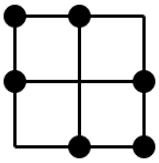
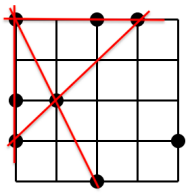
No-three-in-line problem: Creating an algorithm to find solutions on large grids



Introduction: The no-three-in-line problem, posed initially by Henry Dudeney in 1900, asks how many points can be fit onto the vertices of a board of n x n dimensions without there being any 3 points in a line. For example, the rules allow for 6 points to be placed on the intersections of a 3x3 grid like in the figure on the right since no 3 points fall on one line. The figure on the left is a configuration that fails because 3 points occur along the lines shown in red.



On an n x n grid, it is impossible to fit more than $2n$ dots on the intersections of the grid as that would force at least 3 points in one horizontal row, therefore, $2n$ is the upper limit for an n x n grid. However, an unsolved question is, can $2n$ points always fit on an n x n grid such that there are no 3 points in line? Mathematicians have puzzled over this problem for decades, and there exists no proof of this problem. Solutions using $2n$ dots have been found by other mathematicians for values of n from 3 through 46. For grid sizes larger than 46 it is currently unknown what is possible. It becomes increasingly complex to fit $2n$ dots onto larger boards because the possibilities of dot arrangement grow exponentially. Efficient algorithms for generating and evaluating possible solutions are required to make progress on this problem.

My goal is to develop and optimize an algorithm, implemented in Python, to efficiently generate solutions using $2n$ dots in order to determine which values of n are possible on larger grids.

Methods of Generation:

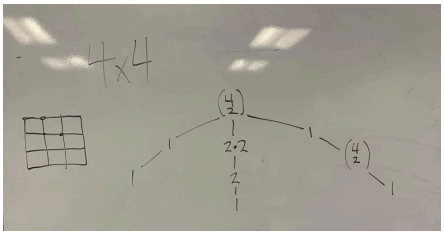
To start, I wrote a python script to randomly place points on a grid. The main challenge with a completely random approach is that the search space grows extremely quickly by the equation $s(n) = \binom{n^2}{2n}$. If we assume a computer can check ten million configurations a second, for a 10x10 grid, we take the total number of possible configurations and divide it by 60 seconds, 60 minutes, 24 hours, and 365 years to find the number of years. It would take us close to $\frac{535983370403832}{60 \cdot 60 \cdot 24 \cdot 365} = 1.7$ million years to check all of the potential solutions within the search space for a 10x10 grid. A more efficient method is needed.

Method B:

A way to increase the efficiency would be to more quickly eliminate the simple configurations that we know will have 3 points in a line. In other words, we can reduce the number of possibilities we have to evaluate by changing the way we initially place points on the grid. For example, it is never possible to select more than 2 points in one row because that solution will then be guaranteed to contain 3 in a line. A strategy instead would then be to choose exactly 2 points from each row. The search space can now be defined by the equation $s(n) = \binom{n}{2}^n$. I got this equation because there are $\binom{n}{2}$ ways to pick two dots from each row and there are n total rows so we raise $\binom{n}{2}$ to the power of n. Although this was an improvement to completely random placement, I thought it was possible to do better.

Method C:

I decided to improve the algorithm by disallowing the placement of more than 2 dots per column along with disallowing more than 2 dots per row like I did in Method B. Following these rules results in placing 2n dots in an n x n grid. It becomes tricky to calculate the search space for this method as the number of options for each point depends on where the previous points were placed. The best way to approach this is by going row by row down the grid and placing two dots in each row until columns are filled up with two dots. For smaller boards I can build trees of the number of possible locations for each dot depending on where we have previously placed dots. I then would multiply down each branch of the tree to find the total possible boards of size $n\ x\ n$ that can be built with two in each row and column.



For a 3x3 board there are $\binom{3}{2}$ ways to pick two dots from the first row because there are 3 locations to choose from and 2 dots. The second row needs to have one dot in the unused column which leaves two possibilities for the second dot. The third and last row only has one possible configuration of the dots because

there will only be two columns with less than two dots. Therefore, there are 6 possibilities for a 3x3 board. A $4x4$ board is more complicated because the possibility tree branches out more as shown in the figure on the left.

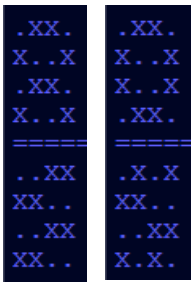
By multiplying down each branch we get a total number of $(6 \cdot 1 \cdot 1) + (6 \cdot 4 \cdot 2 \cdot 1) + (6 \cdot 1 \cdot 6 \cdot 1) = 6 + 48 + 36 = 90$ possibilities. By building these trees, I was able to figure out that there are 2040 ways to fill a $5x5$ grid with two dots per row and column and $67,950$ ways to fill a $6x6$ grid with two dots per row and column. With these data points I was able to search the OEIS to find more terms in this sequence of 4 numbers. (The OEIS is a useful tool for finding a pattern in a string of integers.) The OEIS told me that this series of four numbers: 6, 90, 2040, 67950 is sequence 1499 in OEIS, defined by the equation:

$s(n) = \frac{n}{2} \cdot (2 \cdot a(n-1) + (n-1) \cdot a(n-2))$

This allowed me to calculate that there would be 1,371,785,398,200 possibilities for a 10x10 grid using my method C which is 500 million times fewer possibilities than the 535,983,370,403,809,656,832 possibilities using the completely random method I described above. This is still a large number, but this could be run in days rather than years!

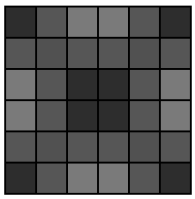
Implementing Method C:

I wrote a python module that generates potential solutions using Method C and then runs them through a checking algorithm. I looped through this code trillions of times and printed out all of the working solutions that had no three in any line. I did this for multiple grid sizes. I then ran another program that removed duplicate solutions so we are only left with unique solutions. Examples of four solutions that were found for a $4x4$ board are shown on the right:

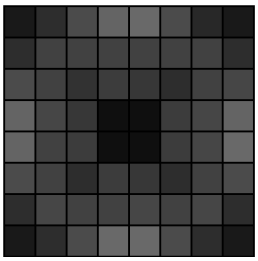


As the boards got larger, it took longer and longer to find working solutions. It took running my computer several days to find all of the solutions for a $10x10$ board, which is where the utility of my algorithm seemed to max out. This was still millions of times faster than the completely random method.

While looking at solutions that I generated, I made an interesting unexpected observation that it seemed that dots were generally placed on the boards in a circular shape for solutions with no 3 in a line. It was uncommon for dots to be placed at the center or corners of the grid. It seemed that designs often contained more dots in the areas shaded red in the figure to the left, while the yellow spots often remained empty.



In order to test this more objectively, I wrote a program to create a heat map by feeding in all of the unique solutions (including reflections and rotations) and adding up the number of dots in each location. The locations that more often had dots would be shaded a lighter color. For $6x6$ boards this resulted in the heatmap to the right. This is really interesting because, as I previously predicted, it appears that the center and corners of the grid rarely contain dots.



On the left is the heat map for $8x8$ board which looks very similar to the $6x6$ board.

Conclusion:

Using brute force algorithms as a way to find solutions to problems can be effective, however there are ways to help make brute force methods more efficient. For the no-three-in-line problem, narrowing down the sample space in which I utilize brute force to find possible solutions allows for a faster computation time as there are less possibilities to check. I found that Method C was the most efficient of my three methods. Even though the solutions I found using Method C seemed random, when I analyzed them using heat maps, I was able to see how a circular region of the grid often contained more dots. As a next step, I would experiment with using these heat maps as a way to more efficiently find possible solutions. Instead of going row by row and randomly picking two dots, my next generation algorithm could pick dots according to the probabilities of the heat map (this would favor picking dots on the grid in which the heat map indicates they are more likely to be in a valid configuration). I intend to continue exploring this.