AUTOMATING THE SEARCH FOR FASTER MATRIX MULTIPLICATION ALGORITHMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF SCIENCE WITH HONORS

Will Monroe
Advised by Virginia Vassilevska Williams and Ryan Williams
May 2013

Abstract

We present an implementation of the automated analysis proposed by Williams in her 2012 generalization of the Coppersmith-Winograd algorithm. We show $\omega < 2.372772$ using the 4th power of the CW construction, an improvement on previous 4th power bounds, in addition to confirming independent work by Williams demonstrating $\omega < 2.372711$ using the 8th power.

Acknowledgements

First and foremost, to Virginia Vassilevska Williams, for her excellent mentorship and finite but unbounded patience; and to Ryan Williams, for being my guide in the undergraduate research labyrinth and opening door after door for me.

To Meredith Hutchin and Yoav Shoham, and to my co-conspirators in the honors program, Dima Brezhnev, Bryce Cronkite-Ratcliff, and Sandy Huang, for their enthusiasm and encouragement, and for filling Friday afternoons with fascinating ideas and delicious sandwiches.

To Mehran Sahami, Eric Roberts, Jerry Cain, Phil Levis, Virginia Williams, and Alexis Wing, for helping ensure that I'll be here doing research for a little while longer. I also would not have been nearly as inclined to be a researcher had it not been for the whirlwind of excitement that was Prof. Levis' CURIS project two summers ago; my thanks to the people who made that project possible, including Ewen Cheslack-Postava, Behram Mistree, and all eight of the fun and friendly undergrads with whom I had the pleasure of working: Alex, Angela, Ben, Elliot, Emily, Jiwon, Kevin, and Kotaro.

And of course, to my parents, Bill and Carol Monroe, for their constant love and support, and for reminding me to keep things in perspective, take care of myself, and have fun. I won't forget my Number One Job.

This is only the teaser drop at the top of the roller coaster—thanks to everyone who powered me up the hill. It's going to be a wild ride.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since Volker Strassen's discovery in 1969 that $n \times n$ matrix multiplication could be accomplished in better than $O(n^3)$ time [Str69], theoretical research in matrix multiplication algorithms has seen a long staircase of gradual improvement in ω, the *exponent of matrix multiplication*. Progress had appeared to stall after Coppersmith and Winograd reached $\omega < 2.376$ [CW87], their second breakthrough in the field; no lower exponent was found until three years ago, when Stothers analyzed the fourth power of CW's tensor to obtain $\omega < 2.3737$ [Sto10], followed shortly thereafter by Virginia Vassilevska Williams' analysis of the eighth power achieving the current bound of $\omega < 2.3727$ [Wil12].

Previous analyses of Coppersmith-Winograd's algorithm and its higher tensor powers, including the analysis accomplished by Stothers, required separate, arduous derivations for each of a number of values that increased quadratically with tensor power. The major achievement of Williams' paper was the construction of an algorithm to generalize the analysis of CW-like tensors, in a way that is both mechanical and computationally tractable for large powers. A tantalizing challenge presented by Williams in her paper is the prospect of performing future analyses of a similar nature entirely by computer. In this report I present my work towards realizing that prospect, under the advisement of Virginia Vassilevska Williams and Ryan Williams.

In chapter 2, we review the nonlinear program derived in Williams' paper. Chapter 3 lists the parameters that can be varied when searching for better solutions. Chapter 4 describes details of the implementation of the search, and the results obtained using this implementation are presented in chapter 5. Finally, chapter 6 provides instructions for obtaining and running the search program and a guide to the layout of the source code.

Chapter 2

Williams' program

Following Schönhage and Coppersmith-Winograd, we instead minimize a bound on $\omega/3$, conventionally labeled τ.

We choose $\mathcal{P}$, the tensor power of the construction we are analyzing, and construct a constraint program in $\lfloor \mathcal{P}^2/3 \rfloor + \mathcal{P} + 1$ variables a_{IJK} (the "small variables"), where I, J, and K are integers, $0 \leq I \leq J \leq K$, and $I + J + K = 2\mathcal{P}$. (The STOC '12 paper includes a more general form of the program with two sets of variables, a and $\bar{a}$; in our analysis here we elect to set $a_{IJK} = \bar{a}_{IJK}$.)

The program is then: minimize τ, subject to

- *Nonnegative*: $a_{IJK} \geq 0$ for all a_{IJK}.
- *Permutations identity*:
$$\sum_{I,J,K} \text{perm}(I,J,K) a_{IJK} = 1,$$
where $\text{perm}(I,J,K)$ is the number of unique permutations of (I,J,K).
- *Rank constraint* (see section 2.1).
- *Free variable constraints* (see section 2.2).

Descriptions of the constraints follow.

2.1 Rank constraint

To compute a matrix product is to evaluate a particular *trilinear form*, a sum $\sum_{i,j,k} t_{ijk} x_i y_j z_k$, where t is the *tensor* of the trilinear form. Coppersmith and Winograd [CW87] defined the *value* $V(\tau)$ of a trilinear form to indicate its similarity to a sum of matrix product tensors.

The rank constraint is a generalization of Schönhage's *asymptotic sum inequality* [Sch81]. It relates the values $V_{IJK}(\tau)$ of a set of trilinear forms that compose the tensor being analyzed to the border rank r of the tensor.

Williams defines additional variables A_ℓ (the "large variables") to be the sum of all a_{ijk} where $\ell \in \{i, j, k\}$, times 2 if the other two indices are unique. (Equivalently, $A_\ell = \sum_{(i,j,k) \text{ if } \ell \in \{i,j,k\}} a_{ijk}$, where here the sum is over *all* triples i, j, k that sum to $2\mathcal{P}$, not just those in sorted order, and $a_{ijk} = a_{\text{sort}(i,j,k)}$.) The rank constraint in its most general form is

$$r^{\mathcal{P}} \leq \frac{\prod_{I,J,K} V_{IJK}{}^{\text{perm}(I,J,K)a_{IJK}}}{\prod_\ell A_\ell{}^{A_\ell}}.$$

In analyzing the $\mathcal{P}$ power of [CW87], we substitute Coppersmith and Winograd's border rank, $r = q + 2$, and take logs to get

$$\boxed{\mathcal{P} \log(q+2) \leq \sum_{I,J,K} \text{perm}(I,J,K) \log(V_{IJK}) a_{IJK} - \sum_\ell A_\ell \log(A_\ell).}$$

The nonlinearity in this constraint is embedded in the second term, as each A_ℓ is a sum involving small variables a_{ijk}.

2.2 Free variable constraints

When analyzing powers above the third, it is necessary to include a number of additional constraints. These constraints take the form

$$a_{IJK} \prod_{I',J',K'} a_{I'J'K'}{}^{p_{IJK}^{I'J'K'}} = 1. \tag{2.1}$$

They are constructed by solving the linear system defining A_ℓ for some subset of a_{IJK} (referred to in the implementation as the "included" set) in terms of A_ℓ and the remaining a_{IJK} (the "excluded" set, S). The included set may be chosen arbitrarily, although its size is fixed by the rank of the matrix $\frac{dA_\ell}{da_{IJK}}$. (For the second and third powers, it is possible to solve for all a_{IJK} in terms of A_ℓ—*i.e.*, the matrix has full rank—and as a result, no constraints are needed here.)

With the coefficients of the solved linear system in hand, we create one additional constraint (2.1) for each variable a_{IJK} in S. The coefficients of the solved linear system give the powers

$$p_{IJK}^{I'J'K'} = \frac{da_{I'J'K'}}{da_{IJK}},$$

where each $a_{I'J'K'}$ is "included" (not in S).

Some of the derivatives $p_{IJK}^{I'J'K'}$ may be negative. To allow for setting certain a_{IJK} to zero when searching for solutions, and to avoid numerical instability in solving the program, Williams rewrote the constraints (2.1) in the form

$$\boxed{a_{IJK} \prod_{\substack{I',J',K' \\ p_{IJK}^{I'J'K'}>0}} a_{I'J'K'}{}^{p_{IJK}^{I'J'K'}} = \prod_{\substack{I',J',K' \\ p_{IJK}^{I'J'K'}<0}} a_{I'J'K'}{}^{-p_{IJK}^{I'J'K'}}.}$$

This is the form we use in our implementation.

2.3 Computing values

In order to complete the constraint program, it is necessary to know a lower bound on the value V_{IJK} of each trilinear form that makes up the construction. Williams gives two algorithms for computing the values. The first algorithm can only be used for even $\mathcal{P}$, while the second can be applied to any tensor power.

2.3.1 Even powers algorithm

Bounds on V_{IJK} are computed recursively in terms of known bounds on the values for lower powers. The structure of the resulting expression is a product of sums:

$$V_{IJK} \geq 2 \underbrace{\left(\sum_\ell \xi_\ell\right)^{1/3} \left(\sum_\ell \psi_\ell\right)^{1/3} \left(\sum_\ell \zeta_\ell\right)^{1/3}}_{\textit{Combined products}} \cdot \underbrace{L_\Delta}_{\textit{LP result}} \tag{2.2}$$

The expressions ξ_ℓ, ψ_ℓ, ζ_ℓ, and L_Δ are computed using a set of temporary variables α_{ijk} indexed by triples similar to the a_{ijk} variables of the constraint program. (Note that a different set of α_{ijk} is constructed for each value V_{IJK}.) The triples (i, j, k) in this step are subject to the constraints that

- $i + j + k = 2\mathcal{P}$ (as in the case of the value triples);
- $i \leq I/2$, and if $i = I/2$, then $j \leq J/2$—that is, (i, j, k) is not lexicographically greater than its "complement" $(I - i, J - j, K - k)$; and
- $j \leq J$, $k \leq K$—no index of the triple may be greater than the corresponding index of the value's triple.

(Unlike the a_{IJK} above, these triples do *not* have to be in sorted order.)

From these, define $X_\ell = \sum_{i,j,k} c(i, I, \ell)\alpha_{ijk}$, where

$$c(i, I, \ell) = \begin{cases} 2 & \text{if } i = \ell = I - \ell; \\ 1 & \text{if } i = \ell \text{ or } i = I - \ell \text{ (but not both);} \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, $Y_\ell = \sum_{i,j,k} c(j, J, \ell)\alpha_{ijk}$, and $Z_\ell = \sum_{i,j,k} c(k, K, \ell)\alpha_{ijk}$.

These definitions form a linear system. As before, we will solve the linear system for some of the α_{ijk} in terms of X_ℓ, Y_ℓ, Z_ℓ, and an excluded set of α_{ijk} (Δ). This yields a matrix of coefficients $\frac{d\alpha_{ijk}}{dX_\ell}$, $\frac{d\alpha_{ijk}}{dY_\ell}$, $\frac{d\alpha_{ijk}}{dZ_\ell}$, and $\frac{d\alpha_{ijk}}{d\alpha_{i'j'k'}}$ for each $\alpha_{i'j'k'}$ in Δ.

Once we have found these coefficients, we can compute ξ_ℓ, ψ_ℓ, and ζ_ℓ, the "combined products" in the value lower bound. These are computed recursively from values of lower powers: define $W_{ijk} = V_{ijk}V_{I-i,J-j,K-k}$. (The code calls W_{ijk} "bases." When evaluating $V_{I-i,J-j,K-k}$, we can use the fact that $V_{IJK} = V_{\mathrm{sort}(I,J,K)}$.) Then

$$\xi_\ell = \prod_{i,j,k} W_{ijk}{}^{3\frac{d\alpha_{ijk}}{dX_\ell}} \qquad \ell < \lfloor I/2 \rfloor; \qquad \xi_{\lfloor I/2 \rfloor} = \begin{cases} 1 & \text{if } I \text{ is odd} \\ 1/2 & \text{if } I \text{ is even} \end{cases}$$

$$\psi_\ell = \prod_{i,j,k} W_{ijk}{}^{3\frac{d\alpha_{ijk}}{dY_\ell}} \qquad \ell < \lfloor J/2 \rfloor; \qquad \psi_{\lfloor J/2 \rfloor} = \begin{cases} 1 & \text{if } J \text{ is odd} \\ 1/2 & \text{if } J \text{ is even} \end{cases}$$

$$\zeta_\ell = \prod_{i,j,k} W_{ijk}{}^{3\frac{d\alpha_{ijk}}{dZ_\ell}} \qquad \ell < \lfloor K/2 \rfloor; \qquad \zeta_{\lfloor K/2 \rfloor} = \frac{1}{2} \prod_{i,j,k} W_{ijk}{}^{6\frac{d\alpha_{ijk}}{dZ_\ell}}.$$

At this point, as in the linear system for the final program above, if the linear system was of full rank ($\Delta = \emptyset$), we are finished; the bound on the value is given by (2.2) (with $L_\Delta = 1$).

If not, we must compute L_Δ. Its value is

$$L_\Delta = \prod_{\alpha_{ijk} \in \Delta} \delta_{ijk}{}^{\alpha_{ijk}},$$

where δ_{ijk} have a form similar to that of ξ_ℓ, *etc.*:

$$\delta_{ijk} = \prod_{i',j',k'} W_{i'j'k'}{}^{\frac{d\alpha_{i'j'k'}}{d\alpha_{ijk}}}.$$

Here, L_Δ still contains some α variables, unlike the previous expressions. Our goal is to maximize the bound on V_{IJK}, so we should maximize L_Δ. We have some freedom to choose the values of the variables α_{ijk}, with some constraints. These constraints, concisely, are that *every* α_{ijk}, whether excluded ($\in \Delta$) or included, must be nonnegative. The included α_{ijk} are defined in terms of X_ℓ, Y_ℓ, Z_ℓ and the excluded α_{ijk} by the solved linear system above. To make them depend only on the

excluded variables, we give values to X_ℓ, Y_ℓ, and Z_ℓ:

$$X_\ell = \frac{\xi_\ell}{\sum_{\ell'} \xi_{\ell'}} \qquad Y_\ell = \frac{\psi_\ell}{\sum_{\ell'} \psi_{\ell'}} \qquad Z_\ell = \frac{\zeta_\ell}{\sum_{\ell'} \zeta_{\ell'}}$$

Assuming we've computed the lower power values, X_ℓ, Y_ℓ, and Z_ℓ are then constants, making all the constraints linear. Furthermore, δ_{ijk} are also constants. Therefore, maximizing

$$\log(L_\Delta) = \sum_{\alpha_{ijk} \in \Delta} \log(\delta_{ijk}) \alpha_{ijk}$$

subject to these constraints is a linear program, which can be solved with any available LP solver. (We employ GLPK for this purpose—see section 6.1.)

2.3.2 General powers algorithm

The algorithm for finding V_{IJK} for arbitrary $\mathcal{P}$ differs from the even powers algorithm only in the indices of the α variables, the computation of the bases W, and some multiplicative factors. The most salient difference in the construction of the programs that i, j, k in $\alpha_{i,j,k}$, and ℓ in ξ_ℓ, *etc.*, are no longer integers but rather $\mathcal{P}$-tuples of integers in $\{0, 1, 2\}$.

For $1 \le p \le \mathcal{P}$, define $i[p]$ to be the pth element of the tuple i. The restrictions on the indices i, j, and k for $\alpha_{i,j,k}$ are:

- $\sum_p i[p] = I$; $\sum_p j[p] = J$; $\sum_p k[p] = K$;
- for each p, $i[p] + j[p] + k[p] = 2$; and
- for each $p < \mathcal{P}$, $(i[p], j[p], k[p]) \le (i[p+1], j[p+1], k[p+1])$, where $\le$ is a lexicographic comparison of the tuples.

For example, one valid α for V_{123} ($\mathcal{P} = 3$) is $\alpha_{(001),(020),(201)}$.

The equations for the large variables X_ℓ, Y_ℓ, Z_ℓ use an extended definition of the perm function, $\mathrm{perm}_\mathcal{P}(i, j, k)$, which is the number of distinct triples of tuples (i', j', k') one can obtain from all possible permutations π of $(1, 2, \ldots, \mathcal{P})$ by permuting i, j, and k each by π: $i'[p] = i[\pi(p)], j'[p] = j[\pi(p)], k'[p] = k[\pi(p)]$. We use the following identity to avoid enumerating all possible π:

$$\mathrm{perm}_\mathcal{P}(i, j, k) = \mathrm{perm}(i) \cdot \mathrm{perm}(j[p] : i[p] = 0) \cdot \mathrm{perm}(j[p] : i[p] = 1)$$

The large variables are

$$X_\ell = \frac{1}{\operatorname{perm}(\ell)} \sum_{i,j,k} c(i,\ell) \operatorname{perm}_{\mathcal{P}}(i,j,k) \alpha_{i,j,k}$$
$$Y_\ell = \frac{1}{\operatorname{perm}(\ell)} \sum_{i,j,k} c(j,\ell) \operatorname{perm}_{\mathcal{P}}(i,j,k) \alpha_{i,j,k}$$
$$Z_\ell = \frac{1}{\operatorname{perm}(\ell)} \sum_{i,j,k} c(k,\ell) \operatorname{perm}_{\mathcal{P}}(i,j,k) \alpha_{i,j,k}$$

with $c(i,\ell) = 1$ if i is a permutation of ℓ and 0 otherwise. This linear system may be overconstrained, so before solving it, we remove two of the equations to make sure its rank is equal to the number of equations. (In our implementation, we choose the lexicographically greatest X_ℓ and lexicographically greatest Y_ℓ; Williams' paper removes the lexicographically least X and Y. The choice is arbitrary.)

Solving this system as above gives the derivatives $\frac{d\alpha_{ijk}}{dX_\ell}$, $\frac{d\alpha_{ijk}}{dY_\ell}$, and $\frac{d\alpha_{ijk}}{dZ_\ell}$. In the general powers algorithm, each base W_{ijk} is a product of $\mathcal{P}$ values: $W_{ijk} = \prod_{p=1}^{\mathcal{P}} V_{i[p],j[p],k[p]}$. Note that although this appears to be a simple generalization of the product of two values that occurs in the even powers algorithm, here $i[p]$, $j[p]$, $k[p]$ are restricted to $\{0,1,2\}$, so the general powers algorithm is not fully recursive, instead giving values for arbitrary powers only in terms of the values V_{002} and V_{011}.

From the derivatives and bases, we construct

$$\xi_\ell = \operatorname{perm}(\ell) \prod_{i,j,k} W_{ijk}^{3\frac{\operatorname{perm}_{\mathcal{P}}(i,j,k)}{\operatorname{perm}(\ell)}\frac{d\alpha_{ijk}}{dX_\ell}}$$
$$\psi_\ell = \operatorname{perm}(\ell) \prod_{i,j,k} W_{ijk}^{3\frac{\operatorname{perm}_{\mathcal{P}}(i,j,k)}{\operatorname{perm}(\ell)}\frac{d\alpha_{ijk}}{dY_\ell}}$$
$$\zeta_\ell = \operatorname{perm}(\ell) \prod_{i,j,k} W_{ijk}^{3\frac{\operatorname{perm}_{\mathcal{P}}(i,j,k)}{\operatorname{perm}(\ell)}\frac{d\alpha_{ijk}}{dZ_\ell}}$$

and if Δ is nonempty,

$$\delta_{ijk} = \prod_{i',j',k'} W_{i'j'k'}^{\operatorname{perm}_{\mathcal{P}}(i',j',k')\frac{d\alpha_{i'j'k'}}{d\alpha_{ijk}}}.$$

In determining L_Δ, the setting of the large variables is also modified with $\operatorname{perm}(\ell)$ coefficients:

$$X_\ell = \frac{\xi_\ell}{\operatorname{perm}(\ell)\sum_{\ell'} \xi_{\ell'}} \qquad Y_\ell = \frac{\psi_\ell}{\operatorname{perm}(\ell)\sum_{\ell'} \psi_{\ell'}} \qquad Z_\ell = \frac{\zeta_\ell}{\operatorname{perm}(\ell)\sum_{\ell'} \zeta_{\ell'}}$$

We then use an LP solver to maximize $\log(L_\Delta)$ and compute V_{IJK} as in (2.2), identically to the even powers algorithm.

2.3.3 "Zero" values

Our implementation includes another lemma from Williams' paper ("Claim 7") which applies to values V_{0JK} from any power. These have a simpler lower bound that is a polynomial in q, raised to the τ power:

$$V_{0JK} \geq \left[\sum_{\substack{b \leq J \\ b \equiv J \pmod 2}} \binom{\mathcal{P}}{b, \frac{J-b}{2}, \frac{K-b}{2}} q^b \right]^\tau .$$

Chapter 3

Parameters

While the nonlinear program itself represents a search over a family of algorithms, defined by the a_{IJK} variables that define the search space of the nonlinear constraint solver, there are five configurable parameters that can be changed to influence the construction of the program itself. A feasible solution to the program generated for any setting of these parameters gives an upper bound on ω, so it is necessary to experiment with the setting of these parameters to ensure that one has explored the entire space of CW-like constructions.

3.1 Power

As described in chapter 2, Williams' algorithm is applied by starting with an analysis of a base case of a particular matrix multiplication construction and recursively generalizing the analysis to higher tensor powers $\mathcal{P}$. Varying this parameter has already been a productive source of improvements in ω: successive generalizations to higher powers of two of the Coppersmith-Winograd [CW87] construction by Stothers [Sto10] and Williams [Wil12] have each resulted in a better bound on ω.

We have applied our automated analysis to powers 2, 3, 4, 5, 7, and 8 of the CW construction, although we were unable to find a feasible solution to the $\mathcal{P} = 7$ program for any $\omega < 2.7$.

3.2 q

The CW family of constructions, starting with [Str86], are assembled from a sum of some number q of tensor products; Coppersmith and Winograd found $\omega < 2.376$ for $q = 6$, but Williams achieved $\omega < 2.3727$ using an analysis with $q = 5$. We allow q to vary in our analysis between 4 and 7. Of these, $q = 5$ appears to consistently give the best results across most powers (see chapter 5.1).

3.3 Zero variables

It is possible to reduce the dimensionality of the search space for the nonlinear solver by requiring that some number of the program parameters be exactly zero. This necessarily prevents the solver from finding some optimal solutions, but if the variables to be set to zero are chosen carefully, this restriction does not make solving the program much more difficult. If a feasible solution exists, reducing the number of variables can increase the probability of finding it and makes the solver converge faster (see section 5.2).

3.4 Free variables

Constructing the program involves several steps of solving underconstrained linear systems by choosing a subset of the variables to treat as constants. There are two points in Williams' algorithm where this strategy is employed: constructing the linear programming piece of the values computation (section 2.3), and constructing the derivatives constraint of the final program (section 2.2). Which variables are treated as constants can affect the bounds obtained on the various tensor values V_{IJK}.

Our implementation includes configurations of free variables used by Williams in [Wil12] and the option to specify the free variables used for each value and for the final program; although our search automation does not currently look for improvements gained in this way, this could be a subject of future work. Since higher values loosen the rank constraint (due to the fact that V_{IJK} is nondecreasing in τ), to minimize ω one would need to search for the LP free variable setting that maximizes each value. An exhaustive search would run in exponential time and may be infeasible already for some of the 8th power values, but if a pattern could be uncovered in the optimal free variable choices, this exhaustive search may be avoidable.

3.5 Substitution

Although the nonlinear program includes both equalities and inequalities, the equalities have simple forms that allow them to be eliminated by substitution in closed form. We implemented a search option that allows such substitution; however, we found that NLopt, the nonlinear solver we employed in searching for feasible solutions to the program, was unable to solve the programs constructed this way in a reasonable amount of time. One likely cause of this is that the closed-form substitutions introduce divisions in the constraints, which lead to discontinuities and precision problems in the solver.

Chapter 4

Implementation

Because the "constants" δ_{ijk} can contain ratios of lower-power values, which depend nonlinearly on τ, constructing symbolic solutions for the values of higher powers, and therefore constructing the complete nonlinear program such that τ can be minimized, is not tractable. Instead, our search proceeds as a series of attempts to find a feasible solution for specific values of τ (and q). With τ and q fixed, the values computed are numeric constants that do not have to be recomputed for every evaluation in the nonlinear solver.

4.1 Parallelization

Since each attempt is independent and CPU-bound, it makes sense to run searches in parallel, up to one search per available core. Our search implementation uses the Python `multiprocessing` library to take advantage of multiple cores, employing a task queue abstraction to schedule searches optimally. The structure of the search parallelization is (Python pseudocode):

```
def search():
    create inter-process queues input_queue, output_queue
    fill input_queue with num_cores random parameter settings
    start num_cores worker processes running worker
    while jobs completed < goal:
        block until output_queue is not empty
        extract a result from output_queue and record the result
        add another random parameter setting to input_queue
    kill worker processes
```

```
def worker(input_queue, output_queue):
    while True:
        get a parameter setting from the input queue
        attempt to solve the program with the parameters
        place the result (which includes success/failure)
            in output_queue
```

4.2 Improving performance

Solving the nonlinear program dominates the running time of the search. Two small optimizations made dramatic differences in the efficiency of the solver.

The first optimization was to use the SymPy `lambdify` function to preprocess the symbolic expressions manipulated by the procedures that constructed the nonlinear program into Python functions that compute the expressions directly. The original SymPy expressions were runtime data structures that required interpreting; `lambdify` transforms these into Python bytecode that runs just as fast as a custom function written for each expression.

The second optimization was to replace the equality in Williams' rank constraint with an inequality. Williams' STOC '12 paper had

$$r^{\mathcal{P}} = \frac{\prod_{I,J,K} V_{IJK}(\tau)^{\mathrm{perm}(I,J,K)a_{IJK}}}{\prod_{\ell} {A_\ell}^{A_\ell}}.$$

However, suppose we have a setting of a_{IJK} such that

$$r^{\mathcal{P}} \leq \frac{\prod_{I,J,K} V_{IJK}(\tau)^{\mathrm{perm}(I,J,K)a_{IJK}}}{\prod_{\ell} {A_\ell}^{A_\ell}}.$$

Since each V_{IJK} is a nondecreasing, continuous function of τ and all a_{IJK} are positive, there exists $\tau' \leq \tau$ for which the strict equality holds, so our τ still gives an upper bound on $\omega/3$. It is therefore acceptable to solve the program using the inequality, which is convenient because satisfying an inequality typically requires far fewer iterations than approximating an equality to a high degree of precision.

The Python program is still significantly slower than the C++ code written by Williams to solve the 8th power program; much of this slowness is likely due to the overhead of calls into and out of the NLopt Python wrappers. Possible future work in improving performance would involve the generation of C or C++ code, which would eliminate this overhead.

Chapter 5

Results

The best exponent bound we obtained solely with our implementation was

$$\omega \leq 2.372771003742,$$

with $\mathcal{P} = 4$, $q = 5$. We were also able to use parts of our implementation interactively to confirm a solution found by Williams with her Maple and C++ code, which gives

$$\omega \leq 2.3727104061$$

for $\mathcal{P} = 8$, $q = 5$. The settings for the variables a_{IJK} that produced these bounds are included in Table 5.2 on page 18.

5.1 Trends

Figure 5.1 plots the best matrix multiplication exponents obtained as a function of $\mathcal{P}$ and q. Powers 4 and 8 gave the best results of the data we collected; the results of the odd-powers attempts were disappointing, giving exponents far higher than the even-power searches. (The fifth power is left out of Figure 5.1 because including it obscures differences between the other powers; Figure 5.1 shows the fifth power in comparison. We were unable to find a feasible solution for the seventh power.) The success of the even powers algorithm suggests that the 16th power could be a fruitful next step in the search.

For all powers except the fifth, $q = 5$ produced the best exponents. The best second- and third-power exponents for $q = 5$ do not differ significantly from $q = 6$, but for the fourth and eighth powers, the difference is notable.

The number of zero variables does impact the exponents found slightly, although it primarily

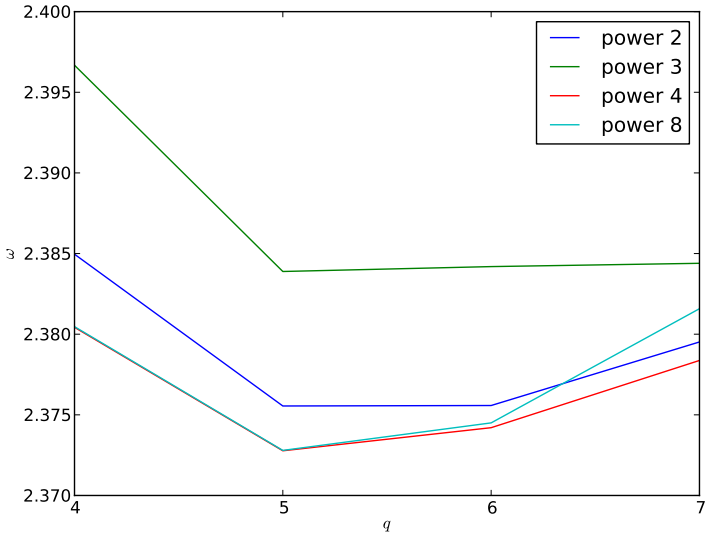

Figure 5.1: Best exponents by power and q

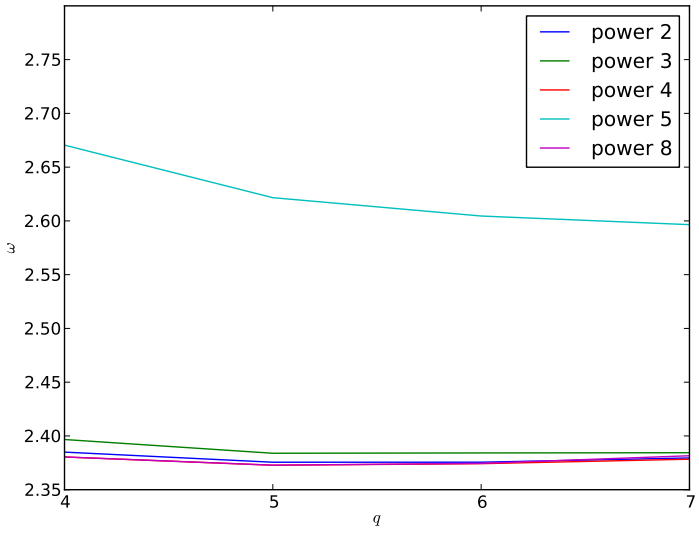

Figure 5.2: Best exponents including 5th power

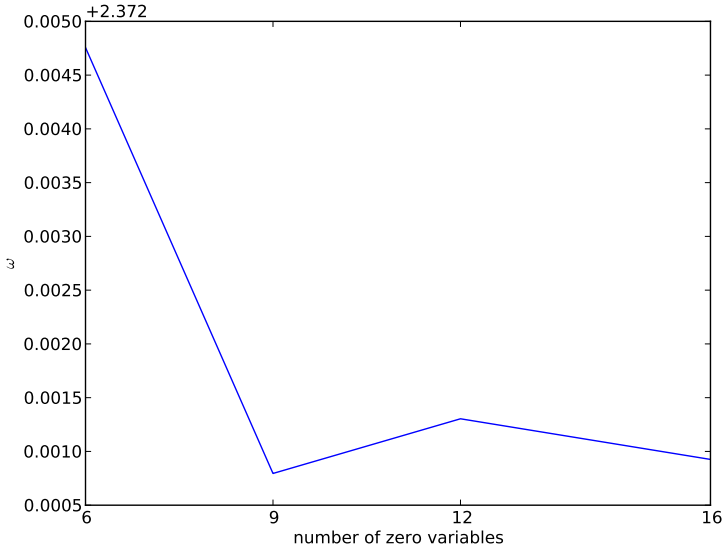

Figure 5.3: Best exponents for the 8th power, varying the number of zero variables

helps, achieving better exponents with higher numbers of variables set to zero. This is likely due to the decrease in program complexity, requiring less searching in the global ISRES algorithm. Figure 5.1 shows the best exponents found for each number of variables set to zero. All four exponents shown in this figure were obtained with $q = 5$.

5.2 Performance

Figure 5.2 shows the number of calls to the constraint functions required to achieve convergence (or determine that the program is not feasible) for the lower powers. Blue dots in the figure are successes, while red are failures; the red line is the kN^2 cap. While failed attempts are highly unpredictable in the amount of time they take to admit failure, successes occur reliably below a certain number of NLopt calls for each power. We take advantage of this by automatically aborting the optimization upon seeing that it has taken an unreasonable number of calls. The heuristic we use is to stop after kN^2 calls, where N is the number of variables in the program (after removing the zero variables), and k is a constant. We found that $k = 20000$ yields useful but conservative bounds.

Limiting the number of variables of the program decreases the time required for convergence dramatically. Figure 5.2 shows three different configurations of zero variables for the 8th power; the difference between using 6 zero variables and 16 is a factor-of-3 decrease in the number of iterations until convergence for successes.

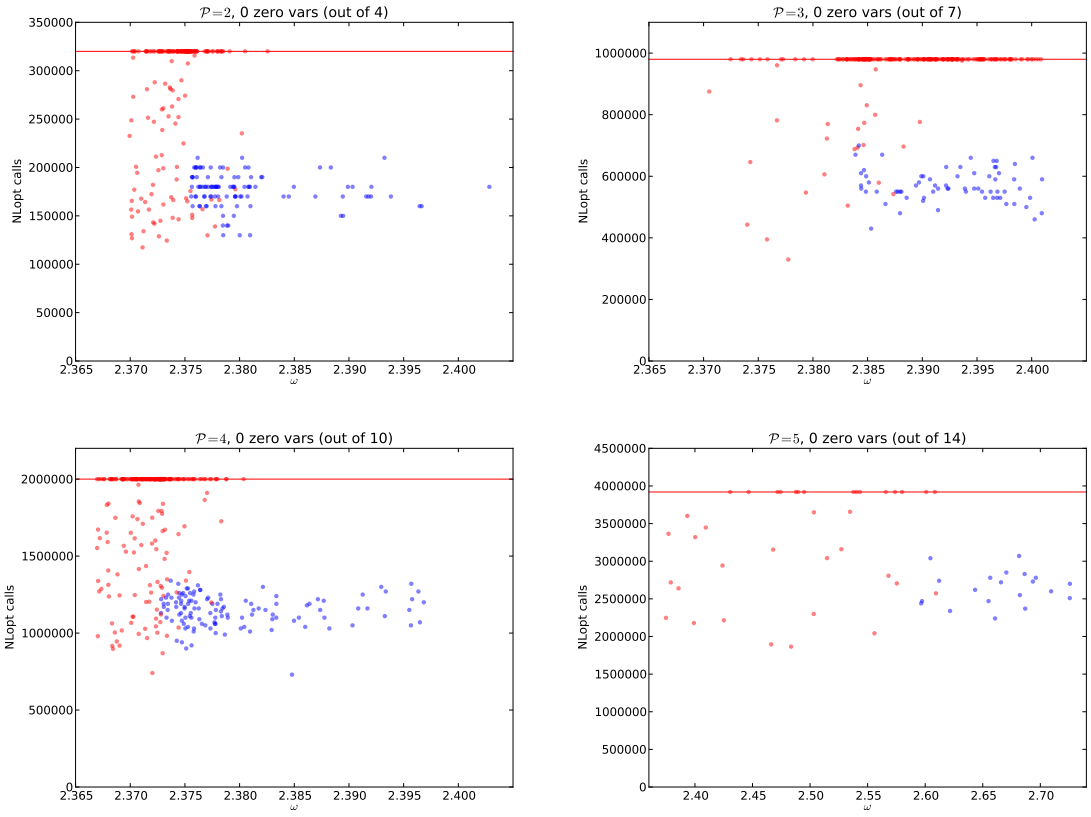

Figure 5.4: Performance of the lower powers, as a function of ω

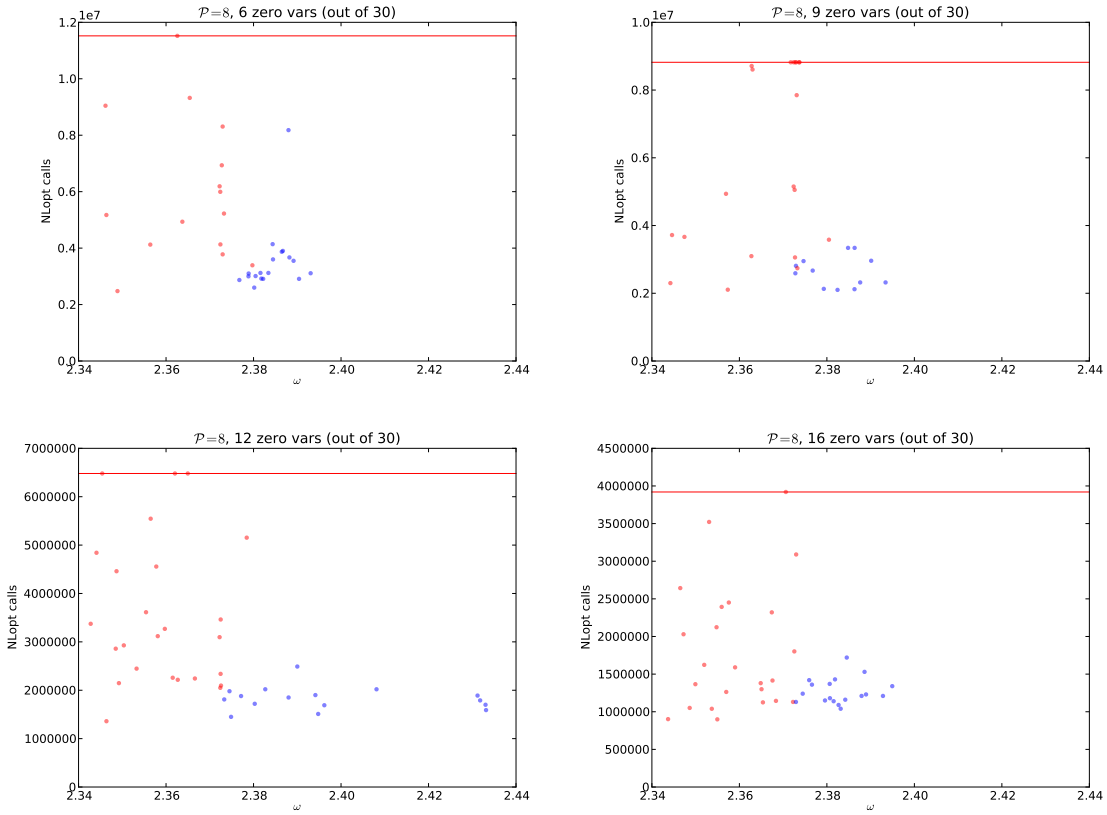

Figure 5.5: Performance of 8th power, for each setting of zero variables

Our result: $\mathcal{P} = 4$, $\omega \leq 2.372771003742$	
q	5
τ	0.790923667914
$a_{0,0,8}$	$3.959562956941811 \times 10^{-8}$
$a_{0,1,7}$	$6.510583719407185 \times 10^{-5}$
$a_{0,2,6}$	0.0006964997378042281
$a_{0,3,5}$	0.012014754526837784
$a_{0,4,4}$	0.0001025196963378948
$a_{1,1,6}$	0.0006309742840397419
$a_{1,2,5}$	0.0021449017957688166
$a_{1,3,4}$	0.05540636001102574
$a_{2,2,4}$	0.08675778797387862
$a_{2,3,3}$	0.1051867695535445

Williams' solution: $\mathcal{P} = 8$, $\omega \leq 2.3727104061$	
q	5
τ	0.7909034687
$a_{0,0,16}$	0
$a_{0,1,15}$	0
$a_{0,2,14}$	0
$a_{0,3,13}$	0
$a_{0,4,12}$	0
$a_{0,5,11}$	$5.607656585969684 \times 10^{-5}$
$a_{0,6,10}$	0.0003442516415535713
$a_{0,7,9}$	$1.0000000000697815 \times 10^{-11}$
$a_{0,8,8}$	$1.000000000151792 \times 10^{-11}$
$a_{1,1,14}$	0
$a_{1,2,13}$	0
$a_{1,3,12}$	0
$a_{1,4,11}$	$1.0000000000650404 \times 10^{-11}$
$a_{1,5,10}$	0.00041278743729968476
$a_{1,6,9}$	0.0006669991694253401
$a_{1,7,8}$	0.0014957209858593493
$a_{2,2,12}$	0
$a_{2,3,11}$	$1.0000000000434667 \times 10^{-11}$
$a_{2,4,10}$	$7.948735480255828 \times 10^{-12}$
$a_{2,5,9}$	0.0021066945258983614
$a_{2,6,8}$	0.006349235640076338
$a_{2,7,7}$	0.009151018807902839
$a_{3,3,10}$	$9.999999987632591 \times 10^{-12}$
$a_{3,4,9}$	0.0033285911678671305
$a_{3,5,8}$	0.012854594323804942
$a_{3,6,7}$	0.024900092793920667
$a_{4,4,8}$	0.016144154713523113
$a_{4,5,7}$	0.04007153148051885
$a_{4,6,6}$	0.053855557825297065
$a_{5,5,6}$	0.06900945039321159

Table 5.1: Variable settings for confirmed bounds

Chapter 6

Source code

Our Python source is available at `http://stanford.edu/~wmonroe4/matrixmult/`, downloadable as a zipped archive or through a `git` repository.

6.1 Dependencies

A Python interpreter (`http://python.org/`) is needed to run the search program. Python 2.7 is recommended to ensure that all the necessary libraries are supported. Our implementation uses three third-party libraries, all of which are free and open-source.

SymPy (`http://sympy.org/`) is an algebra library for Python that is used to automate symbolic calculations. It depends on **NumPy** (`http://numpy.org/`) for array types and numerical algorithms.

PuLP (`http://code.google.com/p/pulp-or/`) is a linear programming modeler that is used to represent the LP step of the values computation and submit it to an LP solver. The LP solver we use is **GLPK** (`http://www.gnu.org/software/glpk/`).

NLOpt (`http://ab-initio.mit.edu/nlopt`) [Joh] is the library we use to solve the final nonlinear program. We employ primarily the ISRES [RY05] global search algorithm in our implementation; the interactive exploration feature also provides a hook for the COBYLA [Pow94] algorithm, for finding local solutions.

Instructions for obtaining and installing these libraries are included on the site listed at the top of the page.

6.2 Getting started

The code is broadly divided into the final program construction, the values computations, utility classes and subroutines, and unit tests.

The creation of the final program is accomplished with `get_final_program` in `analysis.py`. Values are computed for the even powers algorithm and the general powers algorithm with the functions named `value` in `even.py` and `general.py`, respectively.

Included in the code are an extensive set of unit tests comparing the output of various pieces of the code with all expressions published in Williams' STOC '12 paper and included in personal correspondence. These are placed in `analysis_test.py` (for the final program), `even_test.py` (for the even powers value computation), and `general_test.py` (for the general powers value computation).

To start the parallel search strategy in its default configuration (looking for solutions near the best found so far, for powers 2, 3, 4, and 8, $4 \leq q \leq 7$), run

```
matrixmult/src$ python search.py
```

The output is summarized in the console and logged in detail to `exponents.txt`.

The Python modules can also be used to interactively search for better exponents and verify pieces of the algorithm. The module `shortcuts` is intended to make interactive exploration less tedious; figure 6.2 gives a list of the useful functions and classes provided in this module and a handful of other relevant definitions. As an example, the following lines verify that the solution `MAPLE_P8`, found by Williams using Maple, satisfies the rank constraint:

```
>>> from shortcuts import *
>>> gfp(Search(power=8, q=5, tau=MAPLE_P8[tau],
...            free_vars=STOC12_P8_FREE,
...            lp_free_vars=STOC12_P8_LP_FREE,
...            zero_vars=STOC12_P8_ZERO)).subs(MAPLE_P8).positive
set([4.74077080125921e-10])
```

i.e.: get the final program for $\mathcal{P} = 8$, $q = 5$ using the STOC '12 free variables and zero variables, plug in the Maple solution, and print the status of all inequality constraints (a set of numbers that should be positive). The output is $4 \times 10^{-10} > 0$, so the rank constraint is satisfied.

Functions

Full name	Shortcut
`analysis.get_final_program(search)` Construct the final program from the parameters given by the `Search` object `search`. Return it as a `Program` object.	`gfp`
`solutions.all_output()` Return a list of all `Solution` objects stored in `exponents.txt`.	`all_output`
`solutions.short_vars(power)` Return a `dict` mapping long variable symbols `a_i_j_k` to short variable symbols $a, b, \ldots$	`short_vars`
`solutions.long_vars(power)` Return a `dict` mapping short variable symbols to long variable symbols.	`long_vars`
`shortcuts.a(i, j, k)` Return a SymPy `Symbol` suitable for use as an a variable in the final program or an α in value computations.	`a`
`common.value_triples(power)` Return a list of all value triples (I, J, K) for the given power.	`vts`
`solvers.solve_isres(program, ...)` Global solver: look for a `Solution` to `program`.	
`solvers.solve_cobyla(program, starting_point, ...)` Local solver: look for a `Solution` to `program` starting near `starting_point`.	`solve_cobyla`

`even.value(triple, q, tau, ...)`
`general.value(triple, q, tau, ...)`
Compute the V_{IJK} from `q` and `tau`. Lower-power values can be computed symbolically by passing the `Symbols` `q` and `tau`.	

Constants

Full name	Shortcut
`symbols.q`	`q`
`symbols.tau` The SymPy `Symbols` for q and τ.	`tau`
`solutions.STOC12_P2, STOC12_P4, STOC12_P8`	`STOC12_P2, ...`
`solutions.MAPLE_P4, MAPLE_P8` Solutions by Williams for several different powers, in the form of `dicts` mapping `a(i, j, k)` symbols and `tau` to numbers.	`MAPLE_P4, MAPLE_P8`
`solutions.STOC12_P4_FREE`	`STOC12_P4_FREE`
`solutions.STOC12_P4_LP_FREE`	`STOC12_P4_LP_FREE`
`solutions.STOC12_P8_FREE`	`STOC12_P8_FREE`
`solutions.STOC12_P8_LP_FREE` Free variable settings for Williams' STOC '12 results (the same settings are used for the `MAPLE` solutions).	`STOC12_P8_LP_FREE`
`solutions.STOC12_P8_ZERO` Zero variable settings for Williams' STOC '12 power 8 result.	`STOC12_P8_ZERO`

Table 6.1: Useful definitions for interactive searching

Bibliography

[CW87] Don Coppersmith and Shmuel Winograd. "Matrix multiplication via arithmetic progressions". In: *Journal of Symbolic Computation* (1987).

[Joh] Stephen G. Johnson. *The NLopt nonlinear-optimization package*. URL: `http://ab-initio.mit.edu/nlopt`.

[Pow94] M.J.D. Powell. "A direct search optimization method that models the objective and constraint functions by linear interpolation". In: *Advances in optimization and numerical analysis*. Ed. by S. Gomez and J.-P. Hennart. Kluwer Academic: Dordrecht, 1994, pp. 51–67.

[RY05] Thomas Philip Runarsson and Xin Yao. "Search biases in constrained evolutionary optimization". In: *IEEE Trans. on Systems, Man, and Cybernetics Part C: Applications and Reviews*. Vol. 35. 2. 2005, pp. 233–243.

[Sch81] Arnold Schönhage. "Partial and total matrix multiplication". In: *SIAM Journal on Computing* 10.3 (1981), pp. 434–455.

[Sto10] Andrew Stothers. "On the complexity of matrix multiplication". PhD thesis. University of Edinburgh, 2010.

[Str69] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13 (4 1969), pp. 354–356.

[Str86] Volker Strassen. "The asymptotic spectrum of tensors and the exponent of matrix multiplication". In: *Foundations of Computer Science* (1986).

[Wil12] Virginia Vassilevska Williams. "Multiplying matrices faster than Coppersmith-Winograd". In: *Symposium on the Theory of Computing*. 2012.