Freecell Solver - Architecture Document

Shlomi Fish <shlomif@iglu.org.il>

Freecell Solver - Architecture Document

by Shlomi Fish

This is version 0.3.2

Table of Contents

Chapter 1. Introduction

This is the architecture document of Freecell Solver. Its purpose is to serve as an introduction to the code, so future contributors can understand it and work on it.

This document does not aim to cover every detail of the application. Such details are supposed to be documented within the comments of the code. (let me know if there is something there you don't understand, and I'll add a comment) Instead, it should provide an overview of the code and cover the main caveats one would encounter when trying to understand it.

This document covers the Freecell Solver 2.8.x architecture, and will be updated to future versions.

Chapter 2. Coding Conventions

Bottom-Up Design and Evolution

Freecell Solver was designed bottom-up: first by writing functions to input cards and then to handle states and finally the algorithm itself. I find that bottom-up design (as evangelised by Paul Graham and others)- start by writing small utility functions and classes and then integrate into a grander scheme, has been a guiding principle when working on Freecell Solver.

Freecell Solver has many utility classes, and has also been coded incrementally. I also find bottom-up design more natural.

Strict Adherence to the ANSI C Language

Freecell Solver is written in pure ANSI C and uses no gcc extensions, no C99 or C++ extensions. I did allow myself to make use of *inline* in some places, but they are abstraced so they'll be used only with gcc or other supporting compilers. (see the "GCC_INLINE" macro).

Freecell Solver is actively compiled with gcc and with the Visual C++ ANSI C compiler. I expect that it should be compilable with other open-source and proprietary compilers on various systems. As this variety of compilers only support the bare ANSI C standard, I have to stick to it. This is despite the fact that such extensions may have made the code more optimised and my life as a programmer much easier.

Strict Adherence to the ANSI C Standard Library

Freecell Solver is dependant on the ANSI C Standard Library as defined by the standard and on that library alone. The core FCS code is not dependant on glib, apr or any other abstraction or encapsulation libraries of this kind.

Freecell Solver can optionally be compiled to make use of some binary trees and hash implementations found in external libraries. This serves as a drop-in replacement for its internal hash implementation, which was shown to usually give a better performance.

Note that a lot of the logic provided by such libraries is implemented internally in Freecell Solver.

Namespace Purity

All the global symbols of the Freecell Solver modules are prefixed with fc_solve_ prefix. (See this post to the fc-solve-discuss mailing list about why we converted away from freecell_solver_. Some macros supplied to the user are prefixed with the fcs_ prefix for convenience. The API functions in turn are prefixed with freecell_solver_user_, in order to not break compatibility with older versions of the library.

Order of Trade-offs in the Design of the Code

Freecell Solver has the following trade offs in the design of the code:

1. Modularity - the code should be as generic as possible and allow for maximal flexibility. The user should be able to configure the library as he pleases.

2. Speed - the code should be optimised for speed. Many times in the code, it was made a bit less comprehensible to gain speed, and many times extra techniques are implemented to ensure this goal.

3. Memory Consumption - should be reduced as much as possible. A reduced memory consumption usually leads to better speed, because there are less cache misses this way.

4. Smart Algorithms - generally, algorithms with a low complexity and such that give advantage to the code are implemented.

Note that these trade-offs are subject to the programmer's whims, and many times, one was a bit sacrificed to satisfy the other.

No Global or Static Variables

Freecell Solver does not use global variables (except for constants) or such that are statically defined within a function. All the non-temporary variables are present within structs, so they can easily be instantiated.

One should note that several distinct instances of the solving algorithm can be instantiated and made to run side by side or alternately.

Separation of Internals from User and Application Programmer Interfaces

FCS makes a clear distinction between the internals of the program, that are subject to change from version to version, and between the interface supplied to the user or the application programmer. All the modules starting from `instance.c` and below are the internals of the program.

`lib.c` contain the API functions used by the programmer. `cmd_line.c` is an API to analyse a Freecell Solver command line. It itself uses only the `lib.c` API. It provides a more flexible interface for configuring Freecell Solver, even to the application programmer.

`main.c` , `test_multi_parallel.c` , `threaded_range_solver.c` , and `fc_pro_range_solver.c` are command line programs that use the command line interface and programmers interface. Others can be written (and have been written).

Chapter 3. Overview of the Code

Overview of the Program

Freecell Solver solves boards by using Depth-First Search or Best-First-Search scans starting from the initial board. It has a collection of the states, and constructs a tree of the states descending from each state to its "parent". A parent state means the state in which from which it was discovered.

Freecell Solver can run several "tests" on each state. Each test generates a list of derived states. Some of the tests are Freecell meta-move ones, some are Freecell atomic move ones, and the others are Simple Simon meta-move ones. The order of the tests can be specified at the command line, and tests can be grouped, and the order of checking of the tests within a group will be determined by a special callback.

To perform the scans themselves Freecell Solver distinguishes between soft threads, hard threads and instances. An instance is a collection of states and an initial board to explore. Each instance may have several hard threads and each hard thread may have several soft threads. A hard thread is something that would likely be put in a system thread. It contains such resources, that a collision between them among several hard threads running in parallel is undesirable. Nevertheless, every hard thread can access the resources allocated by other hard threads, should it encounter them in its scan.

A soft thread is a singular scan. It performs a scan for a certain number of iterations, at which point the hard thread switches to a different soft thread.

Overview of the Utility Modules

app_str.c

This file contains a utility class that appends sprintf'ed output to a dynamically allocated buffer while growing the buffer if necessary.

It is still conceivable that the sprintf will generate a long enough string that will cause a buffer overflow. However, since the class is only used to render states into strings, and the margin allowed for the buffer is long enough, this cannot actually happen.

rand.h

Implements a random number generating class. The algorithm is identical to that of the Microsoft C Run-Time Library (RTL), but the generator can be instantiated.

fcs_hash.c

An optimised hash implementations. This hash maps keys to values. It stores the hash function values next to the keys, so two keys can be first compared with their hash values, before the costly full comparison is done. It also uses a secondary hash to speed up the detection of two keys with identical primary hash values.

Not all the functions of the hash Abstract Data Type (ADT) are implemented here. Only these that need to be used by Freecell Solver.

fcs_dm.c

This module implements two functions - an advanced binary search one and a function to merge a small sorted array into a larger sorted array. It was used until the newer hash or balanced binary

tree storage were implemented, and may still be used if FCS_STATE_STORAGE_INDIRECT or FCS_STACK_STORAGE_INDIRECT are specified. [1]

alloc.c

This module compactly allocates blocks of arbitrary length, usually used for dynamically allocated Freecell columns. The blocks are allocated one after the other inside malloc()'ed memory segments that are thus guaranteed to retain their position.

cl_chop.c

This module contains a class that implements a chopping of a string into arguments. This is done using a subset of the UNIX Bourne shell functionality. Namely:

1. A backslash (\) makes the next character an actual such character.
2. A newline or a white-space separates a word.
3. A backslash at the end of the line continues the processing.
4. Double quotes (") wrap an argument that may contain white-space.
5. A pound sign (#) makes a comment that extends to the end of the line.

The code itself is very spaghetti-like but it is working.

pqueue.c

This module implements a priority queue as a binary heap. It is derived from Justin Heyes-Jones C++ code which he has kindly donated to Freecell Solver (while re-licensing it under the public domain). Since then, the code has been converted to ANSI C , modified and optimised.

This module is used by the Best-First-Search scan.

Overview of the Core Modules

Bottom-up

card.c

Elementary functions to convert cards to and from their string representations. The u and p within the code stand for "user" and "perl" respectively, and mean user representation and internal C representation.

The first experimental version of Freecell Solver was written in Perl, and since then the naming convention for this case persisted.

state.h

This is a header file, but it can be considered a module due to the large amount of logic that it implements. It defines fcs_state_t (which represents a complete layout of the Freecell board) and of

[1] It is no longer recommended to use a sorted array as a state or stack storage, as they are much slower than using a hash or a balanced binary tree, both in asymptotic complexity ($O(n^2)$) and in average performance.

fcs_state_extra_info_t. It contains many macros for manipulating states and cards. (all of them should behave like function calls)

The fcs_state_extra_info_t contains the real positions of the stacks and freecells (refer to (SECTION_REF Canonisation and Normalisation)) and other things that the system uses but don't uniquely identify the state in the state collection.

state.c

This file contains various functions for manipulating states. Among the many things implemented in it are state canonisation, state duplicating, state comparison and converting to and from string format.

move.c and move.h

This module contains routines for handling individual moves (freecell # stack, stack # freecell, stack # stack) and various special moves as well as entire move stacks, which contain a sequence of moves to be played between two intermediate states.

preset.c

This file manages the presets: configurations of stacks number, freecells number, decks number, and the other parameters that define how a game is played. A preset is a variant of Solitaire such as Freecell, Baker's Game, Simple Simon, Good Measure, etc. Many of them are categorised in PySol under different categories than the Freecell category. Moreover, some "Freecell-like" games such as Penguin are not supported by Freecell Solver yet.

The routines in the file enable applying a preset to an instance (by its name), applying a preset to an instance by a pointer to it, etc. It is also directly used by lib.c to maintain consistency across a sequence of consecutive instances.

freecell.c

This module contains move functions for Freecell tests. A move function receives an origin state and tries to deduce if moves of a certain kind are possible. It fills in a derived states list.

This code uses some macros defined in `meta_move_funcs_helpers.h`. It contains both meta-move tests and atomic moves ones.

simpsim.c

This file is similar in spirit to `freecell.c` only it contains Simple Simon move functions.

caas.c

This file contains the `check_and_add_state` function - a function that is used to determine if a reached state is found in the states collection and if so, to insert it there. (an operation that can be considered atomic).

It has several #ifdef'ed portions used to do it for the various types of states collections supported at compile time. It also has a function to collect the new stacks that were present in a similar fashion.

It is being used by the tests functions to put a state in the state collection.

scans.c

This module contains the functions of the various scans and their auxiliary functions. The scan functions run tests and traverse the graph according to some inherent logic. Currently present are random-dfs/soft-dfs (soft-dfs is random-dfs without randomising groups), Best-First-Search (named A* in the code) and Breadth-First-Search (named BFS in the code) functions.

instance.c

This module contains the logic that manages a solver instance, configures it and runs it. It used the scans module to perform the scan and other modules to configure it. Note that the interface presented here is very raw, and not meant to be used as an API.

lib.c

This module contains the user API. It manages a sequence of instances that can be used to solve a board, and then recycled to solve another. It uses instance.c to perform its operations, and do the actual configuring and solving. It supplies the API header file `fcs_user.h` which contains one function for doing any given operation, and these functions are implemented in lib.c.

cmd_line.c

This module can be used to analyse an array of strings (similar to that given to the `main()` function) and configure a user instance accordingly. It also implements reading such arguments from files and a presets mechanism that can be used to assign names to common configuration and load them.

Deprecated Modules

These are modules that were previously used but have been superseded by different code. They can still be found in the `fc-solve/rejects` directory of the trunk.

lookup2.c

This module implements a hash function that was developed by Bob Jenkins. It is essentially his code, that was just integrated into Freecell Solver for convenience (note that it is also Public Domain).

fcs_isa.c

This module implemented indirect state allocation for states. It allocates states in memory pools (called packs) which have a fixed location in memory and allocates as many such pools as it can.

Each pool contains several states that are placed one after the other, that thus retain their pointer. That way, memory is conserved as an individually malloced state may have a lot of overhead. (a malloced block+a fixed amount of data is rounded to the nearest power of 2)

fcs_isa allows releasing the last allocated state in case it will not be used.

Chapter 4. Interesting Techniques Used throughout the Code

State Representation

As can be seen in `state.h`, Freecell Solver supports three ways to represent a state:

1. *Debug States* - in this configuration, stack counters, and cards are represented as 32-bit quantities. This configuration consumes a lot of memory, and is the slowest of the three. It is however useful for debugging, as the debugger will display the state data-structure very nicely.

2. *Compact States* - in this configuration the data is one buffer of chars, where each card and each stack counter are represented as one character, and each freecell and foundation is one char too.

   Determining the locations of every card is done using offset calculation.

   This configuration consumes much less memory than Debug States, but it doesn't scale well to games where the stacks can contain a lot of cards. The reason is that every stack be of a fixed size (so offset would be determined by means of multiplication).

   This configuration used to be the fastest for limited stack games such as Freecell. After Freecell Solver 2.6.x, it seems that Indirect Stack States has become slightly faster than it.

3. *Indirect Stack States* - in this configuration each stack is a pointer to a stack in memory. The stacks are also collected and there is one copy of each stack organisation (say [KS QH 6H]) in memory. Since a pointer to a stack uniquely identifies a stack, the states can be compared by comparing their memory contents.

   This is now the default configuration, and in the 2.5.x development tree, many enhancements were done to optimise it. It was benchmarked to be slightly faster than Compact States, even for games like Freecell.

Indirect Stack States Algorithms

The stacks are kept in their own stack collection in the freecell_solver_instance struct. When a test wishes to create a derived states, it first copies the state, and then marks the flags of all the stacks as cleared. (check (*Mark STACKS_COW_CLEAR *) in the code).

Later on when a stack is changed, its flag is set, and a stack is copied to a indirect stacks buffer of the hard thread. and modified there. (check (*Mark STACKS_COW_COPY_STACK*)).

The check_and_add_state function then, when checking a new state, ignores those stacks whose flag was not set, and collects the stacks whose flag was set. ((*Mark STACKS_COW_CACHING*)). The memory for the collected stacks is allocated compactly in a segment, where one stack starts after the other (check `alloc.c` and `alloc.h`). If the stack was found in the collection the memory that was allocated is freed for use by future stacks).

Extended States

For each position in the game graph, Freecell Solver maintains a data structure which identifies it called fcs_state_t. This contains the cards in the stacks and the freecells, and the value of the foundations. The

stacks and freecells are uniquely sorted to avoid states that are identical except for a different permutation of the stacks or the freecells.

fcs_state_extra_info_t contains a pointer to the fcs_state_t which it is associated with, and defines some other data. The real locations of the stacks and freecells are stored there for instance, as well as some graph information. See Canonisation and Normalisation in the terminology.

The States Graph and its Use

When a brand new state is discovered its parent is assigned to be the state from which it is derived. (check (*Mark STATE_PARENT*)). Its depth is assigned to be the depth of the parent + 1. There is a command line option (--reparent-states) that specifies that if an existing state whose depth is higher than the depth of state it was derived from + 1 is reached, then its parent would be re-assigned.

An extended state has a num_active_children counter that specifies how many of those states that consider it their parent were still not marked as dead ends. If this counter reaches 0, this state also becomes inactive.

The state has a vector of flags called scan_visited, that specifies if a given scan has visited it yet. If it is a complete scan it can also mark it as dead end should it:

1. Recurse out of it if it's a DFS scan.

2. Find that it has no derived states if it is a Best-First Search scan.

If it is marked as dead end, then its parent's counter would be incremented. If the latter is zero, the process may continue to the grand parent and so forth.

The Life-Cycle of a freecell_solver_instance_t

A freecell_solver_instance_t is allocated by `lib.c` to start solving a board. The logic of solving a board is present in `intrface.c` while the external API functions to use it are implemented in `lib.c`.

After an instance is allocated, it should be parametrised to specify how it will solve the board. Afterwards, `freecell_solver_init_instance()` should be called. After that, `freecell_solver_solve_instance()` should be called for the first time, and `freecell_solver_resume_instance()` afterwards. (these functions solve until they reach a limit of iterations number.)

If one would like to use the instance to solve another board, it is possible to recycle it by calling `freecell_solver_recycle_instance()`. This will keep its configuration but free all its associated resources, and thus will not require parsing the command line again.

The function that calls the actual scans is `run_hard_thread()`, which is called from within `freecell_solver_resume_instance()`.

Compact Allocation

Most resources that are allocated arbitrarily in Freecell Solver are allocated in a compact manner. I.e: instead of being individually malloced, they are allocated in segments and placed one after the other. The segments are dynamically allocated and kept at a fixed location in memory. If more memory is needed, another segment is allocated.

The module that is responsible for this is:

1. `alloc.c`/`alloc.h` - allocates blocks of arbitrary size in a compact manner.

It supports releasing the last allocated block and the last one alone.

Compact allocation is used for the following resources.

1. *States* - a derived state is allocated using the hard thread's allocator, and it is modified with the appropriate moves. If it is found to have already existed, it is released. Else, it is kept and a pointer to it can be found in the states collection.

2. *Card stacks* - if a card stack was modified, it is compactly allocated (see (*Mark COMPACT_ALLOC_CARD_STACKS*)), before one checks to see if it is present in the stacks collection. If it was found there, its memory is released. Else, it is kept there.

3. *Move stacks* - the move stacks leading to the parent are compactly allocated (see (*Mark COMPACT_ALLOC_MOVE_STACKS*)).

4. *Hash Elements* - The elements of the hash linked lists are compactly allocated with a hash-wide hash allocator.

Chapter 5. Software Configuration Management

Game Presets Configuration

The info for generating the game presets is present in the file `gen_presets.pl`. It uses data structure inheritance to determine the exact parameters to be included in each preset. Its output should later be incorporated into `presets.c`.

Generating the Site

The site lies in the sub-directory of fc-solve/site/wml in the Subversion's trunk and is generated using GNU Make, and Website Meta Language .

The main site at BerliOS is generated from a makefile and uploaded to its place using rsync.

The pkg-config File

The CMake process generates a `libfreecell-solver.pc` file that can be used as an aid in programs wishing to compile and link against Freecell Solver. It is generated from `libfreecell-solver.pc.in` by CMake and installed system-wide.

The Win32 Binary

The Windows 32-bit binary can be generated by running CMake on Windows on the Freecell Solver distribution, generating a MinGW makefile and type "make package". There's a script to automate it: `scripts/build-on-win32.pl` .

Chapter 6. Terminology

Terms used throughout the Code

Terms used throughout the Code

Canonisation	An extended state is canonised by its stacks being uniquely sorted according to their contents, and an array of indexes describing their original locations sorted accordingly. This is done to make sure no two states with the same permutation of states exist.
Depth	1. The number of successive state # state.parent operations it take to reach the initial state which is the base of the states graph. 2. In Depth-First-Search (DFS): the position of the state in the recursion stack.
False Impossible	A false impossible is an initial board position for which the solver reports as impossible to solve, yet can be solved in some way. A false impossible may be considered a bug depending on the context. A meta-moves-based scan can potentially have false impossibles, while an atomic moves one (which does not prune in any way) cannot.
False Negative	See False Impossible.
f_s_	Short for "fc_solve_" or "freecell_solver_".
freecell_solver_user	a generic name of the API used by the programmer who wishes to utilise the Freecell Solver library in his application. Named after the prefix of the functions of this library.
Graph	The states in the state collection form a directed graph. Each link is a state # derived state relationship.
Hard DFS	A Depth-First Search scan that uses procedural recursion. Since suspending a scan and resuming it are O(d) operations (where d is the depth) instead of O(1) for Soft-DFS its use is deprecated. As a result of this, Hard-DFS was removed from the code, and when being specified, it is implemented by Soft-DFS. Hard-DFS was the original scan supported by Freecell Solver 0.2.0.
Hard Thread	A collection of soft threads, that should generally be placed in one system thread. Hard thread contains resources that soft threads from different hard threads would interfere with each other in allocating. Hard threads contain a collection of state packs, and various counters and other variables.
Instance	An initial board, a collection of states and all the scans associated with it. An instance is initialised whenever one wishes to solve new board. By using command line parameters it is possible to configure it to solve the board in many ways. Instance logic is implemented in `intrface.c`,

	and the user API is implemented in `lib.c`. Users are advised to make use of the command line interface in `cmd_line.c`.
Intractable	An initial layout of the board that cause the solver to terminate the scan prematurely (due to limitations on the iterations and the such) without determining whether the board was solvable or not.
Iterations	the number of states checked by a scan, or by all the scans of a hard thread or of an instance. An iterations limit (called num_times in the code) is used to restrict a soft thread, hard thread or instance from running too long, and to allocate time quotas for different soft threads.
Meta-Move	A move that consists of several individual moves done as one, to move from state to a derived state. Some of the Freecell tests and all of the Simple Simon tests generate meta-moves.
Move	A one-time displacements of cards from stacks to stacks, from stacks to freecells, or from freecells to stacks. Also contain some special moves such as those for canonising stacks, and separators. Also see Move Stacks.
Move Stacks	A sequence of moves implemented in its own object (check `move.c` and `move.h`).
Normalisation	normalisation is the opposite of canonisation. It is meant to bring the stacks and freecells to their absolute locations. It is normally done only when presenting a state to the user or to a code that uses the API.
Parent	The state from which one state in the state graph was initially derived from. It is possible that this state would eventually be reached from a different state, but its parent in that case, remains the same.
Presets	1. A structure specifying the type of game according to number of stacks, number of freecells, number of decks, whether kings can be placed in empty stacks, if sequences have unlimited moves, and how stacks are built by. Defined in `preset.c`. 2. A set of command line arguments to be processed as if they were given on the command line. Can be used to shorten command lines. For instance "-l cool-jives" or "-l john-galt-line" load the presets "cool-jives" and "john-galt-line" respectively. Implemented mostly in `cmd_line.c`.
Re-parent	Let's suppose state DEST has been derived from state SRC. If the SRC.depth+1 is less than DEST.depth than DEST's parent will be reassigned to SRC. (if reparenting is enabled)
Soft DFS	A depth-first search scan that does not utilise procedural recursion. In Freecell Solver, this utilises a stack of records, each containing the current state, the current test, the list of derived states, and other information. This deviates from the standard scheme that puts every state at the end of one stack scheme (that exists in LM-Solve for example) and is harder to maintain, but can be fine-tuned and conserve resources more easily.
Soft Thread	A singular continuous scan operating on a states collection. It can be Soft-DFS, Hard-DFS or Best First Search. There could be any number of soft threads in a hard thread.

Stacks	1. Move Stacks (refer to them) 2. Columns of the Freecell-like games. 3. The stack used for maintaining the Soft-DFS recursion. 4. The environment recursion stack.
State	The position of the game at any given situation. A state accurately describes the contents of the stacks, freecells, and foundations at any given time. A human seeing a state can solve the game from it without further information.
State Collection	A collection that collects every state once and only once. It can be sought of as an associative array (or a map) of keys (the fcs_state_t) to values (fcs_state_extra_info_t)
Test Groups	A grouping of tests that dictate which one should be performed one after the and placed into the same derived states list. Afterwards, this list can be randomised, or prioritised.
Tests	A function that accepts a source state as input and fills a list of derived states according to the moves it can perform. Each game type has several type of tests, which can be ordered and grouped according to input from the user