Contents

agents are:
Agent 1: Blindfolded agent
Agent 2: 4 Neighbour agent
Agent 3: Example Inference agent
Agent 4 : Constraint Satisfaction/Design Agent

1 Agent 3

- **Init**

  Now we begin with agent 3. In Phases class, we initialise a book which is a dictionary where the information about the current cell's neighbours (Nx), the cells confirmed to be blocked (Cx) and the parent of the cell is stored.

```
class Phases:

    def __init__(self,Klst,Flst,n,m):

        self.exploredset={()}
        """structure {(x,y): {Nx: positive int > 0, Cx: positive int >= 0,
                              P: tuple(positive int >= 0, positive int >= 0)}...}"""
        self.book={}      #dictionary to store details of the cell's neighbours, parents and Cx
        self.Klst=Klst    #Knowledge grid
        self.Flst=Flst    #Final grid
        self.n=n
        self.m=m
        self.trajlst = [] #the trajectory of the cell
```

- **Sensing** The Sensing method returns the number of blocked cells that are in the neighbourhood of the current cell. For this, we check in all 8 directions
  - North
  - South
  - East
  - West
  - Northeast
  - Northwest
  - Southeast
  - Southwest

  Also, our dictionary called 'book' is updated with the Nx, Cx and parents. To know the information about the number of blocked cells, we look up in the Full gridworld that is Flst.

```
def Sensing(self, pos, parents = (None, None)):

    # setting x, y coordinates
    x = pos[0]
    y = pos[1]
    self.Klst[x][y]=1

    # counter for neighbours
    neighbours = 0

    # counter for blocked neighbours
    counter = 0

    #flist is used only during sensing


    # looking up 8 neighbours
    if x-1>=0: # North
        neighbours+=1
        if(self.Flst[x-1][y] == -1):
            counter+=1
        if(y-1>=0): # North-West
            neighbours+=1
            if(self.Flst[x-1][y-1] == -1):
                counter+=1
        if(y+1<self.m): # North-East
            neighbours+=1
            if(self.Flst[x-1][y+1] == -1):
                counter+=1

    if x+1<self.n: # South
        neighbours+=1
        if(self.Flst[x+1][y] == -1):
            counter+=1
        if(y+1<self.m): # South-East
            neighbours+=1
            if(self.Flst[x+1][y+1] == -1):
                counter+=1
        if(y-1>=0): # South-West
            neighbours+=1
            if(self.Flst[x+1][y-1] == -1):
                counter+=1
```

```
    if y+1<self.m: # East
        neighbours+=1
        if(self.Flst[x][y+1] == -1):
            counter+=1

    if y-1>=0: # West
        neighbours+=1
        if(self.Flst[x][y-1] == -1):
            counter+=1

    # adding cells to the book for book-keeping
    self.book[pos] = {'Nx': neighbours, 'Cx': counter, 'P': parents}
```

- **Neighbour info**

  In the Neighbour-info method, the agent goes over all the 8 directions(at max) and updates the

neighbours of the current cell as well as the checks for Hidden cells (Hx), Empty cells (Ex), and Blocked cells that have been confirmed (Bx). To search for these parameters, the Knowledge Grid is used.

```
def Neighbour_info(self, pos):

    # setting x, y coordinates
    x = pos[0]
    y = pos[1]

    # list to store neighbour indices
    neighbour_ix = []

    # initialising other required variables
    hx, ex, bx = 0, 0, 0

    # looking up 8 neighbours and their indices
    if x-1>=0: # North
        neighbour_ix.append((x-1,y))
        if(y-1>=0): # North-West
            neighbour_ix.append((x-1,y-1))
        if(y+1<self.m): # North-East
            neighbour_ix.append((x-1,y+1))

    if x+1<self.n: # South
        neighbour_ix.append((x+1,y))
        if(y+1<self.m): # South-East
            neighbour_ix.append((x+1,y+1))
        if(y-1>=0): # South-West
            neighbour_ix.append((x+1,y-1))

    if y+1<self.m: # East
        neighbour_ix.append((x,y+1))

    if y-1>=0: # West
        neighbour_ix.append((x,y-1))

    for ax,ay in neighbour_ix:
        if(self.Klst[ax][ay] == 0):
            hx+=1
        elif(self.Klst[ax][ay] == 1):
            ex+=1
        else:
            bx+=1  #unconfirmed cell count is updated

    return hx, ex, bx, neighbour_ix
```

- **Inference SubInference: main logic of Agent 3**

For each cell on the Planning Path, Inference method is called. Once the agent enter this method, we fetch the Cx value from 'book' dictionary and Hx, Ex, Bx,and the list of the current cell's neighbours and their coordinates through Neighbour-info. Then SubInfer method is called within this for the current cell, and we update the Knowledge grid based on whichever case the values of Cx, Bx, Ex and Nlist it fits.

Within the Inference method, there runs a while loops, which checks for the parent of the current cell and does the same fetching of values we did for the current cell and the SubInfer method as well. Now, there is a recursive call to the parent of the current cell's parent.

Hence, for every cell, inside the Inference method, this while loop will take the agent back to check for the parent and then its parent and then till the initial cell (0,0) and through the Subinfer method, the

agent updates Cx, Bx, ex, and other parameters for all the cells in the grid. This way Knowledge Grid is gradually filled and the agent understands about the location of the blocked cells.

```
def Inference(self, pos):

    # setting x, y coordinates
    x = pos[0]
    y = pos[1]
    Cx=self.book[pos]['Cx']
    Hx,Ex,Bx,Nlst = self.Neighbour_info(pos)
    self.SubInfer(Hx,Ex,Bx,Nlst,Cx)
    print("CurrentAfter: ", pos, "Hx: ",Hx, "Cx: ",Cx)

    parent=self.book[pos]['P']
    while parent != (None,None):
        Cx=self.book[parent]['Cx']
        Hx,Ex,Bx,Nlst = self.Neighbour_info(parent)
        if Hx==0:
            break
        self.SubInfer(Hx,Ex,Bx,Nlst,Cx)
        print("ParentAfter: ", parent, "Hx: ",Hx, "Cx: ",Cx)
        parent=self.book[parent]['P']
```

– **SubInference**

Consists of three cases split via if-else
case1: if Cx=0 and Hx¿0
case2: if Cx=Bx and Hx¿0
case3: if Nlst - Cx = Ex and Hx¿0

```
def SubInfer(self,Hx,Ex,Bx,Nlst,Cx):
    if Cx==0 and Hx>0:
        for item in Nlst:
            x1=item[0]
            y1=item[1]
            if self.Klst[x1][y1]==0:
                self.Klst[x1][y1]=1

    elif Cx==Bx:
        for item in Nlst:   #Nlst is a tuple which contains coordinates of all th
            x1=item[0]
            y1=item[1]
            if self.Klst[x1][y1]==0:
                self.Klst[x1][y1]=1

    elif len(Nlst)-Cx==Ex:
        for item in Nlst:
            x1=item[0]
            y1=item[1]
            if self.Klst[x1][y1]==0:
                self.Klst[x1][y1]=-1
```

- **Planning**

The Planning phase is the same as Project 1 using AStarResolver.

```
def Planning(self,pos):
    a=AStarResolver(self.Klst,self.n,self.m,self.Flst)
    x=pos[0]
    y=pos[1]
    path= a.Solve(x,y)
    return path
```

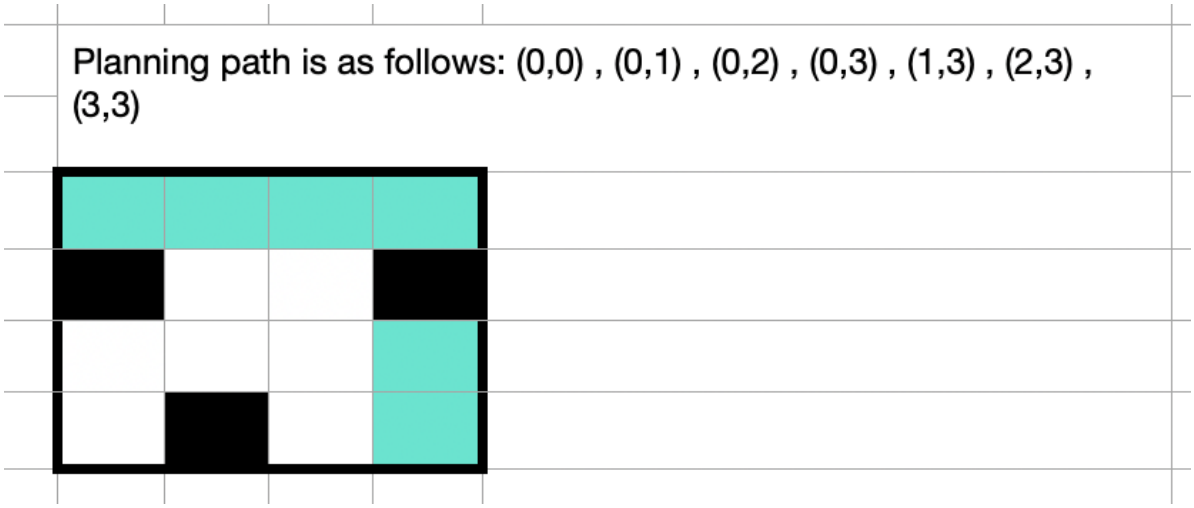
```
def Execution(self,path):
    block=False
    bcell=(0,0)
    print("path=",path)
    for item in range(len(path)):
        x=path[item][0]
        y=path[item][1]
        if path[item] not in self.book:
            self.Sensing(path[item],path[item-1])
        self.Inference(path[item])
        if self.Flst[x][y]==-1:
            self.Klst[x][y]=-1
            block,bcell=True,path[item-1]
            break
        else:
            if((len(self.trajlst) == 0) or (self.trajlst[-1]!=(x,y))):
                self.trajlst.append((x,y))

    if block:
        return True,bcell
    else :
        return False,bcell
```

- **Example**

  Lets take an example of a 4*4 gridworld where the planning path is as follows (0,0)
  (0,1)
  (0,2)
  (0,3)
  (1,3)
  (2,3)
  (3,3)

  And there are three blocked cells at following locations (1,0)
  (1,3)
  (3,1)

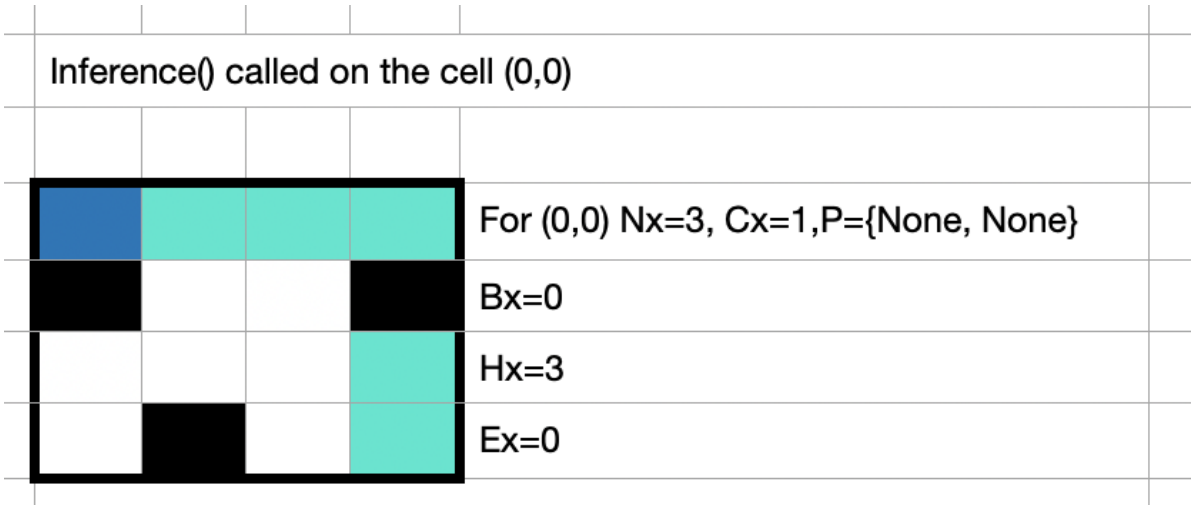


When agent starts from (0,0), Inference method is called and using the book dictionary we get Cx=1.(which was filled using Sensing method)

Then through Neighbour info we fetch
Hx=3(as there are only three neighbours of (0,0)
Bx=0(as we havent found any blocked cells yet
Ex=0(we havent been to any other cells)

On calling the SubInfer method on the (0,0) cell itself,
neither of the three cases are applicable hence it exits the Subinfer method.
case1: Cx=1 not satisfied
case2: Cx=1, Bx=0, not satisfied
case3: Nx=3, Cx=1, Ex=0; not satisfied

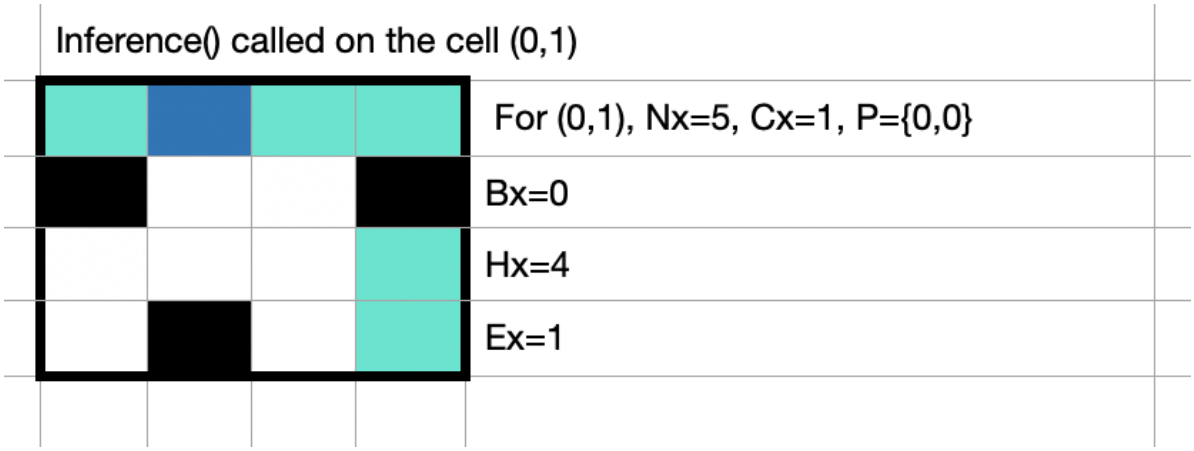
And agent wont enter while loop, because the parents of (0,0) is Null, Null



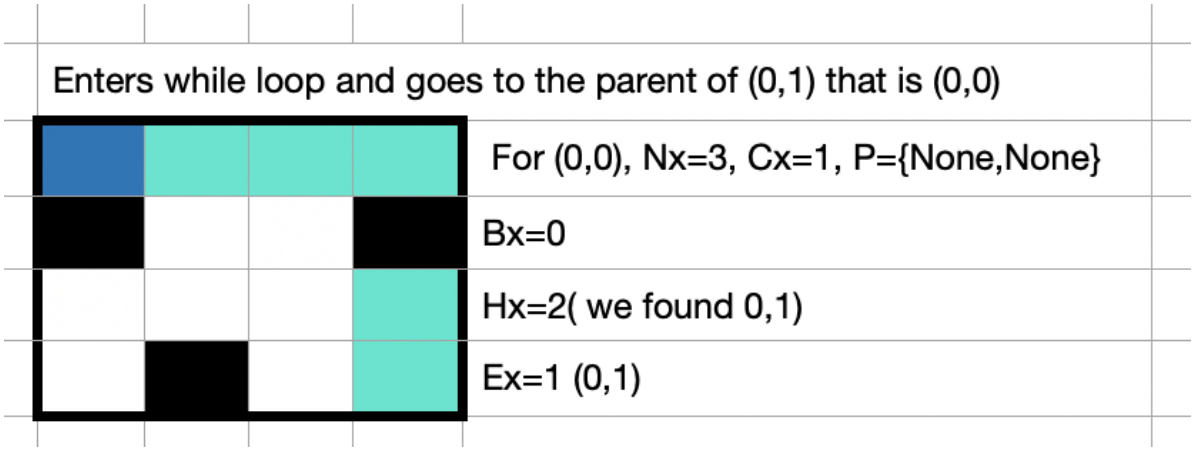
Moving onto the next cell (0,1), Inference method is called and using the book dictionary we get Cx=1.(which was filled using Sensing method)

Then through Neighbour info we fetch
Hx=4(We already visited (0,0)
Bx=0(as we havent found any blocked cells yet
Ex=1(we already found that 0,0 is unblocked)

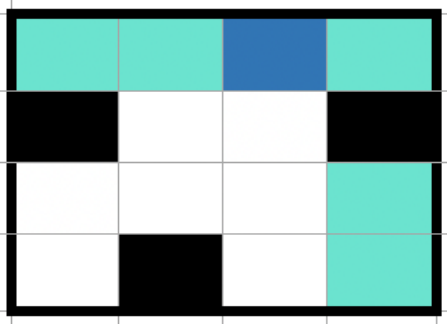
On calling the SubInfer method on the (0,1) cell itself,
neither of the three cases are applicable hence it exits the Subinfer method.
case1: Cx=1 not satisfied
case2: Cx=1, Bx=0, not satisfied
case3: Nx=5, Cx=1, Ex=1; not satisfied



Now, entering the while loop, it goes towards the parent of (0,1) that is (0,0) and again using the dictionary 'book', we fetch Cx=1.
Then through Neighbour info we fetch
Hx=2(we already visited (0,1))
Bx=0(as we havent found any blocked cells yet
Ex=1(we already visited (0,1))



Inference() called on the cell (0,2)



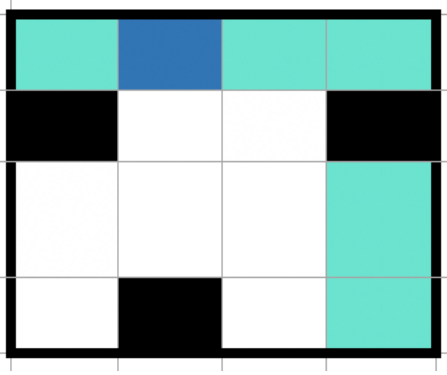
For (0,2), Nx=5, Cx=1, P={0,1}

Bx=0

Hx=4

Ex=1 (0,1) because it was parent

Enters while loop and goes to the parent of (0,2) that is (0,1)



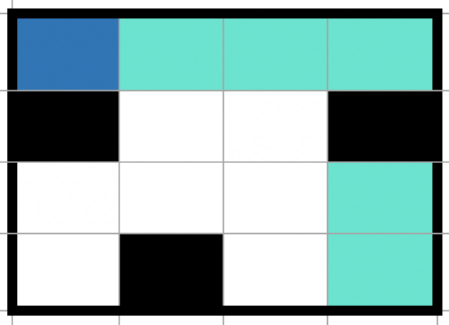
For (0,1), Nx=5, Cx=1, P={0,0}

Bx=0

Hx=3 (Hx is decreased by 1 because (0,2 was visited and found to be empty.

Ex=2 (0,0), (0,2)

Enters while loop and goes to the parent of (0,1) that is (0,0)



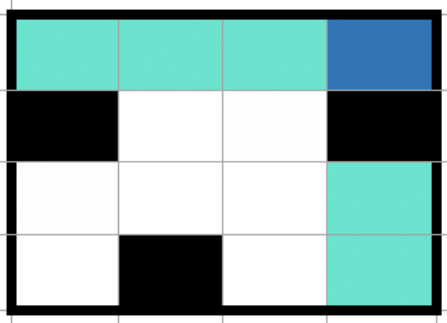
For (0,0), Nx=3, Cx=1, P={None,None}

Bx=0

Hx=2(we found 0,1)

Ex=1 (0,1)

Inference() called on the cell (0,3)



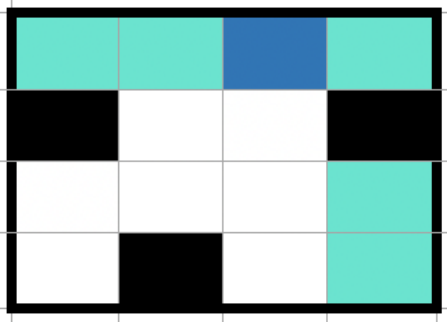
For (0,3), Nx=3, Cx=1, P={0,2}

Bx=0

Hx=2 (we know (0,2) as it was parent)

Ex=1 (0,2) because it was parent

Enters while loop and goes to the parent of (0,3) that is (0,2)



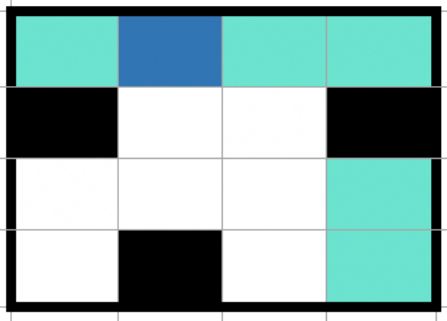
For (0,2), Nx=5, Cx=1, P={0,1}

Bx=0

Hx=3

Ex=2 (0,1) , (0,3)

Enters while loop and goes to the parent of (0,2) that is (0,1)



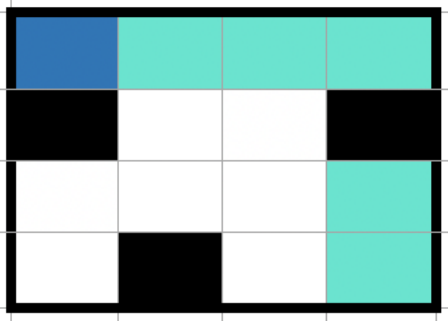
For (0,1), Nx=5, Cx=1, P={0,0}

Bx=0

Hx=3

Ex=2 (0,0), (0,2)

Enters while loop and goes to the parent of (0,1) that is (0,0)



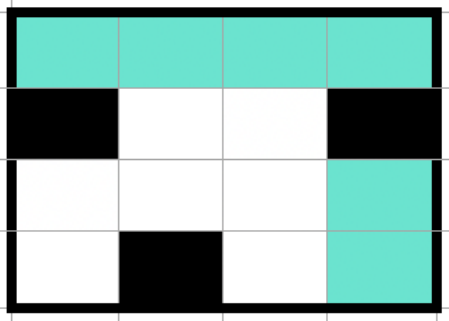
For (0,0), Nx=3, Cx=1, P={None,None}

Bx=0

Hx=2(we found 0,1)

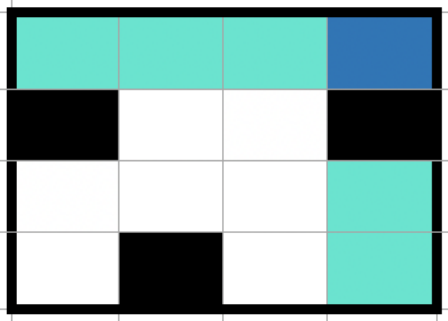
Ex=1 (0,1)

Agents bumps into a block, at (1,3)



Bx will be updated to 1 for (0,3)

Inference() called on the cell (0,3)



For (0,3), Nx=3, Cx=1, P={0,2}

Bx=1(updated)

Hx=1 (1,2)

Ex=2 (0,2) because it was parent,

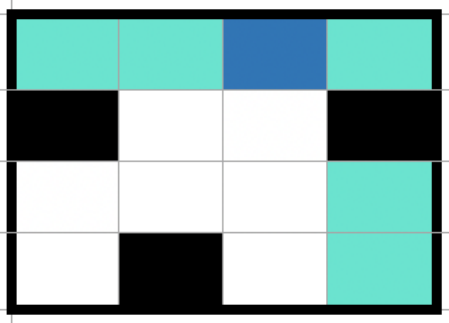
(1,2) through sub inference

In sub inference method,

Case1 won't apply cx=1

Case 2 applies cx=bx hence we update the value in the knowledge grid of 1,2 as 1 hence marking it as an empty/unblocked cell.

Enters while loop and goes to the parent of (0,3) that is (0,2)



For (0,2), Nx=5, Cx=1, P={0,1}

Bx=1 (1,3)

Hx=1 (1,0)

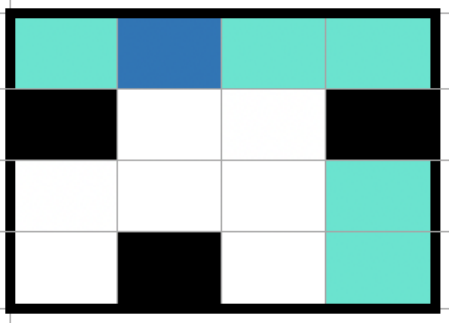
Ex=3 (0,1) , (0,3) , (1,2)

In sub inference method,

Case1 won't apply cx=1

Case 2 applies Cx=Bx hence we update the value in the knowledge grid of 1,1 as 1 hence marking it as an empty/unblocked cell.

Enters while loop and goes to the parent of (0,2) that is (0,1)



For (0,1), Nx=5, Cx=1, P={0,0}

Bx=0

Hx=1 (1,0)

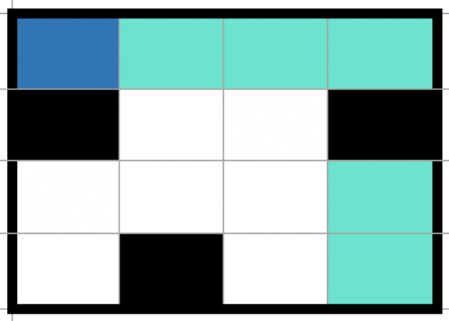
Ex=4 (0,0), (0,2), (1,1) , (1,2)

In sub inference method,

Case1 won't apply cx=1

Case2 won't apply Cx=1, Bx=0

Case 3 will apply, Nx-Cx=Ex; (1,0) is marked as -1 (blocked)

Enters while loop and goes to the parent of (0,1) that is (0,0)
 For (0,0), Nx=3, Cx=1, P={None,None}
Bx=1(1,0)
Hx=0
Ex=2 (0,1), (1,1)
In sub inference method,
Case1 won't apply cx=1 but Hx=0
Case2 won't apply Cx=1, Bx=1 but Hx=0
Case3 won't apply Hx=0

2 Comparing Agent 3 with Agent 1 and 2

Trajectory length of Blindfolded Agent is the longest as it bumps each time and can move in only one direction. Agent 3 has least Trajectory Length hence it is the best.

Couple of tables have been attached to show difference in trajectory length between all the three agents.

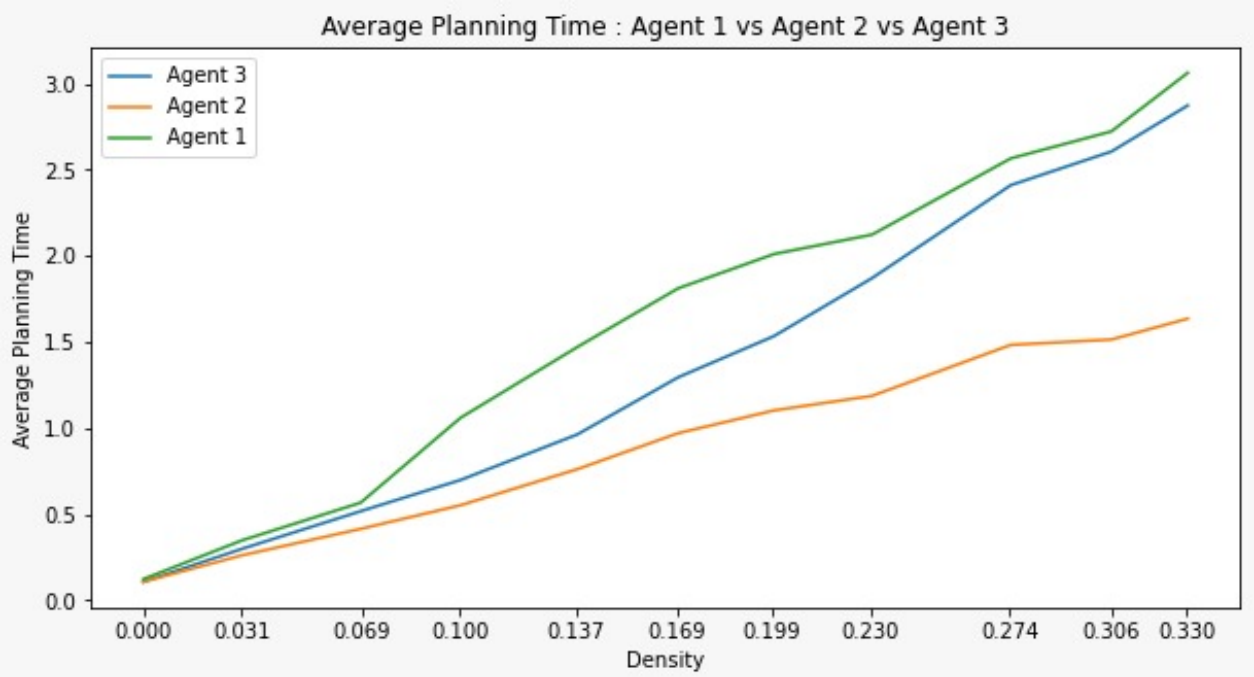
We can observe that initally all agents perform equally well but as the density increases, the ranking of their performances becomes more clearer.

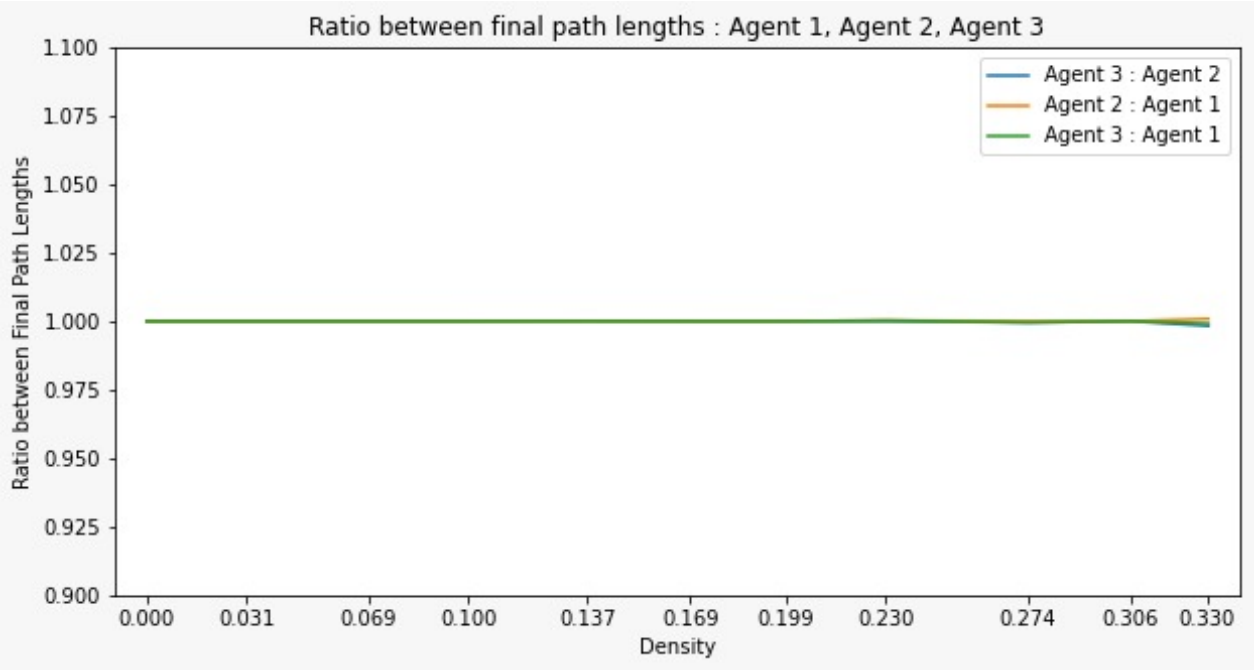
Density	Agent 2	Agent 1	Trajectory Difference
0.330305	296.4	302.2	5.8
0.306148	287.41	293.1	5.69
0.27435	283.39	287.76	4.37
0.23046	260.9	263.28	2.38
0.19941	257.98	260.02	2.04
0.16909	242.51	244.02	1.51
0.13728	232.45	233.71	1.26
0.100344	220.63	220.9	0.27
0.068753	218.42	218.54	0.12
0.031157	207.3	207.32	0.02
0	201	201	0

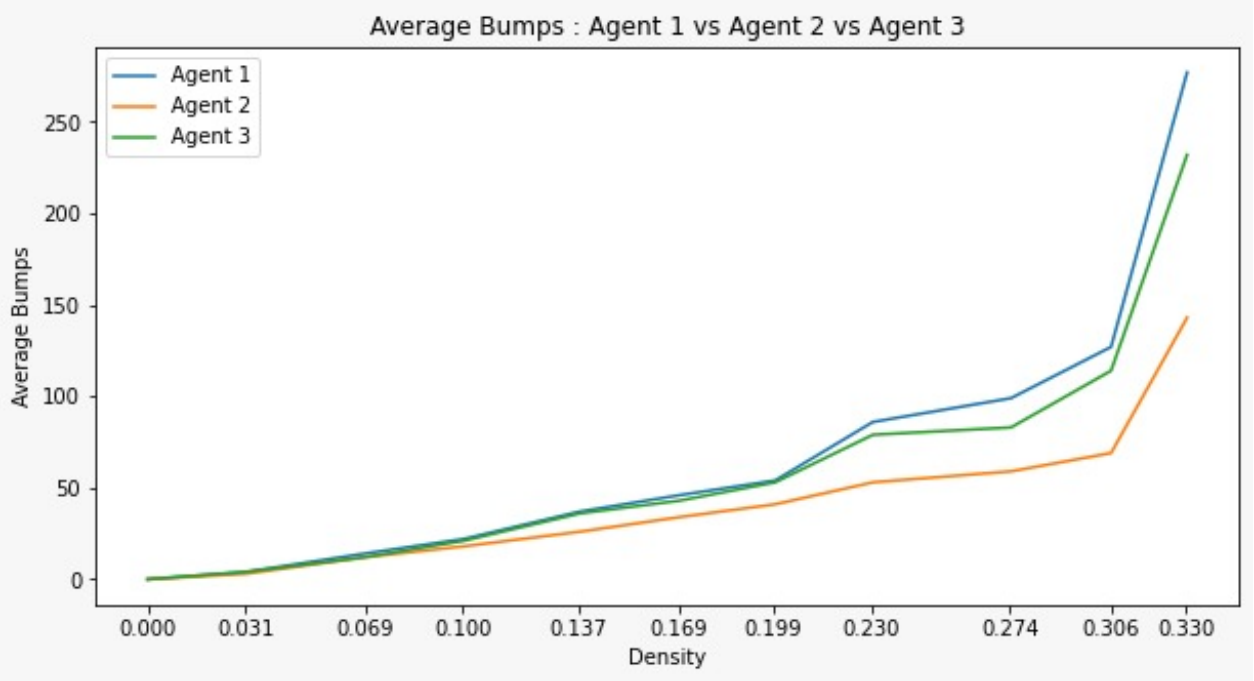
Density	Agent 3	Agent 2	Trajectory Difference
0.330305	292	296.4	4.4
0.306148	283.9	287.41	3.51
0.27435	280.01	283.39	3.38
0.23046	257.35	260.9	3.55
0.19941	254.47	257.98	3.51
0.16909	239.23	242.51	3.28
0.13728	228.57	232.45	3.88
0.100344	217.36	220.63	3.27
0.068753	215.66	218.42	2.76
0.031157	205.1	207.3	2.2
0	201	201	0

Density	Agent 3	Agent 1	Trajectory Difference
0.330305	292	302.2	10.2
0.306148	283.9	293.1	9.2
0.27435	280.01	287.76	7.75
0.23046	257.35	263.28	5.93
0.19941	254.47	260.02	5.55
0.16909	239.23	244.02	4.79
0.13728	228.57	233.71	5.14
0.100344	217.36	220.9	3.54
0.068753	215.66	218.54	2.88
0.031157	205.1	207.32	2.22
0	201	201	0

Average Trajectory Length			
Density	Agent 3	Agent 2	Agent 1
0.330305	292	296.4	302.2
0.306148	283.9	287.41	293.1
0.27435	280.01	283.39	287.76
0.23046	257.35	260.9	263.28
0.19941	254.47	257.98	260.02
0.16909	239.23	242.51	244.02
0.13728	228.57	232.45	233.71
0.100344	217.36	220.63	220.9
0.068753	215.66	218.42	218.54
0.031157	205.1	207.3	207.32
0	201	201	201


Average Planning Time : Agent 1 vs Agent 2 vs Agent 3
Agent 3
Agent 2
Agent 1
Average Planning Time
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.000
0.031
0.069
0.100
0.137
0.169
0.199
0.230
0.274
0.306
0.330
Density


Ratio between final path lengths : Agent 1, Agent 2, Agent 3
Agent 3 : Agent 2
Agent 2 : Agent 1
Agent 3 : Agent 1
Ratio between Final Path Lengths
1.100
1.075
1.050
1.025
1.000
0.975
0.950
0.925
0.900
0.000
0.031
0.069
0.100
0.137
0.169
0.199
0.230
0.274
0.306
0.330
Density


Average Bumps : Agent 1 vs Agent 2 vs Agent 3
Agent 1
Agent 2
Agent 3
Average Bumps
0
50
100
150
200
250
0.000
0.031
0.069
0.100
0.137
0.169
0.199
0.230
0.274
0.306
0.330
Density

3 Agent 4 : Designing and Implementation

- ## Constraint Satisfaction: Rules and Inferences

  A constraint satisfaction problem (CSP) is a type of problem that needs its solution within some limitations or conditions also known as constraints. It consists of the following :

  1)A finite set of variables which stores the solution (V = V1, V2, V3,....., Vn)

  For the particular cell, all its neighbours form the set of variables Hence, for example V(0,0) = (0,1),((1,0),(1,1)

  2)A set of discrete values known as domain from which the solution is picked (D = D1, D2, D3,.....,Dn) Here we have three types of values

  -1: block 0: hidden/unconfirmed 1: empty/unblocked

  Hence D=-1,0,1

  3)A finite set of constraints (C = C1, C2, C3,....., Cn) Here, all the Cx equations that we form for each cell are our set of constraints.

  In our following example, it is shown that C=a,c,c,d where a,b,c,d are Cx equations for each cell.

```
def constraintSatisfaction(self, pos):

    # getting current element's hidden neighbours
    Hx,Ex,Bx,Nlst = self.Neighbour_info(pos)
    current_hx_neighbours = self.hidden_neighbours(Nlst)
    self.SubInfer(Hx,Ex,Bx,Nlst,self.book[pos]['Cx'])
    # getting parent element's hidden neighbours
    parent = self.book[pos]['P']

    # intialising equations
    current_eq = None
    parent_eq = None

    while((parent != (None, None)) and (len(current_hx_neighbours)>0)):

        if(current_eq == None):
            current_eq = current_hx_neighbours + [self.book[pos]['Cx']]

        Hx,Ex,Bx,Nlst = self.Neighbour_info(parent)
        parent_hx_neighbours = self.hidden_neighbours(Nlst)
        if(parent_eq == None):
            parent_eq = parent_hx_neighbours + [self.book[parent]['Cx']]

        current_eq = self.common_neighbours(parent_eq[:-1], current_eq[:-1], parent_eq[-1], current_eq[-1])
        if(current_eq != None):
            if(current_eq[-1] == 0):
                for i in current_eq[:-1]:
                    self.Klst[i[0]][i[1]] = 1
            sensed_blocks = len(current_eq[:-1])
            if(current_eq[-1] == sensed_blocks):
                for i in current_eq[:-1]:
                    self.Klst[i[0]][i[1]] = -1
        self.SubInfer(Hx,Ex,Bx,Nlst,self.book[parent]['Cx'])
        pos = parent
        Hx,Ex,Bx,Nlst = self.Neighbour_info(pos)
        current_hx_neighbours = self.hidden_neighbours(Nlst)
        parent_eq = current_eq
        parent = self.book[pos]['P']
```

- ## Implementing Agent 4 and its Working

at cell 1,

Cx equation for its neighbours will be :

2+5+6 = 1 -a(1 blocked cell amongst the 3 cells)

at cell 2,

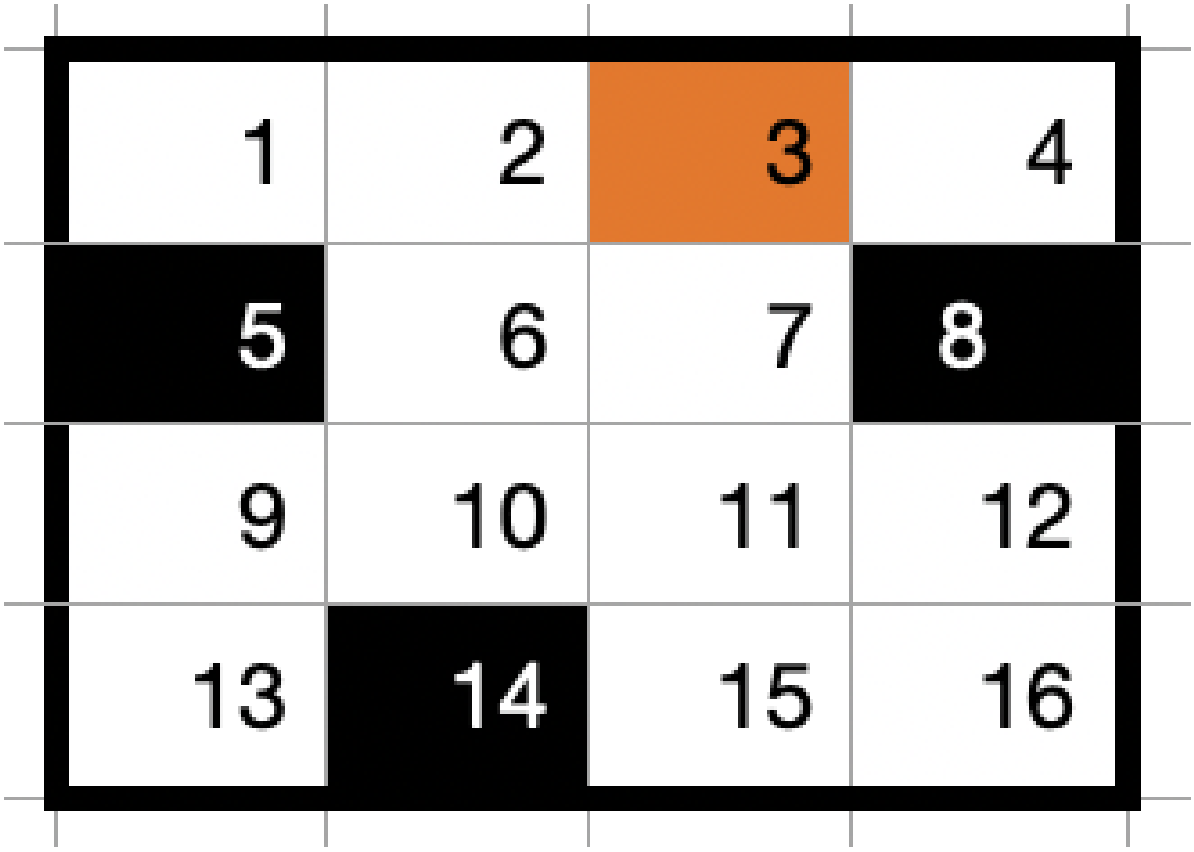
Cx equation for its neighbours will be : 1+5+6+7+3=1 (1 blocked cell)

Now cell no 1 is already visited

5+6+7+3=1. -b

Now 2 is visited, hence its not blocked. Backtracking to the previous node, and fetching equation of cell1, with the informed knowledge that 2 is empty cell, substituting in the equation for cell1, 5+6=1

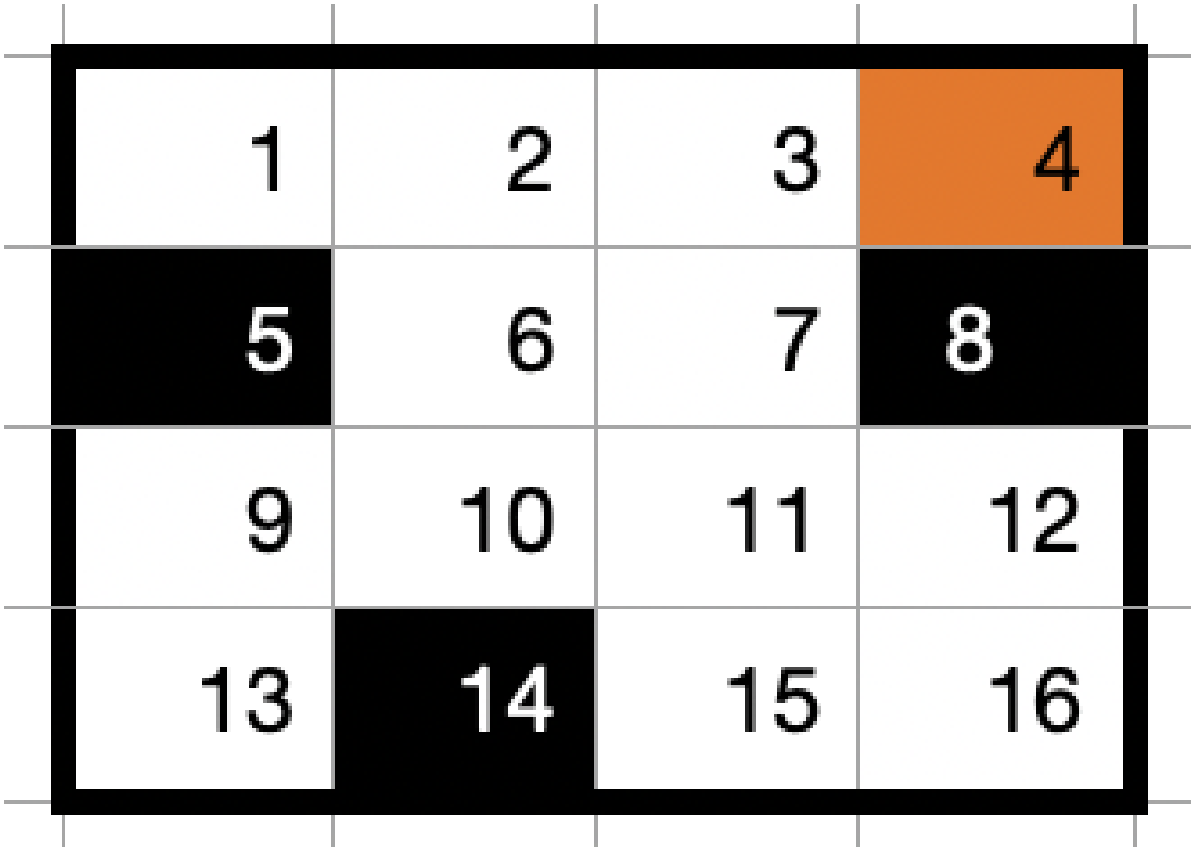
Hence, either of 5 and 6 is a blocked cell. using this inference in equation b, we know that both cells 7 and 3 are empty. 7+3 =0



at cell 3,

2+6+7+8+4 =1. -c

2 , 3 are already visited 7 is an empty cell 6+8+4 =1. -d



at cell 4, 3+7+8=1

3, 7 both are unblocked Hence, using inference, 8 is a block

now, using backtracking from equation d, 6+8+4=1 4 is visited 8 is block 6 is empty

Using this to infer through backpropagation, 5+6 =1 6 is empty Hence, 5 is blocked

This way, we identified both the blocked cells.

- **Comparing Agent 3 and Agent 4**

  From the 4*4 example grid example that we used throughout this project, we can infer that thoy are close in performance because both use inference methods as well as backtrack to their parent nodes, it can be seen that agent 4 is slightly better performing.

  - finding out blocked cell
  - Both the agents needed to visit the first 4 nodes (0,0), (0,1), (0,2), (0,3) but, agent 3 had to bump into the blocked cell (1,3) to find out that it was a block. On the other hand, Agent 4 inferred when it was on (0,3) itself that (1,3) is a block using constraint satisfaction.
  - finding out empty cell

  Also, to find out that (1,2) was a empty/unblocked cell, Agent 3 had to visit till (0,3) hence 4 nodes of traversal. On the other hand, Agent 4 just needed to visit 2 cells, and at (0,1) it inferred that both cells 3 and 7 are unblocked.

  Hence, Agent 4 is better at inferencing and is faster at updating the knowledge grid.

- **Does Agent 3 ever get anything that Agent 4 does not?**

  No, because from our code,

  Agent 4 = Agent 3 (Inferencing) + Constraint Satisfaction

  Hence, Agent 3 is comprised within Agent 4.

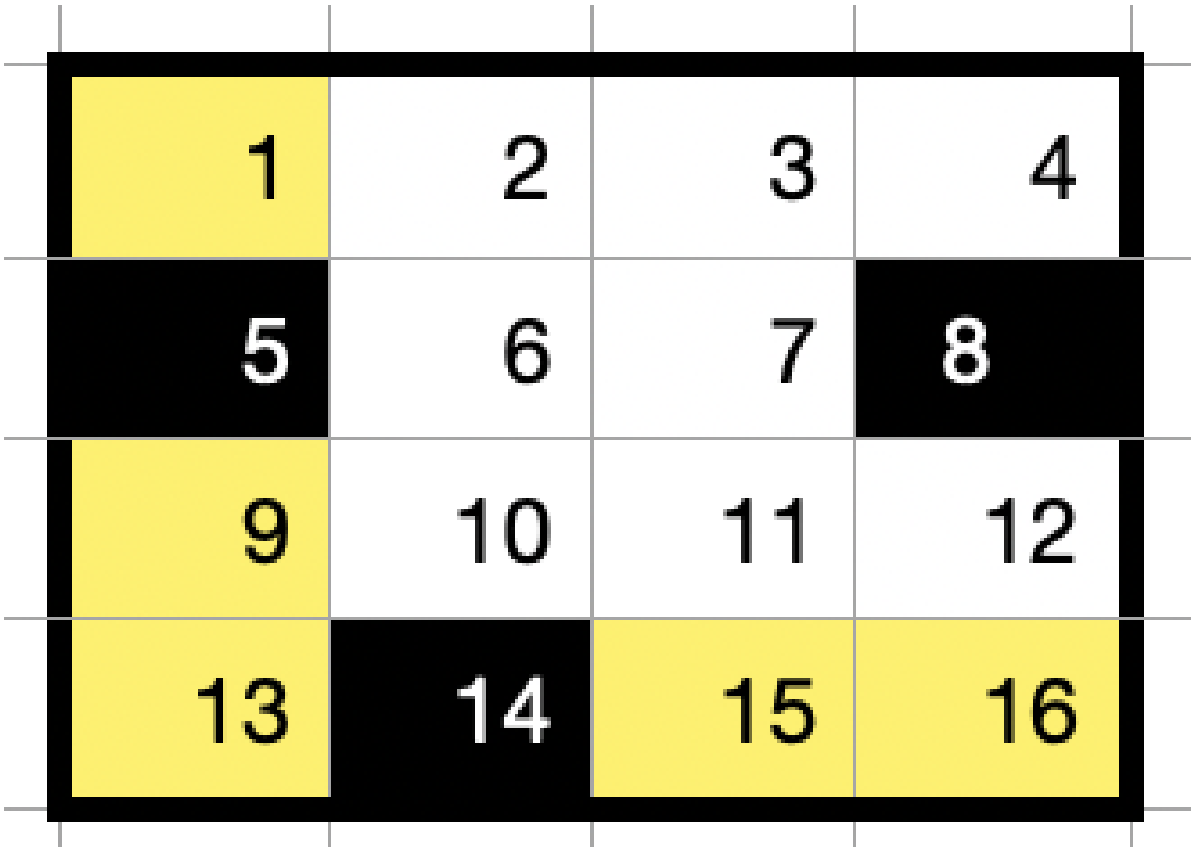
- **Does Agent4 infer everything that is possible to infer?**

  No, there are other methods that can be incorporated like "Proof Of Contradiction" which take the assumption that this particular cell is blocked amongst the two and if that turns out to be false then it recalibrates and performs the execution again. Our Agent4 has its own specific constraints, particularly that of using the neighbours Cx, but that doesnt mean that there cant be other methods that can enhance it.

- **Situation where Inference is possible but Agent 4 cannot?**

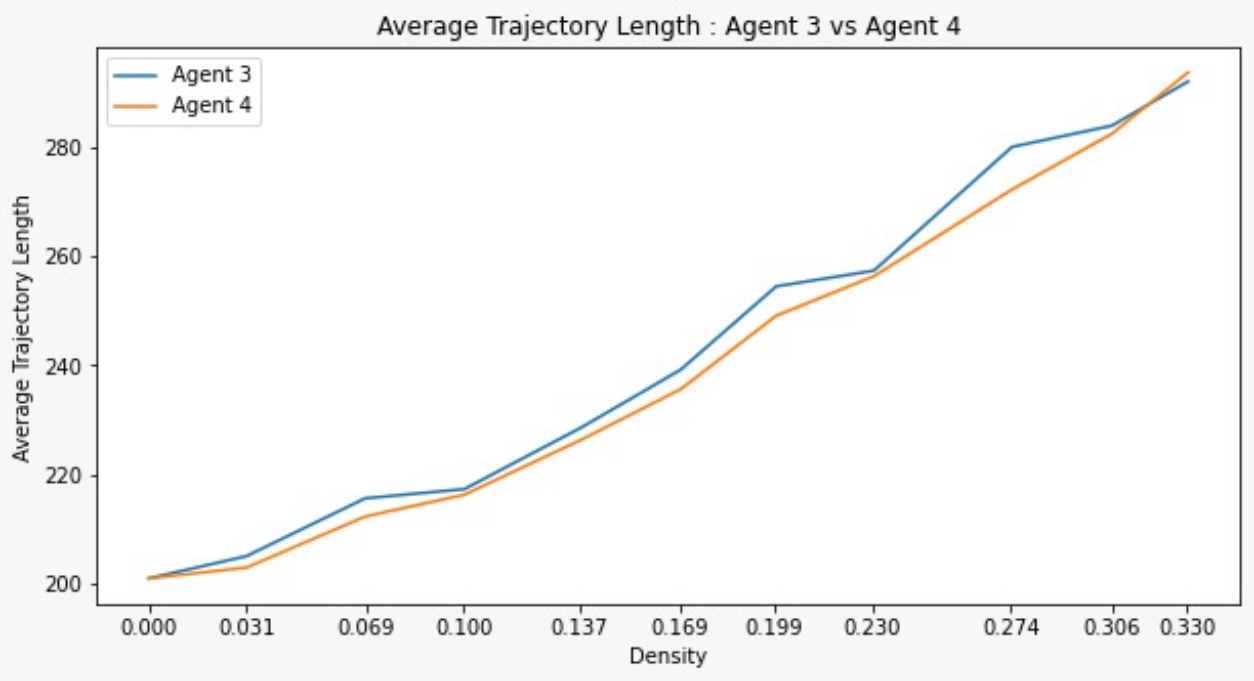
  Yes, there can be such situations such as using Proof of Contradiction. For example, in our 4*4 grid, Agent 4 realises that (1,0) is a block when it reaches the 4th node (0,3).

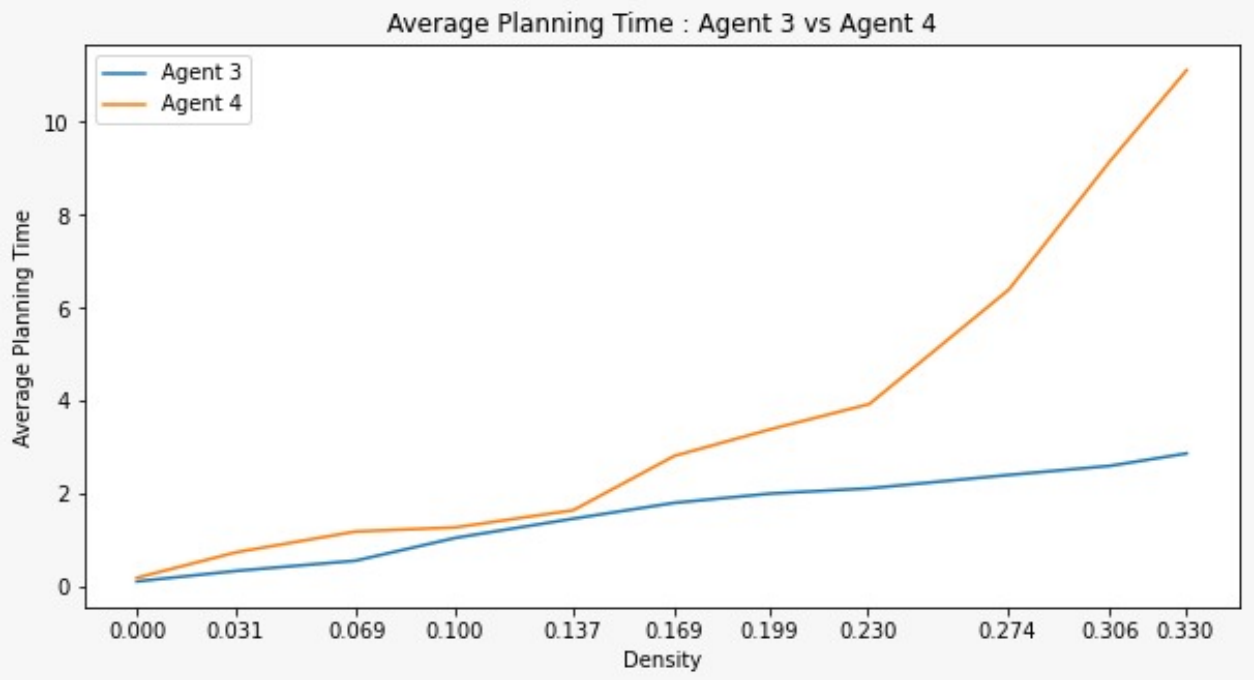
  But, Proof of Contradiction just assumes its planning path and then starts traversing on it. Hence, just when it reaches to the 2nd node, it realised that (1,0) is blocked node.

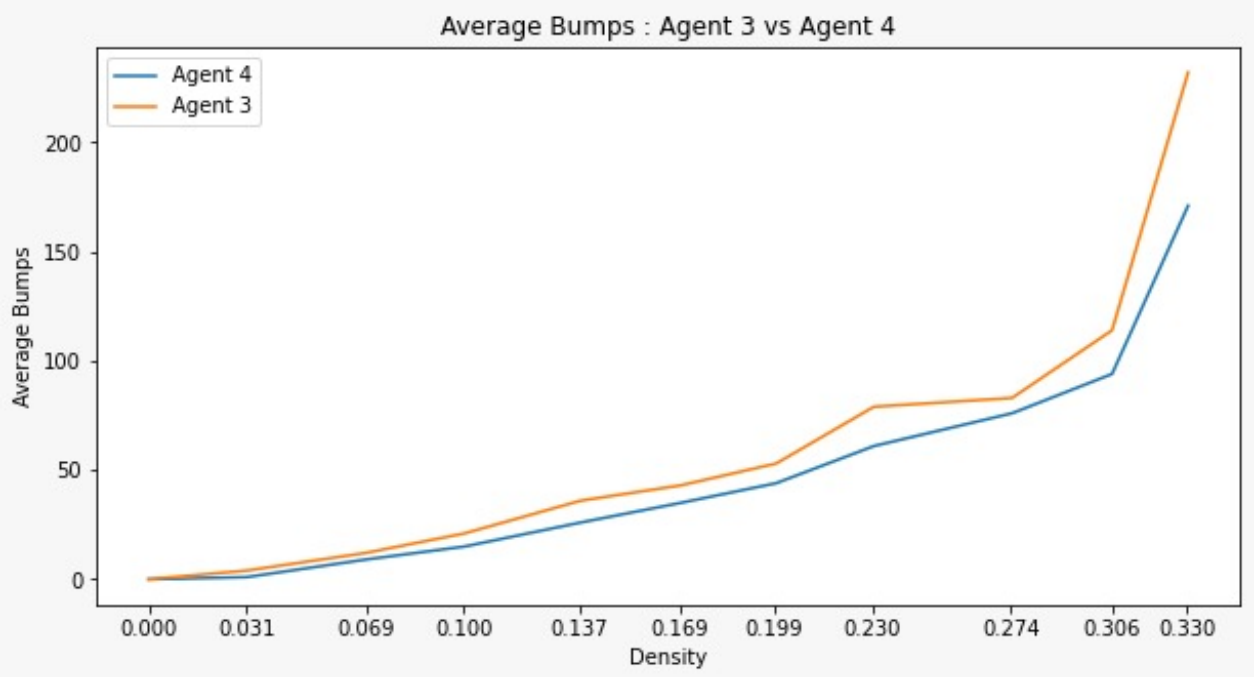


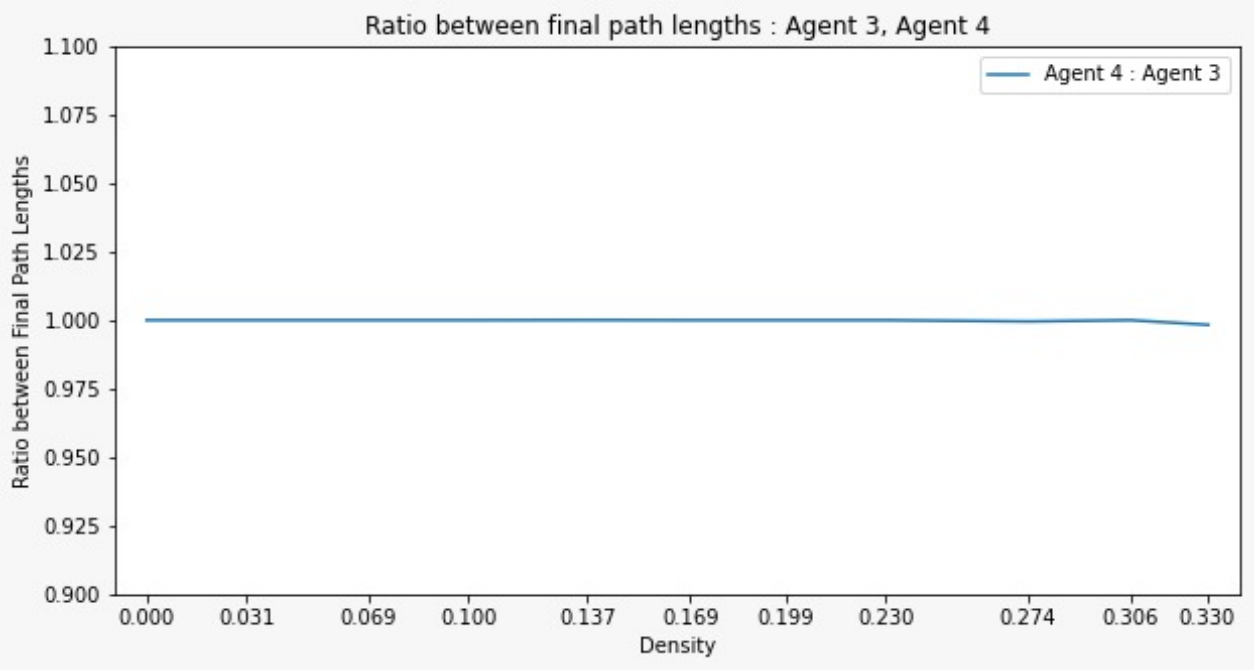
4 Performance Comparison Agent 3 and Agent 4

Agent 4 performs consistently better than Agent 3






Average Bumps : Agent 3 vs Agent 4
Agent 4
Agent 3
Average Bumps
0
50
100
150
200
0.000
0.031
0.069
0.100
0.137
0.169
0.199
0.230
0.274
0.306
0.330
Density


Ratio between final path lengths : Agent 3, Agent 4
Agent 4 : Agent 3
Ratio between Final Path Lengths
0.900
0.925
0.950
0.975
1.000
1.025
1.050
1.075
1.100
0.000
0.031
0.069
0.100
0.137
0.169
0.199
0.230
0.274
0.306
0.330
Density