Free Cell Solver

Copyright © 2001
Kevin Atkinson
Shari Holstege
December 11, 2001

Abstract

We created an agent that plays the Free Cell version of Solitaire by searching through the space of possible sequences of moves until it finds a solution. By using various techniques, including pruning, random restarts, and a rough heuristic function, our search was successful in every solvable instance, although sometimes we had to change the starting parameters for the search. In general, our solutions are shorter than those found by a comparable Free Cell solver found at http://vipe.technion.ac.il/~shlomif/freecell-solver, although the a-star search found in that package seems to do better with some of the most difficult games.

Background

The Free Cell domain is a reasonably simple one. It is accessible, since all of the cards are face up at the start of the game. It is deterministic; there is no element of chance once the cards have been dealt. Because every deal is completely separate from every other deal, it is episodic. It is static, since the cards will not change while the agent is deliberating. Finally, it is discrete. Only a fixed number of moves is available at each turn.

Not all instances of Free Cell are solvable, based on research found at http://www.cs.ruu.nl/~hansb/d.freecell/freecellhtml.html. Only one of the standard Microsoft games is unsolvable, and of all possible games, only about 1 in 76,000 is unsolvable. Of the solvable games, some are considered easy or hard to solve; information about which games are easy and which are hard is found at http://home.earthlink.net/~fomalhaut/fcfaq.html.

Approach

A game consists of several modules, including a representation for a card and a representation for an entire deck of cards. Each card consists of a rank, which is represented by an integer value from one to thirteen, and a suit, which is represented by an integer value from one to three. If the suit is odd, the card is red. Otherwise, the card is black. A deck consists of 52 cards, one of each of the thirteen ranks in each of the four suits. The ranks are numbered from one to thirteen, with one representing the ace, eleven the jack, twelve the queen, and thirteen the king. A game is a specific configuration of the deck of cards. There are four free cells and four home cells, all of which are initially empty, and eight columns of cards. When the game starts, columns 0-3 each contain seven cards, and columns 4-7 each contain six cards. A free cell can hold any card. The player wins when all of the cards have been moved into the proper home cell in order of ascending rank. Deals can be random, as though the cards were actually shuffled, or they can be pre-determined arrangements of the cards. A move consists of a position from which to move a card and a position to which to move the card. Cards are moved one at a time. Valid moves are any that move cards to free cells or empty columns, any that move the next rank of a suit onto its home cell, or any that move one card to a column where the previous bottom card has the next highest rank and the opposite color of the card moved.

We started by applying general search strategies to the Free Cell domain. Initially, we used an A* search to solve the problem in the hope of finding an optimal solution. We quickly learned that this search ran out of memory and therefore failed in nearly every instance, so we abandoned the A* strategy. Ultimately, we used a depth-first search with a number of variations. A plain depth-first search tends to get stuck in a path that either does not contain a solution or has a solution that may be tens of thousands of moves long and could conceivably contain any number of repeated states and pointless moves. Therefore, we used a few techniques to avoid these problems. First of all, our search checks for repeated states. If it reaches a state it has previously encountered, it checks that state against the previously stored version. If the new path to the state is shorter than the path found before, the new version replaces the old version.

Another approach we tried and abandoned was the use of meta-moves. These meta-moves take several cards and move them at once, and might include, for example, moving all the cards off a card that is ready to move to its home cell. Although we did begin implementing the meta-moves, our search did well enough without them that we decided to optimize the search without them and, perhaps, allow an interesting comparison to the on-line solver we used. We were curious to find whether our search using single moves could be as effective as the searches that used meta-moves.

While our search does not guarantee an optimal solution, it generally finds a pretty good solution. It continues searching until it has either exhausted all possible moves or has progressed through a specified number of states since the last time it found a solution, assuming it found any solution at all. Our search is not complete, so it does not actually exhaust every possibility before it gives up. Rather, it will return to the top of the search tree enough times to try every move that is possible when the game starts, but may not examine every possible state in the search space below each branch. After it finds one solution, it stores the solution and the number of moves in that solution, resets the state counter, and continues the search. Subsequently, as it is searching, it will abandon any search path where any solution found will be longer than the solution already found. To evaluate the path length from a state, we sum the number of moves made to get to that state and the number of cards left in play at that state, since we know that at least that many moves must be made to move those cards to their home cells. In general, the search finds several viable solutions to the game and returns the shortest of the solutions it finds.

Our search also uses a deterministic variation of random restarts. If, after a specified number of states, it has been unable to find a solution, it jumps back up the tree by a certain amount. The longer it has been since it found a solution, the higher up in the tree it jumps. This allows the search to continue in a part of the tree that has proven lucrative. Our strategy was to jump approximately twice as far each time a jump was made, repeating the shorter jumps each time. For example, the search might jump 10%, then 20%, then 10%, 40%, 10%, 20%, and so on. However, since our search tree is finite, we needed a function that would converge to zero. Thus, the jumps are scaled by $(1-x)^2$ instead of $1-2x$, where x is the approximate amount

to jump up and $(1-x)$ is the scaling factor. For example, if we want to jump up by 10% initially, x is 0.10 and $(1-x)$ is 0.90; thus, our initial scaling factor is 90% and the next jump level would be scaled by 0.90×0.90 or 0.81, which is equivalent to jumping up 19%. The next jump level will be 0.81×0.81, which is approximately 0.66, and so on. The idea behind squaring the scaling factor rather than doubling the jump factor comes from the fact that $1-2x \approx (1-x)^2$ when x is small. Often, the search will find one solution, and several other slightly shorter solutions will be very near that solution in the tree. Scaling the random restarts not to go very far if the search has found a solution nearby allows the search strategy to find those other solutions. If the search has reached a part of the tree where there are no solutions, a longer jump allows it to escape that part of the tree and continue the search in an entirely different area.

Each node in the search tree has a local minimum that represents the shortest solution that might be found from that node. Each of the node's children is evaluated according to our heuristic function, which adds the number of moves to get to that node to the number of cards left in play at that state. Then, the parent node's local minimum is set to the minimum of its children.

We used several techniques to prevent our search from doing nonsensical things and to save memory. For example, the search is prevented from selecting a move that would reverse a move it just made. This technique prevents many of the loops that the search tree might otherwise have. We implemented this technique before we decided to save the visited states in a hash table. Saving the states also prevents our search from returning to a state that it visited earlier in the tree, since the new, longer path to the state would not replace the original path stored in the hash table. To avoid using up all of the memory, our search re-evaluates all the stored nodes each time it finds a solution. It purges those states from which the smallest possible solution is longer than the best solution found so far. Finally, our search avoids storing leaf nodes. Leaf nodes are those nodes from which no moves can be made. Since they will not have to be expanded, there is no reason to store them.

Another important technique we used is a function for determining which move to try next. Our search strategy prefers moves in the following order:

1) a card to a home cell,
2) a king to an empty column,
3) a card from a column, preferring a move that will either expose a card that can be moved to a home cell or empty a column in the fewest moves,
4) a card to an empty column provided that another card is ready to move on top
5) a card to a column with the reverse of test #3 (in other words, a card to a column that does not have any cards that are ready to move to their home cells nearly free and is not almost empty),
6) a card off a free cell,
7) a card that is the last one on a column,
8) a card to any other column,
9) a card to a free cell, and finally,
10) a card to an empty column.

By trying the different moves in this order, our search is more likely to find a relatively short solution, since it will try the moves that make the most progress toward the goal first.

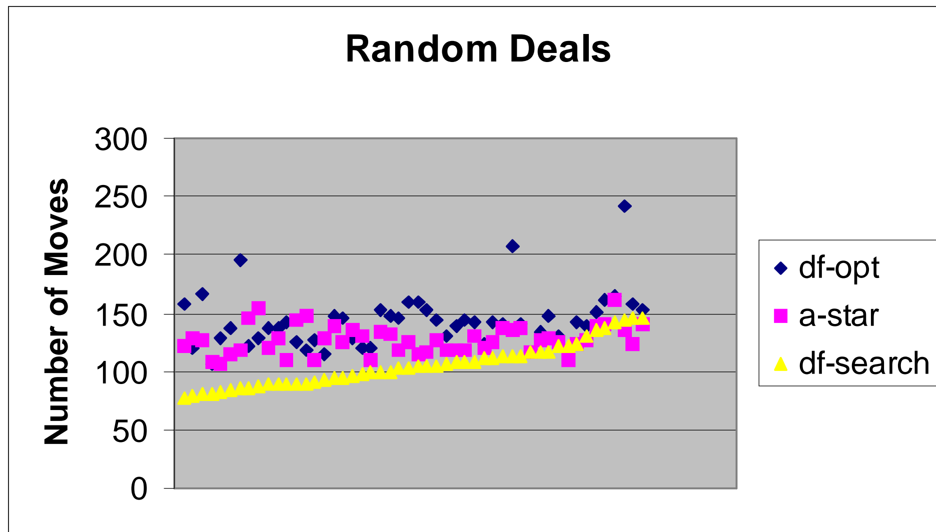
Our search has several parameters that we can change. First of all, there is an optional limit on the number of moves found in a solution. We set our limit to 150 for most of our testing, since most reasonable solutions contain fewer than 150 moves. This prevents the search from wandering down into the tree and never returning. The jump-when parameter tells the search how many states to try in part of the tree before it gives up and jumps. Each time the search finds a solution it resets the count. The jump scale tells the search how far to jump initially. For example, if the parameter is set to 0.9, then the search will jump 0.1 of the way up the tree. The give-up-after parameter tells the search how many states to examine before it gives up on finding any solution. Setting give-up-after to nil (the default) causes the search never to give up until it finds at least one solution. The after-sol parameter tells the search how many states to examine after a solution has been found before it quits searching. If the give-up-after and after-sol

parameters are both set to nil, the search will run to completion. While it is still not guaranteed to return the best solution this way, it will spend more time looking and is likely to return a better solution than the one it would have found with the original parameters.

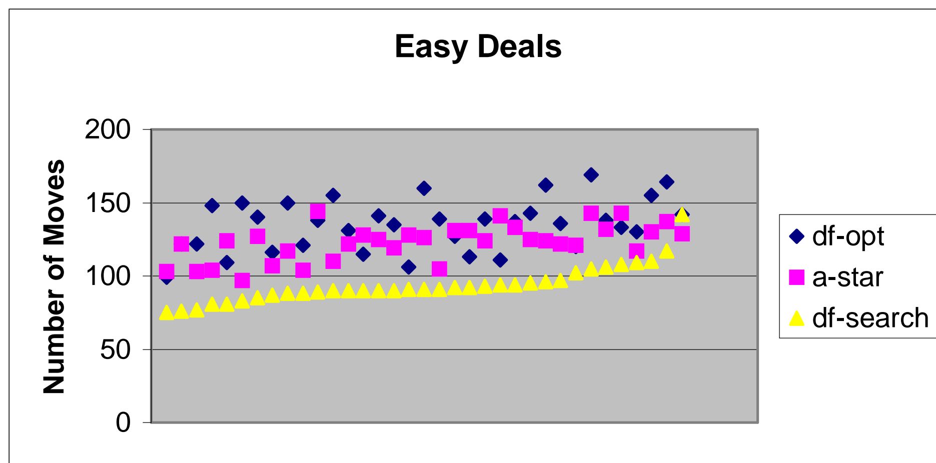
Experimental Results

We ran our search on several sets of games and compared its results to the solutions found by the Free Cell Solver at http://vipe.technion.ac.il/~shlomif/freecell-solver. We used two of the search methods included with the Free Cell Solver we found on-line, the a-star search and the depth-first optimal (df-opt) search. The a-star search is not a true a-star search, since the heuristic it uses is not admissible, and thus, the results are not guaranteed to be optimal. The depth-first optimized search does a simple depth-first search and then tries to optimize the solution by removing redundant moves. Both the a-star and the depth-first optimal searches use meta-moves instead of the single moves our search used. All of the graphed results use the default parameter settings for our search.

First, we ran our search on random deals generated by the deal function in our solver. The results in the chart are sorted by the number of moves in the solution found by our search. In all but a few of the most difficult deals encountered, our search did at least as well as the a-star and depth-first optimized searches to which we compared it. Of the fifty deals, the a-star search's solution was shorter on only four of them. On one of those games, our search was able to find a shorter solution if we lowered the limit on the depth of the search. There were a few other notable games. In two of the random deals, our search initially failed. However, we were able to adjust the parameters so that the search would be successful. In one of those games, the depth-first optimized and a-star searches found solutions with 155 and 140 moves, respectively. Our search found a solution with 168 moves if we changed the depth limit of the search to 200. In another such game, the depth-first optimized and a-star searches found solutions with 174 and 165 moves. Our search, while unable to find a solution in its initial configuration, found a solution of length 163 with a limit of 200 and a solution of length 107 if we allowed it to run to completion.

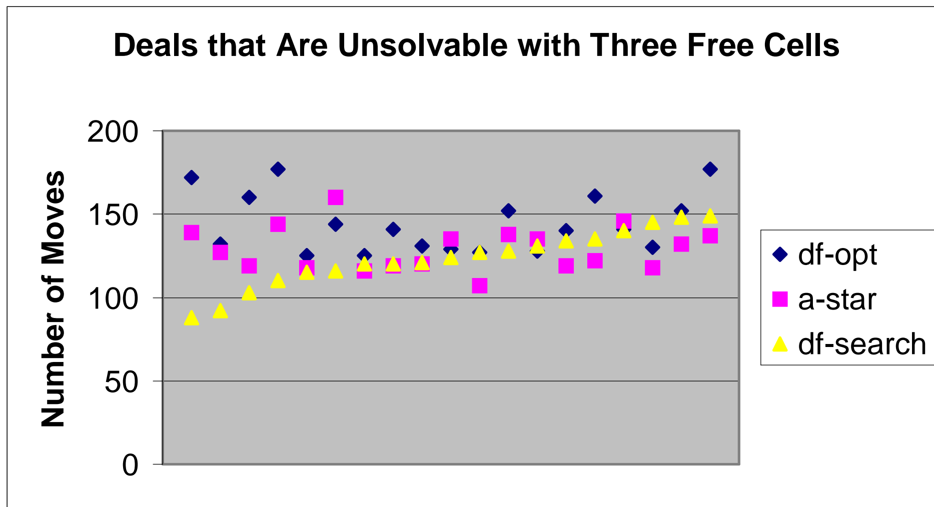


Next, we compared the solutions found by the different solvers for several easy games. We took these games and all of the other pre-determined games that we used from lists of games found at http://home.earthlink.net/~fomalhaut/fcfaq.html. The first set, easy deals, is formally defined: each of them can be solved without using any free cells. Our search seems to do best at simpler deals such as these. There was only one instance in which our search did not do as well as the a-star search. On most of the games, our search did significantly better than either of the other two searches.

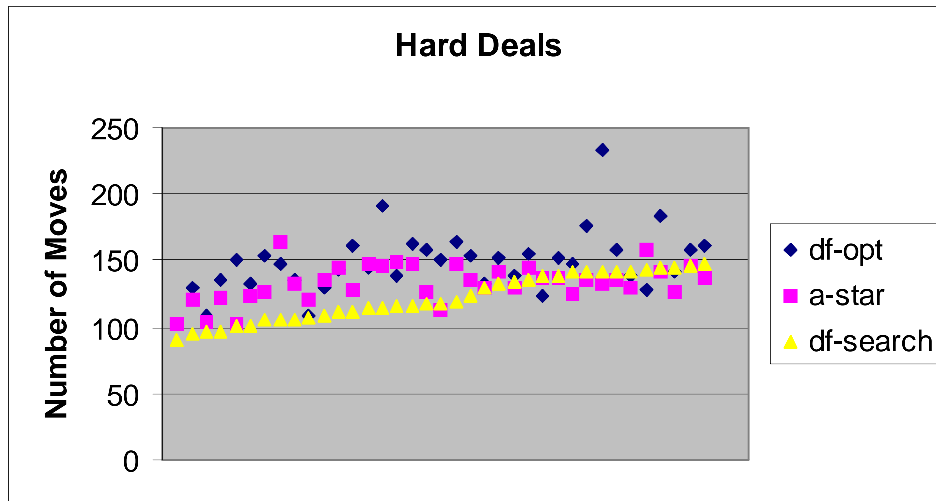


Finally, we ran test cases using difficult deals. We defined these deals in two different ways. First, some of the games are from a list of games that cannot be solved using only three free cells. Of the twenty deals we tried

from this list, our search was able to solve all of them, while the a-star search failed in one instance. However, our search did not generally do as well on these difficult deals as it had done on the simpler deals. Of the nineteen games that all three searches were able to complete, the a-star search found better solutions for nine of them, and the depth-first optimized search found a better solution for one of them. In other words, one of the other searches did better 53% of the time.



The second set of difficult deals was taken from the games listed as difficult or very difficult on the lists created by Dave Ring, Adrian Ettlinger, and Dave Farmer. Of the 37 games we tried, the a-star search found better solutions than our search eleven times, and the depth-first optimized search found better solutions four times, for a total of twelve games (32%) where our search's solution was not as good as the solution found by one of the other solvers.



In general, our search was able to find better solutions than any of the other searches, including our search with the default parameters, if we allowed it to run to completion. However, that process takes at least several minutes and possibly as much as fifteen or twenty minutes on difficult games when solved on a PIII with CMUCL Lisp. Most users would not be willing to wait that long for a solution.

Summary and Future Work

Our search strategy was effective in almost every instance. In the future, we might like to improve its performance on the difficult games, since the a-star and depth-first optimized searches were able to find better solutions one-third of the time on the hard deals and more than half of the time on the games which cannot be solved using only three free cells. One idea we might try is implementing meta-moves and incorporating them into our search, as the optimized depth-first and a-star search did. However, we would like to try a slightly different approach: the optimized depth-first and a-star search in the solver we used only considered meta-moves without considering single moves at all. As a next step in our work, we would like to try considering both meta-moves and single moves and compare those results to the results from using each type of move alone. In addition, we would like to explore further the idea of pseudo-random restarts. In true random restarts, the new search could begin anywhere. In our search, that strategy would be impossible, since our goal is not merely to find a solution state, but rather to find a path to a solution state. If we jumped to a random state, we would not

know how we had gotten there, and we would not have found a solution to the problem. Instead, we try to make the search jump to a part of the search tree that might have a solution. If it has found solutions nearby, it does not jump very far. If the part of the search tree being explored seems hopeless, the search jumps farther in the hope of escaping that part of the tree.

Conclusions

We used various search techniques to solve games of Free Cell, and our search was, on the whole, very successful. Overall, of the three searches tried, our search found the best solution 81% of the time with the initial configuration of our search parameters. It was most likely to find the best solution on the simpler games, although it was almost always successful, even on the difficult games. If we adjusted the parameters, then our search was able to solve every game we tried for which a solution exists.

Appendix: Raw Data

The data from testing the searches appears below. Each of the games was solved using each of the three searches. The data that does not appear in the graphs (in general, the data for which one of the searches failed or in which one of the parameters for our search had to be changed) is below the other data with explanations. The first column on each chart is simply a numbering of the games that we used to help order them in the graphs. The second column in all but the random deals chart, which is missing the column, is the Microsoft game number. The rest of the columns are the number of moves in the solutions found by each of the other searches: the optimized depth-first (df-opt) and a-star from the on-line solver, and our search (df-search). Each set of data is ordered by the number of moves in our solver's solution.