Homework #1

Student ID : 2019117865

Name : 김나형

# Question 1 (3 points) : Finding a Fixed Food Dot using Depth First Search

a. Source Code :

def depthFirstSearch(problem):

# DFS uses stack

stack = util.Stack()

# initial state into stack

stack.push(problem.getStartState())

# DFS

visited = []

visited.append(problem.getStartState())

paths = {}

paths[problem.getStartState()] = []

# DFS actions

while (not stack.isEmpty()):

current = stack.pop()

# visit check

visited.append(current)

# reaches goal -> return path

if problem.isGoalState(current):

return paths[current]

# (successor, action, stepCost)

neighbors = problem.getSuccessors(current)

for successor in neighbors:

# not visited

if successor[0] not in visited:

# successor[0] = successor / successor[1] = action

# save path

paths[successor[0]] = paths[current] + [successor[1]]

# DFS

stack.push(successor[0])

b. Description of DFS Implementation

DFS is implemented using a recursive function, or stack. Therefore, for this project, I used stacks to implement using the given util from the project. After declaring Stack, find the start state and put it in the stack. Then, declare a visited array to check whether to visit or not, and then put a start state/point in the visited array. This is to continue DFS Search thinking that the first starting point has already been visited. It then declares a path dictionary that stores the path that the real Pacman will take. The reason for declaring Paths as a dictionary rather than an array is to use the function getSuccessors to find adjacent nodes of a node, which returns three values (successor, action, stepCost) and uses the dictionary to use these various values.

Proceed with DFS using the While statement. The framework of the existing DFS code has been used and ends when the Goal State is reached. Otherwise, neighboring nodes of the current node are obtained using the getSuccessors function. If there is a neighbor, all of the neighboring nodes are searched using the for statement. If the neighboring node is not visited, it is a path that can be moved, so it is a path that can be moved forward, so it is put in the stack so that it continues to find the next path. And the path is stored in the path, and the corresponding successor is stored in the stack to continue with DFS to find the way to the goal.

c. Experimental Results

- tinyMaze

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

- mediumMaze

Path found with total cost of 130 in 0.0 seconds

Search nodes expanded: 146

Pacman emerges victorious! Score: 380

Average Score: 380.0

Scores: 380.0

Win Rate: 1/1 (1.00)

Record: Win

- bigMaze

Path found with total cost of 210 in 0.0 seconds

Search nodes expanded: 390

Pacman emerges victorious! Score: 300

Average Score: 300.0

Scores: 300.0

Win Rate: 1/1 (1.00)

Record: Win

# Question 2 (7 points): A\* search

a. A\* Search Implementation

- Source code

search.py

def aStarSearch(problem, heuristic=nullHeuristic):

visited = []

# longest distance as possible

pq = util.PriorityQueue()

paths = []

pq.push((problem.getStartState(), paths), 0)

# A\* Search

while not pq.isEmpty():

next\_node, paths = pq.pop()

# reaches goal

if problem.isGoalState(next\_node):

return paths

# not searched node

if next\_node not in visited:

successors = problem.getSuccessors(next\_node)

for child in successors:

if child[0] not in visited:

# A\* Search

# heuristic = distance method

cost = problem.getCostOfActions(paths + [child[1]]) + heuristic(child[0], problem)

pq.push((child[0], paths + [child[1]]), cost)

visited.append(next\_node)

searchAgents.py

def newHeuristic(position, problem, info={}):

xy1 = position

xy2 = problem.goal

return max(abs(xy1[0] - xy2[0]), abs(xy1[1] - xy2[1]), max(abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1]), ( (xy1[0] - xy2[0]) \*\* 2 + (xy1[1] - xy2[1]) \*\* 2 ) \*\* 0.5))

- Description of Implementation

The basic principle is the same as DFS, but DFS uses a stack, whereas A\* Search uses Priority Queue because it needs to find a case with the largest expected value from the current position to the position of goal. Max Heap can be easily implemented using the Priority Queue. To sort this Max Heap, you need to get Cost. To find the Cost, we need to find the largest distance possible for the heuristic that should be admissible. Like DFS, statements are used to navigate all neighboring nodes, and heuristics are used to find and add the cost from the current position to the goal. Then, put this cost into the Priority Queue and do a Heap Sort to proceed with the A\* search.

The difference in performance according to Heuristics depends on how to assume the distance of each vector point. If Manhattan Distance is the sum of the differences between the absolute values on the coordinates of the distance between the two points, Euclidean Distance calculates the linear distance between the two points.

지도이(가) 표시된 사진

자동 생성된 설명

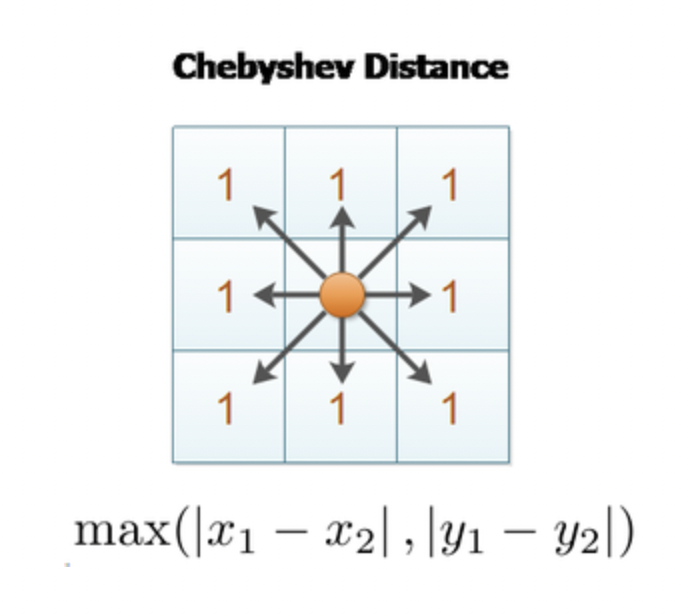
*Manhattan vs Euclidean*

Referring to the above picture, Manhattan Distance is bound to have a greater value than Euclidean Distance. Since heuristics should be admissible, it is recommended to obtain the largest value when measuring the cost (distance measurement). Therefore, in this case, using Manhattan performs better than using Euclidean. It searches for much fewer nodes, and because of its high cost (Distance) value, it is even faster and more complete when looking for directions.

b. New heuristic function

- Method (description of the proposed function)

The first new Heuristic function I tried to make was implemented by referring to Chebyshev Distance.



If Manhattan Distance is the sum of the differences between the absolute values on the coordinates of the distance between the two points, Euclidean Distance calculates the linear distance between the two points. On the other hand, Chebyshev Distance differs in processing all eight adjacent cells from one point at the same distance. Since the movement of the Pacman is done in units of compartments while thinking about devising heuristics, I thought that implementing heuristics using Chebyshev Distance would be able to calculate the cost in consideration of the characteristics of the game.

# Chebyshev distance

def newHeuristic(position, problem, info={}):

xy1 = position

xy2 = problem.goal

return max(abs(xy1[0] - xy2[0]), abs(xy1[1] - xy2[1]))

However, when implemented in this way, the performance was worse during the test because the distance value was much smaller than Euclidean, which is smaller than Manhattan, was obtained. A good heuristic should approximate the longest distance possible from position to the goal, without going over it. If heuristic1 returns a greater or equal value than heuristic2 for every input (while still being valid), then heuristic1 is better. In your case, Manhattan distance is better than Euclidean distance because the value it returns is never smaller than Euclidean distance and is sometimes larger. (Unless you can move diagonally in the maze, in which case Manhattan distance is an invalid heuristic and Euclidean distance might be as well.)

Therefore, I came up with the idea to get the largest distance value possible by utilizing the max used when implementing Chebyshev. If heuristic1 returns a larger value for some inputs, and heuristic2 returns a larger value for others (and both are valid), then heuristic3(x) = max(heuristic1(x), heuristic2(x)) is better than either of them. Therefore, Manhattan, Euclidean, and Chebyshev sets are combined to implement a new function that uses the larger of the two distance values.

def newHeuristic(position, problem, info={}):

xy1 = position

xy2 = problem.goal

return max(abs(xy1[0] - xy2[0]), abs(xy1[1] - xy2[1]), max(abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1]), ( (xy1[0] - xy2[0]) \*\* 2 + (xy1[1] - xy2[1]) \*\* 2 ) \*\* 0.5))

c. Evaluation of the effectiveness of the heuristic functions

- Experimental Results (Manhattan vs Euclidean vs new function)

1. Tiny Maze

1-1) Manhattan

Search nodes expanded: 14

Pacman emerges victorious! Score: 502

Average Score: 502.0

Scores: 502.0

Win Rate: 1/1 (1.00)

Record: Win

1-2) Euclidean

Search nodes expanded: 13

Pacman emerges victorious! Score: 502

Average Score: 502.0

Scores: 502.0

Win Rate: 1/1 (1.00)

Record: Win

1-3) New Function

Path found with total cost of 8 in 0.0 seconds

Search nodes expanded: 14

Pacman emerges victorious! Score: 502

Average Score: 502.0

Scores: 502.0

Win Rate: 1/1 (1.00)

Record: Win

2. Medium Maze

2-1) Manhattan

Path found with total cost of 68 in 0.0 seconds

Search nodes expanded: 221

Pacman emerges victorious! Score: 442

Average Score: 442.0

Scores: 442.0

Win Rate: 1/1 (1.00)

Record: Win

2-2) Euclidean

Path found with total cost of 68 in 0.0 seconds

Search nodes expanded: 226

Pacman emerges victorious! Score: 442

Average Score: 442.0

Scores: 442.0

Win Rate: 1/1 (1.00)

Record: Win

2-3) New Function

Path found with total cost of 68 in 0.0 seconds

Search nodes expanded: 221

Pacman emerges victorious! Score: 442

Average Score: 442.0

Scores: 442.0

Win Rate: 1/1 (1.00)

Record: Win

3. Big Maze

3-1) Manhattan

Path found with total cost of 210 in 0.1 seconds

Search nodes expanded: 549

Pacman emerges victorious! Score: 300

Average Score: 300.0

Scores: 300.0

Win Rate: 1/1 (1.00)

Record: Win

3-2) Euclidean

Path found with total cost of 210 in 0.1 seconds

Search nodes expanded: 557

Pacman emerges victorious! Score: 300

Average Score: 300.0

Scores: 300.0

Win Rate: 1/1 (1.00)

Record: Win

3-3) New Function

Path found with total cost of 210 in 0.1 seconds

Search nodes expanded: 549

Pacman emerges victorious! Score: 300

Average Score: 300.0

Scores: 300.0

Win Rate: 1/1 (1.00)

Record: Win

- Discussion (which one is better, reasons, limitations, ...)

Among Manhattan Heuristic, Euclidean Heuristic, and new function, it can be said that new function is good. Because Manhattan Heuristics are mostly greater than or equal to Euclidean Heuristics. Therefore, in most cases, Manhattan Heuristics performs better than the Heuristics to be used for A\* Search. Because in the case of A\* Search, it was confirmed that Manhattan, which obtains the greatest distance from the current position of Pacman to the goal, has the best performance for Pacman who moves only up, down, left, and right. However, there may be exceptions, because in some cases the Euclidean Distance, or Chebyshev Distance, both Manhattan and Euclidean Distance, and Chebyshev Distance, the new Function, which takes the greater distance, is the best prediction.

To measure the performance of Heuristics, it is recommended to explore the Search Node less and find the fastest way to reach the goal as the best route. The measurements in Tiny Maze show that the score is equal to 502, so all of them have the same shortest path, and in this case, it can be seen that Manhattan with 14 nodes expanded and Euclidean with 13 nodes expanded perform better than New Function. But as the maze grows, it becomes the opposite. It can be seen that in Medium Maze they are similar, in Big Maze, there are 549 Manhattan Heuristics and New Function, while in Euclidean there are 557, which is gradually deteriorating.

However, Manhattan Heuristics' Limitation performs best when moving up, down, left, and right, in four directions, like Pacman. However, it is said that it is advantageous to use Chebyshev when moving diagonally as well as up, down, left, and right, and Euclidean Distance when the restriction on the direction of movement disappears.

* For **4-directions**, use Manhattan distance (L1)
* For **8-directions**, use Chebyshev distance ([L-Infinity](https://en.wikipedia.org/wiki/Chebyshev_distance))
* For **any direction**, you *can* use Euclidean distance, but an alternative map representation may be better (e.g. using Waypoints)

Therefore, the new function is better than Manhattan and Euclidean because it always gets the largest distance to cover all of these cases.