Freecell Solver - Evolution of a C Program

Shlomi Fish <shlomif@cpan.org>

Freecell Solver - Evolution of a C Program

by Shlomi Fish

Table of Contents

List of Tables

Chapter 1. Introduction

One day in the spring break between two semesters I went out running[1] and as I did, I thought to myself how can a computer solve Freecell. I came to some conclusions and decided to test them by programming a working implementation. I first tried doing it in Perl, but it turned out to be too slow. A C Program that I wrote later did the job well enough according to my standards.

I packaged the software, wrote a README, put it on my Technion web-site and placed it on Freshmeat by labeling it simply as Freecell Solver version 0.2.0. Since then, Freecell Solver saw more than twenty releases, each one adding more features or improving speed, and has grown ten folds in its source base (more if you count the code needed to build and maintain it). In this book, I'd like to tell you about some of the changes I incorporated in it, and what I learned in the course of its development.

This book is about programming in general and C programming in particular. I gained many insights by working on Freecell Solver, some of them related to programming game AIs; others relevant to programming many other types of programs. This book is also about the dynamic of a small Open Source project. As you will see, many times, the input that I received from other users and developers affected the development of the program.

[1] Freecell Solver is not the only good idea I got while going out to run. Another one is "The One with the Fountainhead" which is a parody of Ayn Rand's book "The Fountainhead" modelled around an episode of the T.V. show "Friends". You can also find it on my homepage.

Chapter 2. Rules of the Game

Basic Rules of the Freecell Variant

Freecell is played with a standard 52-card deck. At the beginning of play, all cards are dealt to 8 columns (7 cards each in the first four columns and 6 cards each in the other four) facing up.

Besides the columns (which will also be referred to as stacks), there are also four places to hold temporary cards (known as freecells) and four foundations, in which it is possible to place cards of the same suit starting from Aces and ending with Kings.

Sequences of cards on the tableau are built by placing cards on top of cards of a higher rank and of a different colour (black or red).

An atomic move consists of any of the following:

1. Moving one card from the top of a column or from a freecell to the foundation. (which is possible only if its corresponding foundation's value is lower than it by one)

2. Moving a card from the top of a column to a vacant freecell.

3. Moving a card from the top of a column or from a freecell to a parent card on top of a column.

4. Moving a card from the top of a column or a freecell to an empty column.

It is customary and helpful to group the movement of an entire sequence of cards, by moving intermediate cards to freecells or vacant columns. This will be refered to as a *sequence move*. If a sequence move involves temporarily moving a card to an empty column, it is known as a *supermove*.

You should probably play a few games of Freecell, in case you did not already, because knowing the rules alone is not enough to have an intuitive feel of it. There are many available Freecell implementations for various systems, and you should have no problems finding one that you can use. (chances are that it is already installed on your system)

Techniques of Play

There are several strategies that become apparent after a large amount of playing Freecell. One of them, is uncovering top cards so a card can be moved to the foundations. Another one, is doing the same only to move it to a parent card, or for it to serve as the basis of another sequence.

Another technique, that can be helpful at times is moving a card from the freecells back to the tableau, so it can serve as the base for another sequence.

FILL IN

Variations on Freecell

The game Freecell is not the only Solitaire of its kind nor is the first historically. Using Freecell as a basis, several similar Solitaire variants can be constructed by adjusting some of the basic rules:

1. *Sequence Parenthood* - in some variants cards can be placed on consequent cards of the same suit, or of the cards of any suit whatsoever. Examples for this are Baker's Game which is identical to Freecell except that sequences are built by suit.

2. *Policy of Filling Empty Columns* - in Freecell empty columns can be filled by any cards. In other variants, such as Seahaven Towers or Forecell, only kings may be placed there. On other variants, they cannot be filled at all.

3. *Policy of Moving Sequences* - in Freecell, the amount of cards that can be moved as a sequence is determined by the number of vacant Freecells and columns present. In other variants such as Relaxed Freecell, the amount of cards that can be moved as a sequence is unlimited, regardless of how many vacant resources exist on the board.

4. *Variations in the number of Freecells, Columns or Decks* - Some Freecell variants are played with two decks of cards. Many others varry on the number of columns or freecells. Freecell itself is often played with less than 4 freecells, in order to make the game more challenging. (or vice versa)

Below you can find a table that summarized the essential properties of a large number of Solitaire games similar to Freecell. Their names were taken from PySol [http://www.oberhumer.com/opensource/pysol/], which is a free Solitaire suite written in Python. [1]

FILL IN the table

Table 2.1. Freecell Variants

Name	***Columns Number***	***Freecells Number***	***Decks Number***	***Sequence Parenthood***	***Empty Columns Filled By***	***Sequence Move***
Bakers Dozen	13	0	1	Rank	None	Unlimited
Bakers Game	8	4	1	Suit	Any Card	Unlimited
Beleaguered Castle	8	0	1	Rank	Any Card	Unlimited
Cruel	12	0	1	Suit	None	Unlimited
Der Katzenschwanz	9	8	2	Alternate Colour	None	Limited
Die Schlange	9	8	2	Alternate Colour	None	Unlimited
Eight Off	8	8	1	Suit	Kings Only	Unlimited
Fan	18	0	1	Suit	Kings Only	Unlimited
Forecell	8	4	1	Alternate Colour	Kings Only	Unlimited
Freecell	8	4	1	Alternate Colour	Any Card	Unlimited
Good Measure	10	0	1	Rank	None	Unlimited
Kings Only Bakers Game	8	4	1	Suit	Kings Only	Unlimited
Relaxed Freecell	8	4	1	Alternate Colour	Any Card	Limited

[1] Not all of these variants appear in PySol under the "FreeCell Type" category. As I noticed, many games were very similar to Freecell in their spirit, despite the fact that they were classified otherwise.

Name	***Columns Number***	***Freecells Number***	***Decks Number***	***Sequence Parenthood***	***Empty Columns Filled By***	***Sequence Move***
Relaxed Seahaven Towers	10	4	1	Suit	Kings Only	Limited
Seahaven Towers	10	4	1	Suit	Kings Only	Unlimited

Chapter 3. The Version 0.2 Architecture

Conclusions I Reached while Running

As I ran and thought about solving Freecell I reached several conclusions. One of them was that moving card at a time would probably be too slow to be effective, and that when going from one board layout to the another, I should probably perform several moves at once (such a composite of several moves is known as a *meta-move*). That way, I would hopefully reach a solution more quickly.

Then, I speculated whether it would be better to searching "Depth First" or "Breadth First". By depth first, I mean that whenever the solver reachs a new state (a state is a particular configuration of the cards on the board), it will try to recurse into further states before withdrawing from it. Breadth first means that the states would be scanned according to their vicinity to the initial state: first the states that are reachable from it one move, then those that are reachable in two, etc.

I concluded that the number of states in every depth probably expands by a very large factor, and the nearest solution is relatively deep, so the latter strategy would not be very wise to follow. A depth first search has a greater chance of returning a valid path to the solution in a reasonable time.

Another thing I thought about was that I should store the states that were already encountered in the search, so they won't be checked again and again. This would require implementing a collection of previous states

Initial Version in Perl

A short time after I ran, I started coding the solver in Perl 5. I wrote classes to represent a card and an entire Freecell board, and made sure I could input and output boards from and to text.

Then I started to write the algorithm. There were several guiding prinicipals in the code:

State Collection as an Unordered List

I implemented the state collection as an unordered list of states. A new element was added at the end of the list, and to lookup an existing element, I had to compare it with every element of the list.

State Serialization and De-serialization

The states were serialized into a binary form, so they can be stored and compared. They were de-serialized into a nested Perl data structure so I could manipulate the cards on the board.

Monolithic Search Function

The code had one monolithic function for performing the search and trying the moves on the board. The function was recursive and every time it realized it could perform a meta-move and reach a new state it called itself with the new state as a parameter.

Conclusion

I advanced well into writing the code, when I realized that it was running much too slowly to be effective. Therefore I decided to re-implement the solver in C, hoping it would perform better.

Perl is an interpreted high-level languages and is itself written in C. Therefore, I had reason to believe that my code would run faster if converted to C which is compiled. My code was littered with a large number

of loops nested inside each other, and looping is known to be much slower in Perl. Plus, I believed that serializing and de-serializing the states was time-consuming, and there too, C is better, because serial data structures are already serialized.

Converting to C

The first C version I wrote was not identical to the Perl version algorithmically. As some things were more accessible in C than they were in Perl (or vice versa), I was able to take advantage of them (or not).

Representing Cards and States

A card was represented as a C structure which contained two elements of type short, one representing the rank of the card and the other its suit:

```
struct struct_card_t
{
    short card_num;   /* card_num is the rank */
    short deck;       /* I erroneously referred to the suit as a deck */
};

typedef struct struct_card_t card_t;
```

A column was represented as a structure containing an array of 19 cards (the maximal number that can be present in a column in a game of freecell), preceded by an int that specified the number of cards present:

```
struct struct_stack_t
{
    int num_cards;
    card_t cards[19];
};

typedef struct struct_stack_t fc_stack_t;
```

The structure representing a board layout contained 8 column structures like that, 4 cards for the freecells and 4 integers representing the rank of the cards in the foundations.

```
struct struct_state_t
{
    fc_stack_t stacks[8];
    card_t freecells[4];
    int decks[4];  /* I also called the foundations decks. :-) */
};

typedef struct struct_state_t state_t;
```

The code also contained several macros that were used to manipulate this data and perform actions like querying cards, and states, and modifying them. For instance, `stack_len(`*`states`*`)` retrieved the length of the column with the index *`s`* that belonged to the state *`state`*. As another example, the macro `pop_stack_card` was defined as follows:

```
#define pop_stack_card(state, s, into) \
    into = (state).stacks[(s)].cards[(state).stacks[(s)].num_cards-1]; \
    (state).stacks[(s)].cards[(state).stacks[(s)].num_cards-1] = empty_card; \
    (state).stacks[(s)].num_cards--;
```

Note that the columns inside a state were kept sorted according to a lexicographic order, in order to avoid a situation where two states that are identical except for a different order of their columns.

The Algorithm of the Scan

Following is pseudocode for the algorithm used by the scan:

```
Solve-State(state, prev_states, ret)
    if (state == empty_state)
        Push(ret, state)
        return SOLVED
    for each move possible on state
        new_state <- apply(state, move)
        if (new_state in prev_states)
            ; Do nothing
        else
            add new_state to prev_states
            if (Solve-State(new_state, prev_states, ret) == SOLVED)
                Push(ret, state)
                return SOLVED
    return UNSOLVED

Freecell-Solver(init_state)
    if (Solve-State(init_state, Null, ret) == SOLVED)
        print "State was solved"
        while (state <- Pop(ret))
            print state
    else
        print "I could not solve this board";
```

The algorithm uses recursion to trace the solution and it stores all the states it encountered in a state collection (called `prev_states` in the pseudocode), so they won't be checked again.

The States' Collection Implementation

The states' collection of the C version was implemented as a sorted array of whole state structs (i.e: `state_t * prev_states`) . At the end of that array a *sort margin* was kept with unsorted states. After the sort margin grew to a fixed size, the entire array, including the sort margin was sorted using the `qsort()` function. This yielded a bigger sorted array and an empty margin.

When a new state had to be added it was first added to the end of the sort margin. Then, when the sort margin grew to a certain size, it was merged with the main array. When the size of `prev_states` was exceeded, it was realloced.

To lookup a state, I performed a binary search on the sorted part of `prev_states` and then went over all the elements of the sort margin and checked them one by one.

The Moves

The solver was built as one monolithic function (named `solve_for_state`). The function accepted a state and an integer that specified the depth of the recursion, and returned a boolean verdict whether the state was solvable or not. (`prev_states` was implemented as a global variable)

`solve_for_state` tried to do several moves on the board in the following order:

1. Move a card found at the top of a column, into the foundations.
2. Move a card found in one of the freecells, into the foundations.
3. Put a freecell card on top of a parent card, which is present on top of one of the columns. (this does not involve moving a card from a freecell to an empty column)
4. Move a sequence of cards from the top of a column to a parent card on a different column. (again, while not moving cards to a vacant column)
5. Move a sequence of cards from the top of a column to an empty stack.
6. Put cards that occupy a freecell in an empty stack.
7. Move a sequence of cards that is already on top of a valid parent to a different parent.
8. Move a cards that is hidden under some cards, into the foundations, by moving cards above it to vacant freecells and columns.
9. Empty an entire stack into the freecells so other cards can inhabit it.

Code Organization

The code contained three modules and a header file that implemented many macros. The module `card.c` contained several routines for inputting and outputting cards. It knew how to translate a user-readable format such as `AH`, `10S`, `5C` or `QD` to a card struct.

The header file `state.h` defined the data structures used to represnt cards, columns and states, and the macros that were used to query and manipulate them. The module `state.c` implemented input and output to entire states, as well as functions to compare cards and columns, and a function to "canonize" states. (when canonizing means sorting the columns to avoid duplicacies)

The file `freecell.c` contained the `solve_for_state` function and the `main` function. `solve_for_state` called itself recursively, and contained a lot of duplicate code that searched for existing states and added them to the collection. (I managed it using copy and paste)

Conclusion from the C Program

The C Program ran quickly enough for most boards I tried it with. I generated a 1000 random boards, and invoked it upon them one by one. Most of them ran fine, but for some boards it got stuck thinking how to solve them.

Being happy from its relative success, I converted the code to pure ANSI C (it has been C-ish C++ when I started writing it), wrote a README file and prepared a package. I named the program "Freecell Solver", placed it on a page of its own on my web-site, and posted an annoucement for it on Freshmeat [http://freshmeat.net/].

That was the humble beginning of Freecell Solver: humble because since then, it has grown ten-folds in speed, feature-set and code size. But every journey of a thousand miles begins with one small step...

Chapter 4. The States Collection

Overview

Implementing a game AI system requires collectings the positions of the game that were reached so far, in order to mark them and make sure they are not visited times and again.

Such a position (also known as a *state*) has a 1-to-1 representation of the current state of the game. A solver, human or computerized, given an intermediate position in such form would be able to solve it without any other information.

The states collection of Freecell Solver evolved quite a bit since the first version, and I believe many lessons about programming efficient data structures can be learned from this evolution. By playing around with it, I was able to witness great speed improvements, and also come up with very good insights about what data structures would comprise of a good collection or dictionary.

Initial Perl Version - Unordered List

The initial perl version sported a states collection implemented as an unordered list. I kept an array of states (it could have been a linked list, too) and whenever I encountered a newly visited state, I added it to the end of the array. To search for the existence of a state in the state collection, the list was scanned element by element.

Assuming a comparison and an assignment are O(1) operations (an assumption which would be kept for the rest of this chapter), then it can be seen that an insertion takes O(1) time while a lookup take O(n) time. Since we encounter many states, then we want the lookup to be as fast as possible. But in this case it is O(n).

O(n) is the worst possible lookup complexity for a collection, and as more states are collected, it will become worse and worse to lookup one in it. I became aware of this fact, as I noticed that the perl program ran very slowly. O(n) lookup is in most cases very unacceptable and you should avoid it whenever possible.

Sorted Array with a Q-Sorted Sort Margin

The first C version featured a sorted array as a states collection. In order to avoid the costy operation of incrementing all the positions of the higher states by 1, whenever a new state was added, I kept a *sort margin* that contained several unsorted states at the end.

A new state was first added to the end of the sort margin. When enough states were collected there, I merged the sort margin into the main array, by using the ANSI C `qsort()` function. This entire scheme had an acceptable running time.

Let's analyze the complexity of everything involved. Lookup can be done using a binary search on the sorted part followed by a linear search on the unsorted part. Since the unsorted part is at most a constant number of items, then going over it is O(1) and the entire operation is that of a binary search which is an O(log(n)) operation.

Assuming the number of elements collected in the sort margin is k, then adding k elements, would take O(n*log(n)) time, since the function `qsort()` uses Quick-Sort which has this complexity on average.[1] O(n*log(n)) divided by a constant is still O(n*log(n)) so an insertion of an element is O(n*log(n)).

[1] Quick-Sort has a worst time complexity of $O(n^2)$, but it has an average of O(n*log(n))

The accumulative time for adding n elements is $O(n^2*log(n))$ which is the time for insertion multiplied by n.

Merging the Sort Margin Using the "Merge" Algorithm

Incidentally, I took a course about data structures and algorithms, the semester after I coded the first version of Freecell Solver. There I learned about the large complexity involved in quick-sorting, and that there was a better algorithm for merging two arrays: Merge. The Merge algorithm works like this:

1. Take as input two *sorted* arrays A and B and a result buffer R.

2. Keep two pointers to the arrays PA and PB, initialized to the beginning of the arrays, and a pointer PR initialized to the beginning of R.

3. If *PA < *PB copy *PA to *PR and increment PA. Else, copy *PB to *PR and increment PB. In any case, increment PR afterwards.

4. Repeat step No. 3 until PA reaches the end of A, or PB reaches the end of PB.

5. Copy what remains of the other array to R.

This algorithm runs in linear time. I decided to use a variation of the algorithm, due to the fact that the sort margin could become so much smaller than the main array.

In my variation, the sort margin is scanned linearily, but the index of the corresponding element in the main array is found using a binary search. Once found, all the elements up to it, are copied as is without comparison. That way, one can hopefully save a lot of comparisons.

To facilitate the merge, the sort margin was kept sorted all the times, and middle elements pushed some of the elements forward.

Lookup now is still O(log(n)), but this time insertion was reduced to O(n) time. (the merging operation is O(n) and it is done every constant number of elements) The accumulative time now is $O(n^2)$ which is O(n) multiplied by n.

This scheme ran much faster than the qsort one - by a factor of 3 or so. While the qsort() scheme was adequate to give good results most of the time, a merge of the sort margin was a much wiser choice, algorithmically.

Array of Pointers instead of Array of Structs

Freecell Solver 0.2 managed the states collection as an array in which the states' structs appeared one adjacent to the other. I also passed the entire struct as a paremeter to `solve_for_state`. When I tried to run it on NT I realized that sometimes the program ran out of stack, because it descended into a relatively large depth, while passing all the structs in it. To avoid it, I decided to pass pass pointers to the stacks, and to store the states as an array of pointers.

I did not want to individually malloc the structs, because I knew memory was allocated in powers of 2, and so I can have a lot of it wasted. I first tried to allocate the states by allocating them from the end of one array that was realloced when necessary. However, I realized (after a long and frantic debugging session) that the newly allocated array may not retain its original place in memory, thus causing the pointers that were already collected to become defunct.

To avoid this, I devised a better scheme, in which I allocate the states in packs of constant size, and allocate more packs as necessary. Since the packs contain more than one state, and their size can be adjusted to accomodate for the whims of the underlying memory management, it was a good solution as could be found. [2]

With this management of memory, I converted the states collection to use an array of pointers instead of an array of structs. Not only did it consume less stack space, but it also ran much faster. The reason is that swapping and re-organizing pointers (which take 4 or 8 bytes each) is done much more quickly than re-organizing large structs. Plus, passing pointers on the stack is more efficient than passing and duplicating entire stacks.

This time the algorithmical order of growth was not reduced, but still there can be a difference of heaven and earth between a large $O(n^2)$ and a small $O(n^2)$...

[2] What I could have also done was to manage indices to the states inside the allocation block, instead of pointers. However, I prefered not to use this scheme, because I did not want the extra cost of looking up the real pointer from the index.