# 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
* [x] The solver should have a vision interface
  + [x] Read the board from the game with CV
* [x] The vision-solver must have a graphical configuration interface
  + [ ] The configuration interface should 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.

## 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

## GUI

[Labelled image of the GUI](./window_labelled.png)

The GUI has been illustrated, with coloured regions used to identify them. Below is a description of what each region contains.

* Green region
  + Options for the solver
  + Each option in this group corresponds to a single solver option
    - Cheating checkbox: -c
    - Continue spinbox: -t
    - Cache spinbox: -n
    - Max moves spinbox: -m
    - Min depth spinbox: -d
  + This is so that users already familiar with the command-line interface can use the GUI easily, and so that users learning with the GUI are able to switch to the CLI if they so desire
* Blue region
  + The timeout for the subprocess
  + To the user, this appears to fit into the solver parameters, and therefore it is grouped with them; however, it really controls the timeout of the subprocess call, which is used to interface with the solver.
* Magenta region
  + The number of games to solve, automatically
  + This is a parameter entirely separate from the solver options, and as such it is separated from them by a divider
* Yellow region
  + The ‘Go’ button
  + When pressed, it begins a thread which runs the solver wrapper (which has been rewritten to use the status bar instead of stdout), and transforms into a ‘Stop’ button
* Red region
  + The status bar
  + Displays the tooltips for the labels, as well as the last line of output from the solver thread. When not hovering over a label, will display the hint text ‘Hover over a label for more information’

### 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

#### Solver arguments

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
         * Each board is given to the solver as a 42-character string, consisting of 6 columns, each of which consists of up to 15 cards followed by a column terminator.
         * Each card is represented by a digit 6-9, the digit 0, or a letter T, K, D, or V
         * Column terminators are the characters . and !, representing a column without a Cheat and a column with a Cheat, respectively
      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
  + This is the core structure for the project, and there is only ever one, which is global to the main file. It is loaded by parsing a string provided by the user, which takes a specific format (described in more detail in the previous breakdown) and is checked thoroughly in order to avoid segmentation faults.
* 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)

## Analysis of the Graphical User Interface

In order to ease development of the Graphical User Interface (GUI), the library pygubu was used. This allows me to create the components via the editor pygubu-designer, as well as connecting the events and bindings, and then write the code to *implement* the callbacks more quickly. Using pygubu allows the graphics code to stay separate from my own code, with the limited exception that I need to understand that tkinter is being used internally, and as such avoid blocking its thread with my operations

# Evidence of Project Development

## Principles

* Modular development
  + The solver is broken up into small modules which each handle their own tasks; the program contains 8 individual modules (board\_info, common compress hash parse memory timer) which each perform their own tasks and are used by the main program (Note that common is a general-purpose module I lifted straight from my personal C project folder, and as such many of its functions go unused in this project).
  + For example, board\_info handles the board; from [its header file](https://github.com/Starwort/NEA/blob/master/solver_c/board_info.h):
  + bool legal\_stack(const Column\* col, uint8 position);  
    uint8 stack\_begin(const Column\* col);  
    bool can\_move(const Board\* board, Move\* move);  
    int column\_if\_two\_moves(const Board\* board, const Move\* move);  
    bool solved(const Board\* board);  
    void apply\_move(Board\* board, const Move\* move);  
    void unapply\_move(Board\* board, const Move\* move);
  + That is, it can:
    - Determine if stacking from a point is legal
    - Determine where the beginning of a stack is
    - Determine where the intermediary column of a two-step Cheat is
    - Determine if a Board is solved
    - Apply a Move; that is, perform the action described by move on the board
    - Unapply a Move; that is, the reverse of applying it (necessary for the algorithm to backtrack)
  + For an alternative example, memory handles all the memory related tasks (allocation and freeing of complex structures); from [its header file](https://github.com/Starwort/NEA/blob/master/solver_c/memory.h):
  + Board\* allocate\_board();  
    void free\_board(Board\* board);  
    BoardHashTable\_LLNode\* create\_node(const string board\_state, int moves);  
    void deallocate\_list(BoardHashTable\_LLNode\* node);  
    Move\* copy\_move(Move\* move);
  + Its responsibilities are:
    - Creation and destruction of Boards (handled with pointers to allow mutability, and is the reason these functions are necessary)
    - Creation of hash table nodes (also handled with pointers, as this is the only way to link them together)
    - Destruction of hash table linked lists (performed recursively, but likely to be tail-call optimised by the compiler)
    - Copying Moves (used to place moves on the stack while allowing mutation, so the solver may continue attempting to optimise its solution)
* Annotation of the code
  + All main functions in the solver’s code are annotated, for example [the can\_move function](https://github.com/Starwort/NEA/blob/master/solver_c/board_info.c#L33-L104):
  + /\* Check if a given move is legal on a given board  
     \* Sets move->is\_cheat  
     \*/  
    bool can\_move(const Board\* board, Move\* move) {  
     const int from\_col = move->from\_x;  
     const int from\_y = move->from\_y;  
     const int to\_col = move->to\_x;  
     const Column\* fcol = board->cols[from\_col];  
     const Column\* tcol = board->cols[to\_col];  
      
     // cards can never be moved onto a column which contains a Cheated card  
     if (tcol->cheated) {  
     return false;  
     }  
      
     // never move from the beginning of a stack to an empty stack  
     if (fcol->stack\_begin == 0 && tcol->count == 0) {  
     return false;  
     }  
      
     // never move an invalid stack - this could actually be omitted as it's  
     // impossible for such a Move\* to be generated  
     // if (from\_y < fcol->count - 1 && !legal\_stack(fcol, from\_y)) {  
     // return false;  
     // }  
      
     move->is\_cheat = false; // assume not a cheat until we determine it is  
     // if there are no cards in this column then the move is valid  
     if (tcol->count == 0) {  
     return true;  
     }  
      
     // otherwise check the top card and compare it to the card being moved  
     if (tcol->cards[tcol->count - 1] == fcol->cards[from\_y] + 1) {  
     return true;  
     }  
      
     if (from\_y < fcol->count - 1) {  
     // moving a stack; must not Cheat  
     return false;  
     }  
      
     // if we reach here then the attempted move Cheats  
     move->is\_cheat = true;  
     // Columns containing a full stack may not be the target of any move,  
     // including a Cheat  
     if (fcol->count == 9 && fcol->stack\_begin == 0) {  
     return false;  
     }  
     if (fcol->cheated) {  
     // this is only a valid move if fcol->cards[fcol->count - 1] can  
     // also be found in a legitimate position on a column that is neither  
     // fcol nor tcol  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     Column\* col = board->cols[i];  
     // the stack must be more than one card (the card must be  
     // in a valid position)  
     if (col->stack\_begin < col->count - 1  
     // and must end with the card we're trying to move  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return true; // legal move (that is technically 2 moves)  
     }  
     }  
     // not legal; no column permits this move  
     return false;  
     }  
     // Cheating with a card that was previously not Cheated is always valid  
     return true;  
    }
  + This function has a documentation string, and is littered with comments describing what it’s doing and why
* Meaningful variable names
  + The vast majority of the solver’s code uses meaningful variable names (with a few simpler functions using shorter, but still understandable, names such as fcol for the from column, or i for loop counters)
  + A good example is [parse\_input from parse.c](https://github.com/Starwort/NEA/blob/master/solver_c/parse.c#L3-L99):
  + /\* Given a board represented as a string, generate a board  
     \* object and populate its values.  
     \*  
     \* Board data is expected to be in the format  
     \* /([TKDV06-9]{0,15}(\.|!)){6}/ (given as a regex)  
     \*  
     \* T, K, D, V, 9, 8, 7, and 6 represent themselves, and 0  
     \* represents 10  
     \*  
     \* The board needs freeing after use.  
     \*/  
    Board\* parse\_input(const string given\_state) {  
     int num\_cols\_found = 0;  
     for (int i = 0; i < 42; i++) {  
     if (given\_state[i] == '\0') {  
     eprintfln("Bad input string: Expected 42 characters but only got %d", i);  
     exit(EXIT\_FAILURE);  
     }  
     if (given\_state[i] == '.' || given\_state[i] == '!') {  
     num\_cols\_found++;  
     }  
     }  
     if (num\_cols\_found != 6) {  
     eprintfln("Bad input string: Expected 6 columns, got %d", num\_cols\_found);  
     exit(EXIT\_FAILURE);  
     }  
     Board\* board = allocate\_board();  
     int card = 0;  
     int num\_cards\_found = 0;  
     int cards\_found[15] = {0};  
     for (int i = 0; i < 6; i++) {  
     Column\* this\_column = board->cols[i];  
     uint8 current\_card;  
     uint8 y = 0;  
     while ((current\_card = given\_state[card++]) != '.' && current\_card != '!') {  
     uint8 card\_char = current\_card;  
     switch (current\_card) {  
     case 'T':  
     current\_card = 14;  
     break;  
     case 'K':  
     current\_card = 13;  
     break;  
     case 'D':  
     current\_card = 12;  
     break;  
     case 'V':  
     current\_card = 11;  
     break;  
     case '0':  
     current\_card = 10;  
     break;  
     case '9':  
     case '8':  
     case '7':  
     case '6':  
     current\_card -= '0';  
     break;  
     default:  
     eprintfln(  
     "Bad input string: Expected a digit in the range 6-9, '0', "  
     "'T', 'K', 'D', V, '!', or '.'; got '%c'",  
     current\_card);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     cards\_found[current\_card]++;  
     if (cards\_found[current\_card] > 4) {  
     eprintfln(  
     "Bad input string: Impossible state; Found %d %c (expected 4)",  
     cards\_found[current\_card],  
     card\_char);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     if (y > 14) {  
     eprintfln(  
     "Bad input string: Impossible state; Found %d cards in column (max "  
     "15)",  
     y);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     this\_column->cards[y++] = current\_card;  
     num\_cards\_found++;  
     }  
     this\_column->count = y;  
     this\_column->cheated = current\_card == '!';  
     this\_column->stack\_begin = stack\_begin(this\_column);  
     }  
     if (num\_cards\_found != 36) {  
     eprintfln("Bad input string: Expected 36 cards, got %d", num\_cards\_found);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     return board;  
    }
  + Each variable is named in a way that makes it very easy to infer the variable’s purpose, such as this\_column, which is the column being parsed, or num\_cards\_found, which is the number of cards that have been found; etc.
* Validation
  + The above code sample is also an example of validation, as there are numerous points where if invalid data is found, the user is informed and the program terminates (with a failure exit code)
* Use of appropriate data structures for storing data
  + The solver uses a selection of data structures, some specialised (Board, Column, and Move), and some more general (arrays of various types, linked lists for the hash table)
  + These can be seen within [types.h](https://github.com/Starwort/NEA/blob/master/solver_c/types.h):
  + typedef uint8 Card;  
      
    typedef struct \_Column {  
     // largest possible stack:  
     // XXXXXTKDV09876C (15)  
     Card cards[15];  
     uint8 count;  
     uint8 stack\_begin;  
     bool cheated;  
    } Column;  
      
    typedef struct \_Board {  
     Column\* cols[6]; // Board.cols[x][y]  
    } Board;  
      
    typedef struct \_Move {  
     uint8 from\_x;  
     uint8 from\_y;  
     uint8 to\_x;  
     uint8 to\_y;  
     bool is\_cheat;  
     bool was\_cheat;  
    } Move;  
      
    typedef struct \_BoardHashTable\_LLNode {  
     const char\* board\_state;  
     int unsolvable\_in;  
     const struct \_BoardHashTable\_LLNode\* next;  
    } BoardHashTable\_LLNode;
  + Every type used by the solver is defined in this file, with a brief explanation of their design where it isn’t obvious - for instance, the hash table nodes contain a compressed board state used to ensure there are no hash collisions, an integer representing the minimum number of moves a step may have remaining in order for it to even attempt solving the state, and a pointer to the next node
* Error handling
  + Using -Wall -Wextra -Werror -pedantic-errors at compile time ensures that my code avoids the vast majority of possible errors; in addition, since the user’s input is validated (and rejected if it’s bad), there are no places a bug is likely to sneak in (even a segmentation fault due to an incredibly high -m is impossible, as calloc is an *optimistic* memory allocator, and it has been tested on both supported operating systems with -m 2147483647, the largest possible value)

# Iterations

## C solver

### Minimum Viable Product [2021-01-03T14:42Z]

* This version of the solver was very minimalistic, with no configuration, no error handling or input validation, and no Cheating mechanics (not required for many games, so is a viable product)
* As a result of its simplicity it is much simpler than each subsequent version
* Code:
  + board\_info.c (click to expand)
  + #include "board\_info.h"  
      
    bool legal\_stack(Column\* col, uint8 position) {  
     uint8 value = col->cards[position];  
     for (uint8 y = position; y < col->count; y++) {  
     if (col->cards[y] != value--) {  
     return false;  
     }  
     }  
     return true;  
    }  
      
    uint8 stack\_begin(Column\* col) {  
     if (col->count == 0) {  
     return 0;  
     }  
     if (col->cheated) {  
     return col->count - 1;  
     }  
     uint8 y = col->count - 1;  
     uint8 card = col->cards[y];  
     while (col->cards[--y] == ++card && y != 255) {  
     }  
     return y + 1;  
    }  
      
    bool can\_move(Board\* board, uint8 from\_col, uint8 from\_y, uint8 to\_col) {  
     Column\* fcol = board->cols[from\_col];  
     Column\* tcol = board->cols[to\_col];  
     if (tcol->cheated) {  
     return false;  
     }  
     if (from\_y < fcol->count - 1 && !legal\_stack(fcol, from\_y)) {  
     return false;  
     }  
     if (tcol->count == 0) {  
     return true;  
     }  
     return (tcol->cards[tcol->count - 1] == fcol->cards[from\_y] + 1);  
    }  
      
    /\* Determine if the board is solved  
     \*  
     \* The board is solved if all columns are either empty or  
     \* contain a full stack starting from the top of the tableau  
     \* (col->cards == {14, 13, 12, 11, 10, 9, 8, 7, 6} and  
     \* therefore col->count == 9 and col->stack\_begin == 0)  
     \*/  
    bool solved(Board\* board) {  
     for (int col = 0; col < 6; col++) {  
     if (board->cols[col]->count == 0  
     || (board->cols[col]->count == 9 && board->cols[col]->stack\_begin == 0)) {  
     continue;  
     }  
     return false;  
     }  
     return true;  
    }  
      
    /\* Apply move to board.  
     \*  
     \* Does not check validity of move.  
     \*/  
    void apply\_move(Board\* board, Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     uint8 n\_elements = from\_col->count - move->from\_y;  
     memcpy(to\_col->cards + move->to\_y, from\_col->cards + move->from\_y, n\_elements);  
     from\_col->count = move->from\_y;  
     from\_col->stack\_begin = stack\_begin(from\_col);  
     to\_col->count += n\_elements;  
    }  
      
    /\* Unapply move to board; opposite of apply\_move \*/  
    void unapply\_move(Board\* board, Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     uint8 n\_elements = to\_col->count - move->to\_y;  
     memcpy(from\_col->cards + move->from\_y, to\_col->cards + move->to\_y, n\_elements);  
     from\_col->stack\_begin = from\_col->count;  
     from\_col->count = move->from\_y + n\_elements;  
     to\_col->count = move->to\_y;  
    }
  + board\_info.h (click to expand)
  + #pragma once  
    #include "types.h"  
      
    bool legal\_stack(Column\* col, uint8 position);  
    uint8 stack\_begin(Column\* col);  
    bool can\_move(Board\* board, uint8 from\_col, uint8 from\_y, uint8 to\_col);  
    bool solved(Board\* board);  
    void apply\_move(Board\* board, Move\* move);  
    void unapply\_move(Board\* board, Move\* move);
  + compress.c (click to expand)
  + #include "compress.h"  
      
    /\* Compress a board state to a board state string  
     \* Return string needs freeing  
     \*/  
    string compress(Board\* board) {  
     // board state string is always 42 characters: 36 cards + 6 columns  
     string out = malloc(42 \* sizeof(char));  
     uint8 idx = 0;  
     for (int col = 0; col < 6; col++) {  
     Column\* column = board->cols[col];  
     for (int card = 0; card < column->count; card++) {  
     switch (column->cards[card]) {  
     case 14:  
     out[idx++] = 'T';  
     break;  
     case 13:  
     out[idx++] = 'K';  
     break;  
     case 12:  
     out[idx++] = 'D';  
     break;  
     case 11:  
     out[idx++] = 'V';  
     break;  
     case 10:  
     out[idx++] = '0';  
     break;  
     default:  
     out[idx++] = column->cards[card] + '0';  
     break;  
     }  
     }  
     out[idx++] = column->cheated ? '!' : '.';  
     }  
     return out;  
    }  
      
    bool \_compare(Column\* col, string\* compressed) {  
     if ((\*compressed)[col->count] != (col->cheated ? '!' : '.')) {  
     return false;  
     }  
     int card;  
     for (card = 0; card < col->count; card++) {  
     switch ((\*compressed)[card]) {  
     case '.':  
     case '!':  
     return false;  
     case 'T':  
     if (col->cards[card] != 14) {  
     return false;  
     }  
     break;  
     case 'K':  
     if (col->cards[card] != 13) {  
     return false;  
     }  
     break;  
     case 'D':  
     if (col->cards[card] != 12) {  
     return false;  
     }  
     break;  
     case 'V':  
     if (col->cards[card] != 11) {  
     return false;  
     }  
     break;  
     case '0':  
     if (col->cards[card] != 10) {  
     return false;  
     }  
     break;  
     default:  
     if (col->cards[card] != ((\*compressed)[card] - '0')) {  
     return false;  
     }  
     break;  
     }  
     }  
     \*compressed += card + 1;  
     return true;  
    }  
      
    /\* Compare a board state to a compressed board string.  
     \* Column order does not matter and so is ignored.  
     \*/  
    bool compare(Board\* board, string compressed) {  
     bool matched[6] = {0, 0, 0, 0, 0, 0};  
     for (int col\_compressed = 0; col\_compressed < 6; col\_compressed++) {  
     bool found\_match = false;  
     for (int col\_real = 0; col\_real < 6; col\_real++) {  
     if (matched[col\_real]) {  
     continue;  
     }  
     if (\_compare(board->cols[col\_real], &compressed)) {  
     matched[col\_real] = found\_match = true;  
     break;  
     }  
     }  
     if (!found\_match) {  
     return false;  
     }  
     }  
     return true;  
    }
  + compress.h (click to expand)
  + #include "types.h"  
      
    string compress(Board\* board);  
    bool compare(Board\* board, string compressed);
  + hash.c (click to expand)
  + #include "hash.h"  
      
    /\* The MODULUS and GENERATOR are used in the hash function to generate a  
     \* pseudo-random number based on each board position. The MODULUS is a  
     \* prime number with enough bits to fill the hash table, and the GENERATOR  
     \* is a primitive root with respect to that modulus. Conveniently, 2^31-1  
     \* is a prime number, and the generator here is that used in the MINSTD  
     \* Lehmer RNG and is equal to 7^5.  
     \*/  
    #define MODULUS 2147483647  
    #define GENERATOR 16807  
      
    /\* The powmod function calculates (y^x) mod (MODULUS) and is used in the  
     \* hash function. We are assuming that uint32 is long enough to hold the  
     \* modulus and uint64 is long enough to hold two longs multiplied  
     \* together.  
     \*  
     \* Square-and-multiply method. Check each bit of x in turn, square the  
     \* answer so far and multiply by y if the bit was a 1.  
     \*/  
    uint32 powmod(uint32 y, uint32 x) {  
     uint32 answer = 1;  
     int i;  
     for (i = 0; i < 32; i++) {  
     if (i > 0 && answer > 1) {  
     answer = ((uint64)answer \* (uint64)answer) % MODULUS;  
     }  
     if (x & 0x80000000) {  
     answer = ((uint64)answer \* (uint64)y) % MODULUS;  
     }  
     x <<= 1;  
     }  
     return answer;  
    }  
      
    /\* The hash function calculates a hash based on the current board position.  
     \* The hash is a pseudo-random number which must be the same for any  
     \* two identical board positions. In this game of Solitaire we consider  
     \* two board positions similar if they contain the same columns of cards  
     \* in any order, so the hash function is invariant when columns are swapped.  
     \*  
     \* The value of the hash will be (((a^col\_1)^col\_2)^...)^col\_N mod (MODULUS) and  
     \* then further modified by the cell card if any; where a is the RNG generator  
     \* and col\_1...col\_N are hash values computed from the card columns. This  
     \* value is equivalent to a^(col\_1\*col\_2\*...\*col\_N) and therefore the order of  
     \* the columns is not significant. Each column hash must not be zero, and  
     \* so will be 1 if the column is empty.  
     \*/  
    uint32 hash(Board\* board) {  
     uint32 rv = GENERATOR;  
     for (uint8 col = 0; col < 6; col++) {  
     rv = powmod(rv, hash\_column(board->cols[col]));  
     }  
     return rv;  
    }  
      
    /\* The column hash function calculates a hash based on a column's values.  
     \* The hash is a pseudo-random number which must be the same for any two  
     \* identical columns. In this game of Solitaire we consider two columns  
     \* of cards similar if they contain the same cards in the same order and  
     \* have the same 'cheat' flag.  
     \*  
     \* The value of the hash will be:  
     \* (cheated << 31) | (((33+card\_0)\*33+card\_1)\*33+...)\*33+card\_N mod (MODULUS)  
     \* where card\_n is equivalent to column->cards[n] and card\_N = the last card  
     \* in the column.  
     \*/  
    uint32 hash\_column(Column\* column) {  
     uint32 rv = 1;  
     for (uint8 card = 0; card < column->count; card++) {  
     rv = ((uint64)rv \* (uint64)33 + column->cards[card]) % MODULUS;  
     }  
     return rv | (uint32)(column->cheated << 31);  
    }
  + hash.h (click to expand)
  + #include "types.h"  
      
    uint32 powmod(uint32 y, uint32 x);  
    uint32 hash(Board\* board);  
    uint32 hash\_column(Column\* column);
  + memory.c (click to expand)
  + #include "memory.h"  
      
    /\* Allocate a board and its columns  
     \* Needs freeing after use  
     \*/  
    Board\* allocate\_board() {  
     Board\* board = malloc(sizeof(Board));  
     board->depth = 0;  
     for (int i = 0; i < 6; i++) {  
     board->cols[i] = malloc(sizeof(Column));  
     }  
     return board;  
    }  
    /\* Free a board and its columns  
     \*/  
    void free\_board(Board\* board) {  
     for (int i = 0; i < 6; i++) {  
     free(board->cols[i]);  
     }  
     free(board);  
    }
  + memory.h (click to expand)
  + #include "types.h"  
      
    Board\* allocate\_board();  
    void free\_board(Board\* board);
  + parse.c (click to expand)
  + #include "parse.h"  
      
    /\* Given a board represented as a string, generate a board  
     \* object and populate its values.  
     \*  
     \* Board data is expected to be in the format  
     \* /([TKDV06-9]{0,14}(\.|!)){6}/ (given as a regex)  
     \*  
     \* T, K, D, V, 9, 8, 7, and 6 represent themselves, and 0  
     \* represents 10  
     \*  
     \* The board needs freeing after use.  
     \*/  
    Board\* parse\_input(string given\_state) {  
     Board\* board = allocate\_board();  
     int card = 0;  
     for (int i = 0; i < 6; i++) {  
     Column\* this\_column = board->cols[i];  
     uint8 current\_card;  
     uint8 y = 0;  
     while ((current\_card = given\_state[card++]) != '.' && current\_card != '!') {  
     switch (current\_card) {  
     case 'T':  
     current\_card = 14;  
     break;  
     case 'K':  
     current\_card = 13;  
     break;  
     case 'D':  
     current\_card = 12;  
     break;  
     case 'V':  
     current\_card = 11;  
     break;  
     case '0':  
     current\_card = 10;  
     break;  
     default:  
     current\_card -= '0';  
     }  
     this\_column->cards[y++] = current\_card;  
     }  
     this\_column->count = y;  
     this\_column->cheated = current\_card == '!';  
     this\_column->stack\_begin = stack\_begin(this\_column);  
     }  
     return board;  
    }
  + parse.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "memory.h"  
    #include "types.h"  
      
    Board\* parse\_input(string given\_state);
  + solver.c (click to expand)
  + #include "solver.h"  
      
    Move\* moves[1024];  
      
    /\* Search recursively for the next move; backtracking.  
     \* Return -1 if no valid moves found, non-negative to  
     \* indicate number of moves.  
     \*  
     \* TODO: implement cheating  
     \*/  
    int step(Board\* board, int depth, int max\_moves) {  
     printfln("Depth %d:", depth);  
     visualise\_board(board, false);  
     Move\* move = malloc(sizeof(Move));  
     if (depth == max\_moves) {  
     return -1;  
     }  
     for (move->from\_x = 0; move->from\_x < 6; move->from\_x++) {  
     for (move->to\_x = 0; move->to\_x < 6; move->to\_x++) {  
     if (move->from\_x == move->to\_x) {  
     continue;  
     }  
     if (board->cols[move->from\_x]->count == 0) {  
     continue;  
     }  
     move->from\_y = board->cols[move->from\_x]->stack\_begin;  
     move->to\_y = board->cols[move->to\_x]->count;  
     moves[depth] = move;  
     if (!can\_move(board, move->from\_x, move->from\_y, move->to\_x)) {  
     continue;  
     }  
     apply\_move(board, move);  
     if (solved(board)) {  
     return depth + 1;  
     }  
     int found = step(board, depth + 1, max\_moves);  
     if (found != -1) {  
     return found;  
     }  
     unapply\_move(board, move);  
     }  
     }  
     free(move);  
     return -1;  
    }  
      
    int main(int argc, string argv[]) {  
     if (argc < 2) {  
     println("Need starting board state");  
     return 1;  
     }  
     Board\* board = parse\_input(argv[1]);  
      
     for (int max\_depth = 64; max\_depth <= 1024; max\_depth <<= 1) {  
     int n\_moves = step(board, 0, max\_depth);  
     if (n\_moves == -1) {  
     eprintfln("No solution found in %d moves", max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d -> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     }  
     return 0;  
     }  
     }  
     return 0;  
    }
  + solver.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "debug.h"  
    #include "parse.h"  
    #include "types.h"  
      
    int step(Board\* board, int depth, int max\_moves);
  + types.h (click to expand)
  + #pragma once  
    #include "common.h"  
      
    typedef uint8 Card;  
      
    typedef struct \_Column {  
     Card cards[14];  
     uint8 count;  
     uint8 stack\_begin;  
     bool cheated;  
    } Column;  
      
    typedef struct \_Board {  
     Column\* cols[6]; // Board.cols[x][y]  
     int depth;  
    } Board;  
      
    typedef struct \_Move {  
     uint8 from\_x;  
     uint8 from\_y;  
     uint8 to\_x;  
     uint8 to\_y;  
     bool is\_cheat;  
    } Move;

### Cheating [2021-01-19T19:00Z]

* This version of the solver added support for Cheating, as well as experimental support for options (using the argp library, which was later dropped in favour of the cross-platform getopt module).
* Additionally, several algorithm and style improvements were made, including preparation for the use of a hash table to optimise solution search, along with documentation, and adding trailing blank lines to every source file
* Code:
  + board\_info.c (click to expand)
  + #include "board\_info.h"  
      
    /\* Determine if a stack is legal  
     \*/  
    bool legal\_stack(Column\* col, uint8 position) {  
     uint8 value = col->cards[position];  
     for (uint8 y = position; y < col->count; y++) {  
     if (col->cards[y] != value--) {  
     return false;  
     }  
     }  
     return true;  
    }  
      
    /\* Determine where the stack begins  
     \*/  
    uint8 stack\_begin(Column\* col) {  
     // if there are no cards in the column then there is no stack  
     if (col->count == 0) {  
     return 0;  
     }  
     // if there is a cheated card then that is the entire stack  
     if (col->cheated) {  
     return col->count - 1;  
     }  
     uint8 y = col->count - 1;  
     uint8 card = col->cards[y];  
     while (--y != 255 && col->cards[y] == ++card) {  
     }  
     return y + 1;  
    }  
      
    /\* Check if a given move is legal on a given board  
     \* Sets move->is\_cheat  
     \*/  
    bool can\_move(Board\* board, Move\* move) {  
     int from\_col = move->from\_x;  
     int from\_y = move->from\_y;  
     int to\_col = move->to\_x;  
     Column\* fcol = board->cols[from\_col];  
     Column\* tcol = board->cols[to\_col];  
      
     // cards can never be moved onto a column which contains a Cheated card  
     if (tcol->cheated) {  
     return false;  
     }  
      
     // never move from the beginning of a stack to an empty stack  
     if (fcol->stack\_begin == 0 && tcol->count == 0) {  
     return false;  
     }  
      
     // never move an invalid stack - this could actually be omitted as it's  
     // impossible for such a Move\* to be generated  
     // if (from\_y < fcol->count - 1 && !legal\_stack(fcol, from\_y)) {  
     // return false;  
     // }  
      
     move->is\_cheat = false; // assume not a cheat until we determine it is  
     // if there are no cards in this column then the move is valid  
     if (tcol->count == 0) {  
     return true;  
     }  
      
     // otherwise check the top card and compare it to the card being moved  
     if (tcol->cards[tcol->count - 1] == fcol->cards[from\_y] + 1) {  
     return true;  
     }  
      
     // if we reach here then the attempted move Cheats  
     move->is\_cheat = true;  
     if (fcol->cheated) {  
     // this is only a valid move if fcol->cards[fcol->count - 1] can  
     // also be found in a legitimate position on a column that is neither  
     // fcol nor tcol  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     Column\* col = board->cols[i];  
     // the stack must be more than one card (the card must be  
     // in a valid position)  
     if (col->stack\_begin < col->count - 1  
     // and must end with the card we're trying to move  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return true; // legal move (that is technically 2 moves)  
     }  
     }  
     // not legal; no column permits this move  
     return false;  
     }  
     // Cheating with a card that was previously not Cheated is always valid  
     return true;  
    }  
      
    /\* Determine if move is the special two-move operation Cheat -> Cheat  
     \* If it is, return the column of the intermediate column.  
     \* Else, return -1  
     \*  
     \* Assume that move is a legal Cheat move  
     \*/  
    int column\_if\_two\_moves(Board\* board, Move\* move) {  
     int from\_col = move->from\_x;  
     int from\_y = move->from\_y;  
     int to\_col = move->to\_x;  
     Column\* fcol = board->cols[from\_col];  
     Column\* tcol = board->cols[to\_col];  
     if (fcol->cheated) {  
     // find the column using the algorithm from can\_move  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     Column\* col = board->cols[i];  
     if (col->stack\_begin < col->count - 1  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return i; // found the two-move column  
     }  
     }  
     }  
     return -1; // this move cannot be Cheat -> Cheat  
    }  
      
    /\* Determine if the board is solved  
     \*  
     \* The board is solved if all columns are either empty or  
     \* contain a full stack starting from the top of the tableau  
     \* (col->cards == {14, 13, 12, 11, 10, 9, 8, 7, 6} and  
     \* therefore col->count == 9 and col->stack\_begin == 0)  
     \*/  
    bool solved(Board\* board) {  
     for (int col = 0; col < 6; col++) {  
     if (board->cols[col]->count == 0 // column is empty  
     || (board->cols[col]->count == 9 // or contains the full stack  
     && board->cols[col]->stack\_begin == 0)) {  
     continue;  
     }  
     return false;  
     }  
     return true;  
    }  
      
    /\* Apply move to board.  
     \*  
     \* Does not check validity of move.  
     \*/  
    void apply\_move(Board\* board, Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     uint8 n\_elements = from\_col->count - move->from\_y;  
     memcpy(to\_col->cards + move->to\_y, from\_col->cards + move->from\_y, n\_elements);  
     from\_col->count = move->from\_y;  
     from\_col->stack\_begin = stack\_begin(from\_col);  
     to\_col->count += n\_elements;  
    }  
      
    /\* Unapply move to board; opposite of apply\_move  
     \*/  
    void unapply\_move(Board\* board, Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     uint8 n\_elements = to\_col->count - move->to\_y;  
     memcpy(from\_col->cards + move->from\_y, to\_col->cards + move->to\_y, n\_elements);  
     from\_col->stack\_begin = from\_col->count;  
     from\_col->count = move->from\_y + n\_elements;  
     to\_col->count = move->to\_y;  
    }
  + board\_info.h (click to expand)
  + #pragma once  
    #include "types.h"  
      
    bool legal\_stack(Column\* col, uint8 position);  
    uint8 stack\_begin(Column\* col);  
    bool can\_move(Board\* board, Move\* move);  
    int column\_if\_two\_moves(Board\* board, Move\* move);  
    bool solved(Board\* board);  
    void apply\_move(Board\* board, Move\* move);  
    void unapply\_move(Board\* board, Move\* move);
  + compress.c (click to expand)
  + #include "compress.h"  
      
    /\* Compress a board state to a board state string  
     \* Return string needs freeing  
     \*/  
    string compress(Board\* board) {  
     // board state string is always 42 characters: 36 cards + 6 columns  
     string out = malloc(42 \* sizeof(char));  
     uint8 idx = 0;  
     for (int col = 0; col < 6; col++) {  
     Column\* column = board->cols[col];  
     for (int card = 0; card < column->count; card++) {  
     switch (column->cards[card]) {  
     case 14:  
     out[idx++] = 'T';  
     break;  
     case 13:  
     out[idx++] = 'K';  
     break;  
     case 12:  
     out[idx++] = 'D';  
     break;  
     case 11:  
     out[idx++] = 'V';  
     break;  
     case 10:  
     out[idx++] = '0';  
     break;  
     default:  
     out[idx++] = column->cards[card] + '0';  
     break;  
     }  
     }  
     out[idx++] = column->cheated ? '!' : '.';  
     }  
     return out;  
    }  
      
    bool \_compare(Column\* col, string\* compressed) {  
     if ((\*compressed)[col->count] != (col->cheated ? '!' : '.')) {  
     return false;  
     }  
     int card;  
     for (card = 0; card < col->count; card++) {  
     switch ((\*compressed)[card]) {  
     case '.':  
     case '!':  
     return false;  
     case 'T':  
     if (col->cards[card] != 14) {  
     return false;  
     }  
     break;  
     case 'K':  
     if (col->cards[card] != 13) {  
     return false;  
     }  
     break;  
     case 'D':  
     if (col->cards[card] != 12) {  
     return false;  
     }  
     break;  
     case 'V':  
     if (col->cards[card] != 11) {  
     return false;  
     }  
     break;  
     case '0':  
     if (col->cards[card] != 10) {  
     return false;  
     }  
     break;  
     default:  
     if (col->cards[card] != ((\*compressed)[card] - '0')) {  
     return false;  
     }  
     break;  
     }  
     }  
     \*compressed += card + 1;  
     return true;  
    }  
      
    /\* Compare a board state to a compressed board string.  
     \* Column order does not matter and so is ignored.  
     \*/  
    bool compare(Board\* board, string compressed) {  
     bool matched[6] = {0, 0, 0, 0, 0, 0};  
     for (int col\_compressed = 0; col\_compressed < 6; col\_compressed++) {  
     bool found\_match = false;  
     for (int col\_real = 0; col\_real < 6; col\_real++) {  
     if (matched[col\_real]) {  
     continue;  
     }  
     if (\_compare(board->cols[col\_real], &compressed)) {  
     matched[col\_real] = found\_match = true;  
     break;  
     }  
     }  
     if (!found\_match) {  
     return false;  
     }  
     }  
     return true;  
    }
  + compress.h (click to expand)
  + #include "types.h"  
      
    string compress(Board\* board);  
    bool compare(Board\* board, string compressed);
  + hash.c (click to expand)
  + #include "hash.h"  
      
    /\* The MODULUS and GENERATOR are used in the hash function to generate a  
     \* pseudo-random number based on each board position. The MODULUS is a  
     \* prime number with enough bits to fill the hash table, and the GENERATOR  
     \* is a primitive root with respect to that modulus. Conveniently, 2^31-1  
     \* is a prime number, and the generator here is that used in the MINSTD  
     \* Lehmer RNG and is equal to 7^5.  
     \*/  
    #define MODULUS 2147483647  
    #define GENERATOR 16807  
      
    /\* The powmod function calculates (y^x) mod (MODULUS) and is used in the  
     \* hash function. We are assuming that uint32 is long enough to hold the  
     \* modulus and uint64 is long enough to hold two longs multiplied  
     \* together.  
     \*  
     \* Square-and-multiply method. Check each bit of x in turn, square the  
     \* answer so far and multiply by y if the bit was a 1.  
     \*/  
    uint32 powmod(uint32 y, uint32 x) {  
     uint32 answer = 1;  
     int i;  
     for (i = 0; i < 32; i++) {  
     if (i > 0 && answer > 1) {  
     answer = ((uint64)answer \* (uint64)answer) % MODULUS;  
     }  
     if (x & 0x80000000) {  
     answer = ((uint64)answer \* (uint64)y) % MODULUS;  
     }  
     x <<= 1;  
     }  
     return answer;  
    }  
      
    /\* The hash function calculates a hash based on the current board position.  
     \* The hash is a pseudo-random number which must be the same for any  
     \* two identical board positions. In this game of Solitaire we consider  
     \* two board positions similar if they contain the same columns of cards  
     \* in any order, so the hash function is invariant when columns are swapped.  
     \*  
     \* The value of the hash will be (((a^col\_1)^col\_2)^...)^col\_N mod (MODULUS) and  
     \* then further modified by the cell card if any; where a is the RNG generator  
     \* and col\_1...col\_N are hash values computed from the card columns. This  
     \* value is equivalent to a^(col\_1\*col\_2\*...\*col\_N) and therefore the order of  
     \* the columns is not significant. Each column hash must not be zero, and  
     \* so will be 1 if the column is empty.  
     \*/  
    uint32 hash(Board\* board) {  
     uint32 rv = GENERATOR;  
     for (uint8 col = 0; col < 6; col++) {  
     rv = powmod(rv, hash\_column(board->cols[col]));  
     }  
     return rv;  
    }  
      
    /\* The column hash function calculates a hash based on a column's values.  
     \* The hash is a pseudo-random number which must be the same for any two  
     \* identical columns. In this game of Solitaire we consider two columns  
     \* of cards similar if they contain the same cards in the same order and  
     \* have the same 'cheat' flag.  
     \*  
     \* The value of the hash will be:  
     \* (cheated << 31) | (((33+card\_0)\*33+card\_1)\*33+...)\*33+card\_N mod (MODULUS)  
     \* where card\_n is equivalent to column->cards[n] and card\_N = the last card  
     \* in the column.  
     \*/  
    uint32 hash\_column(Column\* column) {  
     uint32 rv = 1;  
     for (uint8 card = 0; card < column->count; card++) {  
     rv = ((uint64)rv \* (uint64)33 + column->cards[card]) % MODULUS;  
     }  
     return rv | (uint32)(column->cheated << 31);  
    }
  + hash.h (click to expand)
  + #include "types.h"  
      
    uint32 powmod(uint32 y, uint32 x);  
    uint32 hash(Board\* board);  
    uint32 hash\_column(Column\* column);
  + memory.c (click to expand)
  + #include "memory.h"  
      
    /\* Allocate a board and its columns  
     \* Needs freeing after use  
     \*/  
    Board\* allocate\_board() {  
     Board\* board = malloc(sizeof(Board));  
     board->depth = 0;  
     for (int i = 0; i < 6; i++) {  
     board->cols[i] = malloc(sizeof(Column));  
     }  
     return board;  
    }  
      
    /\* Free a board and its columns  
     \*/  
    void free\_board(Board\* board) {  
     for (int i = 0; i < 6; i++) {  
     free(board->cols[i]);  
     }  
     free(board);  
    }  
      
    /\* Allocate a BoardHashTable\_LLNode  
     \*/  
    BoardHashTable\_LLNode\* create\_node(string board\_state, int depth) {  
     BoardHashTable\_LLNode\* node = malloc(sizeof(BoardHashTable\_LLNode));  
     node->board\_state = board\_state;  
     node->unsolvable\_in = depth;  
     node->next = NULL;  
     return node;  
    }
  + memory.h (click to expand)
  + #include "types.h"  
      
    Board\* allocate\_board();  
    void free\_board(Board\* board);  
    BoardHashTable\_LLNode\* create\_node(string board\_state, int depth);
  + parse.c (click to expand)
  + #include "parse.h"  
      
    /\* Given a board represented as a string, generate a board  
     \* object and populate its values.  
     \*  
     \* Board data is expected to be in the format  
     \* /([TKDV06-9]{0,14}(\.|!)){6}/ (given as a regex)  
     \*  
     \* T, K, D, V, 9, 8, 7, and 6 represent themselves, and 0  
     \* represents 10  
     \*  
     \* The board needs freeing after use.  
     \*  
     \* TODO: avoid segfault for bad input  
     \*/  
    Board\* parse\_input(string given\_state) {  
     Board\* board = allocate\_board();  
     int card = 0;  
     for (int i = 0; i < 6; i++) {  
     Column\* this\_column = board->cols[i];  
     uint8 current\_card;  
     uint8 y = 0;  
     while ((current\_card = given\_state[card++]) != '.' && current\_card != '!') {  
     switch (current\_card) {  
     case 'T':  
     current\_card = 14;  
     break;  
     case 'K':  
     current\_card = 13;  
     break;  
     case 'D':  
     current\_card = 12;  
     break;  
     case 'V':  
     current\_card = 11;  
     break;  
     case '0':  
     current\_card = 10;  
     break;  
     default:  
     current\_card -= '0';  
     }  
     this\_column->cards[y++] = current\_card;  
     }  
     this\_column->count = y;  
     this\_column->cheated = current\_card == '!';  
     this\_column->stack\_begin = stack\_begin(this\_column);  
     }  
     return board;  
    }
  + parse.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "memory.h"  
    #include "types.h"  
      
    Board\* parse\_input(string given\_state);
  + solver.c (click to expand)
  + #include "solver.h"  
      
    Move\* moves[1024];  
      
    /\* Search recursively for the next move; backtracking.  
     \* Return -1 if no valid moves found, non-negative to  
     \* indicate number of moves.  
     \*  
     \* Cheating should now be implemented  
     \*/  
    int step(Board\* board, int depth, int max\_moves, bool allow\_cheat) {  
     if (depth == max\_moves) {  
     return -1;  
     }  
     Move\* move = malloc(sizeof(Move));  
     Move\* cheat\_step\_one;  
     Move\* cheat\_step\_two;  
     int two\_move\_cheat\_col;  
     for (move->from\_x = 0; move->from\_x < 6; move->from\_x++) {  
     for (move->to\_x = 0; move->to\_x < 6; move->to\_x++) {  
     if (move->from\_x == move->to\_x) {  
     continue;  
     }  
     if (board->cols[move->from\_x]->count == 0) {  
     continue;  
     }  
     move->from\_y = board->cols[move->from\_x]->stack\_begin;  
     move->to\_y = board->cols[move->to\_x]->count;  
     moves[depth] = move;  
     if (!can\_move(board, move)) {  
     continue;  
     }  
     if (move->is\_cheat) {  
     if (!allow\_cheat) {  
     continue;  
     }  
     two\_move\_cheat\_col = column\_if\_two\_moves(board, move);  
     if (two\_move\_cheat\_col != -1) {  
     cheat\_step\_one = malloc(sizeof(Move));  
     cheat\_step\_two = malloc(sizeof(Move));  
     cheat\_step\_one->from\_x = two\_move\_cheat\_col;  
     cheat\_step\_one->from\_y = board->cols[two\_move\_cheat\_col]->count - 1;  
     cheat\_step\_one->to\_x = move->to\_x;  
     cheat\_step\_one->to\_y = move->to\_y;  
     cheat\_step\_one->is\_cheat = true;  
     cheat\_step\_two->from\_x = move->from\_x;  
     cheat\_step\_two->from\_y = move->from\_y;  
     cheat\_step\_two->to\_x = two\_move\_cheat\_col;  
     cheat\_step\_two->to\_y = board->cols[two\_move\_cheat\_col]->count - 1;  
     cheat\_step\_two->is\_cheat = false;  
     moves[depth++] = cheat\_step\_one;  
     moves[depth] = cheat\_step\_two;  
     }  
     } else {  
     two\_move\_cheat\_col = -1;  
     }  
     apply\_move(board, move);  
     if (solved(board)) {  
     return depth + 1;  
     }  
     int found = step(board, depth + 1, max\_moves, allow\_cheat);  
     if (found != -1) {  
     return found;  
     }  
     unapply\_move(board, move);  
     if (move->is\_cheat && two\_move\_cheat\_col != -1) {  
     depth--;  
     free(cheat\_step\_one);  
     free(cheat\_step\_two);  
     }  
     }  
     }  
     free(move);  
     return -1;  
    }  
      
    int main(int argc, string argv[]) {  
     if (argc < 2) {  
     println("Need starting board state");  
     return 1;  
     }  
     Board\* board = parse\_input(argv[1]);  
     struct arguments arguments;  
      
     arguments.cheat = false;  
     argp\_parse(&argp, argc, argv, 0, 0, &arguments);  
      
     for (int max\_depth = 64; max\_depth <= 1024; max\_depth <<= 1) {  
     int n\_moves = step(board, 0, max\_depth, false);  
     if (n\_moves == -1) {  
     eprintfln("No solution found in %d moves", max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d %c> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->is\_cheat ? '!' : '-',  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     }  
     return 0;  
     }  
     }  
     if (arguments.cheat) {  
     for (int max\_depth = 64; max\_depth <= 1024; max\_depth <<= 1) {  
     int n\_moves = step(board, 0, max\_depth, true);  
     if (n\_moves == -1) {  
     eprintfln("No solution found in %d moves (Cheating)", max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d %c> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->is\_cheat ? '!' : '-',  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     }  
     return 0;  
     }  
     }  
     }  
     return 0;  
    }
  + solver.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    // #include "debug.h"  
    #include "parse.h"  
    #include "types.h"  
      
    #include <argp.h>  
      
    const char\* argp\_program\_version = "molek-syntez-solitaire-solver";  
    const char\* argp\_program\_bug\_address = "";  
    static char doc[] = "Solver for MOLEK-SYNTEZ solitaire minigame";  
    static char args\_doc[] = "<STATE>";  
    static struct argp\_option options[] = {  
     {"cheat",  
     'c',  
     0,  
     0,  
     "Allow Cheating (only takes effect if "  
     "no solution is found without Cheating)"},  
     {0}};  
      
    struct arguments {  
     bool cheat;  
    };  
      
    static error\_t parse\_opt(int key, char\* arg, struct argp\_state\* state) {  
     struct arguments\* arguments = state->input;  
     switch (key) {  
     case 'c':  
     arguments->cheat = true;  
     break;  
     case ARGP\_KEY\_ARG:  
     return 0;  
     default:  
     return ARGP\_ERR\_UNKNOWN;  
     }  
     return 0;  
    }  
      
    static struct argp argp = {options, parse\_opt, args\_doc, doc, 0, 0, 0};  
      
    int step(Board\* board, int depth, int max\_moves, bool allow\_cheat);
  + types.h (click to expand)
  + #pragma once  
    #include "common.h"  
      
    typedef uint8 Card;  
      
    typedef struct \_Column {  
     Card cards[14];  
     uint8 count;  
     uint8 stack\_begin;  
     bool cheated;  
    } Column;  
      
    typedef struct \_Board {  
     Column\* cols[6]; // Board.cols[x][y]  
     int depth;  
    } Board;  
      
    typedef struct \_Move {  
     uint8 from\_x;  
     uint8 from\_y;  
     uint8 to\_x;  
     uint8 to\_y;  
     bool is\_cheat;  
    } Move;  
      
    typedef struct \_BoardHashTable\_LLNode {  
     string board\_state;  
     int unsolvable\_in;  
     struct \_BoardHashTable\_LLNode\* next;  
    } BoardHashTable\_LLNode;

### Caching and solution optimisation timer [2021-01-26T12:53Z]

* This version of the solver cleans up some naming and compiles with stricter settings, as well as adding caching, fixing some memory-related bugs, and a timer during which the solver will attempt to optimise its solution
* Additionally, the solver will always try every move which doesn’t Cheat for a step before it will try any move which Cheats (this reduces the number of Cheated cards at any given time, and as many board states are possible without Cheating, it allows the solution to be found more quickly)
* Code:
  + board\_info.c (click to expand)
  + #include "board\_info.h"  
      
    /\* Determine if a stack is legal  
     \*/  
    bool legal\_stack(const Column\* col, uint8 position) {  
     uint8 value = col->cards[position];  
     for (uint8 y = position; y < col->count; y++) {  
     if (col->cards[y] != value--) {  
     return false;  
     }  
     }  
     return true;  
    }  
      
    /\* Determine where the stack begins  
     \*/  
    uint8 stack\_begin(const Column\* col) {  
     // if there are no cards in the column then there is no stack  
     if (col->count == 0) {  
     return 0;  
     }  
     // if there is a cheated card then that is the entire stack  
     if (col->cheated) {  
     return col->count - 1;  
     }  
     uint8 y = col->count - 1;  
     uint8 card = col->cards[y];  
     while (--y != 255 && col->cards[y] == ++card) {  
     }  
     return y + 1;  
    }  
      
    /\* Check if a given move is legal on a given board  
     \* Sets move->is\_cheat  
     \*/  
    bool can\_move(const Board\* board, Move\* move) {  
     const int from\_col = move->from\_x;  
     const int from\_y = move->from\_y;  
     const int to\_col = move->to\_x;  
     const Column\* fcol = board->cols[from\_col];  
     const Column\* tcol = board->cols[to\_col];  
      
     // cards can never be moved onto a column which contains a Cheated card  
     if (tcol->cheated) {  
     return false;  
     }  
      
     // never move from the beginning of a stack to an empty stack  
     if (fcol->stack\_begin == 0 && tcol->count == 0) {  
     return false;  
     }  
      
     // never move an invalid stack - this could actually be omitted as it's  
     // impossible for such a Move\* to be generated  
     // if (from\_y < fcol->count - 1 && !legal\_stack(fcol, from\_y)) {  
     // return false;  
     // }  
      
     move->is\_cheat = false; // assume not a cheat until we determine it is  
     // if there are no cards in this column then the move is valid  
     if (tcol->count == 0) {  
     return true;  
     }  
      
     // otherwise check the top card and compare it to the card being moved  
     if (tcol->cards[tcol->count - 1] == fcol->cards[from\_y] + 1) {  
     return true;  
     }  
      
     if (from\_y < fcol->count - 1) {  
     // moving a stack; must not Cheat  
     return false;  
     }  
      
     // if we reach here then the attempted move Cheats  
     move->is\_cheat = true;  
     if (fcol->cheated) {  
     // this is only a valid move if fcol->cards[fcol->count - 1] can  
     // also be found in a legitimate position on a column that is neither  
     // fcol nor tcol  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     Column\* col = board->cols[i];  
     // the stack must be more than one card (the card must be  
     // in a valid position)  
     if (col->stack\_begin < col->count - 1  
     // and must end with the card we're trying to move  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return true; // legal move (that is technically 2 moves)  
     }  
     }  
     // not legal; no column permits this move  
     return false;  
     }  
     // Cheating with a card that was previously not Cheated is always valid  
     return true;  
    }  
      
    /\* Determine if move is the special two-move operation Cheat -> Cheat  
     \* If it is, return the column of the intermediate column.  
     \* Else, return -1  
     \*  
     \* Assume that move is a legal Cheat move  
     \*/  
    int column\_if\_two\_moves(const Board\* board, const Move\* move) {  
     const int from\_col = move->from\_x;  
     const int to\_col = move->to\_x;  
     const Column\* fcol = board->cols[from\_col];  
     if (fcol->cheated) {  
     // find the column using the algorithm from can\_move  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     const Column\* col = board->cols[i];  
     if (col->stack\_begin < col->count - 1  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return i; // found the two-move column  
     }  
     }  
     }  
     return -1; // this move cannot be Cheat -> Cheat  
    }  
      
    /\* Determine if the board is solved  
     \*  
     \* The board is solved if all columns are either empty or  
     \* contain a full stack starting from the top of the tableau  
     \* (col->cards == {14, 13, 12, 11, 10, 9, 8, 7, 6} and  
     \* therefore col->count == 9 and col->stack\_begin == 0)  
     \*/  
    bool solved(const Board\* board) {  
     for (int col = 0; col < 6; col++) {  
     if (board->cols[col]->count == 0 // column is empty  
     || (board->cols[col]->count == 9 // or contains the full stack  
     && board->cols[col]->stack\_begin == 0)) {  
     continue;  
     }  
     return false;  
     }  
     return true;  
    }  
      
    /\* Apply move to board.  
     \*  
     \* Does not check validity of move.  
     \*/  
    void apply\_move(Board\* board, const Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     const uint8 n\_elements = from\_col->count - move->from\_y;  
     memcpy(to\_col->cards + move->to\_y, from\_col->cards + move->from\_y, n\_elements);  
     if (move->was\_cheat) {  
     from\_col->cheated = false;  
     }  
     if (move->is\_cheat) {  
     to\_col->cheated = true;  
     to\_col->stack\_begin = to\_col->count;  
     }  
     from\_col->count = move->from\_y;  
     from\_col->stack\_begin = stack\_begin(from\_col);  
     to\_col->count += n\_elements;  
    }  
      
    /\* Unapply move to board; opposite of apply\_move  
     \*/  
    void unapply\_move(Board\* board, const Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     uint8 n\_elements = to\_col->count - move->to\_y;  
     memcpy(from\_col->cards + move->from\_y, to\_col->cards + move->to\_y, n\_elements);  
     from\_col->stack\_begin = from\_col->count;  
     from\_col->count = move->from\_y + n\_elements;  
     to\_col->count = move->to\_y;  
     from\_col->cheated = move->was\_cheat;  
     if (move->is\_cheat) {  
     to\_col->cheated = false;  
     to\_col->stack\_begin = stack\_begin(to\_col);  
     }  
    }
  + board\_info.h (click to expand)
  + #pragma once  
    #include "types.h"  
      
    bool legal\_stack(const Column\* col, uint8 position);  
    uint8 stack\_begin(const Column\* col);  
    bool can\_move(const Board\* board, Move\* move);  
    int column\_if\_two\_moves(const Board\* board, const Move\* move);  
    bool solved(const Board\* board);  
    void apply\_move(Board\* board, const Move\* move);  
    void unapply\_move(Board\* board, const Move\* move);
  + compress.c (click to expand)
  + #include "compress.h"  
      
    /\* Compress a board state to a board state string  
     \* Return string needs freeing  
     \*/  
    string compress(const Board\* board) {  
     // board state string is always 42 characters: 36 cards + 6 columns  
     string out = malloc(42 \* sizeof(char));  
     uint8 idx = 0;  
     for (int col = 0; col < 6; col++) {  
     const Column\* column = board->cols[col];  
     for (int card = 0; card < column->count; card++) {  
     switch (column->cards[card]) {  
     case 14:  
     out[idx++] = 'T';  
     break;  
     case 13:  
     out[idx++] = 'K';  
     break;  
     case 12:  
     out[idx++] = 'D';  
     break;  
     case 11:  
     out[idx++] = 'V';  
     break;  
     case 10:  
     out[idx++] = '0';  
     break;  
     default:  
     out[idx++] = column->cards[card] + '0';  
     break;  
     }  
     }  
     out[idx++] = column->cheated ? '!' : '.';  
     }  
     return out;  
    }  
      
    bool \_compare(const Column\* col, string\* compressed) {  
     if ((\*compressed)[col->count] != (col->cheated ? '!' : '.')) {  
     return false;  
     }  
     int card;  
     for (card = 0; card < col->count; card++) {  
     switch ((\*compressed)[card]) {  
     case '.':  
     case '!':  
     return false;  
     case 'T':  
     if (col->cards[card] != 14) {  
     return false;  
     }  
     break;  
     case 'K':  
     if (col->cards[card] != 13) {  
     return false;  
     }  
     break;  
     case 'D':  
     if (col->cards[card] != 12) {  
     return false;  
     }  
     break;  
     case 'V':  
     if (col->cards[card] != 11) {  
     return false;  
     }  
     break;  
     case '0':  
     if (col->cards[card] != 10) {  
     return false;  
     }  
     break;  
     default:  
     if (col->cards[card] != ((\*compressed)[card] - '0')) {  
     return false;  
     }  
     break;  
     }  
     }  
     \*compressed += card + 1;  
     return true;  
    }  
      
    /\* Compare a board state to a compressed board string.  
     \* Column order does not matter and so is ignored.  
     \*/  
    bool equal(const Board\* board, const string compressed) {  
     string consumable = malloc(42);  
     memcpy(consumable, compressed, 42);  
     bool matched[6] = {false, false, false, false, false, false};  
     for (int col\_compressed = 0; col\_compressed < 6; col\_compressed++) {  
     bool found\_match = false;  
     for (int col\_real = 0; col\_real < 6; col\_real++) {  
     if (matched[col\_real]) {  
     continue;  
     }  
     if (\_compare(board->cols[col\_real], &consumable)) {  
     matched[col\_real] = found\_match = true;  
     break;  
     }  
     }  
     if (!found\_match) {  
     return false;  
     }  
     }  
     return true;  
    }
  + compress.h (click to expand)
  + #include "types.h"  
      
    string compress(const Board\* board);  
    bool equal(const Board\* board, const string compressed);
  + hash.c (click to expand)
  + #include "hash.h"  
      
    /\* The MODULUS and GENERATOR are used in the hash function to generate a  
     \* pseudo-random number based on each board position. The MODULUS is a  
     \* prime number with enough bits to fill the hash table, and the GENERATOR  
     \* is a primitive root with respect to that modulus. Conveniently, 2^31-1  
     \* is a prime number, and the generator here is that used in the MINSTD  
     \* Lehmer RNG and is equal to 7^5.  
     \*/  
    #define MODULUS 2147483647  
    #define GENERATOR 16807  
      
    /\* The powmod function calculates (y^x) mod (MODULUS) and is used in the  
     \* hash function. We are assuming that uint32 is long enough to hold the  
     \* modulus and uint64 is long enough to hold two longs multiplied  
     \* together.  
     \*  
     \* Square-and-multiply method. Check each bit of x in turn, square the  
     \* answer so far and multiply by y if the bit was a 1.  
     \*/  
    uint32 powmod(uint32 y, uint32 x) {  
     uint32 answer = 1;  
     int i;  
     for (i = 0; i < 32; i++) {  
     if (i > 0 && answer > 1) {  
     answer = ((uint64)answer \* (uint64)answer) % MODULUS;  
     }  
     if (x & 0x80000000) {  
     answer = ((uint64)answer \* (uint64)y) % MODULUS;  
     }  
     x <<= 1;  
     }  
     return answer;  
    }  
      
    /\* The hash function calculates a hash based on the current board position.  
     \* The hash is a pseudo-random number which must be the same for any  
     \* two identical board positions. In this game of Solitaire we consider  
     \* two board positions similar if they contain the same columns of cards  
     \* in any order, so the hash function is invariant when columns are swapped.  
     \*  
     \* The value of the hash will be (((a^col\_1)^col\_2)^...)^col\_N mod (MODULUS) and  
     \* then further modified by the cell card if any; where a is the RNG generator  
     \* and col\_1...col\_N are hash values computed from the card columns. This  
     \* value is equivalent to a^(col\_1\*col\_2\*...\*col\_N) and therefore the order of  
     \* the columns is not significant. Each column hash must not be zero, and  
     \* so will be 1 if the column is empty.  
     \*/  
    uint32 hash(const Board\* board) {  
     uint32 rv = GENERATOR;  
     for (uint8 col = 0; col < 6; col++) {  
     rv = powmod(rv, hash\_column(board->cols[col]));  
     }  
     return rv;  
    }  
      
    /\* The column hash function calculates a hash based on a column's values.  
     \* The hash is a pseudo-random number which must be the same for any two  
     \* identical columns. In this game of Solitaire we consider two columns  
     \* of cards similar if they contain the same cards in the same order and  
     \* have the same 'cheat' flag.  
     \*  
     \* The value of the hash will be:  
     \* (cheated << 31) | (((33+card\_0)\*33+card\_1)\*33+...)\*33+card\_N mod (MODULUS)  
     \* where card\_n is equivalent to column->cards[n] and card\_N = the last card  
     \* in the column.  
     \*/  
    uint32 hash\_column(const Column\* column) {  
     uint32 rv = 1;  
     for (uint8 card = 0; card < column->count; card++) {  
     rv = ((uint64)rv \* (uint64)33 + column->cards[card]) % MODULUS;  
     }  
     return rv | (uint32)(column->cheated << 31);  
    }
  + hash.h (click to expand)
  + #include "types.h"  
      
    uint32 powmod(uint32 y, uint32 x);  
    uint32 hash(const Board\* board);  
    uint32 hash\_column(const Column\* column);
  + memory.c (click to expand)
  + #include "memory.h"  
      
    /\* Allocate a board and its columns  
     \* Needs freeing after use  
     \*/  
    Board\* allocate\_board() {  
     Board\* board = malloc(sizeof(Board));  
     for (int i = 0; i < 6; i++) {  
     board->cols[i] = malloc(sizeof(Column));  
     }  
     return board;  
    }  
      
    /\* Free a board and its columns  
     \*/  
    void free\_board(Board\* board) {  
     for (int i = 0; i < 6; i++) {  
     free(board->cols[i]);  
     }  
     free(board);  
    }  
      
    /\* Allocate a BoardHashTable\_LLNode  
     \*/  
    BoardHashTable\_LLNode\* create\_node(const string board\_state, int moves) {  
     BoardHashTable\_LLNode\* node = malloc(sizeof(BoardHashTable\_LLNode));  
     node->board\_state = board\_state;  
     node->unsolvable\_in = moves;  
     node->next = NULL;  
     return node;  
    }  
      
    /\* Completely deallocate a BoardHashTable linked list  
     \*  
     \* Should get a Tail Recursion Optimisation  
     \*/  
    void deallocate\_list(BoardHashTable\_LLNode\* node) {  
     if (node == NULL) {  
     return;  
     }  
     BoardHashTable\_LLNode\* next = (BoardHashTable\_LLNode\*)node->next;  
     free((void\*)node->board\_state);  
     free(node);  
     deallocate\_list(next);  
    }  
      
    /\* Copy a move  
     \*  
     \* Returned move needs freeing  
     \*/  
    Move\* copy\_move(Move\* move) {  
     Move\* ret = malloc(sizeof(Move));  
     memcpy(ret, move, sizeof(Move));  
     return ret;  
    }
  + memory.h (click to expand)
  + #include "types.h"  
      
    Board\* allocate\_board();  
    void free\_board(Board\* board);  
    BoardHashTable\_LLNode\* create\_node(const string board\_state, int moves);  
    void deallocate\_list(BoardHashTable\_LLNode\* node);  
    Move\* copy\_move(Move\* move);
  + parse.c (click to expand)
  + #include "parse.h"  
      
    /\* Given a board represented as a string, generate a board  
     \* object and populate its values.  
     \*  
     \* Board data is expected to be in the format  
     \* /([TKDV06-9]{0,14}(\.|!)){6}/ (given as a regex)  
     \*  
     \* T, K, D, V, 9, 8, 7, and 6 represent themselves, and 0  
     \* represents 10  
     \*  
     \* The board needs freeing after use.  
     \*  
     \* TODO: avoid segfault for bad input  
     \*/  
    Board\* parse\_input(const string given\_state) {  
     Board\* board = allocate\_board();  
     int card = 0;  
     for (int i = 0; i < 6; i++) {  
     Column\* this\_column = board->cols[i];  
     uint8 current\_card;  
     uint8 y = 0;  
     while ((current\_card = given\_state[card++]) != '.' && current\_card != '!') {  
     switch (current\_card) {  
     case 'T':  
     current\_card = 14;  
     break;  
     case 'K':  
     current\_card = 13;  
     break;  
     case 'D':  
     current\_card = 12;  
     break;  
     case 'V':  
     current\_card = 11;  
     break;  
     case '0':  
     current\_card = 10;  
     break;  
     default:  
     current\_card -= '0';  
     }  
     this\_column->cards[y++] = current\_card;  
     }  
     this\_column->count = y;  
     this\_column->cheated = current\_card == '!';  
     this\_column->stack\_begin = stack\_begin(this\_column);  
     }  
     return board;  
    }
  + parse.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "memory.h"  
    #include "types.h"  
      
    Board\* parse\_input(const string given\_state);
  + solver.c (click to expand)
  + #include "solver.h"  
      
    Move\*\* moves;  
    BoardHashTable\_LLNode\* cache[0x1000000] = {NULL};  
    int cache\_boundary = 16;  
    int continue\_millis = 500;  
    int max\_depth = 1024;  
    Time timer = {0, 0};  
      
    /\* Clear the cache. Used between no-Cheat and Cheat passes,  
     \* as well as at the end to prevent memory leaks  
     \*/  
    void clear\_cache() {  
     eprintln("Clearing cache...");  
     for (uint32 cache\_idx = 0; cache\_idx < 0x1000000;) {  
     if (cache[cache\_idx] != NULL) {  
     deallocate\_list(cache[cache\_idx]);  
     cache[cache\_idx] = NULL;  
     }  
     cache\_idx++;  
     }  
     eprintln("Done");  
    }  
      
    /\* Free moves[idx] if required, and return whether moves[idx] was freed \*/  
    bool free\_if\_required(int idx) {  
     if (moves[idx] != NULL) {  
     free(moves[idx]);  
     moves[idx] = NULL;  
     return true;  
     }  
     return false;  
    }  
      
    /\* Free all moves from idx onwards  
     \*/  
    void free\_moves\_from(int idx) {  
     // Loop all elements from idx to the end  
     for (int i = idx; i < max\_depth; i++) {  
     if (!free\_if\_required(i)) {  
     // Return if moves[i] is NULL as all other moves will be NULL  
     return;  
     }  
     }  
    }  
      
    /\* Search recursively for the next move; backtracking.  
     \* Return -1 if no valid moves found, non-negative to  
     \* indicate number of moves.  
     \*  
     \* Cheating should now be implemented  
     \*/  
    int step(Board\* board, int depth, int max\_moves, bool allow\_cheat) {  
     if (depth >= max\_moves) {  
     return -1;  
     }  
     int remaining\_moves = max\_moves - depth;  
     uint32 board\_hash = hash(board) & 0xFFFFFF;  
     BoardHashTable\_LLNode\* node = cache[board\_hash];  
     BoardHashTable\_LLNode\* last\_node = NULL;  
     while (node != NULL && !equal(board, (const string)node->board\_state)) {  
     node = (BoardHashTable\_LLNode\*)(last\_node = node)->next;  
     }  
     if (node && node->unsolvable\_in >= remaining\_moves) {  
     return -1;  
     }  
     // Add the node to the cache unconditionally  
     // This prevents the solver from entering loops  
     bool free\_node = false;  
     if (node) {  
     node->unsolvable\_in = remaining\_moves;  
     } else {  
     free\_node = true;  
     string compressed = compress(board);  
     if (last\_node) {  
     last\_node->next = create\_node(compressed, remaining\_moves);  
     node = (BoardHashTable\_LLNode\*)last\_node->next;  
     } else {  
     node = cache[board\_hash] = create\_node(compressed, remaining\_moves);  
     }  
     }  
     Move\* move = malloc(sizeof(Move));  
     Move\* cheat\_step\_one = malloc(sizeof(Move));  
     Move\* cheat\_step\_two = malloc(sizeof(Move));  
     int two\_move\_cheat\_col;  
     bool found\_solution = false;  
     // we want to start with non-Cheats only as these are more likely to work  
     for (move->from\_x = 0; move->from\_x < 6; move->from\_x++) {  
     for (move->to\_x = 0; move->to\_x < 6; move->to\_x++) {  
     if (move->from\_x == move->to\_x) {  
     continue;  
     }  
     if (board->cols[move->from\_x]->count == 0) {  
     continue;  
     }  
     // If we've run out of time, return; otherwise, keep looking for a shorter  
     // solution  
     if (expired(timer)) {  
     goto finalise;  
     }  
     move->from\_y = board->cols[move->from\_x]->stack\_begin;  
     move->to\_y = board->cols[move->to\_x]->count;  
     move->was\_cheat = board->cols[move->from\_x]->cheated;  
     if (!can\_move(board, move)) {  
     continue;  
     }  
     if (move->is\_cheat) {  
     continue;  
     }  
     apply\_move(board, move);  
     if (solved(board)) {  
     // Solving the board this move is the best I can do; start a timer and  
     // return  
     found\_solution = true;  
     free\_moves\_from(depth);  
     moves[depth] = copy\_move(move);  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     max\_moves = depth + 1;  
     unapply\_move(board, move);  
     goto finalise;  
     }  
     int found = step(board, depth + 1, max\_moves, allow\_cheat);  
     if (found != -1) {  
     // Found a solution in fewer moves than before  
     found\_solution = true;  
     free\_if\_required(depth);  
     moves[depth] = copy\_move(move);  
     max\_moves = found;  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     }  
     unapply\_move(board, move);  
     }  
     }  
     if (allow\_cheat) {  
     // we couldn't solve it without Cheating here; try Cheating now  
     for (move->from\_x = 0; move->from\_x < 6; move->from\_x++) {  
     for (move->to\_x = 0; move->to\_x < 6; move->to\_x++) {  
     if (move->from\_x == move->to\_x) {  
     continue;  
     }  
     if (board->cols[move->from\_x]->count == 0) {  
     continue;  
     }  
     // If we've run out of time, return; otherwise, keep looking for a  
     // shorter solution  
     if (expired(timer)) {  
     goto finalise;  
     }  
     move->from\_y = board->cols[move->from\_x]->stack\_begin;  
     move->to\_y = board->cols[move->to\_x]->count;  
     move->was\_cheat = board->cols[move->from\_x]->cheated;  
     if (!can\_move(board, move)) {  
     continue;  
     }  
     if (!move->is\_cheat) {  
     continue;  
     }  
     two\_move\_cheat\_col = column\_if\_two\_moves(board, move);  
     const int two\_move\_col = two\_move\_cheat\_col;  
     const bool is\_two\_move = two\_move\_col != -1;  
     if (is\_two\_move) {  
     // Set up the two steps using the data from move and two\_move\_col  
     cheat\_step\_one->from\_x = two\_move\_col;  
     cheat\_step\_one->from\_y = board->cols[two\_move\_col]->count - 1;  
     cheat\_step\_one->to\_x = move->to\_x;  
     cheat\_step\_one->to\_y = move->to\_y;  
     cheat\_step\_one->is\_cheat = true;  
     cheat\_step\_two->from\_x = move->from\_x;  
     cheat\_step\_two->from\_y = move->from\_y;  
     cheat\_step\_two->to\_x = two\_move\_col;  
     cheat\_step\_two->to\_y = board->cols[two\_move\_col]->count - 1;  
     cheat\_step\_two->is\_cheat = false;  
      
     // Remember that this is a two-step Cheat in order to  
     // finalise correctly  
     depth++;  
     }  
     apply\_move(board, move);  
     if (solved(board)) {  
     // Solving the board this move is the best I can do; start a timer  
     // and return  
     found\_solution = true;  
     free\_moves\_from(depth);  
     if (is\_two\_move) {  
     free\_if\_required(depth - 1);  
     moves[depth - 1] = copy\_move(cheat\_step\_one);  
     moves[depth] = copy\_move(cheat\_step\_two);  
     } else {  
     moves[depth] = copy\_move(move);  
     }  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     max\_moves = depth + 1;  
     unapply\_move(board, move);  
     goto finalise;  
     }  
     int found = step(board, depth + 1, max\_moves, allow\_cheat);  
     if (found != -1) {  
     // Found a solution in fewer moves than before  
     found\_solution = true;  
     free\_if\_required(depth);  
     if (is\_two\_move) {  
     free\_if\_required(depth - 1);  
     moves[depth - 1] = copy\_move(cheat\_step\_one);  
     moves[depth] = copy\_move(cheat\_step\_two);  
     } else {  
     moves[depth] = copy\_move(move);  
     // Found a solution better than any found double-Cheat  
     // so remember that this ISN'T a double-Cheat  
     }  
     max\_moves = found;  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     }  
     unapply\_move(board, move);  
     if (is\_two\_move) {  
     // Decrement depth to return to the correct place  
     depth--;  
     }  
     }  
     }  
     }  
    finalise:  
     // Cut the node out if I had to allocate it and we're not within the minimum cache  
     // boundary, or if I've proven that it's possible to solve the board  
     if (found\_solution  
     || (free\_node && cache\_boundary != -1 && remaining\_moves > cache\_boundary)) {  
     if (last\_node) {  
     last\_node->next = node->next;  
     } else {  
     cache[board\_hash] = (BoardHashTable\_LLNode\*)node->next;  
     }  
     free((void\*)node->board\_state);  
     free(node);  
     }  
     // Free the loop variables to avoid a memory leak  
     free(move);  
     free(cheat\_step\_one);  
     free(cheat\_step\_two);  
     // If I found a solution, return the number of moves it took  
     if (found\_solution) {  
     return max\_moves;  
     }  
     return -1;  
    }  
      
    int main(int argc, string argv[]) {  
     extern string optarg;  
     extern int optind, optopt;  
     bool solver\_allow\_cheat = false;  
      
     int opt;  
     while ((opt = getopt(argc, argv, ":cn:t:m:")) != -1) {  
     switch (opt) {  
     case 'c':  
     solver\_allow\_cheat = true;  
     break;  
     case 'n':  
     cache\_boundary = atoi(optarg);  
     break;  
     case 't':  
     continue\_millis = atoi(optarg);  
     break;  
     case 'm':  
     max\_depth = atoi(optarg);  
     break;  
     default:  
     eprintfln(  
     "Usage: %s [-c] [-n <moves>=16] [-t <milliseconds>=500] [-m <max "  
     "moves>=1024] <STATE>",  
     argv[0]);  
     eprintln(" -c Allow Cheating if no solution is found "  
     "without Cheating");  
     eprintln(" -n <cache moves> How many moves may remain when deciding "  
     "whether to cache or not. -1 to cache all moves (not "  
     "recommended).");  
     eprintln(" -t <milliseconds> How long to continue searching for a "  
     "shorter solution after one is found. -1 to search all moves");  
     eprintln(" -m <max moves> How many moves to allow in total. Should "  
     "be a power of 2 as other numbers will be truncated to the "  
     "previous power of 2");  
     exit(EXIT\_FAILURE);  
     }  
     }  
     if (!argv[optind]) {  
     println("Need starting board state");  
     exit(EXIT\_FAILURE);  
     }  
     Board\* board = parse\_input(argv[optind]);  
      
     int start\_max\_depth = min(64, max\_depth);  
     moves = calloc(max\_depth, sizeof(Move\*));  
      
     for (int step\_max\_depth = start\_max\_depth; step\_max\_depth <= max\_depth;  
     step\_max\_depth <<= 1) {  
     int n\_moves = step(board, 0, step\_max\_depth, false);  
     if (n\_moves == -1) {  
     eprintfln("No solution found in %d moves", step\_max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d %c> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->is\_cheat ? '!' : '-',  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     free(moves[i]);  
     }  
     free\_board(board);  
     clear\_cache();  
     return 0;  
     }  
     }  
     if (solver\_allow\_cheat) {  
     clear\_cache();  
     for (int step\_max\_depth = start\_max\_depth; step\_max\_depth <= max\_depth;  
     max\_depth <<= 1) {  
     int n\_moves = step(board, 0, step\_max\_depth, true);  
     if (n\_moves == -1) {  
     eprintfln("No solution found in %d moves (Cheating)", step\_max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d %c> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->is\_cheat ? '!' : '-',  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     free(moves[i]);  
     }  
     free\_board(board);  
     clear\_cache();  
     return 0;  
     }  
     }  
     }  
     free\_board(board);  
     clear\_cache();  
     return 0;  
    }
  + solver.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "compress.h"  
    #include "hash.h"  
    #include "memory.h"  
    #include "parse.h"  
    #include "timer.h"  
    #include "types.h"  
    #ifdef DEBUG  
     #include "debug.h"  
    #endif  
      
    #include <unistd.h>  
      
    int step(Board\* board, int depth, int max\_moves, bool allow\_cheat);
  + timer.c (click to expand)
  + #include "timer.h"  
      
    /\* Create a timer with extra\_milliseconds on the clock  
     \*/  
    Time make\_timer(int extra\_milliseconds) {  
     // If extra\_milliseconds == -1 then the user has disabled the timer (so don't start  
     // one)  
     if (extra\_milliseconds == -1) {  
     return (Time) {0, 0};  
     }  
     Time now;  
     gettimeofday(&now, NULL);  
     now.tv\_usec += 1000 \* extra\_milliseconds;  
     now.tv\_sec += now.tv\_usec / 1000000;  
     now.tv\_usec %= 1000000;  
     return now;  
    }  
      
    /\* Check if timer has expired  
     \*/  
    bool expired(Time timer) {  
     // timer has not been started; it is not expired  
     if (timer.tv\_sec == 0) {  
     return false;  
     }  
     Time now;  
     gettimeofday(&now, NULL);  
     return now.tv\_sec > timer.tv\_sec  
     || (now.tv\_sec == timer.tv\_sec && now.tv\_usec > timer.tv\_usec);  
    }
  + timer.h (click to expand)
  + #pragma once  
    #include "common.h"  
      
    #include <sys/time.h>  
      
    #define start(timer, time) \  
     if (!timer.tv\_sec) { \  
     timer = make\_timer(time); \  
     }  
      
    typedef struct timeval Time;  
      
    Time make\_timer(int extra\_milliseconds);  
    bool expired(Time timer);
  + types.h (click to expand)
  + #pragma once  
    #include "common.h"  
      
    typedef uint8 Card;  
      
    typedef struct \_Column {  
     // largest possible stack:  
     // XXXXXTKDV09876C (15)  
     Card cards[15];  
     uint8 count;  
     uint8 stack\_begin;  
     bool cheated;  
    } Column;  
      
    typedef struct \_Board {  
     Column\* cols[6]; // Board.cols[x][y]  
    } Board;  
      
    typedef struct \_Move {  
     uint8 from\_x;  
     uint8 from\_y;  
     uint8 to\_x;  
     uint8 to\_y;  
     bool is\_cheat;  
     bool was\_cheat;  
    } Move;  
      
    typedef struct \_BoardHashTable\_LLNode {  
     const char\* board\_state;  
     int unsolvable\_in;  
     const struct \_BoardHashTable\_LLNode\* next;  
    } BoardHashTable\_LLNode;

### Input validation and bugfixes [2021-02-19T16:28Z]

* This version of the solver was created after multiple incremental improvements
* It adds a lot of input validation (specifically to the given board state), in addition to preventing the solver from performing some illegal moves (notably, attempting to Cheat cards onto complete stacks, which is an illegal move as when a column is completed it is locked)
* Code:
  + board\_info.c (click to expand)
  + #include "board\_info.h"  
      
    /\* Determine if a stack is legal  
     \*/  
    bool legal\_stack(const Column\* col, uint8 position) {  
     uint8 value = col->cards[position];  
     for (uint8 y = position; y < col->count; y++) {  
     if (col->cards[y] != value--) {  
     return false;  
     }  
     }  
     return true;  
    }  
      
    /\* Determine where the stack begins  
     \*/  
    uint8 stack\_begin(const Column\* col) {  
     // if there are no cards in the column then there is no stack  
     if (col->count == 0) {  
     return 0;  
     }  
     // if there is a cheated card then that is the entire stack  
     if (col->cheated) {  
     return col->count - 1;  
     }  
     uint8 y = col->count - 1;  
     uint8 card = col->cards[y];  
     while (--y != 255 && col->cards[y] == ++card) {  
     }  
     return y + 1;  
    }  
      
    /\* Check if a given move is legal on a given board  
     \* Sets move->is\_cheat  
     \*/  
    bool can\_move(const Board\* board, Move\* move) {  
     const int from\_col = move->from\_x;  
     const int from\_y = move->from\_y;  
     const int to\_col = move->to\_x;  
     const Column\* fcol = board->cols[from\_col];  
     const Column\* tcol = board->cols[to\_col];  
      
     // cards can never be moved onto a column which contains a Cheated card  
     if (tcol->cheated) {  
     return false;  
     }  
      
     // never move from the beginning of a stack to an empty stack  
     if (fcol->stack\_begin == 0 && tcol->count == 0) {  
     return false;  
     }  
      
     // never move an invalid stack - this could actually be omitted as it's  
     // impossible for such a Move\* to be generated  
     // if (from\_y < fcol->count - 1 && !legal\_stack(fcol, from\_y)) {  
     // return false;  
     // }  
      
     move->is\_cheat = false; // assume not a cheat until we determine it is  
     // if there are no cards in this column then the move is valid  
     if (tcol->count == 0) {  
     return true;  
     }  
      
     // otherwise check the top card and compare it to the card being moved  
     if (tcol->cards[tcol->count - 1] == fcol->cards[from\_y] + 1) {  
     return true;  
     }  
      
     if (from\_y < fcol->count - 1) {  
     // moving a stack; must not Cheat  
     return false;  
     }  
      
     // if we reach here then the attempted move Cheats  
     move->is\_cheat = true;  
     // Columns containing a full stack may not be the target of any move,  
     // including a Cheat  
     if (fcol->count == 9 && fcol->stack\_begin == 0) {  
     return false;  
     }  
     if (fcol->cheated) {  
     // this is only a valid move if fcol->cards[fcol->count - 1] can  
     // also be found in a legitimate position on a column that is neither  
     // fcol nor tcol  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     Column\* col = board->cols[i];  
     // the stack must be more than one card (the card must be  
     // in a valid position)  
     if (col->stack\_begin < col->count - 1  
     // and must end with the card we're trying to move  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return true; // legal move (that is technically 2 moves)  
     }  
     }  
     // not legal; no column permits this move  
     return false;  
     }  
     // Cheating with a card that was previously not Cheated is always valid  
     return true;  
    }  
      
    /\* Determine if move is the special two-move operation Cheat -> Cheat  
     \* If it is, return the column of the intermediate column.  
     \* Else, return -1  
     \*  
     \* Assume that move is a legal Cheat move  
     \*/  
    int column\_if\_two\_moves(const Board\* board, const Move\* move) {  
     const int from\_col = move->from\_x;  
     const int to\_col = move->to\_x;  
     const Column\* fcol = board->cols[from\_col];  
     if (fcol->cheated) {  
     // find the column using the algorithm from can\_move  
     for (int i = 0; i < 6; i++) {  
     if (i == from\_col || i == to\_col) {  
     continue;  
     }  
     const Column\* col = board->cols[i];  
     if (col->stack\_begin < col->count - 1  
     && col->cards[col->count - 1] == fcol->cards[fcol->count - 1]) {  
     return i; // found the two-move column  
     }  
     }  
     }  
     return -1; // this move cannot be Cheat -> Cheat  
    }  
      
    /\* Determine if the board is solved  
     \*  
     \* The board is solved if all columns are either empty or  
     \* contain a full stack starting from the top of the tableau  
     \* (col->cards == {14, 13, 12, 11, 10, 9, 8, 7, 6} and  
     \* therefore col->count == 9 and col->stack\_begin == 0)  
     \*/  
    bool solved(const Board\* board) {  
     for (int col = 0; col < 6; col++) {  
     if (board->cols[col]->count == 0 // column is empty  
     || (board->cols[col]->count == 9 // or contains the full stack  
     && board->cols[col]->stack\_begin == 0)) {  
     continue;  
     }  
     return false;  
     }  
     return true;  
    }  
      
    /\* Apply move to board.  
     \*  
     \* Does not check validity of move.  
     \*/  
    void apply\_move(Board\* board, const Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     const uint8 n\_elements = from\_col->count - move->from\_y;  
     memcpy(to\_col->cards + move->to\_y, from\_col->cards + move->from\_y, n\_elements);  
     if (move->was\_cheat) {  
     from\_col->cheated = false;  
     }  
     if (move->is\_cheat) {  
     to\_col->cheated = true;  
     to\_col->stack\_begin = to\_col->count;  
     }  
     from\_col->count = move->from\_y;  
     from\_col->stack\_begin = stack\_begin(from\_col);  
     to\_col->count += n\_elements;  
    }  
      
    /\* Unapply move to board; opposite of apply\_move  
     \*/  
    void unapply\_move(Board\* board, const Move\* move) {  
     Column\* from\_col = board->cols[move->from\_x];  
     Column\* to\_col = board->cols[move->to\_x];  
     uint8 n\_elements = to\_col->count - move->to\_y;  
     memcpy(from\_col->cards + move->from\_y, to\_col->cards + move->to\_y, n\_elements);  
     from\_col->stack\_begin = from\_col->count;  
     from\_col->count = move->from\_y + n\_elements;  
     to\_col->count = move->to\_y;  
     from\_col->cheated = move->was\_cheat;  
     if (move->is\_cheat) {  
     to\_col->cheated = false;  
     to\_col->stack\_begin = stack\_begin(to\_col);  
     }  
    }
  + board\_info.h (click to expand)
  + #pragma once  
    #include "types.h"  
      
    bool legal\_stack(const Column\* col, uint8 position);  
    uint8 stack\_begin(const Column\* col);  
    bool can\_move(const Board\* board, Move\* move);  
    int column\_if\_two\_moves(const Board\* board, const Move\* move);  
    bool solved(const Board\* board);  
    void apply\_move(Board\* board, const Move\* move);  
    void unapply\_move(Board\* board, const Move\* move);
  + compress.c (click to expand)
  + #include "compress.h"  
      
    /\* Compress a board state to a board state string  
     \* Return string needs freeing  
     \*/  
    string compress(const Board\* board) {  
     // board state string is always 42 characters: 36 cards + 6 columns  
     string out = malloc(42 \* sizeof(char));  
     uint8 idx = 0;  
     for (int col = 0; col < 6; col++) {  
     const Column\* column = board->cols[col];  
     for (int card = 0; card < column->count; card++) {  
     switch (column->cards[card]) {  
     case 14:  
     out[idx++] = 'T';  
     break;  
     case 13:  
     out[idx++] = 'K';  
     break;  
     case 12:  
     out[idx++] = 'D';  
     break;  
     case 11:  
     out[idx++] = 'V';  
     break;  
     case 10:  
     out[idx++] = '0';  
     break;  
     default:  
     out[idx++] = column->cards[card] + '0';  
     break;  
     }  
     }  
     out[idx++] = column->cheated ? '!' : '.';  
     }  
     return out;  
    }  
      
    bool \_compare(const Column\* col, string\* compressed) {  
     if ((\*compressed)[col->count] != (col->cheated ? '!' : '.')) {  
     return false;  
     }  
     int card;  
     for (card = 0; card < col->count; card++) {  
     switch ((\*compressed)[card]) {  
     case '.':  
     case '!':  
     return false;  
     case 'T':  
     if (col->cards[card] != 14) {  
     return false;  
     }  
     break;  
     case 'K':  
     if (col->cards[card] != 13) {  
     return false;  
     }  
     break;  
     case 'D':  
     if (col->cards[card] != 12) {  
     return false;  
     }  
     break;  
     case 'V':  
     if (col->cards[card] != 11) {  
     return false;  
     }  
     break;  
     case '0':  
     if (col->cards[card] != 10) {  
     return false;  
     }  
     break;  
     default:  
     if (col->cards[card] != ((\*compressed)[card] - '0')) {  
     return false;  
     }  
     break;  
     }  
     }  
     \*compressed += card + 1;  
     return true;  
    }  
      
    /\* Compare a board state to a compressed board string.  
     \* Column order does not matter and so is ignored.  
     \*/  
    bool equal(const Board\* board, const string compressed) {  
     string consumable = malloc(42);  
     memcpy(consumable, compressed, 42);  
     bool matched[6] = {false, false, false, false, false, false};  
     for (int col\_compressed = 0; col\_compressed < 6; col\_compressed++) {  
     bool found\_match = false;  
     for (int col\_real = 0; col\_real < 6; col\_real++) {  
     if (matched[col\_real]) {  
     continue;  
     }  
     if (\_compare(board->cols[col\_real], &consumable)) {  
     matched[col\_real] = found\_match = true;  
     break;  
     }  
     }  
     if (!found\_match) {  
     return false;  
     }  
     }  
     return true;  
    }
  + compress.h (click to expand)
  + #include "types.h"  
      
    string compress(const Board\* board);  
    bool equal(const Board\* board, const string compressed);
  + hash.c (click to expand)
  + #include "hash.h"  
      
    /\* The MODULUS and GENERATOR are used in the hash function to generate a  
     \* pseudo-random number based on each board position. The MODULUS is a  
     \* prime number with enough bits to fill the hash table, and the GENERATOR  
     \* is a primitive root with respect to that modulus. Conveniently, 2^31-1  
     \* is a prime number, and the generator here is that used in the MINSTD  
     \* Lehmer RNG and is equal to 7^5.  
     \*/  
    #define MODULUS 2147483647  
    #define GENERATOR 16807  
      
    /\* The powmod function calculates (y^x) mod (MODULUS) and is used in the  
     \* hash function. We are assuming that uint32 is long enough to hold the  
     \* modulus and uint64 is long enough to hold two longs multiplied  
     \* together.  
     \*  
     \* Square-and-multiply method. Check each bit of x in turn, square the  
     \* answer so far and multiply by y if the bit was a 1.  
     \*/  
    uint32 powmod(uint32 y, uint32 x) {  
     uint32 answer = 1;  
     int i;  
     for (i = 0; i < 32; i++) {  
     if (i > 0 && answer > 1) {  
     answer = ((uint64)answer \* (uint64)answer) % MODULUS;  
     }  
     if (x & 0x80000000) {  
     answer = ((uint64)answer \* (uint64)y) % MODULUS;  
     }  
     x <<= 1;  
     }  
     return answer;  
    }  
      
    /\* The hash function calculates a hash based on the current board position.  
     \* The hash is a pseudo-random number which must be the same for any  
     \* two identical board positions. In this game of Solitaire we consider  
     \* two board positions similar if they contain the same columns of cards  
     \* in any order, so the hash function is invariant when columns are swapped.  
     \*  
     \* The value of the hash will be (((a^col\_1)^col\_2)^...)^col\_N mod (MODULUS) and  
     \* then further modified by the cell card if any; where a is the RNG generator  
     \* and col\_1...col\_N are hash values computed from the card columns. This  
     \* value is equivalent to a^(col\_1\*col\_2\*...\*col\_N) and therefore the order of  
     \* the columns is not significant. Each column hash must not be zero, and  
     \* so will be 1 if the column is empty.  
     \*/  
    uint32 hash(const Board\* board) {  
     uint32 rv = GENERATOR;  
     for (uint8 col = 0; col < 6; col++) {  
     rv = powmod(rv, hash\_column(board->cols[col]));  
     }  
     return rv;  
    }  
      
    /\* The column hash function calculates a hash based on a column's values.  
     \* The hash is a pseudo-random number which must be the same for any two  
     \* identical columns. In this game of Solitaire we consider two columns  
     \* of cards similar if they contain the same cards in the same order and  
     \* have the same 'cheat' flag.  
     \*  
     \* The value of the hash will be:  
     \* (cheated << 31) | (((33+card\_0)\*33+card\_1)\*33+...)\*33+card\_N mod (MODULUS)  
     \* where card\_n is equivalent to column->cards[n] and card\_N = the last card  
     \* in the column.  
     \*/  
    uint32 hash\_column(const Column\* column) {  
     uint32 rv = 1;  
     for (uint8 card = 0; card < column->count; card++) {  
     rv = ((uint64)rv \* (uint64)33 + column->cards[card]) % MODULUS;  
     }  
     return rv | (uint32)(column->cheated << 31);  
    }
  + hash.h (click to expand)
  + #include "types.h"  
      
    uint32 powmod(uint32 y, uint32 x);  
    uint32 hash(const Board\* board);  
    uint32 hash\_column(const Column\* column);
  + memory.c (click to expand)
  + #include "memory.h"  
      
    /\* Allocate a board and its columns  
     \* Needs freeing after use  
     \*/  
    Board\* allocate\_board() {  
     Board\* board = malloc(sizeof(Board));  
     for (int i = 0; i < 6; i++) {  
     board->cols[i] = malloc(sizeof(Column));  
     }  
     return board;  
    }  
      
    /\* Free a board and its columns  
     \*/  
    void free\_board(Board\* board) {  
     for (int i = 0; i < 6; i++) {  
     free(board->cols[i]);  
     }  
     free(board);  
    }  
      
    /\* Allocate a BoardHashTable\_LLNode  
     \*/  
    BoardHashTable\_LLNode\* create\_node(const string board\_state, int moves) {  
     BoardHashTable\_LLNode\* node = malloc(sizeof(BoardHashTable\_LLNode));  
     node->board\_state = board\_state;  
     node->unsolvable\_in = moves;  
     node->next = NULL;  
     return node;  
    }  
      
    /\* Completely deallocate a BoardHashTable linked list  
     \*  
     \* Should get a Tail Recursion Optimisation  
     \*/  
    void deallocate\_list(BoardHashTable\_LLNode\* node) {  
     if (node == NULL) {  
     return;  
     }  
     BoardHashTable\_LLNode\* next = (BoardHashTable\_LLNode\*)node->next;  
     free((void\*)node->board\_state);  
     free(node);  
     deallocate\_list(next);  
    }  
      
    /\* Copy a move  
     \*  
     \* Returned move needs freeing  
     \*/  
    Move\* copy\_move(Move\* move) {  
     Move\* ret = malloc(sizeof(Move));  
     memcpy(ret, move, sizeof(Move));  
     return ret;  
    }
  + memory.h (click to expand)
  + #include "types.h"  
      
    Board\* allocate\_board();  
    void free\_board(Board\* board);  
    BoardHashTable\_LLNode\* create\_node(const string board\_state, int moves);  
    void deallocate\_list(BoardHashTable\_LLNode\* node);  
    Move\* copy\_move(Move\* move);
  + parse.c (click to expand)
  + #include "parse.h"  
      
    /\* Given a board represented as a string, generate a board  
     \* object and populate its values.  
     \*  
     \* Board data is expected to be in the format  
     \* /([TKDV06-9]{0,15}(\.|!)){6}/ (given as a regex)  
     \*  
     \* T, K, D, V, 9, 8, 7, and 6 represent themselves, and 0  
     \* represents 10  
     \*  
     \* The board needs freeing after use.  
     \*/  
    Board\* parse\_input(const string given\_state) {  
     int num\_cols\_found = 0;  
     for (int i = 0; i < 42; i++) {  
     if (given\_state[i] == '\0') {  
     eprintfln("Bad input string: Expected 42 characters but only got %d", i);  
     exit(EXIT\_FAILURE);  
     }  
     if (given\_state[i] == '.' || given\_state[i] == '!') {  
     num\_cols\_found++;  
     }  
     }  
     if (num\_cols\_found != 6) {  
     eprintfln("Bad input string: Expected 6 columns, got %d", num\_cols\_found);  
     exit(EXIT\_FAILURE);  
     }  
     Board\* board = allocate\_board();  
     int card = 0;  
     int num\_cards\_found = 0;  
     int cards\_found[15] = {0};  
     for (int i = 0; i < 6; i++) {  
     Column\* this\_column = board->cols[i];  
     uint8 current\_card;  
     uint8 y = 0;  
     while ((current\_card = given\_state[card++]) != '.' && current\_card != '!') {  
     uint8 card\_char = current\_card;  
     switch (current\_card) {  
     case 'T':  
     current\_card = 14;  
     break;  
     case 'K':  
     current\_card = 13;  
     break;  
     case 'D':  
     current\_card = 12;  
     break;  
     case 'V':  
     current\_card = 11;  
     break;  
     case '0':  
     current\_card = 10;  
     break;  
     case '9':  
     case '8':  
     case '7':  
     case '6':  
     current\_card -= '0';  
     break;  
     default:  
     eprintfln(  
     "Bad input string: Expected a digit in the range 6-9, '0', "  
     "'T', 'K', 'D', V, '!', or '.'; got '%c'",  
     current\_card);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     cards\_found[current\_card]++;  
     if (cards\_found[current\_card] > 4) {  
     eprintfln(  
     "Bad input string: Impossible state; Found %d %c (expected 4)",  
     cards\_found[current\_card],  
     card\_char);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     if (y > 14) {  
     eprintfln(  
     "Bad input string: Impossible state; Found %d cards in column (max "  
     "15)",  
     y);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     this\_column->cards[y++] = current\_card;  
     num\_cards\_found++;  
     }  
     this\_column->count = y;  
     this\_column->cheated = current\_card == '!';  
     this\_column->stack\_begin = stack\_begin(this\_column);  
     }  
     if (num\_cards\_found != 36) {  
     eprintfln("Bad input string: Expected 36 cards, got %d", num\_cards\_found);  
     free\_board(board);  
     exit(EXIT\_FAILURE);  
     }  
     return board;  
    }
  + parse.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "memory.h"  
    #include "types.h"  
      
    Board\* parse\_input(const string given\_state);
  + solver.c (click to expand)
  + #include "solver.h"  
      
    Move\*\* moves;  
    BoardHashTable\_LLNode\* cache[0x1000000] = {NULL};  
    int cache\_boundary = 16;  
    int continue\_millis = 500;  
    int max\_depth = 1024;  
    Time timer = {0, 0};  
      
    /\* Clear the cache. Used between no-Cheat and Cheat passes,  
     \* as well as at the end to prevent memory leaks  
     \*/  
    void clear\_cache() {  
     eprintln("Clearing cache...");  
     for (uint32 cache\_idx = 0; cache\_idx < 0x1000000;) {  
     if (cache[cache\_idx] != NULL) {  
     deallocate\_list(cache[cache\_idx]);  
     cache[cache\_idx] = NULL;  
     }  
     cache\_idx++;  
     }  
     eprintln("Done");  
    }  
      
    /\* Free moves[idx] if required, and return whether moves[idx] was freed \*/  
    bool free\_if\_required(int idx) {  
     if (moves[idx] != NULL) {  
     free(moves[idx]);  
     moves[idx] = NULL;  
     return true;  
     }  
     return false;  
    }  
      
    /\* Free all moves from idx onwards  
     \*/  
    void free\_moves\_from(int idx) {  
     // Loop all elements from idx to the end  
     for (int i = idx; i < max\_depth; i++) {  
     if (!free\_if\_required(i)) {  
     // Return if moves[i] is NULL as all other moves will be NULL  
     return;  
     }  
     }  
    }  
      
    /\* Search recursively for the next move; backtracking.  
     \* Return -1 if no valid moves found, non-negative to  
     \* indicate number of moves.  
     \*  
     \* Cheating should now be implemented  
     \*/  
    int step(Board\* board, int depth, int max\_moves, bool allow\_cheat) {  
     if (depth >= max\_moves) {  
     return -1;  
     }  
     int remaining\_moves = max\_moves - depth;  
     uint32 board\_hash = hash(board) & 0xFFFFFF;  
     BoardHashTable\_LLNode\* node = cache[board\_hash];  
     BoardHashTable\_LLNode\* last\_node = NULL;  
     while (node != NULL && !equal(board, (const string)node->board\_state)) {  
     node = (BoardHashTable\_LLNode\*)(last\_node = node)->next;  
     }  
     if (node && node->unsolvable\_in >= remaining\_moves) {  
     return -1;  
     }  
     // Add the node to the cache unconditionally  
     // This prevents the solver from entering loops  
     bool free\_node = false;  
     if (node) {  
     node->unsolvable\_in = remaining\_moves;  
     } else {  
     free\_node = true;  
     string compressed = compress(board);  
     if (last\_node) {  
     last\_node->next = create\_node(compressed, remaining\_moves);  
     node = (BoardHashTable\_LLNode\*)last\_node->next;  
     } else {  
     node = cache[board\_hash] = create\_node(compressed, remaining\_moves);  
     }  
     }  
     Move\* move = malloc(sizeof(Move));  
     Move\* cheat\_step\_one = malloc(sizeof(Move));  
     Move\* cheat\_step\_two = malloc(sizeof(Move));  
     int two\_move\_cheat\_col;  
     bool found\_solution = false;  
     // we want to start with non-Cheats only as these are more likely to work  
     for (move->from\_x = 0; move->from\_x < 6; move->from\_x++) {  
     for (move->to\_x = 0; move->to\_x < 6; move->to\_x++) {  
     if (move->from\_x == move->to\_x) {  
     continue;  
     }  
     if (board->cols[move->from\_x]->count == 0) {  
     continue;  
     }  
     // If we've run out of time, return; otherwise, keep looking for a shorter  
     // solution  
     if (expired(timer)) {  
     goto finalise;  
     }  
     move->from\_y = board->cols[move->from\_x]->stack\_begin;  
     move->to\_y = board->cols[move->to\_x]->count;  
     move->was\_cheat = board->cols[move->from\_x]->cheated;  
     if (!can\_move(board, move)) {  
     continue;  
     }  
     if (move->is\_cheat) {  
     continue;  
     }  
     apply\_move(board, move);  
     if (solved(board)) {  
     // Solving the board this move is the best I can do; start a timer and  
     // return  
     found\_solution = true;  
     free\_moves\_from(depth);  
     moves[depth] = copy\_move(move);  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     max\_moves = depth + 1;  
     unapply\_move(board, move);  
     goto finalise;  
     }  
     int found = step(board, depth + 1, max\_moves, allow\_cheat);  
     if (found != -1) {  
     // Found a solution in fewer moves than before  
     found\_solution = true;  
     free\_if\_required(depth);  
     moves[depth] = copy\_move(move);  
     max\_moves = found;  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     }  
     unapply\_move(board, move);  
     }  
     }  
     if (allow\_cheat) {  
     // we couldn't solve it without Cheating here; try Cheating now  
     for (move->from\_x = 0; move->from\_x < 6; move->from\_x++) {  
     for (move->to\_x = 0; move->to\_x < 6; move->to\_x++) {  
     if (move->from\_x == move->to\_x) {  
     continue;  
     }  
     if (board->cols[move->from\_x]->count == 0) {  
     continue;  
     }  
     // If we've run out of time, return; otherwise, keep looking for a  
     // shorter solution  
     if (expired(timer)) {  
     goto finalise;  
     }  
     move->from\_y = board->cols[move->from\_x]->stack\_begin;  
     move->to\_y = board->cols[move->to\_x]->count;  
     move->was\_cheat = board->cols[move->from\_x]->cheated;  
     if (!can\_move(board, move)) {  
     continue;  
     }  
     if (!move->is\_cheat) {  
     continue;  
     }  
     two\_move\_cheat\_col = column\_if\_two\_moves(board, move);  
     const int two\_move\_col = two\_move\_cheat\_col;  
     const bool is\_two\_move = two\_move\_col != -1;  
     if (is\_two\_move) {  
     // Set up the two steps using the data from move and two\_move\_col  
     cheat\_step\_one->from\_x = two\_move\_col;  
     cheat\_step\_one->from\_y = board->cols[two\_move\_col]->count - 1;  
     cheat\_step\_one->to\_x = move->to\_x;  
     cheat\_step\_one->to\_y = move->to\_y;  
     cheat\_step\_one->is\_cheat = true;  
     cheat\_step\_two->from\_x = move->from\_x;  
     cheat\_step\_two->from\_y = move->from\_y;  
     cheat\_step\_two->to\_x = two\_move\_col;  
     cheat\_step\_two->to\_y = board->cols[two\_move\_col]->count - 1;  
     cheat\_step\_two->is\_cheat = false;  
      
     // Remember that this is a two-step Cheat in order to  
     // finalise correctly  
     depth++;  
     }  
     apply\_move(board, move);  
     if (solved(board)) {  
     // Solving the board this move is the best I can do; start a timer  
     // and return  
     found\_solution = true;  
     free\_moves\_from(depth);  
     if (is\_two\_move) {  
     free\_if\_required(depth - 1);  
     moves[depth - 1] = copy\_move(cheat\_step\_one);  
     moves[depth] = copy\_move(cheat\_step\_two);  
     } else {  
     moves[depth] = copy\_move(move);  
     }  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     max\_moves = depth + 1;  
     unapply\_move(board, move);  
     goto finalise;  
     }  
     int found = step(board, depth + 1, max\_moves, allow\_cheat);  
     if (found != -1) {  
     // Found a solution in fewer moves than before  
     found\_solution = true;  
     free\_if\_required(depth);  
     if (is\_two\_move) {  
     free\_if\_required(depth - 1);  
     moves[depth - 1] = copy\_move(cheat\_step\_one);  
     moves[depth] = copy\_move(cheat\_step\_two);  
     } else {  
     moves[depth] = copy\_move(move);  
     // Found a solution better than any found double-Cheat  
     // so remember that this ISN'T a double-Cheat  
     }  
     max\_moves = found;  
     // If the timer is not already running, start it  
     start(timer, continue\_millis);  
     }  
     unapply\_move(board, move);  
     if (is\_two\_move) {  
     // Decrement depth to return to the correct place  
     depth--;  
     }  
     }  
     }  
     }  
    finalise:  
     // Cut the node out if I had to allocate it and we're not within the minimum cache  
     // boundary, or if I've proven that it's possible to solve the board  
     if (found\_solution  
     || (free\_node && cache\_boundary != -1 && remaining\_moves > cache\_boundary)) {  
     if (last\_node) {  
     last\_node->next = node->next;  
     } else {  
     cache[board\_hash] = (BoardHashTable\_LLNode\*)node->next;  
     }  
     free((void\*)node->board\_state);  
     free(node);  
     }  
     // Free the loop variables to avoid a memory leak  
     free(move);  
     free(cheat\_step\_one);  
     free(cheat\_step\_two);  
     // If I found a solution, return the number of moves it took  
     if (found\_solution) {  
     return max\_moves;  
     }  
     return -1;  
    }  
      
    int main(int argc, string argv[]) {  
     extern string optarg;  
     extern int optind, optopt;  
     bool solver\_allow\_cheat = false;  
      
     int opt;  
     int exit\_code = EXIT\_FAILURE;  
     int min\_max\_depth = 64;  
     while ((opt = getopt(argc, argv, ":chn:t:m:d:")) != -1) {  
     switch (opt) {  
     case 'c':  
     solver\_allow\_cheat = true;  
     break;  
     case 'n':  
     cache\_boundary = atoi(optarg);  
     break;  
     case 'd':  
     min\_max\_depth = atoi(optarg);  
     break;  
     case 't':  
     continue\_millis = atoi(optarg);  
     break;  
     case 'm':  
     max\_depth = atoi(optarg);  
     break;  
     case '?':  
     eprintfln("%s: -%c: invalid option", argv[0], optopt);  
     goto print\_help;  
     case ':':  
     eprintfln("%s: -%c: option requires an argument", argv[0], optopt);  
     goto print\_help;  
     case 'h':  
     exit\_code = EXIT\_SUCCESS;  
     goto print\_help; // this line suppresses -Werror=implicit-fallthrough  
     default:  
     print\_help:  
     eprintfln(  
     "Usage: %s [-ch] [-n <moves>=16] [-t <milliseconds>=500] [-m <max "  
     "moves>=1024] <STATE>",  
     argv[0]);  
     if (exit\_code == EXIT\_SUCCESS) {  
     eprintln(" -c Allow Cheating.");  
     eprintln(  
     " The solver will only attempt to Cheat "  
     "if a solution cannot be found without Cheating.");  
     eprintln(" -h Print this message and exit.");  
     eprintln(  
     " -n <cache boundary> The maximum number of moves remaining "  
     "for the board state to be cached.");  
     eprintln(" Defaults to 16.");  
     eprintln(" If set to -1, all board states are "  
     "cached. (Not recommended).");  
     eprintln(" -t <milliseconds> How long to continue searching for "  
     "a more optimal solution after one is found.");  
     eprintln(" Defaults to 500 (0.5 seconds).");  
     eprintln(  
     " If set to -1, the solver will search "  
     "all possibilities to find the most optimal solution.");  
     eprintln(" -d <maximum moves> The minimum maximum number of "  
     "moves to allow.");  
     eprintln(" Defaults to 64.");  
     eprintln(  
     " Setting this to very low or very high "  
     "values will increase run time and memory consumption.");  
     eprintln(  
     " -m <maximum moves> The maximum number of moves to allow.");  
     eprintln(" Defaults to 1024.");  
     eprintln(" Very large values will cause "  
     "allocation errors.");  
     eprintln(" 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.");  
     }  
     exit(exit\_code);  
     }  
     }  
     if (!argv[optind]) {  
     println("Need starting board state");  
     exit(EXIT\_FAILURE);  
     }  
     Board\* board = parse\_input(argv[optind]);  
     // invalid boards will cause an exit in parse\_input  
      
     int start\_max\_depth = min(min\_max\_depth, max\_depth);  
     moves = calloc(max\_depth, sizeof(Move\*));  
      
     for (int step\_max\_depth = start\_max\_depth; step\_max\_depth <= max\_depth;  
     step\_max\_depth += 4) {  
     int n\_moves = step(board, 0, step\_max\_depth, false);  
     if (n\_moves == -1) {  
     eprintf("No solution found in %d moves\r", step\_max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d %c> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->is\_cheat ? '!' : '-',  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     free(moves[i]);  
     }  
     free\_board(board);  
     clear\_cache();  
     return 0;  
     }  
     }  
     if (solver\_allow\_cheat) {  
     clear\_cache();  
     for (int step\_max\_depth = start\_max\_depth; step\_max\_depth <= max\_depth;  
     step\_max\_depth += 4) {  
     int n\_moves = step(board, 0, step\_max\_depth, true);  
     if (n\_moves == -1) {  
     eprintf("No solution found in %d moves (Cheating)\r", step\_max\_depth);  
     } else {  
     for (int i = 0; i < n\_moves; i++) {  
     printfln(  
     "%d %d %c> %d %d",  
     moves[i]->from\_x,  
     moves[i]->from\_y,  
     moves[i]->is\_cheat ? '!' : '-',  
     moves[i]->to\_x,  
     moves[i]->to\_y);  
     free(moves[i]);  
     }  
     free\_board(board);  
     clear\_cache();  
     return 0;  
     }  
     }  
     }  
     free\_board(board);  
     clear\_cache();  
     return 0;  
    }
  + solver.h (click to expand)
  + #pragma once  
    #include "board\_info.h"  
    #include "compress.h"  
    #include "hash.h"  
    #include "memory.h"  
    #include "parse.h"  
    #include "timer.h"  
    #include "types.h"  
    #ifdef DEBUG  
     #include "debug.h"  
    #endif  
      
    #include <unistd.h>  
      
    int step(Board\* board, int depth, int max\_moves, bool allow\_cheat);
  + timer.c (click to expand)
  + #include "timer.h"  
      
    /\* Create a timer with extra\_milliseconds on the clock  
     \*/  
    Time make\_timer(int extra\_milliseconds) {  
     // If extra\_milliseconds == -1 then the user has disabled the timer (so don't start  
     // one)  
     if (extra\_milliseconds == -1) {  
     return (Time) {0, 0};  
     }  
     Time now;  
     gettimeofday(&now, NULL);  
     now.tv\_usec += 1000 \* extra\_milliseconds;  
     now.tv\_sec += now.tv\_usec / 1000000;  
     now.tv\_usec %= 1000000;  
     return now;  
    }  
      
    /\* Check if timer has expired  
     \*/  
    bool expired(Time timer) {  
     // timer has not been started; it is not expired  
     if (timer.tv\_sec == 0) {  
     return false;  
     }  
     Time now;  
     gettimeofday(&now, NULL);  
     return now.tv\_sec > timer.tv\_sec  
     || (now.tv\_sec == timer.tv\_sec && now.tv\_usec > timer.tv\_usec);  
    }
  + timer.h (click to expand)
  + #pragma once  
    #include "common.h"  
      
    #include <sys/time.h>  
      
    #define start(timer, time) \  
     if (!timer.tv\_sec) { \  
     timer = make\_timer(time); \  
     }  
      
    typedef struct timeval Time;  
      
    Time make\_timer(int extra\_milliseconds);  
    bool expired(Time timer);
  + types.h (click to expand)
  + #pragma once  
    #include "common.h"  
      
    typedef uint8 Card;  
      
    typedef struct \_Column {  
     // largest possible stack:  
     // XXXXXTKDV09876C (15)  
     Card cards[15];  
     uint8 count;  
     uint8 stack\_begin;  
     bool cheated;  
    } Column;  
      
    typedef struct \_Board {  
     Column\* cols[6]; // Board.cols[x][y]  
    } Board;  
      
    typedef struct \_Move {  
     uint8 from\_x;  
     uint8 from\_y;  
     uint8 to\_x;  
     uint8 to\_y;  
     bool is\_cheat;  
     bool was\_cheat;  
    } Move;  
      
    typedef struct \_BoardHashTable\_LLNode {  
     const char\* board\_state;  
     int unsolvable\_in;  
     const struct \_BoardHashTable\_LLNode\* next;  
    } BoardHashTable\_LLNode;

# 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

## Test #9: parse\_input robustness

[Commit of test (used to view the test at the time)](https://github.com/Starwort/NEA/commit/72d426b83270a9aaa66a40963c8af0695ae8ae2a/)

starwort@hedwig ~/Documents/NEA master$ dist/test "NotEnoughCharacters"  
Unexpected end of string; Expected 42 characters but only got 19  
starwort@hedwig ~/Documents/NEA master$ \_.rtn  
1  
starwort@hedwig ~/Documents/NEA master$ dist/test "MoreCharsThanNecessarySoTheUnnecessaryCharsGetIgnored!"  
Bad input string: Expected 6 columns, got 0  
starwort@hedwig ~/Documents/NEA master$ \_.rtn  
1  
starwort@hedwig ~/Documents/NEA master$ dist/test "Enough Characters But No Column Separators"  
Bad input string: Expected 6 columns, got 0  
starwort@hedwig ~/Documents/NEA master$ \_.rtn  
1  
starwort@hedwig ~/Documents/NEA master$ dist/test "Enough.CharsAndSeps.But.All.Character.Bad!"  
Expected a digit in the range 6-9, '0', 'T', 'K', 'D', V, '!', or '.'; got 'E'  
starwort@hedwig ~/Documents/NEA master$ \_.rtn  
1  
starwort@hedwig ~/Documents/NEA master$ dist/test "VTK0KV.89K97V.697070.6TT880.D9T6K6.D7V8D.D" # Column sep in wrong place (extra card in a bad 7th col, card missing from 6th col)  
starwort@hedwig ~/Documents/NEA master$ \_.rtn  
0

Test result: Mostly successful; 3 success, 1 fail: All malformed inputs, except those that are formatted validly but with an invalid layout of cards, get rejected (note that inputs with extra characters are effectively truncated at the 42nd character [note the number of columns being 0 in the second example], this is intended behaviour)

Test action: Validate the number of cards parsed after parsing the board state; there must be exactly 36 cards within the board layout

Post-resolution: Test case now passes

starwort@hedwig ~/Documents/NEA master$ dist/test "VTK0KV.89K97V.697070.6TT880.D9T6K6.D7V8D.D"  
Bad input string: Expected 36 cards, got 35  
starwort@hedwig ~/Documents/NEA master$ \_.rtn  
1

# 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