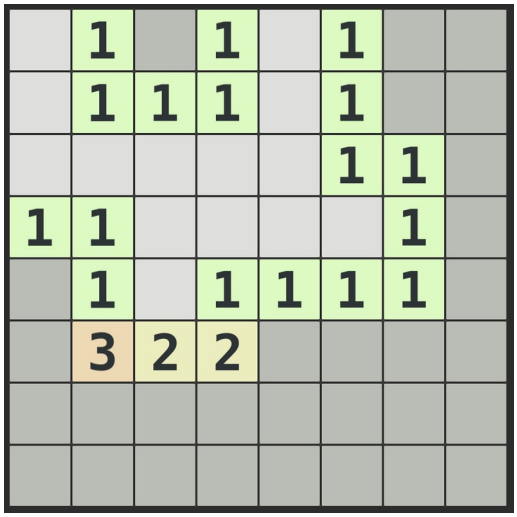
Minesetter

A Computational Intelligence project

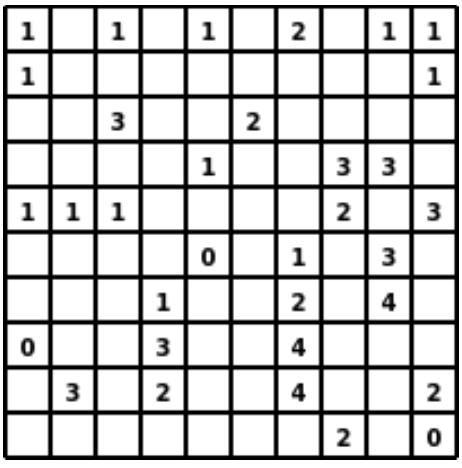
by Dimitrije Jovanović

Introduction

Minesweeper is one of the most well known video games in the world, by being present on many operating systems and online sites over the years as a free game. The concept is pretty straightforward, you have a grid of squares under which there are mines and your goal is to uncover all of them. By clicking on the starting square you are shown squares that are empty and squares that have numbers on them. The squares with numbers on them show how many mines are in their immediate vicinity.



This simple concept has a lot of potential about teaching us many different methods used in Computer Science for finding solutions to hard problems. In this paper we will be taking a look at a special version of Minesweeper which will be named as *Minesetter*. This problem is the opposite of Minesweeper, instead of finding mines on a board we are already given the numerated squares that gives us info how many mines should be around them which aids us in setting the mines. The priority is to cover as much of these fields as possible while at the same time using up as fewer mines as we can.



The idea behind this project came from a lecture done by professor Eryk Kopczyński from the University of Warsaw.[1] In it professor Kopczyński demonstrates how using many different optimization techniques we are able to solve the earlier mentioned Minesetter problem. Here we will try to replicate some of the approaches he mentions as well as approaches covered on a course in Computational Intelligence at the Faculty of Mathematics, University of Belgrade. The algorithms that were covered on that course were altered here to fit the nature of this problem.

Solving the problem

Before talking about the many optimization techniques that were used we should mention the elephant in the room. Optimization techniques give no guarantee that we will get an optimal solution. That is a by-product of relaying on randomness when trying to solve NP hard problems. The alternative is using brute force algorithms that have an exponential time complexity which becomes obsolete in instances where a human is able to solve the problem before the algorithm does. So instead of getting the optimal solution we have to be satisfied with a solution that is *almost* optimal. So just for completion sake there also is an implementation of a brute force algorithm.

Main approaches used for solving Minesetter are metaheuristics. In a general sense heuristics are techniques used for guiding and narrowing searches in problems in which a combinatorial explosion may occur. It has greek roots from the word “heurisko” which means “finding” or “finding out”.[2] Meanwhile *metaheuristics* are general frameworks to build heuristics for combinatorial and global optimization problems.[3] The main metaheuristics used will be the following: local search (invert variant of best improvement inverting), simulated annealing (SA), variable neighborhood search (VNS) and genetic algorithms.

Having said all that it’s time to explain how we will determinate our solution.

We have a matrix which consists of 2 possible elements: a ‘.’ element which represents a free square on the board and an element with a number like ‘1’ or ‘3’ which represent how many mines there are in the vicinity of that square.

We start off by initializing a dictionary. Traversing the matrix we check if element that’s in row i and column j is a free square i.e. ‘.’, and if it is we set the dictionary key as (i,j) and give it a boolean value. The chance of assigning True to any of our keys is 25% which allows us to not be overwhelmed with the amount of mines from the start but at same time to not have too few mines. The reason for using a dictionary is that it makes it much easier to keep track of what is a potential mine square and what is a number square which we can’t place mines on.

In order to determine how good our solution is we have to define a value for every board. This is done by calculating how many mines we were able to correctly place next to the squares with numbers. Say we have a square with the expected number of mines n and in actuality the number of mines is k. To check how far off we are from the the optimal number of mines expected we simply have to use the following formula $| n – k |$ for every number square. The aim is to have the value of our solution be equal to 0 (all mines have been placed accordingly) or at least is very close to 0 (again this is the nature of optimization).

Another thing to keep track of is the number of mines used. Given the nature of what we're doing there's no doubt that we will have some mines that are placed outside the range of numbered squares or are set in such a way that gives us more mines for our solution than needed. This in itself isn't too big of a problem but our goal is to place mines only around numbered squares and use as fewer mines as we can. So we will have to keep an eye out on the total number of mines we have on the field. Our main priority is still to have as much of the mines placed next to the expected areas but if we can still get the same value for our solution using less mines we will take the solution that was able to solve it with less mines.

Brute Force

The Brute Force approach is used only as a demonstration to show how ineffective it is for a board with bigger dimensions. We start with no mines on the board and then try to make every possible combination of mines that we can place.

Random Search

This is done by simply initializing the board over and over again by placing mines on the board at random.

Local search (invert best improvement variation) (LSIBI for short)

Inverting the best improvement of a local search algorithm works as inverting the boolean values in our dictionary to find what square is the best possible one to add/remove a mine by traversing each coordinate of the board. This is done by storing information regarding what pair of coordinates on the board gave us the best board value after inverting. After which we revert the boolean value on the square we visited back to normal. After iterating through the board we then invert the boolean value in our dictionary on the position of the best coordinate to invert because that is the best improvement we could find and then the process continues again until there is no more improvements to be made on the board.

Simulated annealing

We first initialize the solutions and find its value and then iterate the process of finding the best solution and best board value by a given amount of iterations. During the iteration process we choose a random board coordinate at random and calculate the new value we got from changing the boolean value on that part of the board. Here is where the beauty of the algorithm really starts to show. Simulated annealing (SA) uses a random search strategy, which not only accepts new positions that decrease the objective function (assuming a minimization problem), but also accepts positions that increase objective function values.[4] The reason this is done is that if the new board value isn't an improvement there still is a chance of it getting accepted by checking if

a random number we chose is smaller than the p(x) function which in our example is set to 1 / (i) where i is the current iteration we are in.

Variable Neighborhood Search (VNS)

VNS is a metaheuristic, or framework for building heuristics, which exploits systematically the idea of neighborhood change, both in the descent to local minima and in the escape from the valleys which contain them. If there are many local minima, the range of values they span may be large. There are, however, many ways to get out of local optima and, more precisely, the valleys which contain them. VNS exploits systematically the idea of neighbor-hood change, both in descent to local minima and in escape from the valleys which contain them.[3] We iterate various neighborhoods with a size of k in the interval of [k_min, k_max). Choosing k random coordinates changes the boolean values in them and evaluates the board value. Then we send our new board and its value to a LSIBI algorithm that additionally tries finding a better solution. This also comes with implementing the scenario where the new board value and best board value are equal where there is a 50% chance of taking the new board as a best one.

Genetic algorithms (GA)

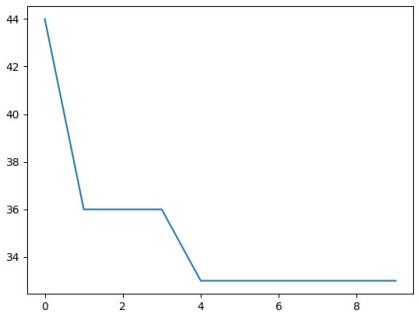
Genetic algorithms are implemented as a computational simulation of a population of individuals where one individual represents one potential solution for our problem. The goal is to find a value that our function reaches as its optimal or near optimal solution.[2] The GA approach was made by making our individuals a single board. The fitness function is the same as the function to determine our board value. Tournament selection was used in order to choose the parents and the crossover function was implemented by choosing a random coordinate (i,j) and giving the first child all the elements that belong to the first parent from coordinate (1,1) until coordinate (i,j) while the rest comes from the second parent. For the second child we do the same thing but the second parent goes first. Mutation function is fairly simple, we change the boolean value with the probability of 5%.

Experimental results

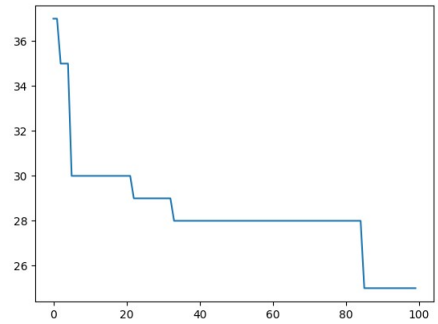
Now we will run our algorithms and comparing how they did between each other. All of the mentioned algorithms will be using the 10x10 board example found in the Introduction section.

Random search

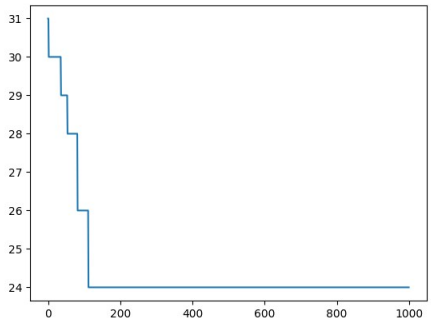
Using Random search we were able to get a value as low as 13 and is among the expected result.[1] This might sound great at first but we needed 100000 and 1000000 iterations to get here. Also we weren't even changing any solution we were just erasing the entire board even tho changing values on some boards could've helped us go down a better solution. Below are the results after 10, 100, 1000, 10000, 100000 and 1000000 iterations. It took around 110 seconds to get the solution with 1000000 iterations.



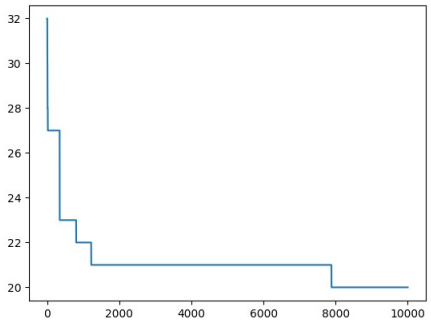
10 iters.



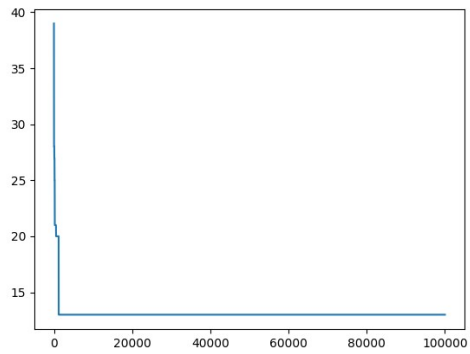
100 iters.

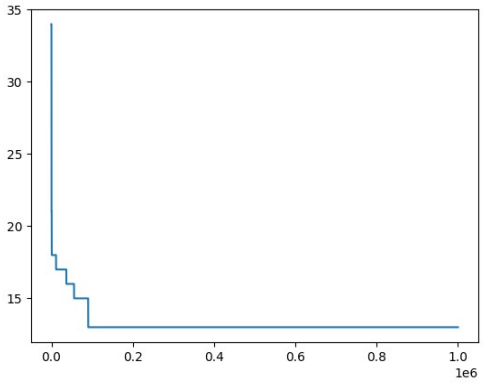


1000 iters.



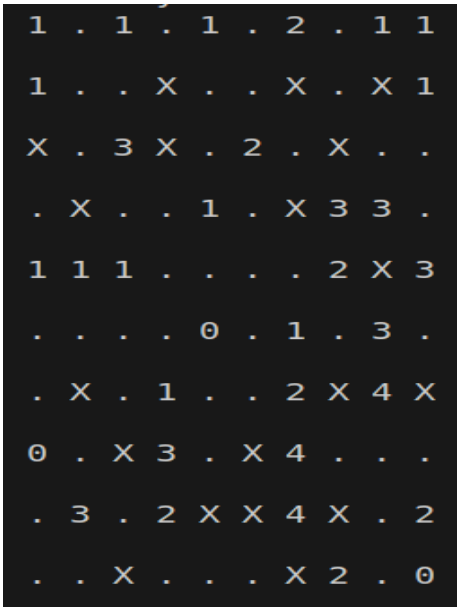
10000 iters.





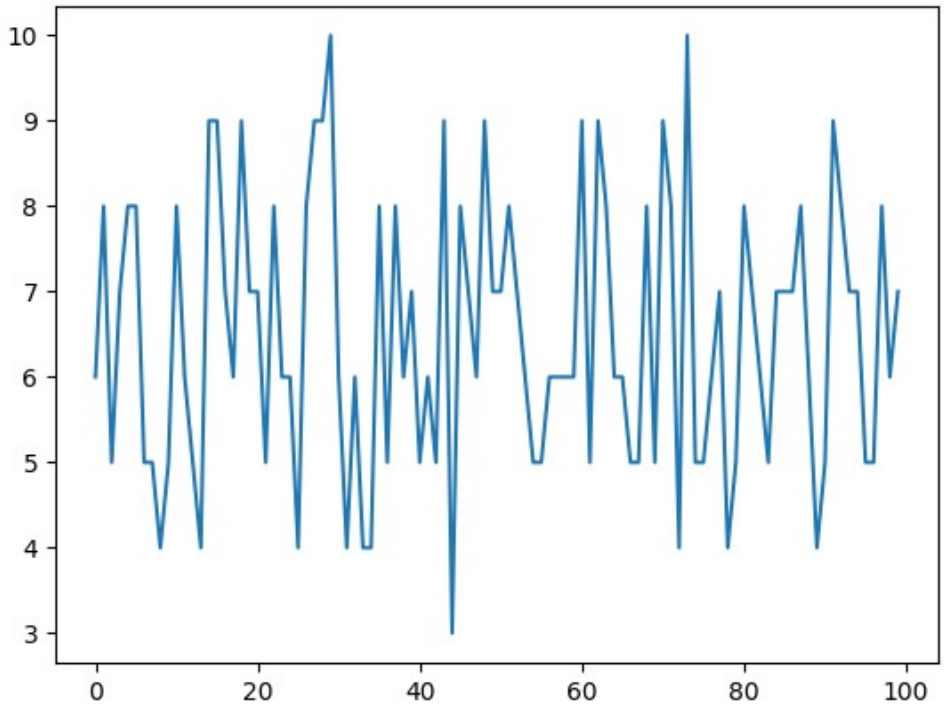
100000 iters. 1000000 iters.

As we can see we don’t have a smart way of finding an optimal or near optimal solutions. We’re just randomly generating boards and hoping we get closer and closer . The picture below is the board made after 1 million iterations.



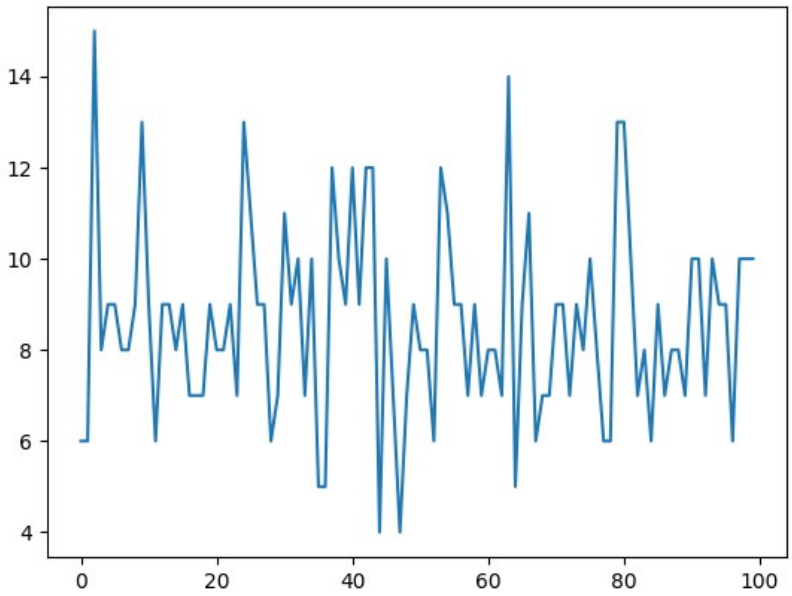
Local Search Invert Best Improvement

Using LSIBI we were able to lower the desired value down to 3. This is a graph which shows the end result. Here we see that solutions were able to take values between 3 and 10. This is a graph that shows the progress during 100 iterations where a single instance took roughly a 10^{th} of a second to solve.



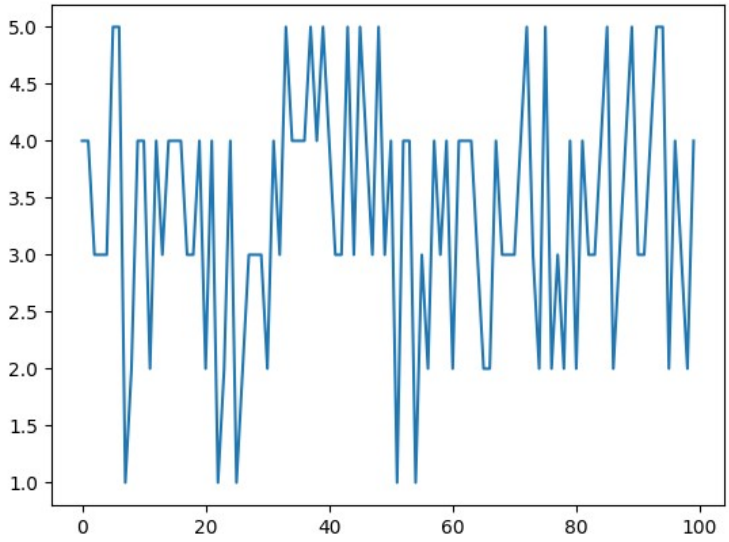
Simulated Annealing

SA results show a somewhat worse performance than local search. While there are instances where SA did achieve better results (up to value 1 in some cases[1]) this wasn't the case when testing as even when setting the number of iterations to 100000 for each SA instance. This graph shows the values SA was able to get when running 100 iterations. The best value it was able to achieve was 4. It takes roughly 20 seconds for one instance to execute.

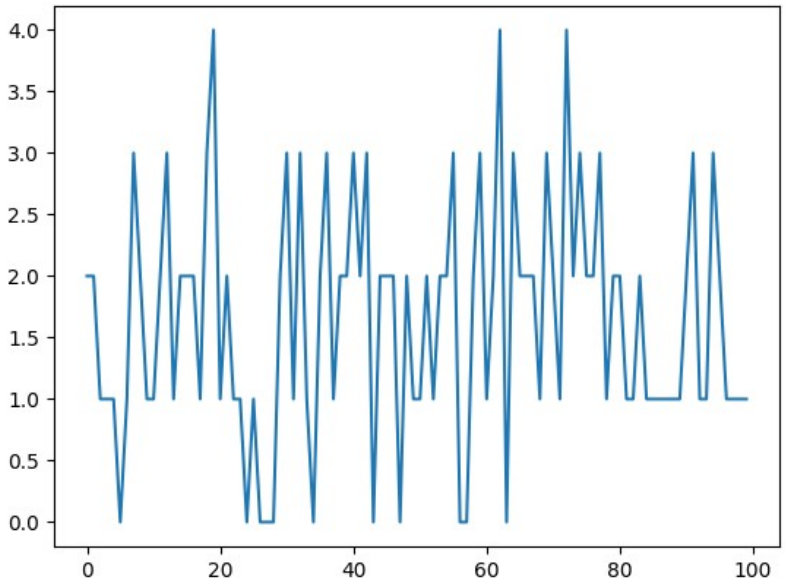


VNS

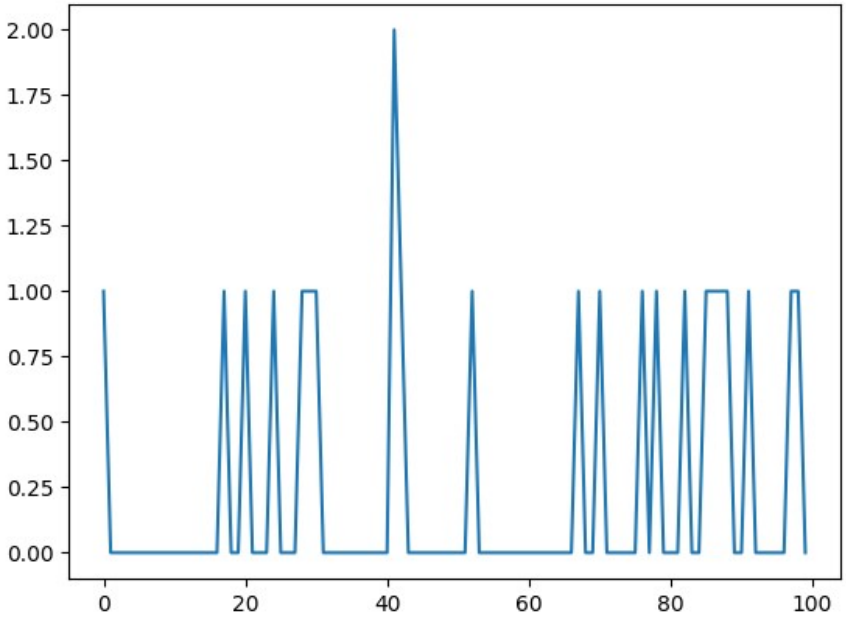
So far VNS has shown the best results compared to other algorithms. However when choosing how many neighbors we should put when assigning both k_min and k_max values we should keep in mind that it doesn't exceed the total number of squares on which we can place mines. That aside we can see that just with setting k_min = 1 and k_max = 2 with 100 iterations we get really good results. Time needed for one iteration was around 1.4 seconds. (Note : This means the for loop in the VNS algorithm is making 100 iterations, while the plots represent running the VNS algorithm 100 times)



For the first time we were able to reach a board value of 1. If we tune the k_min to 1 and k_max to 10 we will get even better results but with around 4 seconds for each iteration.



For the first time in this example we have a board value of 0. Meaning that we finally found an optimal solution. Twerking around with the parameters even more we can make it even more consistent on getting the optimal value like for example setting k_min = 5 and k_max = 10. However it is important to note this took roughly 10 seconds to find.



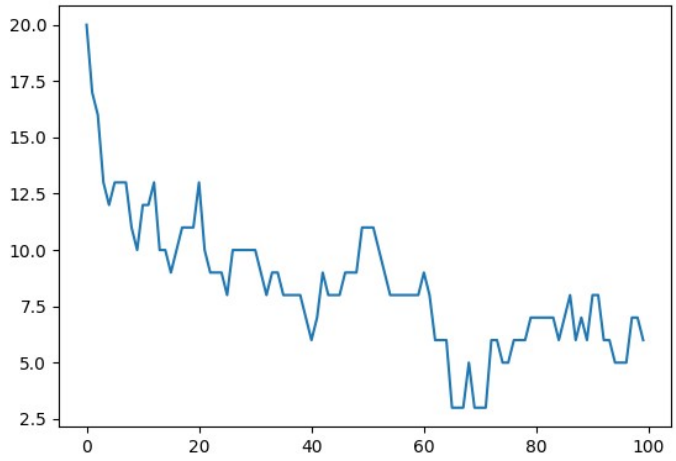
This is what the board is suppose to look like when we find an optimal solution.

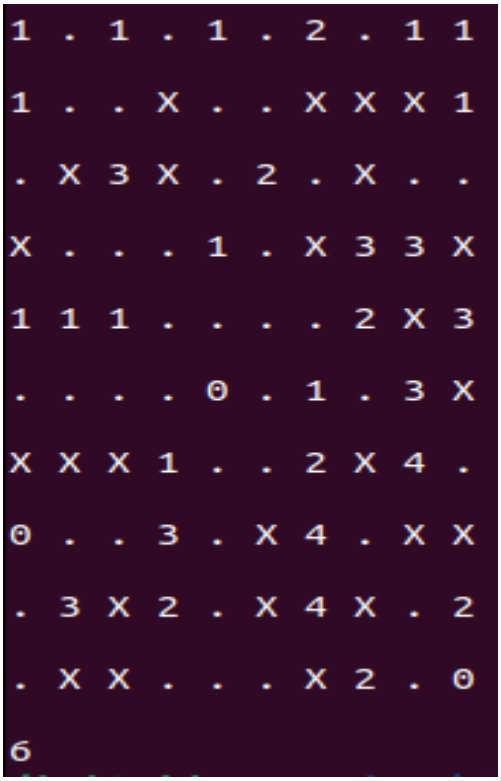
```
1 . 1 . 1 X 2 . 1 1
1 X . . . . X . X 1
. . 3 X . 2 . X . .
. X . . 1 . X 3 3 X
1 1 1 . . . . 2 X 3
. . . . 0 . 1 . 3 X
. . X 1 . . 2 X 4 .
0 . . 3 . X 4 . X X
. 3 X 2 X X 4 X . 2
X X . . . . X 2 . 0
0
```

GA

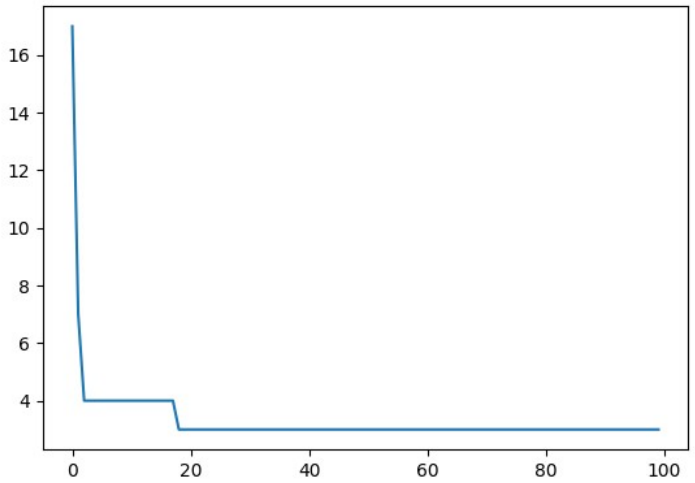
Genetic algorithms have been implemented by changing the population size. We start off with a population of 100 and progressing to 1000 and 2000. All the time we are setting the elitism size to be equal to 30% of the population while the tournament size is set to 70% of the population while keeping the mutation probability at 5% and going through 100 generations.

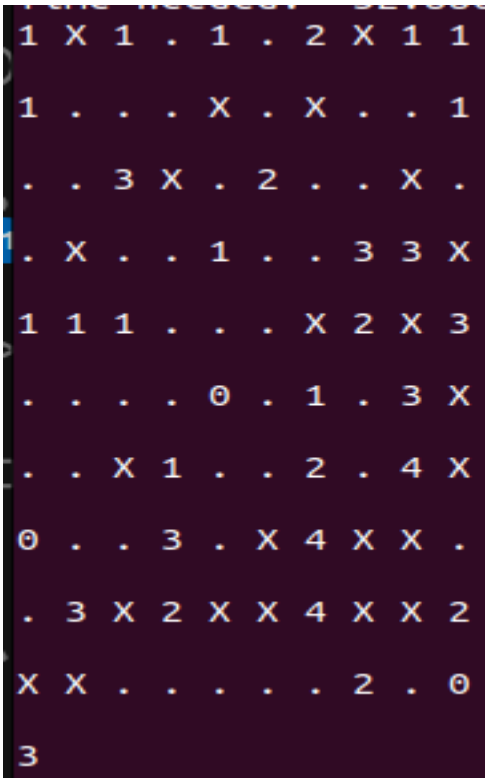
This is how it looks like with having a population of 100. It took around 1 second to execute. The board value is 6.



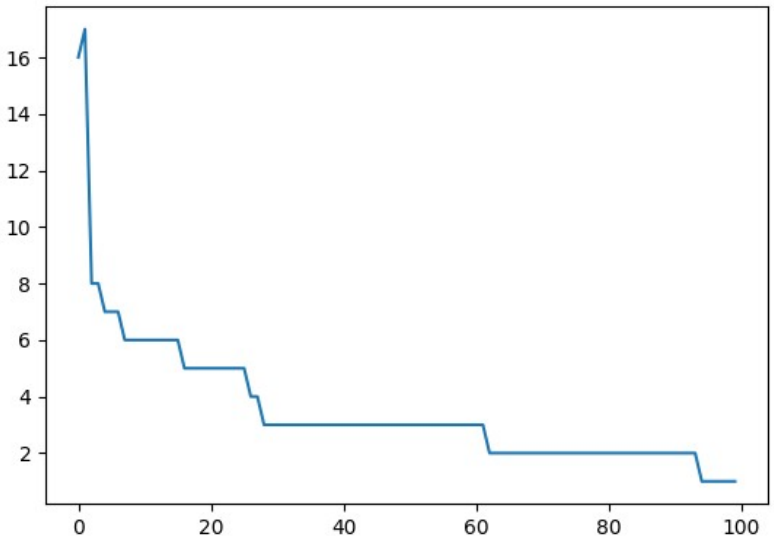


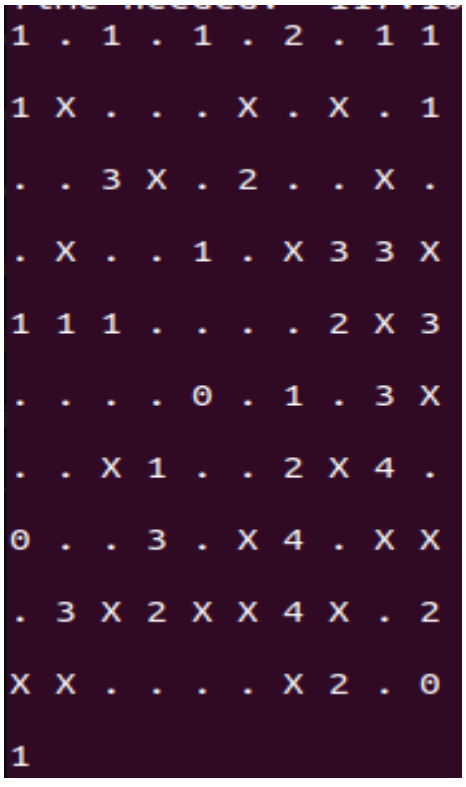
This is how it looks like with having a population of 1000. It took around 30 seconds and the board value is 3.





This is how it looks like with a population of 2000. It took 117 seconds and has a board value of 1. There still might be a chance to hit a board value of 0[1] but it's not as consistent as other methods. Meaning it is nothing more advance than the previous instance even tho we have over twice as many individuals in a population and took nearly 4 times longer to run.





Trying a population that’s anything larger than this would be unnecessary given how it took almost 2 minutes to find a near optimal solution when we already have certain techniques that gives an optimal solution most of the time in 1/10 of time needed here.

Brute Force

On small boards that are dimensions 4x4 we can find the solution almost instantly without any major obstacles. However solving a board that is a size of 10x10 (like the one we used in our example) takes considerably longer. Instead of measuring it in seconds the time needed for solving this problem could take up a couple of hours in order to solve, making it practically worthless. I tried running a brute force algorithm on this example and after **5 hours** I still didn’t have a solution. Of course this is to be expected when handling an algorithm that has an exponential time complexity.

Software and Hardware info

All of the algorithms used were run on a HP Laptop 15s. Memory: 16 GB, Processor: AMD® Ryzen 7 5700u with radeon graphics × 16, Graphics: RENOIR (renoir, LLVM 15.0.7, DRM 3.54, 6.5.0-17-generic), Disk Capacity: 512 GB, Operating System: Ubuntu 22.04.3 LTS, OS Type: 64 bit.

Conclusion

Optimization techniques are an incredible tool for solving NP hard problems whenever it is possible to accept a near optimal solution if the optimal one can’t be reached. However it is important to note that using different techniques will give out a different degree of accuracy. Random search is probably the worst algorithm because it is just giving out a random solution. Simulated annealing comes in as second worse due to the number of iterations needed to achieve a slightly better solution than that of RS when there are much faster and better alternatives. Genetic Algorithm is still a good algorithm given that it needs around a second to find a solution with some small error but if we want to optimize the algorithm even further it might not be the best case to use given how it simply may yield better results than SA for problems where *crossing-over* works well (i.e., solutions that are 'good' in different areas can be successfully combined into a solution that is good everywhere), but in general, they are hard to do right.[1] Local Search Invert Best Improvement is very fast and can give out a very fast near optimal solution in around a tenth of a second. If we are okay with having some minor errors on bigger boards then this might not be a problem but in case of getting nearer to an optimal solution we don’t have many room for improvement. Definitely the best optimization technique used was the VNS algorithm. Even if it doesn’t have the speed that LSIBI has it is still the only algorithm with which we were able to even reach the optimal solution more regularly than we would with using a Genetic Algorithm.

That being said there still is room for improvement. While it didn’t get the desired results perhaps changing the structure of the Simulated Annealing algorithm might improve it. Changing some parameters of the Genetic Algorithm we might get it to have a more consistent optimal solution and potentially even achieve it in a shorter time. When it comes to VNS more testing could be done to make sure that we get an even more consistent amount of optimal solutions, either by changing the number of neighbors or by changing the the number of iterations we are going through. There also exists the possibility of using a SAT solver but that goes beyond the scope of the course this project was made for.

References

[1] https://www.mimuw.edu.pl/~erykk/algods/lecture11.html

[2] *Veštačka inteligencija – Predrag Janičić, Mladen Nikolić – Matematički fakultet 2023*

[3] *Search Methodologies (Introductory Tutorials in Optimization and Decision Support Techniques) - Edmund K. Burke, Graham Kendall – Springer*

[4] *Computational Intelligence An Introduction - Andries P. Engelbrecht – University of Pretoria*