CS440 Project 2: Minesweeper

Achintya Singh, Tapan Patel

March 23, 2021

Exercise 1

How did you represent the board in your program, and how did you represent the information/ knowledge that clue cells reveal? How could you represent inferred relationships between cells?

```
import random
class Env:
    def __init__(self, d, n):
        self.d = d
        self.n = n
        data = [0]*(d*d-n)+[1]*(n)
        random.shuffle(data)
        self.board = [ [data.pop() for j in range(d)] for i in range(d)] # 0 is clear
        #TODO: randomly fill n 1s in board
    def query(self,r,c):
        """ returns -1 if MINE, else num of mined neighbors"""
        if self.board[r][c]:
            return -1
        else:
            counter = 0
            for dx in [-1,0,1]:
                for dy in [-1,0,1]:
                    if dx or dy:
                        x,y=r+dx,c+dy
                        if x>=0 and x<self.d and y>=0 and y<self.d:
                            counter += self.board[x][y]
            return counter
    def __repr__(s):
        res=''
        for i in range(s.d):
            for j in range(s.d):
                res+='{:<3}'.format(s.board[i][j])
            res+='\n'
        return res
```

In our program, the board is represented by a two-dimensional array. The knowledge is represented by three two-dimensional arrays: board, visited, and prob. The board array contains the status of a cell being mine, no mine, or unknown. Visited determines whether a given cell has been previously queried. Prob showed the probability of a given cell being a mine. As for inferred relationships between cells, we compared probabilities of neighboring cells being mines.

Exercise 2

Inference: When you collect a new clue, how do you model/process/compute the information you gain from it? In other words, how do you update your current state of knowledge based on that clue? Does your program deduce everything it can from a given clue before continuing? If so, how can you be sure of this, and if not,how could you consider improving it?

```python
class Agent:
    def __init__(self, d, n):
        self.d = d
        self.n = n
        self.board = [ [0 for j in range(d)] for i in range(d)]
        # -1 is MINE
        # 0 is UNKNOWN
        # 1 is CLEAR
        self.qh = []
        self.score = 0
        self.hits = 0
        self.prob= [ [0 for j in range(d)] for i in range(d)] # probability of seeing a
            mine here
        self.visited = [ [0 for j in range(d)] for i in range(d)] # 0 is unvisited, 1
            is known.
        self.numsides=None

    def boxit(self,box,pad=3,fmt=''):
        res=''
        if fmt:
            template='{{: {}{}}}'.format(pad,fmt)
        else:
            template='{{:>{}}}'.format(pad)
        for i in range(len(box)):
            for j in range(len(box[i])):
                res+=template.format(box[i][j])
            res+='\n'
        return res
    def joinboxes(self,boxesstr,pad='\t'):
        return '\n'.join(pad.join(line) for line in
            zip(*[boxstr.strip('\n').split('\n') for boxstr in boxesstr]))
    def getboard(self):
        board=[ ["?" for j in range(self.d)] for i in range(self.d)]
        for r in range(self.d):
            for c in range(self.d):
                try:
                    o=next(o for x,y,o in self.qh if x==r and y==c)
                    if o==-1:
                        board[r][c]='M' #mine stepped on
                    else:
                        board[r][c]=o
                except StopIteration:
                    if self.board[r][c]==-1:
                        board[r][c]='m' #mine discovered
                elif self.board[r][c]==1:
                    board[r][c]='c' #clear but unvisited
        return board

    def __repr__(s):
        return s.plotprogress()
        head=' BOARD VISITED\t\tPROBABILITY\n'
        return
            head+s.joinboxes([s.boxit(s.getboard(),2),s.boxit(s.visited,2),s.boxit(s.prob,9,'.2f')])
    def getnumsides(self,X,Y):
        if self.numsides!=None:
            return self.numsides[X][Y]
        self.numsides = [ [0 for j in range(self.d)] for i in range(self.d)]
        for r in range(self.d):
            for c in range(self.d):
                for dx in [-1,0,1]:
                    for dy in [-1,0,1]:
                        if dx or dy:
                            x,y=r+dx,c+dy
                            if x>=0 and x<self.d and y>=0 and y<self.d:
                                self.numsides[x][y]+=1
        return self.numsides[X][Y]

    def any_unknown(self):
        return any(0 in row for row in self.board) and sum(row.count(-1) for row in
            self.board)<self.n

    def getprob(s,x,y):
        # prob=-1 if already cleared
        if s.board[x][y]==1:
            return -1000
        return s.prob[x][y]#*8/s.getnumsides(x,y)

    def addprob(self,r,c,v):
        v=v/self.getnumsides(r,c)
        for dx in [-1,0,1]:
            for dy in [-1,0,1]:
                if dx or dy:
                    x,y=r+dx,c+dy
                    if x>=0 and x<self.d and y>=0 and y<self.d:
                        self.prob[x][y]+= (v/self.getnumsides(x,y)) if v<1 else v
                        # Improvement: Rather than bluntly adding prob. of 1/8 per
                            direction, consider to remove all the neighbors that are
                            already CLEAR.

    def my_8_clear(self,r,c):
        for dx in [-1,0,1]:
            for dy in [-1,0,1]:
                if dx or dy:
                    x,y=r+dx,c+dy
                    if x>=0 and x<self.d and y>=0 and y<self.d:
```

```
                self.board[x][y]=1
                self.prob[x][y]=-999
def discovermines(self):
    earned=0
    poses=[]
    for r in range(self.d):
        for c in range(self.d):
            if self.getprob(r,c)>=1:
                if self.board[r][c]==0:
                    earned+=1
                    self.board[r][c]=-1
                    poses.append([r,c,self.getprob(r,c)])

    # TODO: Re-write below check.
    unknownmines=self.n-sum(row.count(-1) for row in self.board)
    visited=sum(row.count(1) for row in self.visited)
    unvisited=sum(row.count(0) for row in self.visited)
    #if self.d**2==visited+unknownmines: # all unvisited are mines
    if unvisited<=unknownmines: # all unvisited are mines
        earned+=unknownmines
        add_poses=[[i,j,999] for i in range(self.d) for j in range(self.d) if
            self.visited[i][j]==0] # unvisited ones.
        poses+=add_poses
        for ii,jj,_ in add_poses: # declaring them as mines
            self.board[ii][jj]=-1

    self.score+=earned
    pos=' '.join('[{},{} = {}]'.format(i,j,k) for i,j,k in poses)
    return 'Discovered {} ({}) new mines without steping on it.
        Score={}'.format(earned, pos,self.score) if earned else ''

def move1(self, env, show_logs=True):
    if not show_logs:
        def pass_through(*_,**__):return True
        myprint=pass_through
    else:
        myprint=print
    if not self.any_unknown(): # check if all is done.
        mined=sum([ [ (i,j) for j in range(self.d) if self.board[i][j]==-1] for i in
            range(self.d)],[])
        myprint("\nAll mines are
            revealed.",mined,"Score=",self.score,"Hits=",self.hits)
        raise StopIteration
    consider_visiting = []

    for i in range(self.d):
        for j in range(self.d):
            # skip if mine
            if self.board[i][j]==-1:
                continue
            # skip if visited
            elif self.visited[i][j]==1:
                continue
            # CLEAR/UNKNOWN, UNVISITED:
            consider_visiting.append([i,j,self.getprob(i,j)])
    assert len(consider_visiting), "Nothing to visit :("
    # pick with lowest prob.
    qx,qy,_p=sorted(consider_visiting,key=lambda v:v[-1])[0]

    # Agent queries Env.
    myprint("\nWill query: {},{} with prob={:.3f}".format(qx,qy,_p),end='. ')
    ans = env.query(qx,qy)
    self.visited[qx][qy]=1
    self.qh.append([qx,qy,ans])

    # agent processes result
    if ans == -1:
        myprint('\n',"--"*15,"\tOops, stepped on a mine!")
        self.board[qx][qy]=-1
        self.hits+=1
    else:
        myprint("Env says", ans)
        self.board[qx][qy]=1
        if ans == 0:
            # set all neighbors as clear
            self.my_8_clear(qx,qy)
        else:
            #add probs of all neighbors:
            self.addprob(qx,qy,ans)
        # check we are determined with a mine position:
        myprint("--"*15,self.discovermines())
    return self


def colors(self,mode):
    if mode=='board':
        cmap,cdef={"M":"red","m":"tomato","c":"lime",0:"white","?":"silver"},"white"
        return [[cmap.get(c,cdef) for c in row] for row in self.getboard()]
    if mode=='visits':
        cmap,cdef={0:"grey",1:"greenyellow"},"salmon"
        return [[cmap.get(c,cdef) for c in row] for row in self.visited]
    if mode=='prob':
        sprob=[i for i in sorted(set(sum(self.prob,[]))) if i>=0 and i<=100]
        t=len(sprob)
        cmap1=lambda v:(lambda r:(r,1-r,.001))(sprob.index(v)/(t))
        cmap=lambda v:('lime' if v<0 else ('red' if v>100 else cmap1(v)))
        return [[cmap(c) for c in row] for row in self.prob]

def plotprogress(self):
    import matplotlib.pyplot as plt
    fig=plt.figure(linewidth=0,
                edgecolor='#96ABA0',
                facecolor='#FFFFFF',
                #tight_layout={'pad':1},
                 figsize=(20,1.5)
               )
        plt1,plt2,plt3=fig.subplots(1,3)
        #ax = plt.gca()
        plt1.set_axis_off()
        plt2.set_axis_off()
        plt3.set_axis_off()
        plt1.set_title("Board")
        plt1.table(cellText=self.getboard(),loc='best',cellLoc='center',cellColours=self.colors('board'
        plt2.set_title("Visited")
        plt2.table(cellText=self.visited,loc='best',cellLoc='center',cellColours=self.colors('visits'))
            1.5)
        plt3.set_title("PROBABILITY")
        probstr=[[f"{c:.4f}" for c in r]for r in self.prob]
        plt3.table(cellText=probstr,loc='best',cellLoc='center',cellColours=self.colors('prob')).scale(
            1.7)
        plt.box(on=None)
        plt.show()
        return ''

class DummyAgent(Agent):
    def __init__(self, d, n):
        super().__init__(d,n)
    def move1(self, env, show_logs=True):
        if not show_logs:
            def pass_through(*_,**__):return True
            myprint=pass_through
        else:
            myprint=print
        if not self.any_unknown(): # check if all is done.
            mined=sum([ [ (i,j) for j in range(self.d) if self.board[i][j]==-1] for i in
                range(self.d)],[])
            myprint("\nAll mines are
                revealed.",mined,"Score=",self.score,"Hits=",self.hits)
            raise StopIteration
        consider_visiting = []

        for i in range(self.d):
            for j in range(self.d):
                # skip if mine
                if self.board[i][j]==-1:
                    continue
                # skip if visited
                elif self.visited[i][j]==1:
                    continue
                # CLEAR/UNKNOWN, UNVISITED:
                consider_visiting.append([i,j,0])
        assert len(consider_visiting), "Nothing to visit :("
        # pick Any
        random.shuffle(consider_visiting)
        qx,qy,_p=consider_visiting[0]

        # Agent queries Env.
        myprint("\nWill query: {},{} with prob={:.3f}".format(qx,qy,_p),end='. ')
        ans = env.query(qx,qy)
        self.visited[qx][qy]=1
        self.qh.append([qx,qy,ans])

        # agent processes result
        if ans == -1:
            myprint('\n',"--"*15,"\tOops, stepped on a mine!")
            self.board[qx][qy]=-1
            self.hits+=1
        else:
            myprint("Env says", ans)
            self.board[qx][qy]=1
            # check if we are left with only mines:
            #left+discovered=total
            if len(consider_visiting)>1 and len(consider_visiting)-1+self.hits==self.n:
                for i,j,_ in consider_visiting[1:]:self.board[i][j]=-1
        return self

class PartialAgent(Agent):
    def __init__(self, d, n):
        super().__init__(d,n)
    def discovermines(self):
        earned=0
        poses=[]
        for r in range(self.d):
            for c in range(self.d):
                if self.getprob(r,c)>=1:
                    if self.board[r][c]==0:
                        earned+=1
                        self.board[r][c]=-1
                        poses.append([r,c,self.getprob(r,c)])

        """  # Removing ending logic.
        # TODO: Re-write below check.
        unknownmines=self.n-sum(row.count(-1) for row in self.board)
        visited=sum(row.count(1) for row in self.visited)
        unvisited=sum(row.count(0) for row in self.visited)
        #if self.d**2==visited+unknownmines: # all unvisited are mines
        if unvisited<=unknownmines: # all unvisited are mines
            earned+=unknownmines
            add_poses=[[i,j,999] for i in range(self.d) for j in range(self.d) if
                self.visited[i][j]==0] # unvisited ones.
            poses+=add_poses
            for ii,jj,_ in add_poses: # declaring them as mines
                self.board[ii][jj]=-1
        """
        self.score+=earned
        pos=' '.join('[{},{} = {}]'.format(i,j,k) for i,j,k in poses)
        return 'Discovered {} ({}) new mines without steping on it.
            Score={}'.format(earned, pos,self.score) if earned else ''

    d=3
    n=3
    env=Env(d,n)
    print(env) #1 is mine, 0 is safe IN ENV.
    ag=Agent(d,n)
    print(ag)
    for _ in range(1+d**2):
        try:
            print(ag.move1(env))
        except StopIteration:
            break
# Legend:
# ? - Unknown
# M - Mine we stepped on
# m - mine we discovered
# c - Clear, not visited
# num - visited, number of mined neighbors
```

When we collect a new clue, the three aforementioned two-dimensional arrays are updated. However, the probabilistic model is not able to deduce everything correctly. For improvements in further iterations, we should consider updating the probabilistic model so that unknown neighbors are updated in a more deterministic way as opposed to re-visiting already examined neighbors.

Exercise 3

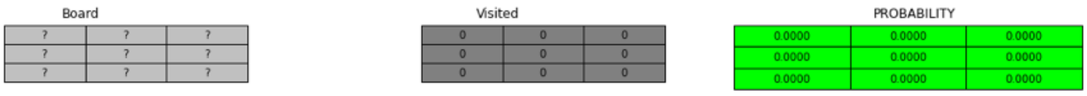
Decisions: Given a current state of the board, and a state of knowledge about the board, how does your program decide which cell to search next?

Given the current state of the board and the three states of knowledge about the board(board, visited, and prob), we are able to decide which cell to search next by identifying neighbors and their mine probability. Whichever mine has the lowest probability, that is the mine that is chosen to be next.

Exercise 4

Performance: For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why was your program able to make that decision?

Start:



Board

?	?	?
?	?	?
?	?	?

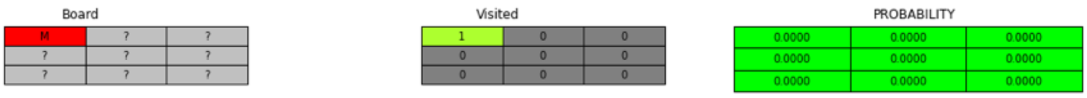
Visited

0	0	0
0	0	0
0	0	0

PROBABILITY

0.0000	0.0000	0.0000
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000

Will query: 0,0 with prob=0.000. Oops, stepped on a mine!
Second:



Board

M	?	?
?	?	?
?	?	?

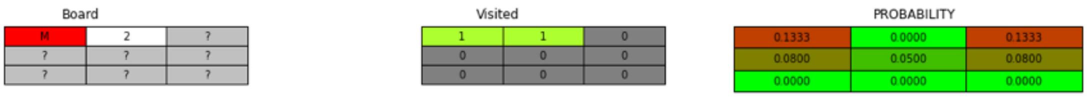
Visited

1	0	0
0	0	0
0	0	0

PROBABILITY

0.0000	0.0000	0.0000
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000

Will query: 0,1 with prob=0.000. Env says 2
Third:



Board

M	2	?
?	?	?
?	?	?

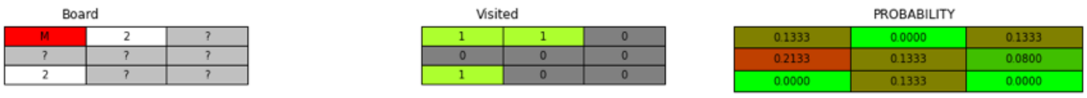
Visited

1	1	0
0	0	0
0	0	0

PROBABILITY

0.1333	0.0000	0.1333
0.0800	0.0500	0.0800
0.0000	0.0000	0.0000

Will query: 2,0 with prob=0.000. Env says 2
Fourth:



Board

M	2	?
?	?	?
2	?	?

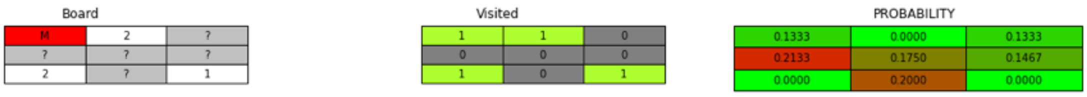
Visited

1	1	0
0	0	0
1	0	0

PROBABILITY

0.1333	0.0000	0.1333
0.2133	0.1333	0.0800
0.0000	0.1333	0.0000

Will query: 2,2 with prob=0.000. Env says 1
Fifth:



Board

M	2	?
?	?	?
2	?	1

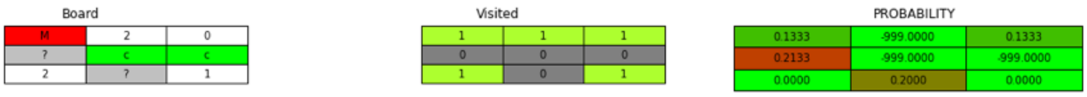
Visited

1	1	0
0	0	0
1	0	1

PROBABILITY

0.1333	0.0000	0.1333
0.2133	0.1750	0.1467
0.0000	0.2000	0.0000

Will query: 0,2 with prob=0.133. Env says 0
Sixth:



Board

M	2	0
?	c	c
2	?	1

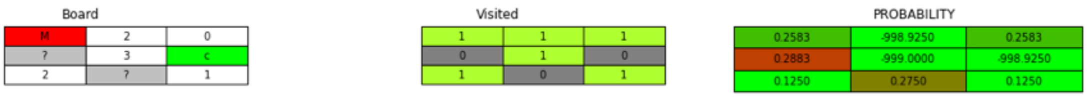
Visited

1	1	1
0	0	0
1	0	1

PROBABILITY

0.1333	-999.0000	0.1333
0.2133	-999.0000	-999.0000
0.0000	0.2000	0.0000

Will query: 1,1 with prob=-1000.000. Env says 3
Seventh:



Board

M	2	0
?	3	c
2	?	1

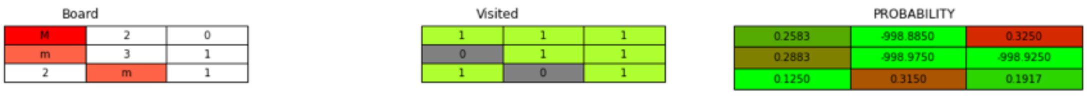
Visited

1	1	1
0	1	0
1	0	1

PROBABILITY

0.2583	-998.9250	0.2583
0.2883	-999.0000	-998.9250
0.1250	0.2750	0.1250

Will query: 1,2 with prob=-1000.000. Env says 1
Eighth:



Board

M	2	0
m	3	1
2	m	1

Visited

1	1	1
0	1	1
1	0	1

PROBABILITY

0.2583	-998.8850	0.3250
0.2883	-998.9750	-998.9250
0.1250	0.3150	0.1917

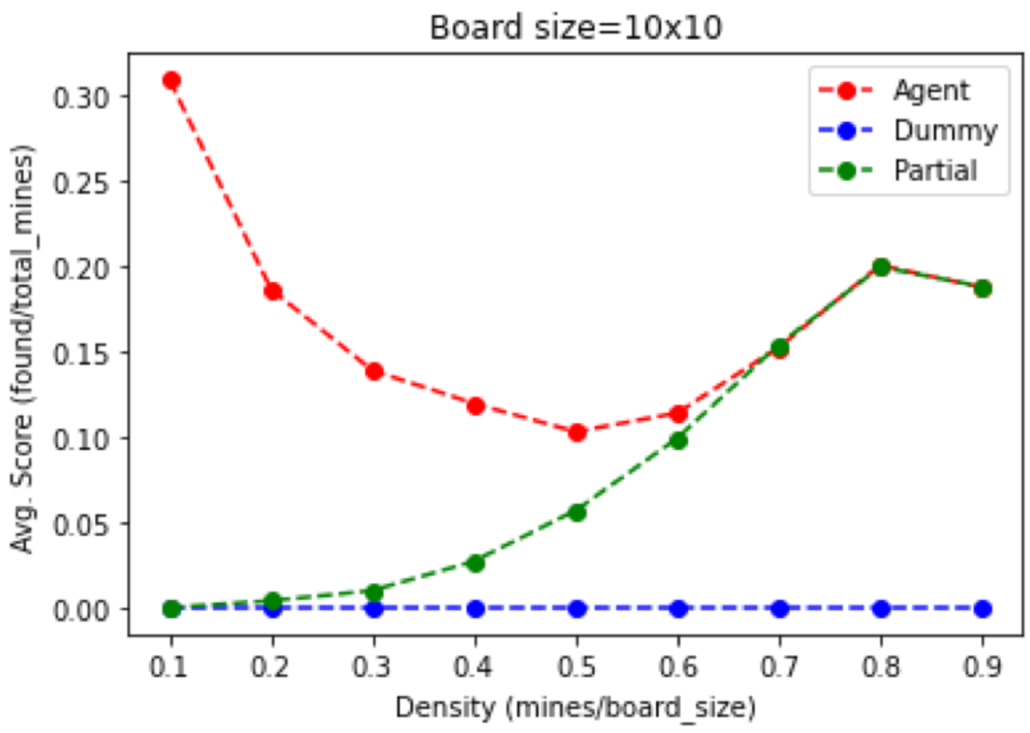
Discovered 2 ([1,0 = 999] [2,1 = 999]) new mines without stepping on it. Score=2.

```
All mines are revealed. [(0, 0), (1, 0), (2, 1)] Score= 2 Hits= 1
```

Sometimes, the agent will jump directly onto a mine, which occurred in the aforementioned play-by-play example. In any case, I disagree with this sporadic and uncalled-for movement and would suggest to improve how the program checks for mines with low probability. On the other hand, the probabilistic model avoids mines that a human like me could sometimes fall for. There is a level of ingenuity to be appreciated behind these models.

Exercise 5

Performance: For a fixed, reasonable size of board, plot as a function of mine density the average final score(safely identified mines / total mines) for the simple baseline algorithm and your algorithm for comparison. This will require solving multiple random boards at a given density of mines to get good average score results. Does the graph make sense/agree with your intuition? When does minesweeper become 'hard'? When does your algorithm beat the simple algorithm, and when is the simple algorithm better? Why? How frequently is your algorithm able to work out things that the basic agent cannot?



The graph does support our intuition that as the density increases, so does our score. In order to plot our data, we ran the program multiple times, each time using a different model to examine performance. Our "dummy" model only contained the global information(number of mines at run-time) and therefore had the worst performance of the three as it did not contain any probabilistic logic to determine where mines could be. Next, we tested the "partial" model wherein we only utilized the probabilistic model and not the global information. In this case, we noticed that the score is directly proportional to the density, implying that an increased density can provide more knowledge about potential mine locations through neighbor traversals. Lastly, our "agent" model utilized both global information and the probabilistic model to determine mine location and achieved the highest success rate, even with low density. This is because at a low density, it was able to leverage the global information in order to assess lesser known mines in order to counteract the low accuracy by the probabilistic model. Our agent model beats the dummy model at all densities and the partial model at low density. Despite this, however, our agent model began to struggle around medium density.

```
## Measuring:
def generate_1p(dim,dens,exps=None,is_dummy=False):
    #dens=n/d2
    d=dim
    n=dens*d*d
    assert int(n)==n
    n=int(n)
    exps=exps or d*d
    tscore=0
    for ei in range(exps):
        env=Env(d,n)
        try:
            ag=((PartialAgent if is_dummy=='partial' else DummyAgent) if is_dummy else
                Agent)(d,n)
            while True:
                #print("move")
                ag.move1(env,show_logs=False)
        except StopIteration:
            tscore+=ag.score
            #print("Exp",ei,"Score=",ag.score,"Final",tscore)
    return tscore/(exps*n)

def generate_all(dim,exps="**2"):
    import matplotlib.pyplot as plt
    space=[round(x * 0.1, 1) for x in range(1, 10)]
    exps=eval("dim"+exps)

    #main agent
    data=[(dens,generate_1p(dim,dens,exps=exps)) for dens in space]
    plt.plot(*zip(*data),'--ro',label="Agent")
    #dummy agent
    data=[(dens,generate_1p(dim,dens,exps=exps,is_dummy=True)) for dens in space]
    plt.plot(*zip(*data),'--bo',label="Dummy")
    #PartialAgent agent
    data=[(dens,generate_1p(dim,dens,exps=exps,is_dummy='partial')) for dens in space]
    plt.plot(*zip(*data),'--go',label="Partial")

    #plotting
    plt.title(f"Board size={dim}x{dim}")
    plt.xlabel(f"Density (mines/board_size)")
    plt.ylabel(f"Avg. Score (found/total_mines)")
    plt.legend()
    plt.show()

# Commented out IPython magic to ensure Python compatibility.
# %%time
```

Exercise 6

Efficiency: What are some of the space or time constraints you run into in implementing this program? Are these problem specific constraints, or implementation specific constraints? In the case of implementation constraints, what could you improve on?

Implementation-Specific Constraints

- Two-dimensional arrays for knowledge bases.
  - Insertion
  - Deletion
  - Traversal
  - Storage

To improve our implementation-specific constraints, we should look to storing the information in other, more compact and performance-efficient data structures or revisit our approach to storing information as such in our knowledge bases. For example, relational information such as a visited cell can be derived without a costly two-dimensional array.

Problem-Specific Constraints

- Computational Power
  - Cell Traversal
  - Flagging Mines
  - Plotting Graphs

Plotting the graph from exercise 5 took four minutes, which, given the board size inputted, makes sense because until all the mines are flagged correctly, most cells, if not all, will have to be visited. Regardless, this is a problem-specific constraint.

Exercise 7

Global Information: Suppose you were told in advance how many mines are on the board. Include this in your knowledge base. How did you model this? Regenerate the plot of mine density vs. average final score with this extra information, and analyze the results.

Create a global instance variable called global that stores the number of mines on the board. Every time a mine is encountered or the program determines a high probability of a cell with a mine, a temporary instance of the variable is decreased by 1. While this will negatively impact performance, it will marginally improve the success rate of the probabilistic model.

*See exercise 5 for plot