỪ KIẾN HOA 18127260

**Ho Chi Minh City – 2020**

# TABLE OF CONTENTS

[TABLE OF CONTENTS 2](#_Toc48487529)

[Assignment plan and overall 3](#_Toc48487530)

[How to play 4](#_Toc48487531)

[Comparison between algorithms 8](#_Toc48487532)

[Explaning code: pacman.py 10](#_Toc48487533)

[Explaning code: Searching\_Algorithm.py 15](#_Toc48487534)

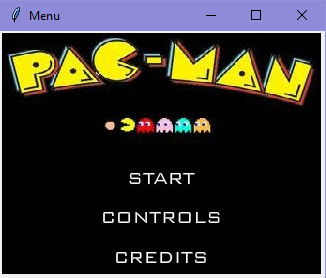
[References 17](#_Toc48487535)

# Assignment plan and overall

|  |  |  |
| --- | --- | --- |
| No. | Specification | Complete |
| 1 | Finish level 1 successfully. | ✓ |
| 2 | Finish level 2 successfully. | ✓ |
| 3 | Finish level 3 successfully. | ✓ |
| 4 | Finish level 4 successfully. | ✓ |
| 5 | Graphical demonstration of each step of the running process. | ✓ |
| 6 | Generate at least 5 maps with difference in number and structure of walls, monsters, and food. | ✓ |
| 7 | Report | ✓ |

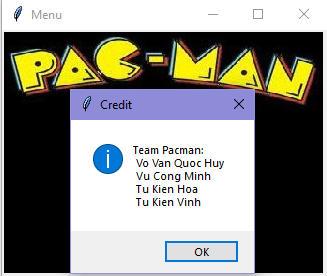
# How to play

Start window



If you click START without setting from CONTROLS, it will be **level 1** and **map1.**

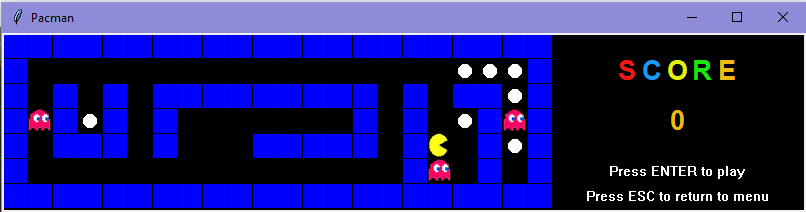
Control window Credit window

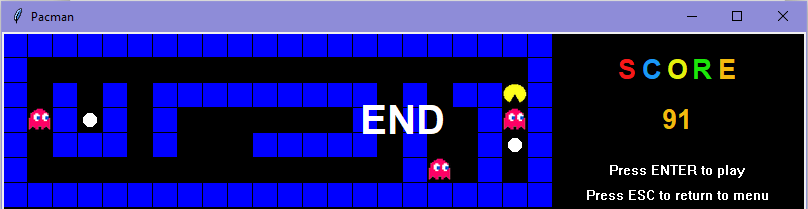
Set up level, input map (random or map name) and click “Play” or “Run” to return start window.

(Both of them will allow Pacman run automatically with algorithms because we haven’t built Play Mode for player vs computer)

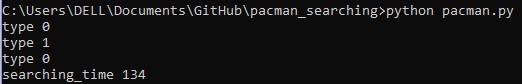
After click START, this is our main window to run algorithms:



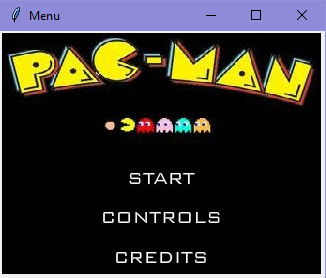
Press Enter to watch Pacman running … and finished: (example at lv2)



There are also some information printed out on console: (explain: there are 3 ghost, two of them have type 0, and the another one has type 1; Searching time means the number of visited node when we call a searching algorithm)

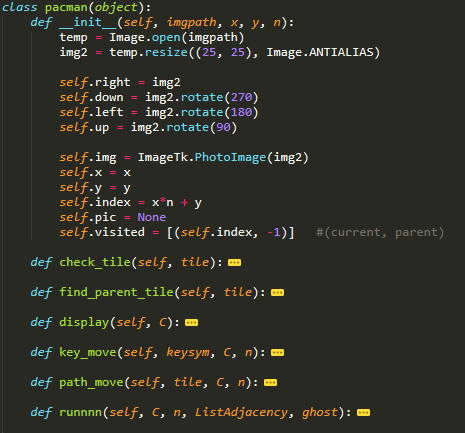


After all, press ESC to return menu (Start window):

Explaning code: Object.py

**class pacman(object):**

* Initialize pacman character from image file, resize to fit a block unit\*unit (unit default is 25).
* Draw pacman at position x, y read from the map file.
* Each time the pacman moves, it erases the previous position and redraws it
* Keep a visitedList for tracing back at lv3 and lv4. Each node is a tuple (current\_tile, parent)
* There are two types of moving: follow key symbol (“Up Right Down Left”) or move to a given index.
* Function runnnn dodge monster by random a move in its AdjacencyList which



**class food(object):**

* Initialize the cells with the value Food as a white circle.
* If the cells are eaten, they will be deleted by themselves.

**class monster(object):**

* Generate from Monster image file and draw in place on maze.
* Like the pacman, each time Monster moves will be deleted and redrawn in a new position.
* Will initialize random positions from array of positions and Monster will randomly move.
* Chase the pacman calculated according to the distance closest to the pacman.
* In lv3: Monster move around initial location (9 tiles available)
* In lv4: Monster is chosen a chase way with its random type.

+ Type 0: 3 steps chasing -> 5 steps random move -> 3 steps chasing …

+ Type 1: If Manhattan distance between it and pacman > 5 then random move, else chasing.



# Comparison between algorithms

1. Level 1:

|  |  |  |
| --- | --- | --- |
| Map 1 | Searching time | Sum length paths |
| BFS | 284 | 53 |
| A\* | 87 | 53 |

|  |  |  |
| --- | --- | --- |
| Map 2 | Searching time | Sum length paths |
| BFS | 430 | 140 |
| A\* | 240 | 140 |

|  |  |  |
| --- | --- | --- |
| Map 3 | Searching time | Sum length paths |
| BFS | 91 | 133 |
| A\* | 153 | 133 |

|  |  |  |
| --- | --- | --- |
| Map 4 | Searching time | Sum length paths |
| BFS | 560 | 119 |
| A\* | 295 | 119 |

|  |  |  |
| --- | --- | --- |
| Map 5 | Searching time | Sum length paths |
| BFS | 120 | 44 |
| A\* | 195 | 44 |

1. Level 2:

|  |  |  |
| --- | --- | --- |
| Map 1 | Searching time | Sum length paths |
| BFS | 284 | 53 |
| A\* | 134 | 46 |

|  |  |  |
| --- | --- | --- |
| Map 2 | Searching time | Sum length paths |
| BFS | 430 | 140 |
| A\* | 229 | 140 |

|  |  |  |
| --- | --- | --- |
| Map 3 | Searching time | Sum length paths |
| BFS |  |  |
| A\* | 154 | 7 |

|  |  |  |
| --- | --- | --- |
| Map 4 | Searching time | Sum length paths |
| BFS | 253 | 23 |
| A\* | 53 | 25 |

|  |  |  |
| --- | --- | --- |
| Map 5 | Searching time | Sum length paths |
| BFS | 253 | 23 |
| A\* | 189 | 25 |

# Explaning code: pacman.py

**Function random\_Maze():**

* Randomly initialize two variables for map size, then create a random matrix with three values of 0, 1, 2 and then surround the matrix with value 1.
* Then random the number and value of Ghost position corresponding to a random position in the matrix.
* Assign matrix into array corresponding to each row of matrix and return the results.

**Function handle\_input():**

* Initialize the variable **lst** as an array, open the file containing the position and matrix.
* For each line read, it is detached and fed into **lst**

**Function create\_maze(C):**

* For each input box from the matrix will be considered at that position as Wall or Food or Ghost.
* For each position there will be additional functions.
* Display all objects on canvas.

**Function create\_data(C):**

* If level = 2 then converts the positions with Monster value to 1. Else its value is 0 because monster’s initial location is viewed as road(visitable).
* Go through the maze array, if it is a sugar or Food value, enter the **ListAdjacency** matrix. Each element in ListAdjacency is a list of tuples **(node\_adjacency, node\_properties).** If current node is wall, ListAdjacency of it will be [None]
* Create List of moveable tiles for each ghost for random around initial location function.

**Function display\_score():**

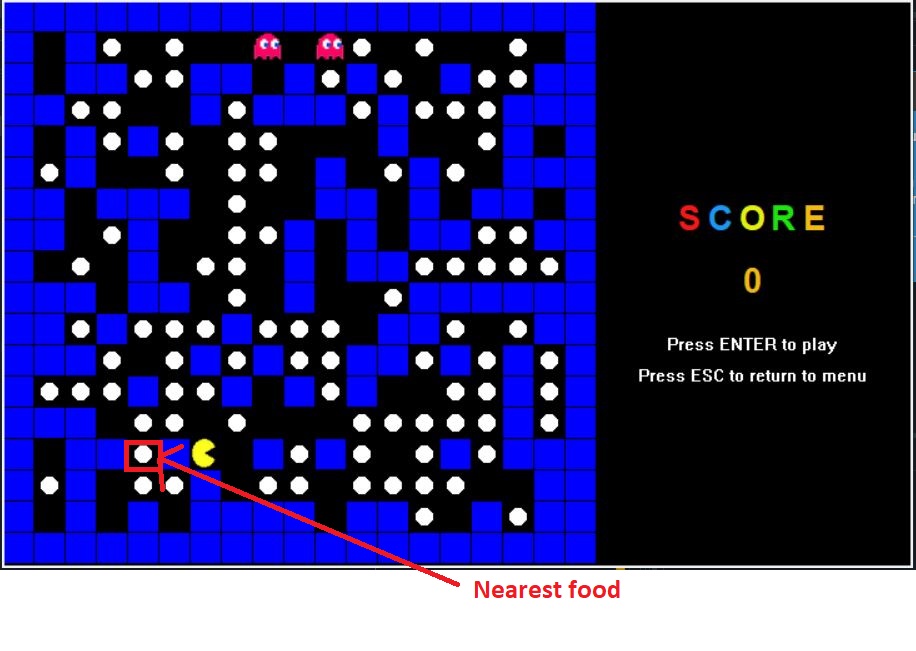
* Draw a score board

**Function sort\_Food():**

* Sort ListFood by ascending distance from pacman location.

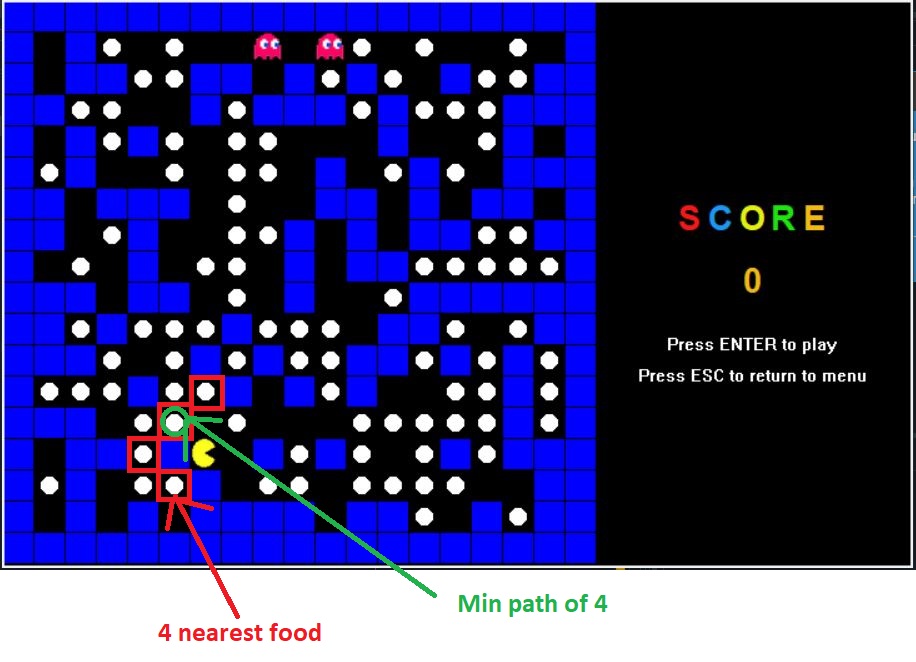
**Function nearest\_food\_tactic1():**

* Iterate each food object in **ListFood**.
* While loop for **ListFood** (While len(**ListFood**) > 0):
* Sort food objects in ListFood by sort\_Food() function.
* Calculate path from Pacman’s current position to ListFood[0] (food that has nearest distance calculated by Manhattan distance between pacman’s position and food’s position) using A\* search, if path not found or the path to food is too far (cost more than 20 while the score of food is only 20), remove the current food from ListFood.
* Check if Pacman’s current position is near any ghost or not, if there is a ghost nearby (calculated distance by Manhattan distance, distance <= 2), Pacman will find the possible direction to evade the ghost (use runnnn function), if there is no possible move, Pacman stays at current position. In lv3, ghosts move with function ghost\_random\_move in ghost class (each ghost just moves around its initial location). In lv4, ghosts chase Pacman by function chase (each ghost will seek and kill Pacman by one of 2 chase modes).
* Pacman moves one step (follow path) and decrease score by 1 (each move cost 1 score), check if the new current position has food or not, if there is a food in current position, increase current score by 20 and delete the food Pacman ate from the map.



**Function nearest\_food\_tactic2():**

* Iterate each food object in **ListFood**.
* While loop for **ListFood** (While len(**ListFood**) > 0):
* Sort food objects in **ListFood** by **sort\_Food()** function.
* Create a copy of **ListFood** (**ListFood\_currentState**) to temporarily remove some food that Pacman can’t reach with current position (the food is blocked by ghosts) out of copy of **ListFood**, these removed food objects in the copy list is still in **ListFood** so in the next loop we can recheck if it is possible to reach them or not.
* Calculate path from Pacman’s current position to **ListFood\_currentState[0]** (food that has nearest distance calculated by Manhattan distance between pacman’s position and food’s position) using **A\* search**. While the path is not found, remove first food in **ListFood\_currentState** and calculate path of first food of new **ListFood\_currentState** until the path is found or there is no food in list.
* If there are at least 4 food objects in the list, calculate path to each of 3 food after, assign shortest path to **path**.
* While loop to move follow the path:
* Check if Pacman’s current position is near any ghost or not, if there is a ghost nearby (calculated distance by Manhattan distance, distance <= 2), Pacman will find the possible direction to evade the ghost (use **runnnn** function), if there is no possible move, Pacman stays at current position. In lv3, ghosts move with function **ghost\_random\_move** in ghost class (each ghost just moves around its initial location). In lv4, ghosts chase Pacman by function **chase** (each ghost will seek and kill Pacman by one of 2 chase modes). Break the while loop of path if Pacman evaded the ghosts, move to next loop and calculate new path.
* Pacman moves one step (follow **path**) and decrease score by 1 (each move cost 1 score), check if the new current position has food or not, if there is a food in current position, increase current score by 20 and delete the food Pacman ate from the map.

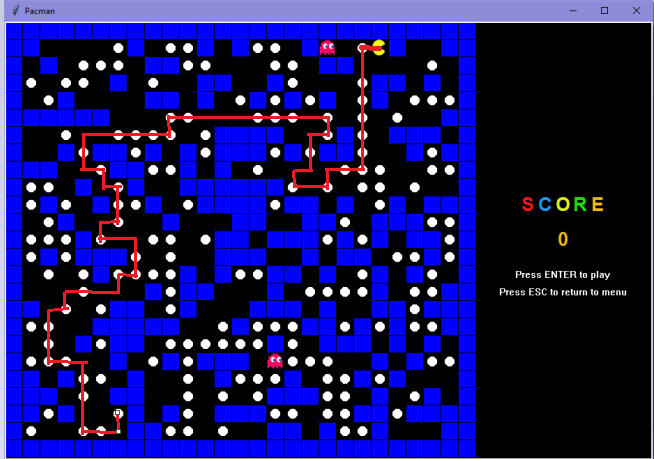


**Function highest\_cost\_tactic():**

* Use **reverse\_A\_Star** algorithm for each food in **ListFood** to create an **ListCost**
* Go to food which has highest score in its path (visit the most food in the path).
* When pacman reach the goal, use **reverse\_A\_Star** again to find from the rest of ListFood.
* **Disadvantage**:

+ We have to use **reverse\_A\_Star** for all food in ListFood, run pacman then update again and again. It cost very long time for searching.

+ This isn’t an optimal tactic because some food near the path can’t be eaten when pacman running. And then, when pacman reach goal, the distance from it to pacman is further, and reverse\_A\_Star will search it again.



Path from the next pacman position is too far.

Near food cannot be eaten…

**Function blind\_check\_tactic():**

* Use **Stack** to iterate like DFS tree searching
* Find **unvisited tile** in **ListAdjacency[pacman\_index]** and that tile **doesn’t near monster** then visit it, push it into **Stack.**
* If **check\_ghost\_1** found that tile near monster (tile in ListAdjacncy[monster\_index] or manhattan distance between pacman and monster is 2), pacman wait a step, watch monster move.
* After monster\_move, **check\_ghost\_2** again, if that tile still near monster, don’t move to that tile, choose another one.
* If we have checked all tiles in **ListAdjacency[pacman\_index],** move back one step at the top (last) of Stack.

# Explaning code: Searching\_Algorithm.py

**Function BFS (adjacency\_list, begin, food\_position):**

* Create the parent array the size of the passed list.
* Check at start is Food or not, if so, then return results.
* Create the queue for the starting value, node at the position taken from the queue. Check to see if it is in the list expanded or not, if the node at location is Food, retrieve it and return the function values.

**Function DFS (adjacency\_list, begin, food\_position):**

* Create the parent array the size of the passed list.
* Check at start is Food or not, if so, then return results.
* Create a stack array and put the initial value in. The node at position will be populated from the stack and checked to see if it is in expand nodes. If not, will update the stack and parent again.
* If it is at that position, Food will trace back and return the values of the function.

**Function DLS (adjacency\_list, food\_pos, explored, parent, current\_path, depth):**

* Implement recursive depth-limited search.
* Base conditions: if the last node in the current path is food\_pos, return True. If the depth is 0 (cut off) return False.
* Iterate adjacency nodes of last node in current path: If the current adjacency node is not in current path, set the parent for adjacency node (parent[adjacency\_node] = last\_node), append current adjacency node to the current path and list of explored nodes.
* Call the DLS function with depth = depth – 1 to explore the current path until cut off or the goal is found, store the result in result. If the goal is found (result == True) return true. Else pop the last node in current path.

**Function IDS (adjacency\_list, current\_position, food\_position, max\_depth):**

* Create 2 lists: explored\_ns and path\_fd to store list of explored nodes and list of nodes on the path found.
* If the current\_position is the same node as food\_position: return 0, explored\_ns, path\_fd.
* Iterate the depth from 0 to max\_depth – 1.
* For each depth:
* Create parent list for each node with default value = -1, create explored list to store list of explored nodes in the current depth, current path list to store the path. Call function DLS with arguments: adjacency\_list, food\_position, explored, parent, current\_path, depth, the return value of DLS function is stored in result.
* Append explored to explored\_ns after the call of function DLS. If DLS function return True (result == True), backtrack the current path to find the path and store path in path\_fd, calculate length of explored nodes and store in esc\_time. Return esc\_time, explored\_ns, path\_fd
* Return “”, “”, “” if the path is not found

**Function get\_manhattan\_heuristic (current\_position, food\_position, maze\_size):**

* Get position and Food position.
* Then calculate the distance between two positions and return the result.

**Function A\_Star (adjacency\_list , current\_position, food\_position, maze\_size):**

* Initialize three arrays: explored, frontier, path
* Initialize the parent array to -1.
* Each node pops out with 2 values, its cost and index. We calculate the fcost and the hcost and then we have the gcost.
* Check if that Food location has the same node value.
* If not, check if that node has been expanded, if not, then enter and update the frontier and parent again.

# References