Combinatorica: A System for Exploring Combinatorics and Graph Theory in *Mathematica*

Sriram V. Pemmaraju* Steven S. Skiena†

July 9, 2004

1 Introduction

Combinatorica is an extension to the computer algebra system *Mathematica* [9] that provides over 450 functions for discrete mathematics. It is distributed as a standard package with every copy of *Mathematica*. *Combinatorica* facilitates the counting, enumeration, visualization, and manipulation of permutations, combinations, integer and set partitions, Young tableaux, partially ordered sets, trees, and (most importantly) graphs. *Combinatorica* users include mathematicians, computer scientists, physicists, economists, biologists, anthropologists, lawyers, and high school students.

Combinatorica (*www.combinatorica.com*) has been widely used for teaching and research in discrete mathematics since its initial release in 1990 [6]. The original *Combinatorica* contained 230 functions, using only 2500 lines of code. Its value lay in the ease with which one could conduct a large variety of experiments on discrete mathematical objects and visualize the results. It was never intended to be a high-performance algorithms library such as LEDA [3], but more as a mathematical research tool and a prototyping environment for effective "technology transfer" of discrete mathematics and algorithms to a diverse applications community. *Combinatorica* received a 1991 EDUCOM award for distinguished mathematics software.

We have recently completed the first significant revision of *Combinatorica* since its initial release over ten years ago [5]. The new package is essentially a complete rewrite of *Combinatorica*. Over 80% of the functions have been rewritten and the package has more than doubled in size to 450 functions and 6700 lines of code. Feedback from users, advances in graph theory and combinatorics, faster and more versatile hardware, better versions of *Mathematica*, and easier access to color graphics were some of factors that motivated this rewrite.

In this paper, we present an overview of the new *Combinatorica* along with a summary of lessons learned along the way. Section 2 presents an overview of *Combinatorica*, including representative graphics generated by the package, a description of new features of the revised version, as well

*Dept. of Computer Science, The University of Iowa, Iowa City, IA 52242, sriram@cs.uiowa.edu
†Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY, 11794-4400, skiena@cs.sunysb.edu

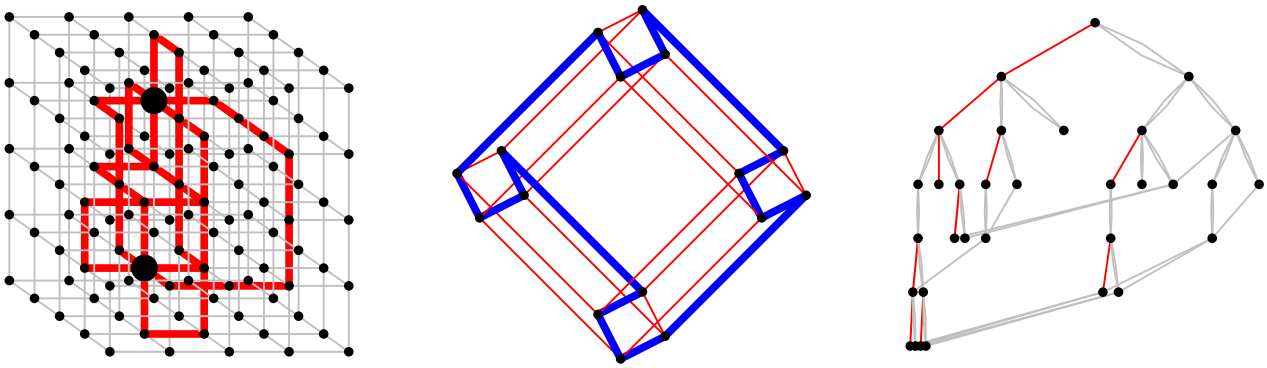


Figure 1: Representative *Combinatorica* graphs I: edge-disjoint paths (left), Hamiltonian cycle in a hypercube (center), animated depth-first search tree traversal (right).

as a review of typical research applications. Section 3 summarizes what we have learned about the design, redesign, and applicability of graph-theoretic software during the long history of the *Combinatorica* project.

2 *Combinatorica* in Action

We begin our introduction to *Combinatorica* with a brief discussion of its design philosophy. We encourage the reader to visit *www.combinatorica.com* for more information on *Combinatorica* and related resources such as algorithm animations, graph database, and Java-based graph editor. Pemmaraju and Skiena [5] is the definitive guide to *Combinatorica*.

2.1 *Combinatorica* Architecture

Combinatorica is best thought of as a library of functions which extend *Mathematica* for combinatorics and graph theory. The arguments to these functions have been carefully designed to that they can be readily composed with each other, i.e. the output of one function serves as an input to another in support of a functional programming style.

Combinatorica is used in one of two primary ways: (1) *interactively*, by responding to a sequence of typed-in function calls, and (2) *programmatically*, by serving as a rich library of basic routines for constructing more sophisticated functions. Each of the illustrations in Figures 1, 2, and 3 were constructed with between one and five lines of *Combinatorica* function calls.

Several advantages accrued to us in developing *Combinatorica* on top of *Mathematica*, including:

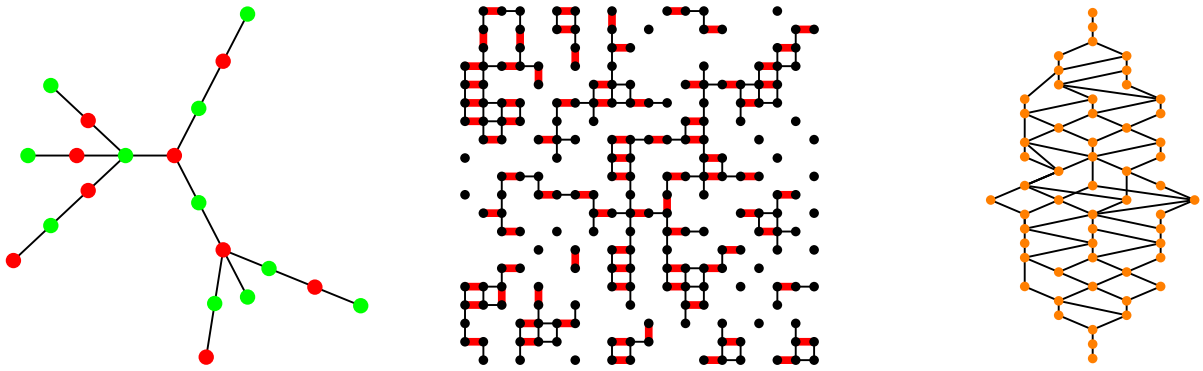


Figure 2: Representative *Combinatorica* graphs II: bicolored tree (left), maximal bipartite matching (center), partition lattice (right).

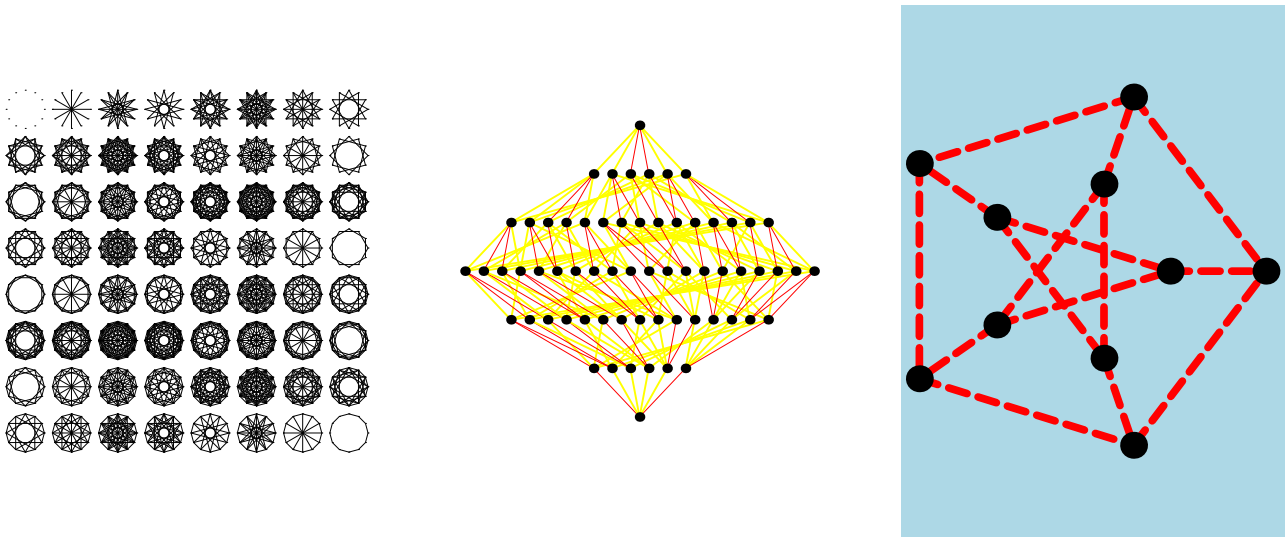


Figure 3: Representative *Combinatorica* graphs III: table of all circulant graphs on 12 vertices (left), Hamiltonian cycle on a six-dimensional Boolean algebra, defining a Grey code (center), Petersen graph to demonstrate visualization options (right).

- *Portability* – Using *Mathematica* as a platform implies that *Combinatorica* is portable to any machine running *Mathematica*, without us having to worry about the peculiarities of different machines. Further, the large and growing installed base of *Mathematica* provides hope that most mathematically-oriented people have access to it[1]. Portability has proven particularly important with respect to *Mathematica* graphics.

- *Density* – *Mathematica* is a very high-level language. Through careful programming, 6700 lines of code suffice to implement all 450 functions. This means that the average function is scarcely more than ten lines long. Because they are so short, most functions are easily readable. Indeed, the complete source code for *Combinatorica* appeared in print in [6]. We reverted to publishing only the most important and instructive functions in [5] due to the increasing size of the system.

- *Power* – Having the rest of *Mathematica* available makes *Combinatorica* much more powerful. For example, chromatic polynomials can be represented as polynomials, which can then be manipulated symbolically. Other aspects of *Mathematica* which are sometimes useful when working with graphs include linear algebra and arbitrary precision arithmetic.

  Perhaps the biggest advantage is the underlying programming language, allowing *Combinatorica* to be used as both an interactive system and as a programming language for working with graphs.

The major disadvantage of using *Mathematica* is that the RAM model of computation traditionally used to analyze the efficiency of algorithms does *not* hold for *Mathematica*. In general, there is no constant-time operation to modify an element in an array or list, so getting efficient performance means designing algorithms which avoid destructive write operations. Further, *Mathematica* is an interpreted language. Because of this, *Combinatorica* is not intended as a tool for performing heavy duty computations. Instead it is a tool for interacting with small examples, and for rapidly implementing programs to provide some experimental insight into specific problems.

About a third of *Combinatorica* 's functionality is devoted to enumerating, counting, and sampling discrete structures. These range from simple functions to count and enumerate permutations to sophisticated functions that use Polya theory to count distinct discrete structures that exhibit inherent symmetries.

2.2 Applications of *Combinatorica*

Combinatorica has been widely used for both teaching and research. The research applications typically fall into one of three types: (1) mathematical research into discrete structures through *Combinatorica* experiments, (2) employing *Combinatorica* to perform discrete simulation modeling, typically by people outside the computer science community, or (3) systematic extensions to *Combinatorica* for particular applications.

[1]Wolfram Research has made considerable progress in recent years in arranging for University-wide site licenses for *Mathematica*, including, for example, both author's home institutions.

Representative research applications of *Combinatorica* include:

- *Unfolding Convex Polytopes* – A longstanding open problem in geometry is the question of whether every convex polytope can be cut along edges and unfolded so that no two faces overlap. Such an unfolding defines a model to construct the polytope. Every potential unfolding is described by a spanning tree of the dual graph of the polytope. This observation is the foundation of Namiki and Fukuda's unfolding package [4], built on top of *Combinatorica*, which has proven to be the best resource available to study this problem.

- *Implementation of Graph Grammars* – *Grammatica* [7] is a prototype implementation of algebraic graph transformation that has been implemented on top of *Combinatorica*, and is available at *http://www.lsi.upc.es/∼valiente/*. *Grammatica* consists of routines for representing, manipulating, displaying and transforming graphs, with special emphasis on algebraic operations on graphs. It supports both interactive and automatic application of double-pushout graph productions, being therefore both a teaching aid and a research tool for algebraic graph transformation.

- *Generalized Lights Out* – The popular puzzle Lights Out, manufactured by Hasbro's Tiger Electronics division, is constructed from a 5×5 grid of lights. Toggling a vertex reverses the on-off state of the vertex *and* all its immediate neighbors. The goal of the puzzle is to start from an initial state where all vertices are lit and turn off all the lights. Initial *Combinatorica* experiments lead to the fascinating theoretical result by Cowen, et. al. [1, 2] that odd-parity colorings over inclusive neighborhoods exist for *all* graphs. These results imply an algorithm to solve Lights Out for any graph.

- *Topologies of Renal Glomerular Microvascular Networks* – In angiogenesis, blood vessels can grow in two ways, i.e. either (1) an existing vessel splits into two parallel vessels, or (2) an existing vessel buds, sending off a shoot which runs into another vessel. The mix of these operations impacts the topology of the resulting network. Wahl, et.al. [8] recently used *Combinatorica* to compute graph invariants on eight previously published rat renal glomerular networks. Invariants calculated include order, size, cycle rank, eccentricity, root distance, planarity, and vertex degree distribution. These invariants enabled the differentiation of the six normal adult glomerular microvascular networks from that of the uremic glomerulus and from that of the normal newborn glomerulus. These invariants might then be used to differentiate between normal and pathological vascular networks.

  Analysis of these networks of hundreds of vertices would not have been possible under the old *Combinatorica* .

2.3 New Features in *Combinatorica*

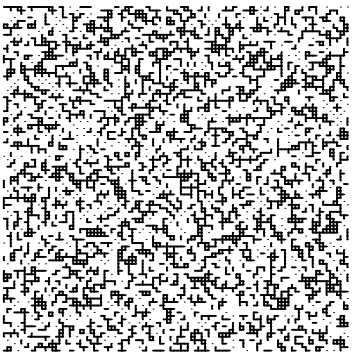
Feedback from users, advances in graph theory and combinatorics, faster and more versatile hardware, better versions of *Mathematica*, and easier access to color graphics are some of the factors that have motivated the rewrite of *Combinatorica* .

Combinatorica now contains (a) an improved data structure for graphs, especially tuned for sparse graphs and for storing drawing information, (b) coverage of additional topics in discrete mathematics, (c) improved graph drawing capabilities, (d) substantially faster code, with some routines sped up asymptotically, and (e) constructors for new graph classes. Here is a quick tour of these features:

Improved graph data structure. The original *Combinatorica* used the adjacency matrix data structure for graphs, for several reasons which were sound at that time. However, with improvements in technology this eventually became a bottleneck in performance. The new *Combinatorica* uses an *edge list* data structure for graphs, partly motivated by increased efficiency and partly motivated by the need to store drawing information associated with the graph. Edge lists are linear in the size of the graph, and this makes a huge difference to most graph related functions. The improvement is most dramatic in "fast" graph algorithms — those that run in linear or near linear-time, such as graph traversal, topological sort, and finding connected/biconnected components. The implications of this change is felt throughout the package; in running time improvements, memory savings, increased functionality, and better graph drawings. The package can now work with graphs that are about 50-100 times larger than graphs that the old package can deal with.

In the following example, we construct a random induced subgraph of a graph with grid graph with 10,000 vertices and about 40,000 edges. The resulting graph with about 5000 vertices and edges is shown below.

```
In[1]:= ShowGraph[ InduceSubgraph[GridGraph[100,100], RandomSubset[Range[10000]]] ];
```



The connected components of this random subgraph are computed in under three seconds on a very modest computer.

Improved graph drawing and animation. One of the highlights of the new package is its new graph drawing capability. Now vertices and edges can be drawn in different shapes, styles, and colors. Vertex numbers, vertex labels, and edge labels can be displayed in a variety of ways with easy control of their sizes and positions. Multiple edges and self-loops are shown correctly. There is now a "zoom" feature, the color of the background can be set, and drawings can be given captions. Figures 1-3 are testimony to some of the new graph drawing capabilities of *Combinatorica*.

The new *Combinatorica* provides several functions to support animations, which can easily be converted to animated gif files. See *www.combinatorica.com* for a nice collection of algorithm animations.

Additional topics. A number of functions relating to *Polya theory* and *set partitions* have been added to *Combinatorica*. Polya theory combines the theory of permutation groups and the power of generating functions to provide ways of counting equivalence classes of families of combinatorial objects, induced by "symmetries." Applications of Polya theory range from counting isomers in chemistry to experimental design in statistics. Polya theory is also useful in counting and enumerating unlabeled graphs. In the following example, we use the *Combinatorica* function `GraphPolynomial` to produce a generating function for the number of unlabeled 6-vertex graphs with different number of edges. Specifically, the coefficient of x^m is the number of unlabeled graphs with 6 vertices and m edges.

```
In[2]:= GraphPolynomial[6, x]

                    2      3      4       5       6       7       8       9
Out[2]= 1 + x + 2 x  + 5 x  + 9 x  + 15 x  + 21 x  + 24 x  + 24 x  + 21 x  +

            10      11      12      13    14    15
       15 x   + 9 x   + 5 x   + 2 x   + x   + x
```

Combinatorica now provides functions on set partitions for enumerating, ranking, unranking, and select uniformly at random.

Improvement in Running Time. Efficiency is a great challenge for *Mathematica*, due to its applicative model of computation and due to the overhead of interpretation (as opposed to compilation). *Mathematica* code is typically 1000 to 5000 times slower than C code and so the original *Combinatorica* could work effectively only on very small graphs (up to 50 vertices or so). In the new *Combinatorica* we can perform interesting computations on graphs with hundreds of thousands of vertices, raising the possibility that *Combinatorica* can now be used not just as a prototyping tool, but even in situations in which high performance is important.

The biggest challenge in speeding up *Combinatorica* functions is due to the overhead of interpretation in *Mathematica*. Also, constant-time destructive write operations are not allowed in *Mathematica*, and this means that operations that we may take for granted as $O(1)$ time operations often take linear time in the *Mathematica* model of computation. However, by using better algorithms, better data structures, more appropriate programming constructs, and new *Mathematica* features we have sped up most functions, many by several orders of magnitude.

For example, the old *Combinatorica* `MinimumSpanningTree` function on a 20×20 grid graph takes more than 120 times longer than the new *Combinatorica* equivalent on a 30×30 grid! The algorithm in the two versions is essentially the same; the huge difference in the running time is entirely due to the change in data structure from adjacency matrix to edge list.

approximate year	1990	1991	1998	2000	2004
command/machine	Sun-3	Sun-4	Sun-5	Ultra 5	SunBlade
PlanarQ[GridGraph[4,4]]	234.10	69.65	27.50	3.60	0.40
Length[Partitions[30]]	289.85	73.20	24.40	3.44	1.58
VertexConnectivity[GridGraph[3,3]]	239.67	47.76	14.70	2.00	0.91
RandomPartition[1000]	831.68	267.5	22.05	3.12	0.87

Table 1: *Combinatorica* Benchmarks on Low-end Sun Workstations, 1990 to date, in seconds.

New instances and classes of graphs. *Combinatorica* now contains constructors for many new classes of graphs and posets: shuffle-exchange graphs, butterfly graphs, boolean algebras, and inversion posets, to name a few. It also contains many instances of interesting specific graphs, encapsulated within the function `FiniteGraphs`. A database of *Combinatorica* graphs containing a complete census (for small n) of non-isomorphic graphs in several interesting classes in available on *www.combinatorica.com*. These are particularly useful to test graph-theoretic conjectures.

3 Lessons from the *Combinatorica* Project

The main lessons we take away from our experience developing and using *Combinatorica* are listed in the subsections below.

3.1 Lessons in Performance Optimization

Lesson 1. To make a program run faster, just wait.

Table 1 presents the results of a series of *Combinatorica* benchmarks over the past 15 years. The differences in running time are partly due to better *Mathematica* implementations, but primarily due to faster machines. All benchmarks are for the original version of *Combinatorica*. In each of these cases we observe a speedup of more than 200-fold. In this context, the further speedups we obtained from upgrading the package become particularly dramatic.

Lesson 2. Asymptotics eventually do matter.

Asymptotic improvements in running time have been obtained for a wide variety of functions. Some of these are due to better algorithms, some to more careful implementations, and others to the new graph data structure. Often these improvements show up as substantial savings in time, even for small inputs.

Lesson 3. Compilation is a win over interpretation.

Mathematica provides a function called `Compile` that can be wrapped around *Mathematica* code to construct *compiled functions*. Specifically, `Compile` creates a `CompiledFunction` object which contains a sequence of simple instructions for evaluating the compiled function. The instructions are chosen to be close to those found in the machine code of a typical

computer, and can thus be executed quickly. Unfortunately, `Compile` expects the inputs to the function and the objects returned by the function to be machine-sized numbers. This means that we cannot pass in graphs, polynomials, huge integers, and all of the other objects we are so accustomed to passing around via *Mathematica* functions. This is a big loss in flexibility, and there is a clear trade-off here between speed and ease of use.

For many *Combinatorica* functions, compilation is not an option since we expect these functions to deal with arbitrary objects. However, it was possible to modify some functions in *Combinatorica* so as to get them to a point where they could be compiled. This has led to significant speedups, as the following example shows.

The function `LexicographicPermutations` generates all permutations of a given list in lexicographic order. In the original *Combinatorica*, it used the standard recursive algorithm in which each element is taken out of the list and then prepended to each permutation of the remaining elements. The new version of the function consists of starting with the list and repeatedly computing lexicographic successors. So the recursion is replaced by iteration and the iterative version is compiled. The differences in timing are a factor of ten in constructing all permutations on nine elements.

Being a compiled function, it can only deal with lists of numbers. If we are asked to permute a list L of n items that are not numbers, we simply compute the lexicographic permutations of the list $\{1, 2, \ldots, n\}$ and then apply each permutation to L. It turns out that even for items that are not numbers, the new version is faster than the old version.

Lesson 4. Not all functions can be made efficient in the *Mathematica* model of computation.

Many efficient algorithms depend on data structures that provide fast implementation of certain operations. For example, on an n-vertex graph with m edges Dijkstra's algorithm runs in $O(n^2)$ time if a linear array is used, $O(m \log n)$ time if a binary heap is used, and $O(m + n \log n)$ time if a Fibonacci heap is used. The run time improvements due to using a heap come mainly because it is possible to update a heap in logarithmic time. In *Mathematica* such updates usually take linear time and so using a linear array ends up being the best option.

Recent versions of *Mathematica* offer the option of storing data in a *packed array* provided the data consists of a sequence or a matrix of machinesized numbers. Using a packed array can reduce memory usage and provide constant-time write operations. However, we face the same trade-off as in trying to compile rather than interpret *Mathematica* code. Sophisticated data structures are often irregularly structured, contain pointers, and are not amenable to being packed. Restricting items in a data structure to be machine-sized numbers requires giving up one of *Mathematica*'s most important features — the ability to handle data of arbitrary size.

3.2 Lessons in System Design

Our *Combinatorica* experience offers several lessons in system design beyond software engineering:

Lesson 1. Sophisticated hardware eventually slithers down to everybody.

Combinatorica currently runs on machines with unimaginably faster speeds and memory sizes than back in 1990, but they could have been predicted as a consequence of Moore's law. This argument suggests designing systems to run on the most powerful hardware available today, instead of aiming to support low-end users.

To make the same point, we designed the original *Combinatorica* to support black and white graphics, because that reflected the typical screens and printers of 1990. Once the then-sophisticated graphics hardware became widely available, color graphics (rather than computational performance) quickly became the biggest demand of *Combinatorica* users.

Lesson 2. Interface consistency and flexibility is more important than performance.

Much of the success of *Combinatorica* is due to its completeness and consistency. The function naming convention, which is inherited from *Mathematica*, capitalizes and completely spelling out each word. Although verbose, it makes it possible for users to routinely use a library as complex as *Combinatorica* and derive the name of functions they have never used before. The completeness of the package encourages exploration of the system instead of the manual.

When conducting experiments in combinatorics, the combinatorial explosion guarantees that you will not be able to search all structures past some small constant. Being able to quickly code up a test of all widgets up to size 10 is usually more useful than a more complicated test up to size 15 or even 20.

One of the simple but important changes of the new *Combinatorica* has been hiding the internals of the graph structure from the view of the naive user. This cue more clearly reveals that the 'right' way to proceed is exclusively by using our access/generation/conversion functions than by cobbling together a graph structure from scratch.

Lesson 3. Hitching a ride to a commercial platform can help with portability and maintenance.

One of the primary reasons why *Combinatorica* has remained in wide use for over ten years is that it rests on a commercial platform. Wolfram Research Inc., the distributors of *Mathematica*, have had the incentive, resources, and responsibility to keep the system running over the years.

Few if any of the "competitor" discrete mathematics programs and graph editors cited in [6] are still running today, because of operating system shifts in the intervening years (DOS to Microsoft Windows, differing flavors of X-windows, Macintosh OS upgrades). Even programming languages are very unstable. During the last ten years, Pascal died, and even old C programs are hard to compile. *Combinatorica* is still alive primarily because somebody else is making it run on other machines for us.

Lesson 4. Commercial platforms do mean a loss of control and visibility.

Hitching our star to a commercial operation has also brought certain problems. Certain function names we used (such as K for complete graph) have been taken from us over the

years by the powers that be. We have also been somewhat of a victim of WRI's pricing strategies. Because *Mathematica* site licenses have until recently been too expensive for most educational institutions, it has not proven as accessible for teaching purposes as we had hoped. Although we have distributed upgrades to *Combinatorica* on the WWW since its inception, it was difficult for many people to justify buying *Mathematica* just to use our package.

In retrospect, did we make the right decision to base *Combinatorica* on *Mathematica*? We believe the answer is yes. *Combinatorica* would have been a substantially different system on any other platform. It is not clear that any other computer algebra system (including Maple) provides a larger user-base. Starting fresh today we might consider using Java, but would you want to bet you will be able to run today's Java programs in ten years?

Lesson 5. Books can greatly increase the visibility of any software product.

One of the key reasons for the success of *Combinatorica* is the associated book [5, 6]. Although complete user documentation is distributed with *Mathematica* and is available at *www.combinatorica.com*, the book has been very important to promoting the package.

Books do many good things. An interesting book cannot just be a user's manual. In [5], we present a substantial volume of theoretical and algorithmic results in a distinctive way, so as to be suitable as a textbook in combinatorics and graph theory courses. Publishing your primary documentation as a book requires thinking more about your target audience than just listing commands and providing examples. They require you to invest as much time in the documentation as the code, and help you anticipate missing features and options. Finally, books considerably expand the awareness of your software; it is your publisher's job to help spread the word about it. They don't do as good a job as we might like, but they do help.

Acknowledgments: We would like to thank Joan Trias for his insights which lead to a preliminary version of the new *Combinatorica* graph structure. Levon Lloyd has developed a Java-based graph editor for the new *Combinatorica*. King Mak developed the graph database and algorithm animation collection available at *www.combinatorica.com*. Finally, we thank several people at WRI (Shiral Devmal, Daniel Lichtblau, Andy Shiekh, Eric Weisstein, and most of all John Novak) who have helped with testing and improving *Combinatorica*.

References

[1] R. Cowen, S. Hechler, J. Kennedy, and A. Ryba. Inversion and neighborhood inversion in graphs. *Graph Theory Notes of New York*, 37:37–41, 1999.

[2] R. Cowen and J. Kennedy. The lights out puzzle. *Mathematica in Education and Research*, 9:28–32, 2000.

[3] K. Mehlhorn and S. Naher. *LEDA: A platform for combinatorical and geometric computing*. Cambridge University Press, 1999.

[4] M. Namiki and K. Fukuda. Strange unfoldings of convex polytopes. http://www.ifor.math.ethz.ch/staff/fukuda/unfold_home/unfold_open.html, 1997.

[5] S. Pemmaraju and S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, New York, 2003.

[6] S. Skiena. *Implementing Discrete Mathematics*. Addison-Wesley, Redwood City, CA, 1990.

[7] G. Valiente. Grammatica: An implementation of algebraic graph transformation on mathematica. In *Sixth Int. Workshop on Theory and Application of Graph Transformations*, pages 261–267, 1998.

[8] E. Wahl, L. Quintas, L. Lurie, and M. Gargano. A graph theory analysis of renal glomerular microvascular networks. *Microvascular Research*, 67:223–230, 2004.

[9] S. Wolfram. *The Mathematica Book*. Wolfram Media, Champaign IL, fourth edition, 1999.