RUTGERS
UNIVERSITY | NEW BRUNSWICK

Utsav Patel (RUID - 211009880, NetID - upp10)
Manan Vakta (RUID - 206003851, NetID - mv651)
Harshal Sinha (RUID - 203008538, NetID - hs1030)

Master of Science in Computer Science

Partial Sensing

October 2021

The School of Graduate Studies

Contents

List of Figures

Chapter 1

Agent Implementation and design

1.1 Agent 1 and 2

The implementation that we have followed is exactly the same implementation that we had done for project 1. All our agents inherit from a parent class called Agent that has the following properties -

1.1.1 Agent.py

- A copy of the maze that is being tested where all cells are objects of the **Cell** class. Each cell is assumed to be unblocked until we get information that confirms a block.
- Here we can also set and store various parameters for the agent such as
  - The algorithm to follow (A* or BFS).
  - Number of Confirmed cells.
  - Number of A* calls.
  - Current position of the agent.
  - Size and coordinates of final path taken to goal, etc.
- We keep a useful **reset_function** that helps us reset all the parameters of the agent and also the maze of the agent. An important thing to note is that this function does not reset the h(n) values that we calculate for each cell in the maze.

- Lastly we have a **planning_function** that helps us change the planning stage of our agent. It is in this function that we calculate the number of cells processed by the agent.

1.1.2 TheBlindFoldedAgent.py

- Our blindfolded agent inherits all of the traits from our Agent base class.
- In the blindfolded agent class we define the execution of the agent:
  - Whenever we go from the planning phase to the execution phase, we call the agent's execution function.
  - When we call the execution function we increase the number of A* calls for our agent by 1.
  - In the execution phase we call the repeated forward A* function to which we specify that the **field_of_view** of our agent is 0.

1.1.3 TheFourNeighborAgent.py

The Four Neighbor agent is exactly the same as our blindfolded agent except when we call the execution function we specify that the **field_of_view** of our agent is 4.

1.2 Agent 3

1.2.1 TheExampleInferenceAgent.py

- The ExampleInference agent inherits the Agent class.
- The following information is maintained for it:
  - **num_neighbor**: the number of neighbors cell x has.
  - **is_visited**: Whether or not cell x has been visited.
  - **is_confirmed** Whether or not cell x has been confirmed as empty or blocked, or is currently unconfirmed.
  - **num_sensed_blocked**: the number of neighbors of x that are sensed to be blocked.
  - **num_sensed_unblocked**: the number of neighbors of x that are sensed to be unblocked.

  - **num_confirmed_blocked**: the number of neighbors of x that have been confirmed to be blocked.
  - **num_confirmed_unblocked**: the number of neighbors of x that have been confirmed to be empty.

- It makes use of the method **sense_current_node()** present in our **helper** class for sensing the neighbourhood of the cell it is currently in. The sensing takes place in the following way:
  - If the neighbouring cell's status is confirmed then we increment the count of both the confirmed cells and the number of sensed cells depending on whether it is blocked or unblocked.
  - If the neighbouring cell's status is unconfirmed then we only increment the count of the number of sensed cells depending on whether it is sensed as blocked or unblocked.
- It makes use of the method **inference()** present in the **helper** class to derives inferences after each movement of the agent in a new cell. The inference followed is the same as shown in the Assignment document and is as follows:
  - For each neighbour the count of the **num_confirmed_blocked** and **num_confirmed_unblocked** is incremented accordingly.
  - If the number of blocks sensed equals the number of cells confirmed as blocked, then we infer that all remaining hidden neighbours are unblocked. This way the agent is able to infer about the neighbours even without having to visit them.
  - Similarly, if the difference of the total number of neighbours and the number of neighbours sensed as blocked equals the number of confirmed_unblocked_cells then we infer that all the neighbours sensed as blocked are definitely blocked. Again, this enables the agent to infer its neighbours without having to visit them.
- The ExampleInference agent uses the **execution()** method to move along the path that has been pre-planned using A* search algorithm.
- When the agent attempts to move into a cell x, the stored info for all cells is updated accordingly depending on whether it is found to blocked or unblocked. The inference method is run until nothing new can be inferred.

1.3 Agent 4

Agent 4 has been designed on top of Agent 3 and thus incorporates all the functionalities that Agent 3 already has. In addition to inference followed by Agent 3 (See section - 1.2.1), to improve the performance of Agent 4 we make use of the following inference strategies:

- 3 Cell Inference
- 2 Cell Inference
- Advanced Inference Strategy

1.3.1 3 Cell Inference

The main strategy that we used to improve the inference done by Agent 4 is to take into consideration how the information given by one cell interacts with information given by other cells. Let us take an example of 3 cells x_1, x_2 and x_3 as given below-

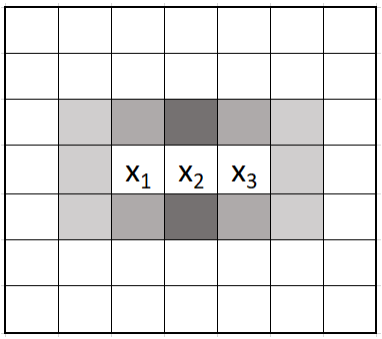

Figure 1.1: x_1, x_2 and x_3

As we can see for this method of inference we need to pick 3 adjacent cells that all have Number of Neighbors = 8. The shaded portion in the grid represents the Neighbors of our Cells x_1, x_2 and x_3. And the degree of the shade represents the number of Cells x_1, x_2 and x_3 are common to.

Our first inference strategy involves adding the information (number of confirmed blocks, number of sensed blocks, number of confirmed unblocked, etc) given by x_1 and x_3 and subtracting x_2 from the information we get i.e -

$$x_1 + x_3 - x_2 \tag{1.1}$$

This leaves us with the information for the following cells show in Figure 1.2-

Let us further illustrate this with an example as shown in Figure 1.3 - where x_1, x_2 and x_3 are

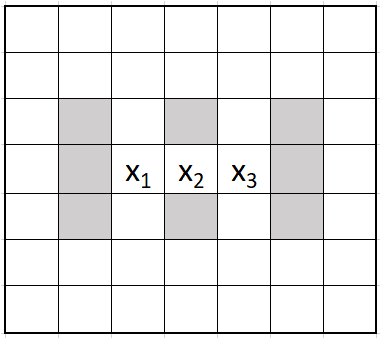

Figure 1.2: x_1, x_2 and x_3

our 3 cells. U denotes an unblocked cell and B denotes a blocked cell. For simplicity's sake we are assuming that the agent starts at x_1 and has explored and confirmed the status of x_1, x_2 and x_3. In a real case we would also have to confirm the cells that the agent has to pass through to get to x_1 -

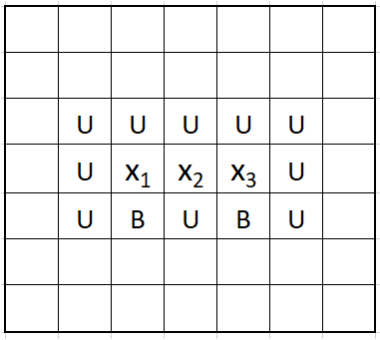

Figure 1.3: x_1, x_2 and x_3

Given below are the values of the variables that we must use for inference for each cell -

x_1	x_2	x_3
$N_{x_1} = 8$	$N_{x_2} = 8$	$N_{x_3} = 8$
$C_{x_1} = 1$	$C_{x_2} = 2$	$C_{x_3} = 1$
$B_{x_1} = 0$	$B_{x_2} = 0$	$B_{x_3} = 0$
$E_{x_1} = 1$	$E_{x_2} = 2$	$E_{x_3} = 1$
$S_{x_1} = 7$	$S_{x_2} = 6$	$S_{x_3} = 7$

Where

- N_x = Number of Neighbors of x
- C_x = Number of Sensed Blocks around x
- B_x = Number of Confirmed Blocks around x

- E_x = Number of Confirmed Empty around x
- S_x = Number of Sensed Empty around x

Let us first check what inference Agent 3 could come up with as it passed through these cells. As it passes through x_1, x_2 and x_3 it will apply its rules of inference individually on each cell, i.e it will check if-

- $C_x = B_x$
- $N_x - C_x = E_x$
- $S_x = E_x$

None of these rules would allow us to further infer and confirm any blocked or unblocked cells around the cells x_1, x_2 and x_3. Even if we take into consideration the status of the x_1, x_2 and x_3 and the fact that we have to pass through some of the neighbors of x_1, x_2 and x_3 to be able to reach them, Agent 3 will not have enough information to infer anything.

Now let us apply the 3-cell inference rule that we will be using for Agent 4. Let us apply $x_1 + x_3 - x_2$

x_1	x_2	x_3	$x_1 + x_3 - x_2$
$N_{x_1} = 8$	$N_{x_2} = 8$	$N_{x_3} = 8$	$N_{x_{123}} = 8$
$C_{x_1} = 1$	$C_{x_2} = 2$	$C_{x_3} = 1$	$C_{x_{123}} = 0$
$B_{x_1} = 0$	$B_{x_2} = 0$	$B_{x_3} = 0$	$B_{x_{123}} = 0$
$E_{x_1} = 1$	$E_{x_2} = 2$	$E_{x_3} = 1$	$E_{x_{123}} = 0$
$S_{x_1} = 7$	$S_{x_2} = 6$	$S_{x_3} = 7$	$S_{x_{123}} = 8$

As we can see here we have obtained the information for the cells shown in Fig 1.2. We clearly get that all these cells must be unblocked. Once the value of these cells is confirmed, we can go back and confirm the status of rest of the cells around x_1, x_2 and x_3 as shown in Fig 1.4

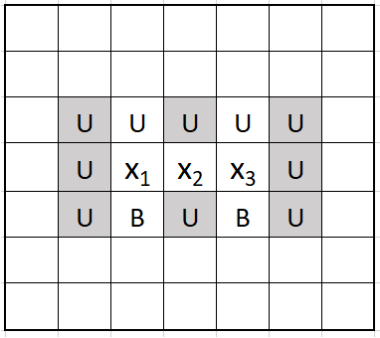

Figure 1.4: x_1, x_2 and x_3

Important Note - This method of inference can be applied anywhere on the grid, vertically or horizontally. We have also implemented this in our code in such a way that it works anywhere on the grid, vertically or horizontally. If x_1 is on the edge of our grid world we assume the 3 cells of x_1 that are not on the grid world are blocked and confirmed. Similarly, we assume neighbors of x_1, x_2 or x_3 that are not on the grid world are blocked. This lets us use $x_1 + x_3 - x_2$ to get information of the above shaded cells almost anywhere on the grid.

1.3.2 2 Cell Inference

Continuing with our theme of using the interaction of information between multiple cells let us take an example of 2 cells x_1 and x_2. For simplicity's sake we are assuming that the agent starts at x_1 and thus has explored and confirmed the status of x_1 and x_2. In a real case we would also be able to confirm the cells that the agent has to pass through to get to x_1 -

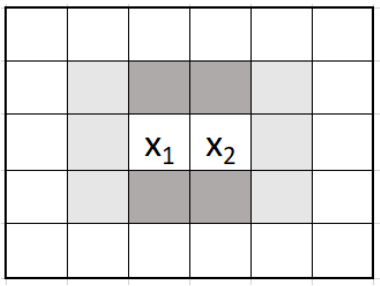

Figure 1.5: x_1, x_2 and x_3

As we can see, here we cannot simply apply $x_1 - x_2$ because we would be subtracting information of 8 neighbors from information of 8 neighbors which would give us information that might be negative or it would be irrelevant information that will not help us find the status of neighbors of x_1 and x_2. So for 2 Cell Inference, we must confirm the status of a few cells from either x_1 or x_2 and then subtract the remainder of the sensed information from the other cells. Let us take a few examples to illustrate this -

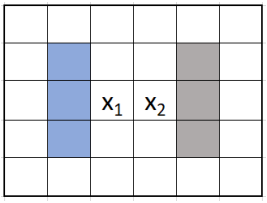

Figure 1.6: Case 1 $x_2 - x_1$

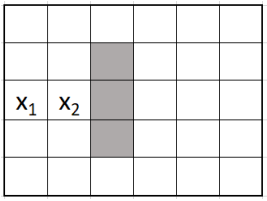

Figure 1.7: Case 2 $x_2 - x_1$

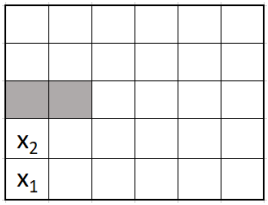

Figure 1.8: Case 3 $x_2 - x_1$

Above are 3 generic cases that our 2 cell inference applies to. In case 1 the blue shaded cells represent cells that we have already confirmed. In cases 2 and 3 cell x_1 is on the border of the grid and thus we assume the neighbors of x_1 that are out of the grid world are blocked and confirmed.

Note - All these cases shown above are generic and can be applied symmetrically, vertically and horizontally anywhere on the grid. We have also implemented this in our code in such a way that it works anywhere on the grid, vertically or horizontally. We ignore the confirmed cells of x_1 and we calculate the sensed neighbors of x_2 in the following way: $x_2 - x_1$ = Number Sensed Neighbors of x_2 − (Number of Sensed Neighbors of x_1− Number of Confirmed Neighbors of x_1 shaded as blue in Figure 1.6) or 0 in case of Figure 1.7, Figure 1.8.

Let us explain each case with some more examples so we can show how Agent 3 cannot infer these same properties and also prove that these inferences give us valuable information.

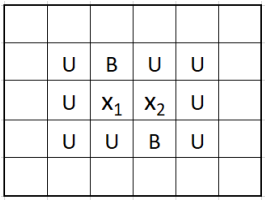

Figure 1.9: Case 1

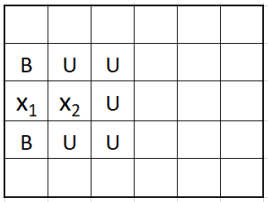

Figure 1.10: Case 2

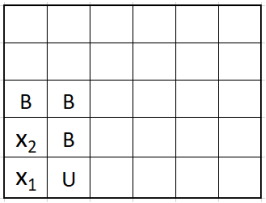

Figure 1.11: Case 3

x_1	x_2	$x_2 - x_1$
$N_{x_1} = 8$	$N_{x_2} = 8$	$N_{x_{12}} = 3$
$C_{x_1} = 2$	$C_{x_2} = 2$	$C_{x_{12}} = 0$
$B_{x_1} = 0$	$B_{x_2} = 0$	$B_{x_{12}} = 0$
$E_{x_1} = 4$	$E_{x_2} = 1$	$E_{x_{12}} = 0$
$S_{x_1} = 6$	$S_{x_2} = 6$	$S_{x_{12}} = 3$

Table 1.1: Case 1

x_1	x_2	$x_2 - x_1$
$N_{x_1} = 5$	$N_{x_2} = 8$	$N_{x_{12}} = 3$
$C_{x_1} = 2$	$C_{x_2} = 2$	$C_{x_{12}} = 0$
$B_{x_1} = 0$	$B_{x_2} = 0$	$B_{x_{12}} = 0$
$E_{x_1} = 1$	$E_{x_2} = 1$	$E_{x_{12}} = 0$
$S_{x_1} = 3$	$S_{x_2} = 6$	$S_{x_{12}} = 3$

Table 1.2: Case 2

x_1	x_2	$x_2 - x_1$
$N_{x_1} = 3$	$N_{x_2} = 5$	$N_{x_{12}} = 2$
$C_{x_1} = 1$	$C_{x_2} = 3$	$C_{x_{12}} = 2$
$B_{x_1} = 0$	$B_{x_2} = 0$	$B_{x_{12}} = 0$
$E_{x_1} = 1$	$E_{x_2} = 1$	$E_{x_{12}} = 0$
$S_{x_1} = 2$	$S_{x_2} = 2$	$S_{x_{12}} = 0$

Table 1.3: Case 3

- **Case 1:** Here we can confirm that the highlighted neighbors of x_2 must all be unblocked as shown in Fig. 1.12
- **Case 2:** Here we can confirm that the highlighted neighbors of x_2 must all be unblocked as shown in Fig. 1.13.
- **Case 3:** Here we can confirm that the highlighted neighbors of x_2 must all be blocked as shown in Fig. 1.14.

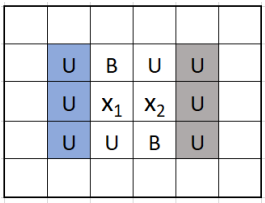

Figure 1.12: Case 1

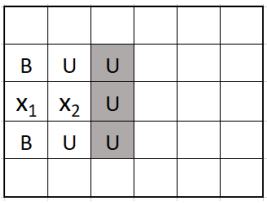

Figure 1.13: Case 2

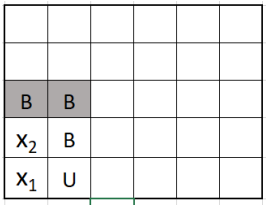

Figure 1.14: Case 3

Note: The inference used by Agent 3 would not be able to use the individual information from x_1 or x_2 to be able to confirm the status of the highlighted cells in any of the given cases even if we take into consideration that x_1 and x_2 are confirmed and the fact that we must path into x_1 and x_2 from its neighboring cells.

1.3.3 Advanced Inference Strategy

In addition the above given 2 cell and 3 cell inference, we are also performing an Advanced Inference Strategy that will help us gain a bit more information. We approached it as a Constraint Satisfaction Problem (CSP) so that we can make use of Constraint Propagation for deriving inferences by using the constraints to reduce the number of legal values for a variable, which in turn can reduce the number of legal values for another variable and so on. The design of agent 4 as well as the Data Structures used for representation of the knowledge base and the constraints are defined below:

- **Constraint:** A constraint is defined as an equation of variables that has an outcome value. It is represented as a *list* containing two elements. The first element is a *Set* which contains the *Tuple* of each variable(each Tuple holds the x and y coordinates of a cell and variable here refers to a cell) and the second element is a value (integer) which denotes the outcome value of the constraint equation.

- For example, suppose the Agent is at a cell X and has the following neighbours represented in the set {a, b, c, d, e, f, g, h}. The agent at cell X senses it's neighbourhood and finds that there are 3 blocked cells and none of them are confirmed. So, we represent this piece of information with following equation : a+b+c+d+e+f+g+h = 3. So, the set defined above will contain tuples for the cells a to h and the value contained in the constraint list will be 3, where each variable in the equation corresponds to a neighbour of X. This is a basic example of the data structure and representation of a constraint for Agent 4.

- **Knowledge Base:** The Knowledge Base for our agent is a list of such constraints and each constraint represents a piece of information that can be used by Agent 4 to derive inferences. We used a *list* data structure to store all the constraints to create a knowledge base.

- We maintain a *dictionary* that maps a variable to a list of constraints that the variable is involved in. This dictionary is used to find the most constrained variable. The most constrained variable is of utmost significance because **it makes inconsistent assignments fail earlier in the search, which enables more efficient pruning.**

- Each variable is mapped to a list of constraints that it is involved in. The most constrained variable is the one that is involved in the maximum number of constraints.

- The cells that are at the borders of the maze will be involved in lesser number of constraints as compared to the ones that are towards the centre of the maze. This will lead to a bias in selecting the most constrained variable. In order to overcome this bias we normalize the value of the number of constraints that a variable is involved in by the length of the list of constraints for that variable. We also add an element of randomization in selection when there are ties, for instance - If we have five variables that have the same normalized value then we randomly select one of these five to avoid picking the same variable again and again.

- **Backtracking Search:** We make use of Backtracking Search to solve our Constraint Satisfaction Problem. Backtracking Search is a depth-first search that chooses values (values can be either 0 or 1) for one variable at a time and backtracks when a variable has no legal values left to assign. Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then backtrack returns failure, causing the previous call to try another value. See appendix B.

- Since the dimension of the grid is 101 x 101 , the number of variables is in the order of 10^4. And as we know that the time complexity of Backtracking Search is of exponential order, it will not be feasible to backtrack and solve for all of the variables hence, we have set a time limit of 0.5 seconds for deriving inferences from the list of most constraint variables. This list contains most constrained variables in a decreasing order of their involvement in constraints. See appendix C.

- The result of Backtracking Search is stored in a dictionary of dictionaries. The dictionary maps values for each variable. Following is an example of a dictionary containing two variables a and b - {a: {0: number of times the constraints were satisfied when a's value was equal to 0, 1: number of times the constraints were satisfied when a's value was equal to 1 }, b: {0: number of times the constraints were satisfied when b's value was equal to 0, 1: number of times the constraints were satisfied when b's value was equal to 1 }}.

- **Computational Optimization:** Suppose during the Backtracking Search we see that the constraints are never satisfied when the value of variable 'a' is 1 but the constraints are satisfied say 'k' number of times when the value of variable 'a' is 0. In such a case, we prune all those variable assignment sub-trees that branch out from the edge having the value of variable 'a' as 1. This essentially means that if for a particular value of a variable we know that there is going to be a constraint violation later on in the variable assignment tree then we prune all such sub-trees. This saves us from carrying out a lot unnecessary computation.

- **Note :** An interesting thing to note here is that even with all the tools we are giving to Agent 4, there might still be some inferences that Agent 4 cannot make. For instance currently we are using backtracking search on our most constrained variable repeatedly for up to 0.5 seconds to be able to infer its status. We are performing this search on the

most constrained variable because this gives us the highest chance to infer something. A problem arises if variables that are not the most constrained variable are actually inferable but we may never infer them since we do not run the backtracking search for every possible variable past the 0.5 seconds time limit.

- **Note :** Another thing to note would be that Agent 3 will never be able to infer something that Agent 4 does not because of the fact that we have built Agent 4 on top of the functionalities of Agent 3. So Agent 4 will always be able to infer the things that Agent 3 already does.

- **Note :** We want to bring to attention that the maximum number of variables possible in a single set . As we can see from Figure 1.15 and Figure 1.16, x_1 can be in a maximum of 8 constraints which contains a maximum of 25 variables. But since we are assuming that the agent has reached x_1, we must have confirmed the status of a minimum of 3 cells. Hence the maximum number of variables possible will be $25 - 3 = 22$

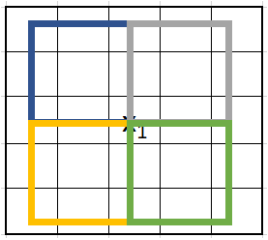

Figure 1.15: 4 Constraints containing x_1

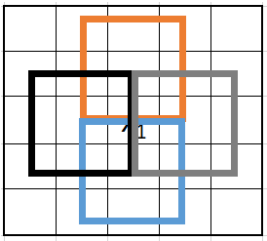

Figure 1.16: 4 more constraints containing x_1

1.3.4 Implementation details

- **OurOwnInferenceAgent** class also inherits the Agent base class.

- We had to add two new member variables to the Agent class namely **knowledge_base** of type *List* and **variable_to_constraint_dict** of type *Dictionary*.

- In the **helper** class we added the method **backtracking_search** which contains the implementation of the Backtracking Search algorithm.

- In the **helper** class we also added the **make_inference_from_most_constraint_variable** method which uses the **backtracking_search** method to derive inferences starting from the most constrained variable. See appendix B, C.

Chapter 2

Analysis

2.1 Agent 1,2 and 3

Here we want to compare the performance of Agent 3 against the performance of Agent 1 and 2. Below are the metrics and results of these comparisons. We tested on 101 probabilities between 0.0 and 0.33 and ran 100 graphs on each probability.

2.1.1 Length of Trajectory

We compared the length of the explored nodes returned by our 3 agents. We also define explored nodes as the length of the path that our agents have taken on the way to the goal. Our results are as follows -

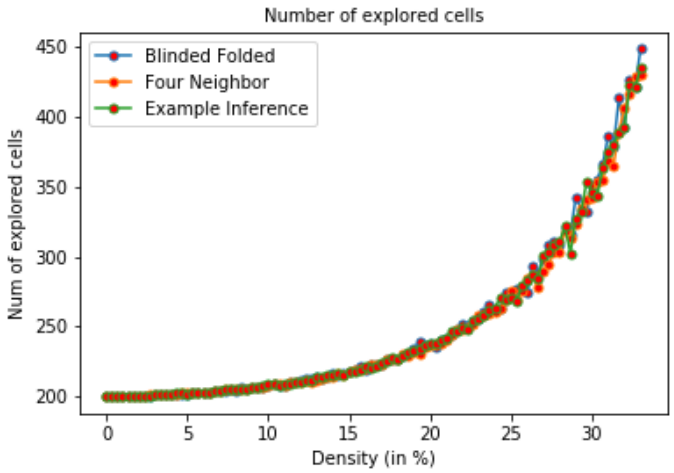

Figure 2.1: Length of Trajectory

As we can see that the length of trajectory does not differ between our agents mainly because

all of our agents still use repeated forward A* search. The amount of information gained by each agent might be different, but the A* algorithm consistently gives our agents the shortest path to goal from their current position.

2.1.2 Length of Trajectory / Length of Shortest Path in Final Discovered Grid World

We compared the average of (Length of Trajectory/Length of Shortest Path in Final Discovered Grid World) returned by our 3 agents. We get the following graph as a result -

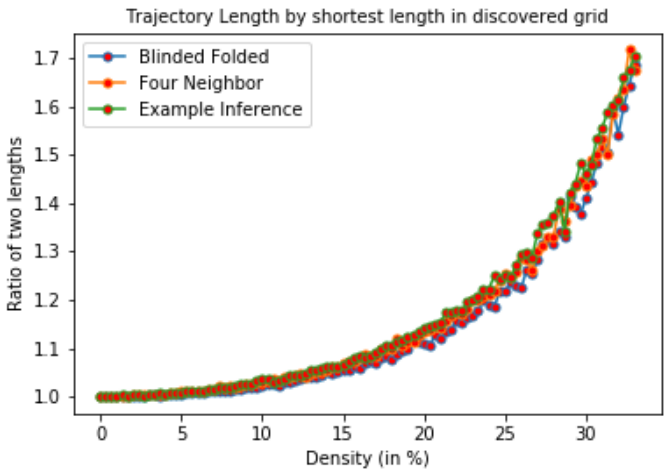

Figure 2.2: Average(Length of Trajectory/Shortest Path in Final)

The ratios are fairly equal for all 3 agents. But we see that since the length of trajectory for all 3 agents is about the same, the length of shortest path in the final discovered grid world must be shortest for Agent 3 by a small margin. This is an expected result since the final discovered grid world for Agent 3 is larger than the final discovered grid world for Agent 1 and 2. And thus the shortest path in the final discovered grid world on average will be shortest for Agent 3 by a small margin.

2.1.3 Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World

We compared the length of the shortest path in discovered grid world / shortest path in full grid world returned by our 3 agents. We get the following graph as a result -

We see that Agent 3 has the lowest ratio of shortest path in discovered/full grid world by a small margin. We know that the shortest path in the full grid world will on average be equal

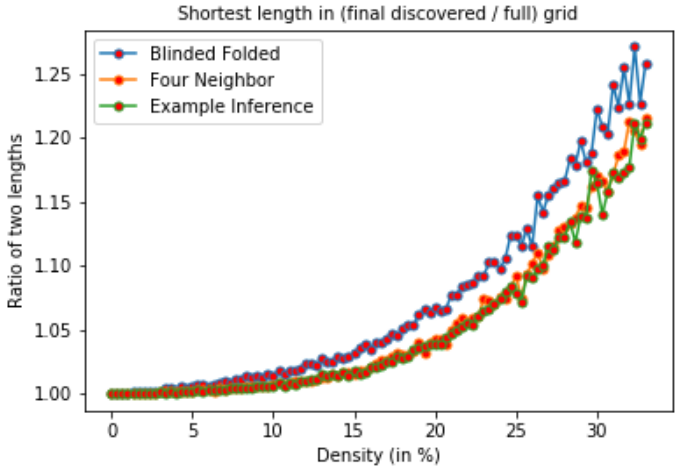

Figure 2.3: Length of Shortest Path in Discovered / Full Grid World

for all 3 agents so a lower ratio must mean that Agent 3 has the shortest path in discovered grid world on average. This is again because of the fact that Agent 3 discovers more of the grid than Agent 1 and 2 because of the extra inference that it gets to propagate through the grid world.

Also we can see that the ratio is almost the same for Agent 2 and 3. This is because as we can see in one of our later comparison metrics (section 2.1.5) the number of confirmed cells (and in turn the discovered grid world) becomes equal for Agent 2 and 3 for higher densities.

2.1.4 Number of Processed cells / Number of A* search calls

Here we define "Processed cells" as the number of cells that are popped off the fringe during our A* search. The Number of Processed cells is directly proportional to the number of A* search calls that the agent makes during its run-time -

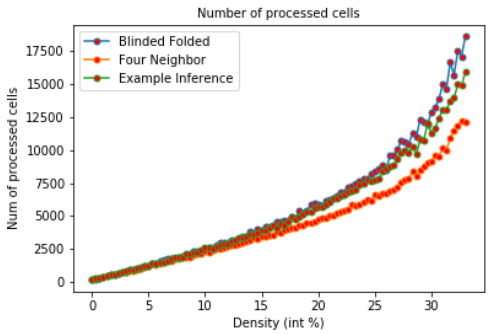

Figure 2.4: Number of Processed cells

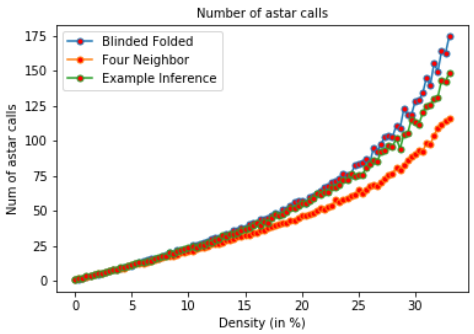

Figure 2.5: Number of A* search calls

As we can see The blindfolded agent processes the most number of cells because of its severely limited field of view. The blindfolded agent has less information about its surroundings and thus

gets blocked multiple times on the way to the goal causing it to call A* an increased number of times.

The example inference agent performs worse than the four neighbor agent mainly because the limited inference of Agent 3 is not enough to help the A* search algorithm pick a path that doesn't contain blocks. This makes it so the agent gets blocked more and thus has to run A* search again.

On the other hand it performs better than the blindfolded agent because even though Agent 3 itself is "blindfolded", it can use its limited inference to plan a path to goal that does not contain blocks and thus on an average will call A* fewer times than the blindfolded agent.

2.1.5 Number of Confirmed Cells

Here we wanted to compare the number of cells in our grid that were confirmed as blocked or unblocked after each of our agents reaches the goal -

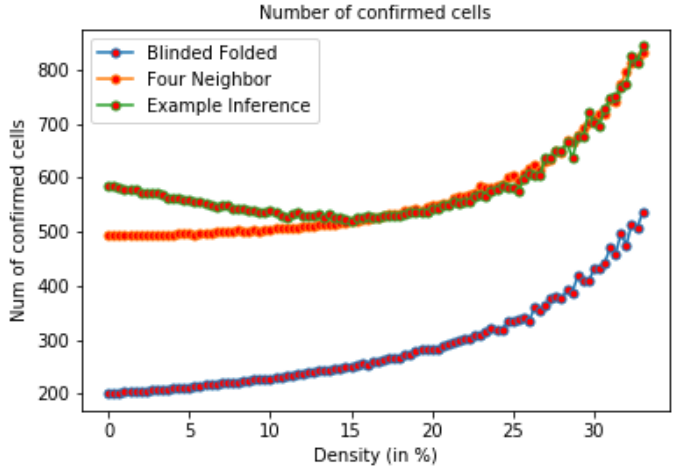

Figure 2.6: Number of Confirmed Cells

As we can clearly see, Agent 3 and 2 far outclass Agent 1 in terms of the number of cells they can confirm as blocked or unblocked. Agent 2 and 3 both discover a lot of the grid world because of their confirmed vision and inference respectively.

An interesting thing to note here is that Agent 3 confirms the status of a lot more cells than the four neighbor agent at low probabilities. This is because a lot of the inference by agent 3 in the low probabilities can be resolved because of the fact that there are very few blocks and thus Agent 3 can confirm almost immediately that the neighbors of most of the cells that it explores are unblocked.

At higher probabilities the number of confirmed cells becomes almost equal for Agent 2 and 3 because it becomes harder for Agent 3 to solve some inferences whereas Agent 2 uses its superior vision to confirm the status of multiple cells.

2.1.6 Number of Times Bumped into Blocked Cell

Here we wanted to calculate the number of times one of our Agents actually bumped into a blocked cell before inferring a blocked cell in its path or in the case of the 4 Neighbor agent before "seeing" a block in its path.

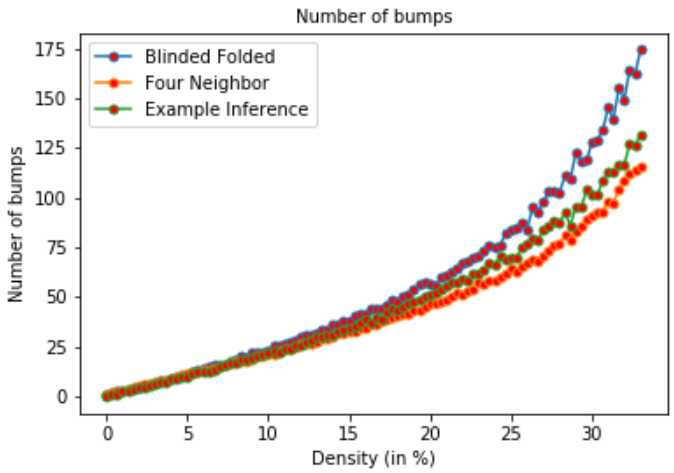

Figure 2.7: Number of bumps into blocked cells

As we can see the blindfolded agent performs the worst in this regard since it must bump into a blocked cell before re-planning. The 4 neighbor agent performs the best because it does not bump into blocked cells that are already in its field of view. Agent 3 performs better than the blindfolded agent because it can sometimes infer blocks in its path and then does not bump into them. It performs worse than the 4 Neighbor agent because Agent 3 is considered blindfolded other than the inference that it can make.

2.1.7 Running Time

We compared the Running time of our 3 agents. We get the following graph as a result 2.8-

As we can see Agent 3 performs the worst in terms of total running time mainly because of the inference that it must calculate and propagate every time our Agent moves on top of the normal A* calls that it must do.

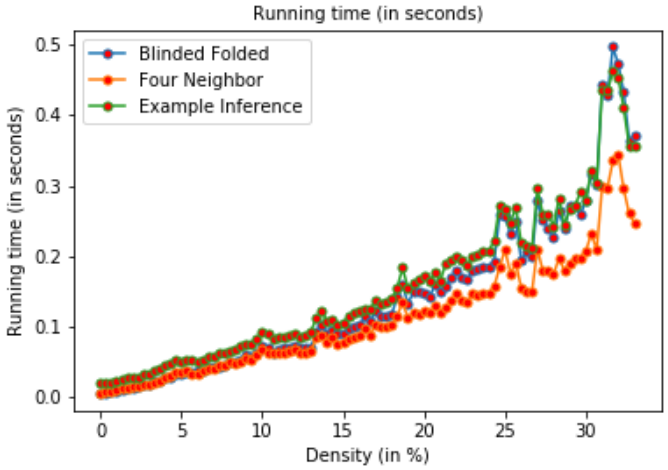

Figure 2.8: Running Time(seconds)

2.2 Agent 1,2,3 and 4

Here we want to compare the performance of Agent 4 against the performance of Agent 1,2 and 3. Below are the metrics and results of these comparisons. We tested on 70 probabilities between 0.0 and 0.33 and ran 70 graphs on each probability.

2.2.1 Length of Trajectory

We compared the length of the final path returned by our 4 agents. We get the following graph as a result -

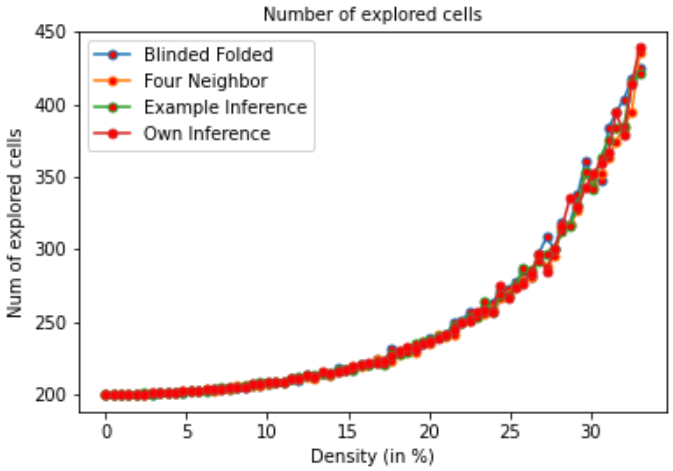

Figure 2.9: Length of Trajectory

As we can see that the length of trajectory does not differ between our agents because of the reasons mentioned in section 2.1.1

2.2.2 Length of Trajectory / Length of Shortest Path in Final Discovered Grid World

We compared the average of (Length of Trajectory/Length of Shortest Path in Final Discovered Grid World) returned by our 4 agents. We get the following graph as a result -

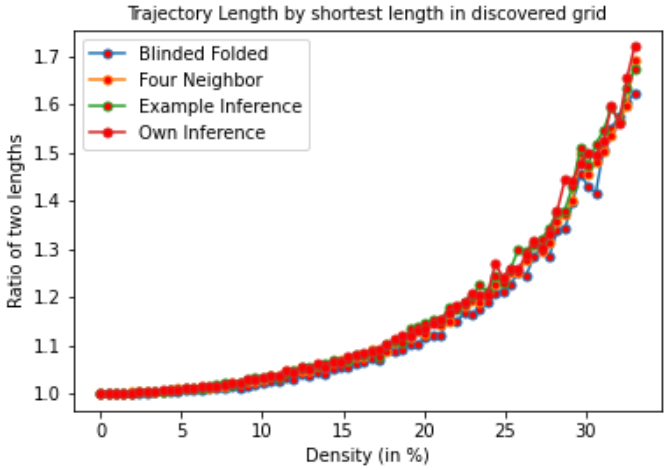

Figure 2.10: Average(Length of Trajectory/Shortest Path in Final)

Similar to the Agent 1,2 and 3 comparison, we see that Agent 4 barely outperforms Agent 3 because even though its inference strategy is better, the shortest path in the discovered grid world does not change by much. This makes Agent 4 only slightly better than Agent 3.

2.2.3 Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World

We compared the length of the shortest path in discovered grid world / shortest path in full grid world returned by our 4 agents. We get the following graph as a result -

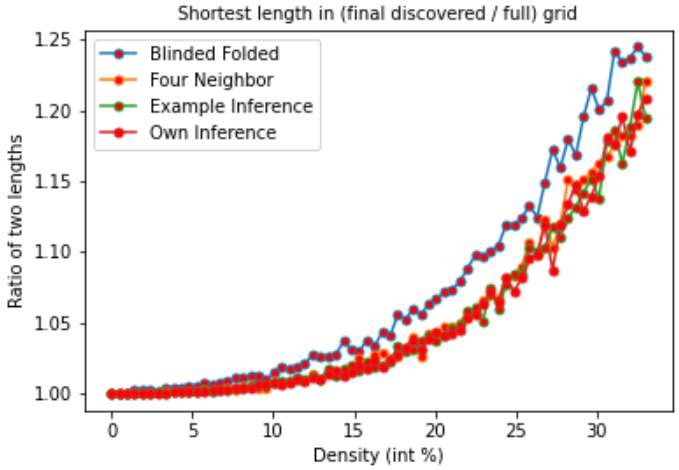

Figure 2.11: Length of Shortest Path in Discovered / Full Grid World

Here we can see that Agent 4 very slightly outperforms Agent 3 because as we talked about in the 3 Agent comparison, this ratio decreases as our agent discovers more of the grid world.

2.2.4 Number of Processed cells / Number of A* search calls

Here we compare the Number of Processed cells and A* calls between Agents 1,2,3 and 4 -

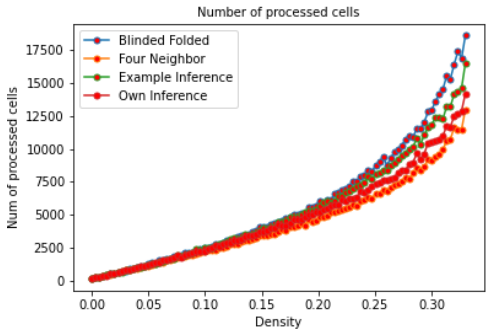

Figure 2.12: Number of Processed cells

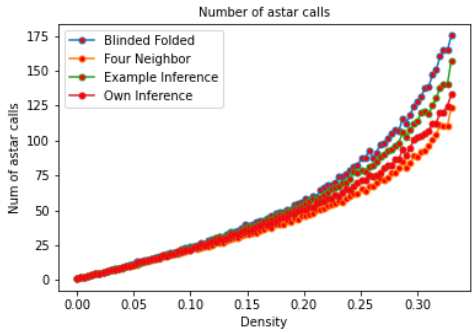

Figure 2.13: Number of A* search calls

We can see that Agent 4 performs better than Agent 3 because of its increased power of inference but is still outperformed in this particular metric by the four neighbor agent; albeit slightly. This is again because of the fact agent 4 encounters more blocks on its path to the goal and thus must call A* search more times to be able to reach the goal.

2.2.5 Number of Confirmed Cells

Here we wanted to compare how many cells of our grid we can confirm as blocked or unblocked after each of our agents reaches the goal. Basically to see how much of the grid world we can

discover -

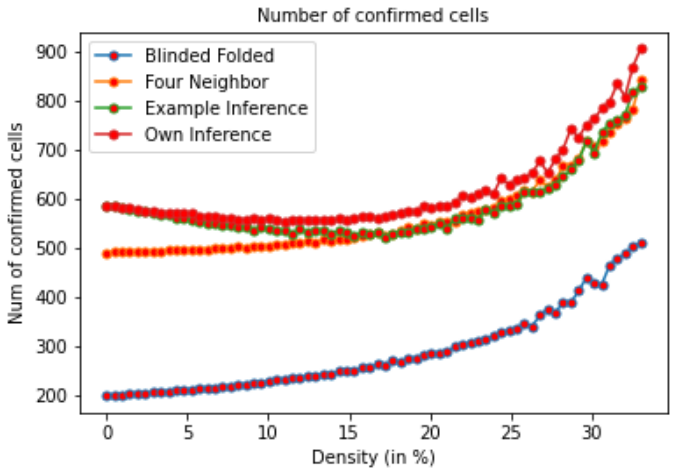

Figure 2.14: Number of Confirmed Cells

Here we see how Agent 4's inference helps it confirm more cells than Agent 3. This is because of the multi cell and advanced inference strategy being used by Agent 4 and which is missing in Agent 3.

2.2.6 Difference in Number of Times Bumped into Blocked Cell

Here we wanted to show the difference in number of times agent 4 bumps into cells when compared to all other agents. We used the difference instead of the total number of bumps to illustrate clearly how Agent 4 is performing better than Agent 3. A negative difference shows that Agent 4 bumped into fewer blocked cells than other Agents.

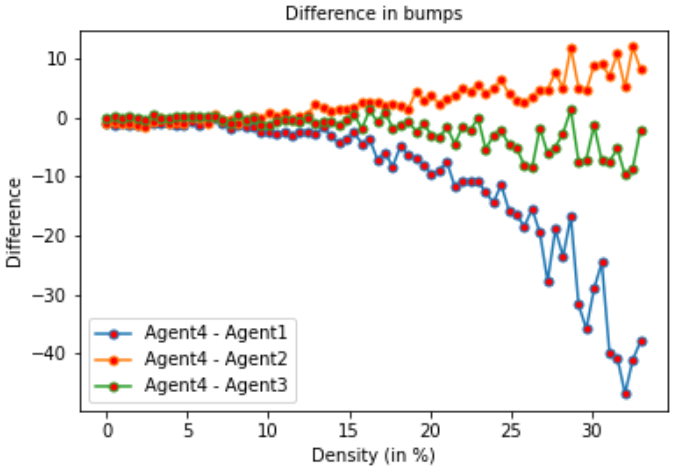

Figure 2.15: Difference in Number of times Bumped into Blocked Cell

We see that our Agent 4 bumps into lesser blocks when compared to Agent 3 and Agent 1 because of the superior inference strategy that allows it to infer some of the blocks that are in its planned path. It still performs slightly worse than Agent 2.

2.2.7 Number of Early Terminations

For Agent 1 and 2 we made sure to only call Repeated A* when we encountered a block in our path. For Agents 3 and 4 we made the calls even if our inference told us that there was a blocked cell in our planned path, we called Repeated A* immediately instead of waiting till we get blocked. We plotted the average number of times our Agents Terminated the Execution stage early versus density and got the following results -

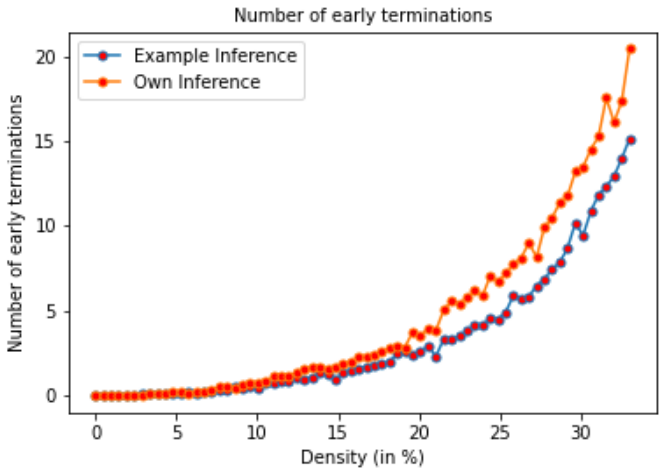

Figure 2.16: Number of Early Terminations

As we can see, Agent 3 can detect blocks in its planned path quite a few times and consequently calls Repeated A* early armed with the knowledge of the block. But agent 3's capability to infer blocks in its path is dwarfed by the number of times Agent 4's inference helps us discover a block in its path as is evident from the graph.

2.2.8 Running Time

We compared the Running time of our 4 agents. We get the following graph as a result -

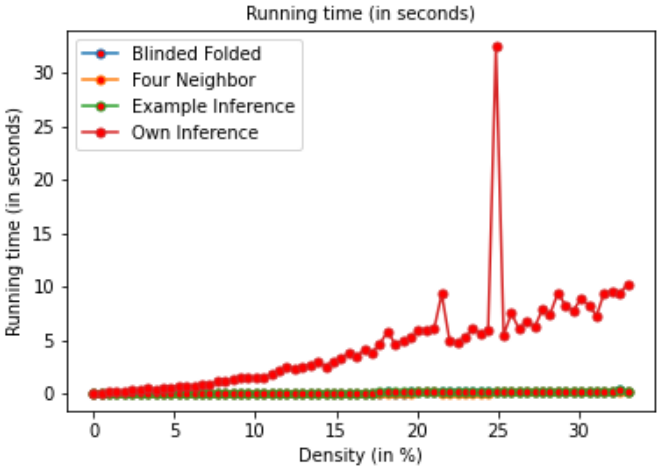

Figure 2.17: Running Time in seconds

Agent 4 takes by far the most amount of time to run because of the high number of calculations it must do to calculate inference for the grid each time the agent moves.

Chapter 3

Agent 5

3.1 Design

Agent 5 has been built on top of Agent 4. The inference strategy for Agent 5 is the same as that of Agent 4 only the planning phase has been improved to utilise the information from the knowledge base so that a better path can be planned. Listed below are the details of the changes that we have made in the planning phase for Agent 5:

- In the Agents 1 to 4, during A* search we were popping the minimum element from the fringe based on the following order: f(n) > g(n) > h(n). If these three values were equal, we randomly popped one of the equal elements from the fringe.
- Now, in order to improve the planning of Agent 5 we are popping elements from the fringe based on the following order: f(n) > Probability of the node being blocked > g(n) > h(n). If still there is a tie between elements, then we randomly pop one of the equal valued nodes.
- We define the probability of a node X being blocked as follows:
  - We check the neighbourhood of X.
  - If X has cells in the neighbourhood that have been visited then choose the neighbour that has the least number of hidden neighbours. Let's call this node X'.

– Then we calculate the probability of X being blocked as

$$Probability(blocked) = \frac{C(X') - B(X')}{H(X')}$$

where C(X') denotes the number of sensed blocks in the neighbourhood of node X', B(X') denotes the number of confirmed blocks in the neighbourhood of node X' and H(X') is the number of hidden neighbours of node X'.

– If X does not have any visited cells in it's neighbourhood then we define the probability of X being blocked as

$$Probability(blocked) = \frac{\text{Number of confirmed blocks in the grid}}{\text{Number of confirmed nodes in the grid}}$$

3.2 Implementation

In the planning phase of Agent 5 since we are taking into consideration the probability of a cell being blocked. We had to add the following methods:

- **compute_probability_of_being_block_for_each_cell_after_execution:** This method calculates the probability of a cell being blocked making use of the calculation described above.
- **reset_probability_of_being_block_for_each_cell_after_execution:** This is a utility method used to reset the previously calculated probability for each cell.
- Apart from the methods listed above, as Agent 5 has been built on top of Agent 4 it makes use of the methods defined for Agent 4.

3.3 Analysis of Performance

Here we want to compare the performance of Agent 5 against the performance of Agents 1,2,3 and 4. Below are the metrics and results of these comparisons. We tested on 34 probabilities between 0.0 and 0.33 and ran 50 graphs on each probability.

3.3.1 Length of Trajectory

We compared the length of the final path returned by our 5 agents. We get the following graph as a result -

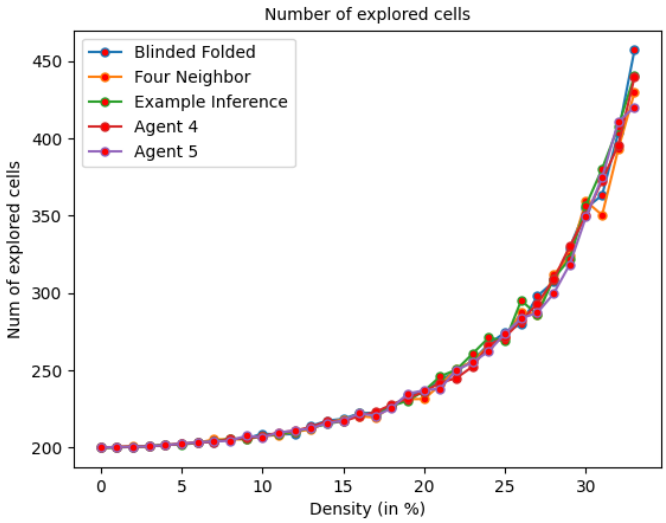

Figure 3.1: Length of Trajectory

As we can see that the length of trajectory does not differ between our agents because of the reasons mentioned in section 2.1.1

3.3.2 Length of Trajectory / Length of Shortest Path in Final Discovered Grid World

We compared the average of (Length of Trajectory/Length of Shortest Path in Final Discovered Grid World) returned by our 5 agents. We get the following graph as a result -

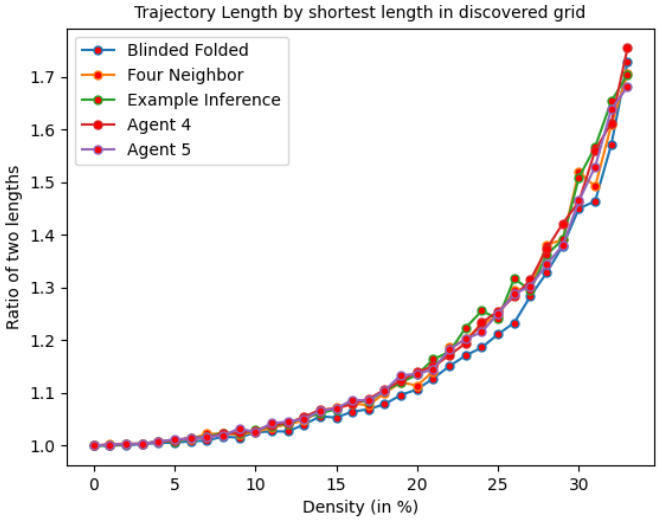

Figure 3.2: Average(Length of Trajectory/Shortest Path in Discovered Grid World)

We can see that Agent 5 performs just barely better than Agent 4 in this metric. Agent 5 returns a shortest path in discovered grid world that is almost equal to the shortest path in discovered grid world returned by Agent 4 this is in spite of having a larger overall discovered grid world. Not all of the extra inference done by Agent 5 helps in giving a shorter path to goal thus giving Agent 4 and 5 a similar ratio.

3.3.3 Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World

We compared the length of the shortest path in discovered grid world / shortest path in full grid world returned by our 4 agents. We get the following graph as a result -

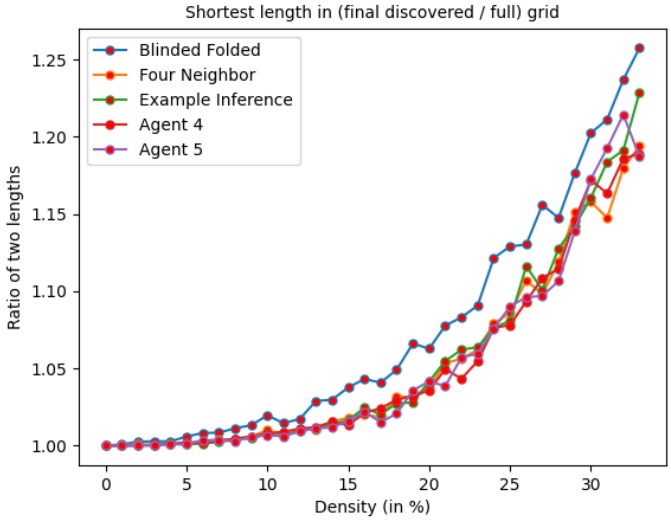

Figure 3.3: Length of Shortest Path in Discovered / Full Grid World

Here we see that Agent 5 again performs about the same as Agent 4 since the discovered grid world of both are around the same size.

3.3.4 Number of Processed cells / Number of A* search calls

Here we compare the Number of Processed cells and A* calls between Agents 1,2,3,4 and 5 -

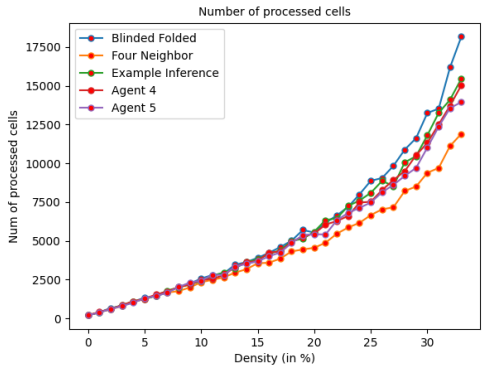

Figure 3.4: Number of Processed cells

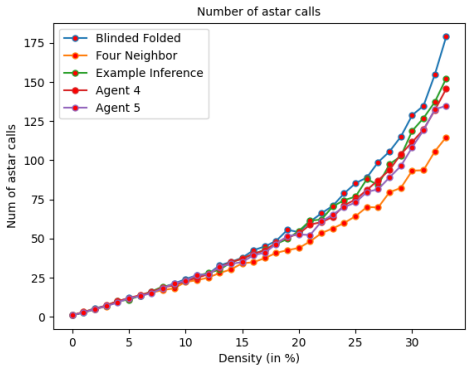

Figure 3.5: Number of A* search calls

We can see that Agent 5 performs slightly better than Agent 4 in this metric. This is mainly because of the probabilistic calculations that we use for Agent 5. This helps Agent 5 use A* search to plan a path that has a low probability of containing blocks. **The limiting factor for Agent 5** is that these probabilistic calculations make use of sensed information of a particular cell and thus this planning doesn't work throughout the grid in areas that we have not visited or inferred anything. This is why we just see a slight improvement over Agent 4 even though Agent 5 has a superior planning strategy.

3.3.5 Number of Confirmed Cells

Here we wanted to compare the number of cells in the grid that can be confirmed as blocked or unblocked after each of our agents reaches the goal -

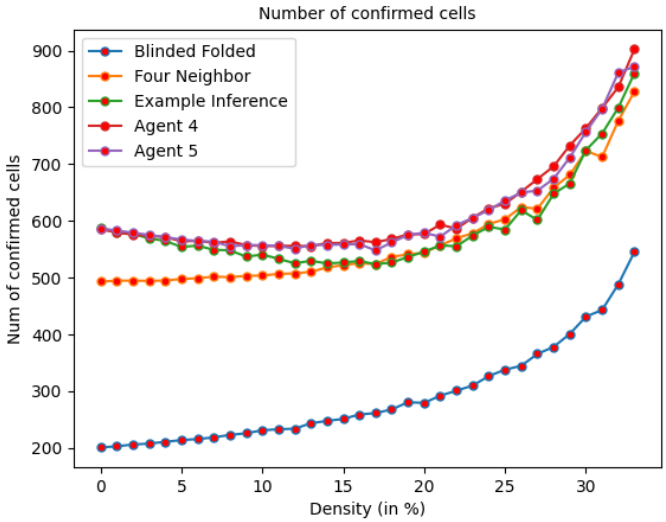

Figure 3.6: Number of Confirmed Cells

Here we see that Agent 5 performs no better than Agent 4 mainly because it is using the same inference strategy as Agent 4 and thus will not show much improvement over it.

3.3.6 Number of Times Bumped into Blocked Cell

Here we wanted to calculate the number of times one of our Agents actually bumped into a blocked cell before inferring a blocked cell in its path or in the case of the 4 Neighbor agent before ”seeing” a block in its path.

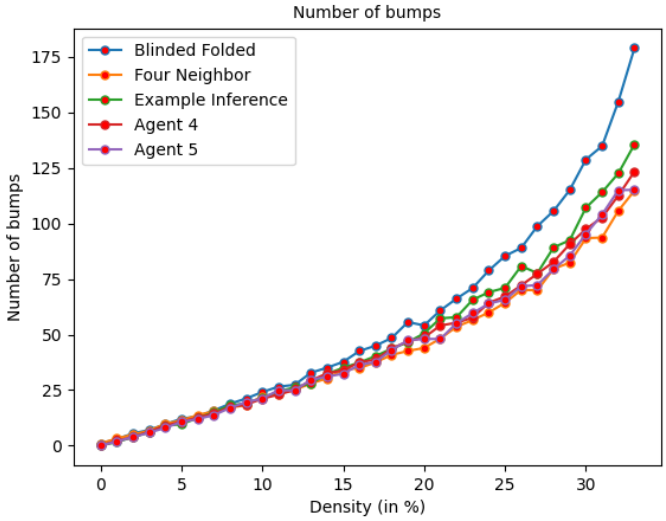

Figure 3.7: Number of Bumps

Here we see that Agent 5 performs slightly better than Agent 4 because it is able to take a path to goal that contains fewer blocks because of its improved planning step and thus it bumps into blocks fewer times than Agent 4. We only see a slight improvement because we can only compute the probability of cells that are neighbors of visited cells. Because of this we cannot compute the probability of far away cells during our planning stage and thus can only slightly reduce the number of times we bump into a blocked cell.

3.4 Discussions of Shortcomings and Optimality

The main shortcoming of our algorithm is the fact that when we calculate the probability of a cell being blocked or being empty, we do the calculation only for the neighbors of a visited cell. Thus our calculations for probability are highly localized and not global on the grid world which makes our solution a local optimum and not a global optimum. Our algorithm also does not tell us about the probability of cells that are not near visited cells which limits its usefulness.

Our algorithm is also not optimal because we do not consider the effect multiple visited cells have on the probability of a single cell. In the future we would like to add a way to combine the information of multiple cells to be able to predict with a higher degree of confidence about the probability of a given cell.

There are also situations where our algorithm plans poorly for example, from the current cell our algorithm will encourage us to go to a cell with a low probability of blocks, but this cell

might lead us to get blocked more in the future whereas there might have been a cell we can choose from our current position that has a higher probability of being a block but later on lets us avoid a large number of blocks.

Appendix A

Inference Code[1]

```
def find_block_while_inference(maze: list, current_position: tuple, full_maze:
                                    np.array, entire_trajectory_nodes=None,

                                    want_to_use_one_node_inference_strategy
                                    : bool = True,
                                    want_to_use_two_node_inference_strategy
                                    : bool = False,
                                    want_to_use_three_node_inference_strategy
                                    : bool = False,
                                    want_to_use_most_constraint_
variable_for_backtracking_search: bool = False,
                                    want_to_use_probability_approach=True,
                                    knowledge_base=None,
                                    variable_to_constraint_dict=None,
                                    list_of_variables_in_the_path=None):
    """
    This function helps us find blocks in the path if they exist while also
                                    performing inference.
    :param maze: This agents copy of the maze.
    :param current_position: This is the current position of the agent.
    :param full_maze: This is the full maze.
    :param entire_trajectory_nodes: This is the set of nodes that are in the
                                    agents current trajectory.
```

[1]We have only put code for the inference function here, please refer to the attached winrar file for any dependant code or for the rest of the code

```
    :param want_to_use_one_node_inference_strategy: This parameter lets us set
                                        if we want use Agent 3's inference
                                        strategy.
    :param want_to_use_two_node_inference_strategy: This parameter lets us set
                                        if we want use Agent 4's 2 cell
                                        inference strategy.
    :param want_to_use_three_node_inference_strategy: This parameter lets us
                                        set if we want use Agent 4's 3 cell
                                        inference strategy.
    :param want_to_use_most_constraint_variable_for_backtracking_search: This
                                        parameter lets us set if we want use
                                        Agent 4's Advanced inference strategy.
    :param want_to_use_probability_approach: This parameter lets us decide how
                                        we want to get the most constrained
                                        variable.
    :param knowledge_base: This is the knowledge base of the Agent
    :param variable_to_constraint_dict: This dictionary maps the variables to
                                        constraints.
    :param list_of_variables_in_the_path: This list stores the variables that
                                        are currently in our path.
    :return: True if there is a block in path else false.
    """
    # We initialize parameters if we get them as None.
    if knowledge_base is None:
        knowledge_base = list()

    if entire_trajectory_nodes is None:
        entire_trajectory_nodes = set()

    if variable_to_constraint_dict is None:
        variable_to_constraint_dict = dict()

    if list_of_variables_in_the_path is None:
        list_of_variables_in_the_path = list()

    # We create this queue to help us store all the variables that are going to
                                         be confirmed and subsequently store
                                        those cells in a set so that we can
                                        access them in constant time.
```

```
    inference_items = Queue()
    items_in_the_queue = set()
    is_block_node_in_current_path = False

    # We put the current position of the agent into the Queue and set.
    inference_items.put(current_position)
    items_in_the_queue.add(current_position)

    # We run this while loop until the queue is empty and there is nothing left
                                        to be inferred.
    while not inference_items.empty():
        while not inference_items.empty():
            current_node = inference_items.get()
            items_in_the_queue.remove(current_node)

            # If the current node is not set as confirmed in the maze then we
                                        set it to confirmed and made updates
                                        accordingly
            if not maze[current_node[0]][current_node[1]].is_confirmed:

                maze[current_node[0]][current_node[1]].is_confirmed = True
                current_node_val = 0

                if full_maze[current_node[0]][current_node[1]] == 1:
                    maze[current_node[0]][current_node[1]].is_blocked = True
                    current_node_val = 1

                    # If current node is in the trajectory then we should
                                        return True for this function
                    if current_node in entire_trajectory_nodes:
                                        is_block_node_in_current_path = True
                else:
                    maze[current_node[0]][current_node[1]].is_blocked = False

                # Remove this confirmed cell from the knowledge base
                remove_variable_from_knowledge_base(knowledge_base,
                                        variable_to_constraint_dict,
                                        current_node, current_node_val)

```

```
65                  # Iterate over eight neighbors, update their status and add it
                                into the queue
66                  for neighbor in maze[current_node[0]][current_node[1]].
                                eight_neighbors:
67                      if not maze[neighbor[0]][neighbor[1]].is_visited:
68                          continue
69                      if maze[current_node[0]][current_node[1]].is_blocked: maze[
                                neighbor[0]][neighbor[1]].
                                num_confirmed_blocked += 1 else:
70                          maze[neighbor[0]][neighbor[1]].num_confirmed_unblocked
                                += 1
71
72                      if not (neighbor in items_in_the_queue):
73                          items_in_the_queue.add(neighbor)
74                          inference_items.put(neighbor)
75
76              # If the current cell is visited, we can inference rules to them
77              if maze[current_node[0]][current_node[1]].is_visited:
78
79                  # This rule applies to the 3rd as well as the 4th agent where
                                we are trying to infer using only one
                                cell's inference
80                  if want_to_use_one_node_inference_strategy:
81                      if can_infer(maze[current_node[0]][current_node[1]].
                                num_sensed_blocked,
82                                   maze[current_node[0]][current_node[1]].
                                num_confirmed_blocked,
83                                   maze[current_node[0]][current_node[1]].
                                num_sensed_unblocked,
84                                   maze[current_node[0]][current_node[1]].
                                num_confirmed_unblocked):
85
86                          for neighbor in maze[current_node[0]][current_node[1]].
                                eight_neighbors:
87                              if (neighbor not in items_in_the_queue) and (not
                                maze[neighbor[0]][neighbor[1]].
                                is_confirmed):
88                                  items_in_the_queue.add(neighbor)
89                                  inference_items.put(neighbor)
```

```
90
91              # Three cells inference strategy
92              if want_to_use_three_node_inference_strategy:
93
94                  # Iterating over all possible mandatory neighbors
95                  for index in range(len
96                  (RELATIVE_POSITION_OF_TWO_MANDATORY_NEIGHBORS)):
97
98                      # Fetching relative position of mandatory neighbors
99                      two_mandatory_neighbors =
100                     RELATIVE_POSITION_OF_TWO_MANDATORY_NEIGHBORS[index]
101                     num_of_neighbors = 0
102
103                     # Check precondition of mandatory neighbors
104                     for relative_position in two_mandatory_neighbors:
105                         neighbor = (current_node[0] + relative_position[0],
                                        current_node[1] + relative_position[1]
                                        )
106                         if check(neighbor) and maze[neighbor[0]][neighbor[1
                                    ]].is_visited:
107                             num_of_neighbors += 1
108
109                     # Check whether you found two neighbors or not
110                     if num_of_neighbors == 2:
111                         num_sensed_blocked = maze[current_node[0]][
                                    current_node[1]].num_sensed_blocked
112                         num_confirmed_blocked = maze[current_node[0]][
                                    current_node[1]].num_confirmed_blocked
113                         num_sensed_unblocked = maze[current_node[0]][
                                    current_node[1]].num_sensed_unblocked
114                         num_confirmed_unblocked = maze[current_node[0]][
                                    current_node[1]].
                                    num_confirmed_unblocked
115
116                         # Current node should not be blocked
117                         if maze[current_node[0]][current_node[1]].
                                    is_blocked:
118                             num_confirmed_blocked += 1
119                             num_sensed_blocked += 1
```

```
120                             assert False
121                         else:
122                             num_confirmed_unblocked += 1
123                             num_sensed_unblocked += 1
124
125                         # Iterate over two mandatory neighbors and compute
                                necessary things
126                         for ind in range(len(two_mandatory_neighbors)):
127                             relative_position = two_mandatory_neighbors[ind
                                    ]
128                             neighbor = ( current_node[0] +
                                    relative_position[0], current_node[1] +
                                     relative_position[1])
129
130                             # Factor is used to subtract the middle element
131                             if ind == 0:
132                                 factor = -1
133                             else:
134                                 factor = 1
135                             num_sensed_blocked += factor * maze[neighbor[0]
                                    ][neighbor[1]].num_sensed_blocked
136                             num_confirmed_blocked += factor * maze[neighbor
                                    [0]][neighbor[1]].num_confirmed_blocked
137                             num_sensed_unblocked += factor * maze[neighbor[
                                    0]][neighbor[1]].num_sensed_unblocked
138                             num_confirmed_unblocked += factor * maze[
                                    neighbor[0]][neighbor[1]].
                                    num_confirmed_unblocked
139
140                             if maze[neighbor[0]][neighbor[1]].is_blocked:
141                                 num_confirmed_blocked += factor
142                                 num_sensed_blocked += factor
143                                 assert False
144                             else:
145                                 num_confirmed_unblocked += factor
146                                 num_sensed_unblocked += factor
147
148                         # Check whether we can infer from the current set
                                of knowledge
```

```
149                         if can_infer(num_sensed_blocked,
                                         num_confirmed_blocked,
                                         num_sensed_unblocked,
150                                       num_confirmed_unblocked):
151
152                             for relative_position in
                                         RELATIVE_POSITION_OF_TWO_SENSED_NEIGHBORS
                                         [index]:
153                             neighbor = (
154                                 current_node[0] + relative_position[0],
                                         current_node[1] + relative_position[1])
155
156                                 if check(neighbor) and (neighbor not in
                                         items_in_the_queue) and (not maze[
                                         neighbor[0]][neighbor[1]].is_confirmed)
                                         :
157                                         items_in_the_queue.add(neighbor)
158                                         inference_items.put(neighbor)
159
160                         # Otherwise we will check the ambiguity and add it
                                 in our knowledge base
161                         elif is_ambiguous(num_sensed_blocked,
                                         num_confirmed_blocked,
                                         num_sensed_unblocked,
                                         num_confirmed_unblocked):
162
163                             # Initialise some variables
164                             unconfirmed_cells = set()
165                             num_blocked_cells_in_unconfirmed_cells = 0
166
167                             # Compute relative position and create one set
                                         for that
168                             for relative_position in
                                         RELATIVE_POSITION_OF_TWO_SENSED_NEIGHBORS
                                         [index]:
169                                 neighbor = (
170                                         current_node[0] + relative_position[0],
                                          current_node[1] + relative_position[1]
                                         )
```

```
171                                 if check(neighbor) and (not maze[neighbor[0
                                        ]][neighbor[1]].is_confirmed):
172                                     unconfirmed_cells.add(neighbor)
173                                     num_blocked_cells_in_unconfirmed_cells
                                        += full_maze[neighbor[0]][neighbor[1]]
174
175                                 # write assert to check some conditions
176                                 assert num_blocked_cells_in_unconfirmed_cells =
                                        = num_sensed_blocked -
                                        num_confirmed_blocked
177                                 assert len(unconfirmed_cells) >=
                                        num_sensed_blocked -
                                        num_confirmed_blocked
178
179                                 # Add constraint to the knowledge
180                                 add_constraint_to_knowledge_base(knowledge_base
                                        , variable_to_constraint_dict,
                                        unconfirmed_cells,
                                        num_blocked_cells_in_unconfirmed_cells)
181
182             # If you want to use two cell inferences
183             if want_to_use_two_node_inference_strategy:
184
185                 # Iterate over four neighbors which are iterable
186                 for index in range(len(X)):
187                     neighbor = (current_node[0] + X[index], current_node[1]
                                        + Y[index])
188
189                     # Check for valid and visited neighbors
190                     if check(neighbor) and maze[neighbor[0]][neighbor[1]].
                                        is_visited:
191                         num_not_confirmed_cells = 0
192                         num_sensed_blocked = 0
193                         num_confirmed_blocked = 0
194                         num_sensed_unblocked = 0
195                         num_confirmed_unblocked = 0
196
197                         # Can comment the below two if and else
```

```
198                         if maze[current_node[0]][current_node[1]].
                                is_blocked:
199                             num_confirmed_blocked += 1
200                             num_sensed_blocked += 1
201                             assert False
202                         else:
203                             num_confirmed_unblocked += 1
204                             num_sensed_unblocked += 1
205
206                         if maze[neighbor[0]][neighbor[1]].is_blocked:
207                             num_confirmed_blocked -= 1
208                             num_sensed_blocked -= 1
209                             assert False
210                         else:
211                             num_confirmed_unblocked -= 1
212                             num_sensed_unblocked -= 1
213
214                         assert (num_confirmed_blocked == 0) and (
                                num_confirmed_unblocked == 0)
215
216                         # check the status of neighbors which has to be
                                confirmed
217                         for relative_position in
                                RELATIVE_POSITION_OF_NEIGHBORS_TO_CHECK
                                [index]:
218                             cell = (current_node[0] + relative_position[0],
                                 current_node[1] + relative_position[1]
                                )
219                             if check(cell):
220                                 if not maze[cell[0]][cell[1]].is_confirmed:
221                                     num_not_confirmed_cells += 1
222                                 elif maze[cell[0]][cell[1]].is_blocked:
223                                     num_confirmed_blocked += 1
224                                     num_sensed_blocked += 1
225                                 else:
226                                     num_confirmed_unblocked += 1
227                                     num_sensed_unblocked += 1
228
```

```
229                             # If number of not confirmed cells is zero, only
                                    then we should move ahead
230                             if num_not_confirmed_cells == 0:
231
232                                 # Do X2 - X1 for each attribute
233                                 num_sensed_blocked += maze[current_node[0]][
                                        current_node[1]].num_sensed_blocked -
                                        maze[neighbor[0]][neighbor[1]].
                                        num_sensed_blocked
234                                 num_confirmed_blocked += maze[current_node[0]][
                                        current_node[1]].num_confirmed_blocked
                                        - maze[neighbor[0]][neighbor[1]].
                                        num_confirmed_blocked
235                                 num_sensed_unblocked += maze[current_node[0]][
                                        current_node[1]].num_sensed_unblocked -
                                         maze[neighbor[0]][neighbor[1]].
                                        num_sensed_unblocked
236                                 num_confirmed_unblocked += maze[current_node[0]
                                        ][current_node[1]].
                                        num_confirmed_unblocked - maze[neighbor
                                        [0]][neighbor[1]].
                                        num_confirmed_unblocked
237
238                                 # Check the hypothesis
239                                 assert (num_sensed_blocked >=
                                        num_confirmed_blocked) and (
                                        num_sensed_unblocked >=
                                        num_confirmed_unblocked)
240
241                                 # Check whether we can infer anything or not
242                                 if can_infer(num_sensed_blocked,
                                        num_confirmed_blocked,
                                        num_sensed_unblocked,
                                        num_confirmed_unblocked):
243
244                                     for relative_position in
                                            RELATIVE_POSITION_OF_NEIGHBORS_TO_UPDATE
                                            [index]:
```

```
245                                 cell = (current_node[0] +
                                    relative_position[0],current_node[1] +
                                    relative_position[1])
246
247                                 # Update all unconfirmed cells into
                                    queue
248                                 if check(cell) and (cell not in
                                    items_in_the_queue) and (not maze[cell[
                                    0]][cell[1]].is_confirmed):
249                                     items_in_the_queue.add(cell)
250                                     inference_items.put(cell)
251
252                             # Check this to make sure we will add valid
                                things into our knowledge base
253                             elif is_ambiguous(num_sensed_blocked,
                                    num_confirmed_blocked,
                                    num_sensed_unblocked,
                                    num_confirmed_unblocked):
254
255                                 # Initialise attributes to store it in our
                                    knowledge base
256                                 unconfirmed_cells = set()
257                                 num_blocked_cells_in_unconfirmed_cells = 0
258                                 for relative_position in
                                    RELATIVE_POSITION_OF_NEIGHBORS_TO_UPDATE
                                    [index]:
259                                     cell = (current_node[0] +
                                    relative_position[0],current_node[1] +
                                    relative_position[1])
260                                     if check(cell) and (not maze[cell[0]][
                                    cell[1]].is_confirmed):
261                                         unconfirmed_cells.add(cell)
262
                                    num_blocked_cells_in_unconfirmed_cells
                                    += full_maze[cell[0]][cell[1]]
263
264                                 # Check some precondition
265                                 assert
                                    num_blocked_cells_in_unconfirmed_cells
```

```
                                    == num_sensed_blocked -
                                    num_confirmed_blocked
266                                 assert (len(unconfirmed_cells) >= (
                                    num_sensed_blocked -
                                    num_confirmed_blocked))
267
268                                 # Add this constraint to our knowledge base
269                                 add_constraint_to_knowledge_base(
                                    knowledge_base,
                                    variable_to_constraint_dict,
                                    unconfirmed_cells,
                                    num_blocked_cells_in_unconfirmed_cells)
270
271         # This is the advance strategy using backtracking search
272         if want_to_use_most_constraint_variable_for_backtracking_search:
273
274             # Getting queue of cells which are inferred
275             queue = make_inference_from_most_constraint_variable_list(
                                    knowledge_base,
                                    variable_to_constraint_dict,
276
                                    want_to_use_probability_approach=
                                    want_to_use_probability_approach, maze=
                                    maze, list_of_variables_in_the_path=
                                    list_of_variables_in_the_path)
277
278             # Adding into queue until it gets empty
279             while not queue.empty():
280                 current_node = queue.get()
281                 if current_node not in items_in_the_queue:
282                     items_in_the_queue.add(current_node)
283                     inference_items.put(current_node)
284
285     return is_block_node_in_current_path
```

Appendix B

Backtracking Search Code

```
1
2  def backtracking_search(current_variable_index: int, total_num_variables: int,
                           variables: list,
                           assigned_values_each_variable: dict,
                           frequency_of_each_output_for_each_variable
                           : dict, knowledge_base: list,
                           variable_to_constraint_dict: dict,
                           most_constraint_variable: tuple):
3      """
4      This function is used for backtracking search. It will recursively take the
                            value of each variable from the domain
                            and try to find possible answers.
5      :param current_variable_index: index of the current variable index
6      :param total_num_variables: total number of indexes for the backtrack
7      :param variables: list of variable in the given constraints
8      :param assigned_values_each_variable: to keep track of assigned value for
                           each variable
9      :param frequency_of_each_output_for_each_variable: to keep track the
                           frequency of valid outputs
10     :param knowledge_base: our knowledge base
11     :param variable_to_constraint_dict: dictionary which can give you list of
                           index of constrain where the given
                           variable is in the knowledge base
12     :param most_constraint_variable: one variable which has most constraint in
                           the given knowledge base
13     :return: Nothing as we are directly updating in the objects
```

```
    """
    # Condition when all the constraints are satisfied with the given set of values
    if current_variable_index == total_num_variables:

        # Increase frequency by one for each valid set
        for variable in assigned_values_each_variable:
            if variable not in frequency_of_each_output_for_each_variable:
                frequency_of_each_output_for_each_variable[variable] = [0, 0]
            frequency_of_each_output_for_each_variable[variable][
                                    assigned_values_each_variable[variable]
                                    ] += 1
        return

    # Loop over the all values which are in the domain.
    for value in [0, 1]:

        # Assign the value to that particular variable
        assigned_values_each_variable[variables[current_variable_index]] =
                                    value

        is_current_value_satisfied_all_constraints = True
        list_of_constraint_index_for_variable = list()

        # Iterate over all the constraints from the knowledge base and make it solved.
        for constraint_index in variable_to_constraint_dict[
                                    most_constraint_variable]:
            if variables[current_variable_index] in knowledge_base[
                                    constraint_index][0]:
                list_of_constraint_index_for_variable.append(constraint_index)
                knowledge_base[constraint_index][0].remove(variables[
                                    current_variable_index])
                knowledge_base[constraint_index][1] -= value
                if not check_constraint(knowledge_base[constraint_index][0],
                                    knowledge_base[constraint_index][1]):
                    is_current_value_satisfied_all_constraints = False

```

```
43          # If all constrains are satisfied after giving value to the variable,
                                        run backtracking search
44          if is_current_value_satisfied_all_constraints:
45              backtracking_search(current_variable_index + 1, total_num_variables
                                        , variables,
                                        assigned_values_each_variable,
                                        frequency_of_each_output_for_each_variable
                                        ,knowledge_base,
                                        variable_to_constraint_dict,
                                        most_constraint_variable)
46
47          # reverse the changes which we made earlier
48          for constraint_index in list_of_constraint_index_for_variable:
49              knowledge_base[constraint_index][0].add(variables[
                                        current_variable_index])
50              knowledge_base[constraint_index][1] += value
```

Appendix C

Inference from Most Constrained Variable

```
1
2
3  def make_inference_from_most_constraint_variable(knowledge_base: list,
                                                    variable_to_constraint_dict: dict,
                                                    most_constraint_variable: tuple):
4      """
5      This function is used to make inferences from the most constraint variable
6      :param knowledge_base: Agent's knowledge base
7      :param variable_to_constraint_dict: variable to list of index in knowledge
                                                    base constraint dictionary
8      :param most_constraint_variable: most constraint variable
9      :return: Queue which contains the nodes which are inferred
10     """
11
12     # Initialise few things
13     constraints_containing_most_constraint_variable_set = set()
14     constraints_containing_most_constraint_variable_list = list()
15     current_constraint_list = list()
16
17     # Iterate over each constraint to find the most constraint variable
18     for constraint_index in variable_to_constraint_dict[
                                                    most_constraint_variable]:
19         current_constraint_list.append(knowledge_base[constraint_index])
```

```
        for variable in knowledge_base[constraint_index][0]:
            if variable not in
            constraints_containing_most_constraint_variable_set:
                constraints_containing_most_constraint_variable_set.add(
                                        variable)
                constraints_containing_most_constraint_variable_list.append(
                                        variable)

    # Check the statement
    assert len(constraints_containing_most_constraint_variable_list) <= 20

    num_variables = len(constraints_containing_most_constraint_variable_list)
    frequency_of_each_output_for_each_variable = dict()
    assigned_values_each_variable = dict()

    # Call backtracking search
    backtracking_search(0, num_variables,
    constraints_containing_most_constraint_variable_list,
    assigned_values_each_variable, frequency_of_each_output_for_each_variable,
    knowledge_base, variable_to_constraint_dict, most_constraint_variable)

    # Check and add inferred items to the queue
    queue = Queue()
    for variable in frequency_of_each_output_for_each_variable:
        if (frequency_of_each_output_for_each_variable[variable][0] == 0 and
            frequency_of_each_output_for_each_variable[variable][1] != 0) or
            (frequency_of_each_output_for_each_variable[variable][1] == 0 and
            frequency_of_each_output_for_each_variable[variable][0] != 0):
            queue.put(variable)

    return queue
```

Appendix D

Most Constrained Variable List

```
1
2
3  def most_constraint_variable_list(variable_to_constraint_dict: dict,
                                     want_to_use_probability_approach: bool,
                                      maze: list):
4      """
5      Find the list of most constraint variables
6      :param variable_to_constraint_dict: dictionary containing key value pair of
                                      variable to index of constraint in
                                     knowledge base
7      :param want_to_use_probability_approach: if you want to get list based on
                                     probability then set it to True
                                     otherwise it should be False
8      :param maze: Maze object
9      :return: list containing most constraint variables
10     """
11
12     # If you want to get most constraint variable based on probability
13     if want_to_use_probability_approach:
14         max_probability = 0
15         cells_with_same_highest_probability = list()
16
17         # Iterate over variable, find probability and update correspondingly
18         for variable in variable_to_constraint_dict:
19             probability = len(variable_to_constraint_dict[variable]) / maze[
                                     variable[0]][variable[1]].num_neighbor
```

```
20              if max_probability < probability:
21                  max_probability = probability
22                  cells_with_same_highest_probability.clear()
23                  cells_with_same_highest_probability.append(variable)
24              elif max_probability == probability:
25                  cells_with_same_highest_probability.append(variable)
26
27          return cells_with_same_highest_probability
28
29      # Otherwise else would look at the absolute value of most constraint
                                        variables choose those variables which
                                        are in most number of constraints
30      else:
31          cells_which_are_most_constraint_variable = list()
32          max_num_constraint = 0
33          for variable in variable_to_constraint_dict:
34              if max_num_constraint < len(variable_to_constraint_dict[variable]):
35                  max_num_constraint = len(variable_to_constraint_dict[variable])
36                  cells_which_are_most_constraint_variable.clear()
37                  cells_which_are_most_constraint_variable.append(variable)
38              elif max_num_constraint == len(variable_to_constraint_dict[variable
                                        ]):
39                  cells_which_are_most_constraint_variable.append(variable)
40
41          return cells_which_are_most_constraint_variable
```