Multi-Agent Path Finding AI

Assignment 1 – AIFA (2021)

Documentation

Team Members:

- Snehal Swadhin (19ME10067)
- Priyam Saha (19HS20030)
- Bhosale Ratnesh Sambhajirao (19MF10010)

User Manual

- To run the algorithm, simply run the command "python main.py" inside the Source_code folder. (Note - All the files inside the Source_code folder must be present for running the algorithm)
- All the parameters of the problem have been defined at the end of the file 'main.py' inside the Source_code folder and passed to the main(_) function.
- The input to the algorithm can be changed by changing the following variables defined at the end of 'main.py'.
- All locations have been defined as (row no., column no.)
- The warehouse has been defined using a list of locations of blocks/obstacles (Variable name -> blocks), along with the height (Variable name -> height) and width (Variable name ->width) of the warehouse. By changing these three variables, a user can define their warehouse layout.
- The Robots have been defined as a list of their start and end locations (Variable name -> bots). Syntax used to define robots : [(R(i).x, R(i).y), (E(i).x, E(i).y)]

  where,
  ```
  R(i) = Start location of ith robot
  R(i).x = Row number of R(i)
  R(i).y = Column number of R(i)
  E(i) = End location of ith robot
  E(i).x = Row number of E(i)
  E(i).y = Column number of E(i)
  ```
- The Tasks have been defined as a list of their pick-up and destination locations (Variable name -> tasks). Syntax used to define robots : [(P(i).x, P(i).y), (D(i).x, D(i).y)]

  where,
  ```
  P(i) = Pick-up location of ith robot
  P(i).x = Row number of P(i)
  P(i).y = Column number of P(i)
  D(i) = Destination location of ith robot
  D(i).x = Row number of D(i)
  D(i).y = Column number of D(i)
  ```
- After setting these variables to your liking, you can execute the code in 'main.py' by running the command "python main.py" inside the Source_code folder.

Algorithm Details

The crux of the multi-agent pathfinding problem comes from the same thing that differentiates it from a classic path finding algorithm. It is the freedom given to us that the work can be divided among multiple agents. Because of this speciality of the problem, we have to come up with methods to ensure that the division of tasks is done optimally. The problem also contains various subproblems that need to be solved in order to complete the big picture.

In this project, we have divided the problem into three major subproblems.

1) Assignment of tasks to various agents
2) Finding the shortest path between two points.
3) Resolving conflicts (what we refer to as collisions in this particular problem)

Task Assignment:

Although this subproblem can also be solved as a Search problem, a more intuitive and possibly sub-optimal method has been used for the same at this current state of the project.

- Proposed search method:

Firstly, we shall highlight how a search algorithm may be used to fulfil this task. We are given a list of tasks, a warehouse layout, and a list of agents/robots with certain initial positions in the warehouse. We define the state of the problem as a list of task-assignments, i.e., a list of lists where the ith lower order list contains all the tasks assigned to the ith agent. And the state space contains the final positions of all agents after they complete their respective tasks.

The state transformation rules will allow any task to be assigned to any agent. For deciding which agent should be allotted that task, we shall take the help of a heuristic or f-value –> (distance from agent position to task pick-up location) + (the distance between the pickup and drop locations of the task).

After a task has been assigned to an agent, the actual path to be travelled is calculated and stored as g-value of the state. Using any search method like A* or Iterative deepening DFS, we should traverse this state-space and a goal node is reached when all the tasks have been allotted to some agent.

- Method currently used in this project:

We have used a list task-assignments just like explained above, i.e., a list of lists where the ith lower order list contains all the tasks assigned to the ith agent. The input list of tasks has been sorted in descending order of its heuristic length (i.e., the distance between pick-up and drop locations). Now these tasks are sequentially assigned to agents having the lowest heuristic f-value among all agents.

Here, f-value = (distance from agent position to task pick-up location) + (the distance between the pickup and drop locations of the task).

So, the working of the algorithm would look like Tetris blocks (predicted task-lengths) of decreasing length falling vertically from the top. And we put the Tetris block in that column which currently has the least blocks already stacked on it.

Shortest Path:

The shortest path between two points in a maze which should not pass through any obstacles is optimally found using an implementation of the A* (best-first) search algorithm. We convert the whole maze into a grid of nodes whose g and f values are yet to be calculated. The heuristic used in this algorithm is the Manhattan distance between two points, because the robot is allowed to travel only single steps vertically and horizontally.

Starting from the source node, we calculate heuristic values of neighbouring nodes and add them in the open list. In the next iteration, we find and remove the node in the open list having the least f-value (i.e., heuristic distance + path length from source). This node is now marked as 'visited' or 'closed', so that it may not be considered again. We also keep track of the parent of these nodes (i.e., the node from which we reached the current node using the state transformation rules). A node in the open list can be reached from some other path, and it can happen that the new path turns out to be shorter than the one found before. In those cases, we update the f and g values of this node along with its new parent.

The process continues till we reach the goal node, or the open list becomes empty. In the case that the list is emptied before reaching the goal, we can conclude that no valid path exists between the two points. In the other case that we do reach the goal node, we trace back the parents of each node starting from the goal node to obtain the shortest path. This is ensured because the parent of any node is the node which connects the source and the current node through the shortest path.

Generally, any A* implementation uses a priority queue to store nodes in the open list so that searching for the smallest f-value among all the nodes can be avoided. This is done because a priority queue's time complexity for addition or deletion is still very less than linearly searching for the minimum f-value. But Python does not have a predefined priority queue implementation. So, instead we have used the heapq library, in order to make the open list as a min-heap, which acts like a substitute for priority queue by keeping the minimum value at the top of the heap, i.e., at the first index of the list.