# NEA

My programming project - A solver for the [*MOLEK-SYNTEZ*](https://store.steampowered.com/app/1168880/MOLEKSYNTEZ/) solitaire game

[](./samples/full.png)

## Analysis

### The problem

The Solitaire minigame within *MOLEK-SYNTEZ* is laid out as follows:

A pack of Russian playing cards (36 cards - 6-10 and T, K, D, V) are dealt into a 6x6 grid. The objective is to form piles in descending order, from T to 6.

Descending runs can be moved as a single stack, so parts of runs can be moved quickly and easily.

If a complete stack (T-6) is placed onto an empty column, the entire column will lock.

If there aren’t any legal moves available, the player may Cheat by placing one card on top of a stack which would normally be illegal. Only one Cheated card can be on each stack at any time, however; and Cheated cards may not be moved to other illegal positions; therefore the game is still tricky.

### Features of the problem that are amenable to computational methods

The possible solution space for a randomly-generated solitaire is very large - while a human may be able to use some intuition, humans make mistakes; as the solitaire comes with no undo button this makes the game much harder to solve by hand.

### Why the problem lends itself to a computational solution

The problem is solvable computationally, by using a backtracking exhaustive algorithm; it tries as many moves as it can make, and if it reaches a point with no valid moves, it will go back until it has another valid move to make. It could be considered to be enumerative.

Furthermore, there is a medal for completing the solitaire game without Cheating; the medal was designed with the intent of being unobtainable (however there are boards which can be solved without Cheating) - a computer is far more likely to find these solutions than a person.

### Stakeholders

*Stakeholders were given the option to remain anonymous when signing up. This means some data will not be in this table or will be less specific.*

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Age | Gender | Operating System(s) |
| Toby Collier | 17 | Male | Windows/Linux |
| id-05012e5c-1462-4510-9fa3-e7e92b1062a9 | 45-50 | *[anonymous]* | Windows |
| id-d1274b61-aa9e-4300-8efe-ba1d5bca03a8 | 15-20 | *[anonymous]* | Windows |
| id-d86d8cf5-156e-497d-bdbf-0ebb3a4a3ca8 | 15-20 | *[anonymous]* | Windows |
| id-4b98d17e-c24e-49c6-a12a-2e95cbeee82a | 15-20 | *[anonymous]* | Windows |

### How the solution is appropriate to the needs of the stakeholders

The solution will be fully automatic, and will be customisable ([don’t] allow the solver to Cheat, limit its search depth, etc) as well as displaying, as part of a graphical interface, what the computer ‘sees’ (the app window will include a live board capture complete with the vision engine’s decisions)

### Objectives

* [ ] Standalone solitaire game
  + [ ] Detect win
  + [ ] Detect legal moves
* [x] The solver must support configuration:
  + [x] Maximum search depth (number of moves)
  + [x] Allow/disallow Cheating
* [ ] The solver should have a vision interface
  + [ ] Read the board from the game with CV
* [ ] The vision-solver must have a graphical configuration interface
  + [ ] The configuration interface must contain a ‘live’ display of the vision

### Research

Previously, my father and I [wrote a similarly-intented solver](https://www.youtube.com/watch?v=1MugAxSGbc8) for the ПАСЬЯНС (patience) minigame in [another Zachtronics game, *EXAPUNKS*](https://store.steampowered.com/app/716490/EXAPUNKS/). The source and release can be found [here](http://starbright.dyndns.org/starwort/patience/).

A user by the name Hegemege has also [created a solver for the solitaire](https://github.com/Hegemege/molek-syntez-solitaire-bot), however their solution [uses a heuristic](https://github.com/Hegemege/molek-syntez-solitaire-bot/blob/master/game_state.py#L244-L267) to narrow down its states and mine will use backtracking to perform a more exhaustive search

### Commentary

I previously (several months ago) began collecting some sample screen captures, and wrote a small framework ([solver/](./solver)). Additionally, the [common.c](solver_c/common.c), [common.h](solver_c/common.h), and [variadicmacros.h](solver_c/variadicmacros.h) are part of my personal C workspace and are copied here for completeness.

I have also created a makefile, and copied my formatter configuration here, to make the build process easier.

Full information about the progress I make will be available in [the changelog](./CHANGELOG.md)

## Roadmap

1. Write standalone player
2. Write solver
3. Write computer vision
4. Write visualisation window

## To-do

* [x] Create a clear description of the problem
* [x] Describe and justify of the features of the problem which are solvable by computational methods
* [x] Explain why the problem is amenable to a computational solution
* [x] Identify and describe the stakeholders in the solution
* [x] Explain how the solution is appropriate to the needs of the stakeholders
* [x] Identify other existing systems or ideas I can research
* [x] Discuss any suitable approaches to a solution
* [x] Create a requirements specification

## Design

### Solver algorithm overview

to solve a given board:  
 see if the board is solved  
 if it is, return no moves  
 for each available move:  
 make the move  
 try to solve the new board  
 if it was solved, return this move and those moves  
 otherwise, undo the move  
 there is no sequence of moves which can solve this board

Moves that are classed as ‘available’:

* Place a stack beginning with T from a non-empty stack onto an empty stack (it will never move from here)
* Place a stack beginning with *n* on a stack ending with *n* + 1 (K on T, D on K, V on D, 10 on V, 9 on 10, etc.)
* *Rules below this point apply only when Cheating is enabled*
* Move a stack consisting of a single card onto a column which does not contain a Cheated card
* Move a Cheated card from one column to another when an identical card is available in a valid position (this is really two moves, but is considered as one by the solver)

### Solver data formats

#### Input

##### Board state

The board state passed to the solver is represented as a single string consisting of 6 columns.

Each column consists of a sequence of up to fourteen characters, representing the cards in the column, consisting solely of T, K, D, V, 0, 9, 8, 7, or 6; terminated by a full stop (.) or exclamation mark (!), representing whether the top card of the column is Cheated (!) or not (.).

The solver accepts 5 options:

* -c
  + Allow Cheating.
  + The solver will only attempt to Cheat if a solution cannot be found without Cheating.
* -n <cache boundary>
  + The maximum number of moves remaining for the board state to be cached.
  + Defaults to 16.
  + If set to -1, all board states are cached. (Not recommended).
* -t <milliseconds>
  + How long to continue searching for a more optimal solution after one is found.
  + Defaults to 500 (0.5 seconds).
  + If set to -1, the solver will search all possibilities to find the most optimal solution.
* -d <maximum moves>
  + The minimum maximum number of moves to allow.
  + Defaults to 64.
  + Setting this to very low or very high values will increase run time and memory consumption.
* -m <maximum moves>
  + The maximum number of moves to allow.
  + Defaults to 1024.
  + Very large values will cause allocation errors.
  + It is recommended to make this a multiple of 4 as the solver will only attempt to solve with maximum depths of multiples of 4.
* -h
  + Print the help message and exit.

### Output

The solver outputs a sequence of moves, each formatted as x y -> new\_x new\_y, or x y !> new\_x new\_y for a Cheat move. For example, to move column 1 from card 3 to column 2 at card 5: 1 3 -> 2 5.

Additionally, if no solution is found in [depth] moves, it will print a message to standard error stating such: No solution found in <depth> moves

The solver attempts to solve with an initial search depth of the minimum maximum search depth; if no solution is found, the maximum search depth is incremented by 4 until the specified maximum is exceeded.

# Design of the Solitaire Solver

## Decomposition

The problem has been broken down as shown below:

1. Solver (written in C)
   1. Board state and parameter parsing
      1. Board state parsing is handled by my code
         * I decided to encode the board state in a simple-to-parse way in order to ease the writing of the solver
      2. Argument parsing is handled by getopt
         * This is a standard C module which is used to handle options
   2. Backtracking solution search
      * The solution search is recursive, using a backtracking algorithm:
        + to solve a given board:
          - see if the board is solved

if it is, return no moves

* + - * + for each available move:

make the move

try to solve the new board

if it was solved, return this move and those moves

otherwise, undo the move

* + - * + there is no sequence of moves which can solve this board
    - The real program differs slightly from this in order (the ‘board is solved’ check occurs after the move rather than at the beginning of the recursive step) and implementation (the sequence of moves is stored in a global array rather than returned dynamically from the functions, as C cannot do this easily)
    - Additionally, the solution search is repeated up to twice:
      * Once without Cheating (always)
        + If the board is possible to solve without Cheating, it is much faster to find this solution
      * Once with Cheating (only if Cheating is enabled, only if the first pass found no solution)
        + This can be *much* slower than searching without Cheating (finding a solution without Cheating can take under a tenth of a second, but with Cheating enabled can take several minutes)
  1. Solution output
     + The solution is printed to STDOUT
     + Each move in the solution is printed on its own line
       - Moves which do not Cheat are printed as X Y -> TO\_X TO\_Y, where X, Y, TO\_X, and TO\_Y represent the columns and rows of the card’s source and destination
       - Moves which Cheat are printed as X Y !> TO\_X TO\_Y, where X, Y, TO\_X, and TO\_Y represent the columns and rows of the card’s source and destination. Note that the only difference is in the arrow, and therefore users (or applications) who do not care if the solution Cheats or not can ignore it

1. Automation (written in Python)
   1. Vision
      1. Screenshot capture
         * Screenshot capture is delegated to the library PyAutoGUI. Where PyGetWindow (a requirement of PyAutoGUI) is available, the screenshot capture should\* be more sophisticated
         * \*Assuming my function works correctly - I have not been able to test this as my primary development system is not supported by PyGetWindow
      2. Screenshot parsing
         * The game has a ‘Pixel-Perfect’ scaling mode, and therefore the vision is simpler than if anti-aliasing needed to be accounted for
         * The game screenshot can be scaled and cropped to the base resolution (960x540), and the positions of the cards can be calculated with a set of constants:
           + The column width, in pixels (82)
           + The row height, in pixels (16)
           + The position of the top left card, as a set of pixel co-ordinates, where the origin is at the top left of the screen ((244, 149))
         * Each card in the 6x6 grid (as the vision solver always starts a new game before recognising the image) can be identified using a 8x9 picture of the card’s rank (these can be found in the samples directory), which are loaded in as constants and compared to the card (as found by cropping the full screenshot to the position of the card) using the ImageChops module of PIL (the Python Imaging Library, or rather its fork Pillow)
         * The board identification function then returns a 6-tuple of 6-tuples of 1-character strings
   2. Solver interface
      * The solver interface takes the Python representation of the board, as well as the required flags, translates them for the solver, and calls the solver.
        + There is also a wrapper function which will use the vision interface to determine the board on-screen
      * The function will determine the appropriate solver executable to use.
      * If running on Linux or MacOS, it will attempt to build the solver using make (which will not attempt to actually build the solver if it has done already).
      * It will then construct an argument list, using the determined solver executable path, the provided flags, and the provided board state, and uses subprocess to run the solver. subprocess is also used to enforce a timeout on the solver, so that it does not take too long (the default timeout is 10 seconds, more than enough for most boards).
      * Using the output of the solver, the function can parse the output into a list of moves which it will return, and optionally print some stats about the found solution.
   3. Input emulation
      * The input emulation interface contains only a couple of functions:
        + click, which clicks (and is used as an alternative to pyautogui.click as the clicks from this function fail to register)
        + run\_solution, which takes a list of moves and executes the given solution. Optionally, it can also click the ‘new game’ button.

## Analysis with Data-Oriented design

The following data structures (which can be found within types.h) were needed for the C code:

* Card, which is an unsigned 8-bit number representing the value of a card.
* Column, which is a structure representing a single column, containing:
  + A fixed-size array of 15 Cards (large enough for the largest possible stack of cards: XXXXXTKDV09876C, where X is any card and C is any Cheated card)
  + An unsigned 8-bit integer, which stores the number of cards in this Column
  + An unsigned 8-bit integer, which stores the position of the beginning of the stack of Cards which may be moved together within this Column
  + A boolean containing whether or not this Column contains a Cheated Card
* Board, which is a structure representing the entire board, containing 6 Column pointers
* Move, which is a structure representing a single move, containing:
  + An unsigned 8-bit integer representing the column the card is moving from (as an index)
  + An unsigned 8-bit integer representing the row the card is moving from (as an index)
  + An unsigned 8-bit integer representing the column the card is moving to (as an index)
  + An unsigned 8-bit integer representing the row the card is moving to (as an index)
  + A boolean containing whether or not the move being made is a Cheat
  + A boolean containing whether or not the Card being moved was Cheated previously
* BoardHashTable\_LLNode, which is a linked-list node for use in the hash table (an optimisation enabling the solver to prematurely discard states, based on whether it has encoutered them before or not)

# Planned steps

## Release 1 - create solver

* Create the solver

## Release 2 - create standalone, integrate solver

* Create a secondary, standalone implementation of the solitaire that a user can play with
* Use solver as a ‘hint’ function for the standalone implementation

## Release 3 - create interface for original

* Complete the vision
* Create the output (for the original version)
* Commandline interface

## Release 4 - final polishing

* Add graphical interface and vision overlay

# Test log

This will log the tests, their output, and whether the test was successful

## Test #1: Board State Parsing

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/ee750d115fd0a794eb092f42938e4e3377302c9c)

starwort@hedwig ~/Documents/NEA master$ dist/test 668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9. TKDV09.876TKD.V09876.TKDV09.876TKD.V09876. TKDV09876.TKDV09876.TKDV09876.TKDV09876... 'TKDV09878!TKDV09878.TKDV09766!TKDV09766...'  
668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9.:  
 06 07 14 11 12 10   
 06 11 08 10 11 12   
 08 06 07 14 06 11   
 07 12 12 09 13 13   
 09 >08< 09 08 >14< 14   
>10< 07 >13<>10< 13 >09<  
^00^^00^^00^^00^^00^^00^  
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
TKDV09.876TKD.V09876.TKDV09.876TKD.V09876.:  
>14< 08 >11<>14< 08 >11<  
 13 07 10 13 07 10   
 12 06 09 12 06 09   
 11 >14< 08 11 >14< 08   
 10 13 07 10 13 07   
 09 12 06 09 12 06   
^00^^00^^00^^00^^00^^00^  
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
TKDV09876.TKDV09876.TKDV09876.TKDV09876...:  
>14<>14<>14<>14<^00^^00^  
 13 13 13 13 00 00   
 12 12 12 12 00 00   
 11 11 11 11 00 00   
 10 10 10 10 00 00   
 09 09 09 09 00 00   
 08 08 08 08 00 00   
 07 07 07 07 00 00   
 06 06 06 06 00 00   
^00^^00^^00^^00^ 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
TKDV09878!TKDV09878.TKDV09766!TKDV09766...:  
>14<>14<>14<>14<^00^^00^  
 13 13 13 13 00 00   
 12 12 12 12 00 00   
 11 11 11 11 00 00   
 10 10 10 10 00 00   
 09 09 09 09 00 00   
 08 08 07 07 00 00   
 07 07 06 06 00 00   
!08! 08 !06! 06 00 00   
^00^^00^^00^^00^ 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00   
 00 00 00 00 00 00

Test result: 3 success, 1 failure

Post-mortem: Bit field limited to 3 bits due to false assumption that valid inputs may never have stacks beginning on a y greater than 7; ‘legitimate’ legal states may not, but ‘cheated’ legal states may (columns 1 and 3 of example 4). Note that columns 2 and 4 of example 4 are illegal states that will still be handled gracefully by the parser, after the fix to types.h.

## Test #2: stack\_begin() profiling/benchmarking

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/a23ce7b2f4f4f0bfc521bbfe2dc150b6c92d8b72)

starwort@hedwig ~/Documents/NEA master$ dist/test TKDV09876.TKDV09876.TKDV09876.TKDV09876...  
After 10,000,000 iterations:  
Elapsed: 0s 338,716µs  
Avg. time per iteration: 33.871601ns  
starwort@hedwig ~/Documents/NEA master$ dist/test TKDV09876.TKDV09876.TKDV09876.TKDV09876... 10000000000  
Using 10,000,000,000 iterations (instead of 10,000,000 iterations)  
After 10,000,000,000 iterations:  
Elapsed: 292s 945,474µs  
Avg. time per iteration: 29.294546ns

Test result: Successful; 10,000,000 calls of the function (in the worst case) can run in under half a second; each worst-case call takes approximately 29.3ns on average.

## Test #3: Board compression and comparison

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/8a86fec1f35856f467f279d8ca910350584e66fd)

starwort@hedwig ~/Documents/NEA master$ dist/test 668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9. TKDV09.876TKD.V09876.TKDV09.876TKD.V09876. TKDV09876.TKDV09876.TKDV09876.TKDV09876... 'TKDV09878!TKDV09878.TKDV09766!TKDV09766...'  
Original: 668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9.  
Compressed: 668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9.  
Identical: true  
Compare equal: true  
Original: TKDV09.876TKD.V09876.TKDV09.876TKD.V09876.  
Compressed: TKDV09.876TKD.V09876.TKDV09.876TKD.V09876.  
Identical: true  
Compare equal: true  
Original: TKDV09876.TKDV09876.TKDV09876.TKDV09876...  
Compressed: TKDV09876.TKDV09876.TKDV09876.TKDV09876...  
Identical: true  
Compare equal: true  
Original: TKDV09878!TKDV09878.TKDV09766!TKDV09766...  
Compressed: TKDV09878!TKDV09878.TKDV09766!TKDV09766...  
Identical: true  
Compare equal: true

Test result: Successful; Compression is the inverse of parsing, and comparison of an inflated (Board\*) state and a compressed (string/char\*) state works correctly.

## Test #4: Board state permutation invariance (comparison)

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/c7bc22b7a4cac4e2709f71ed49e2df7cbbedc77b)

Output: [![asciicast](data:image/svg+xml; charset=utf-8;base64,)](https://asciinema.org/a/2JbOBIMBh1DxGX0zAnPkRYNl3)

starwort@hedwig ~/Documents/NEA master$ bash -c '(read a; while read b; do dist/test \"$a\" \"$b\"; done) < samples/permutations.txt' | grep -B 2 false

(Author’s note: I use [xonsh](https://xon.sh/) so the bash snippet needs to be emulated in bash)

Test result: Successful; All permutations of the board state are accepted as equivalent in comparison

## Test #5: Board state permutation invariance (hashing)

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/4b3894199036135bc13e0ffa85358459d9ae34c8)

starwort@hedwig ~/Documents/NEA master$ bash -c '(while read b; do dist/test \"$b\"; done) < samples/permutations.txt' | python -c "import sys; print(set(sys.stdin.read().splitlines()))"  
{'57555068'}

Test result: Successful; All permutations of the board state are accepted as equivalent in hashing (single result over all permutations)

## Test #6: Hash collision likelihood

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/4b3894199036135bc13e0ffa85358459d9ae34c8)

starwort@hedwig ~/Documents/NEA master$ xargs -a samples/toy.txt dist/test | python -c "import sys; print(\_:=set(sys.stdin.read().splitlines()), len(\_))"; cat samples/toy.txt | wc -l  
{'228303792', '571845447', '1756605383', '1030069549', '328575377', '1811331921', '1590868318', '22301669', '1238129521', '1958393456', '56924340', '1245889', '919853135', '1563027505', '1507860214', '756225619'} 16  
15

Test result: Successful; 16 different hashes are generated; one per board state in the sample file (note that wc has an off-by-one error due to samples/toy.txt not containing a trailing newline)

## Test #7: Hash collision (Cheating)

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/4b3894199036135bc13e0ffa85358459d9ae34c8)

starwort@hedwig ~/Documents/NEA master$ dist/test 668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9.; dist/test "668790.7V6D87.T87D9K.V0T980.DV6KTK.0DVKT9!"  
228303792  
748997643

Test result: Successful; the board states differ only by the cheat state of column 5, and the hash is different

## Test #8: Optimal solution search

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/1472de4cb7bc1216b1e920f19927db1876bba531)

* Test:
  + for line in $(cat samples/toy.txt).splitlines():  
     first = $(dist/solver @(line) -t 0)  
     best = $(dist/solver @(line) -t 10000)  
     if first != best:  
     print(f'Found a more optimal solution for board {line}')  
     break
* Explanation of how the test works:
  + For every example within samples/toy.txt,
    - Run the solver with an optimisation time of 0 milliseconds (do not attempt to optimise)
      * This finds the solver’s first valid solution
    - Run the solver again with an optimisation time of 10,000 milliseconds (10 seconds)
      * This finds an optimal solution without taking forever
        + Using -1 in order to search all possibilities is a bad idea
    - If the optimal solution differs from the initial solution, then say so and exit
* Output of the test:
  + Found a more optimal solution for board VTK0KV.89K97V.697070.6TT880.D9T6K6.D7V8DD.
* Conclusion:
  + The solver successfully searches for (and finds) an additional solution, given enough time

# Debugging sessions

## Session: 2021-01-20

* Compiled with:
  + -O3 -g -lefence
* Relevant GDB session log:
  + (gdb) print compressed  
    $11 = (string) 0x55555d55ce60 "8TK7DK9\207\067\061DV6T\200\310\060Q000000003561", '0' <repeats 19 times>, "q000000Q0000000879DV6T000000000....000000000\220\364!V076VD6."  
    (gdb) print board  
    $12 = (Board \*) 0x55555d55c2a0  
    (gdb) print \*board  
    $13 = {cols = {0x55555d55c2e0, 0x55555d55c300, 0x55555d55c320, 0x55555d55c340, 0x55555d55c360, 0x55555d55c380}, depth = 0}  
    (gdb) print \*board->cols[0]  
    $14 = {cards = "\b\016\r\a\f\r\tW\a\001\f\v\006\016", count = 80 'P', stack\_begin = 152 '\230', cheated = false}
* Conclusion:
  + Something bad is going wrong
  + Somewhere, something is being corrupted
  + The cards buffer is being written to with garbage data far into out of bounds, resulting in the obvious visible garbage data (count = 80, stack\_begin = 152)

## Session: 2021-01-21

* Compiled with:
  + DEBUG=True
* Relevant GDB session log:
  + (gdb) set args -c 997VT8.076VD6.879DV6.T00KD8.09T6K8.VTK7DK.  
    (gdb) break solver.c:137  
    Breakpoint 1 at 0x13c3: file solver\_c/solver.c, line 138.  
    (gdb) run  
    Starting program: /home/starwort/Documents/NEA/dist/solver -c 997VT8.076VD6.879DV6.T00KD8.09T6K8.VTK7DK.  
    No solution found in 64 moves  
    No solution found in 128 moves  
    No solution found in 256 moves  
    No solution found in 512 moves  
    No solution found in 1024 moves  
    Clearing cache...  
      
    Breakpoint 1, main (argc=<optimised out>, argv=<optimised out>)  
     at solver\_c/solver.c:138  
    138 for (uint32 cache\_idx = 0; cache\_idx < 0x1000000;) {  
    (gdb) break fprintf  
    Breakpoint 2 at 0x5555555552a0: fprintf. (6 locations)  
    (gdb) continue  
    Continuing.  
      
    Breakpoint 2, fprintf (\_\_fmt=0x5555555571e4 "Done\n",   
     \_\_stream=0x7ffff7fa05c0 <\_IO\_2\_1\_stderr\_>)  
     at /usr/include/x86\_64-linux-gnu/bits/stdio2.h:100  
    100 return \_\_fprintf\_chk (\_\_stream, \_\_USE\_FORTIFY\_LEVEL - 1, \_\_fmt,  
    (gdb) up  
    #1 main (argc=<optimised out>, argv=<optimised out>)  
     at solver\_c/solver.c:145  
    145 eprintln("Done");  
    (gdb) up  
    Initial frame selected; you cannot go up.  
    (gdb) print \*board  
    value has been optimised out
* Conclusion:
  + Something bad is going wrong
  + For some reason, board is optimised out even before it is done being used
* Compiled with:
  + DEBUG=True
* Relevant GDB session log:
  + (gdb) print \*board  
    $1 = {cols = {0x55555d55c2e0, 0x55555d55c300, 0x55555d55c320,   
     0x55555d55c340, 0x55555d55c360, 0x55555d55c380}, depth = 0}  
    (gdb) print \*board->cols[0]  
    $2 = {cards = "\375\063\320\t\t\a\v\016\b\000\000\000\000", count = 9 '\t',   
     stack\_begin = 8 '\b', cheated = false}
* Conclusion:
  + Somehow, the garbage data 253, 51, 208 is being written to the beginning of the array of cards