* [Skip navigation](#content)
* [Skip to search form](#q)

# **January Column: Algorithm to Solve FreeCell Solitaire Games**

By [George T. Heineman](http://www.oreillynet.com/pub/au/3414)  
January 17, 2009 ([1ed](http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html)) | Revised January 3, 2016 (2ed)

This is the third of a series of monthly columns in the blog associated with the [**Algorithms in a Nutshell, 1ed**](http://oreilly.com/catalog/9780596516246/) book, published October 2008 by [**O'Reilly Media, Inc.**](http://www.oreilly.com/)In preparation for the second edition, we have updated the text of these blogs to correspond to the new source code layout.

**In our last column** ([1ed](http://broadcast.oreilly.com/2008/12/december-column-searching-algo.html) | 2ed) we described:

* **Search** algorithms implemented in Java.
* how to download the December code samples
* how to import the **Algorithm Development Kit (ADK)** into Eclipse.

In **Chapter 6: Graph Algorithms**, we investigate a number of algorithms that perform computations over graph structures. In **Chapter 7: Path Finding in Artificial Intelligence**, we investigate algorithms that process *Search Trees*, graph structures that represent a one-player game.

In this column we describe the use of *recursive backtracking* to solve a search problem. This material is not drawn from the book, but rather helps to place the graph searching algorithms in a better context.

**View January Code Samples**

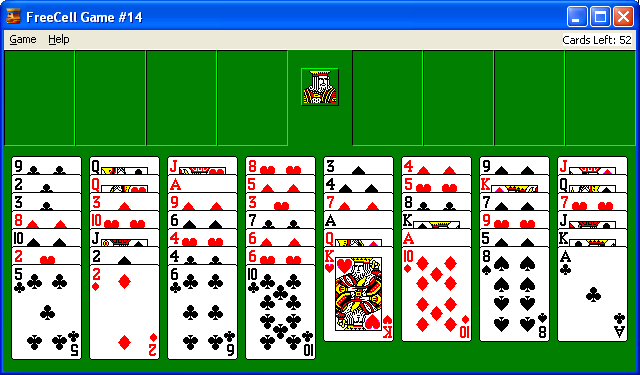
The code samples are found on the github repository (<https://github.com/heineman/algorithms-nutshell-2ed.git>) in the Blogs/ project. You can compile and execute all examples when running from within Eclipse if the entire github repository has been checked out. If you are working from a command-line, be sure that you have fully compiled the ADK code. In addition, you should have the following JAR files in your CLASSPATH

* $ADKHOME/dist/ADK-2.0.jar
* $ADKHOME/Blogs/dist/ADK-2.0-Blog.jar

**The game of FreeCell**

The card game FreeCell is one of the most widely played computer solitaire games because it was traditionally included with the Microsoft Windows operating system (since Windows 95). To play the game, all 52 cards from a standard playing deck are dealt into a tableau of eight columns of slightly-overlapping cards face up: four columns have seven cards while the other four only have six cards. Only the exposed cards of the columns (i.e., the cards that have no overlap on top of them) are available to be moved. There is a set of four "free" cells which can each hold a single card and a set of four foundation piles into which cards are placed by suit in increasing rank from Ace to King. The game is won when the four foundation piles are full.

The following image shows game #14 played using Microsoft's FreeCell implementation:

  
**Figure 1.**Sample FreeCell board (#14 from classic deal)

These are the allowed move types:

1. **Column to Column** -- The exposed card of one column can be moved and placed into an empty column; if the destination column is not empty then the move is valid if the exposed card in the destination column is one rank higher and of a different color than the card being moved.
2. **Free to Column** -- A card can be moved from a free cell to a column in the tableau, according to the restrictions of "Column to Column" moves.
3. **Column to Free** -- A card can be moved from the bottom of one column to an empty free cell.
4. **Free to Foundation** -- A card can be moved from a free cell to a foundation pile if the card is an Ace and the foundation pile is empty; if the foundation pile is not empty then the move is valid if the card being moved is one rank higher and of the same suit as the destination foundation pile.
5. **Column to Foundation** -- The exposed card of a column can be moved to a foundation pile according to the restrictions of "Free to Foundation" moves.

Another move type called a *SuperMove* is a composition of Column to Column moves which makes it possible to move a column of *n* cards (of alternating suit and descending in rank) from one column to another, provided there are at least *n-1* "empty cells" (either in the available free cells or if an entire column is empty).

Of the 32,000 deals playable by the "classic" FreeCell implementations, all but one (the famed #11,982) are solvable. Knowing this information, however, does little to help one write a program that solves a given FreeCell game. In this column we discuss the following topics:

1. How to use **Breadth-first Search (BFS)** to solve FreeCell (and why it won't work)
2. The **Size of the Search Tree** for a FreeCell game
3. How to use **Depth-first Search (DFS)** to solve FreeCell (and why it won't work)
4. How to use **AStar Search** to solve FreeCell (and why it is so hard to make it work!)
5. How to use a **modified DFS algorithm** that reduces its accuracy but still succeeds in solving about 96% of the 32,000 classic deals.
6. Why **Board #11,982 has no solution**
7. Sample **GUI to show solutions live!**

I describe all of the above algorithms within the FreeCell domain. I was motivated to do so because I had heard that writing solitaire solvers was challenging. Indeed, my earliest attempts proved woefully inadequate because I failed to take advantage of the special circumstances of FreeCell games.

**How to use Breadth-first Search (BFS) to solve FreeCell (and why it won't work)**

Instead of trying to produce an program capable of identifying intelligent strategies for solving FreeCell, we pursue a standard approach made possible by high-speed computers. Starting from a Search Tree whose root represents the initial board state, we expand the Search Tree recursively by computing for each of its leaf nodes all available moves, and generating new leaf nodes in the tree by making the moves. When the final goal state is reached, we stop.

Given a FreeCell board state, we can generate a finite set of valid moves. Using this operation recursively, we can construct a Search Tree for an initial FreeCell state using the following code (taken from algs.model.searchtree.BreadthFirstSearch in the ADK):

public Solution search(INode initial, INode goal) {

// Return now if initial is the goal

if (initial.equals(goal)) {

return new Solution (initial, goal);

}

// Store active states in Queue

INodeSet open = StateStorageFactory.create(StateStorageFactory.QUEUE);

open.insert(initial.copy());

// states we have already visited stored in hash table

INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);

// As long as we have an open state, expand upon its moves

while (!open.isEmpty()) {

INode n = open.remove();

closed.insert(n);

// Successor moves translate into appended OPEN states.

DoubleLinkedList moves = n.validMoves();

for (Iterator it = moves.iterator(); it.hasNext();) {

IMove move = it.next();

// make move on a copy

INode successor = n.copy();

move.execute(successor);

// If already visited, search this state no more

if (closed.contains(successor) != null) {

continue;

}

// Record previous move for solution trace. If

// hit goal then leave, otherwise add to OPEN

successor.storedData(new Transition(move, n));

if (successor.equals(goal)) {

return new Solution (initial, successor);

}

open.insert(successor);

}

}

// No solution.

return new Solution (initial, goal, false);

}

This breadth-first exploration of the Search Tree uses a Queue to keep track of the active "horizon" of the search. In doing so, it will uncover a solution with the shortest number of moves. Is this truly what is desired? All we really want to compute is some solution to FreeCell.

To test out the Breadth First exploration of board #14, execute

% java algs.blog.graph.main.StraightBFS 14

where "14" is the command line argument representing the board number to explore). If you are executing the code within an IDE such as Eclipse, you will need some extra steps to supply this argument to the Java program. After a short while, you will likely see the following output:

9C QC JH 8H 3S 4D 9S JD

2C QH AH 5D 4S 5H KD QS

3C 3D 9D 3H 7D 8C 7S 7H

8D TH 6S 7C AS KS 9H JC

TS JS 4H 6D QD AD 5S KC

2H 2S 4C 6H KH TD 8S AC

5C 2D 6C TC

NumMoves: 290482

java.lang.OutOfMemoryError: Java heap space

at freeCell.FreeCellNode.

at freeCell.FreeCellNode.copy

at algs.model.searchtree.BreadthFirstSearch.search

at main.StraightBFS.main

After investigating 290,482 moves (your mileage may vary) the Java virtual machine simply runs out of memory. Now it is possible to extend the size of the heap space, but you will likely exhaust all of your physical memory. The memory runs out because the *closed* and *open* set of states grow without bound. The *closed* set is the one whose size finally exceeds available memory; however, we must keep track of all board states that have already been visited, otherwise we may end up in an infinite cycle.

Is there something about board #14 that has proven to be too difficult? Should we try other boards instead? Before we launch a fruitless check, let's try to estimate how large the search tree is going to be.

**Size of FreeCell Search Tree**

In Chapter 10 (1ed) of **Algorithms in a Nutshell** (Chapter 11, 2ed) we provide code that implements Knuth's method for estimating the size of a Search Tree for the N queens problem. The code below shows a probabilistic algorithm for counting the number of FreeCell board states in the search tree given how many moves (*n*) into the future we wish to consider.

public static long count (FreeCellNode fcn) {

// Run an increasing number of trials, for improved

// accuracy. Look forward just 16 moves.

int LOW\_T = 1024;

int HIGH\_T = 524288;

long aggregate = 0;

long aggregateCount = 0;

for (int n = 1; n <= 16; n++) {

for (int m = LOW\_T; m <= HIGH\_T; m \*= 8) {

TrialSuite ts = new TrialSuite();

for (int t = 0; t < m; t++) {

FreeCellNode here = (FreeCellNode) fcn.copy();

int r = 0;

long lastEstimate = 1;

while (r < n) {

DoubleLinkedList moves = here.validMoves();

int numChildren = moves.size();

if (numChildren == 0) {

lastEstimate = 0;

break;

}

// select one valid move at random.

int rnd = (int)(Math.random() \* numChildren);

DoubleNode node = moves.first();

while (--rnd > 0) { node = node.next(); }

// compute statistics on average number of moves...

aggregateCount++;

aggregate += numChildren;

IMove nm = node.value();

nm.execute(here);

// estimate that all children at this level have

// same number of children

lastEstimate = lastEstimate\*numChildren;

r++;

}

ts.addTrial(n, 0, lastEstimate);

}

System.out.println(n+","+m+","+ts.computeTable());

}

}

double avg = aggregate;

avg /= aggregateCount;

System.out.println("Average number of moves:" + avg);

}

Execute this code via

% java algs.blog.graph.main.FreeCellCount 14

which estimates the size of the search tree for the 14th generated board. The results of this code are shown in the table below. Note the overflow that occurs when counting the size of the Search Tree for just 16 moves into the future. Since 52 cards have to be placed in FreeCell, we can accept the premise that it is simply impossible on current desktop computers to store the entire Search Tree as we explore it! Thus, a [**breadth-first**](#BreadthFirstSearch) approach for searching the Search Tree is not going to work.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1024 trials** | **8192 trials** | **65536 trials** | **524288 trials** |
| 1 | 10 | 10 | 10 | 10 |
| 2 | 97 | 97 | 97 | 97 |
| 3 | 958 | 953 | 952 | 953 |
| 4 | 9670 | 9619 | 9651 | 9639 |
| 5 | 80709 | 81047 | 81737 | 81831 |
| 6 | 521687 | 514008 | 512362 | 514031 |
| 7 | 3148528 | 3145964 | 3158028 | 3160598 |
| 8 | 19229400 | 18874596 | 18897966 | 18691471 |
| 9 | 102380038 | 110320787 | 112600229 | 113277340 |
| 10 | 629428522 | 674500579 | 691503745 | 697487694 |
| 11 | 3683996355 | 4457682162 | 4375374259 | 4496165752 |
| 12 | 29217980095 | 26448488662 | 30181096011 | 29560397738 |
| 13 | 202089900000 | 178279159677 | 205570735002 | 191915309712 |
| 14 | 1357731621016 | 1385267414567 | 1354186049061 | 1341123067292 |
| 15 | 6355922590757 | 8369637109786 | 9178769325917 | 9741519149127 |
| 16 | 89990296563789 | 56981831355020 | 75239486094419 | -1963544077451 |

**Table 1:** Counting Table

One other useful statistic computed by FreeCellCount is the average number of moves in a given FreeCell state. Known as the *branching factor*, this value is 7.78, which means that on average one can make eight moves (which is to be expected when there is at least one empty free cell into which each of the eight columns can make a move).

**How to use Depth-first Search (DFS) to solve FreeCell (and why it won't work)**

The code for depth-first search is rather similar; instead of storing in a Queue the open states being investigated, we use a Stack.

public Solution search(INode initial, INode goal) {

// If goal is initial, return now.

if (initial.equals(goal)) {

return new Solution (initial, goal);

}

// Store active states in Stack

INodeSet open = StateStorageFactory.create(StateStorageFactory.STACK);

open.insert(initial.copy());

// states we have already visited.

INodeSet closed = StateStorageFactory.create(closedStorage);

// As long as we have an open state, expand upon its moves

while (!open.isEmpty()) {

INode n = open.remove();

closed.insert(n);

// Prepare for computations

DepthTransition trans=(DepthTransition) n.storedData();

// Successor moves translate into appended OPEN states.

DoubleLinkedList moves = n.validMoves();

for (Iterator it = moves.iterator(); it.hasNext(); ) {

IMove move = it.next();

// Execute move on copy so we maintain sets of states

INode successor = n.copy();

move.execute(successor);

// If already visited, try another state

if (closed.contains(successor) != null) { continue; }

int depth = 1;

if (trans != null) { depth = trans.depth+1; }

// Record previous move for solution trace. If goal

// is reached leave now, otherwise add to the OPEN

// set if still within depth bound.

successor.storedData(

new DepthTransition(move, n, depth));

if (successor.equals(goal)) {

return new Solution (initial, successor);

}

if (depth < depthBound) { open.insert (successor); }

}

}

// No solution.

return new Solution (initial, goal, false);

}

To test out the Depth First exploration of board #14, execute

% java algs.blog.graph.main.StraightDFS 14.

After a short while (286,620 moves), the depth-first search exhausts the computer's memory also (your mileage may vary)! Is there anything to be done? Well, first recognize that we can set a *depth bound* to prune the depth to which depth-first search searches. However, doing so implies that we know how many moves into the future that we need to search! Let's try it out by invoking

% java algs.blog.graph.main.StraightDFS 14 13

which restricts the depth to 13. After a short while (646,101 moves), the depth-first search once again exhausts memory. Let's try a smaller depth bound by invoking

% java algs.blog.graph.main.StraightDFS 14 5

to look ahead only 5 moves. After 286,620 moves, no solution is found and the search terminates (however we once again exhaust memory with a depth bound of just 7). The following table shows the sample results with differing depth bounds:

|  |  |
| --- | --- |
| **depth bound** | **Number of searched moves** |
| 2 | 109 |
| 3 | 1,111 |
| 4 | 11,615 |
| 5 | 93,339 |
| 6 | 628,197 |
| 7 | 1,836,418 \*exhausts memory |

**Table 2:** Exploring depth-first search with different depth bounds.

You should compare these values with the [**earlier**](#countingTable) to see how accurate the estimation was.

What alternatives exist to depth-first and breadth-first search? These are two "blind search" techniques that simply expand the search tree without care for how good the underlying moves are. What if we take into account the actual board states themselves and try to only make progress on promising boards? Such a search is based on *heuristic* information that can be quickly computed to evaluate how likely a particular board is to reach the goal state. We follow a standard approach where we construct an evaluation function for a FreeCell board state that returns an integer; smaller numbers are better.

**How to use** **AStar Search to solve FreeCell (and why it is so hard to make it work!)**

Using AStarSearch requires an evaluation function scoringFunction that assigns an integer number to a board state. In the code below (from algs.model.searchtree.AStarSearch) the Open states are stored in an ordered tree organized using each node's evaluated score; in this way, one can efficiently remove the board state with minimum evaluation function.

public Solution search(INode initial, INode goal) {

// Start from the initial state

INodeSet open = StateStorageFactory.create(StateStorageFactory.PRIORITY\_RETRIEVAL);

INode copy = initial.copy();

scoringFunction.score(copy);

open.insert(copy);

// states we have already visited

INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);

// As long as we have an open state, expand upon its moves

while (!open.isEmpty()) {

// Remove node with best evaluation and mark closed.

INode best = open.remove();

// Return if goal state reached.

if (best.equals(goal)) {

return new Solution (initial, best);

}

closed.insert(n);

// Compute successor moves and update OPEN/CLOSED lists.

DepthTransition trans = (DepthTransition) best.storedData();

int depth = 1;

if (trans != null) { depth = trans.depth+1; }

for (IMove move : best.validMoves()) {

// Make move and score the new board state.

INode successor = best.copy();

move.execute(successor);

if (closed.contains(successor) != null) {

continue;

}

// Record previous move for solution trace and compute

// evaluation function to see if we have improved upon

// a state already closed

successor.storedData(new DepthTransition(move, best, depth));

scoringFunction.score(successor);

// If already visited state, we may now have lower cost.

// If not, just continue search.

INode exist = open.contains(successor);

if (exist == null || successor.score() < exist.score()) {

// remove old one, if one had existed, and insert better one

if (exist != null) {

open.remove(exist);

}

open.insert(successor);

}

}

}

// No solution.

return new Solution (initial, goal, false);

}

Running this code on board #14 *yet again* runs out of memory:

% java algs.blog.main.StraightAStar 14

The consistent thread throughout all three of the approaches (breadth-first, depth-first, and AStar) is that one needs to store the past states already visited to prevent infinite loops. However, this is not entirely accurate, since you only need to detect if you have visited a prior state. In addition, we can use a technique called *backtracking* that does not require the Search Tree to be constructed in its entirety, but rather executes moves and *reverses* moves as it explores the search space.

**How to use a modified DFS algorithm that reduces its accuracy but still succeeds in solving about 96% of the 32,000 classic deals**

Backtracking relies on recursion and thus mimics the depth-first search. Instead of storing the entire state, what if we just stored a unique key generated from the board state? This would significantly cut-down the storage needs. But we have to go further. We are not (as of yet) taking any advantage of the actual FreeCell game! Here are several points to consider:

1. Given a typical FreeCell state, there are so many different permutations of moves that can be made to lead to a solution; rarely is there a FreeCell deal that demands a very specific sequence of moves that can only be made to reach a solution.
2. Once cards are moved to the foundation piles they can never be used again. This means that we don't need to store old states which can never be visited again.

Putting these observations into effect, however, is a bit more complicated. In this column I present code that implements an algorithm I call *staged deepening*. From an initial board state it explores all possible states that are K=6 moves away. These are all stored in a set S and the board state with lowest score among this set is chosen to expand in the same way. We would quickly exhaust memory if this were allowed to continue unabated. Instead, we clear the set that stores past visited states after it reaches a threshold value arbitrarily set to 200,000. In this way, we bound the memory used to guide the search, so as to never exhaust the computer's physical memory. The outline of the algorithm is as follows:

S = {initial state}

while (S not empty)

n = minimum evaluated board state in S

T = { all boards K moves away from n }

order T by some evaluation function.

S = T

end while

Because we arbitrarily clear out past states (i.e., we do not let the closed set grow without bound), it is possible that we may throw away an important state that is needed for the solution; at the same time, we may find ourselves in an infinite loop since we will not recognize that we have already visited a state. With these warnings in mind, let's review the code from algs.blog.graph.search.StagedDeepening written for this month's blog:

public Result fullSearch (INode start, IScore eval, Comparator<E> comp) {

node = start;

this.eval = eval;

prev = new BalancedTree<E,Integer>(comp);

allIDs = new BalancedTree<E,Integer>(comp);

// conduct search

moveStack = new java.util.Stack<IMove>();

BalancedTree S = new BalancedTree<E,Integer>();

T = new BalancedTree(<Integer,INode>);

S.insert(eval.eval(node), node.copy());

// start is zero

allIDS.insert((E)node.key(), 0);

int lastID;

while (S.size() > 0) {

// invariant: every node that was put into S has a valid ID in allIDs.

node = S.minimum();

// last must be set PRIOR to invoking search, since it is used for

// linking solutions. moveStack must be instantiated anew also, so

// we only create a stack of moves from S.min to new K-distant nodes.

last = (Chain) node.storedData();

moveStack = new Stack<IMove>();

Chain chain = new Chain (stackCopy (moveStack), last);

node.storedData(chain);

if (last == null) {

lastID = 0;

} else {

lastID = last.lastID;

}

if (search(lastID, 0)) {

// update stored move information.

chain = new Chain (stackCopy (moveStack), chain);

node.storedData(chain);

moveStack = computeSolution(node);

Result res = new Result(moveStack);

return res;

}

S = T;

}

return new Result();

}

/\*\* Recursive routine to search up to the given depth. \*/

private boolean search(final int nodeId, int depth) {

if (searchComplete(node)) { return true; }

if (depth > K ) {

if (prev.size() > MAX\_SIZE) {

prev.clear(); // empty out

System.gc();

}

// record that we've been here

prev.insert((K) node.key(), visited);

int score = eval.eval(node);

// store node in T and maintain back-link so we can

// properly reconstruct the move sequence. Last is

// externally set to reflect sequence of moves that got

// to the node from which the initial search(0) invocation

// was executed. We chain moves so we don't have to

// store all previous board states.

INode aCopy = node.copy();

aCopy.storedData(new Chain(stackCopy(moveStack), last));

T.insert(score, aCopy);

return false;

}

// remember where we were BEFORE the move took place.

E kk = (E)node.key();

Integer here = allIDS.search(kk);

DoubleLinkedList s = node.validMoves();

DoubleNode st = s.first();

while (st != null) {

IMove move = st.value();

move.execute(node); // make move

E key = (E) node.key();

Integer exist = prev.search(key);

if (exist == null) {

visited++;

// record ID with this node for later and note we've been here.

allIDS.insert(key, visited);

prev.insert(key, visited);

visitor.visitNode(node, visited);

visitor.visitEdge(nodeId, visited);

if (prev.size() > MAX\_SIZE) {

prev.clear(); // empty out

System.gc();

}

// record we've been here and add to move stack

moveStack.push(move);

if (search (visited,depth+1)) {

return true;

}

// remove from move stack

moveStack.pop();

} else {

Visitor.visitEdge(here, exist); // mark that this edge was visited as well.

}

// undo move and try next one

move.undo(node);

st = st.next();

}

return false;

}

To execute this logic, run the following which increases the amount of memory allowed for the Java virtual machine

% java algs.blog.graph.main.FreeCellExploration

Note that this assumes you have properly configured your CLASSPATH as described in the introduction to this blog. You can select on the command line a *low* and *high* board range to explore as command-line arguments. The table below contains the results of executing the above logic on each of the 32,000 board states of the Microsoft FreeCell implementation.

|  |  |  |
| --- | --- | --- |
| **Result** | **Number FreeCell states** | **Percentage** |
| Solved | 30,859 | 96.4% |
| No Solution | 1,110 | 3.5% |
| Search took too long | 31 | 0.1% |

**Table 3:** Success ratio for *staged deepening*.

These results were compiled over a week-long period. The "search took too long" label was a subjective decision (if there was no response after several hours, the search was manually terminated). These results are extremely promising but there is still room for improvement. Specifically, the computed solutions are not optimal. Consider the "solution" discovered on board #9,559 with 40,661 moves! Not all solutions are so bad, however. The shortest computed solution was for board #11,853 with just 69 moves. The average computed solution (leaving out these two outliers) has 145 moves. Clearly there is room for improvement (how do we identify when search takes "too long"? how do we improve the computed solutions? How do we handle exceptions when the program exceeds memory? how do we deal with the "no solution" cases?) but that is a topic for another column!

**Why Board #11,982 has no solution**

How can we be so sure that Board #11,982 has no solution? To answer this question, run:

% java algs.blog.graph.main.StraightBacktracking 11982

The results are shown below:

Searching board:11982

AH AS 4H AC 2D 6S TS JS

3D 3H QS QC 8S 7H AD KS

KD 6H 5S 4D 9H JH 9S 3C

JC 5D 5C 8C 9D TD KH 7C

6C 2C TH QH 6D TC 4S 7S

JD 7D 8H 9C 2H QD 4C 5H

KC 8D 2S 3S

Solution has 0 moves.

Total number of states: 180045

Cards that didn't move to a free cell

AC AS 3H 4H 6H 6S QC QS

Max Ranks achieved:

0 2 2 0

Positions that are common across all 180045 states

Foundation Cells: CS

Tableau columns:

.. AS 4H AC .. 6S .. ..

.. 3H QS QC .. .. .. ..

.. 6H .. .. .. .. .. ..

.. .. .. .. .. .. .. ..

.. .. .. .. .. .. .. ..

.. .. .. .. .. .. .. ..

.. .. .. .. .. .. ..

.. .. .. .. .. .. ..

.. .. .. .. .. ..

.. .. .. ..

.. ..

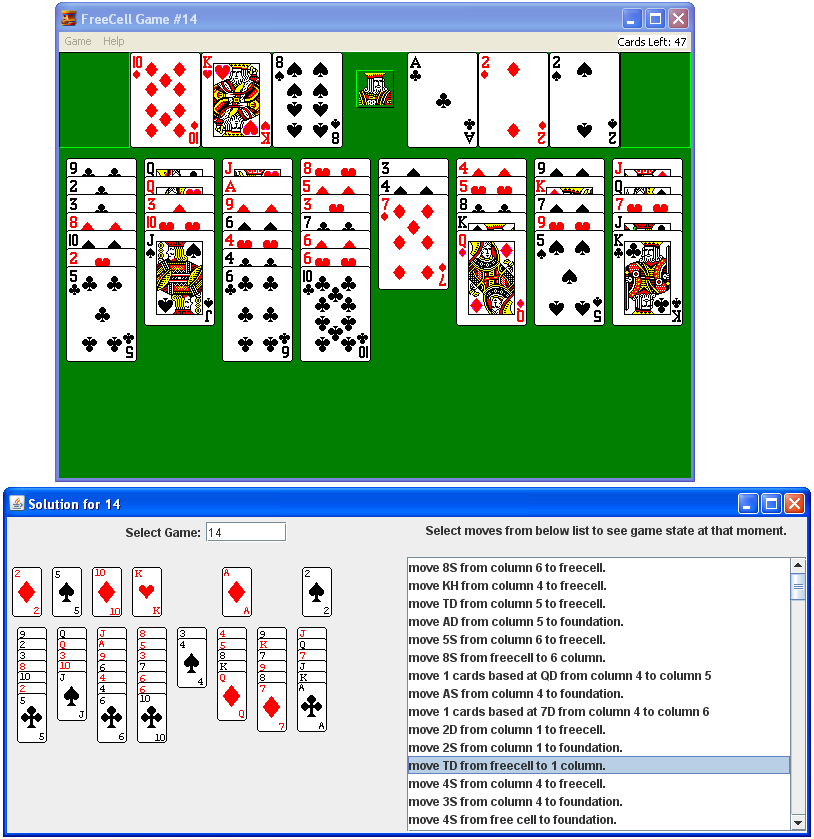
..

After using backtracking to evaluate all 180,045 possible states that are reachable from the initial board state, the program determines there is no solution. It is instructive to review the cards on the tableau which do not move (that is, they are the only cards in common across all of these 180,045 possible board states). There is nothing wrong with having an Ace (or even two) buried at the bottom of two columns. However, for board #11,982 the cards on top of these aces seemingly block any attempt to uncover the Aces.

While the above argument does not constitute a formal *proof*, it is still a fact that all possible board states were explored without finding a solution.

**Sample GUI to show solutions live!**

The code for this Blog contains a sample GUI that can show the playing of a FreeCell game using the solutions generated by the provided algorithm. Here is a sample screenshot:

  
**Figure 2:** Sample solver GUI side-by-side with FreeCell application

To launch the GUI application, execute:

% java algs.blog.graph.gui.Solver

where you will be prompted to enter in the board number you wish to solve. Please enter in a valid board number from 1-32000 which you know can be solved by the staged deepening algorithm (check the included spreadsheet included in $ADKHOME/Blogs/artifacts/graph/FinalReport.xls). You will note that there will be discrepancies that mostly arise from the auto-move feature of the Microsoft FreeCell application. Feel free to use this solver as a "cheat sheet" when playing some of the more complicated board positions!

**Next Column**

In next Month's February column, we will further investigate algorithms from **Chapter 7: Path Finding in AI**. Until next time, we hope you take the opportunity to investigate the numerous algorithms in the Algorithms in a Nutshell book as well as to explore the examples provided in the ADK.

[**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)  
George T. Heineman, Gary Pollice, Stanley Selkow