CS520: Assignment 2 - Partial Sensing

Team Info:

1. Harish Udhayakumar - hu33
2. Parvathi Mahesh Hedathri - pm850
3. Kavya Kavuri - kk1069

Agent 3 Implementation:

Partial_sensing_agent3:

A class with attributes: *Nx*, *Cx*, *Bx*, *Ex*, *Hx, visited*, *state*, *neigbour_list*, *row*, *column*. The attribute *state* is initialized with -1 i.e., unconfirmed(hidden), set to 0 for empty and 1 for blocked.

Matrices in our code:

Full_gridworld: The complete gridworld which has 0 stored for unblocked and 1 for blocked cells.

Agent_gridworld: The discovered girdworld which is initially filled with 0 and updated with 1 for blocks when discovered.

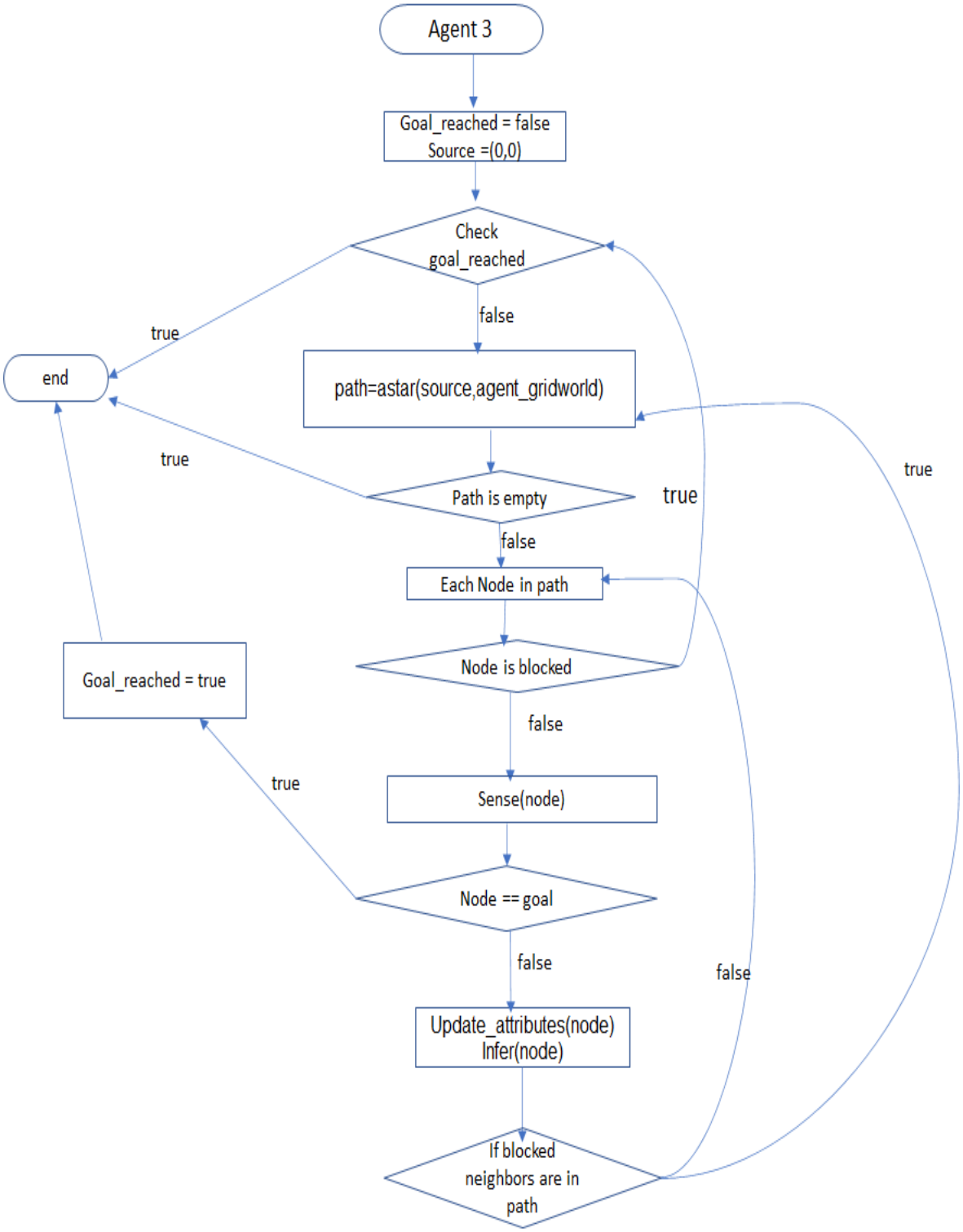
Knowledge_base_agent3: Knowledge base for agent 3 where each cell is an object of Partial_sensing_agent3 class.

Infer function:

A recursive function which checks for 3 inference rules given in assignment. If the state of hidden cells is updated (either by inference or visiting), then we update the knowledge base and recursively run inference on visited neighbors for that cell. The base condition of our recursive function is when Hx==0, that is, nothing else is to be inferred for that cell. Thus, avoiding unnecessary computations.

Appendix has the main code for Agent-3.

Workflow of agent 3:

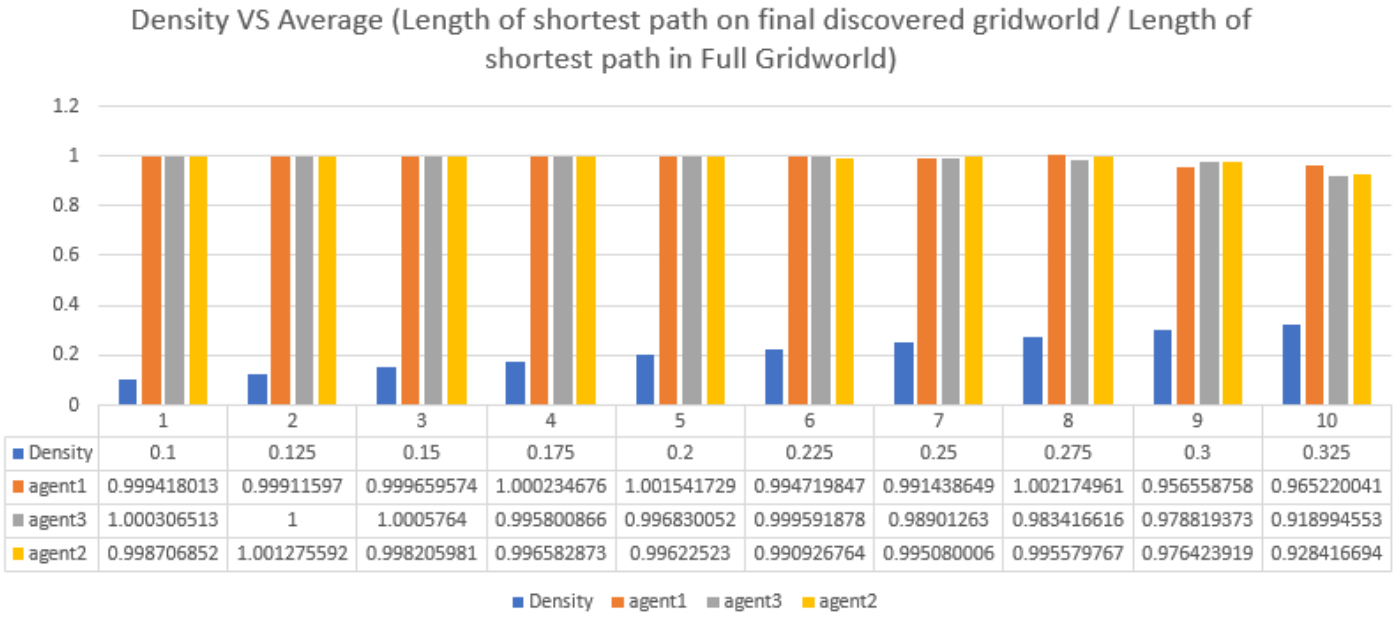


Agent-3 VS other agents:

Assumptions:
Agent 1 - Can see clearly in 4 cardinal directions
Agent 2 - Blindfolded
Agent 3 - Can sense in 8 directions

Graphical representations:

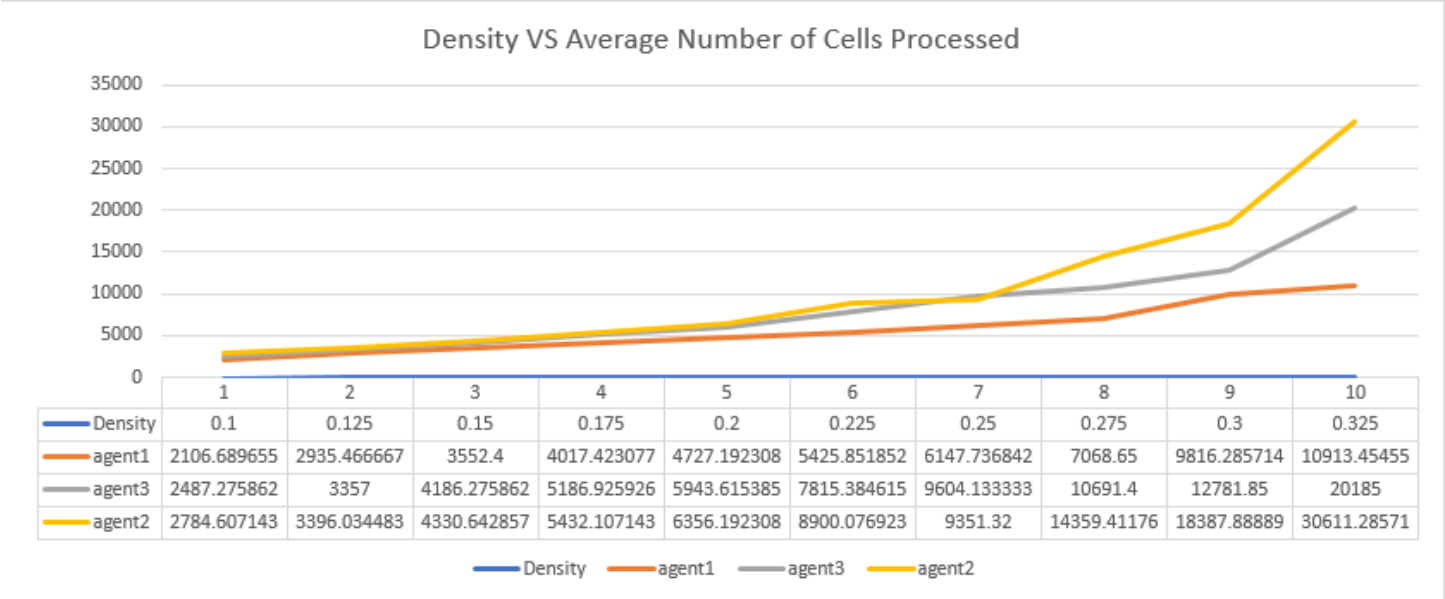
Density VS Average (Length of shortest path on final discovered gridworld / Length of shortest path in Full Gridworld)



	1	2	3	4	5	6	7	8	9	10
Density	0.1	0.125	0.15	0.175	0.2	0.225	0.25	0.275	0.3	0.325
agent1	0.999418013	0.99911597	0.999659574	1.000234676	1.001541729	0.994719847	0.991438649	1.002174961	0.956558758	0.965220041
agent3	1.000306513	1	1.0005764	0.995800866	0.996830052	0.999591878	0.98901263	0.983416616	0.978819373	0.918994553
agent2	0.998706852	1.001275592	0.998205981	0.996582873	0.99622523	0.990926764	0.995080006	0.995579767	0.976423919	0.928416694

The ratios are close to 1 for all densities. The ratios will be equal irrespective of the densities because the Length of Shortest Path in Final Discovered Gridworld and Length of Shortest Path in Full Gridworld will always be equal.

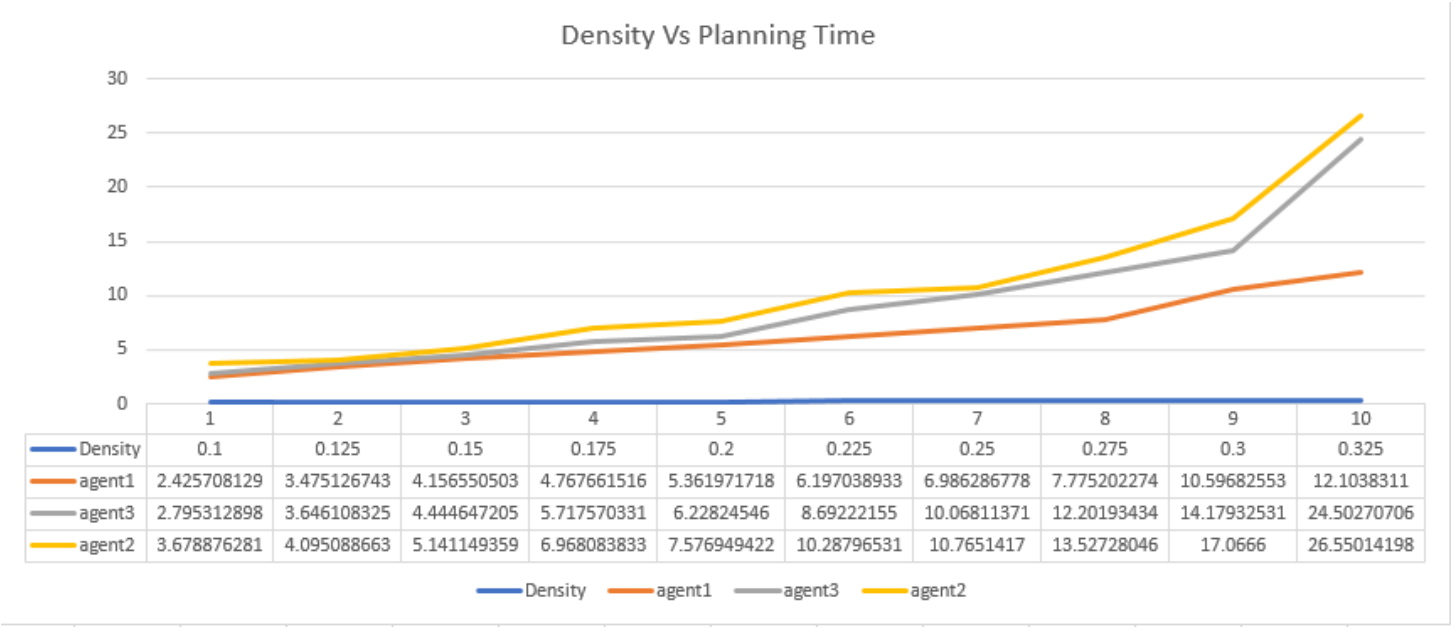
Density VS Average Number of Cells Processed



	1	2	3	4	5	6	7	8	9	10
Density	0.1	0.125	0.15	0.175	0.2	0.225	0.25	0.275	0.3	0.325
agent1	2106.689655	2935.466667	3552.4	4017.423077	4727.192308	5425.851852	6147.736842	7068.65	9816.285714	10913.45455
agent3	2487.275862	3357	4186.275862	5186.925926	5943.615385	7815.384615	9604.133333	10691.4	12781.85	20185
agent2	2784.607143	3396.034483	4330.642857	5432.107143	6356.192308	8900.076923	9351.32	14359.41176	18387.88889	30611.28571

As the grid grows denser with blocks, it is observed that the Agent1 processes less cells as compared to the other agents. Also, Agent3 processes less cells as compared to Agent2. We can say that the trend noticed is Agen1>Agent3> Agent2, if agents are arranged according to their performance by considering less number of cells processed. This is observed because agent1 strictly sees more so it replans less and when it replans less often it avoids processing cells in a path which has a block. Agent3 senses its surroundings and infers blocks, so compared to blindfolded agent2, it also replans lesser times and avoids processing unwanted cells in a path which has a block.

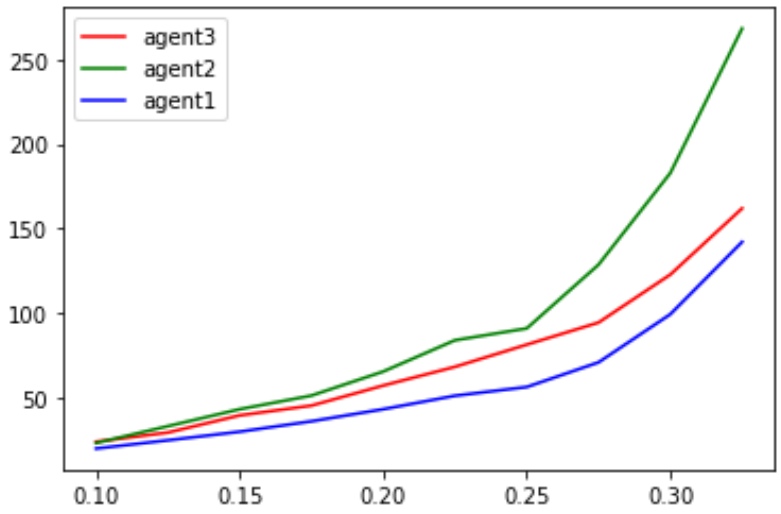
Density Vs Planning Time



	1	2	3	4	5	6	7	8	9	10
Density	0.1	0.125	0.15	0.175	0.2	0.225	0.25	0.275	0.3	0.325
agent1	2.425708129	3.475126743	4.156550503	4.767661516	5.361971718	6.197038933	6.986286778	7.775202274	10.59682553	12.1038311
agent3	2.795312898	3.646108325	4.444647205	5.717570331	6.22824546	8.69222155	10.06811371	12.20193434	14.17932531	24.50270706
agent2	3.678876281	4.095088663	5.141149359	6.968083833	7.576949422	10.28796531	10.7651417	13.52728046	17.0666	26.55014198

It is observed that Agent1 takes less time to plan than other 2 agents for any given density of grid. This observation becomes more prominent when the grid grows denser with blocks. Also, Agent 3 takes less time to plan as compared to Agent2. We can say that the trend noticed is Agen1>Agent3> Agent2, if agents are arranged according to their performance by taking less time in the planning phase. We notice this trend because Agent 1 strictly “sees more”, so planning is faster. Agent 3 senses more, thus, it infers more, and takes lesser time than agent2 to plan. Agent2 being blindfolded takes more time to plan by discovering blocks by bumping into them.

Density Vs Number of Blocks Bumped Into:



We can see agent-1 is bumping into blocks less than agent-3 but following the same trend. And as expected, agent-2 is doing the worst because it is blindfolded.

Design for Agent 4:

To design an inference engine for Agent 4, we use the partially sensed information of cells to get equations.

Rules:

Base rule 1:

If all variables sum up to 0, all the individual variables = 0.

I.e., if a+b+c = 0

=> a=b=c=0

This is basically the same rule of agent 3 which says, if Cx == Bx, then set unknown neighbors to 0

Base rule 2:

If all variables sum upto the number of variables, set all individual variables = 1

I.e., if a+b+c = 3 (here number of variables is 3)

=> a=b=c=1

This is basically the same rule of agent 3 which says, if Nx - Cx == Ex, then set unknown neighbors to 1

Rule to build equations:

Each cell has 3 states 0 -> empty, 1-> blocked, -1 -> unknown

All -1 cells are given variable names, the rest are taken as is (i.e. 0 or 1)

Rule: Sum of state values of cell (x,y) = Cx of cell (x,y)

Example: a cell has Nx = 8 of which 3 are unknown, Cx = 3, Bx =1, Ex = 4. Using this we can get equation:

a+b+c+0+0+0+0+1 = 3

=> a+b+c = 2

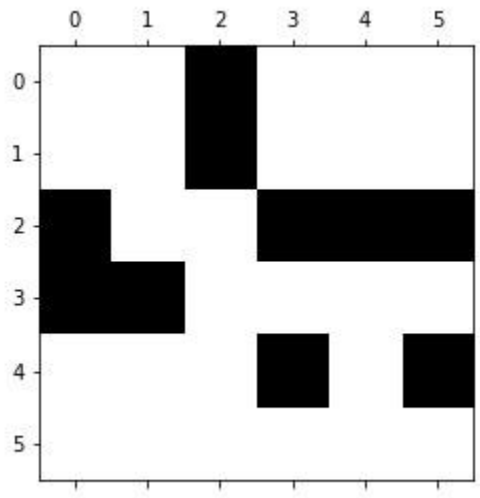
We add this to the system of equations.

Thus, we get linear equations with the left hand side of the equation containing the cell's sensed information and the right hand side of the equation will be the sum of variables assigned to neighbors. Solving these simultaneous equations will help us determine if the cells are empty or blocked. Therefore, the method that Agent-4 is using to infer blocks is solving simultaneous linear equations.

Difference between Agent-4 and Agent-3:

Agent-4 is different from Agent-3 as it is able to infer cells which are not inferred from inference rules of Agent-3. For some cases where inference rules of Agent-3 couldn't infer anything from given information, Agent-4 is capable of finding the state of a cell(blocked or empty) by solving equations.

For example: consider the full gridworld below:



On assigning variables and solving the equations, the knowledge base is updated as below:

(0,0) Visited State=0 Cx=0	(0,1) Visited State=0 Cx=3	(0,2) Visited (bumped) State=1	(0,3) a		
(1,0) Inferred State=1	(1,1) Visited State=0 Cx=3	(1,2) Inferred State=1	(1,3) Inferred State=0		
(2,0) Inferred State=1	(2,1) Visited State=0 Cx=4	(2,2) Visited State=0 Cx=3	(2,3) Visited(bumped) State=1		
(3,0) Inferred State=1	(3,1) Visited (bumped) State=1	(3,2) Visited State=0 Cx=3	(3,3) Inferred State=0		
	b	(4,2) Visited State=0 Cx=2	d		
	e	f	g		

For cell (4,2) in agent 3:
Nx=8, Cx=2, Bx=1,Ex=2,Hx=5
From the given information, nothing new can be inferred. b,d,e,f,g values are still not inferred.

For Agent 4:
For cell(3,2):
We get equation: b+d=1 ---(1)
For cell (4,2):
We get equations:
b+d+e+f+g = 2 ---(2)

On solving the equations (1) and (2):
We get that e=f=g =0

This way agent4 is able to infer better than agent3.

Inferencing of Agent-4:
Although Agent4 infers better than Agent3, there is a possibility that it doesn't infer everything that can be inferred. This could be a possible situation because there can exist few unknown variables and equations that aren't solved by our system of equations after we reach the goal node. Since, our intention is about the fastest and geographically shortest possible way to reach the goal node, we ignore the unknown variables. So, there can be other inference methods to get the values of those unknown variables.

Agent 4 Implementation:

Linear Expression: Node.Cx = neighbor1.state + neighbor1.state + neighbor1.state . . .

Example: if current node is (3,0)

It's neighbors are (2,0) , (2,1), (3,1) , (4,1), (4,0).

The equation becomes = node(30).Cx = node(20).state + node(21).state + node(31).state + node(41).state + node(40).state

What we know -

· If a node is blocked then the node.state = 1 else it is 0.

· Cx of the node (2 in this example)

What we compute or solve –

node(30).Cx = node(20).state + node(21).state + node(31).state + node(41).state + node(40).state

=> 2 = node(20).state + node(21).state + node(31).state + node(41).state + node(40).state

=> **node(20).state + node(21).state + node(31).state + node(41).state + node(40).state – 2 = 0**

- **Equation (1)**

1) If the current node is blocked, then update the knowledge base and substitute it's value in existing expressions and return. We do not compute an expression since we aren't aware of Cx of a blocked node/cell.

2) If solving an expression for a node gives an output = 0, then the node is free. In other words, if the expression contains just the node, it is considered to be free.

Example:
Expression: [node(45)]

Equating it to zero, we get the node(45) to be a free node.

3) Similarly, solving an expression for a node gives an output = 1, then the node is blocked.
Example:
Expression: [node(89) - 1]
Equating it to zero we get node(89) to be 1 that reveals that node(89) is a blocked node

4) If node1 + node 2 = 2, then both the nodes are evaluated as blocked since the maximum value that a node's blocked status can have is 1 to satisfy this condition

5) If node1 + node2 = 0, then both the nodes are evaluated as free nodes, since the least possible value that a node's status can have is 0, to satisfy this condition.

6) Iteratively, these rules are run against every new node that is explored to update the expressions obtained from previously visited nodes

7) The system of equations are solved using python package SymPy

Appendix has the main code for Agent-4.

Computational Issues:

To reduce computational burden:
We can prioritize solving the equations containing variables(unknown cells) which are in the path provided by A-star or the neighbors of the path provided by A-star.
We don't add equations of those cells which have only one known neighbor.
We can maintain a variable dictionary which keeps track of the equations in which that variable is present. So for a given variable, we need to check 8 equations in the worst case, because any given unknown can be a neighbor of 8 cells at the most, thereby be in 8 equations at the most.
We can start creating equations after a couple of inferences. So that we may get less variables in equations.
We don't write equations for those cells which are univisited.

Individual Contribution to the project:

Except for the sympy library for solving linear equations, this program was coded completely by the team. The team split up the work equally amongst each other.

Each individual solved a couple of questions along with individual input to the code build. Answering the question involved analysis of the code output, data collection, plotting graphs, and adding the answer to the report.

Harish: Worked on Agent-4 implementation, report
Kavya: Worked on Agent-3 implementation, graphs for Agent-3 vs other agents, report
Parvathi: Worked on Agent-3 implementation, graphs for Agent-3 vs other agents, report

Link to the code:
Agent-3: https://github.com/harish-udhayakumar/partial_sensing/tree/agent_3
Agent-4: https://github.com/harish-udhayakumar/partial_sensing/tree/agent_4

Appendix

Agent3 function:

```
def agent3():
"""
    This function runs agent 3 and
    returns the shortest path on discovered grid world, trajectory_length, and number of cells processed.
    knowledge_base_agent3: this matrix stores information of partial sensing objects.
    These objects have attributes like Nx,Cx,Ex,Hx,state,visited.
    agent_gridworld: knowledge base/discovered grid world
"""

    global knowledge_base_agent3
    global maze_size
    global full_grid_world_matrix
    global agent_gridworld
    goal = (maze_size-1,maze_size-1)
    source = (0,0,0)
    goal_reached = False
    trajectory_length = 0
    num_of_cells_processed = 0
    parent = ()
    planning_time =0


    while not goal_reached:
```

```
    start_time = time.time()
    _,_,astar_num_cells,path = astar(source,agent_gridworld)
    planning_time += (time.time()-start_time)
    num_of_cells_processed += astar_num_cells
    path.reverse()
    if (len(path) == 0):
        print(path)
        return [],0,0
    for node in path:
        row = node[0]
        col = node[1]
        if full_grid_world_matrix[row,col] == 1:
            agent_gridworld[row,col]=1
            knowledge_base_agent3[row,col].state = 1
            trajectory_length += 2
            source = parent
            break
        elif full_grid_world_matrix[row,col] == 0:
            knowledge_base_agent3[row,col].visited = True
            knowledge_base_agent3[row,col].state = 0
            knowledge_base_agent3[row,col].update_visited_neighbor_attributes(knowledge_base_agent3)
            trajectory_length += 1
            parent = (row,col,0)
            if node == goal:
                goal_reached = True
                set_agent_gridworld_from_knowledge_base()
                _,_,_,shortest_path_final_agent_gridworls = astar((0,0,0),agent_gridworld)
                return shortest_path_final_agent_gridworls, trajectory_length, num_of_cells_processed,planning_time

            knowledge_base_agent3[row,col].update_attributes(maze_size,helper_matrix,knowledge_base_agent3)

            #Calling inference engine
            infer(node)

            #if there is block in path, then discard this path, and get new path from astar with this node as source
            blocked_neighbors = knowledge_base_agent3[row,col].get_blocked_neighbors(knowledge_base_agent3)
            if len(list(set(tuple(blocked_neighbors))& set(tuple(path)))) != 0:
                source = (node[0],node[1],0)
                set_agent_gridworld_from_knowledge_base()
                break
    set_agent_gridworld_from_knowledge_base()
    _,_,_,shortest_path_final_agent_gridworld = astar((0,0,0),agent_gridworld)
    return shortest_path_final_agent_gridworld, trajectory_length, num_of_cells_processed,planning_time
```

Inference engine for agent3:

```
"""
    This function runs the inference engine for agent 3 by following the given inference rules.
    """
def infer(node):
    global knowledge_base_agent3
    row = node[0]
    col = node[1]
# if Hx=0, then nothing is left to infer
    if knowledge_base_agent3[row, col].Hx == 0:
        return

# if Cx=Bx, remaining hidden cells are empty
    if knowledge_base_agent3[row, col].Cx == knowledge_base_agent3[node].Bx:
        knowledge_base_agent3[row, col].Ex += knowledge_base_agent3[node].Hx
        knowledge_base_agent3[row, col].Hx = 0
        uknown_neighbors = knowledge_base_agent3[row, col].get_unknown_neighbors(knowledge_base_agent3)
        knowledge_base_agent3[row, col].set_unknown_neighbors_empty(knowledge_base_agent3)
        for i in uknown_neighbors:
            knowledge_base_agent3[i[0],i[0]].update_visited_neighbor_attributes(knowledge_base_agent3)
        for i in uknown_neighbors:
            for j in knowledge_base_agent3[i[0],i[0]].neighbor_list:
                if knowledge_base_agent3[j[0],j[1]].visited == True:
                    infer(j)

# if Nx - Cx=Ex, remaining hidden cells are blocked
    elif knowledge_base_agent3[row, col].Nx - knowledge_base_agent3[row, col].Cx == knowledge_base_agent3[row,
col].Ex:
        knowledge_base_agent3[row, col].Bx += knowledge_base_agent3[row, col].Hx
        knowledge_base_agent3[row, col].Hx = 0
        uknown_neighbors = knowledge_base_agent3[row, col].get_unknown_neighbors(knowledge_base_agent3)
        knowledge_base_agent3[row, col].set_unknown_neighbors_blocked(knowledge_base_agent3)
        for i in uknown_neighbors:
            knowledge_base_agent3[i[0],i[1]].update_visited_neighbor_attributes(knowledge_base_agent3)
        for i in uknown_neighbors:
            for j in knowledge_base_agent3[i[0],i[1]].neighbor_list:
                if knowledge_base_agent3[j[0],j[1]].visited == True:
                    infer(j)
```

Agent 4 function:

```
def agent4(maze_size, source):
    global visited_list
    global number_of_bumps
    global knowledge_base
```

```
    global agent_gridworld
    global full_grid_world_matrix
    global confirmed_path
    global infered_list
    global expression_list
    global number_of_bumps
    global confirmed_path

    _,_,_,path = astar(source,agent_gridworld)
    path = path[::-1]
    if path == []:
        goal_reached = False
        return visited_list, number_of_bumps, infered_list, goal_reached
    goal_reached = False
    knowledge_base[0][0].blocked = 0
    i = 0

    for node in path:
        row = node[0]
        col = node[1]

        # terminate the program if we reach the goal node
        if (row, col) == (maze_size - 1, maze_size - 1):
            goal_reached = True
            print("goal reached")
            visited_list.append((row,col))
            confirmed_path.append((row,col))
            return visited_list, number_of_bumps, infered_list, goal_reached

        # retrieve the respective object from knowledge base
        node_object = knowledge_base[row][col]
        node_object.blocked = full_grid_world_matrix.item((row, col))
        if node_object.blocked == 0:
            previously_unblocked_node = (row,col)

        # if the current node isn't visited previously then add it to visited list and trajectory path
        if ((row,col)) not in visited_list:
            visited_list.append((row,col))
            confirmed_path.append((row,col))
            print(visited_list)

            #if node is blocked, update the agent grid world and run inference engine with the current node's blocked status
        # recursively call the agent4 function to fetch an alternative path to goal node
        if node_object.blocked == 1:
            if not ((row, col)) in infered_list:
```

```
                number_of_bumps += 1
            print(row, col, "is blocked")
            agent_gridworld.itemset((row, col), 1)
            node_object.visited = True
             knowledge_base, infered_list, expression_list, agent_gridworld  = node_object.inference_engine(visited_list, infered_list, knowledge_base, expression_list, agent_gridworld)
            confirmed_path.remove((row,col))
             visited_list, number_of_bumps, infered_list, goal_reached = agent4(maze_size, previously_unblocked_node + (0,))
            if goal_reached == True:
                return confirmed_path, number_of_bumps, infered_list, goal_reached

        else:
            # this block runs if the node is free, it updates the knowledge base and runs the inference engine
            print(row, col, "is free")
            neighborlist = get_neighbors(row, col, maze_size)
            node_object.neighbor_list = neighborlist
              knowledge_base, infered_list,expression_list, agent_gridworld  = node_object.inference_engine(visited_list, infered_list, knowledge_base, expression_list, agent_gridworld)
        node_object.visited = True
        print('infered_list',infered_list)
        i+=1
    return confirmed_path, number_of_bumps, infered_list, goal_reached
```

Inference Engine:

```
    def inference_engine(self, visited_list, infered_list, knowledge_base, expression_list, agent_gridworld):
        visited_plus_infered_list = []
        list_of_neighbor_status = []
        expr = ""
        visited_plus_infered_list.extend(visited_list)
        visited_plus_infered_list.extend(infered_list)

        # if current node is blocked substitute current node's blocked status into expressions and return
        # we donot compute the Cx value for the blocked node
        if self.blocked == 1:
            for expression in expression_list:
                node_to_substitute = sym.Symbol('node(' + str(self.row) + str(self.column) + ')')
                index = expression_list.index(expression)
                expression_list[index] = expression.subs(node_to_substitute, self.blocked)
            return knowledge_base, infered_list, expression_list, agent_gridworld

        # initializing an expression
      # if the current node is free, we compute an expression as "self.Cx = neighbor_1.blocked + neighbor_2.blocked.."
        # i.e., The equation becomes => neighbor_1.blocked + neighbor_2.blocked .. - self.Cx = 0
        if self.visited == False:
```

```
        a = sym.Symbol('node(' + str(self.row) + str(self.column) + ')')
        for neighbor in self.neighbor_list:
            neighbor_row = neighbor[0]
            neighbor_col = neighbor[1]
            neighbor_node_object = knowledge_base[neighbor_row][neighbor_col]
            if not expr:
                expr = sym.Symbol('node(' + str(neighbor_node_object.row) + str(neighbor_node_object.column) + ')')
            else:
                expr += sym.Symbol('node(' + str(neighbor_node_object.row) + str(neighbor_node_object.column) + ')')
        expr -= a
        expr = expr.subs(a, self.Cx)
        # expression_list has the list of expressions to be computed at any given point in time
        expression_list.append(expr)

    # substitute blocked values for previously visited nodes
    for expression_index in range(len(expression_list)):
        expression = expression_list[expression_index]
        #updating current node's value in the expression
        for neighbor in visited_plus_infered_list:
            neighbor_row = neighbor[0]
            neighbor_col = neighbor[1]
            neighbor_node_object = knowledge_base[neighbor_row][neighbor_col]
                                    node_to_substitute = sym.Symbol('node(' + str(neighbor_node_object.row) +
str(neighbor_node_object.column) + ')')
            substituted_expression = expression.subs(node_to_substitute, neighbor_node_object.blocked)
            substituted_expression = substituted_expression.subs(node_to_substitute, self.blocked)
            expression_list[expression_index] = substituted_expression
            expression = substituted_expression

    # solving the expression
    for expression in expression_list:
        # condition 1 if the expression evaluates to zero the single node in the expression is free
        if sym.solve(expression)[0] == 0:
            # fetch the node's row and column values adn retrieve it's respective object from knowledge base
            print("Infered the node is: ", str(expression))
            infered_row = int(str(expression)[5:7][0])
            infered_col = int(str(expression)[5:7][1])
            infered_object = knowledge_base[infered_row][infered_col]

            # if the node isn't attached to visited or inferred list we append it to solve the equations for next iteration
            if (infered_row, infered_col) not in visited_plus_infered_list:
                infered_list.append((infered_row, infered_col))
            expression_list.remove(expression)
            knowledge_base[infered_row][infered_col].blocked = 0  # updating the knowledge base
        # condition 2 if the expression evaluates to one the single node in the expression is blocked
        elif sym.solve(expression)[0] == 1:
```

```
            # fetch the node's row and column values and retrieve its respective object from knowledge base
            print("Inferred the node is: ", str(expression))
            infered_row = int(str(expression)[5:7][0])
            infered_col = int(str(expression)[5:7][1])
            infered_object = knowledge_base[infered_row][infered_col]

            # if the node isn't attached to visited or inferred list we append it to solve the equations for next iteration
            if (infered_row, infered_col) not in visited_plus_infered_list:
                infered_list.append((infered_row, infered_col))
            expression_list.remove(expression)
            knowledge_base[infered_row][infered_col].blocked = 0  # updating the knowledge base
        # condition 3 if node1 = node 2 then both the nodes are free eg:- -node(21) = -node(23)
        elif sym.solve(list(sym.solve(expression)[0].values())[0])[0] == 0:

            # node(23)
            infered_row1 = int(str(list(sym.solve(expression)[0].values())[0]).replace("-","")[5:7][0])
            infered_col1 = int(str(list(sym.solve(expression)[0].values())[0]).replace("-","")[5:7][1])

            infered_row2 = int(str(list(sym.solve(expression)[0].keys())[0]).replace("-","")[5:7][0])
            infered_col2 = int(str(list(sym.solve(expression)[0].keys())[0]).replace("-","")[5:7][1])

            knowledge_base[infered_row1][infered_col1].blocked = 0   # updating the knowledge base

            # if the node isn't attached to visited or infered list we append it to solve the equations for next iteration
            if (infered_row1, infered_col1) not in visited_plus_infered_list:
                infered_list.append((infered_row1, infered_col1))

            knowledge_base[infered_row2][infered_col2].blocked = 0   # updating the knowledge base

            # if the node isn't attached to visited or infered list we append it to solve the equations for next iteration
            if (infered_row2, infered_col2) not in visited_plus_infered_list:
                infered_list.append((infered_row2, infered_col2))

            expression_list.remove(expression)

            print('infered node is: ', list(sym.solve(expression)[0].values())[0])
            print('infered node is: ', list(sym.solve(expression)[0].values())[0])

    return knowledge_base, infered_list, expression_list, agent_gridworld
```