CUCS-297-87

Reset Sequences for Finite Automata with Application to Design of Parts Orienters

David Eppstein

Computer Science Department
Columbia University
New York, NY 10027

Natarajan reduced the problem of designing a certain type of mechanical parts orienter to that of finding reset sequences for monotonic deterministic finite automata. He gave algorithms that in polynomial time either find such sequences or prove that no such sequence exists. In this paper we present a new algorithm based on breadth first search that runs in faster asymptotic time than Natarajan's algorithms, and in addition finds the shortest possible reset sequence if such a sequence exists. We give tight bounds on the length of the minimum reset sequence. We further improve the time and space bounds of another algorithm given by Natarajan, which finds reset sequences for general automata in the special case that all states are initially possible.

Keywords: automated design of parts orienters, deterministic finite automata, reset sequences, breadth first search

October 31, 1987

Introduction

Natarajan [5] considered the design of automated parts orienters; that is, devices which accept mechanical parts in any orientation or in a wide class of orientations, and output them in some predetermined orientation. One such orienter is a pan handler, in which the part slides around on a tray as that tray is tilted, turning in a well-defined way when it hits the walls of the tray. These devices had been previously been described in [2] and [4].

For a given tray and object, and for a given set of possible initial orientations for the object, one has the problem of determining whether there is a sequence of tilt angles that will cause the object to always end up in the same orientation. Natarajan made the assumptions that the set of angles is finite, that the set of orientations in which the part can rest on a tray face is also finite, that tilting the tray with a given angle and with the object in a given initial orientation always results in the same final orientation, and that this relation between angles, initial orientations, and final orientations is known. With these assumptions he reduced the problem to the following combinatorial one.

One is given a deterministic finite automaton (S, Σ). $S = \{s_1, s_2, \ldots, s_n\}$ is the set of the states of the automaton, corresponding to orientations of the part to be oriented. $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ is the set of the transition functions of the automaton, which we also identify with the input alphabet; these functions correspond to the angles at which the pan may be tilted. One is further given a set of initial states, or orientations, $X \subset S$.

In what follows, sequences of input symbols to the automaton will be denoted using the letter τ. The effect of the input sequence τ on the states of the automaton is given by the composition of the transition functions for each input symbol of τ; as with the input symbols themselves, we identify τ with its effect as a transition function. Note that, if $\tau = \tau_1\tau_2$, then $\tau(s) = \tau_2(\tau_1(s))$. If τ is the empty input sequence, $\tau(s) = s$. We denote the set of all possible input sequences by Σ^*.

The problem to which Natarajan reduced the pan handler problem is to find an input sequence $\tau \in \Sigma^*$ such that $|\tau(X)| = 1$; that is, such that the application of τ will leave the automaton in one particular state no matter which state in X it started at. We call τ a *reset sequence* for (S, Σ) and X.

In the special case that $X = S$, Natarajan gave an algorithm that finds a reset sequence if it exists. This algorithm takes $O(kn^4)$ time. The sequence produced is not guaranteed to be the shortest possible, but Natarajan bounded its length by $O(kn^3)$. As one of the results of this paper, we improve this algorithm to take time $O(n^3 + kn^2)$, and working space bounded by $O(n^2)$. We also prove a tighter bound of $O(n^3)$ on the length of the resulting sequence, and show that finding the minimum length reset sequence is NP-complete.

It turns out that for general automata and general X, finding a reset sequence is PSPACE-complete. However Natarajan observed that the automata arising in the pan handler problem have a property which he called monotonicity, and that with this property the problem becomes solvable in polynomial time. He gave algorithms with asymptotic time complexity $O(kn^4)$ (or $O(kn^3 \log n)$ when $X = S$), which find sequences of length at most $O(kn^3)$ (respectively $O(kn^2 \log n)$). The sequences found are again not guaranteed to be optimal.

This paper presents a new algorithm for finding reset sequences on monotonic automata, which takes time $O(kn^2)$ and is guaranteed to find the shortest possible sequence. Further, this leads to tight worst case bounds of n^2-2n+1 on the number of input symbols in the optimal reset sequence. The algorithm works by defining a new automaton, the states of which correspond to intervals in the cyclic order of the original automaton's states. Reset sequences in the original automaton correspond to paths in the new automaton leading to a singleton interval. Therefore we can find our desired sequence using a simple breadth first search technique.

Definitions and Lemmas

First, we define monotonic automata. Assume that the states of a given deterministic finite automaton (DFA) are arranged in some known cyclic order $s_1, s_2, \ldots, s_n$. A transition function σ is *monotonic* if it preserves the cyclic order of the states. Formally, the sequence of states $\sigma(s_1), \sigma(s_2), \ldots, \sigma(s_n)$, after removal of possible adjacent duplicate states, must be a subsequence of a cyclic permutation of $s_1, s_2, \ldots, s_n$. If a set of transition functions is monotonic, then all compositions of those transition functions will also be monotonic.

We call an automaton monotonic when all of its transition functions are monotonic. From now on when we refer to the automaton (S, Σ) we will assume that it is monotonic.

Next let us define an interval $[s_i, s_j]$. This consists of all those states between s_i and s_j (inclusive) in the cyclic order of the states; e.g., $[s_1, s_3] = \{s_1, s_2, s_3\}$. Note that there are n different ways of representing the full set of states S as an interval $[s_i, s_{i-1}]$; any other set of states that can be represented as an interval has exactly one such representation. We say that an interval $[s_h, s_i]$ is contained in another interval $[s_g, s_j]$ when the endpoints of the intervals appear in the cyclic order s_g, s_h, s_i, s_j. Containment as an interval implies containment as a set of states, but the reverse may be false in the case that the containing interval is all of S.

Lemma 1. For all $\tau \in \Sigma^*$, and for any interval I, $\tau^{-1}(I)$ is an interval.

Proof: If not, there would be s_{i_1}, s_{i_2}, s_{i_3}, and s_{i_4} in cyclic order such that $\tau(s_{i_1})$ and $\tau(s_{i_3})$ are in I but $\tau(s_{i_2})$ and $\tau(s_{i_4})$ are not; but this violates monotonicity. •

It turns out that, unlike the inverses of transition functions, the original transition functions of the DFA do not necessarily take intervals to intervals. However we can define a new set of transition functions, corresponding to the original ones, that do take intervals to intervals. For $\sigma \in \Sigma$, let $\sigma'([s_i, s_j])$ be (1) $[\sigma(s_i), \sigma(s_j)]$ if $\sigma(s_i) \neq \sigma(s_j)$; (2) $\sigma([s_i, s_j])$ if this is a singleton; that is, if σ maps the whole interval to one state; and (3) undefined otherwise.

The new transition functions we have defined give us a new DFA $(S \times S, \Sigma')$ whose states are the intervals of the original automaton, and which takes the same input alphabet as the original automaton. Note that this DFA, which is of size $O(kn^2)$, can be constructed in time linear in its size. The only complication is how to determine whether the result of a transition in which the two endpoints are mapped to a single point should be that point or undefined. This can be done by first constructing for each $\sigma \in \Sigma$ and $s \in S$ the interval $\sigma^{-1}(s)$, which must exist by lemma 1. This construction takes time $O(n)$, and there are $O(kn)$ intervals to construct, so all such intervals can be constructed in time $O(kn^2)$. Then determining which transitions are undefined can be done in constant time, as follows. If the endpoints of interval I are mapped by σ to the same state s, $\sigma'(I)$ is defined if and only if $I \subset \sigma^{-1}(s)$. The containment above should be interpreted as being

between sets rather than as the interval containment defined earlier; it can be calculated using a constant number of comparisons between interval endpoints to determine interval containment, together with a test for the special case that $\sigma^{-1}(s) = S$, which is the only case in which set and interval containment can differ.

If τ is a composition of transition functions $\sigma_i\sigma_j\sigma_k\cdots$, we define τ' to be the corresponding composition of interval transition functions $\sigma_i'\sigma_j'\sigma_k'\cdots$. If any of the individual transitions in $\tau'(I)$ is undefined, the result as a whole is also undefined. If τ is the empty input sequence we define $\tau'(I) = I$.

Lemma 2. For all $\sigma \in \Sigma$, and all intervals I, if $\sigma'(I)$ is defined, then $\sigma(I) \subset \sigma'(I)$.

Proof: Let $I = [s_i, s_j]$. If $\sigma(s_i) = \sigma(s_j)$, then $\sigma(I) = \sigma'(I) = \{\sigma(s_i)\}$, so we need only consider the case that σ takes s_i and s_j to different states, in which case $\sigma'(I) = [\sigma(s_i), \sigma(s_j)]$. Now clearly both $\sigma(s_i)$ and $\sigma(s_j)$ are in $[\sigma(s_i), \sigma(s_j)]$. For each remaining $s \in [s_i, s_j]$, if $\sigma(s) = \sigma(s_i)$ or $\sigma(s) = \sigma(s_j)$ then again $\sigma(s) \in [\sigma(s_i), \sigma(s_j)]$. Otherwise $\sigma(s_i)$, $\sigma(s)$, and $\sigma(s_j)$ are distinct, so monotonicity forces them to appear in that cyclic order; but this is the same as saying that $\sigma(s) \in [\sigma(s_i), \sigma(s_j)]$. Thus all states in $\sigma(I)$ are contained in $[\sigma(s_i), \sigma(s_j)]$. •

Lemma 3. For all τ in Σ^*, and all intervals I, if $\tau'(I)$ is defined, then $\tau(I) \subset \tau'(I)$.

Proof: We use induction on the length of τ. If τ is empty, $\tau'(I) = I = \tau(I)$. Otherwise assume $\tau = \bar{\tau}\sigma$; i.e., let σ be the last input symbol of τ. Then $\bar{\tau}'(I)$ and $\sigma'(\bar{\tau}'(I))$ must be defined, or $\tau'(I)$ would be undefined. By induction we have that $\bar{\tau}(I) \subset \bar{\tau}'(I)$; further, by lemma 2, $\sigma(\bar{\tau}'(I)) \subset \sigma'(\bar{\tau}'(I))$. Putting this together gives $\tau(I) = \sigma(\bar{\tau}(I)) \subset \sigma(\bar{\tau}'(I)) \subset \sigma'(\bar{\tau}'(I)) = \tau'(I)$. •

Lemma 4. Given $\tau \in \Sigma^*$, and an interval I such that $\tau'(I)$ is defined, then for all intervals $\bar{I} \subset I$, with containment as intervals rather than as sets, $\tau'(\bar{I})$ is also defined and $\tau'(\bar{I}) \subset \tau'(I)$.

Proof: We use induction on the length of τ. If τ is empty the lemma obviously holds, so assume $\tau = \bar{\tau}\sigma$. By induction $\bar{\tau}'(\bar{I}) \subset \bar{\tau}'(I)$. Let $\bar{\tau}'(\bar{I}) = [s_i, s_j]$. If $s_i = s_j$, then $\sigma'([s_i, s_j])$ is easily seen to satisfy the conditions of the lemma, so assume the two states are different. If $\sigma(s_i) \neq \sigma(s_j)$, then by monotonicity these two states appear in the correct order within $\tau'(I)$, and again the lemma is satisfied. The remaining case to check is that $\sigma(s_i) = \sigma(s_j)$, and that there is some state s in $[s_i, s_j]$ such that $\sigma(s) \neq \sigma(s_i)$. But for this to occur without violating monotonicity, it must be that $\sigma(s) \notin \sigma'(\bar{\tau}'(I))$, and since $s \in \bar{\tau}'(I)$ this contradicts either lemma 2 or the assumption that $\tau'(I) = \sigma'(\bar{\tau}'(I))$ is defined. •

Lemma 5. Given $\sigma \in \Sigma$, and states of the original automaton s_i and s_j, if $\sigma'([s_i, s_j])$ is undefined, then $\sigma'([s_j, s_i])$ is defined and a singleton.

Proof: Assume for a contradiction that both $\sigma'([s_i, s_j])$ and $\sigma'([s_j, s_i])$ are undefined. Then we would have two states $s_h \in [s_i, s_j]$ and $s_k \in [s_j, s_i]$, such that $\sigma(s_h) \neq \sigma(s_i) = \sigma(s_j) \neq \sigma(s_k)$. But these states occur in cyclic order s_i, s_h, s_j, s_k, so the above equalities and inequalities violate monotonicity. Therefore, if the conditions of the lemma hold, $\sigma'([s_j, s_i])$ must be defined, and since $\sigma'([s_i, s_j])$ is undefined $\sigma(s_i) = \sigma(s_j)$ and $\sigma'([s_j, s_i])$ is a singleton. •

Lemma 6. For all $\sigma \in \Sigma$ and all intervals $\bar{I}$, if I is a representation of $\sigma^{-1}(\bar{I})$ as an interval (which by lemma 1 must exist), and if I is not all of S, then $\sigma'(I)$ is defined and a subset of $\bar{I}$.

Proof: Let $I = [s_i, s_j]$. If $\sigma'([s_i, s_j])$ were undefined, then $\sigma'([s_j, s_i])$ would be defined and a singleton by lemma 5. But then $\sigma(S) = \sigma(I \cup [s_j, s_i]) = \sigma(I) \cup \sigma([s_j, s_i]) \subset \bar{I} \cup \{\sigma(s_j)\} = \bar{I}$, which contradicts the assumption that $I = \sigma^{-1}(\bar{I}) \neq S$. Therefore $\sigma'(I)$ is defined. By the definition of I, $\sigma(s_i)$ and $\sigma(s_j)$ are both in $\bar{I}$; by monotonicity, they must appear in the correct cyclic order within that interval. It follows that $\sigma'(I) \subset \bar{I}$. •

We now prove the main lemma, which shows the equivalence between reset sequences in the original automaton and paths to a singleton in the interval automaton.

Lemma 7. Given $\tau \in \Sigma^*$, then $|\tau(X)| = 1$ if and only if there is a representation of $\tau^{-1}(\tau(X))$ as an interval I such that $\tau'(I)$ is defined and $|\tau'(I)| = 1$.

Proof: If there is some such I then, by lemma 3, $\tau'(I) \supset \tau(I) \supset \tau(X)$, so $|\tau(X)| = 1$. In the other direction, assume we are given τ and X with $|\tau(X)| = 1$. We want to find a representation of $\tau^{-1}(\tau(X))$ as an interval meeting the terms of the lemma. We will use induction on the length of τ. As the base case, if τ is empty then X is itself a singleton interval satisfying the lemma. Otherwise let $\tau = \sigma\bar{\tau}$; i.e., unlike the proofs of the previous lemmas let σ be the first input symbol in τ. Now $\bar{\tau}(\sigma(X)) = \tau(X)$ is a singleton, so by induction there is a representation of $\bar{\tau}^{-1}(\bar{\tau}(\sigma(X))) = \bar{\tau}^{-1}(\tau(X))$ as an interval $\bar{I}$ such that $\bar{\tau}'(\bar{I})$ is defined and a singleton. Then $\tau^{-1}(\tau(X)) = \sigma^{-1}(\bar{I})$.

First assume $I = \sigma^{-1}(\bar{I})$ is not all of S. By lemma 6 we see that $\sigma'(I) \subset \bar{I}$. Therefore, using lemma 4, $\tau'(I) = \bar{\tau}'(\sigma'(I)) \subset \bar{\tau}'(\bar{I})$ is defined and a singleton, as was to be shown.

The remaining case is that $\sigma^{-1}(\bar{I}) = S$. Choose the interval $[s_i, s_j]$ such that s_i and s_j are both in $\sigma(S)$ (and therefore also in $\bar{I}$), and also such that $\sigma(S) \subset [s_i, s_j] \subset \bar{I}$, with the first inclusion as sets and the second as intervals. This can be done by taking s_i to be the first member in $\bar{I}$ that is also in $\sigma(S)$, and s_j to be the last such member.

Now if $s_i = s_j$, then $\sigma(S) = s_i$, and any representation of S as an interval I will give us $\tau'(I)$ defined and a singleton, satisfying the lemma. So assume $s_i \neq s_j$. This implies that $\sigma^{-1}(s_j)$ is not all of S, so by lemma 1 this set has a unique representation as an interval $[s_h, s_k]$. Using monotonicity and the fact that $\sigma(S) \subset [s_i, s_j]$, it can be shown that that $\sigma(s_{k+1}) = s_i$. Therefore $\sigma'([s_{k+1}, s_k])$ is defined and equal to $[s_i, s_j]$. By lemma 4 we see that $\tau'([s_{k+1}, s_k]) = \bar{\tau}'([s_i, s_j]) \subset \bar{\tau}'(\bar{I})$ is defined and a singleton. •

Reset Sequences for Monotonic Automata

Theorem 1. A minimum length reset sequence for monotonic automaton (S, Σ) and initial states $X = \{s_{i_1}, s_{i_2}, \ldots\}$ can be found in time bounded by $O(kn^2)$.

Proof: First one constructs the automaton $(S \times S, \Sigma')$ as described above. By lemma 7, a minimum reset sequence for the original automaton will also be a minimum length path in the new automaton from some interval containing X to a singleton interval, and vice versa. By lemma 4, we need only consider starting from the minimal intervals containing X, rather than all intervals containing X; these minimal intervals can be found as $[s_{i_j}, s_{i_{j-1}}]$. Finally, the shortest path from one of these intervals to a singleton can be found using the standard breadth first search algorithm. •

Theorem 2. If a minimum length reset sequence for monotonic automaton (S, Σ) and initial states X exists, its length is $\leq n^2 - 2n + 1$.

Proof: The path constructed by the breadth first search in theorem 1 will visit each state of the constructed automaton at most once, and there are n^2 states. But in fact the sequence need involve at most one interval representing all of S, and at most one singleton; thus there are at least $2n-2$ states not included in the minimum length path. •

Theorem 3. For any n, there exists a monotonic automaton (S,Σ) with $|S| = n$, and a set of initial states X, such that the minimum length reset sequence for (S,Σ) and X has length $n^2 - 2n + 1$.

Proof: Name the states $s_1, s_2, \ldots s_n$, in that cyclic order. Let Σ consist of only two transition functions, σ_1 and σ_2. Let both functions take s_n to s_1, but let σ_1 take all states s_i other than s_n to s_{i+1}, and let σ_2 take all states s_i other than s_n to themselves. We take $X = S$.

Assume we have a reset sequence τ, and define τ_i to be the prefix of τ consisting of the first i symbols of τ. Also define $l(i)$ to be the length of the shortest interval containing all the states in $\tau_i(S)$. If by $|\tau|$ we denote the number of input symbols in τ, then clearly $l(|\tau|) = 1$. Finally, define $t(j)$, for each j, to be the least i such that $l(i) \leq j$.

We prove below that, for each $j < n-1$, $t(j) \geq t(j+1) + n$; that is, there must be at least n input symbols processed between each point at which the shortest interval containing the states becomes shorter. The theorem then follows, because the total number of steps in the reset sequence must be at least $n(n-2)$ for the $n-2$ gaps of n steps each, plus one initial step to reduce $l(t)$ from n to $n-1$.

First note that, if $j \neq 1$, the ith input symbol is σ_1, and $\tau_i(S) \subset [s_j, s_k]$, then $\tau_{i-1}(S) \subset [s_{j-1}, s_{k-1}]$. If $j \neq 1$, the ith input symbol is σ_2, and $\tau_i(S) \subset [s_j, s_k]$, then $\tau_{i-1}(S) \subset [s_j, s_k]$. Therefore, no matter what the input symbols of τ are, if $\tau_i(S) \subset [s_j, s_k]$, we can see using induction that $\tau_{i-j+1}(S) \subset I$ for some interval I of length $k-j+1$.

Next observe that if the ith input symbol is σ_1, then $l(i-1) = l(i)$; therefore for each j the input symbol at position $t(j)$ must be σ_2, and further it must be the case that $\tau_{t(j)-1}(S) \subset [s_n, s_j]$. Using the previous observation we see that $\tau(t(j)-n) \subset I$ for some interval I of length $j+1$, and therefore $t(j+1) \leq t(j) - n$ as was to be proved. The theorem then follows as described above. •

Various generalizations of the algorithms and bounds above may be taken. For instance, let us consider the case that what is desired as the result of the reset sequence is a particular state rather than just any single state. The same algorithm as in theorem 1, but with the breadth first search terminating only when it reaches the singleton interval corresponding to the desired state, will always find the minimum such reset sequence when it exists, again taking time $O(kn^2)$. The upper bound of theorem 2 must be relaxed to $n^2 - n$, because it is now possible for the path in the interval automaton to go through all singleton intervals before it gets to the desired one. And the example used in theorem 3, with the desired singleton state being s_n, requires $n^2 - n$ steps for a reset sequence, showing that this new bound is tight.

Reset Sequences for General Automata

In this section we will relax the requirement that the automaton (S,Σ) be monotonic, and instead restrict our attention to reset sequences for all of S; that is, we will assume that the automaton may initially be in any of its states, rather than in a state drawn from some subset X of its states.

The reason that the case we study is easier than the general case is that we can never get stuck: if there exists a reset sequence τ, then no matter what sequence $\bar{\tau}$ we have chosen already, $\bar{\tau}\tau$ will still be a reset sequence for the whole set. We can proceed by reducing the size of $\bar{\tau}(S)$ a step at a time, without ever having to worry about backtracking.

The following algorithm, due to Natarajan, works in the above manner to find a reset sequence for any automaton (S, Σ), with the initial set of states being all of S. The reset sequence it finds is not necessarily the shortest possible such sequence. We will put off describing the implementation details of some of the steps until later.

```
Algorithm 1:
    begin
        X ← S;
        τ ← the empty sequence;
        while |X| > 1 do begin
            pick s_i, s_j ∈ X with s_i ≠ s_j;
            find a sequence τ̄ taking s_i and s_j to the same state;
            X ← τ̄(X);
            τ ← ττ̄;
        end;
    end
```

Theorem 4. Assuming the steps in the loop of algorithm 1 can be computed, the algorithm terminates after $O(n)$ repetitions of the loop, and finds a reset sequence for (S, Σ) if such a sequence exists. If the algorithm ever chooses a pair of states s_i, s_j such that no sequence $\bar{\tau}$ takes the two states to a single state, then no reset sequence exists.

Proof: Each time through the loop, the size of X decreases by at least one; therefore the loop can be executed at most n times. When the size of X has decreased to one, τ will then be a reset sequence. If any reset sequence τ exists, it will a fortiori satisfy the conditions for $\bar{\tau}$. •

The two steps of the algorithm that take the most time are finding $\bar{\tau}$ and applying it to X. We now describe some preprocessing that allows these steps to be done quickly and with little space.

Theorem 5. Algorithm 1 can be executed in time $O(n^3 + kn^2)$.

Proof: As in the monotonic case, we first form a new automaton of size $O(kn^2)$ and perform a breadth first search in it. The states of the new automaton consist of each (unordered) pair of states from the original automaton, together with one state for each of the original automaton's states. The result of applying any the original automaton's input symbols σ to a pair of states (s_i, s_j) will be $(\sigma(s_i), \sigma(s_j))$; if $\sigma(s_i) = \sigma(s_j)$ then the result will be that singleton state.

Before we run algorithm 1 itself, we perform a breadth first search on the new automaton, finding for each pair of original states (s_i, s_j) a shortest input sequence $\tau_{i,j}$ taking that pair to a singleton state. This can be performed in time $O(kn^2)$, and the result can be represented as a shortest path forest in space $O(n^2)$; paths in this forest lead from each pair to a singleton, along the sequence of pairs found by applying each transition function in $\tau_{i,j}$ successively to the pair (s_i, s_j).

In the following description we will call the above breadth first search stage 1. If we only desire to know whether there is a reset sequence, without needing to know what that reset sequence is,

then we may stop now, having taken time $O(kn^2)$: a reset sequence exists if and only if for every pair (s_i, s_j) such a sequence $\tau_{i,j}$ leading to a singleton can be found.

Next, as stage 2 of our preprocessing, for each pair of states (s_i, s_j) and each state s_k of the original automaton, we compute $\tau_{i,j}(s_k)$. This is done by performing a pre-order traversal of the shortest path forest computed in stage 1. Whenever we visit a pair (s_i, s_j), we compute in constant time $\tau_{i,j}(s_k)$, for all states s_k, as follows. Let $\tau_{i,j} = \sigma\tau_{g,h}$, where σ is the first transition function in $\tau_{i,j}$, and $\sigma((s_i, s_j)) = (s_g, s_h)$. If $s_g = s_h$ let $\tau_{g,h}$ be the empty sequence of transition functions, which corresponds to the identity function. Then $\tau_{i,j}(s_k) = \tau_{g,h}(\sigma(s_k))$ can be computed as one function evaluation of $\sigma(s_k)$ followed by a table lookup of the value of $\tau_{g,h}(\sigma(s_k))$; because we are performing a pre-order traversal the latter value will have already been computed. Since there are $O(n^3)$ computations to be performed, each taking constant time, the total time for this stage is $O(n^3)$.

Now we show how to perform the steps of the main algorithm described above. To find $\bar{\tau}$ for s_i and s_j, we simply look up $\tau_{i,j}$ in the forest we calculated in the first stage; there are $O(n^2)$ pairs, so the shortest sequence $\tau_{i,j}$ resulting in a singleton is at most $O(n^2)$ symbols long, and therefore this step takes time bounded by $O(n^2)$. To find $\bar{\tau}(X)$ we simply look up, for each member s of X, $\bar{\tau}(s)$ as calculated in the second stage; $|X| \leq n$ so this step takes time $O(n)$. The inner loop is executed $O(n)$ times, so the execution of algorithm 1 as a whole takes time $O(n^3 + kn^2)$, which is also the time taken by the preprocessing stages. •

Recall that we claimed that we could reduce the working space used to $O(n^2)$ while keeping the time bounds described above. We do not count the length of the output sequence, for which the best bound we have is $O(n^3)$, as part of this space bound.

The pair automaton we constructed would seem to take $O(kn^2)$ space, but in fact we need only to use constant storage space for each pair of the automaton, and construct the outgoing arcs from each pair as needed from the original automaton. A more serious obstacle to reducing the space is that the space required to store $\tau_{i,j}(s_k)$ is $\Omega(n^3)$. However it turns out to be possible to reduce the space required, by keeping $\tau_{i,j}(s_k)$ only for certain pairs (s_i, s_j) rather than all such pairs. First let us describe an algorithm to compute the pairs for which we will calculate the values of $\tau_{i,j}$. This algorithm is given as input a forest of size x, and another integer parameter y. It calculates a partition of the forest into $O(x/y)$ subtrees, each of depth at most y.

Algorithm 2:

```
for each vertex v of the forest, in a post-order traversal, do begin
    size(v) ← 1;
    for each vertex w such that (w, v) is an edge in the forest do
        if mark(w) = 0 then size(v) ← size(v) + size(w);
    if size(v) < y then mark(v) ← 0;
    else mark(v) ← 1;
end
```

Lemma 8. Algorithm 2 takes time linear in x, the number of vertices in the forest it processes. After it has been executed, there will be at most x/y vertices v of the forest with $mark(v) = 1$. Further, if we break the outgoing link of each such marked vertex, no tree in the new forest so created will have depth greater than y.

Proof: The post-order traversal guarantees that $size(w)$ and $mark(w)$ in the inner loop of the algorithm will have been calculated before we process vertex v. For each vertex v, $size(v)$ computes the number of vertices in the subtree of unmarked vertices rooted at v. Each marked vertex has at least $y-1$ unmarked vertices in its subtree, so there can be at most x/y marked vertices. If any tree of unmarked vertices rooted at a marked vertex had depth greater than y, the number of unmarked vertices on any path of length greater than y to the root would be enough to have caused one of the vertices along that path to have been marked; therefore the depth of each such tree is at most y. •

Theorem 6. Algorithm 1 can be executed in time $O(n^3 + kn^2)$ as in theorem 5. using working space bounded by $O(n^2)$.

Proof: We compute stage 1 as before. But before performing stage 2, we run algorithm 2 on the shortest path forest computed in stage 1, with x being the number $n(n+1)/2$ of pairs and singletons in the forest, and y equal to n, the number of states in the original automaton.

In stage 2 we now only compute $\tau_{i,j}(s_k)$ for those pairs (s_i, s_j) that were marked by algorithm 2. Again we will process each such pair in order by a pre-order traversal of the forest. We first compute the shortest prefix of $\tau_{i,j}$ that takes (s_i, s_j) to another marked pair (s_g, s_h); call this shortest prefix $\bar{\tau}$. By lemma 8, the number of transition functions in $\bar{\tau}$ is at most n. Then $\tau_{i,j}(s_k) = \tau_{g,h}(\bar{\tau}(s_k))$, which can be computed with at most n function evaluations followed by a table lookup. Using lemma 8 again we see that there are at most $O(n)$ marked pairs, and for each such pair we have to perform n computations each taking time $O(n)$, so the total time for the new version of stage 2 is again bounded by $O(n^3)$. We store $\tau_{i,j}(s_k)$ for only $O(n)$ pairs (s_i, s_j), so the total space used is bounded by $O(n^2)$.

In algorithm 1 itself, the only changed step is in computing $\tau_{i,j}(X)$. Here (s_i, s_j) might not be marked, but as in stage 2 we can find a shortest prefix $\bar{\tau}$ of $\tau_{i,j}$ taking (s_i, s_j) to a marked pair (s_g, s_h). Again $\bar{\tau}$ has length at most n, so for each member s of X we can find $\tau_{i,j}(s) = \tau_{g,h}(\bar{\tau}(s))$ by $O(n)$ function evaluations followed by a table lookup. The entire computation of $\tau_{i,j}(X)$ takes time bounded by $O(n^2)$, which does not reduce the running time of the algorithm from that of theorem 5 by more than a constant factor. •

Theorem 7. The reset sequence found by algorithm 1, as implemented in theorems 5 and 6, has length at most $O(n^3)$.

Proof: There are $O(n^2)$ pairs and singletons in the derived automaton, so each $\tau_{i,j}$ has length bounded by $O(n^2)$. The reset sequence as a whole is the concatenation of at most n such sequences, so its length is bounded by $O(n^3)$. •

A more exact analysis shows that, if the pair (s_i, s_j) is always chosen to have the shortest sequence $\tau_{i,j}$ among all pairs remaining in X, which can be done within the same asymptotic time and space bounds as before, the length of the resulting reset sequence will be at most $n^3/3 - n^2 + 5n/3 - 1$. •

Theorem 8. Finding the shortest possible reset sequence for an automaton is NP-complete.

Proof: More precisely the problem is, given an automaton and an integer parameter m, to test whether the automaton has a reset sequence of length less than or equal to m. By theorem 7, such

a sequence need have at most polynomial length, so the problem is in NP. We prove completeness by reducing 3-SAT [1] to the problem.

Assume we are given a satisfiability problem in conjunctive normal form, with m variables $x_1, x_2, \ldots x_m$, and with n clauses. The automaton we construct will need only two transition functions σ_1 and σ_2. There will always be a reset sequence of length $m+1$ (in fact any input sequence of that length will reset the automaton), but any reset sequence of length m or less will correspond to a satisfying assignment. The assignment is constructed by letting x_j be true if the jth input symbol of the reset sequence is σ_1, or false if the jth input symbol is σ_2. Conversely the opposite transformation will produce a reset sequence of length m from any satisfying assignment.

The automaton itself is constructed as follows. It will have one special state r, and $mn+m$ other states $s_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq m+1$. For all states $s_{i,m+1}$, and for state r, both transition functions will lead to state r. If the ith clause of the formula contains x_j, σ_1 will take state $s_{i,j}$ to r; if that clause contains $\bar{x}_j$, σ_2 will take $s_{i,j}$ to r. We call these transitions to r from states other than $s_{i,m+1}$ *shortcuts*. All remaining transitions take $s_{i,j}$ to $s_{i,j+1}$.

It can be seen that, as we stated above, any input sequence will take all states to r in at most $m+1$ steps. Further, all states except $s_{i,1}$ will always be taken to r in m steps, so we need only concern ourselves with the former states. If a reset sequence of length m or less exists, then each of these initial states $s_{i,1}$ must be taken by that sequence across a shortcut transition, because otherwise an initial state $s_{i,1}$ would progress through all the states $s_{i,j}$ before reaching r, and that would take $m+1$ steps.

The variable assignment computed from the reset sequence must have a true variable in each clause, corresponding to the shortcut taken by the initial state corresponding to that clause. Thus we see that a satisfying assignment to the formula can be derived from a short reset sequence. Conversely, if an assignment satisfies the formula, the derived input sequence would cause each initial state $s_{i,1}$ to take a shortcut corresponding to the first true variable in the corresponding clause, and we would have a reset sequence of length m. Thus we see that the formula will be satisfiable if and only if the derived automaton has a short reset sequence. •

Conclusions and Open Problems

We have shown that, given a monotonic DFA and a set of initial states it may be in, we can construct a minimum length sequence (if one exists) that takes all the initial states to one particular state. This construction can be performed in time bounded by $O(kn^2)$. Further, we have shown that the length of the resulting sequence is at most $n^2 - 2n + 1$; there are DFAs for which the minimum reset sequence exists and is this long, so this bound is tight.

We have also shown that, in the general case in which the automaton is not monotonic, we can still find a reset sequence for all of S in time bounded by $O(n^3 + kn^2)$ and working space bounded by $O(n^2)$. The length of the resulting sequence is not necessarily optimal, but is bounded by $O(n^3)$.

Some questions remain open. For instance, the algorithm for monotonic automata may be performed in polylogarithmic parallel time using Kučera's breadth first search algorithm [3], and similarly we may perform the preprocessing in stages 1 and 2 of our algorithm for general automata in NC. But the main part of the latter algorithm seems to be inherently sequential; a natural question is whether it too can be performed in parallel, or whether some other algorithm exists

that can find reset sequences in parallel. A partial result in this direction is that a reset sequence can be found in random NC; this can be done by choosing a long random sequence of pairs of states (s_i, s_j) and concatenating the sequences $\tau_{i,j}$ that take each random pair to a singleton. If there are at least n^3 pairs in the random sequence, then with very high probability the corresponding input sequence will be a reset sequence, and this can be tested in parallel.

Another open problem is the gap between the $O(n^3)$ upper bound on the length of reset sequences for general automata, and the $\Omega(n^2)$ lower bound given for the special case of monotonic automata.

References

[1] M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

[2] D.D. Grossman and M.W. Blasgen, *Orienting Parts by Computer Controlled Manipulation*, IEEE Trans. Systems, Man and Cybernetics 5(5), 1975, pp. 561–565.

[3] L. Kučera, *Parallel Computation and Conflicts in Memory Access*, Info. Proc. Letters 14(2), April 1982, pp. 93–96.

[4] M.T. Mason and M. Erdmann, *An Exploration of Sensorless Manipulation*, 3rd IEEE Intl. Conf. Robotics and Automation, San Francisco, April 1986.

[5] B.K. Natarajan, *An Algorithmic Approach to the Automated Design of Parts Orienters*, Proc. 27th Annual Symp. Foundations of Computer Science, IEEE, 1986, pp. 132–142.