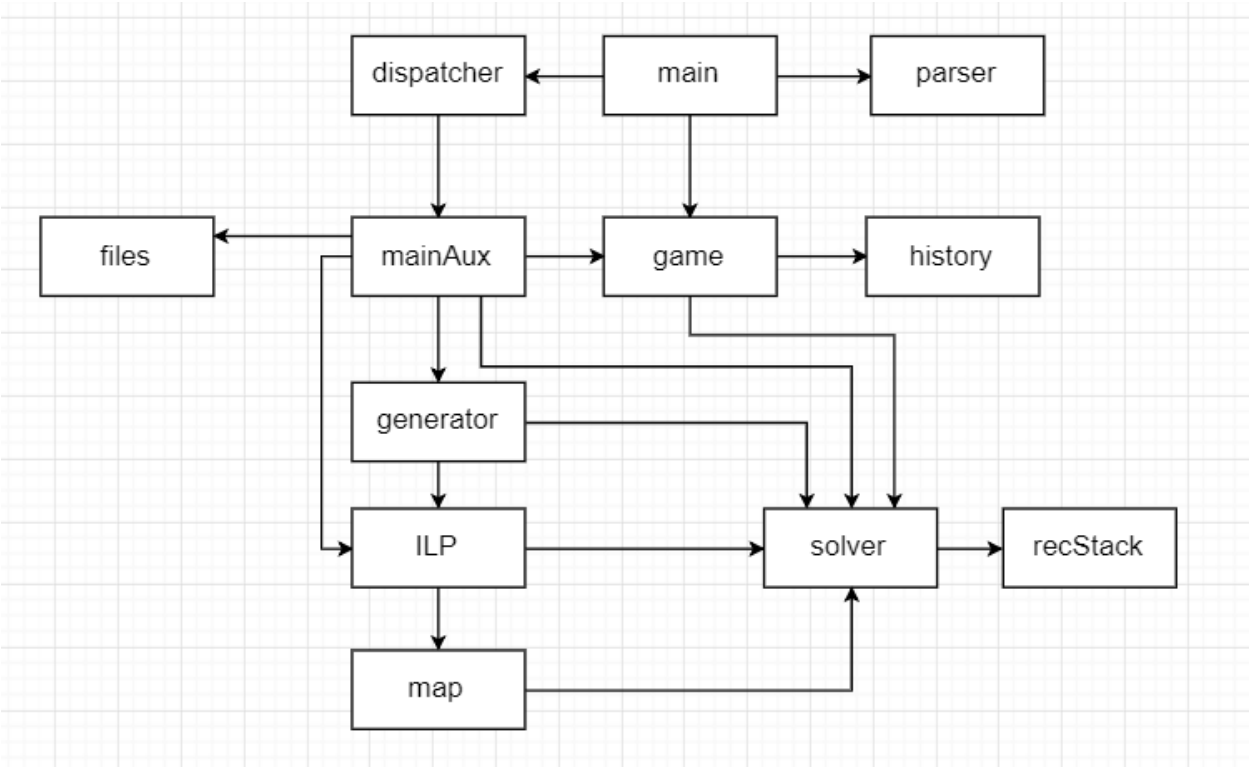
Sudko project

1. *Preface:*
    1. *As defined in the assignment, my project is an interactive Sudoku game with a command line interface.*
    2. *My code implements all the requirements stated in the assignment except the guess and guess hint features (the two features requiring continuous LP, inseat ILP), as T.A. Sulami announced that solo implementations of the project are exempt from these sections.*
    3. *Implementing the project, the main principals that guided me were simplicity, modularity, code reuse and where not significantly contradicting the former: space and runtime efficiency.*
    4. *The purpose of this document is to compliment the documentation in the code itself, present the general structure of each module and provide an easier to read API than in the header files.*

2. *Project architecture:*

    The project is devised of 12 modules (each in it's own .c file), 13 header files (11 header files for each of the modules except main, and 2 headers defining constants or enums) and a makefile.



The modules:

1. ***Main**: Contains one relatively thin function (main) that holds the game stage (represented by an enum "mode"), and pointer to the game's board (if exists).*

2. ***Game**: Defines the data structure containing the game's board and the API to using and manipulating it.*

3. ***History**: Auxiliary module for game. Manages a doubly linked list data structure recording move history for a board.*

4. ***Solver**: This module is in charge of the game's logic, or rule set. Also contains num_solutions function.*

5. ***RecStack**: Auxiliary module that supplies an interface of a pseudo recursion stack that is used in solver module.*

6. ***ILP**: Provides API to solve Sudoku boards using ILP. The module is dedicated to prime a board to be solved by ILP, pass it to GUROBI optimizer and process the library's output. Uses the map module to manage all variables passed to module.*

7. ***Map**: Supplies a data structure that helps create efficient linear programs. the data structure maps between a cell and the variables that represent it's possible assignments and are sent to the optimizer.*

8. ***Generate**: Provides the Generate function, and the frameworks that helps keep it's implementation as efficient as possible.*

9. ***Files**: Provide the API to load and store boards to files.*

10. ***Parser**: parses user input to program format and checks its validity.*

11. ***MainAux**: used to invoke functions (or combinations of functions) and print data (including board) according to user commands.*

12. ***Dispatcher**: used to call specific MainAux function according to parsed command. Separated from main, and mainAux in order to reduce their length.*

13. ***Sizes.h**: Defines constants to be used across the program.*

14. ***Mode**: Defines the mode enum.*

Details:

1. *Main:* *Contains one relatively thin function (main) that holds the game stage (represented by an enum "mode"), and pointer to the game's board (if exists). This function's job is to handle user input, pass it to the parser and pass the parsed data to the dispatcher.*

2. *Game:* *Defines the data structure containing the game's board and the API to using and manipulating it.*

   The board itself is represented by a "board" structure containing Meta-data about the board's general state, the base of the data structure recording the moves history and pointers to two arrays: The first contains the cell values and the second their abstract state (whether they are set or fixed).

   As we want to encapsulate (as much as we can in C) this data structure, any other module performing complex changes to the board will receive an array representation of the board and only manipulate it through this API.

   I choose to represent the board with 1d arrays in order to simplify handling this structure. The cells index is it's order in the board starting from 0 as the left top-most corner. The coordinate system can be easily translated to index from using the cordToInd function.

3. *History: Provides a data structure to document player's move history and an API to use and manipulate it.*

   The history struct itself is consisted of three pointers: the first move made, the last move that was affected the boards current state, and the last move made to the board (might have been undone already).

   Every move documents: the index of effected cell, the previous value in the cell, new value placed to the cell, pointers to next and previous move, and an ID field that helps separate between actual changes to the board. If two moves contain the same ID, it means they were both made during the same user command (such as generate or autofill).

4. *Solver: Contains logic for legality of rules- when is placing a value to a cell legal? A change in rules will only need to be modified here (and for some changes the create constraint in ILP).*

Also implements the num_solutions function that counts the number of solutions possible for current board using exhaustive backtracking. As per assignment requirements this is done using a pseudo recursion stack that is defined in the recStack module.

5. *RecStack: supplies the solver module with a pseudo recursion stack that is used in num solutions. My implementation requires only passing the last edited cell in the board between "recursion" steps. The stack itself is only a pointer to the topmost node in it. Each node (recursion step) holds the last cell that has been edited in the board and calculates the next cell that can be edited or backtracks if there are none.*

6. *ILP: This module solves sudoku boards using ILP. Apart from working directly with the GUROBI optimization library, this module's main job is simplifying the linear program passed to the library as much as possible to reduce runtime and memory usage. This is done by: by trying to fill as much of the board before trying to solve it analytically (apply autofill iteratively to reduce number of empty cells or conclude the board is infeasible), by using the map data structure that allows us to minimize the number of variables in the proSgram and by recognizing certain unsolvable states (during the constraint priming).*

7. *Map: Supplies a data structure that helps create efficient linear programs. the data structure maps between a cell and the variables that represent possible assignments to it, and are sent to the optimizer. This module allows us to "remember" exactly what each variable passed to the linear program represents.*

   The data structure itself wraps an array with the same dimensions as the board. Each cell of the array is a custom structure:
   - *int field, contains the index of the first variable belonging to this cell in the optimizer (this helps us map what variable belongs to what cell more easly.*
   - *pointer to a list where each node representing a variable in the optimizer*

   besides this array, the map holds additional helpful meta-data:
   - *total: number of variables in play.*
   - *size: size of the board that is mapped.*

   Apart from the data structure, this model provides an API to manipulate it.

8. *Generator: Tries to generate a new game by using X random (but only legal) placements, using ILP to complete the board, then clear all but Y random cells.*

   As a naïve implementation of this process might be computational costly, most of the module is dedicated to help this function run more efficiently: arrays which are reused

are only allocated once, empty cells in the board that were filled are kept separately so every randomization will yield a "fresh" cell, and between iterations only edited cells need to be reverted to their original state.

9. *Files: Provide the API to load and store boards to files.*

10. *Parser: parses user input to program format and checks its validity.*