Viet Nam National University, Ho Chi Minh City

University Of Science

Faculty Of Information Technology

Project 1

Search In Pacman

Trần Đại Chí	18127070
Thái Nhật Tân	18127204
Thái Hoàng Long	18127140
Trần Minh Quang	18127192

Lecture

Dr. Nguyễn Hải Minh

Dr. Nguyễn Ngọc Thảo

Ms. Lê Ngọc Thành

Course: Introduction to artificial intelligence

1) **Detail Information**:

Name	**Student ID**
Trần Đại Chí	18127070
Thái Nhật Tân	18127204
Thái Hoàng Long	18127140
Trần Minh Quang	18127192

2) **Assignment Plan:**

Name	**Job**	**Evaluation**
Trần Đại Chí	Level 3 and 4, graphic game	100%
Thái Nhật Tân	Level 3 and 4, graphic game	100%
Thái Hoàng Long	Level 1 and 2, graphic game	100%
Trần Minh Quang	Level 1 and 2, graphic game	100%

3) **Environment to compile and run the program**: pycharm

4) **Estimating the degree of completion for each requirement**: 100% completed

5) **Input/Output**:

- Input:

+ 5 maps of game correspond to 5 files: map1.txt, map2.txt, map3.txt, map4.txt and map5.txt. File map1.txt is designed for level 1, map2.txt is designed for level 2 and map3.txt, map4.txt, map5.txt are designed for level 3 and level 4.

+ Structure of input file: first line contains two integers NxM which is the size of map, N next lines represent NxM map matrix with value 0: empty path, value 1: wall, value 2: food, value 3: monster and value 4: pacman. We put value 4 directly in empty path of map's file, so we can check the intelligence of pacman to solve the problem in any positions of map if it's an empty path.

- Structure of output file: we have use two graphic libraries (turtle graphic for level 1 and 2, pygame graphic for level 3 and 4) for displaying results . We also print result to console include final state result of pacman (win or lose), the length of the discovered paths, point of pacman and time to finished and a folder path_discovered contains text files for these informations.

6) **Structure file and algorithm for both level 1 and level 2**:

+ level1.py: handle for level 1 of pacman, readFile(myFile): read input file of map game, in order to detect pacman collision with monster or destroy food after eating, we use euclidean distance between two points a, b to check that situation. setup_maze(adjacent_matrix): compute x_coordinate, y_coordinate follow the formula x_coordinate = -288 + (y*24), y_coordinate = 288 – (x*24) (each cell of turtle graphic has pixel = 24, our map designed for level 1 is 25x25, so width = row * 24 = 25 * 24 = 600, height = column * 24 = 25 * 24 = 600, if column or row is even: (width or height) / 2, else if column or row is odd: (width or height – 24) / 2, our row and column is 25 which is an odd number, so its row = column = (600 – 24) / 2 = 288, then our map will have four coordinate correspond: topLeft(-288, 288), topRight(288, 288), bottomLeft(-288, -288), bottomRight(288, -288)) and use x_coordinate, y_coordinate to check characterMap at that position is 0: empty path, 1: wall, 2: food or 4: pacman to add. getCoordinate(path): we also use formula to calculate x_coordinate, y_coordinate as above, then use function BackTrack to get path after searching by a* for pacman going to food. eat(): check if pacman is collision with food, then print the point of the pacman, mainLevel1(): get current position of pacman and use function getCoordinate(BackTrack) to get path for pacman going to food, then handle for movement of pacman. each step pacman move, its point will decrease by 1 and if pacman successfully eat food, its point will increase by 20).

+ level2.py: everything is the same with level1.py, we just create a class Monster and add monsters (if character = 3) in function setup_maze(adjacent_matrix) and because pacman just stay at a fixed position, we see them as a wall.

+ searchLevel1_2: use a* for both level 1 and level 2, set_not_visited(adjacent_matrix): default we'll set all row and column of map as not visited (0: not visited, 1: visited), path_visited(adjacent_matrix): make all vertices as not visited (path = [False] * -1). And about a*: we proceed to move go down, go top, go right, go left and check next step condition if it is not wall (for level 1) or not monster (for level 2). After each step moving we add to the queue. The next time we'll check to see which in the queue should choose that position to proceed through at minimum cost, which is equal to the sum of the heuristic length and its actual length. After each taking out, we mark it as over and continue to find its new path. If the new

path is not in the queue and has not been passed yet, we add it to the queue and save the new location by assigning its parent node to the path, which is very useful for post-encounter tracing. If the new road is in the queue and has not been passed yet and the cost of the new road is less than the cost of the old road, we remove the old position in the queue and the old road, and add to the queue. the new position enters the queue and saves the new location by assigning its parent node across the path. If pacman encounters food, we stop and use the backtrack function to track from pacman's initializer counter. Backtrack(path, x_pacman, y_pacman, x_food, y_food): get path from source (current position of pacman) to destination (food) for path returns of each node relative to its parent, example: we have a graph with 6 nodes (1, 2, 3, 4, 5, 6), assume that destination of our graph is 6 and its parent is 4, parent of 4 is 2 and parent of 2 is 1, so backtrack from down to top we have result: 6->4->2->1, then we'll reverse this result to have path returns: 1->2->4->6.

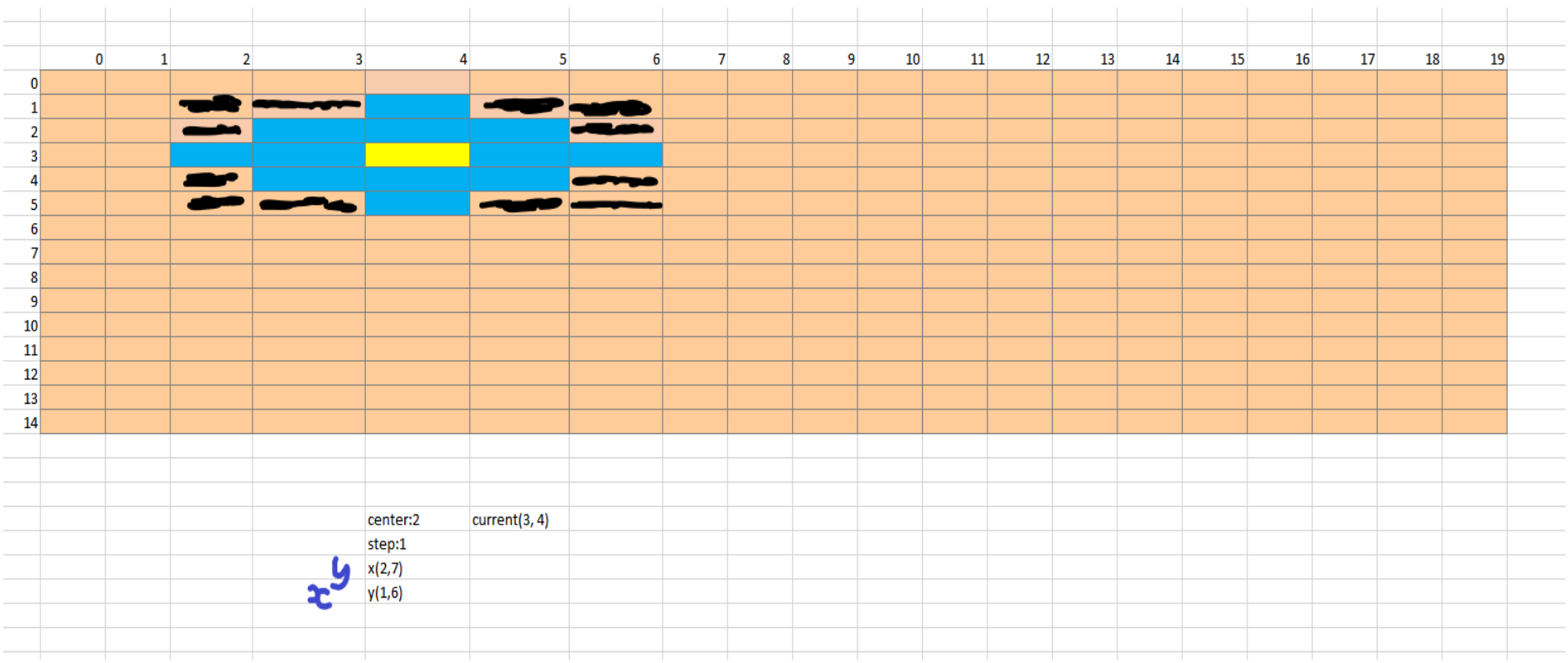
7) **Structure file and algorithm for both level 3 and level 4**:

+ settings.py: default settings for pygame (screen resolution, color and font).

+ util.py is the support file for init character of map game, check win/lose of pacman, some functions such as stack, queue, priorityqueue for implementing searching (if necessary) and 4 heuristic distance that board game usually use are: manhattan distance, euclidean distance, chebyshev distance, octile distance. Both level 3 and level 4 we just use manhattan distance for main optimize heuristic of game, but for testing purpose, we can change this heuristic to euclidean, chebyshev or octile to check which heuristic is the best for our game.

+ setup_map.py is the initial map for our game. In this file, we have declare function update(self, current_location, new_location, characterMap) that update new location of agents (pacman, monsters) by removing characterMap (value 3 for monsters and value 4 for pacman) from current location then we copy information of new location and add characterMap to that new location. We also have start_draw(self) function for drawing theme, map, walls, monsters, pacman, foods and some support functions such as: getResolution(self): get default resolution of map game, getFood(self, foods): add foods to map, getWall(self, walls): add walls to map, getAgent(self, agents): add agents (pacman, monsters) to map, destroyFood(self, food_index): remove food from map after eating.

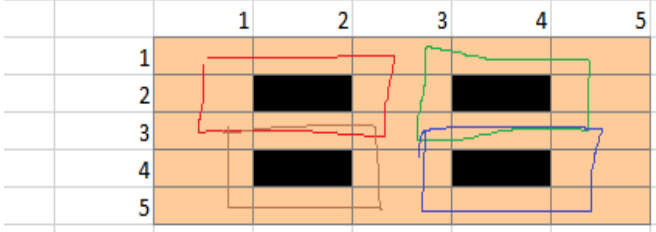
+ ghostAgents.py is the main file for handling monsters. getDistribution(self, direction): remove an action from list valid actions of monster (this help monster always move to find pacman instead of stopping at a fixed position), checkValidGhostPosition(self): check if monster move out of bound of map, random_move(self, move): random movement of monsters in list valid actions and try to avoid monsters from stopping action, update(self, new_location): update new location of monsters in map game from its predecessor (current location), scan_pacman(self, mapGame): this function help monster find pacman if pacman in 8 tiles x 3



As we see in this picture above, assume yellow cell is current position of pacman or monster, so from level 3, both pacman and monster will see each others in this area (nearest three steps). Thereforce, we'll declare a diameter (a fixed number) = 5 => radius = 5 / 2 = 2. In this picture, current position of agent is (3, 4), we use first loop for row from (self.location[0] – center, self.location[0] + center + 1) with step = 1 will get x_coordinate = (3 – 2, 3 + 2 + 1) = (1, 6). Similarly, we have a second loop for column from (self.location[1] – center, self.location[1] + center + 1) with step = 1 will get y_coordinate = (4 – 2, 4 + 2 + 1) = (2, 7). Both 1 to 6 and 2 to 7 take 5 steps as blue cell of picture, calculate from yellow cell, we'll get

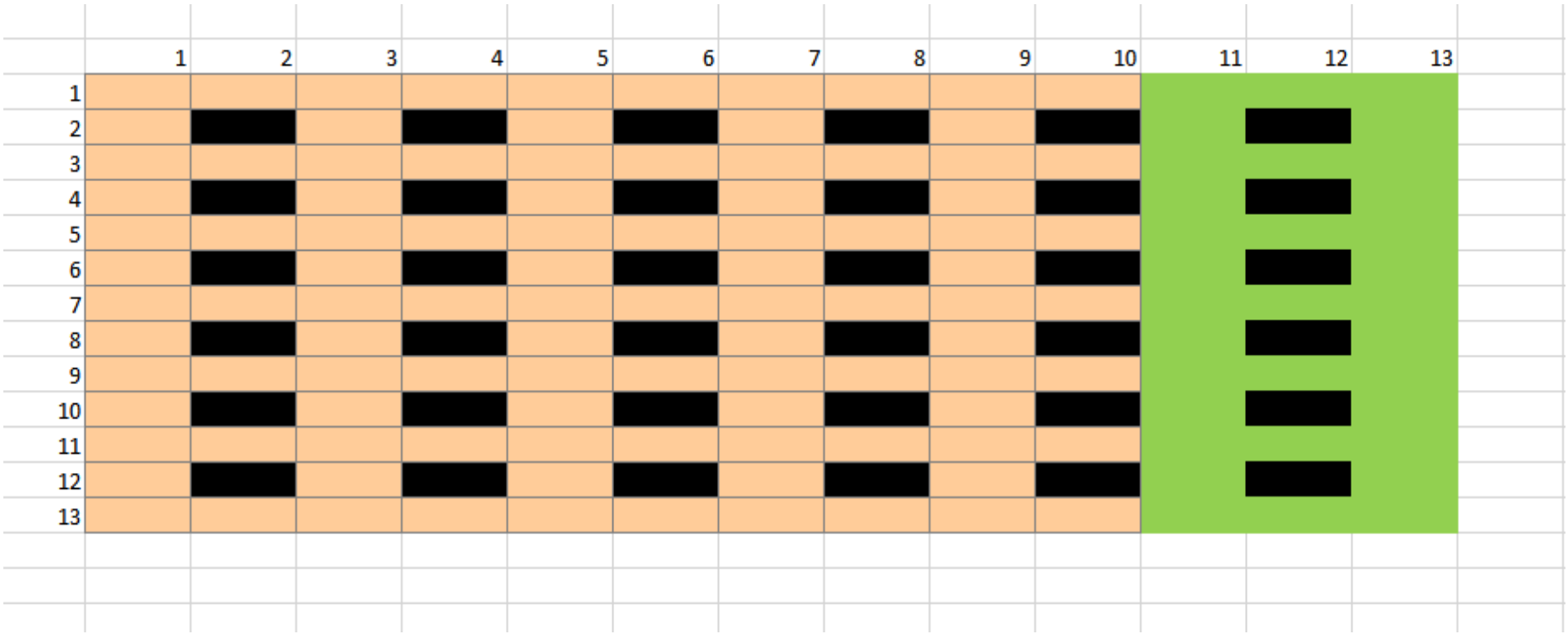
nearest three steps and if pacman appear in this area, monsters'll have pacman position. seek_pacman(self, mapGame): get pacman position from function scan_pacman(self, mapGame) and add to list bestMove actions need to move to that position. move(self, mapGame): level of game is 3: pacman move randomly, level of game is 4: use function scan_pacman(self, mapGame) to take possible actions from list bestMove then check its validity, then move to that position to kill pacman, each step moving, we also update new location of monsters from map game and update new location of itself from its predecessor (current location).

+ pacman.py is the main file for handling pacman. update(self, new_location): update new location of pacman in map game from its predecessor (current location). getDistribution(self, direction): remove an action from list valid actions of pacman (this function use to remove actions which can lead pacman to monsters). avoid_monster(self, monsters): use function getDistribution(self, direction) for pacman to avoid monsters, example: if monster is in left direction, pacman is continue moving left and detect that left direction has monsters, so pacman will remove this action immediately. checkValidPacmanPosition(self): check if pacman move out of bound of map. calculate_manhattan_distance(self, mapGame): we also use formula for 8 tiles x 3 as function scan_pacman(self, mapGame) declared in ghostAgents.py, in addition, in both level 3 and 4 pacman don't have full view of map, so we will create self.map for class PacmanRules which use map resolution (function getResolution(self) in setup_map.py), this will help us copy information from 8 tiles x 3 of map game to self.map (a map create specifically for pacman to scan around) for scanning in that area if pacman detect new_food which not exits in food_position (list store food positions that pacman detect during scanning), then we'll add positions of new_food to list tiles (list store new food positions pacman detect during scanning) and decrease food_count (variable store total food of map, if food_count = 0, it means pacman had successfully scanned all food positions of the map), because pacman just see in nearest three steps, so manhattan distance from pacman location to goal (new_food) must be less than 3. scan_monster_stay_near(self, mapGame): use function calculate_manhatan_distance(self, mapGame) to dectect if in 8 tiles x 3 that monsters appear, add these positions to list monster_location which help pacman avoid collision monsters in this area. Because pacman don't have full view of map, so firstly, we have function create_important_positions(self) for pacman go to these positions to scan around area of that important position to know whether that area has foods, walls, monsters.

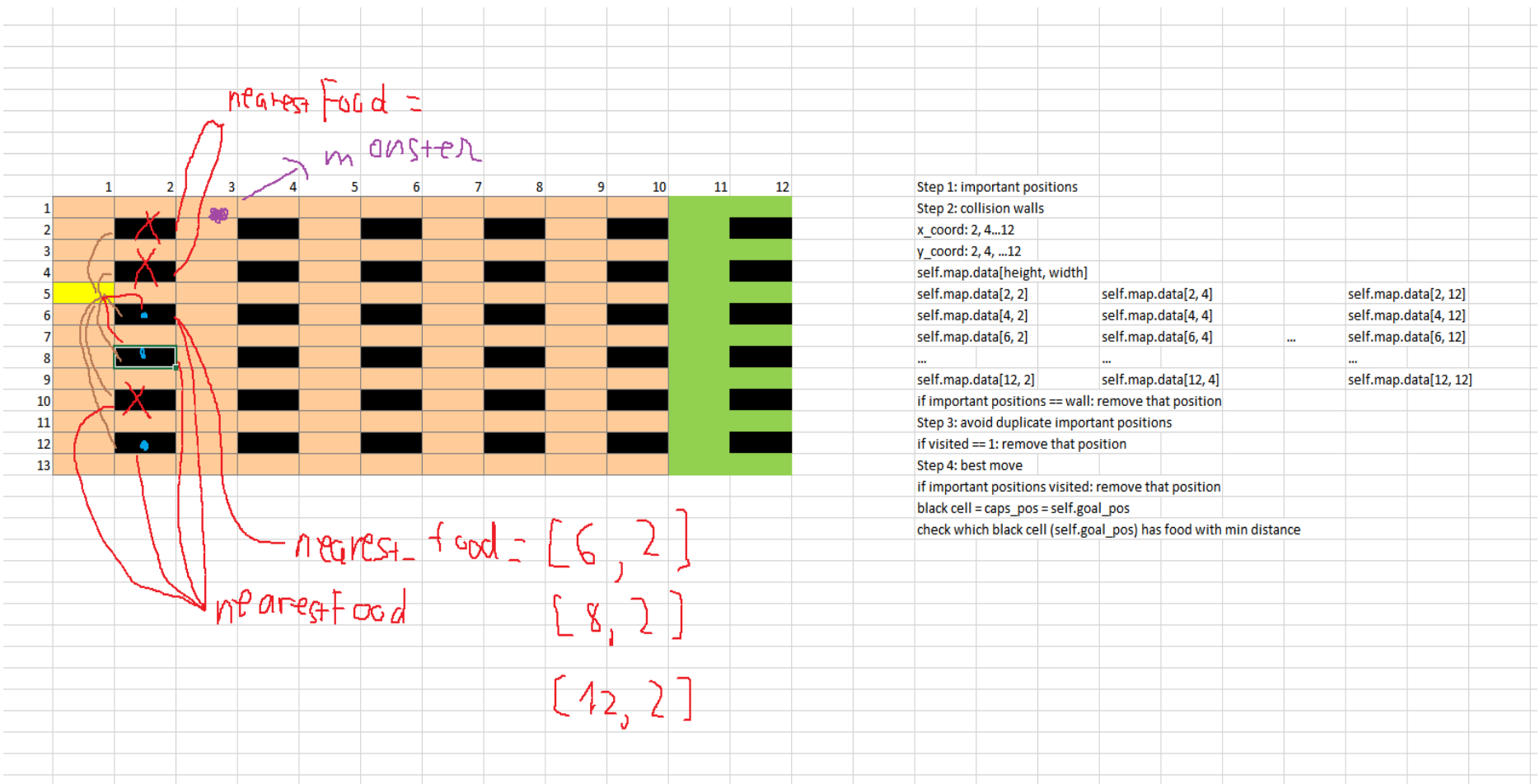


center = 2	x=0<5	y=0<5	x=2<5	y=0<5
x, y = 0	x=0+2=2	y=0+2=2	x=2+2=4	y=0+2=2
w=5,h=5	(2, 2)		(4, 2)	
	x=0<5	y=2<5	x=2<5	y=2<5
	x=0+2=2	y=2+2=4	x=2+2=4	y=2+2=2
	(2, 4)		(4, 4)	

Assume that our map game is 5x5 as picture above, we reuse radius = diameter / 2 = 5 / 2 = 2, take x = 0, y = 0 as coordinate for row and column, width of map = row of map = 5, while x < width of map (0 < 5), x = x + radius = 0 + 2 = 2, while y < height of map (0 < 5), y = y + radius = 0 + 2 = 2 and continue doing with 3 positions left, we'll get 4 important positions for map are: (2, 2), (2, 4), (4, 2), (4, 4). Because pacman don't know about its world, we'll help pacman learn to know about its world by creating these positions as goal positions for pacman moving to these positions to scanning around for foods, monsters, walls. In case our map just 13x10, not a square matrix like 5x5, 10x10, 15x15…, it can't create a map with fully important positions, so when read size of map, we have changed mapGame from (mapSize[0], mapsize[1]) become (mapsize[0], mapSize[1] + abs(mapsize[0] – mapSize[1]) = (13, 10 + abs(13 – 10)) = (13, 13). Now, our map become 13x13 and we can fully create important positions as picture bellow



Next, we have function check_important_positions_collision_walls(self) to check if there is any important positions collision with wall, we will remove it from goal positions to pacman avoid going to these position. In case if pacman reach fully these important positions but still exist food pacman didn't detect during scanning, check these important positions collision wall that we have removed before. eat_food(self): decrease total foods map provided if pacman eat a food, if food less than 0, it means pacman ate all foods and won the game. closest_capsule(self, caps_pos): get current position of pacman, then we use manhattan distance to get min distance from current position of pacman to new_food which pacman detected during scanning and add this position (position contain food with min distance) to list nearestCapsule to help pacman make best move for maximizing food pacman could eat. remove_duplicate_important_positions(self): because we want pacman to scan full view of map as much as possible, so each goal position (important positions) pacman had reached, we'll mark it as visited and remove that position from list goal_position (list contains important positions) to make pacman avoid going these position too many times if pacman have other possible directions to move. search_for_best_move(self): if food_count > 0 (still have foods pacman didn't detect during scanning), we'll use function check_important_positions_collision_walls(self) because pacman didn't reach to these positions previously, so maybe these positions (positions collision with wall) have food around, then use function remove_duplicate_important_positions(self) because we want pacman to maximizing its food and point, so pacman should avoid go to these positions (positions not collision with wall) too many times.



After that, we declare a variable nearestFood for getting min distance from pacman position to goal_position (important positions) and if food_count (food pacman scanned and detected in map) = 0, it means pacman scanned full view of map

and had fully food positions, so nearestFood will an empty list because we don't need check these positions any more. Next, we declare a variable nearest_food for getting min distance from pacman position to food_position (coordinate of food pacman had while scanning) and if nearest_food is not empty, we'll add position of nearest_food to list nearestCapsule to help pacman know which food has min distance to eat first and then, come back to nearestFood to continue scan around for foods until there are no more foods. We also have a list bestMove that contains actions to help pacman move to all food positions with min distance from list nearestCapsule. random_move(self, move): make pacman move randomly but avoid pacman stopping at a fixed position if pacman have other possible directinos to move and we also limit pacman moving a cell too many times because we are trying to make pacman scan full view of map as much as possible. scan_positions_around_important_positions(self): mark current position of pacman as visited and if pacman didn't fully scan all view of map, remove that position if it's important positions, and pacman also destroy food from map if position pacman moved has food. fullViewOfMap(self, mapGame): support function for level 1 and 2 to provide all food locations for pacman. move(self, mapGame): we priority to escape monsters first, then use function search_for_best_move(self) to find best moves to foods and check validity of these actions if it exists in list self.actions of pacman and move pacman randomly.

+ mainLevel3_4: main game for handling level 3 and 4, printResult(pacman): print final state result of pacman (win or lose), valid_actions(agent, mapGame, level): get valid actions for agents (pacman, monsters) to remove actions when facing walls or avoid going out of bound, draw_text(words, screen, position, size, color, fontName): draw text graphic when game finished, check_current_state(pacman, mapGame, screen): check current state of pacman is win (ate fully foods) or lose (collision with monsters), mainLevel3_4(): handle for reading file, drawing map, adding agents (pacman, monsters), foods, walls to map and writing all results to text file (check in folder path_discovered).

8) **Some reflection and comments**:

+ In level 1 and 2, if our map is complex, we should use search strategies a* for optimizing shortest path. But each moving steps, pacman's point will be decrease by 1 (cost is the same), so in normal map, bfs is also good enough for pacman searching shortest path to foods (in case we don't care so much about cost).

+ In level 4, we should also update pacman to handle some difficult situations such as if we have a food in a corner of map, pacman is chased by many monsters and when that food (food in corner of map) is in min distance, pacman detect and try to eat that food which can lead to a lose game (because that corner of map just have an entrance which can make pacman to be stuck). Pacman still have a little bit problem is that: if the corner just has one cell (I means it just have an entrance as situation I mentioned above) to move inside and pacman scanned and ate all food inside of that corner but current position of any monsters is in entrance of that corner, pacman detect and try to avoid monster first but all cells inside of corner pacman had fully scanned but we are trying to make pacman scan full view of map and avoid duplicate paths as much as possible, so it can lead pacman moves as an endless loop in that corner (just happens occasionally). Therefore, in level 4 we need update and optimize our algorithm to make pacman more intelligent for handling some difficult suitations like that.

+ We also have a different problem with map is that: because our map has multi agents and each step moving of any agents(pacman, monsters), our map has to update again new position of agents which make our map draw graphic of that agent again, so graphic image of agents sometimes disappear and appear again, it don't affect so many to our game but we should make some graphic optimization for better display quality.

9) **Compile and run**: open main.py and run it (just install the pygame, no more packages are required). When we run the program, we'll have 3 options: easy challenge (use to run level 1 of pacman), medium challenge (use to run level 2 of pacman), hard challenge (use to run level 3 and level 4 of pacman). After running the program, you can see all the necessary results of that level in console or you can quickly find the result of four levels I've made for you before in folder path_discovered.

Graphic references:

https://www.youtube.com/watch?v=-0q_miviUDs (graphic references for level 1 and level 2)

https://www.youtube.com/watch?v=ZdwRcte00oU (graphic references for level 3 and level 4)