Asymptotic estimates of the number of combinatorial structures using dynamic programming

Michal Forišek[1]
forisek@dcs.fmph.uniba.sk

[1] Comenius University, Bratislava, Slovakia

Keywords: dynamic programming, forbidden submatrices, tiling with squares, asymptotic estimates
Abstract: In the paper we present a universal technique that gives asymptotically exact estimates of the number of certain combinatorial structures. The technique can be applied in cases where the exact number of said structures can be computed using a specific linear form of dynamic programming. We demonstrate the technique on counting the number of matrices with a forbidden submatrix, and on counting the number of tilings with squares.

1 Overview

In the paper we apply results from linear algebra to efficient combinatorial algorithms. In this part of the paper we present an overview of known results we use.

1.1 Vector length under a repeated linear transformation

Given a vector u and a matrix A, we will consider a sequence of vectors defined as follows: $u_i = uA^i$. In particular, we will be interested in $|u_i|$ as a function of i. Theorem 1 addresses this.

Note that all vectors and matrices in this paper will have nonnegative integer elements. Still, we state Theorem 1 in a more general way.

Theorem 1 *Let u be a k-dimensional row vector, and let A be a $k \times k$ matrix (both over the complex numbers field). Additionally, assume that A has k linearly independent eigenvectors. Let f be the function defined as $f(n) = |uA^n|$. Then f belongs*

to $\Theta(\alpha^n)$, where α is the maximum modulus (i.e., absolute value) of an eigenvalue of A that corresponds to an eigenvector v that is not orthogonal to u.

The proof is obvious: it is sufficient to consider A as a linear transformation with the set of linearly independent eigenvectors as its base. The length of the projection of u onto v is multiplied by α in each iteration.

Corollary 1 *The absolute value of the largest element of the vector uA^n must be $\Omega(\alpha^n/\sqrt{k})$. If k is a fixed constant, the largest element of uA^n is asymptotically equal to α^n, and thus the sum of absolute values of all elements of uA^n is also asymptotically equal to α^n.*

1.2 Dynamic programming

Dynamic programming is a technique used to design efficient algorithms. This technique can be applied to problems with a so-called optimal substructure: the solution of any nontrivial instance can be formulated in terms of solutions of other, smaller instances (usually subinstances of the original instance). Whenever we observe such a structure, we can turn it into an algorithm that recursively solves instances, making sure each of the necessary instances is solved only once. Alternately, we can fix any valid topological order for the instances we need to compute, and compute their solutions iteratively. From a mathematical point of view, dynamic programming represents an efficient way of computing a subset of values of a given recurrence.

The computation of the n-th Fibonacci number is a trivial example of dynamic programming. The definition gives us a recursive formula: $\forall n \geq 2 : F_n = F_{n-1} + F_{n-2}$. However, a program that simply rewrites this formula into a recursive function will run in exponential time. In order to obtain a polynomial time complexity, we have to either apply memoization[1] or we have to compute the values iteratively in the order $F_2, F_3, \ldots, F_n$, always remembering at least the last two computed values.

1.3 Dynamic programming as a linear transformation

In some cases of dynamic programming, the computation of a new value of the recurrence can be written as a linear combination of some previous values (and possibly of some easy-to-compute values such as polynomials). Whenever this happens and the instances happen to have a suitable structure, we can express the computation of successive values of the recurrence as an iteration of a suitably chosen linear transformation. For example, we can easily verify that Fibonacci numbers satisfy the following:

$$(F_n, F_{n+1}) \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_{n+1}, F_{n+2})$$

[1] I.e., store the return value of each function call, and only make a function call if we don't have its return value stored yet.

As matrix multiplication is associative, we can then write $F_n = (0,1)\begin{pmatrix}0 & 1\\ 1 & 1\end{pmatrix}^n\begin{pmatrix}1\\ 0\end{pmatrix}$.

This notation has many practical applications. First of all, we can now use it to obtain a more efficient algorithm to compute the value F_n. (In order to do so, we need to use exponentiation by squaring, and a fast algorithm to multiply big integers.) Additionally, our Corollary 1 tells us that Fibonacci numbers grow asymptotically as fast as φ^n, where $\varphi = (1+\sqrt{5})/2$ (the golden ratio) is the bigger of the two eigenvalues of the above matrix.

Our goal in this paper is to show how to generalize this observation to more complicated forms of dynamic programming, and to use it to easily obtain asymptotic estimates of the number of objects counted by said dynamic programming.

2 Forbidden submatrices of a constant size

For simplicity, all matrices in this part of the paper are 0-1 matrices, i.e., each element is either 0 or 1. This is only for the sake of a simpler presentation, all results can easily be generalized.

The problem we are now going to solve can be viewed as a generalization of the well-known problem of counting strings with a forbidden substring. (See section 4.7 in [1].) The main question we will be asking is the following one: Given r, c, and a matrix Z, what is the number of matrices of dimensions $r \times c$ that do not contain Z as a (contiguous) submatrix?

We are going to focus on a subproblem of this problem: we will consider r a fixed constant and we will analyze the number of matrices as a function of c.

2.1 Counting matrices using dynamic programming

As an example, consider the forbidden matrix $Z = \begin{pmatrix}0&1&0\\1&0&1\\0&1&0\end{pmatrix}$. (Again, all our constructions can be generalized to arbitrary dimensions and content of Z.) Let $P[r,c]$ be the number of $r \times c$ matrices that do not contain Z.

How can we use dynamic programming to compute the values $P[r,c]$? The trick is to use a larger set of states. Instead of computing $P[r,c]$ directly, we shall compute the values $P[r,c,s_{c-1},s_c]$, where s_{c-1} and s_c are the full contents of columns $c-1$ and c of the $r \times c$ matrix. For example, $P[3,10,001,011]$ is the count of matrices of dimensions 3×10 that do not contain Z, and have $\begin{pmatrix}0\\0\\1\end{pmatrix}$ and $\begin{pmatrix}0\\1\\1\end{pmatrix}$ in the last two columns. (In general, the state $P[r,c,\dots]$ will contain the last $x-1$ columns, where x is the number of columns in Z.)

For these new values easily write a linear recurrence. In our example with a 3×3

matrix Z these are:

$$
\begin{aligned}
P[r,\, 2,\, y_1y_2\dots y_r,\, z_1z_2\dots z_r] &= 1 \\
P[r,\, c+1,\, y_1y_2\dots y_r,\, z_1z_2\dots z_r] &= \sum P[r,\, c,\, x_1x_2\dots x_r,\, y_1y_2\dots y_r]
\end{aligned}
$$

where the sum in the general case ranges over all contents of column $c-1$ such that the three columns $c-1$, c, and $c+1$ taken together do not contain the forbidden submatrix Z.

For example, $P[4, 5, 1101, 1010]$ counts good matrices of the form $\begin{pmatrix} ?&?&?&1&1 \\ ?&?&?&1&0 \\ ?&?&?&0&1 \\ ?&?&?&1&0 \end{pmatrix}$. Our recurrence computes this value by trying all possibilities for the contents of the third column. Once we fix the contents of the third column (and verify that Z does not appear in the last three columns), we can forget the fifth column – clearly, the number of valid ways to fill in the first two columns doesn't depend on the contents of the fifth column. This reduces the computation of $P[4, 5, 1101, 1010]$ to the computation of some values $P[4, 4, ????, 1101]$. More precisely, in our example $P[4, 5, 1101, 1010]$ is the sum of all such values except for $P[4, 4, 0010, 1101]$ and $P[4, 4, 1010, 1101]$. (For these two we have an occurrence of the forbidden submatrix Z in columns 3-5.)

The above algorithm needs roughly $rc2^{3r}$ arithmetic operations (with big integers) to compute a particular value $P[r, c]$. In practice, this algorithm is therefore viable only if the number of rows happens to be small enough. (A more clever application of dynamic programming can reduce the exponential part of the time complexity to just 2^{2r}, but this is beyond the scope of this paper. Additionally, this optimization is not compatible with the construction shown in the next section.)

2.2 Counting matrices using an iterated linear transformation

If we know all the values $P[r, c, ???, ???]$, we can use them to compute all the values $P[r, c+1, ???, ???]$: for each new value, we compute the sum of some of the old values. This computation can be succinctly represented as a linear transformation. More precisely, for any fixed value r we can find a matrix A_r with the following property: for each c, if we take the vector of all values $P[r, c, ???, ???]$ and multiply it by A_r, we will obtain the vector of all values $P[r, c+1, ???, ???]$. (Note that the matrix A_r is the same for all values of c.)

The dimensions of A_r are $2^{2r} \times 2^{2r}$, but the matrix is sparse: there are at most 2^r ones in each row and column.

Hence, for any $c \geq 2$ we can compute the vector of all $P[r, c, ???, ???]$ as $(1, 1, \dots, 1)A_r^{c-2}$. (Here, the vector of all 1s is the vector of all values $P[r, 2, \dots]$.) Then, the value $P[r, c]$ is simply the sum of all elements of this vector. As with the Fibonacci number example, this now gives us a new algorithm to compute $P[r, c]$: now the number of arithmetic operations is exponential in r, but only logarithmic (as opposed to linear) in c. For example, we can easily evaluate $P[7, 10^{100}]$.

2.3 Asymptotic estimate of the number of matrices

For any forbidden submatrix Z and any fixed r (both need to be small enough in practice) we can now apply the technique shown in the Overview section: we will compute the matrix A_r, verify that it has the required properties by finding its eigenvectors and eigenvalues, and then we find α as the maximum modulus over all eigenvalues that correspond to eigenvectors that are not orthogonal to the vector $(1, \ldots, 1)$. We can then conclude that $P[r, c]$ as a function of c grows asymptotically exactly as quickly as α^c.

(In practice, for problems of this type α is usually simply the largest eigenvalue, which is a positive real.)

2.4 The number of matrices with a forbidden 3×3 pattern

In this section we analyze the number of $r \times c$ matrices that do not contain our example forbidden submatrix $Z = \begin{pmatrix} 0&1&0 \\ 1&0&1 \\ 0&1&0 \end{pmatrix}$. In the table we give two different coefficients for each r. The coefficient α_r is the base of the exponential function: the number of good $r \times c$ matrices, as a function of c, grows as fast as α_r^c. The coefficient β_r is computed as $(\log_2 \alpha_r)/r$. Hence, the number of good $r \times c$ matrices is proportional to $2^{\beta_r rc}$.

Table 1: Count of matrices with a specific forbidden 3×3 submatrix.

r	3	4	5	6	7	8	9
α_r	7.98456	15.9398	31.8217	63.5276	126.824	253.186	505.451
β_r	0.999071	0.998641	0.998388	0.998218	0.998098	0.998007	0.997937

Values up to $r = 6$ were computed by a straightforward application of the algorithm described above. For larger values of r we added an extra step: the reduction of symmetries. Before computing A_r we merged equivalent states together. This speeds up the computation significantly. For example, for $r = 9$ we obtained a matrix with side 1146 instead of a matrix with side $2^{18} = 262\,144$.

To conclude this example, we note that the values given in the table have been rounded for presentation, but in principle these are exact values. We can compute them to any precision, or give an integer polynomial that has the value as a root.

2.5 The number of noise matrices

In this section we derive exact asymptotic estimates of the growth of the number of noise matrices, i.e., matrices that do not contain any of the following patterns: $\begin{pmatrix} & 0 & \\ 0 & 0 & 0 \\ & 0 & \end{pmatrix}$ and $\begin{pmatrix} & 1 & \\ 1 & 1 & 1 \\ & 1 & \end{pmatrix}$. In [2] the number of these matrices was experimentally estimated for $3 \leq r \leq 6$. We note that the experimentally obtained estimates of

constants β_r in [2] differ from correct values in the sixth decimal place. This is because the method used to estimate those values did not converge fast enough. In the table below we present exact results up to $r = 9$.

Table 2: The number of noise matrices.

r	3	4	5	6	7	8	9
α_r	7.53113	14.2986	27.1246	51.4594	97.6259	185.211	351.371
β_r	0.970955	0.959450	0.952305	0.947561	0.944170	0.941628	0.939650

3 Tiling using squares

As the second example of the strength of this technique we will show a solution to the following problem: for a fixed r, what is (as a function of c) the number of ways in which we can tile the rectangle $r \times c$ using arbitrary squares with integer side lengths? (The tiles must be non-overlapping and they must cover the entire rectangle.) A sample tiling for $r = 5$ and $c = 19$ is shown in Figure 1.

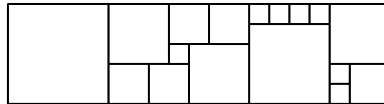

Figure 1: A sample tiling using squares.

Again, we can use dynamic programming to count such tilings exactly. The main idea is to construct the tiling incrementally. In each step we will find the rightmost column that is not tiled completely, and we will try all possibilities how this column can be tiled. Figure 2 shows one possible first step of tiling a rectangle with $r = 6$.

Figure 2: One possible way of tiling the last column.

For each possible way of tiling the last column we make a recursive call to compute the valid tilings of the rest of the rectangle. Now, obviously, these are *not* instances of the original problem. This is because some rows already have the last few unit squares covered. Hence, the general problem we are going to solve recursively looks as follows: “What is the number of valid tilings of an $r \times c$ rectangle, given that the last p_1 cells in the first row, p_2 cells in the second row, ..., and p_r cells in the last row are already covered?”

Figure 3 shows an example of a state where we already have $(1,1,1,0,0,2,2)$ cells tiled. When counting the tilings for this state, we need to try two possibilities: either the last column contains two 1×1 squares on the empty cells, or it contains a single 2×2 square there. Both possibilities lead into states where the number of columns left to tile is one smaller. In the first case the numbers of already tiled cells in rows are $(0,0,0,0,0,1,1)$, while in the second case these would be $(0,0,0,1,1,1,1)$.

Figure 3: An example of continuing an existing tiling.

Clearly, each p_i will always be less than r. This gives us an upper estimate: the number of instances we need to solve in order to compute the number of tilings of an $r \times c$ rectangle is $O(cr^r)$. (This estimate is way too loose, as only a small fraction of all states is actually reachable in valid tilings. It is possible to give a better estimate, but we don't need it. For our purpose, it is sufficient to note that if r is a fixed constant, we have an algorithm for which the number of arithmetic operations is linear in c.)

As in the previous section, we can now write the computation of our recurrence as an iterated application of a suitable linear transformation. We can algorithmically construct the corresponding matrix, do the necessary checks, and find the largest suitable eigenvalue to get an exact asymptotic estimate of the number of tilings. (The vector of initial values for this recurrence is a vector of all zeros, with a single 1 that corresponds to the state where all p_i are zero.)

The results of our computation are shown in Table 3. Row s gives the number of reachable states for a specific value of c – that is, the number of different $(p_1, \ldots, p_r)$ that can actually be obtained. The constant α_r is the size of the corresponding eigenvalue: the number of valid tilings of an $r \times c$ rectangle, as a function of c, grows asymptotically as fast as α_r^c. The values α_r are rounded.

Table 3: The number of tilings using squares.

r	1	2	3	4	5	6	7	8	9	10
s	1	2	5	11	24	53	118	261	577	1276
α_r	1.0000	1.6180	2.1479	2.9615	4.0551	5.5573	7.6159	10.4372	14.3035	19.6020

The count of these tilings is in OEIS as [3] (and several related sequences). In addition to the material available in OEIS we gave an efficient exact algorithm for small r and almost arbitrarily large c, and exact values of asymptotic growth rates

for small r.

4 Conclusion

The technique shown in the paper should be applicable for many similar combinatorial problems.

As a direction for future research we note that some similar combinatorial problems also exhibit simpler recurrences – however, these are hard to find and prove. It would be interesting to try applying this technique to be able to find and prove the correctness of such simplified recurrences automatically.

References

[1] R. Stanley, *Enumerative Combinatrics, vol. 1.* Cambridge University Press, 1997.

[2] M. Opial, "Bounded Locally Testable Matrices," 2015, Bachelor's Thesis, Comenius U.

[3] OEIS Foundation Inc., "The On-Line Encyclopedia of Integer Sequences, sequence A219924," 2016, http://oeis.org/A219924 (retrieved March 2016).