Dynamic Programming As a Tool For Solving Exact Cover Problems Efficiently With an Eye Towards Tiling Problems in Particular

Alexander Neuschotz

Summer 2021

Abstract

I begin by formalizing the notion of tiling problems with the intent to strengthen the connection to generalized exact cover problems and to provide insight into the way I structured my code, the Exact Cover Problem Solver, the third version of which I describe step-by-step before considering further adjustments I intend to make.

1 Overview

The purpose of this paper is to construct a formalization of the notion of tiling problems that makes it clear that they are special cases of exact cover problems and to present the method of my code, the Exact Cover Problem Solver, that solves the computational problem called #ExactCover.

2 Tiling Problems

Before we can consider tiling problems or the interpretation of them as exact cover problems, we need to understand what they are.

Definition 2.1. A *tiling problem* is a problem pertaining to how many ways a given region can be perfectly covered, or tiled, by non-overlapping copies of specified tile types.

Surely the reader will agree that this explanation is intuitive enough to put the image of the problem in anyone's head, but it is far from rigorous. For one thing, it relies on a presupposed intuition for what a region is and does not specify what is meant by "copies" of tile types. For these reasons, let us first reconstruct this definition with a more formal foundation.

Definition 2.2. Let B be an n-by-m box containing nm unit squares S, and designate any subset U of these unit squares as the *tileable units* with the requirement that either $|U| = 1$ or every element of U is adjacent to at least one other. The corresponding region *region* R is U contained in B. We shall call the complement of U in S, U^c, the set of *non-tileable units* and refer to U^c contained in B as the *non-region*.

Now, our definition is not quite where we want it to be. Consider a particular region R associated with an n-by-m box B and tileable units U. If we leave U unchanged, and merely expand the dimensions of B by adding one or more rows or columns of non-tileable units to the non-region, then the resulting region R' with associated box B' ought to be considered the same region. From an informal perspective, this is because padding a region with surrounding non-tileable units does not alter the tiling problem if the tile types are kept constant. More formally, the justification for this can be seen by the fact that there is a geometry-preserving bijection from U to U', which we can, by abuse of notation, refer to as a geometry-preserving bijection from R to R'. Therefore, we must consider regions not as individual objects, but as equivalence classes, where we consider two regions R and R' to be equal if there is a geometry-preserving bijection between R and R'. These bijections include one or more of the following geometry-preserving operations any number of times:

1. Adding or removing a row of non-tileable units to S and adjusting B accordingly.
2. Adding or removing a column of non-tileable units to S and adjusting B accordingly.
3. Shifting U over in B by one unit either up, down, left, or right.
4. Rotating U in B by 90 degrees clockwise or anti-clockwise.

Conjecture 2.3. These are the only operations on R that preserve the geometry of U.

For the sake of simplicity, we shall assume from this point on that all regions contain no rows or columns that are disjoint from U.

Definition 2.4. Given a region R with an associated n-by-m box B and tileable units U, a *subregion* R' is a subset $U' \subseteq U$ contained in an n'-by-m' box B', where $n' \leq n$ and $m' \leq m$.

That is, a subregion is a region that can be "contained" in a "larger" region. From this formal perspective, tile types are just subregions of the region being tiled. To reflect this fact, we shall denote tile types using r, which will also allow us to reserve T for denoting tiles, which shall be defined shortly. To avoid confusion from the fact that tile types are themselves regions, we shall heretofore only use the term region to refer to the specific region being tiled, unless otherwise specified, and we shall refer to tile types as such.

Definition 2.5. Given a region R with associated B and U, and tile types $r_1, r_2, \ldots, r_t$ with associated tileable unit sets $u_1, u_2, \ldots, u_t$, a *tile t* is the image of a geometry-preserving injection f from a u_i to U. We say that t *covers* a tileable unit $u \in U$ when $u \in t$.

Now that we have a formal definition of regions, tile types, and tiles, we are ready to formalize the meaning of a perfect tiling.

Definition 2.6. Given a region R with associated B and U, and a set of tiles T, a perfect tiling is a subset of tiles $T' \subseteq T$ such that every $u \in U$ is covered by exactly one tile.

In this context, we can now define tiling problems more carefully.

Definition 2.7. A *tiling problem* is a problem pertaining to how many ways a given region R with associated B and U can be perfectly tiled by a subset T' of tiles T derived from given tile types $r_1, r_2 \dots, r_t$.

Not only does this formulation of tiling problems make it clear that they can be interpreted as specialized cases of exact cover problems, but, as we shall see, it provides us with the perspective to do so. In order to see this, let us take a step back and look at generalized exact cover problems.

3 Tiling Problems As Exact Cover Problems

Before we can understand what an exact cover problem is, we first need to define set covers. Let us start with a set $X = \{1, \dots, n\}$ for some $n \in \mathbb{Z}^+$. This will be the *universe*, or *n-universe*, of our set cover problem. Consider a family of subsets $S \subseteq X$.

Definition 3.1. Given an n-universe X and a family S, a *set cover* is a subfamily $S' \subseteq S$ such that $\bigcup_{s \in S'} = X$.

There are several questions one might ask when given (X, S). Is there a non-trivial set cover? That is, is there a set cover S' that is a strict subset of S? How many set covers are there? For our purposes, we are interested in particular types of set covers called exact set covers.

Definition 3.2. Given (X, S), an *exact set cover*, or simply, *exact cover*, is a pair-wise disjoint set cover.

Exact set cover problems are all about finding the number of these exact covers given (X, S).

Definition 3.3. An *exact cover problem* is a problem pertaining to the number of exact covers of an n-universe X given a family of allowable subsets S.

It is not too difficult to see that we can interpret a tiling problem as an exact cover problem. Let R be a region associated with B and U. If we associate each tileable unit $u \in U$ with a unique number from 1 to $|U|$, then we can consider tiles as tuples that contain the numbers associated with the tileable units that they cover. Then, the set $X = \{1, \dots, |U|\}$ becomes our universe, the set T of all tiles becomes our family S, and there is a bijection between each perfect tilings of R by tiles in T and exact covers S' of X.

The point of formalizing the notion tiling problems the way that we did was to make manifest this connection to exact cover problems, which we can more easily approach with programming methods. Exact cover problems do not always provide favorable solutions. In fact, certain regions and tile types give rise to no perfect tilings. Such instances of the non-existence of tilings is neither surprising, nor difficult to confirm, since we can imagine a region made up of three tiles in an "L" position and imagine that the tile type is a domino, which clearly cannot perfectly tile the region because there are an odd number of tilable units and each tile must cover an even number of tileable units. The conditions for when no perfect tilings exist is a related interesting problem and is answered in part by Conway and Lagarias [2].

Among the tiling problems that do have perfect tilings, there are certain regions and tile types that provide "round" answers, which is to say that the solutions to a sequence of tiling problems, where we increase the dimension of the region in a particular and consistent way, form a sequence of numbers that are all products of small primes and increase with some pattern to be discovered. Robbins discusses round enumerations in his work with alternating sign matrices, wherein he observed that the sequence you get by letting A_n be the number of n-by-n alternating sign matrices forms a sequence of round numbers identical to the sequence you get by letting D_n be the number of descending plane partitions with no part exceeding n [6]. Patterns like these motivate the development of programs that can solve exact cover problems and provide data to locate and explain round answers. Propp discusses many of the concepts expressed in this paper in the context of round numbers [5].

4 Dynamic Programming

Robbins' method involves dynamic programming, which we explain and apply in our own program. Given (X,S), one might attempt to solve an exact cover problem with the following naïve algorithm.

Algorithm 4.1. *Start with the trivial node that is the empty set. For each node N, find the smallest element $x \in X$ not contained in N. For each subset $s_x \subseteq S$ that covers x and is disjoint from N, branch off and create a child node $s_x \cup N$. If no such s_x exists, that branch of the tree will terminate. Repeat the process until every node terminates. If a branch terminates at S, it is a path to an exact cover. Therefore, count the number of exact covers by counting the branches that terminate at S.*

This algorithm will certainly work, but it is not very fast. To be more precise, it is exponential in the cardinality of X. Unfortunately, any algorithm for solving exact cover problems will be exponential, but we can do better.

Definition 4.2. Given (X,S), a *partial covering* is a subset S' that exactly covers a subset of X.

Algorithm 4.3. *Start with a family of sets C that only contains the empty set and note that there is one of this particular partial covering. For each set $c \subseteq C$, let m denote the multiplicity of c, the number of times that it has been reached, and find the smallest element $x \in X$ that is not covered by c. For each subset $s_x \subseteq S$ that covers x and is disjoint from c, consider $s_x \cup c$. If $s_x \cup c$ has not been reached before, mark it as a new partial covering with multiplicity m and add it C. If it has been reached before with multiplicity n, update its multiplicity to be $n+m$. If c is not S, subtract it from C. Repeat this process until the only element of C is S and take the multiplicity of S to be the number of exact covers.*

Keeping track of multiplicity allows the algorithm to minimize repeated work. That is, if a partial tiling has multiplicity m and has n children, the naïve algorithm would require a separate branch for each of the m ways to achieve the partial tiling for each of the n children. However, with this new approach, called *dynamic programming*, we only have to look at each "child" once. This "merging" of multiplicity saves the algorithm a lot of counting time. In fact, if we consider exact cover problems that are tiling problems, this algorithm is exponential in the width of the geometric shape formed by the tilable units associated with R, rather that in its area. There is, however, a trade-off, as the dynamic programming algorithm requires a lot more memory storage.

Of course, this explanation of dynamic programming is specifically about tiling, and thus is better termed dynamic tiling. Laaksonen details the method for more general programs in which time and not memory is a limiting factor [4].

5 The Exact Cover Problem Solver

David desJardins implemented dynamic programming in his code for solving tiling problems [3]. His code, TilingCounts, takes in regions, including tile types, as ASCII art and uses dynamic programming to calculate the number of perfect tilings. As it stands, his code is maximally efficient, but not generalized to exact set covers.

After the fashion of his code, to which he kindly allowed me access, I was able to write a program, the Exact Cover Problem Solver (ECPS), that does two things: converts a tiling problem to an exact cover problem, and solves exact cover problems. This way, it can be used to solve any exact cover problem, not just tiling problems, and still has the capacity to function as a tiling problem solver without the perspective of exact cover problems.

Suppose that we have a tiling problem with region R associated with an n-by-m box B and a set of nm unit squares S with tileable units $U \subseteq S$, and tile types $r_1, r_2, \ldots, r_t$ with associated tileable units $u_1, u_2, \ldots, u_t$. When acting as a tiling problem solver, the ECPS, can take in ASCII art of the tileable region in the form of an n-by-m grid of text, where elements of U are represented by hashtags and elements of U^c are represented by periods. Each tile type is similarly drawn out with ASCII art in appropriate grid sizes, with a horizontal blank line separating the region from the tile types and the tile types from one another. So long as the region is first, the order of tile types does not matter.

The region is then converted into a list of lists, which allows for a coordinate system, where the symbol at (x, y) can be found by starting at the top left and moving to the right x times and down y times. The coordinates of each element of U (*i.e.*, each hashtag) are associated with a unique number from 1 to $|U|$ in a dictionary, called a cdict, with coordinates as keys and non-negative integers as values. A similar cdict is created for each of the tile types, but only for the purpose of knowing the coordinates of each tileable unit of a given tile type.

Now, we are ready to form our tiles.

Algorithm 5.1. *Suppose that we have a tile type r_i associated with an n_i-by-m_i box and tileable units u_i. For each integer tuple (j, k) such that $0 \leq j \leq n_i - 1$, $0 \leq k \leq m_i - 1$, check the image of the geometry-preserving injection from u_i to S that sends the tileable unit u_i with coordinates (x, y) to the element of S in the region whose coordinates are $(x + j, y + k)$. If the image is in U, it is a tile; represent it as a tuple that contains the numbers associated through the cdict of the region with the coordinates of the tileable units that it covers. If the image is not in U, it is not a tile. Regardless, since this algorithm has checked all possible images of geometry-preserving injections from u_i to S, it has checked all possible images of geometry-preserving injections from u_i to U and therefore has found all possible tiles formed from r_i. Repeat this process for all tile types and obtain the set of all tiles T.*

At this point, we have successfully converted our tiling problem to an exact cover problem with a universe of $X = \{1, \ldots, |U|\}$ and family of subsets $S = T$. If one wanted to skip this conversion and deal with general exact cover problems, they would be able to enter their own n-universe and S in place of the above steps and continue with the part of the program to follow. In other words, despite the terms "tiles" for S, "tile" for subset, and "partial tiling" for partial covering, which were chosen largely for consistency with the beginning of the program, there is nothing about the second half of the program that relies on tiling or regions.

Definition 5.2. Given a partial tiling or tile t and an n-universe, the *key representation* or *key rep* of t is the non-negative number in base 10 associated with the binary sequence of length n that contains a 1 in the i'th digit if and only if $i \in t$.

These key reps efficiently encode the information of which tileable units each tile covers, which will later allow us to compare the compatibility (*i.e.*, pair-wise disjointedness) of partial tilings and new tiles when it comes time to implement dynamic programming. We will search for offspring tiles in the dynamic programming with a list of lists, called a q-set, where the i'th list contains all of the the key reps of tiles that cover i. Lastly, we create a list called counts, which contains an integer, which is 0 at first and increases by one each time an exact cover is found, and n dictionaries, where the i'th dictionary contains the keys reps whose smallest uncovered element of U is i as keys and their multiplicities as values. To start, the first dictionary has $\{0 : 1\}$ to represent that there is a trivial partial tiling with multiplicity 1, and the other dictionaries are empty, since there are no other partial tilings yet. As dynamic programming is implemented, these other dictionaries will be updated accordingly.

Algorithm 5.3. *Iterate over the integer i for $1 \leq i \leq n$. At the i'th step, for each key reps in the i'th dictionary of counts (*i.e.*, the ones that need to have i covered), for each key rep in the i'th q-set (*i.e.*, the tiles that can cover i), if the key rep for the tile, k_1, and the key rep of the partial tiling, k_2, do not have mutual 1's, take the sum $k_3 = k_1 + k_2$ to be a new partial tiling. If k_3 is a perfect tiling, update the count of perfect tilings. Otherwise, update the multiplicity of k_3 accordingly, check for the smallest $j \in U$ that is uncovered, and send k_3 along with its multiplicity as a key/value pair to the j'th dictionary of counts to be further tiled. After the process is complete for all i, take the count of perfect tilings to be the number of exact covers.*

6 Future Work

The program right now is able to do exactly what it says: it converts tiling problems to exact cover problems and it solves exact cover problems, but there are a few ways in which it can be improved. For one thing, the entire code is currently written in Python, which is good for the most part, but the computationally-intensive part of the code, the part dealing with dynamic programming, ought to be written in a faster language, like *C*++ for further efficiency. Furthermore, there is a way to do it without worrying about dictionaries, which would speed up the process significantly. To that end, I intend to transfer the code to Cython and adjust that part accordingly with ideas presented to me by Greg Kuperberg.

I would also like to implement a function that would find the optimal way to enumerate the region in order to minimize, or at least reduce, the amount of work for the dynamic programming part of the code. This involves issues of pathwidth optimization. The program currently numbers the tileable region from left to right and up to down, but that might not always be the most efficient numbering system. Given a region and tile types, we ideally want to find a way to number the region so that the ECPS has to do the least number of iterations. In effect, this means that we want to minimize the number of choices that it has to iterate over when we are trying to find a cover for the i'th tileable unit, which is related to how we choose to number the tileable units. Bodlaender [1] relates the notion of treewidth, as formalized by Robertson and Seymour [8, 7], to the efficient construction of combinatorial solutions and it is not difficult to see the connection to efficient enumeration from there. This connection was further demonstrated by Samer and Szeider [9] for #SAT and, by the equivalence of #SAT and #ExactCover, we can apply it towards constructing a more efficient version of the ECPS that uses treewidth rather than pathwidth.

Lastly, this code is very flexible in its ability to solve general exact cover problems, which means it has potential for more capabilities that I intend to implement, such as calculating the number of perfect tilings of a region with specified tile types that satisfy specific criteria (*e.g., symmetry under rotation*). Needless to say, my work on making the ECPS the best it can be continues.

References

[1] Hans L. Bodlaender. “Dynamic programming on graphs with bounded treewidth”. In: *Automata, languages and programming (Tampere, 1988)*. Vol. 317. Lecture Notes in Comput. Sci. Springer, 1988, pp. 105–118.

[2] John H. Conway and Jeffrey C. Lagarias. “Tiling with polyominoes and combinatorial group theory”. In: *J. Combin. Theory Ser. A* 53.2 (1990), pp. 183–208.

[3] David desJardins. *TilingCount*. May 15, 2021.

[4] Antti Laaksonen. *Guide to competitive programming: Learning and improving algorithms through contests*. 2nd ed. Undergraduate Topics in Computer Science. Springer, 2020.

[5] James Propp. “Tilings”. In: *Handbook of enumerative combinatorics*. Discrete Math. Appl. (Boca Raton). CRC Press, 2015, pp. 541–588.

[6] David P. Robbins. “The story of 1,2,7,42,429,7436,...”. In: *Math. Intelligencer* 13.2 (1991), pp. 12–19.

[7] Neil Robertson and Paul D. Seymour. “Graph minors. I. Excluding a forest”. In: *J. Combin. Theory Ser. B* 35.1 (1983), pp. 39–61.

[8] Neil Robertson and Paul D. Seymour. “Graph minors. II. Algorithmic aspects of tree-width”. In: *J. Algorithms* 7.3 (1986), pp. 309–322.

[9] Marko Samer and Stefan Szeider. *A fixed-parameter algorithm for #SAT with parameter incidence treewidth*. arXiv:cs/0610174. 2006.