AI Project - Hidato

A Tale of Love & Numbers

Authors:

Peleg Zour
Dor Dahuki
Saef Aliyan
Hala Omari

Made for 67842: Introduction To Artificial Intelligence at HUJI, 2021

41	43	44		49	52	51	
42	40	45	48	63	50	53	55
39	38	46	47	22	62		
37	36	26	23	21	61	17	
35	27	24	25	20	18	60	16
33	34	28	5	19		10	15
32	29	3	4	6	9	14	11
30	31	2	1	7	8	13	12

Introduction

Hidato (from the Hebrew word “Hida”-”חידה” which means “riddle”) is a logic puzzle game invented by the Israeli mathematician Dr. Gyora M. Benedek.
The starting point of the game is a grid board partially filled with numbers; The goal of the game is to fill the empty grid cells with the missing numbers from 1 to the size of the board such that each two consecutive numbers are in adjacent cells on the board, creating a “path” from 1 to the maximal number on the board.

Complexity of Hidato

Hidato is an NP-Complete problem[1], meaning that finding a perfect solution in the worst case can take a number of steps that is exponential in the size of the board.

That means that solving a 10X10 Hidato might take up to 100^{100}, more than the number of atoms in the universe!

Our Approach

The naive approach to solving Hidato is using brute-force backtracking, trying all possible options of filling the board; As shown above, this is computationally-infeasible even for a relatively small Hidato board.
In order to solve the problem we decided to model the problem in two different ways:
As a Constraint Satisfaction Problem (CSP), and as a Local Search Problem.

As a CSP

A Constraint Satisfaction Problem (CSP) model consists of three parts:

1. Variables $x_1, x_2, \ldots, x_n$
2. Domains $D_1, D_2, \ldots, D_n$ where D_i is a set of possible assignments to x_i
3. Constraints $C_1, C_2, \ldots, C_k$ where each C_i is a pair; The first member of the pair is a subset $x_{i_1}, x_{i_2}, \ldots, x_{i_r}$ of the variables; The second member of the pair is a subset of $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_r}$ for a subset $D_{i_1}, D_{i_2}, \ldots, D_{i_r}$ of the domains.

A complete assignment to a CSP is an assignment which gives a value to each variable;
A consistent assignment to a CSP is an assignment which does not violate any constraint.
A solution to a CSP is an assignment which is both complete and consistent.

Modeling Hidato as a CSP sounds somewhat intuitive, but there is a challenge in defining which parts of the CSP correspond to which parts of Hidato; After several attempts, we ended up with the following model:

1. $x_1, \ldots, x_t$ are variables such that t is the size of the board.

[1] http://www.nearly42.org/cstheory/hidato-is-np-complete/

2. The domain D_i is the set of possible locations for the number i on the board, i.e all empty cells in the initial board.
3. For each fixed number (i.e a number given in the initial board) the constraint C_i is the singleton containing its location on the board.
   For each pair of numbers i, j the constraint $C_{i,j}$ is the set of possible combinations of locations that can be assigned to i and j.

In practice, the domains and constraints are updated according to the current state of the board.
The domain D_i is defined as the set of all legal locations for i on the board, and is updated with each assignment change.
For a number i, after i is assigned its domain is updated to be the singleton set containing its assigned location.
For two numbers i, j:

- if i, j are not consecutive the constraint $C_{i,j}$ is the set of all combinations of currently empty locations on the board.
- if i, j are consecutive, then if WLOG i is on the board then the constraint $C_{i,j}$ is the set of all empty locations directly adjacent to the location of i.
  if i, j are not on the board then the constraint $C_{i,j}$ is the set of all combinations of currently empty locations on the board.

Heuristics used to optimize backtracking

As described above, Backtracking is the basic algorithm for solving CSP, but its computational complexity is not ideal.
By using heuristics we can cut down the number of backtracking steps.
We have used the following heuristics:

- Minimum Remaining Values (MRV): The next number placed on the board will be the number which has the minimal number of legal locations currently left empty on the board.

- Least Constraining Value: Given a number, the next location we will try to place it in is the location which rules out the fewest locations for other numbers, i.e the location which currently appears in the least domains.

- Forward Checking / Arc-Consistency: After each assignment, keep track of the remaining locations for the other numbers; if any number is left with no legal locations, terminate and backtrack.

  Arc-Consistency goes even further: we represent the problem as a complete graph, where each vertex represents a number; The edge (i, j) is called consistent if for every possible remaining location for i there is a legal location for j.
  The idea is to make sure all edges are always consistent; If (i, j) is inconsistent, i.e there is a location x for i such that it leaves no legal location for j then we remove x from the set of possible locations for i, and recheck consistency of all of i's neighbors; if there are no more possible locations for i we terminate.

For Arc-Consistency we used the AC-3 algorithm[2].

As a Search Problem

The second approach we have decided to try was modeling the problem as a local search problem.
Local search algorithms are a heuristic method of solving computationally hard optimization problems. The search space is made of all possible problem states; Each state is given a score according to the criterion optimized and the search is performed by moving from state to state by applying local changes to the current state until a good enough solution is found or some time / steps limit is reached.
In our model, each state is defined by the content of the board and scored by counting **how many consecutive numbers are not in adjacent locations on the board** (From now on referred to as the loss function).
For solving the local search problem we used two variants of the hill-climbing algorithm:

Note: a neighbor state is generated by swapping two numbers on the board.

The first variant is first-choice hill-climbing with random restart:

1. Initialization: Start at a random state; Save it as the best state seen.
2. Generate neighbor states of the current state in some arbitrary order.
3. If a neighbor state has a higher score than the current state, move to the neighbor state and go back to step 2; If the neighbor state is better than the best state seen - save it as the best state seen.
4. If none of the neighboring states are better, move to a random state and go back to step 2. if the random state is better than the best state seen - save it as the best state seen.
5. Termination: after running for the given max steps return the best state seen.

The second variant is random-choice hill climbing:

1. Initialization: Start at a random state.
2. Move to a random neighbor state; repeat while steps made < max steps.
3. Termination: return last achieved state.

Both algorithms are run for a specified maximal number of iterations.

Hill-climbing algorithms usually fail to find an optimal solution, as they can get “stuck” on local maxima / minima; This problem can be addressed by random-restarts, or, put simply, moving to a random state if no better neighbor state is found.

Furthermore, finding the best neighbor of a current state is computationally hard;
Given a board with size n there are $\binom{n}{2}$ possible swaps from each state, which is $O(n^2)$ possible swaps.
In order to avoid computing all possible neighbor states at each move we chose the first better neighbor, thus ensuring an uphill ascent at each step with the trade-off of making sub-optimal steps.

---

[2] https://en.wikipedia.org/wiki/AC-3_algorithm

Results

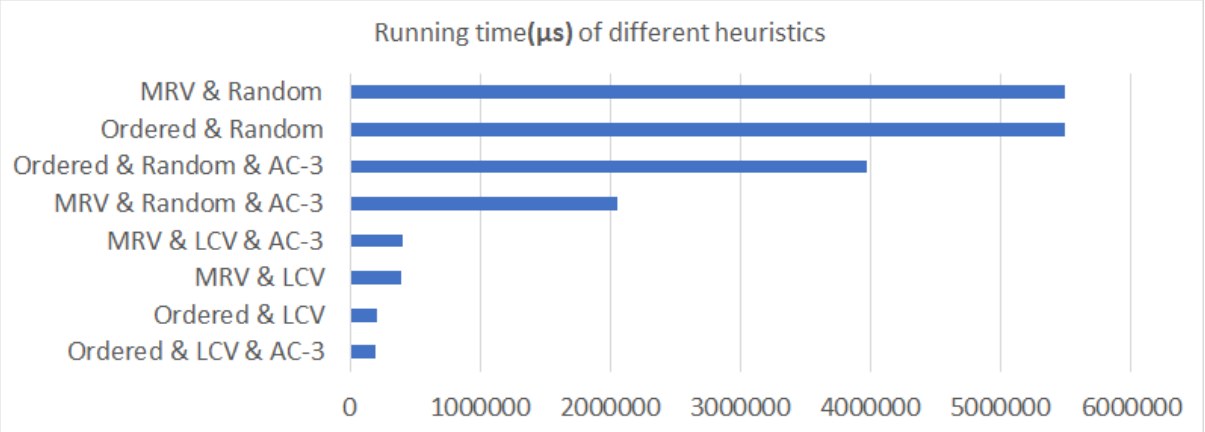
Comparing CSP solvers to Local Search solvers is quite difficult.
While the CSP solver is complete, meaning it always finds an optimal solution, the Local Search algorithms usually only find an approximate sub-optimal solution.
Therefore, comparison between them by any metric is quite meaningless, as they achieve different goals.
Following the above conclusion we decided to measure the performance of the two different approaches using different metrics.
All following measurements were made on an 8X8 Hidato board with 32 numbers ($\frac{1}{2}$ of the board) missing.

Constraint Satisfaction Problem Solver

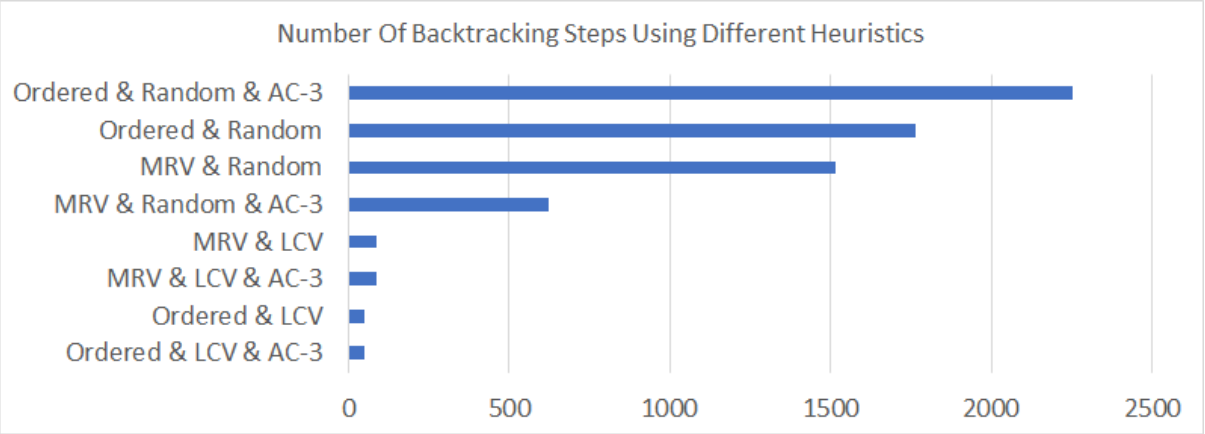
CSP Solvers always find an optimal solution, However the runtime might be exponential;
As our goal was to cut backtracking runtime, we chose to measure CSP Solver performance based on runtime / number of backtracking nodes visited.

Abbreviations used for the different heuristics:

- AC-3 - Arc Consistency
- Ordered - Select numbers from smallest to largest
- Random - Select locations on the board in random order.
- MRV - Minimum Remaining Values
- LCV - Least Constraining Value



Runtimes as recorded on a computer using Intel® i7 10th gen processor, with 16GB RAM



The following tables show for each heuristic a comparison of results for all combinations of other heuristics, with or without it; For each column the better heuristic combination is colored green, the worst is colored red and equal results are colored yellow.

Runtimes (µs) with / without MRV:

	Random	**Random & AC-3**	**LCV**	**LCV & AC-3**
MRV	5,493,935	2,050,590	398,589	407,108
Ordered	5,492,267	3,967,804	206,223	199,984

Backtracking steps with / without MRV:

	Random	**Random & AC-3**	**LCV**	**LCV & AC-3**
MRV	1,517	625	89	89
Ordered	1,763	2,254	49	49

Runtimes (µs) with / without LCV:

	MRV	**MRV & AC-3**	**Ordered**	**Ordered & AC-3**
LCV	398,589	407,108	206,223	199,984
Random	5,493,935	2,050,590	5,492,267	3,967,804

Backtracking steps with / without LCV:

	MRV	**MRV & AC-3**	**Ordered**	**Ordered & AC-3**
LCV	89	89	49	49
Random	1,517	625	1,763	2,254

Runtimes (µs) with / without AC-3:

	Ordered & Random	**MRV & Random**	**Ordered & LCV**	**MRV & LCV**
With AC-3	3,967,804	2,050,590	199,984	407,108
Without AC-3	5,492,267	5,493,935	206,223	398,589

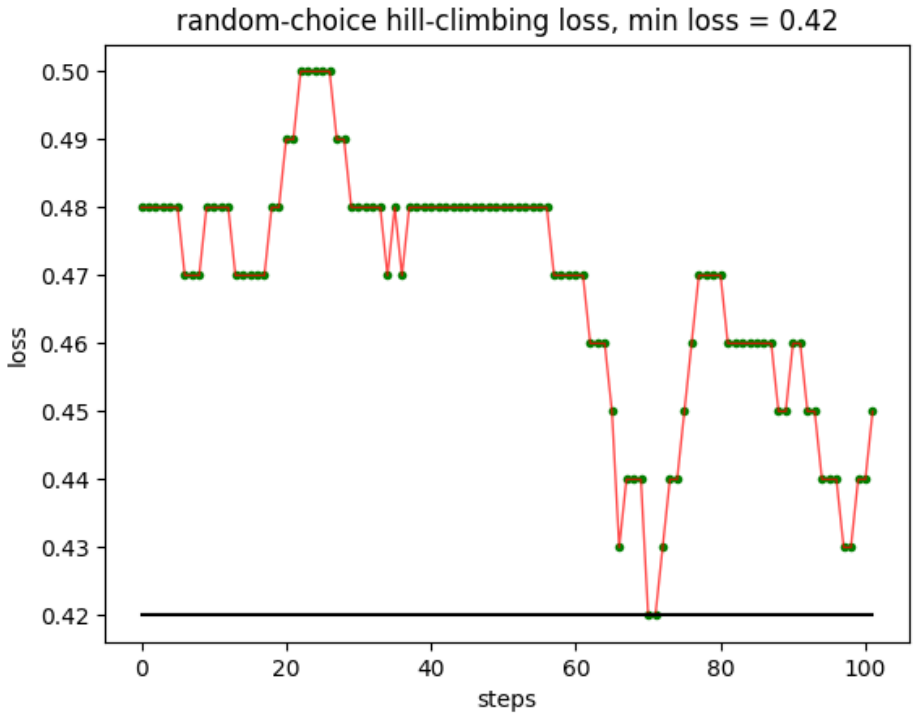
Backtracking steps with / without AC-3:

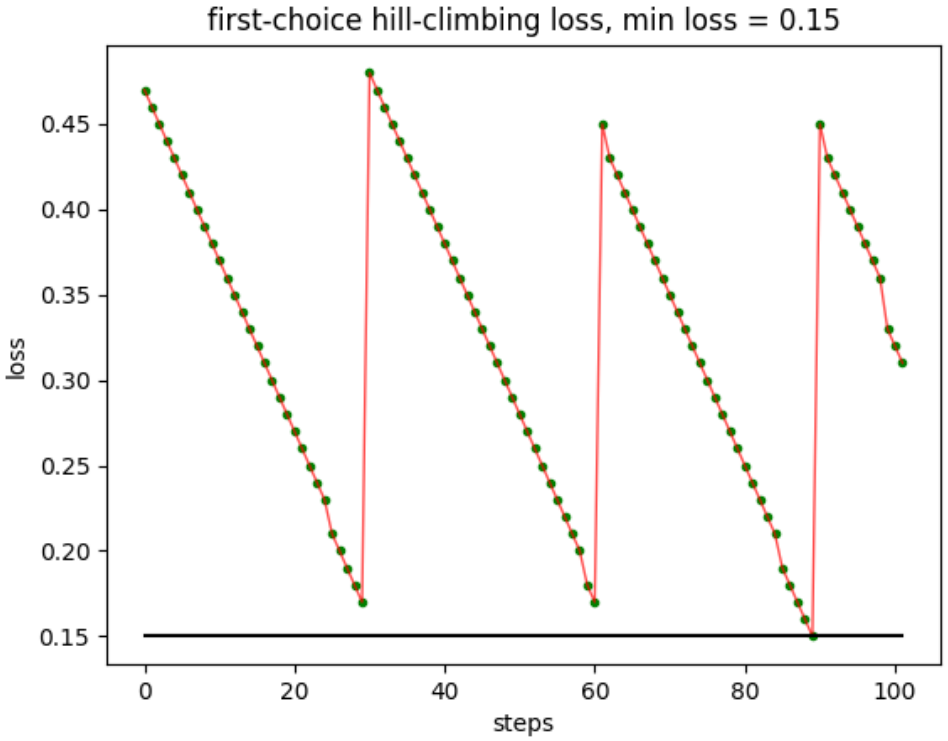
	Ordered & Random	**MRV & Random**	**Ordered & LCV**	**MRV & LCV**
With AC-3	2,254	625	49	89
Without AC-3	1,763	1,517	49	89

Hill-Climbing search

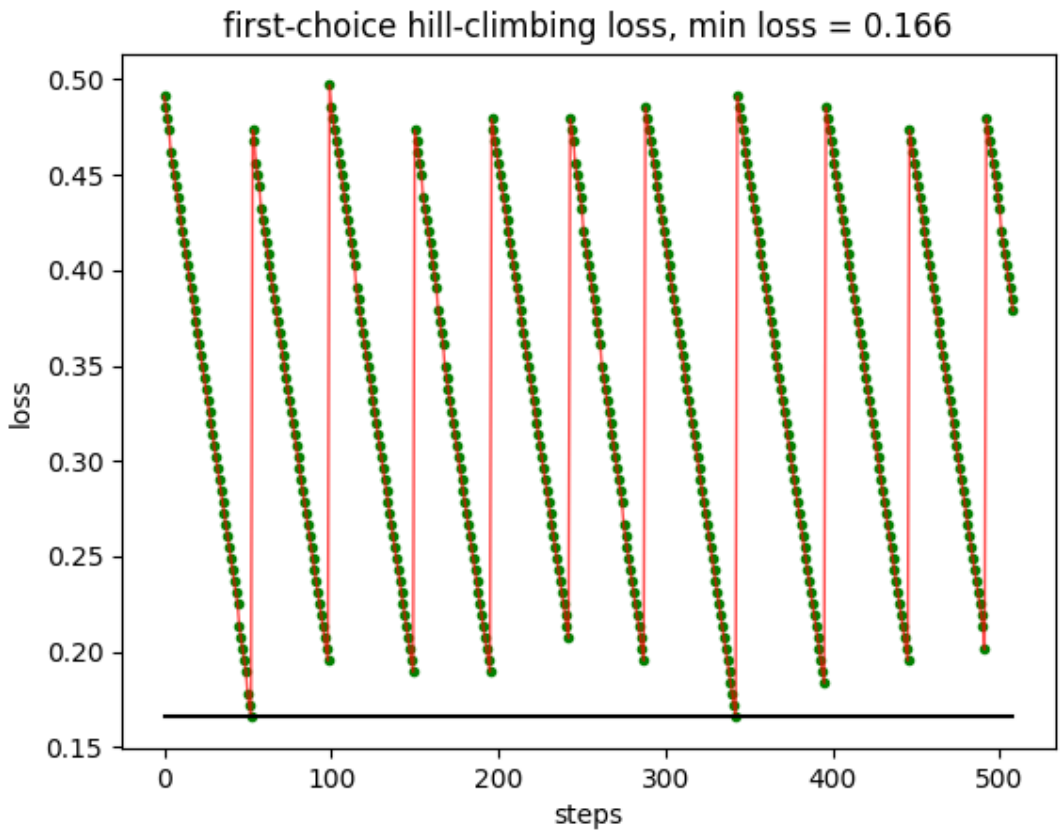
Unlike the CSP solver, the Hill-Climbing search usually does not find a complete solution. Therefore we decided to simply plot the loss at each step, and compare the loss progression of the two different Hill-Climbing algorithms.

Note: #_errors is the number of unfixed numbers on the board not adjacent to their consecutive or previous number; Loss is measured as $\frac{\#\_errors}{size\ of\ board}$.





In addition to the above, we also ran the Hill-Climber on a 13x13 board, with the following results:



Conclusions

The goal of our project was to find an efficient algorithm to solve the Hidato problem.
We implemented two different algorithmic approaches, each with different strengths and weaknesses.

Constraint Satisfaction Problem

By looking at the result charts it is clear that the **best** heuristics combination by both metrics is **ordered variables & LCV & AC-3**; The **worst** heuristics combination differs between the two metrics, with **MRV & Random values** being the worst in runtime and **Ordered variables & Random values & AC-3** being the worst in backtracking steps.

A few notable points:

- It is worth noting that Random & Ordered, i.e the “vanilla” backtracking algorithm did quite well on the 8x8 board, being a viable solution for solving Hidato, obviously much faster than a human could solve it.

- The heuristic combinations including LCV always outperformed the ones that chose values in random order, both in terms of backtracking steps and running time.

- Ordered & Random & AC-3 performed worse than Ordered & Random on backtracking steps; Assuming our implementation of the AC-3 algorithm is correct, as AC-3 will never add backtracking steps but rather, at the very least, not make a difference - the most probable cause for this result is the random order itself, which is different between each heuristic trial run.

- The combination of Ordered values and LCV was by far the best, outperforming MRV & LCV and MRV & Random value.
  This might be due to the nature of the Hidato game itself - variable order is crucial in Hidato, and it seems that the “natural” way of placing numbers on the board in increasing order is indeed the best.
  However, it is not clear then why the combination of MRV & Random outperforms MRV & LCV; This is something we could not find an explanation for.

- Somewhat surprisingly, AC-3 did quite well on runtimes, even with the additional computations involved, improving the runtimes of all heuristic combinations except MRV & LCV.
  In contrast, the results of AC-3 on backtracking steps are inconclusive; When value order was not random, i.e values were chosen in the same order with and without AC-3, the number of backtracking steps did not change after applying AC-3;
  While when choosing values in a random order, the results are contradictory - with AC-3 improving the results of MRV & Random and worsening the results of Ordered & Random; This as well is probably due to the fact that these tests were each run with a different random order of variables, as mentioned above.
  A conclusion of the above might be that AC-3 is unnecessary when using LCV together with MRV.

- MRV runtimes are also inconclusive; The results with and without MRV on Random and on Random & AC-3 are again contradictory, probably due to the above

mentioned problem with random order being inconsistent between runs.
In contrast, on backtracking the results are more conclusive, showing a minor improvement on Random variables and a major improvement on Random & AC-3. The results on LCV and LCV & AC-3 are conclusive and show that MRV worsened both runtimes and backtracking steps.

Search Problem

The Hill-Climbing algorithm we implemented did quite well on the Hidato problem, getting a very low error rate in somewhat constant time; In comparison, the random-choice climber algorithm performed terribly (as expected), not gaining any significant improvement over its starting loss.

Looking at the included loss chart for our Hill-Climber, it appears that the surface of our chosen loss function has many local minima, making random restart a smart choice; And indeed, the chart shows that without random restart we would have gotten a worse loss.
It is also worth noting that most steps improved loss by 1, with a minor portion of the steps improving it by more than 1 (mostly by 2); This might imply that Best-Choice rather than First-Choice was the better option for our Hill-Climbing algorithm; However when we tried implementing Best-Choice runtimes suffered greatly, with runs not terminating after a very long time, while First-Choice would terminate in mere seconds.
Indeed it seems That the gains in efficiency from using First-Choice far outweigh the gains in descent rate we would have gotten by using Best-Choice, as was our hypothesis.

Looking at the loss chart for our 13x13 board run, it appears that the number of steps taken has no clear correlation to loss gained - the best loss was achieved before the first (out of 10) random restart; It is true however that making more steps increases the number of local minima found, thus increasing the probability of finding a better minima.

CSP vs. Search Problem: Final Conclusions

When running the algorithms on 8x8 Hidato boards with $\frac{1}{2}$ of the numbers missing, it might appear that using the CSP solver is the obvious choice, and in fact it is not clear why we would need a Hill-Climber algorithm which does not even achieve a correct solution.

However, the differences appear when changing the size of the board, or when increasing the rate of missing numbers.
On larger boards, runtimes for the CSP solver increase drastically, with runtimes for a 13x13 board being over 1 hour; In contrast, the run of the Hill-Climber on a 13x13 board took a mere 25 minutes, achieving very low loss.
The advantage of Hill-Climbing becomes even more pronounced when increasing the rate of missing numbers on the board; On an 8x8 board with a rate of 0.7 missing numbers our CSP solver did not terminate after 1 hour; The Hill-Climber on the other hand terminated its run within 43 seconds, achieving a loss of ~0.14.

The main difference lies in complexity - while the worst case runtime of the CSP solver is exponential in the size of the board, the maximal number of steps the Hill-Climber can take is linear[3].

In conclusion, the CSP solver and Hill-Climber serve somewhat different purposes.
While the CSP solver will always find a perfect solution, it can be extremely inefficient for large boards with many missing numbers.
On the other hand, the Hill-Climber, while not finding a perfect solution, does find a good approximate solution while running quite fast, especially with respect to the runtimes of the CSP solver; This makes Hill-Climbing the preferred method in situations where a perfect solution is not strictly required.

[3] max steps is computed as board_size * missing_number_rate * constant

Running the code

In order to run the program, you will need to compile the generator which is written in C++ using the Makefile by running the command $make$ in the command line.

Then, use the following command to solve a random Hidato problem:
$python3\ main.py\ \{--\ csp\ |\ --\ hill\}\ [--\ dim\ value]\ [--\ a\ value]\ [--\ gui]\ [--\ benchmark]$

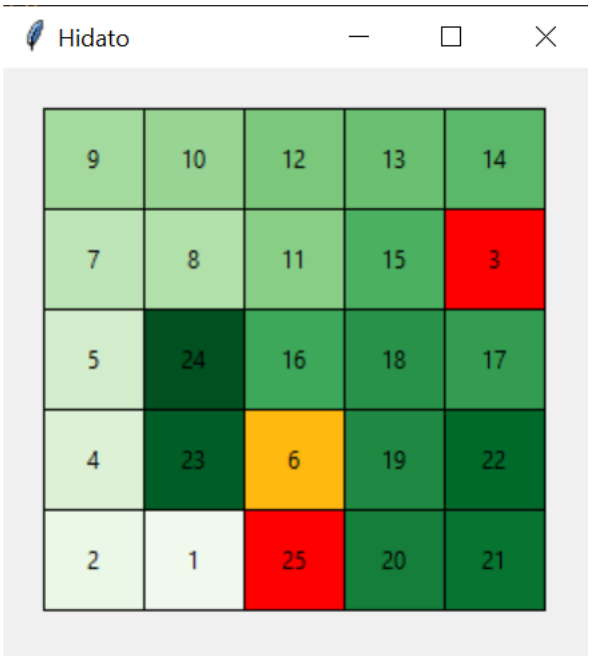
where:

- $--\ csp$ Solve the problem using the CSP solver.
- $--\ hill$ Solve the problem using Hill-Climbing.
- $--\ dim$ An integer; Specifies the width and the height of the board that will be generated (boards are strictly square for simplification).
  Default value is 5.
- $--\ a$ A number between 0 and 1; Specifies the rate of missing numbers in the board.
  This parameter is used to set the difficulty of the problem.
  Default value is 0.5.
- $--\ gui$ Shows a GUI animation of the solution steps (recommended).
- $--\ benchmark$ Run a performance test of the chosen model, and output the results to a CSV file.

Examples:

- $python3\ main.py\ --\ csp\ --\ dim\ 8$
  Solve an 8x8 board using the CSP model.
- $python3\ main.py\ --\ csp\ --\ dim\ 8\ --\ benchmark$
  Test performance of the CSP model on an 8x8 board and output results to CSV.
- $python3\ main.py\ -\ hill\ --\ dim\ 8\ --\ gui$
  Solve an 8x8 board using Hill-Climbing and Display GUI animation at the end.

About the GUI:



The meaning of the different colors is as follows:

- Green: a correctly placed number. i.e a number which is adjacent to its previous number and its consecutive number, if they are on the board.
- Red: An error.
- Gold: Cells will flash gold before being swapped.

Appendix

Project Github repository: https://github.com/dor-d/Hidato
The Wikipedia page for Hidato: https://en.wikipedia.org/wiki/Hidato