* [Skip navigation](#content" \o "skip navigation)
* [Skip to search form](#q)

# **February Column: Improving Performance Of Algorithms**

By [George T. Heineman](http://www.oreillynet.com/pub/au/3414)  
February 28, 2009 ([1ed](http://broadcast.oreilly.com/2009/02/february-column-improving-algo.html)) | Revised October 12, 2015 (2ed)

This is the fourth of a series of monthly columns in the blog associated with the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)**, 1ed** 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/2009/01/january-column-graph-algorithm.html) | 2ed) we described:

* A FreeCell solving algorithm based on recursive backtracking

In this column we explain how to improve on the algorithm from last month. This task is quite common, really, in both algorithm design and in programming. We will describe our attempt to improve the performance of the Java-based **Staged Deepening** algorithm introduced last month for solving FreeCell. We also explain why certain optimizations were made when we implemented **Prim's Algorithm** for computing the Minimum Spanning Tree for a graph.

**View February Code Samples**

The code samples are found on the github repository (<https://github.com/heineman/algorithms-nutshell-2ed.git>) in the Blogs/ project.

**Identify Opportunities For Optimization**

The **Staged Deepening** algorithm computed solutions for 32,000 FreeCell boards over a week-long period. Below are the results as computed from last month's column.

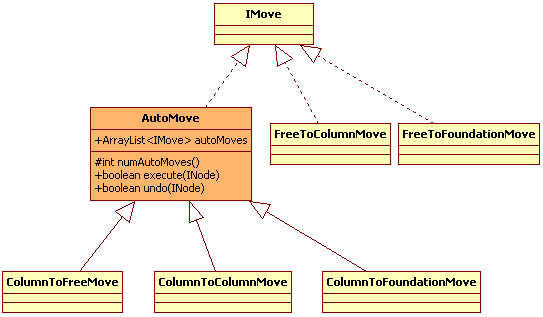
|  |  |  |
| --- | --- | --- |
| **Result** | **Number FreeCell states** | **Percentage** |
| Solved | 30,859 | 96.4% |
| No Solution | 1,110 | 3.5% |
| Search took too long | 31 | 0.1% |

**Table 1:** Success ratio for **Staged Deepening**.

Two questions need to be addressed: Why did it take a week to complete its execution? and what can be done to improve upon the "No Solution" and "Search took too long" results?

The inherent weakness in **Staged Deepening** is that it carries out a *blind search* over the search tree that represents the space of potential games to be played from an initial FreeCell state. Like **DFS** (Depth First Search) and **BFS** (Breadth First Search), **Staged Deepening** only considers each move in isolation. This trait is exactly the opposite that one would want when playing FreeCell since the player must often consider strategies (i.e., "empty out a column so a card can be moved from the FreeCell") as well as sequences of moves (i.e., a super-move in FreeCell occurs when a column of cards are to be moved from one column in the tableau to another). Any blind search algorithm suffers from this inability to string together sequences of moves while attempting a solution.

One idea to pursue is to introduce "Auto Moves" into FreeCell. As more cards are moved to the foundation piles, opportunities arise where cards can automatically be moved to their proper spot on the foundation pile. Specifically, after each player move, the board can be inspected to see if any available card can automatically be moved to a foundation pile. This logic extends the obvious idea to immediately move Aces to the foundation pile as they are uncovered. Indeed, the standard Microsoft FreeCell application has such an Auto Move feature built in.



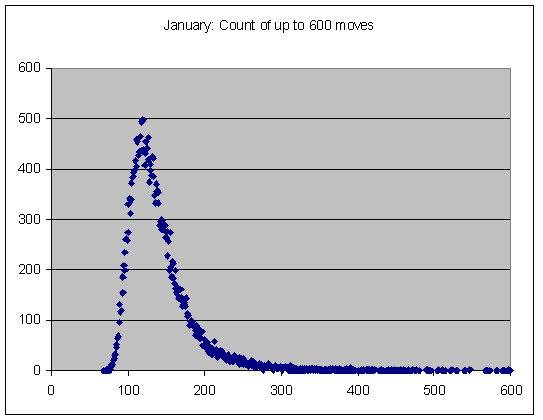
Including the Auto Move feature is straightforward; essentially, a new *AutoMove* class is created which wraps all Move classes which potentially may expose a card that can automatically be played. These three move types are: Column to Column, Column to Free, and Column to Foundation.

Instituting this change leads to the following results. The primary benefit was that the execution only required two days instead of seven (on the same machine platform). That's nearly a 70% speed-up factor; well-worth the small investment in adding the Auto Move feature.

|  |  |  |
| --- | --- | --- |
| **Result** | **Number FreeCell states** | **Percentage** |
| Solved | 31,609 | 98.78% |
| No Solution | 380 | 1.19% |
| Search took too long | 11 | 0.03% |

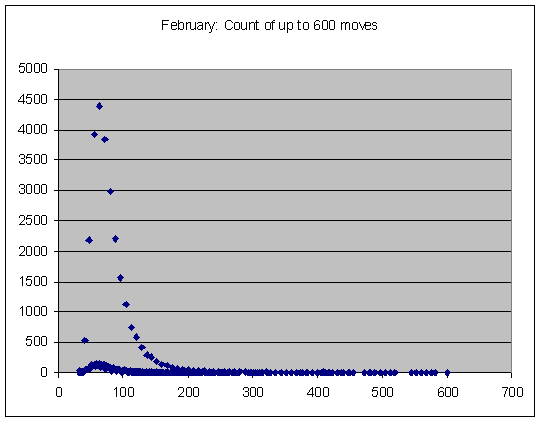
**Table 2:** Success ratio for **Staged Deepening** with auto moves.

At the same time that the performance improved, the quality of the solutions also improved. That is, the returned solution for board #1 using the original **Staged Deepening** algorithm required 133 moves; the modified algorithm only uses 90 moves. For full accounting purposes, there were thirteen cards automatically placed on the foundation pile, and the game ends with a twenty-one card "flourish" of cards that can all be automatically placed. The nature of FreeCell is that the end-game is quite simple once sufficient cards have been placed; in this regard, the **Staged Deepening** algorithm is a perfect fit. The following graph depicts the quality of solutions as generated from the earlier algorithm:



The shortest solution had 69 moves, and the 120-move solution had the highest frequency of occurring. The image is actually misleading, since 92 solutions were omitted (ranging from 607 to 40,661 moves) to make it possible to graph the solutions within a single chart. The quality of the moves is quite poor.

With the improved algorithm from this month's column, the results are stunning. Now 13,894 (or roughly 43% of the boards) have a solution with less than 69 moves. The 64-move solution had the highest frequency of occurring (fully 13.7% of the time).



To improve the quality of the solutions, we would need to move to a different algorithm which "intelligently" solves FreeCell, rather than relying on the blind search algorithms described so far. However, we leave that discussion for another time.

**Add Storage For Increased Performance**

Another principle you can follow is to find ways to store the results of commonly executed computations to avoid unnecessary computation. Consider **Prim's algorithm** as described in Chapter 6 for computing the Minimum Spanning Tree for an undirected, connected Graph g.

void mst\_prim (Graph const &graph, vector &pred) {

// initialize pred[] and key[] arrays. Start with arbitrary

// vertex s=0. Priority Queue PQ contains all v in G.

const int n = graph.numVertices();

pred.assign(n, -1);

vector key(n, numeric\_limits::max());

key[0] = 0;

BinaryHeap pq(n);

vector inQueue(n, true);

for (int v = 0; v < n; v++) {

pq.insert(v, key[v]);

}

while (!pq.isEmpty()) {

int u = pq.smallest();

inQueue[u] = false;

// Process neighbors of u to find if any edge beats best

for (VertexList::const\_iterator ci = graph.begin(u);

ci != graph.end(u); ++ci) {

int v = ci->first;

if (inQueue[v]) {

int w = ci->second;

if (w < key[v]) {

pred[v] = u;

key[v] = w;

pq.decreaseKey (v, w);

}

}

}

}

}

**Figure 3:** C++ code for **Prim's Algorithm**.

The implementation of this algorithm has two optimizations for efficiency:

1. A separate inQueue vector records whether a vertex is in the priority queue
2. A separate key vector stores the priority values of the vertices.

The code relies on a *BinaryHeap* class (provided in the code Repository) to maintain a priority queue of the vertices in the graph. This class was designed to support **Prim's Algorithm**, but what if a programmer wanted to only use the standard STL structures for the implementation? The essential problem is that **Prim's Algorithm** requires the ability to check to see whether a vertex exists within the priority queue.

It is striking how many times that beginning programmers ask for "the iterator" for a priority queue so they can determine whether it contains a specific value. It is exactly to maintain its optimal performance that the STL intentionally leaves out any such iterator method. Here are [**some notes on STL priority queue**](http://www.sgi.com/tech/stl/priority_queue.html) to help explain its design.

What if these optimizations were not present? That is, what if we had coded a rather naive (though simpler) implementation using just STL classes? The following code results:

/\*\* Each entry represents a vertex with its (key) priority. \*/

class Entry {

public:

Entry(int \_id, int \_key) : id(\_id), key(\_key) { }

bool operator<(const Entry&) const;

bool operator==(const Entry&) const;

int id, key;

};

/\*\* Enable heap to properly place lowest key first. \*/

bool Entry::operator<(const Entry& right) const { return key > right.key; }

/\*\* Enable removal from vector based on id. \*/

bool Entry::operator==(const Entry& right) const { return id == right.id; }

void mst\_prim (Graph const &graph, vector &pred) {

// initialize pred[] and key[] arrays. Start with arbitrary

// vertex s=0. Priority Queue PQ contains all v in G.

const int n = graph.numVertices();

pred.assign(n, -1);

// Create Initial Heap

vector pq;

pq.push\_back(Entry(0, 0));

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

pq.push\_back(Entry(v, numeric\_limits::max()));

}

make\_heap (pq.begin(), pq.end());

while (!pq.empty()) {

// remove vertex from priority queue with lowest key value

int u = pq.front().id;

pq.erase(remove(pq.begin(), pq.end(), pq.front()), pq.end());

// Process neighbors of u to find if any edge beats best

for (VertexList::const\_iterator ci = graph.begin(u); ci != graph.end(u); ++ci) {

int v = ci->first;

int w = ci->second;

// If neighbor is in queue, may have to adjust priority

for (vector::iterator ei = pq.begin(); ei != pq.end(); ++ei) {

if (ei->id == v && w < ei->key) {

pred[v] = u;

ei->key = w;

make\_heap (pq.begin(), pq.end());

}

}

}

}

}

**Figure 4:** C++ code for slower implementation of **Prim's Algorithm**.

|  |  |  |  |
| --- | --- | --- | --- |
| **V** | **E** | **Prim MST** | **Slow Prim MST** |
| 6 | 8 | 0.01 | 0.011 |
| 18 | 56 | 0.013 | 0.026 |
| 66 | 464 | 0.027 | 0.301 |
| 258 | 3,872 | 0.114 | 6.8 |
| 1,026 | 31,808 | 0.82 | 181.3 |
| 4,098 | 258,176 | 6.5 | 5,106 |
| 16,386 | 2,081,024 | 50 | 146,889 |

**Table 4:** Average Time (in ms.) for MST and slow MST on benchmark data with ten trial runs.

Running these two side by side produces the results in Table 4. On small graphs, the difference will appear negligible, but the performance rapidly spirals out of control for the slower implementation. You must be careful to select both the proper data structure to use for your implementations as well as the proper way to access the data.

**Postscript**

After completing my **January column (**[**1ed**](http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html)**)**, I received an email from Gary Campbell who has developed his own Freecell Player and Solver. You might want to download his [**Faslo FreeCell Solver**](http://www.numin8r.us/programs) by comparison.

**Next Column**

In next Month's March column, we will turn our attention to investigate algorithms from **Chapter 8: Network Flow**. 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