Searching for Additive-Multiplicative Magic Squares

Krit Boonsiriseth, CJ Quines, Steven Raphael

16 May 2023

Abstract

It is unknown whether there are 6×6 magic squares that are simultaneously additive and multiplicative. With less than 10,000 hours of single-core compute, we find over 750,000 6×6 semi-magic squares, which are magic except for their diagonals. The best semi-magic square is almost magic, except for one diagonal with a different sum. Code is public at https://github.com/talkon/magic-squares.

1 Introduction

An *(additive) semi-magic square* is an $n \times n$ grid of distinct positive integers, whose rows and columns sum to the same number, called its *magic sum*. A semi-magic square whose main diagonals also sum to the magic sum is called a *magic square*. If, instead of magic sums, the square had a *magic product*, we call it a *multiplicative (semi-)magic square*; an *additive-multiplicative (semi-)magic square* has both.

Perhaps unintuitively, smaller magic squares are generally harder to construct than larger ones, as there are more constraints relative to the number of variables. An open problem is to find the smallest possible additive-multiplicative magic square. The smallest known is the 7×7 square given below, which was found in 2016 by Sébastien Miquel using around 600 hours of computing on a personal computer. This square, shown in Table 1, has magic sum 465, magic product $2^{10} \cdot 3^7 5^3 7^2 11 = 150885504000$, and largest number 252.

126	66	50	90	48	1	84
20	70	16	54	189	110	6
100	2	22	98	36	72	135
96	60	81	4	10	49	165
3	63	30	176	120	45	28
99	180	14	25	7	108	32
21	24	252	18	55	80	15

Table 1: Miquel's 7×7 additive-multiplicative magic square.

Throughout the paper we call the magic product P and the magic sum S; for example, Miquel's square has $S = 465$ and $P = 2^{10}3^7 5^3 7^2 11$. As finding magic squares is hard, existing progress is specified in terms of semi-magic squares satisfying diagonal constraints. We adopt the following abbreviations to discuss partial progress:

- A 0-*type square* has no magic diagonals.
- An S-*type square* has one diagonal with magic sum.
- A P-*type square* has one diagonal with magic product.
- An SP-*type square* has one magic diagonal (with both magic sum and product.)

- When both diagonals of the square has some magic properties (sum or product), we use the plus sign + and the type of each of the diagonals. For example, an $S+P$*-type square* is a square whose one diagonal has the magic sum, and the other diagonal has the magic product.

In Table 2 we summarize the latest known progress using this notation.

Year	Author	Size	Type	S	P
2007	Lee Morgenstern	6×6	0	327	$2^{10}3^{4}5^{3}7^{2}$
2007	Lee Morgenstern	6×6	$S+P$	289	$2^{11}3^{4}5^{3}7^{1}11^{1}$
2008	Lee Morgenstern	6×6	$S+S$	360	$2^{10}3^{6}5^{3}7^{1}13^{1}$
2010	Toshihiro Shirakawa	7×7	$S+S$	310	$2^{10}3^{4}5^{2}7^{2}11^{1}13^{1}$
2010	Toshihiro Shirakawa	7×7	$SP+P$	380	$2^{11}3^{5}5^{2}7^{2}11^{1}13^{1}$
2016	Sébastian Miquel	7×7	$SP+SP$	465	$2^{10}3^{7}5^{3}7^{2}11^{1}$

Table 2: Best known progress for finding additive-multiplicative magic squares. [1]

For our project we sought to find 6×6 magic squares. Our main contributions in this paper are describing and releasing the code we used to do our search, as there seems to be no existing project doing so. We also analyze the results from around a year of searching to produce quantitative estimates of the difficulty of finding such magic squares. Through our searches, we have also found $SP+S$-type and $SP+P$-type magic squares.

The rest of the paper is organized as follows. In section 2 we describe the algorithm design, and section 3 describes our implementation. In section 4 we describe how we conducted our search experiments, and present the results in section 5. We evaluate our performance optimizations in section 6, and then discuss our results in section 7.

2 Algorithm design

The algorithm we use is similar to Miquel's, and has three main phases:

1. **Enumeration**: Fix S and P, then enumerate all sets of 6 integers with this sum and product. We call such sets *vectors*.
2. **Arrangement**: Through backtracking, find all ways to arrange 12 vectors to form 6 rows and 6 columns. Note this is a purely combinatorial step; this can be done while ignoring S and P. If we place 12 such vectors, we produce a semi-magic square, which we output.
3. **Postprocessing**: For each semi-magic square, we check its *traversals*. A traversal is a set of 6 squares, no two in the same row or column. By permuting the rows and columns, any traversal can be made into a diagonal. The postprocessing stage counts how many traversals have sum S and product P, and summarizes the results.

We provide high-level pseudocode for enumeration and arrangement in algorithm 1 as these are the main ones; postprocessing is a straightforward implementation.

Almost all time is spent in Search, a backtracking procedure that tries to arrange the vectors into rows and columns. To discuss Search, we first discuss algorithm 2, which represents the backtracking state. The most important part of the state are *t.rows* and *t.cols*, which represent the vectors that have been taken for that axis, and maintain the vectors that can still be added.

Algorithm 1 Enumeration and arrangement

```
Enumerate($P$)
    $allVecs$ = list of sorted 6-vectors of distinct positive integers with product $P$
    group $allVecs$ by their sum $S$
    for $vecs$ in $allVecs$ with sum $S$
        while $length(vecs) \geq 12$
            remove each vector in $vecs$ with an element not in another vector
            if none removed, break
        if $length(vecs) < 12$, remove $vecs$ from $allVecs$
    return $allVecs$

Arrange($vecs$, $S$)
    filter $vecs$ for vectors with sum $S$
    $elts$ = distinct elements in $vecs$, sorted in descending order of frequency
    $vecs$ = replace each element of each vector in $vecs$ with its index in $elts$
    sort $vecs$ by descending max label
    $g$ = MakeGlobalState($vecs$)
    $table$ = MakeSearchState($g$)
    return Search($g$, $table$)
```

In the global state, we use *g.inters* to update *axis.valid*. Each new row should have have an intersection of size 0 with previous rows, and of size 1 with previous columns; and the same holds for columns. In theory, all we need in Search is *t.rows*, *t.cols*, and *g.inters*. This makes Search solve a purely combinatorial problem: find 6 indices in *t.rows.valid*, and 6 indices in *t.cols.valid*, such that all indices are distinct, all pairs of rows and pairs of columns are disjoint, and all pairs of rows and columns have intersection size 1.

The rest of the state is used for a maximum label heuristic. Recall that *g.vecs*, and thus *axis.valid*, is sorted from highest to smallest label. That means there's an index where, for every vector after that index, their maximum labels are too small to match the maximum unmatched label. This is why we have *t.unmatched*, a bitarray storing the table's unmatched labels, which we update with *g.bitarrays* whenever we add a new vector.

The heart of the algorithm is Search, pseudocode in algorithm 3. Note that in the actual algorithm, we'd print solutions in line 2. The crucial heuristic here is in lines 7–8, where we enforce that the maximum unmatched number is in the next vector. This is a symmetry breaking constraint and empirically results in good pruning. A critical part of the code here is FillValids function, which filters the remaining possible rows and columns based on their intersections with the last vector added. It's where we spend most of the time, although not a majority of the time.

The Search function contains a few other symmetry breaking constraints: the first vector added must be a row, must contain the maximum element in the magic square, and must have a smaller index than the first column.

Before starting the search, the Arrange procedure relabels each number based on descending order of frequency across all vectors with the given sum. The least-frequent elements are assigned the largest labels, and the most-frequent elements are assigned the smallest labels. This relabeling process is designed to reduce the branching factor of the search based on the maximum label heuristic: at every branch of the search, the element with the maximum label is likely to be contained in the

Algorithm 2 Search state

```
MakeGlobalState($vecs$)
    $g = \varnothing$                                  global search state (doesn't change during search)
    $g.vecs = vecs$                                    $vecs[i]$ is 6-vector of labels; sorted by max label
    $g.bitarrays$ = make bitarrays                     $bitarray[i][j] = j \in vecs[i]$
    $g.inters$ = make inters                           $inters[i][j] = length(vecs[i] \cap vecs[j])$
    return $g$

MakeAxisState($g$)
    $axis = \varnothing$                               search state for rows or cols)
    $axis.vecs = \varnothing$                          list of vec indices taken for axis
    $axis.valid = [0 \,..\, length(g.vecs) - 1]$       list of vec indices that can go in
    return $axis$

MakeSearchState($g$)
    $t = \varnothing$
    $t.rows$ = MakeAxisState($g$)
    $t.cols$ = MakeAxisState($g$)
    $t.unmatched = \varnothing$                        $unmatched[i]$ if label $i$ isn't in both rows and cols
    return $t$
```

smallest number of remaining vectors. Relabeling empirically improves the performance of the search.

Previously, we tried pruning by ensuring that every vector added has a smaller index than any future vectors. We added a symmetry breaking constraint: every vector's maximum element must be at least the maximum unmatched element. This method is incompatible with our current method of pruning, and empirically results in a larger branching factor.

We also tried several algorithm-level optimizations that did not result in any speedup. For example, we tried representing the list of remaining vectors with a bitset, so FillValids could filter the vectors with a series of bitwise instructions. This did not result in any speedup, possibly because iterating over the list of remaining vectors is more intensive in bitset format.

3 Implementation

Our first implementation of enumeration, arrangement, and postprocessing was in Python, which we call our baseline implementation. We spent some time making building and testing infrastructure too. We then translated the arrangement step to C, as it was the most intensive step. Because arrangement is task-parallelizable, there was no need to add parallelism proper; we could instead run several copies of the program. The overhead is negligible enough that this works well.

We made several optimizations to the C code after doing some benchmarks. For example, we learned from profiling that the hottest parts of algorithm 3 was in FillValids, so we carefully optimized that part, like making line 21 branchfree. We later rewrote FillValids to take advantage of SIMD, using AVX-512 instructions to run it vectorized.

The main difficulty in vectorizing it is that it implicitly involves a gather instruction. The logic in

Algorithm 3 Search

```
1  SEARCH($g, t$)
2      if $t$ is a solution, return
3      for $axis$ in $rows$, $cols$
4          if $axis = cols$ and $t.rows.vecs = \emptyset$, break
5          if $length(t.axis.vecs) = 6$, continue
6          for $vec$ in $t.axis.valid$
7              if $max(g.vecs[vec]) < max(t.unmatched)$, break
8              if $max(t.unmatched) \notin vec$ and ($axis = cols$ or $t.rows.vecs \neq \emptyset$), continue
9              $u = copy(t)$
10             $u.axis = t.axis \cup \{vec\}$
11             $u.unmatched = t.unmatched \oplus g.bitarrays[vec]$
12             $minVec = 0$ if $length(t'.rows) > 0$ or $length(t'.cols) > 0$ else $vec$
13             FILLVALIDS($g$, $t.rows$, $u.rows$, $[axis = cols]$, $vec$, $minVec$)
14             FILLVALIDS($g$, $t.cols$, $u.cols$, $[axis = rows]$, $vec$, $minVec$)
15             SEARCH($g$, $u$)
16
17 FILLVALIDS($g, oldAxis, newAxis, numInters, newVec, minVec$)
18     $newAxis.valid = \emptyset$
19     for $vec$ in $oldAxis.valid$
20         if $vec < minVec$, continue
21         if $g.inters[newVec][vec] \neq numInters$, continue
22         $newAxis.valid = newAxis.valid \cup \{vec\}$
```

FILLVALIDS is that we want to filter for vec that satisfy $g.inters[newVec][vec] = numInters$. We can find these using a mask, but to produce the output list of vectors, we need to gather the ones that satisfy the mask.

Optimizations that we tried and did not see a speed improvement included changing line 19 in algorithm 3 to use bitsets to cut a for loop, or replacing the recusivecall in line 15 with a simulated stack. Empirically it seems that the code is compute-bound, so it's unclear what low-level optimizations can help.

4 Search methodology

For our large-scale search, we deployed the algorithm we developed on MIT SuperCloud, utilizing up to eight CPU nodes at once. Each CPU node has an Intel Xeon 8260 Platinum processor with 48 CPU cores, and 192 GB shared memory. Enumeration and arrangement are task-parallelized across these CPU cores (with each core running the search on a different value of P), and postprocessing on the output files is done locally on a personal computer.

In order to run our search on SuperCloud, we made some minor modifications to our tooling, including the following.

- We modified the arrangement algorithm to take in a target node count (on the order of 10^{11}), and stopping when the target count is exceeded. Since nodes-per-second count is fairly constant across values of S, P, this ensures that we utilize each core roughly equally. As we discuss later in this section, this also increases our search efficiency when compared to an exhaustive search on each value of P.

- We updated our shell scripts that are used to run the search to automatically start the search at the next value of S for each P.

With these changes, running a search batch on SuperCloud amounted to running a job submission script on an input file containing target values of P, one per line. This allowed us to scale up our search relatively quickly. For larger values of P, however, our enumeration step would reach the memory bottleneck (SuperCloud CPU nodes have 4 GB of memory per core when all cores in a node are used), so we had to separate out the enumeration and the arrangement steps.

Across all our search batches, a total of 414.32 days of single-core compute time was spent on our arrangement step (including around 15 days of compute on a personal computer during the development process of our algorithm). We did not precisely track the time spent on enumeration and postprocessing, but we estimate it to be a small fraction of the time spent on the arrangement step. The overall statistics from the arrangement step are given in Table 3.

number of P searched	1,136
number of (P, S) searched	7,273,367
maximum number of (P, S)-vectors	5,551
semi-magic squares found	758,949
total serial runtime	414.32 days
total number of search nodes	1.58×10^{14}
overall nps	4,416,000
runtime per semi-magic square	47.17 seconds

Table 3: Overall statistics from the arrangement step in our search

As shown in Table 3, a total of 1,136 distinct values of P are included in our search, including values like $2^{14}3^8 5^5 7^3$, $2^{13}3^8 5^4 7^2 11^1$, and $2^{11}3^6 5^4 7^2 11^1 13^1$. Of these, 645 have been exhaustively searched, and 491 have been partially searched. The values of P are chosen manually for each search batch, for both the enumeration and arrangement steps.

The choice of P is based on some heuristics and from data from the previous values of P searched. For example, values of P with "nicely decreasing" exponents, like $2^{13}3^8 5^4 7^2 11^1$, tend to have more semi-magic squares. When a value of $P = 2^{a_1}3^{a_2}5^{a_3}\cdots$ works well (has many semi-magic squares per node count), it is likely that $2^{a_1+1}3^{a_2}5^{a_3}\cdots$ will also work well.

When running an enumeration batch, the included values of P is also manually chosen to fit within the 192 GB of memory on each SuperCloud node; we were able to estimate the amount of memory each $P = 2^{a_1}3^{a_2}\cdots$ would take using the following formula:

$$\text{memory usage in GB} = \frac{1}{3.2 \times 10^9} \cdot \prod_i \binom{i+5}{5}.$$

We heuristically found that the rate of finding semi-magic squares generally decreases as S increases, so semi-magic squares are found most quickly near the beginning of the search. Our methodology takes advantage of this observation by stopping the search for a given value of P after a target number of nodes, instead of exhaustively searching over all values of S for a given P.

5 Search results

Our search found a total of 758,949 additive-multiplicative semi-magic squares. This includes squares of type $SP + P$, $SP + S$, SP, and $P + P$ (in order of decreasing rarity). As far as we know, each of these types of squares have not been found prior to our search. Table 4 displays statistics about squares of each type in our search. The smallest known examples of each of the four new square types are given in Table 5, Table 6, Table 7, and Table 8.

(To be precise, the number shown here is the number of semi-magic squares whose rows and columns can be permuted to form a square with the given type. Also, these counts include some non-primitive squares — squares where the greatest common divisor of the numbers in the square is not 1 — but these are a negligible fraction of the squares we found.)

A summary file with further results can be located in the repository.

Type	Squares	Smallest P	S	Notes
$SP + SP$	0			None known
$SP + P$	1	$2^{13}3^55^37^213^1$ = 158,505,984,000	632	**New type**, see Table 5
$SP + S$	9	$2^{11}3^65^37^213^1$ = 118,879,488,000	577	**New type**, see Table 6
SP	519	$2^{11}3^65^37^111^1$ = 14,370,048,000	492	**New type**, see Table 7
$P + P$	721	$2^{11}3^55^37^213^1$ = 5,660,928,000	433	**New type**, see Table 8
$S + P$	3,531	$2^{11}3^45^37^111^1$ = 1,596,672,000	289	Found by Morgenstern (2007)
$S + S$	6,492	$2^{10}3^55^37^111^1$ = 2,395,008,000	386	
P	178,153	$2^83^65^37^2$ = 1,143,072,000	392	
S	427,869	$2^73^45^37^213^1$ = 825,552,000	346	
0	758,949	$2^{10}3^45^37^2$ = 508,032,000	327	Found by Morgenstern (2007)

Table 4: Types of squares found in our search

112	200	81	16	28	195
320	39	98	54	96	25
60	30	40	42	208	252
78	147	240	120	15	32
27	24	160	260	105	56
35	192	13	140	180	72

Table 5: 6×6 $SP+P$-type square with smallest P found so far in our search, with $P = 2^{13}3^55^37^213^1 =$ 158,505,984,000 and $S = 632$. Here, both diagonals have product P, and the diagonal from bottom-left to top-right has sum S.

252	42	96	125	36	26
52	15	108	128	225	49
28	64	50	39	126	270
30	210	117	84	16	120
135	90	196	12	104	40
80	156	10	189	70	72

Table 6: 6×6 $SP+S$-type square with smallest P found so far in our search, with $P = 2^{11}3^{6}5^{3}7^{2}13^{1} =$ 118,879,488,000 and $S = 577$. Here, both diagonals have sum S, and the diagonal from top-left to bottom-right has product P.

243	14	100	60	64	11
28	180	33	192	9	50
110	36	8	108	20	210
15	18	84	22	225	128
16	200	27	105	132	12
80	44	240	5	42	81

Table 7: 6×6 SP-type square with smallest P found so far in our search, with $P = 2^{11}3^{6}5^{3}7^{1}11^{1} =$ 14,370,048,000 and $S = 492$. Here, the diagonal from bottom-left to top-right has sum S and product P.

175	117	5	48	72	16
144	36	10	8	105	130
13	14	80	90	20	216
9	100	104	12	180	28
32	6	108	210	52	25
60	160	126	65	4	18

Table 8: 6×6 $P+P$-type square with smallest P found so far in our search, with $P = 2^{11}3^{5}5^{3}7^{1}13^{1} =$ 5,660,928,000 and $S = 433$. Here, both diagonals have product P.

6 Performance evaluation

We compare our current arrangement results to a baseline Python implementation written during the first week of the project, using the values $P = 2^{10}3^{4}5^{3}7^{2}$ and $P = 2^{10}3^{5}5^{3}7^{2}$ as input. This baseline implementation is semi-optimized, and already has around 4x fewer search nodes and is already around 15x faster than the naive Python implementation mentioned in our project proposal.

We perform our comparison in two steps: first, between the Python code and an optimized but unvectorized version of our current arrangement code, and second, between the unvectorized and vectorized versions of our current code. This choice is made because we used PyPy3 on a personal computer to time our baseline implementation, but PyPy3 is not available on SuperCloud (and using CPython would be even slower and consume unnecessary resources), and we used the AVX512 instruction set for vectorization, which is not available on our local machine.

Local computations are performed on an Apple Macbook Pro with an 8-core Apple M1 Pro CPU. Also note that we task-parallelized our baseline Python implementation (splitting tasks by the value of S), but ran our current optimized C code serially. To adjust for this difference, we used a multiplier of 6× to derive a speedup corrected for parallelization.

Results for $P = 2^{10}3^4 5^3 7^2$ are given in Table 9. Including vectorization, our current implementation sees a 13.7x reduction in number of search nodes, and a 545× corrected speedup in terms of nodes per second, resulting in around a 7,500× overall speedup.

$P = 2^{10}3^4 5^3 7^2$	machine	nodes	nps	**runtime (s)**
Baseline (Python, parallelized on 8 cores)	MacBook Pro	108,357,783	75,000	**1435.49**
Semi-optimized (C, serial, unvectorized)	MacBook Pro	7,914,087	5,450,000	**1.45**
Semi-optimized (C, serial, unvectorized)	SuperCloud	8,037,151	2,770,000	**2.90**
Optimized (C, serial, vectorized)	SuperCloud	8,037,151	3,480,000	**2.31**
Raw speedup without vectorization		13.7x	72x	**990x**
Correction factor from parallelization			6x	**6x**
Speedup from vectorization			1.26x	**1.26x**
Total speedup (incl. correction factor)		**13.7x**	**545x**	**7,500x**

Table 9: Performance comparison for $P = 2^{10}3^4 5^3 7^2$.

Results for $P = 2^{10}3^5 5^3 7^2$ are given in Table 10. For this value of P, we see a similar reduction in number of search nodes, and around a 960× corrected speedup in terms of nodes per second, resulting in around a 13,350× overall speedup. This shows that our current algorithm scales at least as well as the baseline algorithm.

$P = 2^{10}3^5 5^3 7^2$	machine	nodes	nps	**runtime (s)**
Baseline (Python, parallelized on 8 cores)	MacBook Pro	1,451,170,693	55,000	**26575.88**
Semi-optimized (C, serial, unvectorized)	MacBook Pro	104,759,431	6,700,000	**15.63**
Semi-optimized (C, serial, unvectorized)	SuperCloud	106,502,463	3,940,000	**27.04**
Optimized (C, serial, vectorized)	SuperCloud	106,502,463	5,160,000	**20.64**
Raw speedup without vectorization		13.9x	122x	**1,700x**
Speedup from vectorization			1.31x	**1.31x**
Correction factor from parallelization			6x	**6x**
Total speedup (incl. correction factor)		**13.9x**	**960x**	**13,350x**

Table 10: Performance comparison for $P = 2^{10}3^5 5^3 7^2$.

7 Discussion

7.1 Estimated remaining compute time with current algorithm

With our current algorithm, we expect that it will take on the order of 400 CPU years to find an add-mult magic square. This estimate is derived as follows. First, using our preprocessing script, we computed the number of *traversals* (sets of six cells that lie on distinct rows and columns; a traversal can be made into a diagonal using row and column permutations) with the correct magic sum or product or both. Aggregating across the 758,949 semi-magic squares, we obtain the statistics in Table 11.

	Total	S-type	P-type	SP-type
Count	$6! \cdot 758{,}949 = 546{,}443{,}820$	657,724	208,272	519
Probability		1 in 830	1 in 2600	1 in 1,050,000

Table 11: Number of traversals with given magic properties

There are $\frac{1}{2} \cdot 6! \cdot \frac{6!}{2!2!2!3!} = 5{,}400$ distinct unordered pairs of diagonals we can obtain by permuting the rows and columns of a semi-magic square, therefore we estimate that a semi-magic square has probability

$$5400 \times \frac{1}{830^2} \times \frac{1}{2600^2} \approx 1 \text{ in } 860 \text{ million}$$

of being magic. At our current search rate of ≈ 45 seconds per semi-magic square, we can estimate that it will take

$$860 \text{ million} \times 45 \text{ seconds} \approx 1{,}200 \text{ years}$$

of CPU time. Note that this is a rough estimate – for example, SP-type traversals seem to be more common than one would expect if the properties of having a magic sum and product are independent, as

$$\frac{1}{1{,}050{,}000} > \frac{1}{830} \cdot \frac{1}{2600}$$

by a factor of 2.05×. A similar calculation also gives an estimate of 1.46 and 4.58 for the number of $SP+P$ and $SP+S$-squares we should expect so far, compared to the observed values of 1 and 9. As P increases, we also expect magic traversals to become rarer, so the estimates in this section are likely underestimates. Empirically, in a semi-magic square with sum S and product P, the probability that a traversal has correct sum seems to behave like $1/S$, and the probability that a traversal has correct product seems to behave like $1/\tau(P)$, where $\tau(P)$ is the number of divisors of P.

7.2 Current bottlenecks

Unfortunately, as we exhaust smaller values of P, there are some bottlenecks that make it harder to find magic squares as we search larger values of P. First, we expect the rate at which we find semi-magic squares with some magic diagonals to decrease as the values of P get larger. The rate at which we find semi-magic squares seems to empirically remain constant, but as mentioned in the previous section, the probability that any diagonal has the correct sum or product should go down as the sums and products increase.

Additionally, as we search larger values of P, our current process of enumeration becomes more memory-intensive as more vectors need to be listed. This has already led to out-of-memory errors on SuperCloud.

7.3 Directions for future work

We suggest the following possible directions for future work.

- **Optimizing enumeration.** A low-hanging fruit for future work would be to optimize the memory usage of the enumeration code. Although enumeration still takes less time than arrangement by far, its memory use meant that as we could only enumerate so many P in parallel). As such, we had to spent some search time searching deep into each value of P, which is suboptimal as the smallest sums for each P have the highest rate of semi-magic squares.

  One possible way to do this would be to only enumerate tuples with a given product P and sum $S \leq S_{\text{max}}$ for some upper bound S_{max}.

- **Improving arrangement complexity.** Empirically, the complexity of our current algorithm, if we start with N valid vectors with the same sum and product, is roughly $N^{4.5}$. So far, we have not found any way to improve this complexity, but it seems like some improvement here would be necessary to reduce the remaining search to a reasonable amount. It might be possible to reduce the number of configurations searched by altering the search order, but the ordering already seems to be very efficient.

- **Search for 5×5 magic squares.** The problem of finding 5×5 magic squares is also open, and we have briefly tested our current code on 5×5 magic squares. It turns out that for 5×5, enumeration becomes much more of a bottleneck, as arrangement becomes much faster. However, traversals with magic properties also become much rarer, so it is likely that more compute power will be required to find a 5×5 additive-multiplicative magic square.

7.4 Contribution

- Krit: Worked on build systems, tooling, testing, and measuring, compiling, running the search, and analyzing results.
- CJ: Structured and architected the code base, wrote low-level optimizations, did external communications.
- Steven: Suggested and wrote many of the core algorithms, optimizations, and heuristics; vectorized the code base.

Things that are harder to divide between us include writing this report and project management.

Acknowledgements

The authors acknowledge the MIT SuperCloud and Lincoln Laboratory Supercomputing Center for providing HPC resources that have contributed to the research results reported within this report.

References

[1] Boyer, C. The smallest possible additive-multiplicative magic square. Retrieved from http://www.multimagie.com/English/SmallestAddMult.htm