Artificial Intelligence: Foundations and Applications
AI61005

Report on

Multi-Agent Path Finding

Under the guidance of

Prof. P. P. Chakraborty, Prof. Arijit Mondal

By -

Amber Shekhar Shrivastava(18NA30003)

Ravindra Vijaybahadur Patel(18NA30014)

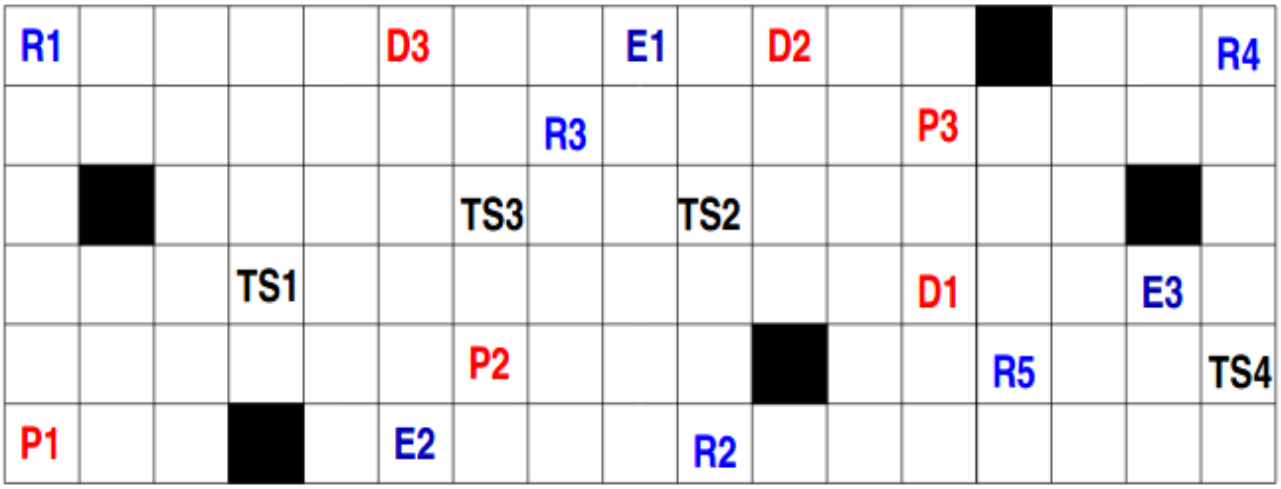
Sarthak Vijay(18NA30018)

Problem Statement :

[Multi Agent Path Finding] Consider a warehouse and a set of robots which pick up items from designated places, deliver those in desired locations, and finally the robots go to the end location. Assume that the warehouse is represented as a 2-D grid of size $n \times m$. Each location (L_i) on warehouse floor can be denoted by a pair of integers $L_i = (x_i, y_i)$. Each cell on the grid can be categorized in one of the following ways (see diagram below) - source location (P1- P3), destination location (D1-D3), temporary storage location (TS1-TS4), obstacle (black square), normal (rest of the cells). Source & destination denote pick-up and drop locations respectively. Temporary storage location denotes the place where robot can keep some items. Obstacles represent a location where no robot can move. Rest of the cells are considered as normal cell. Let there be k number of robots and r number of tasks. The details of robot location and tasks are provided as per the following table.

Robot	Location		Task	Location	
	Init	Final		pick-up	deliver
Robot-1	R1	E1	Task-1	P1	D1
Robot-2	R2	E2	Task-2	P2	D3
Robot-3	R1	E2	Task-3	P1	D3
. . .	. . .	. . .	. . .	. . .	. . .

Assume that a robot can move at most one cell (either vertically or horizontally) at a time step, a normal cell can be occupied by at most one robot. Source, destination, temporary storage locations can accommodate multiple robots simultaneously. Our target here is to develop a work schedule that minimizes the time to complete all tasks. You need to develop both optimal as well as heuristic algorithms.



Theory

Single Agent Path Finding

In single agent path finding, we are given two(or more) nodes and there is only robot/agent to complete that task and we have to minimise the cost/time. It has many applications such as GPS Navigation, Robot routing path, planning and many more. It can be done by **A* Algorithm** which have been Implemented in "A_Star1.py" file submitted along with other files. In A* Algorithm i) First solution is the optimal solution. ii) No node in closed is ever reopened. iii) we make use of heuristic estimates (the more accurate they are, less the search will be).
We used the heuristic as follows in line 261 of "CBS.py" file.

```
def admissible_heuristic(self, state, robot_name):
        goal = self.robot_dict[robot_name]["Final"]
        return fabs(state.location.x - goal.location.x) + fabs(
        state.location.y - goal.location.y
        )
```

Where our heuristic estimation is |goal(x) - current(x)| + |goal(y) - current(y)| and thus the fabs were used to determine its absolute value which works as a good enough estimation.

Multi Agent Path Finding

The multi - agent pathfinding (MAPF) problem is a generalization of the single - agent pathfinding problem for k>1 agents. It consists of a graph and a number of agents/robots. For each agent, a unique start state and a unique goal state are given, and the task is to find paths for all agents from their start states to their goal states, under the constraint that agents cannot collide during their movements. In many cases there is an additional goal of minimizing cumulative cost function such as the sum of the time steps required for every agent to reach its goal. MAPF has practical applications in video games, traffic control, robotics and aviation. Algorithms for solving MAPF can be divided into two classes : optimal and sub - optimal solvers. Finding an optimal solution for the MAPF problem is tough, as the state space grows exponentially with the number of agents. Sub-optimal solvers are usually used when the number of agents is large. In such cases, the aim is to quickly find a path for the different agents, and it is often intractable to guarantee that a given solution is optimal. MAPF is very different from Single agent path finding problem as conflicts can occur due to which the optimality of the solution fails as number of agents increase and thus A* is not a very wise option for solving MAPF problems.
The traditional technique to solve MAPF is by using A* based searches but it is not very efficient as it grows exponential in size. Thus, we used Conflict Based Search (CBS) which is a two level algorithm divided in high level and low level search.
The agents are initialized with default paths, which may contain conflicts. The high-level search is performed in a constraint tree (CT) whose nodes contain time and location constraints for a single agent. At each node in the CT, a low-level search is performed for all agents. The low-level search returns single-agent paths that are consistent with the set of constraints given at

any CT node. If, after running the low level, there are still conflicts between agents, i.e. two or more agents are located in the same location at the same time, the associated high-level node is declared a non-goal node and the high-level search continues by adding more nodes with constraints that resolve the new conflict which we will see subsequently in our executed code.

Analysis of the Algorithm

We have submitted the CBS algorithm by the name “CBS.py” in the github repository where we have provided the part by part algorithms to process Nodes, Vertex Constraint, conflicts, states, location, Edge constraints, constraints and finally the Environment.

We have to provide the information about the problem through the “Input.yaml” file as shown:

```yaml
Robots:
-   Name: Robot1
    Initial: [5, 0]
    Final: [3, 12]

-   Name: Robot1
    Initial: [4, 6]
    Final: [0, 10]

-   Name: Robot1
    Initial: [1, 12]
    Final: [0, 5]

-   Name: Robot2
    Initial: [5, 0]
    Final: [0,5]

-   Name: Robot2
    Initial: [4, 6]
    Final: [0, 10]

    Name: Robot2
    Initial: [1, 12]
    Final: [3, 12]

Map:
    Dimension: [6, 17]

    Obstacles:
    - !!python/tuple [2, 1]
    - !!python/tuple [5, 3]
    - !!python/tuple [4, 10]
    - !!python/tuple [0, 13]
    - !!python/tuple [2, 15]

```

Here, the dimension of the warehouse is given on the line 27 as 6 x 17 i.e. 6 rows and 17 columns.
The obstacles are provided below from Line 30 to Line 34, where the coordinate of the obstacle is given in the form of a Tuple.
The name, initial position and final position of the robots are given above.

Once we provide the Input file and follow the Steps mentioned in the Installation guide, the algorithm will generate an output file which will be as follows -

```
Robot1:
- t: 0
  x: 5
  y: 0
- t: 1
  x: 5
  y: 1
- t: 2
  x: 4
  y: 1
- t: 3
  x: 4
  y: 2
- t: 4
  x: 4
  y: 3
- t: 5
  x: 4
  y: 4
- t: 6
  x: 4
  y: 5
```

For each robot, this information will be provided which will provide us with its location at a given instant of time. Like in the above case,
For Robot 1:
at t = 0, its x coordinate is 5 and its y coordinate is 0 (i.e. its initial position)
At t=1, its x coordinate is 5 and its y coordinate is 1(i.e. It moved one step in y coordinate)
And so on.

In the main(), we access the given information through the Input.yaml and pass it to the respective classes-

```
def main():
    with open(r"D:\AIFA Assignment 1\Input.yaml") as param_file: #Change location of Input file
        try:
            param = yaml.load(param_file, Loader=yaml.FullLoader)
        except yaml.YAMLError as exc:
            print(exc)

    dimension = param["Map"]["Dimension"]
    obstacles = param["Map"]["Obstacles"]
    robots = param["Robots"]

    new_robots = []
    robot_name = []
    remaining_robots = []
    robot_time = {}
    time = 0
    k = 1
```

Here we take out the values of dimension, obstacles and robots

we defined the location as

```
class Location(object):
    def __init__(self, x=-1, y=-1):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return str((self.x, self.y))
```

And then we defined the state as

```
class State(object):
    def __init__(self, time, location):
        self.time = time
        self.location = location

    def __eq__(self, other):
        return self.time == other.time and self.location == other.location

    def __hash__(self):
        return hash(str(self.time) + str(self.location.x) + str(self.location.y))

    def is_equal_except_time(self, state):
        return self.location == state.location

    def __str__(self):
        return str((self.time, self.location.x, self.location.y))
```

We make a dictionary of Robots(i.e. Their location at each time-step) as follows -

```
  def make_robot_dict(self):
      for robot in self.robots:

          start_state = State(
              self.robots_time[robot["Name"]],
              Location(robot["Initial"][0], robot["Initial"][1]),
          )
          goal_state = State(
              self.robots_time[robot["Name"]],
              Location(robot["Final"][0], robot["Final"][1]),
          )

          self.robot_dict.update(
              {robot["Name"]: {"Initial": start_state, "Final": goal_state}}
          )
```

In the environment, we first find the the conflicts using the low level algorithm and then we then update the constraints keeping the constraints in mind and returning the updated constraints dictionary as follows -

```
def create_constraints_from_conflict(self, conflict):
    constraint_dict = {}
    if conflict.type == Conflict.VERTEX:
        v_constraint = VertexConstraint(conflict.time, conflict.location_1)
        constraint = Constraints()
        constraint.vertex_constraints |= {v_constraint}
        constraint_dict[conflict.robot_1] = constraint
        constraint_dict[conflict.robot_2] = constraint

    elif conflict.type == Conflict.EDGE:
        constraint1 = Constraints()
        constraint2 = Constraints()

        e_constraint1 = EdgeConstraint(
            conflict.time, conflict.location_1, conflict.location_2
        )
        e_constraint2 = EdgeConstraint(
            conflict.time, conflict.location_2, conflict.location_1
        )

        constraint1.edge_constraints |= {e_constraint1}
        constraint2.edge_constraints |= {e_constraint2}

        constraint_dict[conflict.robot_1] = constraint1
        constraint_dict[conflict.robot_2] = constraint2

    return constraint_dict
```

Drawbacks of A* for MAPF

A* always begins by expanding a state and inserting its successors into the openlist (denoted OPEN). All states expanded are maintained in a closed list (denoted CLOSED). Because of this, A* for MAPF suffers from two drawbacks. First, the size of the state space is exponential in the number of agents(k), meaning that CLOSED cannot be maintained in memory for large problems. Second, the branching factor of a given state may be exponential ink. Consider a state with 20 agents on a 4-connected grid. Each agent may have upto 5 possible moves (4 cardinal directions and `wait`). Fully generating all the 5^20 = 9.53 × 10^14 neighbors of even the start state could be computationally infeasible. The following enhancements have been proposed to overcome these drawbacks.

i) Reducing the effective number of agents with Independence detection(ID).
ii) Avoiding Surplus Nodes
iii) Operator decomposition
iv) Enhanced Partial expansion

END OF PROJECT