# Minesweeper: Knowledge Bases & Logical Inference

Authors:

1. **Rohan Rele** (rsr132)
2. **Aakash Raman** (abr103)
3. **Alex Eng** (ame136)
4. **Adarsh Patel** (aap237)

This project was completed for Professor Wes Cowan's Fall 2019 offering of the CS 520: Intro to Artificial Intelligence course, taught at Rutgers University, New Brunswick.

# Baseline Algorithm

We begin the assignment by implementing the basic solver agent algorithm that relies solely on local inference and iterative deduction.

## Implementation

Throughout this assignment, we maintain a very object-oriented approach. The baseline case includes two main component: the Agent class and the MineSweeper class. The Agent represents the "player" of the game while the MineSweeper class represents the game as it transitions through its different states. Below are the class details for the two.

### Agent class

In [1]:

```
class agent:
    def __init__(self, game, order):
        # copy game into agent object
        self.game = deepcopy(game)
        # track player knowledge per cell: initially all hidden
        self.playerKnowledge = np.array([[HIDDEN] * self.game.dim for _ in range
(self.game.dim)])

        # keep track of these throughout game to yield final mineFlagRate
        self.numFlaggedMines = 0
        self.numDetonatedMines = 0

        # these are the two metrics we can judge performance with
        self.mineFlagRate = 0
        self.totalSolveTime = 0
        self.logging = False

        # used (later) to control for variance in random selection when comparin
g agents
        self.order = order
        self.current_in_order = 0

        # used (later) to toggle uncertainty type for clues
        self.uncertaintyType = 'none'
```

As we can see, the agent's members include:

1. **`game` (Type: Minesweeper):** A MineSweeper object, which specifies the current instance of the game that agent is playing on
2. **`playerKnowledge` (Type: Numpy Array 2d):** An integer matrix that keeps track of what cells have been queried, their *value* (vicinity to a mine) and/or their "mine status"

And the rest are simple metrics used to track the agent's progress:

1. **`numFlaggedMines` (Type: Integer):** Number of flagged mines
2. **`numDetonatedMines` (Type: Integer):** Number of detonated mines
3. **`mineFlagRate` (Type: Integer):** Ratio between safely flagged mines and total mines present
4. **`totalSolveTime` (Type: Integer):** Time taken to solve
5. **`logging` (Type: Boolean):** Toggle the logger output to .txt file
6. **`order` (Type: List):** A fixed (but random) order of cell coordinates that we use to minimize variance of repeated trials
7. **`current_in_order` (Type: Integer)**: index of current 'random' cell coordinate in `order`, used for iteration through `order`
8. **`uncertaintyType` (Type: String):** Uncertainty toggle for the bonus: when the clue is randomly revealed, optimistic, or cautious. Defaults to 'none' i.e. a revealed clue is accurate

See `Agent.py` for more implementation details.

## MineSweeper class

In [2]:

```
dirs = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (-1, 1), (1, -1)]

MINE = -6
DETONATED = -8
HIDDEN = -1
SAFE = 0

class MineSweeper:
    dim = 0
    num_mines = 0
    board = [[]]
    success = False

    def __init__(self, dim, num_mines):
        self.dim = dim

        if num_mines < 0 or num_mines > dim**2:
            print("Invalid number of mines specified --> set to 0")
            num_mines = 0
        self.num_mines = num_mines

        board = [[0 for _ in range(dim)] for _ in range(dim)]

        for n in range(num_mines):
            x = random.randint(0, dim-1)
            y = random.randint(0, dim-1)
            while board[x][y] == MINE:
                x = random.randint(0, dim-1)
                y = random.randint(0, dim-1)
            board[x][y] = MINE

        temp = np.array(board)

        coords = zip(*np.where(temp == MINE))
        for x, y in coords:
            for i, j in dirs:
                if 0 <= x + i < dim and 0 <= y + j < dim and temp[x+i][y+j] != M
INE:
                    temp[x + i][y + j] += 1
                else:
                    continue

        self.board = temp.tolist()
```

The Minesweeper class is even more straightforward. It:

1. Specifies coordinates to easily iterate through local cells
2. Specifies a key for the board (mine, hidden, safe...etc.)
3. Places random mines throughout our data structure based on some params (dim = dimension, n = # of mines)
4. Updates data structure with clues (+= 1 to every surrounding cell of a mine)

Its members are:

1. **`dim` (Type: Integer):** Dimension
2. **`num_mines` (Type: Integer):** Number of mines
3. **`board` (Type: 2d list):** Board
4. **`success` (Type: Boolean):** Whether or not the board was solved

See `Minesweeper.py` for more implementation details.

## Baseline solver

As it is the baseline or "naive" approach to Minesweeper, our solver algorithm begins with a random first move (i.e. revealing a cell). It then tries to deduce, based solely on the $8$ cells adjacent to it, whether or not any of the $9$ cells (total) in question are definitely safe or definitely a mine.

Clearly, the chosen cell does not need to be inferred, but the $8$ cells adjacent can generally not be inferred on the first move *unless* the clue is $0$, as there is not enough information yet. Mathematically, we can denote this as: $P(X_{i,j} = 0|C) = (1 - d)^9$ where $C$ is the board configuration and $d$ is the mine density (Note: we are assuming the first cell queried is *not* a corner or edge cell, but our implementation handles all cases).

This inference relies only on the local data of the adjacent cells; the following rules are followed:

1. If the clue minus the number of revealed mines is equal to the number of hidden neighbors, then every hidden neighbor is a mine.
2. If the number of safe neighbors minus the number of revealed safe neighbors is the number of hidden neighbors, then every hidden neighbor is safe.

These rules are deterministic, i.e. they **decisively** find mine or safe cells. All cases in between the above two rules are not deterministic, and the baseline falls through to random choice of the remaining hidden cells (after re-crunching the knowledge base: see below section).

See `solve(...)` method in `Agent.py` for more implementation details.

For a trial run, see attached `/data/log.txt` for full breakdown of moves made by baseline agent. The following snippet illustrates the above methodology:

---

```
BASELINE SOLVER LOG (initial moves)
```

```
Cell (8, 2) safely revealed with clue: 3.
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8

Revealing cell (8, 2) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

    Re-processing did not find new safe cells; proceeding to randomly sele
ct hidden cell.

----------------------------------------

Cell (3, 5) safely revealed with clue: 1.
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8

Revealing cell (3, 5) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

    Re-processing did not find new safe cells; proceeding to randomly sele
ct hidden cell.

----------------------------------------

BOOM! Mine detonated at (5, 5).


Revealing cell (5, 5) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
```

```
or add random if none available.

    Re-processing did not find new safe cells; proceeding to randomly sele
ct hidden cell.

----------------------------------------

Cell (3, 7) safely revealed with clue: 1.
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8


Revealing cell (3, 7) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

    Re-processing did not find new safe cells; proceeding to randomly sele
ct hidden cell.

----------------------------------------

Cell (11, 7) safely revealed with clue: 0.
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8


All neighbors of (11, 7) must be safe.

    Neighbor (11, 8) flagged as safe and enqueued for next visitation.
    Neighbor (11, 6) flagged as safe and enqueued for next visitation.
    Neighbor (12, 7) flagged as safe and enqueued for next visitation.
    Neighbor (10, 7) flagged as safe and enqueued for next visitation.
    Neighbor (12, 8) flagged as safe and enqueued for next visitation.
    Neighbor (10, 6) flagged as safe and enqueued for next visitation.
    Neighbor (10, 8) flagged as safe and enqueued for next visitation.
    Neighbor (12, 6) flagged as safe and enqueued for next visitation.

...
```

When enough of the board has been filled, the metrics of each cell start to be the main source of knowledge for the agent. At every cell $(x, y)$, the agent keeps track of $(x, y)$'s total neighbors, its safe neighbors, mine neighbors and hidden neighbors, which can be used for marking more cells as safe/unsafe. This is **local inference.** For example, if a cell has 8 total neighbors, 7 safe neighbors, 1 hidden neighbor and a clue of 1, the agent can determine with 100% certainty that the final neighbor is a mine. Once the agent can no longer make any reliable inferences, it returns to the last safe, visited cell which is stored in a queue. If the queue is empty, the agent returns to random selection.

**Recrunching the KB via local inference prior to random selection**

Note that in cases where the algorithm falls to random selection, it first reprocesses its entire knowledge base by repeating how it parses clues (as above) to conduct local inference. That is, it iterates over all currently safe, uncovered cells and re-calculates number of safe neighbors, mine neighbors, hidden neighbors, etc. and possibly infers mine/safety. That way, it deduces as much knowledge as possible prior, possibly enqueueing safe cells to visit next or marking cells as mines.

The motivation behind this is to minimize the number of mines which are accidentally detonated by random selection when, in fact, the current knowledge base contains that information. We do not need to uncover new cells to retrieve that information; our KB just needs to be "re-crunched" in terms of local inference to retrieve this insight. Then we filter out some of those mines that could have been randomly selected and improve our baseline without doing anything beyond simple local inference.

Below is a snippet from `/data/log.txt` illustrating this more concretely; see the `Reprocessing-KB found that: ...` lines below.

---

```
BASELINE SOLVER LOG (intermediate moves)

...

----------------------------------------

Cell (6, 5) safely revealed with clue: 4.
    # safe neighbors: 2
    # mine neighbors: 2
    # hidden neighbors: 4
    # total neighbors: 8


Revealing cell (6, 5) led to no conclusive next move (either DETONATED or
all neighbors MINES).
```

```
Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

    Re-processing KB found that: All neighbors of (2, 5) must be safe.

        Neighbor (1, 6) flagged as safe and enqueued for next visitation.

----------------------------------------

Cell (1, 6) safely revealed with clue: 3.
    # safe neighbors: 4
    # mine neighbors: 1
    # hidden neighbors: 3
    # total neighbors: 8


Revealing cell (1, 6) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

    Re-processing KB found that: All neighbors of (0, 5) must be mines.

        Neighbor (0, 6) flagged as a mine.

    Re-processing did not find new safe cells; proceeding to randomly sele
ct hidden cell.

----------------------------------------

Cell (12, 4) safely revealed with clue: 5.
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8


Revealing cell (12, 4) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
```

or add random if none available.

    Re-processing did not find new safe cells; proceeding to randomly sele
ct hidden cell.

---

...

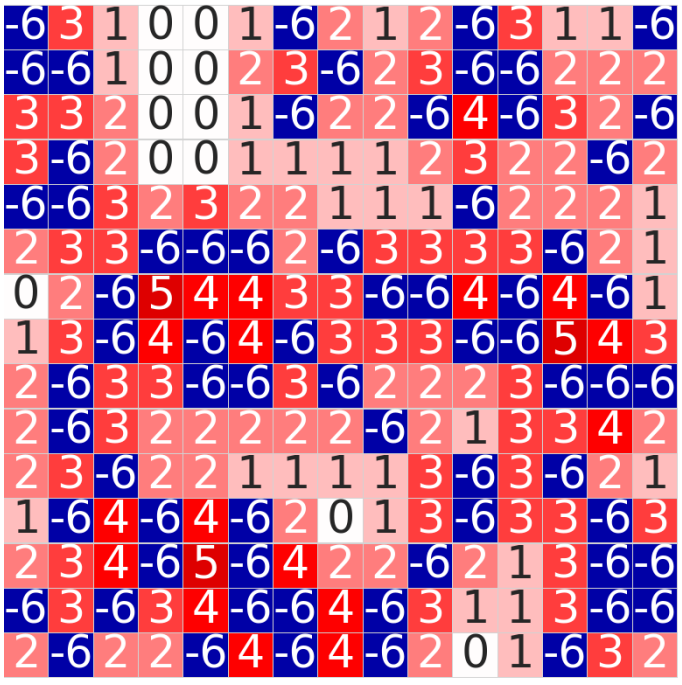
---

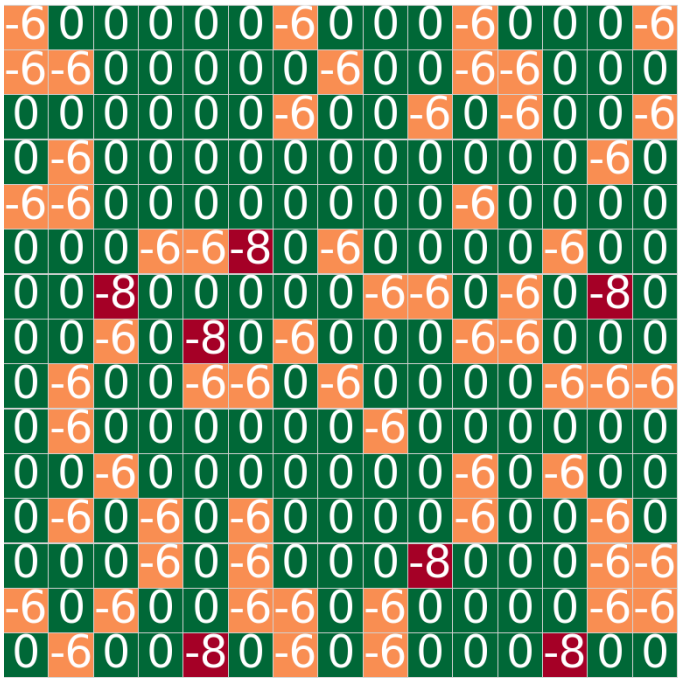
***** GAME OVER *****

Game ended in 0.11594223976135254 seconds

Safely detected (without detonating) 89.55223880597015% of mines

The above log was from the following initial & final game boards:




Initial Board (Unhidden)

Solved Board

For the initial board, the blue $-6$ cells correspond to mines, and the other numbers are clues (color coded based on intensity of clue).

For the final solved player knowledge board, the green $0$ cells are safely uncovered cells, orange $-6$ cells are safely detected mines, and red $-8$ cells are detonated mines.

We can see the baseline algorithm performing quite well here, with a mine flag rate of ~90%. This can be seen visually above on the solved board where cells in orange are mines that are safely detected, and cells in red are mines that are detonated.

# Performance & Results

The following tables show how the baseline algorithm performs under different parametrized conditions ( $dim$, densities).

## Baseline algorithm: Average mine flag rates over various board sizes and mine densities

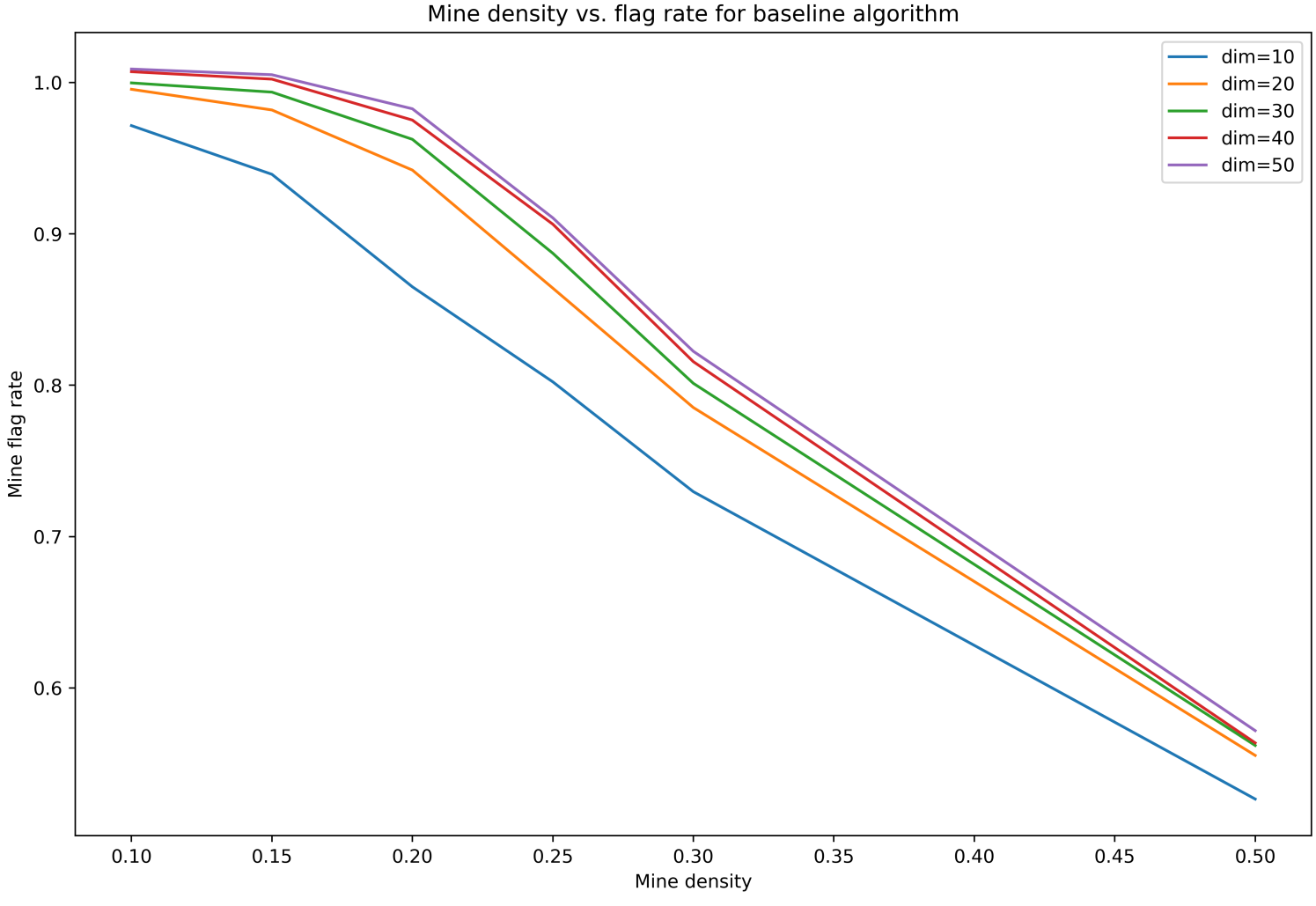
In [3]:

```
import pandas as pd
baselineFlagRates = pd.read_csv("./data/baseline_solo_performance_flagRates.csv"
, index_col=0)
baselineFlagRates
```

Out[3]:

| | d=0.1 | d=0.15 | d=0.2 | d=0.25 | d=0.3 | d=0.5 |
|---|---|---|---|---|---|---|
| **dim=10** | 0.971500 | 0.939333 | 0.865000 | 0.802200 | 0.729667 | 0.526600 |
| **dim=20** | 0.995500 | 0.981833 | 0.942125 | 0.864100 | 0.785250 | 0.555425 |
| **dim=30** | 0.999722 | 0.993630 | 0.962444 | 0.887156 | 0.801241 | 0.562000 |
| **dim=40** | 1.007125 | 1.002208 | 0.975156 | 0.906350 | 0.815563 | 0.563713 |
| **dim=50** | 1.008880 | 1.005120 | 0.982620 | 0.910544 | 0.822400 | 0.571800 |

*Note:* Some of the flag rates are $> 1$ very slightly, this is due to minor data rounding errors. Each cell in the table is an average value of 100-200 trials.



This table depicts the average flag rates (detected mines/total mines) per dim/density and clearly we can see a strictly decreasing trend as the mine density increases. This is because more mines not only forces the baseline agent to make more random choices but also makes it more prone to choosing mines on those random iterations.

We also see that although this holds for all $dim$, the flag rates are higher for higher $dim$ than they are for lower $dim$ with the same density. This is interesting and likely because with more safe cells available to reveal, the algorithm can perform more local inference and deterministically find more safe/mine cells than is the case for lower $dim$.

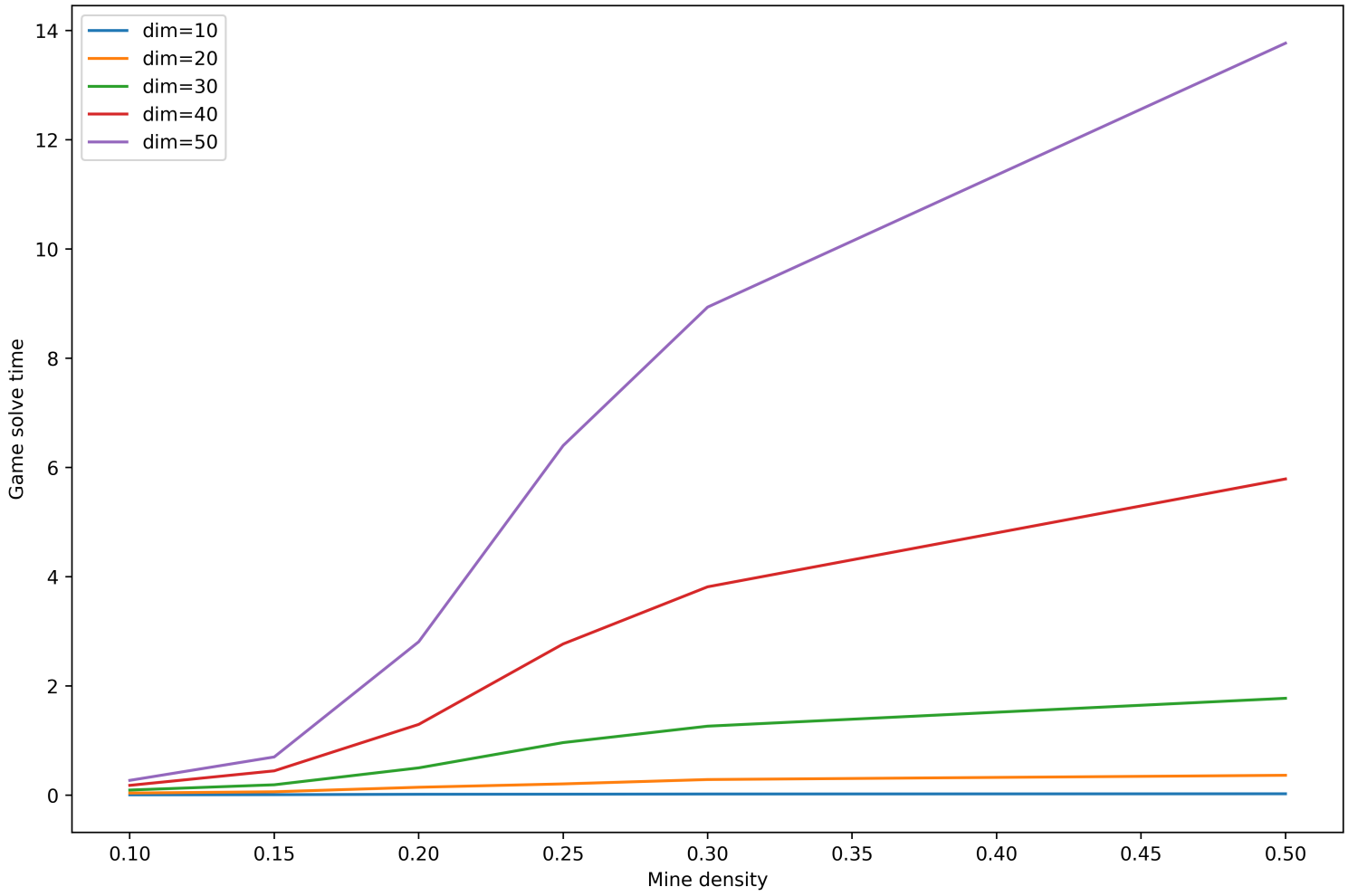
## Baseline algorithm: Average runtime over various board sizes and mine densities

In [4]:

```
baselineTimes = pd.read_csv("./data/baseline_solo_performance_times.csv",index_c
ol=0)
baselineTimes
```

Out[4]:

| | d=0.1 | d=0.15 | d=0.2 | d=0.25 | d=0.3 | d=0.5 |
|---|---|---|---|---|---|---|
| **dim=10** | 0.008125 | 0.011161 | 0.018602 | 0.020182 | 0.023605 | 0.027591 |
| **dim=20** | 0.041218 | 0.064340 | 0.146992 | 0.208602 | 0.288224 | 0.366105 |
| **dim=30** | 0.095811 | 0.192290 | 0.501797 | 0.964674 | 1.265326 | 1.775726 |
| **dim=40** | 0.181584 | 0.447265 | 1.297337 | 2.770875 | 3.816501 | 5.791174 |
| **dim=50** | 0.273127 | 0.701709 | 2.810102 | 6.401690 | 8.937176 | 13.768566 |




This table depicts the average performance times (in seconds) per dim/density. Again, we test each pairing 100-200 times and average the runtimes of the trials. Like the previous table, the pattern is evident; higher dims/densities correspond to longer runtimes due to more random choicing and recomputation of the knowledge base.

Notably, although runtimes always get worse for higher mine densities, this 'worsening' effect is 'worse' for higher $dim$ than for lower $dim$. Again, this is because more inference is completed, which leads to better mine flag rates but more computations which contribute to higher runtimes.

# Improved Strategy (v1): Linear Algebra Approach

To improve upon the baseline approach, we devise a strategy that relies not solely on local inference but also on an evaluation of all the clues given to evaluate the whole board at every iteration. We will refer to this as the **Linear Algebra Approach,** as it relies on core principles of linear algebra.

# Implementation

## Representation

Our Linear Algebra Agent (see `LinAlg.py` ) inherits all members & methods from our generic Agent class, and thus its object representation is nearly identical (except one extra metric for computational convenience); as is the board structure that it operates on.

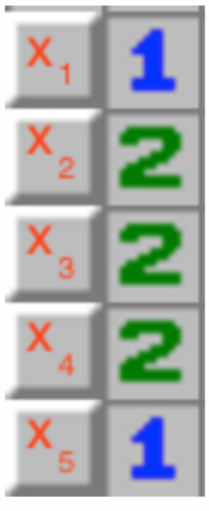
In [5]:

```
class lin_alg_agent(agent):
    def __init__(self, game, useMineCount, order):
        agent.__init__(self,game, order)
        self.useMineCount = useMineCount
```

## Inference

Rather than performing inference based on a restricted set of clues/information about adjacent data, the Linear Algebra approach does inference on the entire board, which effectively expands the knowledge base by a factor of $(dim - 3)^2$. It then uses that expanded knowledge base to create a constraint satisfaction problem or, more concretely, a system of linear equations that analytically describes what cells are *guaranteed* to be mines.

This is illustrated below:




Example block of game board with variable labels

$$
\begin{aligned}
x_1 + x_2 &= 1 \\
x_1 + x_2 + x_3 &= 2 \\
x_2 + x_3 + x_4 &= 2 \\
x_3 + x_4 + x_5 &= 2 \\
x_4 + x_5 &= 1
\end{aligned}
$$

Corresponding System of Equations

$$
\left(\begin{array}{ccccc|c}
1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 2 \\
0 & 1 & 1 & 1 & 0 & 2 \\
0 & 0 & 1 & 1 & 1 & 2 \\
0 & 0 & 0 & 1 & 1 & 1
\end{array}\right)
$$

Corresponding Matrix

Images courtesy of [https://massaioli.wordpress.com/2013/01/12/solving-minesweeper-with-matricies/comment-page-1/ (https://massaioli.wordpress.com/2013/01/12/solving-minesweeper-with-matricies/comment-page-1/)](https://massaioli.wordpress.com/2013/01/12/solving-minesweeper-with-matricies/comment-page-1/).

The advantage of this approach is that it makes full use of every clue by relating it to every other known clue on the board. In other words, the Linear Algebra Agent uses all of the mine proximity clues in its knowledge base to the fullest extent.

In early iterations of our implementation, the agent solely relied on the system of linear equations.

In its final form, the agent uses not only every single clue that is available on the board, but also the number of hidden mines, the number of total mines, the number of safe mines and the rest of the local data that the baseline approach was using to make the best possible decision. Usage of the number of total mines is described in the sections below.

## Algorithm

Here is how we implemented the Linear Algebra approach.

In [ ]:

```
...

#if we have information, solve the system
if information_cells:
    # create matrix of zeroes to start
    # each row will represent information provided by one clue
    # we take each hidden neighbor (nx,ny) of the cell with the clue
    # and then set row[dim*nx + ny] = 1, and the last value in row
    # equal to clue - already discovered mines. This encodes a
    # boolean of whether a spot contributes to a clue, and when we solve
    # the equation for the dim*dim values, constraining each to 1 or 0
    # we can know if (x,y) is a mine iff the variable corresponding to
    # the (dim * x + y)th column is 1
    matrix = np.zeros((len(information_cells) + (1 if self.useMineCount else 0),
dim*dim + 1), dtype = 'float64')
    for i in range(len(information_cells)):
        x, y, clue, mineNeighbors, numHiddenNbrs = information_cells[i]
        hidden_nbrs = self.get_hidden_neighbors(x,y)
        for nx, ny in hidden_nbrs:
            matrix[i][dim * nx + ny] = 1
        matrix[i][dim * dim] = clue - mineNeighbors

    rref(matrix)

    for row in matrix:
        positives = []
        negatives = []
        #scale the row for code reuseability
        negated = False
        if row[-1] < 0 :
            row *= -1
            negated = True
        #for each variable
        for i in range(dim*dim):
```

```
            #first get the cell it corresponds to

            x = i // dim
            y = i % dim
            # if the coeffecient in this row is positive put it in
            # positives. if negative then negatives, if 0 then we don't care
            if row[i] >0:
                positives.append((x,y, row[i]))
            elif row[i] < 0:
                negatives.append((x,y, row[i]))

        #if the row has no information move on
        if len(positives) == len(negatives) == 0:
            continue

        #if the row sums to 0, we can potentially find definitively safe cells
        if row[-1] == 0:
            # if we have both positive and negative coeffecients, the variables
could all be 0
            # or there could be some mines, the sum of the coeffecients of the m
ines is 0
            # which can happen since we have positive and negative coeffecients
            # so we don't know anything definitive (multiple combinations satisf
y the equation)
            if len(positives) > 0 and len(negatives) > 0:
                continue
            else:
                # however if only one list is populated then everything
                # in the list must have value 0 for equation to hold (so everyth
ing is safe)
                # we can do positives + negatives since exactly one has values
                for x,y,_ in positives + negatives:
                    assert self.game.board[x][y] != MINE
                    if self.logging:
                        print("deduced ({},{}) to be safe via lin alg".format(x,
y))
                    self.playerKnowledge[x][y] = SAFE
                    q.append((x,y)) #add the safe cell to the queue to visit nex
t
                    safesFound = True
        else:
            # if the row sums to a positive then since each variable is only
            # 0 or 1 (ie. can not be negative), if the sum of the
            # positive coeffecients exactly matches the last value of the row
            # (the other side of the equation), then all the variables with
            # positive coeffecients must be 1, and so we flag them. The sum can
never be less since
            # each variable is at most 1, and if it is greater, then multiple co
mbinations
            # of the variables in the positive and negative lists can be mines
            if sum([coeffecient for x,y,coeffecient in positives]) == row[-1]:
                for x,y,_ in positives:
                    if self.playerKnowledge[x,y] == MINE:
                        continue
```

```
                    if self.logging:

                        print("deduced ({},{}) to be a mine via lin alg".format(
x,y))
                    assert self.game.board[x][y] == MINE
                    self.playerKnowledge[x][y] = MINE
                    self.numFlaggedMines += 1
                for x,y,_ in negatives:
                    if self.playerKnowledge[x,y] == SAFE:
                        if self.logging:
                            print()
                        continue
                    if self.logging:
                        print("deduced ({},{}) to be safe via lin alg".format(x,
y))
                    assert self.game.board[x][y] != MINE
                    self.playerKnowledge[x][y] = SAFE
                    q.append((x,y))
                    safesFound = True
```

Due to the inheritance structure of our agent classes, the Linear Algebra approach will attempt to deterministically find safe/mine cells via the baseline method, then execute the above code to use more than local inference, and then, in cases where no safe/mine cells can be found and enqueued, it falls to random choice.

One criticism of the purely Linear Algebra approach is how it handles random guessing when there are no available solutions at any given move. Essentially, when the system of linear equations cannot find a solution, it falls through to random choice, as in the baseline. This will be improved upon in the latest iterations and will be highlighted further in this assignment report.

## Decisions

Decision-making is fairly straightforward: once the matrix is row reduced, we can mark cells that correspond to reduced rows with a $0$ in the final column vector as being safe and similarly reduced rows with value $1$ in the final column vector as being a mine. Unreduced or $0$ row vectors are disregarded.

The Linear Algebra Agent's decision-making process is logged out in a snippet below (see `/data/lin_alg_log.txt`):

---

```
***** GAME STARTING *****

4 by 4 board with 4 mines

Solving with LINEAR ALGEBRA strategy

...
```

```
----------------------------------------

Cell (1, 1) safely revealed with clue: 4.
    # safe neighbors: 2
    # mine neighbors: 1
    # hidden neighbors: 5
    # total neighbors: 8


Revealing cell (1, 1) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

game:
[-6, 3, -6, 1]
[-6, 4, 2, 1]
[2, -6, 1, 0]
[1, 1, 1, 0]

knowledge:
[[-8  0 -1 -1]
 [-1  0 -1  0]
 [-1 -1  0 -1]
 [ 0 -1 -1 -1]]

generated following row using (0,1):
[0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 2.]

generated following row using (1,1):
[0. 0. 1. 0. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 3.]

generated following row using (1,3):
[0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1.]

generated following row using (2,2):
[0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 1.]

generated following row using (3,0):
[0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1.]

information matrix:
[[0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 2.]
```

```
 [0. 0. 1. 0. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 3.]
 [0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1.]]

rref'd matrix:
[[ 0.  0.  1.  0.  1.  0.  0.  0.  0. -1.  0. -1.  0.  0. -1. -1.  1.]
 [ 0.  0.  0.  1. -1.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  1.  0.  0.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  1.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]]

using row:
[ 0.  0.  1.  0.  1.  0.  0.  0.  0. -1.  0. -1.  0.  0. -1. -1.  1.]

using row:
[ 0.  0.  0.  1. -1.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0. -1.]
deduced (1,0) to be a mine via lin alg
deduced (0,3) to be safe via lin alg
deduced (2,3) to be safe via lin alg

using row:
[0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 0. 0. 1. 1. 1.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
deduced (3,1) to be safe via lin alg

...

----------------------------------------


***** GAME OVER *****

Game ended in 0.014050960540771484 seconds

Safely detected (without detonating) 75.0% of mines
```

Here we see the linear algebra approach taking place after the typical baseline local inference, parsing the system of equations out of clues and hidden cells, and then solving them using Gaussian elimination.

# Performance

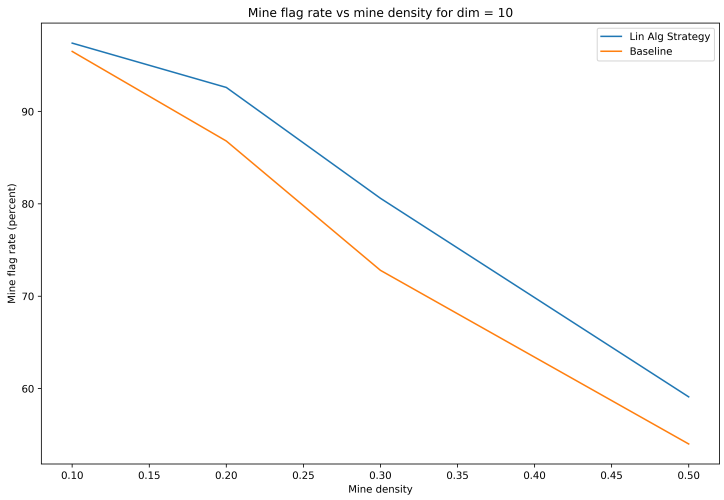
Below is a comparison table of the Linear Algebra approach with the baseline approach in terms of mine detection rate (once again, DETECTED/total) as a percentage and performance time in seconds.

In [6]:

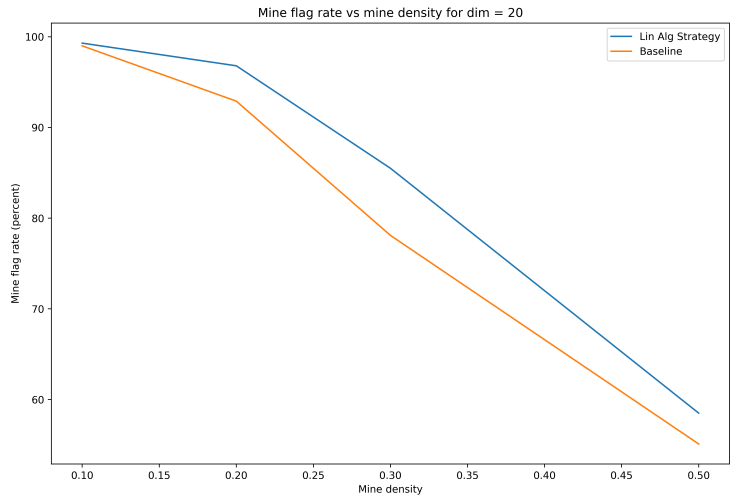
```
df = pd.read_excel("./data/comparisondata.xlsx",sheet_name=0,header=0).dropna()
df.style.hide_index()
base_linalg_df = df[['Dim', 'Density', 'Baseline', 'Lin Alg']]
base_linalg_df.transpose()
```

Out[6]:

| | 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 11 | 12 | 13 | 14 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dim** | 10.0 | 10.0 | 10.0 | 10.0 | 20.0 | 20.0 | 20.0 | 20.0 | 30.0 | 30.0 | 30.0 | 30.0 | 40.0 | 40.0 | 40.0 |
| **Density** | 0.1 | 0.2 | 0.3 | 0.5 | 0.1 | 0.2 | 0.3 | 0.5 | 0.1 | 0.2 | 0.3 | 0.5 | 0.1 | 0.2 | 0.3 |
| **Baseline** | 96.5 | 86.8 | 72.8 | 54.0 | 99.0 | 92.9 | 78.1 | 55.1 | 99.5 | 95.7 | 79.8 | 55.7 | 99.7 | 96.5 | 81.1 |
| **Lin Alg** | 97.4 | 92.6 | 80.6 | 59.1 | 99.3 | 96.8 | 85.5 | 58.5 | 99.6 | 98.4 | 86.5 | 58.9 | 99.7 | 99.0 | 88.4 |




Dim 10




Dim 20




Dim 30




Dim 40

The above data is representative of 50-100 trials (depending on dim/density) which are averaged together. We can see that using the linear algebra approach consistently yields an improvement from the baseline case.

We can also see that the game gets 'hard' when the density surpasses approximately $d = 0.20$, at which point mine flag rate drops drastically.

# Improvements

Suppose the agent is told in advance how many total mines exist on the board. Then, recalling the system of equations used by the linear algebra approach, it follows that we would be able to add one additional equation to our system which conveys the total mine count (on the RHS of the equation). This would give us more mine information at each step in the algorithm. Consider the situation where 199 of 200 mines have been found already, and one pass of linear algebra finds the 200th mine. Then, subsequent passes of linear algebra will have an equation which essentially says all possible mines have been found, and the algorithm may use this to mark mines as safe, cutting down runtime and opportunities to fall through to random selection.

## Mine count implementation

Implementation-wise, all agents are equipped with a `useMineCount` boolean field which makes usage of mine counts toggleable. Setting it to `True` essentially adds the following equation at the end of the system of equations:

$$x_1 + x_2 + \cdots + x_{dim^2-1} = N_m - N_f - N_d$$

where each $x_i$ is $dim * x + y$ (as before) for a given hidden cell $(x, y)$ and $N_m, N_f, N_d$ correspond to the numbers of total mines, flagged mines, and detonated mines, respectively. This makes the RHS above the number of remaining hidden mines.

In [ ]:

```
if self.useMineCount:
    for x, y in hidden_cells:
        matrix[len(information_cells), dim * x + y] = 1
    matrix[len(information_cells),dim*dim] = self.game.num_mines - self.numFlaggedMines - self.numDetonatedMines
```

The additional equation is appended to the system prior to being sent to reduced row-echelon form and solved. The addition of this one row simply gives the algorithm another row upon which to use inference, e.g. for an iteration of vanilla linear algebra on a $4 \times 4$ game with $4$ mines we are able to achieve a 100% solve rate:

---

```
...

Cell (2, 2) safely revealed with clue: 3.
    # safe neighbors: 4
    # mine neighbors: 1
    # hidden neighbors: 3
    # total neighbors: 8

Revealing cell (2, 2) led to no conclusive next move (either DETONATED or
all neighbors MINES).
```

```
Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

game:
[0, 0, 0, 0]
[1, 1, 1, 1]
[-6, 2, 3, -6]
[2, -6, 3, -6]

knowledge:
[[ 0  0  0  0]
 [ 0  0  0  0]
 [-6  0  0 -6]
 [-1 -1 -1 -1]]

generated following row using (2,1):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1.]

generated following row using (2,2):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 2.]

generated following row using total mine count:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 2.]

information matrix:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 2.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 2.]]

rref'd matrix:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1.]]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
deduced (3,0) to be safe via lin alg

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 1.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1.]
```

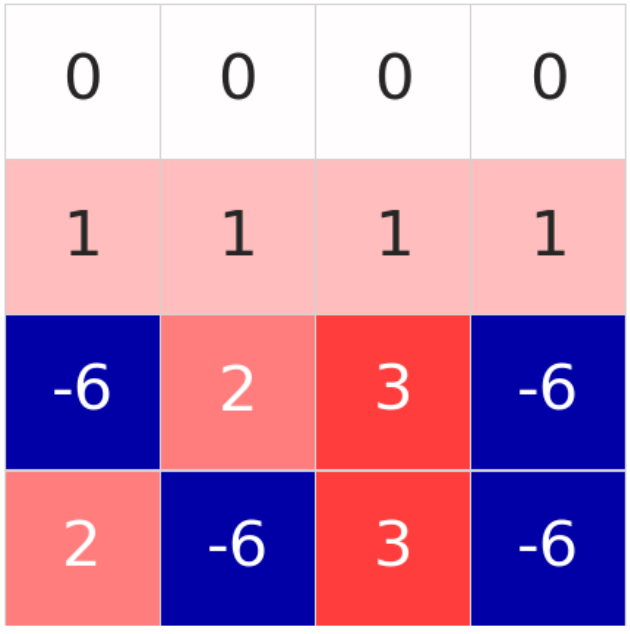
```
deduced (3,3) to be a mine via lin alg

----------------------------------------

Cell (3, 0) safely revealed with clue: 2.
    # safe neighbors: 1
    # mine neighbors: 1
    # hidden neighbors: 1
    # total neighbors: 3


All neighbors of (3, 0) must be mines.

    Neighbor (3, 1) flagged as a mine.

Revealing cell (3, 0) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,
or add random if none available.

game:
[0, 0, 0, 0]
[1, 1, 1, 1]
[-6, 2, 3, -6]
[2, -6, 3, -6]

knowledge:
[[ 0  0  0  0]
 [ 0  0  0  0]
 [-6  0  0 -6]
 [ 0 -6 -1 -6]]


generated following row using (2,1):

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

generated following row using (2,2):

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

generated following row using total mine count:
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

information matrix:

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]


rref'd matrix:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
deduced (3,2) to be safe via lin alg

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

----------------------------------------

Cell (3, 2) safely revealed with clue: 3.
    # safe neighbors: 2
    # mine neighbors: 3
    # hidden neighbors: 0
    # total neighbors: 5


All neighbors of (3, 2) are already revealed; nothing to infer.

----------------------------------------
...

----------------------------------------


***** GAME OVER *****

Game ended in 0.0106010437011718 75 seconds
```

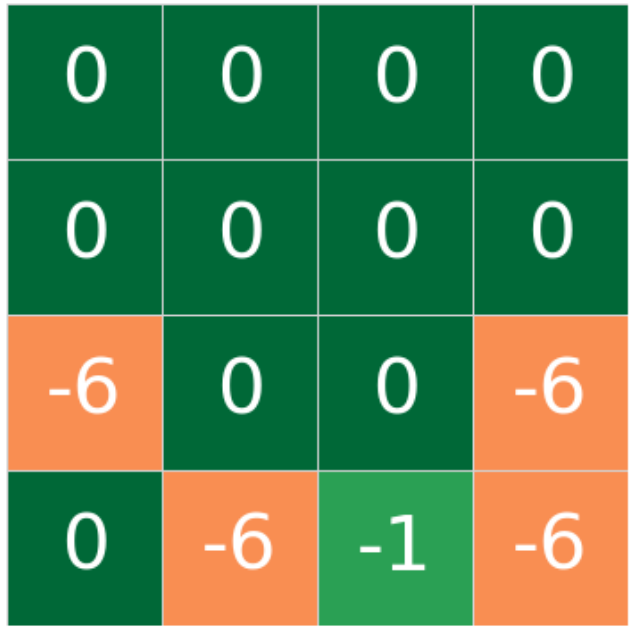
```
Safely detected (without detonating) 100.0% of mines
```

---

For the full log, see `/data/lin_alg_log_mineCt.txt` . The game board & final result that yielded the above log is displayed here:




Initial Board (Unhidden)




Solved Agent Knowledge via Linear Algebra

Notice that in the final state, there is one block still HIDDEN (light green, $-1$ value), however the agent terminated its gameplay because the number of mines is in its knowledge base. Had the number of mines not been specified, the agent would have continued until all the remaining hidden cells were revealed, adding slightly more computations (in this low-dimensional case). In higher dimensions, these extra computations could be sizeable though.

Our intuition is that knowing the mine count definitely helps the driver, but we seek a further improvement to our strategy to better exploit this fact.

# Improved Algorithm (v2): Linear Algebra + Brute Force Probability Approach

Our second iteration is a modified version of the **Linear Algebra approach** that makes improvements to guessing. Any algorithm to solve Minesweeper will have to eventually make a random guess. The improvements outlined in this version of our approach attempts to address this problem and instead make an *educated guess* using a combination of brute force and probabilistic inference. We call this the combined **Linear Algebra + Brute Force approach,** and we use a new agent called the **Brute Force Agent.**

## Implementation

The Brute Force Agent inherits all of the Linear Algebra Agent's (and thus, the baseline Agent's) members/functions. The object-oriented schema remains unchanged throughout this assignment.

In [7]:

```
class brute_force_agent(lin_alg_agent):
    def __init__(self, game, useMineCount, order):
        lin_alg_agent.__init__(self,game,useMineCount,order)
```

The main advantage of adding brute force functionality is that upon failure of the Linear Algebra Agent's main method (i.e. solving the system of equations via matrix algebra), it no longer has to randomly guess what the next move will be. Instead, we iterate through all possible configurations of relevant cells and keep track of how many times a mine appeared in each cell. We then simply choose the cell with the least probability of having a mine.

The methods that perform these computations are listed below.

### Configuration Generation Function

This function finds all possible configurations of mines that are valid in terms of consistency with the existing player knowledge base.

In [8]:

```
# potential mine configuration finder
def get_configs(self, configCells, consistency_cells):
    # configs is our current list of potential configs, we have a tuple with thr
ee elements
    # 1. a copy othe KB that will be modified to add mines
    # 2. an index of the next cell in the component to set as a mine, this never
decreases to force
    #    an order on how we visit configurations, and removes looping. If index
>= len(component) then we terminate
    # 3. a list of cells set to mines in the component to be returned if the con
fig is valid
    configs = [(deepcopy(self.playerKnowledge), i, [])  for i in range(len(confi
```

```
gCells))]

    # list of all valid configs
    out = []
    while configs:
        # essentially doing a bfs over configurations, instead of a queue we tak
e all the nodes at a given depth
        # and then add all their children to next_set, and then use that for nex
t iteration
        next_set = []
        for board, index, currentConfig in configs:
            current_board = deepcopy(board)
            current_config = deepcopy(currentConfig)
            dim = self.game.dim

            # terminate if index is out of bounds, or if using mine count and th
ere are too many mines
            if index >= len(configCells) or (self.useMineCount and len(current_c
onfig) == \
                                                    self.game.num_mines-self.numFlagged
Mines-self.numDetonatedMines):
                # if the config is consitent, add it to our output list
                if self.confirm_full_consistency(current_board, consistency_cell
s):
                    out.append(current_config)
                continue

            # get the next cell we want to set as a mine and do so
            cell = configCells[index]
            current_config.append(cell)
            x = cell[0]
            y = cell[1]
            current_board[x,y] = MINE

            # initialize the config as valid, but set valid false if any neighbo
r has been made inconsistent
            valid = True
            for dx, dy in dirs:
                nx = x + dx
                ny = y + dy
                if 0 <= nx < dim and 0 <= ny < dim and self.playerKnowledge[nx][
ny] == SAFE \
                                        and not self.confirm_consistency(current_boa
rd, nx, ny):
                    valid = False
            # if every neighbor was found to be valid then add each of the node
(the config's) children to
            # the next list
            # it's children are adding any of the cells after index as a mine, o
r terminating
            #(index = len(configCells))
            if valid:
                for i in range(index + 1, len(configCells) + 1):
                    next_set.append((current_board,i,current_config))
```

```
        configs = next_set
    return out
```

## Probability Generating Function

This function acts on what we call **components:** A component is a set of hidden cells that are neighboring cells we have clues on, and can thus generate a probability it is a mine. Two such hidden cells are in the same component if they both neighbor the same safe cell, and this is transitive. Basically, a component is a set of cells that are potentially mines, and one cell in the component being a mine affects whether the other cells in the component are mines.

Then, the above configuration generation function generates valid configurations of these components.

On these components, this function maps each cell in a component to the percentage of configurations of the component it appears in. We take this as the **probability of that cell being a mine.**

In [9]:

```
# overriding probability_method to use brute force probabilities
# broad strokes implementation:
# First create a list of components;
# This list of components of hidden cells is config_cells below
# We also store the associated safe cells for each component in consistency_cell
s
# cap is the maximum size for a component that we are willing to compute probabi
lities for, as larger components take exponentially longer to compute
def probability_method(self, cap = 20):
    dim = self.game.dim
    # list of safe cells separated by corresponding component, we will need to u
se confirm_full_consistency using the set of safe cells
    # to verufy a confuration of mines' consistency
    consistency_cells = list()
    # list of hidden cells separated by corresponding component, called config_c
ells as
    # each component will be used to generate potential configurations of mines
    config_cells = list()
    for x in range(dim):
        for y in range(dim):
            # if the cell has a clue
            if self.playerKnowledge[x][y] == SAFE:
                numSafeNbrs, numMineNbrs, numHiddenNbrs, numTotalNbrs = self.get
CellNeighborData(x, y)
                # make sure there are hidden neighbors to try making mines
                if numHiddenNbrs > 0:
                    # this cell we be needed for the consistency_cells component
for all its children (neighbors)
                    children_consistency = {(x,y)}
                    children = set(self.get_hidden_neighbors(x,y))
                    intersects = []
                    # find all other components that have use any of the same hi
```

```
dden cells
                    for i,s in enumerate(config_cells):
                        if len(children.intersection(s)) > 0:
                            intersects.append(i)
                    # if any such components were found, then we add those compo
nents to the one we are currently creating,
                    # as they should actually be the same component. Then delete
the old version separated versions those
                    # components.
                    if len(intersects) > 0:
                        for i in intersects:
                            children.update(config_cells[i])
                            children_consistency.update(consistency_cells[i])
                        for i in intersects[::-1]:
                            del config_cells[i]
                            del consistency_cells[i]
                    # Then we add our current component to our lists.
                    config_cells.append(children)
                    consistency_cells.append(children_consistency)

    # if we found no components then we just use pure random as we cannot comput
e probabilities
    if len(consistency_cells) == 0:
        return self.get_next_random(set())
    # just turn the config_cells into a list for simplicity
    config_cells = [sorted(list(y), key = lambda x: x[0] * dim + x[1]) for y in
config_cells]
    # configs is potential configurations for a given component
    configs = []
    # we will remove some components for being too large to compute
    to_remove = []

    for i,s in enumerate(config_cells):
        # if a component is too large, note that we should remove it
        if len(s) > cap:
            to_remove.append(i)
            continue
        # otherwise get the list of possible mine configurations for the config,
passing the corresponding
        # safe cell list (consistency_cells[i]) so we can ensure the configurati
ons are possible
        start_search_time = time.time()
        current_configs = self.get_configs(s, consistency_cells[i])
        configs.append(current_configs)

    # remove the components that are too large
    for i in to_remove[::-1]:
        del config_cells[i]
        del consistency_cells[i]

    if len(configs) == 0:
        return self.get_next_random(set())
```

```
    # minimum number of mines present among all configs

    min_mine_count = sum([ 0 if len(s) == 0 else min([len(config) for config in
s]) for s in configs])
    if self.useMineCount:
        for s in configs:
            min_for_set =  0 if len(s) == 0 else min([len(config) for config in
s])
            to_remove = []
            for i,config in enumerate(s):
                # for each config for the component, if the number of mines in t
he config + the minimum number
                # of mines in other configs (min_mine_count - min_for_set) excee
ds the number of mines in the game,
                # then remove the config as it is impossible
                if len(config) + min_mine_count - min_for_set > self.game.num_mi
nes-self.numFlaggedMines-self.numDetonatedMines:
                    to_remove.append(i)
            for i in to_remove[::-1]:
                del s[i]

    # map each cell in a component to the percentage of configs of the component
it appears in
    # this is it's probability of being a mine
    probabilities = dict()
    for i,s in enumerate(configs):
        if len(s) > 0:
            counts = {x:0 for x in config_cells[i]}
            for config in s:
                for coordinates in config:
                    counts[coordinates] += 1
            for k , v in counts.items():
                probabilities[k] = v / len(s)
        else:
            for cell in config_cells[i]:
                probabilities[cell] = 0

    # get cell with lowest probability
    best_cell = None
    best_probability = len(configs)

    for cell, probability in probabilities.items():
        if probability <= best_probability:
            best_cell = cell
            best_probability = probability

    # the probability that a cell not in any component is a mine is uniform amon
g all such cells
    # it is at most the maximum numnber of mines not in any component divided by
the number of such cells
    other_cells = self.numHiddenCells() - sum([len(component) for component in c
onfig_cells])
    mines_left = self.game.num_mines-self.numFlaggedMines-self.numDetonatedMines
    max_mines_in_hidden_cells_not_in_config_Cells = mines_left - min_mine_count
```

```
    other_cell_max_probability = 1 if other_cells == 0 else max_mines_in_hidden_

cells_not_in_config_Cells / other_cells

    # we can only compute and useother cell probabilities if using mine count
    # when we can use it, them if other cells have a lower probability, then ins
tead
    # just return the next random excluding all hidden cells in any component
    # otherwise just return the best cell found above
    if self.useMineCount and best_probability > other_cell_max_probability:
        to_exclude = []
        for l in config_cells:
            to_exclude.extend(l)
        return self.get_next_random(set(to_exclude))
    else:
        assert best_cell is not None
        return best_cell
```

See `BruteForceAgent.py` for more implementation details.

As mentioned earlier, the Brute Force agent takes advantage of the added knowledge of **total mine count.** What makes this algorithm effective is that it uses the total mine count to compare the cell with lowest probability of being a mine in any component to that of the other cells not in any component. Without total mine count, this comparison (and the added knowledge) cannot be used.

A concrete example of this would be if our first random guess to start the game gives a clue of $1$. If the density is $0.2$, then we want to choose a cell surrounding the initial guess, as $\frac{1}{8} < 0.2$ (assuming the initial guess has $8$ neighbors). But if density is $0.1$, then we want to look outside the initial guess, as $\frac{1}{8} > 0.1$.

Knowing the mine count also helps the brute force agent rule out possible configurations for a component. For example, if we have $2$ components with possible mine counts $7-9$ and $1-2$ with $10$ total mines, we can rule out the configurations with $7$ mines for component $1$, since $7 + (1 or 2) < 10$. Similarly if we have $8-12$ and $6-10$ for the possible mine counts, and $16$ total mines, we can truncate to $8-10$ and $6-8$. These removals are done after computing all configurations for each component, and thus do not speed up computations. However, they ensure that the agent is using the most possibly accurate probabilities.

## Further Improved Decision-Making

Below is a single iteration of the new algorithm which incorporates both the linear algebra method and, when there are no matrix solutions, the brute force method, to compute the safest next move via probabilities. The below iteration illustrates the linear algebra agent being insufficient to determine the next move, and thus prompts the brute force agent to determine it.

The full decision log for our v2 algorithm can be found in `/data/lin_alg_with_brute_log.txt`.

...

---

```
Cell (8, 8) safely revealed with clue: 1.
```

```
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8


Revealing cell (8, 8) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge, or add random if none available.

recrunching baseline

game:
[0, 0, 0, 1, -6, 1, 1, 1, 1, 0]
[0, 0, 1, 2, 2, 1, 1, -6, 2, 1]
[0, 0, 1, -6, 2, 1, 2, 1, 2, -6]
[0, 1, 2, 3, 3, -6, 3, 2, 2, 1]
[0, 1, -6, 2, -6, 4, -6, -6, 2, 0]
[0, 1, 2, 4, 3, 5, -6, -6, 3, 0]
[0, 0, 1, -6, -6, 4, -6, -6, 2, 0]
[1, 1, 2, 2, 3, -6, 4, 3, 2, 1]
[1, -6, 1, 0, 2, 3, -6, 1, 1, -6]
[1, 1, 1, 0, 1, -6, 2, 1, 1, 1]

knowledge:
[[ 0  0  0  0 -6  0  0 -1 -1 -1]
 [ 0  0  0  0  0  0  0 -1  0 -1]
 [ 0  0  0 -6  0  0  0  0  0 -1]
 [ 0  0  0  0  0 -6  0 -1  0 -1]
 [ 0  0 -6  0 -6  0 -6 -1 -1 -1]
 [ 0  0  0  0  0  0 -6 -1 -1 -1]
 [ 0  0  0 -6 -6  0 -6 -1 -1 -1]
 [ 0  0  0  0  0 -6  0 -1 -1 -1]
 [ 0 -6  0  0  0  0 -1 -1  0 -1]
 [ 0  0  0  0  0 -6 -1 -1 -1 -1]]

generated following row using (0,6):
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]
```

```
generated following row using (1,6):
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]

generated following row using (1,8):
[0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 2.]

generated following row using (2,6):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]

generated following row using (2,7):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]

generated following row using (2,8):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 2.]

generated following row using (3,6):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]

generated following row using (3,8):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1.
```

```
1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 2.]

generated following row using (7,6):
    [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 2.]

generated following row using (8,5):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
0. 1.]

generated following row using (8,8):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1.
1. 1.]

information matrix:
[[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 2.]
 ...
 [0. 0. 0. ... 0. 0. 2.]
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 1. 1. 1.]]

rref'd matrix:
[[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]

using row:
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
```

```
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]

using row:
[ 0.  0.  0.  0.  0.  0.  0.  0.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0. 0
.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 0.
0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.]

using row:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.]

using row:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1. -1.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.  1
. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0
. 0. 0. 1.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1.]

using row:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
```

```
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0. -1. -1.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0.  0.
0.  0. -1. -1. -1. -1.  0.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1.
1. 1.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
0. 1.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0.]

using row:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0.]

    Re-processing via lin alg not find new safe cells; proceeding to rando
mly select hidden cell.

couldn't find definitively safe location, beginning computing brute force
probabilities

game:
[0, 0, 0, 1, -6, 1, 1, 1, 1, 0]
[0, 0, 1, 2, 2, 1, 1, -6, 2, 1]
[0, 0, 1, -6, 2, 1, 2, 1, 2, -6]
[0, 1, 2, 3, 3, -6, 3, 2, 2, 1]
```

```
[0, 1, -6, 2, -6, 4, -6, -6, 2, 0]
[0, 1, 2, 4, 3, 5, -6, -6, 3, 0]
[0, 0, 1, -6, -6, 4, -6, -6, 2, 0]
[1, 1, 2, 2, 3, -6, 4, 3, 2, 1]
[1, -6, 1, 0, 2, 3, -6, 1, 1, -6]
[1, 1, 1, 0, 1, -6, 2, 1, 1, 1]

knowledge:
[[ 0  0  0  0 -6  0  0 -1 -1 -1]
 [ 0  0  0  0  0  0  0 -1  0 -1]
 [ 0  0  0 -6  0  0  0  0  0 -1]
 [ 0  0  0  0  0 -6  0 -1  0 -1]
 [ 0  0 -6  0 -6  0 -6 -1 -1 -1]
 [ 0  0  0  0  0  0 -6 -1 -1 -1]
 [ 0  0  0 -6 -6  0 -6 -1 -1 -1]
 [ 0  0  0  0  0 -6  0 -1 -1 -1]
 [ 0 -6  0  0  0  0 -1 -1  0 -1]
 [ 0  0  0  0  0 -6 -1 -1 -1 -1]]

separated into components, now printing each component and possible sets o
f mines within component:

for component:
[(0, 7), (0, 8), (0, 9), (1, 7), (1, 9), (2, 9), (3, 7), (3, 9), (4, 7), (
4, 8), (4, 9)]
took 0.03 seconds to compute following mine configurations
[(0, 7), (2, 9), (3, 7)]
[(1, 7), (2, 9), (4, 7)]
[(0, 7), (0, 8), (3, 7), (3, 9)]
[(0, 7), (0, 9), (3, 7), (3, 9)]
[(0, 7), (1, 9), (3, 7), (4, 8)]
[(0, 7), (1, 9), (3, 7), (4, 9)]
[(0, 8), (1, 7), (3, 9), (4, 7)]
[(0, 9), (1, 7), (3, 9), (4, 7)]
[(1, 7), (1, 9), (4, 7), (4, 8)]
[(1, 7), (1, 9), (4, 7), (4, 9)]

for component:
[(6, 7), (7, 7), (7, 8), (7, 9), (8, 6), (8, 7), (8, 9), (9, 6), (9, 7), (
9, 8), (9, 9)]
took 0.01 seconds to compute following mine configurations
[(7, 7), (8, 6)]
[(8, 6), (8, 7)]
[(6, 7), (7, 7), (9, 6)]
[(6, 7), (7, 8), (8, 6)]
```

```
[(6, 7), (7, 9), (8, 6)]
[(6, 7), (8, 6), (8, 9)]
[(6, 7), (8, 6), (9, 7)]
[(6, 7), (8, 6), (9, 8)]
[(6, 7), (8, 6), (9, 9)]
[(6, 7), (8, 7), (9, 6)]

mapping each cell to percentage of times it occurs in it's component's pos
sible mine configurations

found best cell (9,9) that was in 0.1% of it's component's configurations

...
```

---

This hybrid algorithm is quite powerful largely because it relies solely on computational methods and utilizes the full extent of the knowledge base at every iteration. In the above excerpt, which depicts a single iteration by the Linear Algebra/Brute Force Agent, the cell (8,8) was revealed but did not provide enough information to the knowledge base for the linear algebra method to yield useful results. The Brute Force Agent then calculated all the possible mine configurations for the remaining cells and was able to find that cell (9,9) had the least probability of being a mine, which it then chose to visit next. Had there been two or more minimum probabilities, the agents would have had no choice but to arbitrarily (randomly) pick one of the corresponding cells.

To improve the hybrid approach even further, we turn to the main weakness of the algorithm, which is the inefficiency of the brute force approach. The algorithm performance is discussed in detail in the following sections; however, the brute force approach as a whole could, in theory, be superceded by a more optimized algorithm. This might be one that does not require computing every possible mine configuration as a means of generating the probabilities of choosing a safe cell. In other words, the brute force approach is the least optimal "Plan-B" to the linear algebra approach, hence its name. Because of theoretical results relating to NP-Completeness, any algorithm for solving Minesweeper will require evaluating all configurations in a certain window. Using algorithms like linear algebra or Bayesian Expectation simply reduce the amount of computations that the brute force agent has to perform.

In summary, the only way to improve this hybrid approach is to add more algorithms on top of linear algebra + brute force, which would further diminish time efficiency.

## Performance

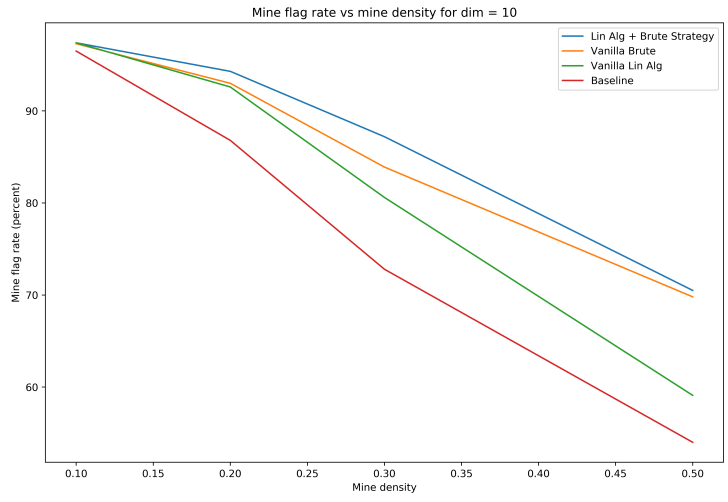
Below is a comparison table of the Brute Force (linear algebra + brute force probabilities) approach with the baseline approach, vanilla linear algebra approach, and even vanilla brute force approach (with no linear algebra used) in terms of mine detection rate (once again, DETECTED/total) as a percentage and performance time in seconds.

In [10]:

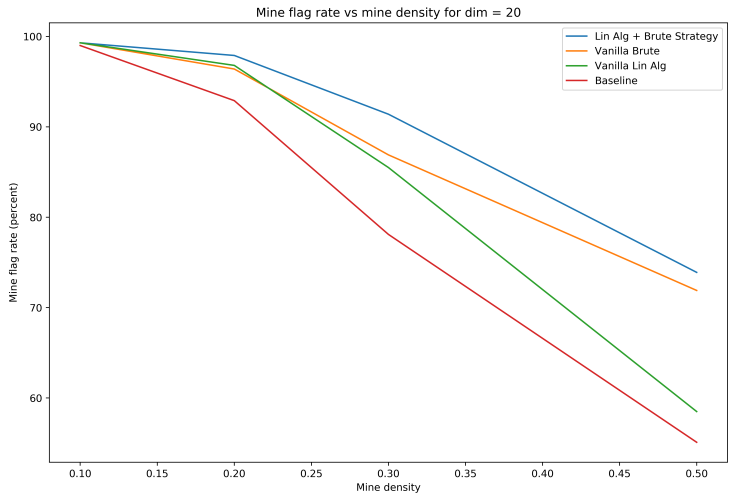
```
base_linalgBrute_df = df[['Dim', 'Density', 'Baseline', 'Lin Alg', 'Brute', 'Lin Alg + Brute']]
base_linalgBrute_df.transpose()
```

Out[10]:

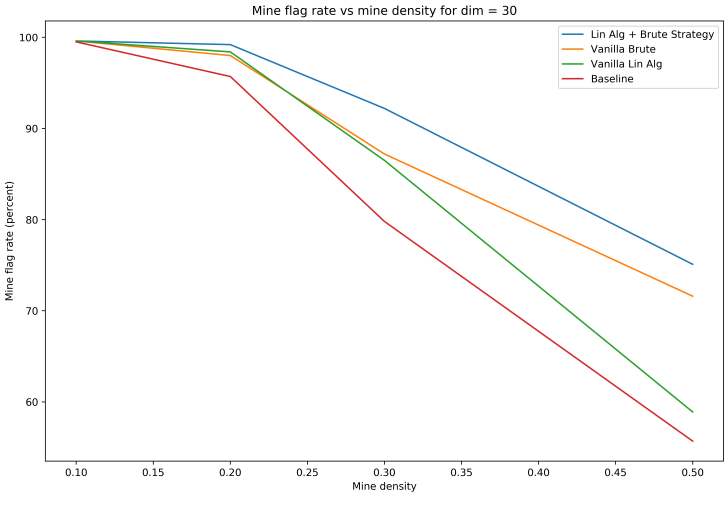
| | 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 11 | 12 | 13 | 14 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dim** | 10.0 | 10.0 | 10.0 | 10.0 | 20.0 | 20.0 | 20.0 | 20.0 | 30.0 | 30.0 | 30.0 | 30.0 | 40.0 | 40.0 | 40.0 |
| **Density** | 0.1 | 0.2 | 0.3 | 0.5 | 0.1 | 0.2 | 0.3 | 0.5 | 0.1 | 0.2 | 0.3 | 0.5 | 0.1 | 0.2 | 0.3 |
| **Baseline** | 96.5 | 86.8 | 72.8 | 54.0 | 99.0 | 92.9 | 78.1 | 55.1 | 99.5 | 95.7 | 79.8 | 55.7 | 99.7 | 96.5 | 81.1 |
| **Lin Alg** | 97.4 | 92.6 | 80.6 | 59.1 | 99.3 | 96.8 | 85.5 | 58.5 | 99.6 | 98.4 | 86.5 | 58.9 | 99.7 | 99.0 | 88.4 |
| **Brute** | 97.3 | 93.0 | 83.9 | 69.8 | 99.3 | 96.4 | 86.9 | 71.9 | 99.6 | 98.0 | 87.2 | 71.6 | 99.7 | 98.5 | 88.0 |
| **Lin Alg + Brute** | 97.4 | 94.3 | 87.2 | 70.5 | 99.3 | 97.9 | 91.4 | 73.9 | 99.6 | 99.2 | 92.2 | 75.1 | 99.7 | 99.4 | 93.2 |



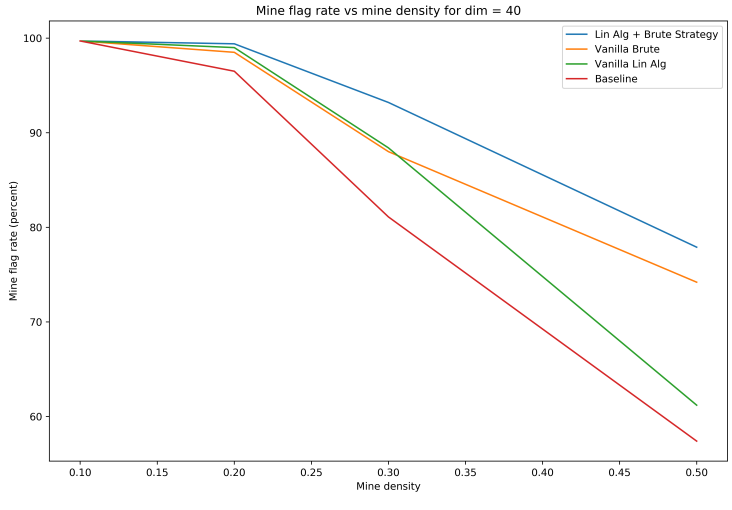

Dim 10




Dim 20




Dim 30




Dim 40

Each data point is obtained via 100-200 trials. Note again that our linear algebra + brute strategy is using **total mine count.**

Since the vanilla linear algebra approach already uniformly outperformed baseline, it is no surprise that the linear algebra + brute force approach does as well. Again, we have uniformly outperformed strategy v1 (linear algebra) with strategy v2 (linear algebra + brute force) in terms of mine flag rates.

Interestingly, one can see how for higher $dim$ (30-40), vanilla brute and vanilla linear algebra seem to get entangled in terms of mine flag rates for low mine densities (0.10-0.30). This is likely because we coded brute force to short circuit once its component size exceeds 20 for runtime reasons, after which point it reverts to baseline. Without this cap, we would expect vanilla brute force to outperform vanilla linear algebra nearly always.

And ultimately, brute force tends to decisively beat linear algebra for higher mine densities, as seen above. This is because there are lots of possible ways to configure mines, but the brute force agent will find the cells that are least likely. On the other hand, vanilla linear algebra (system of equations) can not do that, as it purely finds deterministic inferences.

# Efficiency

Runtime data collected was quite variable because multiple processes were running concurrently. However, we found was that runtimes tended towards the following pattern:

**Vanilla brute force $\geq$ brute force + lin alg $\geq$ vanilla lin alg $\geq$ baseline**

**Implementation-specific constraints**

In general, it makes sense that vanilla brute force has higher runtimes, because although it will still locate cells that are definitively safe, it currently only picks one spot, i.e. the one with the least probability of being a mine. We could decrease its runtime by enqueuing all cells with zero mine probability (if they exist) so that we avoid redoing the expensive brute force operation to find out something is safe, especially if we already knew it was safe. We do not currently do this as our implementation ( `probabilityMethod(...)` ) currently only returns one coordinate.

The combined brute force + linear algebra strategy has lower runtimes than vanilla brute force because it executes the linear algebra inference step first. This allows for a lot of simple deductions to take place, growing the KB. Consequently, the agent ends up calling brute force fewer times, and it also has a smaller space of configurations to calculate, on average, when trying to find probabilities.

**Problem-specific constraints**

Across various $dim$ and densities, we observed:

1. At low $dim$ and mine densities, linear algebra was close to baseline in terms of runtime, and the other two options had extremely high runtimes.
2. At higher $dim$ and mine densities, linear algebra, vanilla brute, and combined linear algebra + brute had extraordinarily high runtimes.

In fact, we noticed several $dim$, density threshold "walls" beyond which runtimes became excessively long. For example, for density $d = 0.3$, $dim = 30$ was one such "wall." For $d = 0.5$, $dim = 20$ was a "wall." In those cases, runtimes blew up even though they performed better, in terms of mine flag rate, than vanilla linear algebra.

This is simply because higher $dim$ and mine densities are 'harder' games to solve. Higher density means less fast inferences by baseline, and more time spent in expensive linear algebra and brute steps. Linear algebra closes in on brute for higher dim simply because we cap the component size we are willing to compute brute force probabilties for at 20. If we had a fast enough computer and did not cap brute, then at high dim we would still see brute force take significantly longer than linear algebra.

# Performance of brute force + linear algebra without mine count

Earlier, we stated that brute force exploits the knowledge of the total mine count, and this contributes towards its efficacy. Now, we compare the brute force + linear algebra driver both with and without the mine count in use for a variety of $dim$ and mine densities.

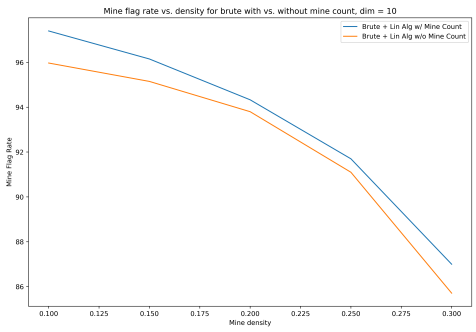
Note that each data point below is the result of 100-200 controlled trials (for the same game).

In [2]:

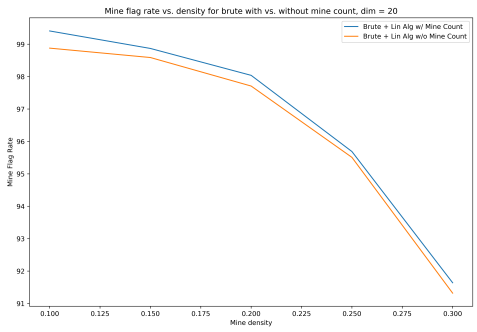
```
mineCtRates = pd.read_csv('./data/mineCtRates.csv', index_col=0)
mineCtRates.transpose()
```

Out[2]:

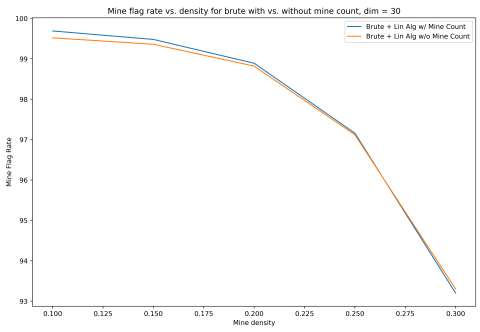
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **dim** | 10.00 | 10.00 | 10.00 | 10.00 | 10.00 | 20.00 | 20.00 | 20.00 | 20.00 | 20.00 | 30.00 | 30.00 |
| **density** | 0.10 | 0.15 | 0.20 | 0.25 | 0.30 | 0.10 | 0.15 | 0.20 | 0.25 | 0.30 | 0.10 | 0.15 |
| **baseRate** | 95.97 | 95.15 | 93.80 | 91.10 | 85.71 | 98.88 | 98.59 | 97.71 | 95.51 | 91.32 | 99.52 | 99.36 |
| **mineCtRate** | 97.40 | 96.15 | 94.33 | 91.70 | 87.00 | 99.41 | 98.87 | 98.04 | 95.69 | 91.64 | 99.69 | 99.48 |




Dim 10




Dim 20




Dim 30

From above, we can see that the usage of the mine count marginally improves the mine flag rates (when using brute force + linear algebra), but this improvement is small, with somewhere between a $0$ and $2$ percent improvement. On the other hand, we can conclude that mine flag rates when the mine count is utilized are better or equal to mine flag rates when the mine count is not utilized.

This reiterates the fact that if the brute force agent does not have the total mine count, then it cannot make the comparison between cells with the lowest mine probability and cells not in any component, i.e. not bordering any safe cell. This added information gives the agent a performance boost.

As mentioned earlier, the Brute Force agent takes advantage of the added knowledge of **total mine count.** What makes this algorithm effective is that it uses the total mine count to compare the cell with lowest probability of being a mine in any component to that of the other cells not in any configuration.

Moreover, we can also see the following trends:

- The added benefit of using the mine count decreases over higher $dim$; note the smaller gap for $dim = 30$ vs that of $dim = 10$.
- The added benefit of using the mine count decreases over higher mine densities, and this effect is especially pronounced for higher $dim$.

This is due to the increasing complexity of Minesweeper games for higher $dim$ and mine densities. The more complex the game, the harder time the brute force + linear algebra agent has when solving it, regardless of whether or not it knows the total mine count. More specifically, mine count helps linear algebra primarily near the end of the game, as early on the row added using mine count has too many variables. For high $dim$, end game is a much smaller portion of the overall board, so the effect is less pronounced. For brute force, a possible explanation of why mine count helps less at high $dim$ is that at high $dim$, component sizes get large and we end up not even bothering to compute anything, so we can't use the mine count anyways.

# Dealing with Uncertainty

This section deals with a variation of the MineSweeper game in which the clue itself is not assumed to be correct. That is, we deal with uncertainty on the accuracy of the clue itself. Accordingly, instead of merely revealing the actual clue via `clue = self.game.board[x][y]` , we now use the `getClue(...)` function found in `Agent.py` which returns an uncertain clue.

Recall that toggling between these types of uncertain clues, as well as having no uncertainty (as before), is done via the `self.uncertaintyType` attribute in the `Agent` class.

# Accurate information, random reveal

In this case, the revealed clue is accurate, but it is only revealed with some random probability $p$. This complicates the game because the knowledge base is not updated as frequently as before, leading to less opportunities for inference to deterministically detect mines or safe cells.

## Uncertainty implementation

For the random reveal type of uncertainty, the `getClue` function needs to be passed $p$, the probability that any given clue will be revealed. Then, the code to reveal the clue randomly is simple:

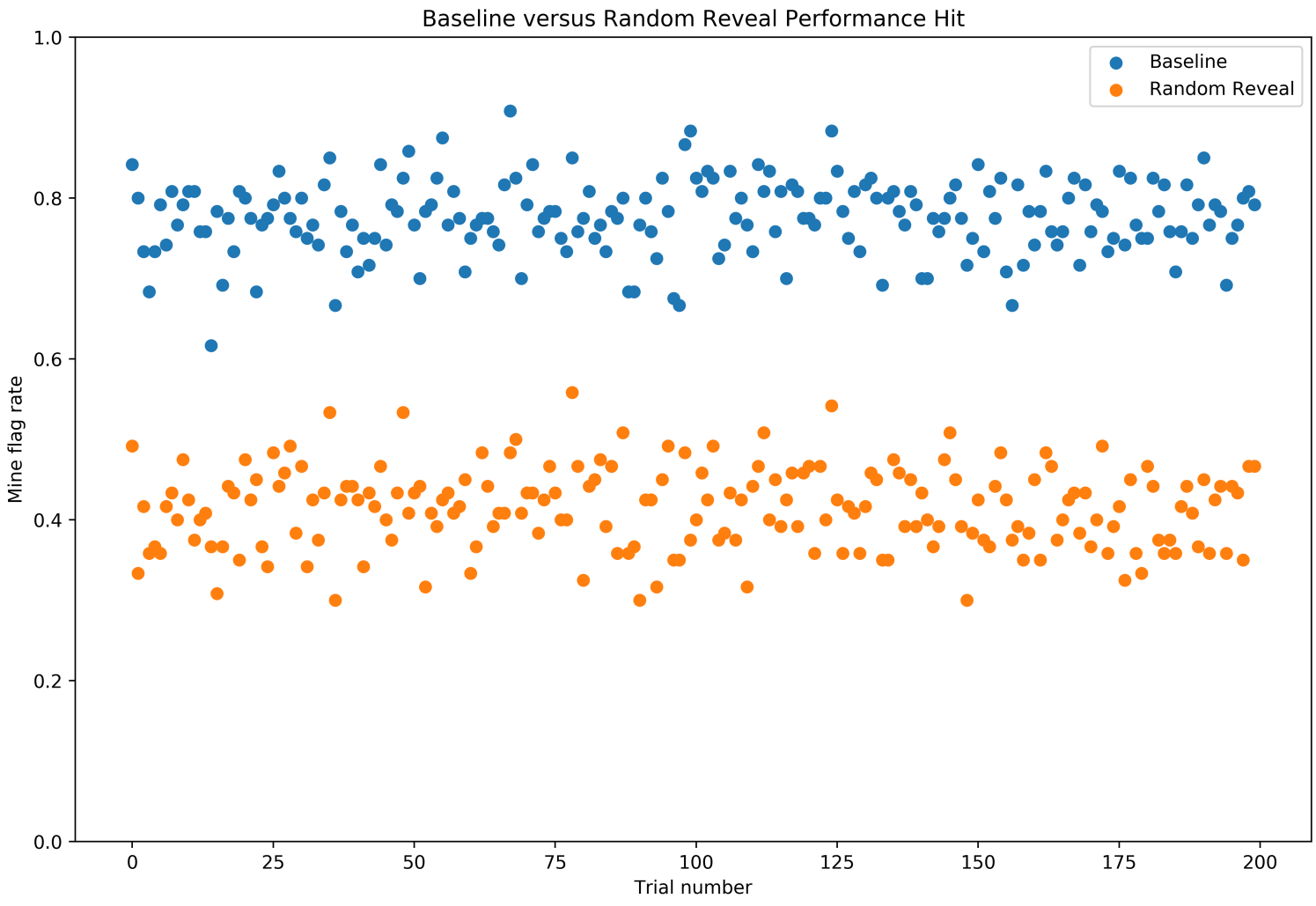
In [ ]:

```
if random.random () < p:
    return self.game.board[x][y]
else:
    return -1
```

where -1 is returned in place of an actual clue. The driver is coded such that if it receives a -1 clue, it will simply continue.

## Performance hit

We compare Random Reveal against the baseline strategy to get a sense of how the mine flag rate reacts to this uncertainty. For the following 200 trials, we use Random Reveal with $p = 0.02$; that is, a significantly small probability of revealing the clue.




We see that, expectedly, the agent is uniformly worse with the random reveal level of uncertainty, with flag rates approximately 40% lower than baseline, on average. As mentioned earlier, the agent has strictly less information in its knowledge base at any point in time than it would have at that step without any uncertainty measures.

Note also that we used $p = 0.02$, a very small probability, because while higher probabilities tested in the range $[0.3, 0.7]$ also resulted in strictly worse flag rates, they did not produce dramatic disparities as seen here. We found that it takes a relatively small $p$ value in order to significantly affected the flag rates.

## Strategy adaptation

We first need to assume that we are provided with the information of the number of mines given in the grid. Without this information, it becomes incredibly difficult to estimate the probability. The problem with this situation is that we cannot trust any of our knowledge base, since our clues are all based on probabilities.

What we could do is use probabilistic inference here. But the problem remains, since there is still no way of knowing if our knowledge base will consistently be correct. And if it is not correct, then the game will typically not last too long since mine detection will be quite off from the baseline approach where the clues are given to you with full certainty.

When we approach this problem, we are given a board with clear cells that have attached clues with a certain probability, or mines. We first must calculate the probability of landing on a mine using Bayes' Theorem: which would be taking the (total number of mines -1) multiplied by (total number of cells -1) divided by (total number of cells choose total number of mines). We are assuming the total number of mines is a constraint provided to us. Each cell now has an associated probabiltiy. This will be our base probability. We can store this as `base_prob`.

When we approach a cell that has a probabilistic, "true" clue, our implementation idea is to use the clue and its neighbors with an equal probability. Assuming that none of a cell's neighbors are uncovered, each probability would be 1/8. And the rest of the cells would have a probability of (total number of mines -1) divided by (total number of cells - 8). Now we can calculate the probabilies of each of the surrounding cells and the chance of there being a mine in the surrounding cells. But to make more accurate predicitions, we should choose to reveal neighbors, one by one, with the associated lowest probability. Once we have the probabilities of two neighbors, we have doubled our information and also doubled the accuracy by doing a rerun of all the probabilites surrounding the cell.

This step can be repeated over and over again until the table is solved. There is no certainty that the table will be accurately solved each time since everything is a probability and not an accurate clue. The chance of explosions is still significantly higher to any approach we have discussed previously in regards to accurate clues. Since, each step of this process, our clues start off with weak probabilistic inference, the chance of explosion is greater in the beginning. But as the game progresses (if it does progress), the chance of survival also significantly increases.

# Flawed Knowledge-Base: Optimistic clue

In this case, the clue is always revealed, but it may underestimate the clue with uniform probability, i.e. uniform distribution over the range of possible neighboring mines.

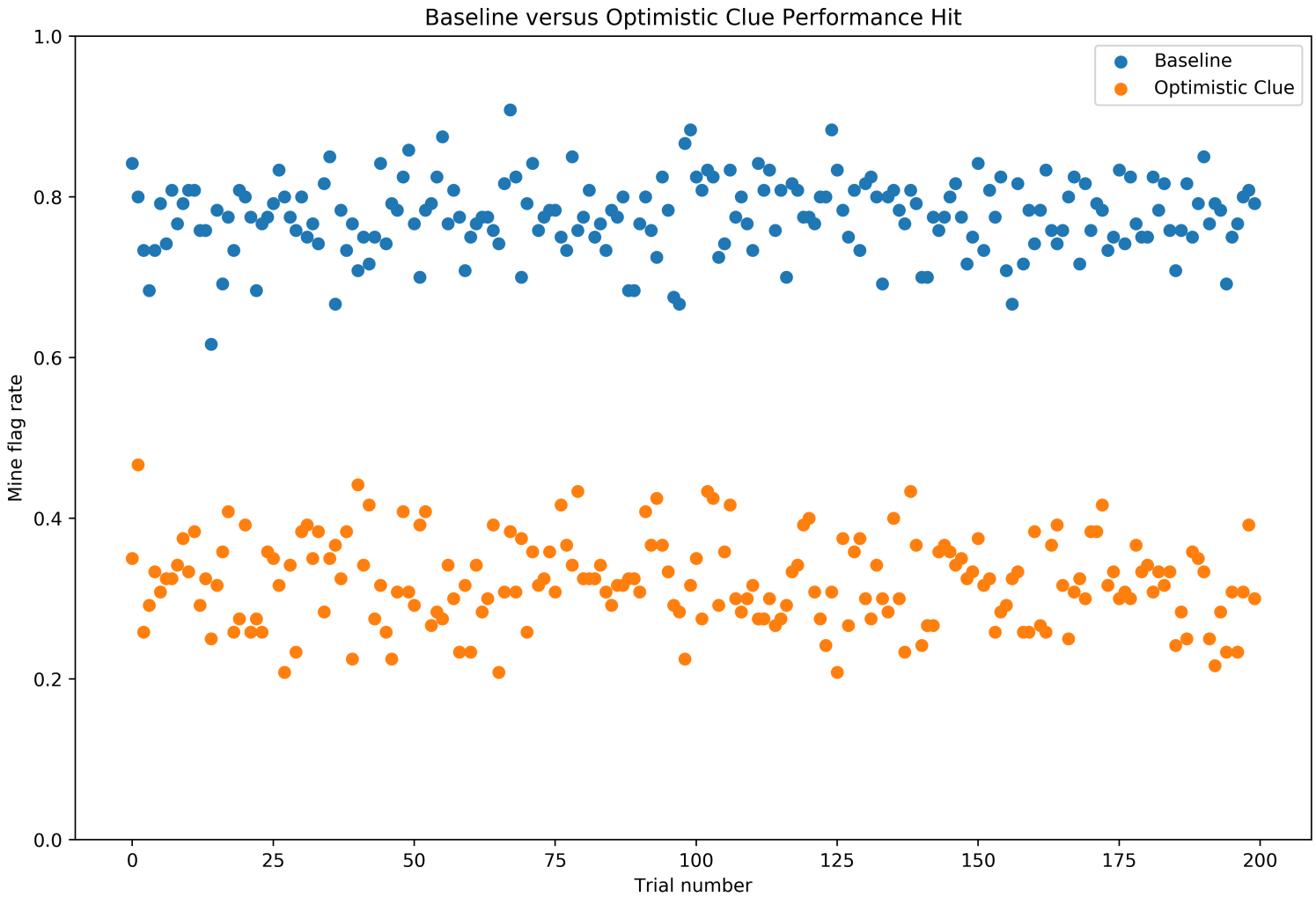
## Uncertainty implementation

Here, the `getClue` function will return an integer uniformly at random from the range $[m, C]$, where $m$ is the number of mine neighbors already revealed, i.e. the minimum number of possible total mine neighbors, and $C$ is the true clue value. That way, the returned value will be somewhere in the range of possible values but always underestimating the true clue.

In [ ]:

```
if numMineNbrs < self.game.board[x][y]:
    return random.randint(numMineNbrs, self.game.board[x][y])
elif numMineNbrs == self.game.board[x][y]:
    return self.game.board[x][y]
```

## Performance hit



We see that the agent acting on optimistic clue knowledge is uniformly worse than the baseline agent, with flag rates approximately 50% lower. This is because with clues which possibly reveal fewer mine neighbors, the agent is unable to deduce as many deterministically mine neighbors. It also may incorrectly determine some cells are safe, because the clue is an underestimate. This fact holds true using the linear algebra + brute force approach as well.

## Strategy adaptation

Similarly to our linear algebra discussion earlier, consider the equation $x_1 + x_2 + \cdots + x_n \geq C$, where $x_i$ is a cell which is 1 if mine, 0 if safe, and $C$ is the given clue.

Then we naturally see that $\sum_i x_i$ gives the true number of mines, and $C$ underestimates this: representing exactly this case of **optimistic clues.**

Leveraging this, we turn our discussion of linear algebra to the discussion of [linear optimization, (https://en.wikipedia.org/wiki/Linear_programming)](https://en.wikipedia.org/wiki/Linear_programming) which is the related field considering systems of linear equations where each equation represents not an equality but a constraint ($\geq$ or $\leq$), and a solution is optimal if it maximizes or minimizes a certain objective function of the variables $z = f(x_i)$.

Because we utilize systems of linear equations in the linear algebra method above, we see it convenient to adapt this portion to a linear optimization model.

### Linear optimization theory & implementation

We write another linear optimization agent which inherits from the linear algebra + brute force agent above, with the following simple change: **Instead of finding a solution to the system, we are trying to find the optimal one.** If none can be found, then simply revert to solving the system as before, regardless of the clues' accuracy.

See `LinOpt.py` for more implementation details.

For our optimal clue case, we are considering systems where each equation is of the form: $x_1 + x_2 + \cdots + x_n \geq C$. The linear programming problem we use is:

Max $z = \sum_i x_i$ subject to constraints $Ax \geq b, x_i \geq 0$.

Here, the expression $Ax \geq b$ is nothing more than the system of equations laid out in a matrix with $\geq$ symbols instead of $=$ symbols.

The primary method of solving linear programming problems is the [simplex method, (https://personal.utdallas.edu/~scniu/OPRE-6201/documents/LP4-Simplex.html)](https://personal.utdallas.edu/~scniu/OPRE-6201/documents/LP4-Simplex.html) which takes in the augmented matrix $[A|b]$ with a row added to represent the objective function, called the **initial tableau.** Then, it iteratively uses row operations to return a matrix with the solution to the problem.

However, the simplex method requires initial tableaus be converted to a specific [canonical form: (https://www.usna.edu/Users/math/wakefiel/_files/documents/sa305/notes/note3-25.pdf)](https://www.usna.edu/Users/math/wakefiel/_files/documents/sa305/notes/note3-25.pdf)

Max $z$ s.t. $Ax = b, x_i \geq 0$.

We use the Two Phase Simplex Method (http://www.maths.qmul.ac.uk/~ffischer/teaching/opt/notes/notes8.pdf) here to solve this system. Without diving too much into linear optimization theory, this requires us to pass in a different augmented matrix to mediate the problem of having $\geq$ instead of $=$ symbols. Then, we solve the optimization problem and return the appropriate solution matrix. The below code walks through this process of transforming and solving, but see the `simplexOptimistic(...)` method in `LinOpt.py` for more implementation details.

In [ ]:

```
# optimistic matrix: has inequalities [row] >= clue-numMineNbrs
def simplexOptimistic(matrix):
    # Phase I: solve for artificial variables
    # concatenate -I_n to right (without last col) for slacking vars to resolve
>= into =
    tableau = np.column_stack((matrix[:, :-1], -1*np.identity(matrix.shape[0], d
type=int)))
    # concatenate I_n to right (without last col) for artificial vars (to get in
it basic fsbl soln)
    tableau = np.column_stack((tableau, np.identity(matrix.shape[0])))
    # concatenate back the last col
    tableau = np.column_stack((tableau, matrix[:, -1]))
    # add obj row: max -(all artificial vars)
    obj_row = np.concatenate(([0]*(matrix.shape[1]-1+matrix.shape[0]), [1]*matri
x.shape[0], [0]))
    tableau = np.row_stack((tableau, obj_row))
    # row reduce obj row (art vars basic)
    tableau = makeBasic(tableau)
    # solve Phase I with simplex
    tableau = simplex(tableau)
    # Phase I fail: if z != 0, then no optimal solution exists
    if tableau is None or tableau[-1, -1] != 0:
        return None

    # Phase II: update and simplex
    # delete artificial vars' columns
    tableau = np.column_stack((tableau[:, :(matrix.shape[1]-1 + matrix.shape[0])
], tableau[:, -1]))
    # update objective function: all -1s except for slacking vars
    obj_row = np.concatenate(([-1]*(matrix.shape[1] - 1), [0]*(matrix.shape[0] +
1)))
    tableau[-1] = obj_row
    # row reduce to get basic vars std col vectors
    tableau = makeBasic(tableau)
    # simplex as usual
    tableau = simplex(tableau)

    # if sol'n, cut out cols for slackvars and obj row (only returning row vals
as in rref)
    if tableau is not None:
        tableau = np.column_stack((tableau[:, :matrix.shape[0]], tableau[:, -1])
)[:-1, :]
    return tableau
```

This of course relies on the `simplex(...)` method to carry out the necessary iterative algorithmic steps:

In [ ]:

```
def simplex(tableau):
    i = 0
    while (not checkOptimal(tableau)):
        e = enteringVar(tableau)
        d = departingVar(tableau,e)
        tableau = rowReduce(tableau,d,e)
        if tableau is None or hasNoSolution(tableau):
            return None
        i+=1
        # hardcode short-circuit if we get stuck in degeneracy / bland's rule cy
cling issues
        if i > 20000:
            return None
    return tableau
```

To see more of how we implemented the linear optimization operations (e.g. row operations, selecting departing/entering variables, handling termination conditions and cycling) behind this, see the `simplex(...)` and related methods in `LinOpt.py` .

**Trial results**

Here is a test run of the linear optimization agent (on a 10 by 10 board with 20 mines) attempting to mediate optimistic clues with the Two Phase Simplex Method:

---

```
Solving with LINEAR OPTIMIZATION strategy

Uncertainty: optimistic

Cell (1, 5) safely revealed with clue: 1.
    # safe neighbors: 0
    # mine neighbors: 0
    # hidden neighbors: 8
    # total neighbors: 8


Revealing cell (1, 5) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,

or add random if none available.

...
```

```
generated following row using total mine count:
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.  1. 1
.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. 1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1. 20.]

Linear optimization for optimistic uncertainty FAILED. Proceeding with reg
ular Lin Alg.

solved matrix:
[[ 1.  1.  1.  1.  0.  0.  0.  1.  1.  1.  1.  1.  1.  1.  0.  0.  0.  1.
1.  1.  1.  1.  1.  1.  0.  0.  0.  1.  1.  1.  1.  1.  1.  1.  1.   1
.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.   1.
1.  1.  1. 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1  1
.  1.  1.  1.  1.  1.  1.  1. 17.]

[ 0.  0.  0.  0.  1.  1.  1.  0.  0.  0.  0.  0.  0.  0.  1.  0.  1.  0. 0
.  0.  0.  0.  0.  0.  1.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.   0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0
.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.   0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  3.]]

...


***** GAME OVER *****

Game ended in 2.826066255569458 seconds

Safely detected (without detonating) 35.0% of mines
```

Here and throughout the log (which can be read in full at `/data/optimisticLog.txt`), we see that the Two Phase Simplex Method failed to find an optimal solution of equations, and reverted to Lin Alg approach. In this case, our strategy is unable to overcome the significant performance hit of the inaccurate (optimistic) clue, with a low flag rate of 35%.

Before analyzing why this failure occurs, we proceed to also implement a similar solution for cautious clues.

# Flawed Knowledge-Base: Cautious clue

In this case, the revealed clue has the potential to be over-estimating; it is also chosen uniformly over the range of possible values for the true number of mine neighbors, assuming it is greater than or equal to the real clue.

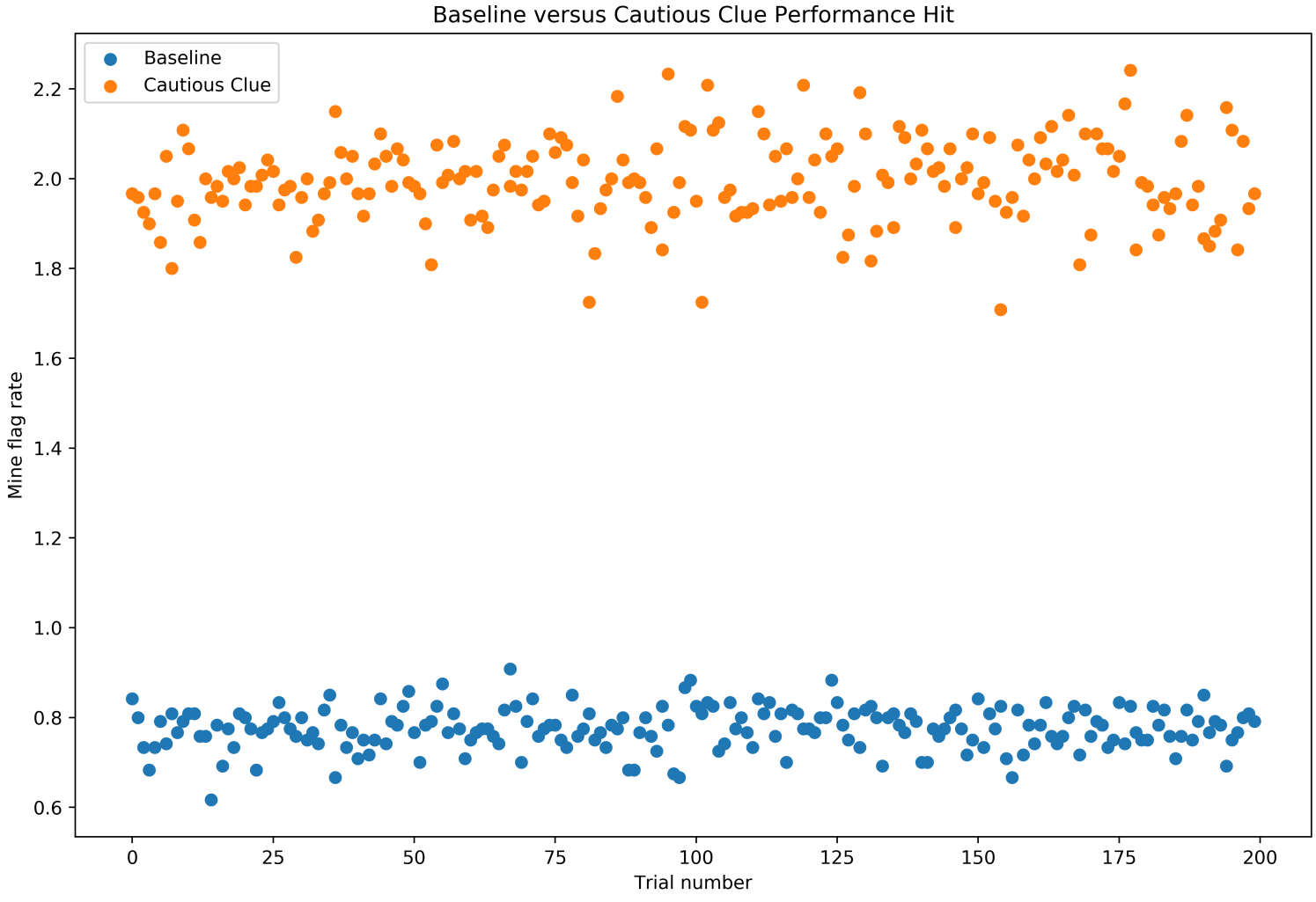
## Uncertainty implementation

The `getClue` function will return an integer uniformly at random from the range $[C, M]$, where $C$ is the true clue as before, and $M$ is the number of mine neighbors already revealed plus the number of hidden neighbors, i.e. the maximum number of possible total mine neighbors. That way, the returned value will be somewhere in the range of possible values but always overestimating the true clue.

In [ ]:

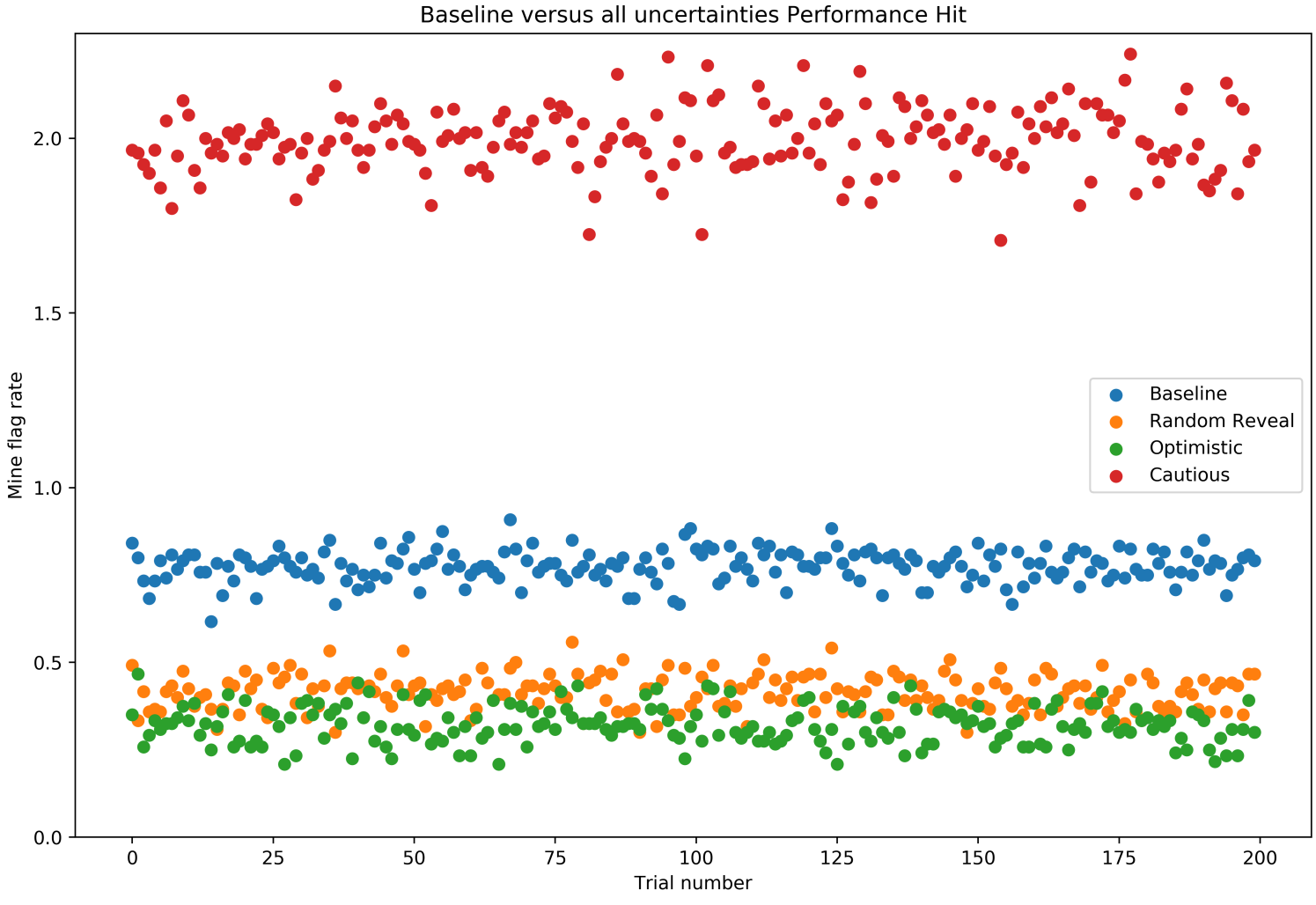
```
if self.game.board[x][y] < numMineNbrs + numHiddenNbrs:
    return random.randint(self.game.board[x][y], numMineNbrs + numHiddenNbrs)
elif self.game.board[x][y] == numMineNbrs + numHiddenNbrs:
    return self.game.board[x][y]
```

## Performance hit




We see that the agent acting on optimistic clue knowledge is uniformly worse than the baseline agent, but in a different way: one should immediately see that the mine flag rates are absurdly above 1.0. This is because the given clues possibly reveal more mine neighbors than there really are, and so the agent is trigger-happy in terms of deducing that certain cells are deterministically mines when in fact they are not. Then, the agent doesn't uncover those cells, even if they are safe, jacking the mine flag rate above 100%.

Now that all three complications have been implemented, we can compare our baseline algorithm's performance hit in each case:




One can see that each successive complication is successively and strictly worse, if we assume that overestimating mines is the worst case scenario of them all. That is, cautious clues are worse than optimistic clues, which are in turn worse than random reveals and no uncertainty at all.

## Strategy adaptation

With cautious clues, we are concerned of equations of the form: $x_1 + x_2 + \cdots + x_n \leq C$. Similarly to the above case with optimistic clues, we use the simplex method, although it is simpler in the sense that constraints having $\leq$ signs is easier to remediate than if they have $\geq$ signs.

See the `simplexCautious(...)` and related methods in `LinOpt.py` for more implementation details.

In [ ]:

```
# cautious matrix: has inequalities [row] <= clue-numMineNbrs
def simplexCautious(matrix):
    # concatenate I_n to right (without last col) for slacking vars to resolve <
= into =
    tableau = np.column_stack((matrix[:, :-1], np.identity(matrix.shape[0])))
    # concatenate back the last col
    tableau = np.column_stack((tableau, matrix[:, -1]))
    # add row for objective function: all -1s except for slacking vars
    obj_row = np.concatenate(([-1]*(matrix.shape[1] - 1), [0]*(matrix.shape[0]+1
)))
    tableau = np.row_stack((tableau, obj_row))

    # simplex as usual
    tableau = simplex(tableau)

    # if sol'n, cut out cols for slackvars and obj row (only returning row vals
as in rref)
    if tableau is not None:
        tableau = np.column_stack((tableau[:, :matrix.shape[0]], tableau[:, -1])
)[:-1, :]
    return tableau
```

### Trial results

Just as with the optimistic clue, we run into the same issue where the simplex method fails. Below is a sample from a trial run (similar to before), and the full log can be read in `/data/cautiousLog.txt`.

---

```
Solving with LINEAR OPTIMIZATION strategy

Uncertainty: cautious

BOOM! Mine detonated at (9, 2).


Revealing cell (9, 2) led to no conclusive next move (either DETONATED or
all neighbors MINES).

Will attempt to re-deduce & enqueue new safe cell(s) from all of current k
nowledge,

or add random if none available.

...

generated following row using total mine count:
```

```
[ 1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. 1
.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.
1.  1.  1.  1.  1.  1. 17.]
```

```
Linear optimization for cautious uncertainty FAILED. Proceeding with regul
ar Lin Alg.
solved matrix:
```

```
[[ 1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.  1.  0.  0.  1.  1.  1.  1.   1
.  1.  1.  1.  0.  0.  1.  1.  1.  1.  1.  1.  1.  1.  0.  0.  1.  1.   1.
1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.   1.  1
.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.   1.  1.
0.  1.  1.  1.  1.  1.  1.  1. 16.]

[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  1.  0.  0.  0.  0.  0.
0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  1.  1.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0
.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  1.]]

...


***** GAME OVER *****

Game ended in 0.463541030883789 06 seconds

Safely detected (without detonating) 170.0% of mines
```

As discussed earlier, note that the simplex method procedure fails, and the cautious clue results in a hyperactive mine flag rate of 170% due to false positives.

## Implementation challenges for linear optimization

This method (for both optimistic and cautious clues) fails precisely because our problem is not well-suited to the mathematical reality of linear optimization. This is true even when using the total mine count.

Linear optimization theory rests upon a very precise set of geometric conditions. Imagine plotting each constraint inequality, and the region bounded by all these lines. For our system of linear Minesweeper equations, would this region ever exist? What would it look like? In fact, every linear programming problem must have a feasible and convex region of points, i.e. the [feasible region (https://en.wikipedia.org/wiki/Feasible_region)](https://en.wikipedia.org/wiki/Feasible_region), which satisfies the conditions presented by the constraints; without this, the problem will not return a solution. It does not appear that the systems of linear equations we are solving yield such a feasible region, given the failures of our trials.

Moreover, another issue may lie in our objective function: to maximize the total number of mines detected at each pass. This seems right in the sense of intelligent decision-making, but does not correspond to mathematical reality. Consider the case of optimistic clues. Intuitively, if each constraint is uses a $\geq$ symbol and the objective function is to maximize a certain value, then it would appear that even if there was a feasible region, then there wouldn't be an optimal solution; one might be able to increase the objective function infinitely.

To fix this, we would have to re-think both of these points. We might end up needing to implement a different method of getting a system of linear equations, e.g. one which solves for safe cells, and not mine cells. Then the system would want to maximize the number of safe cells detected at each pass, because those cells give us new actionable information for inference.

We might also change our objective function to minimize the number of detected mines, to be as conservative as possible. That would mean trying to define the number of detected mines as a function of the mine cells in the system of linear equations used here, or perhaps a modified system.

We might even want to explore looking at the implications of the theory of [duality, (http://web.mit.edu/15.053/www/AMP-Chapter-04.pdf)](http://web.mit.edu/15.053/www/AMP-Chapter-04.pdf) finding dual problems to the ones at hand, and attempting to solve those.

All in all, linear optimization seems to head in the right intuitive direction, but either is not well-suited to solving Minesweeper with uncertain clues, or needs significant massaging which would depart from the core of the linear algebra systems relayed here.