Matrix Multiplication: Verifying Strong Uniquely Solvable Puzzles

Matthew Anderson(✉), Zongliang Ji, and Anthony Yang Xu

Department of Computer Science, Union College,
Schenectady, NY, USA
{andersm2,jiz,xua}@union.edu

Abstract. Cohn and Umans proposed a framework for developing fast matrix multiplication algorithms based on the embedding computation in certain groups algebras [9]. In subsequent work with Kleinberg and Szegedy, they connected this to the search for combinatorial objects called strong uniquely solvable puzzles (strong USPs) [8]. We begin a systematic computer-aided search for these objects. We develop and implement algorithms based on reductions to SAT and IP to verify that puzzles are strong USPs and to search for large strong USPs. We produce tight bounds on the maximum size of a strong USP for width $k < 6$, and construct puzzles of small width that are larger than previous work. Although our work only deals with puzzles of small-constant width and does not produce a new, faster matrix multiplication algorithm, we provide evidence that there exist families of strong USPs that imply matrix multiplication algorithms that are more efficient than those currently known.

Keywords: Matrix multiplication · Strong uniquely solvable puzzle · Arithmetic complexity · Integer programming · Satisfiability · Reduction · Application

1 Introduction

An optimal algorithm for matrix multiplication remains elusive despite substantial effort. We focus on the square variant of the matrix multiplication problem, i.e., given two n-by-n matrices A and B over a field $\mathcal{F}$, the goal is to compute the matrix product $C = A \times B$. The outstanding open question is: How many field operations are required to compute C? The long thought-optimal naïve algorithm based on the definition of matrix product is $O(n^3)$ time. The groundbreaking work of Strassen showed that it can be done in time $O(n^{2.808})$ [24] using a divide-and-conquer approach. A long sequence of work concluding with Coppersmith and Winograd's algorithm (CW) reduced the running time to $O(n^{2.376})$ [10,21,22,25]. Recent computer-aided refinements of CW by others reduced the exponent to $\omega \leq 2.3728639$ [13,18,26].

L. Pulina and M. Seidl (Eds.): SAT 2020, LNCS 12178, pp. 464–480, 2020.
https://doi.org/10.1007/978-3-030-51825-7_32

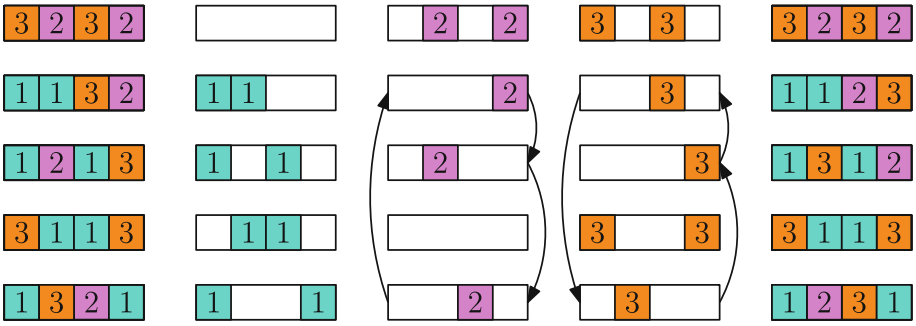

Fig. 1. The leftmost diagram is a width-4 size-5 puzzle P. The middle three diagrams are the three sets of subrows of P. The rightmost diagram is the puzzle P' resulting from reordering the subrows of P as indicated by the arrows and then recombining them. Since P can be rearranged as $P' \neq P$ without overlap, P is not uniquely solvable.

Approach. Cohn and Umans [9] introduced a framework for developing faster algorithms for matrix multiplication by reducing this to a search for groups with subsets that satisfy an algebraic property called the *triple-product property* that allows matrix multiplication to be embedded in the group algebra. Their approach takes inspiration from the $O(n \log n)$ algorithm for multiplying degree-n univariate polynomials by embedding in the group algebra of the fast Fourier transform, c.f., e.g., [11, Chapter 30]. Subsequent work [8] elaborated on this idea and developed the notion of combinatorial objects called *strong uniquely solvable puzzles* (strong USPs). These objects imply a group algebra embedding for matrix multiplication, and hence give a matrix multiplication algorithm as well.

A *width-k* puzzle P is a subset of $\{0, 1, 2\}^k$, and the cardinality of P is the puzzle's *size*. Each element of P is called a *row* of P, and each row consists of three *subrows* that are elements of $\{0, *\}^k$, $\{1, *\}^k$, $\{2, *\}^k$ respectively. Informally, a puzzle P is a *uniquely solvable puzzle* (USP) if there is no way to permute the subrows of P to form a distinct puzzle P' without cells with numbers overlapping. Figure 1 demonstrates a puzzle that is not a USP. A uniquely solvable puzzle is *strong* if a tighter condition for non-overlapping holds (see Definition 2). For a fixed width k, the larger the size of a strong USP, the faster matrix multiplication algorithm it gives [8]. In fact Cohn et al. show that there exist an infinite family of strong USPs that achieves $\omega < 2.48$.

We follow Cohn et al.'s program by: (i) developing **verification** algorithms to determine whether a puzzle is a strong USP, (ii) developing **search** algorithms to find large strong USPs, and (iii) implementing and running practical **implementations** of these algorithms. The most successful of the verification algorithms function by reducing the problem through 3D matching to SAT and IP which are then solved with existing tools. The algorithms we develop are not efficient—they run in worst-case exponential time in the natural parameters. However, the goal is to find a sufficiently large strong USP that would provide

a faster matrix multiplication algorithm, and the resulting algorithm's running time is independent of the running time of our algorithms. The inefficiency of our algorithms limit the search space that we can feasibly examine.

Results. Our experimental results give new bounds on the size of the largest strong USP for small-width puzzles. For small-constant width, $k \leq 12$, we beat the largest sizes of [8, Proposition 3.8]. Our lower bounds on maximum size are witnessed by strong USPs we found via search. For $k \leq 5$ we give tight upper bounds determined by exhaustively searching all puzzles up to isomorphism. Although our current experimental results do not beat [8] for unbounded k, they give evidence that there may exist families of strong USPs that give matrix multiplication algorithms that are more efficient than those currently known.

Related Work. There are a number of negative results known. Naïvely, the dimensions of the output matrix C implies that the problem requires at least $\Omega(n^2)$ time. Slightly better lower bounds are known in general and also for specialized models of computation, c.f., e.g., [16,23]. There are also lower bounds known for a variety of algorithmic approaches to matrix multiplication. Ambainis et al. showed that the laser method cannot alone achieve an algorithm with $\omega \leq 2.3078$ [4]. A recent breakthrough on arithmetic progressions in cap sets [12] combined with a conditional result on the Erdös-Szemeredi sunflower conjecture [3] imply that Cohn et al.'s strong USP approach cannot achieve $\omega = 2 + \epsilon$ for some $\epsilon > 0$ [7]. Subsequent work has generalized this barrier [1,2] to a larger class of algorithmic techniques. Despite this, we are unaware of a concrete lower bound on ϵ implied by these negative results. There remains a substantial gap in our understanding between what has been achieved by the positive refinements of LeGall, Williams, and Stothers, and the impossibility of showing $\omega = 2$ using the strong USP approach.

Organization. Section 2 begins with the formal definition of a strong USP. Sections 3 and 4, respectively, discuss our algorithms and heuristics for verifying that and searching for a puzzle that is a strong USP. Section 5 discusses our experimental results.

2 Preliminaries

For an integer k, we use $[k]$ to denote the set $\{0, 1, 2, \ldots, k-1\}$. For a set Q, Sym_Q denotes the symmetric group on the elements of Q, i.e., the group of permutations acting on Q. Cohn et al. introduced the idea of a *puzzle* [8].

Definition 1 (Puzzle). *For $s, k \in \mathcal{N}$, an (s,k)-*puzzle *is a subset $P \subseteq [3]^k$ with $|P| = s$. We call s the* size *of P, and k the* width *of P.*

We say that an (s, k)-puzzle has s rows and k columns. The columns of a puzzle are inherently ordered and indexed by $[k]$. The rows of a puzzle have no inherent ordering, however, it is often convenient to assume that they are ordered and indexed by the set of natural numbers $[s]$.

Cohn et al. establish a particular combinatorial property of puzzles that allows one to derive group algebras that matrix multiplication can be efficiently embedded into. Such puzzles are called *strong uniquely solvable puzzles*.

Definition 2 (Strong USP). *An (s, k)-puzzle P is* strong uniquely solvable *if for all $\pi_0, \pi_1, \pi_2 \in \mathrm{Sym}_P$: Either (i) $\pi_0 = \pi_1 = \pi_2$, or (ii) there exists $r \in P$ and $i \in [k]$ such that exactly two of the following hold: $(\pi_0(r))_i = 0$, $(\pi_1(r))_i = 1$, $(\pi_2(r))_i = 2$.*

Note that strong uniquely solvability is invariant to the (re)ordering of the rows or columns of a puzzle. We use this fact implicitly.

Cohn et al. show the following connection between the existence of strong USPs and upper bounds on the exponent of matrix multiplication ω.

Lemma 1 ([8, Corollary 3.6]). *Let $\epsilon > 0$, if there is a strong uniquely solvable (s, k)-puzzle, there is an algorithm for multiplying n-by-n matrices in time $O(n^{\omega+\epsilon})$ where*

$$\omega \leq \min_{m \geq 3, m \in \mathcal{N}} \frac{3 \log m}{\log(m-1)} - \frac{3 \log s!}{sk \log(m-1)}.$$

This result motivates the search for large strong USPs that would result in faster algorithms for matrix multiplication. In the same article, the authors also demonstrate the existence of an infinite family of strong uniquely solvable puzzles, for width k divisible by three, that achieves a non-trivial bound on ω.

Lemma 2 ([8, Proposition 3.8]). *There is an infinite family of strong uniquely solvable puzzles that achieves $\omega < 2.48$.*

3 Verifying Strong USPs

The core focus of this article is the problem of verifying strong USPs, i.e., given an (s, k)-puzzle P, output YES if P is a strong USP, and NO otherwise. In this section we discuss the design of algorithms to solve this computational problem as a function of the natural parameters s and k. Along the way we also discuss some aspects of our practical implementation that informed or constrained our designs. All the exact algorithms we develop in this section have exponential running time. However, asymptotic worst-case running time is not the metric we are truly interested in. Rather we are interested in the practical performance of our algorithms and their capability for locating new large strong USPs. The algorithm that we ultimately develop is a hybrid of a number of simpler algorithms and heuristics.

Algorithm 1: Brute Force

Input: An (s, k)-puzzle P.
Output: YES, if P is a strong USP and NO otherwise.

```
1: function VerifyBruteForce(P)
2:   for π1 ∈ Sym_P do
3:     for π2 ∈ Sym_P do
4:       if π1 ≠ 1 ∨ π2 ≠ 1 then
5:         found = false.
6:         for r ∈ P do
7:           for i ∈ [k] do
8:             if δ_{r_i,0} + δ_{(π1(r))_i,1} + δ_{(π2(r))_i,2} = 2 then found = true.
9:         if not found then return NO.
10:  return YES.
```

3.1 Brute Force

The obvious algorithm for verification comes directly from the definition of a strong USP. Informally, we consider all ways of permuting the ones and twos pieces relative to the zeroes pieces and check whether the non-overlapping condition of Definition 2 is met. A formal description of the algorithm is found in Algorithm 1.

The ones in Line 4 of Algorithm 1 denote the identity in Sym_P, and $\delta_{a,b}$ is the Kronecker delta function which is one if $a = b$ and zero otherwise. Observe that Algorithm 1 does not refer to the π_0 of Definition 2. This is because the strong USP property is invariant to permutations of the rows and so π_0 can be thought of as an arbitrary phase. Hence, we fix $\pi_0 = 1$ to simplify the algorithm. Seeing that $|\mathrm{Sym}_P| = s!$, we conclude that the algorithm runs in time $O((s!)^2 \cdot s \cdot k \cdot \mathrm{poly}(s))$ where the last factor accounts for the operations on permutations of s elements. The dominant term in the running time is the contribution from iterating over pairs of permutations. Finally, notice that if P is a strong USP, then the algorithm runs in time $\Theta((s!)^2 \cdot s \cdot k \cdot \mathrm{poly}(s))$, and that if P is not a strong USP the algorithm terminates early. The algorithm's poor performance made it unusable in our implementation, however, its simplicity and direct connection to the definition made its implementation a valuable sanity check against later more elaborate algorithms (and it served as effective onboarding to the undergraduate students collaborating on this project).

Although Algorithm 1 performs poorly, examining the structure of a seemingly trivial optimization leads to substantially more effective algorithms. Consider the following function on triples of rows $a, b, c \in P$: $f(a, b, c) = \vee_{i \in [k]}(\delta_{a_i,0} + \delta_{b_i,1} + \delta_{c_i,2} = 2)$. We can replace the innermost loop in Lines 7 & 8 of Algorithm 1 with the statement $found = found \vee f(r, \pi_1(r), \pi_2(r))$. Observe that f neither depends on P, r, nor the permutations, and that Algorithm 1 no longer depends directly on k. To slightly speed up Algorithm 1 we can precompute and cache f before the algorithm starts and then look up values as the algorithm runs. We precompute f specialized to the rows in the puzzle P, and call it f_P.

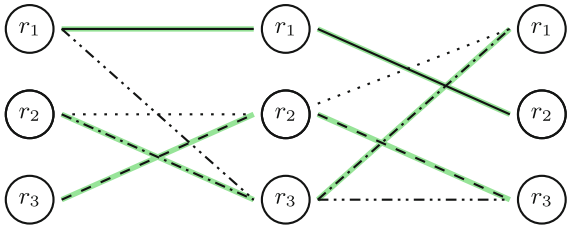

Fig. 2. An example hypergraph G with edges $E = \{(r_1, r_1, r_2), (r_1, r_3, r_3), (r_2, r_2, r_1), (r_2, r_3, r_1), (r_3, r_2, r_3)\}$. The highlighted edges are a non-trivial 3D matching $M = \{(r_1, r_1, r_2), (r_2, r_3, r_1), (r_3, r_2, r_3)\}$ of G.

3.2 Strong USP Verification to 3D Matching

It turns out to be more useful to work with f_P than with P. It is convenient to think of f_P as a function $f_P : P \times P \times P \to \{0, 1\}$ that is the complement of the characteristic function of the relations of a tripartite hypergraph $H_P = \langle P \sqcup P \sqcup P, \bar{f_P} \rangle$ where the vertex set is the disjoint union of three copies of P and f_P indicates the edges that are not present in H_P.

Let $H = \langle P \sqcup P \sqcup P, E \subseteq P^3 \rangle$ be a tripartite 3-hypergraph. We say H has a *3D matching* (3DM) iff there exists a subset $M \subseteq E$ with $|M| = |P|$ and for all distinct edges $e_1, e_2 \in M$, e_1 and e_2 are *vertex disjoint*, i.e., $e_1 \cap e_2 = \emptyset$. Determining whether a hypergraph has a 3D matching is a well-known NP-complete problem (c.f., e.g., [14]). We say that a 3D matching is *non-trivial* if it is not the set $\{(r, r, r) \mid r \in P\}$. Figure 2 demonstrates a 3-hypergraph with a non-trivial 3D matching.

The existence of non-trivial 3D matchings in H_P is directly tied to whether P is a strong USP.

Lemma 3. *A puzzle P is a strong USP iff H_P has no non-trivial 3D matching.*

Proof. We first argue the reverse. Suppose that H_p has a non-trivial 3D matching M. We show that P is not a strong USP by using M to construct $\pi_0, \pi_1, \pi_2 \in \mathrm{Sym}_P$ that witness this. Let π_0 be the identity permutation. For each $r \in P$, define $\pi_1(r) = q$ where $(r, q, *) \in M$. Note that q is well defined and unique because M is 3D matching and so has vertex disjoint edges. Similarly define $\pi_2(r) = q$ where $(r, *, q) \in M$. Observe that by construction

$$M = \{(\pi_0(r), \pi_1(r), \pi_2(r)) \mid r \in P\}.$$

Since M is a matching of H_P, $M \subseteq \bar{f_P}$. Because M is a non-trivial matching at least one edge in $(a, b, c) \in M$ has either $a \neq b$, $a \neq c$, or $b \neq c$. This implies, respectively, that as constructed $\pi_0 \neq \pi_1$, $\pi_0 \neq \pi_2$, or $\pi_1 \neq \pi_2$. In each case we have determined that π_0, π_1, and π_2 are not all identical. Thus we determined permutations such that for all $r \in P$, $f(\pi_0(r), \pi_1(r), \pi_2(r)) = 0$. This violates Condition (ii) of Definition 2, hence P is not a strong USP.

The forward direction is symmetric. Suppose that P is not a strong USP. We show that H_P has a 3D matching. For P not to be a strong USP there must exist $\pi_0, \pi_1, \pi_2 \in \mathrm{Sym}_P$ not all identical such that Condition (ii) of Definition 2 fails. Define $e(r) = (\pi_0(r), \pi_1(r), \pi_2(r))$ and $M = \{e(r) \mid r \in P\}$. Since Condition (ii) fails, we have that $f_P(e(r)) = false$ for all $r \in P$. This means that for all $r \in P$, $e(r) \in \bar{f}_P$ and hence $M \subseteq \bar{f}_P$. Since π_0 is a permutation, $|M| = |P|$. Observe that M is non-trivial because not all the permutations are identical and there must be some $r \in P$ with $e(r)$ having non-identical coordinates. Thus M is a non-trivial 3D matching. □

Note that although 3D matching is an NP-complete problem, Lemma 3 does not immediately imply that verification of strong USPs is coNP-complete because H_P is not an arbitrary hypergraph. As a consequence of Definition 2, verification is in coNP. It remains open whether verification is coNP-complete. Lemma 3 implies that to verify P is a strong USP it suffices to determine whether H_P has a 3D matching. In the subsequent sections we examine algorithms for the later problem. We can, in retrospect, view Algorithm 1 as an algorithm for solving 3D matching.

The realization that verification of strong USPs is a specialization of 3D matching leads to a dynamic programming algorithm for verification that runs in linear-exponential time $O(2^{2s}\mathrm{poly}(s) + \mathrm{poly}(s, k))$. Applying more advanced techniques like those of Björklund et al. can achieve a better asymptotic time of $O(2^s\mathrm{poly}(s) + \mathrm{poly}(s, k))$ [6]. For brevity, we defer the details of our algorithm to the long version of this article.

3.3 3D Matching to Satisfiability

By Lemma 3, one can determine whether a puzzle P is a strong USP by constructing the graph H_P and deciding whether it has a non-trivial 3D matching. Here we reduce our 3D matching problem to the satisfiability (SAT) problem on conjunctive normal form (CNF) formulas and then use a state-of-the-art SAT solver to resolve the reduced problem. To perform the reduction, we convert the graph H_P into a CNF formula Ψ_P, a depth-2 formula that is the AND of ORs of Boolean literals. We construct Ψ_P so that Ψ_P is satisfiable iff H_P has a non-trivial 3D matching.

Let $H_P = \langle V = P \sqcup P \sqcup P, E \subseteq P^3 \rangle$ be the 3D matching instance associated with the puzzle P. Our goal is to determine whether there is a non-trivial 3D matching $M \subseteq E$. A naïve reduction would be to have variables $M_{u,v,w}$ indicating inclusion of each edge $(u, v, w) \in P^3$ in the matching. This results in a formula Ψ_P with s^3 variables and size $\Theta(s^5)$ because including an edge $e \in P^3$ excludes the $\Theta(s^2)$ edges e' with $e \cap e' \neq \emptyset$. To decrease the size of Ψ_P we instead use sets of variables to indicate which vertices in the second and third part of V are matched with each vertex in the first part. In particular we have Boolean variables $M^1_{u,v}$ and $M^2_{u,w}$ for all $u, v, w \in P$, and these variable map to assignments in the naïve scheme in the following way: $M^1_{u,v} \wedge M^2_{u,w} \Leftrightarrow M_{u,v,w}$.

We now write our CNF formula for 3D matching. First, we have clauses that prevents non-edges from being in the matching:

$$\Psi_P^{\text{non-edge}} = \bigwedge_{(u,v,w)\in\overline{E}} (\neg M^1_{u,v} \vee \neg M^2_{u,w}). \tag{1}$$

Second, we add clauses require that every vertex in H_P is matched with some edge:

$$\begin{aligned}\Psi_P^{\geq 1} = &\left(\bigwedge_{u\in P} (\vee_{v\in P}\ M^1_{u,v}) \wedge (\vee_{w\in P}\ M^2_{u,w})\right) \\ &\wedge \left(\bigwedge_{v\in P} (\vee_{u\in P}\ M^1_{u,v})\right) \wedge \left(\bigwedge_{w\in P} (\vee_{u\in P}\ M^2_{u,w})\right).\end{aligned} \tag{2}$$

Third, we require that each vertex be matched with at most one edge and so have clauses that exclude matching edges that overlap on one or two coordinates.

$$\Psi_P^{\leq 1} = \bigwedge_{i\in\{1,2\}} \bigwedge_{(u,v),(u',v')\in P^2} (u = u' \vee v = v') \wedge (u, v \neq u', v') \Rightarrow \neg M^i_{u,v} \vee \neg M^i_{u',v'}. \tag{3}$$

Fourth, we exclude the trivial 3D matching by requiring that at least one of the diagonal edges not be used: $\Psi_P^{\text{non-trivial}} = \bigvee_{u\in P} \neg M^1_{u,u} \vee \neg M^2_{u,u}$. Finally, we AND these into the overall CNF formula: $\Psi_P = \Psi_P^{\text{non-edge}} \wedge \Psi_P^{\leq 1} \wedge \Psi_P^{\geq 1} \wedge \Psi_P^{\text{non-trivial}}$. The size of the CNF formula Ψ_P is $\Theta(s^3)$, has $2s^2$ variables, and is a factor of s^2 smaller than the naïve approach. Thus we reduce 3D matching to satisfiability by converting the instance H_P into the CNF formula Ψ_P.

To solve the reduced satisfiability instance we used the open-source solver MapleCOMPSPS from the 2016 International SAT Competition [5]. This solver is conflict driven and uses a learning rate branching heuristic to decide which variables are likely to lead to conflict and has had demonstrable success in practice [19]. We chose MapleCOMPSPS because it was state of the art at the time our project started. It is likely that more recently-developed solvers would achieve similar or better performance on our task.

3.4 3D Matching to Integer Programming

In parallel to the previous subsection, we use the connection between verification of strong USPs and 3D matching to reduce the former to integer programming, another well-known NP-complete problem (c.f., e.g., [17]). Again, let $H_P = \langle V, E\rangle$ be the 3D matching instance associated with P. We construct an integer program Q_P over $\{0, 1\}$ that is infeasible iff P is a strong USP. Here the reduction is simpler than the previous one because linear constraints naturally capture matching.

We use $M_{u,v,w}$ to denote a variable with values in $\{0, 1\}$ to indicate whether the edge $(u, v, w) \in P^3$ is present in the matching. To ensure that M is a subset of E we add the following edge constraints to Q_P: $\forall u, v, w \in P, \forall (u, v, w) \notin$

$E, M_{u,v,w} = 0$. We also require that each vertex in each of the three parts of the graph is incident to exactly one edge in M. This is captured by the following vertex constraints in Q_P: $\forall w \in P, \sum_{u,v \in P} M_{u,v,w} = \sum_{u,v \in P} M_{u,w,v} = \sum_{u,v \in P} M_{w,u,v} = 1$. Lastly, since we need that the 3D matching be non-trivial we add the constraint: $\sum_{u \in P} M_{u,u,u} < |P|$.

To check whether P is a strong USP we determine whether Q_P is not feasible, i.e., that no assignment to the variables M satisfy all constraints. In practice this computation is done using the commercial, closed-source, mixed-integer programming solver Gurobi [15]. We note that reduction from 3D matching to IP is polynomial time and that there are s^3 variables in Q_P, and that the total size of the constraints is $s^3 \cdot \Theta(1) + 3s \cdot \Theta(s^2) + 1 \cdot \Theta(s^3) = \Theta(s^3)$, similar to size of Ψ_P in the SAT reduction.

3.5 Heuristics

Although the exact algorithms presented in the previous sections make substantial improvements over the brute force approach, the resulting performance remains impractical. To resolve this, we also develop several fast verification heuristics that may produce the non-definitive answer MAYBE in place of YES or NO. Then, to verify a puzzle P we run this battery of fast heuristics and return early if any of the heuristics produce a definitive YES or NO. When all the heuristics result in MAYBE, we then run one of the slower exact algorithms that were previously discussed. The heuristics have different forms, but all rely on the structural properties of a strong USP. Here we discuss the two most effective heuristics, downward closure and greedy, and defer a deeper discussion of these and several less effective heuristics, including projection to 2D matching, to the full version of this article.

Downward Closed. The simplest heuristics we consider is based on the fact that strong USPs are downward closed.

Lemma 4. *If P is a strong USP, then so is every subpuzzle $P' \subseteq P$.*

Proof. Let P be a strong USP and $P' \subseteq P$. By Definition 2, for every $(\pi_1, \pi_2, \pi_3) \in \mathrm{Sym}_P^3$ not all identity, there exist $r \in P$ and $i \in [k]$ such that exactly two of the following hold: $(\pi_0(r))_i = 0$, $(\pi_1(r))_i = 1$, $(\pi_2(r))_i = 2$. Consider restricting the permutations to those that fix the elements of $P \backslash P'$. For these permutations it must be the case that $r \in P'$ because otherwise $r \in P \backslash P'$ and there is exactly one $j \in [3]$ for which $(\pi_j(r))_i = j$ holds. Thus we can drop the elements of $P \backslash P'$ and conclude that for every tuple of permutations in $\mathrm{Sym}_{P'}$ the conditions of Definition 2 hold for P', and hence that P' is a strong USP. □

This leads to a polynomial-time heuristic that can determine that a puzzle is not a strong USP. Informally, the algorithm takes an (s, k)-puzzle P and $s' \leq s$, and verifies that all subsets $P' \subseteq P$ with size $|P'| = s'$ are strong USPs. If any subset P' is not a strong USP, the heuristic returns NO, otherwise it

returns MAYBE. This algorithm runs in time $O(\binom{s}{s'} \cdot T(s', k))$ where $T(s', k)$ is the runtime for verifying an (s', k)-puzzle. In practice we did not apply this heuristic for s' larger than 3, so the effective running time was $O(s^3 \cdot T(3, k))$, which is polynomial in s and k using the verification algorithms from the previous subsections that eliminate dependence on k for polynomial cost. This heuristic can be made even more practical by caching the results for puzzles of size s', reducing the verification time per iteration to constant in exchange for $\Theta(\binom{3^k}{s'} \cdot T(s', k))$ time and $\Theta(\binom{3^k}{s'})$ space to precompute the values for all puzzles of size s'. From a practical point of view, running this heuristic is free for small constant $s' \leq 3$, as even the reductions in the exact verification algorithms have a similar or higher running time.

Greedy. This heuristic attempts to greedily solve the 3D matching instance H_P. The heuristic proceeds iteratively, determining the vertex of the first part of the 3D matching instance with the least edges and randomly selecting an edge of that vertex to put into the 3D matching. If the heuristic successfully constructs a 3D matching it returns NO indicating that the input puzzle P is not a strong USP. If the heuristic reaches a point were prior commitments have made the matching infeasible, the heuristic starts again from scratch. This process is repeated some number of times before it gives up and returns MAYBE. In our implementation we use s^3 attempts because it is similar to the running time of the reductions and it empirically reduced the number of instances requiring full verification in the domain of puzzles with $k = 6, 7, 8$ while not increasing the running time by too much.

3.6 Hybrid Algorithm

Our final verification algorithm (Algorithm 2) is a hybrid of several exact algorithms and heuristics. The size thresholds for which algorithm and heuristic to apply were determined experimentally for small k and were focused on the values were our strong USP search algorithms were tractable $k \leq 6$ (or nearly tractable $k \leq 8$). We decided to run both the reduction to SAT and IP in parallel because it was not clear which algorithm performed better. Since verification halts when either algorithm completes, the wasted effort is within a factor of two of what the better algorithm could have done alone. We also chose to do this because we experimentally observed that there were many instances that one of the algorithms struggled with that the other did not—this resulted in a hybrid algorithm that out performed the individual exact algorithms on average.

4 Searching for Strong USPs

In some ways the problem of constructing a large strong USP is similar to the problem of constructing a large set of linearly independent vectors. In both cases, the object to be constructed is a set, the order that elements are added does not

Algorithm 2: Hybrid Verification Algorithm

Input: An (s, k)-puzzle P.
Output: YES, if P is found to be strong USP, and NO otherwise.

```
if s ≤ 2 then return VerifyBruteForce(P).
if s ≤ 7 then return VerifyDynamicProgramming(P).
if s ≤ 10 then
    Return result if HeuristicDownwardClosed(P, 2) is not MAYBE.
    return VerifyDynamicProgramming(P).
Return result if HeuristicDownwardClosed(P, 3) is not MAYBE.
Return result if HeuristicGreedy(P) is not MAYBE.
Run VerifySAT(P) and VerifyIP(P) in parallel and return the first result.
```

matter, the underlying elements are sequences of numbers, and there is a notion of (in)dependence among sets of elements. There are well-known polynomial-time algorithms for determining whether a set of vectors are independent, e.g., Gaussian elimination, and we have a practical implementation for deciding whether a puzzle is a strong USP.

There is a straightforward greedy algorithm for constructing maximum-size sets of independent vectors: Start with an empty set S, and repeatedly add vectors to S that are linearly independent of S. After this process completes S is a largest set of linearly independent vectors. This problem admits such a greedy algorithm because the family of sets of linearly independent vectors form a matroid. The vector to be added each step can be computed efficiently by solving a linear system of equations for vectors in the null space of S.

Unfortunately this same approach does not work for generating maximum-size strong USPs. The set of strong USPs does not form a matroid, rather it is only an independence system, c.f., e.g., [20]. In particular, (i) the empty puzzle is a strong USP and (ii) the set of strong USP are downward closed by Lemma 4. The final property required to be a matroid, the augmentation property, requires that for every pair of strong USPs P_1, P_2 with $|P_1| \leq |P_2|$ there is a row of $r \in P_2 \backslash P_1$ such that $P_1 \cup \{r\}$ is also a strong USP. A simple counterexample with the strong USPs $P_1 = \{32\}$ and $P_2 = \{12, 23\}$ concludes that neither $P_1 \cup \{12\} = \{12, 32\}$ nor $P_1 \cup \{23\} = \{23, 32\}$ are strong USPs, and hence the augmentation property fails. One consequence is that naïve greedy algorithms will likely be ineffective for finding maximum-size strong USPs. Furthermore, we do not currently have an efficient algorithm that can take a strong USP P and determine a row r such that $P \cup \{r\}$ is a strong USP aside from slight pruning of the $\leq 3^k$ possible next rows r.

That said, we have had some success applying general purpose search techniques together with our practical verification algorithm to construct maximum-size strong USPs for small k. In particular, we implemented variants of depth-first search (DFS) and breadth-first search (BFS). We defer the details of this to the full version of this article.

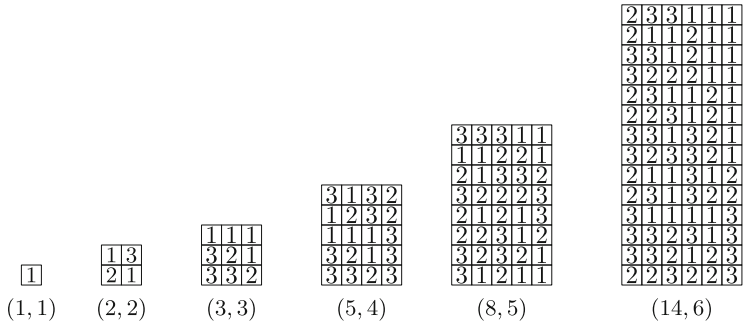

Fig. 3. Representative maximum-size strong USPs found for width $k = 1, 2, \ldots, 6$.

The actual running times of both of these algorithms are prohibitive even for $k > 5$, and the greater memory usage of BFS to store the entire search frontier is in the tens of terabytes even for $k = 6$. There are some silver linings, DFS can report intermediate results which are the maximally strong USPs that it has discovered so far. Both algorithms admit the possibility of eliminating puzzles from the search that are equivalent to puzzles that have already been searched, though it is easier to fit into the structure BFS as the puzzles are already being stored in a queue.

5 Experimental Results

Our experimental results come in three flavors for small-constant width k: (i) constructive lower bounds on the maximum size of width-k strong USPs witnessed by found puzzles, (ii) exhaustive upper bounds on the maximum size of width-k strong USPs, and (iii) experimental run times comparing the algorithms for verifying width-k strong USPs. BFS and DFS when able to run to completion search the entire space, up to puzzle isomorphism, and provide tight upper and lower bounds. When unable to run to completion they provide only results of form (i) and are not guaranteed to be tight.

5.1 New Bounds on the Size of Strong USPs

Figure 3 contains representative examples of maximum-size strong USPs we found for $k \leq 6$. Table 1 summarizes our main results in comparison with [8]. The lower bounds of [8] are from the constructions in their Propositions 3.1 and 3.8 that give families of strong USPs for k even or k divisible by three. The upper bounds of [8] follow from their Lemma 3.2 (and the fact that the capacity of strong USPs is bounded above by the capacity of USPs). The values of ω in this table are computed by plugging s and k into Lemma 1 and optimizing over m. For clarity we omit ω's that would be larger than previous lines.

We derive tight bounds for all $k \leq 5$ and constructively improve the known lower bounds for $4 \leq k \leq 12$. The strong uniquely solvable (14, 6)-puzzles we

Table 1. Comparison of bounds on maximum size of strong USPs with [8] for small k.

	[8]		This work	
k	Maximum s	ω	Maximum s	ω
1	$1 = s$	3.000	$1 = s$	3.000
2	$2 \leq s \leq 3$	2.875	$2 = s$	2.875
3	$3 \leq s \leq 6$	2.849	$3 = s$	2.849
4	$4 \leq s \leq 12$	2.850	$5 = s$	2.806
5	$4 \leq s \leq 24$		$8 = s$	2.777
6	$10 \leq s \leq 45$	2.792	$14 \leq s$	2.733
7	$10 \leq s \leq 86$		$21 \leq s$	2.722
8	$16 \leq s \leq 162$		$30 \leq s$	2.719
9	$36 \leq s \leq 307$	2.739	$42 \leq s$	2.718
10	$36 \leq s \leq 581$		$64 \leq s$	2.706
11	$36 \leq s \leq 1098$		$112 \leq s$	2.678
12	$136 \leq s \leq 2075$	2.696	$196 \leq s$	2.653

found represent the greatest improvement in ω versus the construction of [8]. Further, our puzzle for $k = 12$ is the result of taking the Cartesian product of two copies of a strong uniquely solvable (14, 6)-puzzle. Repeating this process with more copies of the puzzle gives a strong USP implying $\omega < 2.522$. Note that Proposition 3.8 of [8] gives an infinite family of strong USPs that achieves $\omega < 2.48$ in the limit.

Based on the processing time we spent on $k = 6$, we conjecture that $s = 14$ is tight for $k = 6$ and that our lower bounds for $k > 6$ are not. Our results suggests there is considerable room for improvement in the construction of strong USPs, and that it is likely that there exist large puzzles for $k = 7, 8, 9$ that would beat [8]'s construction and perhaps come close to the Coppersmith-Winograd refinements. It seems that new insights into the search problem are required to proceed for $k > 6$.

5.2 Algorithm Performance

We implemented our algorithms in C++ (source code to be made available on github) and ran them on a 2010 MacPro running Ubuntu 16.04 with dual Xeon E5620 2.40 Ghz processors and 16 GB of RAM. Figure 4 contains log plots that describe the performance of our algorithms on sets of 10000 random puzzles at each point on a sweep through parameter space for width $k = 5 \ldots 10$ and size $s = 1 \ldots 30$. We chose to test performance via random sampling because we do not have access to a large set of solved instances. This domain coincides with the frontier of our search space, and we tuned the parameters of the heuristics and algorithms in the hybrid algorithm to perform well in this domain. We did not deeply investigate performance characteristics outside of this domain.

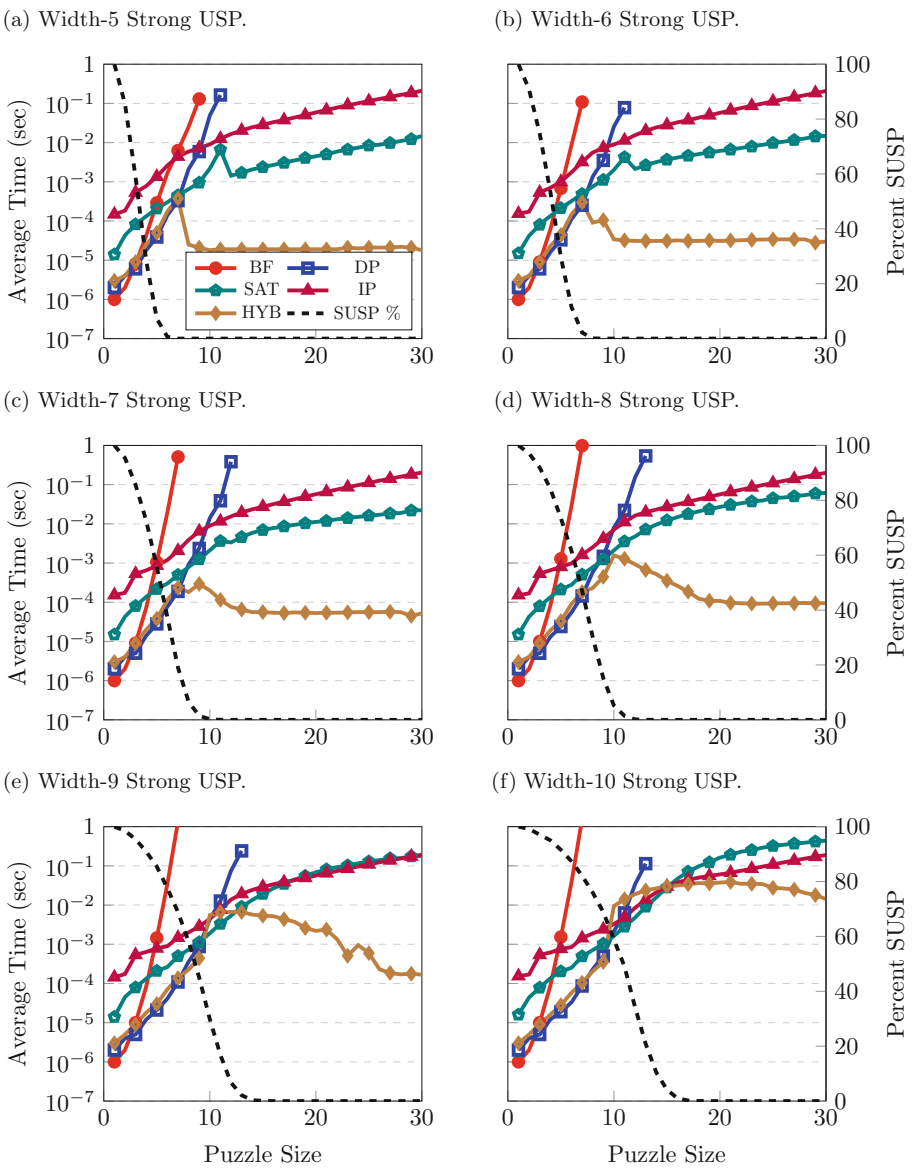

Fig. 4. Log plots of the average running times for verifying 10000 random puzzles of widths five to ten. Note that the legend in (a) applies to all six plots, and that the axes are named and labeled only on the edges of the page. Each plot describes the behavior of five algorithms brute force (BF), dynamic programming (DP), reduction to satisfiability (SAT), reduction to integer programming (IP), and the hybrid algorithm (HYB). The final dashed line indicates the percentage of strong USP found among the 10000 random puzzles.

The brute force and dynamic programming algorithms perform poorly except for very small size, $s \leq 8$, and their curves loosely match the $2^{\Omega(n)}$ time bounds we have. The plots for the two reduction-based algorithms (SAT and IP) behave similarly to each other. They are slower than brute force and dynamic programming for small values of s, and their behavior for large s is quite a bit faster. We speculate that the former is due to the cost of constructing the reduced instance and overhead of the third party tools. Further observe that the SAT reduction handily beats the IP reduction on large size for $k = 5$, but as k increases, the IP reduction becomes faster. We also note that across the six plots the IP reduction has effectively the same running time and is independent of k, this is likely because the size of the IP instance depends only on s. The hybrid algorithm generally performs best or close to best. Notice that it matches the dynamic programming algorithm closely for small values of s and then diverges when the reduction-based algorithms and heuristics are activated around $s = 10$. Observe that the hybrid algorithm is effectively constant time for large s. We expect this is because the density of strong USPs decreases rapidly with s, and that the randomly selected puzzles are likely far from satisfying Definition 2 and are quickly rejected by the downward closure heuristic.

Overall, our hybrid verification algorithm performs reasonably well in practice, despite reductions through NP-complete problems.

6 Conclusions

We initiated the first study of the verification of strong USPs and developed practical software for both verifying and searching for them. We give tight results on the maximum size of width-k strong USPs for $k \leq 5$. Although our results do not produce a new upper bound on the running time of matrix multiplication, they demonstrate there is promise in this approach. There are a number of open questions. Is strong USP verification coNP-complete? What are tight bounds on maximum size strong USPs for $k \geq 6$ and do these bound lead to asymptotically faster algorithms for matrix multiplication? The main bottleneck in our work is the size of the search space—new insights seem to be required to substantially reduce it. We have preliminary results that indicate that the size of the search space can be reduced by modding out by the symmetries of puzzles, though this has not yet led to new lower bounds.

Acknowledgments. The second and third authors thank Union College for Undergraduate Summer Research Fellowships funding their work. The authors thank the anonymous reviewers for their detailed and thoughtful suggestions for improving this work.

References

1. Alman, J., Williams, V.V.: Further limitations of the known approaches for matrix multiplication. In: Leibniz International Proceedings Information LIPIcs of the 9th

Innovations in Theoretical Computer Science (ITCS), vol. 94, p. 15, Article no. 25. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern (2018)
2. Alman, J., Williams, V.V.: Limits on all known (and some unknown) approaches to matrix multiplication. In: 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 580–591, October 2018. https://doi.org/10.1109/FOCS.2018.00061
3. Alon, N., Shpilka, A., Umans, C.: On sunflowers and matrix multiplication. Comput. Complex. **22**(2), 219–243 (2013)
4. Ambainis, A., Filmus, Y., Le Gall, F.: Fast matrix multiplication: limitations of the coppersmith-winograd method. In: 47th Annual ACM Symposium on Theory of Computing (STOC), pp. 585–593. ACM (2015)
5. Balyo, T., Heule, M.J., Jarvisalo, M.: SAT competition 2016: recent developments. In: 31st AAAI Conference on Artificial Intelligence (AAAI) (2017)
6. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. J. Comput. Syst. Sci. **87**, 119–139 (2017)
7. Blasiak, J., Church, T., Cohn, H., Grochow, J.A., Umans, C.: Which groups are amenable to proving exponent two for matrix multiplication? arXiv preprint arXiv:1712.02302 (2017)
8. Cohn, H., Kleinberg, R., Szegedy, B., Umans, C.: Group-theoretic algorithms for matrix multiplication. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 379–388, October 2005. https://doi.org/10.1109/SFCS.2005.39
9. Cohn, H., Umans, C.: A group-theoretic approach to fast matrix multiplication. In: 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 438–449, October 2003. https://doi.org/10.1109/SFCS.2003.1238217
10. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. J. Symbolic Comput. **9**(3), 251–280 (1990)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
12. Croot, E., Lev, V.F., Pach, P.P.: Progression-free sets in are exponentially small. Ann. Math. **185**, 331–337 (2017)
13. Davie, A.M., Stothers, A.J.: Improved bound for complexity of matrix multiplication. Proc. R. Soc. Edinb. Sect. A Math. **143**(2), 351–369 (2013)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness (1979)
15. Gurobi Optimization LLC: Gurobi optimizer reference manual (2018). http://www.gurobi.com
16. Kaminski, M.: A lower bound on the complexity of polynomial multiplication over finite fields. SIAM J. Comput. **34**(4), 960–992 (2005)
17. Korte, B., Vygen, J.: Combinatorial Optimization, vol. 2. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24488-9
18. Le Gall, F.: Powers of tensors and fast matrix multiplication. In: 39th International Symposium on Symbolic and Algebraic Computation (ISSAC), pp. 296–303. ACM (2014)
19. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_9
20. Oxley, J.G.: Matroid Theory, vol. 3. Oxford University Press, USA (2006)

21. Pan, V.Y.: Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In: 19th Annual Symposium on Foundations of Computer Science (FOCS), pp. 166–176. IEEE (1978)
22. Schönhage, A.: Partial and total matrix multiplication. SIAM J. Comput. **10**(3), 434–455 (1981)
23. Shpilka, A.: Lower bounds for matrix product. SIAM J. Comput. **32**(5), 1185–1200 (2003)
24. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. **13**(4), 354–356 (1969)
25. Strassen, V.: The asymptotic spectrum of tensors and the exponent of matrix multiplication. In: 27th Annual Symposium on Foundations of Computer Science (FOCS), pp. 49–54. IEEE (1986)
26. Williams, V.V.: Multiplying matrices faster than Coppersmith-Winograd. In: 44th Annual ACM Symposium on Theory of Computing (STOC), pp. 887–898. ACM (2012)