Opt Art

"How to create works of art through optimization problems."

Robert Bosch
Oberlin College

Optimization is the branch of mathematics concerned with finding the best way to complete a task. Certainly, some tasks are easy, and we need not rely on optimization to tackle them. But many others are much more difficult, so much so that we may have very little hope of completing them satisfactorily—let alone optimally—without optimization.

For example, suppose a friend of ours does volunteer work for a Meals-on-Wheels program. Once a week she bikes to the Meals-on-Wheels headquarters and picks up meals and a list of twenty names and addresses. She then gets in the Meals-on-Wheels van, delivers the meals, and returns to headquarters. Her goal is to drop off the meals in an order that will minimize the number of miles she'll travel, as this will minimize fuel consumption, pollutant emissions, and the amount of time she'll spend on the job.

Is her task—planning the route she'll take—an easy one? It depends. If all of the addresses are on the same road, then it is extremely easy; the optimal route will be obvious to anyone who takes a look at a map. But if not, it can be extremely difficult, especially in the case in which the addresses appear to have been scattered about the city at random. (Why? Why not just list and evaluate every single route? The answer is that there are $20! \approx 2.43 \times 10^{18}$ routes, one for every permutation of the 20 addresses. Even if our friend has a laptop that can evaluate one trillion (10^{12}) routes per second, she'll have to run it for about 28 days if she wants to find the optimal route via complete enumeration!) Incidentally, this task is an instance of the Traveling Salesman Problem (TSP), one of the most difficult, important, and well-studied problems in the optimization field.

Optimization has a seemingly unlimited number of applications. It has been put to good use in a large number of diverse disciplines: advertising, agriculture, biology, business, economics, engineering, manufacturing, medicine, telecommunications, and transportation (to name but a few). In this article, we showcase its amazing utility by describing some applications in the area of art, which at first glance would seem to have no use for it whatsoever!

Photomosaics

A photomosaic is, as the name suggests, a mosaic comprising photographs. When we examine a photomosaic from up close, we are able to identify each individual "building-block" photograph. When we back away from it, we lose this ability, but we gain something else: our eyes somehow manage to merge the arrangement of photographs into a recognizable image. For example, in Figure 1, using digits 1, 2, 3, 5, and 8, we see Fibonacci in a photomosaic with dimensions 34×55.

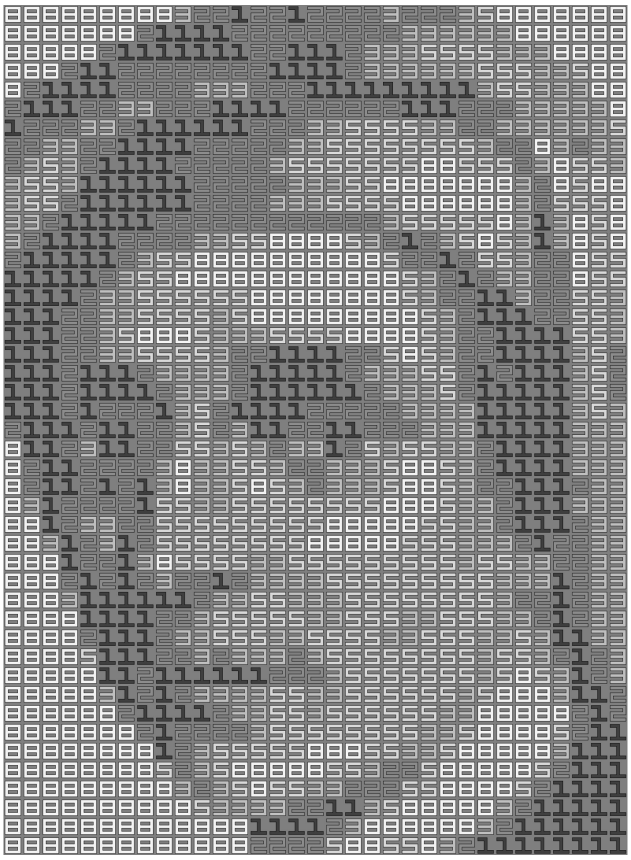


Figure 1. Photomosaic of Fibonacci using numbers 1,2,3,5, and 8.

In this section, we describe how to use optimization to create an $m \times n$ photomosaic (one with m rows and n columns of building-block photographs) that resembles a given target image from photographs that belong to a given set F of building-block photographs. Usually, each photograph $f \in F$ is square (a $k \times k$ array of pixels) and can be used no more than some given number u_f times, with no rotations or reflections. (In our opinion, the best photomosaics have $u_f = 1$ for each $f \in F$.) To keep things simple here, we assume that each photograph is black-and-white, and we denote the average brightness of photograph f by $b_f \in [0,1]$, where 0 stands for a completely black photograph and 1 stands for a completely white one.

Accordingly, we begin by partitioning both our target image and our initially blank canvas into m rows and n columns of

congruent squares. We denote the brightness of the row-i-column-j square—square (i, j)—of our target image by β_{ij}, using the same 0-to-1, black-to-white scale we use for the b_fs. Our task amounts to placing photographs from F onto our canvas, one photograph per square. Our goal is to pick the photographs and place them on the canvas in such a way that the arrangement of photographs resembles our target image as closely as possible.

This task is very well suited to optimization. When translated into the language of mathematics, it becomes an *integer program* (IP), an optimization problem with a linear objective function, linear constraints (inequalities and/or equations), and variables that take on integer values:

Photomosaic IP

$$\text{minimize} \quad \sum_{f\in F}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(b_f-\beta_{ij}\right)^2 x_{fij}$$

$$\text{subject to} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{fij} \le u_f \quad \text{for each } f \in F$$

$$\sum_{f\in F} x_{fij} = 1 \qquad \text{for each } 1 \le i \le m,\ 1 \le j \le n$$

$$x_{fij} \in \{0,1\} \qquad \text{for each } f \in F,\ 1 \le i \le m,\ 1 \le j \le n$$

Note that we have a binary variable x_{fij} for each photograph $f \in F$ and each square (i, j) of our canvas. We interpret $x_{fij} = 1$ to mean that we should place photograph f in square (i, j), and $x_{fij} = 0$ to mean that we definitely should not make this particular placement.

The idea behind the objective function is fairly simple: Suppose we place photograph f in square (i, j) (i.e., we set $x_{fij} = 1$). If the brightness of f is equal to the brightness of square (i, j) in our target image, we've dealt with square (i, j) as well as anyone could have. But if not (the more likely case), we should charge ourselves a cost. A reasonable cost is $(b_f - \beta_{ij})^2$, the square of the discrepancy between the brightness values. Hence the objective function is the total cost we charge ourselves. Our goal is to make this cost as small as possible.

The constraints are even easier to interpret. The first set of constraints ensures that no photograph $f \in F$ is used more than u_f times. The second set of constraints guarantees that each square (i, j) of our canvas receives exactly one photograph.

Although integer programming is NP-hard (which means that there is likely to be no efficient algorithm that can solve any integer programming problem), some classes of integer programs are quite easy to solve. The Photomosaic IP, being an integer programming formulation of an instance of the Assignment Problem, is very easy to solve. It turns out that if we were to "forget" that the variables must take on integer values and solve the integer program as a linear program (LP)—replacing each $x_{fij} \in \{0,1\}$ with $0 \le x_{fij} \le 1$—we would be certain to get lucky: the optimal solution to this linear program is guaranteed to be integer-valued! Moreover, there are efficient non-LP-based algorithms for solving instances of the Assignment Problem.

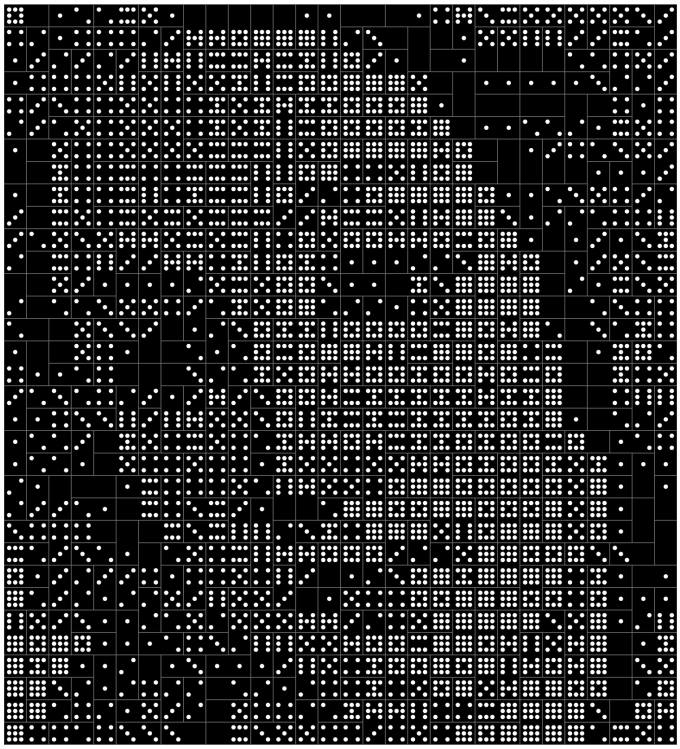


Figure 2. A domino portrait of Venus from Botticelli's "The Birth of Venus" created with nine complete sets of double-nine dominoes.

Thus, from an optimization standpoint, photomosaics are easy to make! (But note that this does not imply that it is easy to make a *good* photomosaic. Optimization cannot help us solve the difficult task of selecting a set F of building-block photographs that will fit well with—or perhaps provide commentary on—the target image.)

Historical Note

Of the numerous artists who have constructed photomosaics, Robert Silvers is the most well known. His algorithm, which is described in his 1996 MIT MS Thesis, *Photomosaics: Putting Pictures in Their Place*, is *not* based on treating the Photomosaic Problem as an Assignment Problem. It is a greedy algorithm. Silvers writes, "Currently [my] mosiacs are made from the top down. The negative consequences of this are that the quality of matches is worse at the bottom of the mosaic than at the top because the best images are used up first."

Domino Artwork

In this section we describe how to create a portrait out of *complete* sets of double-nine dominoes, as in Figure 2. Here,

our task is to place s sets of double-nine dominoes on the canvas—with each domino positioned either horizontally or vertically, covering precisely two squares of the canvas—in such a way that the resulting arrangement resembles the target image as closely as possible.

Note that in place of a set of building-block photographs F, we have a set of double-nine dominoes $D = \{d = (d_1, d_2) : 0 \le d_1 \le d_2 \le 9\}$, and we need to use each domino $d \in D$ exactly s times. Since domino $d = (d_1, d_2)$ is black and has d_1 white dots painted on one square and d_2 white dots painted on the other, domino $d = (d_1, d_2)$ can be thought of as a domino-shaped photograph, half of which has brightness d_1 and half of which has brightness d_2, with both brightness values measured on a 0-to-9, black-to-white scale.

Also note that since there are 55 dominoes per set, we need to make sure that when we partition the target image and canvas into m rows and n columns of congruent squares, m and n satisfy $mn = 110s$. The portrait in Figure 2 has $m = 33$, $n = 30$, and $s = 9$.

Finally, note that it is convenient here to denote the brightness of the row-i-column-j square of our target image by an integer $0 \le \beta_{ij} \le 9$ (or if we want higher resolution, with a real number $-0.5 \le \beta_{ij} \le 9.5$). A completely black square will be given a brightness of 0 (or –0.5), and a complete white square will be given a brightness of 9 (or 9.5).

Recall that constructing a photomosaic requires us to make a yes-no decision for each possible assignment of a photograph f to a square (i, j) of the canvas, and that in our Photomosaic IP we modeled this via binary variables x_{fij}. Constructing a domino portrait is more complicated in that we need to make a yes-no decision for each possible assignment of a domino d to a *pair* of adjacent squares of the canvas. But if we construct the set of all adjacent pairs of squares

$$P = \{\{(i,j),(i+1,j)\} : 1 \le i \le m-1,\ 1 \le j \le n\} \cup \{\{(i,j),(i,j+1)\} : 1 \le i \le m,\ 1 \le j \le n-1\},$$

we can then proceed as we did before: we can introduce a binary variable x_{dp} for each domino d in D and each pair p in P. We interpret $x_{dp} = 1$ to mean that we should place domino d on the board in such a way that it covers the squares in pair p; we interpret $x_{dp} = 0$ to mean that we shouldn't do this. And we quickly arrive at the following integer program:

Domino IP

$$\text{minimize} \quad \sum_{d \in D} \sum_{p \in P} c_{dp} x_{dp}$$

$$\text{subject to} \quad \sum_{p \in P} x_{dp} = s \qquad \text{for each } d \in D$$

$$\sum_{d \in D} \sum_{\substack{p \in P: \\ p \ni (ij)}} x_{dp} = 1 \quad \text{for each } 1 \le i \le m,\ 1 \le j \le n$$

$$x_{dp} \in \{0,1\} \qquad \text{for each } d \in D,\ p \in P.$$

The objective function measures the total cost of the resulting arrangement of dominoes; c_{dp} is the cost of placing domino d so that it covers the squares in pair p. Our method for computing c_{dp} is easy to understand, but hard to capture in a concise formula. Suppose we place domino $d = (3,5)$ so that it covers the squares of pair $p = \{(10,10), (10,11)\}$ and the squares have brightness values of $\beta_{10,10} = 6$ and $\beta_{10,11} = 4$. If we place the domino with its '3' in square (10,10) and its '5' in square (10,11), we'll charge ourselves a cost of $(3–6)^2 + (5–4)^2 = 10$. If we place the domino with its '3' in square (10,11) and its '5' in square (10,10), we'll do much better, incurring a cost of only $(3–4)^2 + (5–6)^2 = 2$. In this example, $c_{dp} = \min\{10,2\} = 2$. In general, if $d = (d_1, d_2)$, and $P = \{(i_1, j_1), (i_2, j_2\}$, we have

$$c_{dp} = \min\{(d_1 - \beta_{i_1 j_1})^2 + (d_2 - \beta_{i_2 j_2})^2, (d_1 - \beta_{i_2 j_2})^2 + (d_2 - \beta_{i_1 j_1})^2\}.$$

As for the constraints, they are very easy to understand. The first set makes sure that all of the dominoes are placed on the canvas. The second set makes sure that each square of the canvas is covered by exactly one domino.

At first glance, the Domino Problem appears to be another instance of the Assignment Problem. After all, we are assigning dominoes to pairs of adjacent squares of the canvas, making sure that each domino is used s times. But note that instead of requiring that each *pair* of adjacent squares receives

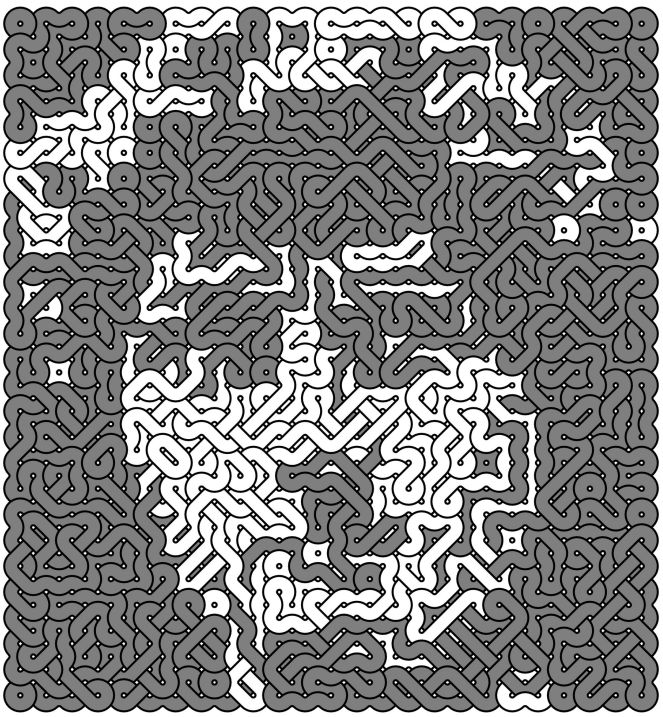


Figure 3. Michaelangelo's "David" using diamond-shaped tiles with edge constraints.

at most one domino, we require something stronger: that each *square* be covered by exactly one domino. Consequently, the Domino Problem is actually an instance of an Assignment Problem *with side constraints*.

When one adds side constraints to an easy-to-solve problem, several things can happen. The best-case scenario is that the problem remains easy to solve (i.e., one can find an efficient algorithm for solving instances of the "new" problem). The next-to-best-case scenario is that the problem becomes hard to solve in theory, yet is still easy to solve in practice (i.e., there exist some instances that appear to require an enormous amount of time to solve, but most instances encountered on a day-to-day basis are easily handled). The worst-case scenario is that the problem becomes hard to solve, both in theory and in practice. The Domino Problem falls into the second category.

Historical Note

Ken Knowlton, one of the pioneers of the field of computer graphics, was the first artist to construct mosaics using complete sets of dominoes, though his techniques differ from what we describe here.

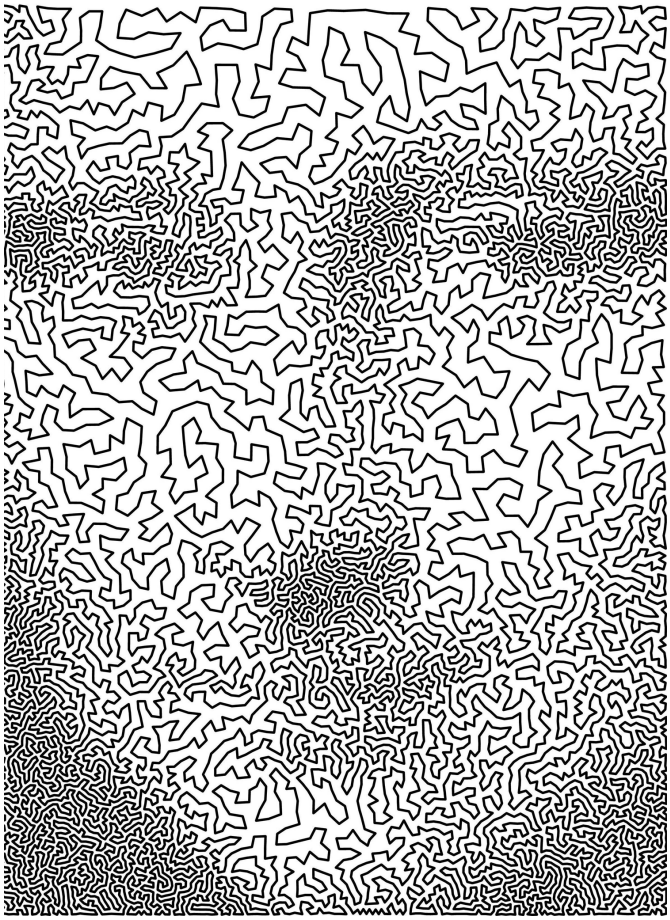
Other Tiles

Another possibility is to use tiles instead of dominoes or building-block photographs and require that the edges of neighboring tiles match. We can use, for example, square tiles that have precisely two white and/or black strands drawn on them, with exactly one end of a strand exiting each side of the square. The resulting mosaics resemble Celtic knot drawings, as in Figure 3.

Here, we have a binary variable x_{tij} for each tile t and each square (i, j) of the canvas, and we interpret $x_{tij} = 1$ to mean that we should place tile t in square (i, j) and $x_{tij} = 0$ to mean that we shouldn't. Our objective is to minimize the total cost of our tile arrangement, and we have two sets of constraints: constraints that ensure that each square receives exactly one tile, and constraints that guarantee that the edges of neighboring tiles match. As in the case of the Domino Problem, we end up with an Assignment Problem with side constraints.

Continuous Line Drawings

We can also use optimization to create continuous line drawings. The idea is very simple: First, we place dots down on a blank canvas in such a way that the group of dots resembles the target image. Next, we construct an instance of the TSP, viewing the dots as a collection of cities. Here, the salesman is assumed to be able to travel as the crow flies, so city-to-city distances are given by the Euclidean formula. Next, we use a good TSP heuristic (the Lin-Kernighan heuristic from the Concorde TSP package by Applegate, Bixby, Chvátal, and Cook) to obtain a high quality (but not necessarily optimal) solution to the TSP instance. Finally, we draw the salesman's tour.



All figures provided by Robert Bosch, dominoartwork.com. All rights reserved.

Figure 4. This continuous line drawing of Leonardo DaVinci's "Mona Lisa" uses 10,000 cities and is topologically equivalent to a circle.

The resulting picture is a continuous line drawing. Since the TSP instance is Euclidean, the tour will not intersect itself. (Can you see why?) It will be topologically equivalent to a circle! ■

Further Reading

R.A. Bosch, "Constructing domino portraits," in *Tribute to a Mathemagician*, ed. B. Cipra et al., A.K. Peters, 2004, 251-256.

R. Bosch and A. Herman, "Continuous line drawings via the traveling salesman problem," *Operations Research Letters* 3 (2004) 302-303.

C.S. Kaplan and R. Bosch, "TSP Art," *Proceedings of Bridges 2005: Mathematical Connections in Art, Music and Science* (2005) 301-308.