JUGGLING WITH PATTERN MATCHING

Jean Cardinal
Université Libre de Bruxelles
jcardin@ulb.ac.be

Steve Kremer
University of Birmingham
S.Kremer@cs.bham.ac.uk

Stefan Langerman[a]
Université Libre de Bruxelles
slanger@ulb.ac.be

[a]Chargé de recherche FNRS

Abstract

In the late eighties, it was shown that juggling patterns can be described by strings of numbers with fascinating combinatorial properties that have since then been studied by many mathematicians and computer scientists. In this paper, we propose to study juggling patterns from a pattern matching point of view. Inspired by pattern matching algorithms based on convolution, we propose a fast algorithm for finding transitions between juggling states. Apart from being a fun application of pattern matching theory, it provides a practical tool in the experimental design of (large) juggling patterns.

1. *Introduction*

Juggling can be defined as the art of tossing objects in the air and catching them, keeping several in the air at the same time. Different kinds of objects can be used: balls, sticks, clubs, rings, torches, etc., and juggling acts can involve more than one juggler. Objects are usually thrown according to some kind of regular pattern, the beauty and complexity of which make juggling popular among mathematicians, physicists, and, quite naturally, computer scientists.

There exist many scientific papers about juggling. The earliest ones are probably from Claude Elwood Shannon, the inventor of information theory, in the early fifties [12, 13]. Among other results, he derived a physical law relating the time an object spends in the air and the time it spends in the hands of the juggler. He even designed a simple mechanical juggling machine.

An actual mathematical theory of juggling was however not available before the invention, by many different jugglers, of the *siteswap* notation, around 1985 [8, 14]. In this notation, a two-hand juggling pattern is denoted by a string of positive integer numbers, where each number represents a throw, the value of the number is related to the height of the throw, and the length of the string is equal to the period of the juggled pattern. This notation assumes that the throws are made alternately by each hand and that no two balls can be caught or thrown at the same time. A number of contributions have been made in the analysis of siteswap patterns. The number of patterns with b balls and a bounded throw height was established and generalized by different authors [15, 1, 2]. We summarize the essential properties of siteswaps in section 2.1. In section 2.2, we define the *landing state graph*, and explain how periodic juggling sequences correspond to cycles in this infinite graph. These background notions can be found in the recent book from Polster [11], probably the most complete available source of information on mathematical aspects of juggling.

Our contribution is about finding transitions between juggling patterns, i.e. sequences of throws that allow to move from one pattern to another. This kind of problem was tackled only once in [9], where heuristic algorithms – neural networks and genetic algorithms – were used to find "nice" transitions.

It is not always possible to find a transition sequence of a given length. So our first concern will be to find what are the achievable lengths. In section 3, we describe the problem of finding all achievable transition sequence lengths between two juggling patterns as that of finding paths in the landing state graph. We also show that it can be seen as a kind of pattern matching problem, where the destination landing state has to be suitably matched with the source landing state. A trivial algorithm with quadratic complexity is derived quickly. In section 4, we review a recent variation from Kalai [5] of the Karp-Rabin pattern matching method [6]. It puts forward the relevance of fast convolution in the context of generalized pattern matching. In section 5, inspired by the above algorithm, we describe an extremely simple method for finding all possible path lengths between two landing states. The algorithm has $O(n \log n)$ complexity and is deterministic. We prove its optimality and give some examples. In section 6, we discuss ways of generating transition sequences of a given length. We provide a simple algorithm for enumerating those sequences and a formula for counting them. The latter could be used to generate a transition sequence uniformly at random. We conclude with some open problems.

2. *Preliminaries*

The content of this section is the essential background material on mathematical juggling.

2.1. *Siteswaps*

Siteswap patterns (or just "siteswaps") allow jugglers to communicate their tricks without having to actually show them. The juggling patterns that can be represented with this notation are those in which:

- two hands are used alternately,
- two objects (say, balls) cannot be thrown at the same time,
- two objects cannot be caught at the same time.

Furthermore, the notation does not characterize the movements of the hands (crossings, for instance), nor does it provide any precise information on the trajectories of the balls in the air.

Let us represent throws and catches on a time axis, and label them according to whether they are performed by the right or left hand. This is done on Fig. 1. Note that catching and throwing a ball is modelled as an atomic action. In that figure, we represented the trajectories in time of the balls. Since a ball cannot be sent in the past, all trajectories are hopping forward from throws to throws. The number of distinct trajectories, or *orbits*, is equal to the number of balls b, which must be finite.

The *height* of a throw is the number of throws occuring between this throw and the next throw of the same ball. A (infinite) juggling sequence is nothing more than the sequence of heights. It can be formalized as a bijection $f : \mathbb{Z} \rightarrow \mathbb{Z}$ such that $f(x) > x$ (see [1]). In Fig. 1, each throw is labeled with its height, showing the portion of the sequence being juggled. Note that odd heights correspond to a ball thrown at the other hand, while even heights corresponds to balls that are thrown at the current hand. Empty throws are possible if no ball is scheduled to land at the current beat: they are denoted by **0**.

What is commonly referred to as siteswaps are simply periodic juggling sequences. Clearly, not every sequence of positive integers can be a siteswap, since the restrictions above must hold. The sequence depicted in Fig. 1 actually corresponds to the siteswap **441**. The simplest patterns are those in which the height is constant. They can be represented by a single number. The siteswap **3**, for instance, represents the classical 3-ball cascade in which each ball is thrown alternately to the other hand. We can realize that the height is equal to the number of balls, since two balls must be

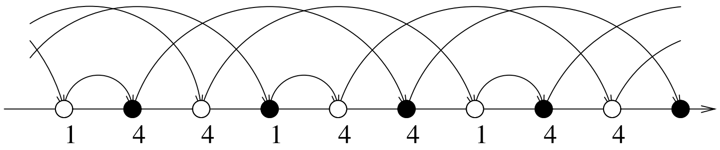


Figure 1: Representation of a juggling sequence on the time axis

thrown at height three between two throws of a given ball. Similarly, the siteswaps **4**,**5** and **6**, etc. represent respectively the standard 4-, 5- and 6-ball cascades. There are many ways to generate siteswaps, either exhaustively or randomly [11]. Examples of valid siteswaps include **423**,**522**, or **441** (3 balls), **534** (4 balls), **75**, **756**, or **774** (6 balls). An example of invalid siteswap is **324**, as the two first throws will arrive at the same time, which violates the third condition on siteswaps.

A nice property of siteswaps, called the *average theorem*, is that the sum of the throw heights divided by the pattern length is equal to the number of balls b. This is true in particular, as we have seen, when the siteswap has length one. We can compute the number of balls in the example of Fig. 1:$(4 + 4 + 1)/3 = 3$. A rigorous proof of the theorem is provided in [1] and reproduced in [11]. Note that the average theorem yields a necessary condition for a siteswap to be valid, however, as witnessed by the siteswap **324**, this condition is not sufficient. A necessary and sufficient condition is as follows: $s_0 s_1 \ldots s_{p-1}$ is a valid siteswap if and only if the application $i \mapsto (i + s_i) \mod p$ is a permutation of $\{0, 1, \ldots, p-1\}$.

The word siteswap comes from the *siteswapping* operation, that consists in swapping the landing times of two balls in a given siteswap in order to generate a new valid siteswap. Swapping for instance the landing times of the first **4** throw and the **1** throw in the pattern **441** leads to a new, valid pattern **342**.

2.2. *Landing State Graph*

Let us freeze the action described on Fig. 1 at some point t in time, and represent the expected landing times of the balls currently in the air. This is shown on top of Fig. 2, where we froze just before the **1** throw. A ball is planned to land right before time $t + 1$, and two more before times $t + 3$ and $t + 4$, respectively. We do not care which ball they are, and simply denote by a '1' an expected landing, and by a '0' the absence of any landing. The bit vector obtained this way is called the *landing state* at time t. On top of Fig. 2, the landing state is 1011.

Let us now move one step forward in time, at time $t + 1$, and see what we can do

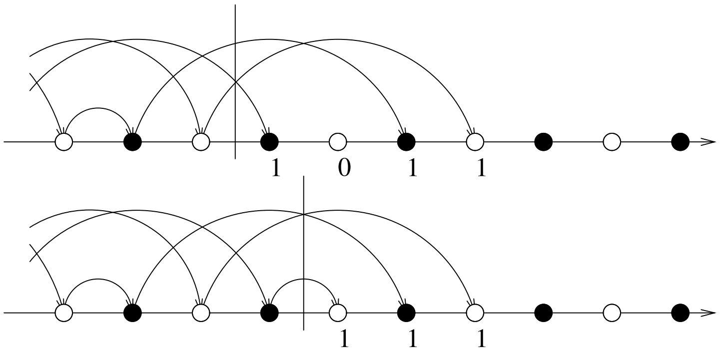


Figure 2: Landing states

in the meanwhile with the landing ball. Clearly, it must be thrown to some point in the future, and it cannot be sent in the future at a time when a landing is expected. In Fig. 2, we show how the ball can be sent to the next beat using a **1** throw, following the **441** siteswap. The new landing state is 111.

So a landing state for b balls is simply a bit vector in which exactly b bits are set to '1'. It can be thought of as followed by an infinity of '0's, but we will use the convention to represent landing states up to the last '1' bit.

A throw acts on a landing state as a left shift if the leftmost bit of the state is '0', and otherwise as a combination of left shift and setting a '0'-bit to '1'. This is exactly how an edge is defined in the landing state graph $\mathcal{G}_b$.

Definition 1. *The* b-ball landing state graph $\mathcal{G}_b$ *is an directed graph with vertex set defined as the set of states* $(0^*1)^b$ *and in which there exists an edge from state* $X = x_1x_2 \dots x_n$ *to state* Y *if and only if* Y *can be obtained by modifying* X *following Algorithm 1. Each edge of* $\mathcal{G}_b$ *is labeled with the height of the corresponding throw, equal to* **0** *if* $c = 0$*, or to the index of the new '1' bit otherwise.*

Landing state graphs are usually partly represented, according to a given maximum throw height. A number of typeset landing state graphs with bounded throw height are given by Lundmark [7].

From the definition of a landing state graph, a siteswap is nothing but the sequence of edge labels in any cycle of $\mathcal{G}_b$. On Fig. 3, we represented the cycle in $\mathcal{G}_3$ corresponding to a long siteswap.

$c \leftarrow x_1$
shift X left of one position
if $c = 1$ **then**
   either
   choose i such that $x_i = 0$
   $x_i \leftarrow 1$
   or
   concatenate X with a string of the form 0^*1
end if

Algorithm 1: Transition in the landing state graph

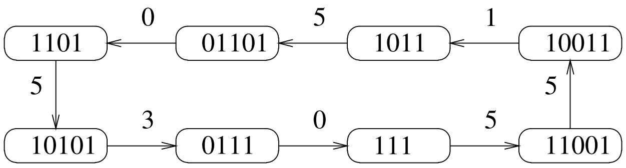


Figure 3: The cycle in $\mathcal{G}_3$ corresponding to the 3-ball siteswap **55150530**

3. *Finding Transitions*

Finding transition between siteswap patterns is a challenging problem that occurs frequently when designing juggling routines [9].

Since a siteswap pattern is a cycle in the state graph, finding a transition between two patterns – if we do not consider circular rotations of a pattern as equivalent – amounts to finding a path between two given states in the graph. The number of edges (throws) in the path cannot be chosen freely: it might be impossible to reach a state from another in a given number of throws.

Problem 1. *Find all achievable path lengths between two landing states X and Y.*

Definition 2. *A* transition sequence *between two states X and Y is the sequence of edge labels (throw heights) on a path from X to Y in $\mathcal{G}_b$.*

We now show that finding achievable path lengths between two landing states can be seen as a special kind of pattern matching problem. Let us, for instance, compute the shortest path from state $X = 1001101$ to state $Y = 1010101$. From the following alignment of X and Y, it is impossible to reach Y from X in a single throw, because some bits in Y are set to zero while the corresponding bits in X are set to 1:

X= 1001101
Y=   101**0**1**0**1.

If we test with a shift of two positions, we obtain:

X= 1001101
Y=   1**0**10101,

which is still impossible. The first shift that corresponds to feasible transition sequences is the 4-position shift:

X= 1001101
Y=   1010101.

There exists a feasible transition sequence because the two '1's in the unmatched prefix of X can be used to fill the two '1' positions in the unmatched suffix of Y. Hence a shortest path between X and Y has length 4 and a corresponding transition sequence may be for instance **8 0 0 7**. Checking all possible shifts yields a simple quadratic algorithm for finding the shortest path between X and Y.

This matching procedure can actually be seen as an instance of a "pattern matching with don't cares" (PMWDC) problem [3, 5, 10]. In a PMWDC problem, some symbols in the pattern are "don't care" symbols, denoted by '$*$', that match any symbol in the text.

For this, transform Y as follows:

1. pad Y, if needed, by a sequence of '0' so that $|Y| \geq |X|$,

2. replace each '1' by a '$*$'.

Let Y' be the new string. Now any (possibly right-hanging) match of this new string with some position in X defines a path from X to Y in $\mathcal{G}_b$. The shortest path length is the position of the first match from left to right.

The above proposition can be shown to be true simply by noticing that only '0's in Y force a '0' as matched symbol in X, while a '1' in Y can be matched with either a '0' or a '1' in X. Also, when a match occurs between Y' and X, there are always enough '1's in the unmatched prefix of X to fill the unmatched '1's in Y.

Note that we can find a simple upper bound on the shortest path length by trying to match exactly a suffix of state X with a prefix of state Y. This can be done in linear time using a standard pattern matching algorithm. In the example above, this upper bound appears to be optimal.

4. *Pattern Matching with Don't Cares*

We briefly recall a recent randomized algorithm from Kalai [5] for PMWDC. This algorithm is inspired by the well-known fingerprint method of Karp and Rabin [6] for standard pattern matching. It runs in $O(n \log m)$ time on a RAM machine, where $n = |X|$ and $m = |Y|$. It returns all matches, as well as, with an arbitrarily small probability, some false matches.

The algorithm receives $X = x_1 x_2 \dots x_n$ and $Y = y_1 y_2 \dots y_m$ as input, and the probability of false match is controlled by a parameter N. It is given as Algorithm 2. The don't care symbols are taken into account in the fingerprinting step by letting the factor in the sequence r be equal to zero.

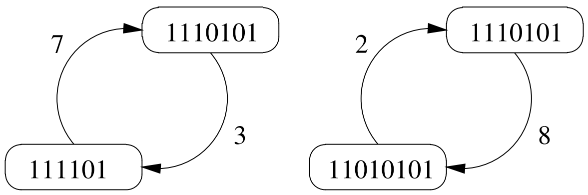
Clearly, a match cannot be missed. The probability that a false match occurs at a given position is the probability that $s(j) = t$ has a solution given that there is at least one wrong symbol in the text, which is equal to $1/N$. The complexity of the algorithm is essentially that of the convolution in the second **for** loop. This convolution can be computed in time $O(n \log m)$ using a FFT.

for $1 \leq i \leq m$ **do**
  if $y_i = *$ **then**
    $r_i \leftarrow 0$
  else
    $r_i \leftarrow$ random from $\{1, 2, \dots, N\}$
  end if
end for
$t \leftarrow \sum_{i=1}^{m} y_i r_i$
for $0 \leq j \leq n - m$ **do**
  $s(j) \leftarrow \sum_{i=1}^{m} x_{i+j} r_i$
end for
output $\{j : s(j) = t\}$

Algorithm 2: PMWDC

5. *Algorithm and Examples*

We now present a simple algorithm inspired by the previous PMWDC method. If we apply the method blindly, we realize that t can only be the zero vector, hence that any nonzero value of $s(j)$ will not be a match. Also, zero values of $s(j)$ can only be matches and the values of r_i for $y_i = 0$ has no importance.



(a) **37** (b) **82**

Figure 4: Shortest cycles going through state 1110101

We can therefore let $r_i = 1$ whenever $y_i = 0$, and $r_i = 0$ whenever $y_i = 1$. The algorithm reduces to the convolution of X with the complement of Y, as described in Algorithm 3. It is deterministic and has complexity $O(n \log n)$ if we use a FFT for the **for** loop, where $n = \max\{|X|, |Y|\}$.

pad Y with '0's if necessary so that $|Y| \geq |X|$
pad X with $|Y|$ '0's
for $0 \leq j \leq |X| - |Y|$ **do**
  $s(j) \leftarrow \sum_{i=1}^{|Y|} x_{i+j}\overline{y_i}$
end for
output $\{j : j > 0$ and $s(j) = 0\}$

Algorithm 3: Algorithm for problem 1

Algorithm 3 returns all matches, i.e. all possible transition lengths up to the trivial case where the unmatched prefix of X contains b '1's.

An interesting special case of the problem occurs when $X = Y$. In that case, our procedure finds the lengths of the cycles going through the given state. The first match corresponds to the shortest such cycle. Let us see for instance what happens when $X = Y = 1110101$, with $b = 5$. The first match is as follows:

X= 1110101
Y=   1110101.

We conclude that the two shortest cycles going through this state have length two, and are respectively **82** and **37**. The intermediate state in both cases is given in Fig. 4.

The complexity of the algorithm matches that of the best known algorithm for boolean convolution. Note that a number of other pattern matching problems can be shown to be harder than computing a boolean convolution [4].

Theorem 1. *If we assume that a boolean convolution cannot be computed in less than $O(n \log n)$ time, then algorithm 3 is an optimal algorithm to solve problem 1.*

Proof. What we actually want to compute is the boolean convolution of two binary words X and $\overline{Y}$, where the number of '1's in X is equal to the number of '0's in $\overline{Y}$. We show that if we can compute this in time less than $O(n \log n)$ we can also compute any boolean convolution in time less than $O(n \log n)$.

To show this, we encode an instance of the boolean convolution problem, made of two binary words A and B, into an instance $(X, \overline{Y})$ of the restricted problem as follows. If the number of '1's in A is greater than the number of '0's in B, we can let $\overline{Y}$ be equal to the concatenation of B with a sufficient number of '0's, and X be equal to A. This clearly does not change the result of the convolution. On the other hand, if the number of '1's in A is less than the number of '0's in B, we can let X be equal to A concatenated with $|B|$ '0's and a sufficient number of '1's, and $\overline{Y}$ be equal to B. The result of the original convolution is a prefix of the convolution of X and $\overline{Y}$. □

6. *Enumerating Transition Sequences*

Now that we have an algorithm for finding the lengths of the paths leading from landing states X to Y, we may ask ourselves what can be the transition sequences for these lengths.

Problem 2. *Given an achievable path length ℓ, output all transitions sequences of length ℓ.*

We first remark that the balls corresponding to the '1's in the unmatched prefix of X need not necessarily be thrown at their final position in the future. They may as well be thrown multiple times before landing in their final position. As an illustration, consider two distinct transition sequences corresponding to the following alignment of two landing states with $b = 4$:

X= 1010101
Y=   1111.

The first obvious transition sequence is the one sending the first ball in the first position to fill in the target state, and the second in the second such position. This is the sequence **5050**. For coherence, **0**-throws should be seen as an instantaneous throw from one hand to itself. Now let us apply siteswapping and permute the landing positions of some of the throws to obtain a new valid sequence of throws. If we permute the landing positions of the first two throws, and then permute the landing positions of the two next throws, we obtain another valid transition sequence: **1414**. In fact, we can jump from any transition sequence to any other by siteswapping, which is also true for traditional siteswap [11].

From this description, it is a simple matter to enumerate all possible transition sequences: we choose the landing time of the first ball and recurse. The landing time can be chosen among the times corresponding to all '0's in the unmatched prefix of X, and all '0's in X that are matched with '1's in Y. The new landing state obtained after this throw is made is still, by definition, correctly matched with Y. It is easy to show that the sum of the heights must be constant (it is preserved by siteswapping), but that the average theorem is clearly not true anymore.

We also give the number of transition sequences of a given length. This result can be proved by induction. Due to lack of space the actual proof is omitted.

Theorem 2. *If ℓ is an achievable path length between X and Y, then the number of distinct transition sequences (paths) of length ℓ is equal to*

$$k! \prod_{i=1}^{k} (i+1)^{n_i}, \tag{1}$$

where:

- *k is the number of '1's in the ℓ-prefix of X,*
- *n_i is the number of '0's between the ith and the $(i+1)$th '1' in the ℓ-prefix of X if $0 < i < k$, or the number of '0's between the kth '1' and position $\ell + 1$ if $i = k$.*

7. *Conclusion*

An interesting open question, which is important in practice, is whether we can find quickly the shortest transition between two states if we restrict the maximum height of a throw. This is the shortest path problem in the finite directed landing state graph $\mathcal{G}_{b,h}$, where b is the number of balls and h the maximum throw height. Another question is whether we can find a transition between two siteswaps without actually computing the landing states. It is also interesting in practice, because the length of the siteswap can be arbitrarily smaller than the number of balls, and *a fortiori* arbitrarily smaller than the final landing state. This can be seen on the b-ball cascade, where b can be arbitrarily large but whose siteswap has always unit length.

References

[1] J. Buhler, D. Eisenbud, R. Graham, and C. Wright. Juggling drops and descents. *Amer. Math. Monthly*, 101:507–519, 1994.

[2] R. Ehrenborg and M. Readdy. Juggling and applications to q-analogues. *Discrete Math.*, 157:107–125, 1996.

[3] M. Fischer and M. Paterson. *Complexity of Computation*, chapter String Matching and Other Products, pages 113–125. SIAM-AMS, 1974. (SIAM-AMS Proceedings 7, R.M. Karp ed.).

[4] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. In *Proceedings of the 39th Symposium on Foundations of Computer Science*, 1998.

[5] A. Kalai. Efficient pattern-matching with don't cares. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 655–656, 2002.

[6] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[7] H. Lundmark. Siteswap state diagrams. `http://www.mai.liu.se/~halun/`, 2003.

[8] B. Magnusson and B. Tiemann. A notation for juggling tricks. *Juggler's World*, pages 31–33, summer 1991.

[9] C. Miller. Finding juggling pattern transitions with smart engineering systems. `http://www.juggling.org/papers/miller/`, 1997.

[10] S. Muthukrishnan and K. Palem. Non-standard stringology: Algorithms and complexity. In *Proceedings of the 26th Symposium on the Theory of Computing*, 1994.

[11] B. Polster. *The Mathematics of Juggling*. Springer-Verlag, 2003.

[12] C.E. Shannon. Claude shannon's no-drop juggling diorama. In N. Sloane and A. Wyner, editors, *Claude Elwood Shannon – Collected Papers*, pages 847–849. IEEE Press, 1993.

[13] C.E. Shannon. Scientific aspects of juggling. In N. Sloane and A. Wyner, editors, *Claude Elwood Shannon – Collected Papers*, pages 850–864. IEEE Press, 1993.

[14] C. Simpson. Juggling on paper. *Juggler's World*, page 31, winter 1986.

[15] E. Steingrimmson. *Permutation Statistics of Indexed and Poset Permutations*. PhD thesis, MIT, Cambridge, Massachussets, 1991.