SYMBOL-CRUNCHING with the TRANSFER-MATRIX Method
In Order to Count
SKINNY Physical Creatures

Doron ZEILBERGER [1]

The transfer-matrix method, like the Principle of Inclusion-Exclusion and the Möbius inversion formula, has simple theoretical underpinnings but a very wide range of applicability.

– Richard P. Stanley ([S], p. 241)

Abstract: We describe Maple packages for the automatic generation of generating functions (and series expansions) for three notoriously hard-to-count combinatorial objects that arise in statistical physics: lattice animals (alias polyominoes), self-avoiding polygons, and self-avoiding walks, in the two-dimensional square lattice, of bounded, but *arbitrary* width. The novelty of this work is in its *generality*, *reproducibility*, *explicitness of details*, and *availability*. Our Maple packages (complete with *source code*) are easy-to-use and downloadable free of charge. But perhaps, most important, it is hoped that this work will *illustrate by example* an admittedly crude and semi-amateurish, yet honest, attempt at a foundation for a *work ethics* and *research methodology*, of what will soon be a major part of 3rd millennium mathematical research, and that for lack of a better name we call *mathematical engineering.*

From Number-Crunching to Symbol-Crunching in Computational Combinatorial Statistical Physics

Combinatorial Statistical Physics has several *impossible* problems that define it. For example, the Ising model in a magnetic field, percolation, and the enumeration of lattice animals, self-avoiding walks and self-avoiding polygons. Even though they are (probably) impossible, or at least intractable, there is a vast literature on them. One tries to find, both rigorously and non-rigorously, both exactly and approximately, important numbers associated with these problems, like connective constants, and more interestingly *critical exponents*. One also uses *number-crunching* to find *series expansions* that enable estimates of these important numbers. Another cottage industry is that of tractable *toy models*, that are subsets or supersets of the 'impossible-to-count' sets.

Even today there is a sharp dichotomy between 'theoretical research' that tries to find analytical solutions by pencil-and-paper human reasoning, and 'computational research' that uses the computer to crank-out important numbers. The advent of computer-algebra has enabled (e.g. the remarkable work of the Bordeux school [DV][B]) to use it as a *tool* for *solving* human-derived equations. In this research mode the *human* uses *human reasoning* (possibly aided by computers, but still with human

[1] Department of Mathematics, Temple University, Philadelphia, PA 19122, USA. `zeilberg@math.temple.edu` `http://www.math.temple.edu/~zeilberg/` . Feb. 25, 2000. Supported in part by the NSF. This article is accompanied by seven Maple packages, downloadable from `http://www.math.temple.edu/~zeilberg/tm.html` , where sample input and output files can also be found.

guidance) to *derive* the *equations* for the desired quantities. Once the equations are obtained, they are fed into the computer-algebra system (like Maple, Mathematica, etc.), that solves them.

But even *deriving* the set of equations whose solution would produce the desired generating function could be a very difficult, and often impossible, task for a mere human. The next natural step is to 'teach' (i.e. program) the computer to *find the equations, all by itself.*

Indeed, in order to take full advantage of the computer revolution, WE MUST TEACH THE COMPUTER HOW TO DO RESEARCH ON ITS OWN. The difficulty of this 'idea crunching' is that at present, Maple, Mathematica, and their likes are really only one notch above Fortran, C, and their like. In principle we can do symbolic computation in C and even in Fortran (or for that matter, even in machine language), after all, everything reduces to a Turing Machine, and in fact the core of Maple is actually written in C! However, from the user's interface point of view this is very awkward.

The same difficulty faces us right now, when we try to do, essentially, 'meta-symbolic computation', or, more accurately, 'idea-crunching' (idea-ics, for short). Doing 'idea-ics' in Maple is the higher-level analog of doing symbolics in Fortran. I am sure that in a few years we would have a Meta-Maple that would make the task of the present undertaking much easier. Conversely, it is hoped that this effort, and efforts like it, would stimulate and guide future system-developers.

Once this higher-level Maple (or Mathematica, etc.) will become available, it would be possible to state, for example, the following command (in the appropriate formal language, and perhaps even in English):

Write a Maple program that inputs an integer k and a variable s and outputs the rational function $\sum_{n=0}^{\infty} a_k(n) s^n$, where $a_k(n) :=$ the number of self-avoiding walks of length n that is confined to the strip $0 \leq y \leq k$.

We would also need to tell the computer what is a *self avoiding walk* but this can be done in one line in any formal language.

Since this Meta-Maple is not yet available, I had to spend a month of my precious time to *write the program by hand*, (which, incidentally, turned out to be more complicated than I anticipated). Even though this is very crude compared to the future, it is definitely progress compared to the previous efforts discussed below.

With the exception of Mikovsky's remarkable thesis[M], in past work, both for number- and symbol crunching, for example of the Australian (e.g. [BY], [CG]) and Bordelaise (e.g. [DV],[B]) schools, one had to find the 'alphabet' and 'grammar' (or equivalently the 'transfer matrix') by hand, for one k at a time (where k denotes the width of the counted creatures). Also, the source-code was normally not published, nor made available upon request. With the Maple packages accompanying this article, *anybody* with access to the web, and access to Maple (that can be purchased for less than $100), can download, for example, my Maple program `SAW`, go into Maple, and type, `read`

`SAW:`, followed by say, `GFW(5,s);`, and after a few minutes he or she would get the *exact answer* for the generating function for the sequence: the number of n-step self-avoiding walks, on the two-dimensional square lattice such that the largest y coordinates minus the smallest y coordinates is ≤ 5. He or she, can of course type `GFW(k,s)`, for any desired k. Even though, with current computers, it would be hopeless to get `GFW(100,s)` or even `GFW(12,s)`, it is very gratifying that *in principle* the program can do it. Also he or she can read and enjoy the source-code, and later modify it for different problems.

There is very little 'conceptual originality' in this article. The transfer-matrix method is a standard *tool of the trade* in this area, used in computational and theoretical work alike. So the "traditionalist" (who is usually computer-illiterate and hence unable, and very often also unwilling, to appreciate the effort that went into such an endeavor) might dismiss it as 'trivial', something to be published in "lowly" "software-engineering journals". To him I retort: don't be a theoretical snob! Don't you know that: THE IMPLEMENTATION IS THE MESSAGE.

And believe me, it was not easy! Especially for the Self-Avoiding Walks program (`SAW`), I had to *think* much harder than I did for my previous 'human' stuff! So here is another point I am trying to make (already made in my Opinion 37 [Z1]): Programming to do a specific task is difficult and important research! Much more important than proving, humanly, yet another theorem. True, the programs of this project will be superseded and made obsolete by future developments, (since, as I argued above, it would be enough to *state* the general problem to the computer, and it would *write the program* for you). But the proof of Fermat's Last Theorem will enjoy the same fate of obsolescence and 'triviality' as the present project. All *current* math proofs (and computer programs) would be completely computer-generated in less than fifty years. So, although Wiles's proof and my Maple programs are both destined to become obsolete in the not-too-distant future, I hold that, from a 'future-history' point of view, this article, and articles like it, constitute a more significant contribution to mathematics than the proof of Fermat's Last Theorem (I conceded that from a 'past-history' viewpoint, Wiles's proof wins).

Indeed, Wiles's proof does not contribute a iota to the really important problem that faces 21st-century mathematics: TEACH THE COMPUTER HOW TO DO RESEARCH. By contrast, my present effort is an admittedly modest, yet strictly positive, step in the right direction. It teaches the computer how 'to do research', albeit very narrowly-focused, in the sense that it does 'theoretical research' that goes beyond routine numeric and even symbolic computation. As I have already mentioned above, the computer is used not just to *solve* the humanly-generated or computer-aided *set of equations*, but it is used to *generate the equations* for an *infinite* family of problems, *ab initio*.

I know that this is a tiny advance, but Rome was not built in a day, and as we know from our experience of teaching human students, we have to teach, *step-by-step*, *very gradually* delegating more responsibility and independence to them. One of the reasons the original efforts at AI were such a flop was that we tried to teach the computer too much too soon, and also, in a true species-centric way, we tried to make 'artificial' intelligence in our own image. As we get to know the computer better, we would learn how to teach it better, and take advantage of its strengths.

But enough of prophecy. Let me now describe the specific contents of this paper. A substantial part of it is an enhancement, correction, and *extension* of a large part of Anthony Mikovsky's thesis[M], so I'll have to explain first why his work had to be redone.

A Critique of Anthony Mikovsky's 1997 Ph.D. Thesis

In a very impressive Penn 1997 Ph.D. thesis [M], under the direction of Herb Wilf, Mikovsky used Maple to derive generating functions for the enumeration of lattice animals and self-avoiding walks, confined to a finite strip, as well as convex polyominoes (that we do not consider here). The thesis is beautifully written, and the algorithms are described clearly both in English and in Maple. It is indeed a pioneering effort in computer-generated research in combinatorial statistical physics, but it has some weaknesses.

The major weakness is 'trivial' but nevertheless VERY IMPORTANT. The source code, while printed-out in full, is not available for download from the web! Much good does it do me! Typing is not my idea of fun! I much rather write my own program. Even worse, Mikovsky's thesis was never published in a journal, and the only way to get it is by paying (like I did) more than \$40 to University Microfilms.

A second weakness is long-winded human rambling. After describing the ideas behind the algorithms, he actually goes on to write down, in full detail, for each of the cases $k = 1, 2, 3$, the 'grammar' and set of equations, thereby wasting many pages. I agree that for pedagogical reasons it is a good idea to have the grammar and set of equations listed for the smallest and second-smallest cases, but the rest should be reproducible by the reader (or rather by her computer), if desired. We humans should learn not to micro-manage the computer too much, in particular, not ask to see intermediate results, and trust it. Of course, only after the program has been completely debugged.

A third weakness is the style of the Maple code. It does not use modularity, and has no procedures (subroutines). It starts out with the line : `rownum=?;`, where the user is supposed to fill-in the question-mark, rather than defining a procedure `SAW:=proc(rownum) local ...`, and using smaller procedures. Mikovsky gives very few comments, and the code is very hard to follow and verify.

Finally, at least for his Maple program on self-avoiding walks, Mikovsky failed to test the output against independently generated, or readily available, data. While his output and mine agree for lattice-animals, they disagree in the case of self-avoiding walks. According to his corollary 5.5 (p. 163), half the number of self-avoiding walks with 4 steps and width ≤ 3 equals 57, while it is well-known, (and easy to find by direct counting) that there are only 100 such walks altogether, and two of them have width 4, hence the true number is $(100 - 2)/2 = 49$ rather than 57. So his program must be flawed. By contrast, I know that my version (`SAW`), is correctly implemented, since the output was compared to published tables, and independently generated output from 'empirical' programs.

The Specific Goals and Results of This Work

In addition to the rather pompous 'justification' presented in the abstract, there are more down-to-earth reasons for taking the trouble. Thanks to the Maple package `ANIMALS` one can now compute the generating function for lattice-animals of width $\leq k$, for *arbitrary* k (at present it is feasible to go up to $k = 6$, but as computers get faster and bigger one should be able to go beyond). If one is content with *series expansions*, i.e. the first 200 (or whatever) terms of the generating functions, then one can go much further. Of course, in principle our programs can handle any integer k.

Thanks to the Maple package `SAP` one can now find generating functions, and series expansions, for *self-avoiding polygons* of width $\leq k$ for *any* k.

Thanks to the Maple package `SAW`, one can do the same for *self-avoiding walks* of bounded width.

Counting More Inclusive Classes of Creatures: Only restricting the width of individual 'letters'

Of course, the holy grail is the enumeration of the original 'impossible-to-count' objects, and the purpose of the studied 'toy models' is to approximate, both numerically and conceptually, the 'real-thing'. I noticed that once the transfer-matrix method for counting creatures confined to a strip is implemented correctly, then a slight modification of the 'Transfer Matrix' enables us to count the much wider class of creatures in which we do not insist that the whole creature should fit in a finite, fixed, strip, but only that each *individual vertical cross-section* has bounded width. The matrices get smaller (since there are fewer letters), and the 'connective constants' for these subclasses (which imply lower bound for the 'real' connective constant) are higher than their counterparts. In particular, we were able to improve the previous record of [WS] (see [F]) for the lower bound for 'Klarner's constant' (the connective constant for lattice-animals) from 3.791 (and its updated value using the extended series expansion, 3.8228, see [F]) to 3.8499.

The Maple packages enumerating these extended sets of creatures are `freeANIMALS`, `freeSAP`, and `freeSAW`.

Coming Up Soon: The Umbral Transfer Matrix Method

But the *main* reason for spending so much time developing Maple packages and implementing seemingly minor enhancements of largely known algorithms is that we need it as a starting point for the 'Umbral Transfer Matrix Method' [Z2]. Now, we no longer have a 'finite alphabet' but an 'infinite alphabet', and the 'approximation' both conceptually and computationally to the 'real things' is much better. Hence the Maple packages presented in this article are not the shortest possible, for the specified task. Rather they are written in a form that would facilitate climbing from the (finite) Transfer-Matrix method to the (infinite) Umbral Transfer-Matrix method.

In the present case we had a generalization in mind, but any project should be considered as a link in an infinite chain of consecutive generalizations, hence it should be written as clearly and modularly, as possible, and carefully documented, so that future researchers can use it both as a ‘black box’, or as a starting point for tweaking and modifying. Of course these precepts are part of the *practice of programming*, (see e.g. the classic [KP]), the trite-but-true dos and donts of the professional software engineer and developer. But even we, professional mathematicians but as yet amateur programmers, would do well to adhere to them. Within reason, of course, after all, we can't spend *all* our time programming and documenting.

The Transfer-Matrix Method

There are several equivalent formulations of the Transfer-Matrix Method. One can talk about ‘finite-automata’, a ‘Markov-Process’ , or a ‘type-3 grammar in normal form’, but the most straightforward way is in terms of weighted counting of paths in a directed graph.

We will follow Stanley[S], but things will be even simpler for us, since we don't have to know any linear algebra, because Maple does. All we have to do is know how to set up the system of equations.

Definition: A directed graph $(V, E, init, fin)$, consists of a finite set of "vertices", V, a finite set of "directed edges", E, and two functions $init : E \to V$ and $fin : E \to V$.

For $e \in E$ we say that e goes from vertex $init(e)$ to vertex $fin(e)$.

Definition: A path in a directed graph $(V, E, init, fin)$ is a sequence

$$v_1, e_1, v_2, \ldots, v_i, e_i, v_{i+1}, \ldots, e_k, v_{k+1}$$

with $v_i \in V, (1 \le i \le k+1)$, $e_i \in E, (1 \le i \le k)$, and $init(e_i) = v_i, fin(e_i) = v_{i+1}, (1 \le i \le k)$.

Definition: A Combinatorial Markov Process is a 6-tuple, $(V, E, init, fin, Start, Finish)$, where $(V, E, init, fin)$ is a directed graph defined above, and $Start$ and $Finish$ are subsets of V.

Definition: A Vertex-Weighted (resp. Edge-Weighted) Combinatorial Markov Process is a seven-tuple $(V, E, init, fin, Start, Finish, wt)$, where $(V, E, init, fin, Start, Finish)$ is a Combinatorial Markov Process and wt is a function from V (respectively E) to the positive integers.

Definition: The *weight* $Wt(P)$ of a path $P = v_1, e_1, v_2, e_2, \ldots, v_i, e_i, v_{i+1}, \ldots, v_k, e_k, v_{k+1}$, in a Vertex-Weighted Markov Process is: $Wt(P) := wt(v_1) + wt(v_2) + \ldots + wt(v_k) + wt(v_{k+1})$.

Definition: The *weight* of a path $P = v_1, e_1, v_2, e_2, \ldots, v_i, e_i, v_{i+1}, \ldots, v_k, e_k, v_{k+1}$, in an Edge-Weighted Markov Process is: $wt(e_1) + wt(e_2) + \ldots + wt(e_k)$.

Note that we could combine the two notions and put weight both on vertices and edges, but this is unnecessary, since the weights of the vertices can always be incorporated by modifying the weights of the edges. Please be warned that, unlike [S], weights of paths are additive, not multiplicative.

Example: if $V = \{1, 2, 3\}$ and $E = \{[1, 2]^1, [1, 2]^2, [2, 3], [3, 1]^1, [3, 2]^2\}$, $Start = \{1, 2\}$, $Finish = \{2, 3\}$, where an edge $[i, j]^k$ goes from i to j, and if, $wt(1) = 2$, $wt(2) = 5$, $wt(3) = 1$, then the weight of the path $P = 1, [1,2]^2, 2, [2,3], 3, [3,1]^2, 1, [1,2]^1, 2$ in this Vertex-Weighted Markov-Process is $Wt(P) = wt(1) + wt(2) + wt(3) + wt(1) + wt(2) = 2 + 5 + 1 + 2 + 5 = 15$.

Another Example: Let V, E, $Start$, and $Finish$, be as above, but let's make it into an Edge-Weighted Markov-Process with the weight function defined by $wt([1,2]^1) = 3, wt([1,2]^2) = 4, wt([2,3]) = 1, wt([3,1]^1) = 4$, $wt([3,2]^2) = 2$, then the weight of the above path P is $Wt(P) = wt([1,2]^2) + wt([2,3]) + wt([3,1]^2) + wt([1,2]^1) = 4 + 1 + 2 + 3 = 10$.

Our goal is to compute the weight-enumerator (generating function)

$$F(t) = \sum_{P} t^{Wt(P)} \quad ,$$

where the sum extends over the (usually infinite) set of paths in the directed graph that start with a vertex of $Start$ and ends with a vertex of $Finish$.

For any statement, [statement] equals 1 if it is true, 0 if it is false.

Let's first consider the Vertex-Weighted Markov Process, and assume that there are no multiple edges, i.e. there is at most one edge between any two vertices. For any $v \in V$, let $F_v(t)$ be the sum of $t^{Wt(P)}$, over all path P that start with v (even if v is not a member of $Start$), and that end with a vertex in $Finish$. Let $Followers(v)$ be the set of vertices v' connected to v by a single edge. We have

$$F_v(t) = [v \in Finish] \cdot t^{wt(v)} + t^{wt(v)} \sum_{v' \in Followers(v)} F_{v'}(t) \quad . \qquad (Eq1)$$

This gives us a system of $|V|$ equations for the $|V|$ unknowns $\{F_v(t) \,|\, v \in V\}$. We are guaranteed that it has a solution since we know that $F_v(t)$ exists as a formal power series. Once we (or rather Maple, or rather the computer) solves this system, we discard the $F_v(t)$ for which $v \notin Start$, and get the final result

$$F(t) = \sum_{v \in Start} F_v(t) \quad .$$

If we have multiple edges then the system of equations $(Eq1)$ should be modified to read

$$F_v(t) = [v \in Finish] \cdot t^{wt(v)} + t^{wt(v)} \sum_{v' \in Followers(v)} a(v,v') F_{v'}(t) \quad , \qquad (Eq2)$$

where $a(v,v')$ is the number of edges between v and v'.

For the Edge-Weighted Markov Process, we have the equation

$$F_v(t) = [v \in Finish] + \sum_{e, init(e)=v} t^{wt(e)} F_{fin(e)}(t) \quad . \qquad (Eq3)$$

Finding Series Expansions

Even though the system of equations $(Eq1)$ can always be solved, it often happens that the system is too big to be solved by Maple. It is then still possible to find what physicists call "series expansion" of the generating function $F(t)$, i.e. the first L terms of its power-series expansion: $F(t) = a(0)+a(1)t+a(2)t^2+\ldots+a(L)t^L+\ldots$, where $a(i)$ is the number of paths in the Combinatorial Markov Process whose weight equals i. Let $a_v(i)$ be the number of paths of weight i that start with the vertex v. Then

$$a(i) = \sum_{v \in Start} a_v(i) \quad .$$

The system of linear equations $(Eq1)$, for Vertex-Weighted simple Markov Processes is equivalent to the recurrences:

$$a_v(i) = [wt(v) = i] \cdot [v \in Finish] + \sum_{v' \in Followers(v)} a'_v(i - wt(v)) \quad , i \geq 0, \qquad (Eq1')$$

with the initial conditions $a_v(i) = 0$, when $i < 0$. The recurrence counterparts of $(Eq2)$ and $(Eq3)$ are similar.

Maple Representation of Combinatorial Markov Processes

In order to solve the system of equations $(Eq1)$, $(Eq2)$, or $(Eq3)$ we really don't need to know the 'nature' of the edges, only how many they are of any given weight. This leads to the notion of *profile* of a Weighted Combinatorial Markov Process.

Let's call a vertex-weighted Combinatorial Markov Process, with no multiple edges, a Markov Process of type I. Its *profile* is a list of length four: $[Start, Finish, ListOfOutgoingNeighbors, ListOfWeights]$. The length of $ListOfOutgoingNeighbors$ is the number of vertices, let's call it N. We assume that the vertices are labelled $\{1, 2, \ldots, N\}$. $Start$ and $Finsih$ are subsets of $\{1, 2, \ldots, N\}$. The i^{th} component of $ListOfOutgoingNeighbors$ $(i = 1, \ldots, N)$ is the set of vertices j such that there is an edge between i and j (recall that right now we assume that there is at most one edge between vertex i and j, for $1 \leq i, j \leq N$). Finally, $ListOfWeights$ is a list of length N whose i^{th} component $(i = 1, \ldots, N)$ is the weight of the vertex labelled i.

For example

$$MC1 := [\{1\}, \{1\}, [\{1\}], [1]] \quad ,$$

is a very simple example of the profile of a type I Markov Process. It has one vertex, of weight 1, and a single edge, going from that vertex to itself.

Let's call a vertex-weighted Combinatorial Markov Process, *with* multiple edges, a Markov Process of type II. Its *profile* is a list of length 4, $[Start, Finish, ListOfOutgoingNeighbors, ListOfWeights]$. The length of $ListOfOutgoingNeighbors$ is the number of vertices, let's call it N. We assume that the vertices are labelled $\{1, 2, \ldots, N\}$. $Start$ and $Finsih$ are subsets of $\{1, 2, \ldots, N\}$. The i^{th} entry of $ListOfOutgoingNeighbors$ $(i = 1, \ldots, N)$ is a *multiset* $j_1^{m_1} j_2^{m_2} \ldots j_k^{m_k}$ represented in the form

$\{[j_1, m_1], \ldots, [j_k, m_k]\}$, which means that out of the vertex labelled i there are m_1 edges going to j_1, m_2 edges going to j_2, etc. Finally, $ListOfWeights$ is a list of length N whose i^{th} component $(i = 1, \ldots, N)$ is the weight of the vertex labelled i.

For example the Markov Process represented by $MC1$ can also be viewed as a type II Markov Process with profile:

$$MC2 := [\{1\}, \{1\}, [\{[1, 1]\}], [1]] \quad .$$

Let's call a simple edge-weighted Combinatorial Markov Process, (i.e. without multiple edges), a Markov Process of type III. Its *profile*, is a list of length 3 $[Start, Finish, WeightedListOfOutgoingNeighbors]$. The length of the list $WeightedListOfOutgoingNeighbors$ is the number of vertices, let's call it N. We assume that the vertices are labelled $\{1, 2, \ldots, N\}$. $Start$ and $Finsih$ are subsets of $\{1, 2, \ldots, N\}$. The i^{th} component of $WeightedListOfOutgoingNeighbors$ $(i = 1, \ldots, N)$ is the set of pairs $[j, wt_{i,j}]$ such that there is an edge between i and j and $wt_{i,j}$ is the weight of the edge connecting vertex i to vertex j.

For example the Markov Process

$$MC3 := [\{1\}, \{1\}, [\{[1, 1]\}]] \quad ,$$

is a very simple example of the profile of a type III Markov Process. It has one vertex and a single edge, of weight 1, going from vertex 1 to itself.

For a slightly bigger example consider:

$$MC3a := [\{1\}, \{1, 2\}, [\{[1, 3], [2, 4]\}, \{[1, 2], [2, 8]\}]] \quad ,$$

Here the underlying digraph has two vertices labelled 1 and 2. A path must start at the vertex 1, but may end at either 1 or 2. Out of 1 there is one edge, of weight 3 going into itself, and an edge, of weights 4 going into 2. Out of 2 there an edge, of weight 2, going to 1, and an edge, of weight 8, going back to 2.

Let's call an edge-weighted Combinatorial Markov Process, with possibly multiple edges, a Markov Process of type IV. Its *profile*, w.r.t. the variable t, is a list of length three, $[Start, Finish, WeightedListOfOutgoingNeighbors]$.

The length of $WeightedListOfOutgoingNeighbors$ is the number of vertices, let's call it N. We assume that the vertices are labelled $\{1, 2, \ldots, N\}$. $Start$ and $Finsih$ are subsets of $\{1, 2, \ldots, N\}$. The i^{th} component of $WeightedListOfOutgoingNeighbors$ $(i = 1, \ldots, N)$ is the set of pairs $[j, poly_{i,j}(t)]$ such that there is at least one edge between i and j, and $poly_{i,j}(t)$ is the weight-enumerator of the set of edges that go from i to j, i.e. the coeff. of t^k in $poly_{i,j}(t)$ is the number of edges from i to j that have weight k.

For example the Markov Process

$$MC4 := [\{1\}, \{1\}, [\{[1, t]\}]] \quad ,$$

is a very simple example of the profile of a type IV Markov Process. It has one vertex and a single edge, of weight 1, going from that vertex to itself.

For a slightly bigger example consider:

$$MC4a := [\{1\}, \{1, 2\}, [\{[1, t], [2, t^2 + t^3]\}, \{[1, t^2], [2, 3t^5 + 2t^7]\}]] \quad ,$$

Here the underlying digraph has two vertices labelled 1 and 2. A path must start at the vertex 1, but may end at either 1 or 2. Out of 1 there is one edge, of weight 1 going into itself, and two edges, of weights 2 and 3 going into 2. Out of 2 there is one edge, of weight 2, going to 1, and five edges, three of weight 5 and two of weight 7, going back to 2.

The Maple package `MARKOV` computers the generating functions that weight-enumerate paths in Combinatorial Markov Processes of types I-IV.

A User's Manual for `MARKOV`

First you should go to my website, and download `MARKOV`. Assuming that you are still in the same directory, go into Maple by typing: `maple` (or `xmaple`), followed by `Enter`, or click on the Maple icon.

Once in Maple, type: `read MARKOV;` , and follow the instructions given there. The main procedures are `SolveMC1`, `SolveMC2`, `SolveMC3`, `SolveMC4`, that find the generating functions weight-enumerating paths in Combinatorial Markov Processes of type I, II, III, and IV respectively.

For example, for the Markov Processes above, do `SolveMC1(MC1,t);`, `SolveMC2(MC2,t);`, `SolveMC3(MC3,t);`, `SolveMC4(MC4,t);`, to get, in all cases, $-t/(t-1)$.

If the input Markov Processes are too big for `SolveMC1` and its siblings to handle, try `SolveMC1series`, etc. to find the first L (for a specified L) terms of the power-series expansion of the corresponding generating function. For example `SolveMC1series(MC1,10);` should output $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$.

A Brief Explanation of the Nuts and Bolts of `MARKOV`

We will only describe `SolveMC1`, since the other procedures are straightforward modifications. Please refer to the source code of the package `Markov`.

`SolveMC1(MC,s)` inputs MC, the profile of a type I Markov Process, and a variable `s`. First the program checks that `s` is indeed a variable, and `MC` is indeed a legal type I Markov Process, using the procedure `KosherMC1`.

The line "`LL:=MC[1]:RL:=MC[2]:Nei:=MC[3]:Wts:=MC[4]:` " unpacks, MC. `LL` is the set vertices that we call *Start*, `RL` is the set of vertices *Finish*, *Nei* is the list of neighbor-sets and `Wts` is the list of weights. The line "`N:=nops(Wts):`" finds the number of vertices. `eq` and `var` are the set of equations and variables, respectively, that are initialized to the empty set. We use the indexed variable `A[i]` for what we called above F_v, i.e. for the weight-enumerator of paths that start at the vertex labelled `i`.

The program now do-loops over `i` from `1` to `N`, adding, one at a time, `A[i]` to `var` and constructing `eq1` the equation representing paths that start at the vertex labelled `i`. If `i` is a member of `RL` (i.e. it belongs to *Finish*, i.e. the length-0 path that starts and ends at i and has no edges, is to be counted), then `eq1` is initialized to be `s**(Wt[i])` , the weight of this trivial path. `Sakh` is `Nei[i]`, the set of neighbors of the vertex `i`. The inner do-loop, w.r.t `j`, sums `A[Sakh[j]]`, over all outgoing neighbors , `Sakh[j]` of `i`. We call the result `lu`. Then the current equation, `eq1`, is updated in the next line: `eq1:=eq1-s**Wts[i]*lu:`, and the new equation, `eq1`, that considers all paths that start at vertex `i`, is added to the set of equations, `eq` with the command: "`eq:=eq union {eq1}:`". Exiting the `i` do-loop, equipped with the set of linear equations, `eq`, and the set of unknowns, `var`, we ask Maple to solve the system with the line: "`var:=solve(eq,var):`". The last do-loop, w.r.t. `i`, sums the solved values for the weight-enumerators of paths that start at one of the allowed starting points, the members of *Start*, that is called `LL` in this program. The very last line normalizes the answer.

Counting Lattice Animals Confined to a Strip

A (2D site) *lattice-animal* (alias fixed polyomino, henceforth animal) is a connected set of lattice points in the square lattice Z^2.

Two animals are equivalent if they are translations of each others. The problem is to find $a(n)$, the number of equivalence classes (under translation) of animals with n cells. For example $a(1) = 1$, $a(2) = 2$, $a(3) = 6$, $a(4) = 19$ etc. To date (see Steve Finch's superb website [F]) $a(n)$ is known for $n \le 28$. It is probably hopeless to look for a 'closed-form' formula, or even for a polynomial-time algorithm for computing $a(n)$, so, one should be content, at present, with counting interesting subsets and supersets, that yield as a bonus, lower and upper bounds, for the so-called *Klarner* constant: $\mu := \lim_{n\to\infty} a(n)^{1/n}$. Klarner ([K1], see also [K2]) proved the existence of his constant and that $\mu > 3.72$. With Rivest ([KR]) he proved that $\mu < 4.65$.

It was Read [R] who first proposed to use the transfer-matrix method to enumerate animals confined to a strip of width k. This was nicely implemented by Mikovsky [M]. Here we will use a different data structure, which logically is equivalent, but that would be amenable to the generalization that I hope to present in [Z2].

For a given integer k, consider all animals $S = \{(x, y)\}$, with the restriction that for each $(x, y) \in S$, $x \ge 0$, $0 \le y < k$, and $\min\{x \,|\, (x, y) \in S\} = 0$. We can look, for $x = 0, 1, \ldots$, at the set of corresponding y coordinates with the specified x for which $(x, y) \in S$. This yields a word in the alphabet consisting of the non-empty subsets of $\{0, 1, \ldots, k-1\}$. For example the animal

$$\{(0,1),(0,2),(0,4),(1,0),(1,2),(1,4),(1,5),(2,0),(2,2),(2,3),(2,4),(3,0),(3,1),(3,2)\}$$

can be coded as the word

$$\{1,2,4\},\{0,2,4,5\},\{0,2,3,4\},\{0,1,2\} \quad .$$

While this is an injection into the set of words in the alphabet whose letters are the $2^k - 1$ non-empty subsets of $\{0, \ldots, k-1\}$, it is impossible to construct a reasonable (type-3) 'grammar' describing this language. Following Read[R], we need a larger alphabet.

For an animal S, and for an integer $i = 0, 1, 2, \ldots$, consider the subset of S, S_i, obtained by only retaining the points of S with $x \leq i$:

$$S_i := \{(x, y) \in S \mid x \leq i\}$$

For example, for the animal above,

$$S_0 = \{(0,1), (0,2), (0,4)\} \quad , \quad S_1 = \{(0,1), (0,2), (0,4), (1,0), (1,2), (1,4), (1,5)\} \quad ,$$

$$S_2 = \{(0,1), (0,2), (0,4), (1,0), (1,2), (1,4), (1,5), (2,0), (2,2), (2,3), (2,4)\} \quad ,$$

and

$$S_3 = S = \{(0,1), (0,2), (0,4), (1,0), (1,2), (1,4), (1,5), (2,0), (2,2), (2,3), (2,4), (3,0), (3,1), (3,2)\} \quad .$$

Note that the S_i do not have to be animals, since removing the points with $x > i$ may disconnect the animal S. For each vertical cross section $x = i$, the points of S for which $x = i$ can be partitioned into equivalence classes according to whether or not they belong to the same component of S_i. So the natural alphabet is not just subsets of $\{0, 1, \ldots, k-1\}$ but certain set-partitions. According to this coding, the animal S is coded as the 'word':

$$\{\{1,2\}, \{4\}\}, \{\{0\}, \{2\}, \{4,5\}\}, \{\{0\}, \{2,3,4\}\}, \{\{0,1,2\}\} \quad .$$

It is easy to see that not all set-partitions show up. First, if (i, j) and $(i, j+1)$ both belong to S then they must belong to the same component. Hence it is more beneficial, both conceptually and computationally, to abbreviate the set of $j - i + 1$ consecutive integers $\{i, i+1, i+2, \ldots, j-1, j\}$, by the "interval-notation": $[i, j]$. So from now on we will denote sets of integers as unions of such closed intervals. For example the set $\{1, 2, 3, 5, 7, 8, 9, 11, 12, 13, 17, 19, 20, 22\}$ will be denoted by $\{[1,3], [5,5], [7,9], [11,13], [17,17], [19,20], [22,22]\}$. Because of the observation above, the kind of set-partitions that arise from lattice animals (in which all the members of a closed interval must belong to the same component), can be written as set-partitions of intervals. For example, in the new notation the animal S above is coded as the word:

$$\{\{[1,2]\}, \{[4,4]\}\}, \{\{[0,0]\}, \{[2,2]\}, \{[4,5]\}\}, \{\{[0,0]\}, \{[2,4]\}\}, \{\{[0,2]\}\} \quad .$$

It is easy to see that the set-partitions that show up as letters coding lattice animals by the above process are all *non-crossing*, but this fact is not needed, since the computer can always find the complete alphabet, once it knows the "left-alphabet", i.e. the letters that may be starters, and how to compute the *followers* of a letter. So the next task is to decide who is eligible to be a first (i.e. leftmost) letter, in an animal word. Two intervals $[a, b]$ and $[c, d]$ belong to the same block in the set-partition that constitutes the $(i+1)^{th}$ letter of an animal-word, if and only in the animal it came from, the set of lattice points $\{(i,a),(i,a+1),\ldots,(i,b)\}$ is "connected from the left" to the set of latticed points $\{(i,c),(i,c+1),\ldots,(i,d)\}$. In other words, belong to the same component in the set obtained from the original animal by only retaining the points with $x \leq i$. When $i=0$, then there is nothing to the left, and hence the left-letters must be set-partitions of intervals each of whose blocks are singletons. For example when $k=1$ there is only one left-letter, $\{\{[0,0]\}\}$. When $k=2$ then we have the three letters$\{\{[0,0]\}\},\{\{[1,1]\}\},\{\{[0,1]\}\}$. When $k=3$ then we have the seven letters $\{\{[0,0]\}\},\{\{[1,1]\}\},\{\{[0,1]\}\},\{\{[0,2]\}\},\{\{[1,2]\}\},\{\{[2,2]\}\},\{\{[0,0]\},\{[2,2]\}\}$

In general there are 2^k-1 left-letters corresponding to all non-empty subsets of $\{0,\ldots,k-1\}$, written in interval notation, in which each participating interval forms its own singleton block in the set-partition.

What animal-letters can be the last (rightmost) letter? Obviously, all the intervals in the rightmost cross-section must belong to the same component, since there are no points to the right to help them out. Hence the rightmost letters consist of set-partitions consisting of one block. Their number again is 2^k-1. For example when $k=1$ there is only one right-letter, $\{\{[0,0]\}\}$. When $k=2$ then we have the three letters$\{\{[0,0]\}\},\{\{[1,1]\}\},\{\{[0,1]\}\}$. When $k=3$ then we have the seven letters $\{\{[0,0]\}\},\{\{[1,1]\}\},\{\{[0,1]\}\},\{\{[0,2]\}\},\{\{[1,2]\}\},\{\{[2,2]\}\},\{\{[0,0],[2,2]\}\}$.

Given a letter in the animal-alphabet, who can follow it? Let's call a *pre-letter* any non-empty subset of $\{0,1,\ldots,k-1\}$, written in interval notation. Of course there are 2^k-1 pre-letters. As we saw above, the set-partitions all of whose blocks are singletons correspond to leftmost letters, and the set-partitions having exactly one block correspond to rightmost letters. So we must now answer the question: What pre-letters can follow a given letter?

Since the partial animals (obtained by only retaining the points to the left of the considered vertical-cross section) must be all "connected from the right", we must make sure that each of the components of the letter has non-empty intersection with the pre-letter that comes right after it. For example, after the letter

$$\{\{[0,1],[3,3]\},\{[5,8],[15,18],[21,24]\},\{[10,13]\}\},$$

the pre-letter $\{[4,8],[10,26]\}$ may *not* follow since the component $\{[0,1],[3,3]\}$ has nothing to hang on to. Also the pre-letter $\{[0,4],[11,14],[19,20]\}$ may *not* follow, since now the component $\{[5,8],[15,18],[21,24]\}$ does not intersect this pre-letter. On the other hand the pre-letter $\{[1,1],[12,12],[22,22]\}$ may follow, since each of the three components gets touched.

Once we have a letter, and a legitimate pre-letter follower, the pre-letter becomes a letter in a unique way. Indeed, given a letter, and a pre-letter following it, define an (undirected) graph whose vertices are the intervals of the pre-letter and where there is an edge between two such intervals if they both touch the same component. The set-partition induced from the connected components of this graph is the desired follow-up letter. For example, the pre-letter $\{[2,6],[12,16],[24,24]\}$ following the letter

$$\{\{[0,1],[3,3]\},\{[5,8],[15,18],[21,24]\},\{[10,13]\}\},$$

becomes $\{\{[2,6],[12,16],[24,24]\}\}$.

Now that we know how to find all the followers of any letter, we can dynamically construct the table of followers of each letter that shows up, and at the same time keep track of our current set of letters, and keep going until there are no new letters encountered. This is accomplished by our Maple package `ANIMALS` described below.

A User's Manual for `ANIMALS`

First download `ANIMALS` to your directory (either directly from INTEGERS or from my website). Then go into Maple by typing: `maple` (or, if you prefer, `xmaple`) followed by `Enter`, or click on the Maple icon. Then, once in Maple, type: `read ANIMALS;`, assuming you are still in the same directory, or, e.g. `read 'research/animals/ANIMALS' ;` (i.e. the full path-name of the file `ANIMAL`) if you are not.) Then follow the on-line help. To see the names of the main procedures type: `ezra();` . To get help on a specific procedure type `ezra(ProcName);` . For example, to get help on `GF` type `ezra(GF);` .

To find the generating function that enumerates animals of width ≤ 3, type `GF(3,s);` . To get the number of animals with n cells, for $1 \leq n \leq L$, type $Khaya(L)$. I was able to get $Khaya(18)$, but $Khaya(20)$ took too long.

A Brief Explanation of the Nuts and Bolts of `ANIMALS`

The procedure `LeftLetters(a,b)` outputs all set-partitions of $\{a, a+1, \ldots, b\}$ (written in interval notation, see above) with one interval per block. The procedure `Followers(n,LETTER)` finds all the possible followers of the letter `LETTER`, in the alphabet of animals confined to the strip $0 \leq y \leq n-1$. It does that by testing each of the preletters (obtained from procedure `PreLeftLetters` by calling `PreLeftLetters(0,n-1)`). If every component of `LETTER` is touched by the tested pre-letter, `mu1`, it is approved, and it is made into a letter by procedure `PreLetToLet`, by the call `PreLetToLet(Let1,mu1)`.

Procedure `PreLetToLet(Letter,PreLetter)` inputs a letter `Letter` (a set-partition of intervals) and a pre-letter `PreLetter` (a set of intervals) and decides how to turn `PreLetter` into a letter. It defines two tables `T` and `S`, defined on the blocks of the set-partition `Letter` and the members of the set `PreLetter` respectively. At the end of the do-loop `T[block]` is the set of members of `PreLetter` touching the block `block`, and `S[member]` is the set of blocks of `Letter` touching the given `member` of `PreLetter`.

If any of the `T[component]` is empty then `PreLetter` is rejected by returning `0`. Otherwise, we construct a graph `G` whose vertex-set is the set of members of `PreLetter`. The graph is given in terms of an adjacency table and the last double do-loop constructs the graph where `interval1` is adjacent to `interval2` if they both touch at least one of the components of `Letter`, i.e. if `S[interval1] intersect S[interval2]` is non-empty. Finally a call to the procedure `Comp`, `Comp(G,PreLetter)`, gives the induced set-partition of the set of intervals `PreLetter`, by partitioning it according to the connected components of the graph `G`. The procedure `Support` simply finds the support of a letter, for example `Support`$(\{\{[0,2],[4,5]\},\{[7,7]\}\})$ should yield $\{0,1,2,4,5,7\}$.

Procedure `FollowersS` takes as input a set of letters, and outputs the union of all the followers. Procedure `Alphabet(n)`, starting with `LeftLetters(0,n-1)`, applies `FollowersS` iteratively until we get all letters. Since `Followers` is with `option remember`, by the time `Alphabet(n)` is finished, Maple already knows all the followers of all the letters.

Procedure `MarCha(n)` constructs the profile of the type I Markov Process describing the animals confined to the strip $0 \leq y \leq n-1$, in canonical, compact form. First a call to `Alphabet`, `gu:=Alphabet(n);` , gets the alphabet. By this time Maple already knows all the followers, but in terms of their long-winded set-partition names. To compactify it, the letters are given integer names from `1` to `N:=nops(gu)`, and the table `T` remembers the new names. Then the set of left-letters, `Left0` is found, and `Left1` and `Right1` become the set of left-letters and right-letters using the new, abbreviated names. The list `mu` is the list of lists where the i^{th} term is the set of followers, using their new, abbreviated names, of the letter whose abbreviated name is i. Finally `Wts` is the table of weights of the letters. The output is the type I Markov Process profile `[Left1,Right1,mu,Wts]`.

Procedure `gf(n,s)` uses `SolveMC1` to find the generating function for all animals that live in the strip $0 \leq y \leq n-1$, but what we are really interested in is the generating function for all animals of width $\leq n$, since, `gf` counts, for example both the animal $\{(0,0)\}$ and the animal $\{(0,1)\}$ as distinct entities, even though they are equivalent. This is achieved by `GF(n,s)` which is simply `gf(n,s)-gf(n-1,s)`.

If you are interested in the generating function for animals with width *exactly* n, then use `Gf(n,s);`, which is `GF(n,s)-GF(n-1,s)`.

Once $n \geq 6$, it takes too long, at least for Shalosh, to compute `GF(n,s)` exactly, but one can go much further with `GFseries(n,L)` which uses `SolveMC1series` to find the series expansion up to L terms. `GfseriesS(n,L,s)` uses `SolveMC1seriesS` to find the series-expansion where we also keep track of the length of the animal (alias the number of letters in the animal word). In particular the coefficient of x^i in the L^{th} entry of `GfseriesS(n,L,x)`, let's call it $A(L,n,i)$, is the number of animals with L cells, of width *exactly* n, and length *exactly* i. Since width+length$\leq L$, by symmetry the total number of animals with L cells equals $A(L,L/2,L/2)[k \equiv 0 \ mod\ 2]$ plus twice the sum of $A(L,n,i)$ with $n < i \leq L/2$. This is how `Khaya(L)` computes the first L terms of the notorious animal-series. `Khaya(18)` is fairly fast, but `Khaya(20)` already takes too long.

More Animals For Less Money: Counting Locally-Skinny Animals

The Maple package `ANIMALS` counted animals that are *globally* skinny, i.e. the whole animal can be fitted in a *fixed* strip. Let's define, for $x = 0, 1, \ldots$

$$M(x) := max\{y|(x,y) \in S\} \quad , \quad m(x) := min\{y|(x,y) \in S\} \quad .$$

The package `ANIMALS` described above counts, for a given n, the number of animals such that

$max_x(M(x)) - min_x m(x) \leq n-1$.

By contrast, the Maple package `FreeANIMALS`, to be described in this section, counts the much larger subset of animals such that $M(x) - m(x) \leq n-1$ for *each* x. For example the 'staircase' animal: $\{(0,0),(0,1),(1,1),(1,2),(2,2),(2,3),(3,3),(3,4),(n-1,n-1),(n-1,n)\dots\}$ is not counted by `GF(n,s)` of `ANIMALS`, but is already counted by `GF(2,s)` of `FreeANIMALS`, since each vertical cross-section, individually, had width ≤ 2.

The modification is straightforward. Now we have a normalized alphabet, where the lowest member must be 0. So we have half as many left-letters for a given n. Unlike the previous case, however, given a letter of width a, say, and a pre-letter of width b following it, the bottom of the pre-letter is placed, in turn at $y = -(b-1), -(b-2), \dots, 0, \dots, a-1$, and for each of these vertical translates we determine whether it is a legal interface, and if yes, what is the corresponding letter. For each of these resulting letters, we normalize them so that the bottom is 0. So now, in general, every letter has a *multi-set* of followers, and we have to use the Type II Markov Process Model.

Example: Consider the letter $\{\{[0,0]\},\{[2,2]\}\}$, and the pre-letter $\{[0,3]\}$. The translates $\{[-3,0]\}$,$\{[-2,1]\}$, $\{[1,4]\}$,$\{[2,5]\}$, do not produce bona-fide followers, while $\{[-1,2]\}$, $\{[0,3]\}$ do both give rise to the letter $\{\{[0,3]\}\}$. Hence in the digraph of this Markov Process, we have two edges between $\{\{[0,0]\},\{[2,2]\}\}$ and $\{\{[0,3]\}\}$. On the other hand there are six edges between $\{\{[0,0],[2,2]\}\}$ and $\{\{[0,3]\}\}$, since all the translates $\{[-3,0]\}$,$\{[-2,1]\}$,$\{[-1,2]\}$,$\{[0,3]\}$,$\{[1,4]\}$, $\{[2,5]\}$, are legal followings and all of them normalize to $\{\{[0,3]\}\}$. In general the followers that result from all the vertical translates of a pre-letter do not have to be the same.

A User's Manual for the Maple Package FreeANIMALS

The main procedures are: `gf, gfSeries, gfList, gfSeriesList` .

For a positive integer n, and a variable s, typing `gf(n,s);` would give the generating function

$$f_n(s) := \sum_{i=1}^{\infty} a_n(i) s^i \quad ,$$

where $a_n(i)$ is the number of (2D site) animals such that for each vertical cross-section $x = x_0$ the difference between the biggest y such that (x_0, y) belongs to the animal and the smallest such y is $\leq n-1$.

`gf(n,s)` works, on my computer, up to $n = 6$. Beyond that, you may wish to use `gfSeries(n,L)`, where L is also a positive integer, in order to get the first L coefficients of $f_n(s)$.

Both `gf(n,s)` and `gfSeries(n,L)` enumerate animals where each vertical cross-section has width $\leq n$. But we may want to consider more general sets of animals. Suppose that whenever the "letter" (i.e. vertical cross-section) has i boards, then it is allowed to have width $\leq R_i$, for a list $[R_1, R_2, \dots, R_m]$, say. The corresponding generating functions are given by typing `gfList(List,s);` and `gfSeriesList(List,s);`. For example `gfList([5,3],s);` would give the generating function for animals whose vertical cross-sections with one interval (board) have width ≤ 5 and vertical cross-sections with two intervals have width ≤ 3, or equivalently, the "free animal-language" that only uses the letters $\{\{[0,1]\}\}, \{\{[0,2]\}\}, \{\{[0,3]\}\}, \{\{[0,4]\}\}$, $\{\{[0,0],[2,2]\}\}, \{\{[0,0]\},\{[2,2]\}\}$.

A Brief Explanation of the Nuts and Bolts of `FreeANIMALS`

Now `Alphabet(n);` gives the smaller set of normalized letters, where the smallest integer is always 0. `MarCha(n)` gives the Markov Process for the language of the kind of animals considered here, but unlike `ANIMALS` it is what we called a Markov Process of type II, i.e. the underlying graph is still vertex-weighted, but multiple edges are allowed. Consequently, `Followers(n,Letter)` does not return a set but a list (that allows duplication). Procedure `ListToMultiSet` simply converts from list to a mutliset. For example `ListToMultiSet([1,2,2,1,2,1,1]);` yields $\{[1,4],[2,3]\}$, since 1 shows up 4 times and 2 shows up 3 times.

Counting Skinny Self-Avoiding Polygons

We will only consider the two-dimensional square-lattice, but the approach can be adapted to other plane lattices, and somewhat less obviously, to higher dimensions.

In the sequel we will use lower-case: *vertices* and *edges*, to talk about vertices and edges that live in the two-dimensional square-lattice, i.e. the natural habitat where self avoiding polygons roam. We will use Capitals: *Vertices* and *Edges* to talk about vertices and edges in the type III Markov Process describing the structure of these self-avoiding-polygons.

A (vertex) lattice point will be denoted by an (ordered) pair of integers, $[m,n]$, and, somewhat cumbersomely, but convenient for the computer implementation, an edge will be denoted by the (unordered) pair consisting of its end-points. For example the horizontal edge that joins $[4,5]$ and $[5,5]$ will be denoted by $\{[4,5],[5,5]\}$.

A *self-avoiding polygon* (henceforth sap) in a lattice, is a simple closed "curve" in the lattice. Equivalently, a sap is a set of edges of the square-lattice such that, in the induced graph, every vertex had degree exactly two, i.e. every vertex is adjacent to exactly two edges. We are interested in the sequence $a(n)$:= the number of saps with $2n$ edges, up to translation equivalence. For example

$$\{\{[0,0],[1,0]\},\{[1,0],[1,1]\},\{[1,1],[0,1]\},\{[0,1],[0,0]\}\}$$

is the only (up to trivial translation-equivalence) sap with 4 edges, while

$$\{\{[0,0],[1,0]\},\{[1,0],[2,0]\},\{[2,0],[2,1]\},\{[2,1],[1,1]\},\{[1,1],[1,0]\},\{[1,0],[0,0]\}\}$$

and

$$\{\{[0,0],[1,0]\},\{[1,0],[1,1]\},\{[1,1],[1,2]\},\{[1,2],[0,2]\},\{[0,2],[0,1]\},\{[0,1],[0,0]\}\}$$

are the only saps with 6 edges.

Let $vert(S)$ denote the set of vertices (lattice-points) that participate in a the sap S. For example

$$vert(\{\{[0,0],[1,0]\},\{[1,0],[1,1]\},\{[1,1],[0,1]\},\{[0,1],[0,0]\}\}) = \{[0,0],[1,0],[0,1],[1,1]\}$$

Let's standardize, and take, from each translation-equivalence class, the unique sap S, such that $\min_x\{(x,y) \in S\} = 0$ and $\min_y\{(x,y) \in S\} = 0$. In other words we place it such that its "left-most walls" lie on the y-axis and its "bottom floors" lie on the x-axis.

For any integer K, we would like to find the generating function

$$f_K(s) := \sum_{n=0}^{\infty} a_K(n) s^{2n} \quad ,$$

where $a_K(n)$ is the number of "skinny" saps with $2n$ edges, i.e. standardized saps such that $\max_y\{(x,y) \in vert(S)\} \leq K$.

It is possible to code any (standardized) sap S, as a "word" in its vertical cross-sections, i.e. for $k = 1, 2, \ldots$, write it as $A_0A_1A_2\ldots$, where the k^{th} letter is the subset of the horizontal edges of the sap that form $\{[k-1,y],[k,y]\}$. Since k is predetermined, it is enough to list it as a word in the alphabet consisting of the non-empty subsets of $\{0,1,\ldots,K\}$. Also the vertical edges are implied, even though the "letters" only list the horizontal edges. However, it is impossible to describe a reasonable "grammar", definitely not a type-3, Markovian one, on the language obtained from all skinny saps. Hence, just as with animals, we have to extend the alphabet to contain more information.

Looking at the vertical cross-section obtained by intersecting the sap with $\{k-1 \leq x \leq k\}$, i.e. at the above set of horizontal edges, lets's call it H_k, $H_k := \{\{[k-1,y],[k,y]\}\}$, we see that every member of H_k has a unique "mate", within H_k, such that it is possible to walk counter-clockwise, from the lower one to the upper one, along the sap, such that all the intermediate vertices are to the left of the vertical line $y = k-1$. It is also obvious, on geometrical grounds (the discrete Jordan Curve Theorem), that these pairings form a *legal bracketing*. Thus the size of the alphabet is

$$\sum_{i=1}^{[(K+1)/2]} \binom{K+1}{2i} C_i \quad ,$$

where $C_i := \binom{2i}{i}/(i+1)$ are the Catalan numbers.

We will describe legal bracketings by using the letters L and R, to denote "left bracket" and "right bracket" respectively. The only legal bracketing of length 2 is $[L,R]$. The two legal bracketings of length 4 are $[L,R,L,R]$ and $[L,L,R,R]$, etc. Recall that the set of legal bracketing consists of the empty word and all words in the alphabet $\{L,R\}$, that may be written as Lw_1Rw_2, where w_1, w_2 are shorter legal bracketings.

A typical, say, k^{th} "letter", in the sap-alphabet consists of a pair of lists of the same length, that length being an even integer. The second list is the increasing sequence of integers $[i_1,\ldots,i_{2r}]$

indicating which horizontal edges join the vertical lines $x = k-1$ and $x = k$, i.e. the edges $\{[k-1, i_1], [k, i_1]\}, \{[k-1, i_2], [k, i_2]\}, \ldots, \{[k-1, i_{2r}], [k, i_{2r}]\}$. The first list is the corresponding legal-bracketing, indicating the pairing according to "accessibility from the left". So if i_n is an "L", and its R-mate is i_m, then it is possible to walk along the sap, between edge $\{[k-1, i_n], [k, i_n]\}$ and $\{[k-1, i_m], [k, i_m]\}$, without ever venturing to the right of the vertical line $x = k-1$.

For example, the alphabet for saps confined to the horizontal strip $0 \leq y \leq 4$, (i.e. $K = 4$ above), consists of the following $\binom{5}{2}C_1 + \binom{5}{4}C_2 = 20$ letters:

$$\{[[L,R],[0,1]],[[L,R],[0,2]],[[L,R],[1,2]],[[L,R],[0,3]],[[L,R],[2,3]],[[L,R],[1,3]],$$

$$[[L,R,L,R],[0,1,2,3]],[[L,L,R,R],[0,1,2,3]],[[L,R,L,R],[0,1,2,4]],$$

$$[[L,R,L,R],[0,1,3,4]],[[L,R,L,R],[0,2,3,4]],[[L,R,L,R],[1,2,3,4]],[[L,L,R,R],[0,1,2,4]],$$

$$[[L,L,R,R],[0,1,3,4]],[[L,L,R,R],[0,2,3,4]],[[L,L,R,R],[1,2,3,4]],[[L,R],[0,4]],[[L,R],[2,4]],$$

$$[[L,R],[3,4]],[[L,R],[1,4]]\} \quad .$$

The leftmost letters, in a sap-word are not arbitrary, but necessarily must have its first list be of the form $[(L,R)^r]$, for $r = 1, \ldots, [(K+1)/2]$. Hence there are exactly

$$\sum_{i=1}^{[(K+1)/2]} \binom{K+1}{2i} \quad ,$$

letters that may be starters.

For example amongst the 20 letters above only the following $\binom{5}{2} \cdot 1 + \binom{5}{4} \cdot 1 = 15$ may start a sap word:

$$\{[[L,R],[0,1]],[[L,R],[0,2]],[[L,R],[1,2]],[[L,R],[0,3]],[[L,R],[2,3]],[[L,R],[1,3]],$$

$$[[L,R,L,R],[0,1,2,3]],[[L,R,L,R],[0,1,2,4]],$$

$$[[L,R,L,R],[0,1,3,4]],[[L,R,L,R],[0,2,3,4]],[[L,R,L,R],[1,2,3,4]],$$

$$[[L,R],[0,4]],[[L,R],[2,4]],[[L,R],[3,4]],[[L,R],[1,4]]\} \quad .$$

The rightmost letters, in a sap-word are not arbitrary either, but necessarily must have its first list, irreducible, i.e. of the form $[L, \phi, R]$, where ϕ is legal. In other words the mate of the first L must be the last R.

Hence there are exactly

$$\sum_{i=1}^{[(K+1)/2]} \binom{K+1}{2i} C_{i-1} \quad ,$$

letters that may be finishers. For example amongst the 20 letters above only the following $\binom{5}{2} \cdot C_0 + \binom{5}{4} \cdot C_1 = 15$ may end a sap word:

$$\{[[L,R],[0,1]],[[L,R],[0,2]],[[L,R],[1,2]],[[L,R],[0,3]],[[L,R],[2,3]],[[L,R],[1,3]],$$
$$[[L,L,R,R],[0,1,2,3]],[[L,L,R,R],[0,1,2,4]],$$
$$[[L,L,R,R],[0,1,3,4]],[[L,L,R,R],[0,2,3,4]],[[L,L,R,R],[1,2,3,4]],[[L,R],[0,4]],[[L,R],[2,4]],$$
$$[[L,R],[3,4]],[[L,R],[1,4]]\} \quad .$$

The next item on the agenda is: "Which letters can follow a given letter?". This will be accomplished in three stages. First we have to find the *pre-pre-followers* of the given letter, then the *pre-followers* and finally the *followers*. The pre-pre-followers indicate the possible ways of continuing our sap (viewed globally) from $x < k$ into $x \leq k$, by adding vertical edges joining the "loose ends", thereby possibly tying some of them to each other, which results in their disappearance. The pre-followers are obtained by deciding what to do with the surviving open ends. Going from pre-pre-followers to pre-followers does not change the first component of the pre-pre-letter. Finally the followers are obtained from the pre-followers by inserting new adjacent L, R's in the underlying legal bracketing, using the free space that is left. In every phase, we also keep track of the number of edges (of the sap, not of the Markov Process!) that are used, that contribute to the weight of the Edge (of the Markov Process, not of the original sap!).

Let's first describe the first phase, of determining the pre-pre-followers. Let's consider a typical sap, and cut it at the vertical line $x = k$. As we saw above, the horizontal edges between $x = k-1$ and $x = k$ define a certain letter, the first part describing the induced legal bracketing, and the second part listing the y coordinates. So let's look only at the part of the sap that lies to the left of $x = k$. Suppose that the second part of k^{th} letter is $[i_1, \ldots, i_{2r}]$. This means that there are $2r$ "open ends" waiting to be extended beyond the "longitude" $x = k$. First note that we are allowed to add vertical edges on $x = k$, and in the process change the underlying legal bracketing. For a very simple example, suppose the letter is of the form $[[L,R],[i1,i2]]$. Then we can close the partial sap by adding the $i2-i1$ edges joining $[k,i1]$ and $[k,i2]$ on $x = k$, and finish up, adding a final letter "Period", to indicate that we are finished. Another example is $[[L,L,R,R],[i1,i2,i3,i4]]$. We can add the $i2-i1$ vertical edges joining $[k,i1]$ and $[k,i2]$, thereby making the $i3$ an "L", turning it effectively into $[[L,R],[i3,i4]]$. Another option is to add the $i4-i3$ edges joining $[k,i3]$ and $[k,i4]$, thereby making the $i2$ an "R", turning it effectively into $[[L,R],[i1,i2]]$. We could also do both, eliminating the bracketing altogether, which signals that we are done, so we only add the last letter, the Period.

In general, we can iterate this process of "shrinking the bracketing", but every time we create new vertical edges, we lose territory, so that we have to keep track of the "open space" left. In addition, since we want to determine the type III Combinatorial Markov Process (in which the edges are assigned weights), we also have to have another variable, *extw*, that describes the length of the portion of the underlying sap corresponding to the transition from the region $x < k$ to $x \leq k$, in the first phase, and from $x \leq k$ to $x < k+1$ in the second and third phases.

So we are now ready to formally define the set of *pre-pre-followers* of a letter $LET = [[w_1, \ldots, w_{2r}], [i_1, \ldots, i_{2r}]]$, where $[w_1, \ldots, w_{2r}]$ is a legal bracketing, and $0 \leq i_1 < \ldots < i_{2r} \leq K$. To this end, we need to define the notion of *pre-pre-letter*.

A *pre-pre-letter* is a triple $[LET, FS, extw]$, where LET is a letter, FS is a subset of $\{0, 1, 2, \ldots, K\}$, and $exwt$ is an integer denoting the extra-weight acquired so far in the transition from $x < k$ to $x < k + 1$. At the beginning $FS = FS_0 := \{0, 1, 2, \ldots, K\} \backslash \{i_1, \ldots, i_{2r}\}$, and $extw = 0$.

Initially the set of pre-pre-followers only has one element: the triple $[LET, FS_0, 0]$. We keep enlarging it by iteratively applying three "legal moves" to the present members of this set of pre-pre-followers of LET, until no new members can be inducted.

The three possible legal moves are RR, LL, and RL. We will now describe them in turn.

RR move

an "RR move" is obtained by joining two adjacent Rs, erasing them both and making the widow of the lower deceased R, formerly an L, into an R. More precisely, an RR move can be performed whenever there is a b between 1 and $2r - 1$ such that $w_b = w_{b+1} = R$. Letting the L-mate of w_{b+1} be $w_{a1} (= L)$ and that of w_b be $w_{a2} (= L)$, (of course we must have $a1 < a2$), we erase the two R's, w_b and w_{b+1}, from the first component of LET, changing w_{a2} from L to R. We also erase i_b and i_{b+1}. The new $extw$, of the derived pre-pre-letter, is the old $extw$ plus $i_{b+1} - i_b$ (the extra length of the sap, corresponding to performing this particular RR move), and the new set of free-space, FS, is the old FS minus $\{i_b + 1, i_b + 2, \ldots, i_{b+1} - 1\}$, corresponding to the set of lattice points $\{[k, i_b + 1], [k, i_b + 2], \ldots, [k, i_{b+1} - 1]\}$ taken-up by this particular RR move.

In short, given a pre-pre-letter $[LET, FS, extw]$, where $LET = [[w_1, \ldots, w_{2r}], [i_1, \ldots, i_{2r}]]$, then whenever there are two consecutive Rs in $w_1 \ldots w_{2r}$, say, $w_b = w_{b+1} = R$, applying the RR-move at b, consists of the following operation, where $a1$ is the location of the L-mate of $w_{b+1} (= R)$ and $a2$ is the location of the L-mate of $w_b (= R)$.

$$[[[w_1, \ldots, w_{a1-1}, L, w_{a1+1}, \ldots, w_{a2-1}, L, w_{a2+1}, \ldots, w_{b-1}, R, R, w_{b+2}, \ldots, w_{2r}], [i_1, \ldots, i_{2r}]], FS, extw] \rightarrow$$

$$[[[w_1, w_{a1-1}, L, w_{a1+1}, \ldots, w_{a2-1}, R, w_{a2+1}, \ldots, w_{b-1}, w_{b+2}, \ldots, w_{2r}], [i_1, \ldots, i_{b-1}, i_{b+2}, \ldots i_{2r}]],$$
$$FS \backslash \{i_b + 1, \ldots, i_{b+1} - 1\}, extw + i_{b+1} - i_b]$$

LL move

An "LL move" is obtained by joining two adjacent Ls, erasing them both and making the widower of the upper deceased L, formerly an R, into an L. More precisely, an LL move can be performed whenever there is an a between 1 and $2r - 1$ such that $w_a = w_{a+1} = L$. Letting the R-mate of w_{a+1} be $w_{b1} (= R)$ and that of w_a be $w_{b2} (= R)$, (of course we must have $b1 < b2$), we erase the 2 L's w_a and w_{a+1}, from the first component of LET, and change w_{b1} from R to L. We also erase i_a and i_{a+1}. The new $extw$, of the derived pre-pre-letter, is the old $extw$ plus $i_{a+1} - i_a$ (the extra length of the sap, corresponding to performing this particular RR move), and the new set of free-space, FS, is the old FS minus $\{i_a+1, i_a+2, \ldots, i_{a+1}-1\}$, corresponding to the set of lattice points $\{[k, i_a+1], [k, i_a+2], \ldots, [k, i_{a+1}-1]\}$ taken-up by this particular RR move.

In short, given a pre-pre-letter $[LET, FS, extw]$, where $LET = [[w_1, \ldots, w_{2r}], [i_1, \ldots, i_{2r}]]$, then whenever there are two consecutive L's in $w_1 \ldots w_{2r}$, $w_a = w_{a+1} = L$, applying the LL-move at a, consists of the following operation, where $b1$ is the location of the R-mate of $w_{a+1} (= L)$ and $b2$ is the location of the R-mate of $w_a (= L)$.

$$[[[w_1, \ldots, w_{a-1}, L, L, w_{a+2}, \ldots, w_{b1-1}, R, w_{b1+1}, \ldots, w_{b2-1}, R, w_{b2+1}, \ldots w_{2r}], [i_1, \ldots, i_{2r}]], FS, extw] \rightarrow$$

$$[[[w_1, \ldots, w_{a-1}, w_{a+2}, \ldots, w_{b1-1}, L, w_{b1+1}, \ldots, w_{b2-1}, R, w_{b2+1}, \ldots w_{2r}], [i_1, \ldots, i_{a-1}, i_{a+2}, \ldots i_{2r}]],$$
$$FS \backslash \{i_a+1, \ldots, i_{a+1}-1\}, extw + i_{a+1} - i_a]$$

RL move

an "RL move" may be obtained whenever an L immediately follows an R, and erasing them both, and leaving the widowers unchanged, thereby making them marry each other (luckily they are of the same sex now, so that the "sex-change operation" required for one of the survivors of the RR and LL moves, so that they can marry each other, is no longer necessary). More precisely, an RL move can be performed whenever there is an a between 1 and $2r-1$ such that $w_a = R$ and $w_{a+1} = L$. The new $extw$, of the derived pre-pre-letter, is the old $extw$ plus $i_{a+1} - i_a$ (the extra length of the sap, corresponding to performing this particular RR move), and the new set of free-space, FS, is the old FS minus $\{i_a+1, i_a+2, \ldots, i_{a+1}-1\}$, corresponding to the set of lattice points $\{[k, i_a+1], [k, i_a+2], \ldots, [k, i_{a+1}-1]\}$ taken-up by this particular RR move.

In short, given a pre-pre-letter $[LET, FS, extw]$, where $LET = [[w_1, \ldots, w_{2r}], [i_1, \ldots, i_{2r}]]$, then whenever there is an L immediately following an R in $w_1 \ldots w_{2r}$, say: $w_a = R, w_{a+1} = L$; applying the RL-move at a, consists of the following operation,

$$[[[w_1, \ldots, w_{a-1}, R, L, w_{a+2}, \ldots, w_{2r}], [i_1, \ldots, i_{2r}]], FS, extw] \rightarrow$$

$$[[[w_1, \ldots, w_{a-1}, w_{a+2}, \ldots, w_{2r}], [i_1, \ldots, i_{a-1}, i_{a+2}, \ldots i_{2r}]],$$
$$FS \backslash \{i_a+1, \ldots, i_{a+1}-1\}, extw + i_{a+1} - i_a]$$

From Pre-Pre-Followers to Pre-Followers

The set of pre-pre-followers of any given letter LET might contain a pre-letter that has the form $[[[], []], FS, extw]$, i.e. where the underlying letter shrunk to nothing, which means that the sap has been closed, and can't be continued. This means that there is an Edge between LET to the final letter "FINISH". So the set of followers of LET, Followers(LET), starts out with $[FINISH, extw]$. If the pre-pre-follower is not empty, then, if it is $[LET1, FS, extw]$, where $LET1 = [[w_1, \ldots, w_{2s}], [i_1, \ldots, i_{2s}]]$, then we have $2s$ open ends, and we have to decide how to continue them along the vertical line $x = k$, before they venture into a horizontal edge connecting the vertical lines $x = k$ and $x = k + 1$. Of course, for the vertical edges, we can only use vertices on $x = k$ whose y coordinates belong to FS. Also, for each of these decisions where to extend the $2s$ free ends, we keep track of the extra weight, and modify the free-space set, FS. For example, in the language for saps confined to $[0, 11]$ (i.e. $K = 11$), consider the letter $[[L, R, L, R], [1, 5, 7, 10]]$. Its only pre-pre-follower is obtained by performing an RL-move at the second and third places, resulting in the pre-letter $[[[L, R], [1, 10]], \{0, 2, 3, 4, 7, 8, 9, 11, 12\}, 2]$. Now we have two free ends to worry about. The one at $[k, 1]$ corresponding to the L has the option either to go down one unit, to $[k, 0]$ before going to $[k+1, 0]$, or to "cross the river" right way, straight to $[k+1, 1]$, or to go up one unit, to $[k, 2]$ (and then to $[k + 1, 2]$), or go up two units, to $[k, 3]$ (and then to $[k + 1, 3]$), or go up three units, to $[k, 4]$ (and then to $[k+1, 4]$). Of course it may not go up four units, since $[k, 5]$ is not free. Similarly the free end that is now at $[k, 10]$ may go one or two units up, to $[k, 11]$, or $[k, 12]$, respectively, before crossing the river to $[k+1, 11]$, $[k+1, 12]$ respectively, or it may decide to go to $[k+1, 10]$ straight away, or it may decide to go down to $[k, 9]$, or $[k, 8]$, or $[k, 7]$, before crossing horizontally to $[k + 1, 9]$,$[k + 1, 8]$, $[k + 1, 7]$, respectively. So the open end at $[k, 1]$ has 4 options, while the open end at $[k, 10]$ has 6 options. These are independent (in this case, sometimes the options of each of the individual free ends conflict with each other), so in this case there are $4 \times 6 = 24$ pre-followers that follow the pre-pre-follower $[[[L, R], [1, 10]], \{0, 2, 3, 4, 7, 8, 9, 11, 12\}, 3]$ considered here. For example one of them, obtained when we decide to make the L go down one unit and the R go up two units, results in the following pre-follower $[[[L, R], [0, 11]], \{2, 3, 4, 7, 8, 9, 12\}, 7]$.

From Pre-Followers to Followers

Each pre-follower is automatically a follower, but in addition, there are many other kinds of followers. These are obtained by inserting pairs LR in the free-space set FS, just like we did for the Left-Letters above, individually for each consecutive stretch of free-space. For example, the pre-letter above $[[[L, R], [0, 11]], \{2, 3, 4, 8, 9, 12\}, 7]$, may be followed, by itself, of course (i.e. do nothing extra), or by $[[[L, L, R, R], [0, 2, 3, 11]], \{4, 8, 9, 12\}, 10]$, or $[[[L, L, R, R], [0, 8, 9, 11]], \{2, 3, 4, 12\}, 10]$. Every time we insert a pair L, R in a stretch of free space, say that we place L at location a and R at location b, the resulting new pre-letter has its $extw$ increased by $b - a + 2$.

Constructing the type III Combinatorial Markov-Process for SAPS

Recall that in order not to confuse edges of the original sap, that live on the 2D square-lattice, with edges of the combinatorial Markov Process describing them, we are calling the latter kind Edges.

To create the Type III (i.e. Edge-weighted) Combinatorial Markov Process describing saps confined to $0 \le y \le K$, we create two extra letters, let's call them $START$, and $FINISH$. The followers of the letter $START$ are all the left-letters described above, i.e. all letters of the form $[[L, R, L, R, \ldots, L, R], [i_1, \ldots, i_{2r}]]$, for $r = 1, 2, \ldots, [(K + 1)/2]$, and $0 \le i_1 < i_2 < \ldots < i_{2r} \le K$. So we have one Edge between $START$ and each one of these left-letters, and we assign the weight

$$\sum_{a=1}^{r} (i_{2a} - i_{2a-1} + 2) \quad ,$$

to that Edge, because these are the number of edges that live in $0 \le x < 1$.

For each letter, we find its followers, as described above, getting followers that are pre-letters of the form $[LET1, FS, extw]$. FS is no longer needed, but $extw$ is the weight of the Edge connecting LET and $LET1$. Also, if one of the pre-pre-followers is of the form $[[], []], FS, extw]$, then we make $FINISH$ one of the followers of LET, and the weight of the Edge connecting LET and $FINISH$ is $extw$, and FS is discarded.

So in the resulting type III Combinatorial Markov Process, $[Start, Finish, ListOfNeighbors]$, the set $Start$ consists of the singleton $\{START\}$, the set $Finish$ consists of the singleton $\{FINISH\}$, and the list of outgoing neighbors is constructed as above.

A User's Manual for the Maple Package SAP

You must first download the package `SAP`, saving it as SAP, either from my website, or directly from INTEGERS. To use it, stay in the same directory, get into Maple, and type: `read SAP;` Then follow the on-line help. In particular, to get a list of the main procedures, type: `ezra();` .

The main procedures are: `GF, GFSeries , SAPseries , gfBatchelorYung` .

For a positive integer n, and a variable s, typing `GF(n,s);` would give the generating function

$$f_n(s) := \sum_{i=1}^{\infty} a_n(i) s^{2i} \quad ,$$

where $a_n(i)$ is the number of self-avoiding polygons in the two-dimensional square lattice whose perimeter has $2i$ edges, and whose width (the difference between the largest and smallest y coordinates) is $\le n$.

`GF(n,s)` works, on my computer, up to $n = 6$. Beyond that, you may wish to use `GFSeries(n,L)`, where L is also a positive integer, in order to get the first L coefficients of $f_n(s)$.

In order to get the first L terms in the sequence enumerating *all* self-avoiding polygons (with unrestricted width), type `SAPseries(L);` .

Finally `gfBatchelorYung(n,t);` gives the generating function for enumerating self-avoiding *walks* immersed in the strip $[0, n]$ and such that the leftmost wall touches the x-axis (i.e. contains the origin). It gives the quantity described in M.T. Batchelor, and C.M. Yung's paper [BY] (the denominators on p. 4059 and 4066 and the numerators on p. 4064).

A Brief Explanation of the Nuts and Bolts of `SAP`

`gf(n,t)` is the generating function for SAPS immersed in $[0,n]$, and it is obtained by calling procedure `SolveMC3`, borrowed from the package `MARKOV`, that solves the type III Markov Process obtained via `MarCha(n)`. `MarCha(n)` follows the outline given above for constructing the Markov Process describing saps confined to $[0,n]$. Since `gf(n,t)` counts all saps immersed in $[0,n]$, not only those that lie on the x-axis, and what we really want are only those that do lie (in other words we only want every sap to be counted once up to translation), we have to find $gf(n,t) - gf(n-1,t)$. This is done by `GF(n,t)`, the main procedure.

`SAPseries` is designed in an analogous way to `KHAYA`.

`gfBatchelorYung(n,t)` uses a slight modification of the Markov Process, obtained from `MarChaBY(n)`, so that to count the set of self-avoiding walks counted by [BY].

More SAPs For Less Money: Counting Locally-Skinny SAPs

The Maple package SAP counts *globally* skinny saps, i.e., for any given positive integer n, it counts the saps for which the difference between the largest y coordinate of any of the vertices, and the smallest y coordinate, is less than n. The package `FreeSAP` counts the larger set of saps in which we only demand that the difference between the largest and smallest y coordinate, *for each specific* vertical slice $x = k$, is smaller than $\leq n$.

To go from SAP to FreeSAP, we use the same methodology that was employed in going from ANIMALS to FreeANIMALS. For the details, see the source code of the Maple package FreeSAP.

A User's Manual for the Maple Package FreeSAP

You must first download the package `FreeSAP`, saving it as FreeSAP, either from my website, or directly from INTEGERS. To use it, stay in the same directory, get into Maple, and type: `read FreeSAP;` Then follow the on-line help. In particular, to get a list of the main procedures, type: `ezra();` .

The main procedures are `gff` and `gffSeries`.

For a positive integer n, and a variable s, typing `gff(n,s);` would give the generating function

$$g_n(s) := \sum_{i=1}^{\infty} b_n(i) s^{2i} \quad ,$$

where $b_n(i)$ is the number of self-avoiding polygons in the two-dimensional square lattice whose perimeter has $2i$ edges, and such that for any vertical cross-section, $x = k$, the difference between the largest y such that (k, y) belong to the sap, and the smallest such y, is $\leq n$.

`gff(n,s)` works, on my computer, up to $n = 6$. Beyond that, you may wish to use `gffSeries(n,L)`, where L is also a positive integer, in order to get the first L coefficients of $g_n(s)$.

A Brief Explanation of the Nuts and Bolts of `FreeSAP`

The underlying Combinatorial Markov Process is of type IV. `MarChaf(n,t);` constructs it, using the machinery of `SAP`, but introducing multiple edges as necessary (corresponding to various interfaces). Then `SolveMC4`, borrowed from the package `MARKOV`, described above, finds the weight-enumerator, in procedure `gff(n,t)` . `SolveMC4series`, also borrowed from `MARKOV`, is used to find the series expansion in `gffSeries(n,L)`.

Counting Skinny Self-Avoiding Walks

The methodology is the same as for counting self-avoiding polygons, but the details are more complicated. The reader is invited to study the source-code of the Maple package `SAW` in detail. Here we will only briefly sketch how to adapt the algorithm for counting skinny self-avoiding polygons to that of counting skinny self-avoiding walks.

The vertical cross-sections look a little different, since in addition to $L-R$ pairs, we may have zero, one, or two "loners" that do not connect, from the left, to any other horizontal edge. Let's denote these loners by the letter A. So now the alphabet still contains the same letters as in the SAP alphabet, but each of these letters give rise to several more letters, obtained by inserting either one or two A's in any available slot.

For example, if the strip is $0 \leq y \leq 5$, then the letter $[[L,R],[0,3]]$ also gives rise to the letters $[[L,A,R],[0,1,3]]$, $[L,A,R,A],[0,1,3,5]]$ and eight others (altogether $\binom{4}{1}+\binom{4}{2}=10$).

The pre-left-letters are obtained from those of the SAP case by sticking 0, 1, or 2 A's. To get the pre-pre-followers of a letter, we have, in addition to RR, LL, and RL moves, also the following four moves.

AL move: in which the lonely A connects to an L residing right above it, making L's ex-R-mate into a lonely A.

LA move: in which the lonely A connects to an L residing right under it, making L's ex-R-mate into a lonely A.

AR move: in which the lonely A connects to an R residing right above it, making R's ex-L-mate into a lonely A.

RA move: in which the lonely A connects to an R residing right under it, making R's ex-L-mate into a lonely A.

The phase of going from pre-pre-followers to pre-followers is similar, it preserves the $L-R-A$ part, i.e. the first component of the letter, but changes the second part, as well as the available free space.

Finally, the phase between pre-followers to followers is also similar, except that in addition we may stick-in one or two A's up to a total of two A's.

We also have two special letters $START$ and $FINISH$, to denote the beginning and end. Every time it is possible to end the walk, i.e. there are only (one or two) As left (possibly after doing AbortL or AbortR, which means that an L or R, formerly paired, decides to "commit suicide", thereby leaving its mate a lonely A.

To fully understand what's going on, you *must* study `SAW` carefully.

A User's Manual for the Maple Package SAW

The main functions are `GFW, GFSeriesW`, and `SAWseries`.

`GFW(n,t);` would give the (ordinary) generating function, in the variable t, that enumerates (undirected) saws whose (global) width is $\leq n$. In other words, the rational function

$$W_n(t) = \sum_{i=0}^{\infty} a_n(i) t^i \quad ,$$

where $a_n(i)$ is the number of i-step self-avoiding walks (divided by 2), whose width is $\leq n$.

`GFW(n,t)` works, on my computer, up to $n = 4$. Beyond that, you may wish to use `GFSeriesW(n,M)`, where M is also a positive integer, in order to get the first M coefficients of $W_n(t)$.

Finally, if you want to find the first L terms in the notorious enumerating sequence for (all, unrestricted) saws (in the 2D square-lattice), type `SAWseries(L)`. Of course, it can't compete with the very efficient computations of Conway and Guttmann [CG], but its agreement, up to $L = 24$, with the published data, is the best *proof* for us, of the validity of our approach. If Mikowsky[M] would have done it, he would have found his mistake right away (his output starts to disagree at $L = 4$).

A Very Brief Explanation of the Nuts and Bolts of `SAW`

`MarChaW` finds the type IV Combinatorial Markov Process that describes the language of saws of bounded width. Please note that we use a slightly different representation than the one used in `MARKOV`, using lists rather sets and generating polynomials. Likewise, `SolveMarCha` is a version of `MARKOV`'s `SolveMC4` that handles the present representation. `gf(n,t);` gives the generating function for all saws confined to the strip $0 \leq y \leq n$, which over-counts saws of width $< n$. In order to get the right count, we use `GF(n,t);`, which is simply $gf(n,t) - gf(n-1,t)$.

More SAWs For Less Money: Counting Locally-Skinny SAWs

The Maple package SAW counted *globally* skinny saws, i.e., for the given positive integer n, it counted saps for which the difference between the largest y coordinate of any of the vertices, and the smallest y coordinate, is $\leq n$. The package `FreeSAW` counts the larger set of saws in which we only demand that the difference between the largest and smallest y coordinate, *for each specific* vertical slice $x = k$, is $\leq n$.

To go from SAW to FreeSAW, we use the same methodology that was employed in going from ANIMALS to FreeANIMALS, and from SAP to FreeSAP. We refer the reader is to the source code of the Maple package FreeSAW.

A User's Manual for the Maple Package FreeSAW

You must first download the package `FreeSAW`, saving it as FreeSAW, either from my website, or directly from INTEGERS. To use it, stay in the same directory, get into Maple, and type: `read FreeSAW;` . Then follow the on-line help. In particular, to get a list of the main procedures, type: `ezra();` . The main procedures are `gfWf` and `gfSeriesWf`.

For a positive integer n, and a variable s, typing `gfWf(n,s);` would give the generating function

$$f_n(s) := \sum_{i=1}^{\infty} b_n(i) s^i \quad ,$$

where $b_n(i)$ is the number of i-step self-avoiding walks in the two-dimensional square lattice such that for any vertical cross-section, $x = k$, the difference between the largest y such that (k, y) belong to the saw, and the smallest such y, is $\leq n$.

`gfWf(n,s)` works, on my computer, up to $n = 5$. Beyond that, you may want to use `gfSeriesWf(n,L)`, where L is also a positive integer, in order to get the first L coefficients of $f_n(s)$.

A Very Brief Explanation of the Nuts and Bolts of `FreeSAW`

The underlying Combinatorial Markov Process is of type IV. `MarChaWf(n,t);` constructs it, using the machinery of `SAW`, but introducing even more multiple edges as necessary (corresponding to various interfaces). Then `SolveMarCha` solves it to give `gfWf(n,t)`.

Conclusion

We have described seven Maple packages: `MARKOV, ANIMALS, FreeANIMALS, SAP, FreeSAP, SAW` and `FreeSAW`. The first of these, `MARKOV` is of very general scope, and should be useful in many other situations where the Transfer-Matrix method is employable. The other packages enumerate "skinny" lattice-animals, self-avoiding polygons and self-avoiding walks, with two definitions of skinniness: *global*, where the object has to fit completely within a prescribed horizontal strip, and *local*, where the whole object has unbounded width, but each vertical cross-section has bounded width.

We hope that in the future, similar treatment would be given for other venerable models of statistical physics, like the Ising model (in two and three dimensions, with magnetic field), Percolation, and the Monomer-Dimer problem. This methodology should also be useful in the study of plane and solid partitions.

Most important, the systematic, careful and lucid "software engineering" of this project should make it easier for me to implement the *Umbral Transfer-Matrix Method*[Z2].

References

[BY] M.T. Batchelor, and C.M. Yung, *Exact transfer-matrix enumeration and critical behaviour of self-avoiding walks across finite strips*, J. Physics A: Math. Gen. **27** (1994) 4055-4067.

[B] Mireille Bousquet-Mélou, *Convex polyominoes and Algebraic Languages*, J. Physics A: Math. Gen. **25** (1992), 1935-1944.

[CG] A.R. Conway and Anthony J. Guttmann, *Square lattice self-avoiding walks and corrections-to-scaling*, Phys. Rev. Letts. **77** (1996), 5284-5287.

[DV] Maylis Delest and Xavier G. Viennot, "*Algebraic Languages and Polyominoes Enumeration*", Theor. Comp. Sci. **34**, 169-206.

[F] Steven Finch, *Mathematical Constants Website*, `http://www.mathsoft.com/asolve/constant/constant.html`.

[KP] Brian W. Kernighan and Bob Pike, "*The Practice of Programming*", Addison-Wesley, Reading, Mass, 1999.

[K1] David A. Klarner, *Cell Growth Problems*, Canad. J. Math. **19** (1967), 851-863.

[K2] David A. Klarner, *My Life Among the Polyominoes*, in "The Mathematical Gardener", D. A. Klarner, ed., Wadsworth, 1981, 243-262.

[KR] David A. Klarner and Ronald L. Rivest, *A procedure for improving the upper bound for the number of n-ominoes*, Canad. J. Math. **25** (1973), 585-602.

[M] Anthony A. Mikovsky, "*Convex Polyominoes, General Polyominoes, and Self-Avoiding Walks using algebraic languages*", Ph.D. dissertation, University of Pennsylvania Mathematics Department, 1997. Available from UMI Dissertation Services, `http://www.umi.com`.

[R] Ronald C. Read, *Contributions to the Cell Growth Problem*, Canad. J. Math. **14** (1962), 1-20.

[S] Richard P. Stanley, "*Enumerative Combinatorics*", volume 1, Wadsworth, Monterey, California, 1986. Reprinted by Cambridge University Press.

[WS] S.G. Whittington and C.E. Soteros, *Lattice Animals: Rigorous results and wild guesses*, in: "Disorder in Physical Systems: A Volume in Honour of J.M. Hammersley, ed. G.R. Grimmet and D.J.A. Welsh, Oxford, 1990.

[Z1] Doron Zeilberger, *Opinion 37: Guess What? Programming is Even More Fun Than Proving, and, More Importantly It Gives As Much, If Not More, Insight and Understanding*, `http://www.math.temple.edu/~zeilberg/Opinion37.html`.

[Z2] Doron Zeilberger, *The Umbral Transfer-Matrix Method*, in preparation.