Assignment 1: A study of multi-agent based environment

IT2018045, Ananya Aparajita Mohanty

V Semester, B.Tech, Department of Studies in Information Technology,

IIIT Allahabad , Prayagraj, India

Abstract--In this paper, we have devised an algorithm which determines the path of 'n' agents from their respective initial coordinates to final goal coordinates in a two dimensional matrix of chosen dimension.

I. INTRODUCTION

Given 'n' number of vehicles which try to reach a target without overlapping with each other anywhere in their traversal is a classic problem. For each vehicle to reach the final destination, it has to not only opt for the shortest possible route but also ensure that its every step is taken towards an unoccupied coordinate. In the field of artificial intelligence, the "vehicles" are the "agents" who should be "rational". A rational agent is the one that chooses an action, at every step, with the intention to maximise one's performance measure [1], which in turn contributes towards making the system intelligent.

II. DEFINITIONS AND CONCEPTS USED

(A) Environment

It is the place on which the agents act/operate upon [1]. In this case, it is the 2D matrix of NxM that contains given agents.

(B) Agent

An agent perceives its environment with the help of sensors and acts using actuators or effectors. [1]

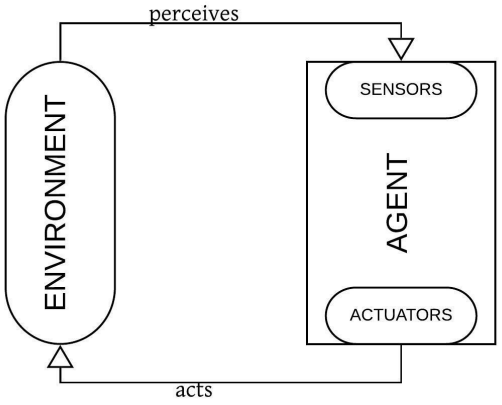


Fig.1 Representation of Agent and Environment Interaction

They can be categorized into four types [1] :

1. Simple reflex agents
2. Model-based reflex agents
3. Goal-based agents
4. Utility-based agents

In accordance with the problem statement, we choose the agents to be goal-based since our prime objective is to move them to the target coordinates. Such agents make decisions based on their current distance from reaching their goal, and their every action focuses to reduce this distance [3].

(C) Multithreading

It is the process by which we execute multiple threads (light-weight subprocess) simultaneously. By using threads, multitasking can be obtained easily since they not only share memory (thus saving considerable memory space) but also function smoothly as context switching is faster and more efficient than multiprocessing [2]. Here, we use multithreading in order to make more than one agent function in the same environment (shared memory) to reach their target destination.

III. DESIGN AND IMPLEMENTATION

We have divided the problem at hand into 4 parts with increasing constraints and have devised a solution for each part. The agent, environment, sensor, stimulator shall alter throughout the solution.
The following are the steps of our solution.

(A) For the first part, we have an nXm grid with no obstacles as an environment and an agent which has a sensor to determine the current position and the goal coordinates are known.
In order to facilitate communication between the environment and the agent, we create a stimulator which invokes the agent until it reaches the goal co-ordinates.

Step 1: Create an object of Environment class and initialize the values of n, m, goalX and goalY.
Step 2: Create an object of Agent class and initialize its x and y coordinates.
Step 3: Invoke the run function of Agent until the coordinates of the agent equals that of the goal, i.e. the agent reaches the destination.
Step 4: Agent determines the next position (whether right or left or up or down) depending on whether the next step is safe (i.e. within the boundaries of the grid) and the goal coordinates. Since there is only one agent, we need not check for the cell to be empty before moving it to the next position.

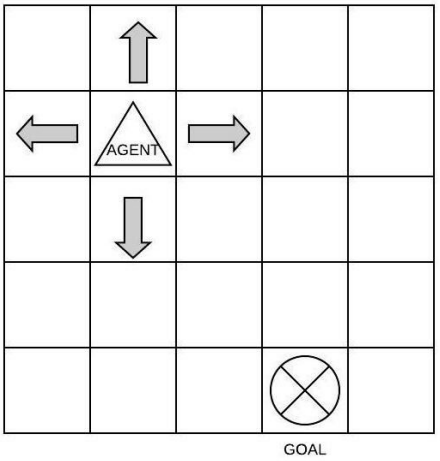


Fig.2 Single agent can move in these four directions to reach the goal

Step 5: The agent then invokes the stimulator with the updated coordinates and the agent marks them for it by calling the environment.
Step 6: When it reaches the target coordinates, the function stops.

Implementation- Class: Environment: variables: n , m (grid size), matrix[][] , x, y (coordinates of agent); functions-> update Position

1: x<- a, y<-b
Class: Agent: variables: x,y, goalX, goalY
Function: nextPosition, input -> none, output->integer
2: diffX <- absolute value(goalX-x),
diffY <- absolute(goalY-y)
3: **if** diffX>diffY **then**
if goalX-x > 0 **then** return 1
else return -1
4: **else if** y<goalY **then** return 2
5: **else if** y>goalY **then** return -2
6: **else** return 0
Function: run , input -> x1,y1, output->void
7: x <- x1, y<- y1, flag <- nextPosition( )
8: **if** flag = 1 or flag = -1 **then** x <- x + flag
9: **else if** flag = 2 or flag = -2 **then** y <- y + flag/2
10: Simulator -> Send position
Class: Simulator: variables: n,m, env (object of class Environment)
Function: sendPosition, input-> x1,y1, output->void
11: env.updatePosition(x1, y1)
Function: MAIN

12: Agent a <- (goalX, goalY, x, y)
13. **while**(a.x != goalX && a.y!= goalY) **do**
14: a.run(x,y);
15: x<-a.x, y <-a.y
16: **end while**

(B) Next, we proceed with the assumption that the goal coordinates are not known by the agent, instead it has a sensor that returns the distance (through function findDistance) between said coordinates and the destination. As a prerequisite, we are using the distance formula in the code for findDistance.
Here, the entire algorithm remains the same except the nextPosition() function which becomes –

Function: nextPosition, input -> none, output->integer
1: int []xi <- {-1, 1}, int []yi <- {-1,1}, minDist <- findDistance(x,y)
2: **for** k<-0 till k<2 **do**
3: distance <- findDistance(x+xi[k])
4: **if** minDist > distance **then**
5: minDist <- distance, temp <- k
6: k<-k+1
7: **end for**
8: **for** k<-0 **to** k<2 **do**
9: distance <- findDistance(y+yi[k])
10: **if** minDist > distance **then**
11: minDist <- distance, temp <- yi[k]*2
12: **end for**
13: return k

(C) For the next part, we implement multi-threading to coordinate 10 or more agents from their initial paths to the common target destination, while ensuring that one doesn't collide with the other and choosing the shortest path as far as possible. We use the same findDistance as we did in part (B). Since there are more than one agent, every such agent views the others as "obstacles" in the environment and thus avoids such a cell.

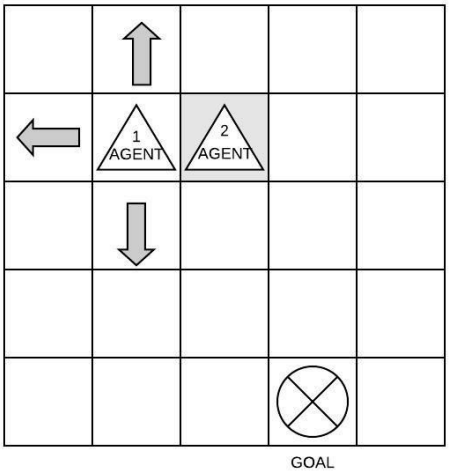


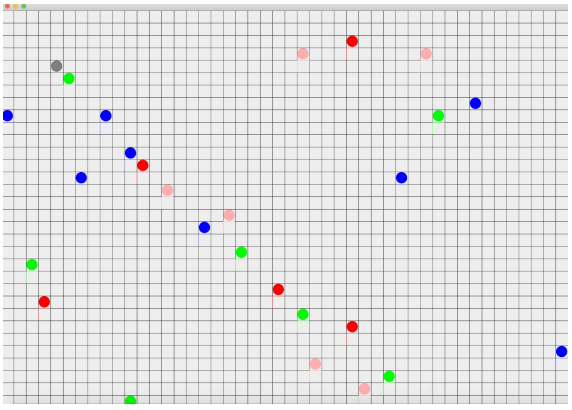
Fig.3 Agent 1 cannot move right since it is occupied by Agent 2 (obstacle).

We have the additional variable 'index' that maintains the position of the respective agent.
The updatePosition() function of the environment class changes as follows –
Function: updatePosition, input ->a,b, x,y,index output->void
1: matrix[x][y] <- 0,
2: **if** a== goalX and b = goalY **then** return
3: **else** matrix[a][b] <-index
4: return
Additionally, we also have the isSafe function in the environment class.While determining the nextPosition, this function is invoked every time to ensure that the next cell is not an obstacle. Also, it is again invoked in the updatePosition function to recheck inconsistency due to parallelism.
It is as follows –
Function: isSafe, input ->x,y, output->integer
1: **if** 0<= x < n and 0<=y<m and matrix[x][y] =0 **then** return 1
2: **else** return 0
The main function alters to create the threads and join them as follows –
Function: MAIN,
1: ArrayList xi (Agent)
2: **for** i<-0 **to** i < noOfThreads
3: Agent agent()<- goalX, goalY, random_X, randomY, i
4: xi.add(agent)
5: i <- i+1
6: **end for**

7: **for** i<-0 **to** i < noOfThreads
8: xi.get(i).start()
9: i<- i+1
10: **end for**
11: **for** i<-0 **to** i < noOfThreads
12: xi.get(i).join()
13: i<- i+1
14: **end for**

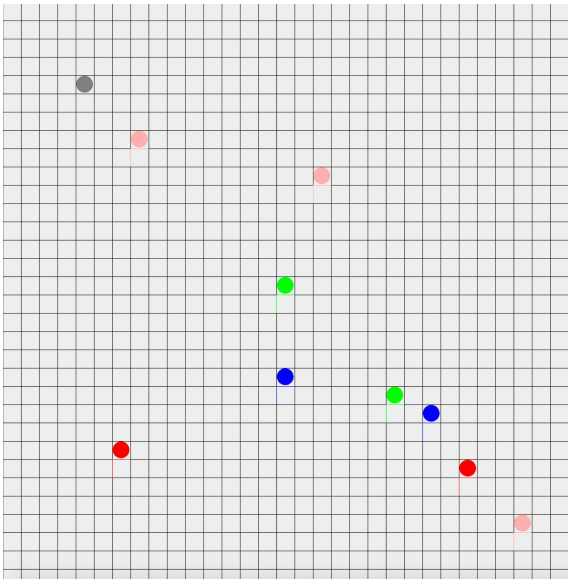
(D) For the next and final part, we alter the algorithm to depict a real world model where the agent doesn't move as per the nextPosition 10% of the time.
To achieve this, in the run function of Agent class, after getting the nextPosition, we generate a random integer in the range 0 to 9 and proceed with updation only if the integer is not 0 (probability of 0.9). Else, we recalculate the nextPosition.

IV. RESULT AND CONCLUSION



The given multiagent algorithm was implemented using JAVA swing for obtaining the GUI of the same. It is seen how the agents move towards the goal (grey circle on top left corner) without colliding with each other.

For 10 agents, the result is –



V. REFERENCES

[1] Stuart Russel, Peter Norvig, "Artificial Intelligence, A Modern Approach", Third Edition
[2]Multithreading,https://www.javatpoint.com/multithreading-in-java
[3]https://www.geeksforgeeks.org/agents-artificial-intelligence/