Unshuffling a Square is NP-Hard

Preliminary version — comments appreciated

Sam Buss*
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093-0112, USA
sbuss@math.ucsd.edu

Michael Soltys†
Department of Computing & Software
McMaster University
Hamilton, Ontario L8S 4K1, Canada
soltys@mcmaster.ca

March 9, 2013

Abstract

A shuffle of two strings is formed by interleaving the characters into a new string, keeping the characters of each string in order. A string is a *square* if it is a shuffle of two identical strings. There is a known polynomial time dynamic programming algorithm to determine if a given string z is the shuffle of two given strings x, y; however, it has been an open question whether there is a polynomial time algorithm to determine if a given string z is a square. We resolve this by proving that this problem is NP-complete via a many-one reduction from 3-Partition.

1 Introduction

If u, v, and w are strings over an alphabet Σ, then w is a *shuffle* of u and v provided there are (possibly empty) strings x_i and y_i such that $u = x_1x_2\cdots x_k$ and $v = y_1y_2\cdots y_k$ and $w = x_1y_1x_2y_2\cdots x_ky_k$. A shuffle is sometimes instead called a "merge" or an "interleaving". The intuition for the definition is that w can be obtained from u and v by an operation similar to shuffling two decks of cards. We use $w = u \odot v$ to denote that w is a shuffle of u and v; note, however, that in spite of the notation there can be many different shuffles w of u and v. The string w is called a *square*

*Supported in part by NSF grant DMS-1101228.

†Supported in part by an NSERC Discovery Grant. This project was carried out while the second author was visiting UCSD in Fall 2012.

provided it is equal to a shuffle of a string u with itself, namely provided $w = u \odot u$ for some string u. This paper proves that the set of squares is NP-complete; this is true even for (sufficiently large) finite alphabets.

The initial work on shuffles arose out of abstract formal languages, and shuffles were motivated later by applications to modeling sequential execution of concurrent processes. To the best of our knowledge, the shuffle operation was first used in formal languages by Ginsburg and Spanier [6]. Early research with applications to concurrent processes can be found in Riddle [19, 20] and Shaw [21]. A number of authors, including [7, 8, 10, 11, 12, 13, 14, 17, 18, 22] have subsequently studied various aspects of the complexity of the shuffle and iterated shuffle operations in conjunction with regular expression operations and other constructions from the theory of programming languages.

In the early 1980's, Mansfield [15, 16] and Warmuth and Haussler [24] studied the computational complexity of the shuffle operator on its own. The paper [15] gave a polynomial time dynamic programming algorithm for deciding the following shuffle problem: Given inputs u, v, w, can w be expressed as a shuffle of u and v, that is, does $w = u \odot v$? In [16], this was extended to give polynomial time algorithms for deciding whether a string w can be written as the shuffle of k strings $u_1, \ldots, u_k$, so that $w = u_1 \odot u_2 \odot \cdots \odot u_k$, for a *constant* integer k. The paper [16] further proved that if k is allowed to vary, then the problem becomes NP-complete (via a reduction from Exact Cover with 3-Sets). Warmuth and Haussler [24] gave an independent proof of this last result and went on to give a rather striking improvement by showing that this problem remains NP-complete even if the k strings $u_1, \ldots, u_k$ are equal. That is to say, the question of, given strings u and w, whether w is equal to an *iterated shuffle* $u \odot u \odot \cdots \odot u$ of u is NP-complete. Their proof used a reduction from 3-Partition.

The second author [23] has recently proved that the problem of whether $w = u \odot v$ is in AC^1, but not in AC^0. Recall that AC^0 (resp., AC^1) is the class of problems recognizable with constant-depth (resp., logarithmic depth) Boolean circuits.

As mentioned above, a string w is defined to be a *square* if it can be written $w = u \odot u$ for some u. Erickson [4] in 2010, asked on the *Stack Exchange* discussion board about the computational complexity of recognizing squares, and in particular whether this is polynomial time decidable. This problem was repeated as an open question in [9]. An online reply to [4] by Austrin [1] showed that the problem of recognizing squares is polynomial time decidable provided that each alphabet symbol occurs at most four times in w (by a reduction from 2-Sat); however, the general question has remained open.

The present paper resolves this by proving that the problem of recognizing squares is NP-complete, even over a sufficiently large fixed alphabet.

The NP-completeness proof uses a many-one reduction from the strongly NP-complete problem 3-PARTITION (see [5]). 3-PARTITION is defined as follows: The input is a sequence of natural numbers $S = \langle n_i : 1 \leq i \leq 3m \rangle$ such that $B = (\sum_{i=1}^{3m} n_i)/m$ is an integer and $B/4 < n_i < B/2$ for each $i \in [3m]$. The question is: can S be partitioned into m disjoint subsequences $S_1, \ldots, S_m$ such that each S_k has exactly three elements with the sum of the three members of S_k equal to B? Since 3-PARTITION is *strongly* NP-complete, it remains NP-complete even if the integers n_i are presented in unary notation.

2 Mathematical preliminaries

Let w be a string of symbols over the alphabet Σ with $w = w_1 \cdots w_n$ for $w_i \in \Sigma$, so $n = |w|$. A string u is a *subword* of w if $w = v_1 u v_2$ for some strings v_1, v_2. A string u' is a *subsequence* of w if $w = u' \odot v$ for some string v. Both the subword u and the subsequence u' contain symbols selected in increasing order from w; the symbols of u must appear consecutively in w but this is not required for u'. The exponential notation u^i, for $i \geq 0$, indicates the word obtained by concatenating i copies of u. If $u_1, \ldots, u_k$ are strings, the product notation $\prod_{\ell=1}^{k} u_\ell$ indicates the concatenation $u_1 u_2 \cdots u_{k-1} u_k$.

Now suppose that w is a square. Figure 1 gives an example of how a square shuffle $w = u \odot u$ gives rise to a bipartite graph G on the symbols of w. The graph G is defined based on a particular computation of w as a shuffle $u \odot u$, as obtained by shuffling two copies of u.[1] The vertices of G are the symbols $w_1, \ldots, w_n$ of w, and, for each i, G contains an edge joining the symbol of w corresponding to the i-th symbol of one copy of u to the symbol of w corresponding to the i-th symbol of the other copy of u. W.l.o.g., if G contains an edge joining w_j and w_k with $j < k$, then w_j corresponds to a symbol in the first copy of u, and w_k corresponds to a symbol in the second copy of u. This can be done without loss of generality, possibly by changing the order in which the symbols of the u's are shuffled out to form w. (So we could instead define G as a *directed* graph if we wished.)

The bipartite graph G has a special "non-nesting" property: if G contains an edge from w_k to w_ℓ and an edge from w_p to w_q, then it is not the case that $k < p < q < \ell$. This is because there are indices i and i' such that

[1] In general, there may be several such ways to express w as a square shuffle, even for the same u.

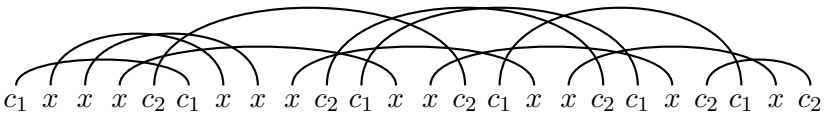

Figure 1: Let w be the string $(c_1x^3c_2)^2(c_1x^2c_2)^2(c_1xc_2)^2$, or in product notation $w = \prod_{k=0}^{2}(c_1x^{3-k}c_2)^2$. This figure shows the bipartite graph G associated with the square shuffle $w = u \odot u$ with u equal to $c_1xxc_2xc_2c_1xc_1xc_2$. It is not pictured, but we also have $w = v \odot v$ with $v = c_1x^3c_2c_1x^2c_2c_1xc_2$.

Figure 2: Examples of two crossing (and hence non-nested) edges for a graph on *abab*, and two nested edges for a graph on *abba*. Nested edges cannot appear in a graph obtained from a shuffle.

w_k and w_ℓ correspond to the i-th symbols of the first and second copies of u, and such that w_p and w_q correspond to the i'-th symbols of the two copies of u. (Compare to Figure 2.) But then $k < p$ implies $i < i'$ whereas $q < \ell$ implies that $i' < i$, and this is a contradiction.

In fact, as is easy to prove, if there is a complete bipartite graph G of degree one (i.e., a perfect matching) on the symbols of w which is non-nesting, then w can be expressed as a square shuffle $w = u \odot u$ so that G is the bipartite graph associated with this shuffle.

The non-nesting property for G can also be viewed as an "anti-Monge" condition, namely as the opposite of the Monge condition. A bipartite graph on the symbols of the string w is said to satisfy the *Monge condition* provided that, instead of having the non-nesting condition, it is prohibited that $k < p < \ell < q$. In other words, the Monge condition allows nested edges but prohibits crossing edges. The Monge condition has been widely studied for matching problems and transportation problems. Many problems that satisfy the Monge condition or the "quasi-convex" condition are known to have efficient polynomial time algorithms; for these see [3] and the references cited therein. There are fewer algorithms known for problems that satisfy the anti-Monge property, and some special cases are known to be NP-hard [2]. This is another reason why we find the NP-completeness of the square problem to be interesting: it provides a hardness result for anti-Monge matching in a very simple and abstract situation.

The non-nesting property means that we can define a non-deterministic

finite-state queue automaton which accepts precisely the strings which are squares, and this provides a convenient algorithmic characterization of the strings which are squares. A *queue automaton* is defined similarly to a non-deterministic PDA but with a queue instead of a stack. A queue automaton reads its input w from left to right. The automaton's queue is initially empty and supports the operations push-right (enqueue) and pop-left (dequeue), and the automaton accepts if its queue is empty after the last symbol of w has been read.

The non-deterministic algorithm for our queue automaton which accepts precisely the set of squares is as follows:

Repeatedly do one of the following:
a. Read the next input symbol σ and push it onto the queue, or
b. If the next input symbol σ is the same as the symbol at the top of the queue, read past the input symbol σ and pop σ from the queue.

When either step a. or b. is performed, we say that the input symbol σ has been *consumed*. In case b., we say that the symbol σ on the queue has been *matched* by the input symbol. Note that a. is always allowed, and b. only when the symbols match.

A *configuration* of the automaton is a "snapshot" of the computation, and consists of the queue contents Q and the remaining part x of the input to be read. A configuration is denoted $Q\|x$. A single step from configuration C to configuration C' is denoted $C \vdash C'$. A sequence of zero or more steps is denoted $C\vdash^* C'$. The condition $C \vdash C'$ can hold in one of two ways: if C is $Q\|\sigma x$, then either (a) C' is $Q\sigma\|x$, or (b) C' is $Q'\|x$ where $Q = \sigma Q'$. The input w is *accepted* if $\varepsilon\|w \vdash^* \varepsilon\|\varepsilon$, where ε is the empty string. More generally, a configuration C is *accepted* provided $C\vdash^* \varepsilon\|\varepsilon$.

If a computation proceeds as

$$u_1u_2u_3\|x_1x_2x_3 \vdash^* u_2u_3z_1\|x_2x_3 \vdash^* u_3z_1z_2\|x_3, \tag{1}$$

then we say that the subword x_2 of the input is *consumed* by the subword u_2 of the queue. This means that the symbols of x_2 are either matched against symbols from u_2, or are pushed onto the queue only after all the symbols of u_1 have been popped and before any symbol of u_3 is popped. In addition, no symbol of x_1 or x_3 is matched against a symbol from u_2. The word z_2 which is pushed onto the stack while x_2 is consumed by u_2 is called the *resultant*.[2] The following two simple lemmas, which will be used in the next

[2]Note that in (1) also x_1 is consumed by u_1 with resultant z_1.

section, illustrate these concepts.

Lemma 1. *If x_2 is consumed by u_2 yielding the resultant z_2, then u_2 and z_2 are subsequences of x_2. Furthermore, $x_2 = u_2 \odot z_2$.*

Proof. This holds since u_2 is equal to the subsequence of symbols of x_2 that are matched against symbols of u_2, and z_2 is the subsequence of symbols of x_2 which are enqueued and so not matched against symbols from u_2. □

Lemma 2. *Suppose e_0, e are symbols that do not appear in the strings u_i, x_i, or v. Consider the string $w = e_0 u_1 e u_2 e \cdots e u_k e e_0 x_1 e x_2 e \cdots e x_k e v$. Any accepting computation of w must proceed as:*

$$\begin{aligned} \varepsilon \| w &\vdash^* u_1 e u_2 e u_3 e \cdots e u_k e \| x_1 e x_2 e x_3 e \cdots e x_k e v \\ &\vdash^* u_2 e u_3 e \cdots e u_k e z_1 \| x_2 e x_3 e \cdots e x_k e v \\ &\vdash^* u_3 e \cdots e u_k e z_1 z_2 \| x_3 e \cdots e x_k e v \\ &\vdash^* u_k e z_1 \cdots z_{k-1} \| x_k e v \vdash^* z_1 \cdots z_k \| v \vdash^* \varepsilon \| \varepsilon, \end{aligned} \tag{2}$$

so that each x_i is consumed by the corresponding u_i with resultant z_i.

Proof. The two occurrences of e_0 must be matched with each other during the accepting computation. By the non-nesting property, this means that all the symbols between the two e_0's must be pushed onto the queue instead of matching any prior symbol. At this point, there are exactly k many e's on the queue and an equal number of e's remaining in the input. The non-nesting property thus implies that the i-th occurrence of e pushed onto the queue must be matched against the i-th occurrence of e in the second half of w. From this it is evident, again by the non-nesting property, that the accepting computation follows the pattern (2); therefore each x_i is consumed by u_i. □

3 Main Result

Theorem 3. *The set* SQUARE *of squares is NP-complete. This is true even for sufficiently large finite alphabets.*

We shall prove the theorem for an alphabet with 9 symbols. As we discuss later, a modification of our proof shows that the theorem also holds for alphabets of size 7. We conjecture that Theorem 3 holds even for alphabets of size 2, but this would require substantially new proof techniques. (Over a unary alphabet, SQUARE is just the set of even length strings.)

The rest of the paper is devoted to the proof of Theorem 3. Clearly the set of squares is in NP. To prove the NP-completeness, we shall give a logspace computable many-one reduction from 3-PARTITION to SQUARE.

Consider an instance of 3-PARTITION $S = \langle n_i : 1 \le i \le 3m \rangle$ such that the n_i's are given in unary notation and such that $B = (\sum_{i=1}^{3m} n_i)/m$ is an integer. We also have $B/4 < n_i < B/2$ for each i, but shall not use this fact. Without loss of generality, the values n_i are given in *non-increasing* order, since 3-PARTITION remains strongly NP-complete with the n_i's sorted. The many-one reduction to SQUARE constructs a string w_S over the alphabet

$$\Sigma = \{a_1, a_2, b, e_0, e, c_1, c_2, x, y\},$$

such that w_S is a square iff S is a "yes" instance of 3-PARTITION. The string w_S consists of three parts:

$$w_S \;:=\; \langle \text{loader}_S \rangle \langle \text{distributor}_S \rangle \langle \text{verifier}_S \rangle.$$

These are defined by

$$\begin{aligned}
\langle \text{loader}_S \rangle &= e_0 \prod_{i=1}^{m} (b^{2B} e) \\
\langle \text{distributor}_S \rangle &= e_0 \prod_{i=1}^{m} ((a_1 b^B a_2)^3 e) \\
\langle \text{verifier}_S \rangle &= \prod_{k=1}^{3m} [v_{4k-3} D_k v_{4k-3} v_{4k-2} D_k v_{4k-2} v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k}]
\end{aligned}$$

where

$$\begin{aligned}
v_\ell &= c_1 x^\ell y^\ell c_2 \\
D_k &= (a_1^2 b^{n_k} a_2^2)^{3m-k+1} \\
E_k &= (a_1^2 b^B a_2^2)^{3m-k} (a_1 b^{n_k} a_2) (a_1^2 b^B a_2^2)^{3m-k} \\
F_k &= (a_1^2 b^B a_2^2)^{2(3m-k)}
\end{aligned}$$

It is useful to let $U_\ell := a_1^2 b^\ell a_2^2$ as this lets us shorten the expressions for D_k, E_k, and F_k, so $D_k = U_{n_k}^{3m-k+1}$, $E_k = U_B^{3m-k} a_1 b^{n_k} a_2 U_B^{3m-k}$, and $F_k = U_B^{2(3m-k)}$.

The length of w_S is quadratic in $m + \sum_i n_i$, so w_S is polynomially bounded. It is clear that w_S can be constructed from S by a logspace computation.

The actions of the loader and distributor are relatively easy to understand, so we describe them first. As the next lemma states, the intended function of the loader is to place m many blocks of $2B$ many b's, separated by e's, onto the queue.

Lemma 4. *Any accepting computation for w_S starts off as*

$$\varepsilon \| w_S \vdash^* e_0 (b^{2B} e)^m \| \langle \text{distributor}_S \rangle \langle \text{verifier}_S \rangle.$$

In the subsequent part of the accepting computation, the i-th occurrence of the subword $(a_1 b^B a_2)^3$ in $\langle \text{distributor}_S \rangle$ will be consumed by the i-th occurrence of b^{2B} in the queue.

Proof. This is an immediate consequence of Lemma 2 since there are only two occurrences of e_0 in w_S, and since e has the same number of occurrences between the two e_0's as after the second e_0. □

Consider how the subword $(a_1 b^B a_2)^3$ can be consumed by b^{2B}. Since there are no a_1's or a_2's in b^{2B}, the a_1's and a_2's must be pushed onto the queue. In addition, exactly $2B$ of the $3B$ many occurrences of b in $(a_1 b^B a_2)^3$ must be matched against the symbols of b^{2B}. Thus, when the subword $(a_1 b^B a_2)^3$ is consumed by b^{2B} a resultant string of the form $a_1 b^{j_1} a_2 a_1 b^{j_2} a_2 a_1 b^{j_3} a_2$ must be pushed onto the queue where $j_1 + j_2 + j_3 = 3B - 2B = B$. Since the automaton is non-deterministic, any such values for j_1, j_2, j_3 can be achieved. These observations, together with Lemma 4, prove Lemma 5:

Lemma 5. *Given any sequence of non-negative integers $\langle i_k \rangle_{k=1}^{3m}$ such that*

$$\forall j \in \{1, 2, \ldots, m\}, \quad i_{3j-2} + i_{3j-1} + i_{3j} = B, \tag{3}$$

there exists a computation $\varepsilon \| w_S \vdash^* \prod_{k=1}^{3m} (a_1 b^{i_k} a_2) \| \langle \text{verifier}_S \rangle$. Conversely, if $\varepsilon \| w_S \vdash^* W \| \langle \text{verifier}_S \rangle$ then W must be of the form $\prod_{k=1}^{3m} (a_1 b^{i_k} a_2)$, so that condition (3) *holds.*

We now turn to analyzing the effect of $\langle \text{verifier}_S \rangle$. By Lemma 5, any accepting computation for $\varepsilon \| w_S$ reaches a configuration $\prod_{k=1}^{3m} (a_1 b^{i_k} a_2) \| \langle \text{verifier}_S \rangle$ satisfying (3). The intuition is that the sets $S_j := \{i_{3j-2}, i_{3j-1}, i_{3j}\}$ should be a solution to the 3-Partition problem S. By (3), the members of each S_j sum to B. Thus, the sets S_j are a solution to the 3-Partition iff the sequence $\langle i_k \rangle_{k=1}^{3m}$ is a permutation (a reordering) of $S = \langle n_k \rangle_{k=1}^{3m}$.

By Lemma 5, to complete the proof of Theorem 3, it suffices to show that $\prod_{k=1}^{3m} (a_1 b^{i_k} a_2) \| \langle \text{verifier}_S \rangle$ is accepted if and only if the sequence $\langle i_k \rangle_{k=1}^{3m}$ is a permutation of S. We first prove the easier direction of this equivalence:

Lemma 6. *Suppose $\langle i_k \rangle_{k=1}^{3m}$ is a permutation of S. Then the configuration $\prod_{k=1}^{3m}(a_1 b^{i_k} a_2) \| \langle \text{verifier}_S \rangle$ is accepted. Therefore, if S is a "Yes" instance of* 3-PARTITION*, then $\varepsilon \| w_S \vdash^* \varepsilon \| \varepsilon$ and w_S is in* SQUARE.

We prove Lemma 6 after first proving Lemmas 9 and 10.

Definition 7. A computation accepting w_S satisfies the *V-Condition* provided that for each ℓ (for $1 \leq \ell \leq 12m$) the second occurrence of the subword v_ℓ in w_S is consumed by the first occurrence of v_ℓ in w_S. This means that the symbols of the second v_ℓ are completely matched by those of the first v_ℓ.

Theorem 14, at the end of the paper, will prove that the V-Condition must hold, but for now it suffices to just assume it.

Definition 8. A string z *has k alternations of the symbols* a_1, a_2 provided $(a_1 a_2)^k$ is a subsequence of z but $(a_1 a_2)^{k+1}$ is not.

Lemma 9. *Let $i_1, \ldots, i_{3m-k+1}$ be natural numbers, and $W = \prod_{j=1}^{3m-k+1}(a_1 b^{i_j} a_2)$. Suppose the V-Condition holds for a computation containing the subcomputation*

$$W \| v_{4k-3} D_k v_{4k-3} v_{4k-2} D_k v_{4k-2} (\cdots) \vdash^* W' \| (\cdots). \tag{4}$$

(The "$(\cdots)$" denotes the rest of the input string.) Then $W' = W$, and $i_j \leq n_k$ for all j. Conversely, if each $i_j \leq n_k$, then the subcomputation (4) *can be carried out.*

Since $W' = W$, the computation (4) might seem to achieve nothing, and thus be pointless; the point, however, is that it ensures that the values i_j are $\leq n_k$. This will be useful for the proof of Lemma 11.

Proof. By the V-Condition, and the non-nesting property, the computation (4) must have the form

$$W \| v_{4k-3} D_k v_{4k-3} v_{4k-2} D_k v_{4k-2} (\cdots) \vdash^* W'' \| v_{4k-2} D_k v_{4k-2} (\cdots) \vdash^* W' \| (\cdots),$$

where W'' is the resultant when the first D_k is consumed by W, and W' is similarly the resultant when the second D_k is consumed by W''.

W and D_k both have $3m - k + 1$ alternations of a_1, a_2. Therefore, when D_k is consumed by W, the j-th a_1 (resp., a_2) symbol in W must match an a_1 (resp., a_2) from the j-th block $a_1 a_1$ (resp, $a_2 a_2$) in D_k. The other a_1 (resp., a_2) in that block is pushed onto the queue as part of W''. Furthermore, the subword b^{i_j} in the j-th component of W must match i_j of the b's in the j-th occurrence of b^{n_k} in D_k; this leaves $n_k - i_j$ many b's to be pushed onto the

queue as part of W''. This is possible if and only if $i_j \leq n_k$ for all j, and if so, $W'' = \prod_{j=1}^{3m-k+1}(a_1 b^{n_k - i_j} a_2)$.

The second D_k must be consumed by W'', and the same argument shows that this means $W' = \prod_{j=1}^{3m-k+1}(a_1 b^{i_j} a_2) = W$. □

Lemma 10. *Let $i_1, \ldots, i_{3m-k+1}$ be natural numbers, and $W = \prod_{j=1}^{3m-k+1}(a_1 b^{i_j} a_2)$. Suppose $J \leq 3m-k+1$ satisfies $i_J = \max_j\{i_j\} = n_k$. Let $i'_1, \ldots, i'_{3m-k}$ be the sequence $\langle i_j \rangle_j$ with i_J omitted, and let $W' = \prod_{j=1}^{3m-k}(a_1 b^{i'_j} a_2)$. Then there is a computation*

$$W \| v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k}(\cdots) \vdash^* W' \|(\cdots). \tag{5}$$

The computation (5) will satisfy the V-Condition. Lemma 12 below will prove a converse to Lemma 10 under the additional assumption of the V-Condition. Lemma 10, however, is all that is needed for Lemma 6.

Proof. We construct a computation of the form

$$W \| v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k}(\cdots) \vdash^* W'' \| v_{4k} F_k v_{4k}(\cdots) \vdash^* W' \|(\cdots). \tag{6}$$

Recalling that $E_k = U_B^{3m-k} a_1 b^{n_k} a_2 U_B^{3m-k}$ and using $i_J = n_k$, the first half

of the computation (6) has the form

$$
\begin{aligned}
&\prod_{j=1}^{3m-k+1} (a_1 b^{i_j} a_2) \| v_{4k-1} U_B^{3m-k} a_1 b^{n_k} a_2 U_B^{3m-k} v_{4k-1} \\
\vdash^* \quad &\prod_{j=J}^{3m-k+1} (a_1 b^{i_j} a_2) v_{4k-1} \prod_{j=1}^{J-1} (a_1 b^{B-i_j} a_2) \| U_B^{3m-k-(J-1)} a_1 b^{n_k} a_2 U_B^{3m-k} v_{4k-1} \\
\vdash^* \quad &\prod_{j=J}^{3m-k+1} (a_1 b^{i_j} a_2) v_{4k-1} \prod_{j=1}^{J-1} (a_1 b^{B-i_j} a_2) U_B^{3m-k-(J-1)} \| a_1 b^{n_k} a_2 U_B^{3m-k} v_{4k-1} \\
\vdash^* \quad &\prod_{j=J+1}^{3m-k+1} (a_1 b^{i_j} a_2) v_{4k-1} \prod_{j=1}^{J-1} (a_1 b^{B-i_j} a_2) U_B^{3m-k-(J-1)} \| U_B^{3m-k} v_{4k-1} \\
\vdash^* \quad &\prod_{j=J+1}^{3m-k+1} (a_1 b^{i_j} a_2) v_{4k-1} \prod_{j=1}^{J-1} (a_1 b^{B-i_j} a_2) U_B^{3m-k} \| U_B^{3m-k-(J-1)} v_{4k-1} \\
\vdash^* \quad &v_{4k-1} \prod_{j=1}^{J-1} (a_1 b^{B-i_j} a_2) U_B^{3m-k} \prod_{j=J+1}^{3m-k+1} (a_1 b^{B-i_j} a_2) \| v_{4k-1} \\
\vdash^* \quad &\prod_{j=1}^{J-1} (a_1 b^{B-i_j} a_2) U_B^{3m-k} \prod_{j=J+1}^{3m-k+1} (a_1 b^{B-i_j} a_2) \| \varepsilon \\
= \quad &\prod_{j=1}^{J-1} (a_1 b^{B-i'_j} a_2) U_B^{3m-k} \prod_{j=J}^{3m-k} (a_1 b^{B-i'_j} a_2) \| \varepsilon \;=\; W'' \| \varepsilon.
\end{aligned}
$$

The first and fifth steps shown above use the fact that when $a_1^2 b^B a_2^2$ is consumed by $a_1 b^{i_j} a_2$ the resultant is $a_1 b^{B-i_j} a_2$ as shown in the proof of Lemma 9. The third step matches $a_1 b^{i_J} a_2$ with the equal $a_1 b^{n_k} a_2$. The final step matches v_{4k-1}. The other steps push words v_{4k-1} and U_B from the input to the queue.

The second half of the computation (6) proceeds as follows:

$$\prod_{j=1}^{J-1}(a_1 b^{B-i'_j} a_2) U_B^{3m-k} \prod_{j=J}^{3m-k}(a_1 b^{B-i'_j} a_2) \| v_{4k}(a_1^2 b^B a_2^2)^{2(3m-k)} v_{4k}$$

$$\vdash^* \quad U_B^{3m-k} \prod_{j=J}^{3m-k}(a_1 b^{B-i'_j} a_2) v_{4k} \prod_{j=1}^{J-1}(a_1 b^{i'_j} a_2) \| (a_1^2 b^B a_2^2)^{2(3m-k)-(J-1)} v_{4k}$$

$$\vdash^* \quad \prod_{j=J}^{3m-k}(a_1 b^{B-i'_j} a_2) v_{4k} \prod_{j=1}^{J-1}(a_1 b^{i'_j} a_2) \| (a_1^2 b^B a_2^2)^{3m-k-(J-1)} v_{4k}$$

$$\vdash^* \quad v_{4k} \prod_{j=1}^{3m-k}(a_1 b^{i'_j} a_2) \| v_{4k} \vdash^* \prod_{j=1}^{3m-k}(a_1 b^{i'_j} a_2) \| \varepsilon \;=\; W' \| \varepsilon.$$

This is easily seen to be a correct computation. This proves Lemma 10. □

We can now prove Lemma 6. Suppose that $S = \langle n_j \rangle_{j=1}^{3m}$ and that $\langle i_j \rangle_{j=1}^{3m}$ is a permutation of $\langle n_j \rangle_{j=1}^{3m}$ witnessing that S is a "Yes" instance of 3-PARTITION. Let W_k be the string $\prod_{j=1}^{3m-k+1}(a_1 b^{i'_j} a_2)$ where $i'_1, \ldots, i'_{3m-k+1}$ is the sequence obtained by removing $k-1$ of the largest elements of the sequence $\langle i_j \rangle_{j=1}^{3m}$. (When there are multiple equal values i_j, they can be removed from the sequence in arbitrary fixed order, say according to the order they appear in the sequence). The n_k's are non-increasing, so the maximum i'_j is equal to n_k. Therefore, Lemmas 9 and 10 imply that

$$W_k \| v_{4k-3} D_k v_{4k-3} v_{4k-2} D_k v_{4k-2} v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k} \vdash^* W_{k+1} \| \varepsilon.$$

Combining these computations for $1 \le k \le 3m$ gives $W_1 \| \langle \text{verifier}_S \rangle \vdash^* \varepsilon \| \varepsilon$. Lemma 5 gives $\varepsilon \| w_S \vdash^* W_1 \| \langle \text{verifier}_S \rangle$. Thus $\varepsilon \| w_S \vdash^* \varepsilon \| \varepsilon$. This proves Lemma 6. □

The next lemma gives the converse of Lemma 6, under the assumption that the V-Condition holds. This, together with Theorem 14 stating that the V-Condition must hold, will prove Theorem 3.

Lemma 11. *Let S be an instance of* 3-PARTITION, *$\langle i_k \rangle_{k=1}^{3m}$ satisfy condition* (3) *of Lemma 5, and $W = \prod_{k=1}^{3m}(a_1 b^{i_k} a_2)$. Suppose that $W \| \langle \text{verifier}_S \rangle \vdash^* \varepsilon \| \varepsilon$ with a computation that satisfies the V-Condition, so $\varepsilon \| w_S \vdash^* \varepsilon \| \varepsilon$ and w_S is in* SQUARE. *Then S is a "Yes" instance of* 3-PARTITION.

The main new tool needed for proving Lemma 11 is a converse of Lemma 10:

Lemma 12. *Let $1 \le k \le 3m$, let $i_1, \ldots, i_{3m-k+1}$ be natural numbers, and $W_k = \prod_{j=1}^{3m-k+1}(a_1 b^{i_j} a_2)$. Suppose that $\max_j\{i_j\} \le n_k$. Further suppose there is a computation*

$$W_k \| v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k} (\cdots) \vdash^* W_{k+1} \| (\cdots) \tag{7}$$

that satisfies the V-Condition. Then there is a J such that $i_J = \max_j\{i_j\} = n_k$ and such that, letting $i'_1, \ldots, i'_{3m-k}$ be the sequence $\langle i_j \rangle_j$ with i_J omitted, we have $W_{k+1} = \prod_{j=1}^{3m-k}(a_1 b^{i'_j} a_2)$.

Before we prove Lemma 12, we indicate how it, and the V-Condition assumption, imply Lemma 11 and thus imply Theorem 3. Suppose C is a computation $\varepsilon \| w_S \vdash^* \varepsilon \| \varepsilon$ that obeys the V-Condition. For $1 \le k \le 3m+1$, define the strings V_k to be such that C contains the configurations

$$V_k \parallel \prod_{\ell=k}^{3m} [v_{4\ell-3} D_\ell v_{4\ell-3} v_{4\ell-2} D_\ell v_{4\ell-2} v_{4\ell-1} E_\ell v_{4\ell-1} v_{4\ell} F_\ell v_{4\ell}] .$$

Of course, these V_k's are the intermediate queue contents as $\langle \text{verifier}_S \rangle$ is consumed. For $1 \le k \le 3m$, define V'_k to be the strings such that C contains the configuration

$$V'_k \parallel v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k} \prod_{\ell=k+1}^{3m} [v_{4\ell-3} D_\ell v_{4\ell-3} v_{4\ell-2} D_\ell v_{4\ell-2} v_{4\ell-1} E_\ell v_{4\ell-1} v_{4\ell} F_\ell v_{4\ell}] .$$

Claim 13. *We have:*

(a) *V_1 is equal to $\prod_{j=1}^{3m-k+1}(a_1 b^{i_j} a_2)$ for some sequence $\langle i_j \rangle_{j=1}^{3m}$ satisfying (3).*

(b) *For $1 \le k \le 3m+1$, V_k equals $\prod_{j=1}^{3m-k+1}(a_1 b^{i'_j} a_2)$ for some sequence $\langle i'_j \rangle_j$ which is obtained from $\langle i_j \rangle_{j=1}^{3m}$ by removing (instances of) the $k-1$ largest entries of $\langle n_j \rangle_{j=1}^{3m}$.*

(c) *For $1 \le k \le 3m$, V'_k equals V_k, and its maximum i'_j value is less than or equal to n_k.*

The claim is proved by induction on k. Part (a), and the equivalent $k = 1$ case of (b), follows from Lemma 5. Part (c) for a given k follows from Lemma 9 and from the induction hypothesis that (b) holds for the same value of k. Part (b) for $k > 1$ follows from Lemma 12 and from the induction hypothesis that (b) and (c) hold for $k-1$. Since $V_{3m+1} = \varepsilon$, part (b) implies that the sequence $\langle i_j \rangle_{j=1}^{3m}$ is a reordering of $\langle n_j \rangle_{j=1}^{3m}$. And,

since (3) holds, $\langle i_j \rangle_{j=1}^{3m}$ witnesses that S is a "Yes" instance of 3-PARTITION. This completes the proof of Lemma 11, and thereby Theorem 3, modulo the proofs of Lemma 12 and Theorem 14. □

Proof. (of Lemma 12.) Consider a particular computation C as in (7) that satisfies the V-Condition. C has the form

$$W_k \| v_{4k-1} E_k v_{4k-1} v_{4k} F_k v_{4k} (\cdots) \vdash^* Z \| v_{4k} F_k v_{4k} \vdash^* W_{k+1} \| (\cdots)$$

where Z is the resultant of E_k being subsumed by W_k. By assumption, W_k has $3m-k+1$ alternations of a_1, a_2, whereas E_k has $2(3m-k)+1$ and F_k has $2(3m-k)$. The string E_k is a concatenation of "blocks" of the form $a_1 b^{n_k} a_2$ or the form $U_B = a_1^2 b^B a_2^2$. Each subword $a_1 b^{i_j} a_2$ in W has its symbol a_1 matched by some a_1 in E_k and its a_2 matched by some a_2 in the same block or a later block of E_k: these symbols a_1 and a_2 in E_k determine a contiguous sequence of blocks in E_k which is consumed by $a_1 b^{i_j} a_2$. We call these blocks the "j-consumed" portion of E_k, and denote it Y_j. The resultant of $a_1 b^{i_j} a_2$ and its j-consumed portion is denoted Z_j. There may also be blocks of E_k which are not part of any j-consumed portion, and these are called "non-matched" blocks of E_k. The string Z is then the concatenation of the words Z_j, for $1 \le j \le 3m-k+1$, interspersed with the non-matched blocks of E_k.

Let us consider the possible resultants Z_j. We can write E_k as $E_k = P_1 P_2 P_3$ where $P_1 = P_3 = U_B^{3m-k}$ and $P_2 = a_1 b^{n_k} a_2$. There are several cases to consider.

Case a. Y_j is $(a_1^2 b^B a_2^2)^\ell$ for some $\ell \ge 1$, and thus is a subword of either P_1 or P_3 in E_k. When Y_j is consumed by $a_1 b^{i_j} a_2$, one of the two initial a_1's, any i_j of the b's, and then one of the two final a_2's are matched; the remaining symbols of Y_j become the resultant Z_j and are pushed onto the queue. Therefore, Z_j is equal to

$$Z_j \;=\; a_1 b^{B-m_1} \prod_{s=2}^{\ell} (a_2^2 a_1^2 b^{B-m_s}) a_2 \tag{8}$$

where $m_1 + m_2 + \cdots + m_\ell = i_j$. Note that Y_j and Z_j both have ℓ alternations of a_1, a_2.

Case b. Y_j spans from P_1 to P_3 and equals $(a_1^2 b^B a_2^2)^{\ell_1} a_1 b^{n_k} a_2 (a_1^2 b^B a_2^2)^{\ell_2}$ where $\ell_1, \ell_2 \ge 1$. Arguing as in the previous case, Z_j is equal to

$$a_1 b^{B-m_1} a_2^2 \prod_{s=2}^{\ell_1} (a_1^2 b^{B-m_s} a_2^2) a_1 b^{n_k - m_{\ell_1+1}} a_2 \prod_{s=\ell_1+2}^{\ell_1+\ell_2} (a_1^2 b^{B-m_s} a_2^2) a_1^2 b^{B-m_{\ell_1+\ell_2+1}} a_2$$

where $m_1 + m_2 + \cdots + m_{\ell_1+\ell_2+1} = i_j$. In this case, Y_j and Z_j both have $\ell_1 + \ell_2 + 1$ alternations of a_1, a_2.

Case c. Y_j is $a_1 b^{n_k} a_2$, namely, $Y_j = P_2$. In this case, Z_j is equal to just $b^{n_k - i_j}$. If $i_j = n_k$, then Z_j is just ε: this is called a "full cancellation" case. Note that Z_j has zero alternations of a_1, a_2, whereas Y_j has one alternation.

Case d. Y_j is $(a_1^2 b^B a_2^2)^\ell a_1 b^{n_k} a_2$. We now have

$$Z_j \;=\; a_1 b^{B-m_1} a_2^2 \prod_{s=2}^{\ell} (a_1^2 b^{B-m_s} a_2^2) a_1 b^{n_k - m_{\ell+1}} \tag{9}$$

where $m_1 + \cdots + m_{\ell+1} = i_j$. Z_j consists of a part with ℓ alternations of a_1, a_2 followed by a subsequent a_1 (and possibly b's). In the "full cancellation" case, $m_{\ell+1} = n_k$, and since $i_j \leq n_k$, we have $n_k = m_{\ell+1} = i_j$ and, for $s \leq \ell$, $m_s = 0$. Otherwise, Z_j ends with one or more b's.

Case e. The case where Y_j is $a_1 b^{n_k} a_2 (a_1^2 b^B a_2^2)^\ell$ is completely analogous to case d., and we omit it.

For simplicity, let's assume for the moment that neither case d. nor e. occurs. This means that there is at most one occurrence of either case b. or c., and the rest of the cases are case a. In cases a. and b., Z_j has the same number of alternations of a_1, a_2 as Y_j. Of course the number of alternations in the non-matched blocks does not change. Therefore, Z has $2(3m-k)+1$ alternations of a_1, a_2 if case c. does not occur, and has $2(3m-k)$ alternations if case c. does occur. The word F_k has $2(3m-k)$ alternations of a_1, a_2, and since F_k is consumed by Z, Lemma 1 implies that Z cannot have more alternations of a_1, a_2 than F_k. Therefore, it must be that case c. occurs and case b. does not.

We claim that case c. must occur as a full cancellation case. If not, then Z will consist of a subword with $3m-k$ alternations of a_1, a_2 that came from P_1, followed by some non-zero number of b's from the Z_j of case c., and then by another subword with $3m-k$ alternations of a_1, a_2 that came from P_3. In other words, $(a_1 a_2)^{3m-k} b (a_1 a_2)^{3m-k}$ is a subsequence of Z. It is not, however, a subsequence of F_k, contradicting the fact that F_k is consumed by Z. It follows by Lemma 1 that case c. must have occurred in the full cancellation version. Let J be the value of j for which case c. occurred; since it was a case of full cancellation, $i_J = n_k$.

Therefore, Z has $2(3m-k)$ alternations of a_1, a_2, and is the concatenation of the $3m-k$ many Z_j's that arose in case a. (the empty Z_J has been dropped) and of zero or more non-matched $a_1^2 b^B a_2^2$'s. The fact that F_k and Z both have $2(3m-k)$ alternations of a_1, a_2, means that the way F_k can be consumed by Z is tightly constrained. First, any non-matched block $a_1^2 b^B a_2^2$ in Z must consume (and fully match) an identical block in F_k leaving a resultant of ε. Second, any Z_j with ℓ alternations of a_1, a_2 will be of the form (8) and must consume a subword $G_j = (a_1^2 b^B a_2^2)^\ell$ of F_k. The first a_1 of Z_j must match one of the two first a_1's of G_j; the final a_2 of Z_j must match one the final two a_2's of G_j; the other subwords a_1^2 and a_2^2 of Z_j must match identical subwords in G_j; and the $\ell B - i_j$ many b's in Z_j all must match b's in G_j. This can always be done, no matter what the values of the m_s's in Z_j are. Since G_j has ℓB many b's, the consumption of G_j by Z_j yields a resultant W'_j equal to $a_1 b^{i_j} a_2$.

It follows that, when F_k is consumed by Z, the resultant equals the concatenation of the strings $W'_j = a_1 b^{i_j} a_2$, omitting the word w_J (which triggered case c.). In other words, the resultant is just W_{k+1}, proving Lemma 12 in this case.

We still have to consider the case where case d. or e. occurs. The cases are symmetric, so suppose case d. occurs, and thus the rest of the Z_j's are generated by case a. Suppose Z_j is obtained via case d., and so is equal to (9). We claim that this must be a full cancellation case of case d., with $n_k = i_j$. If not, then Z contains $3m-k$ alternations of a_1, a_2 up through Z_j, followed by the final a_1 of Z_j and at least one b at the end of Z_j, and then followed by $3m-k$ alternations of a_1, a_2 in the remaining part of Z. In other words, $(a_1 a_2)^{3m-k} a_1 b (a_1 a_2)^{3m-k}$ is a subsequence of Z. It is not a subsequence of F_k however, contradicting the fact that F_k is to be consumed by Z. Thus we must have a full cancellation case of case d.

Now consider what immediately follows Z_j in Z. It must either be of the form $a_1^2 b^B a_2^2$ (obtained from a non-matched block), or, referring to (8), be the word of the form

$$Z_{j+1} \;=\; a_1 b^{B-m'_1} \prod_{s=2}^{\ell'} (a_2^2 a_1^2 b^{B-m'_s}) a_2,$$

obtained from case a. for Y_{j+1}. We claim it is impossible for $Z_j a_1^2 b^B a_2^2$ to be a subword of Z. If so, $(a_1 a_2)^{3m-k} a_1^2 (a_1 a_2)^{3m-k}$ is a subsequence of Z, and thus Z is not a subsequence of F_k. As before, this is a contradiction.

We have eliminated the other possibilities, so $Z_j Z_{j+1}$ is a subword of Z

and $n_k = m_{\ell+1}$. Therefore, $m_s = 0$ for all $s \leq \ell$, and we have

$$Z_j Z_{j+1} \;=\; a_1 b^B (a_2^2 a_1^2 b^B)^\ell \prod_{s=1}^{\ell'} (a_2^2 a_1^2 b^{B-m'_s}) a_2.$$

Note that $Z_j Z_{j+1}$ contains $\ell + \ell'$ alternations of a_1, a_2. Also note that the subword $Y_j Y_{j+1}$ contains $\ell + \ell' + 1$ many such alternations. Therefore Z has $3m - k$ alternations of a_1, a_2, namely one fewer than E_k (as desired). Similarly to the argument four paragraphs above, it follows that $Z_j Z_{j+1}$ must consume a subword G of F_k of the form $(a_1^2 b^B a_2^2)^{\ell+\ell'}$. Since $m'_1 + \cdots + m'_{\ell'} = i_{j+1}$, $Z_j Z_{j+1}$ has $(\ell+\ell')B - i_j$ many b's. Hence the resultant when G is consumed by Z_j is equal to $a_1 b^{i_{j+1}} a_2$. If follows again that when F_k is consumed by Z it yields the resultant W_{k+1} as desired.

This completes the proof of Lemma 12. □

The V-Condition. The proof of Theorem 3 will be finalized once we prove that the V-Condition must hold:

Theorem 14. *Any accepting computation $\varepsilon||w_S \vdash^* \varepsilon||\varepsilon$ satisfies the V-condition.*

Let

$$V \;=\; \prod_{i=0}^{\ell-1} v_{\ell-i}^2 \;=\; \prod_{j=\ell,\ldots,2,1} (c_1 x^j y^j c_2)^2, \tag{10}$$

i.e., $V = v_\ell v_\ell \cdots v_2 v_2 v_1 v_1$. (The dependence of V on ℓ is suppressed in the notation.) The symbols c_1, x, y, c_2 occur only in the subwords v_ℓ of w_S, and V is the subsequence of w_S containing these symbols, but in reversed order. (We use the reversed order since it makes the proof below a little simpler to state.) Clearly, any expression of w_S as a square shuffle induces a square shuffle for V. Therefore Theorem 14 is a consequence of Theorem 15:

Theorem 15. *Let $\ell \geq 1$. The only accepting computation $\varepsilon\|V \vdash^* \varepsilon\|\varepsilon$ is the one that matches each v_k in V with the other v_k in V.*

As a side remark, it is interesting to note that Figure 1 illustrates that Theorem 15 would not hold if the v_j's were instead defined to equal $c_1 x^j c_2$ with the y's omitted. Theorem 15 follows from the next three lemmas.

Definition 16. Each subword x^j or y^j shown in the definition of V in (10) is called an *x-block* or a *y-block*, respectively. We also refer to them as *full x-blocks* or *full y-blocks* after they have been pushed onto the queue to emphasize that the complete subword x^j or y^j has been pushed onto the queue without any x or y from the block being matched.

Lemma 17. *If C is an accepting computation of V, then C does not match any x (resp., y) with another symbol from the same x-block (resp., y-block).*

Proof. V contains an even number of c_1's and an even number of c_2's. Consider some x- or y-block β in C. There is either an odd number of c_1's before (and therefore, after) β in V, or an odd number of c_2's before (and after) β in V. If there are, say, odd numbers of c_1's then some c_1 before β must match some c_1 after β during C. The non-nesting condition now implies that no two symbols in β can be matched. □

Lemma 18. *Suppose C is an accepting computation for V, and C does not completely match the first subword v_ℓ of V with the second v_ℓ of V (i.e., at least one symbol from the second v_ℓ of V is pushed onto the queue). Then there is a point in C where the queue contains either two full x-blocks or two full y-blocks.*

Proof. The proof splits into cases depending on how C starts off. For the first case, suppose the first c_1 of V does not match the second c_1 of V. By the non-nesting condition, this implies that the subword $x^\ell y^\ell c_2 c_1 x^\ell y^\ell$ is pushed onto the queue. This puts two full x-blocks and two full y-blocks on the queue, so the lemma holds in this case. So, henceforth assume that the first c_1 matches the second c_1.

Now suppose the first x-block x^ℓ does not completely match the second x^ℓ. Therefore, some of the x's in the first x^ℓ match symbols from some x^j with $j < \ell$. This x-block x^j comes *after* the first two y-blocks (which equal y^ℓ), so by the non-nesting condition, these two y-blocks are on the queue by the time the algorithms consumes the x-block x^j. So the lemma holds in this case as well. Assume henceforth that the first $c_1 x^\ell$ is completely matched with the second $c_1 x^\ell$ by C.

Finally, suppose that the first subword $y^\ell c_2$ does not completely match the second $y^\ell c_2$. In this case, we claim that, after consuming the second c_2, C's queue will contain $y^m c_2 y^m c_2$. To see this note that either the two y^ℓ's completely match (so $m = 0$) and then the c_2's are not matched by assumption, or the two y^ℓ's do not completely match (so $m > 0$) and then the c_2's must be pushed to the queue since they cannot be matched while a y is at the top of the queue. At any rate, the queue contains two c_2's once the second c_2 is consumed. By the non-nesting property, the second c_2 on the queue must match the *fourth* c_2 of V or a later c_2 of V. Therefore, the two x-blocks $x^{\ell-1}$ that come prior to the fourth c_2 are pushed onto the queue, and the lemma holds again in this case. □

Lemma 19. *If C is an accepting computation for V and at some point in C the queue contains two full x-blocks (respectively, contains two full y-blocks), then there is a later point at which the queue contains two full y-blocks (respectively, contains two full x-blocks).*

Proof. Suppose C has two full x-blocks x^m and then x^j in the queue. Note $j \leq m$. Let the computation continue until x^m has been matched, and then until x^j has been matched. The symbols of x^m are matched by symbols from x-blocks x^s that have $s < m$ (since the block x^j was intervening). Therefore, x^m's symbols must match x's from at least two distinct x-blocks. Between these two x-blocks there is a y-block, and by the non-nesting condition this y-block is pushed onto the queue in its entirety. Similarly the x-block x^j is matched against symbols from at least two distinct x-blocks, and again there is a y-block between those two x-blocks that is entirely pushed onto the queue. Therefore, once the x^j is matched, there are at least two full y-blocks in the queue.

The dual argument works with x and y interchanged. □

We can now prove Theorem 15:

Proof. The proof is by induction on ℓ. The base case $\ell = 1$ is trivial. Suppose $\ell > 1$. If an accepting computation C matches the first two subwords $c_1 x^\ell y^\ell c_2$ against each other completely, then the rest of the computation C is an accepting computation on the rest of V, namely V minus these first two subwords. By the induction hypothesis, the latter accepting computation matches each pair of subwords $c_1 x^j y^j c_2$, and the theorem holds. Otherwise, if the first two subwords $c_1 x^\ell y^\ell c_2$ of V are not completely matched by C, then Lemma 18 states that C contains some point where its queue contains either two full x-blocks or two full y-blocks. Lemma 19 then implies that C's queue must contain two full x- or y-blocks infinitely often, which is a contradiction. □

That completes the proof of Theorem 15, and thereby the proof of Theorem 14, giving us the V-Condition that was needed for the proof of Theorem 3. □

We conclude by indicating how to modify the proof of Theorem 3 to use an alphabet size of only 7. The first idea is that the symbol e_0 can be eliminated by using two extra occurrences of v_ℓ subwords. Eliminating the

use of the symbol e is more difficult however. We briefly sketch how this can be done with a modified definition of w_s:

$$w'_S \;:=\; \langle \text{loader}'_S \rangle \langle \text{distributor}'_S \rangle \langle \text{verifier}'_S \rangle.$$

The string $\langle \text{verifier}'_S \rangle$ is defined like $\langle \text{verifier}_S \rangle$, but with each subword v_ℓ replaced by $v_{\ell+4}$. Then define

$$\begin{aligned}
\langle \text{loader}'_S \rangle &= v_1 \prod_{i=1}^{m} \Big(a_1^2 b^{2B+3} a_2^2 \Big) v_1 \\
\langle \text{distributor}'_S \rangle &= v_2 \prod_{i=1}^{m} \Big(a_1^2 (a_1 b^{B+1} a_2)^3 a_2^2 \Big) v_2 v_3 (a_1^2 b^B a_2^2)^{3m} v_3 v_4 (a_1^2 b^B a_2^2)^{3m} v_4.
\end{aligned}$$

We claim that Lemma 5 still holds for $\langle \text{loader}'_S \rangle \langle \text{distributor}'_S \rangle$, and thus w'_S defines a logspace computable many-one reduction from 3-PARTITION to SQUARE. We leave the details of the proof to the reader.

References

[1] P. AUSTRIN, *How hard is unshuffling a string (reply)*. CS Theory Stack Exchange reply to [4]. `http://cstheory.stackexchange.com/q/692`, August 2010.

[2] R. E. BURKHARD, E. ÇELA, G. ROTE, AND G. J. WOEGINGER, *The quadratic assignment problem with a monotone anti-Monge and a symmetric Toeplitz matrix: Easy and hard cases*, Mathematical Programming, 82 (1998), pp. 125–158.

[3] S. R. BUSS AND P. N. YIANILOS, *Linear and $O(n \log n)$ time minimum-cost matching algorithms for quasi-convex tours*, SIAM Journal on Computing, 27 (1998), pp. 170–201. An extended abstract of this paper appeared in *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994, pp. 65-76.

[4] J. ERICKSON, *How hard is unshuffling a string?* CS Theory Stack Exchange posting. `http://cstheory.stackexchange.com/questions/34/how-hard-is-unshuffling-a-string`, August 2010.

[5] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[6] S. Ginsburg and E. Spanier, *Mappings of languages by two-tape devices*, Journal of the A.C.M., 12 (1965), pp. 423–434.

[7] J. Gischer, *Shuffle languages, petri nets, and context-sensivite grammars*, Communications of the ACM, 24 (1981), pp. 597–605.

[8] H. Gruber and M. Holzer, *Tight bounds on the descriptional complexity of regular expressions*, in Proc. Intl. Conf. on Developments in Language Theory (DLT), Springer Verlag, 2009, pp. 276–287.

[9] D. Henshall, N. Rampersad, and J. Shallit, *Shuffling and unshuffling*, Bulletin of the EATCS, 107 (2012), pp. 131–142.

[10] M. Jantzen, *The power of synchronizing operations on strings*, Theoretical Computer Science, 14 (1981), pp. 127–154.

[11] ——, *Extending regular expressions with iterated shuffle*, Theoretical Computer Science, 38 (1985), pp. 223–247.

[12] J. Jedrzejowicz, *Structural properties of shuffle automata*, Grammars, 2 (1999), pp. 35–51.

[13] J. Jedrzejowicz and A. Szepietowski, *Shuffle languages are in P*, Theoretical Computer Science, 250 (2001), p. 31=53.

[14] ——, *On the expressive power of the shuffle operator matched with intersection by regular sets*, Theoretical Informatics and Applications, 35 (2005), pp. 379–388.

[15] A. Mansfield, *An algorithm for a merge recognition problem*, Discrete Applied Mathematics, 4 (1982), pp. 193–197.

[16] ——, *On the computational complexity of a merge recognition problem*, Discrete Applied Mathematics, 1 (1983), pp. 119–122.

[17] A. J. Mayer and L. J. Stockmeyer, *The complexity of word problems — this time with interleaving*, Information and Computation, 115 (1994), pp. 293–311.

[18] W. F. Ogden, W. E. Riddle, and W. C. Rounds, *Complexity of expressions allowing concurrency*, in Proc. 5th ACM Symposium on Principles of Programming Languages (POPL), 1978, pp. 185–194.

[19] W. E. Riddle, *A method for the description and analysis of complex software systems*, SIGPLAN Notices, 8 (1973), pp. 133–136.

[20] ——, *An approach to software system modelling and analysis*, Computer Languages, 4 (1979), pp. 49–66.

[21] A. C. Shaw, *Software descriptions with flow expressions*, IEEE Transactions on Software Engineering, SE-4 (1978), pp. 242–254.

[22] T. Shoudai, *A P-complete language describable with iterated shuffle*, Information Processing Letters, 41 (1002), pp. 233–238.

[23] M. Soltys, *A tight circuit-complexity bound for shuffle*. Submitted for publication, October 2012.

[24] M. K. Warmuth and D. Haussler, *On the complexity of iterated shuffle*, Journal of Computer and System Sciences, 28 (1984), pp. 345–358.