Names: Fionna Zhang, Katherine Lee, YuJu Ku
NetIds: fwz2, ksl103, yk544

Partial Sensing

Example Inference Agent

In order to create the Example Inference Agent (Agent 3), we began with our implementation of the Repeated Forward A* algorithm, which generated a path through an empty gridworld using A* and then filled in blocks as and when the agent ran into them. From here, we added the additional fields for each cell, keeping track of:

- Blocked status
- N_x: the number of neighbors that cell x has
- C_x: the number of neighbors x that are sensed to be blocked
- B_x: the number of neighbors x that have been confirmed to be blocked
- E_x: the number of neighbors of x that have been confirmed to be empty
- H_x: the number of neighbors x that are still hidden or unconfirmed

We stored this information in a separate grid, where the cell (row, column) corresponds to a list of the above values. Initially our Blocked value is *None*, while all the rest of our initial values are 0. At each node, we fill in our gridworld further, first by generating the Blocked status, N_x, and C_x values of our current node. From here, we generate all neighbors, and if we have seen any of our neighbors previously, we take their blocked or empty status and it to our B_x and E_x values, respectively. Finally, our H_x value is calculated by subtracting B_x and E_x from our N_x. Then, we begin our inference.

If we find that $C_x = B_x$, we know that we have found all of our blocked neighbors, and therefore all of our other hidden neighbors are confirmed empty. From here, we set all the neighbors which are not blocked to be empty, and iterate through their neighbors, giving a +1 E_x to each of the neighbors and seeing if we can make any new inferences on the neighbors.

Alternatively if we find that $N_x - C_x = E_x$, we know that all of our remaining hidden neighbors are blocked. We set all the remaining neighbors to be blocked and iterate through their neighbors, giving a +1 B_x to each of its neighbors and then seeing if we can make any new inferences on the neighbors. Both inference rules stop, when we are no longer able to make any new inferences, and then we move forward in our path.

Each time that we run into a block in our path or we infer a block that is in our projected path, we rerun A*. If we find the block by trying to move into that cell, we rerun A* using are last empty node as the new start and our partially filled gridworld as the gridworld. If we are inferring that not our current cell is blocked, but a cell in our projected path down the line is blocked, we immediately rerun A* using our current node as the start, and avoiding the cell that is inferred to

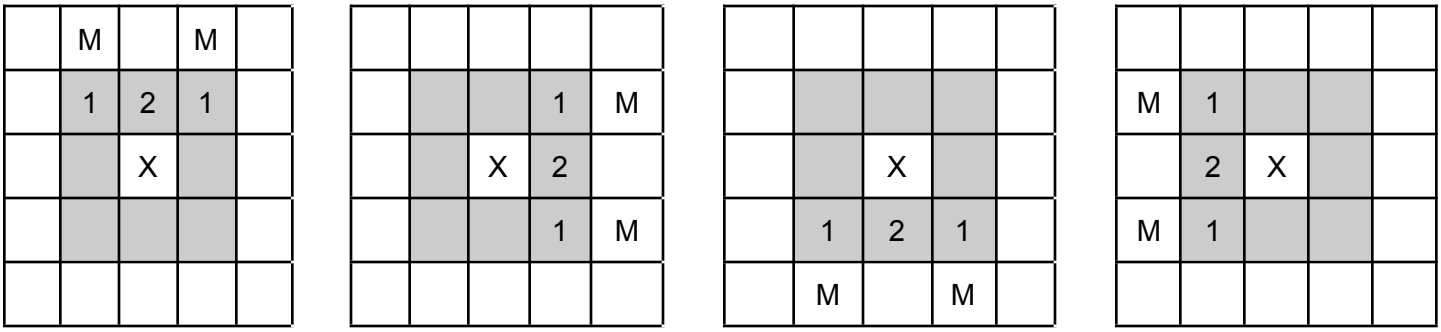
be blocked. The algorithm stops once it senses that we have reached the goal node, or if it finds that there is no path to the goal node.

Inference Agent (Agent 4)

In order to further our inference and be able to detect blocked cells earlier, thereby shortening the total number of cells needed to be traversed to find the goal node, we decided to implement some strategies inspired by the game Minesweeper. In order to better and more efficiently keep track of blocked and potentially blocked nodes, we added an additional parameter $CB_x = C_x - B_x$

1-2-1 Rule (Design/Implementation)

The 1-2-1 rule operates as follows:



X denotes location of current node, 1-2-1 are CB_x values of the neighbors, M is the inferred location of a block

Any time you see a series of neighbors with CB_x values in a 1-2-1 pattern, you know that the cell adjacent to the cells with the CB_x value of 1 is blocked, as demonstrated above.

From the current node, we check each of the edges:

- North Edge: North Cell, Northeast Cell, Northwest Cell
- East Edge: East Cell, NorthEast Cell, Southeast Cell
- West Edge: West Cell, NorthWest Cell, Southwest Cell
- South Edge: South Cell, Southeast Cell, Southwest Cell

If in any of our edges, we see the pattern 1-2-1, as demonstrated above, in our CB_x parameter, we are able to infer that there is a blocked cell next to one of our cardinal direction neighbors. From here, we update our internal gridworld copy to have these two new blocks. When we find new blocks, we update the neighbors of all the B_x values to have a +1 additional confirmed block, and then run our `update_neighbors_bx` function in order to recursively infer further any new blocked or empty cells that may have come about from our new blocked cell.

Each time that using this pattern infers blocked cells, we add the blocks to our internal gridworld, and also check if any of these new blocks are in our projected path. If they are in our projected path, we rerun A* to preemptively avoid them.

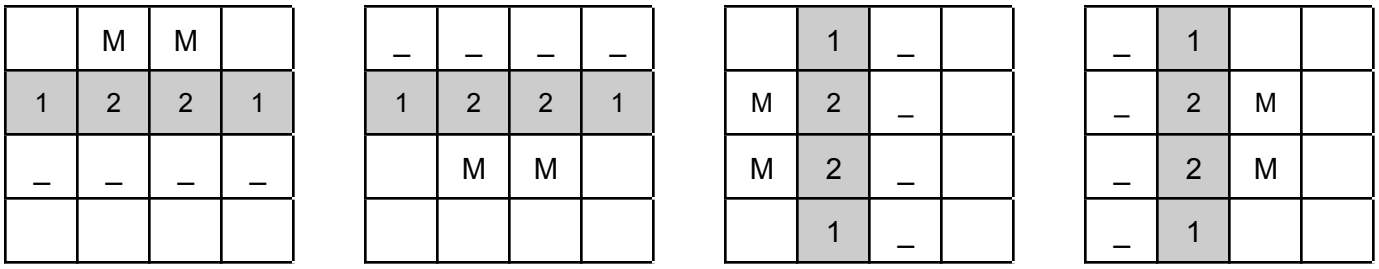
1-2-1 Rule (Computational Issues)

We do not, as we do when we find new blocked cells, continue to run the 1-2-1 rule on our neighbors for two reasons: First, because of performance and computational overhead. If, at each cell, we were to traverse through our entire internal gridworld, looking for the 1-2-1 pattern in our stored CB_x value, our algorithm would take significantly longer than it already does. Second, because having a CB_x value relies on us already having traversed the node (since it requires having a C_x value), it is unlikely that we are going to find any 1-2-1 patterns in areas further away from the current path. Also, because where we are currently is where we believe the shortest path to be, it may not be helpful to traverse the entire grid for blocks. Blocks opposite to our current shortest path may not be relevant to us, and may create a burden if we have to continuously check for them.

One of the drawbacks of this approach is that it requires us to have traversed our neighbors and attained their CB_x values, therefore early on in the gridworld and in cases where there is a relatively straightforward path, this rule does not come into play very often. During our own testing, in gridworlds of small dimensions, we rarely saw this pattern. And if we were considering performance testing, needing to check every edge could take significantly longer than not.

1-2-2-1 Rule (Design/Implementation)

The 1-2-2-1 rule operates as follows:



1-2-2-1 are CB_x values of the nodes, M is the inferred location of a block, _ is a confirmed empty space

The 1-2-2-1 rule is similar to the 1-2-1 rule in that it is looking for patterns in our internal CB_x values. However, it differs in that it is also considering the CB_x value of our current node. The 1-2-2-1 rule is only run when we have a CB_x value of 1 or 2 in our current node, meaning that our current node is a direct part of the pattern.

If our current node does have a CB_x value of either 1 or 2, we then check the CB_x values of its neighbors horizontally or vertically to see if we can see the 1-2-2-1 pattern. Since the CB_x value has already subtracted the number of confirmed blocks, what we have to check is whether the 4 cells above, below or beside the 1-2-2-1 pattern are unvisited. If we see a 1-2-2-1 pattern in the row of our current node, we check the four cells directly above or below to see if they are unvisited. If the four cells above are unvisited, we can infer that there are two blocked mines

above our 1-2-2-1 pattern, which are the cells above 2-2. In the same way, we can infer that there are two blocked mines below the 1-2-2-1 pattern if the four cells below are unvisited. The same holds true for if we find the 1-2-2-1 pattern in a column.

Each time that our 1-2-2-1 pattern finds new blocked nodes, we add the blocks to our internal gridworld. We then run our `update_neighbors_bx` function in order to find out if these new blocked nodes change any of the values for our previously found nodes and to add a +1 B_x value to all the neighboring nodes of our newly inferred blocks. Finally, we add the blocks to our internal gridworld and check if any of them appear in our projected pattern. If they do, A* is rerun to avoid them.

1-2-2-1 Rule (Computational Issues)

Similar to 1-2-1 and for the same reasons, we do not traverse the entire grid each time looking for this pattern. We do, however, each time we find blocks from one of these patterns, continue updating neighbors until we can infer no more. In this way, neither the 1-2-1 rule nor the 1-2-2-1 rule infer “everything that there is to infer.” It is possible that if we were to continue traversing all of our neighbors for their CB_x values that we would find more instances of the 1-2-2-1 and 1-2-1 pattern and therefore be able to infer more blocks. We do not do this because of the computational cost of having to traverse through our entire internal gridworld to only potentially find more instances of the pattern.

This problem remains the same from the 1-2-1 strategy, and to an even greater extent because we need to be able to consider the CB_x value of more nodes. Many times, and in fact most of the time, because we can only find a C_x value when we go into a node, we are not able to infer any CB_x value because most nodes are not concentrated around our current path. And yet, if we were to continue looking for the 1-2-2-1 pattern throughout the entire grid, we would have still spent a significant amount of time not only searching for the pattern and not finding it, but potentially finding blocks that are not relevant to us because they are too far away from our projected path and may never be traversed.

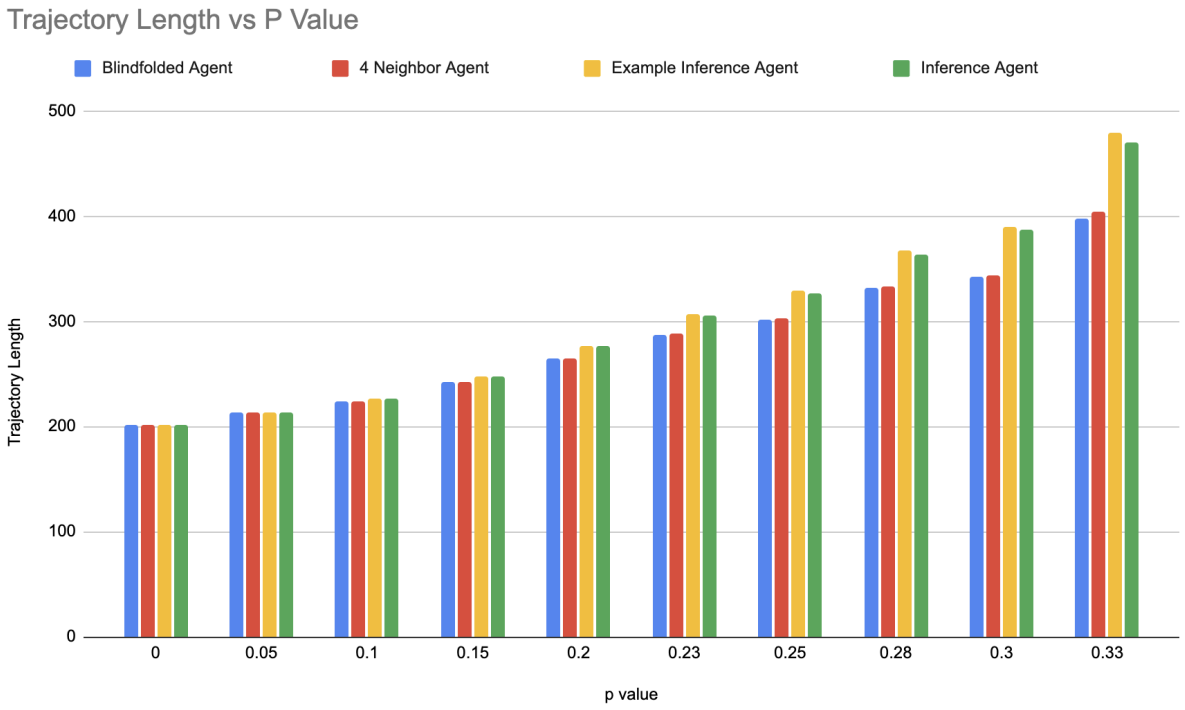
Testing

All testing is done on a 101x101 gridworld. The data presented is the average trajectory length of each of the four inference agents at specific *p* values between 0.0 and 0.33. For each test, the agents are run on the same 100 gridworlds, and the trajectory lengths for all the runs are then averaged.

1-2-1 Rule Only

Trajectory Length:

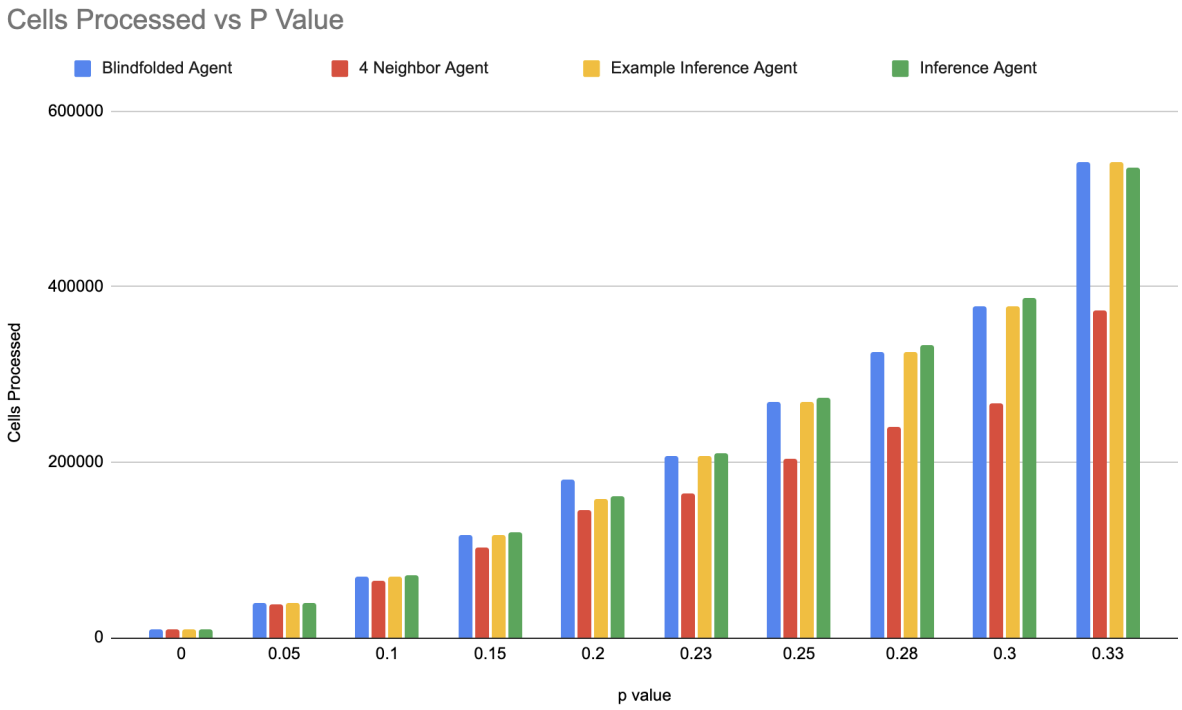
p value	Solvable: /100	Blindfolded Agent (Agent 1)	4 Neighbor Agent (Agent 2)	Example Inference Agent (Agent 3)	Inference Agent (Agent 4)
0.0	100	201	201	201	201
0.05	100	213	213	213	213
0.1	97	224	224	227	226
0.15	96	242	242	248	247
0.20	80	264	264	276	276
0.23	72	287	288	307	305
0.25	70	302	303	329	327
0.28	56	332	333	367	363
0.3	44	342	343	390	387
0.33	28	397	404	479	470



Cells Processed:

p value	Solvable: /100	Blindfolded Agent (Agent 1)	4 Neighbor Agent (Agent 2)	Example Inference Agent (Agent 3)	Inference Agent (Agent 4)
0.0	100	10201	10201	10201	10201
0.05	100	39224	38244	39224	39410
0.1	97	69638	64063	69638	70546
0.15	96	117680	102373	117680	119569
0.20	80	180083	145534	158311	160890
0.23	72	207055	164908	207055	209765
0.25	70	268024	203776	268024	273708
0.28	56	325695	240671	325695	333122
0.3	44	377967	267478	377967	386416

0.33	28	541694	373613	541694	534983



Analysis

On average, the inference agent (agent 4), using only the 1-2-1 rule yields shorter trajectory lengths than the example inference agent (agent 3). At *p* values of 0.0 to 0.1, the example inference agent and the inference returned trajectories of exactly the same length. This is likely because at *p* values, of 0.0 to 0.1, there are very few blocks in the full gridworld and all representations use A* to plan their paths, even if they cannot see the blocks to begin with. This also explains why at *p* values of 0.0 and 0.05, the example inference agent and the inference agent have the same trajectory as the blindfolded agent and the 4 neighbor agent.

Beginning at *p* of 0.1, the inference agent performed better than the example inference agent for most *p* values or the same (but never worse) because it was able to predict blocks earlier and replan its path earlier than the example inference agent. On average, the trajectory lengths produced by the inference agents are longer. We suspect that this may be because the inference agents are allowed to make “wrong inferences,” and if they infer that a cell may be blocked even though it is not actually blocked, they will try to avoid it, thereby increasing the length of the trajectory.

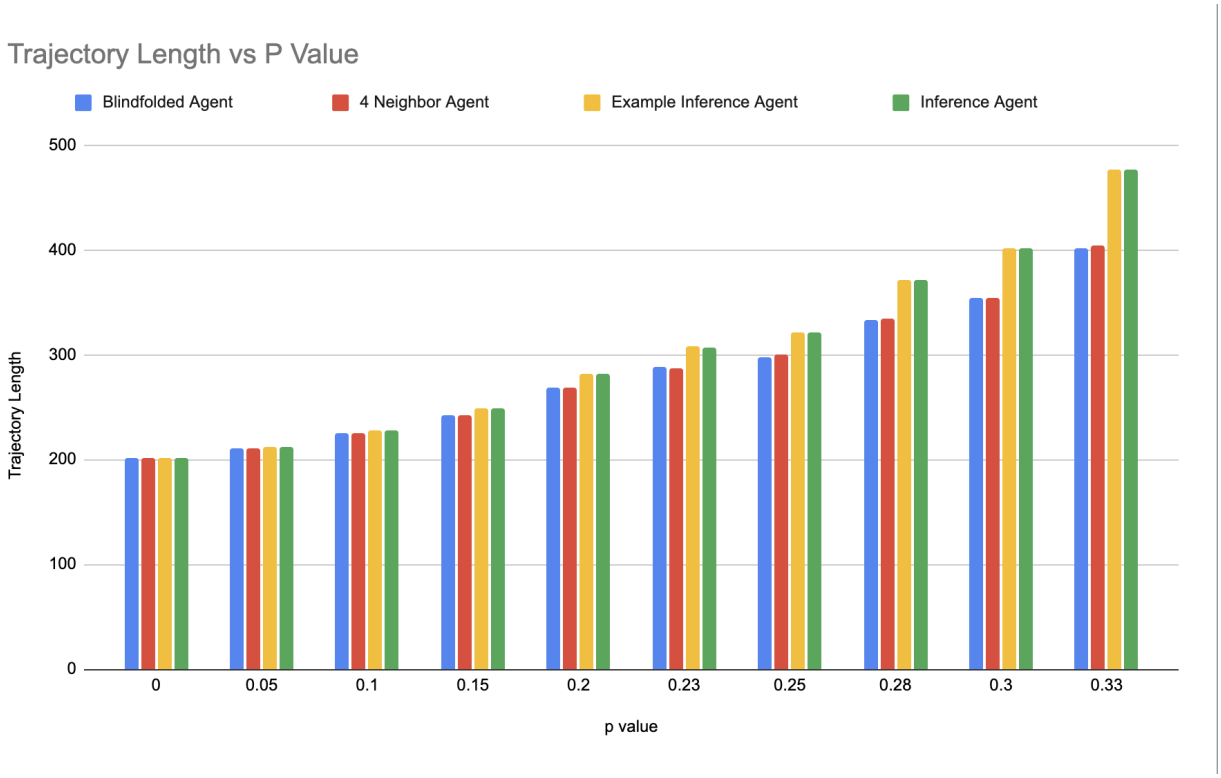
On average, looking at the total number of cells processed, the blindfolded agent, example inference agent, and inference agent all processed about the same number of cells, give or

take, with the inference agent skewing toward processing slightly more cells. This makes sense because the inference agent is making the same inferences as the example inference agent and more, and so it may rerun the generation of the path more times, increasing the number of cells processed. Overall, when comparing the number of cells processed, the example agent and the inference agent perform significantly worse than the 4-neighbor agent, which also makes sense as the 4-neighbor agent is able to see its immediate neighbors and consistently confirm blocks without having to move directly into any blocks. We suspect that the example agent (agent 3) often performs the same as the blindfolded agent, especially in cases where it is not making many inferences.

1-2-2-1 Rule Only

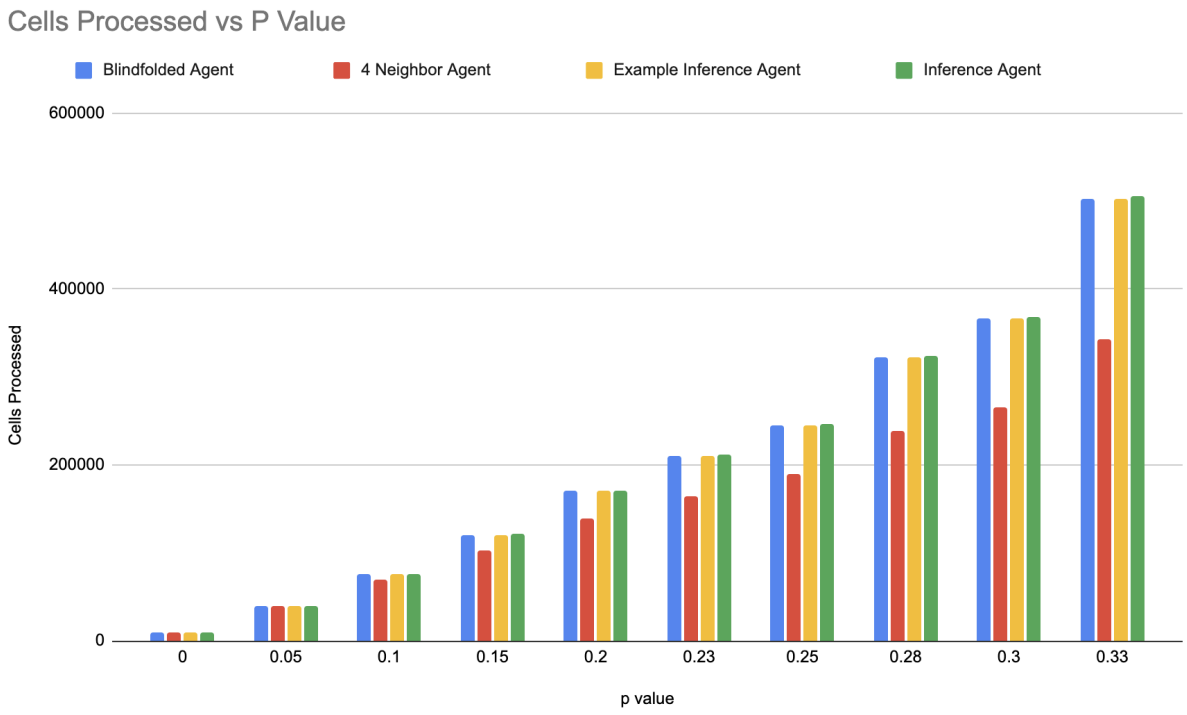
Trajectory Length:

p value	Solvable: /100	Blindfolded Agent (Agent 1)	4 Neighbor Agent (Agent 2)	Example Inference Agent (Agent 3)	Inference Agent (Agent 4)
0.0	100	201	201	201	201
0.05	99	211	211	212	212
0.1	97	225	225	228	228
0.15	90	242	242	249	249
0.20	85	269	269	282	282
0.23	85	288	287	308	307
0.25	68	298	300	321	321
0.28	54	333	335	371	371
0.3	52	354	354	402	402
0.33	37	402	404	477	476



Cells Processed:

p value	Solvable: /100	Blindfolded Agent (Agent 1)	4 Neighbor Agent (Agent 2)	Example Inference Agent (Agent 3)	Inference Agent (Agent 4)
0.0	100	10201	10201	10201	10201
0.05	99	40027	38836	40027	40027
0.1	97	75908	69617	75908	76504
0.15	90	120874	102731	120874	121259
0.20	85	170597	139833	170597	171318
0.23	85	210772	163815	210772	211951
0.25	68	245266	189045	245266	246430
0.28	54	322416	238886	322416	323304
0.3	52	366246	265439	366246	368364
0.33	37	502916	343078	502916	506337



Analysis

The 1-2-2-1 rule on its own provides less consistent results than the 1-2-1 rule on its own. Though we do see the inference agent yielding smaller trajectories than the example inference agent at *p* values 0.23 and 0.33, for the most part, the average trajectory length yielded by the inference agent is consistent with the example inference agent. This is likely because the pattern 1-2-2-1 appears so infrequently in the gridworld. Additionally, the 1-2-2-1 rule is relatively stricter to find mines, which we also have to consider if one side of the four cells adjacent to 1-2-2-1 are unvisited and the other side of them are visited. This leads the agent to check if the 12 cells meet the condition whenever we find a pattern, which is too strict to meet and we may just infer the same way as example inference agent. When testing on individual gridworlds, we saw that the inference agent using only the 1-2-2-1 rule does yield a shorter trajectory, generally by at least 1 node, when compared against the example inference agent. However, this pattern appears so infrequently that likely is not a good measure of inference for gridworlds of dimension 101.

Both Agents 3 and 4 yield longer trajectory lengths than Agents 1 and 2 as density grows. While the trajectory lengths are not significantly longer, this may tell us that agents 3 and 4 are inferring blocked cells where there are none and preemptively blocking off cells that we should be able to traverse through.

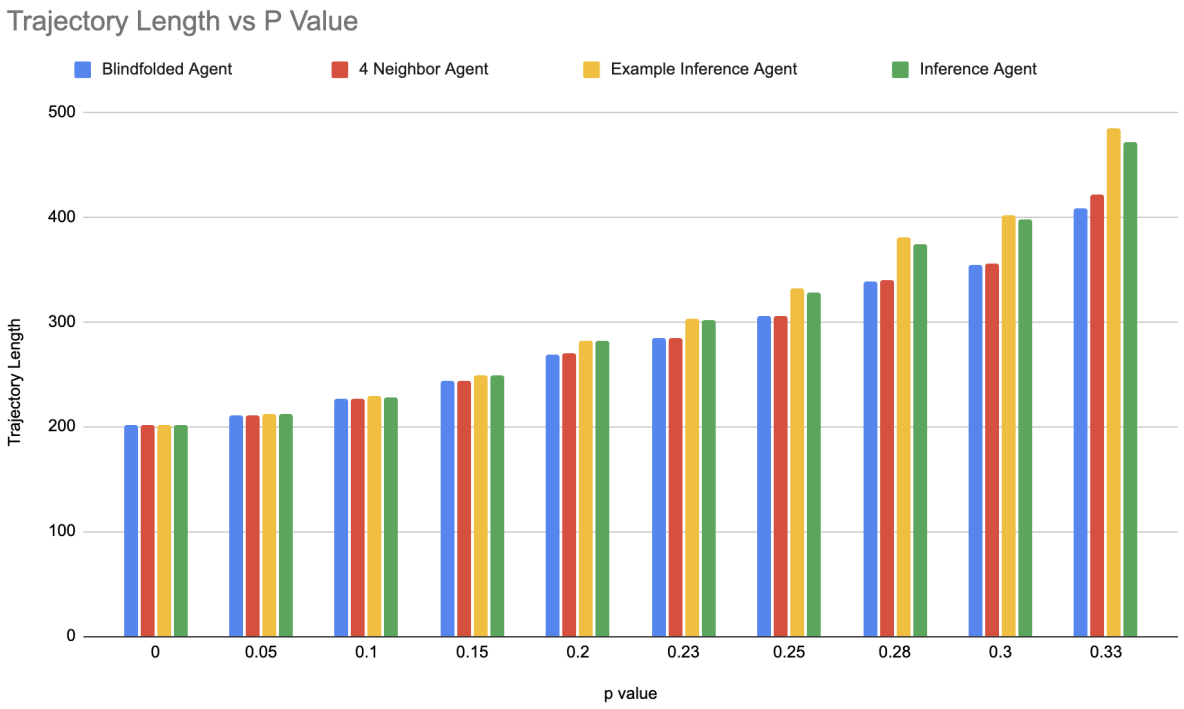
In terms of the total numbers of cells processed, we can find that from p value of 0.1, the inference agent starts to consistently process more cells than the example inference agent

does. Besides, among the 4 agents, the inference agent processes the most cells, which is reasonable because in order to make additional inferences Agent 3 makes, the 1-2-2-1 rule requires processing more cells. Similar to the inference agent applying the 1-2-1 rule, once it infers a block, it replans the path, which results in processing more cells than the example inference agent does. We can also notice that Agent 1, Agent 3 and Agent 4 process far more cells than Agent 2, which again demonstrates that the 4-neighbor agent could process less cells due to its ability to confirm all four cardinal neighbors at once without actually visiting.

Both Rules

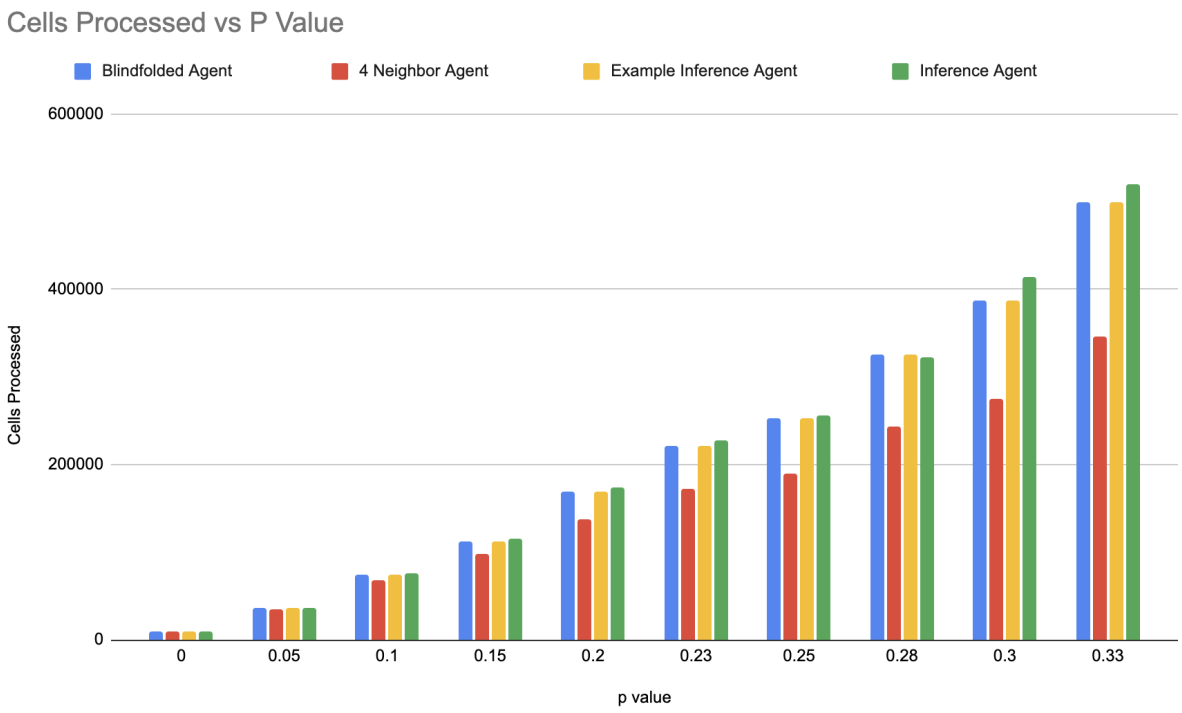
Trajectory Length:

p value	Solvable: /100	Blindfolded Agent (Agent 1)	4 Neighbor Agent (Agent 2)	Example Inference Agent (Agent 3)	Inference Agent (Agent 4)
0.0	100	201	201	201	201
0.05	98	211	211	212	212
0.1	97	226	226	229	228
0.15	91	243	243	249	249
0.20	82	269	270	282	282
0.23	74	284	285	303	301
0.25	63	306	306	332	328
0.28	51	338	340	380	374
0.3	42	354	355	402	398
0.33	25	408	421	485	471



Cells Processed:

p value	Solvable: /100	Blindfolded Agent (Agent 1)	4 Neighbor Agent (Agent 2)	Example Inference Agent (Agent 3)	Inference Agent (Agent 4)
0.0	100	10201	10201	10201	10201
0.05	98	35768	34502	35768	36192
0.1	97	74365	68598	74365	76401
0.15	91	112526	97690	112526	115480
0.20	82	169535	137720	169535	173438
0.23	74	221499	171714	221499	226866
0.25	63	252390	189937	252390	256359
0.28	51	326069	242654	326069	322600
0.3	42	387040	275512	387040	414369
0.33	25	499428	346373	499428	519679



Analysis

Overall, the agent using both the 1-2-1 rule and the 1-2-2-1 rule yields more consistent results than either of the rules on their own. Similar to the previous tests, the inference agent with both rules returns the same trajectory as the Agent 1, 2, and 3 at *p* value of 0 to 0.1 because very few cells are blocked in the full gridworld. Starting from *p* value of 0.15, excluding *p* value 0.33, we can see that the average trajectory length generated by the inference agent is consistently lower than the example inference agent. This is likely because it is able to detect blocks earlier and replan far before it would bump into these blocks, which Example Agent 3 is not always able to do because it is not looking for specific patterns which imply blocks. At *p* of 0.33, however, we can find the only case that the inference agent has a longer trajectory length than the example inference agent. The reason may be that the inference agent infers more wrong blocks in this testing. Yet, from an objective perspective, the inference agent still performs better than the example inference agent regarding the other *p* values, as most of its trajectory lengths are consistently less than the example inference agent, and the case of *p* value of 0.3, where there were only 25 solvable gridworlds, could be determined as an outlier.

The agent with both rules appears to have more consistent results than 1-2-2-1 by itself. Based on our analysis of the trajectory length of the two rules respectively, we know that the 1-2-2-1 pattern appears too infrequently and that the rules to consider are too strict as well. Therefore, we may imply that the performance of the inference agent with both rules is mostly contributed by the 1-2-1 rule.

The agent using both rules together does not process significantly more cells, on average, than the 1-2-1 rule on its own or the 1-2-2-1 rule on its own. However, compared with other agents on the same gridworlds, the inference agent using both rules appears to process more cells. Where, in the previous agents, we saw small increases in the number of cells processed using agent 4, when we use both rules, we see more significant increases in the number of cells processed especially as *p* increases. It is possible that a higher density gridworld results in more inferences being made using either the 1-2-1 or 1-2-2-1 rule, causing the agent to replan its path more often on average. It is also possible that the significant increase in cells processed by Agent 4 at p values of 0.3 and 0.33 is an outlier due to the smaller number of solvable gridworlds.

Appendix

```
def example_inference_agent(dim, gridworld, heuristic):
    start = (0,0)
    goal = (dim-1,dim-1)
    cells_processed = 0
    gridworldcopy = astar.generategridworld(dim, 0)

    info_grid = [[[None, None, None, 0, 0, None] for i in range(dim)] for j in range(dim)]
    """
    info_grid keeps track of information from each cell:

    info_grid[row][col]: (blocked, nx, cx, bx, ex, hx)

    - blocked: boolean value, 0 unblocked, 1 blocked, None if unvisited
    - nx: number of neighbors
    - cx: number of neighbors sensed to be blocked
    - bx: number of neighbors confirmed to be blocked
    - ex: number of neighbors confirmed to be empty
    - hx: number of neighbors not confirmed to be blocked or empty
    """
    astarresult = astar.astarwithstart(dim, gridworldcopy, start, heuristic)
    cells_processed += astarresult[0]
    path = astarresult[1]
    finalpath = []
    prevNode = (-1,-1)
    currNode = (0,0)

    i = 0
    while ((currNode[0] != dim-1) and (currNode[1] != dim-1)) or ((currNode[0] != -1) and (currNode[1] != -1)):
        if (currNode[0] == dim-1) and (currNode[1] == dim-1):
            break
        row = currNode[0]
        col = currNode[1]
        # blocked, nx, cx always recalculated
        blocked = None
        nx = 0
        cx = 0
        # bx, ex, hx come from dictionary if available
        bx = 0
        ex = 0
        hx = 0

        # check if currnode is blocked
        if (gridworld[currNode[0]][currNode[1]] == 1):
```

```
    blocked = 1
    gridworldcopy[row][col] = 1
else:
    blocked = 0
nx = calculate_nx(currNode, dim)
cx = calculate_cx(currNode, gridworld, dim)
neighbor_list = generate_neighbor_list(currNode, dim)

for neighbor in neighbor_list:
    info_grid[neighbor[0]][neighbor[1]][1] = calculate_nx(neighbor, dim)
    if info_grid[neighbor[0]][neighbor[1]][0] == 1:
        bx+=1
    elif info_grid[neighbor[0]][neighbor[1]][1] == 0:
        ex+=1

hx = nx - bx - ex

info_grid[currNode[0]][currNode[1]] = [blocked, nx, cx, bx, ex, hx]

update_result = (info_grid, gridworldcopy)

if (info_grid[currNode[0]][currNode[1]][0] is None):
    if blocked == 1:
        update_result = update_neighbors_bx(currNode, info_grid, gridworldcopy, dim)
    elif blocked == 0:
        update_result = update_neighbors_ex(currNode, info_grid, gridworldcopy, dim)

info_grid = update_result[0]
gridworldcopy = update_result[1]

"""
if blocked:
    get new path with previous node as start, gridworldcopy as gridworld
    currNode is second node in new path
else:
    add node to final path
    make the previous node the current nodes
    make curr node the next node in the path
"""
#checking all nodes in the planned path to find any blocks
replan_path = False
for node in path:
    if gridworldcopy[node[0]][node[1]] == 1:
        replan_path = True
        break

if (blocked == 1 or replan_path == True):
    if (blocked == 1):
        astarresult = astar.astarwithstart(dim, gridworldcopy, prevNode, heuristic)
        cells_processed += astarresult[0]
        path = astarresult[1]
    else:
        # cell not blocked, but still need to replan
        astarresult = astar.astarwithstart(dim, gridworldcopy, currNode, heuristic)
        cells_processed += astarresult[0]
        path = astarresult[1]
    if (len(path) == 0):
        break
    elif (len(path) == 1):
        if (path[0][0] == dim-1) and (path[0][1] == dim-1):
            finalpath.append(path[0])
```

```
                return(len(finalpath, cells_processed, gridworldcopy, finalpath))
            else:
                i = 1
                currNode = path[i]
        else:
            # cell is not blocked
            finalpath.append(currNode)
            prevNode = currNode
            currNode = path[i+1]
            i = i+1
    if (len(path) != 0):
        finalpath.append(currNode)
    else:
        return(-1, cells_processed, gridworldcopy, finalpath)
    return(len(finalpath), cells_processed, gridworldcopy, finalpath)

def inference_agent(dim, gridworld, heuristic):
    start = (0,0)
    goal = (dim-1,dim-1)
    cells_processed = 0
    gridworldcopy = astar.generategridworld(dim, 0)

    info_grid = [[[None, None, None, 0, 0, None, None] for i in range(dim)] for j in range(dim)]
    """
    info_grid keeps track of information from each cell:
    info_grid[row][col]: (blocked, nx, cx, bx, ex, hx)
    - blocked: boolean value, 0 unblocked, 1 blocked, None if unvisited
    - nx: number of neighbors
    - cx: number of neighbors sensed to be blocked
    - bx: number of neighbors confirmed to be blocked
    - ex: number of neighbors confirmed to be empty
    - hx: number of neighbors not confirmed to be blocked or empty
    - cbx: cx - bx
    """
    astarresult = astar.astarwithstart(dim, gridworldcopy, start, heuristic)
    cells_processed += astarresult[0]
    path = astarresult[1]
    finalpath = []
    prevNode = (-1,-1)
    currNode = (0,0)

    i = 0
    while ((currNode[0] != dim-1) and (currNode[1] != dim-1)) or ((currNode[0] != -1) and (currNode[1] != -1)):
        if (currNode[0] == dim-1) and (currNode[1] == dim-1):
            break
        row = currNode[0]
        col = currNode[1]
        blocked = None
        nx = 0
        cx = 0
        bx = 0
        ex = 0
        hx = 0
        cbx= 0

        # check if currnode is blocked
        if (gridworld[currNode[0]][currNode[1]] == 1):
            blocked = 1
            gridworldcopy[row][col] = 1
        else:
            blocked = 0
```

```
nx = calculate_nx(currNode, dim)
cx = calculate_cx(currNode, gridworld, dim)
neighbor_list = generate_neighbor_list(currNode, dim)

for neighbor in neighbor_list:
    info_grid[neighbor[0]][neighbor[1]][1] = calculate_nx(neighbor, dim)
    if info_grid[neighbor[0]][neighbor[1]][0] == 1:
        bx+=1
    elif info_grid[neighbor[0]][neighbor[1]][1] == 0:
        ex+=1

hx = nx - bx - ex
cbx = cx-bx

info_grid[currNode[0]][currNode[1]] = [blocked, nx, cx, bx, ex, hx, cbx]

update_result = (info_grid, gridworldcopy)

if (info_grid[currNode[0]][currNode[1]][0] is None):
    if blocked == 1:
        update_result = update_neighbors_bx(currNode, info_grid, gridworldcopy, dim)
    elif blocked == 0:
        update_result = update_neighbors_ex(currNode, info_grid, gridworldcopy, dim)

info_grid = update_result[0]
gridworldcopy = update_result[1]

"""
if blocked:
    get new path with previous node as start, gridworldcopy as gridworld
    currNode is second node in new path
else:
    add node to final path
    make the previous node the current nodes
    make curr node the next node in the path
"""
#checking all nodes in the planned path to find any blocks

one_two_one_result = one_two_one(currNode, info_grid, gridworldcopy, dim)
found_mines = one_two_one_result[0]
info_grid = one_two_one_result[1]
gridworldcopy = one_two_one_result[2]


one_two_two_one_result = one_two_two_one(currNode, info_grid, gridworldcopy, dim)
found_mines_two_two = one_two_one_result[0]
info_grid = one_two_one_result[1]
gridworldcopy = one_two_one_result[2]

replan_path = False
for node in path:
    if gridworldcopy[node[0]][node[1]] == 1:
        replan_path = True
        break

if (len(found_mines) > 0):
    replan_path = True

if (len(found_mines_two_two) > 0):
    replan_path = True
```

```
        if (blocked == 1 or replan_path == True):
            if(blocked == 1):
                astarresult = astar.astarwithstart(dim, gridworldcopy, prevNode, heuristic)
                cells_processed += astarresult[0]
                path = astarresult[1]
            else:
                # cell not blocked, but still need to replan
                astarresult = astar.astarwithstart(dim, gridworldcopy, currNode, heuristic)
                cells_processed += astarresult[0]
                path = astarresult[1]
            if (len(path) == 0):
                break
            elif (len(path) == 1):
                if (path[0][0] == dim-1) and (path[0][1] == dim-1):
                    finalpath.append(path[0])
                    return(len(finalpath, cells_processed, gridworldcopy, finalpath))
            else:
                i = 1
                currNode = path[i]
        else:
            # cell is not blocked
            finalpath.append(currNode)
            prevNode = currNode
            currNode = path[i+1]
            i = i+1
    if (len(path) != 0):
        finalpath.append(currNode)
    else:
        return(-1, cells_processed, gridworldcopy, finalpath)
    return(len(finalpath), cells_processed, gridworldcopy, finalpath)

def update_neighbors_bx(cell, info_grid, gridworld_copy, dim):
    neighbor_list = generate_neighbor_list(cell, dim)
    for neighbor in neighbor_list:
        info_grid[neighbor[0]][neighbor[1]][1] = calculate_nx(neighbor, dim)
        info_grid[neighbor[0]][neighbor[1]][3]+=1
        #since bx of neighbor changed, we check for cx==bx type inference
        inference_result = cx_bx_inference(neighbor, info_grid, gridworld_copy, dim)
        info_grid = inference_result[0]
        gridworld_copy = inference_result[1]
    return (info_grid, gridworld_copy)

def update_neighbors_ex(cell, info_grid, gridworld_copy, dim):
    neighbor_list = generate_neighbor_list(cell, dim)
    for neighbor in neighbor_list:
        info_grid[neighbor[0]][neighbor[1]][1] = calculate_nx(neighbor, dim)
        info_grid[neighbor[0]][neighbor[1]][4]+=1
        #since ex of neighbor changed, we check for nx-cx==ex type inference
        inference_result = nx_cx_ex_inference(neighbor, info_grid, gridworld_copy, dim)
        info_grid = inference_result[0]
        gridworld_copy = inference_result[1]
    return (info_grid, gridworld_copy)

def cx_bx_inference(cell, info_grid, gridworld_copy, dim):
    #if cx = bx
    if (info_grid[cell[0]][cell[1]][2] is not None):
        if (info_grid[cell[0]][cell[1]][2] == info_grid[cell[0]][cell[1]][3]):
            #print("cx == bx inference made!")
            neighbor_list = generate_neighbor_list(cell, dim)
            for neighbor in neighbor_list:
                #all unconfirmed neighbors of cell are unblocked
```

```
            if info_grid[neighbor[0]][neighbor[1]][0] is None:
                info_grid[neighbor[0]][neighbor[1]][0] = 0
                gridworld_copy[neighbor[0]][neighbor[1]] = 0
                update_result = update_neighbors_ex(neighbor, info_grid, gridworld_copy, dim)
                info_grid = update_result[0]
                gridworld_copy = update_result[1]
    return (info_grid, gridworld_copy)

def nx_cx_ex_inference(cell, info_grid, gridworld_copy, dim):
    #if nx - cx = ex
    if (info_grid[cell[0]][cell[1]][2] is not None):
        if (info_grid[cell[0]][cell[1]][1] - info_grid[cell[0]][cell[1]][2]) == info_grid[cell[0]][cell[1]][4]:
            #print("nx - cx == ex inference made!")
            neighbor_list = generate_neighbor_list(cell, dim)
            for neighbor in neighbor_list:
                #all unconfirmed neighbors of cell are blocked
                if info_grid[neighbor[0]][neighbor[1]][0] is None:
                    info_grid[neighbor[0]][neighbor[1]][0] = 1
                    gridworld_copy[neighbor[0]][neighbor[1]] = 1
                    update_result = update_neighbors_bx(neighbor, info_grid, gridworld_copy, dim)
                    info_grid = update_result[0]
                    gridworld_copy = update_result[1]
    return (info_grid, gridworld_copy)

def one_two_one(currNode, info_grid, gridworld_copy, dim):
    # order neighbors_list: [north, northEast, east, southEast, south, southWest, west, northWest]
    found_mines = []
    neighbors_list = generate_neighbor_list_with_none(currNode, dim)

    north = neighbors_list[0]
    northEast = neighbors_list[1]
    east = neighbors_list[2]
    southEast = neighbors_list[3]
    south = neighbors_list [4]
    southWest = neighbors_list[5]
    west = neighbors_list[6]
    northWest = neighbors_list[7]

    # north edge: northwest, north, northeast
    # east edge: northeast, east, southeast
    # south edge: southwest, south, southeast
    # west edge: northwest, west, southwest

    if (isChildValidRepeatedForward(northWest, dim) and isChildValidRepeatedForward(north, dim) and
isChildValidRepeatedForward(northEast, dim)):
        nw_cbx = info_grid[northWest[0]][northWest[1]][6]
        n_cbx = info_grid[north[0]][north[1]][6]
        ne_cbx = info_grid[northEast[0]][northEast[1]][6]
        if ((nw_cbx == 1) and (n_cbx == 2) and (ne_cbx == 1)):
            # if valid, blocks about nw and ne are blocked
            nw_block = (northWest[0]-1, northWest[1])
            ne_block = (northEast[0]-1, northEast[1])
            if (isChildValidRepeatedForward(nw_block, dim) and isChildValidRepeatedForward(ne_block, dim)):
                found_mines.append(nw_block)
                found_mines.append(ne_block)

                # add blocks to info grid and gridworld copy
                info_grid[nw_block[0]][nw_block[1]][0] = 1
                info_grid[ne_block[0]][ne_block[1]][0] = 1
                gridworld_copy[nw_block[0]][nw_block[1]] = 1
                gridworld_copy[ne_block[0]][ne_block[1]] = 1
```

```
            # update neighbors northwest block
            nw_block_changes = update_neighbors_bx(nw_block, info_grid, gridworld_copy, dim)
            info_grid = nw_block_changes[0]
            gridworld_copy = nw_block_changes[1]

            #update neighbors northeast block
            ne_block_changes = update_neighbors_bx(ne_block, info_grid, gridworld_copy, dim)
            info_grid = ne_block_changes[0]
            gridworld_copy = ne_block_changes[1]

    if (isChildValidRepeatedForward(northEast, dim) and isChildValidRepeatedForward(east, dim) and isChildValidRepeatedForward(southEast,
dim)):
        ne_cbx = info_grid[northEast[0]][northEast[1]][6]
        e_cbx = info_grid[east[0]][east[1]][6]
        se_cbx = info_grid[southEast[0]][southEast[1]][6]
        if ((ne_cbx == 1) and (e_cbx == 2) and (se_cbx == 1)):
            # if valid, blocks east of ne and se are blocked
            ne_block = (northEast[0], northEast[1]+1)
            se_block = (southEast[0], southEast[1]+1)
            if (isChildValidRepeatedForward(ne_block, dim) and isChildValidRepeatedForward(se_block, dim)):
                found_mines.append(ne_block)
                found_mines.append(se_block)

                # add blocks to info grid and gridworld copy
                info_grid[ne_block[0]][ne_block[1]][0] = 1
                info_grid[se_block[0]][se_block[1]][0] = 1
                gridworld_copy[ne_block[0]][ne_block[1]] = 1
                gridworld_copy[se_block[0]][se_block[1]] = 1

                # update neighbors northwest block
                ne_block_changes = update_neighbors_bx(ne_block, info_grid, gridworld_copy, dim)
                info_grid = ne_block_changes[0]
                gridworld_copy = ne_block_changes[1]

                #update neighbors northeast block
                se_block_changes = update_neighbors_bx(se_block, info_grid, gridworld_copy, dim)
                info_grid = se_block_changes[0]
                gridworld_copy = se_block_changes[1]

    if (isChildValidRepeatedForward(southWest, dim) and isChildValidRepeatedForward(south, dim) and
isChildValidRepeatedForward(southEast, dim)):
        sw_cbx = info_grid[southWest[0]][southWest[1]][6]
        s_cbx = info_grid[south[0]][south[1]][6]
        se_cbx = info_grid[southEast[0]][southEast[1]][6]
        if ((sw_cbx == 1) and (s_cbx == 2) and (se_cbx == 1)):
                # if valid, blocks south of sw and se are blocked
            sw_block = (southWest[0]+1, southWest[1])
            se_block = (southEast[0]+1, southEast[1])
            if (isChildValidRepeatedForward(sw_block, dim) and isChildValidRepeatedForward(se_block, dim)):
                found_mines.append(sw_block)
                found_mines.append(se_block)

                # add blocks to info grid and gridworld copy
                info_grid[sw_block[0]][sw_block[1]][0] = 1
                info_grid[se_block[0]][se_block[1]][0] = 1
                gridworld_copy[sw_block[0]][sw_block[1]] = 1
                gridworld_copy[se_block[0]][se_block[1]] = 1

                # update neighbors northwest block
                sw_block_changes = update_neighbors_bx(sw_block, info_grid, gridworld_copy, dim)
```

```
            info_grid = sw_block_changes[0]
            gridworld_copy = sw_block_changes[1]

            #update neighbors northeast block
            se_block_changes = update_neighbors_bx(se_block, info_grid, gridworld_copy, dim)
            info_grid = se_block_changes[0]
            gridworld_copy = se_block_changes[1]


    if (isChildValidRepeatedForward(northWest, dim) and isChildValidRepeatedForward(west, dim) and
isChildValidRepeatedForward(southWest, dim)):
        nw_cbx = info_grid[northWest[0]][northWest[1]][6]
        w_cbx = info_grid[west[0]][west[1]][6]
        sw_cbx = info_grid[southWest[0]][southWest[1]][6]
        if ((nw_cbx == 1) and (w_cbx == 2) and (sw_cbx == 1)):
            #if valid, blocks west of nw and sw are blocked
            nw_block = (northWest[0], northWest[1]-1)
            sw_block = (southWest[0], southWest[1]-1)
            if (isChildValidRepeatedForward(nw_block, dim) and isChildValidRepeatedForward(sw_block, dim)):
                found_mines.append(nw_block)
                found_mines.append(sw_block)

                # add blocks to info grid and gridworld copy
                info_grid[nw_block[0]][nw_block[1]][0] = 1
                info_grid[sw_block[0]][sw_block[1]][0] = 1
                gridworld_copy[nw_block[0]][nw_block[1]] = 1
                gridworld_copy[sw_block[0]][sw_block[1]] = 1

                # update neighbors northwest block
                nw_block_changes = update_neighbors_bx(nw_block, info_grid, gridworld_copy, dim)
                info_grid = nw_block_changes[0]
                gridworld_copy = nw_block_changes[1]

                #update neighbors northeast block
                sw_block_changes = update_neighbors_bx(sw_block, info_grid, gridworld_copy, dim)
                info_grid = sw_block_changes[0]
                gridworld_copy = sw_block_changes[1]


    return (found_mines, info_grid, gridworld_copy)

def one_two_two_one(curr_Node, info_grid, gridworldcopy, dim):
    # infer when current cell's cbx value = 2
    # order neighbors_list: [north, northEast, east, southEast, south, southWest, west, northWest]


    found_mines = []
    current_cbx = info_grid[curr_Node[0]][curr_Node[1]][6]
    if curr_Node[0]>=3 and curr_Node[0]<=dim-3 and curr_Node[1]>=4 and curr_Node[1]<=dim-3:
      if current_cbx >= 1:
        neighbors_list = generate_neighbor_list_with_none(curr_Node, dim)
        #print(curr_Node)
        #print(neighbors_list)

        north = neighbors_list[0]
        north2 = (north[0]-1, north[1])
        northEast = neighbors_list[1]
        northEast2 = (northEast[0], northEast[1]+1)
        north2East = (northEast[0]-1, northEast[1])

        east = neighbors_list[2]
```

```
        east2 = (east[0], east[1]+1)

        southEast = neighbors_list[3]
        southEast2 = (southEast[0], southEast[1]+1)
        south2East = (southEast[0]+1, southEast[1])

        south = neighbors_list [4]
        south2 = (south[0]+1, south[1])

        southWest = neighbors_list[5]
        southWest2 = (southWest[0], southWest[1]-1)
        southWest3 = (southWest[0], southWest[1]-1)
        south2West = (southWest[0]+1, southWest[1])

        west = neighbors_list[6]
        west2 = (west[0], west[1]-1)
        west3 = (west[0], west[1]-2)

        northWest = neighbors_list[7]
        northWest2 = (northWest[0], northWest[1]-1)
        northWest3 = (northWest[0], northWest[1]-2)
        north2West = (northWest[0]-1, northWest[1])

        if current_cbx == 1:
            if (isChildValidRepeatedForward(west3, dim) and isChildValidRepeatedForward(west2, dim) and isChildValidRepeatedForward(west,
dim)):
                www_cbx = info_grid[west3[0]][west3[1]][6]
                ww_cbx = info_grid[west2[0]][west2[1]][6]
                w_cbx = info_grid[west[0]][west[1]][6]

                if ((www_cbx == 1) and (ww_cbx == 2) and (w_cbx == 2)):
                    up_neighbors = [northWest3, northWest2, northWest, north]
                    low_neighbors = [southWest3, southWest2, southWest, south]

                    # if up neighbors not visited yet
                    if all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in up_neighbors):
                        # if valid, blocks about nw and nw2 are blocked
                        for block in up_neighbors[1, 2]:
                            found_mines.append(block)
                            (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)
                    # if low neighbors not visted yet
                    elif all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in low_neighbors):
                        # if valid, blocks about sw and sw2 are blocked
                        for block in low_neighbors[1, 2]:
                            (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)


        if current_cbx == 2:
            # horizontal inference

            if (isChildValidRepeatedForward(west, dim) and isChildValidRepeatedForward(east, dim) and isChildValidRepeatedForward(east2,
dim)):
                w_cbx = info_grid[west[0]][west[1]][6]
                e_cbx = info_grid[east[0]][east[1]][6]
                ee_cbx = info_grid[east2[0]][east2[1]][6]
                if ((w_cbx == 1) and (e_cbx == 2) and (ee_cbx == 1)):
                    up_neighbors = [northWest, north, northEast, northEast2]
                    low_neighbors = [southWest, south, southEast, southEast2]

                    if all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in up_neighbors):
                        # if valid, blocks about n and ne are blocked
```

```
                for block in up_neighbors[1, 2]:
                    found_mines.append(block)
                    (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)
            elif all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in low_neighbors):
                # if valid, blocks about s and se are blocked
                for block in low_neighbors[1, 2]:
                    found_mines.append(block)
                    (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)

        if (isChildValidRepeatedForward(west2, dim) and isChildValidRepeatedForward(west, dim) and isChildValidRepeatedForward(east,
dim)):
            ww_cbx = info_grid[west2[0]][west2[1]][6]
            w_cbx = info_grid[west[0]][west[1]][6]
            e_cbx = info_grid[east[0]][east[1]][6]
            if ((ww_cbx == 1) and (w_cbx == 2) and (e_cbx == 1)):
                up_neighbors = [northWest2, northWest, north, northEast]
                low_neighbors = [southWest2, southWest, south, southEast]

                # if up neighbors not visited yet
                if all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in up_neighbors):
                    # if valid, blocks about nw and n are blocked
                    for block in up_neighbors[1, 2]:
                        found_mines.append(block)
                        (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)
                # if low neighbors not visted yet
                elif all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in low_neighbors):
                    # if valid, blocks about sw and s are blocked
                    for block in low_neighbors[1, 2]:
                        (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)


        # vertical inference
        if (isChildValidRepeatedForward(north, dim) and isChildValidRepeatedForward(south, dim) and isChildValidRepeatedForward(south2,
dim)):
            n_cbx = info_grid[north[0]][north[1]][6]
            s_cbx = info_grid[south[0]][south[1]][6]
            ss_cbx = info_grid[south2[0]][south[1]][6]
            if ((n_cbx == 1) and (s_cbx == 2) and (ss_cbx == 1)):
                right_neighbors = [northEast, east, southEast, south2East]
                left_neighbors = [northWest, west, southWest, south2West]

                # if right neighbors not visited yet
                if all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in right_neighbors):
                    # if valid, blocks about e and se are blocked
                    for block in right_neighbors[1, 2]:
                        found_mines.append(block)
                        (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)

                # if low neighbors not visted yet
                elif all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in left_neighbors):
                    # if valid, blocks about w and sw are blocked
                    for block in left_neighbors[1, 2]:
                        found_mines.append(block)
                        (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)

        if (isChildValidRepeatedForward(north2, dim) and isChildValidRepeatedForward(north, dim) and isChildValidRepeatedForward(south,
dim)):
            nn_cbx = info_grid[north2[0]][north2[1]][6]
            n_cbx = info_grid[north[0]][north[1]][6]
            s_cbx = info_grid[south[0]][south[1]][6]
            if ((nn_cbx == 1) and (n_cbx == 2) and (s_cbx == 1)):
```

```
            right_neighbors = [north2East, northEast, east, southEast]
            left_neighbors = [north2West, northWest, west, southWest]

            # if right neighbors not visited yet
            if all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in right_neighbors):
                # if valid, blocks about e and se are blocked
                for block in right_neighbors[1, 2]:
                    found_mines.append(block)
                    (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)

            # if low neighbors not visted yet
            elif all(info_grid[key[0]][key[1]][2] == None and isChildValidRepeatedForward(key, dim) for key in left_neighbors):
                # if valid, blocks about w and sw are blocked
                for block in left_neighbors[1, 2]:
                    found_mines.append(block)
                    (info_grid, gridworldcopy) = update_neighbors_bx(block, info_grid, gridworldcopy, dim)

    return (found_mines, info_grid, gridworldcopy)
```